# Week 9 Part 2

Kyle Dewey

# Overview

- Announcement
- More with structs and memory
- Assertions
- Exam #2
- Course review

# Next “Lab”

- No pre-lab
- The lab will be entirely review
- It **will be graded**

# More with Structs

# Nesting Structs

- We can also nest structs, like so:

```
struct Point {
  int x;
  int y;
};
struct Circle {
  struct Point center;
  int radius;
};
```

```
struct Address {
  int streetNumber;
  char* street;
  int zip;
  int state;
};
struct Date {
  int month;
  int day;
  int year;
};
struct Person {
  struct Address address;
  char* name;
  struct Date birthday;
};
```

# Pointers to Structs

- Remember, we can also have pointers to other structs in a struct, or even pointers to a struct of the same type

```
struct IntegerList {
  int integer;
  struct IntegerList* nextInteger;
};
```

# Returning Structs

- Structs can be returned just like any other data
- The same rules about structs versus pointers to structs apply

# Returning Structs

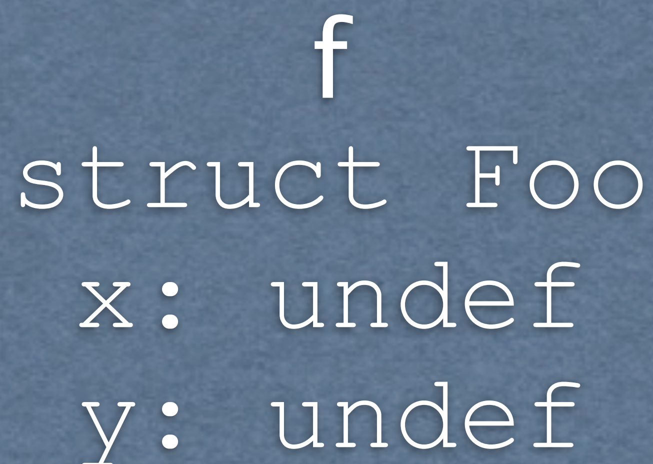
- This code is perfectly valid:

```
struct Foo { int x; int y; };

struct Foo doSomething( int a ) {
  struct Foo retval = { a, a + 1 };
  return retval;
}
int main() {
  struct Foo f;
  f = doSomething( 1 );
  return 0;
}
```

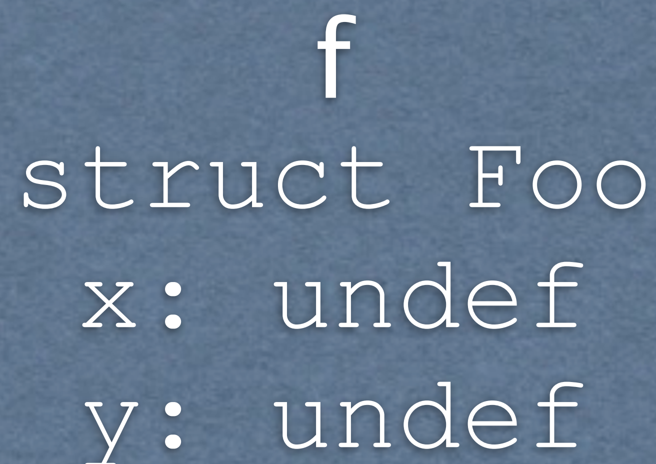
# Internal Representation

```
struct Foo doSomething( int a ) {
  struct Foo retval = { a, a + 1 };
  return retval;
}
int main() {
  struct Foo f;
  f = doSomething( 1 );
  return 0;
}
```



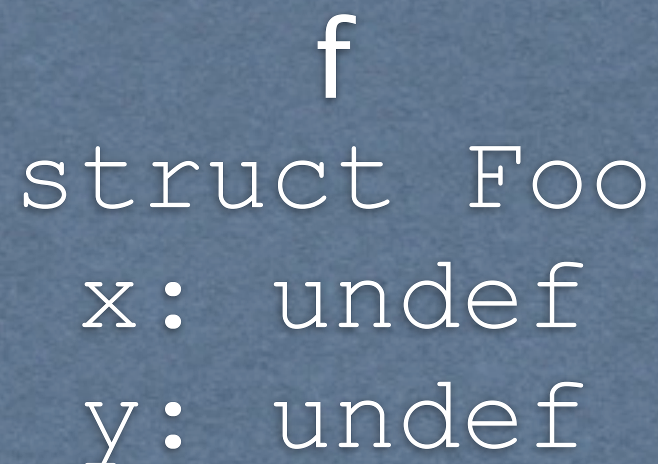
# Internal Representation

```
struct Foo doSomething( int a ) {
  struct Foo retval = { a, a + 1 };
  return retval;
}
int main() {
  struct Foo f;
  f = doSomething( 1 );
  return 0;
}
```



# Internal Representation

```
struct Foo doSomething( int a ) {
  struct Foo retval = { a, a + 1 };
  return retval;
}
int main() {
  struct Foo f;
  f = doSomething( 1 );
  return 0;
}
```



# Internal Representation

```
struct Foo doSomething( int a ) {
  struct Foo retval = { a, a + 1 };
  return retval;
}
int main() {
  struct Foo f;
  f = doSomething( 1 );
  return 0;
}
```

| f | retval |
|---|---|
| struct Foo | struct Foo |
| x: undef | x: 1 |
| y: undef | y: 2 |

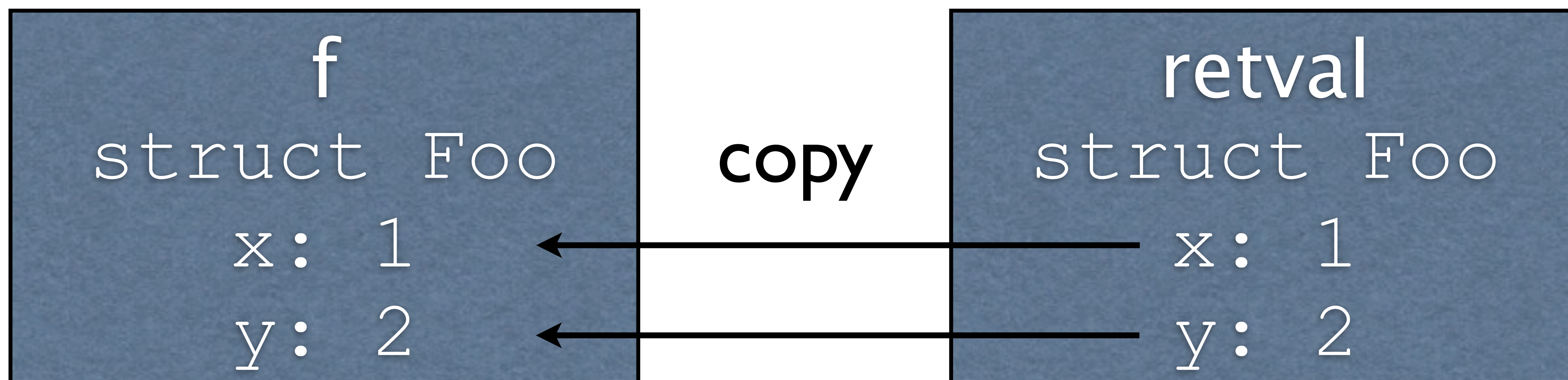
# Internal Representation

```
struct Foo doSomething( int a ) {
  struct Foo retval = { a, a + 1 };
  return retval;
}
int main() {
  struct Foo f;
  f = doSomething( 1 );
  return 0;
}
```

| f | retval |
|---|---|
| struct Foo | struct Foo |
| x: undef | x: 1 |
| y: undef | y: 2 |

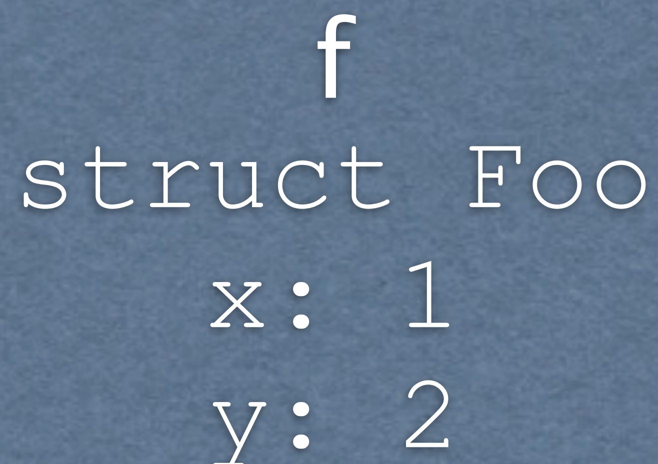
# Internal Representation

```
struct Foo doSomething( int a ) {
  struct Foo retval = { a, a + 1 };
  return retval;
}
int main() {
  struct Foo f;
  f = doSomething( 1 );
  return 0;
}
```



# Internal Representation

```
struct Foo doSomething( int a ) {
  struct Foo retval = { a, a + 1 };
  return retval;
}
int main() {
  struct Foo f;
  f = doSomething( 1 );
  return 0;
}
```



# Returning Structs

- This code has an issue:

```
struct Foo { int x; int y; };

struct Foo* doSomething( int a ) {
  struct Foo retval = { a, a + 1 };
  return &retval;
}
int main() {
  struct Foo* f;
  f = doSomething( 1 );
  return 0;
}
```

# Problem

```
struct Foo retval = { a, a + 1 };
```

Stack

retval
`struct Foo`
`x: 1`
`y: 2`

Heap

# Problem

```
return &retval;
```

retval

Stack

Heap

retval

`struct Foo`

`x: 1`

`y: 2`

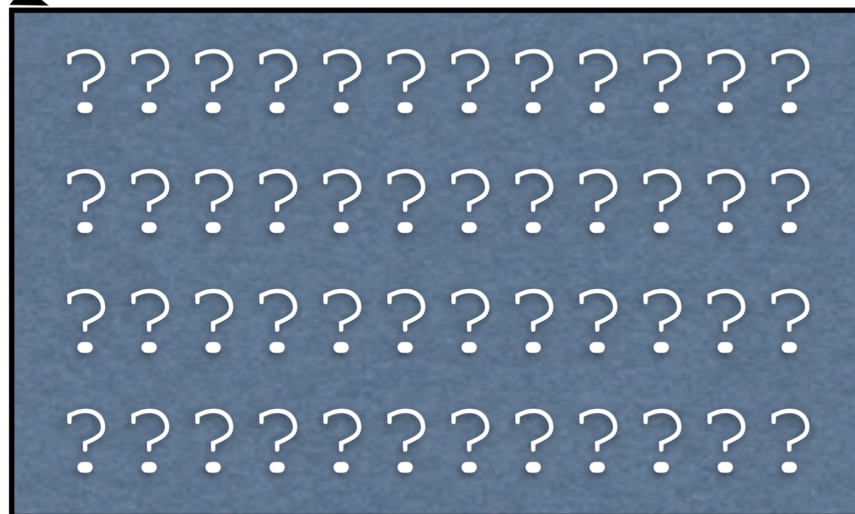
# Problem

<<doSomething returns without copy>>

retval

Stack

Heap



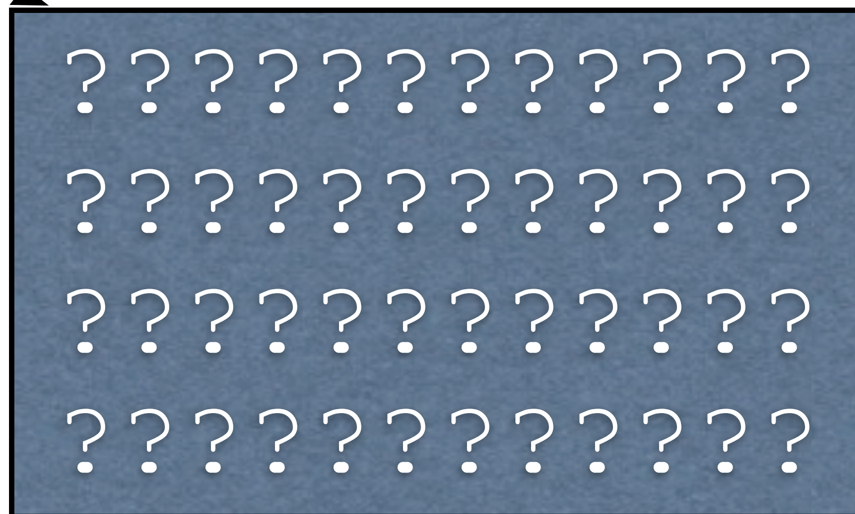
# Problem

```
struct Foo* f = doSomething( 1 );
```

retval

Stack

Heap



# Returning Structs

- This code is ok:

```
struct Foo { int x; int y; };
struct Foo* doSomething( int a ) {
  struct Foo* retval =
    malloc( sizeof( struct Foo ) );
  retval->x = a;
  retval->y = a + 1;
  return retval;
}
int main() {
  struct Foo* f = doSomething( 1 );
  free( f );
  return 0;
}
```

# Why it's Ok

```
struct Foo* retval =
    malloc( sizeof( struct Foo ) );
```

| Stack | Heap |
| --- | --- |
| retval → | struct Foo<br>x: undef<br>y: undef |

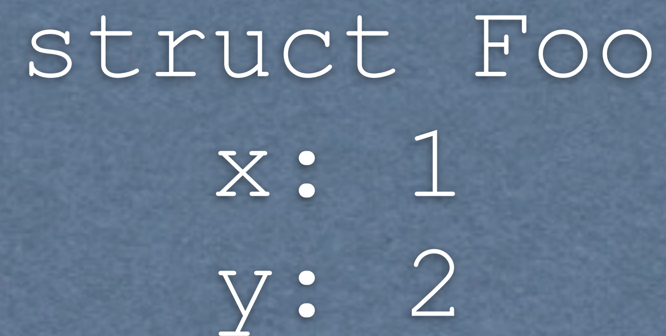
# Why it's Ok

```
retval->x = a;
retval->y = a + 1;
```

Stack

Heap

retval

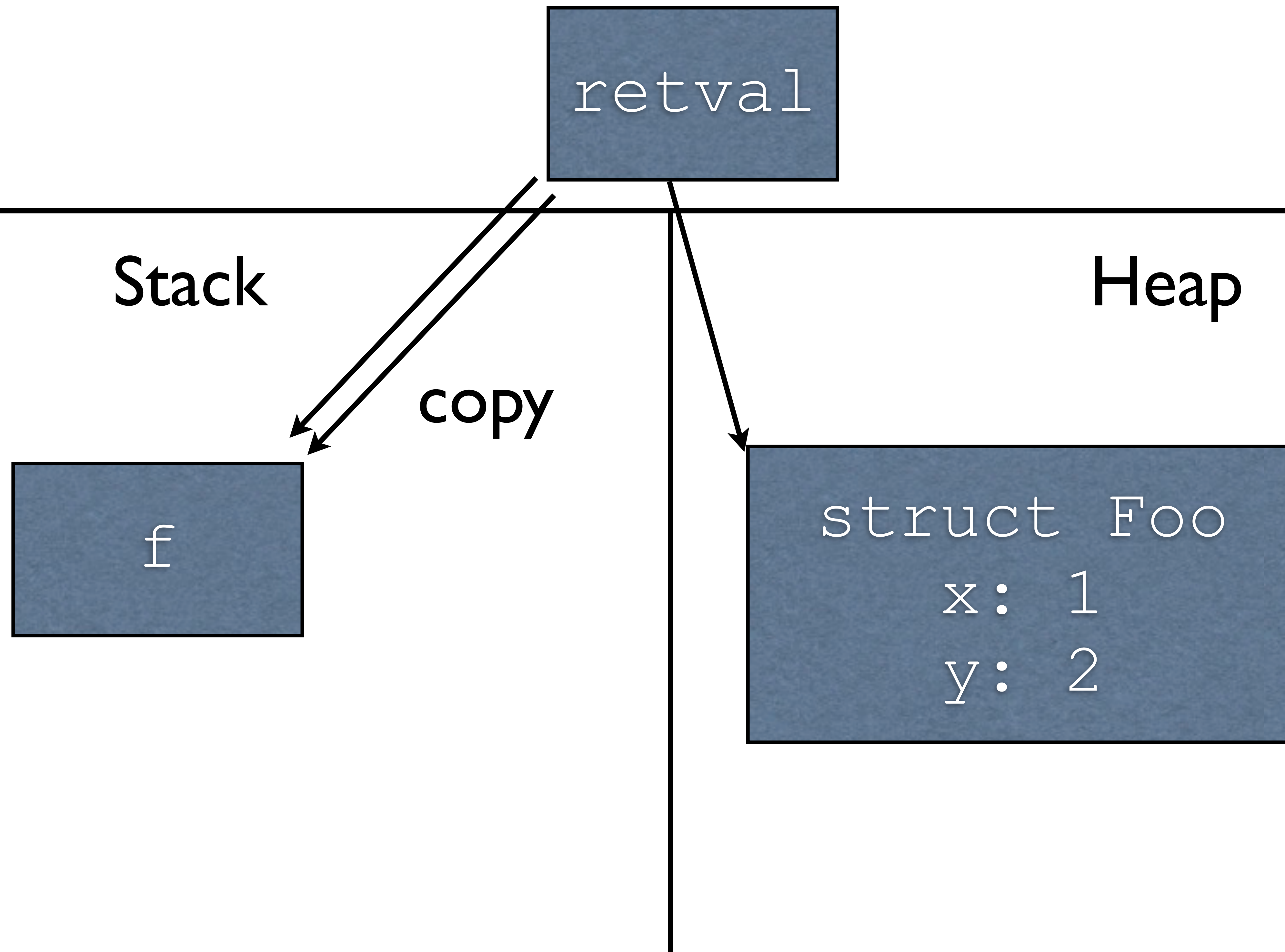


# Why it's Ok

```
return retval;
```

# Why it's Ok

```
struct Foo* f = doSomething( 1 );
```

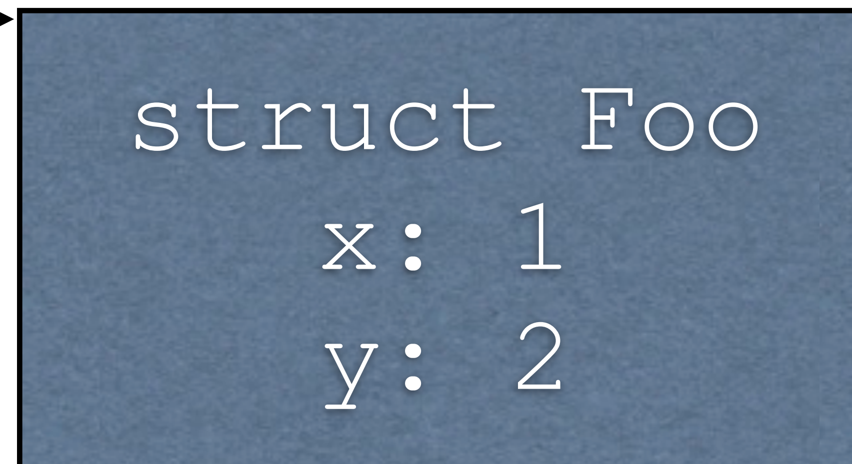


# Why it's Ok

```
struct Foo* f = doSomething( 1 );
```

Stack

Heap

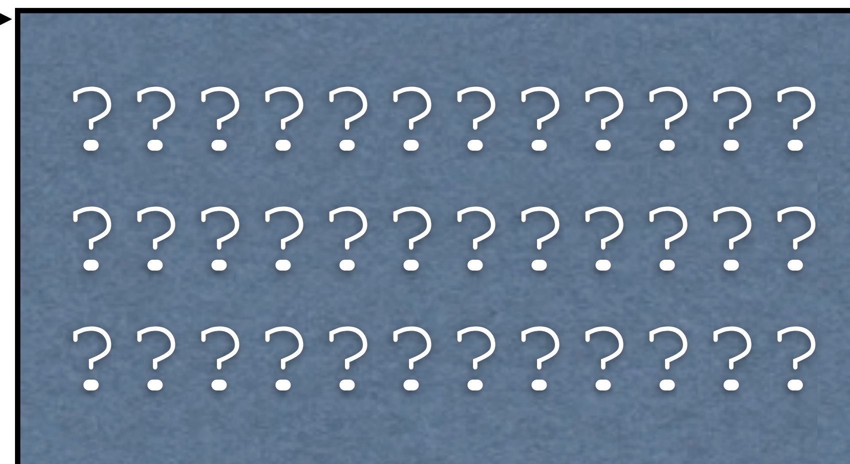


# Why it's Ok

```
free( f );
```

Stack

Heap



# Recall

- The stack is automatically managed
  - Allocation: variable / struct / array declaration
  - Deallocation: leave a block
- The heap is manually managed
  - Allocation: `malloc` / `calloc` / `realloc`
  - Deallocation: `free`

# Assertions

# Assertions

- Used to *assert* that something is true at a given point
- If it is true, the program goes on
- If it is not true, then the program terminates

# Using Assertions

```
#include <assert.h>
int main() {
  assert( 3 * 2 > 7 );
  return 0;
}
```

```
Assertion failed: (3 * 2 > 7),
function main, file assert.c, line 4.
Abort trap: 6
```

# Usefulness

- Great for debugging
- Can put assumptions into code
    - Acts as executable documentation

```
void doSomething( int* pointer ) {
  // assume pointer isn’t NULL
  assert( pointer != NULL );
  printf( “%i\n”, *pointer );
}
```

# Caveats

- They are intended as a debugging tool
- They can be shut off like so:

```
#define NDEBUG
#include <assert.h>
int main() {
  assert( 3 * 2 > 7 );
  return 0;
}
```

# Question #1

```
#include <assert.h>
int main() {
  int x = 0;
  assert( x = 3 );
  // what does x equal?
  return 0;
}
```

# Question #1

```
#include <assert.h>
int main() {
  int x = 0;
  assert( x = 3 );
  // x == 3
  return 0;
}
```

# Question #2

```
#define NDEBUG
#include <assert.h>
int main() {
  int x = 0;
  assert( x = 3 );
  // what does x equal?
  return 0;
}
```

# Question #2

```
#define NDEBUG
#include <assert.h>
int main() {
  int x = 0;
  assert( x = 3 );
  // x = 0
  return 0;
}
```

# The Point

- No work should be done in an assertion

# Exam #2

# Statistics

- Average: 78
- Min: 52
- Max: 97

- A's: 8
- B's: 12
- C's: 15
- D's: 4
- F's: 2

# Course Review