COS 217: Introduction to Programming Systems

Structures,
Command Line Arguments,
Dynamic Memory

PRINCETON UNIVERSITY

C STRUCTURES

- Java classes can have many fields

@ericgilkes
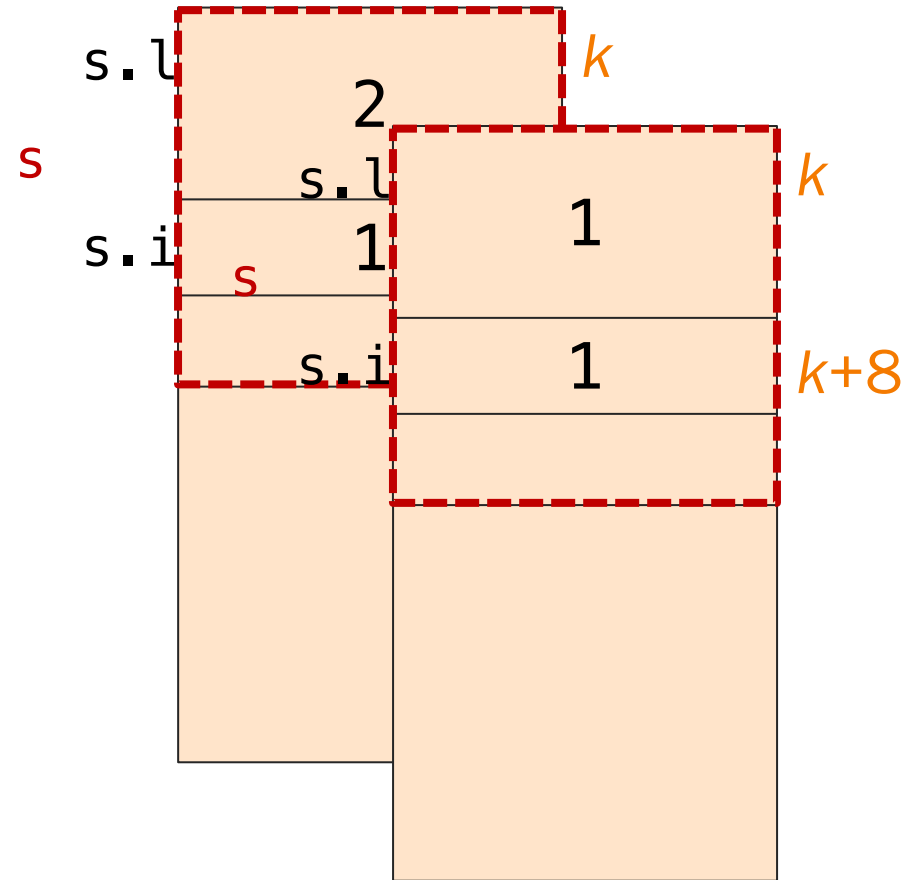
@simonesecci

- How to get the equivalent in C?

Add some structure to your program

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

struct S s = {2, 1};

s.l = s.i;
```
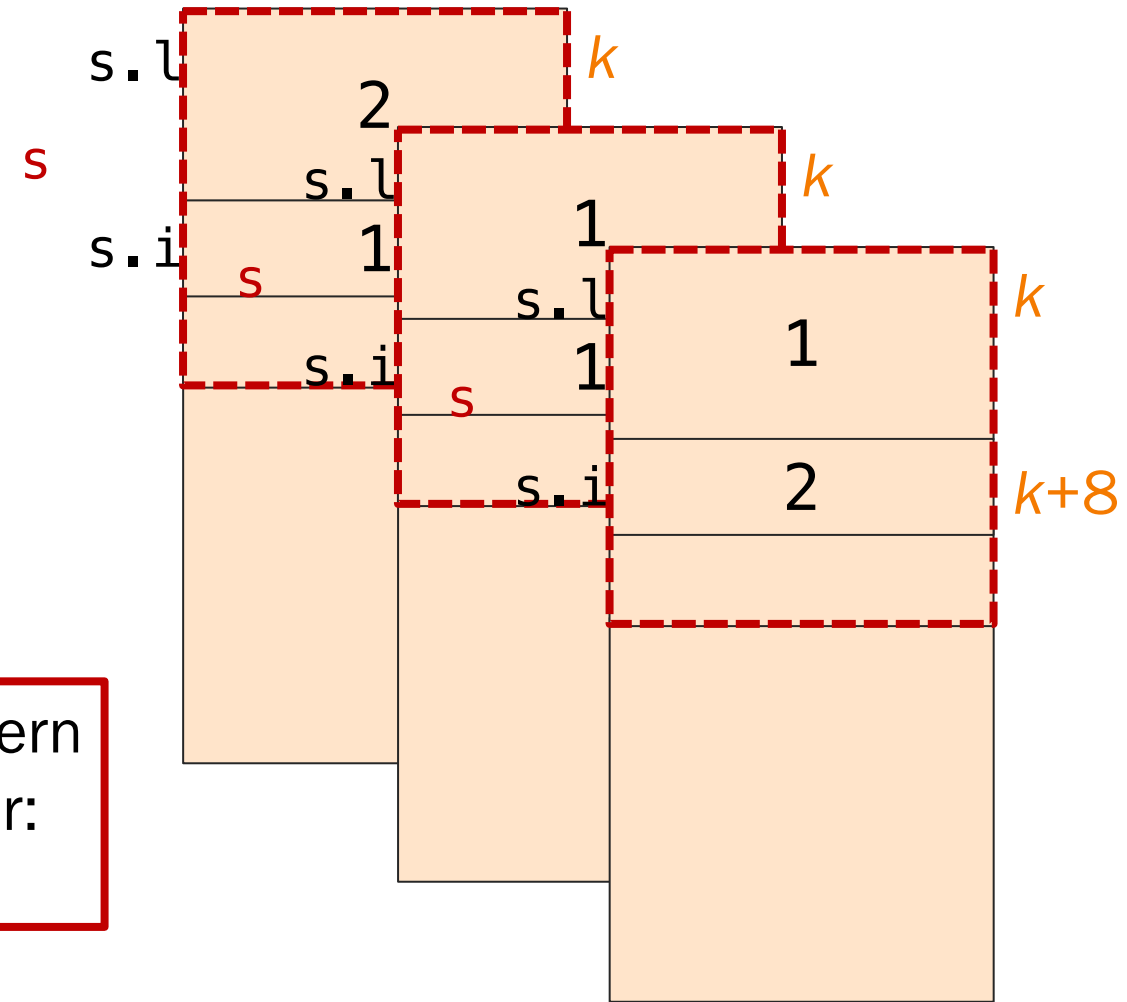
Add some structure to your program

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

struct S s = {2, 1};
struct S *ps = &s;

s.l = s.i;

(*ps).i *= 2;
```
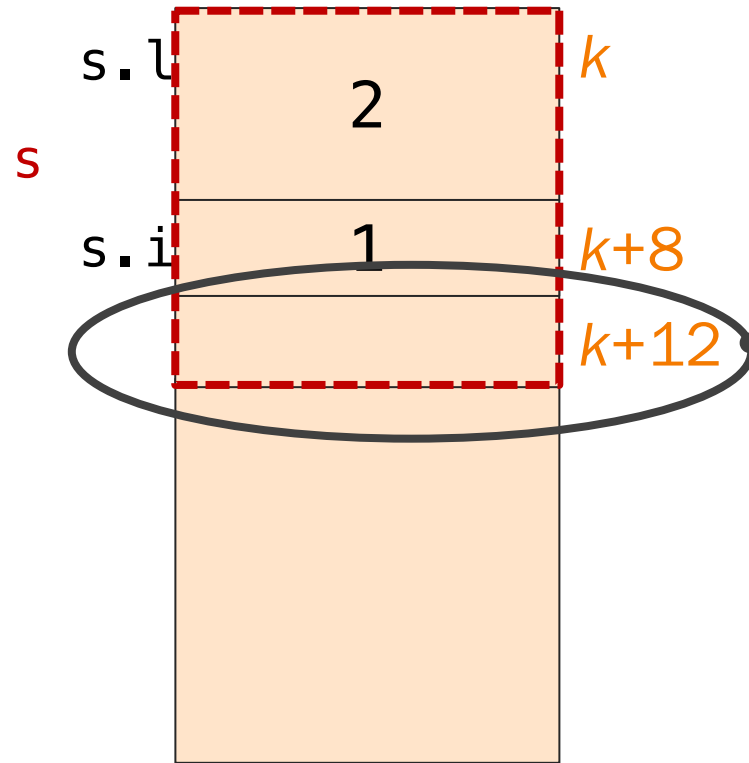
This is such a common pattern that it has its own operator:
ps->i

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

struct S s = {2, 1};
```
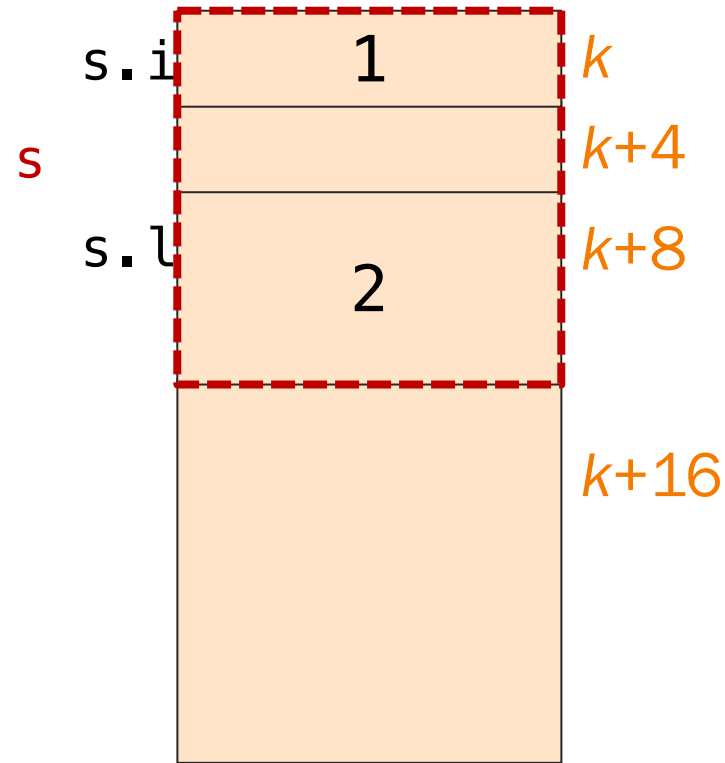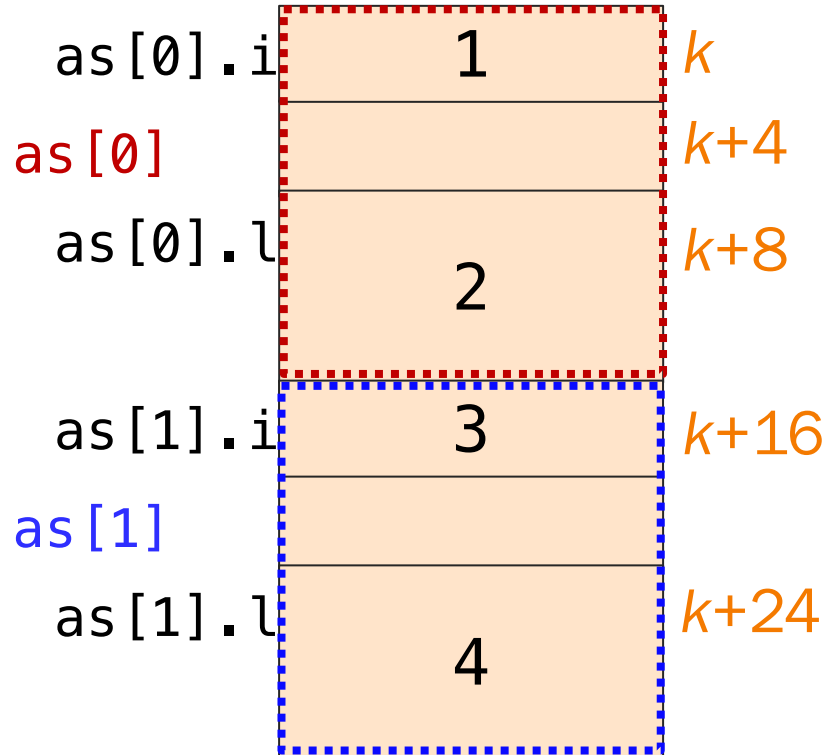
s.l
s
s.i

2 k

1 $k+8$

$k+12$

?

6

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

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

s.i `1` k

s

$k+4$

s.l `2` $k+8$

$k+16$

@agatacreate

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

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

as[0].i   1   *k*

as[0]   *k+4*

as[0].l   2   *k+8*

as[1].i   3   *k+16*

as[1]

as[1].l   4   *k+24*

8

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

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


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

as[0].i   1   *k*

as[0]   *k+4*

as[0].l   2   *k+8*

as[1].i   1   *k+16*

as[1]

as[1].l   2   *k+24*

9

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

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

10

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

void printS(struct S s) {
  int i;
  for (i = 0; i < 10; i++)
    printf("%d ", s.arr[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)!

11

@athulca

COMMAND LINE ARGUMENTS

What's my name?

- String[] args was COS 126 day 1

@wordsmithmedia

- 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

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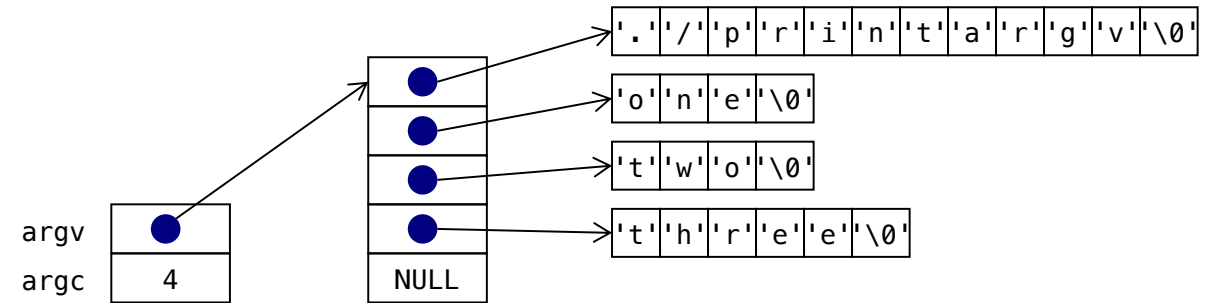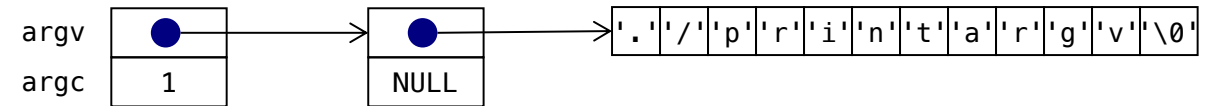
$ ./printargv one two three

'.''/''p''r''i''n''t''a''r''g''v''\0'
'o''n''e''\0'
't''w''o''\0'
't''h''r''e''e''\0'

argv
argc   4   NULL

$ ./printargv

argv
argc   1   NULL

'.''/''p''r''i''n''t''a''r''g''v''\0'

15

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

B:

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


A. 3

B. 4

C. 5

D. Syntax error at runtime

16

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



```
$ ./printargv
```
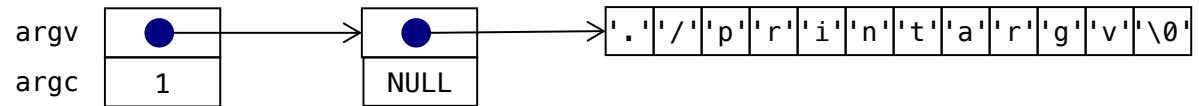


```
int main(int argc, char *argv[])
{
    char **ppc = argv;
    printf("argc:  %d\n", argc);

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

17

```
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:

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

18

```
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;`

@jorgetung

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.)
  - …

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

Memory Allocation at Runtime

Thus far we have seen 3 memory sections:

Stack
- Function parameters and local variables

Text
- Program machine language code
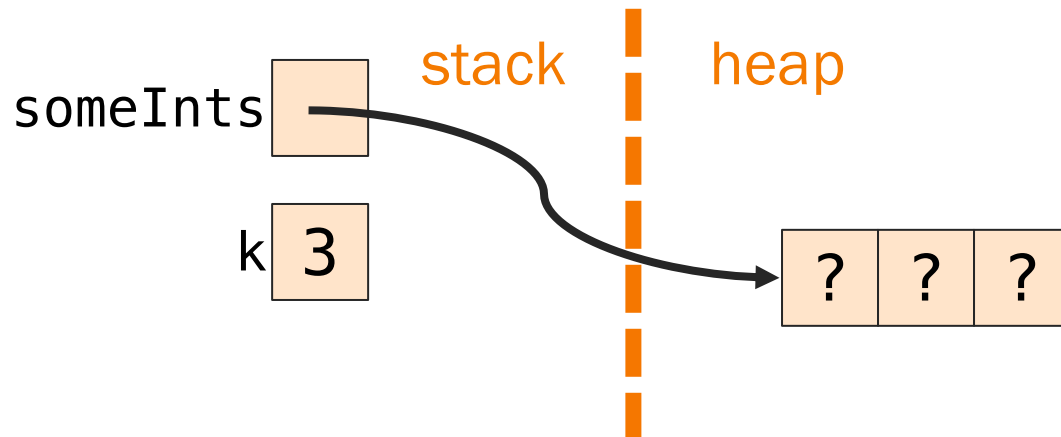
RODATA
- Read-only data, e.g. string literals

Now: "Heap"

Your New Friends: malloc

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

```
int k;
int *someInts;
printf("How many ints?");
scanf("%d", &k);
someInts =
    calloc(k, sizeof(int));
```
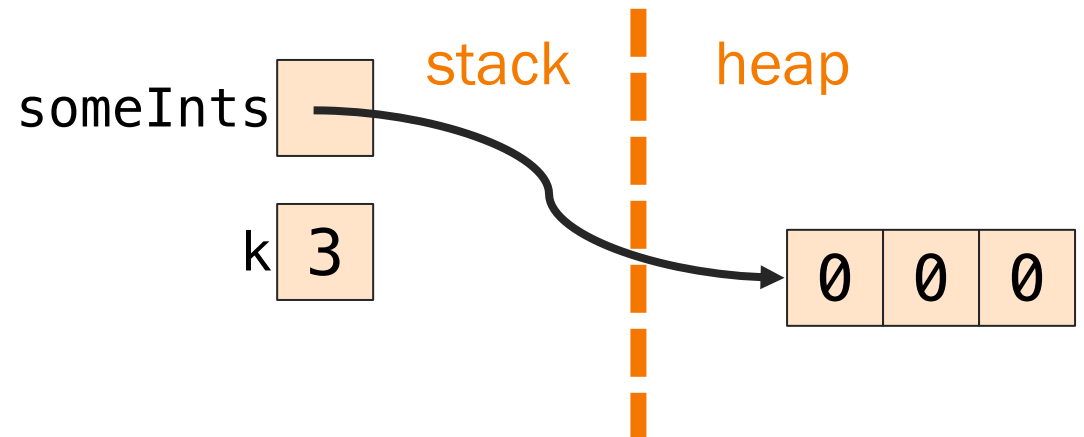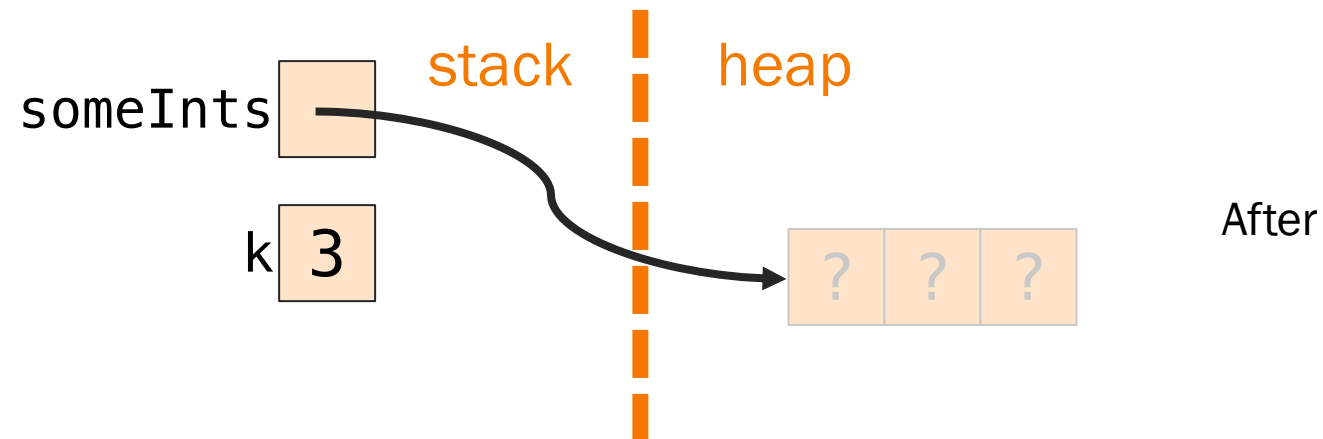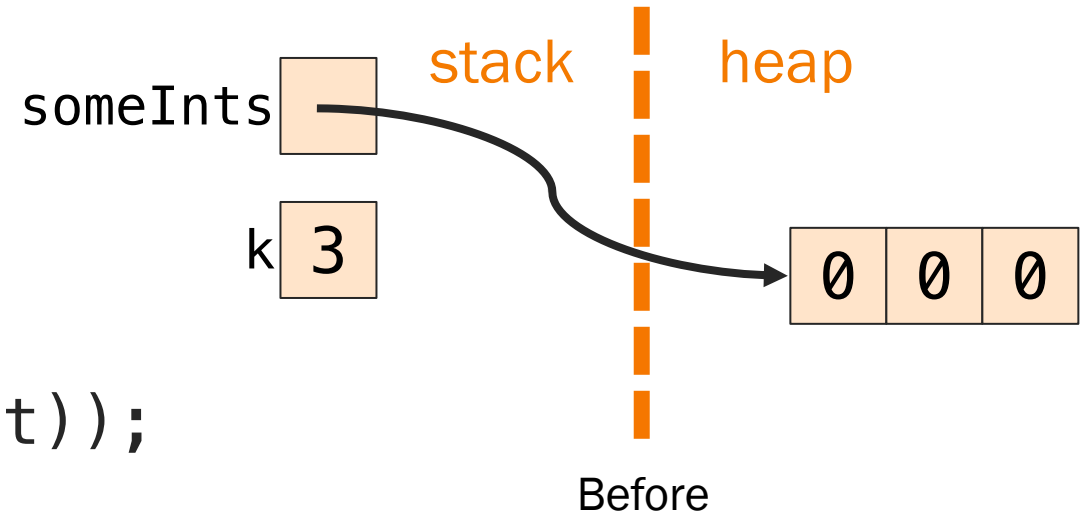
Your New Friends: calloc

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


someInts   stack   heap

k 3   0 0 0

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

free(someInts);
```

stack | heap

someInts → | | 0 | 0 | 0 |

k | 3 |

Before
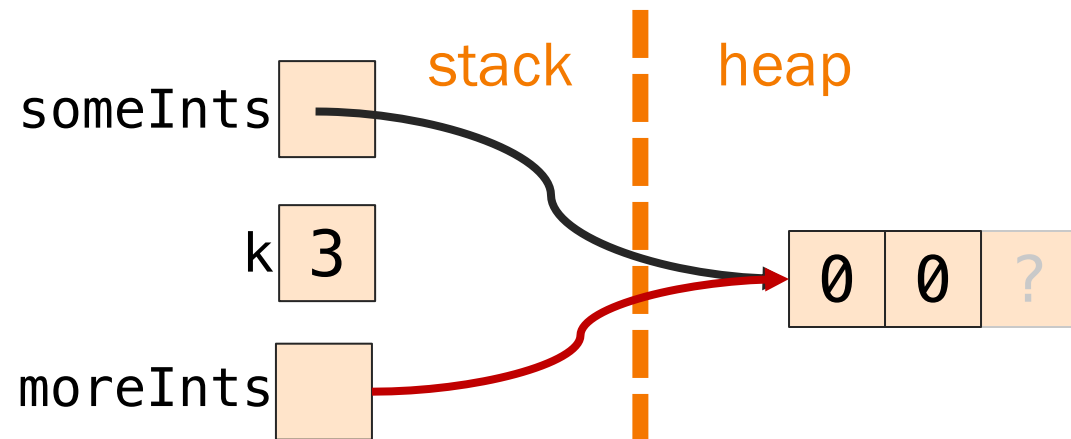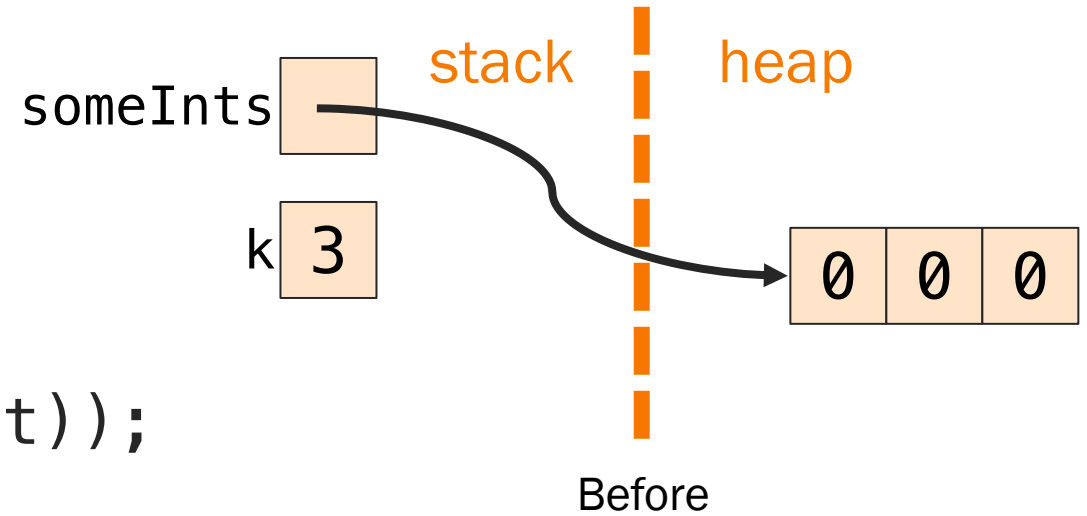
stack | heap

someInts → | ? | ? | ? |

k | 3 |

After

26

Your New Friends: realloc

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

moreInts = realloc(someInts, (k-1)*sizeof(int));
```

someInts

k 3

stack | heap

0 | 0 | 0

Before

someInts

k 3
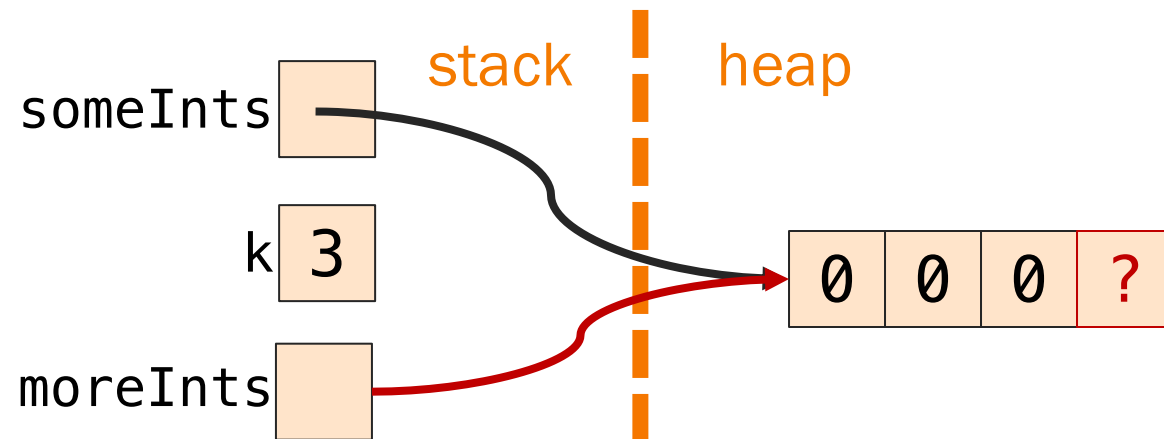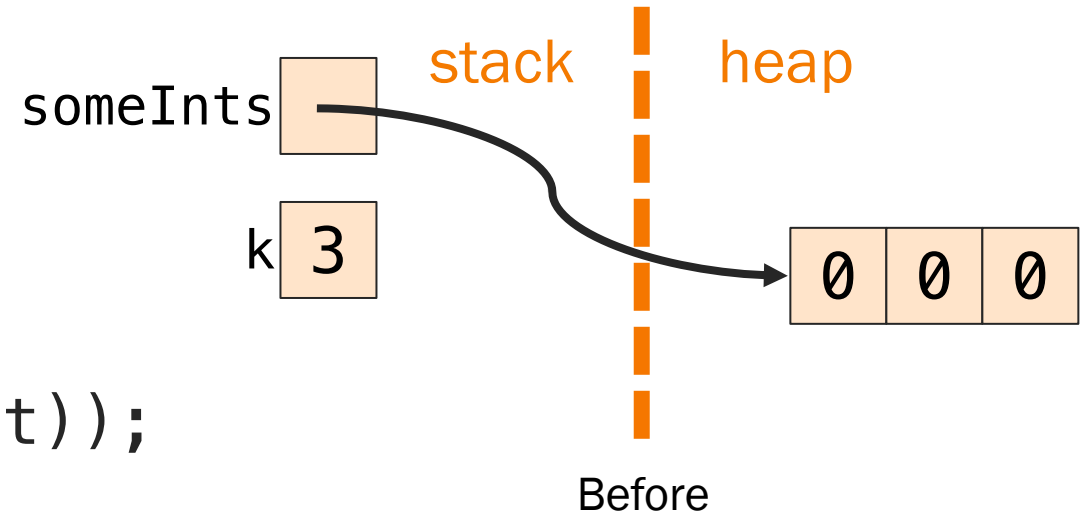
moreInts

stack | heap

0 | 0 | ?

After
(typically, but not guaranteed
by the C standard)

Your New Friends: realloc

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

moreInts = realloc(someInts, (k+1)*sizeof(int));
```


someInts — stack — heap

k 3

0 | 0 | 0

Before

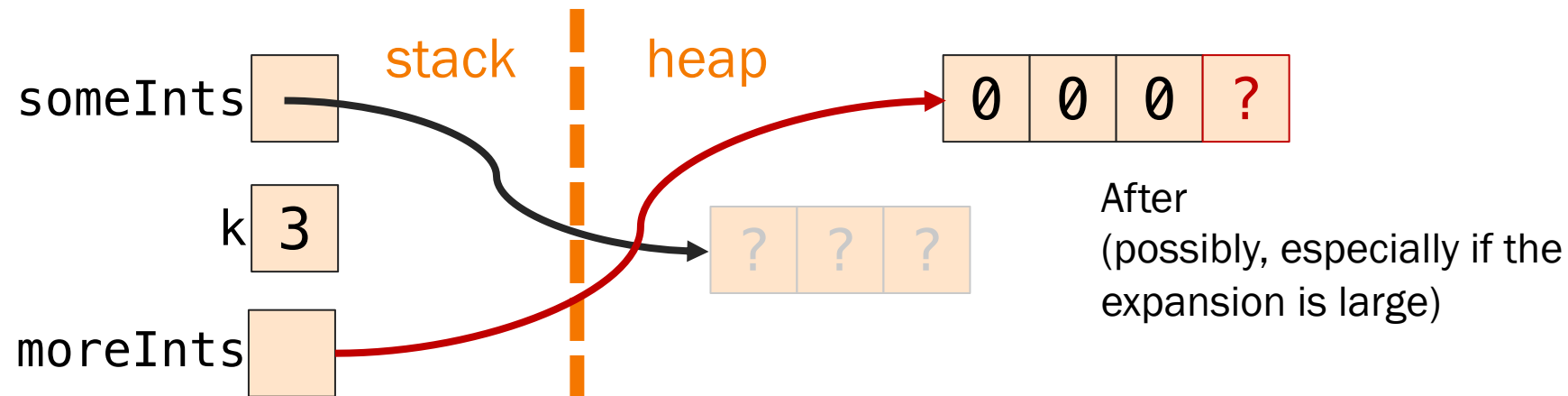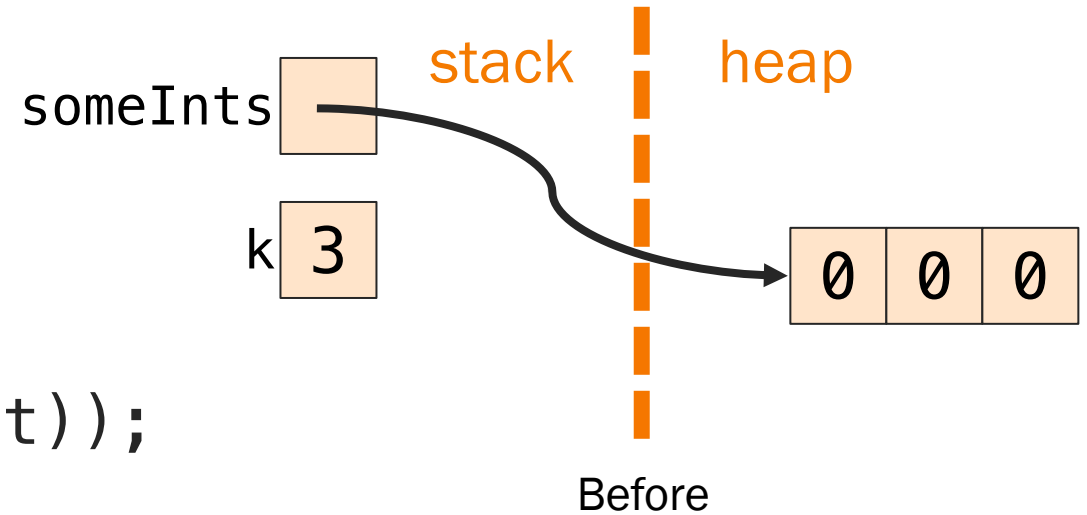someInts — stack — heap

k 3

moreInts

0 | 0 | 0 | ?

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

Your New Friends: realloc

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

moreInts = realloc(someInts, (k+1)*sizeof(int));
```
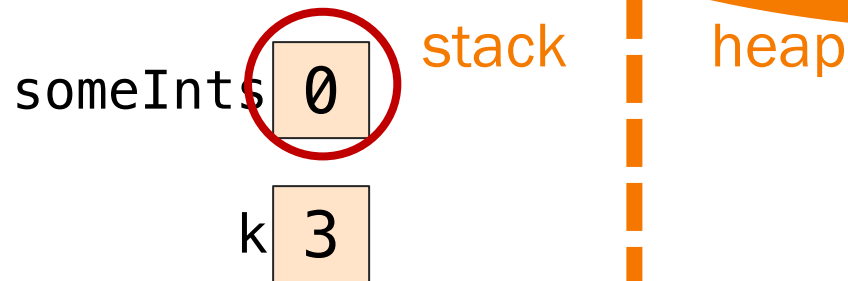
stack   heap

someInts

k 3

0 0 0

Before

stack   heap

someInts

k 3

moreInts

? ? ?

0 0 0 ?

After
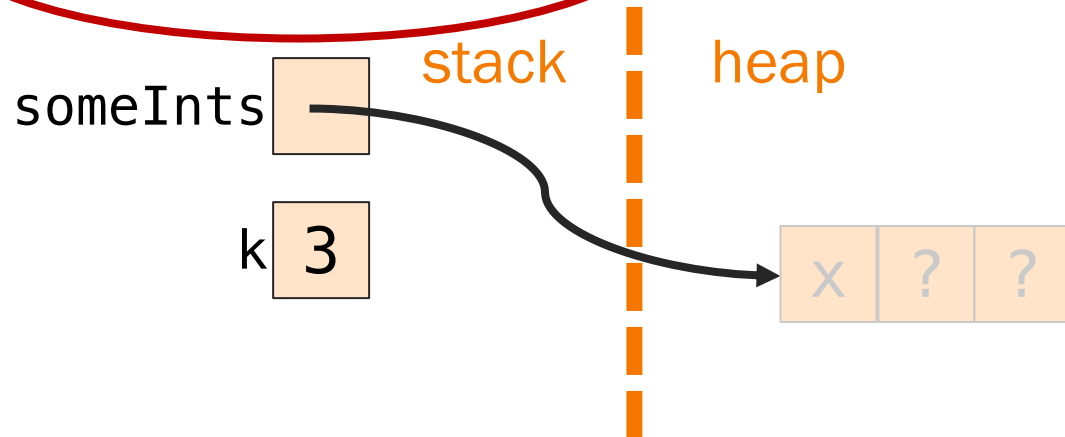(possibly, especially if the
expansion is large)

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

stack   heap

someInts 0

k 3
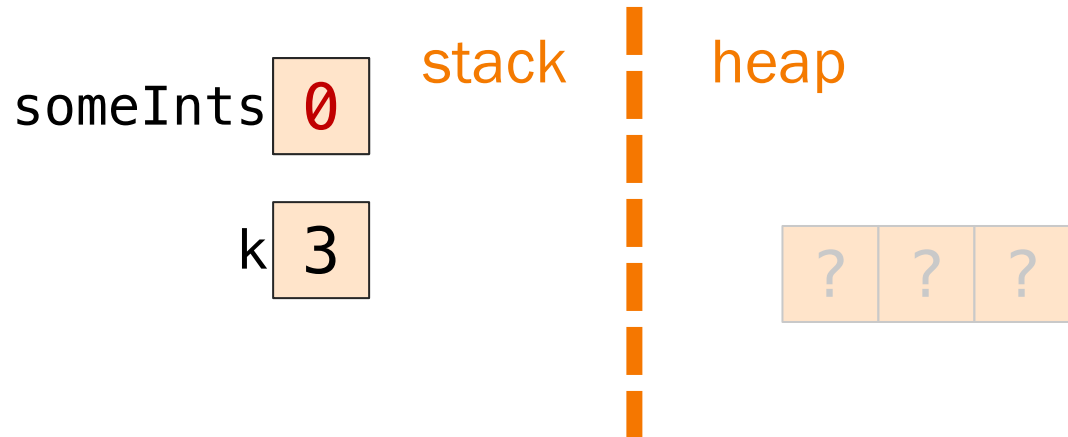

30

What could go wrong (free)?

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

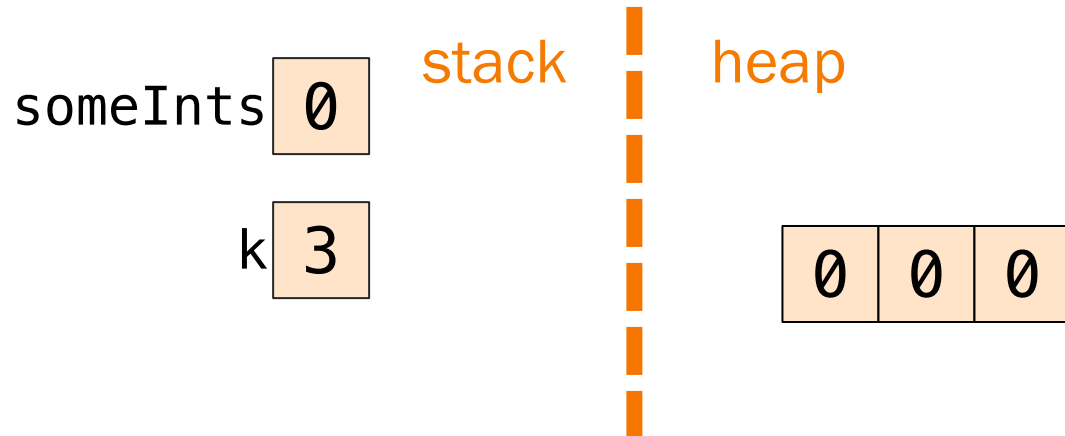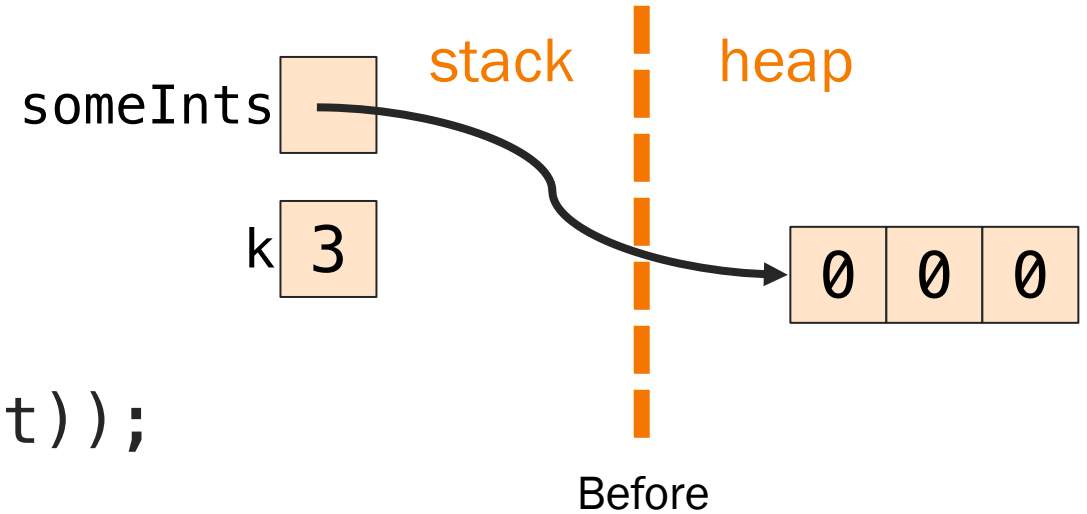stack   heap

someInts

k   3

x   ?   ?

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

stack   heap

someInts | 0 |

k | 3 |

? ? ?

32

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

stack | heap

someInts

k 3

| 0 | 0 | 0 |

Before

stack | heap

someInts 0

k 3

| 0 | 0 | 0 |

After:
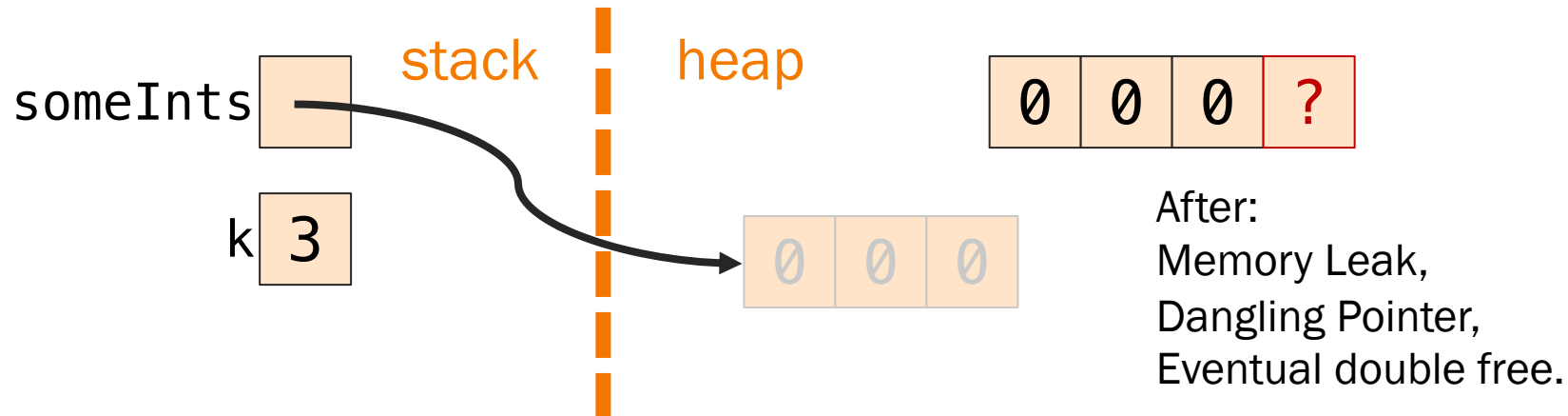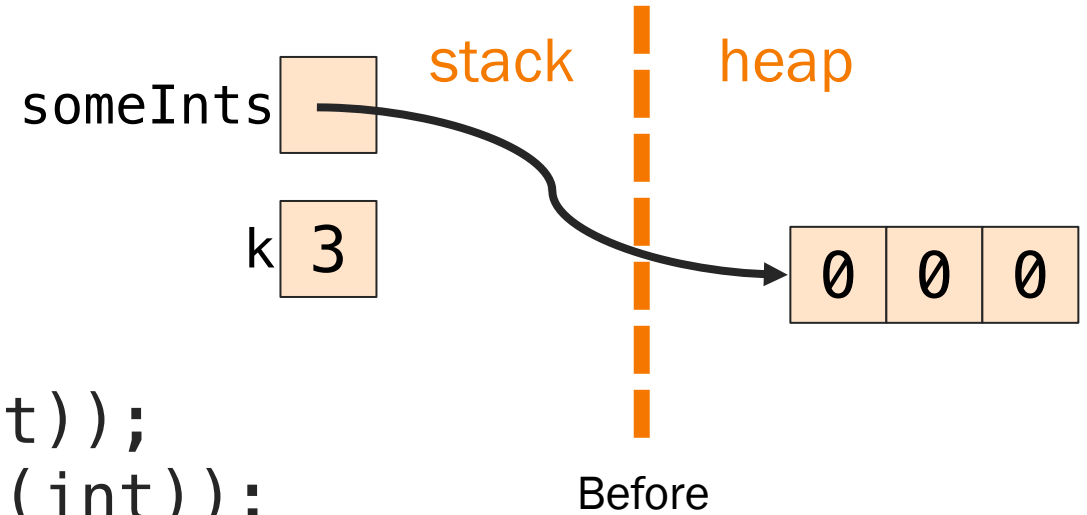If realloc returns NULL,
Memory Leak

What could go *really* wrong: realloc

```
int k;
int *someInts, *moreInts;
printf("How many ints?");
scanf("%d", &k);
someInts = calloc(k, sizeof(int));
realloc(someInts, (k+1)*sizeof(int));
if(someInts == NULL)...
```
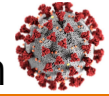
someInts [ ] stack   heap

k [3]   [0 | 0 | 0]

Before

someInts [ ] stack   heap

k [3]   [0 | 0 | 0]

[0 | 0 | 0 | ?]

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

Yes, even more common than

```
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`'.

35

```
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`