Programming Abstractions

CS106B

Cynthia Lee

Today's Topics

ADTs

- Containers within containers
  - Example: anagram finder

Recursion

- First steps in this fun/crazy new concept
  - Factorial!

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);
```

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

C++ bonus details:

This works by returning a reference (!)

C++ allows you to define a return type to be a reference

In the case of map, this returns a *reference* to the value at map[key]:

```
ValueType & operator[](const KeyType & key);
```

Stanford library Map *(selected member functions)*

```
template <typename KeyType, typename ValueType> class Map {
public:
    void add(const KeyType& key, const ValueType& value);

    bool containsKey(const KeyType& key) const;

    ValueType get(const KeyType& key) const;

    ValueType operator [](const KeyType& key) const;

    ValueType& operator [](const KeyType& key);
...
private:

}
```

Redacted…
until the second half of the quarter!

Anagram Finder

An application of compound Map

“Abstractions”

Bacon artists
Cab stain rots
Crab in toasts
Bonsai tracts
…

http://www.wordsmith.org/anagram/

What would be a good design for this problem?

Concept:

- Unlike the website, we will only show anagrams that are 1 word ↔ 1 word ("moored" ↔ "roomed", not "abstractions" ↔ "bacon artists")
- Have a string that is a "representative" of a group of words that are anagrams of each other
- Have that string map to a list of those words
- `Map<string, Vector<string>> anagrams;`

- Key trick idea: the representative is the string with the letters sorted (use a function: `string sortWord(string word)`)
  - *moored* becomes *demoor*
  - *roomed* becomes *demoor*

What would be a good design for this problem?

Concept:

- `Map<string, Vector<string>> anagrams;`

How would we add a word stored in the string variable `word` to our collection?

A. `anagrams[word] += word;`

B. `anagrams[word] += sortWord(word);`

C. `anagrams[sortWord(word)] += word;`

D. `anagrams[sortWord(word)] += sortWord(word);`

E. Other/none/more

What would be a good design for this problem?

Concept:

- `Map<string, Vector<string>> anagrams;`

To add a word to our collection:

```
anagrams[sortWord(word)] += word;
```

To look up a word in our collection to find its anagrams:

```
Vector<string> matches = anagrams[sortWord(query)];
```

Recursion


Which image resul
following code s
>> im = imread('rainbow.jpg');
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)

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 examples

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

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

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 {
       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) {
        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 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

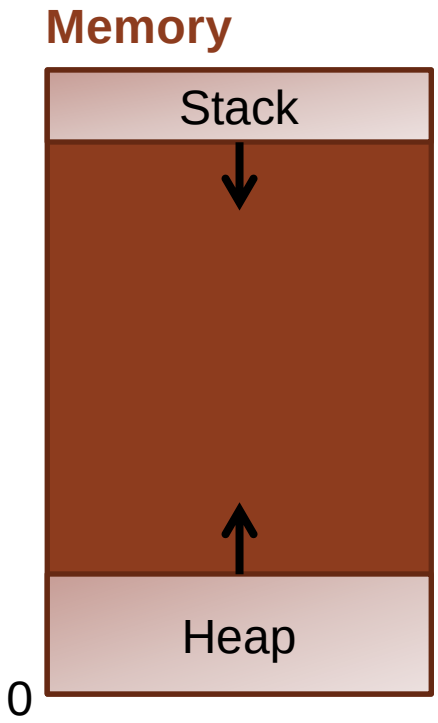
Factorial!

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

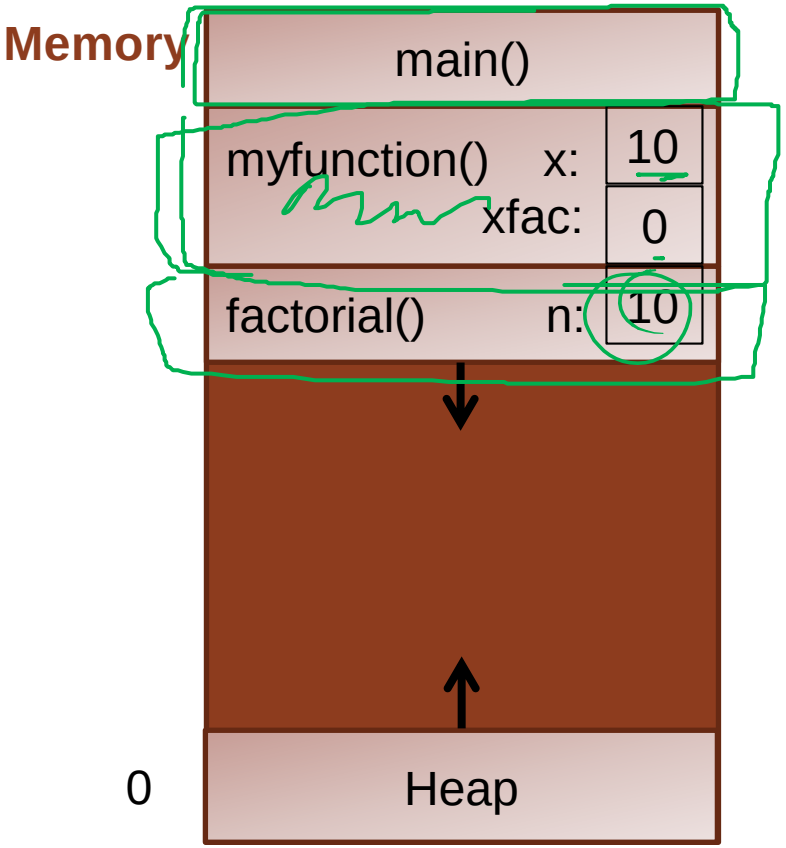
What is the **third** thing **printed** when we call factorial(10)?

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

How does this look in memory?



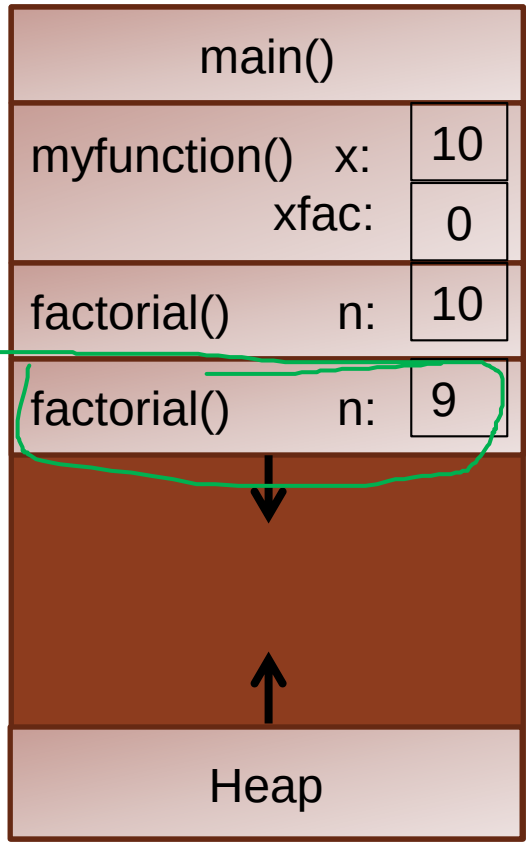
How does this look in memory?



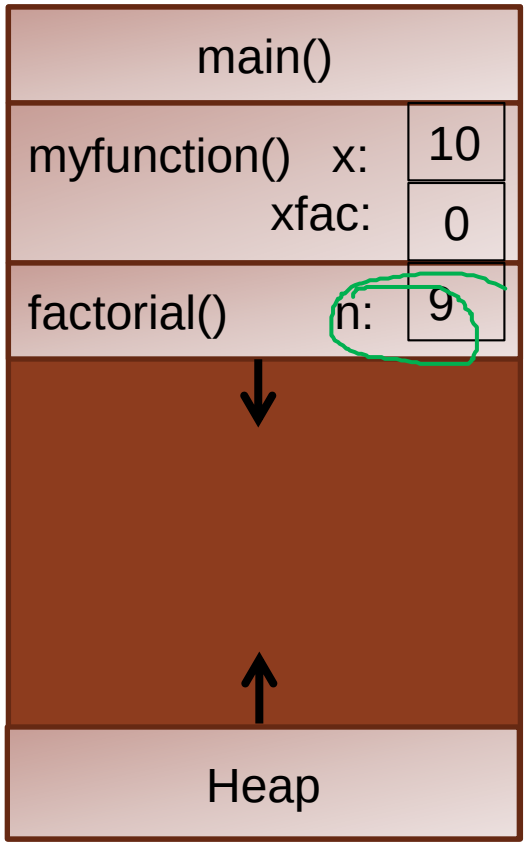

```
long factorial(int n) {
    cout << n << endl;
    if (n == 1) {
       return 1;
    } else {
       return n * factorial(n - 1);
    }
}

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

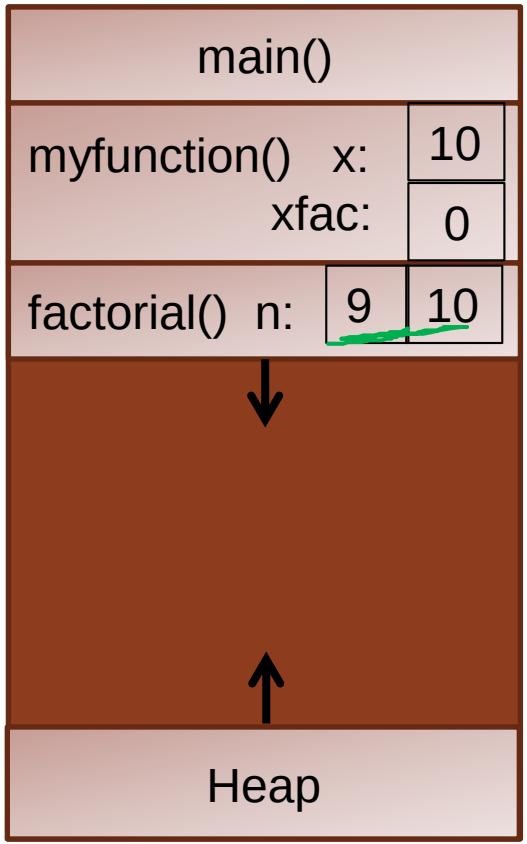
(A)



(B)



(C)



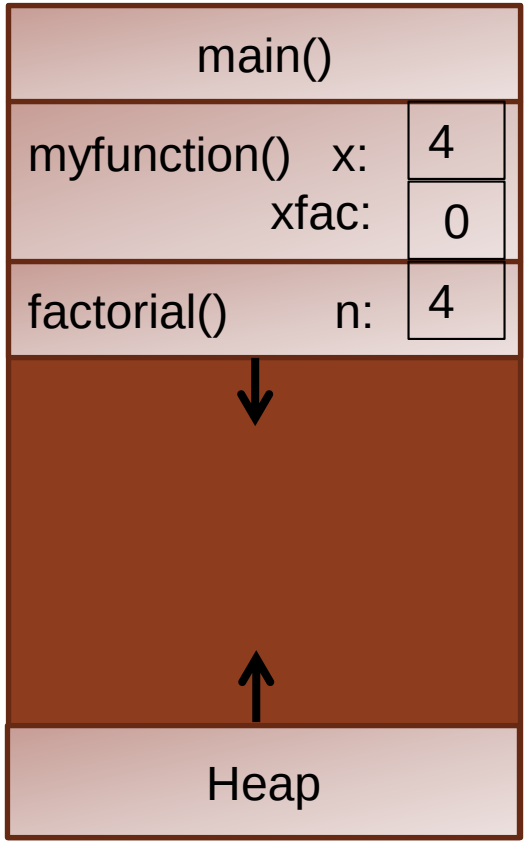
(D) Other/none of the above

The “stack” part of memory is a stack

Function call = push

Return = pop

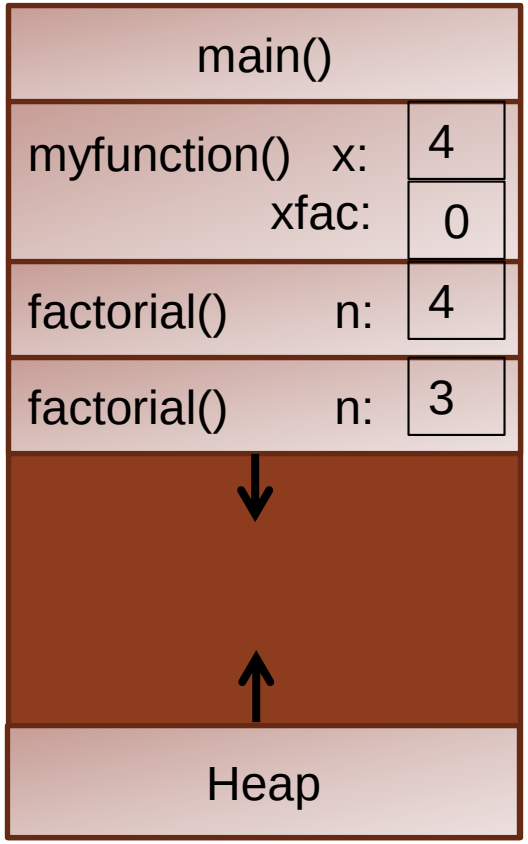
The “stack” part of memory is a stack




```
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);
}
```

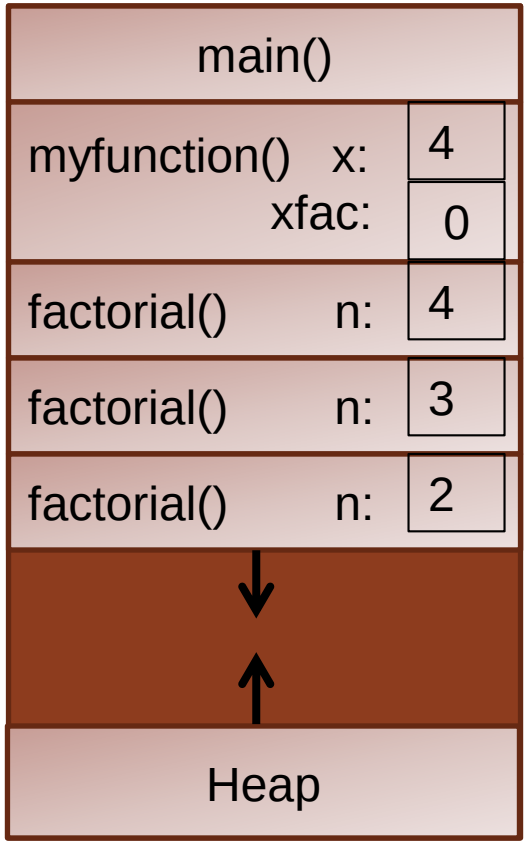
The “stack” part of memory is a stack




```
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




```
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


```
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);
}
```

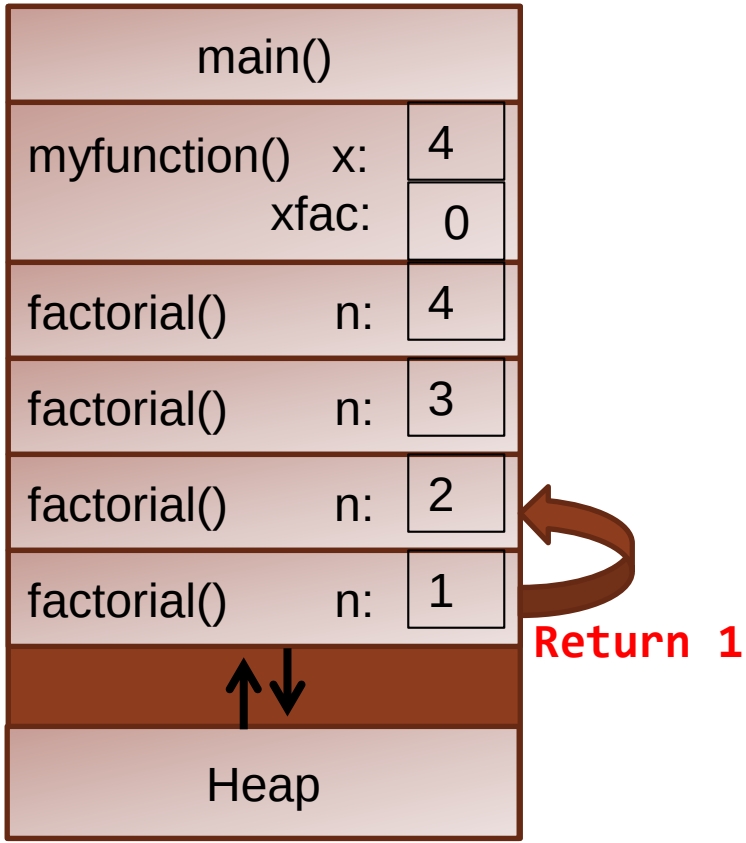
Factorial!

```
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

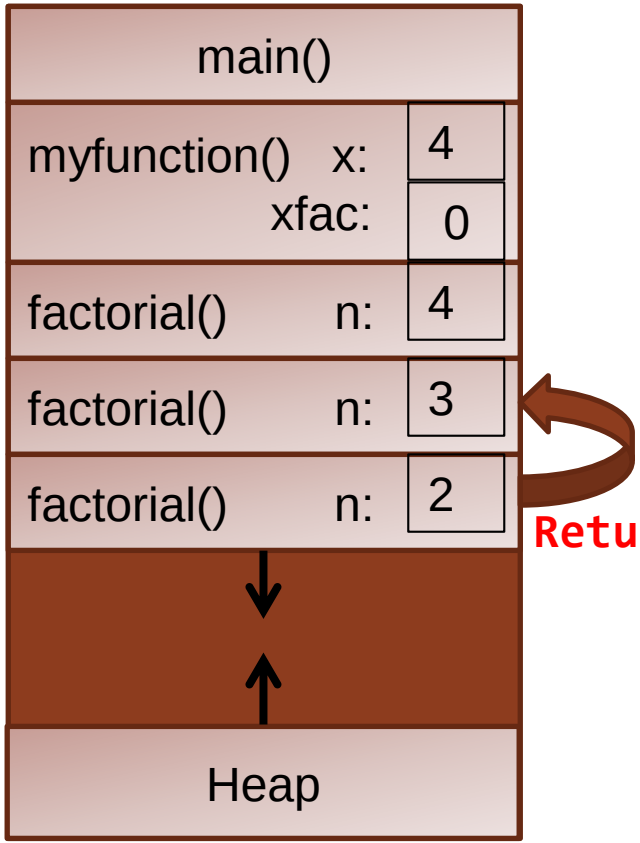
The “stack” part of memory is a stack




```
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

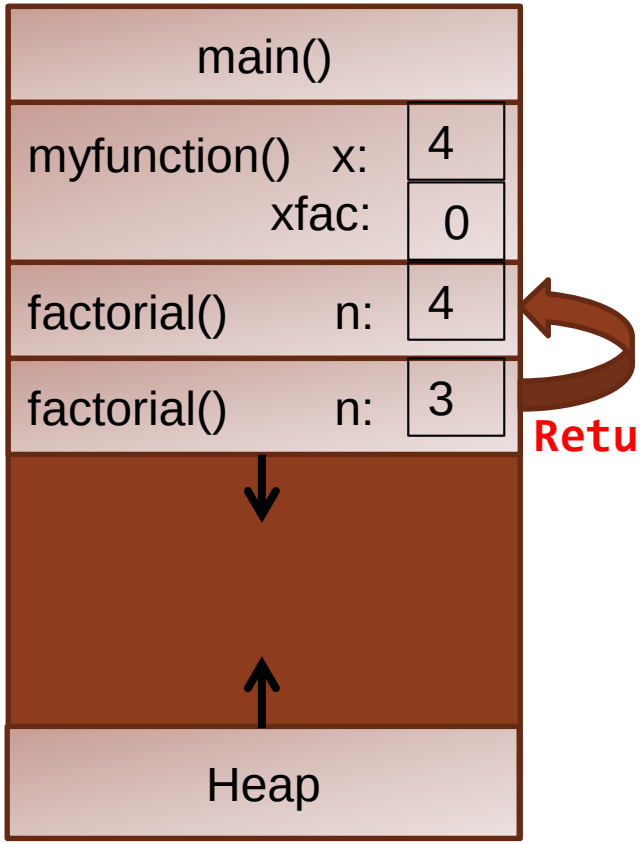


Return 2

```
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



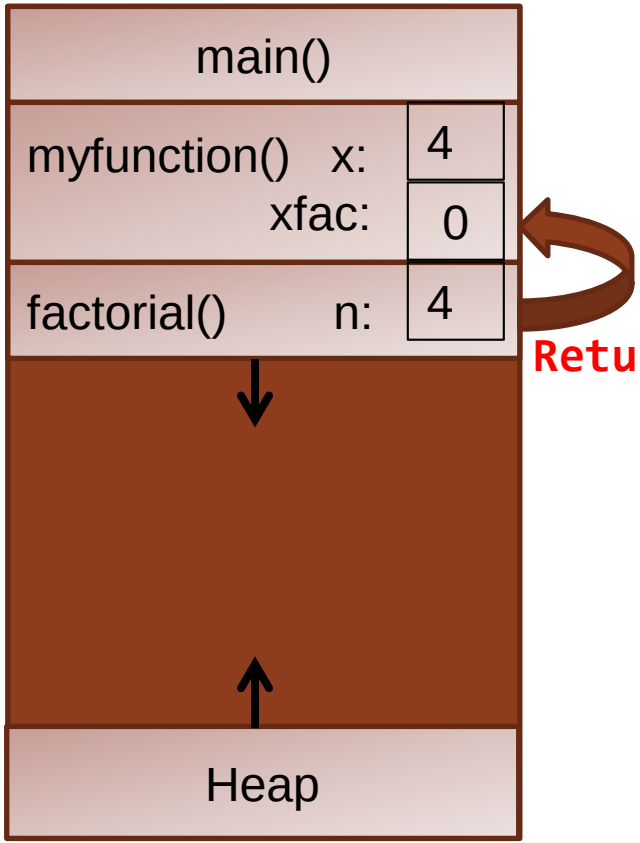

```
long factorial(int n) {
    cout << n << endl;
    if (n == 1) {
       return 1;
    } else {
       return n * factorial(n - 1);
    }
}
```

3 2

Return 6

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

The “stack” part of memory is a stack



Return 24

```
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.