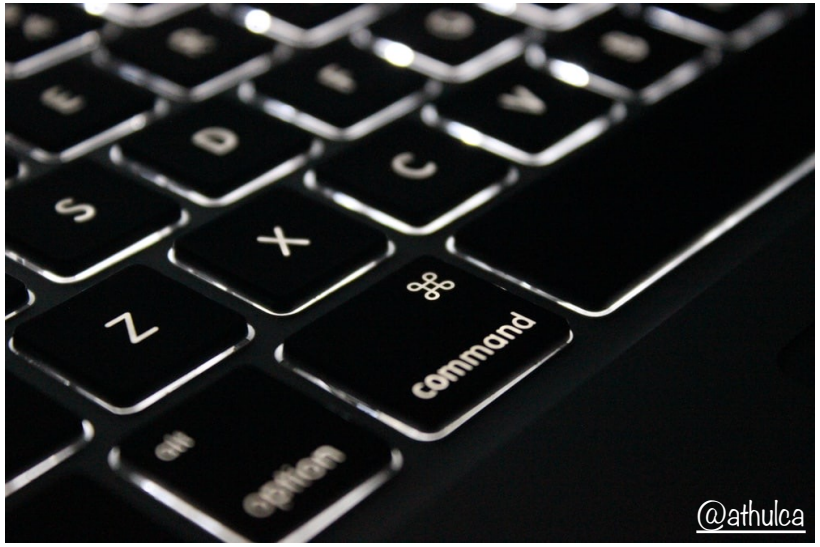
COS 217: Introduction to Programming Systems

Indirection

Command Line Arguments, Structures, and Dynamic Memory

PRINCETON UNIVERSITY



INDIRECTION IN COMMAND LINE ARGUMENTS

```
$ ./printargv one two three
```

What's My Name?

- `String[] args` was COS 126 day 1

- `main()` receives command line parameters in an array of strings in Java

In C

- `main()` also receives arguments in an array of strings
  - Array of arrays of characters
- But they are represented and accessed differently

Indirection in Receiving Command Line Parameters

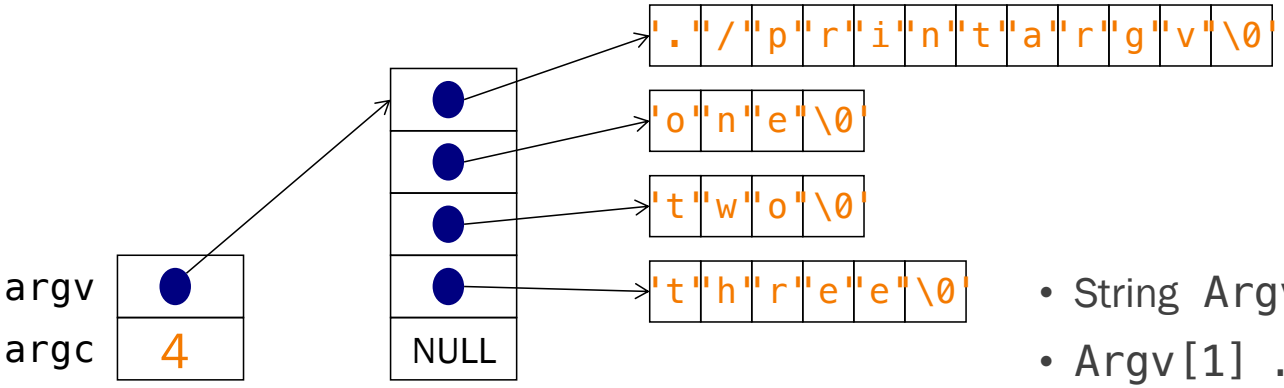
- `main()` receives arguments in array `argv`, a parameter to it

```
int main(int argc, char *argv[])
```

- `main()` doesn't know how many parameters, or how long each is
- Indirection easily allows variable numbers and lengths of parameters
  - `argv`: array of variable no. of pointers, each to variable-length string
  - Note: As parameters, `char *argv[]` and `char **argv` are identical
- How many parameters? How long is each?
  - Unlike Java, in C arrays aren't objects with known lengths
  - Can't use `sizeof(argv[])` in `main` to find out, as it results in 8 bytes
  - Instead, teminating NULL pointer and null character
  - For convenience, 1st parameter to `main()`, `argc`, holds # of arguments

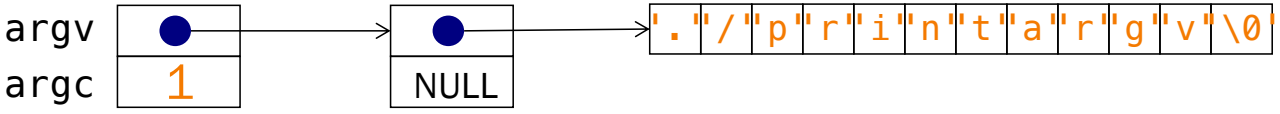
So, What's in `argc` and `argv[]`?

```
$ ./printargv one two three
```



- String `Argv[0]` is command name
- `Argv[1] … argv[argc-1]` are other command line parameters
- `Argv[argc]` is `NULL`

```
$ ./printargv
```

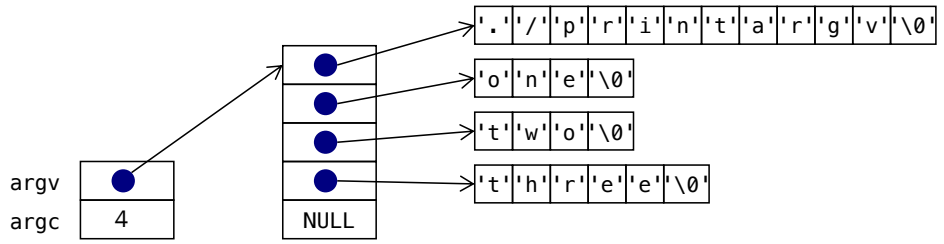


Can Access in Code Using `for` Loop and `argc`

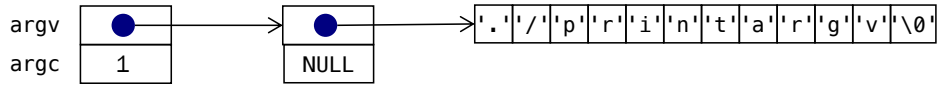
printargv.c:

```
int main(int argc, char *argv[])
{
  int i;
  printf("argc:  %d\n", argc);

  for (i = 0; i < argc; i++)
    printf("argv[%d]:  %s\n", i, argv[i]);

  return 0;
}
```

```
$ ./printargv one two three
```



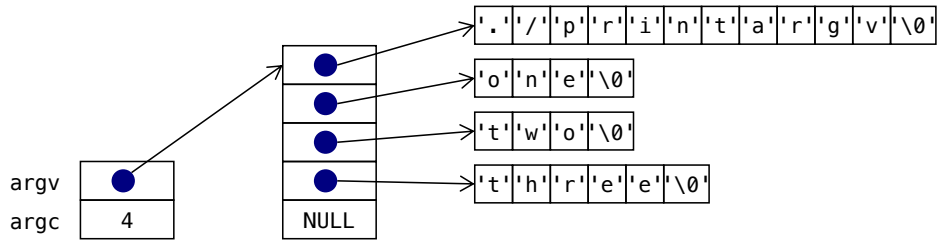

```
$ ./printargv
```



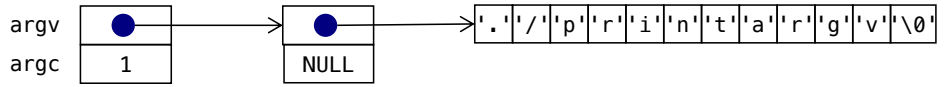
Or `while` Loop and Pointers, Terminating at NULL

```
int main(int argc, char *argv[])
{
  char **ppc = argv;
  int i = 0;
  printf("argc:  %d\n", argc);

  while(*ppc != NULL)
    printf("argv[%d]:  %s\n", i++, *ppc++);

  return 0;
}
```

```
$ ./printargv one two three
```

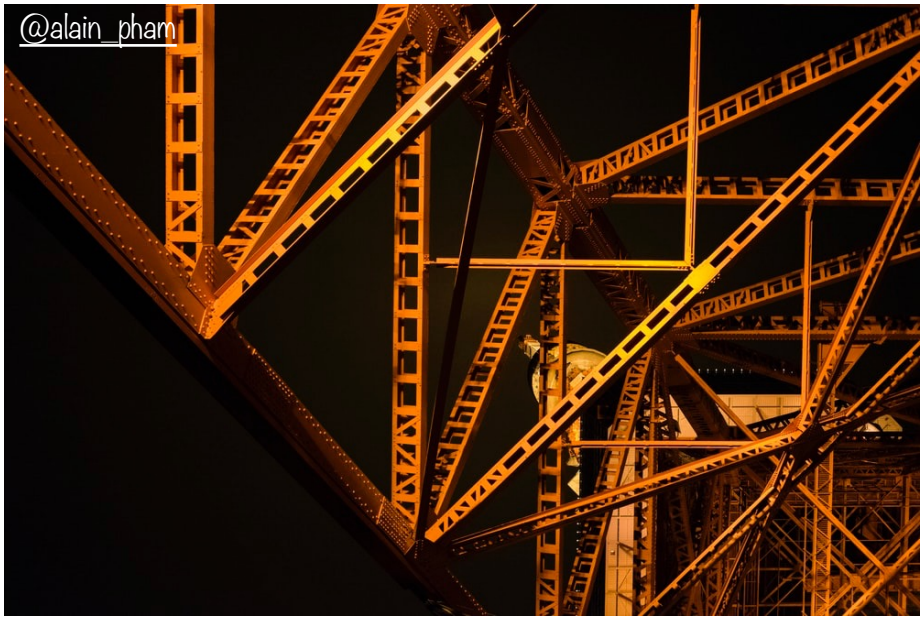



```
$ ./printargv
```



Post-increment with pointers is just like with integer types: emits **old** value of `ppc`. The dereference then happens on value emitted.

This could be parenthesized as: `*(ppc++)`



INDIRECTION AND VARIABLE-FORM C STRUCTURES

Why Structures

- Arrays are multi-element types; i.e. a collection of N elements
- But 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

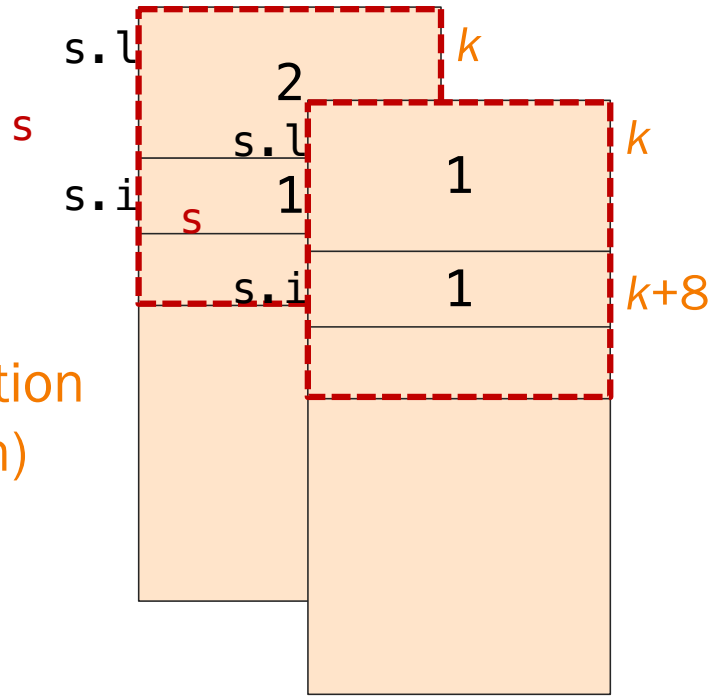
```
struct S {
  long l;
  int i;
};
```

Type

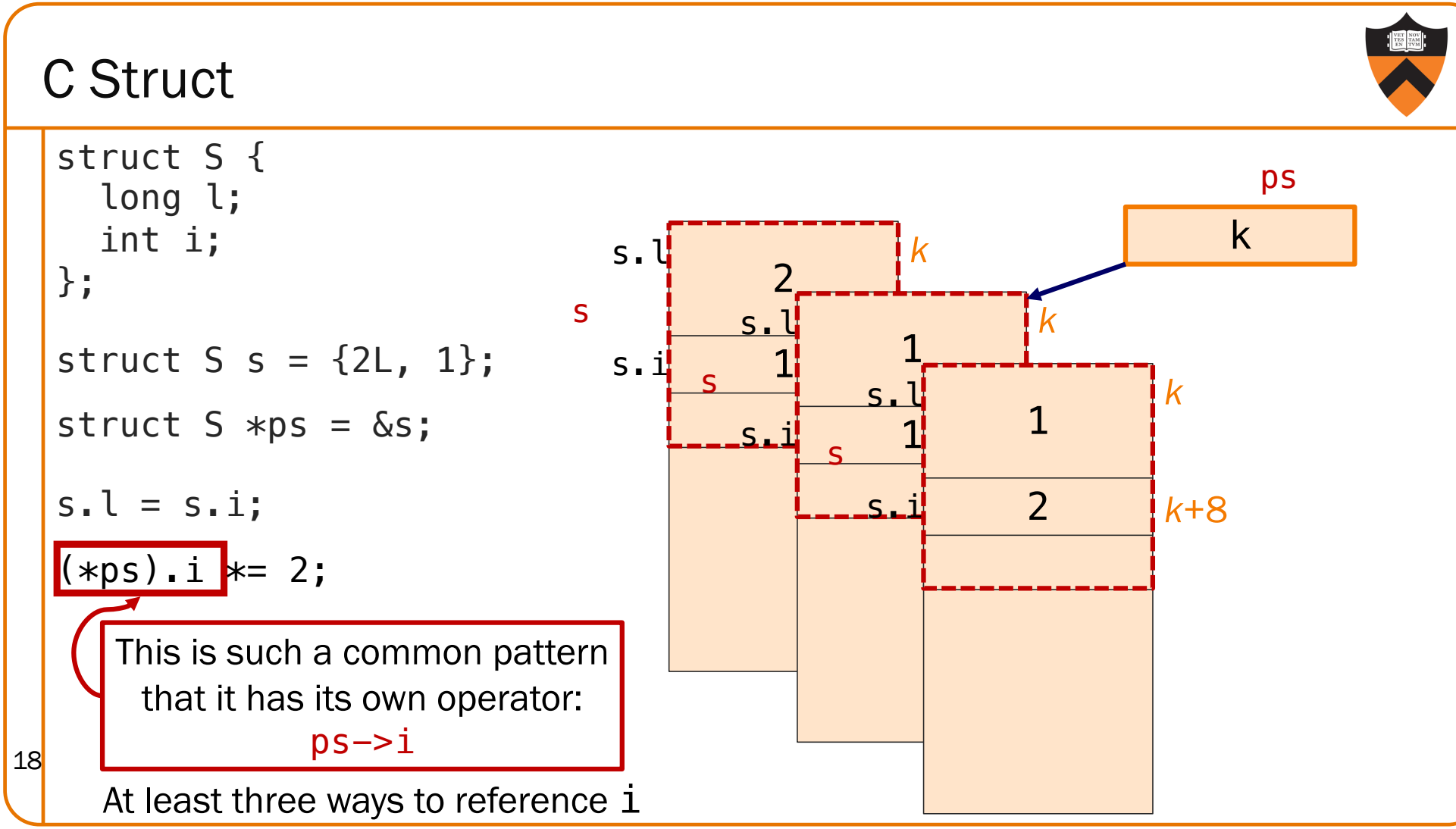
```
struct S s = {2L, 1};
```

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;
```

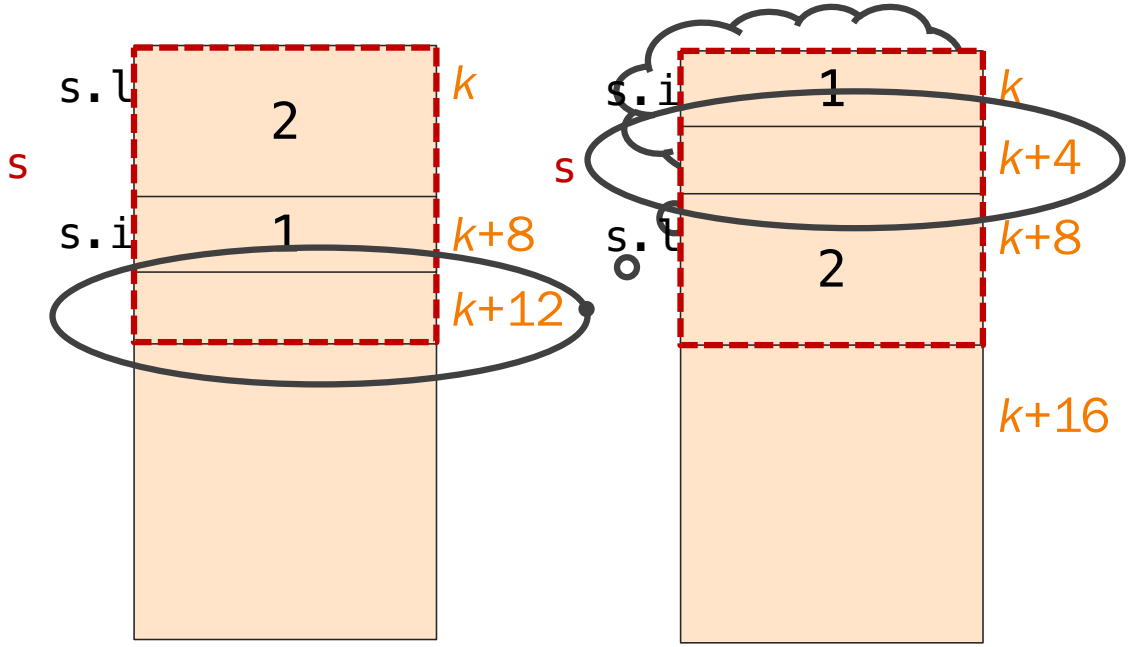

This is such a common pattern that it has its own operator:

`ps->i`

At least three ways to reference `i`

Interface and Implementation: Padding in Structs

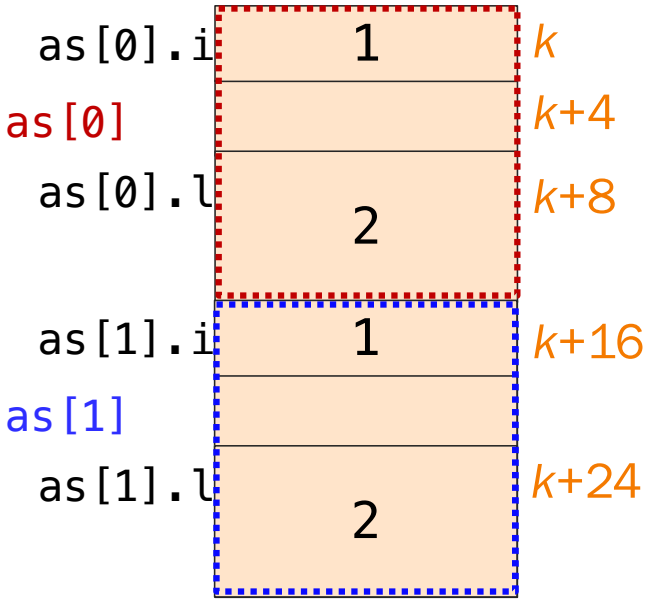

```
struct S {
  long l;
  int i;
};

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} };


as[1] = as[0];
```



- Assigning one struct variable to another makes a “deep copy” (copies the values)

Structs and 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

Structs and Functions

```
void printS(struct S s) {
  printf("%d %ld\n", s.i, s.l);
}
void swap1(struct S s) {
  int iTemp = s.l;
  s.l = s.i;
  s.i = iTemp;
}
struct S swap2(struct S s) {
  int iTemp = s.l;
  s.l = s.i;
  s.i = iTemp;
  return s;
}
void swap3(struct S *ps) {
  int iTemp = ps->l;
  ps->l = ps->i;
  ps->i = iTemp;
}
```

```
int main(void) {
   struct S s = {1, 2L};
   printS(s);

   swap1(s);
   printS(s);

   s = swap2(s);
   printS(s);

   swap3(&s);
   printS(s);
   return 0;
}
```

```
armlab01:~/Test$ ./sswap
1 2
1 2
2 1
1 2
```

Structs and Functions

```
struct S {
  int aiSomeInts[10];
};

void printS(struct S s) {
  int i;
  for (i = 0; i < 10; i++)
    printf("%d ", s.aiSomeInts[i]);
  printf("\n");
}
```

```
int main(void) {
  struct S s = { {0,1,2,3,4,5} };
  struct S s2 = s;
  printS(s2);
  return 0;
}
```

```
armlab01:~/Test$ ./a.out
0 1 2 3 4 5 0 0 0 0
```

How many int arrays are stored in memory?

A. 0: arrays in a struct aren't really arrays
B. 1: arrays are copied/passed as a pointer
C. 2: structs are copied on assignment
D. 3: C, plus structs are passed by value
E. Arrays can't be fields of a structure.

The correct answer is D.

Passing, returning, or assigning a structure with an array field copies the array by value (a deep copy)

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[]`
- This is often not feasible;
  memory needs are often dependent on runtime state
  - E.g. User input (number of students records)

```
How many records are being entered?
```

  - E.g. Reading from a resource (file, network, etc.)
  - E.g. Creating new nodes in a tree a threshold value is met

Dynamically Managed Memory Goes on the Heap

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

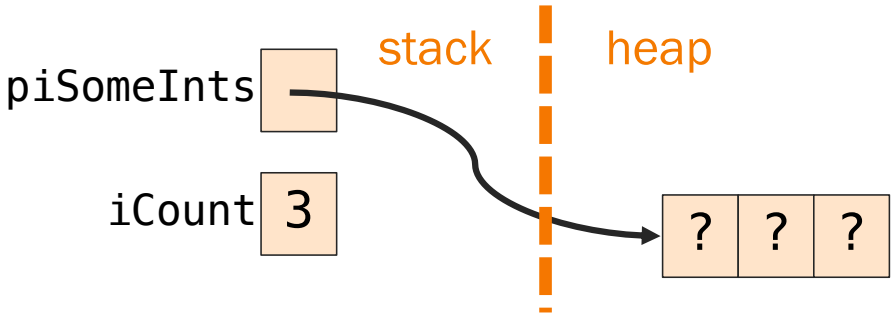
Stack

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

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

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

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



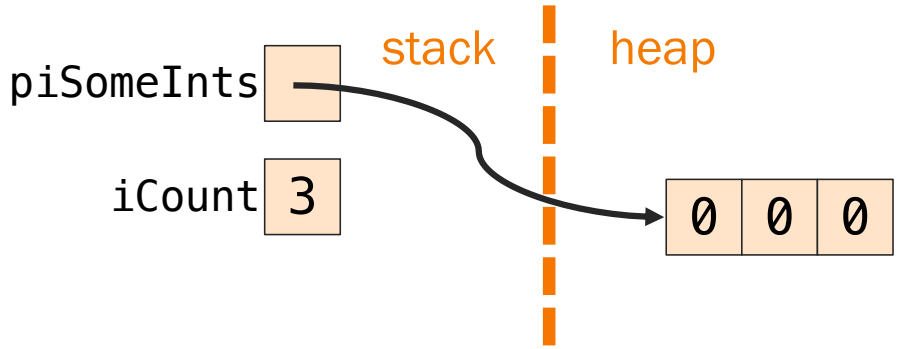
Interfaces and implementations: Use `sizeof`.

`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));
```

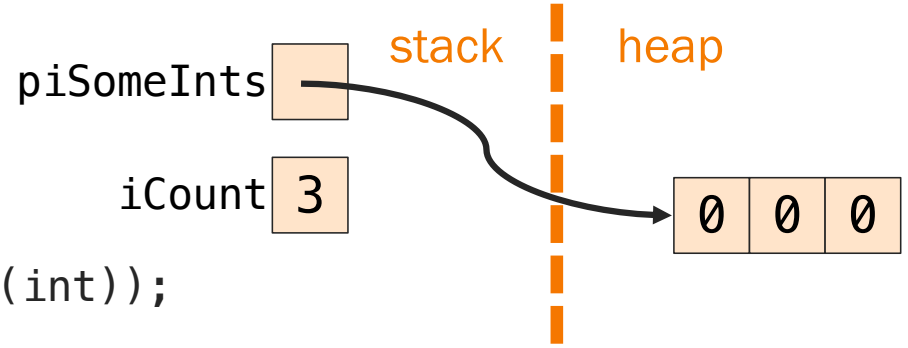
```
int iCount;
int *piSomeInts;
printf("How many ints?");
scanf("%d", &iCount);
piSomeInts =
   calloc(iCount, sizeof(int));
```



calloc() initializes data to 0

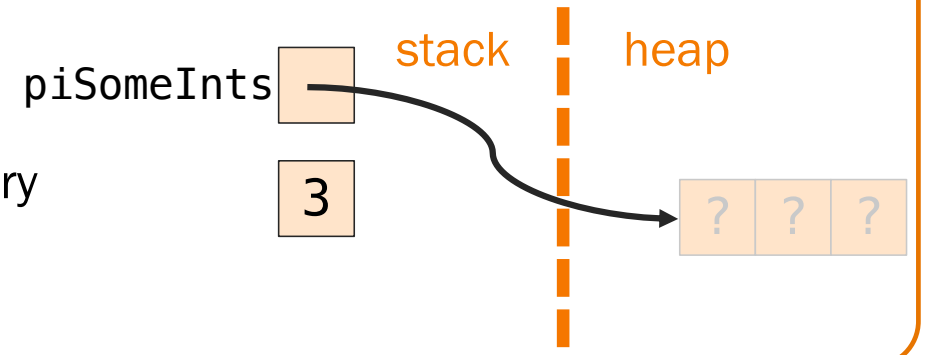
Your New Friends: `malloc`, `calloc` and `free`

```
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?

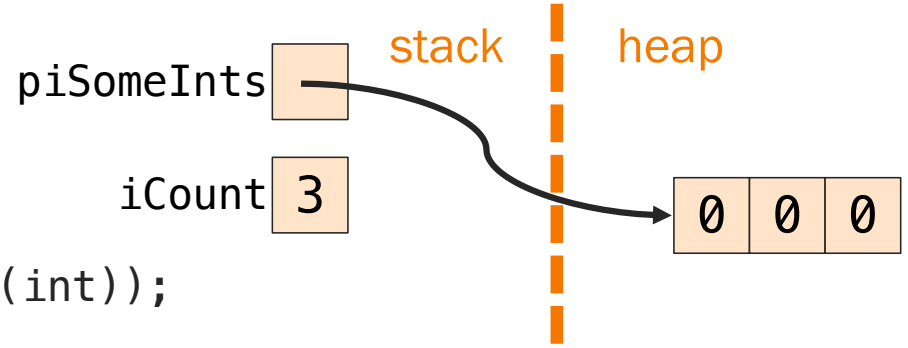
```
free(piSomeInts);
```

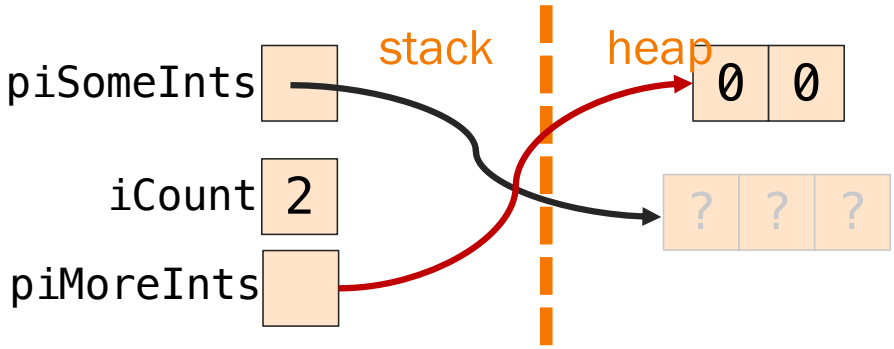


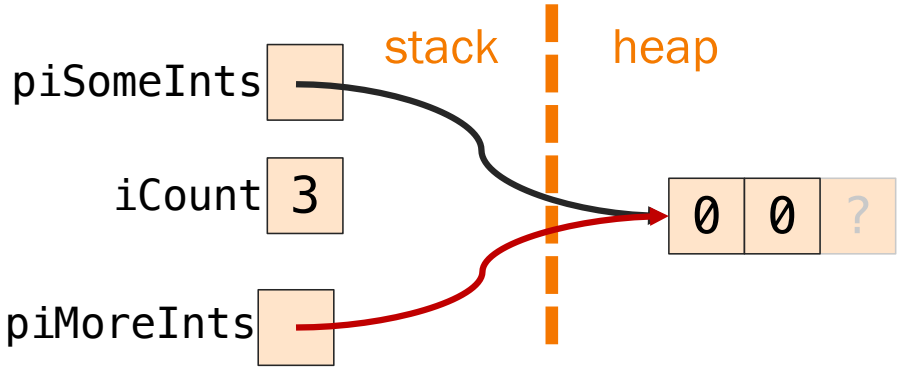
`piSomeInts` keeps pointing to the memory
Why?
Hmmm…. "Dangling pointer"

Your New Friends: realloc

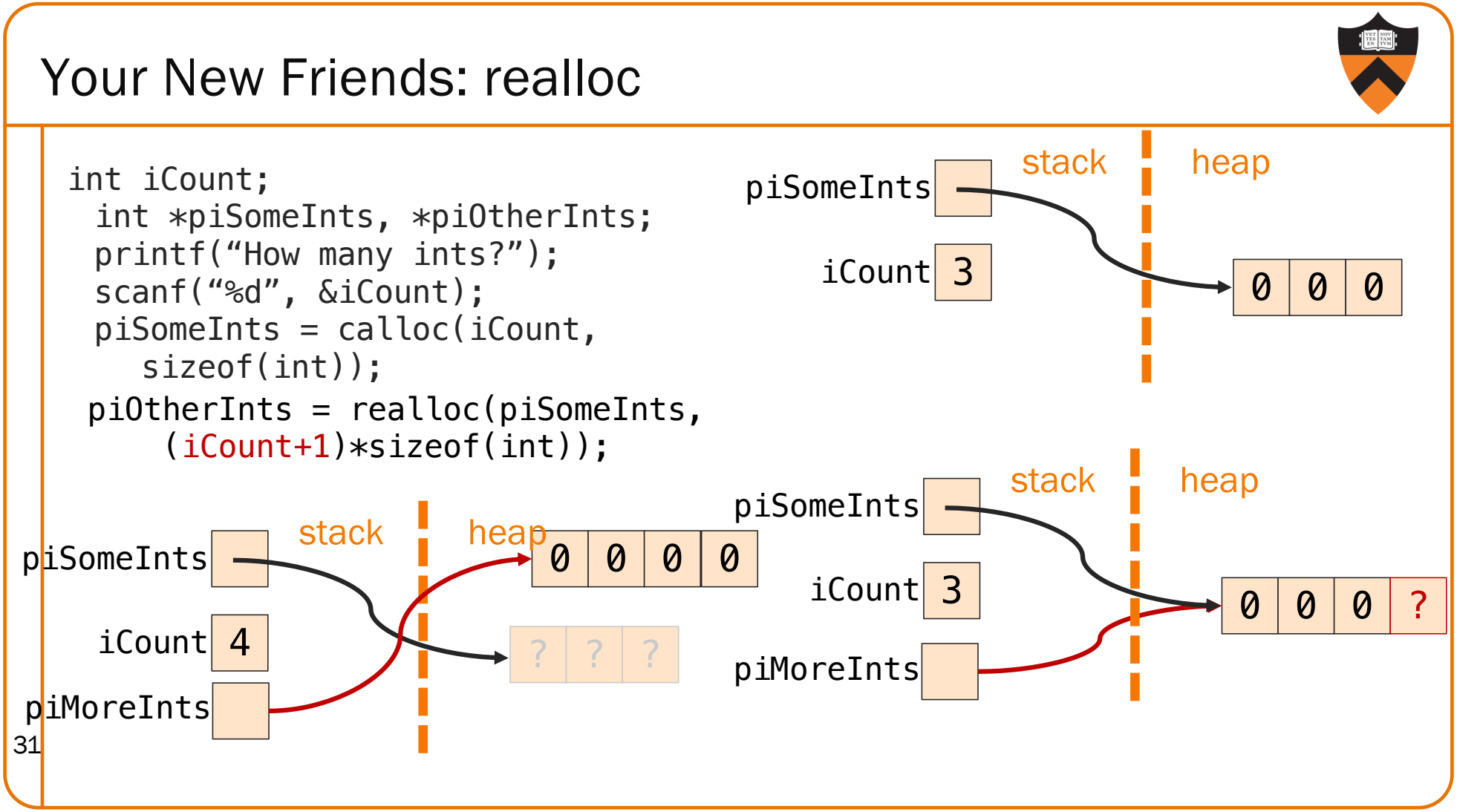
```
int iCount;
int *piSomeInts, *piMoreInts;
printf("How many ints?");
scanf("%d", &iCount);
piSomeInts = calloc(iCount, sizeof(int));
piMoreInts = realloc(piSomeInts,
     (iCount-1)*sizeof(int));
```







Your New Friends: realloc



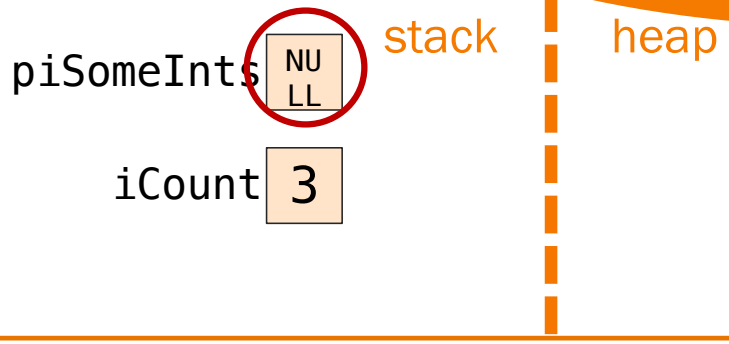
```
int iCount;
  int *piSomeInts, *piOtherInts;
  printf("How many ints?");
  scanf("%d", &iCount);
  piSomeInts = calloc(iCount,
     sizeof(int));
 piOtherInts = realloc(piSomeInts,
     (iCount+1)*sizeof(int));
```

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?

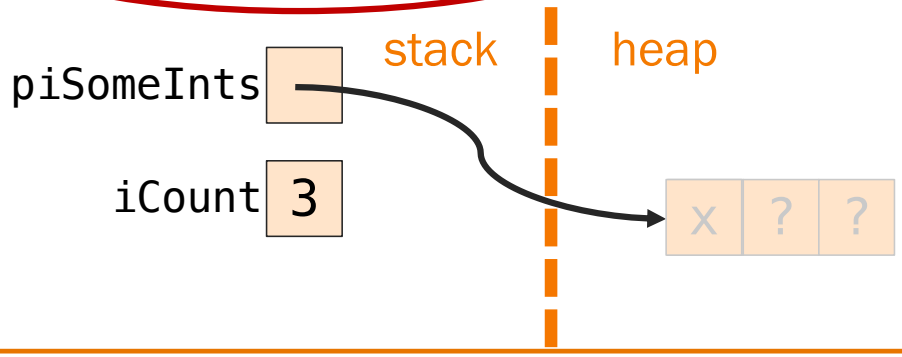


What Could Go Wrong (free)?

```
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?

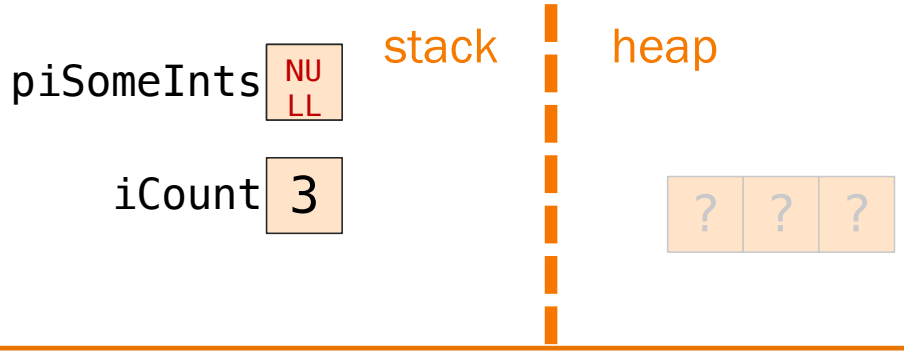


What Could Go Wrong (free)?

```
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);
```

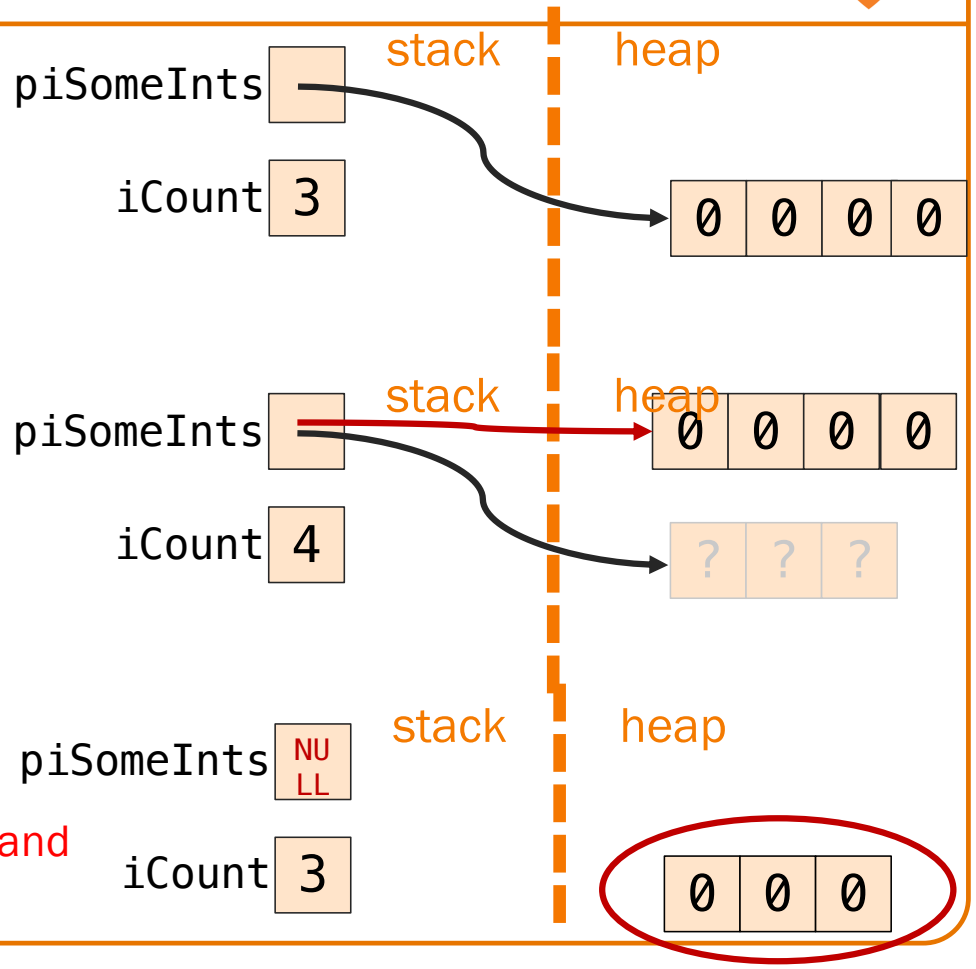
Will crash. But this is a bug, so that's good

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

Solution: realloc to temp pointer, check NULL, and only then update original pointer accordingly

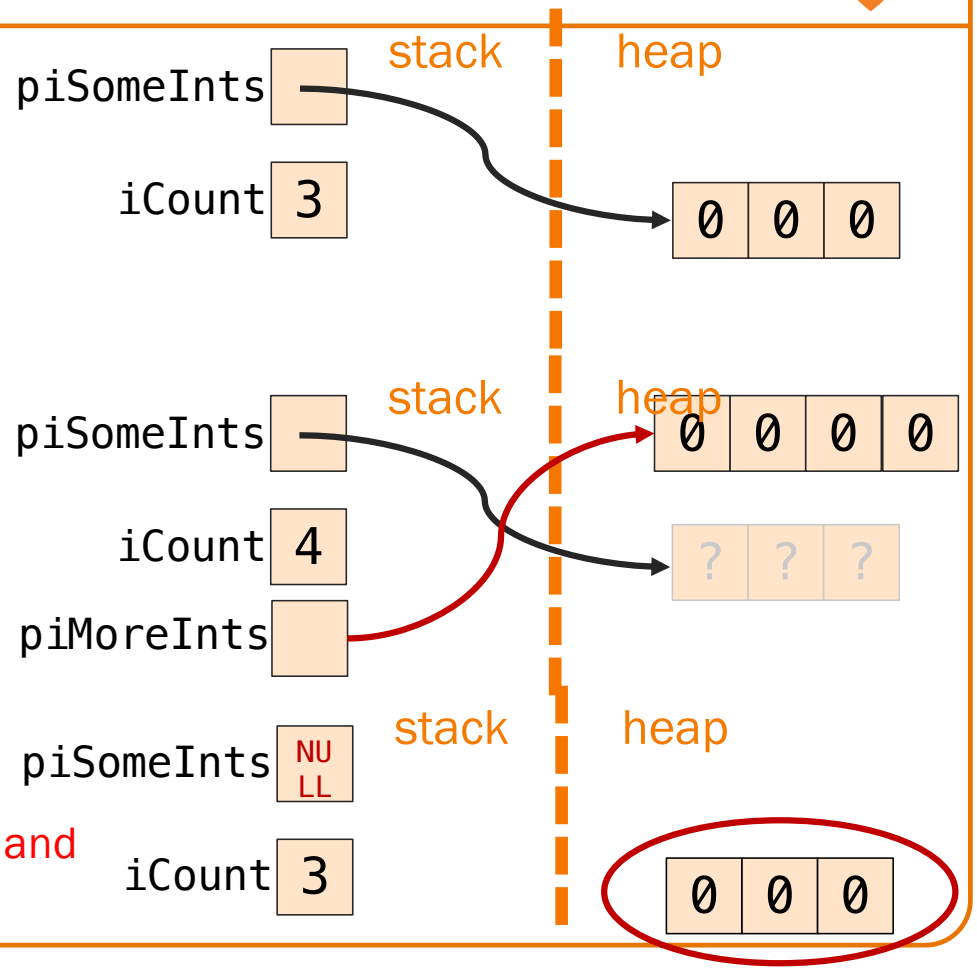
What could go wrong: realloc

```
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

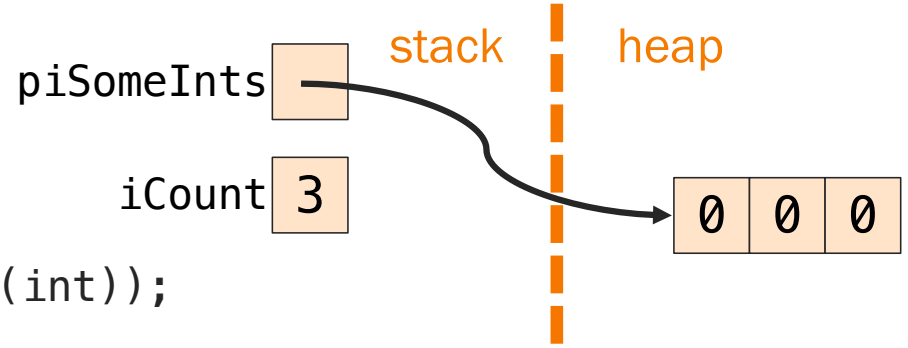
Regardless, if realloc fails: memory leak

Solution: realloc to temp pointer, check NULL, and only then update original pointer

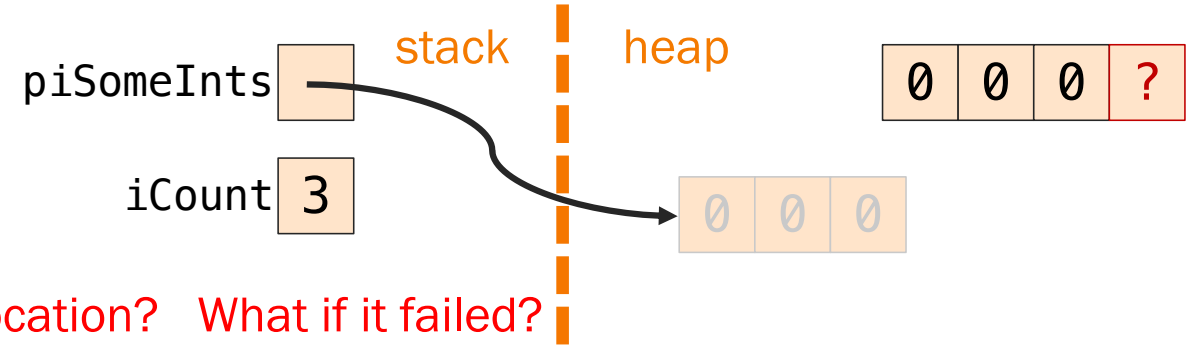


What could go *really* wrong: realloc

```
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)...
```



Memory Leak

Dangling Pointer

Likely eventual double free

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

Catch the Common Bug

```
newCopy = malloc(strlen(oldCopy));
strcpy(newCopy, oldCopy);
```

Does this work?

A. Totally.

B. Nope. The bug is …

B:

This allocates 1 too few bytes for `newCopy`, because `strlen` doesn't count the trailing `'\0'`

Save a line?

```
newCopy = strcpy(malloc(strlen(oldCopy)+1), oldCopy);
```

Does this work?

A. So *that's* why `strcpy` returns the destination. Sure

B. Eh, okay, but this is less clear.

C. Nope

C:

If `malloc` returns NULL, this fails the precondition for `strcpy`

(This was also an issue on the previous slide.)

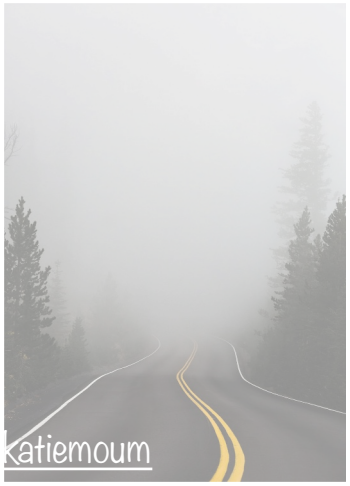
Check for `malloc` returning NULL first, so keep it on separate line

Don't get ahead of yourself …

Assignment 2 does **NOT** use dynamic memory

- Assignments 3 and 4 will use it extensively
- We will not test it on the midterm

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(⓪ int argc, ① char** argv) {
② int a[10] = {-1, 0, 1};
③ double x = 10.75;
④ double* px = &x;
⑤ char* s;
⑥ char* f = ⑦ "¥"%s¥"¥n";
s = ⑧ calloc(*px, sizeof(*s));
printf(f, s);
return strlen(s);
}
```

a. 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`.

b. What does this program print to standard output?

c. 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`.)