# Programming Abstractions

CS106B

Cynthia Lee

# Today's Topics

Abstract Data Types

- One final detail: containers containing containers
  - Containerception!

Recursion!

- One final detail: containers containing containers

Next time:

- More recursion! It's Recursion Week!
- Like Shark Week, but more nerdy

# Compound Containers

It's turtles all the way down...

# Comparing two similar codes:

```
Vector<int> numbers;
numbers.add(1);
numbers.add(2);
numbers.add(3);
Map<string, Vector<int>> mymap;
mymap["123"] = numbers;
```

Code option #1

```
mymap["123"].add(4);
```

```
cout << "New size: " << mymap["123"].size() << endl;
```

# Comparing two similar codes:

```
Vector<int> numbers;
numbers.add(1);
numbers.add(2);
numbers.add(3);
Map<string, Vector<int>> mymap;
mymap["123"] = numbers;
```

Code option #2

```
Vector<int> test = mymap["123"];
test.add(4);
```

```
cout << "New size: " << mymap["123"].size() << endl;
```

# Comparing two similar codes:

```
Vector<int> numbers;
numbers.add(1);
numbers.add(2);
numbers.add(3);
Map<string, Vector<int>> mymap;
mymap["123"] = numbers;
```

Code option #1

```
mymap["123"].add(4);
```

Code option #2

```
Vector<int> test = mymap["123"];
test.add(4);
```

```
cout << "New size: " << mymap["123"].size() << endl;
```

**Predict the outcome:**

(A) Both print 3 (B) Both print 4 (C) One prints 3, other prints 4

(D) Something else or error

# Comparing two similar codes:

You don't need to worry too much about the details of how the two cases differ in terms of behind-the-scenes mechanism—I just wanted to flag it as a potential issue in case you accidentally encounter this in your code.

```
ctor<int> numbers;
nbers.add(1);
nbers.add(2);
nbers.add(3);
o<string, Vector<int>> mymap;
nap["123"] = numbers;
```

Code option #2

```
mymap["123"].add(4);
```

```
Vector<int> test = mymap["123"];
test.add(4);
```

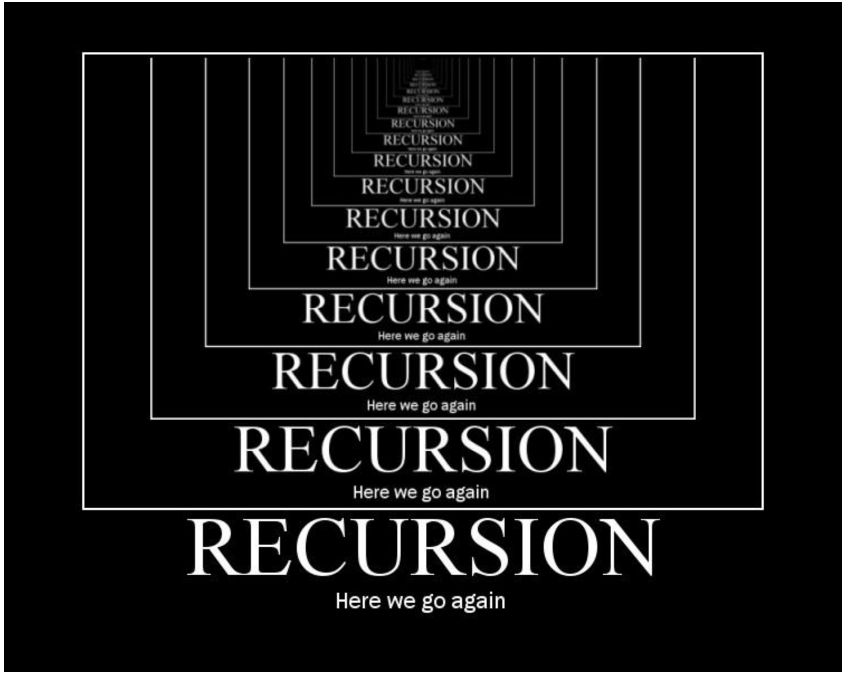
```
cout << "New size: " << mymap["123"].size() << endl;
```

**Predict the outcome:**

(A) Both print 3 (B) Both print 4 (C) One prints 3, one prints 4
(D) Something else or error

# Recursion!

The exclamation point isn't there only because this is so exciting; it also relates to our first recursion example….



# 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 * pretendIJustMagicallyKnowFactorialOfThis(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);
    }
}
```

# 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) {   // Easy! Return trivial answer
        return 1;
    } else {         // Not easy enough yet! Break into "smaller" problem
        return n * factorial(n - 1);  // delegate smaller problem
    }
}
```

# 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 **ourselves** (the function we are in now) on the smaller bits to find out the answer to the problem we face

# Digging deeper in the recursion

Looking at how recursion works “under the hood”

# Factorial!

## Recursive definition

***n*! =**

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

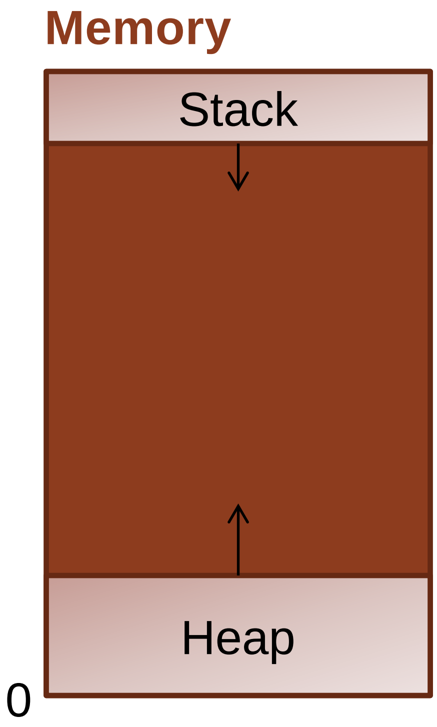
## Recursive code

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

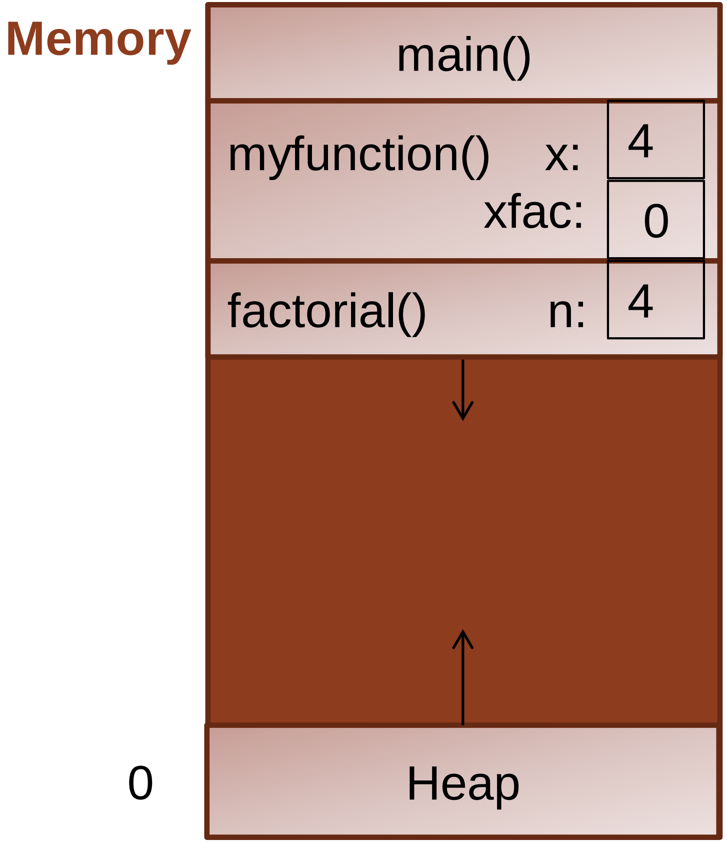
What is the **third** thing **printed** when we call factorial(4)?

A. 1
B. 2
C. 3
D. 4
E. Other/none/more

# How does this look in memory?



# How does this look in memory?

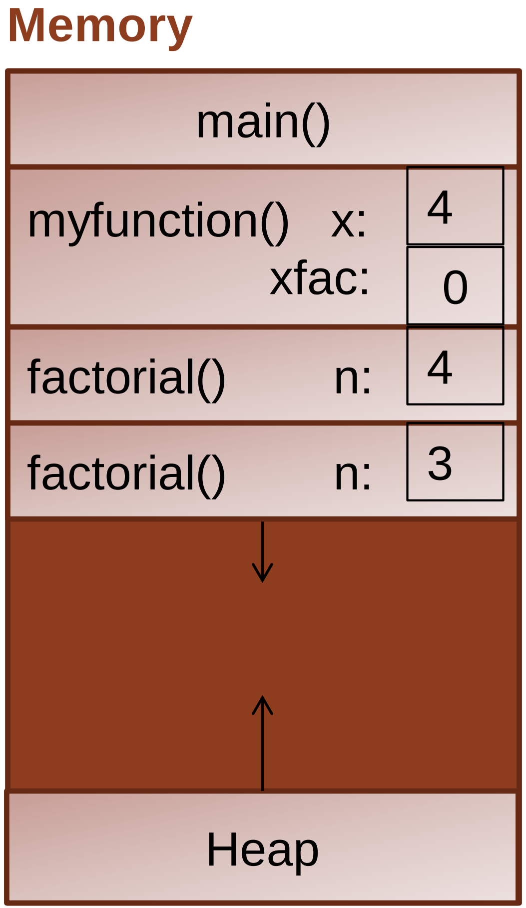



**Recursive code**

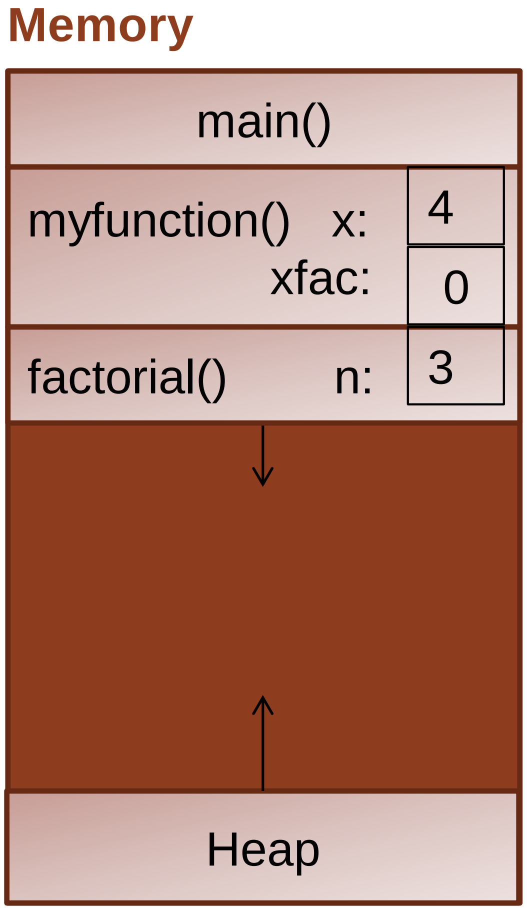
```
long factorial(int n) {
    cout << n << endl;
    if (n == 1) return 1;
    else return n * factorial(n – 1);
}

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

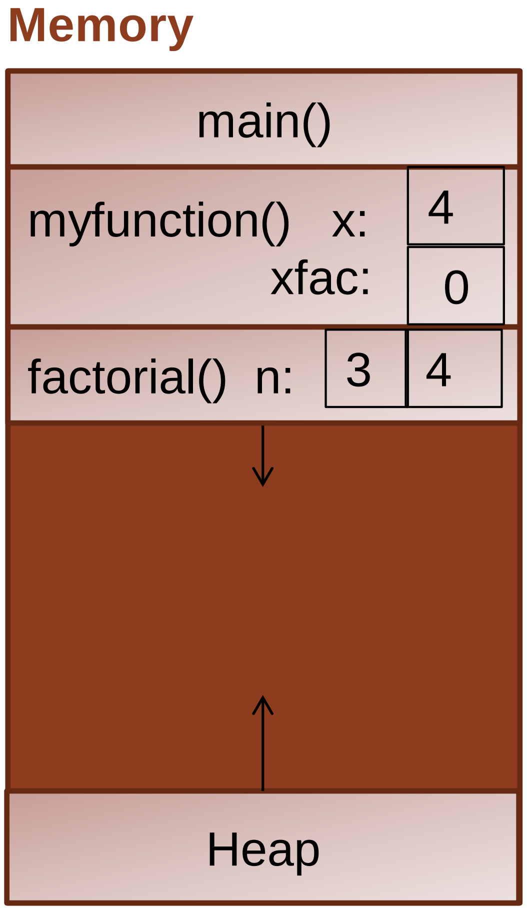
(A)




(B)




(C)



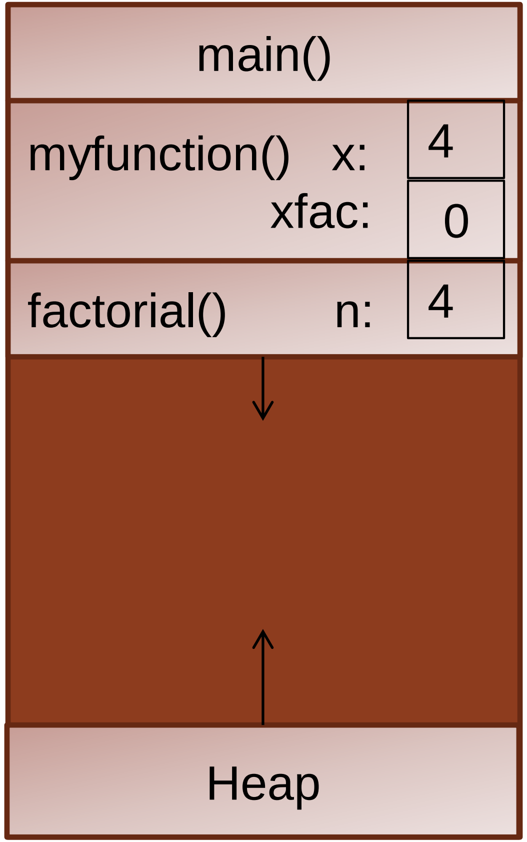

(D) Other/none of the above

# The “stack” part of memory is a stack

Function call = push

Return = pop

# The "stack" part of memory is a stack

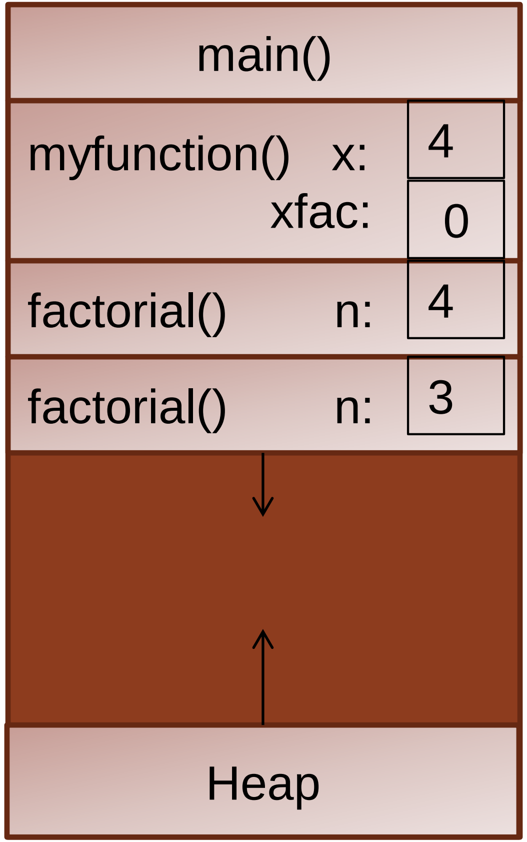



## Recursive code

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

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

# The "stack" part of memory is a stack

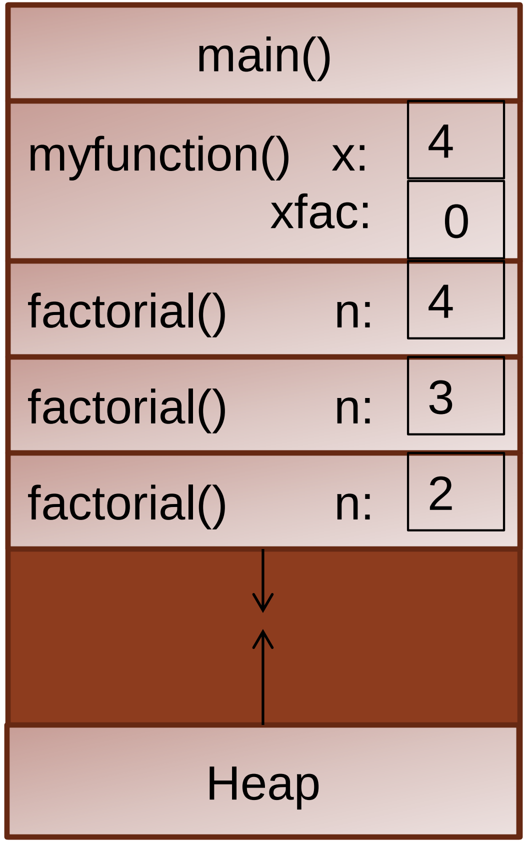



**Recursive code**

```
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

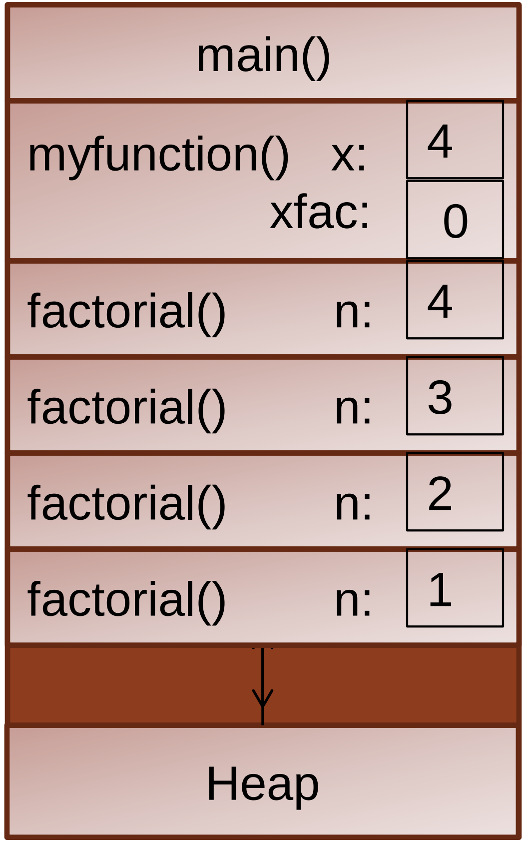



**Recursive code**

```
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);
}
```

# Factorial!

What is the **fourth** value ever **returned** when we call factorial(4)?

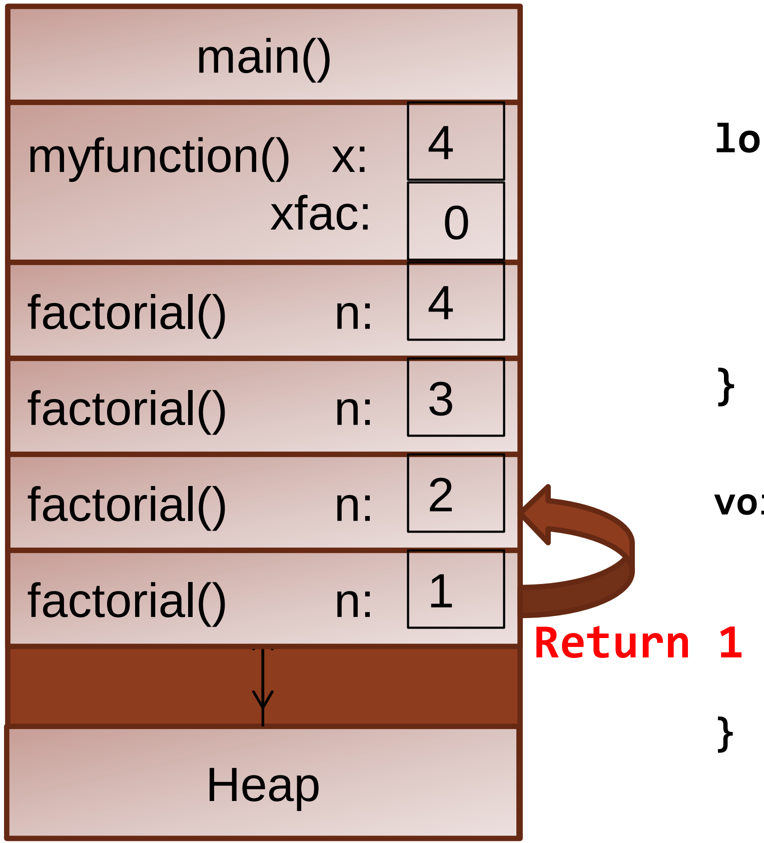
A. 4
B. 6
C. 10
D. 24
E. Other/none/more than one

## 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

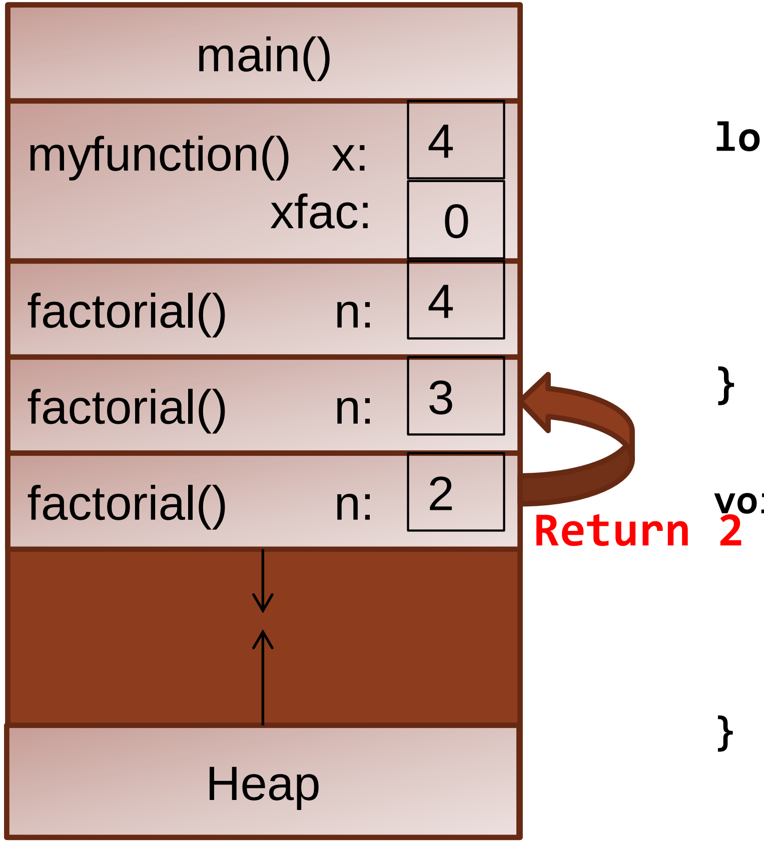



**Recursive code**

```
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

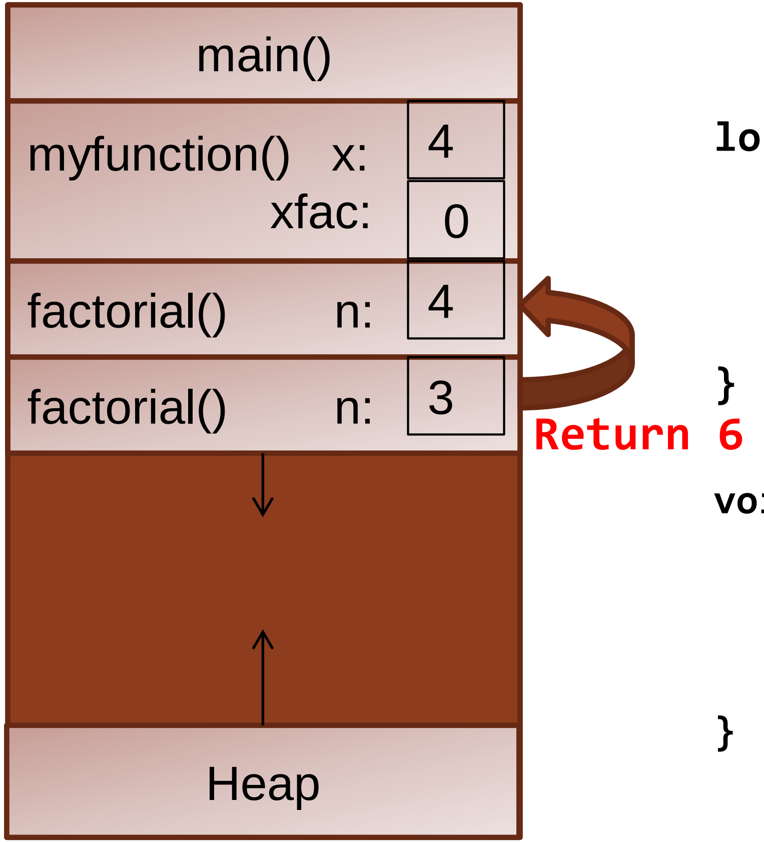



## Recursive code

```
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);
}
```

# The “stack” part of memory is a stack

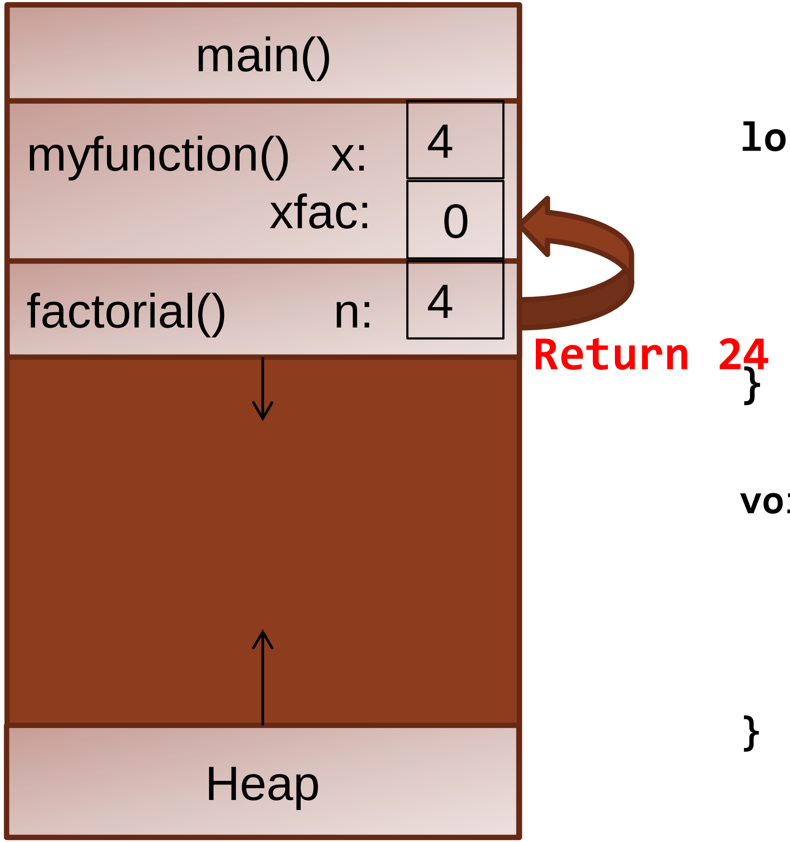



### Recursive code

```
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);
}
```

# 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) {
    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. Method calls have overhead in terms of space *and* time to set up and tear down.