Week 9 Part 1

Kyle Dewey

Overview

- Dynamic allocation continued
- Heap versus stack
- Memory-related bugs
- Exam #2

Dynamic Allocation

Recall...

- Dynamic memory allocation allows us to request memory on the fly
- A way to use only what you need

```
// size_t is an integral defined
// elsewhere
void* malloc( size_t numBytes );
void* calloc( size_t num, size_t size );
void* realloc( void* ptr, size_t size );
```

Recall...

- Requesting some unknown number of integers at runtime

```
int numToAllocate;
scanf( “%i”, &numToAllocate );
int* nums =
  malloc(sizeof( int ) * numToAllocate);
int nums2[ numToAllocate ]; // ERROR
```

Multidimensional Arrays

- Need two separate allocations:
  - Array of columns
  - Each column individually

Example

- Function that makes a matrix with a given number of rows and columns, all initialized to 0

```
int** makeMatrix( int rows, int cols ){
  int** retval =
    calloc(rows, sizeof(int*));
  int row;
  for( row = 0; row < rows; row++ ) {
    retval[ row ] =
      calloc(cols, sizeof(int));
  }
  return retval;
}
```

Question

- What differs here?

```
int** makeMatrix( int rows, int cols ){
  int** retval =
    calloc(rows, sizeof(int*));
  int* temp = calloc(cols, sizeof(int));
  int row;
  for( row = 0; row < rows; row++ ) {
    retval[ row ] = temp;
  }
  return retval;
}
```

With Structs

- Works the same way

```
struct Foo {int x; int y;};
int main() {
   struct Foo* f =
     malloc( sizeof( struct Foo ) );
   f->x = 10;
   f->y = f->x;
   return 0;
}
```

With Structs

- Works the same way

```
struct Foo {int x; int y;};
int main() {
   int x;
   struct Foo** fs =
    calloc( NUM_STRUCTS,
            sizeof( struct Foo* ) );
   for(x = 0; x <  NUM_STRUCTS; x++ ){
     fs[ x ] =
       malloc( sizeof( struct Foo ) );
   }
   return 0;
}
```

Example

- We want to calculate the sample standard deviation of some input numbers
- This formula is below:

$$s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2}$$

Problems

- We need the average in order to calculate it, so we need to keep the numbers around
  - Can't simply calculate as we go
- Don't know how many numbers we need to read in
- Solution: Dynamic memory allocation!

stddev.c

Heap vs. Stack

Recall...

- Say we have multiple function calls

```
int foo() {
   int x = 7;
   return x * 2;
}
int bar() {
   int y = 13;
   return y + foo();
}
int baz() {
   int z = 24;
   return z * 3;
}
```

```
int main() {
   printf( “%i”,
           bar() );
   return 0;
}
```

Recall...

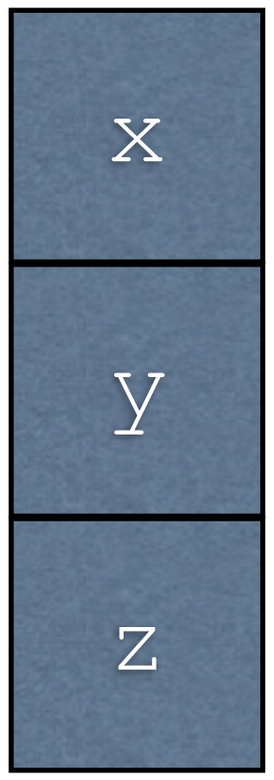
- How is this allocated in memory?

```
int foo() {
   int x = 7;
   return x * 2;
}
int bar() {
   int y = 13;
   return y + foo();
}
int baz() {
   int z = 24;
   return z * 3;
}
```

```
int main() {
   printf( "%i",
           bar() );
   return 0;
}
```

One Possibility

```
int foo() {
   int x = 7;
   return x * 2;
}
int bar() {
   int y = 13;
   return y + foo();
}
int baz() {
   int z = 24;
   return z * 3;
}
```



Address

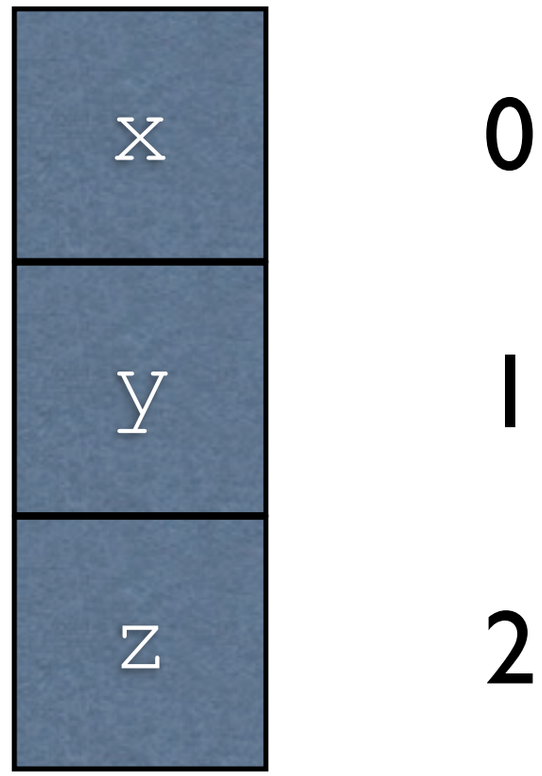
0

1

2

Pros

- Simple
- Fast



Cons

- Wasteful (z is allocated but is unused in this code)
- Does not generally allow for recursion

```
int fact( int n ) {
   if ( n == 0 ) {
      return 1;
   } else {
      return n * fact(n-1);
   }
}
```

Address

0

Another Possibility

- Use a stack
  - As we call functions and declare variables, they get added to the stack
  - As function calls end and variables are no longer alive, they get removed from the stack
  - Do everything relative to the top of the stack

Stack

```
int foo() {
   int x = 7;
   return x * 2;
}
int bar() {
   int y = 13;
   return y + foo();
}
int baz() {
   int z = 24;
   return z * 3;
}
```

```
int main() {
   printf( "%i",
           bar() );
   return 0;
}
```

Address

Stack

```
int foo() {
   int x = 7;
   return x * 2;
}
int bar() {
   int y = 13;
   return y + foo();
}
int baz() {
   int z = 24;
   return z * 3;
}
```

```
int main() {
   printf( "%i",
           bar() );
   return 0;
}
```

Address

Stack

```
int foo() {
   int x = 7;
   return x * 2;
}
int bar() {
   int y = 13;
   return y + foo();
}
int baz() {
   int z = 24;
   return z * 3;
}
```

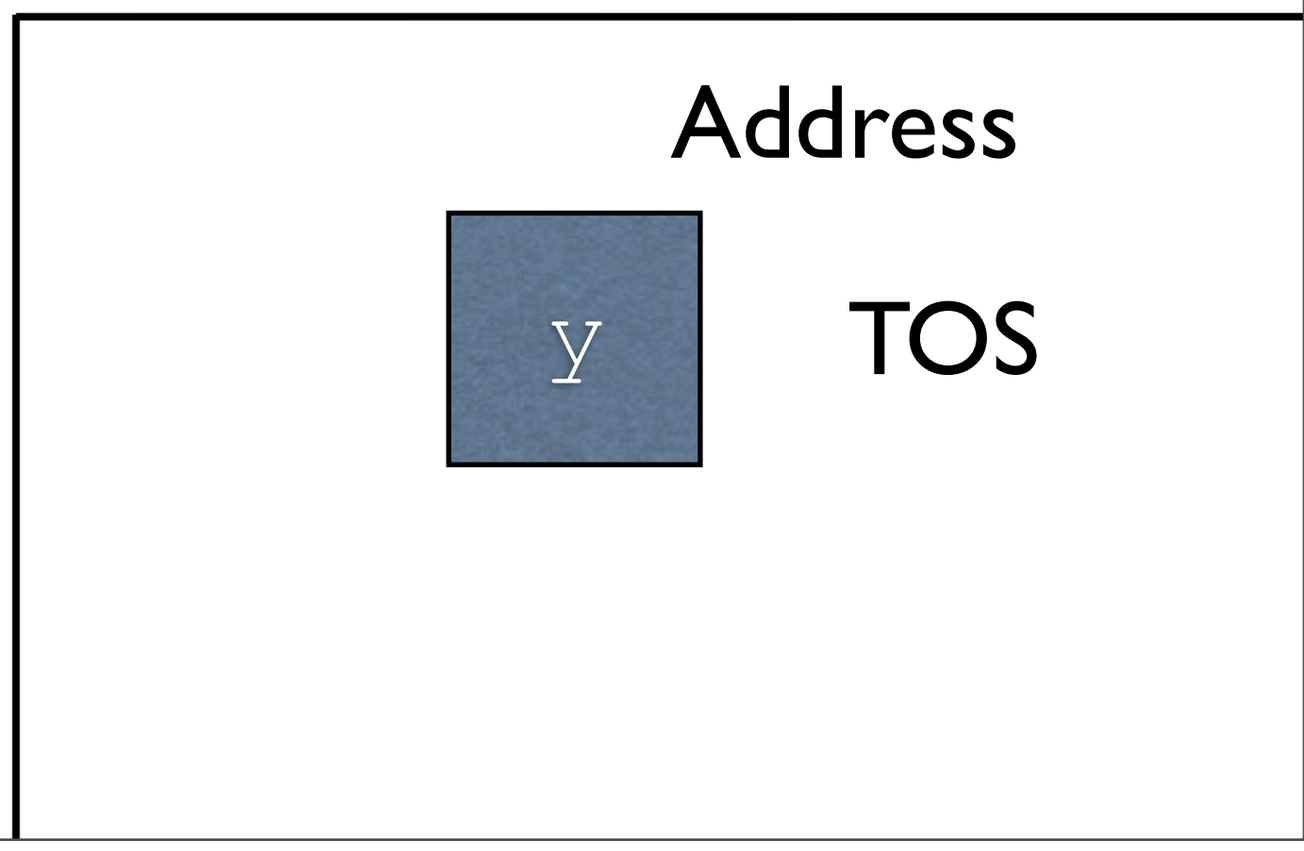
```
int main() {
   printf( "%i",
           bar() );
   return 0;
}
```

Address

Stack

```
int foo() {
   int x = 7;
   return x * 2;
}
int bar() {
   int y = 13;
   return y + foo();
}
int baz() {
   int z = 24;
   return z * 3;
}
```

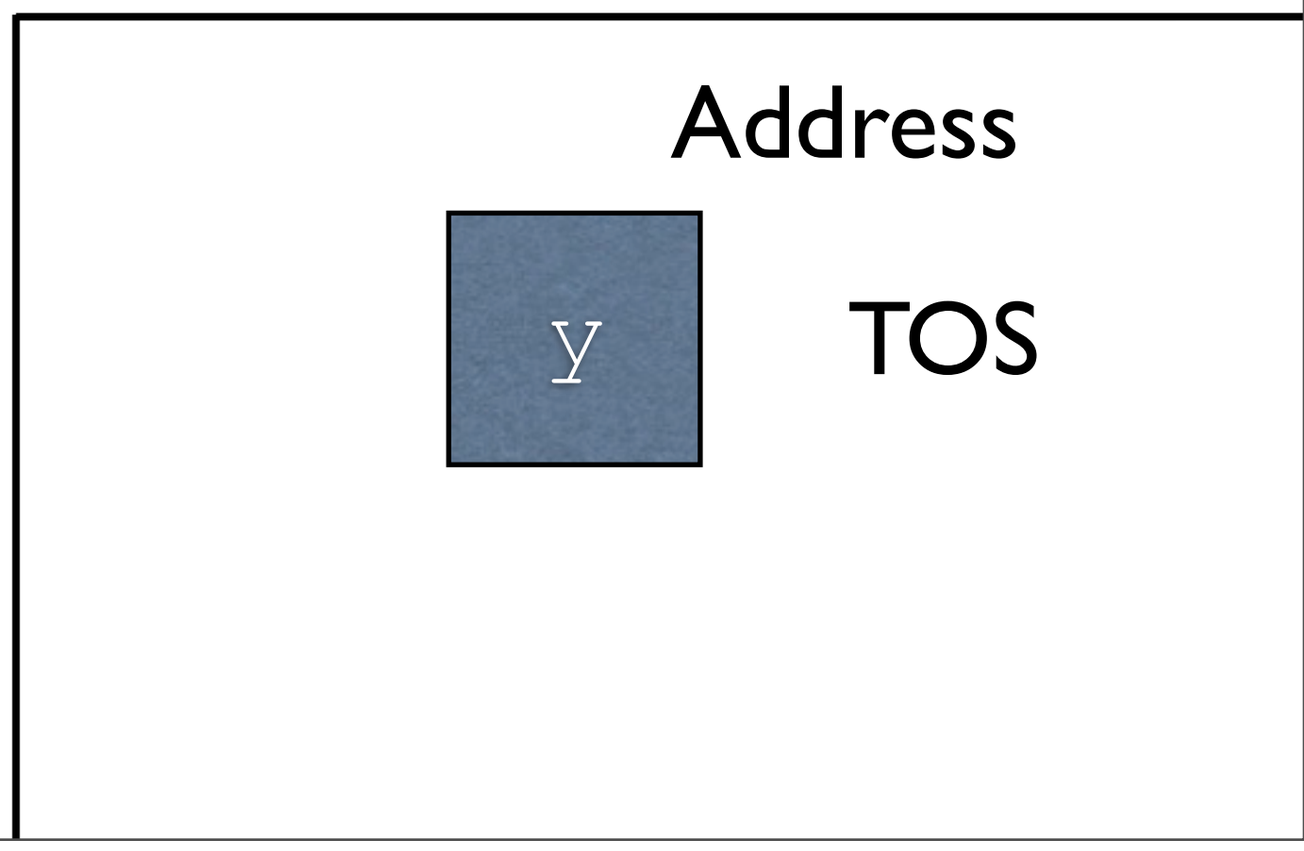
```
int main() {
   printf( "%i",
           bar() );
   return 0;
}
```



Stack

```
int foo() {
   int x = 7;
   return x * 2;
}
int bar() {
   int y = 13;
   return y + foo();
}
int baz() {
   int z = 24;
   return z * 3;
}
```

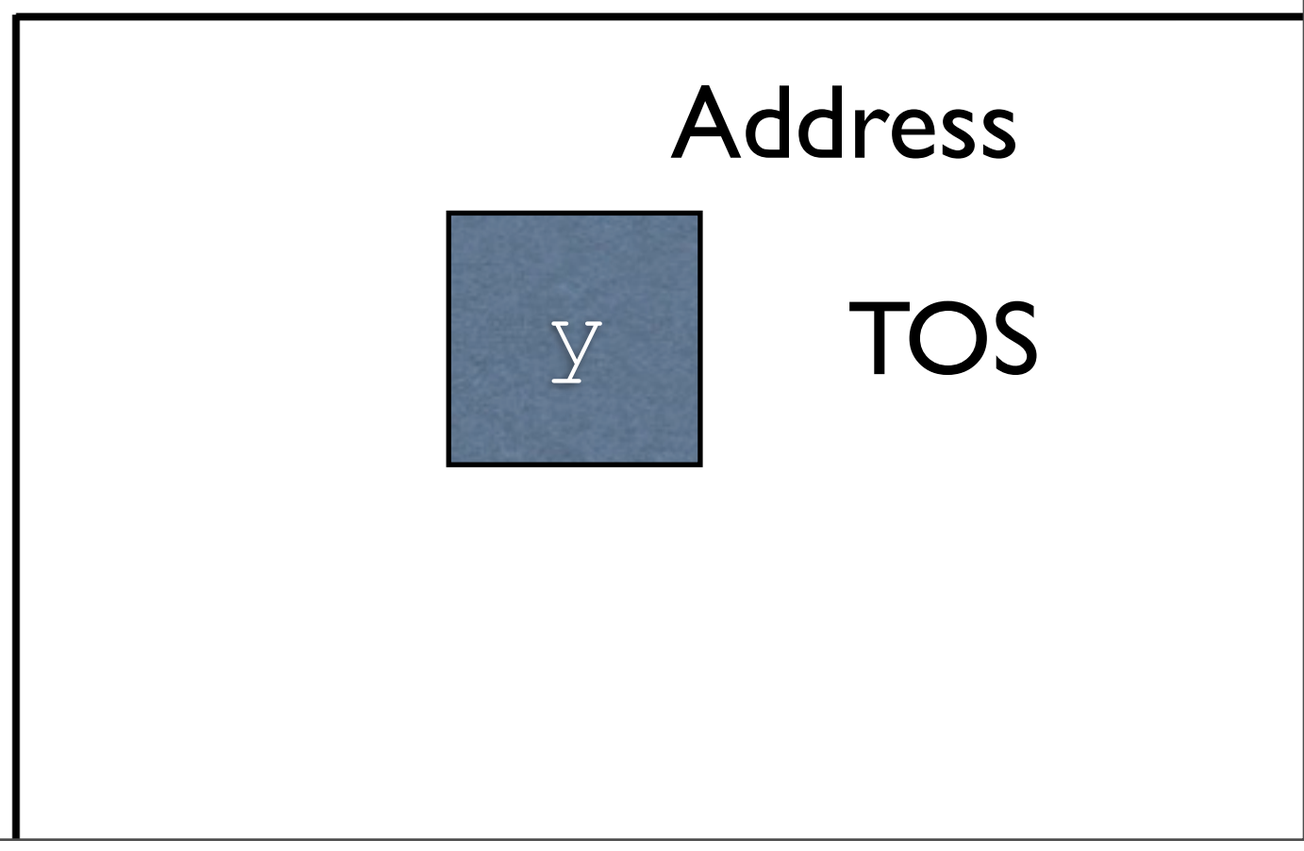
```
int main() {
   printf( "%i",
           bar() );
   return 0;
}
```



Stack

```
int foo() {
   int x = 7;
   return x * 2;
}
int bar() {
   int y = 13;
   return y + foo();
}
int baz() {
   int z = 24;
   return z * 3;
}
```

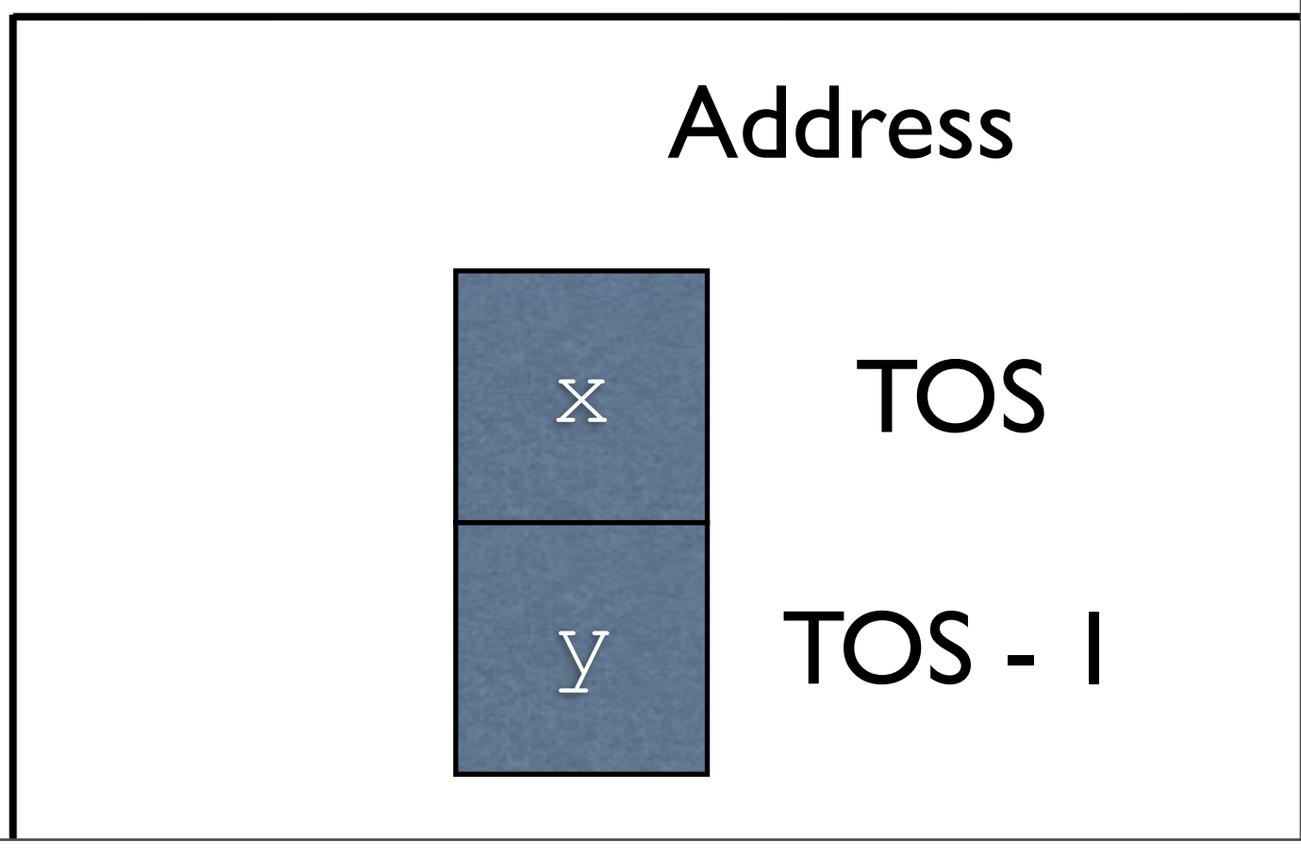
```
int main() {
   printf( "%i",
           bar() );
   return 0;
}
```



Stack

```
int foo() {
   int x = 7;
   return x * 2;
}
int bar() {
   int y = 13;
   return y + foo();
}
int baz() {
   int z = 24;
   return z * 3;
}
```

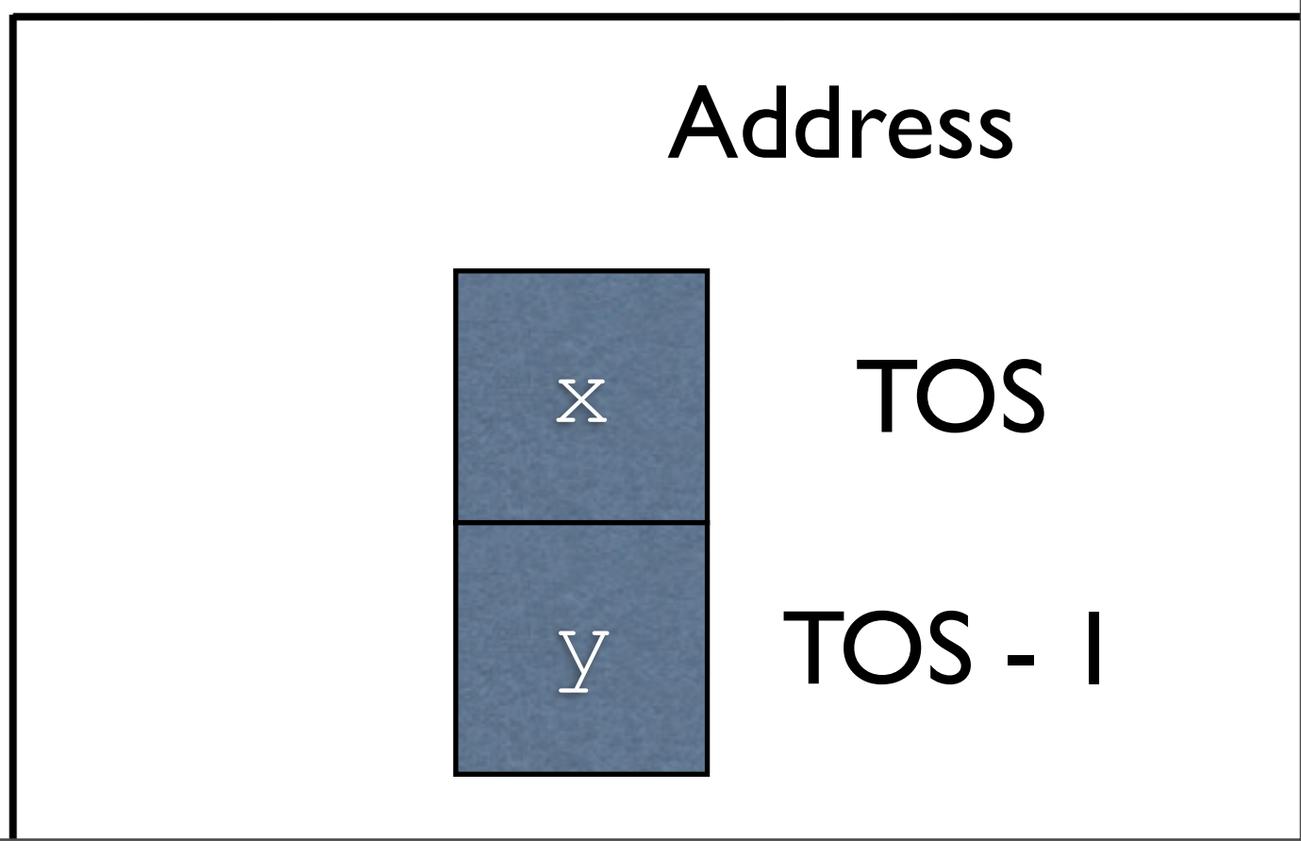
```
int main() {
   printf( "%i",
           bar() );
   return 0;
}
```



Stack

```
int foo() {
   int x = 7;
   return x * 2;
}
int bar() {
   int y = 13;
   return y + foo();
}
int baz() {
   int z = 24;
   return z * 3;
}
```

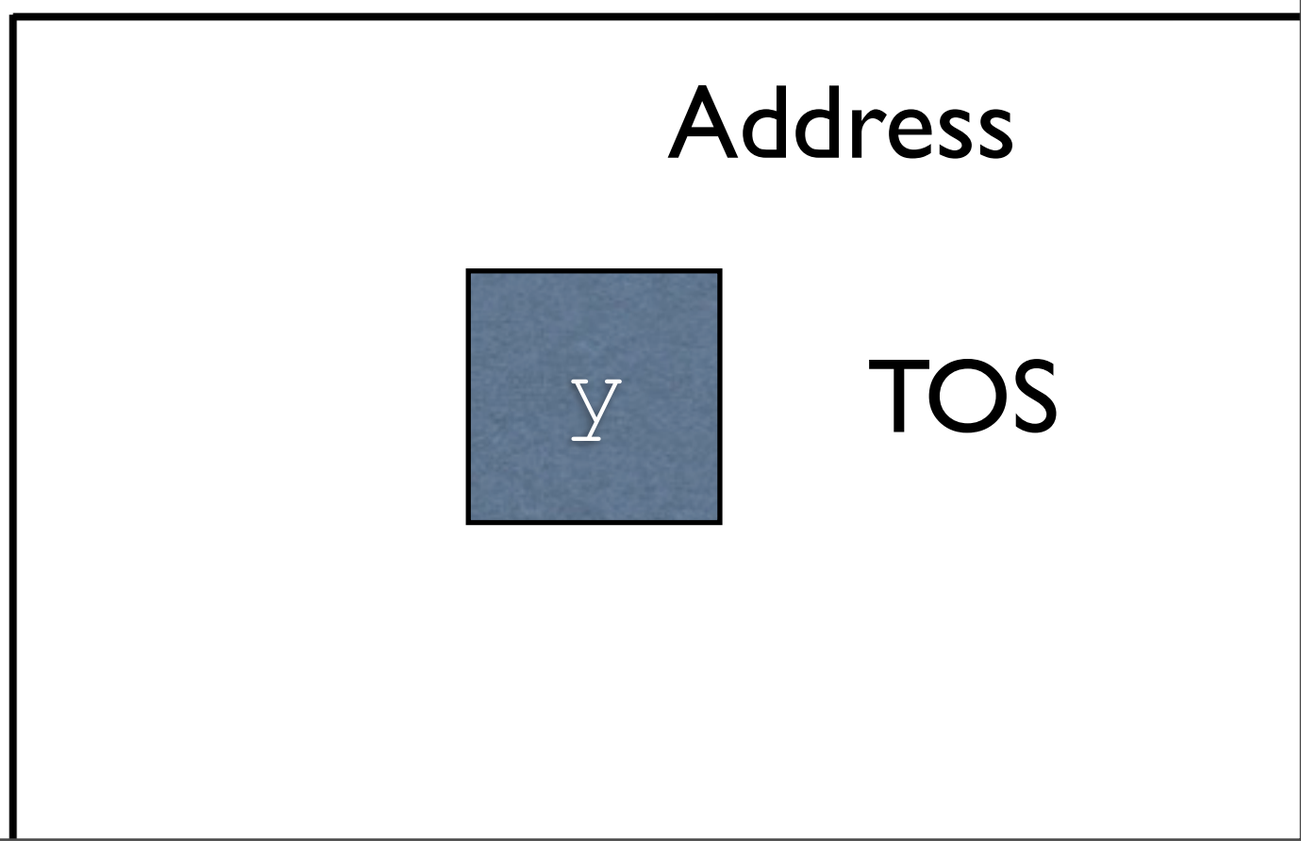
```
int main() {
   printf( "%i",
           bar() );
   return 0;
}
```



Stack

```
int foo() {
   int x = 7;
   return x * 2;
}
int bar() {
   int y = 13;
   return y + foo();
}
int baz() {
   int z = 24;
   return z * 3;
}
```

```
int main() {
   printf( "%i",
           bar() );
   return 0;
}
```



Stack

```
int foo() {
   int x = 7;
   return x * 2;
}
int bar() {
   int y = 13;
   return y + foo();
}
int baz() {
   int z = 24;
   return z * 3;
}
```

```
int main() {
   printf( "%i",
           bar() );
   return 0;
}
```

Address

Notice

- The function `baz` was never called, and the variable `z` was thusly never put on the stack
- At all points, only exact as much memory as was needed was used

Recursion

- Since stacks grow dynamically and allocate on the fly, we can have recursion

```
int fact( int n ) {
   if ( n == 0 ) {
      return 1;
   } else {
      return n * fact(n-1);
   }
}
int main() {
   fact( 5 );
   return 0;
}
```

Recursion

```
int fact( int n ) {
   if ( n == 0 ) {
      return 1;
   } else {
      return n * fact(n-1);
   }
}
int main() {
   fact( 5 );
   return 0;
}
```

Recursion

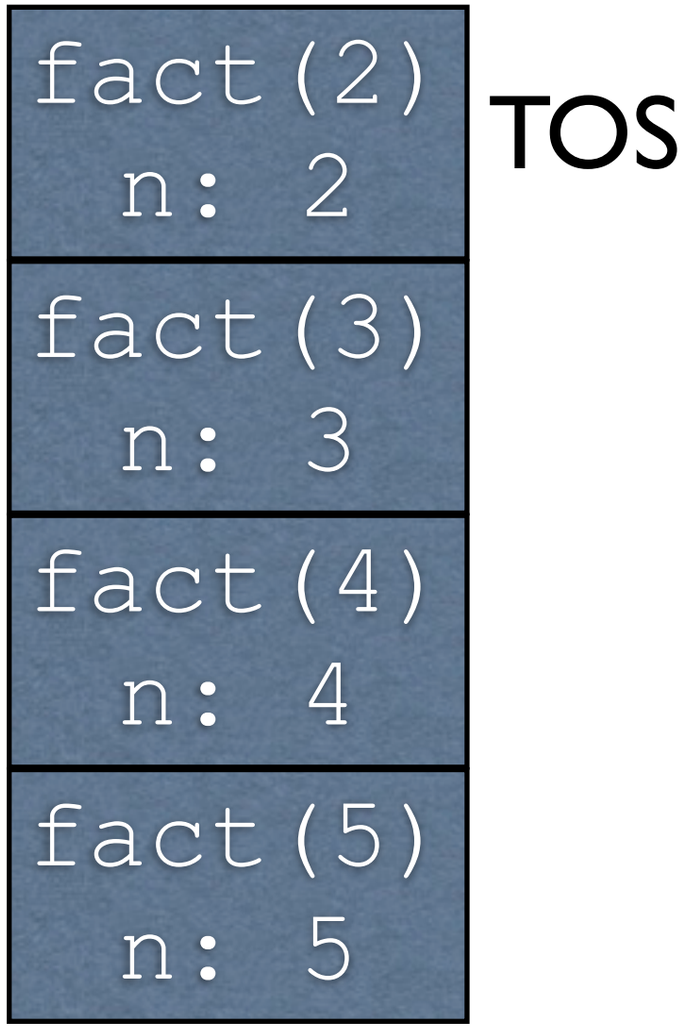
```
int fact( int n ) {
   if ( n == 0 ) {
      return 1;
   } else {
      return n * fact(n-1);
   }
}
int main() {
   fact( 5 );
   return 0;
}
```

Recursion

```
int fact( int n ) {
   if ( n == 0 ) {
      return 1;
   } else {
      return n * fact(n-1);
   }
}
int main() {
   fact( 5 );
   return 0;
}
```

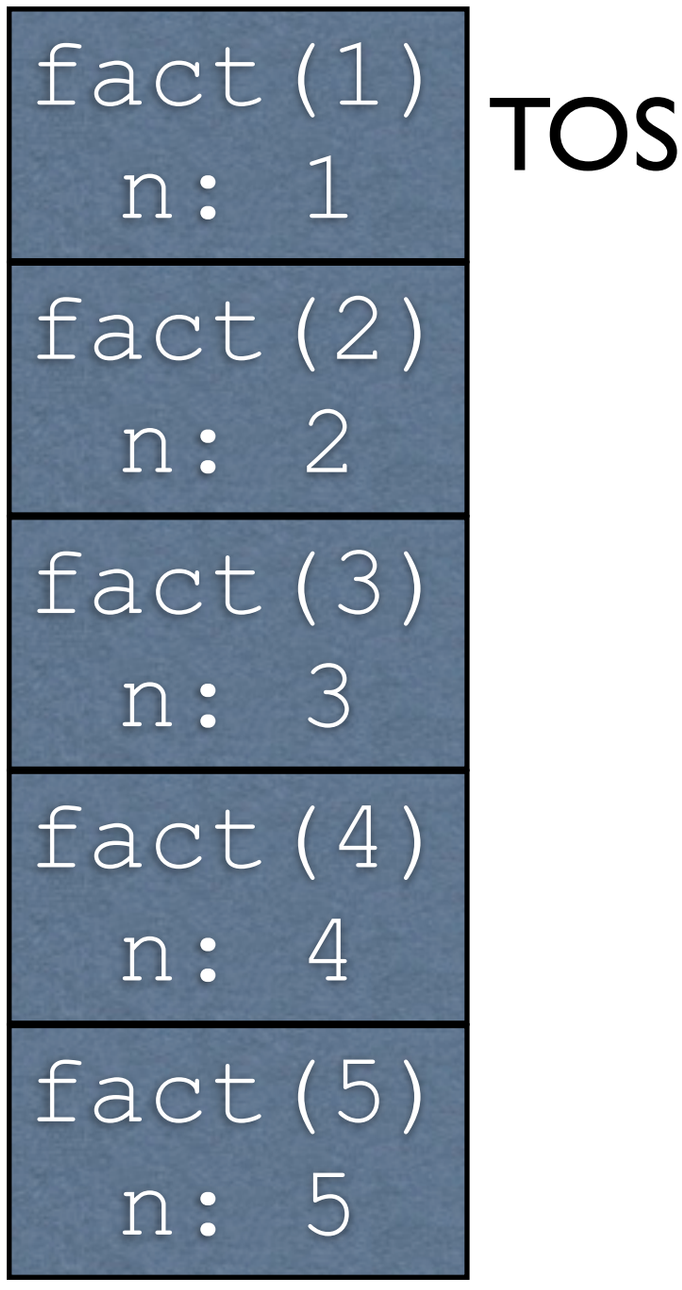
Recursion

```
int fact( int n ) {
   if ( n == 0 ) {
      return 1;
   } else {
      return n * fact(n-1);
   }
}
int main() {
   fact( 5 );
   return 0;
}
```



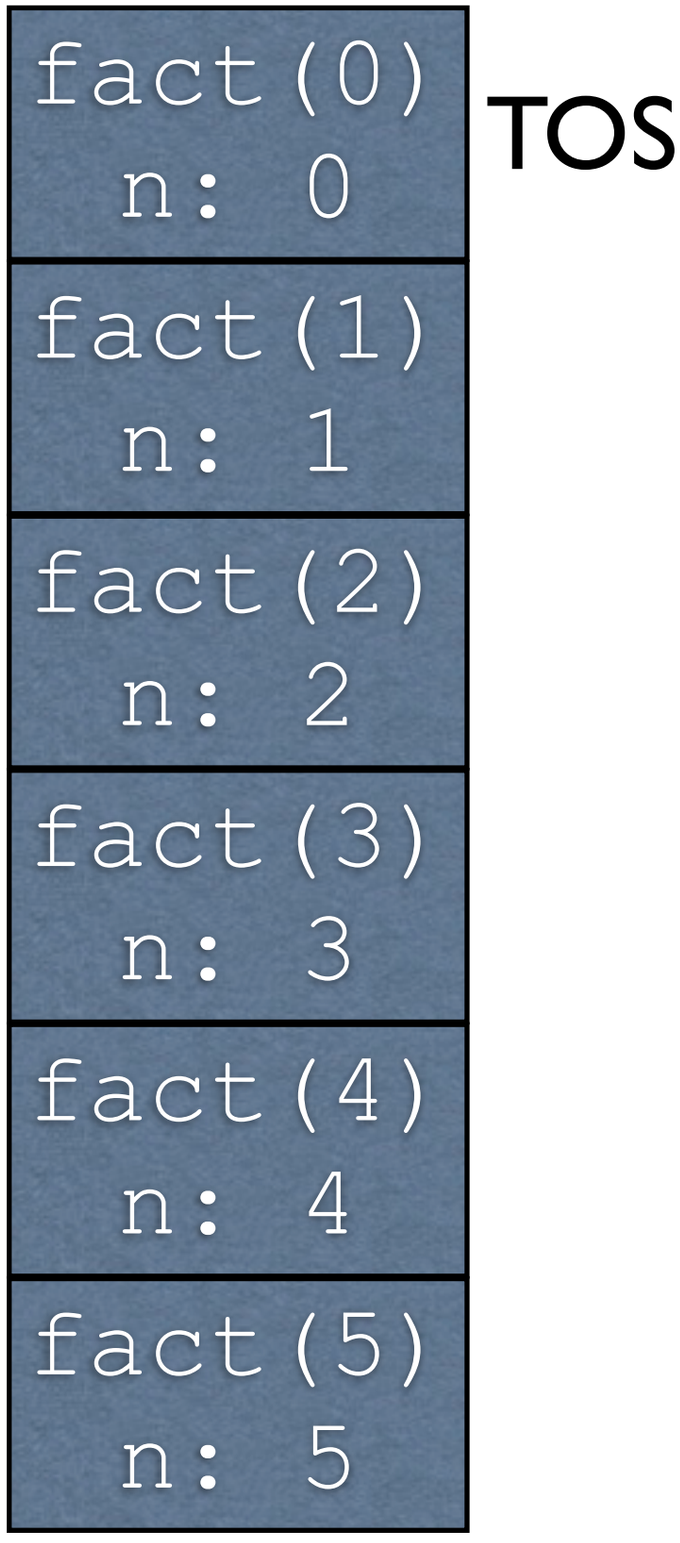
Recursion

```
int fact( int n ) {
   if ( n == 0 ) {
      return 1;
   } else {
      return n * fact(n-1);
   }
}
int main() {
   fact( 5 );
   return 0;
}
```



Recursion

```
int fact( int n ) {
   if ( n == 0 ) {
      return 1;
   } else {
      return n * fact(n-1);
   }
}
int main() {
   fact( 5 );
   return 0;
}
```



Stack Pros/Cons

- Only memory that is required is allocated
- Can have recursion
- Slower (everything now relative to the top of the stack instead of absolute)

Question

- Is there anything odd with this code?

```
int* doSomething() {
   int x;
   return &x;
}
int main() {
   int w;
   int* p = doSomething();
   *p = 5;
   return 0;
}
```

Question Continued

```
int* doSomething() {
   int x;
   return &x;
}
int main() {
   int w;
   int* p = doSomething();
   *p = 5;
   return 0;
}
```

Question Continued

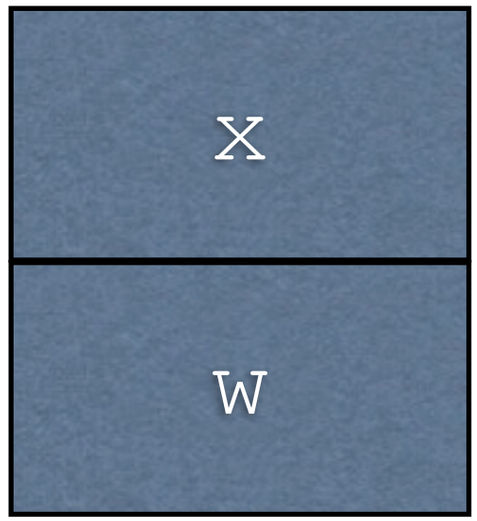
```
int* doSomething() {
   int x;
   return &x;
}
int main() {
   int w;
   int* p = doSomething();
   *p = 5;
   return 0;
}
```

Question Continued

```
int* doSomething() {
   int x;
   return &x;
}
int main() {
   int w;
   int* p = doSomething();
   *p = 5;
   return 0;
}
```

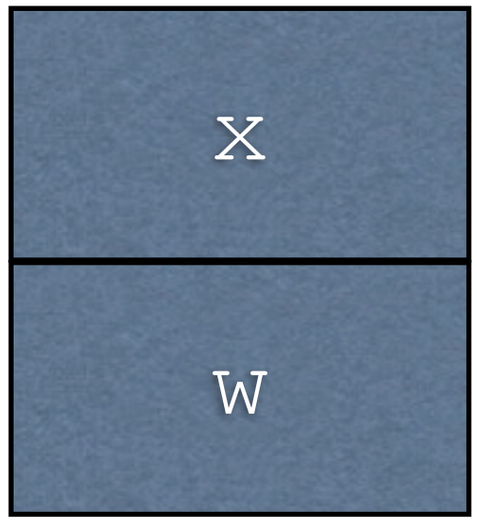
Question Continued

```
int* doSomething() {
   int x;
   return &x;
}
int main() {
   int w;
   int* p = doSomething();
   *p = 5;
   return 0;
}
```



Question Continued

```
int* doSomething() {
   int x;
   return &x;
}
int main() {
   int w;
   int* p = doSomething();
   *p = 5;
   return 0;
}
```



TOS

TOS - 1

Question Continued

```
int* doSomething() {
   int x;
   return &x;
}
int main() {
   int w;
   int* p = doSomething();
   *p = 5;
   return 0;
}
```

TOS

Question Continued

```
int* doSomething() {
   int x;
   return &x;
}
int main() {
   int w;
   int* p = doSomething();
   *p = 5;
   return 0;
}
```

TOS

Once TOS is updated, `w`'s address is the same as `x`'s address

In General

- Stack allocation is done automatically
- Pointers to places on the stack are ok, as long as you make sure the address will always be alive
- Pointers to the stack can be safely passed as function parameters, but not safely returned

Heap

- Dynamic memory allocation allocates from a section of memory known as the heap
- Completely manual allocation
  - Allocate when you need it
  - Deallocate when you don't
- The computer assumes you know what you're doing

Heap Allocation

- Reconsider the code from before:

```
int* doSomething() {
//   int x;
//   return &x;
   return malloc( sizeof( int ) );
}
int main() {
   int w;
   int* p = doSomething();
   *p = 5;
   return 0;
}
```

Heap Allocation

- This code is perfectly fine
  - There is nothing that will be automatically reclaimed
- Can pass pointers any which way

Heap Allocation Bugs

Memory Leak

- Allocated memory is not freed after it is needed (wasted)
- Generally, no pointers exist to the portion allocated, and so it can never be reclaimed
- Why do we need a pointer?

```
void makeLeak() {
  malloc( sizeof( int ) );
}
```

Dangling Pointer

- We keep a pointer to something that has been freed
- That memory can do just about anything after it is freed

```
int* dangle() {
   int* p = malloc( sizeof( int ) );
   free( p );
   *p = 5;
   return p;
}
```

Double Free

- We called free on something that was already freed

```
void doubleFree() {
   int* p = malloc( sizeof( int ) );
   free( p );
   free( p );
}
```

On Memory Bugs

- Determining when something can be freed is generally difficult
  - Theoretically impossible to determine precisely in general
- Try to use stack allocation as much as possible

Exam #2

Statistics

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

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