Programming Abstractions

CS106B

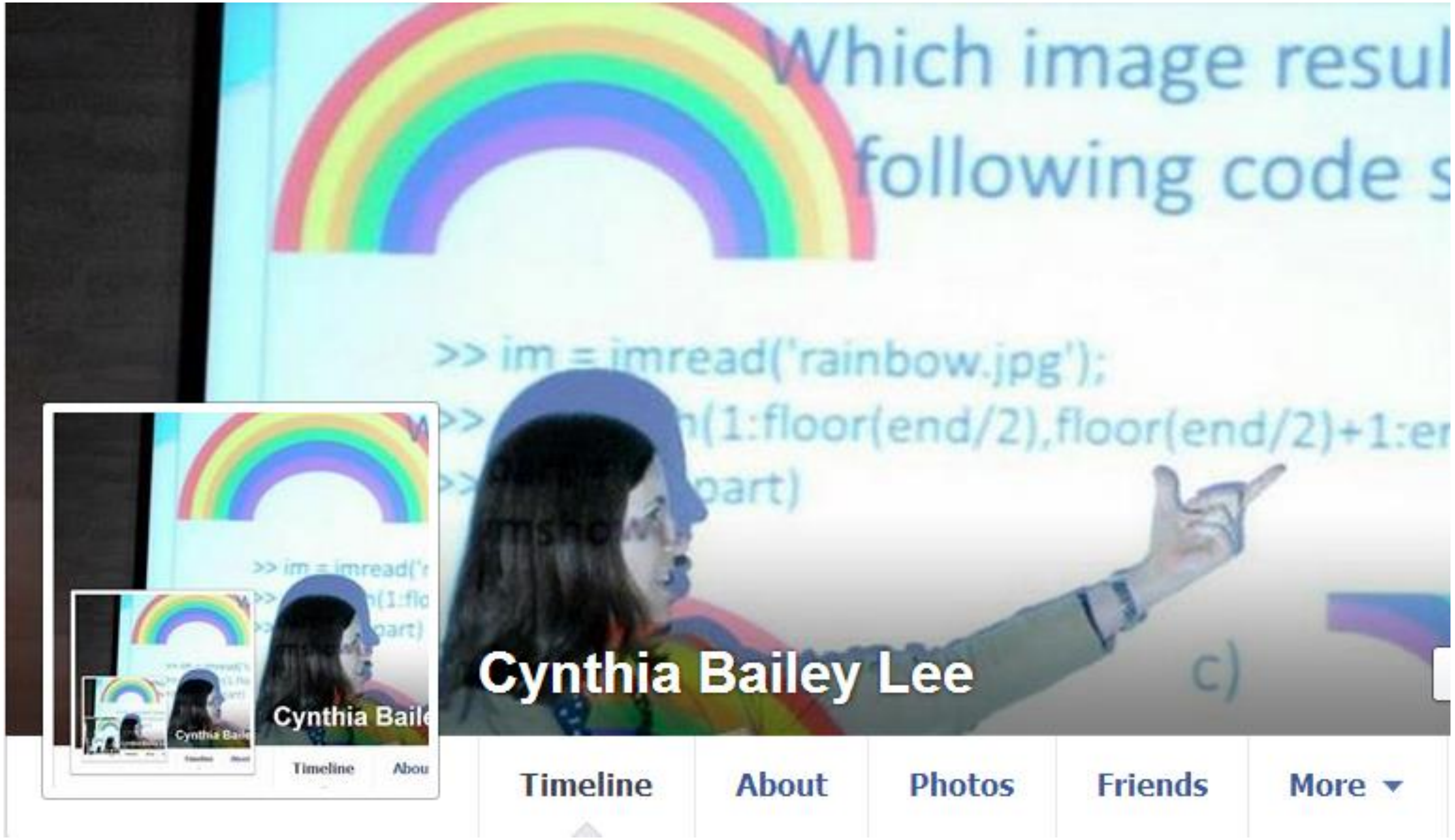
Cynthia Lee

Recursion!

The exclamation point isn’t there only because this is so exciting, it also relates to one of our recursion examples….

Stanford University

Recursion


Which image resul
following code s
>> im = imread('rainbow.jpg');
>> n(1:floor(end/2),floor(end/2)+1:er
>> part)
c)
Cynthia Bailey Lee
Timeline
About
Photos
Friends
More

Factorial!

Recursive mathematical definition

***n*! =**

- if n is 1, then $n! = 1$
- if $n > 1$, then $n! = n * (n - 1)!$
- (0!=1 but for simplicity we'll just consider the domain n>0 for today)

Recursive code

Designing a recursive algorithm

- Recursion is a way of taking a big problem and repeatedly breaking it into smaller and smaller pieces until it is so small that it can be so easily solved that it almost doesn't even need solving.
- There are two parts of a recursive algorithm:
  - base case: where we identify that the problem is so small that we trivially solve it and return that result
  - recursive case: where we see that the problem is still a bit too big for our taste, so we chop it into smaller bits and call *our self* (the function we are in now) on the smaller bits to find out the answer to the problem we face

Factorial!

Recursive definition

***n*! =**

- if n is 1, then $n! = 1$
- if $n > 1$, then $n! = n * (n - 1)!$

Recursive code

```
long factorial ( int n ) {
    if (n==1) return 1;
    else return n*factorial(n-1);
}
```

Factorial!

Recursive definition

***n*! =**

- if *n* is 1, then *n*! = 1
- if *n* > 1, then *n*! = *n* * (*n* – 1)!

Recursive code: imagining more concrete examaples

```
long factorialOf6 () {
       return 6 * factorialOf5();
}

long factorialOf5() {
       return 120;
}
```

Stanford University

Factorial!

Recursive definition

***n*! =**

- if n is 1, then $n! = 1$
- if $n > 1$, then $n! = n * (n - 1)!$

Recursive code: imagining more concrete examaples

```
long factorial ( int n ) {
    if (n==1) return 1;
    else {
        int nminus1fact = pretendIJustMagicallyKnowFactorialOfThis(n-1);
        return n*nminus1fact;
    }
}
```

Factorial!

Recursive definition

***n*! =**

- if *n* is 1, then *n*! = 1
- if *n* > 1, then *n*! = *n* * (*n* – 1)!

Recursive code

```
long factorial ( int n ) {
    if (n==1) return 1;
    else {
        int nminus1fact = factorial(n-1);
       return n*nminus1fact;
    }
}
```

Factorial!

Recursive definition

***n*! =**

- if n is 1, then $n! = 1$
- if $n > 1$, then $n! = n * (n - 1)!$

Recursive code

```
long factorial ( int n ) {
    if (n==1) return 1;
    else return n*factorial(n-1);
}
```

Factorial!

Recursive definition

***n*! =**

- if *n* is 1, then *n*! = 1
- if *n* > 1, then *n*! = *n* * (*n* – 1)!

Recursive code

```
long factorial ( int n ) {
    if (n==1) return 1;
    else return n*factorial(n-1);
}
```

Pro tip: the recursive "leap of faith"

- This concept has become part of the mythology of Stanford's CS106B/X classes. It speaks to the idea that recursion will start to make sense to you when you just trust that the recursive part works.
- One way of tricking your brain into summoning this trust is imagining that the recursive call instead calls some *different* (non-recursive) function that calculates the same thing, like we did at first for factorial().

Digging deeper in the recursion

I know I just told you about the recursive leap of faith and that, for algorithm design purposes, you should mentally flatten the recursion. But before we put that tip into practice, it helps to orient ourselves to the full complexity.

Factorial!

Recursive definition

***n*! =**

- if *n* is 1, then *n*! = 1
- if *n* > 1, then *n*! = *n* * (*n* – 1)!

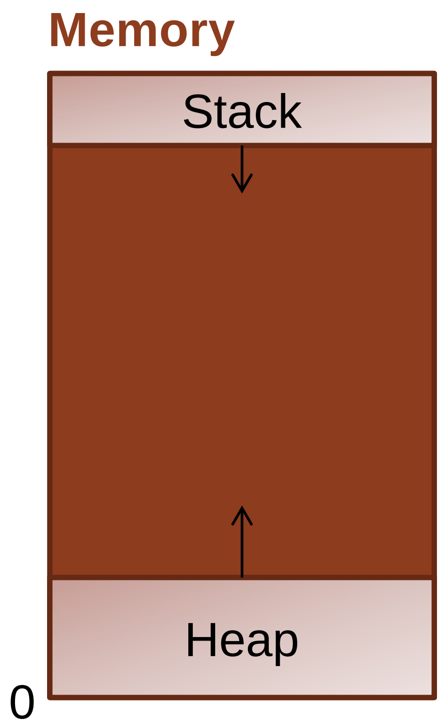
Recursive code

```
long factorial ( int n ) {
    cout << n << endl; //added code
    if (n==1) return 1;
    else return n*factorial(n-1);
}
```

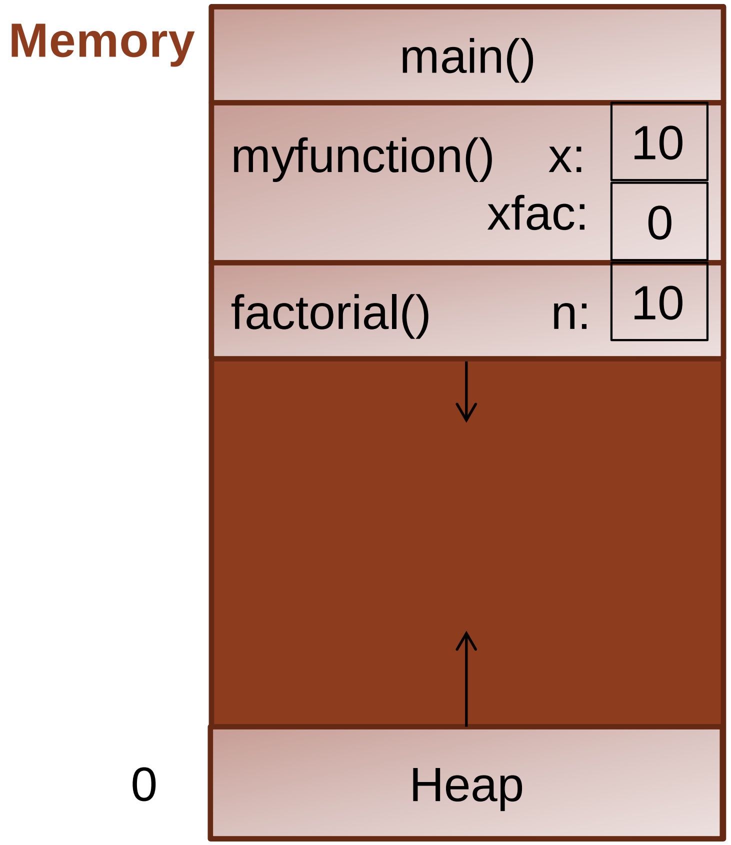
What is the **third** thing **printed** when we call factorial(10)?

A. 2
B. 3
C. 7
D. 8
E. Other/none/more

How does this look in memory?



How does this look in memory?

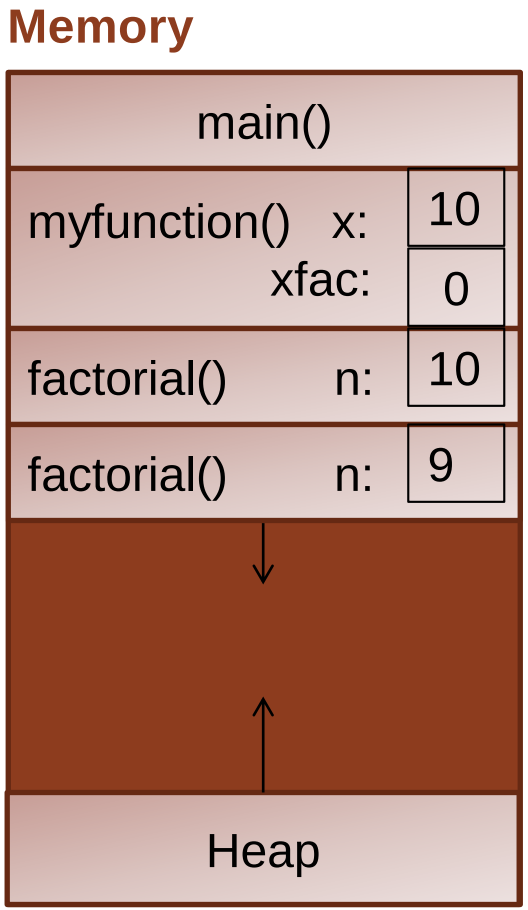


Recursive code

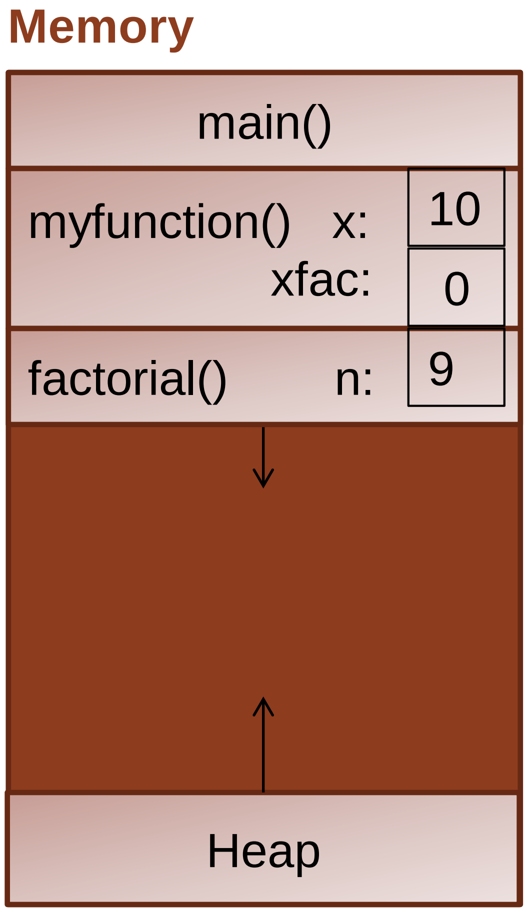
```
long factorial ( int n ) {
     cout << n << endl;
     if (n==1) return 1;
     else return n*factorial(n-1);
}

void myfunction(){
     int x = 10;
     long xfac = 0;
     xfac = factorial(x);
}
```

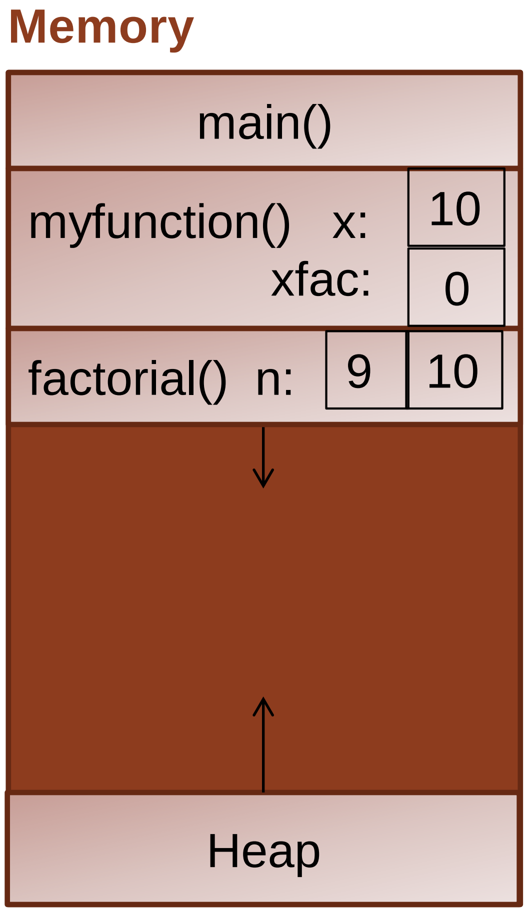
(A)



(B)



(C)



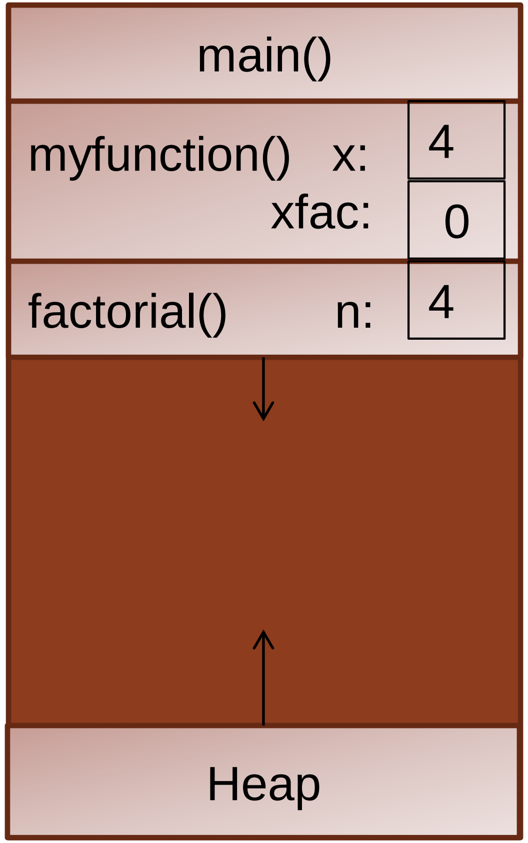
(D) Other/none of the above

The “stack” part of memory is a stack

Function call = push

Return = pop

The “stack” part of memory is a stack

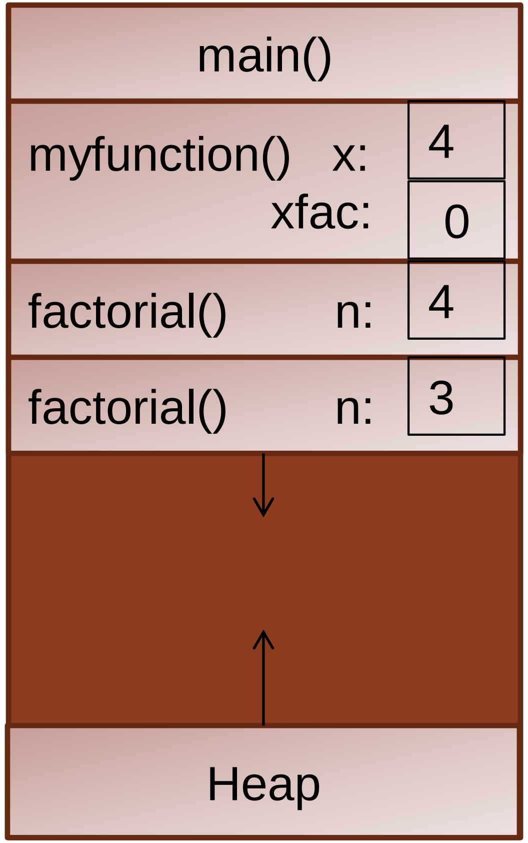


Recursive code

```
long factorial ( int n ) {
    cout << n << endl;
    if (n==1) return 1;
    else return n*factorial(n-1);
}

void myfunction(){
    int x = 4;  //smaller test case
    long xfac = 0;
    xfac = factorial(x);
}
```

The “stack” part of memory is a stack

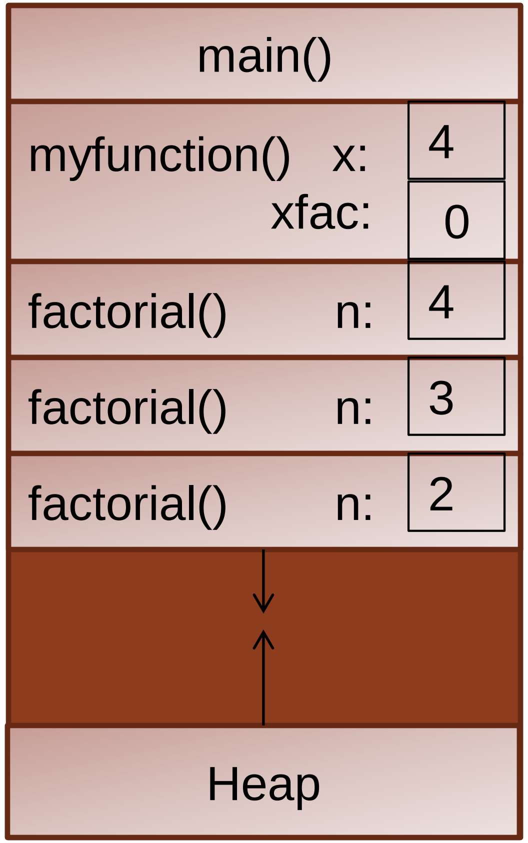


Recursive code

```
long factorial ( int n ) {
    cout << n << endl;
    if (n==1) return 1;
    else return n*factorial(n-1);
}

void myfunction(){
    int x = 4;
    long xfac = 0;
    xfac = factorial(x);
}
```

The "stack" part of memory is a stack

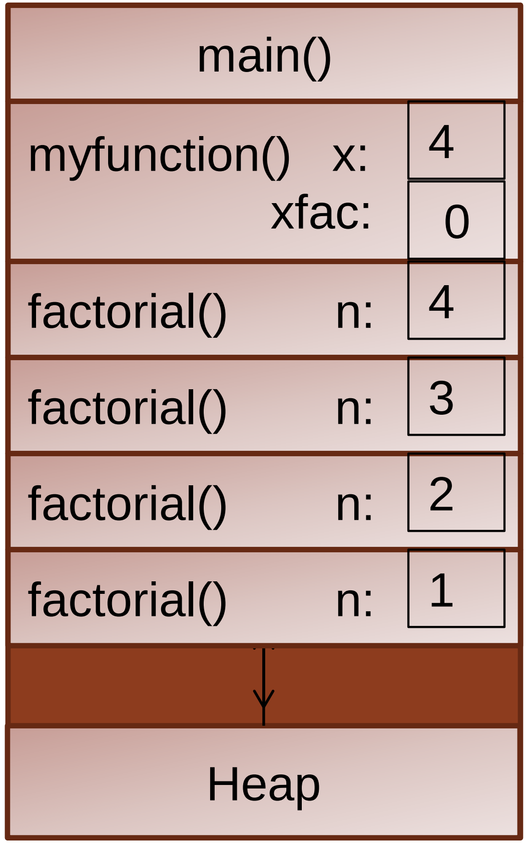


Recursive code

```
long factorial ( int n ) {
    cout << n << endl;
    if (n==1) return 1;
    else return n*factorial(n-1);
}

void myfunction(){
    int x = 4;
    long xfac = 0;
    xfac = factorial(x);
}
```

The “stack” part of memory is a stack



Recursive code

```
long factorial ( int n ) {
    cout << n << endl;
    if (n==1) return 1;
    else return n*factorial(n-1);
}

void myfunction(){
    int x = 4;
    long xfac = 0;
    xfac = factorial(x);
}
```

Stanford University

Factorial!

Recursive definition

***n*! =**

- if *n* is 1, then *n*! = 1
- if *n* > 1, then *n*! = *n* * (*n* – 1)!

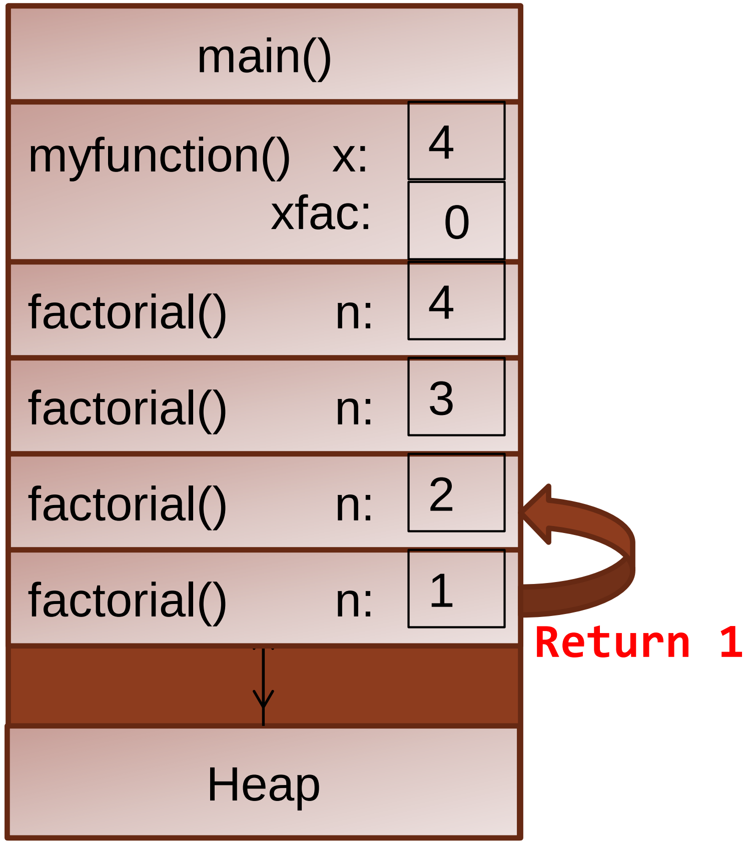
Recursive code

```
long factorial ( int n ) {
    cout << n << endl;
    if (n==1) return 1;
    else return n*factorial(n-1);
}
```

What is the **fourth** value ever **returned** when we call factorial(10)?

A. 4
B. 6
C. 10
D. 24
E. Other/none/more than one

The “stack” part of memory is a stack

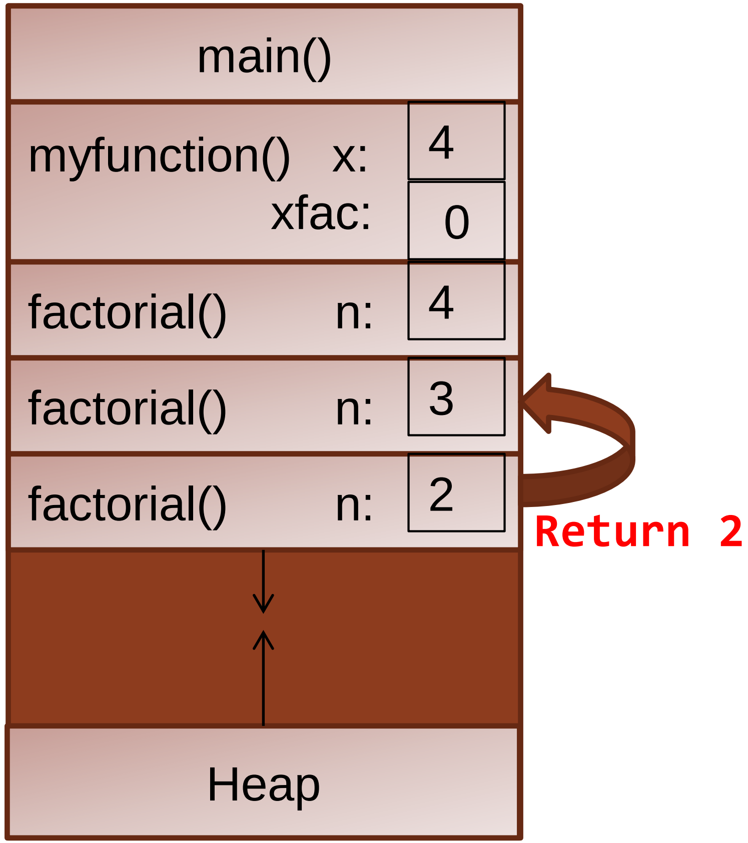


Recursive code

```
long factorial ( int n ) {
    cout << n << endl;
    if (n==1) return 1;
    else return n*factorial(n-1);
}

void myfunction(){
    int x = 4;
    long xfac = 0;
    xfac = factorial(x);
}
```

The “stack” part of memory is a stack

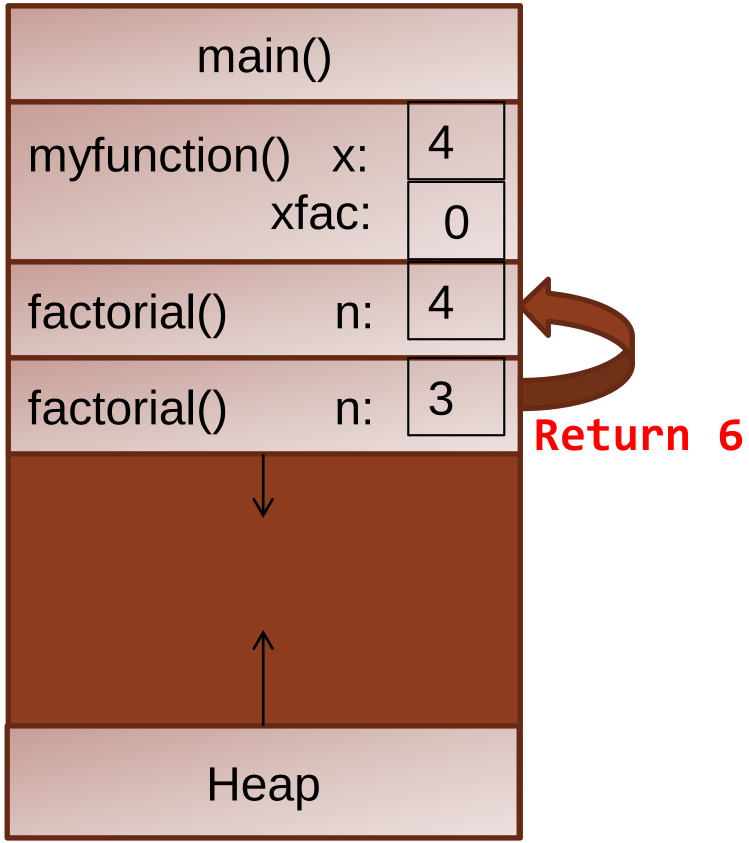


Recursive code

```
long factorial ( int n ) {
    cout << n << endl;
    if (n==1) return 1;
    else return n*factorial(n-1);
}

void myfunction(){
    int x = 4;
    long xfac = 0;
    xfac = factorial(x);
}
```

The "stack" part of memory is a stack



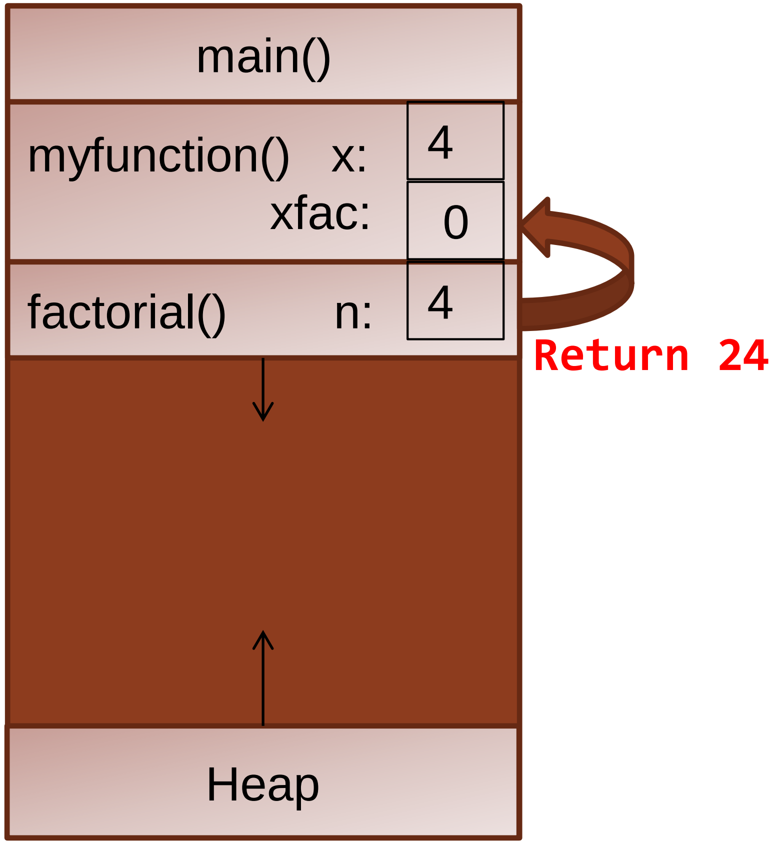
Recursive code

```
long factorial ( int n ) {
    cout << n << endl;
    if (n==1) return 1;
    else return n*factorial(n-1);
}

void myfunction(){
    int x = 4;
    long xfac = 0;
    xfac = factorial(x);
}
```

Stanford University

The “stack” part of memory is a stack



Recursive code

```
long factorial ( int n ) {
     cout << n << endl;
     if (n==1) return 1;
     else return n*factorial(n-1);
}

void myfunction(){
     int x = 4;
     long xfac = 0;
     xfac = factorial(x);
}
```

Factorial!

Iterative version

```
long factorial(int n)
{
  long f = 1;
  while ( n > 1 ) {
    f = f * n;
    n = n – 1;
  }
  return f;
}
```

Recursive version

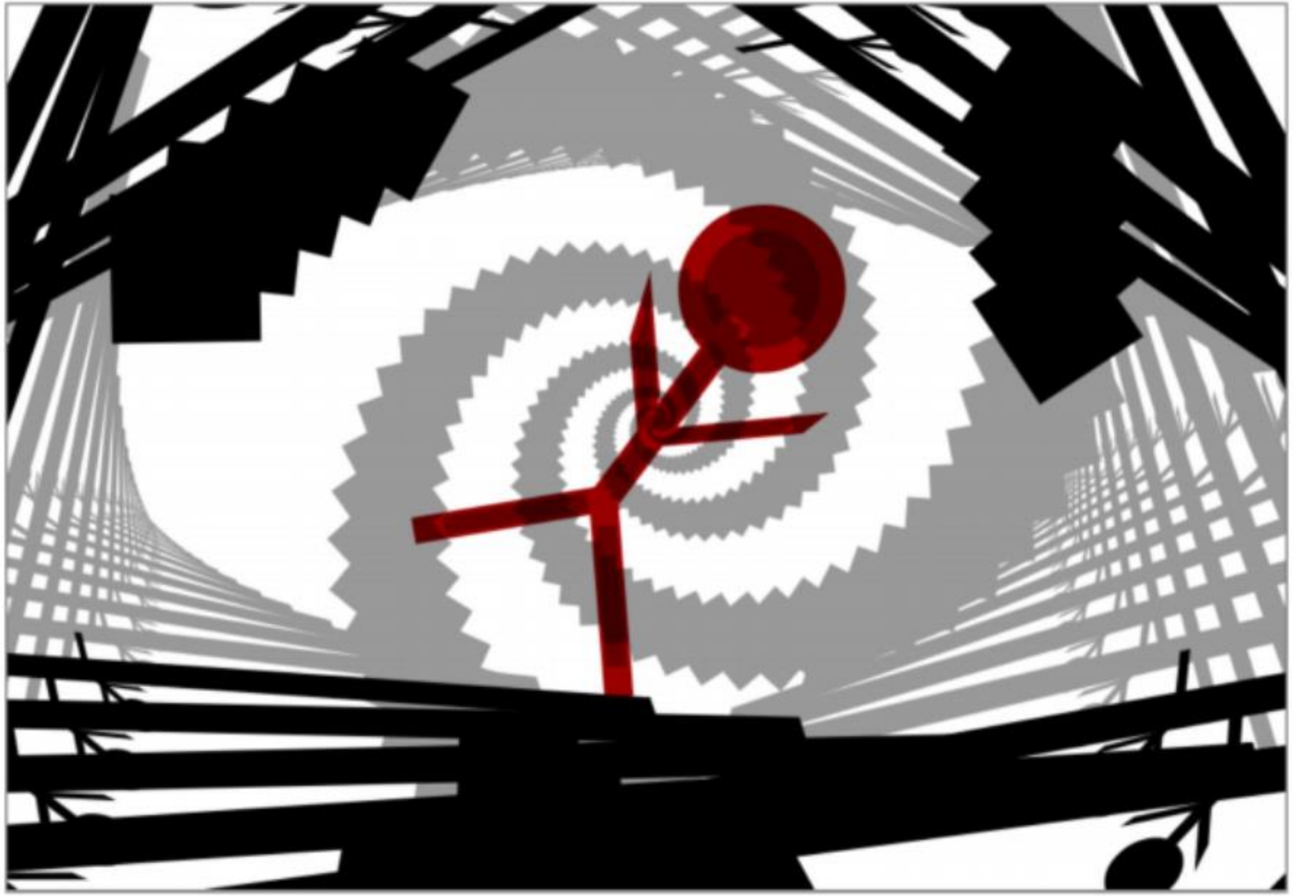
```
long factorial ( int n ) {
    cout << n << endl;
    if (n==1) return 1;
    else return n*factorial(n-1);
}
```

NOTE: sometimes **iterative can be much faster** because it doesn't have to push and pop stack frames. Function calls have overhead in terms of space *and* time to set up and tear down.

Announcement: Recursive art contest!

- Go to http://recursivedrawing.com/
- Make recursive art
  - Win prizes!
- Come to my office hours and see my Wall of Fame of past recursive art submissions!
- Submission deadline:
  - Wednesday of Week 4 (October 14)
- Submission procedure:
  - Email me: cbl@stanford.edu

Art contest



Catherine Wong

Autumn 2013

Wall of Fame

Classic CS problem: searching

Imagine storing **sorted** data in an array

How long does it take us to find a number we are looking for?

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

Imagine storing **sorted** data in an array

How long does it take us to find a number we are looking for?

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

If you start at the front and proceed forward, each item you examine rules out 1 item

Imagine storing **sorted** data in an array

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

If instead we **jump right to the middle**, one of three things can happen:

1. The middle one happens to be the number we were looking for, yay!
2. We realize we went too far
3. We realize we didn't go far enough

Imagine storing **sorted** data in an array

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

If instead we **jump right to the middle**, one of three things can happen:

1. The middle one happens to be the number we were looking for, yay!
2. We realize we went too far
3. We realize we didn't go far enough

Ruling out HALF the options in one step is so much faster than only ruling out one!

Binary search

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

Let’s say the answer was 3, “we didn’t go far enough”

We ruled out the entire first half, and now only have the second half to search

We could start at the front of the second half and proceed forward…

Binary search

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

Let’s say the answer was 3, “we didn’t go far enough”

We ruled out the entire first half, and now only have the second half to search

We could start at the front of the second half and proceed forward…but why do that when we know we have a better way?

Jump right to the middle of the region to search

Binary search

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

Let’s say the answer was 3, “we didn’t go far enough”

We ruled out e en elf and now only have the secon

We could econd half and pr that when we know we ha

Jump **e** of the region to search

RECURSION!!

Stanford University

Designing a recursive algorithm

- Recursion is a way of taking a big problem and repeatedly breaking it into smaller and smaller pieces until it is so small that it can be so easily solved that it almost doesn't even need solving.
- There are two parts of a recursive algorithm:
  - base case: where we identify that the problem is so small that we trivially solve it and return that result
  - recursive case: where we see that the problem is still a bit too big for our taste, so we chop it into smaller bits and call *our self* (the function we are in now) on the smaller bits to find out the answer to the problem we face

To write a recursive function, we need base case(s) and recursive call(s)

What would be a good base case for our Binary Search function?

A. Only three items remain: save yourself an unnecessary function call that would trivially divide them into halves of size 1, and just check all three.

B. Only two items remain: can’t divide into two halves with a middle, so just check the two.

C. Only one item remains: just check it.

D. No items remain: obviously we didn’t find it.

E. More than one

Binary Search

```
bool binarySearch(Vector<int>& data, int key){
  return binarySearch(data, key, 0, data.size()-1);
}

bool binarySearch(Vector<int>& data, int key,
     int start, int end){

    //to be continued…
}
```