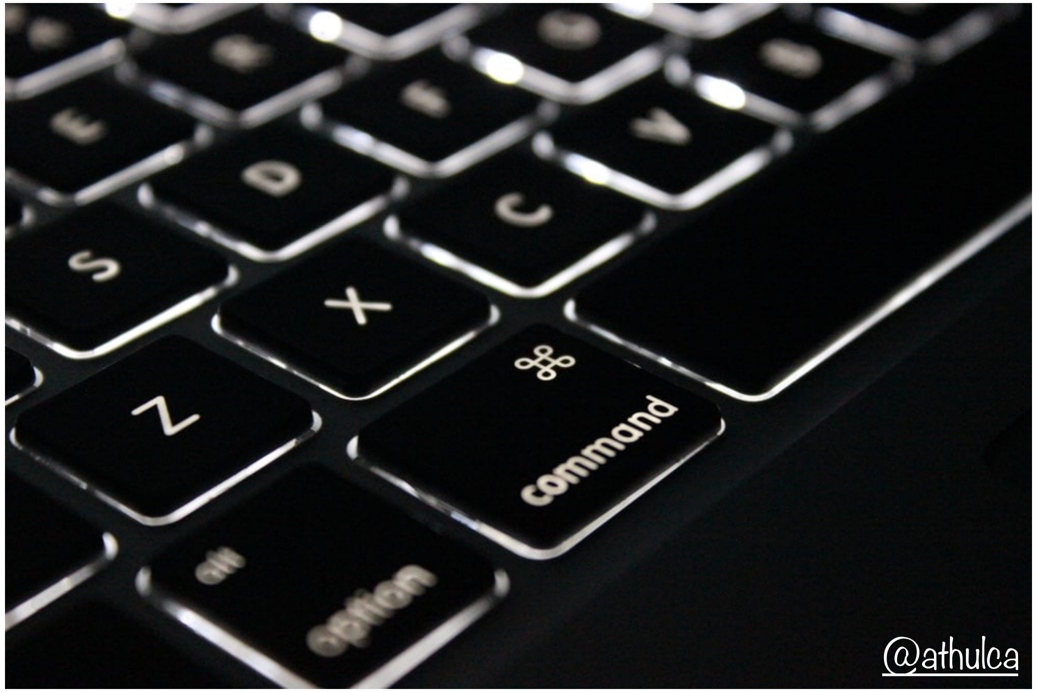
COS 217: Introduction to Programming Systems

Command Line Arguments,
Structures,
Dynamic Memory

PRINCETON UNIVERSITY



COMMAND LINE ARGUMENTS

What’s my name?

- String[] args was COS 126 day 1

- How to get the equivalent in C?

With sed `s/s/v/`, natch.

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


   /* Write the command-line argument count to stdout. */
   printf("argc:  %d\n", argc);

   /* Write the command-line arguments to stdout. */
   for (i = 0; i < argc; i++)
      printf("argv[%d]:  %s\n", i, argv[i]);


   return 0;
}
```

As parameters, these are identical:

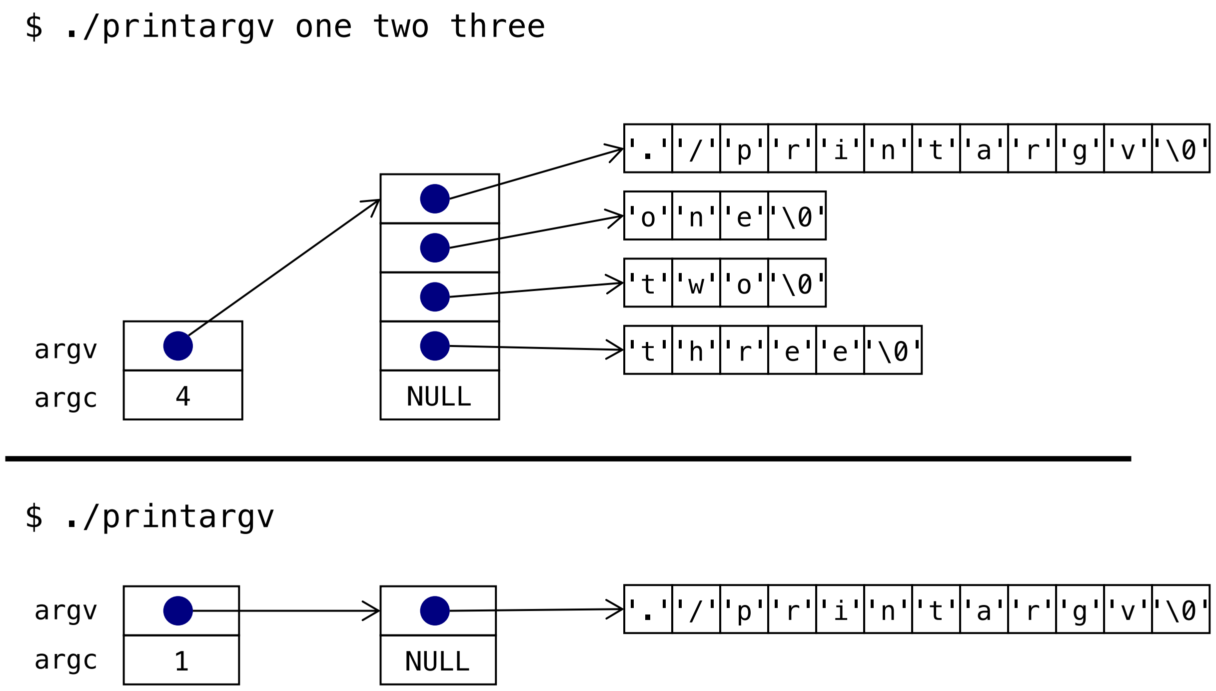
`char a[]` and `char *a`

So it follows that, as parameters, these are, too:

`char *argv[]` and `char **argv`

Elucidating Example: Explanatory echo

```
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;
}
```

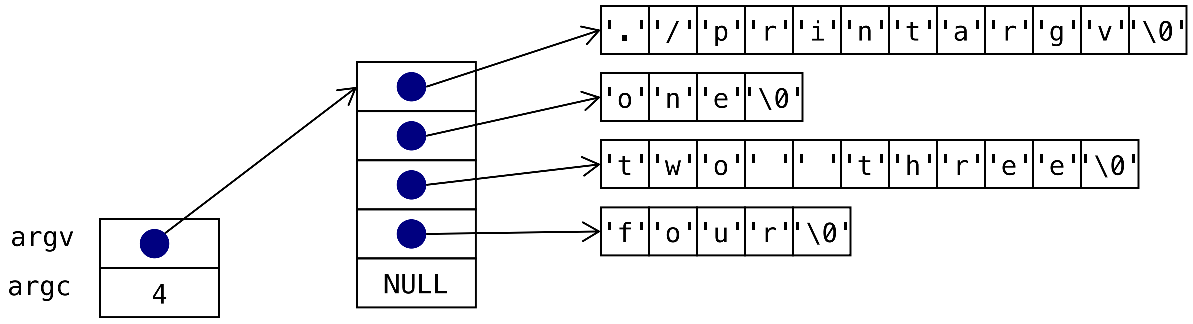


What's argc?

```
./printargv one "two  three" four
```

B:

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



A. 3

B. 4

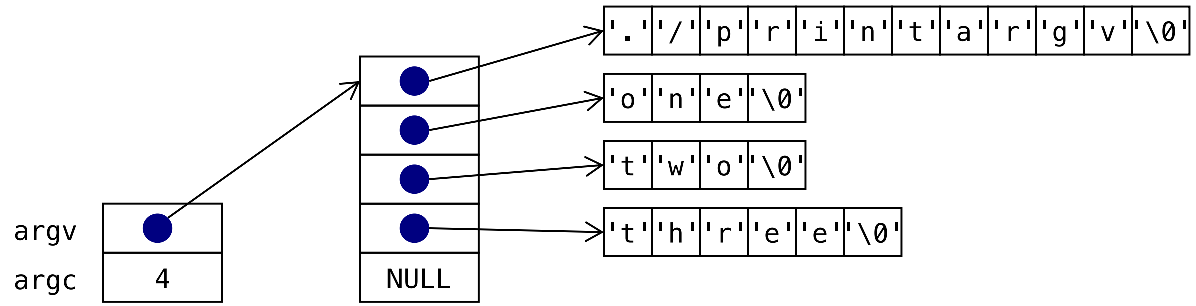
C. 5

D. Syntax error at runtime

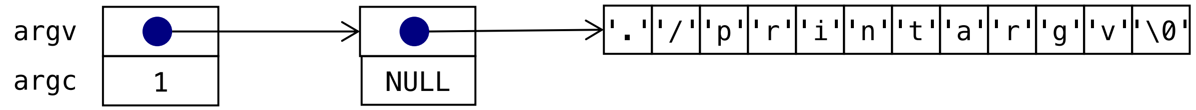
A2-inspired: rewrite everything in arrays to use pointers

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

  while (*ppc != NULL)
    printf("argv[%d]:  %s\n",
                   ppc-argv, *ppc++);
  return 0;
}
```

`$ ./printargv one two three`



`$ ./printargv`



Post-increment with pointers is just like we saw with integer types: emits the **old** value of `ppc`

The dereference then happens on value emitted, so this could be parenthesized as: `*(ppc++)`

Kicking the extra point?

```
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;
}
```

?

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

   while(pc != NULL)
     printf("argv[%d]:  %s\n", i++,
                          pc++);
   return 0;
}
```

A. Yes! This works and is clearer.

B. Maybe. This works but is less clear.

C. No! This is incorrect!

D. No! This doesn't even compile!

C: When run with no arguments:

```
argc: 1
argv[0]: ./pcla-wrong
argv[1]: /pcla-wrong
argv[2]: pcla-wrong
argv[3]: cla-wrong
...
```

Challenge for the bored: mainly nonsense

```
int main(int argc, char **argv) {
  int retVal;
  if (argc == 0) {
    return 0;
  } else {
    retVal = main(argc-1, argv+1);
    printf("%d: %s\t", argc-1, argv[0]);
    return retVal;
  }
}
```

What does this program do?

A. prints arguments

B. prints arguments in reverse order

C. recurs infinitely: `argc` is always ≥ 1

D. prints only the last argument: `return` from `main` exits the program

The correct answer is B:

```
armlab01:~/Test$./recur-r a b c; echo
0: c 1: b 2: a 3: ./recur-r
```

C is only the case at the start of execution,
and does not hold if the program changes `argc`.

D would be the behavior with `exit(retVal);` instead of `return retVal;`

C STRUCTURES

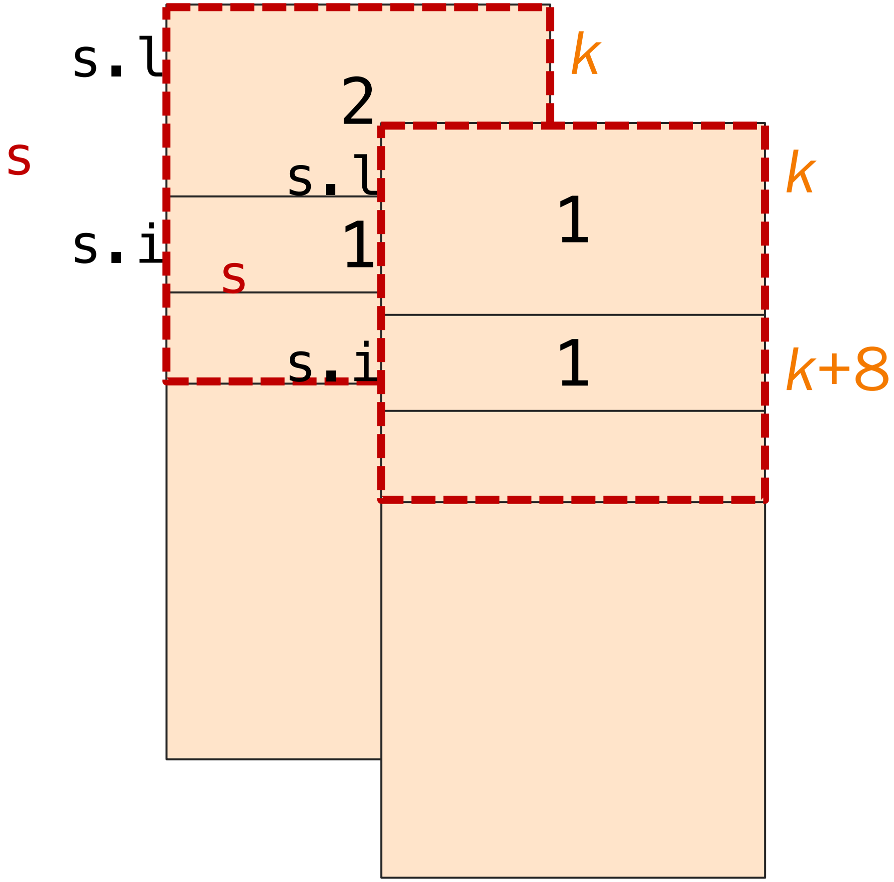
{new state, updated line number} would've worked

- Java classes can have many fields

- How to get the equivalent in C?

Add some structure to your program

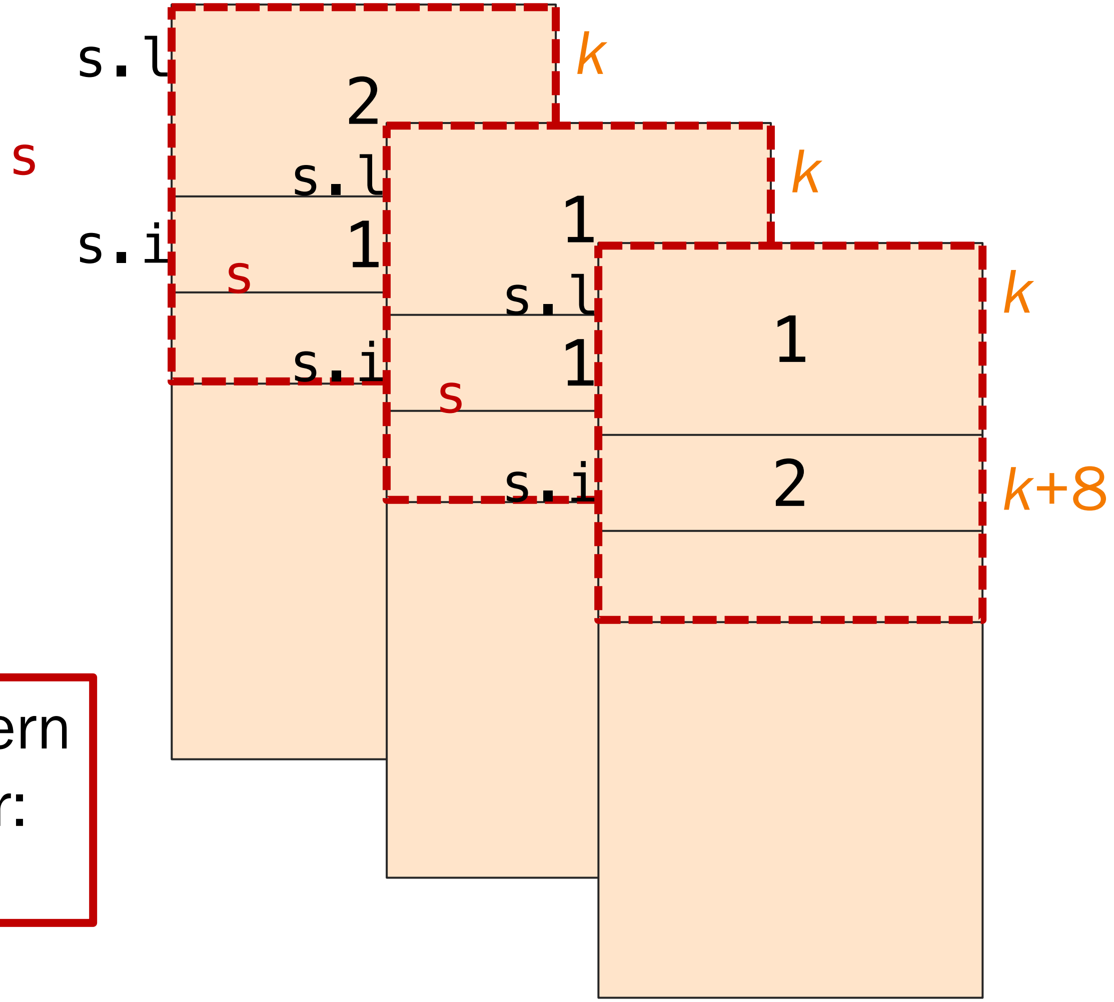
```
struct S {
  long l;
  int i;
};

struct S s = {2L, 1};


s.l = s.i;
```



Add some structure to your program

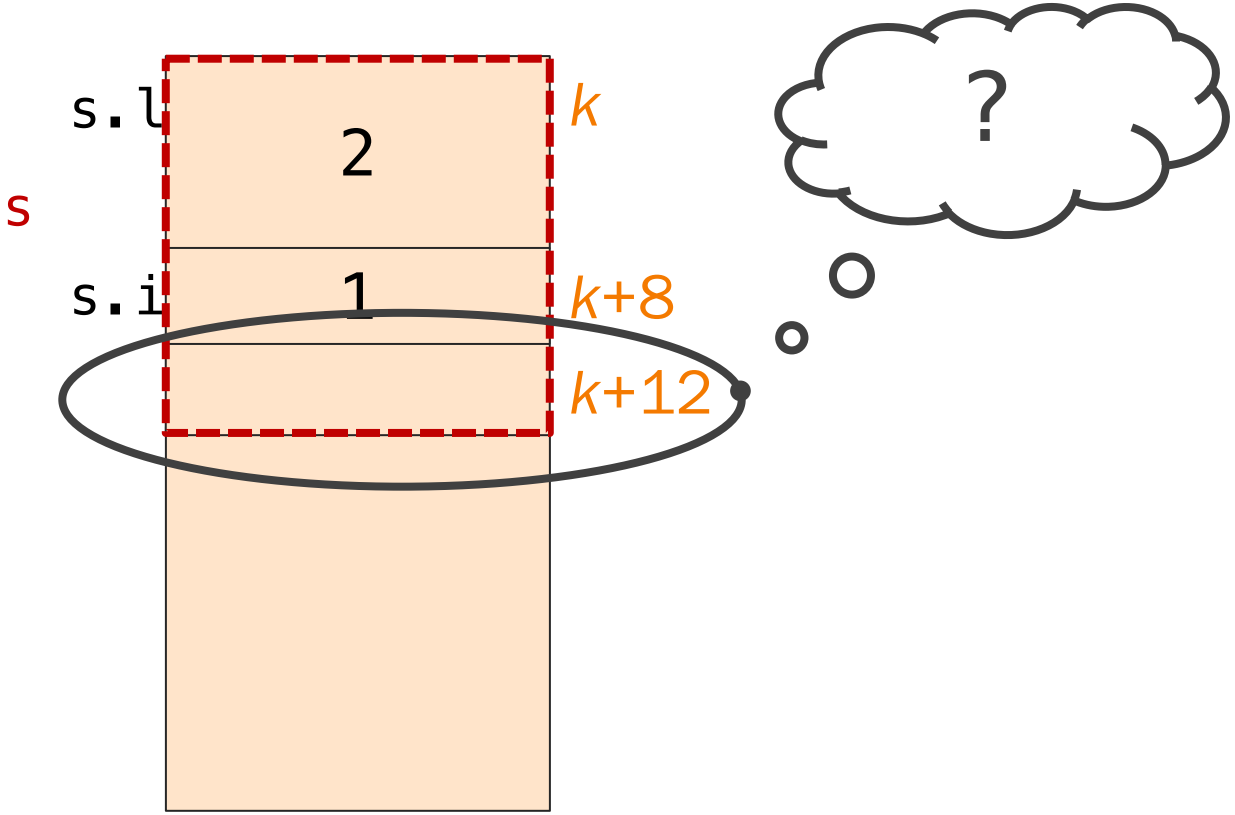
```
struct S {
  long l;
  int i;
};

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`



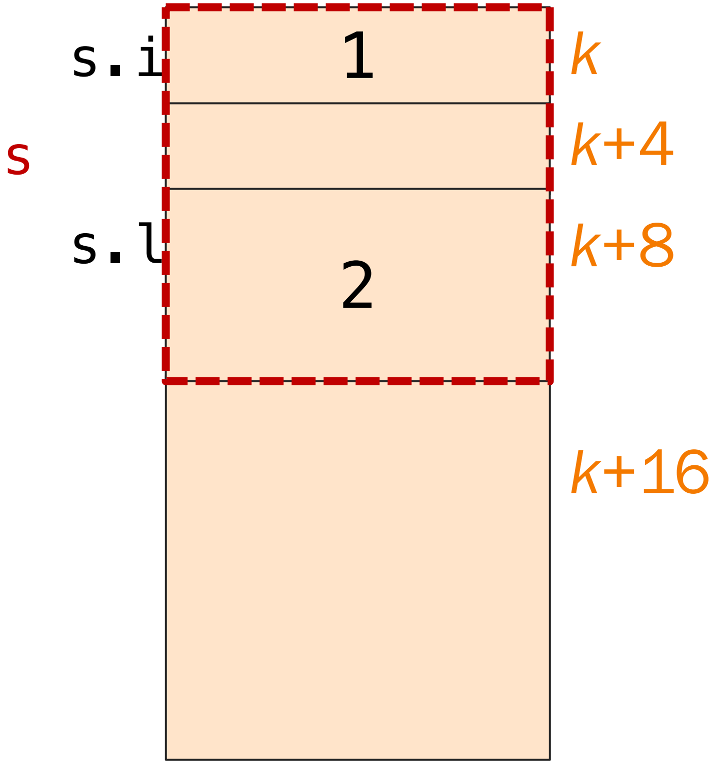
struct instruction

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

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



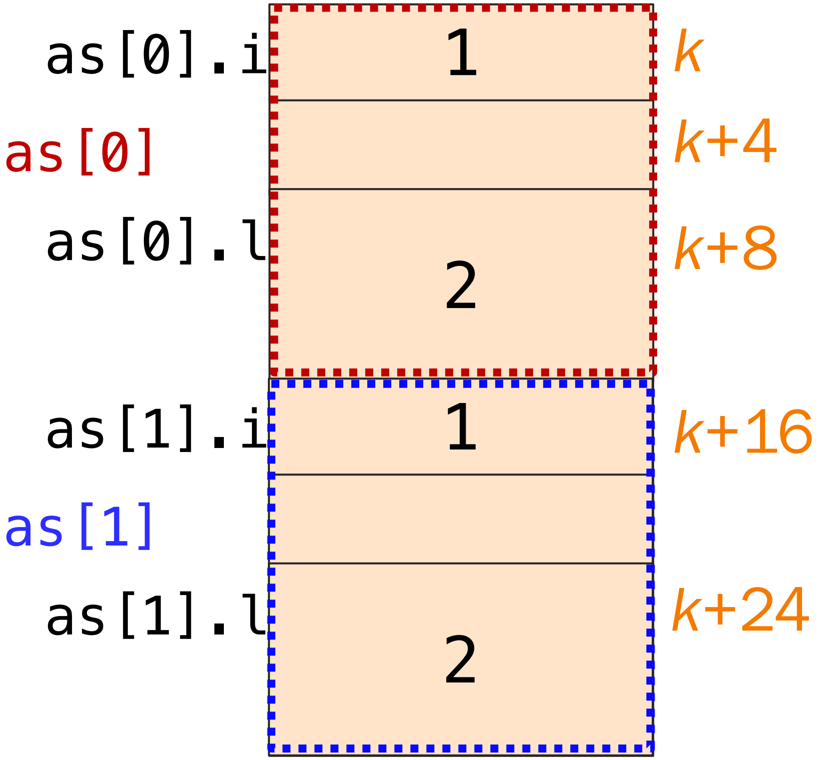
Internal Padding

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

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



structstructstructstructstructstruct

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

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


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



struct construction, what's your function?

```
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
```

Whose Rules Rule?

```
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$ ./sa
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.

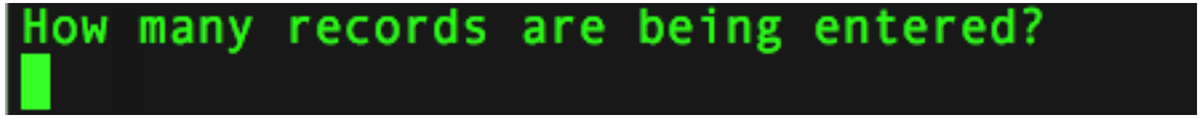
The correct answer is D.

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

DYNAMIC
MEMORY

Why, though?

- Thus far, all memory that we have used has had to be known at compile time.
- This is not feasible for realistic workloads; many times memory needs are dependent on runtime state
  - User input
  - Reading from a resource (file, network, etc.)
  - …



Memory Allocation at Runtime

Thus far we have seen 3 memory sections:

Stack

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

Text

- Program machine language code

RODATA

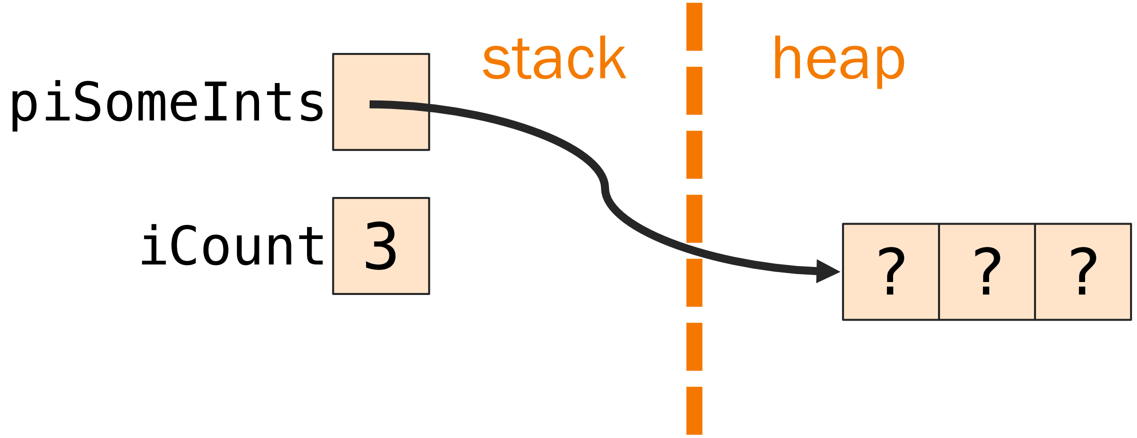
- Read-only data, e.g. string literals

Now, a 4th: the “Heap”

Your New Friends: malloc

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

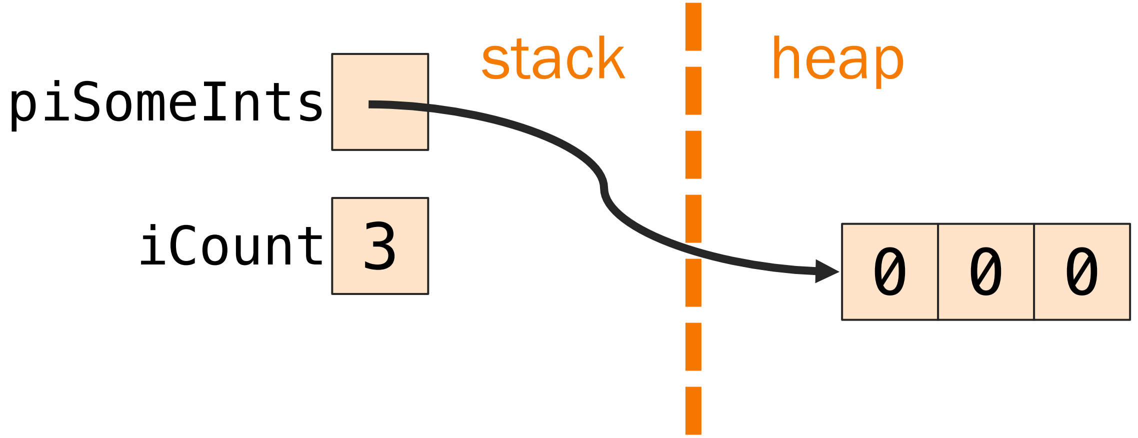
```
int iCount;
int *piSomeInts;
printf("How many ints?");
scanf("%d", &iCount);
piSomeInts =
   calloc(iCount,
           sizeof(int));
```



Your New Friends: calloc

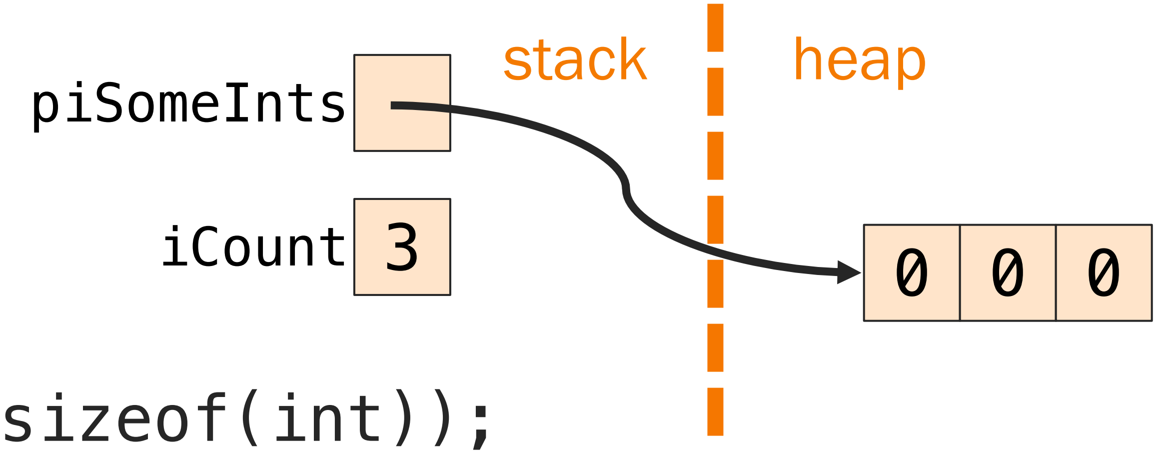
```
int iCount;
int *piSomeInts;
printf("How many ints?");
scanf("%d", &iCount);
piSomeInts =
   malloc(iCount *
          sizeof(int));
```

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

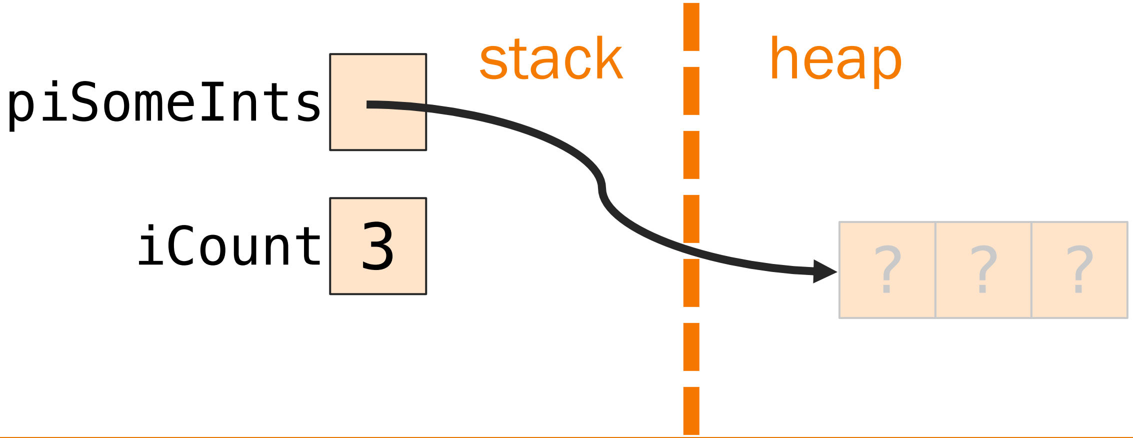


Your New Friends: free

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

free(piSomeInts);
```



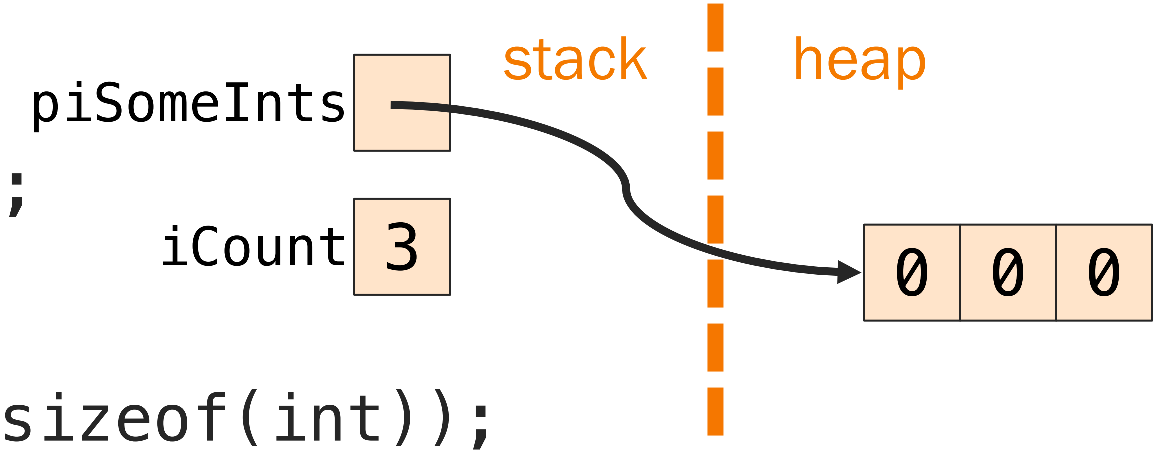
Before



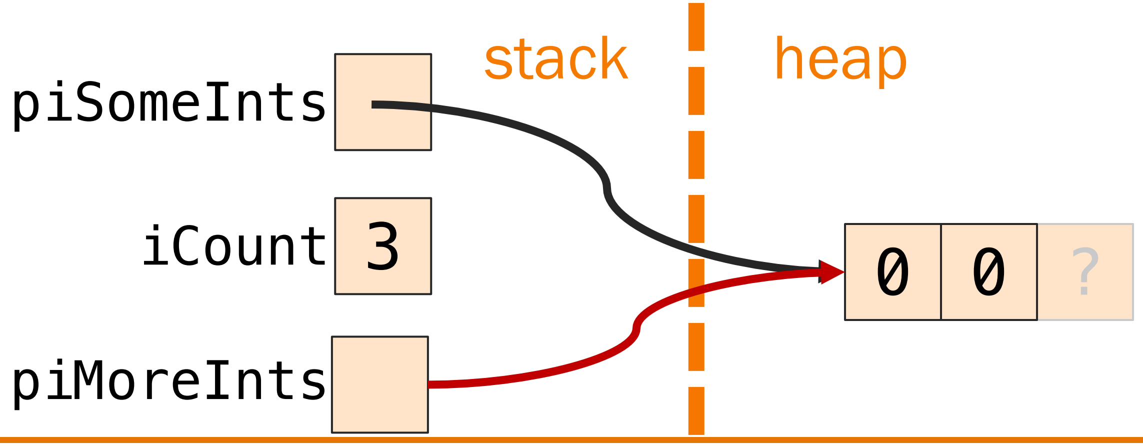
After

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));
```



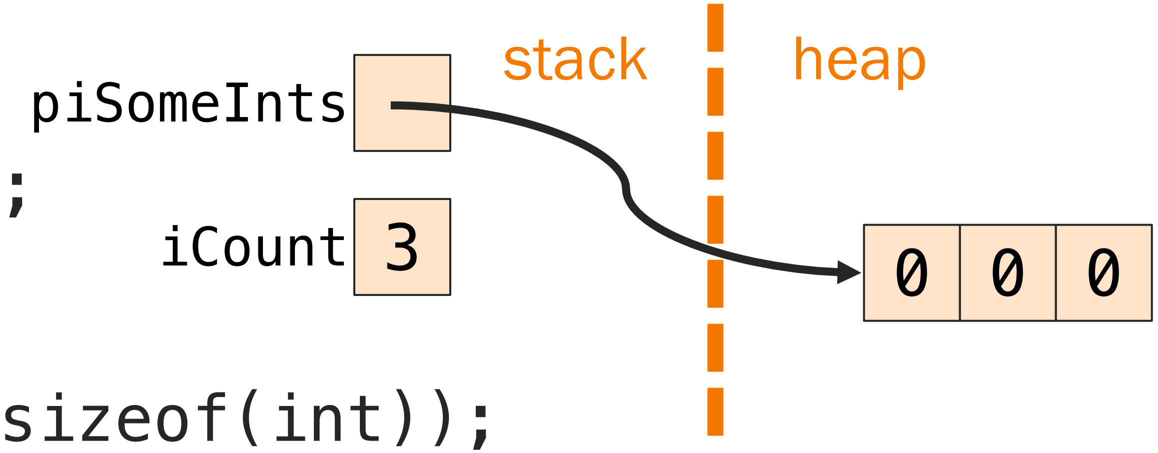
Before



After
(typically, and definitely on armlab, but not guaranteed by the C standard)

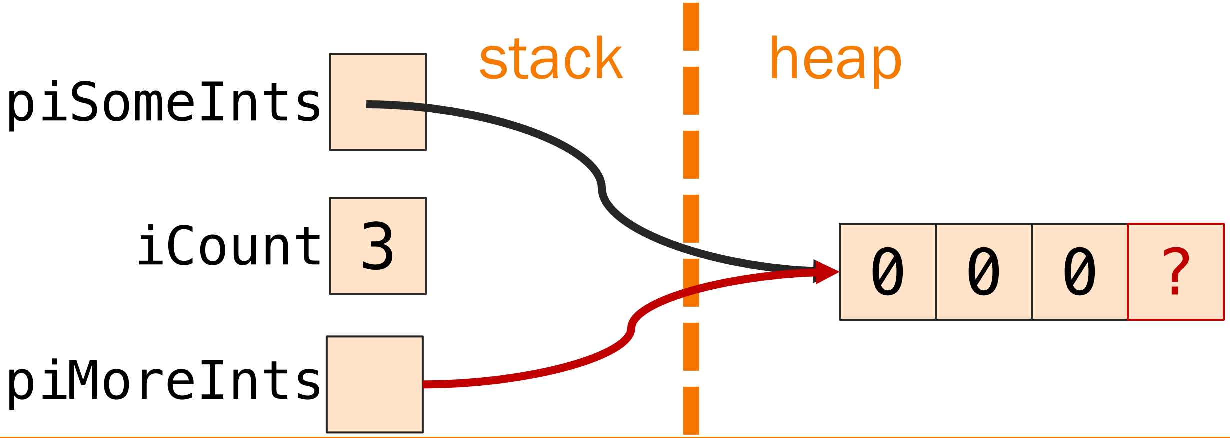
Your New Friends: realloc

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



Before

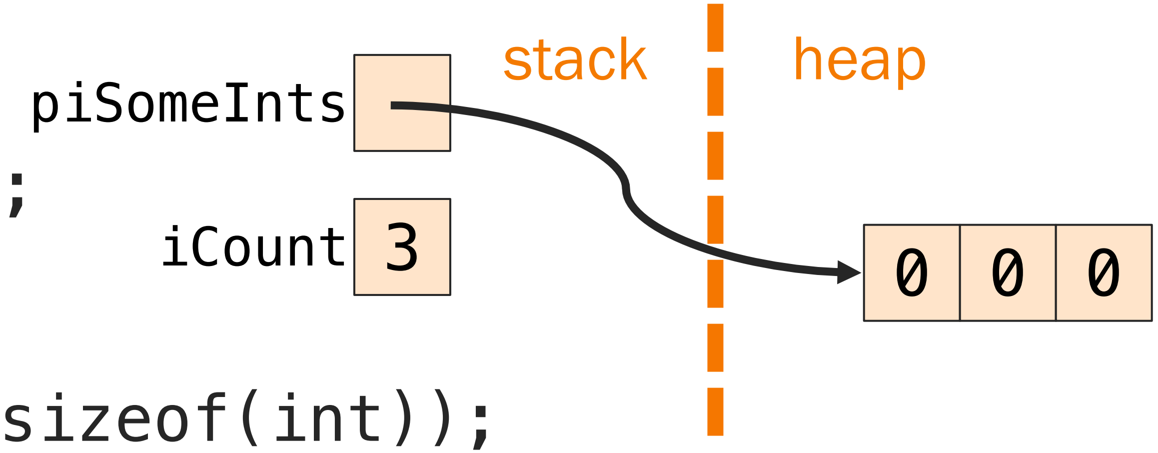
```
piMoreInts = realloc(piSomeInts, (iCount+1)*sizeof(int));
```



After
(typically, but not guaranteed,
especially if instead of (iCount+1)
you want, say, 2^{iCount})

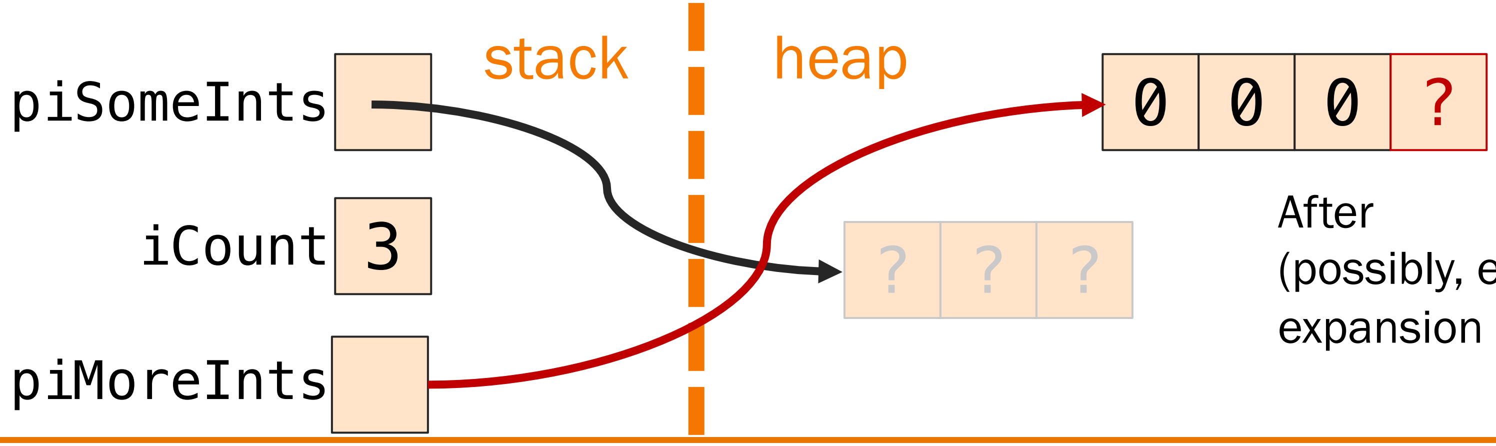
Your New Friends: realloc

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



Before

```
piMoreInts = realloc(piSomeInts, (iCount+1)*sizeof(int));
```



After
(possibly, especially if the expansion is large)

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] = ...
```

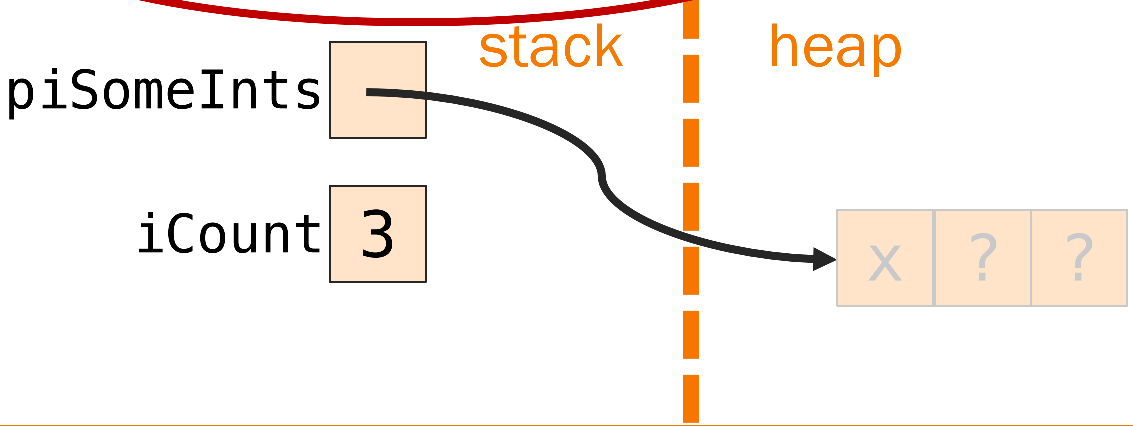
stack | heap

piSomeInts NULL

iCount 3

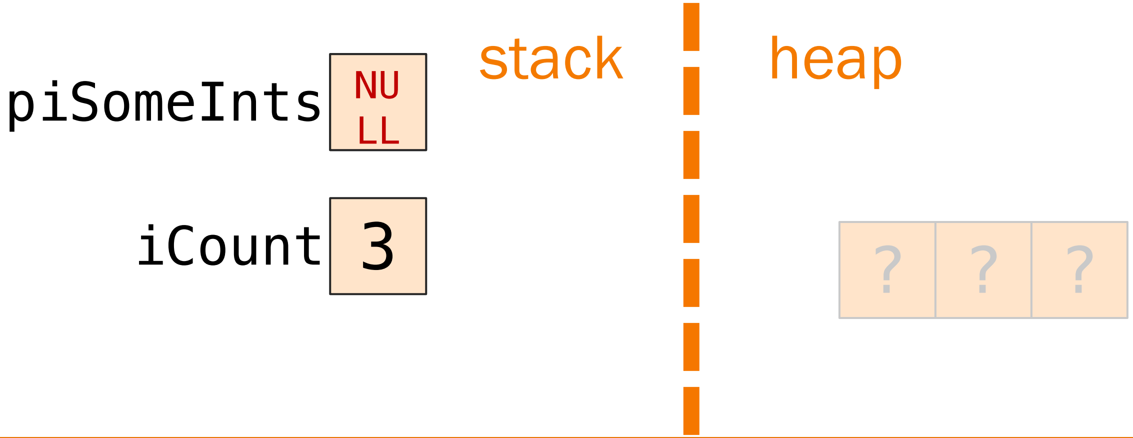
What could go wrong (free)?

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



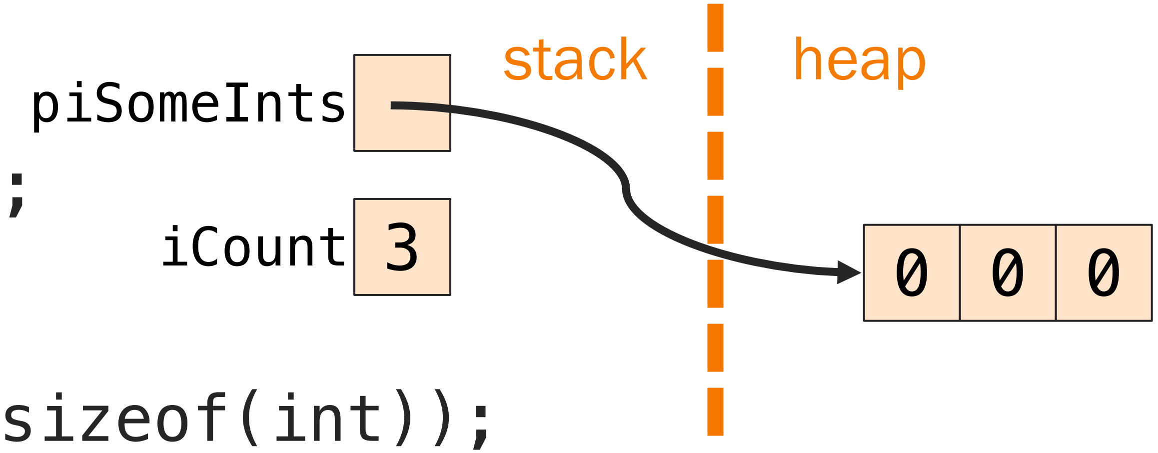
It’s still a bug! (But now you’ll find it "easily"!)

```
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);
```

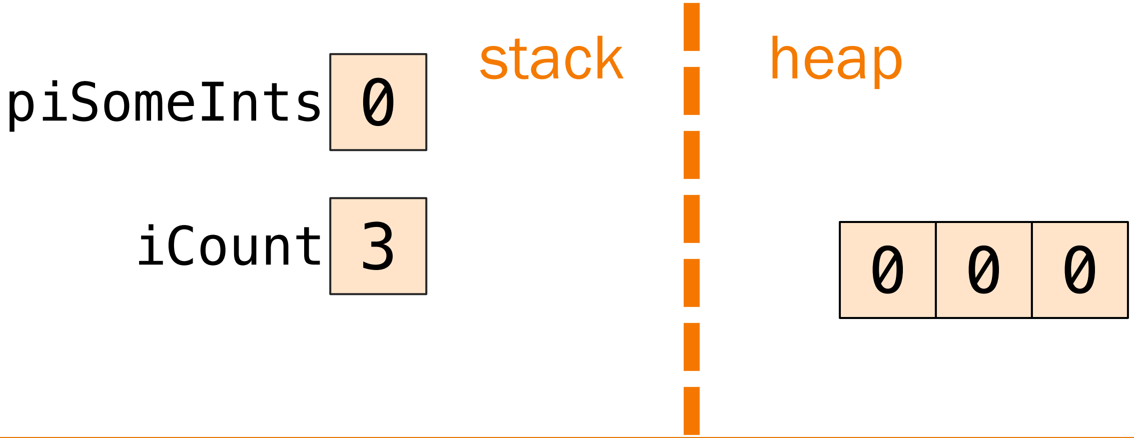


What could go wrong: realloc

```
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)...
```



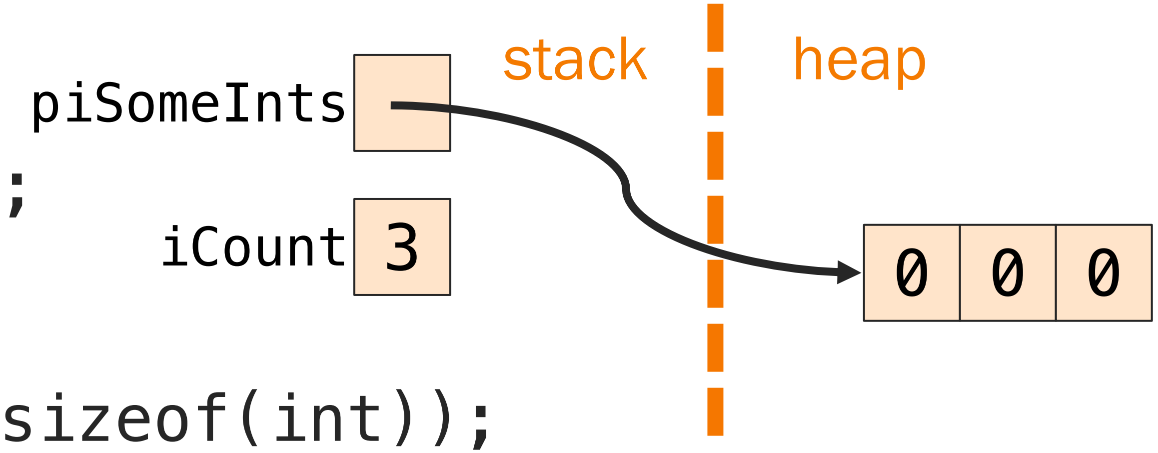
Before



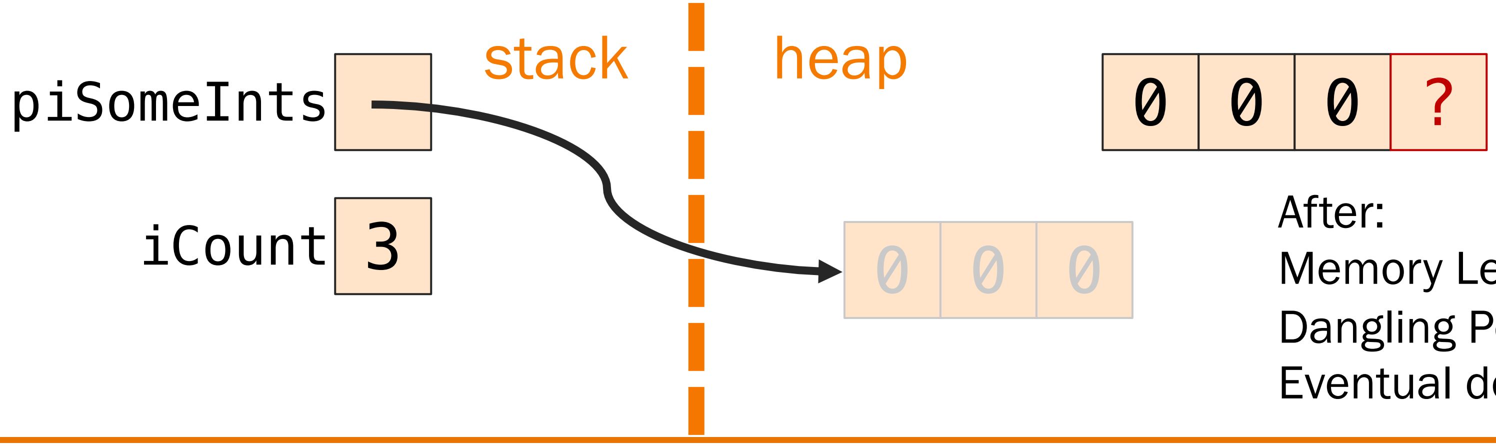
After:
If realloc returns NULL,
Memory Leak
NULL pointer dereference if not checked

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)...
```



Before



After:
Memory Leak,
Dangling Pointer,
Eventual double free.

Catch the Most Common Bug

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

Does this work?

A. Totally! (Wait, what's the title of this slide again?)

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.)

Don't get ahead of yourself!

Assignment 2 does **NOT** use dynamic memory!

Assignments 3 and 4 will use it extensively.
The topic **is** fair game for the midterm.

But **DO NOT** use `{m,c,re}alloc` + `free` on A2!