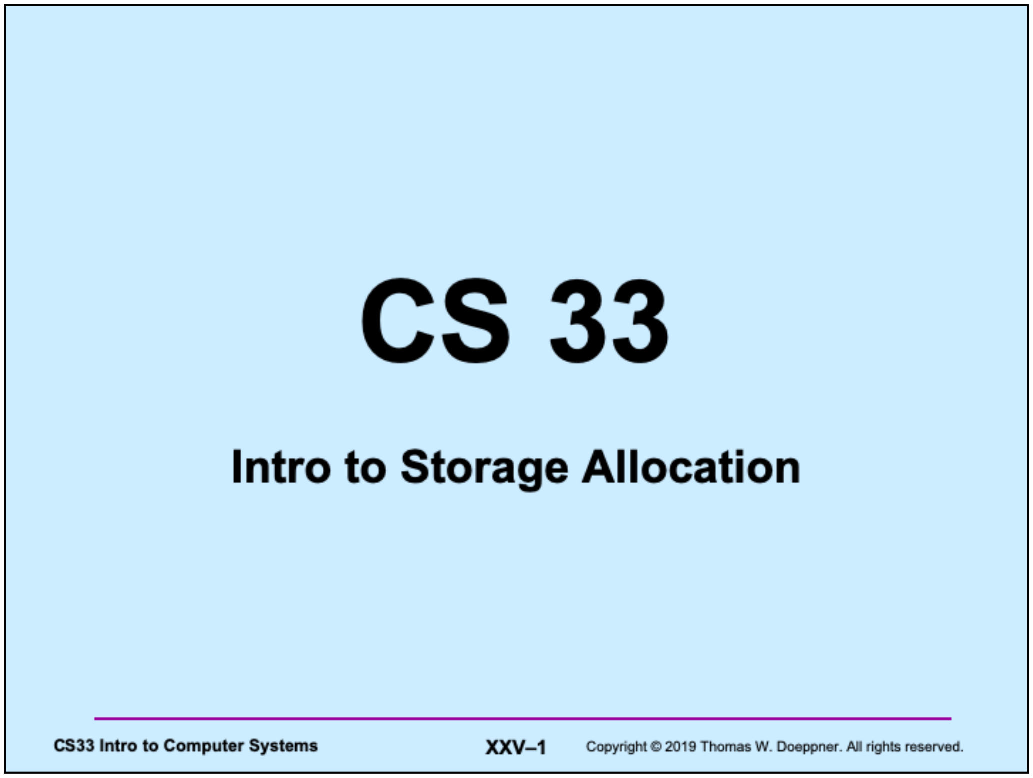
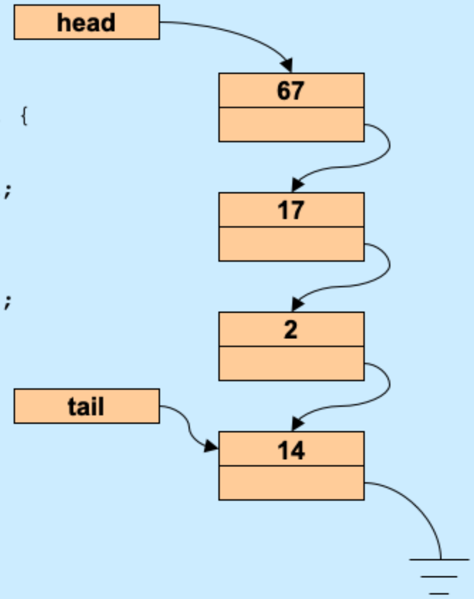


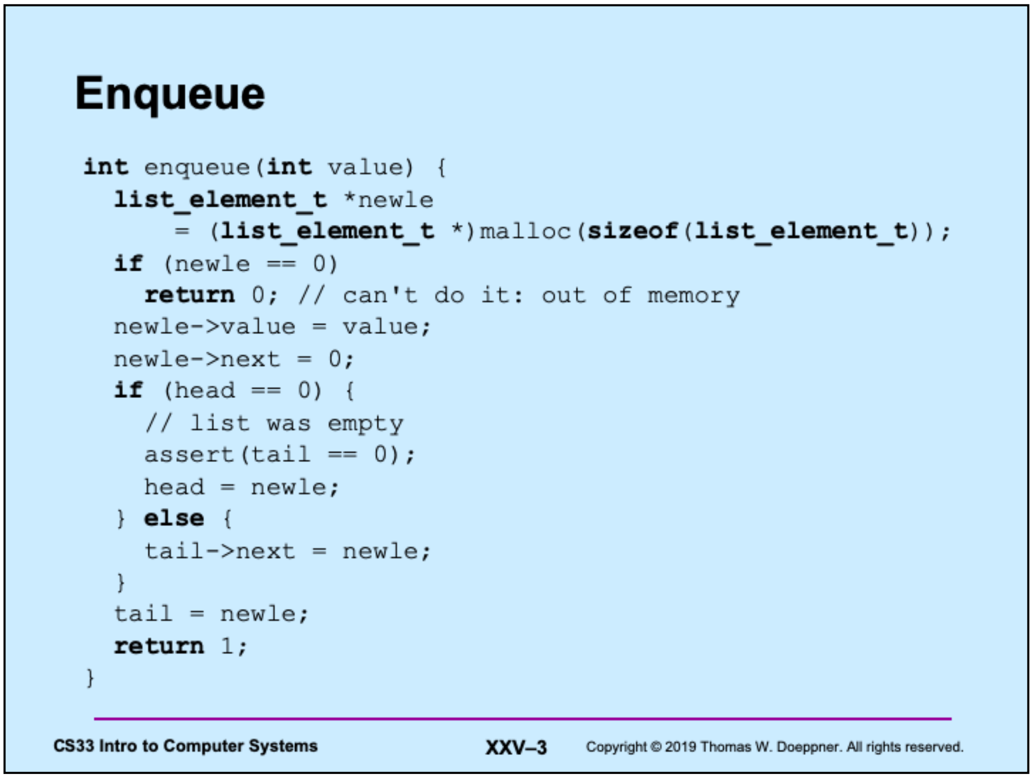
CS 33

Intro to Storage Allocation

A Queue

```
typedef struct list_element {
  int value;
  struct list_element *next;
} list_element_t;

list_element_t *head, *tail;
```

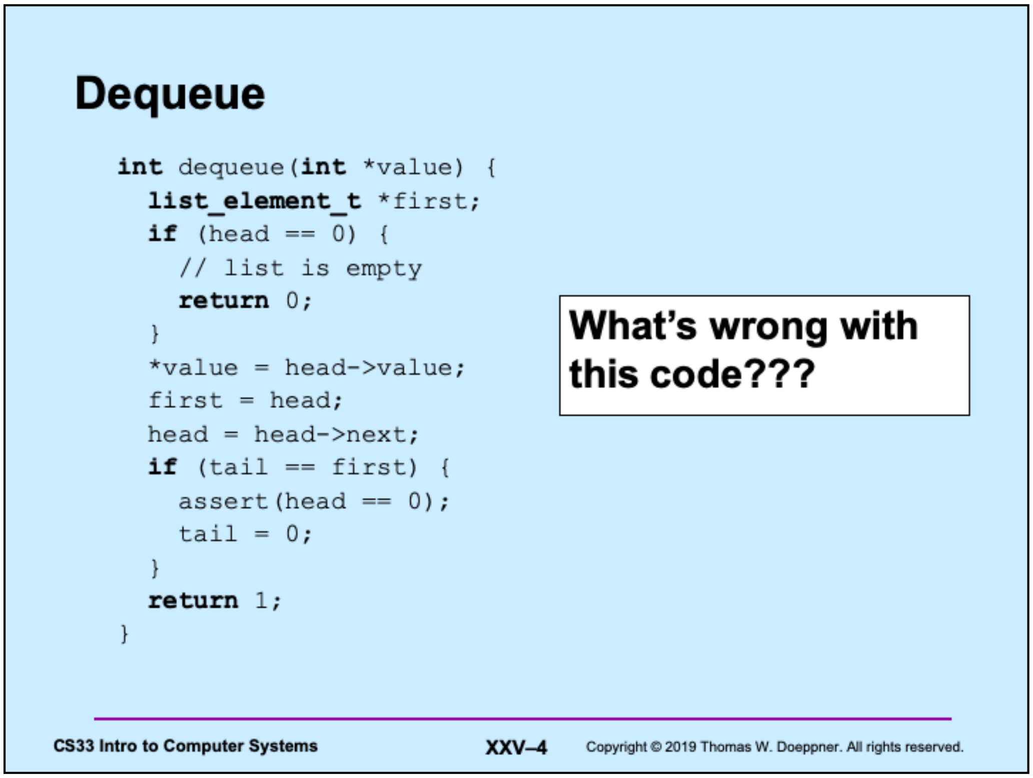




Enqueue

```
int enqueue(int value) {
  list_element_t *newle
      = (list_element_t *)malloc(sizeof(list_element_t));
  if (newle == 0)
    return 0; // can't do it: out of memory
  newle->value = value;
  newle->next = 0;
  if (head == 0) {
    // list was empty
    assert(tail == 0);
    head = newle;
  } else {
    tail->next = newle;
  }
  tail = newle;
  return 1;
}
```

Note that *malloc* allocates storage to hold a new instance of *list_element_t*.

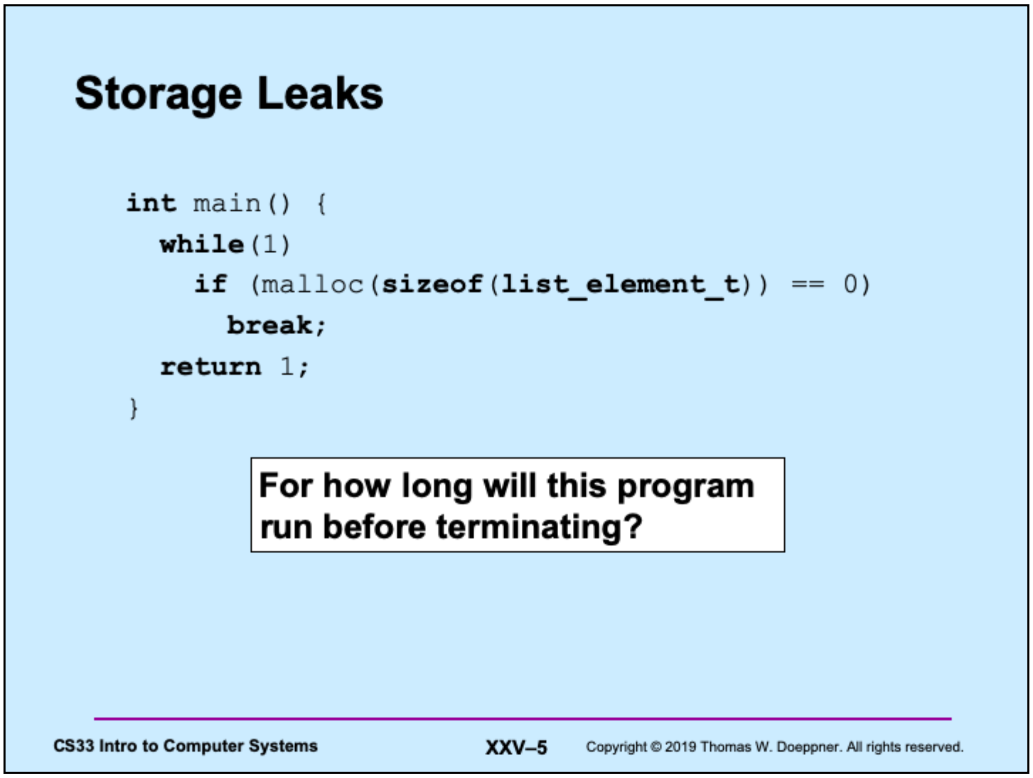


Dequeue

```
int dequeue(int *value) {
  list_element_t *first;
  if (head == 0) {
    // list is empty
    return 0;
  }
  *value = head->value;
  first = head;
  head = head->next;
  if (tail == first) {
    assert(head == 0);
    tail = 0;
  }
  return 1;
}
```

What's wrong with this code???

The problem with this code is that the list element being removed is lost – its storage is not returned to the pool of free memory.

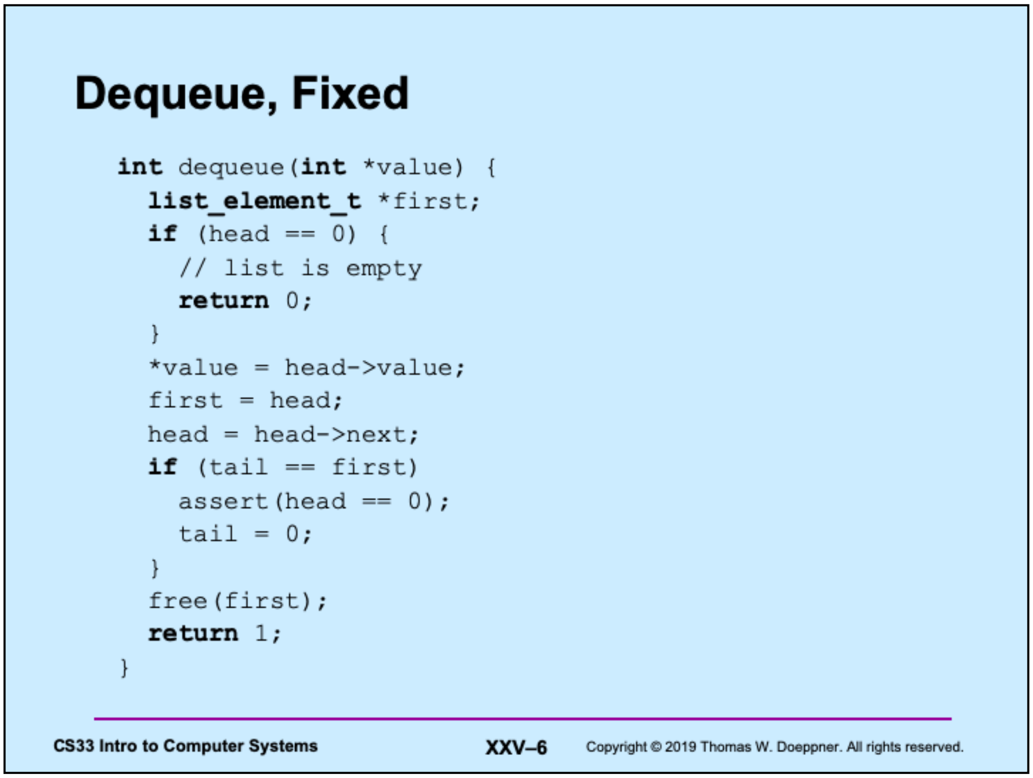


Storage Leaks

```
int main() {
  while(1)
    if (malloc(sizeof(list_element_t)) == 0)
      break;
  return 1;
}
```

For how long will this program run before terminating?

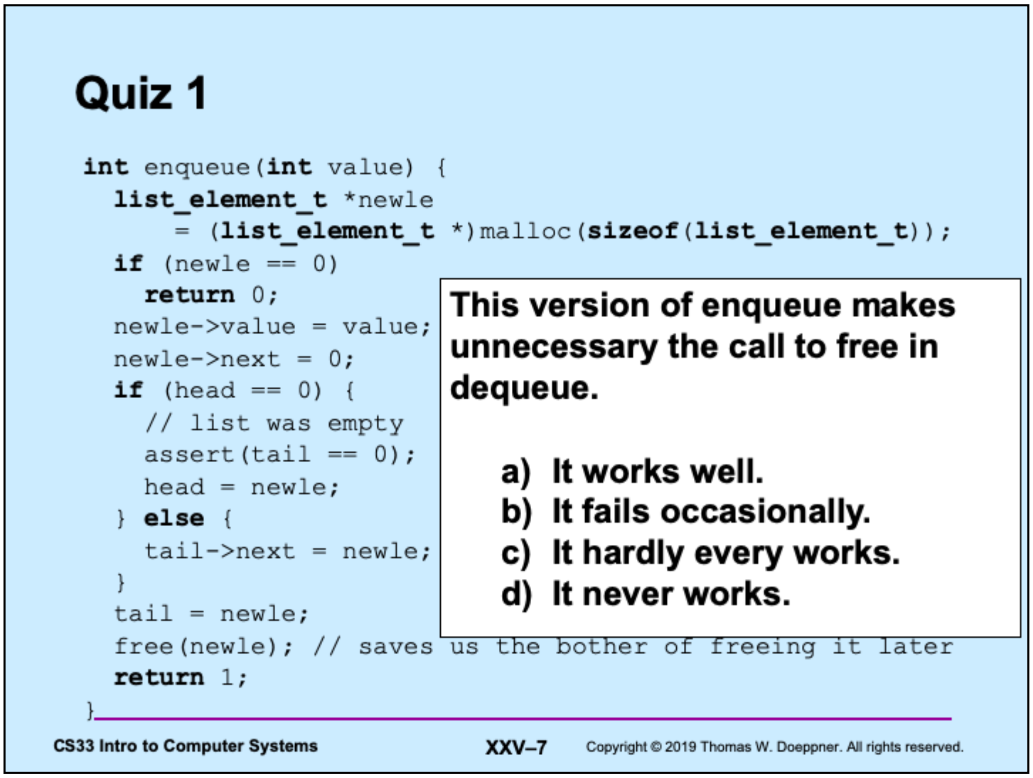
Answer: around 3 minutes on a SunLab machine.



Dequeue, Fixed

```
int dequeue(int *value) {
  list_element_t *first;
  if (head == 0) {
    // list is empty
    return 0;
  }
  *value = head->value;
  first = head;
  head = head->next;
  if (tail == first)
    assert(head == 0);
    tail = 0;
  }
  free(first);
  return 1;
}
```

Here after removing the list element from the list, we return it to the pool of free memory by calling *free*.

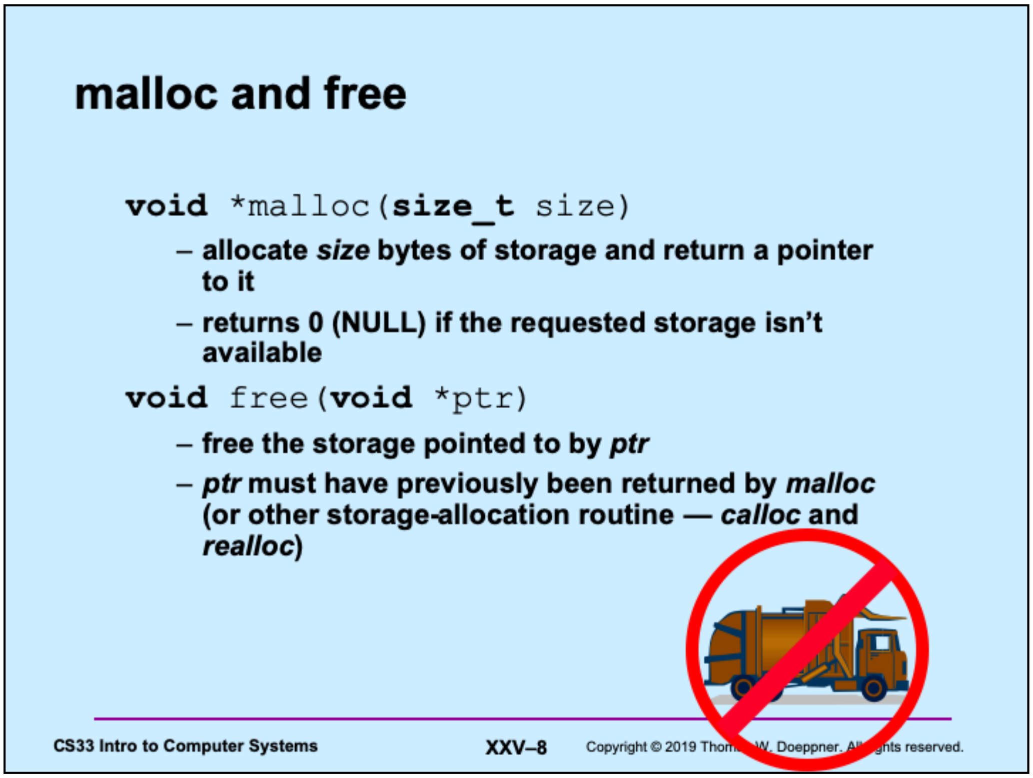


Quiz 1

```
int enqueue(int value) {
  list_element_t *newle
      = (list_element_t *)malloc(sizeof(list_element_t));
  if (newle == 0)
    return 0;
  newle->value = value;
  newle->next = 0;
  if (head == 0) {
    // list was empty
    assert(tail == 0);
    head = newle;
  } else {
    tail->next = newle;
  }
  tail = newle;
  free(newle); // saves us the bother of freeing it later
  return 1;
}
```

This version of enqueue makes unnecessary the call to free in dequeue.

a) **It works well.**
b) **It fails occasionally.**
c) **It hardly every works.**
d) **It never works.**



malloc and free

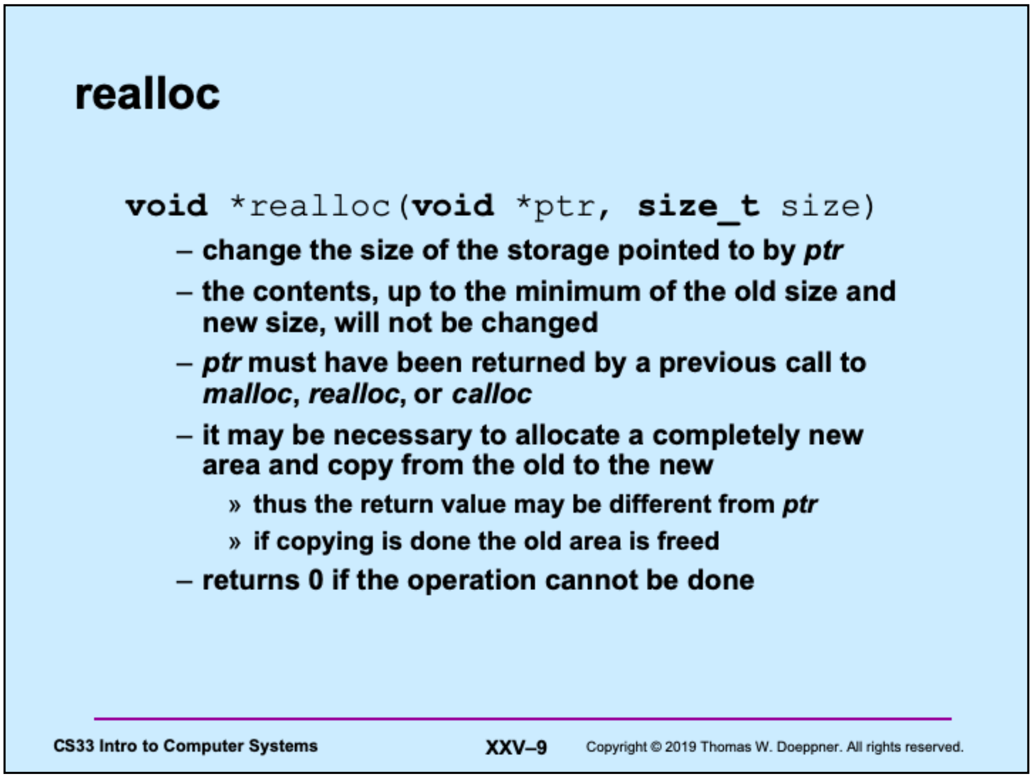
```
void *malloc(size_t size)
```

- allocate *size* bytes of storage and return a pointer to it
- returns 0 (NULL) if the requested storage isn't available

```
void free(void *ptr)
```

- free the storage pointed to by *ptr*
- *ptr* must have previously been returned by *malloc* (or other storage-allocation routine — *calloc* and *realloc*)

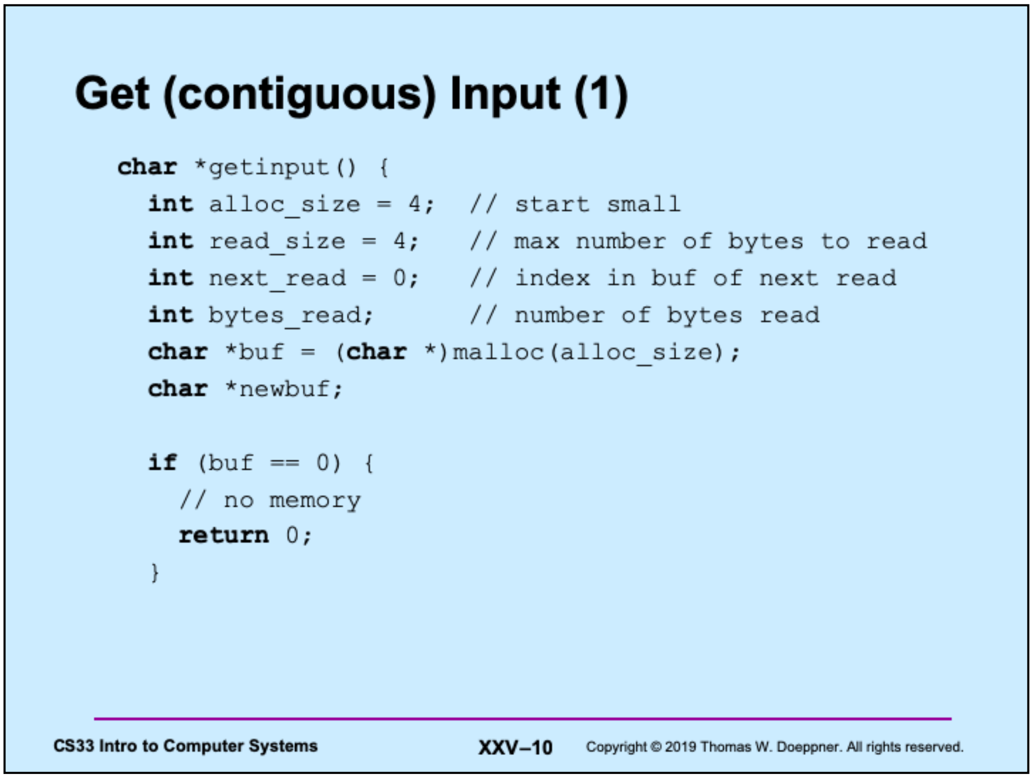
When something is malloc'd, the system must keep track of its size. Thus when it's freed, the system know how much storage is being freed.



realloc

```
void *realloc(void *ptr, size_t size)
```

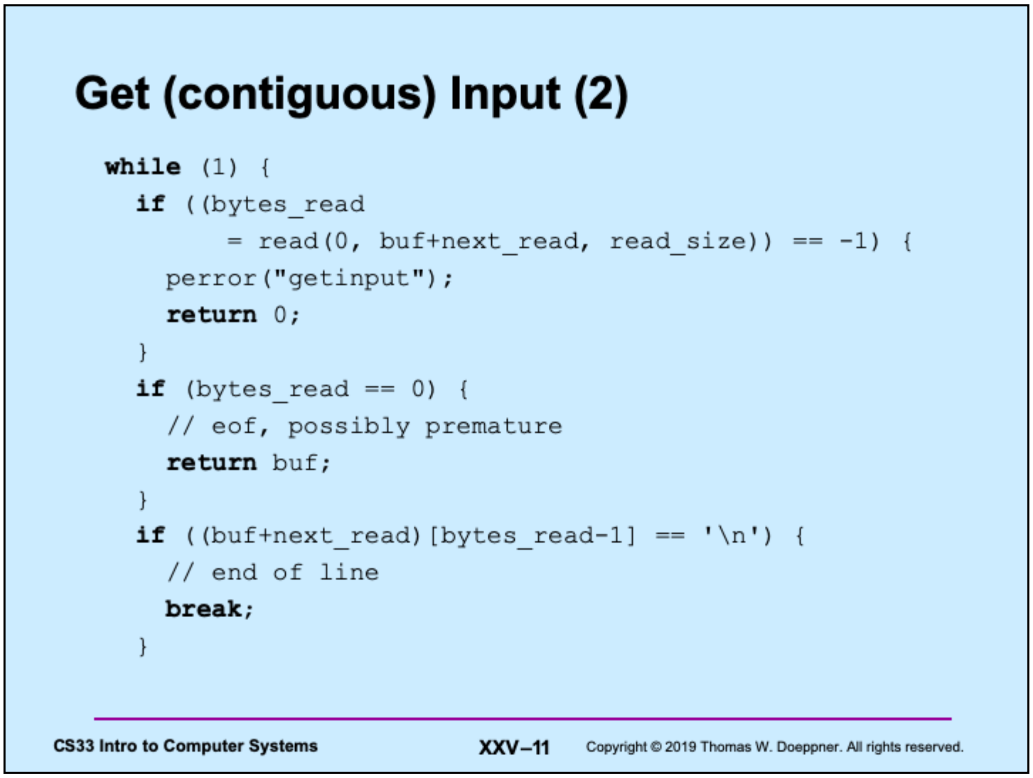
- change the size of the storage pointed to by *ptr*
- the contents, up to the minimum of the old size and new size, will not be changed
- *ptr* must have been returned by a previous call to *malloc*, *realloc*, or *calloc*
- it may be necessary to allocate a completely new area and copy from the old to the new
  - thus the return value may be different from *ptr*
  - if copying is done the old area is freed
- returns 0 if the operation cannot be done



Get (contiguous) Input (1)

```
char *getinput() {
  int alloc_size = 4;  // start small
  int read_size = 4;   // max number of bytes to read
  int next_read = 0;   // index in buf of next read
  int bytes_read;      // number of bytes read
  char *buf = (char *)malloc(alloc_size);
  char *newbuf;

  if (buf == 0) {
    // no memory
    return 0;
  }
```

In this example, we're to read a line of input, where a line is delineated by a newline character. However, we have no upper bound on its length. So we start by allocating four bytes of storage for the line. If that's not enough (the four bytes read in don't end with a '\n'), we then double our allocation and read in more up to the end of the new allocation, if that's not enough, we double the allocation again, and so forth. When we're finished, we reduce the allocation, giving back to the system that portion we didn't need.

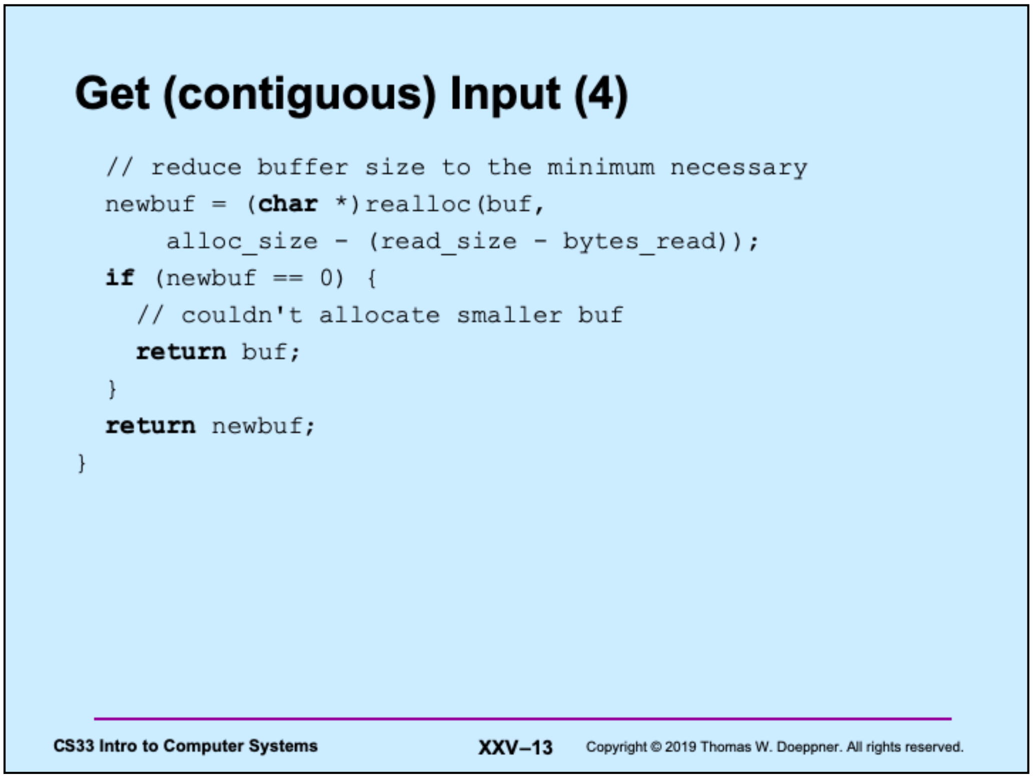


Get (contiguous) Input (2)

```
while (1) {
  if ((bytes_read
        = read(0, buf+next_read, read_size)) == -1) {
    perror("getinput");
    return 0;
  }
  if (bytes_read == 0) {
    // eof, possibly premature
    return buf;
  }
  if ((buf+next_read)[bytes_read-1] == '\n') {
    // end of line
    break;
  }
```

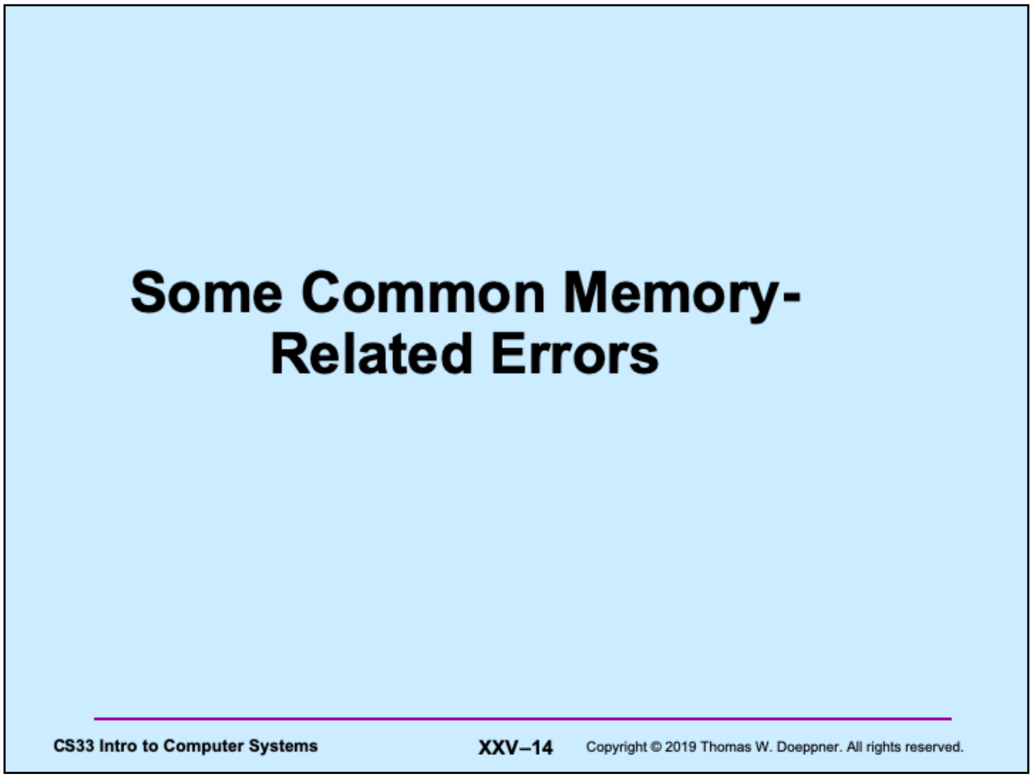
Get (contiguous) Input (3)

```
    next_read += read_size;
    read_size = alloc_size;
    alloc_size *= 2;
    newbuf = (char *)realloc(buf, alloc_size);
    if (newbuf == 0) {
      // realloc failed: not enough memory.
      // Free the storage allocated previously and report
      // failure
      free(buf);
      return 0;
    }
    buf = newbuf;
  }
```

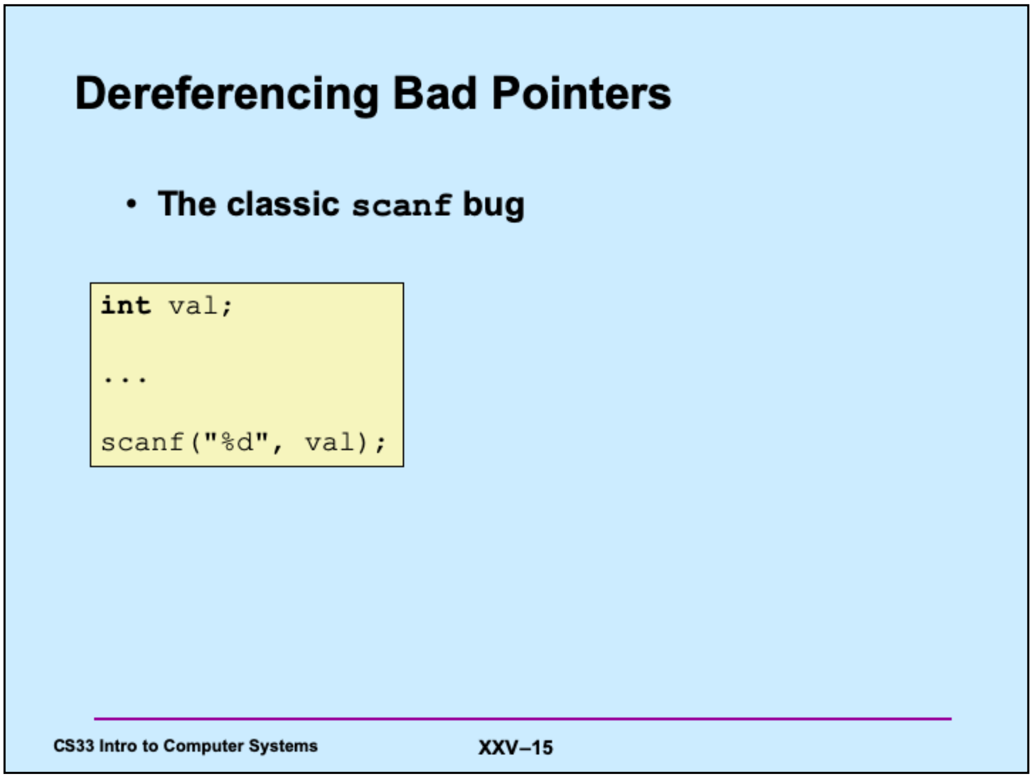


Get (contiguous) Input (4)

```
  // reduce buffer size to the minimum necessary
  newbuf = (char *)realloc(buf,
      alloc_size - (read_size - bytes_read));
  if (newbuf == 0) {
    // couldn't allocate smaller buf
    return buf;
  }
  return newbuf;
}
```



Some Common Memory-Related Errors

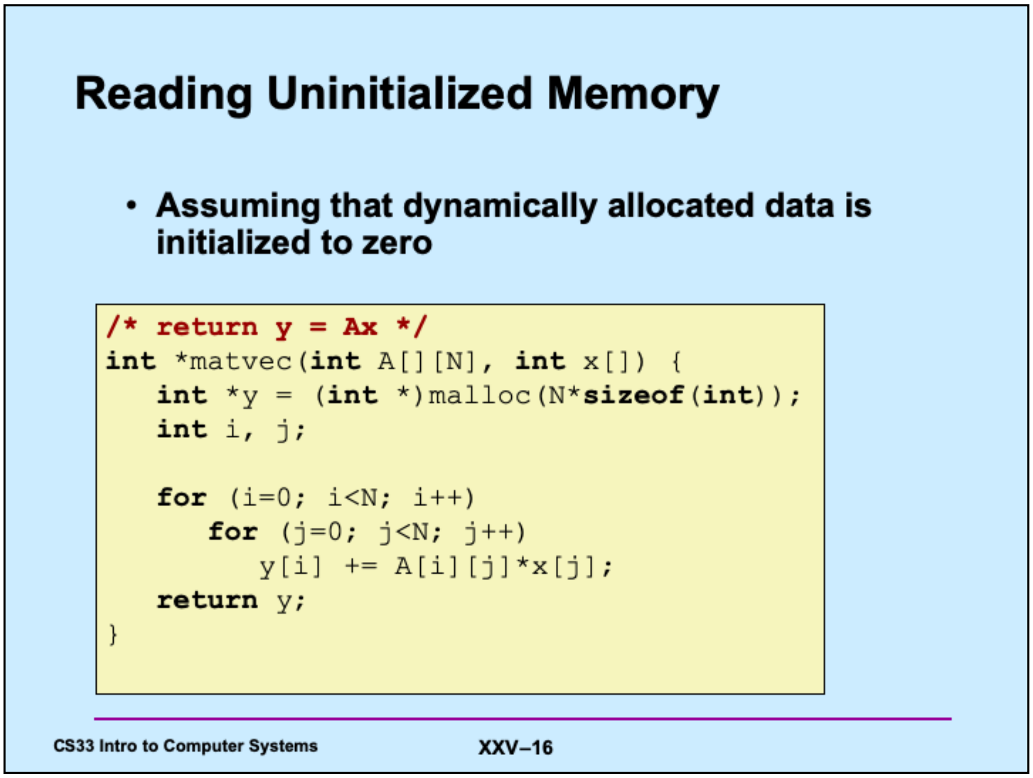


Dereferencing Bad Pointers

- The classic `scanf` bug

```
int val;

...

scanf("%d", val);
```

Supplied by CMU.

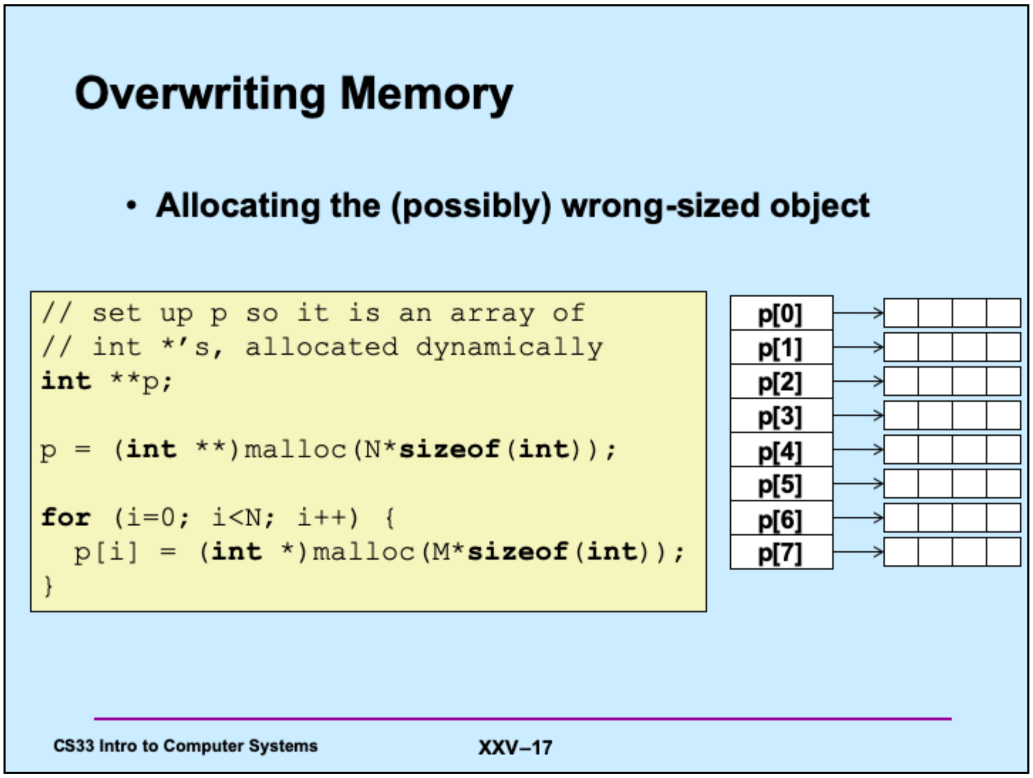


Reading Uninitialized Memory

- Assuming that dynamically allocated data is initialized to zero

```
/* return y = Ax */
int *matvec(int A[][N], int x[]) {
   int *y = (int *)malloc(N*sizeof(int));
   int i, j;

   for (i=0; i<N; i++)
      for (j=0; j<N; j++)
         y[i] += A[i][j]*x[j];
   return y;
}
```

Supplied by CMU.



Overwriting Memory

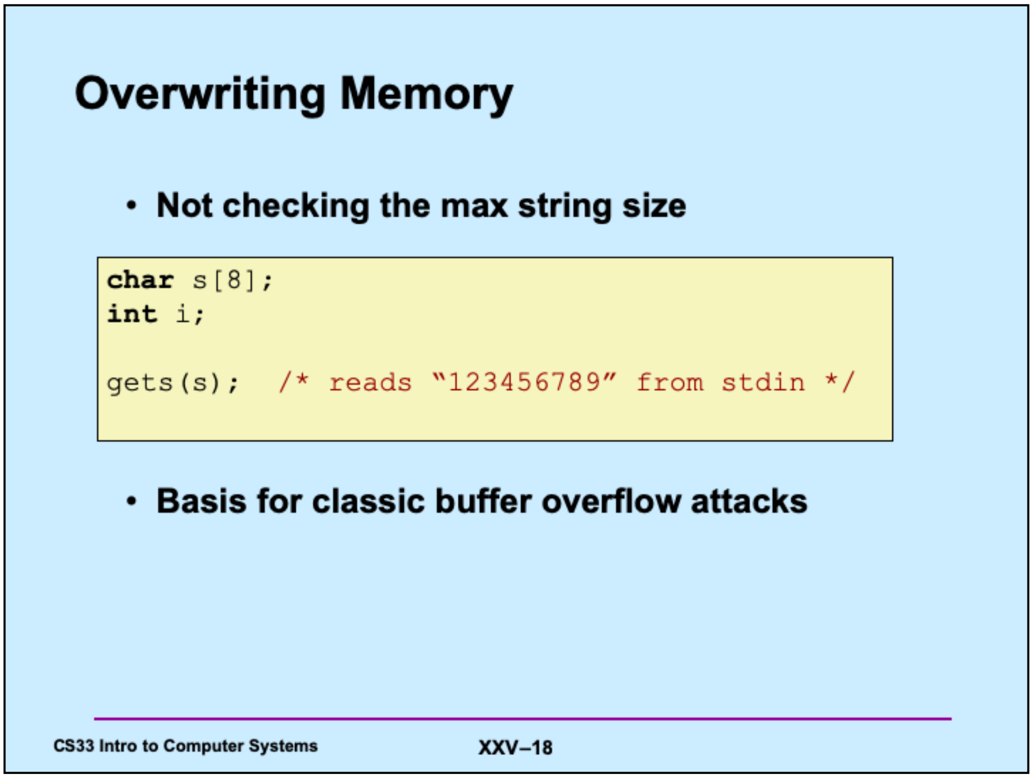
- **Allocating the (possibly) wrong-sized object**

```
// set up p so it is an array of
// int *'s, allocated dynamically
int **p;

p = (int **)malloc(N*sizeof(int));

for (i=0; i<N; i++) {
  p[i] = (int *)malloc(M*sizeof(int));
}
```

p[0]
p[1]
p[2]
p[3]
p[4]
p[5]
p[6]
p[7]

Supplied by CMU.

The problem here is that the storage allocated for p is of size N*sizeof(int), when it should be N*sizeof(int *) — on a 64-bit machine, p won’t have been assigned enough storage.



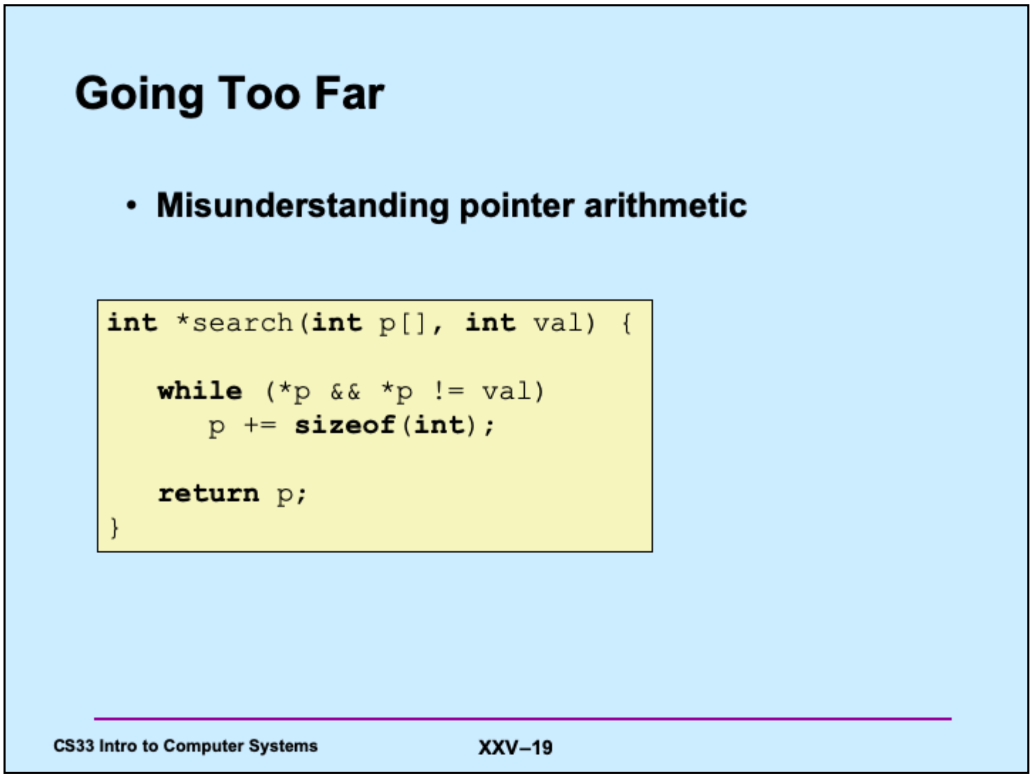
Overwriting Memory

- Not checking the max string size

```
char s[8];
int i;

gets(s);  /* reads "123456789" from stdin */
```

- Basis for classic buffer overflow attacks

Supplied by CMU.

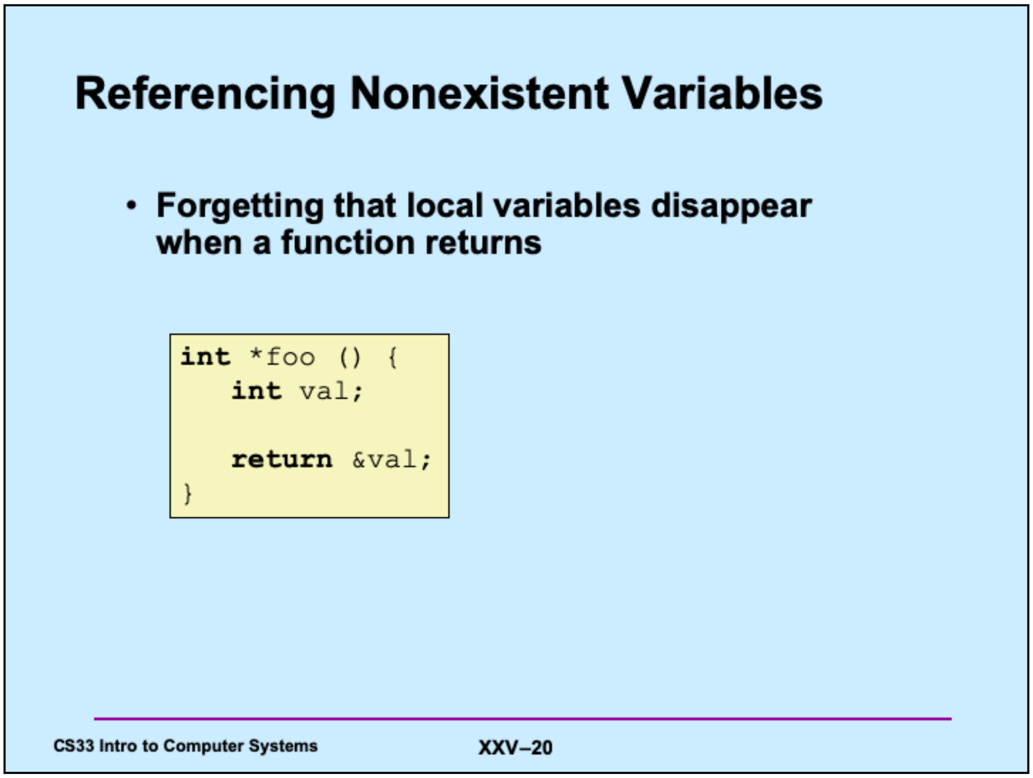


Going Too Far

- **Misunderstanding pointer arithmetic**

```
int *search(int p[], int val) {

   while (*p && *p != val)
      p += sizeof(int);

   return p;
}
```

Supplied by CMU.

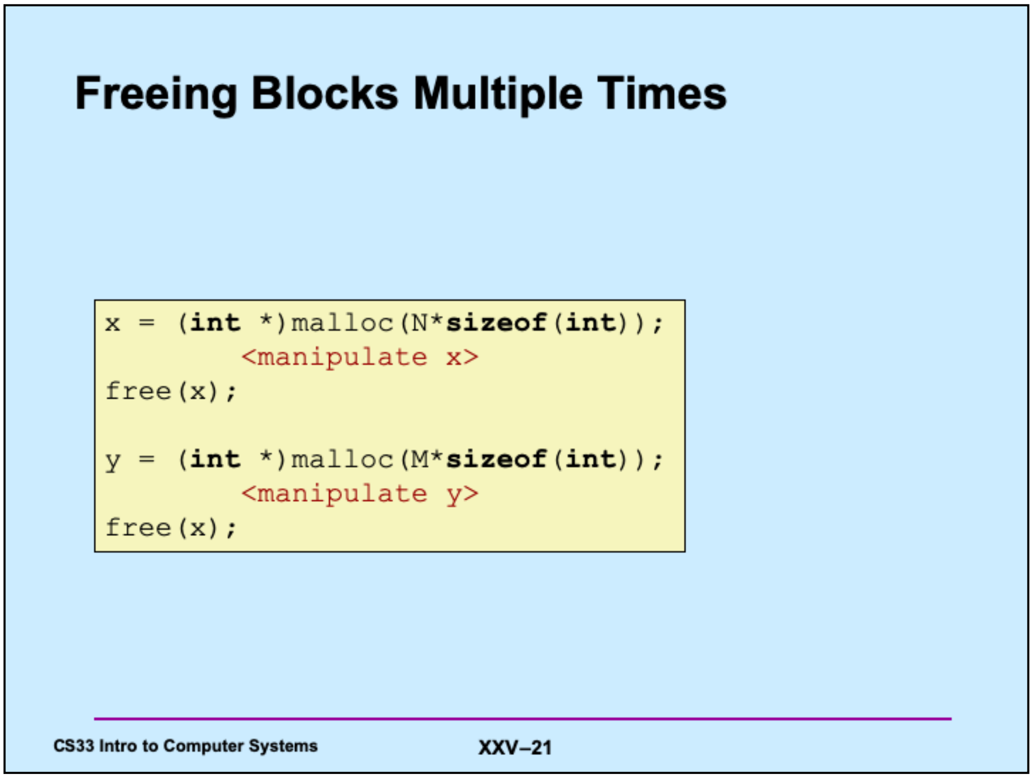


Referencing Nonexistent Variables

- Forgetting that local variables disappear when a function returns

```
int *foo () {
   int val;

   return &val;
}
```

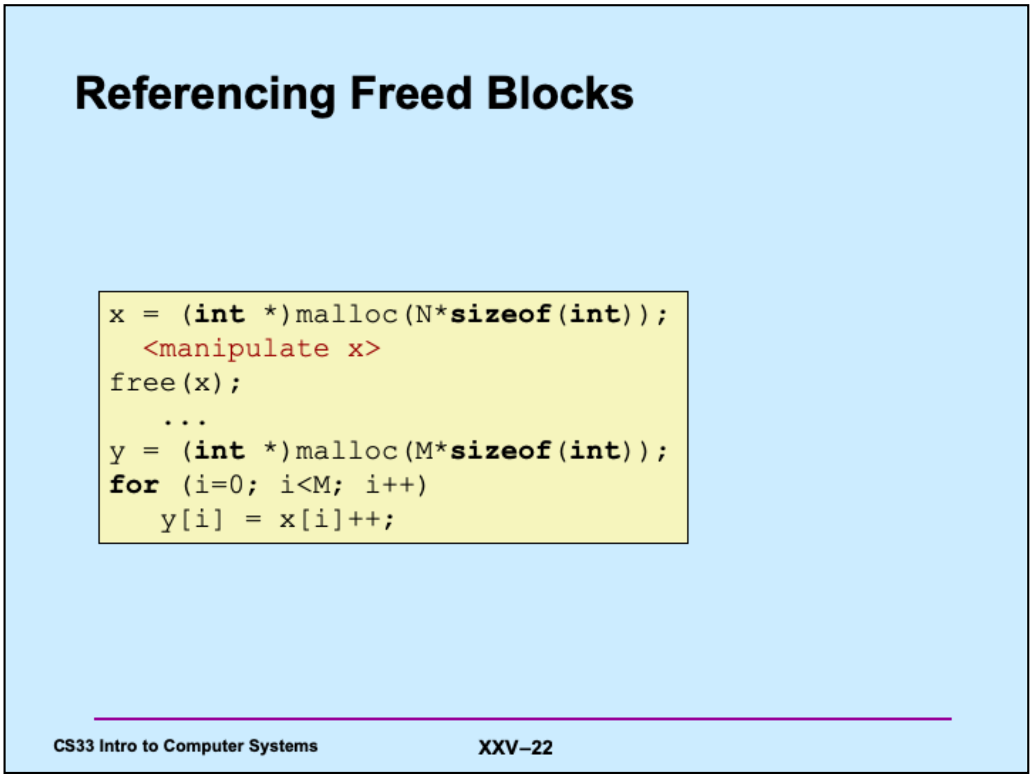
Supplied by CMU.



Freeing Blocks Multiple Times

```
x = (int *)malloc(N*sizeof(int));
        <manipulate x>
free(x);

y = (int *)malloc(M*sizeof(int));
        <manipulate y>
free(x);
```

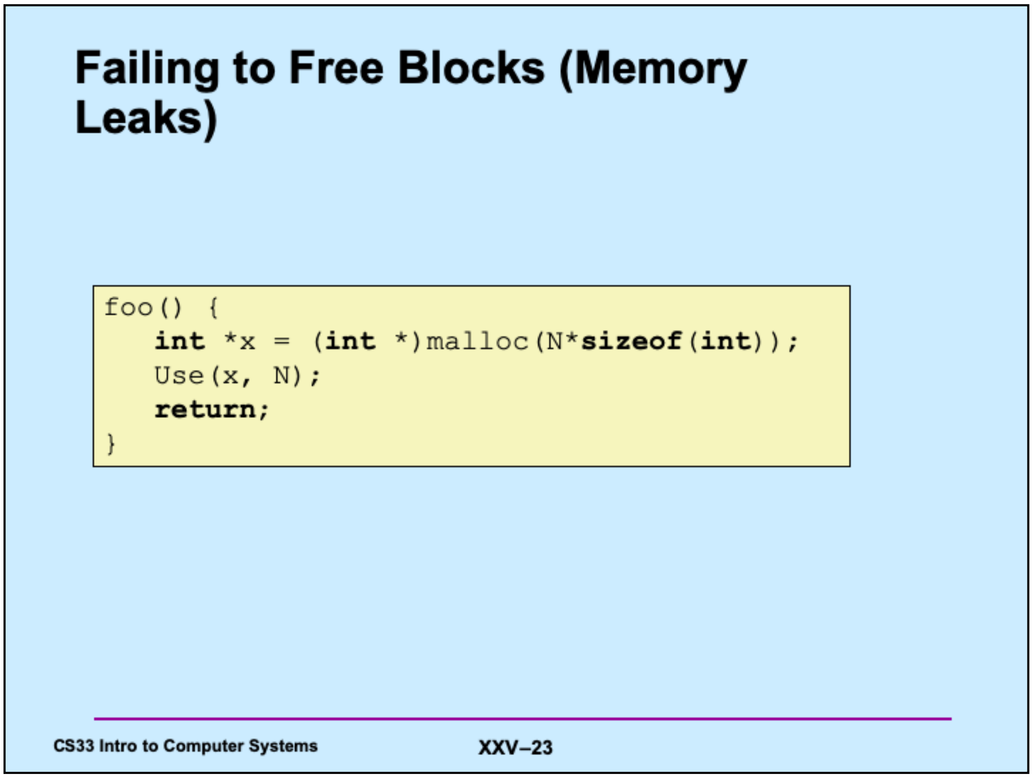
Supplied by CMU.



Referencing Freed Blocks

```
x = (int *)malloc(N*sizeof(int));
  <manipulate x>
free(x);
   ...
y = (int *)malloc(M*sizeof(int));
for (i=0; i<M; i++)
   y[i] = x[i]++;
```

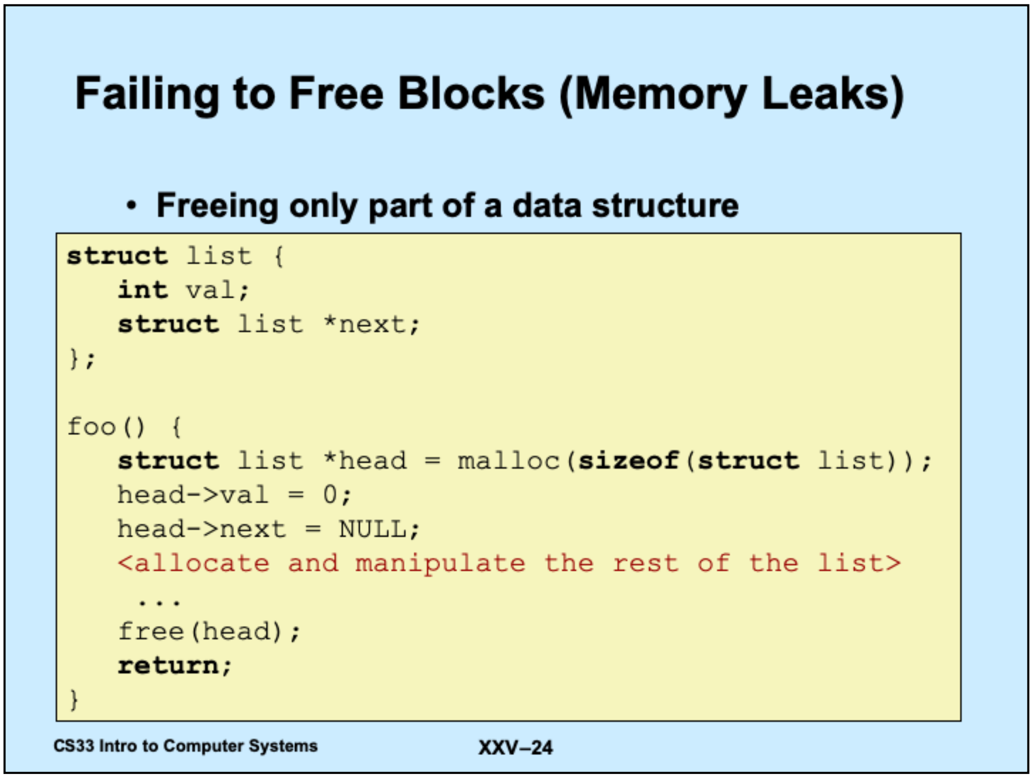
Supplied by CMU.



Failing to Free Blocks (Memory Leaks)

```
foo() {
   int *x = (int *)malloc(N*sizeof(int));
   Use(x, N);
   return;
}
```

Supplied by CMU.

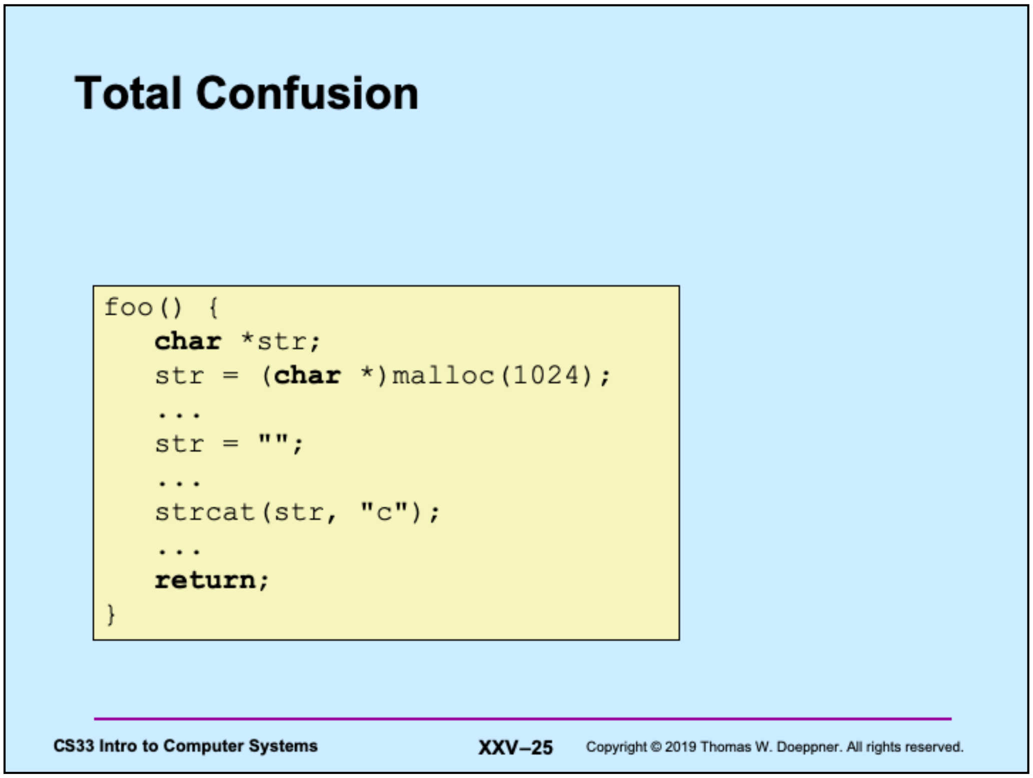


Failing to Free Blocks (Memory Leaks)

- **Freeing only part of a data structure**

```
struct list {
   int val;
   struct list *next;
};

foo() {
   struct list *head = malloc(sizeof(struct list));
   head->val = 0;
   head->next = NULL;
   <allocate and manipulate the rest of the list>
    ...
   free(head);
   return;
}
```

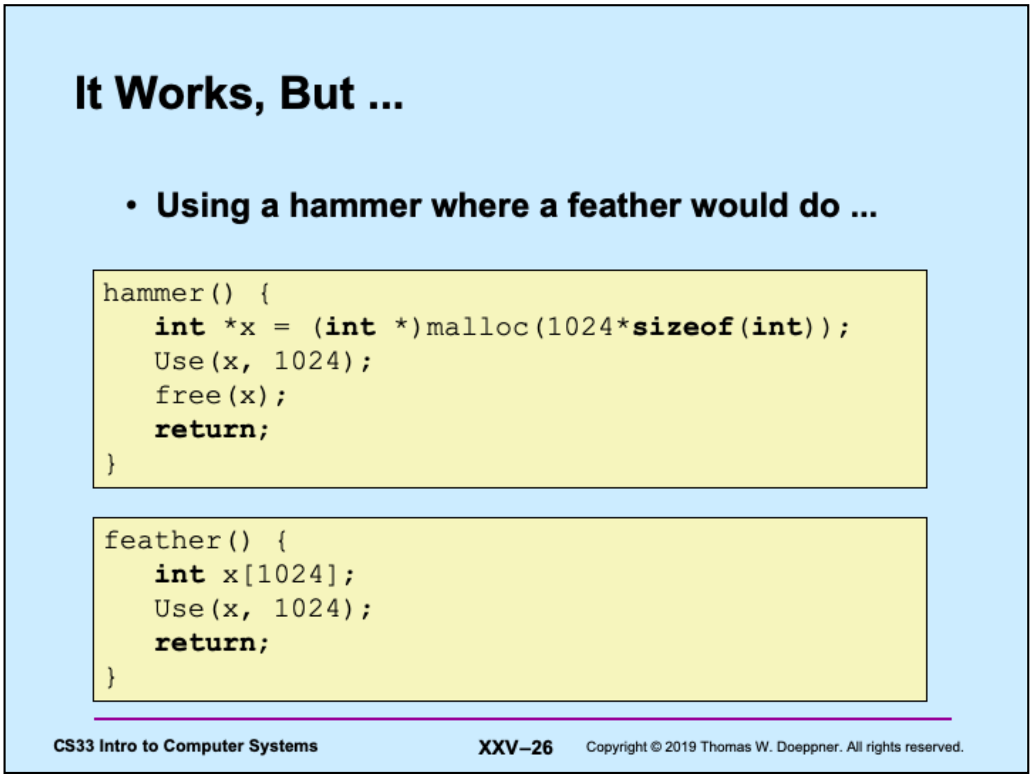
Supplied by CMU.



Total Confusion

```
foo() {
   char *str;
   str = (char *)malloc(1024);
   ...
   str = "";
   ...
   strcat(str, "c");
   ...
   return;
}
```

There are two problems here: space is allocated for str to point to, but the space is not freed when str no longer points to it. str now points to the string “”, a string consisting of just the null byte that’s in read-only storage. The strcat attempts to copy a string into the storage, but not only is the string to be copied too long, but there will be a seg fault when the attempt is made to copy it into the read-only storage.

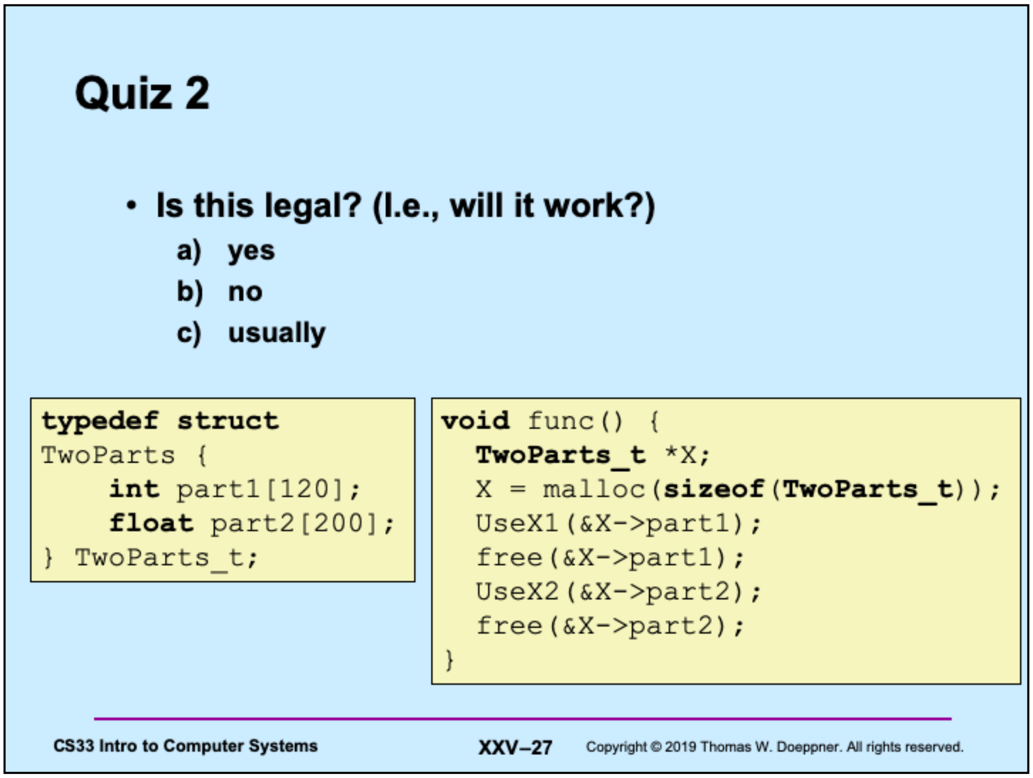


It Works, But ...

- Using a hammer where a feather would do ...

```
hammer() {
   int *x = (int *)malloc(1024*sizeof(int));
   Use(x, 1024);
   free(x);
   return;
}
```

```
feather() {
   int x[1024];
   Use(x, 1024);
   return;
}
```



Quiz 2

- Is this legal? (I.e., will it work?)
  a) yes
  b) no
  c) usually

```
typedef struct
TwoParts {
    int part1[120];
    float part2[200];
} TwoParts_t;
```

```
void func() {
  TwoParts_t *X;
  X = malloc(sizeof(TwoParts_t));
  UseX1(&X->part1);
  free(&X->part1);
  UseX2(&X->part2);
  free(&X->part2);
}
```