Recursion

Instructor: Jeeho Ryoo

Announcements

- Feb 21
    - No lecture
- Assignment 1
    - Grade will be posted this week
    - No email
    - Any discussion with assignment 1 grade in person during the lecture time in the lecture hall on Feb 21
- No lab next week
    - All submission to learning hub during designated lab time
    - Lab6 released in the week of Mar 4
    - No lab in the week of Mar 4

Announcements

- Quiz 2
  - Grades will be released this weekend
  - Any discussion to be hold on Feb 21 (same as assignment 1)
- Accessibility services
  - Please submit your request
- Midterm (Feb 27 @ 8:30AM SW09 110)
  - Closed book
  - No cheat sheet
  - MCQ
  - Output
  - Coding
  - Coverage including this week's lecture
  - 90 min

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);
    }
}
```

Digging deeper in the recursion

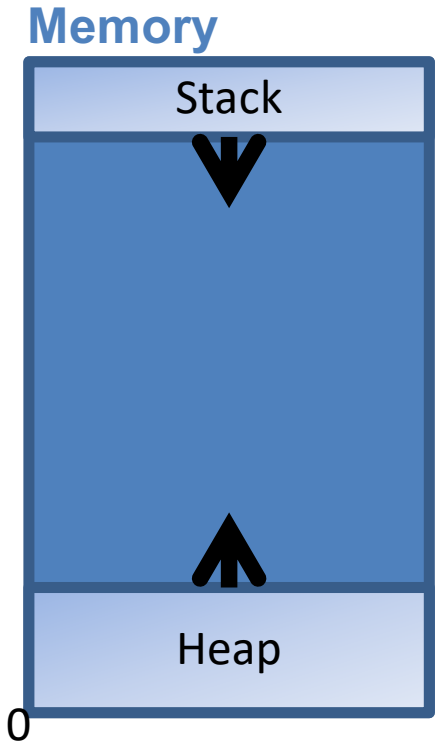
Factorial!

```
long factorial(int n) {
    printf("%d\n", n); //added code
    if (n == 1) {
       return 1;
    } else {
       return n * factorial(n - 1);
    }
}
```

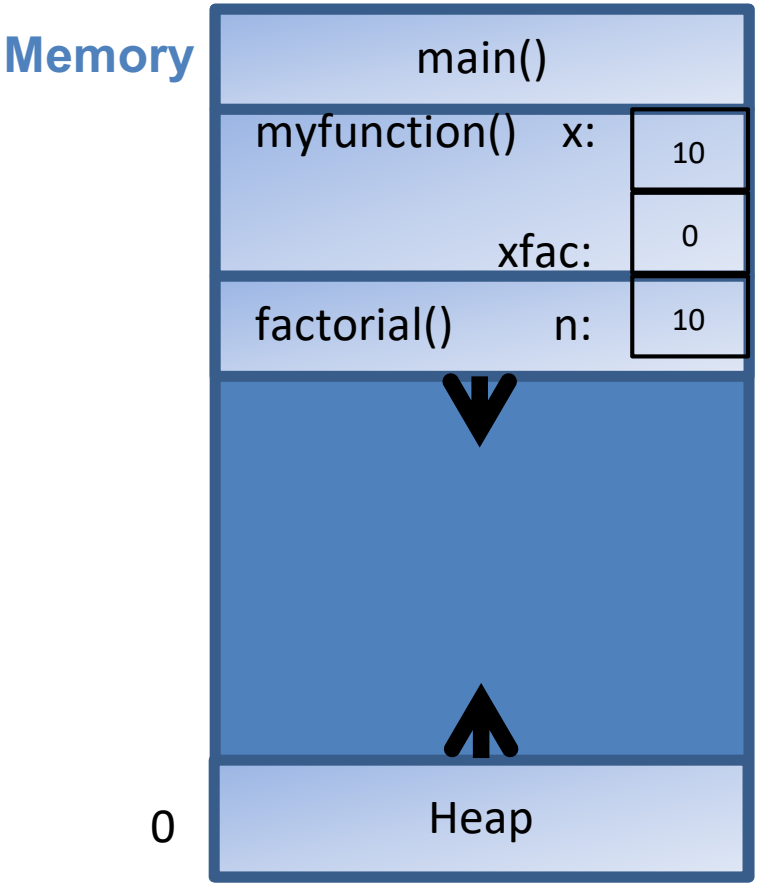
What is the **third** thing **printed** when we call factorial(10)?

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

How does this look in memory?



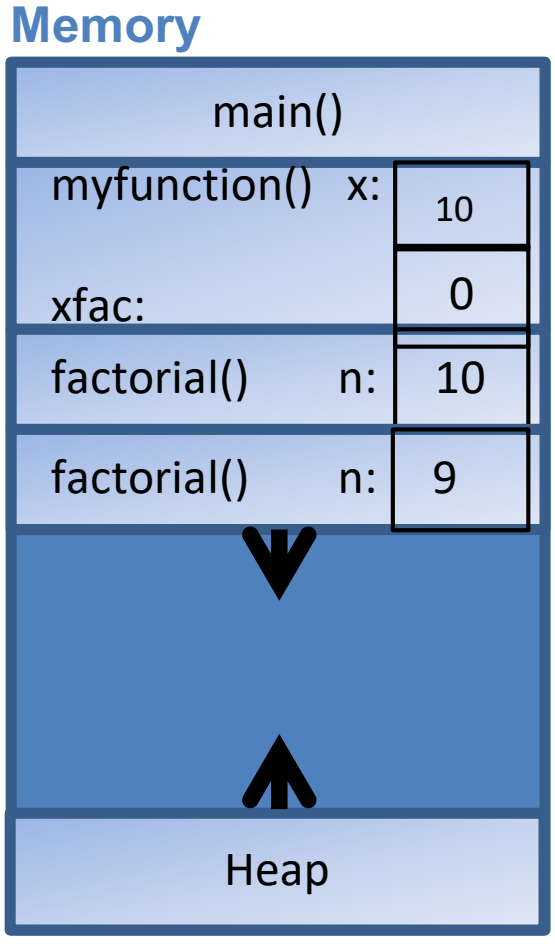
How does this look in memory?



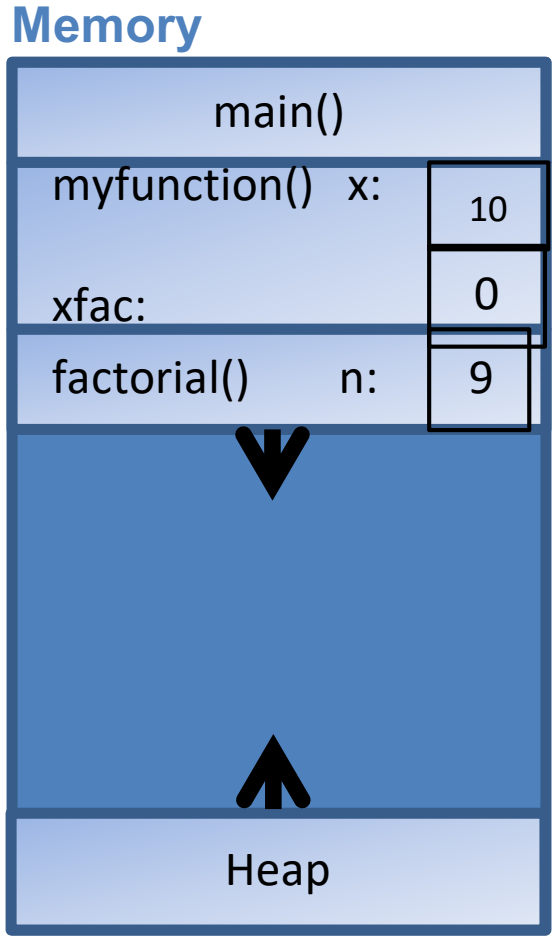

```
long factorial(int n) {
    printf("%d\n", n); //added code
    if (n == 1) {
       return 1;
    } else {
       return n * factorial(n - 1);
    }
}

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

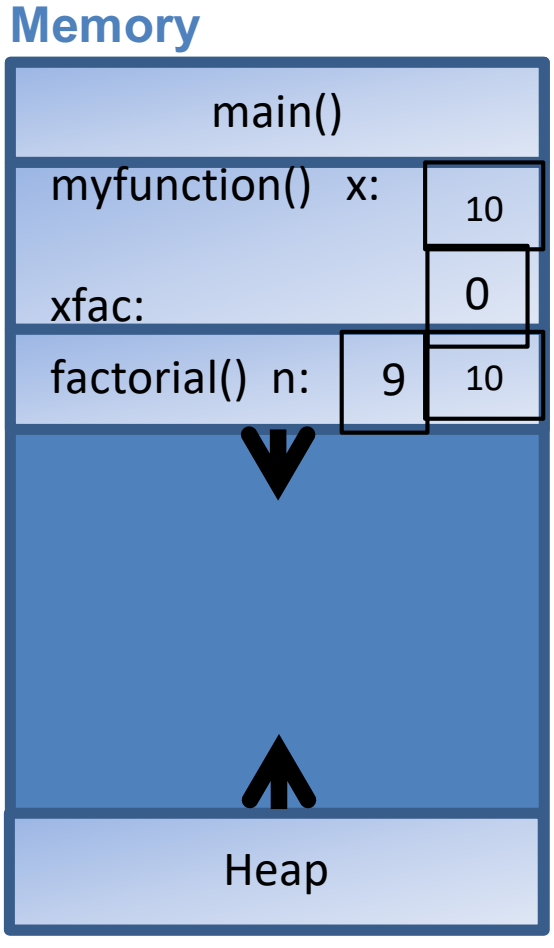
(A)



(B)



(C)

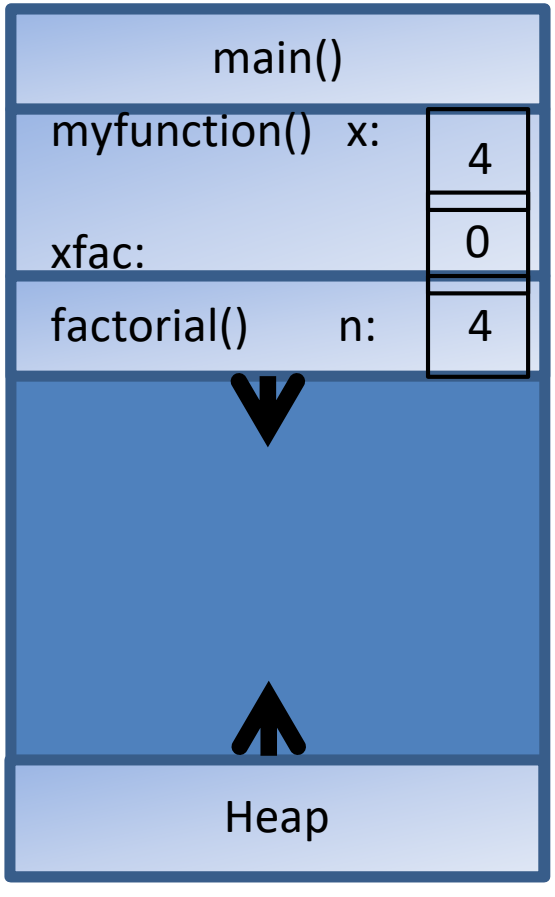


(D) Other/none of the above

The “stack” part of memory is a stack

Function call = push
Return = pop

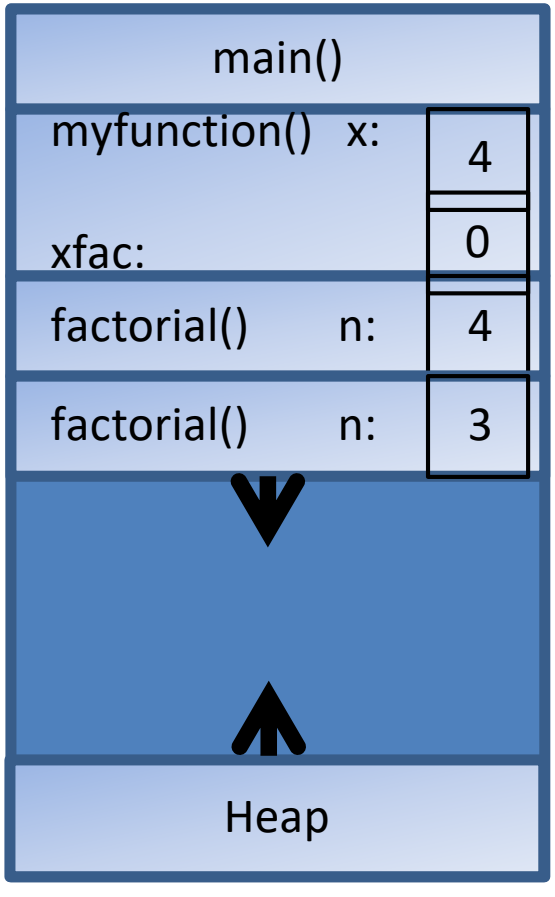
The “stack” part of memory is a stack




```
long factorial(int n) {
    printf(“%d\n”, n); //added code
    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);
}
```

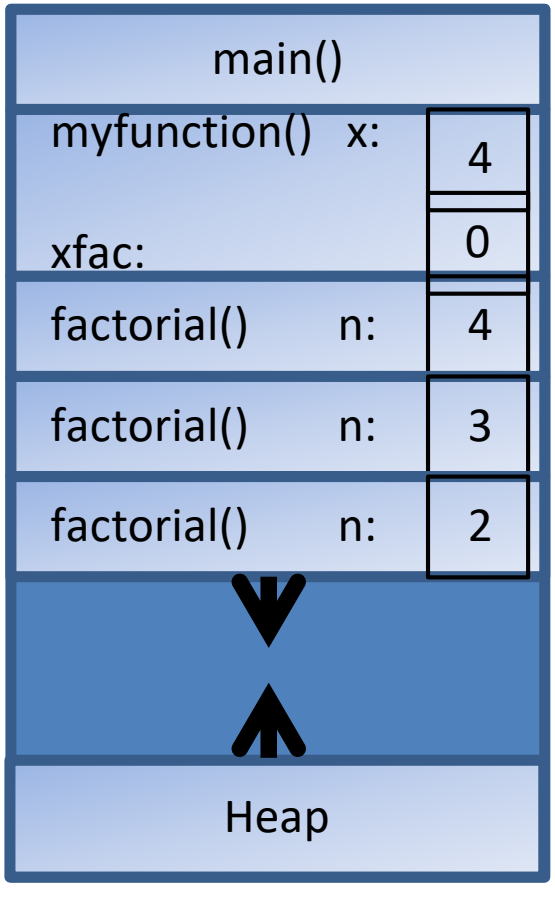
The “stack” part of memory is a stack




```
long factorial(int n) {
    printf(“%d\n”, n); //added code
    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);
}
```

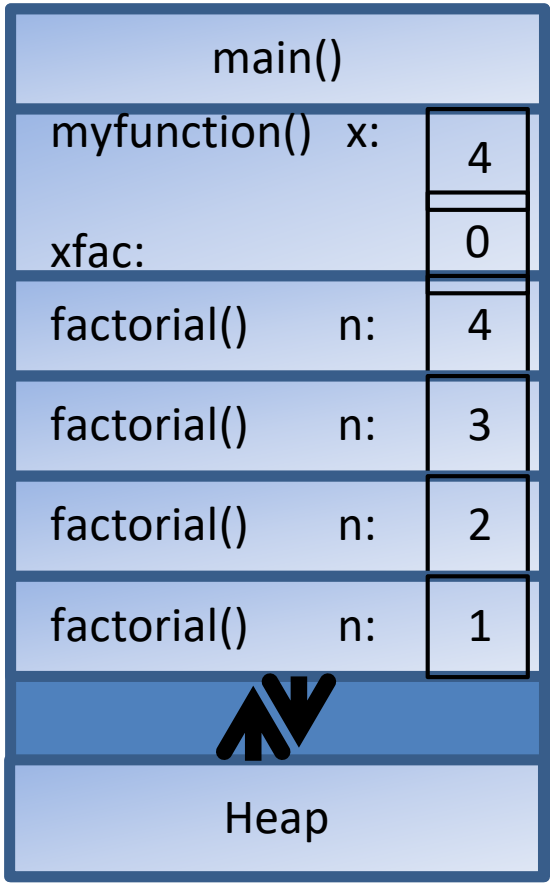
The “stack” part of memory is a stack




```
long factorial(int n) {
    printf(“%d\n”, n); //added code
    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) {
    printf(“%d\n”, n); //added code
    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);
}
```

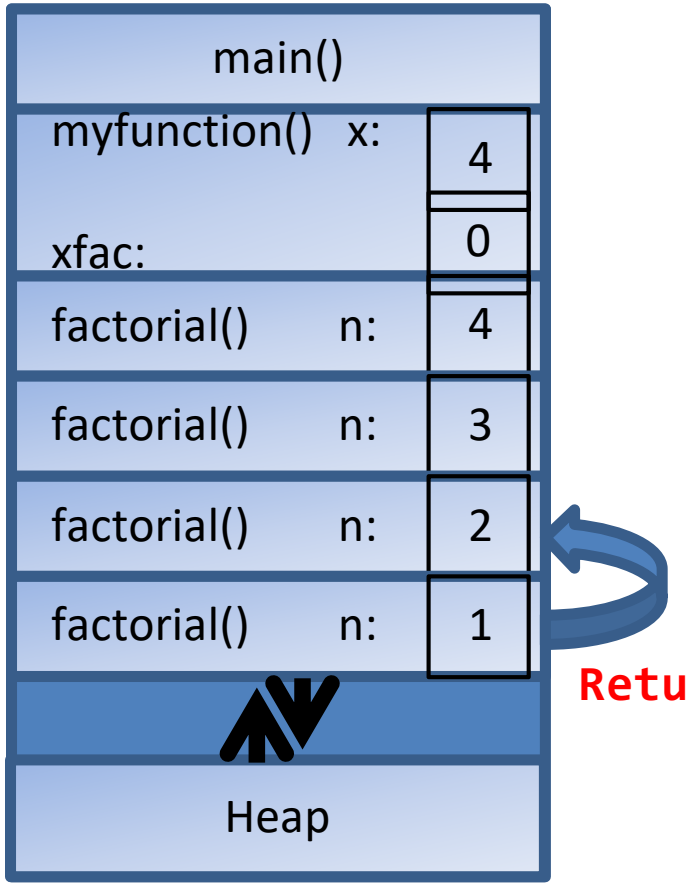
Factorial!

```
long factorial(int n) {
     printf("%d\n", n);
     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

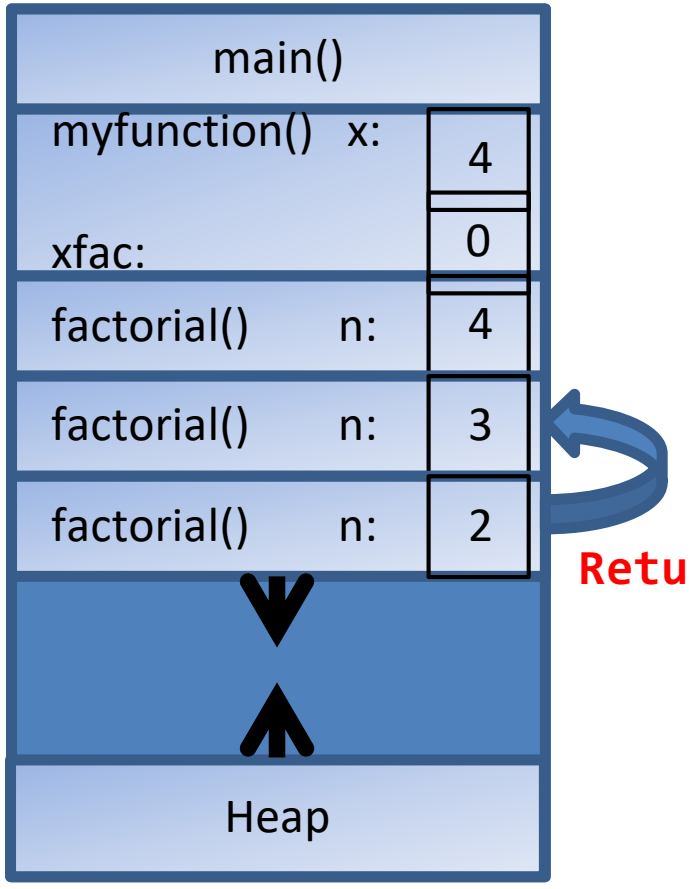



```
long factorial(int n) {
    printf("%d\n", n);
    if (n == 1) {
       return 1;
    } else {
       return n * factorial(n - 1);
    }
}

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

Return 1

The “stack” part of memory is a stack

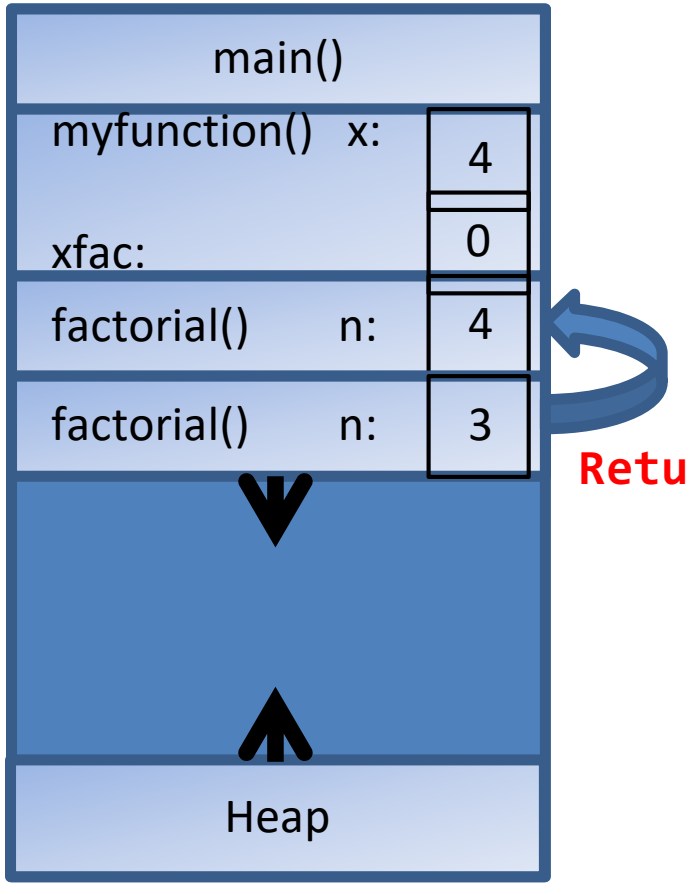


Return 2

```
long factorial(int n) {
    printf(“%d\n”, n);
    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

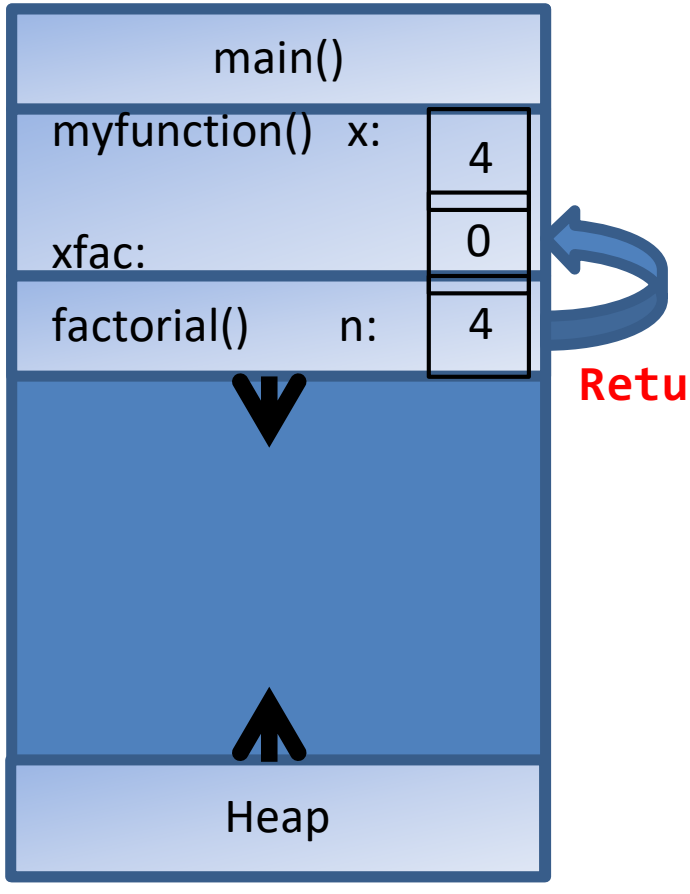


Return 6

```
long factorial(int n) {
    printf(“%d\n”, n);
    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 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!

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. Function calls have overhead in terms of space *and* time to set up and tear down.

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. Function calls have overhead in terms of space *and* time to set up and tear down.

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 case 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 case 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 case 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

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?

Binary Search

```
int binarySearch(int arr[], int l, int
r, int x)
{

}
```

Binary Search

```
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l) {
        int mid = l + (r - l) / 2;

        if (arr[mid] == x)
            return mid;

        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);

        return binarySearch(arr, mid + 1, r, x);
    }

    return -1;
}
```