Introduction to Recursion

Amrita Kaur

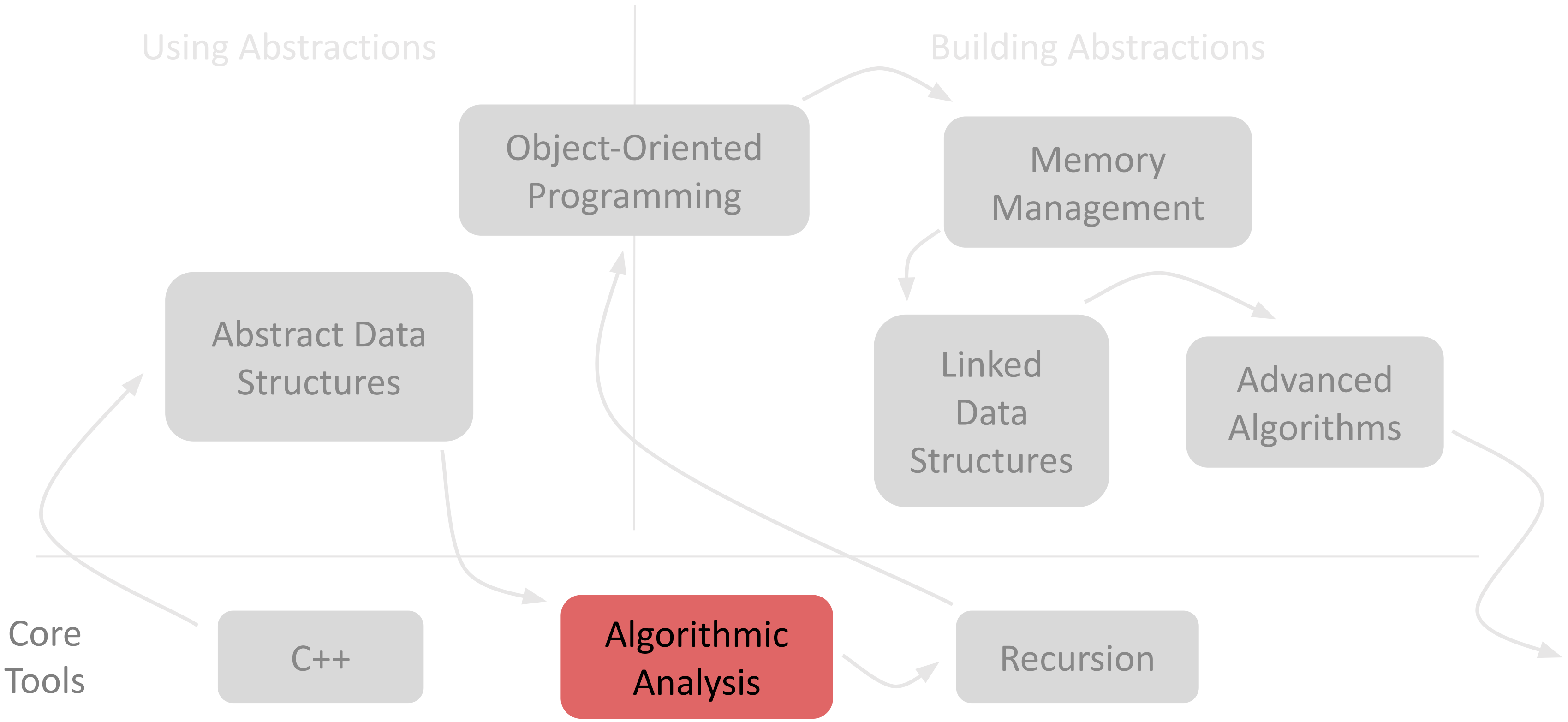
July 10, 2023

Contributions made from previous CS106B Instructors

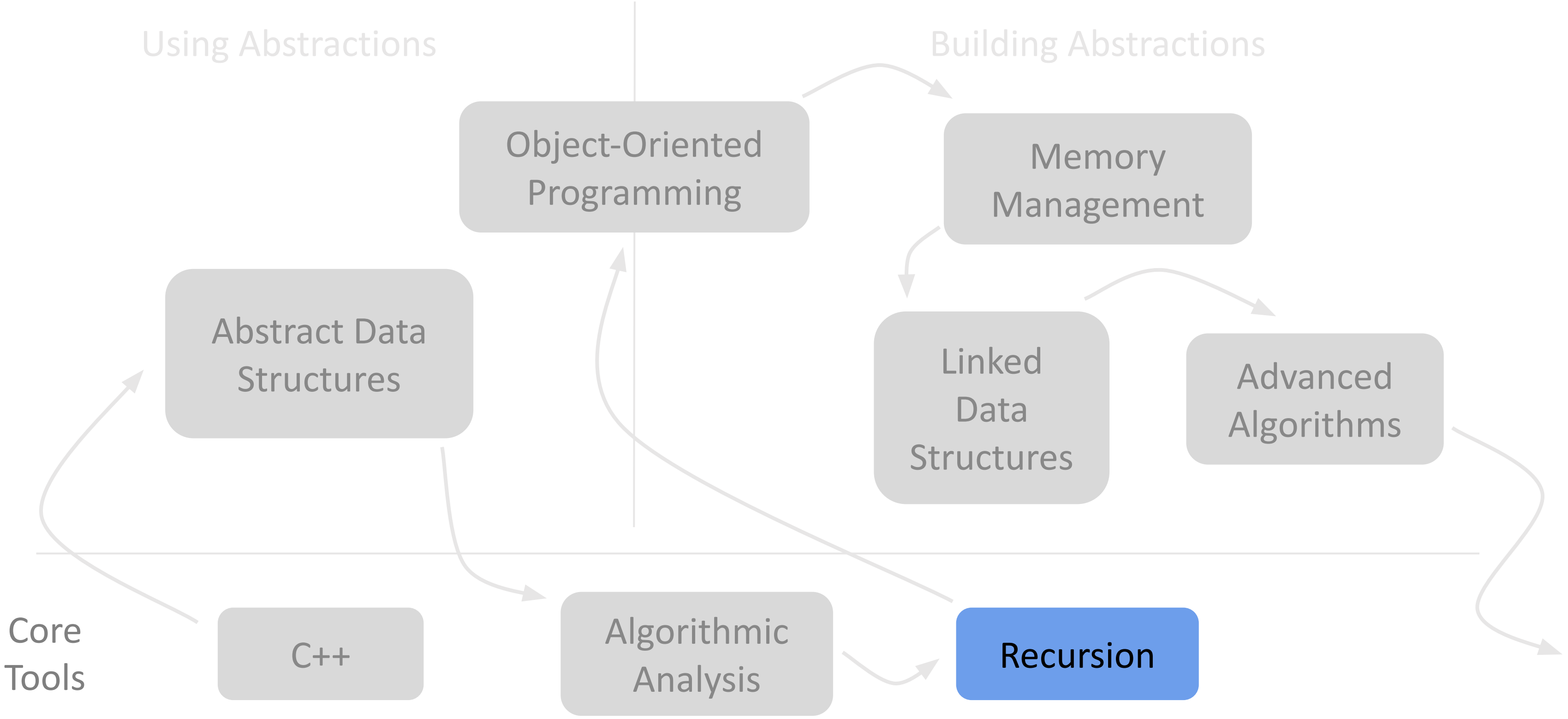
Announcements and Reminders

- Assignment 2 due Friday at 11:59pm
- IGs with your SL on Assignment 1 this week
- Midterm next Monday from 7-9pm
  - Talk more about this at the end of today’s class!

Roadmap


Using Abstractions
Building Abstractions
Object-Oriented Programming
Memory Management
Abstract Data Structures
Linked Data Structures
Advanced Algorithms
Core Tools
C++
Algorithmic Analysis
Recursion

Roadmap


Using Abstractions
Building Abstractions
Object-Oriented Programming
Memory Management
Abstract Data Structures
Linked Data Structures
Advanced Algorithms
Core Tools
C++
Algorithmic Analysis
Recursion

Jumble - July 10, 2023

TEYPT

□ □ ○ ○ □

TOTOH

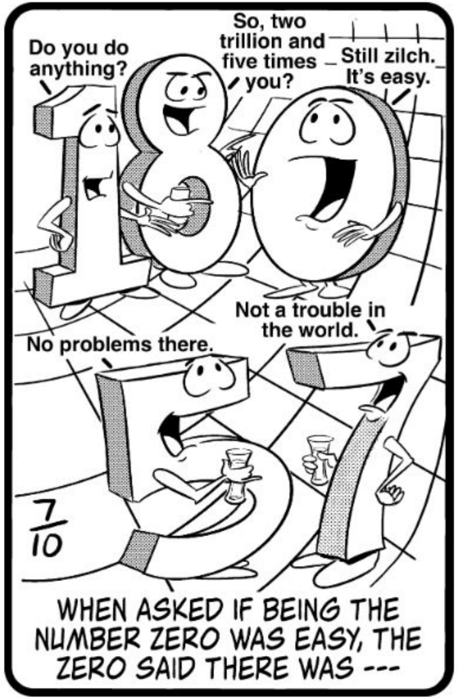
○ □ ○ □ ○

NINWOM

□ ○ ○ ○ □ □

LAIEGO

○ ○ □ □ ○ □



WHEN ASKED IF BEING THE NUMBER ZERO WAS EASY, THE ZERO SAID THERE WAS ---

○ ○ ○ ○ ○ ○ ○

○ ○ ○ ○

Code it up

```
void permute4(string s) {
      for (int i = 0; i < 4; i++) {
            for (int j = 0; j < 4 ; j++) {
                  if (j == i) {
                        continue; // ignore
                  }
                  for (int k = 0; k < 4; k++) {
                        if (k == j or k == i) {
                              continue; // ignore
                        }
                        for (int w = 0; w < 4; w++) {
                              if (w == k or w == j or w == i) {
                                    continue; // ignore
                              }
                              cout << s[i] << s[j] << s[k] << s[w] << endl;
                        }
                  }
            }
      }
}
```

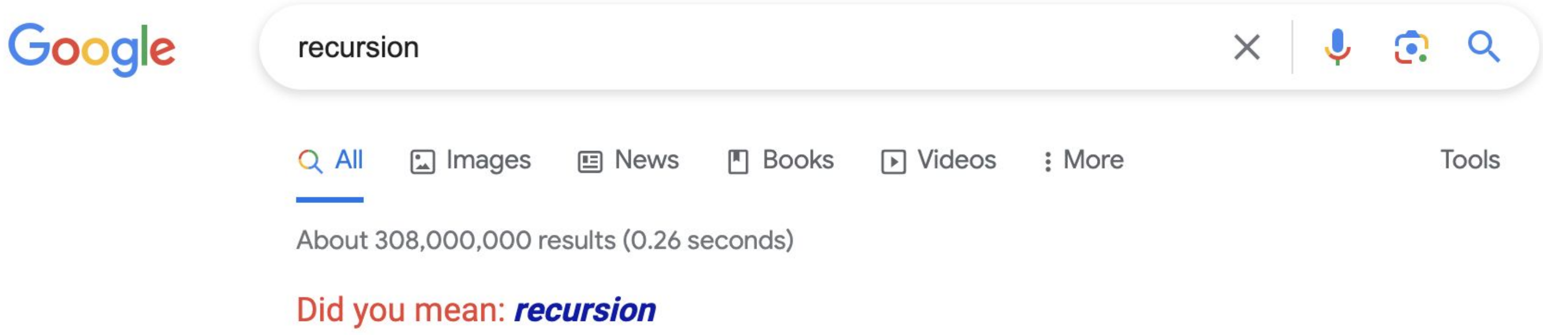
Code it up

```
void permute5(string s) {
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4 ; j++) {
            if (j == i) {
                continue; // ignore
            }
            for (int k = 0; k < 4; k++) {
                if (k == j or k == i) {
                    continue; // ignore
                }
                for (int w = 0; w < 4; w++) {
                    if (w == k or w == j or w == i) {
                        continue; // ignore
                    }
                    for (int x = 0; x < 5; x++) {
                        if (x == k or x == j or x == i or x == w) {
                            continue;
                        }
                        cout << "  " << s[i] << s[j] << s[k] << s[w] << s[x] << endl;
                    }
                }
            }
        }
    }
}
```

Recursion

What is recursion?

Wikipedia: “concept or process depends on a simpler version of itself”

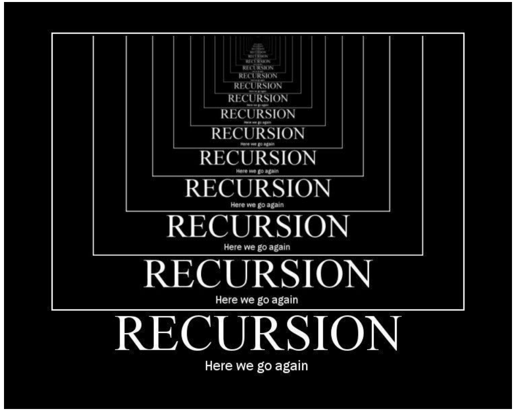


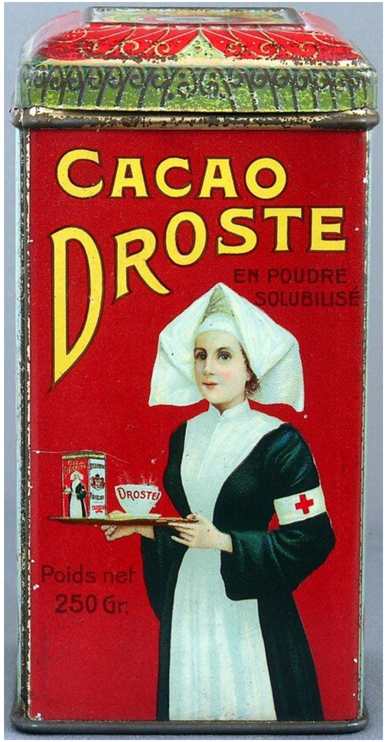
recursion

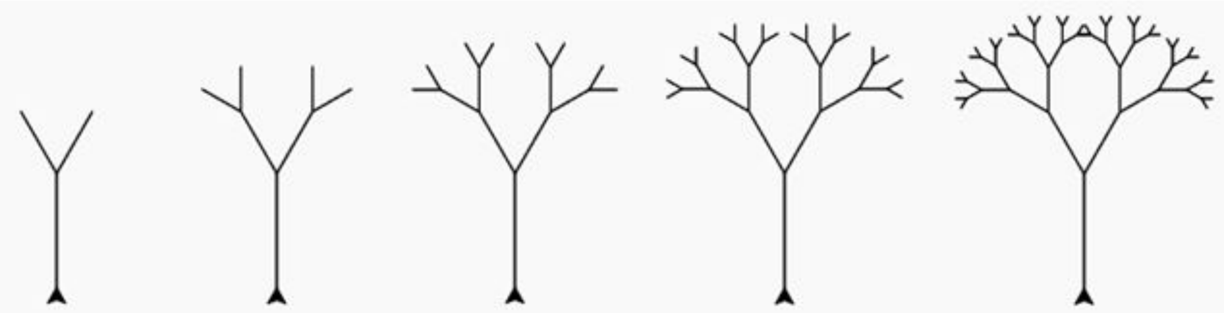
About 308,000,000 results (0.26 seconds)

Did you mean: ***recursion***

What is recursion?



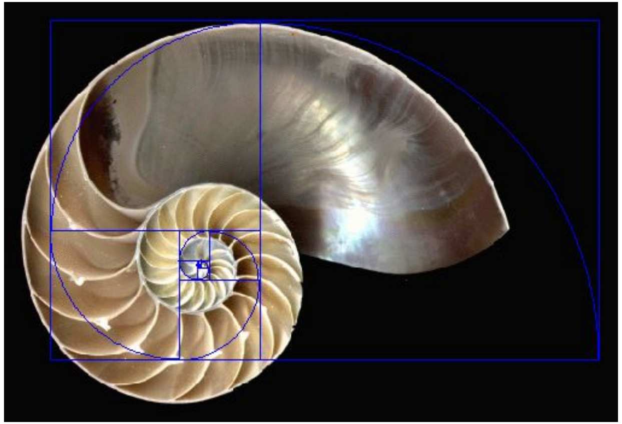


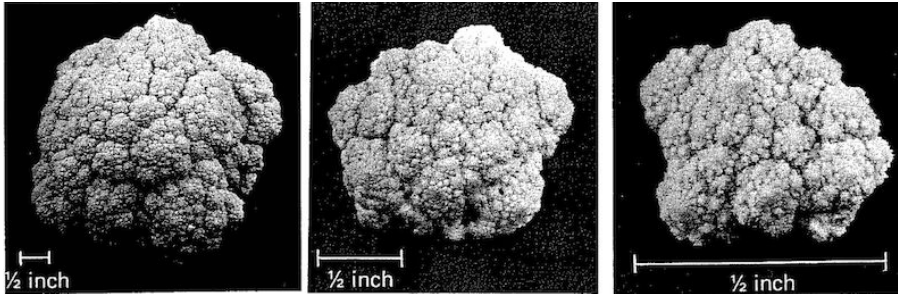


What is recursion?

- A problem-solving technique in which tasks are completed by reducing them into **repeated, smaller tasks of the same form**.
- Powerful substitution for iteration (loops)
    - Start by seeing the difference between iterative vs. recursive solutions
    - Later will see problems that can only be solved by recursion
- Results in elegant, often shorter code
- Can be used to express patterns seen in nature

Recursion in nature





Using recursion in real life

Solve puzzle:

1. Is the puzzle finished? If yes, stop.
2. Find one correct piece and place it
3. Solve the rest of the puzzle

Using recursion in real life

- I want to figure out how many students came to class today
- I want to recruit your help, but I also want to minimize each individual's amount of work

Counting students

- Focus on counting a single row first
    - I ask the person on the very left “How many people are to your right?”
    - Student’s algorithm:
        - If there is no one to your right, answer 0.
        - If someone is sitting to your right
            - Ask that person, “How many people are to your right?”
            - When they respond with a value N, respond (N+1) to the person who asked you
- Can generalize to the entire lecture hall

Counting students

- Focus on counting a single row first
  - I ask the person on the very left “How many people are to your right?”
  - S
    - ?”
    - When they respond with a value N, respond (N + 1) to the person who asked you
- Can generalize to the entire lecture hall

recursion

problem-solving technique in which tasks are completed by reducing them into repeated, smaller tasks of the same form

What is recursion?

- In programming, it means that the function calls itself
- Every time the function is called, the problem becomes a little smaller

```
void recurse() {
    recurse();
}
```

*never ever write code like this

Two main components

- Base case
  - The simplest version of your problem that all other cases reduce to
  - An occurrence that can be answered directly

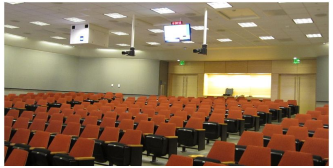
Is the puzzle finished? If yes, stop.

If there is no one to your right, answer 0.

Two main components

- Base case
  - The simplest version of your problem that all other cases reduce to
  - An occurrence that can be answered directly
- Recursive case
  - More complex version of the problem that cannot be directly answered
  - Break down the task into smaller occurrences
  - Take the "recursive leap of faith" and trust the smaller tasks will solve the problem for you!

Place one piece and solve rest of puzzle



If someone is sitting to your right…

Two main components

- Base case
  - The simplest version of your problem that all other cases reduce to
  - An occurrence that can be answered directly
- Recursive case
  - More complex version of the problem that cannot be directly answered
  - Break down the task into smaller occurrences
  - Take the "recursive leap of faith" and trust the smaller tasks will solve the problem for you!

Three “Musts” of Recursion

1. Your code must have a case for all valid inputs.
2. You must have a base case that does not make recursive calls.
3. When you make a recursive call it should be to a simpler instance of the same problem, and make progress towards the base case.

Compute Factorial!

Factorial Example

- The number **`n factorial`**, denoted as **`n!`**, is

  `n × (n-1) × … × 3 × 2 × 1`

- For example,
  - `3! = 3 × 2 × 1 = 6`
  - `4! = 4 × 3 × 2 × 1 = 24`
  - `5! = 5 × 4 × 3 × 2 × 1 = 120`
  - `0! = 1` (by definition)
- Let’s implement a function to compute factorials!

Computing Factorials

$$5! = 5 \times 4 \times 3 \times 2 \times 1$$

Computing Factorials

$$5! = 5 \times \underbrace{4 \times 3 \times 2 \times 1}_{4!}$$

Computing Factorials

5! = 5 × 4!

Computing Factorials

```
5! = 5 × 4!
4! = 4 × 3 × 2 × 1
```

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times \underbrace{3 \times 2 \times 1}_{3!}$$

Computing Factorials

$5! = 5 \times 4!$

$4! = 4 \times 3!$

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2 \times 1$$

Computing Factorials

```
5! = 5 × 4!
4! = 4 × 3!
3! = 3 × 2 × 1
```

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times \underbrace{2 \times 1}_{2!}$$

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

$$2! = 2 \times 1$$

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

$$2! = 2 \times \underbrace{1}_{1!}$$

Computing Factorials

$5! = 5 \times 4!$

$4! = 4 \times 3!$

$3! = 3 \times 2!$

$2! = 2 \times 1!$

Computing Factorials

```
5! = 5 × 4!
4! = 4 × 3!
3! = 3 × 2!
2! = 2 × 1!
1! = 1 × 1
```

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

$$2! = 2 \times 1!$$

$$1! = 1 \times \underbrace{1}_{0!}$$

Computing Factorials

$5! = 5 \times 4!$

$4! = 4 \times 3!$

$3! = 3 \times 2!$

$2! = 2 \times 1!$

$1! = 1 \times 0!$

$0! = 1$

More views of factorials

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

More views of factorials

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

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

Three “Musts” of Recursion

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

1. Your code must have a case for all valid inputs.

Three “Musts” of Recursion

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

1. Your code must have a case for all valid inputs.
2. You must have a base case that does not make recursive calls.

Three “Musts” of Recursion

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

1. Your code must have a case for all valid inputs.
2. You must have a base case that does not make recursive calls.
3. When you make a recursive call it should be to a simpler instance of the same problem, and make progress towards the base case.

Recursion in action

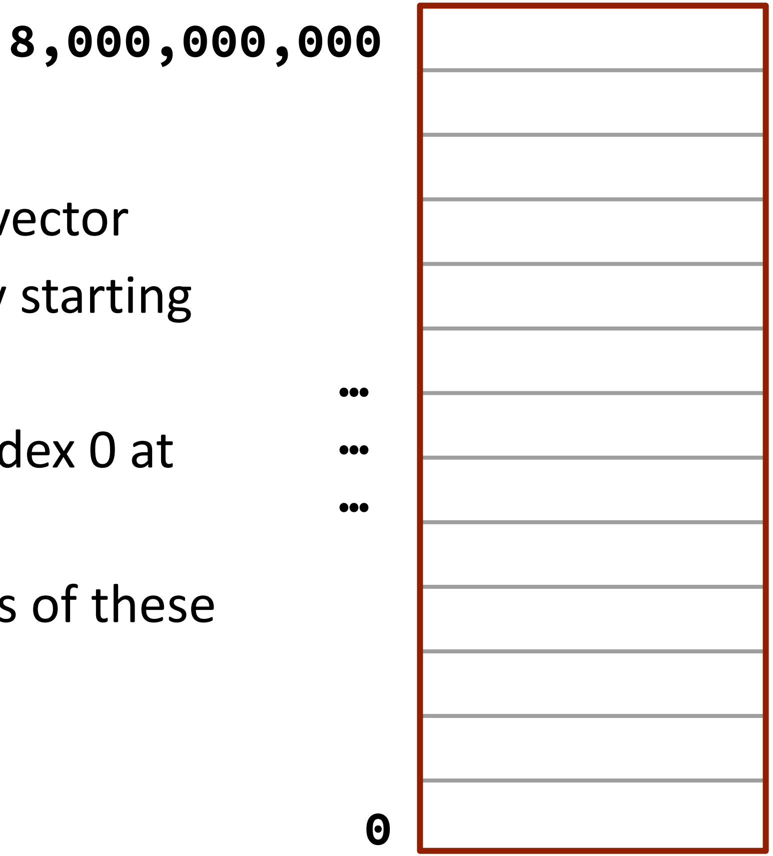
```
int main () {
    int n = factorial(5);
    cout << “5! = ” << n << endl;
    return 0;
}
```

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

Aside on Computer Memory

Computer Memory

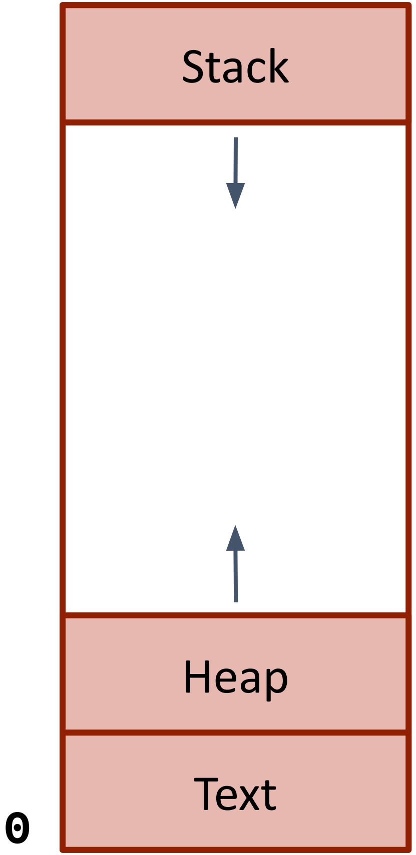
- Computer’s memory is like a giant vector
- Like a vector, we can index memory starting from 0.
- We draw memory vertically with index 0 at the bottom
- Typical laptop’s memory has billions of these indexed slots (one byte each)



Computer Memory

Divide memory in a few main regions

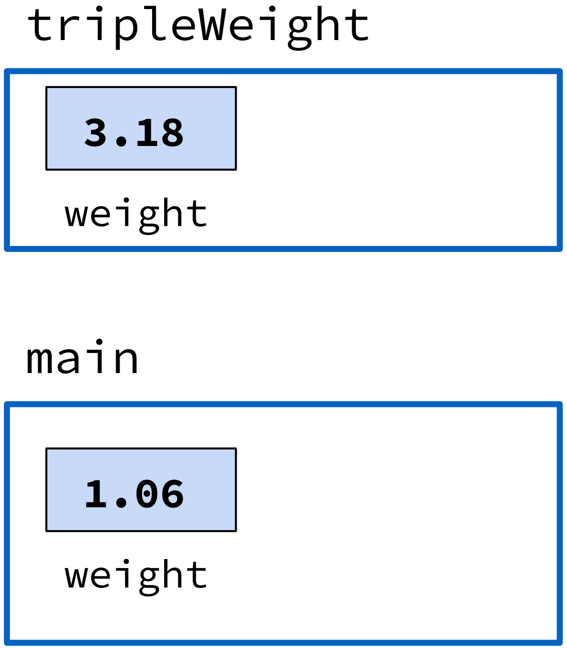
- Text: program’s own code
- Heap: where dynamically allocated memory resides
- Stack: where local variables for each function are stored



Recall this program

```
void tripleWeight(double weight) {
    weight *= 3;
}

int main() {
    double weight = 1.06;
    tripleWeight(weight);
    cout << weight << endl;
}
```



tripleWeight

3.18

weight

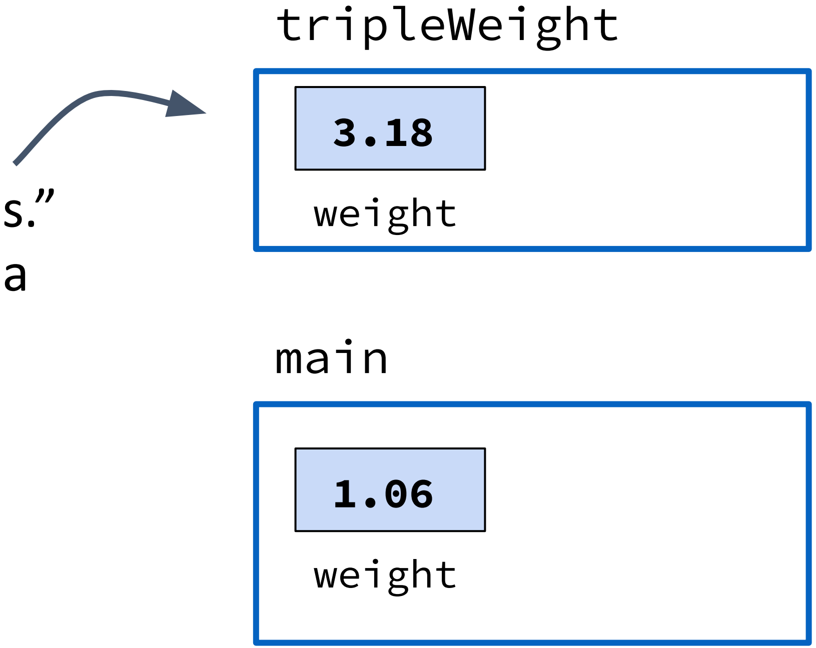
main

1.06

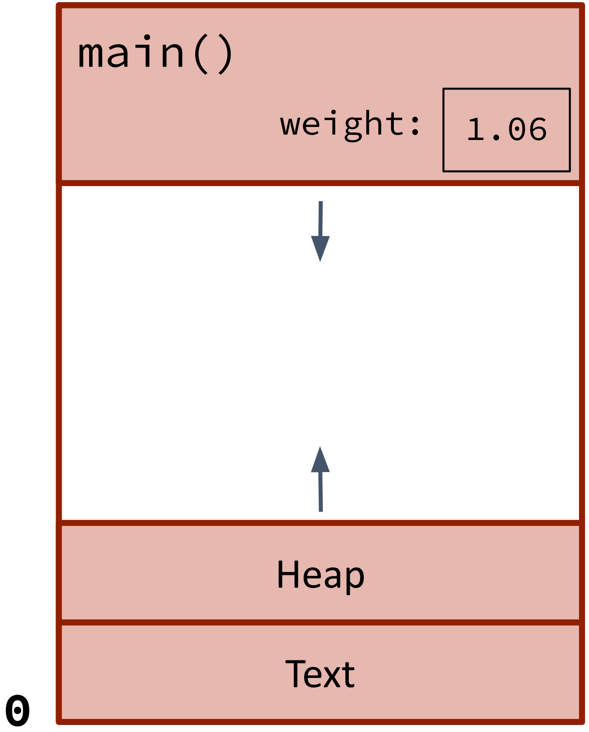
weight

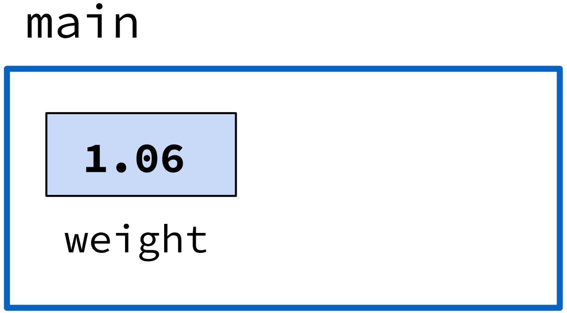
Stack Frames

These are called “stack frames.” One gets created each time a function is called.

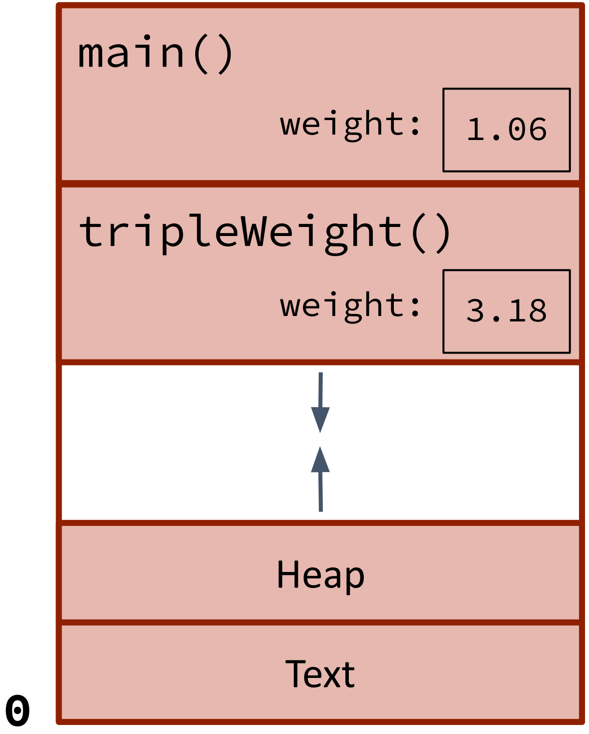


Stack Frames





Stack Frames


main()

weight: 1.06

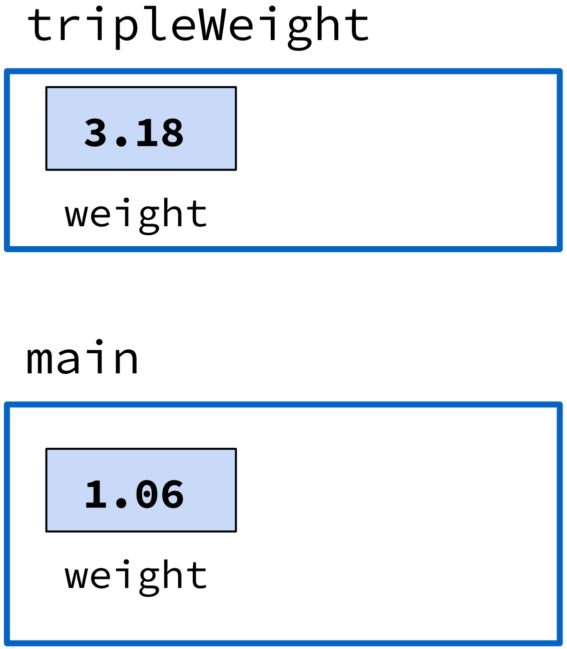
tripleWeight()

weight: 3.18

Heap

Text

0


tripleWeight

3.18

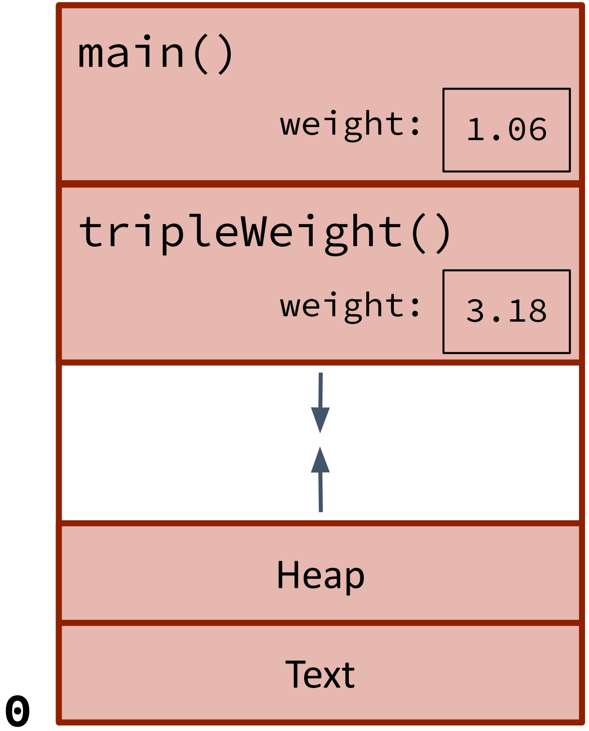
weight

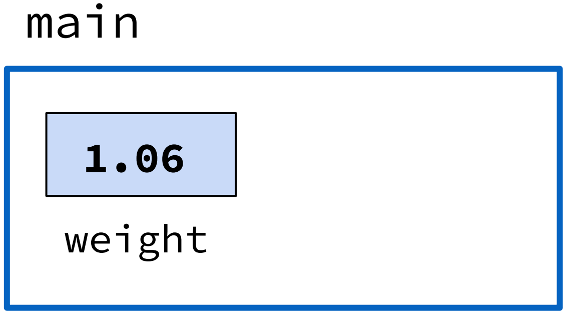
main

1.06

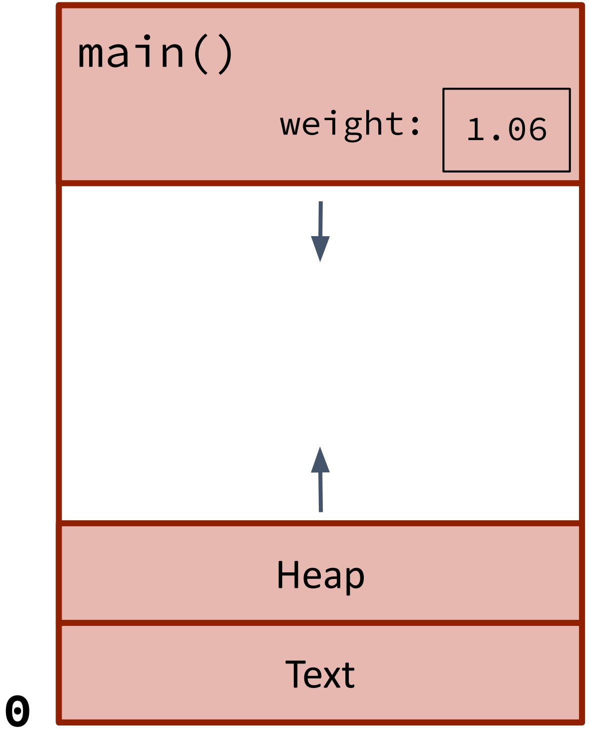
weight

Stack Frames

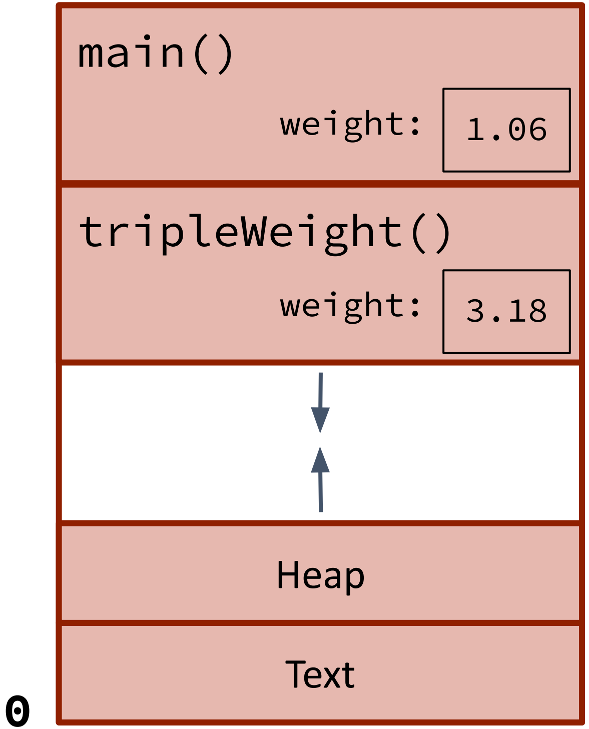




Stack Frames



Stack Frames



The “stack” part of memory is a stack!

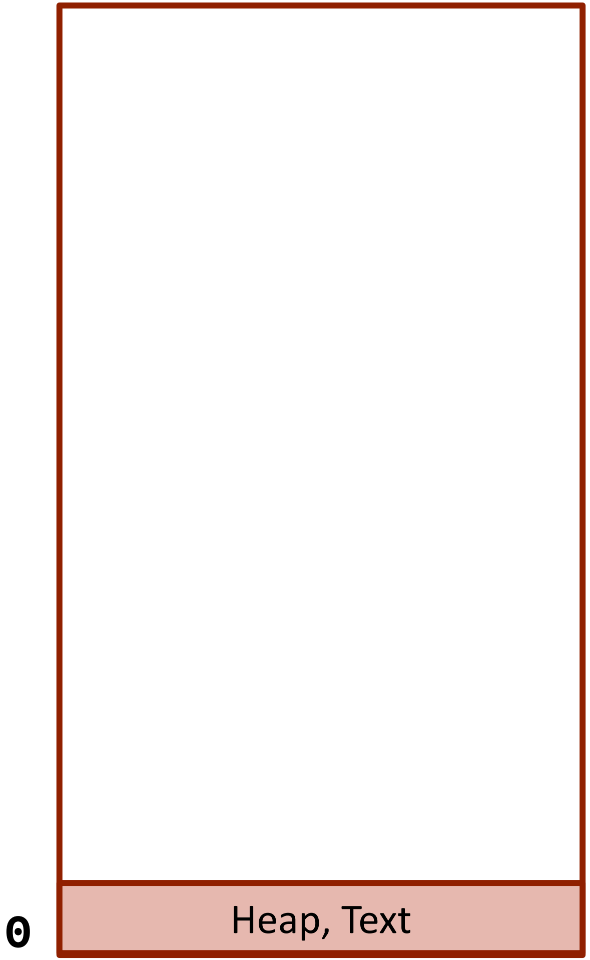
- A function call pushes a stack frame onto the stack
- A function return pops a stack from from the stack

Back to Factorial!

Recursion in action

```
int main () {
    int n = factorial(5);
    cout << “5! = ” << n << endl;
    return 0;
}
```

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

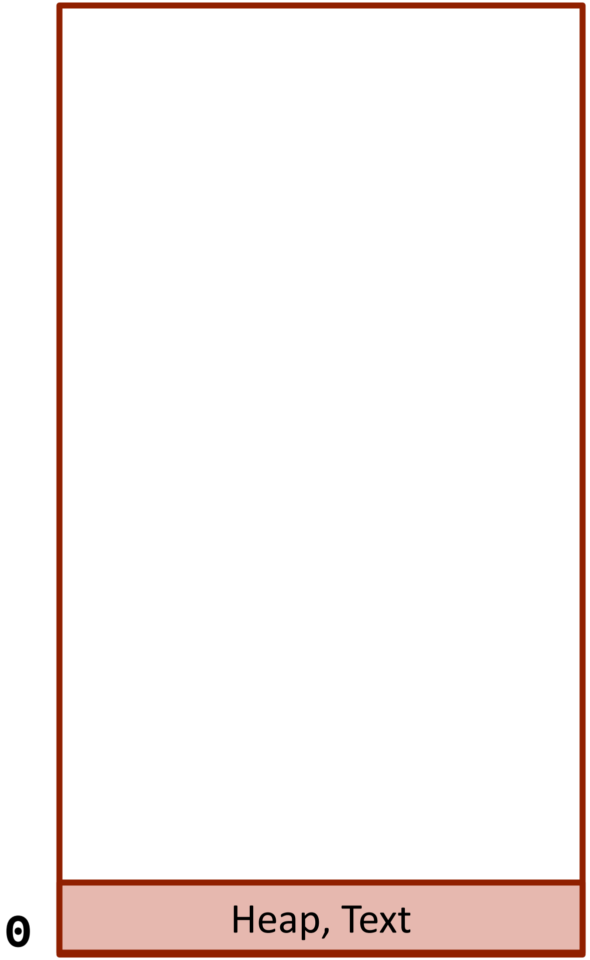


0

Heap, Text

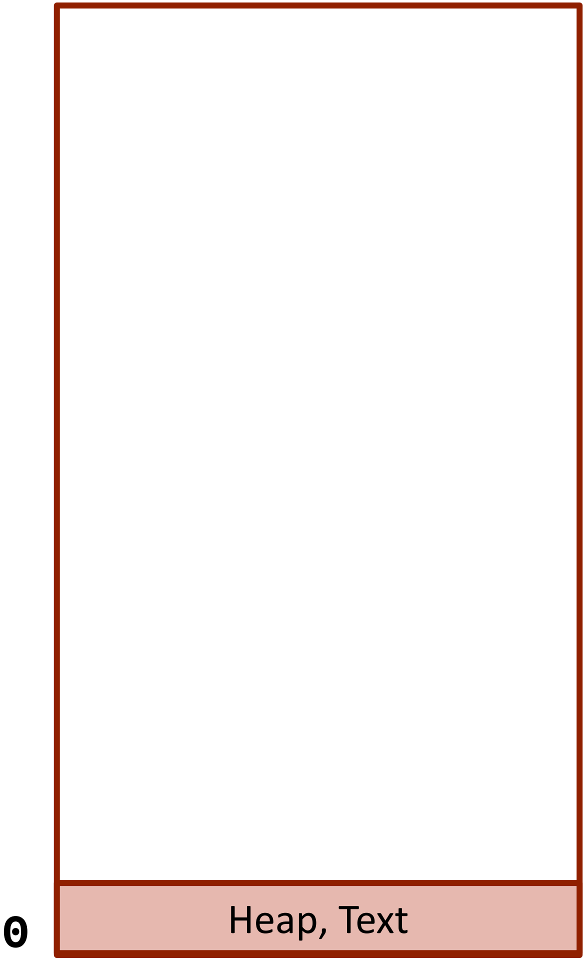
Recursion in action

```
int main () {
    int n = factorial(5);
    cout << “5! = ” << n << endl;
    return 0;
}
```



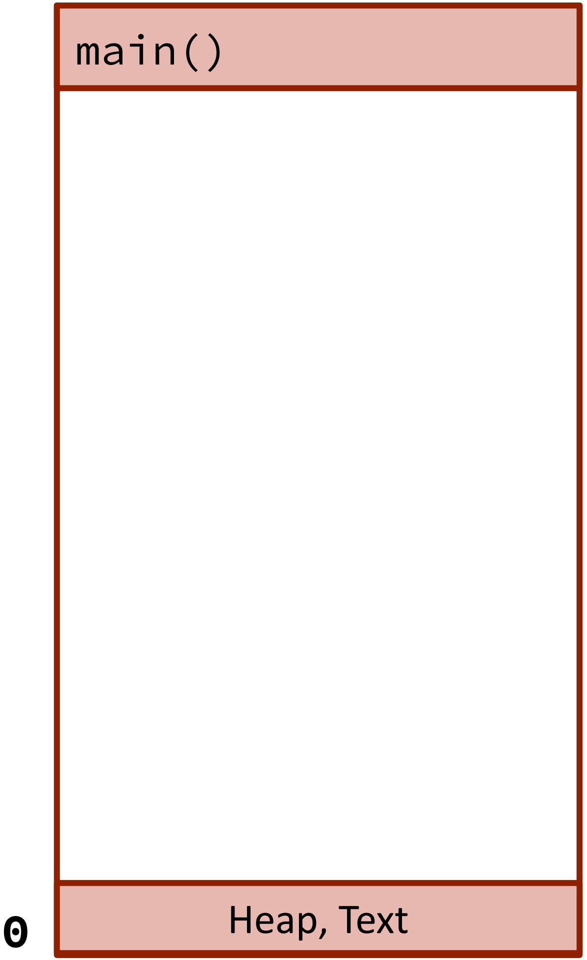
Recursion in action

```
int main () {
    int n = factorial(5);
    cout << “5! = ” << n << endl;
    return 0;
}
```



Recursion in action

```
int main () {
    int n = factorial(5);
    cout << “5! = ” << n << endl;
    return 0;
}
```



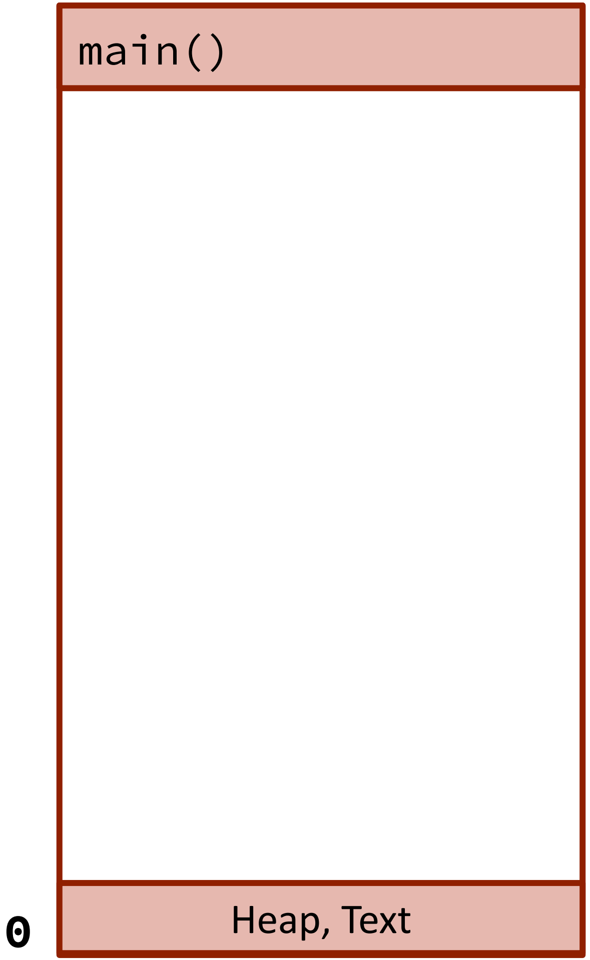
main()

0

Heap, Text

Recursion in action

```
int main () {
    int n = factorial(5);
    cout << “5! = ” << n << endl;
    return 0;
}
```



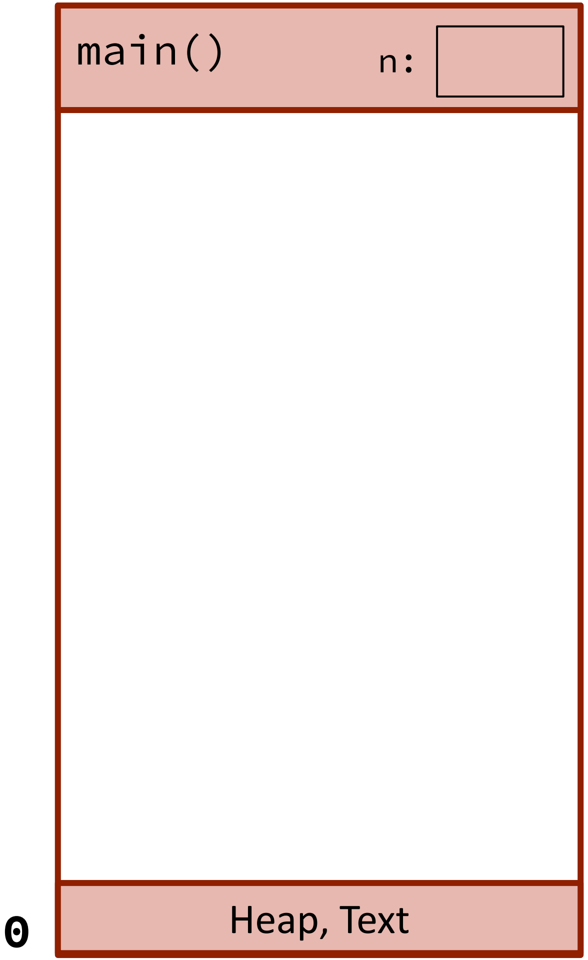
main()

0

Heap, Text

Recursion in action

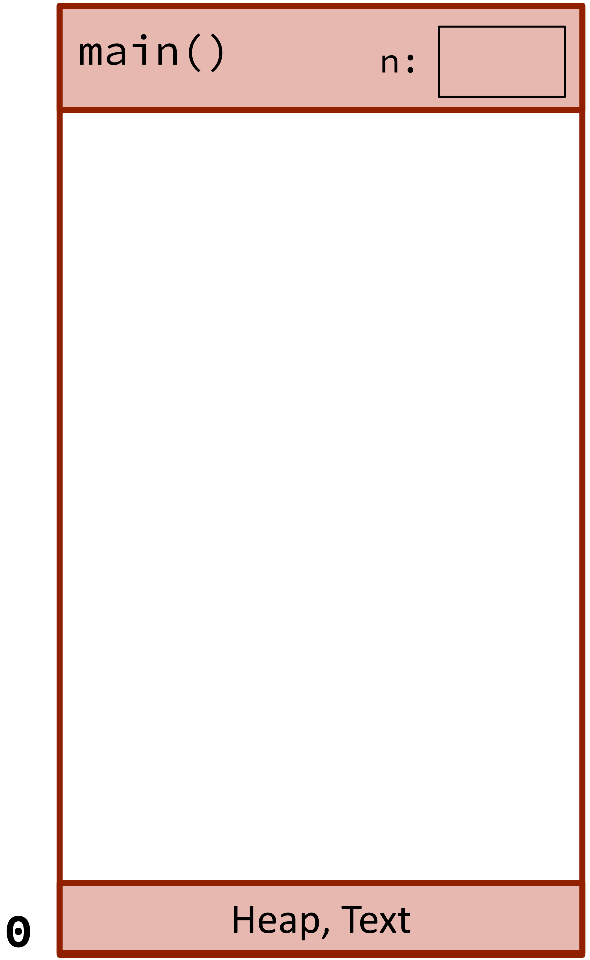
```
int main () {
    int n = factorial(5);
    cout << “5! = ” << n << endl;
    return 0;
}
```



Recursion in action

```
int main () {
```

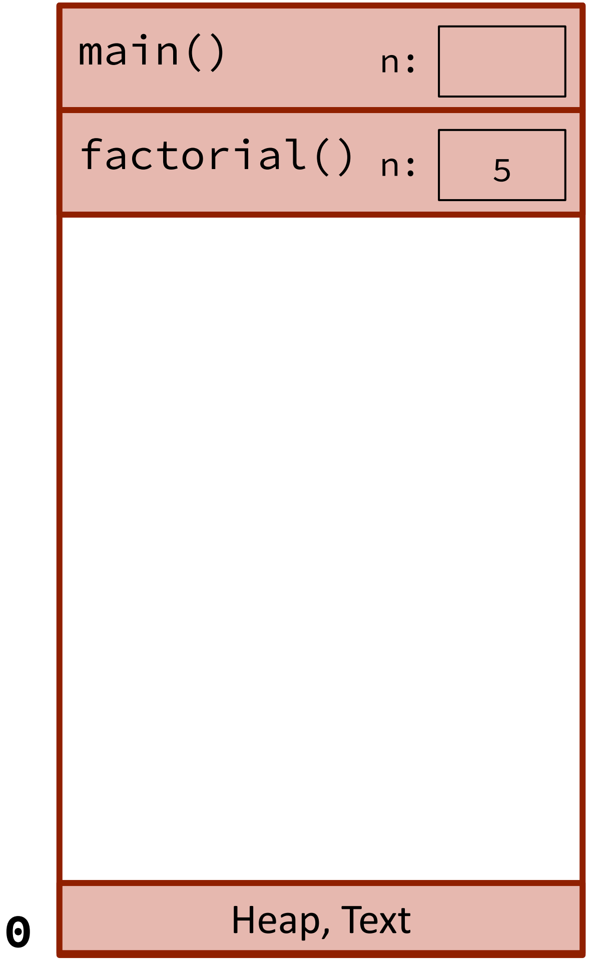
```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```



Recursion in action

```
int main () {
```

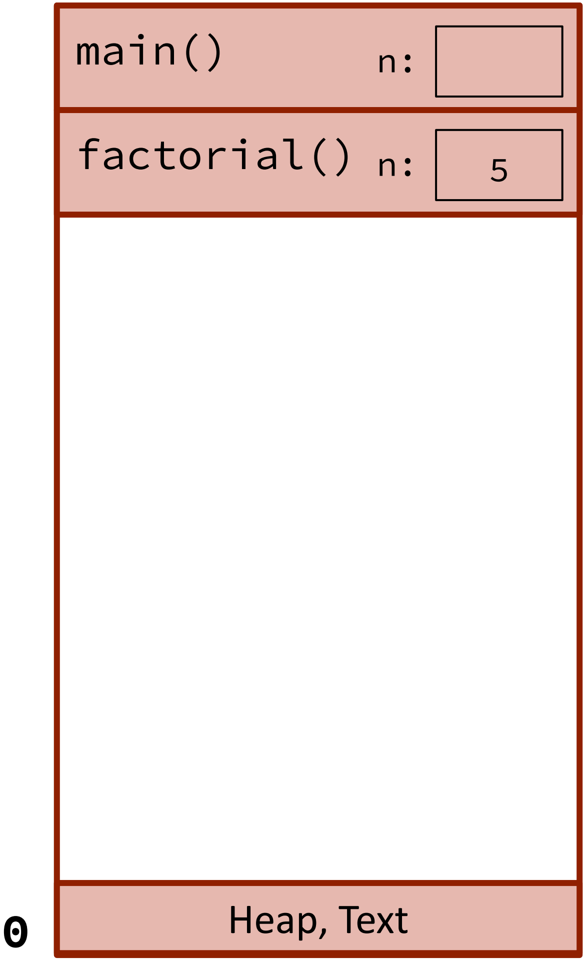
```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```



Recursion in action

```
int main () {
```

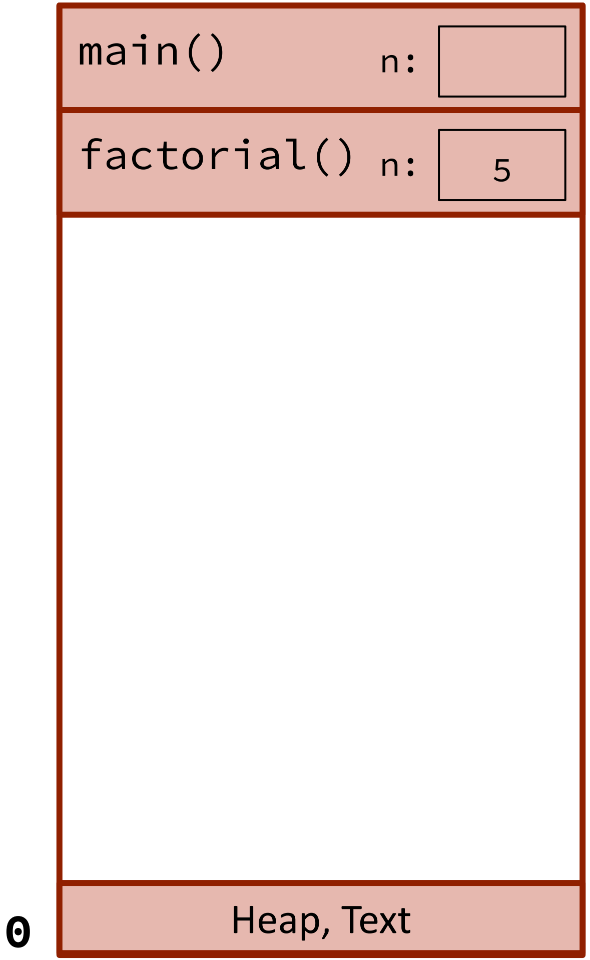
```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```



Recursion in action

```
int main () {
```

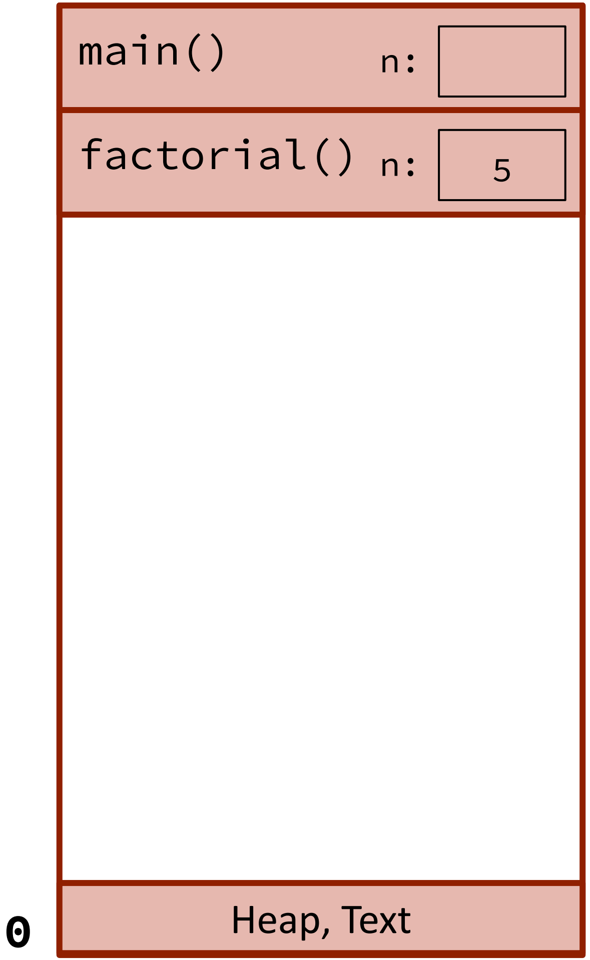
```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```



Recursion in action

```
int main () {
```

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

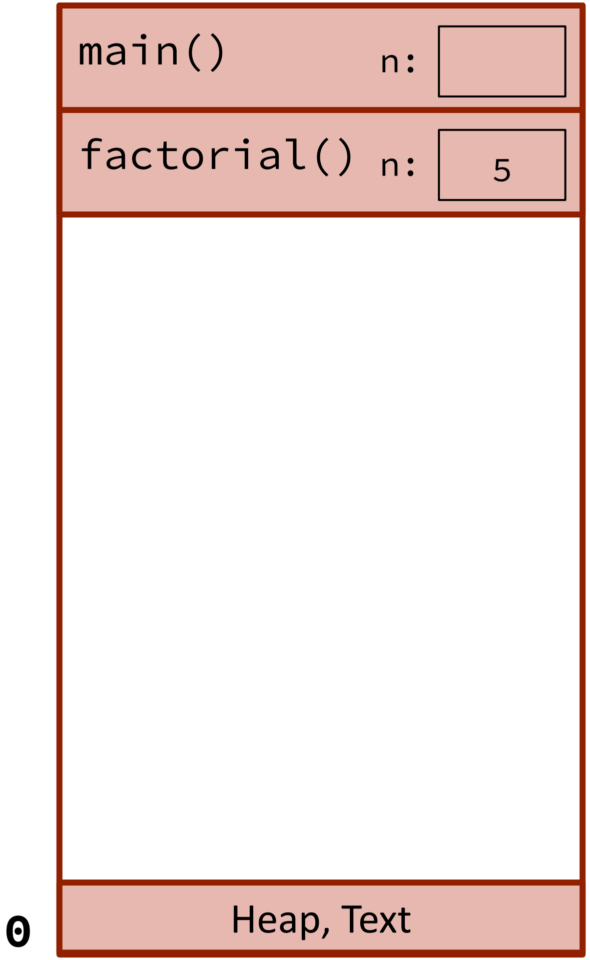


Recursion in action

```
int main () {
```

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

5



main() n:

factorial() n: 5

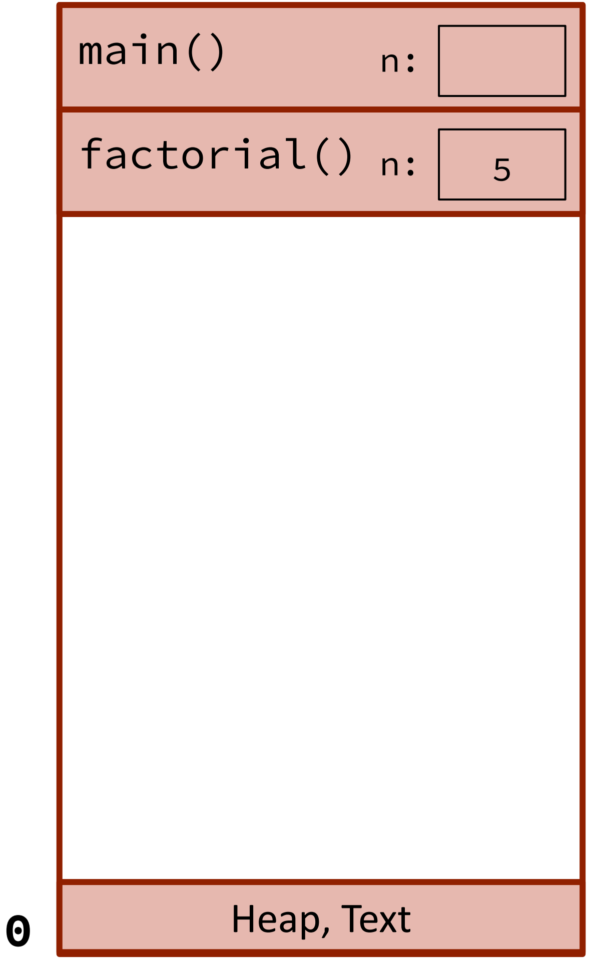
0

Heap, Text

Recursion in action

```
int main () {
```

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

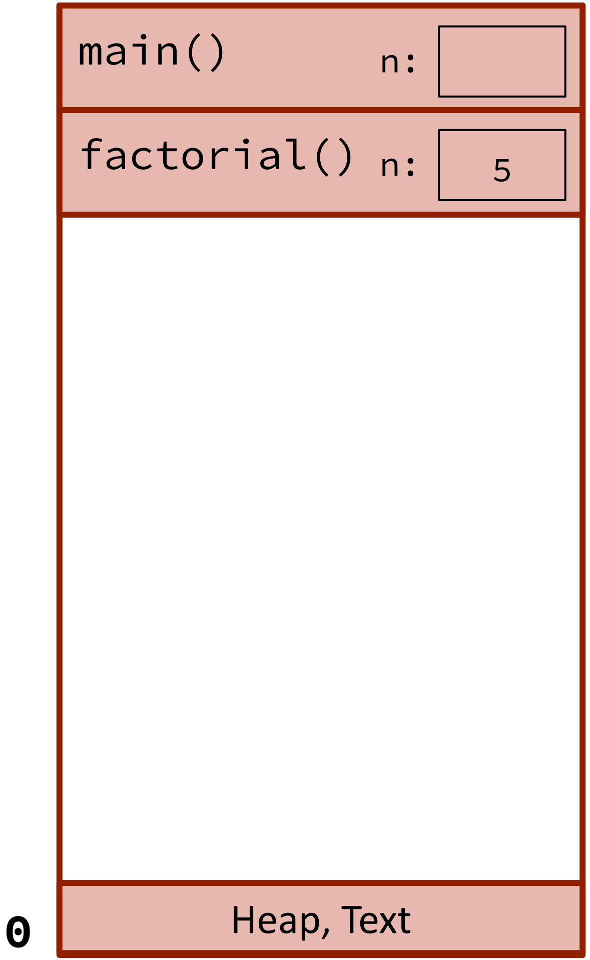


Recursion in action

```
int main () {
```

```
int factorial (int n) {
```

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

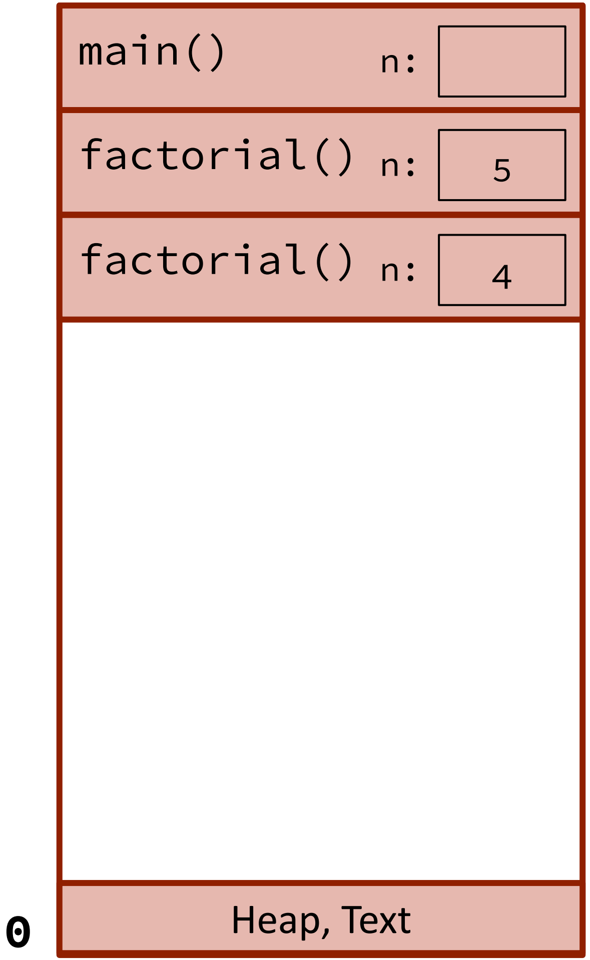


Recursion in action

```
int main () {
```

```
int factorial (int n) {
```

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



main() n:

factorial() n: 5

factorial() n: 4

0

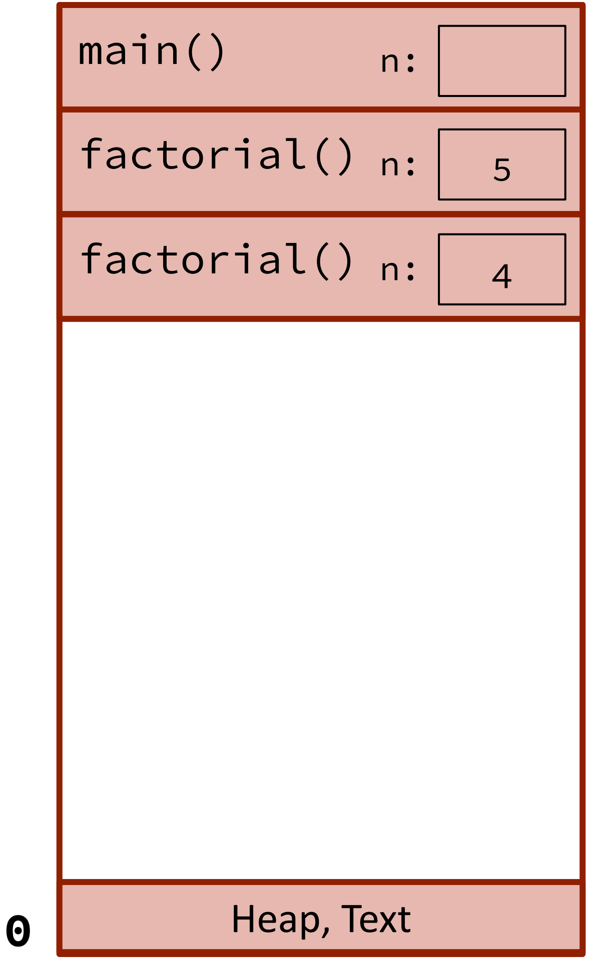
Heap, Text

Recursion in action

```
int main () {
```

```
int factorial (int n) {
```

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



main() n:

factorial() n: 5

factorial() n: 4

0

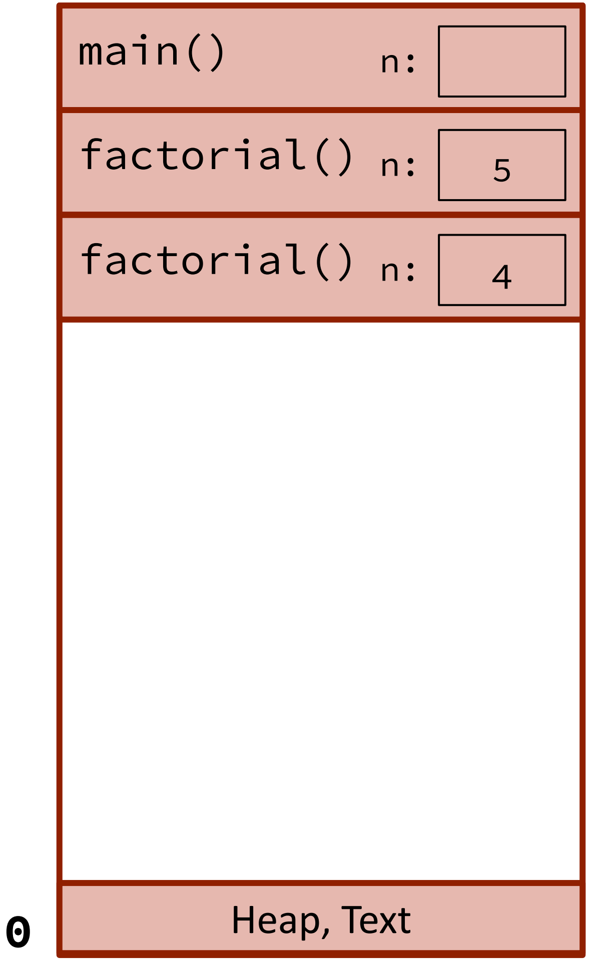
Heap, Text

Recursion in action

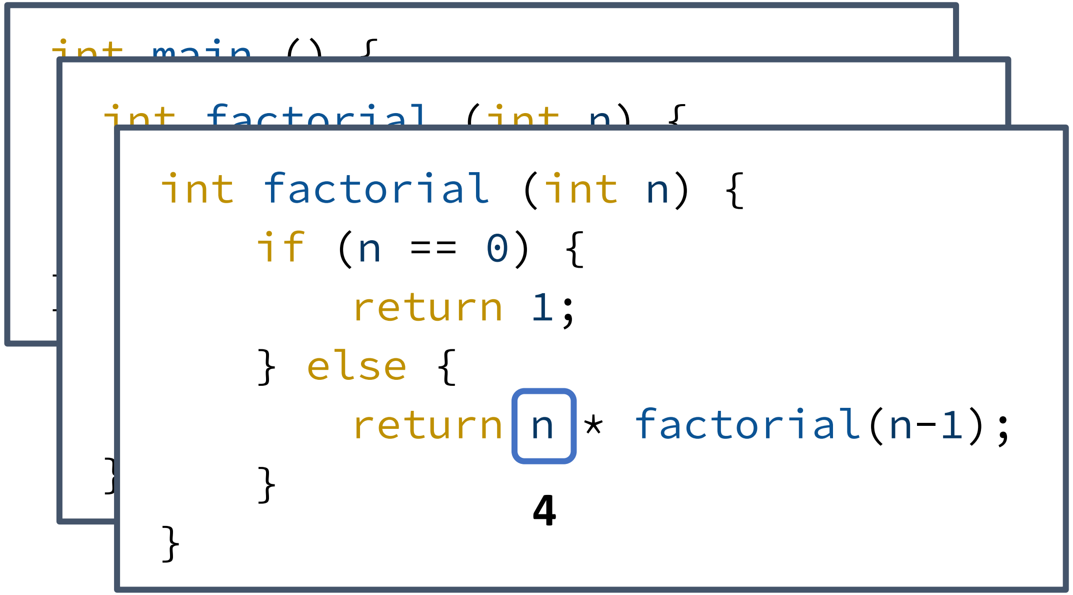
```
int main () {
```

```
int factorial (int n) {
```

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



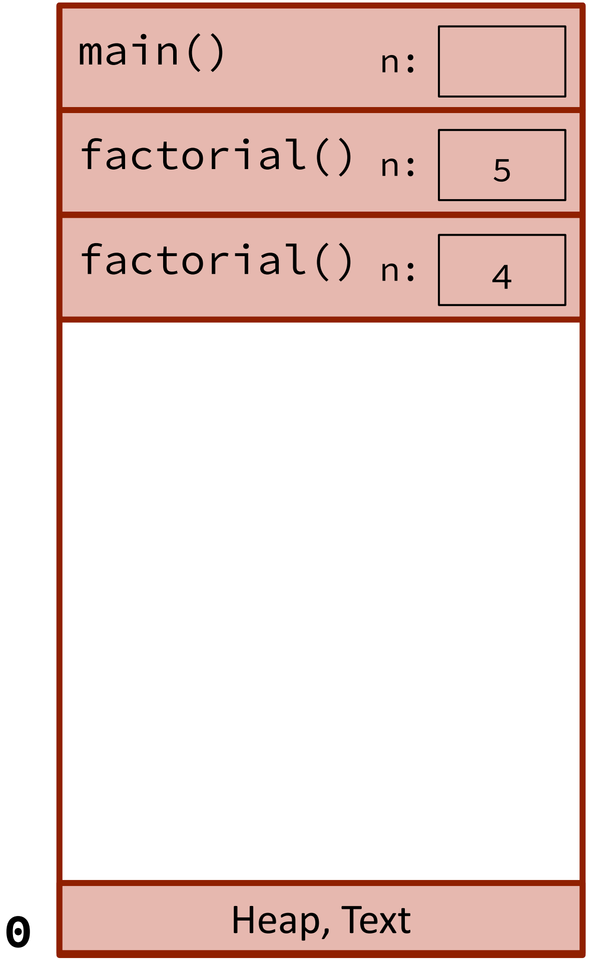
Recursion in action



```
int main () {
```

```
int factorial (int n) {
```

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

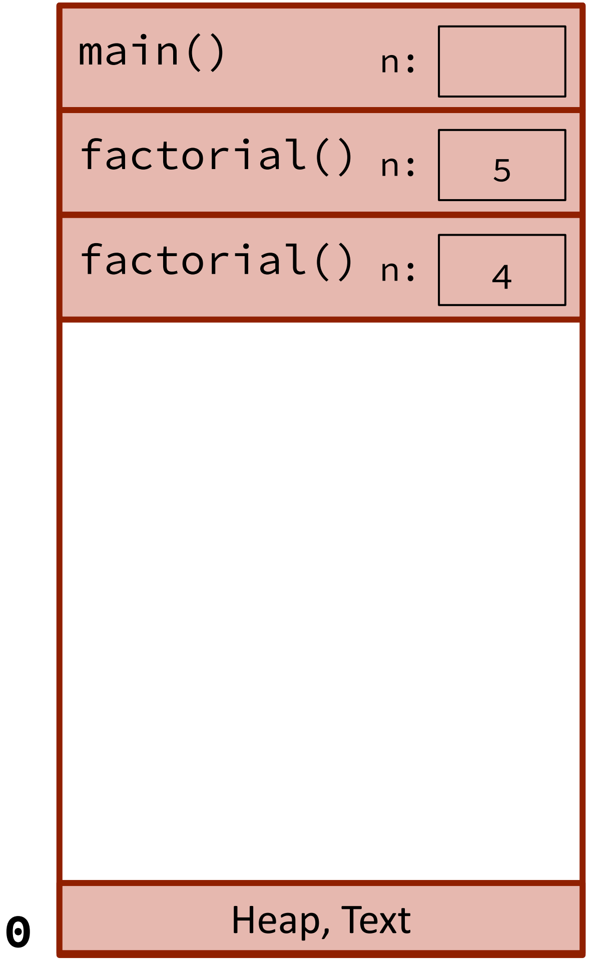


Recursion in action

```
int main () {
```

```
int factorial (int n) {
```

```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }           4
}
```



main() n:

factorial() n: 5

factorial() n: 4

0

Heap, Text

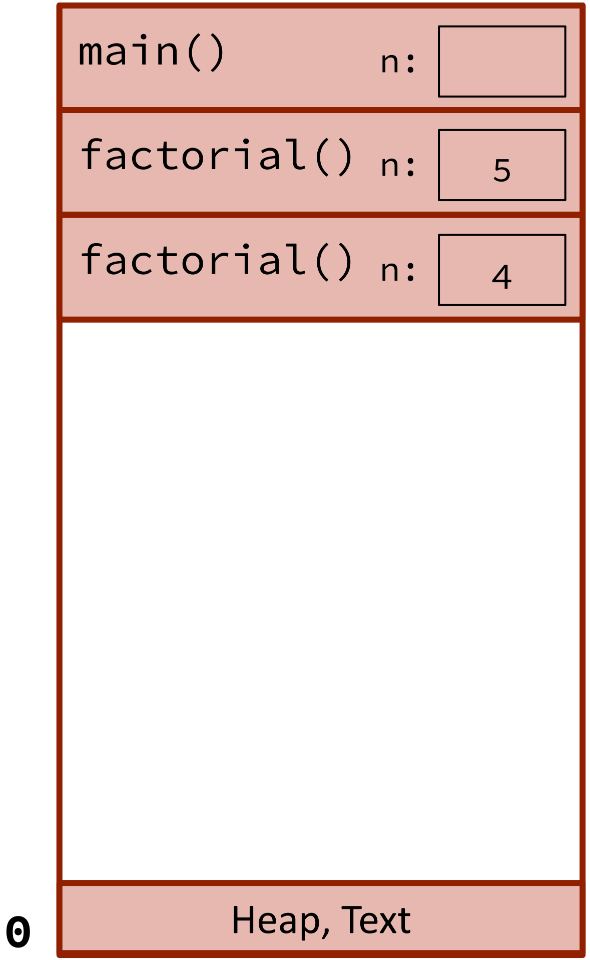
Recursion in action

```
int main () {
```

```
int factorial (int n) {
```

```
int factorial (int n) {
```

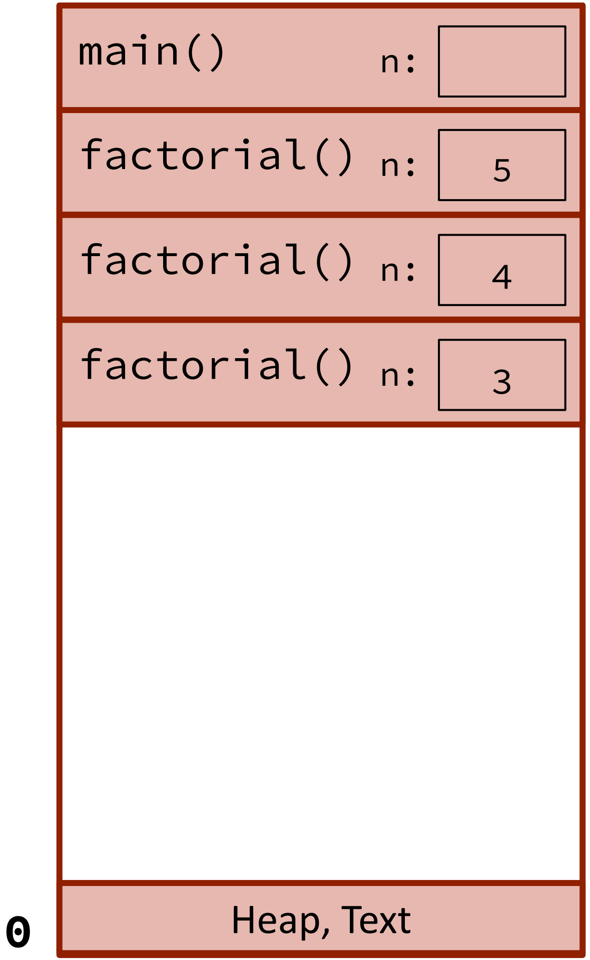
```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```



Recursion in action

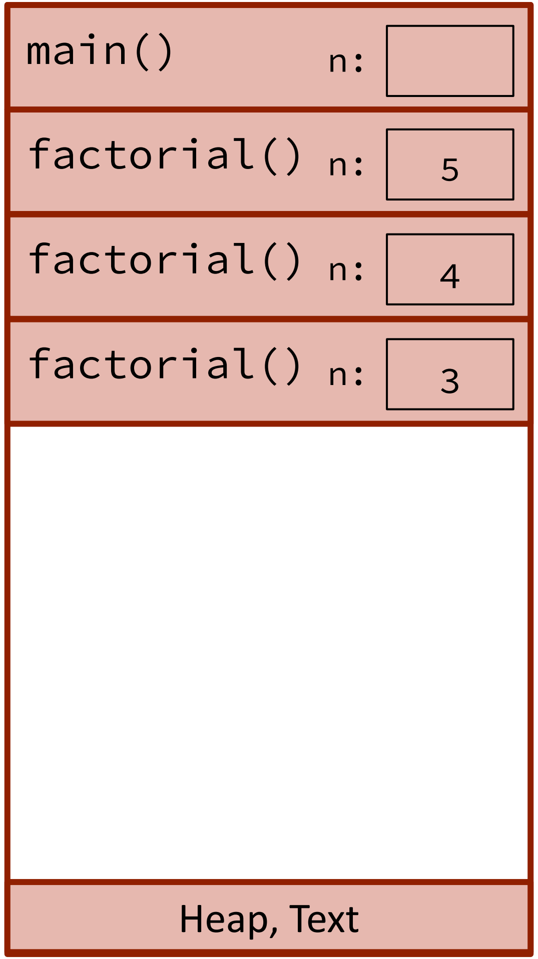
```
int main () {
  int factorial (int n) {
    int factorial (int n) {
```

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



Recursion in action

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



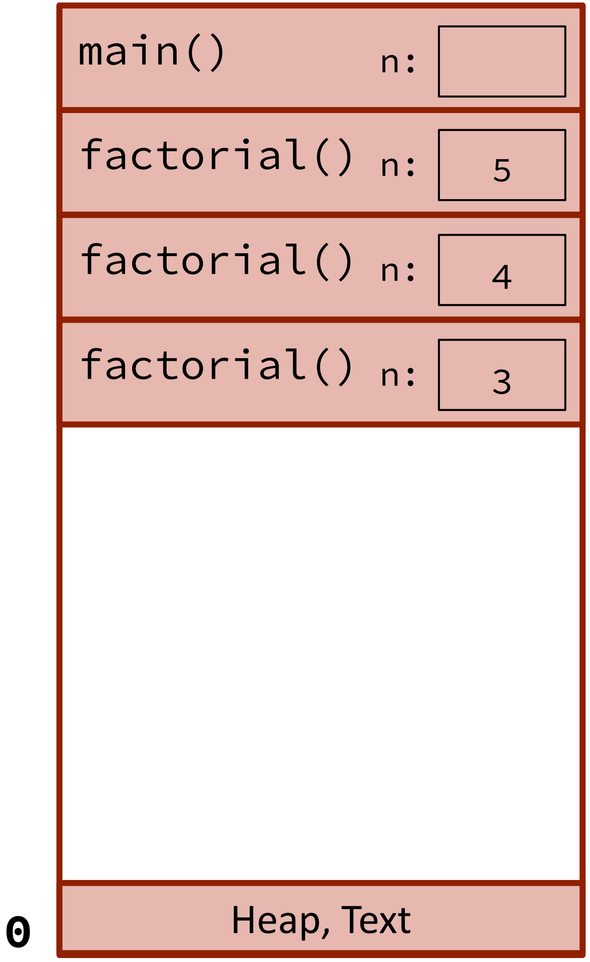
Recursion in action

```
int main () {
```

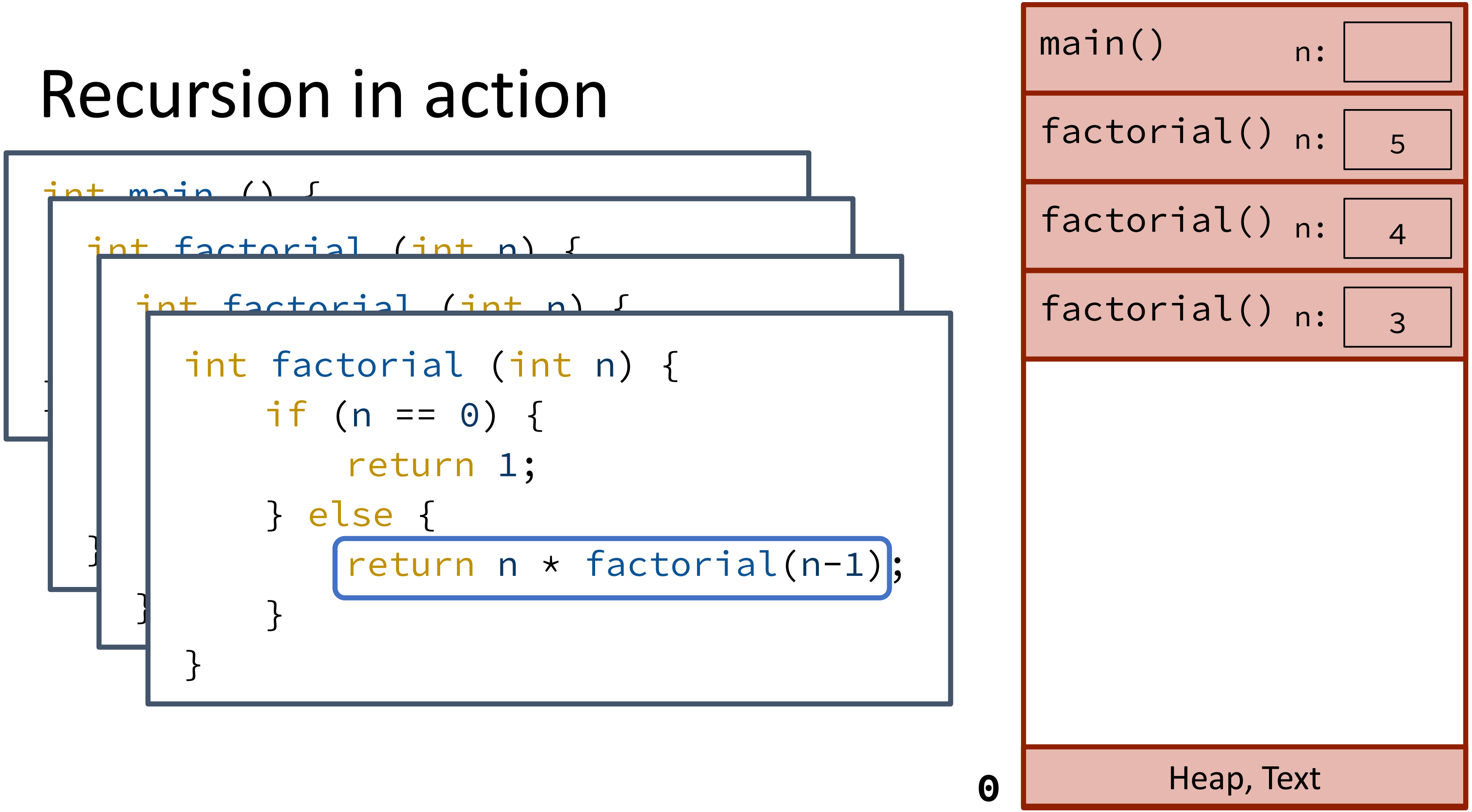
```
int factorial (int n) {
```

```
int factorial (int n) {
```

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



Recursion in action



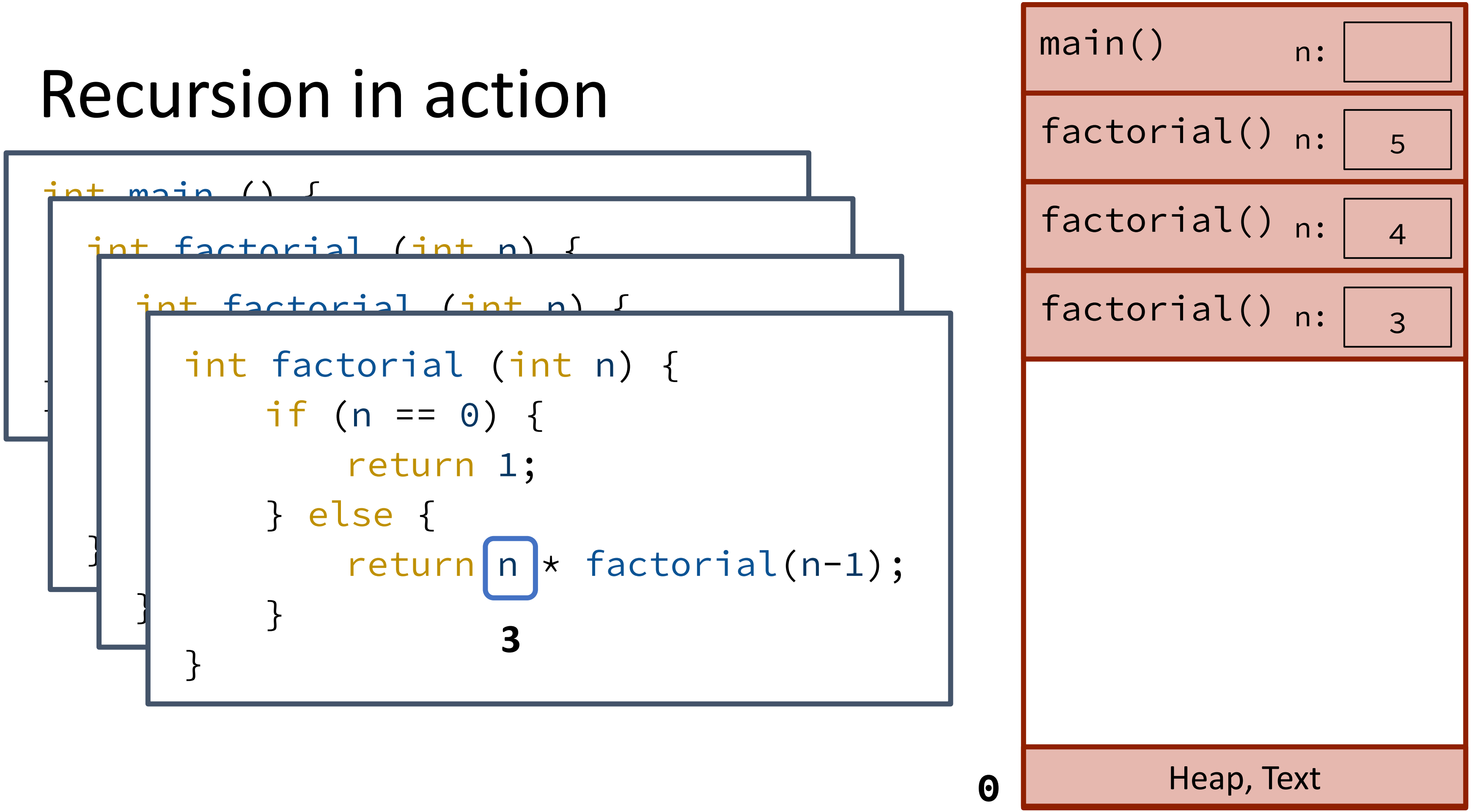
```
int main () {
```

```
int factorial (int n) {
```

```
int factorial (int n) {
```

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

Recursion in action



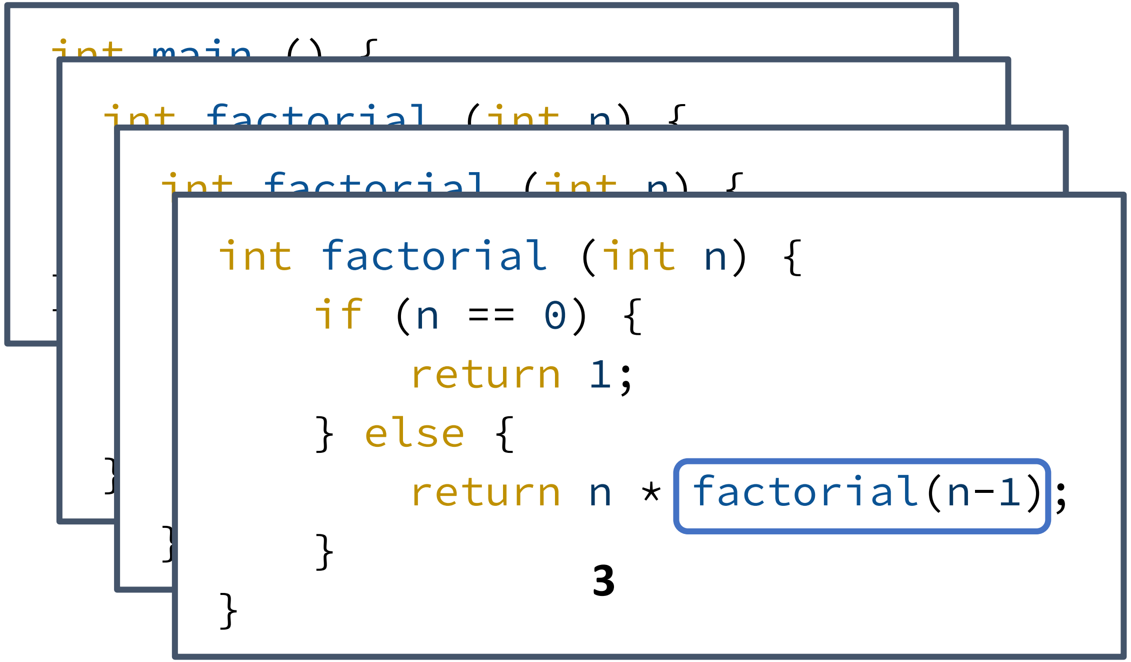
```
int main () {
```

```
int factorial (int n) {
```

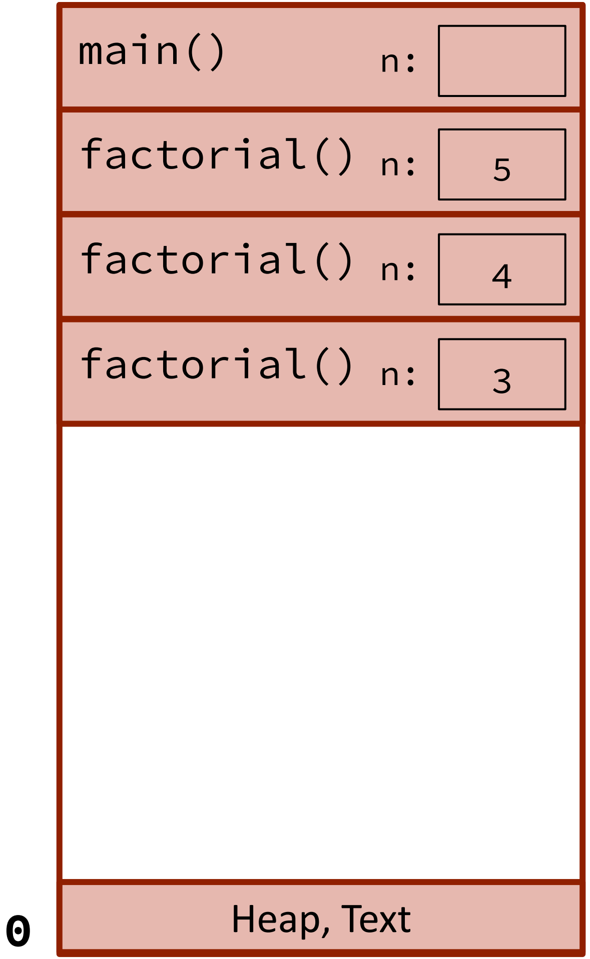
```
int factorial (int n) {
```

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

Recursion in action



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



main() n:

factorial() n: 5

factorial() n: 4

factorial() n: 3

0

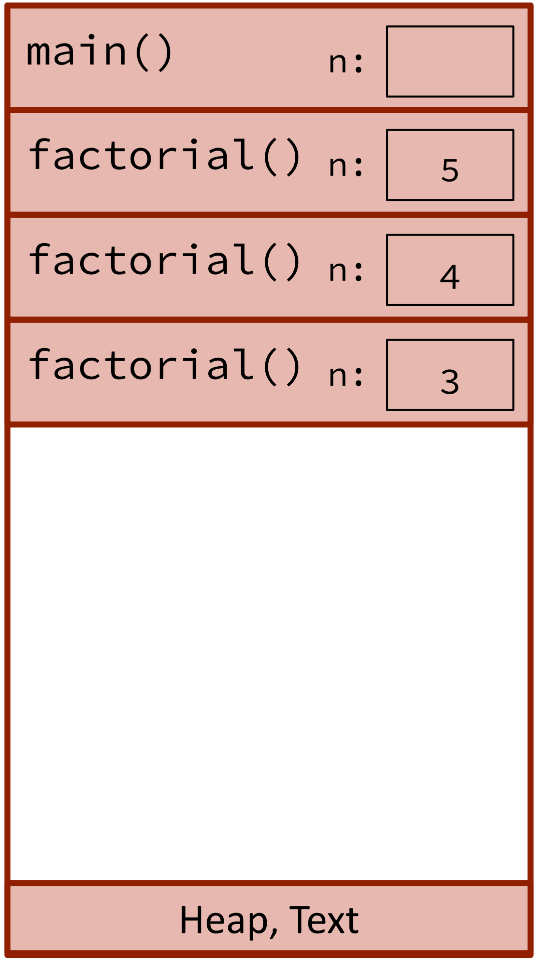
Heap, Text

Recursion in action

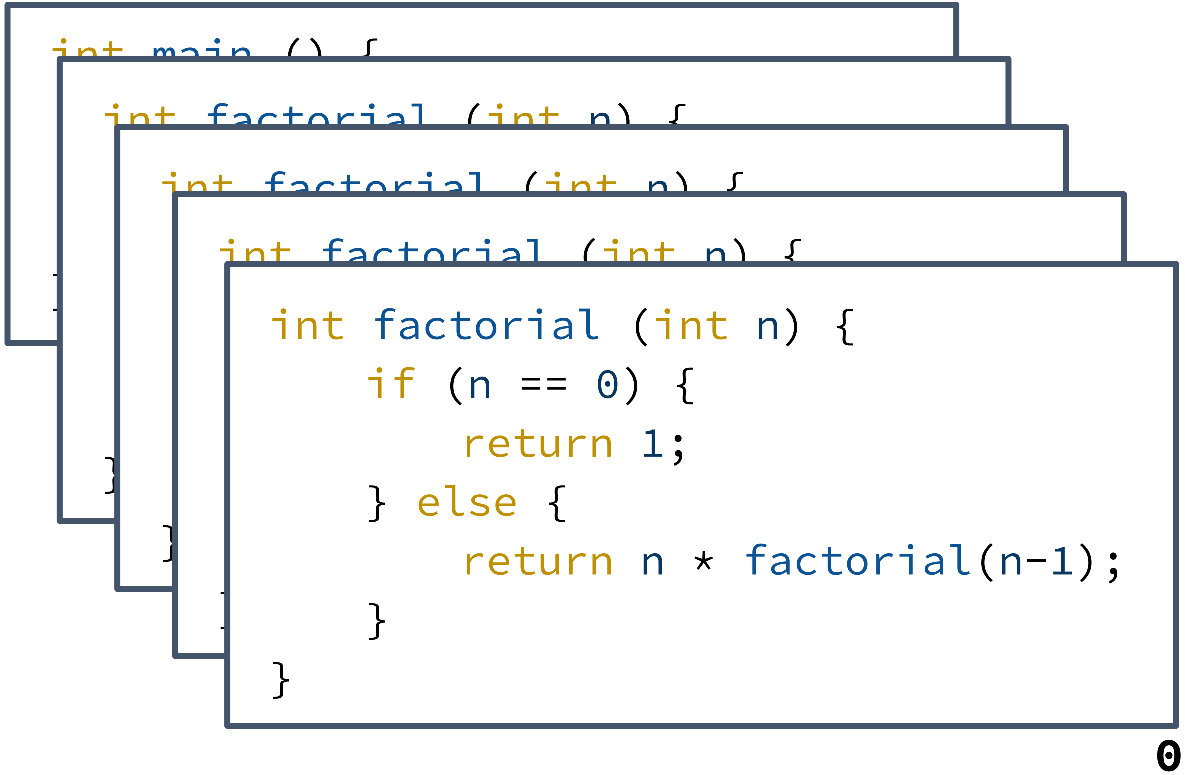
```
int main () {
  int factorial (int n) {
    int factorial (int n) {
      int factorial (int n) {
```

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

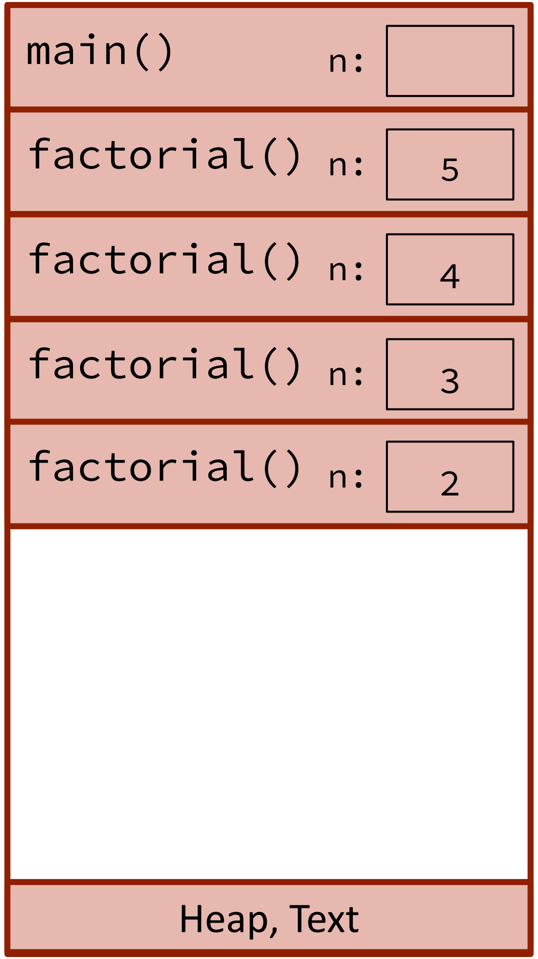
0



Recursion in action

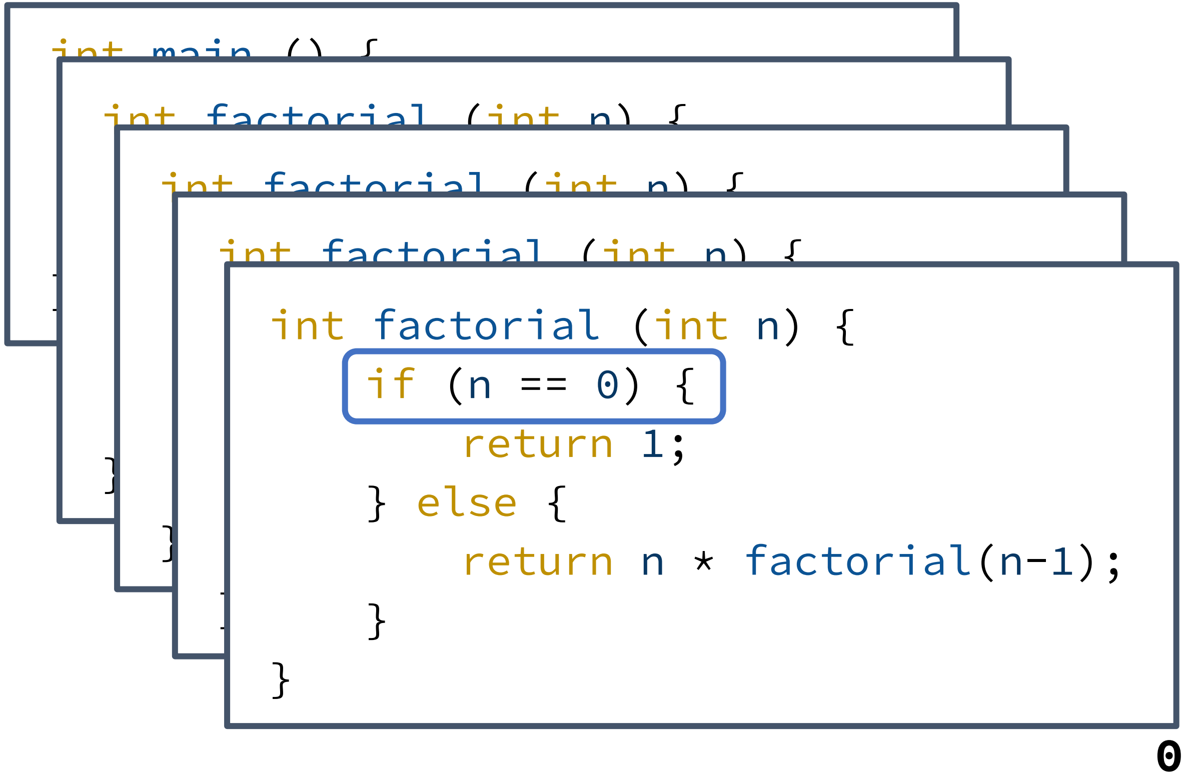


```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```



main()	n:
factorial()	n: 5
factorial()	n: 4
factorial()	n: 3
factorial()	n: 2

Recursion in action



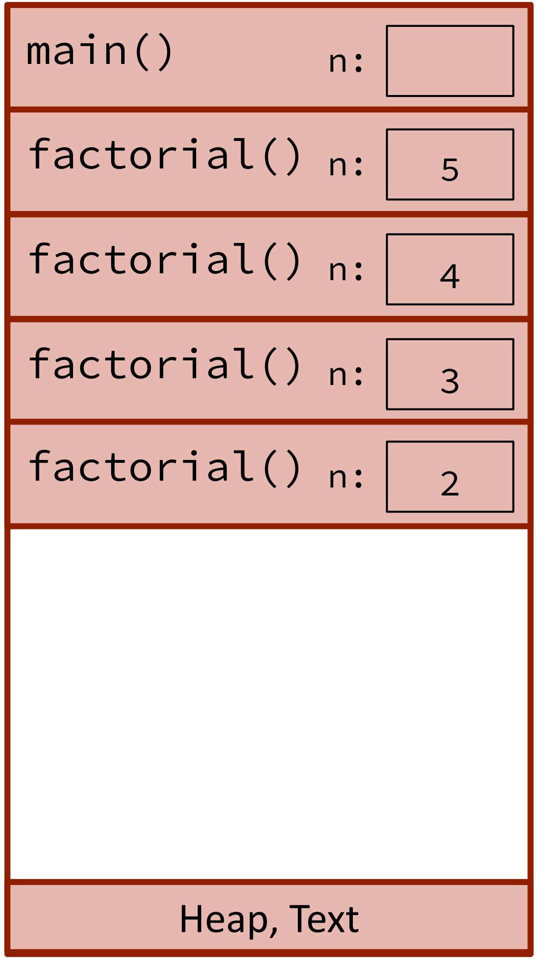
```
int main () {
```

```
int factorial (int n) {
```

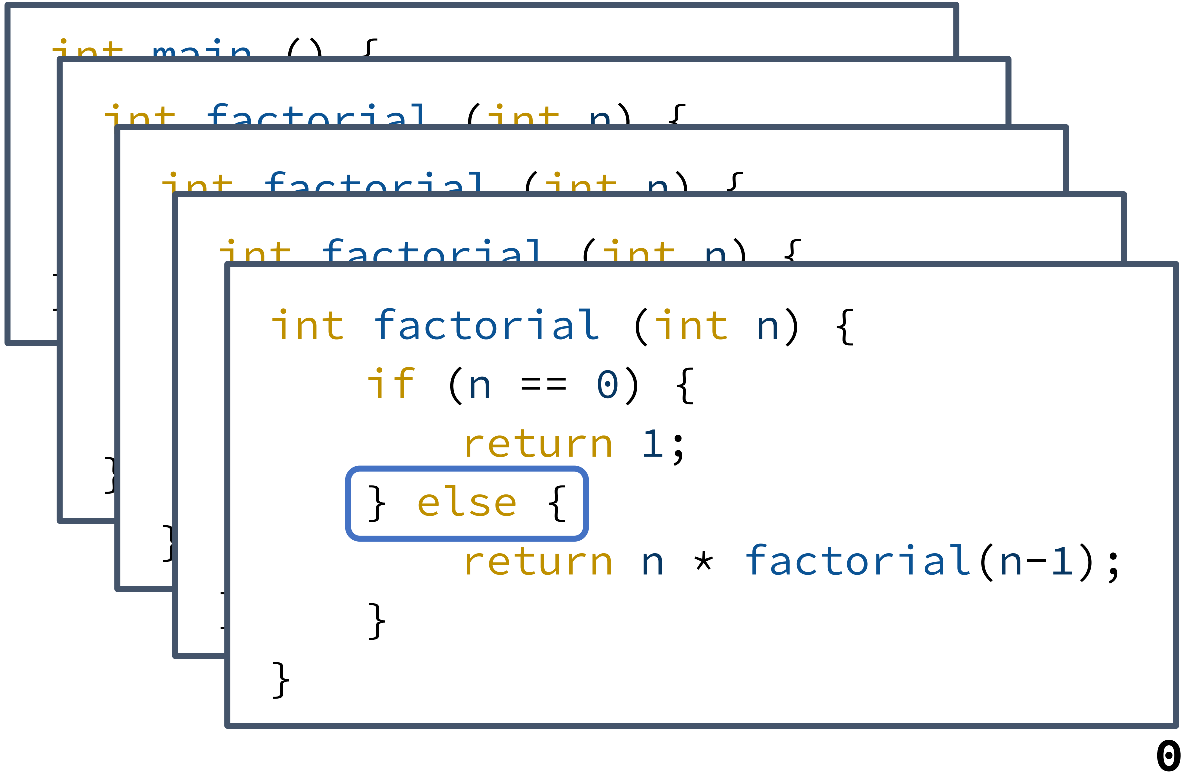
```
int factorial (int n) {
```

```
int factorial (int n) {
```

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



Recursion in action



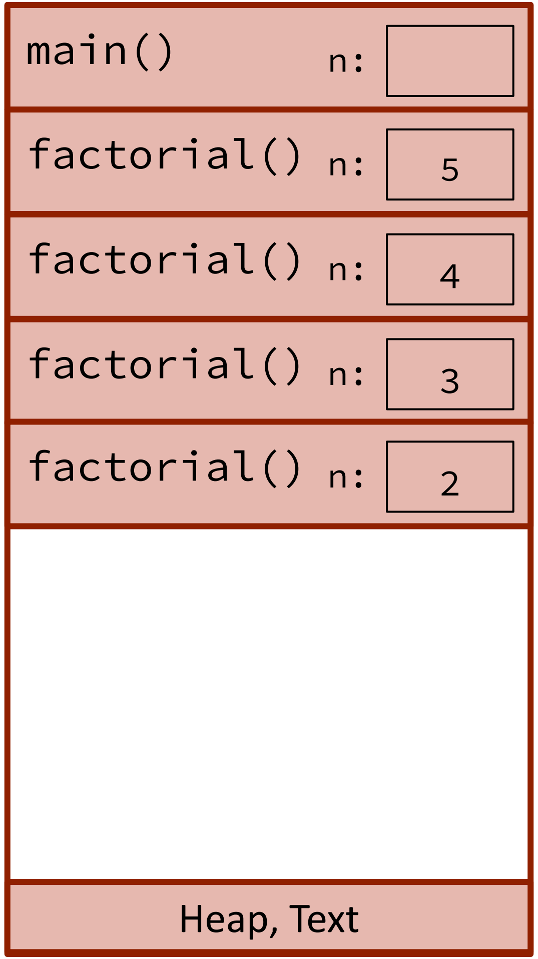
```
int main () {
```

```
int factorial (int n) {
```

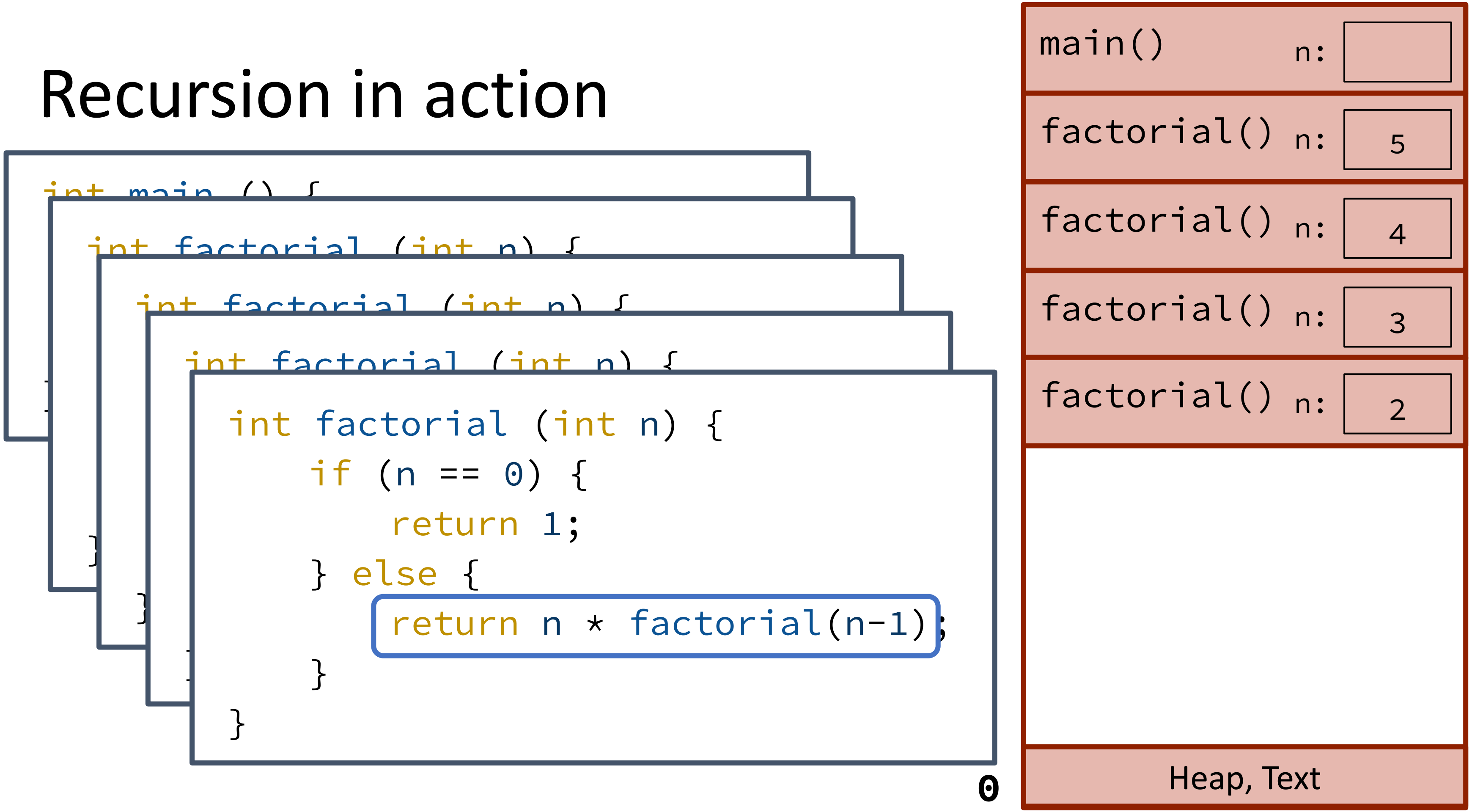
```
int factorial (int n) {
```

```
int factorial (int n) {
```

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

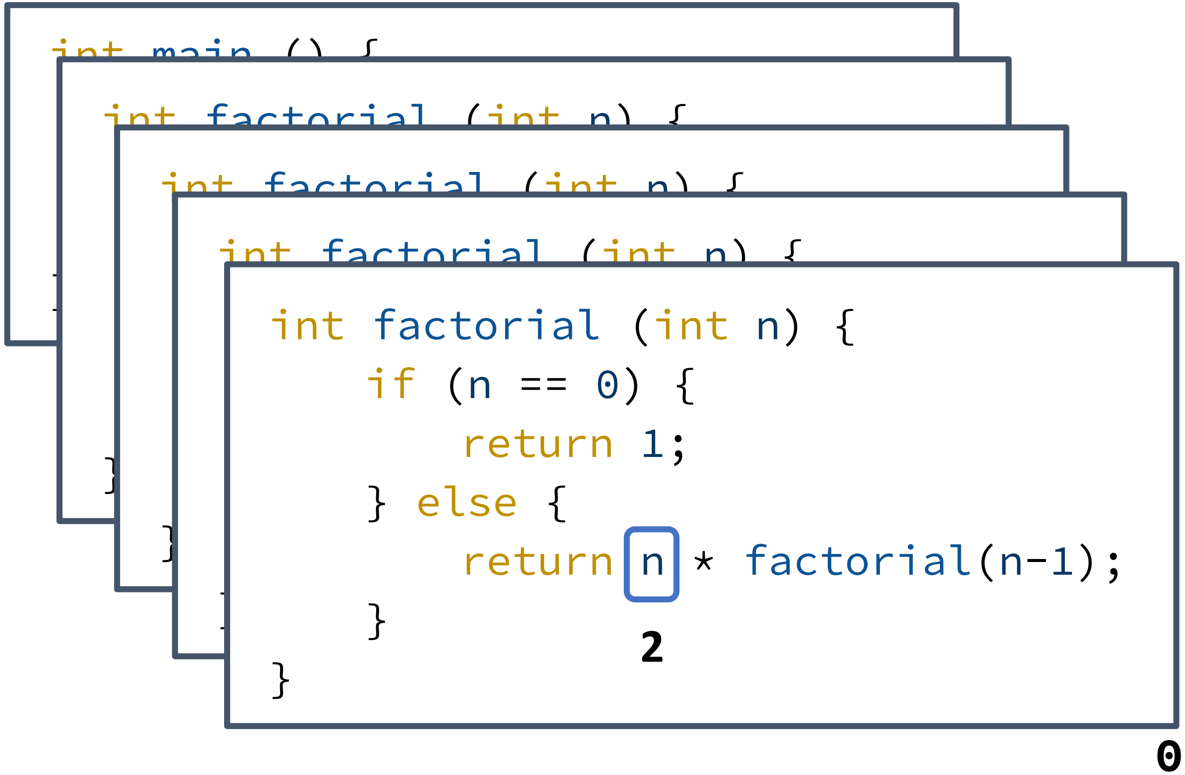


Recursion in action



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

Recursion in action



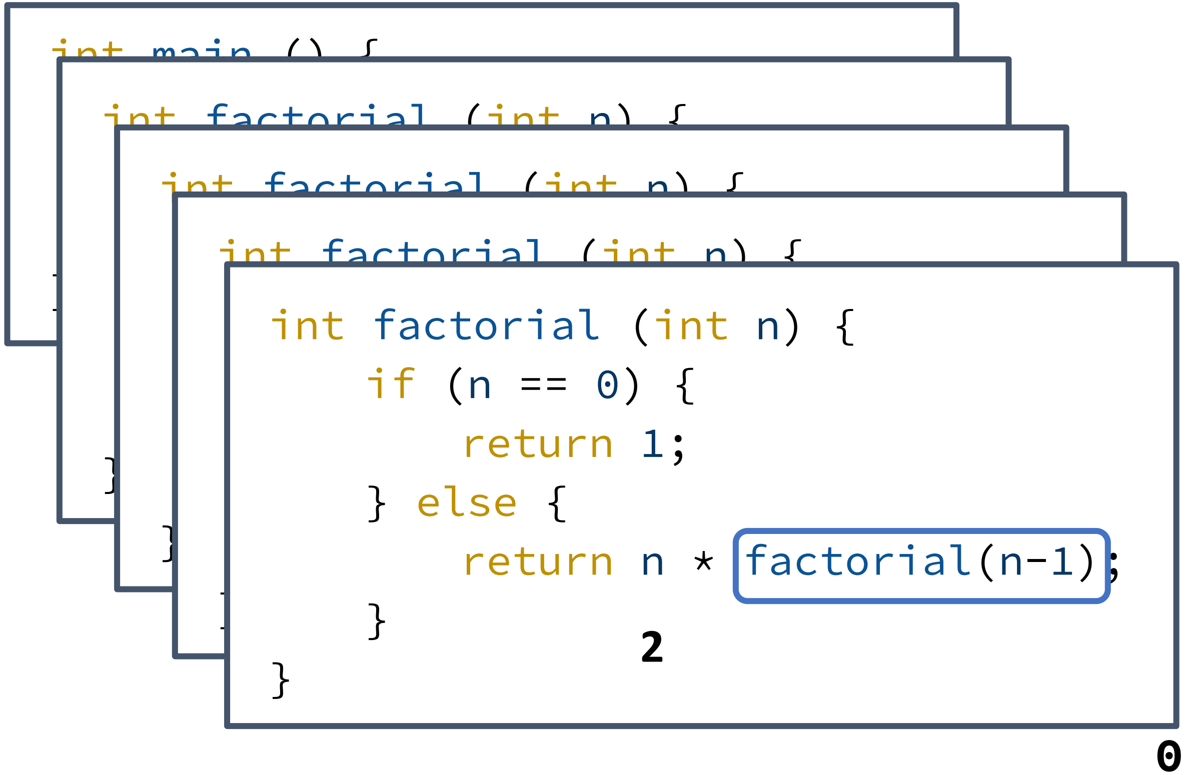
```
int main () {
int factorial (int n) {
int factorial (int n) {
int factorial (int n) {
```

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

main()	n:	
factorial()	n:	5
factorial()	n:	4
factorial()	n:	3
factorial()	n:	2

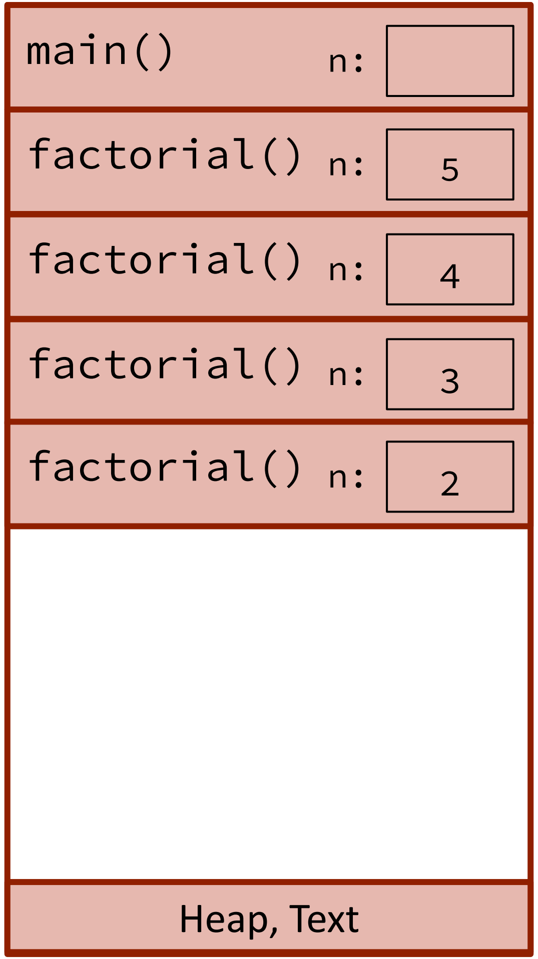
Heap, Text

Recursion in action



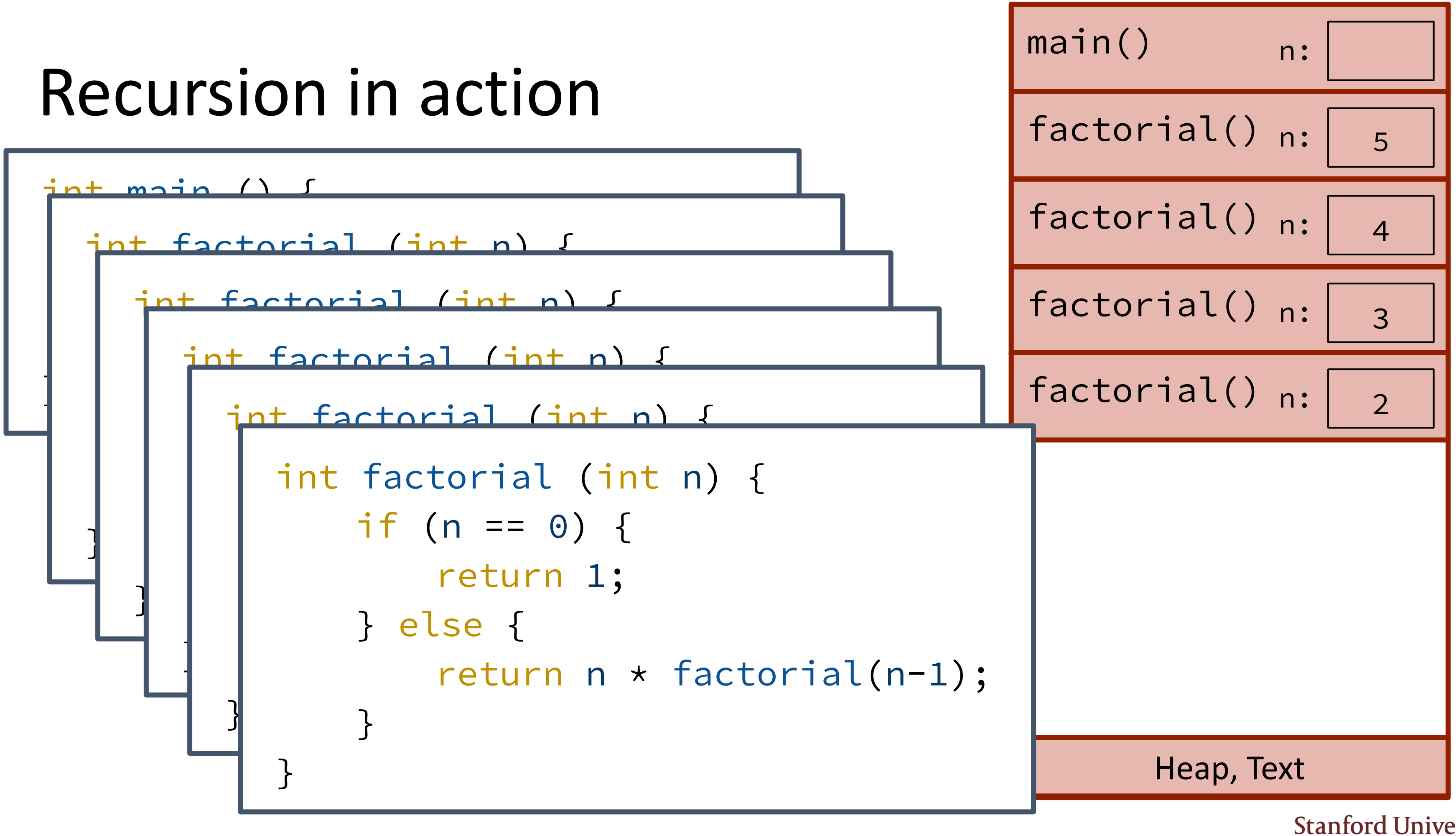
```
int main () {
int factorial (int n) {
int factorial (int n) {
int factorial (int n) {
```

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



Stanford University

Recursion in action



```
int main () {
```

```
int factorial (int n) {
```

```
int factorial (int n) {
```

```
int factorial (int n) {
```

```
int factorial (int n) {
```

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

main() n:

factorial() n: 5

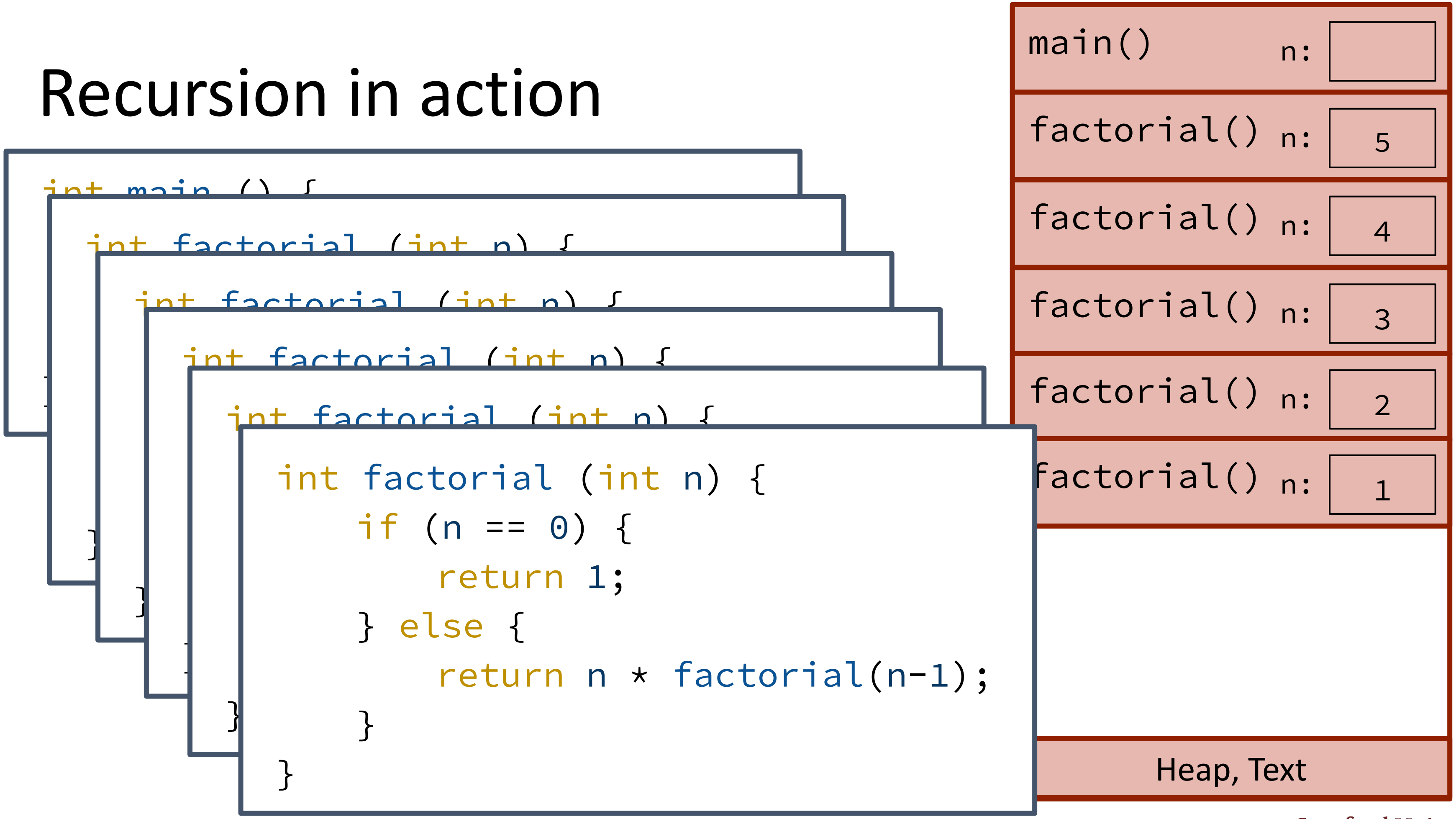
factorial() n: 4

factorial() n: 3

factorial() n: 2

Heap, Text

Recursion in action



```
int main () {
```

```
int factorial (int n) {
```

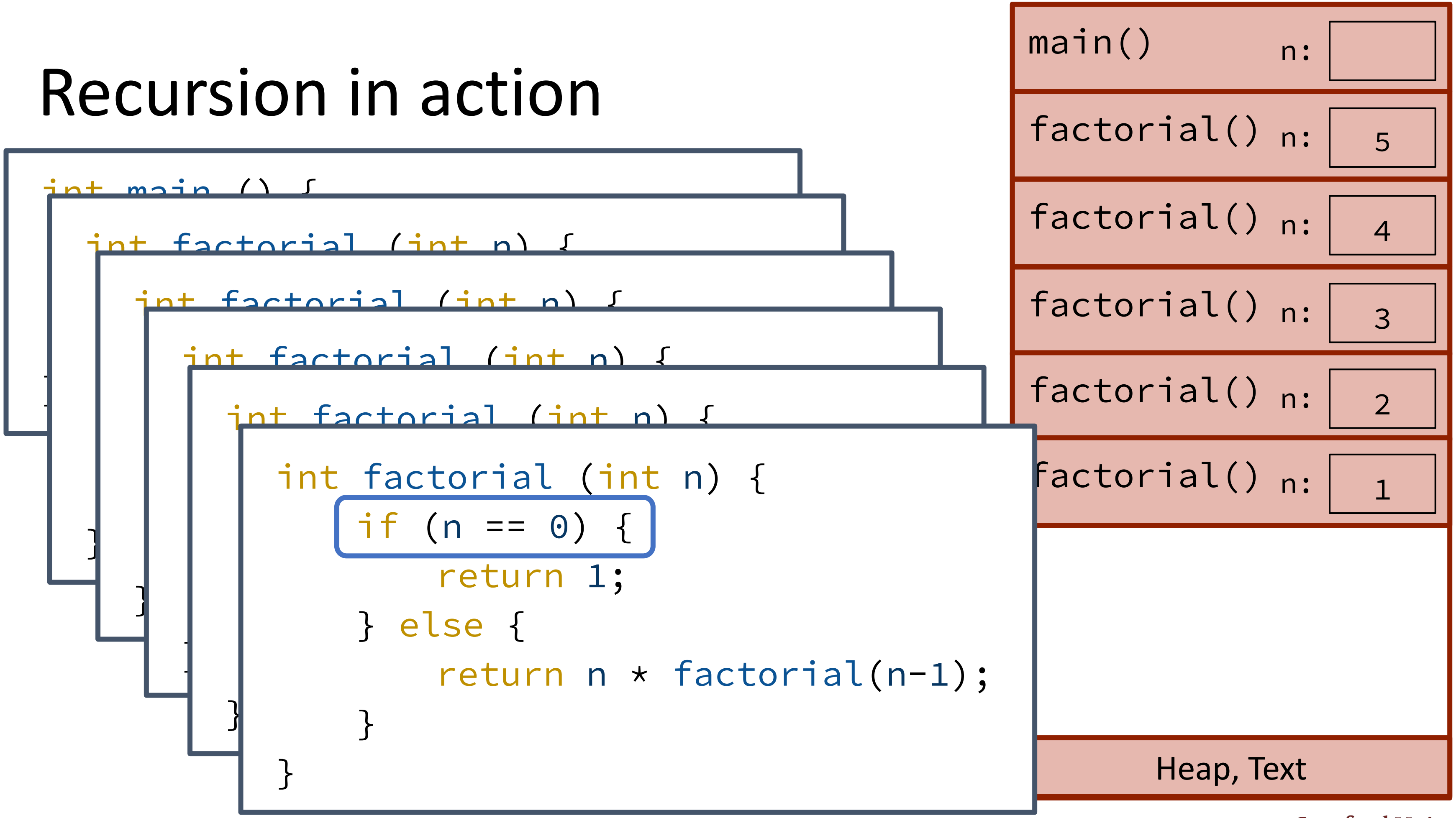
```
int factorial (int n) {
```

```
int factorial (int n) {
```

```
int factorial (int n) {
```

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

Recursion in action



```
int main () {
```

```
int factorial (int n) {
```

```
int factorial (int n) {
```

```
int factorial (int n) {
```

```
int factorial (int n) {
```

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

main() n:

factorial() n: 5

factorial() n: 4

factorial() n: 3

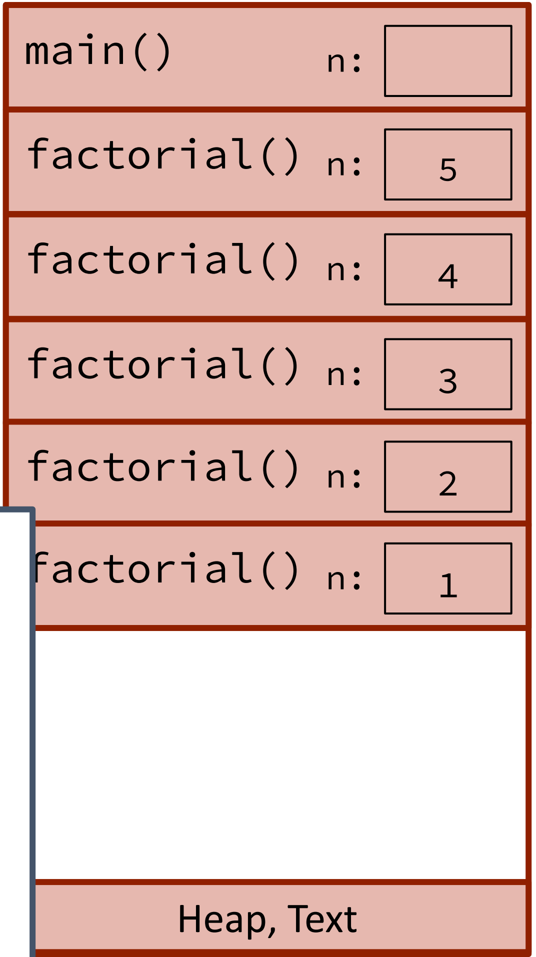
factorial() n: 2

factorial() n: 1

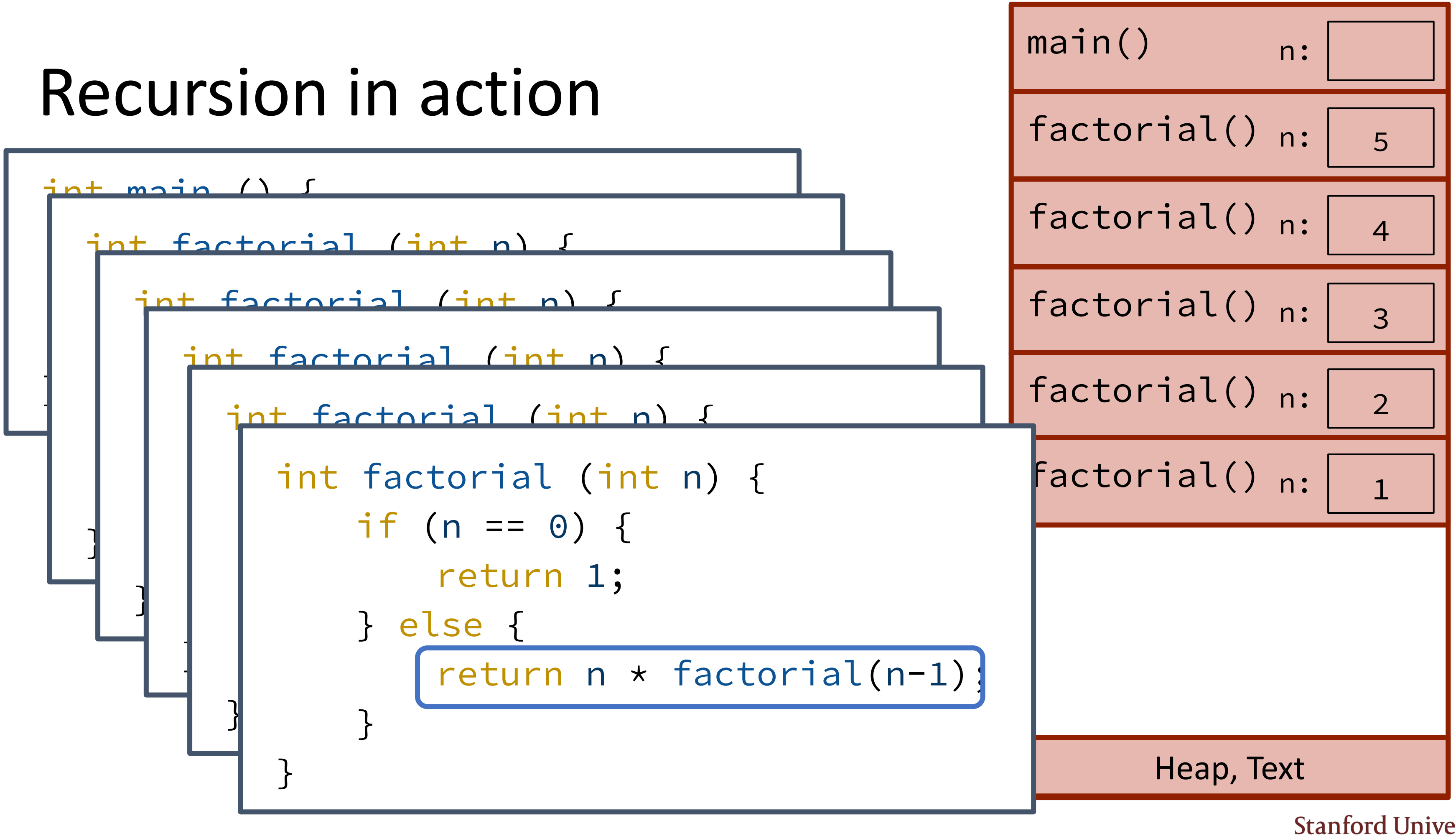
Heap, Text

Recursion in action

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



Recursion in action



```
int main () {
```

```
int factorial (int n) {
```

```
int factorial (int n) {
```

```
int factorial (int n) {
```

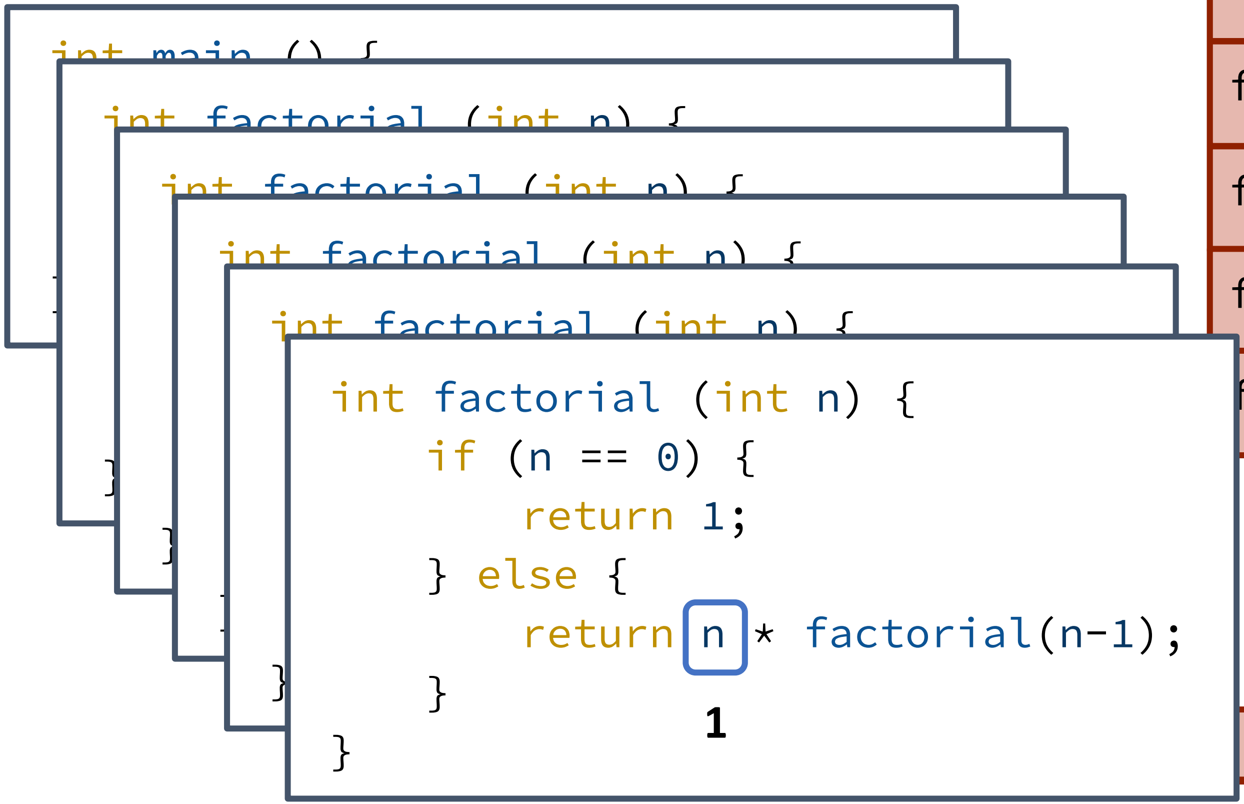
```
int factorial (int n) {
```

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

main()	n:	
factorial()	n:	5
factorial()	n:	4
factorial()	n:	3
factorial()	n:	2
factorial()	n:	1

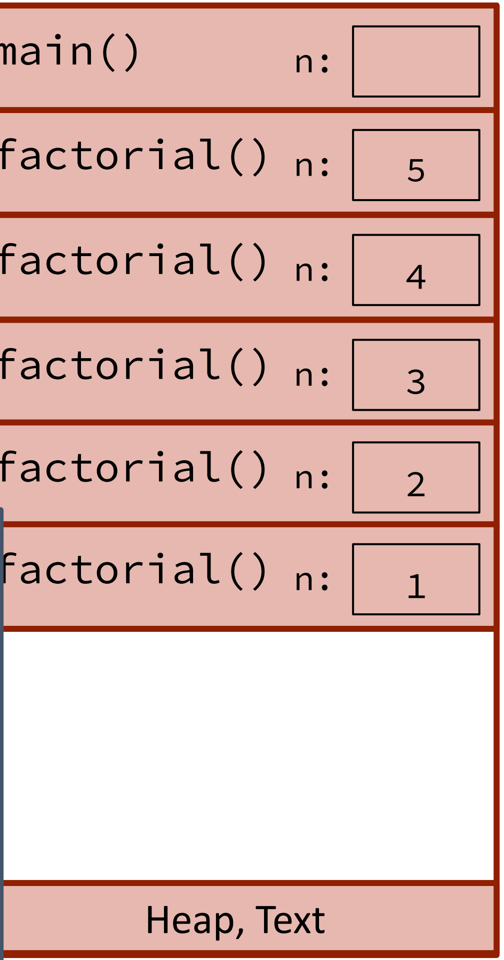
Heap, Text

Recursion in action



```
int main () {
  int factorial (int n) {
    int factorial (int n) {
      int factorial (int n) {
        int factorial (int n) {
```

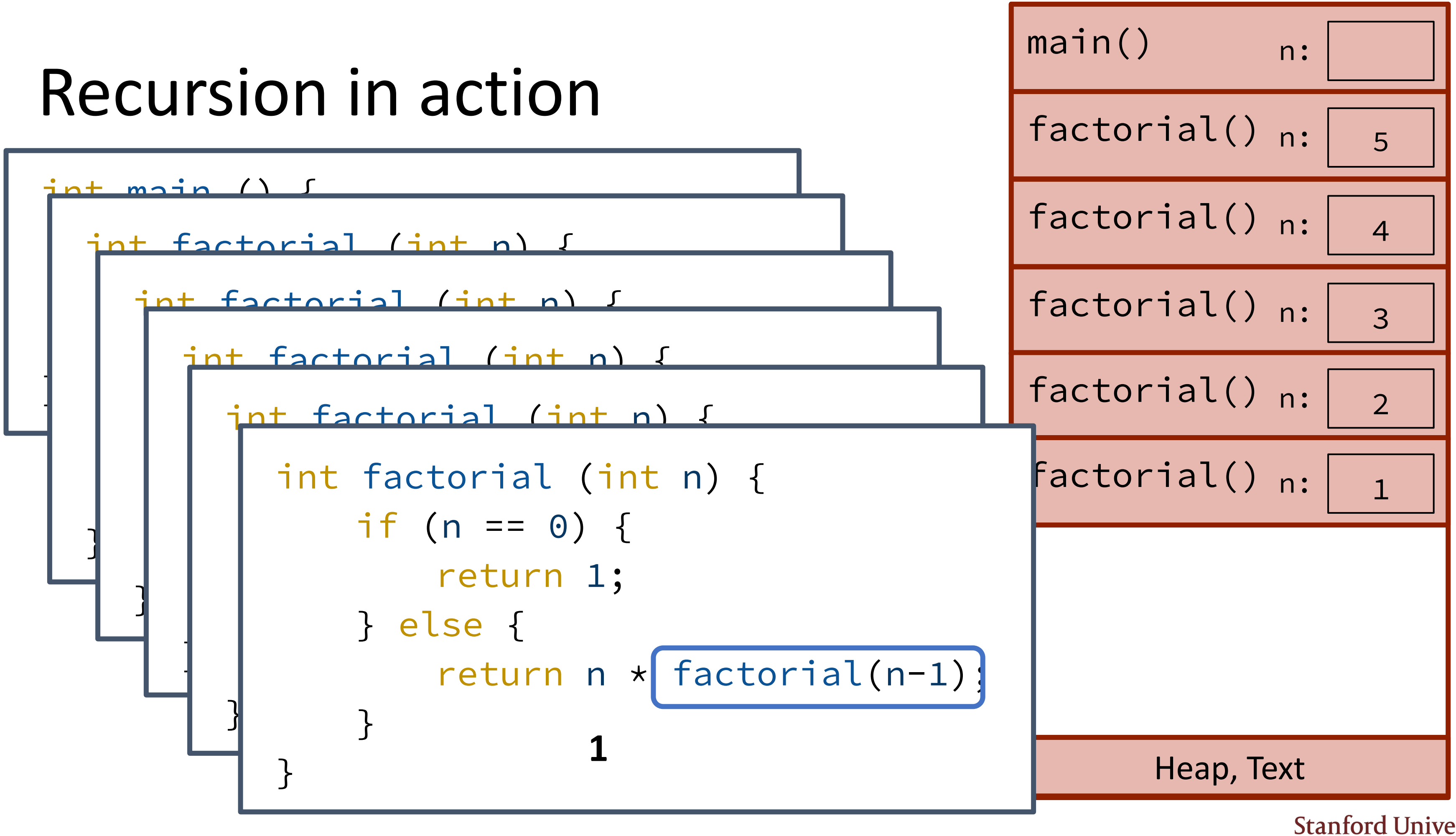
```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
                 1
}
```



main()	n:
factorial()	n: 5
factorial()	n: 4
factorial()	n: 3
factorial()	n: 2
factorial()	n: 1

Heap, Text

Recursion in action



```
int main () {
```

```
int factorial (int n) {
```

```
int factorial (int n) {
```

```
int factorial (int n) {
```

```
int factorial (int n) {
```

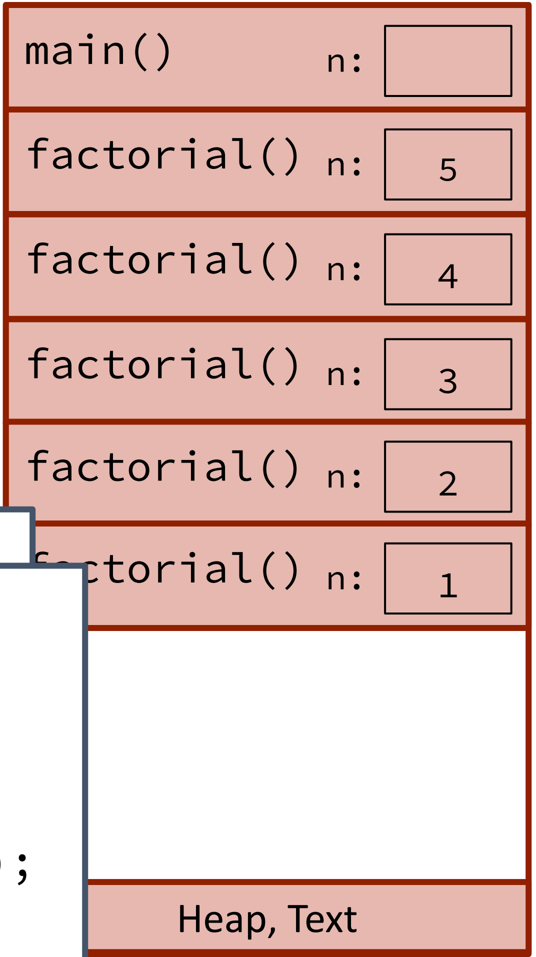
```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```

main()	n:
factorial()	n: 5
factorial()	n: 4
factorial()	n: 3
factorial()	n: 2
factorial()	n: 1
Heap, Text	

Recursion in action

```
int main () {
int factorial (int n) {
int factorial (int n) {
int factorial (int n) {
int factorial (int n) {
int factorial (int n) {
```

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



main() n:

factorial() n: 5

factorial() n: 4

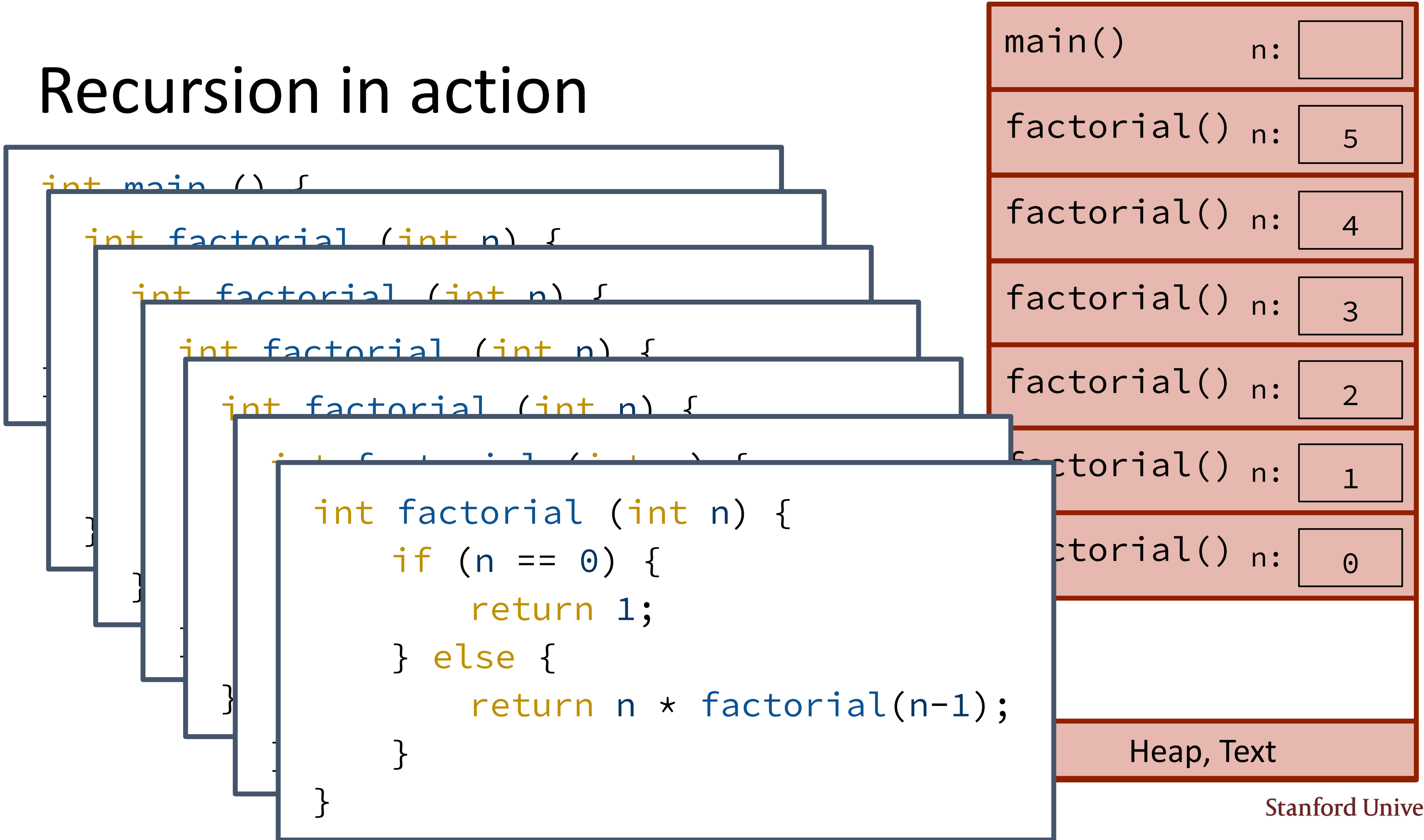
factorial() n: 3

factorial() n: 2

factorial() n: 1

Heap, Text

Recursion in action



```
int main () {
```

```
int factorial (int n) {
```

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

main() n:

factorial() n: 5

factorial() n: 4

factorial() n: 3

factorial() n: 2

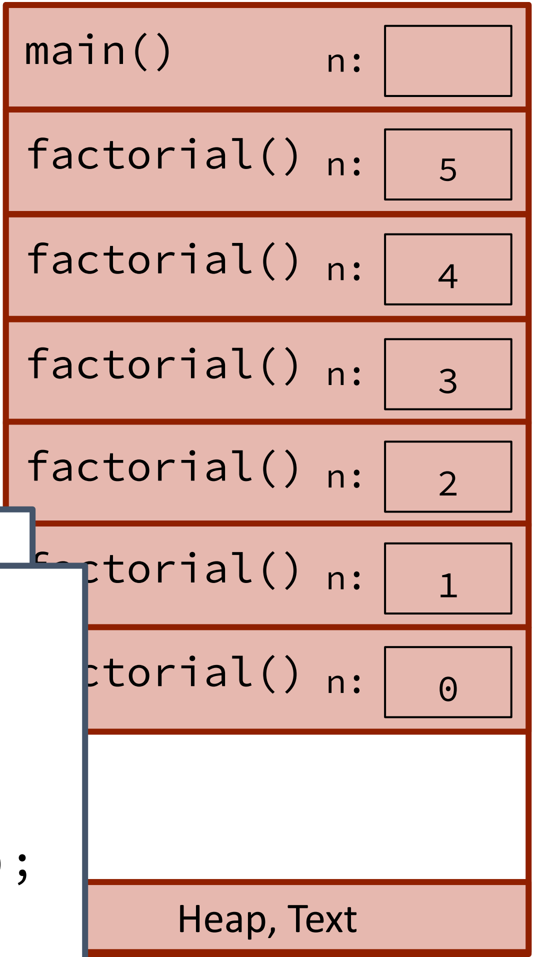
factorial() n: 1

factorial() n: 0

Heap, Text

Recursion in action

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



main() n:

factorial() n: 5

factorial() n: 4

factorial() n: 3

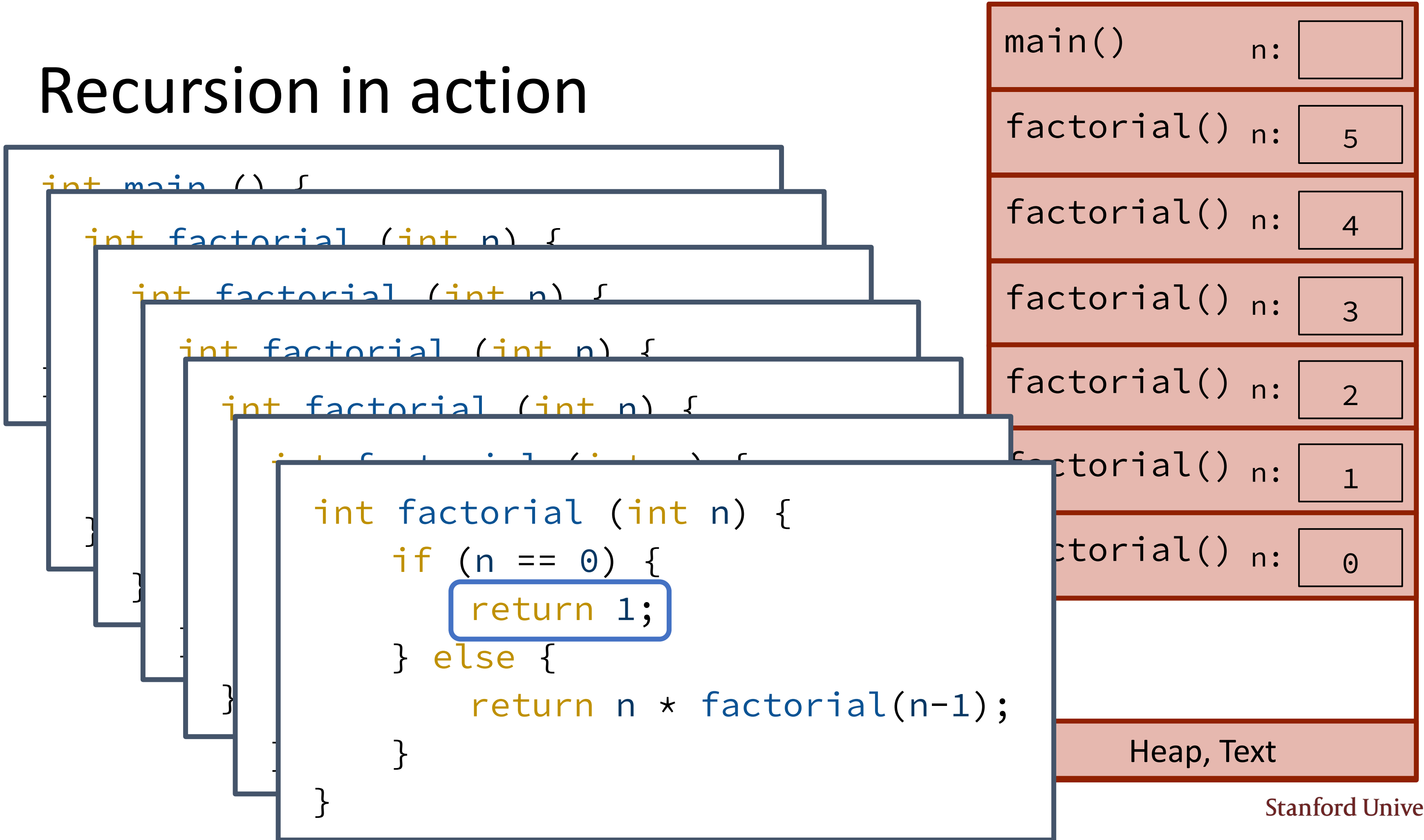
factorial() n: 2

factorial() n: 1

factorial() n: 0

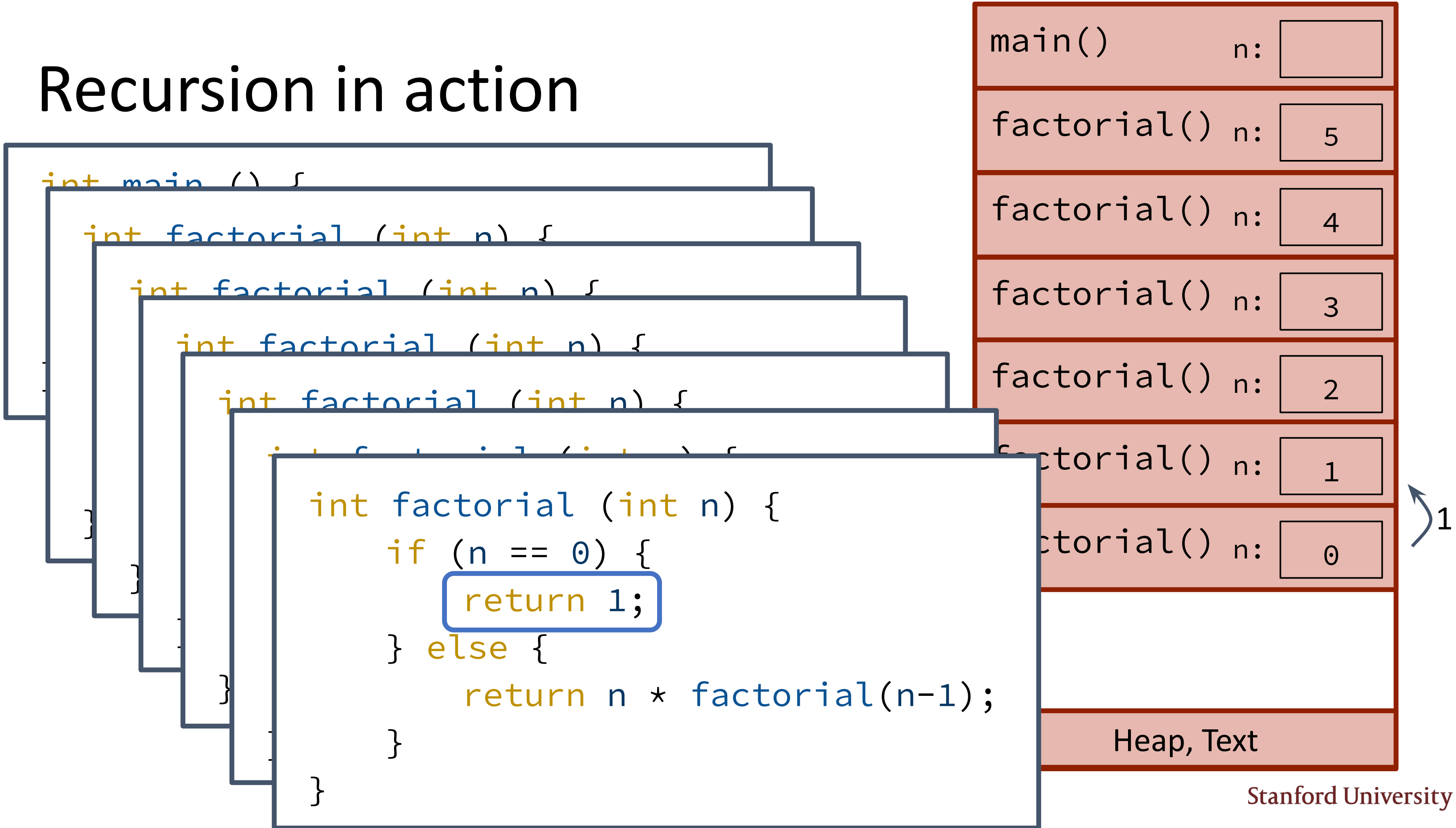
Heap, Text

Recursion in action



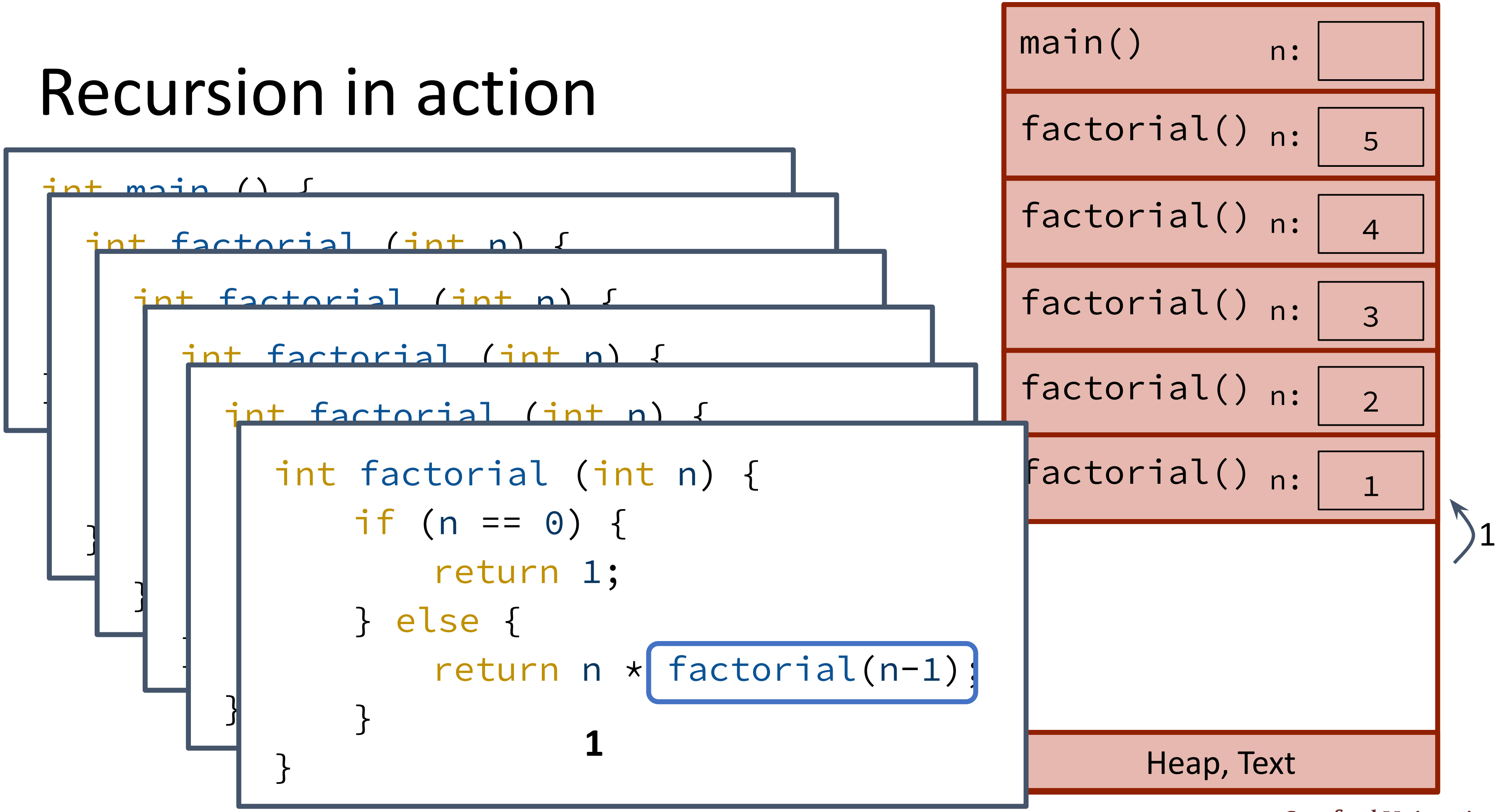
```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```

Recursion in action



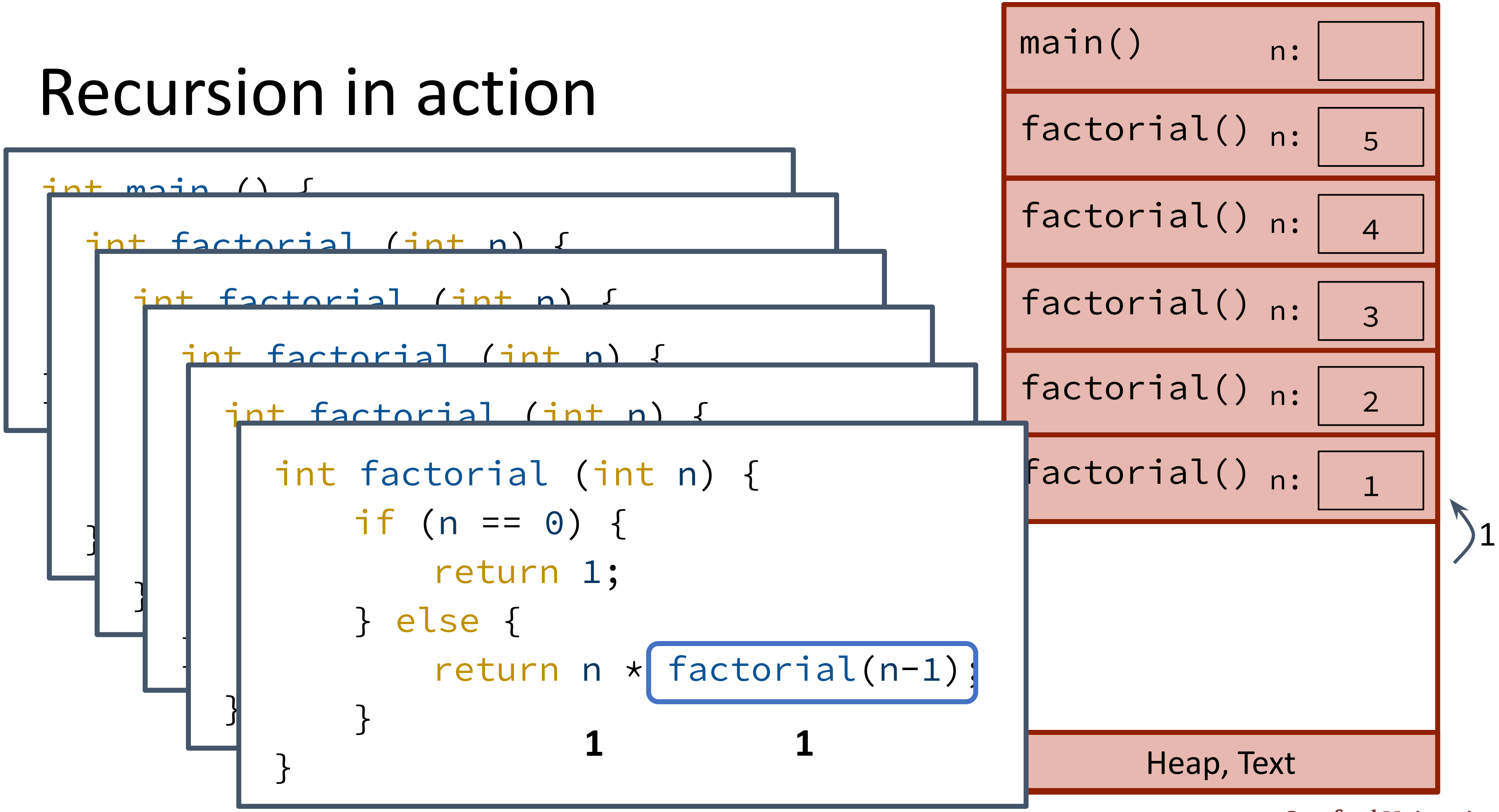
```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```

Recursion in action



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

Recursion in action



```
int main () {
```

```
int factorial (int n) {
```

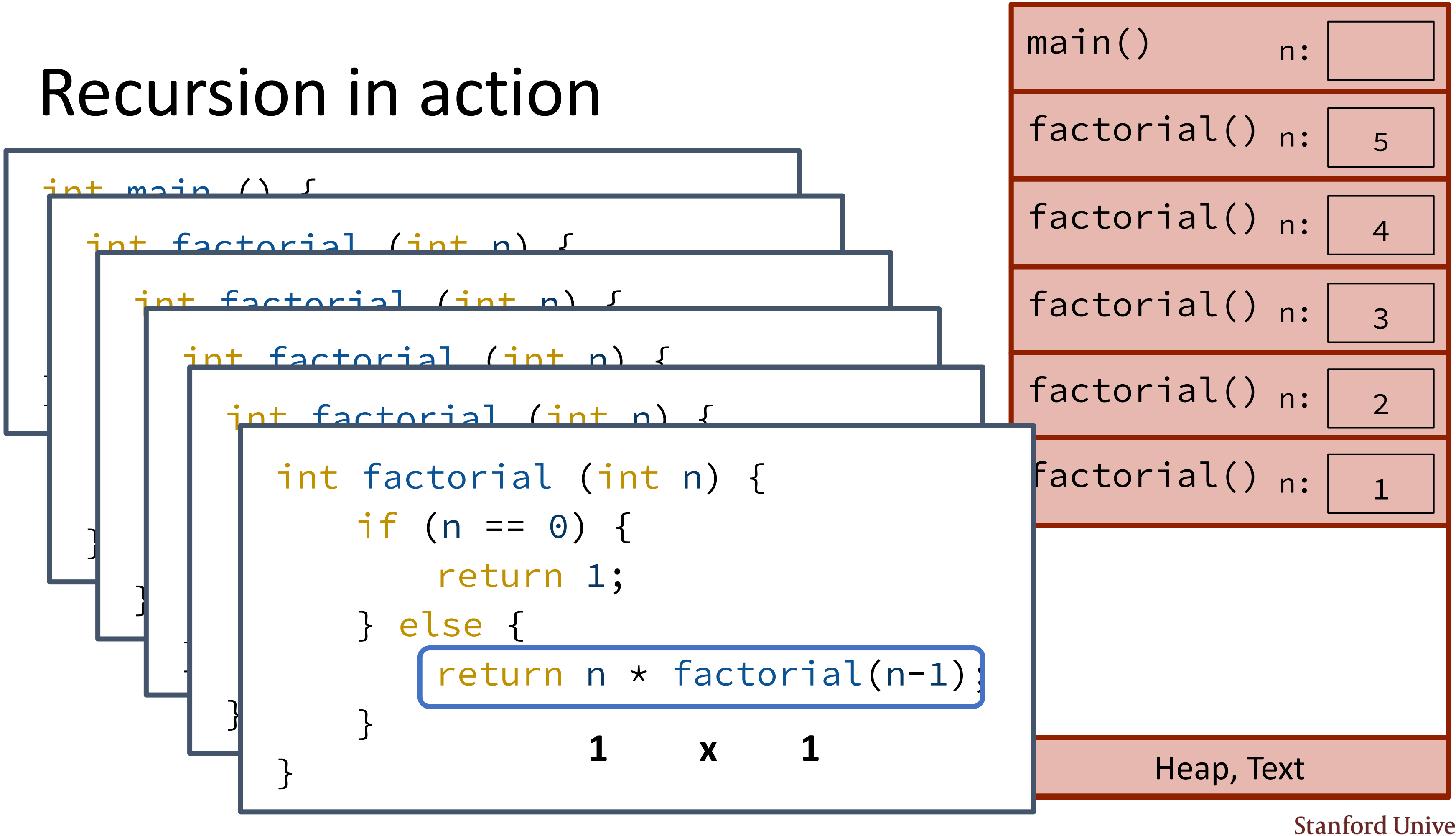
```
int factorial (int n) {
```

```
int factorial (int n) {
```

```
int factorial (int n) {
```

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

Recursion in action



```
int main () {
```

```
int factorial (int n) {
```

```
int factorial (int n) {
```

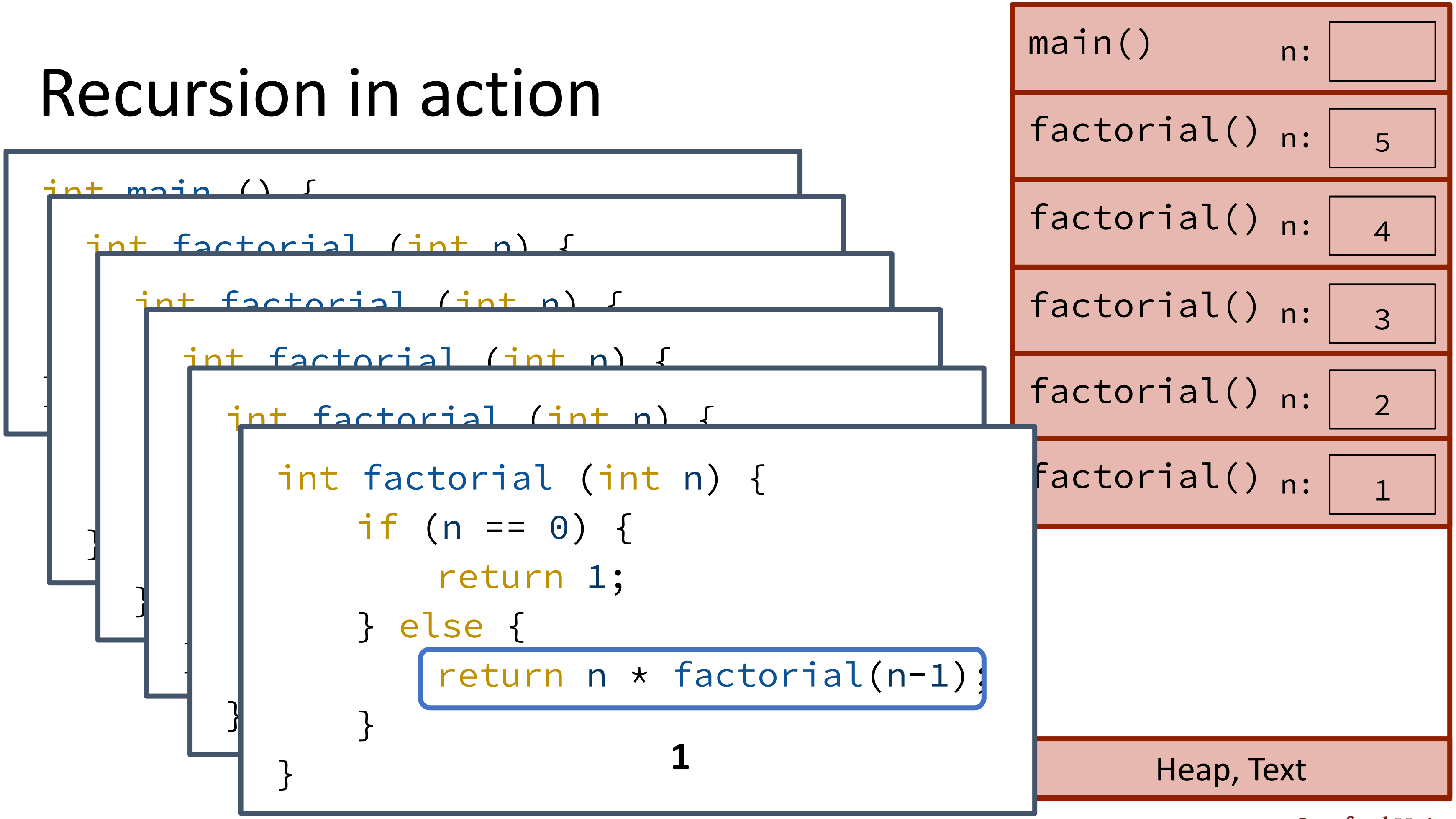
```
int factorial (int n) {
```

```
int factorial (int n) {
```

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

1 x 1

Recursion in action



```
int main () {
```

```
int factorial (int n) {
```

```
int factorial (int n) {
```

```
int factorial (int n) {
```

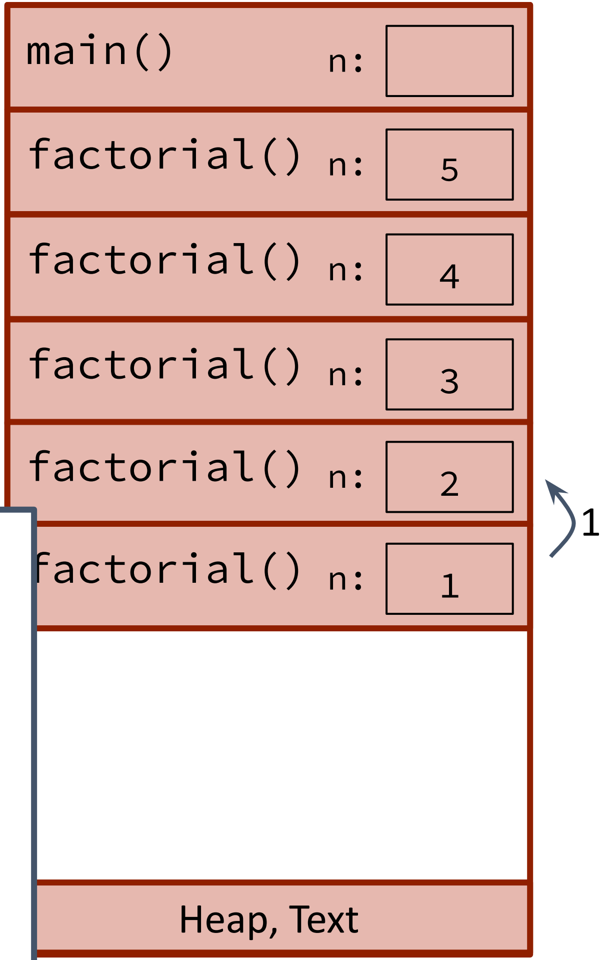
```
int factorial (int n) {
```

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

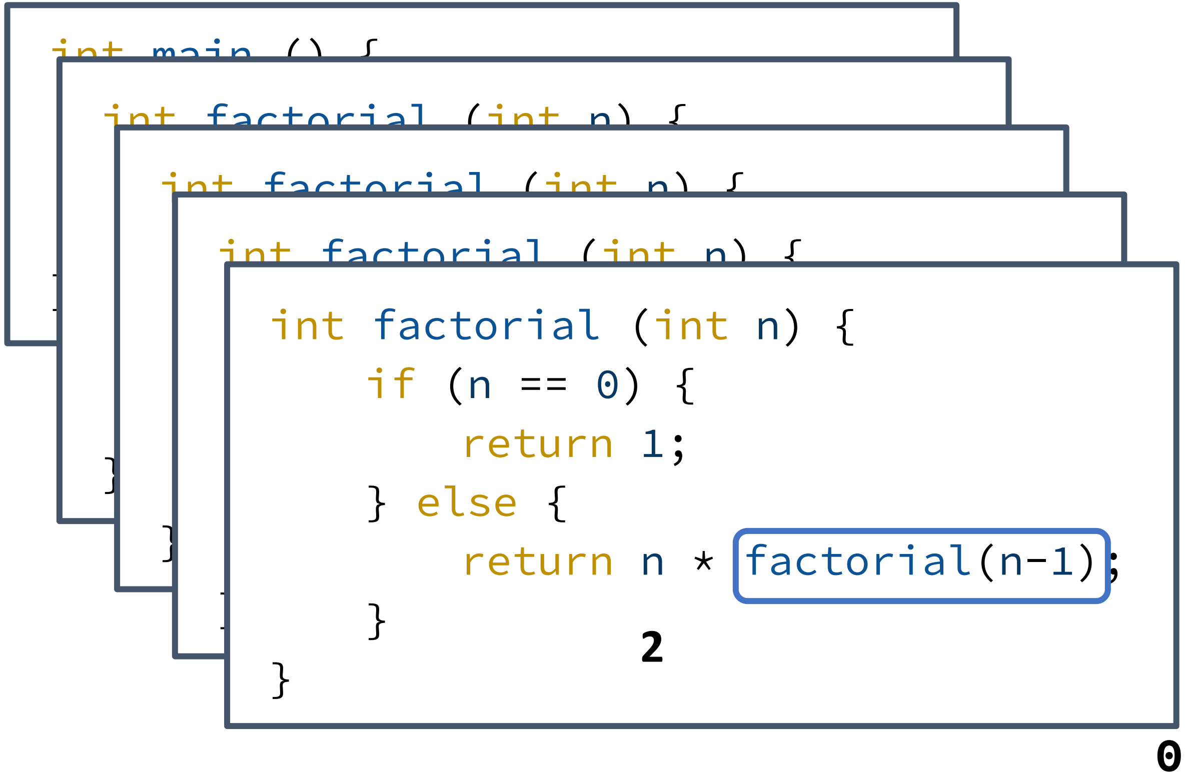
Recursion in action

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

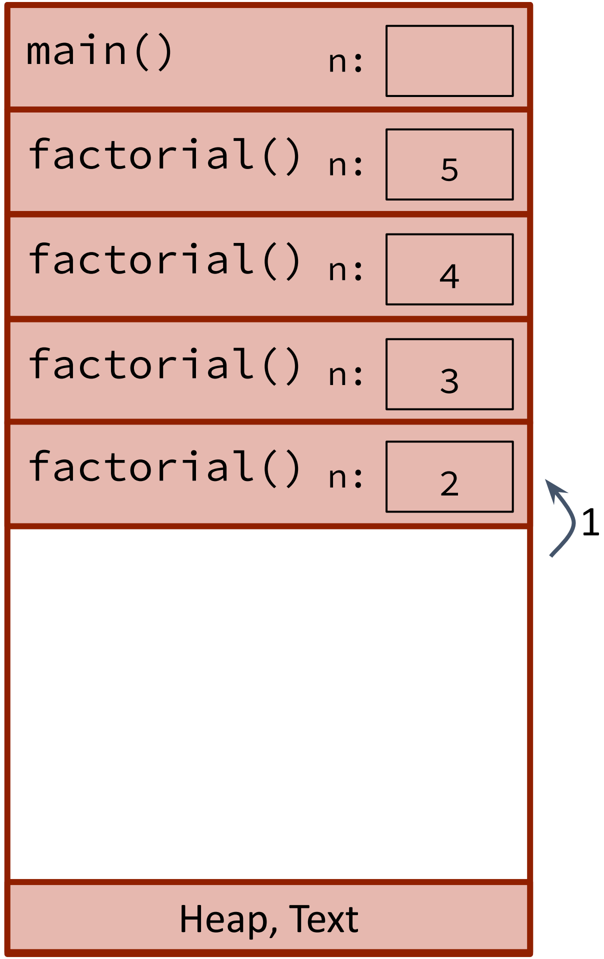
1



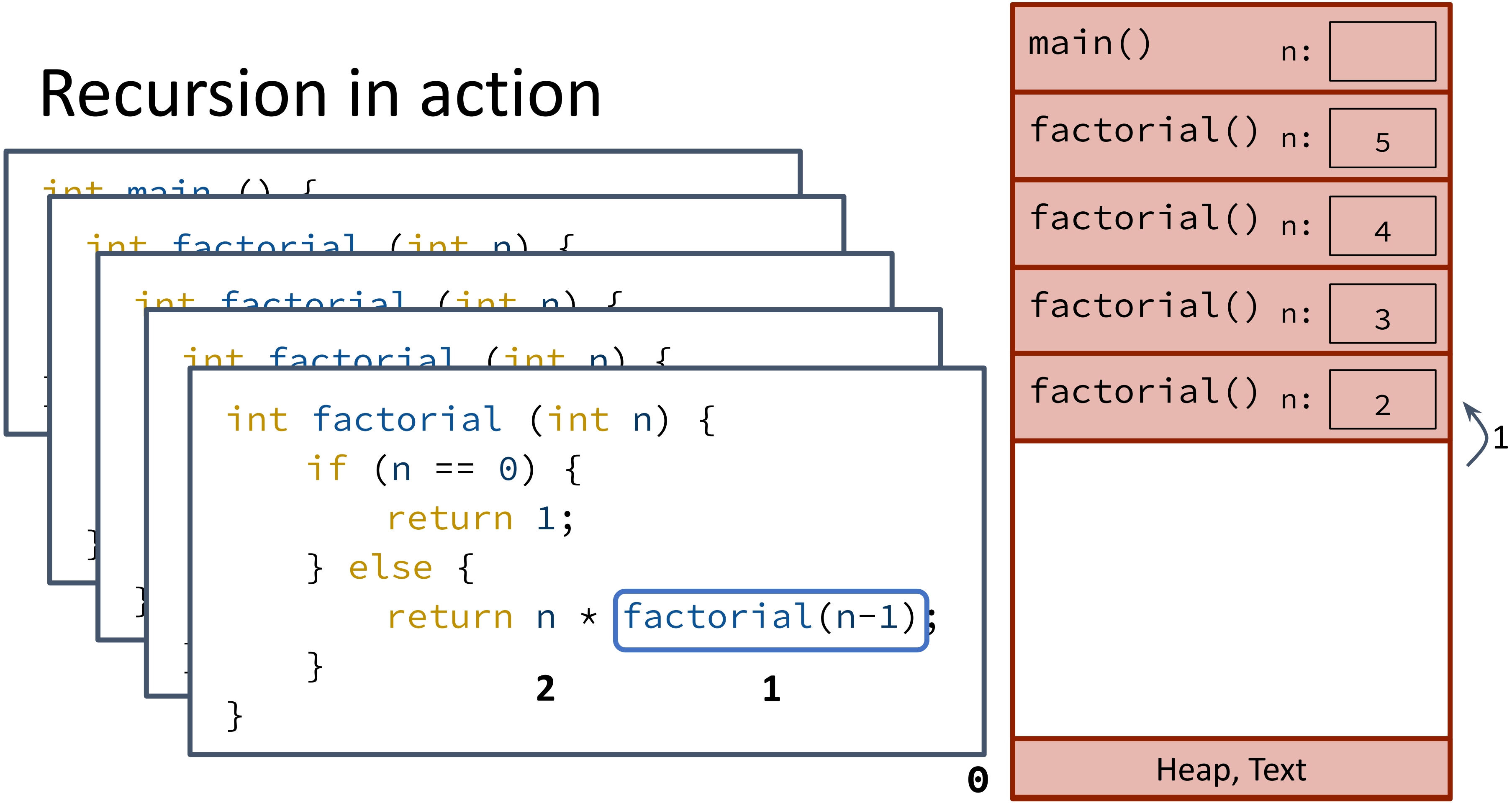
Recursion in action



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

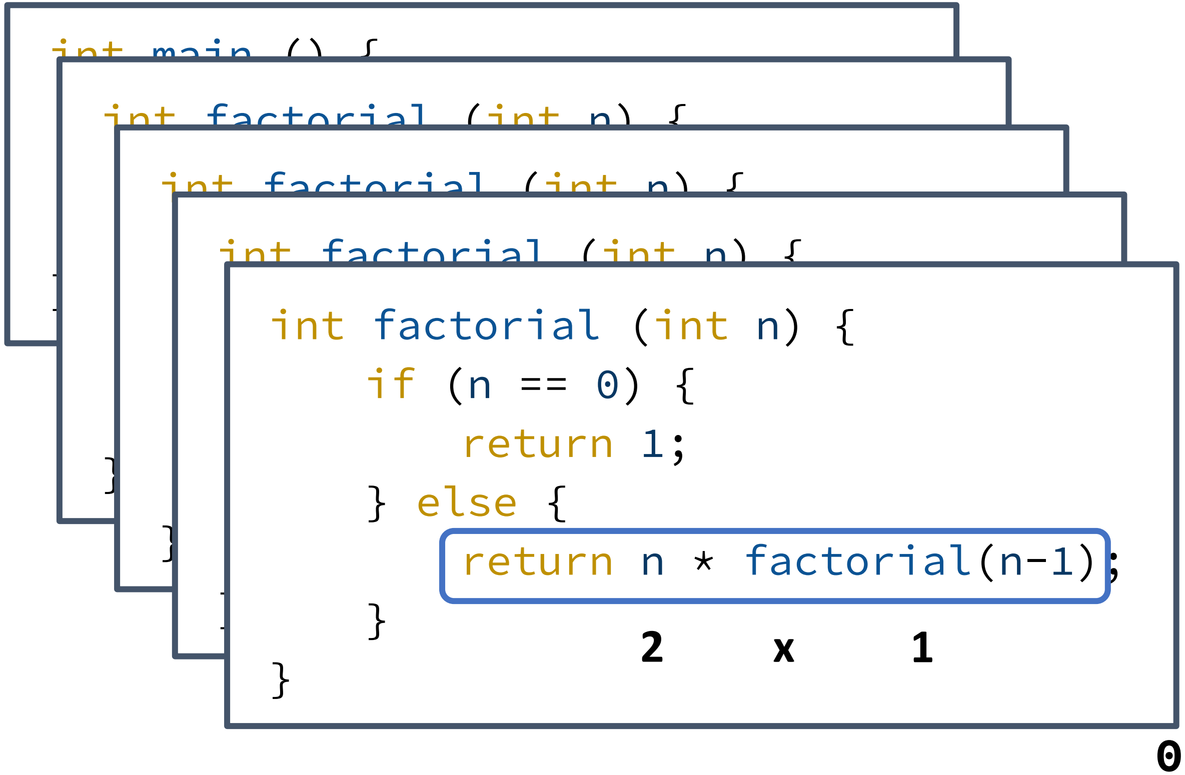


Recursion in action



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

Recursion in action



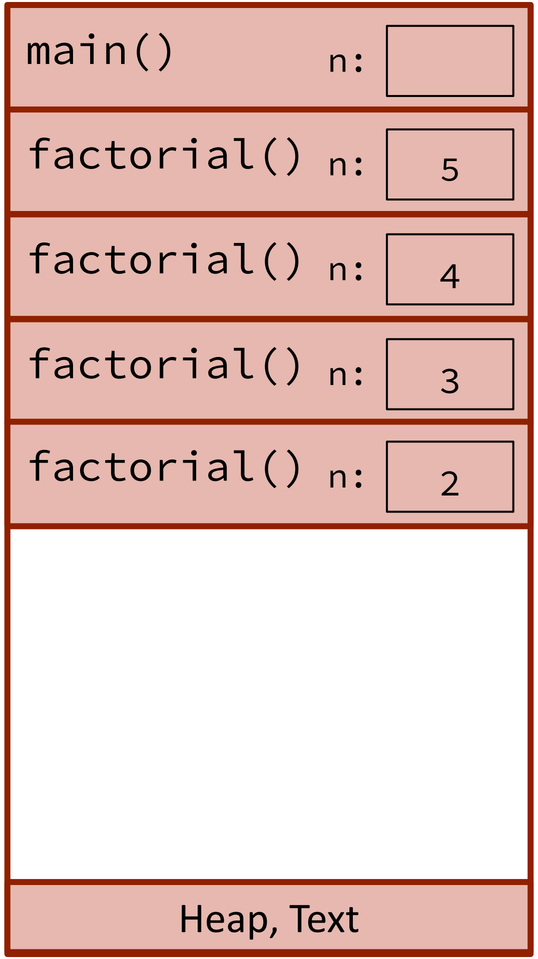
```
int main () {
```

```
int factorial (int n) {
```

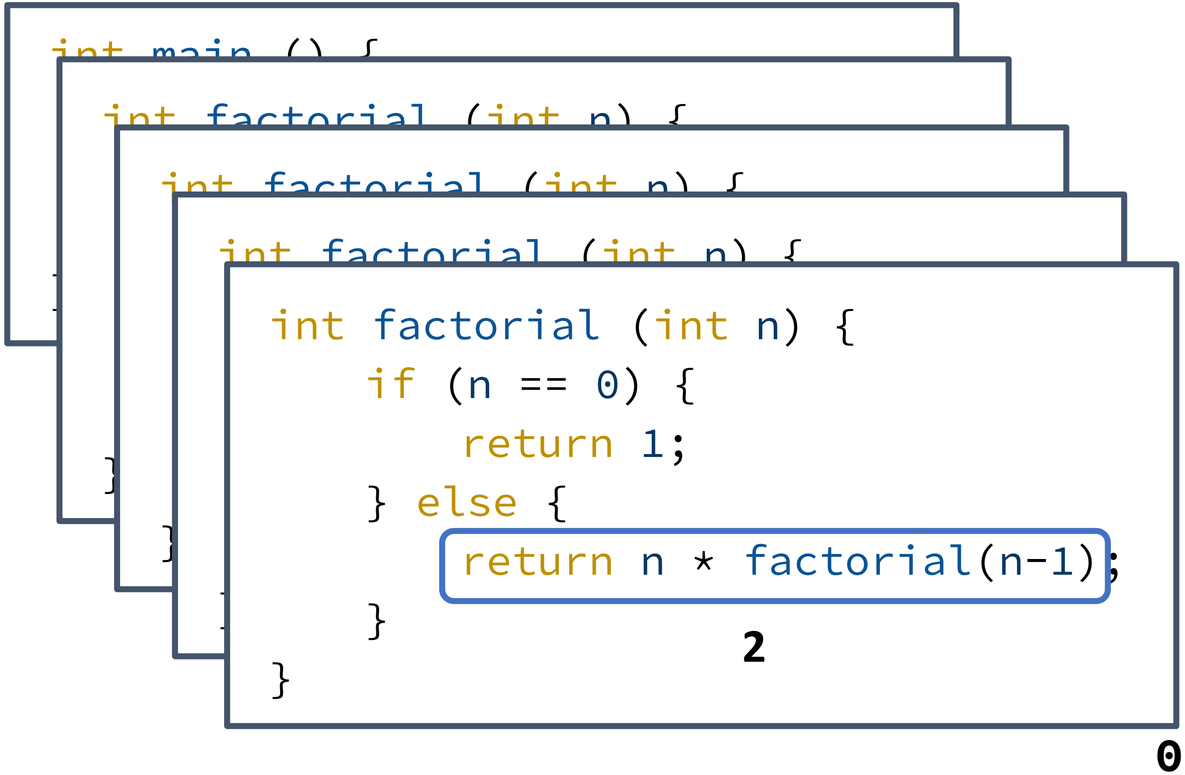
```
int factorial (int n) {
```

```
int factorial (int n) {
```

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



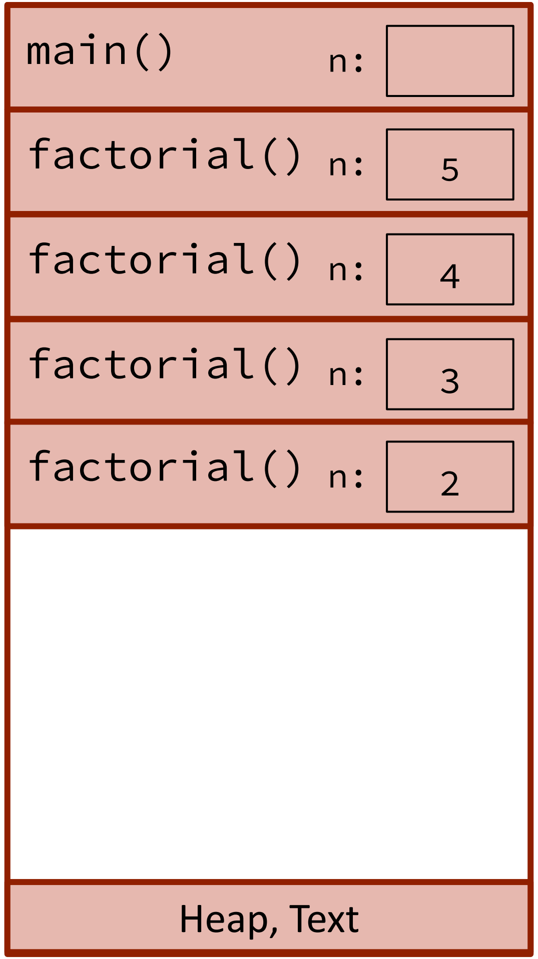
Recursion in action



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

2

0



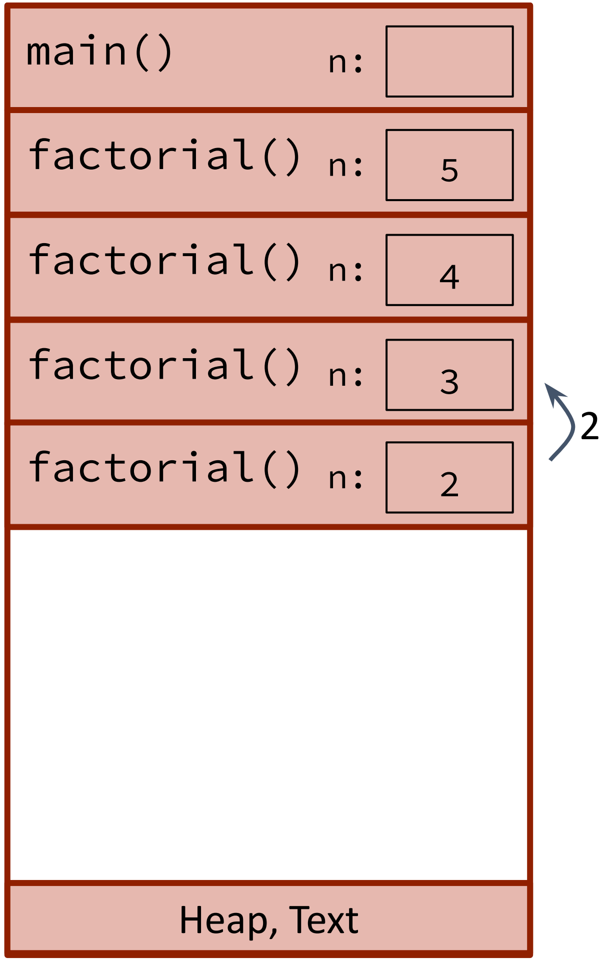
Recursion in action

```
int main () {
  int factorial (int n) {
    int factorial (int n) {
      int factorial (int n) {
```

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

2

0



main() n:

factorial() n: 5

factorial() n: 4

factorial() n: 3

factorial() n: 2

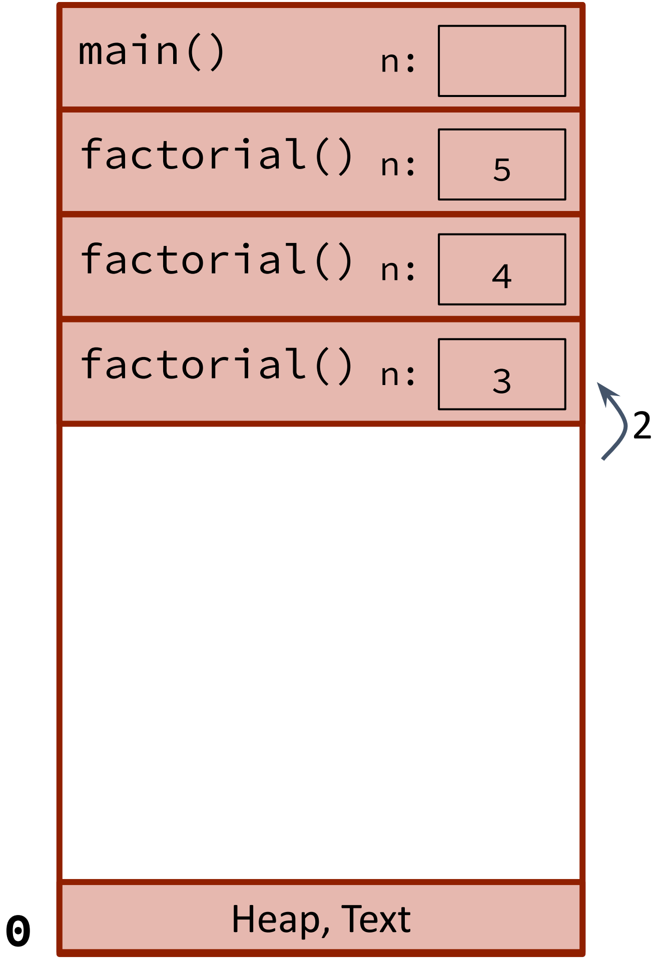
2

Heap, Text

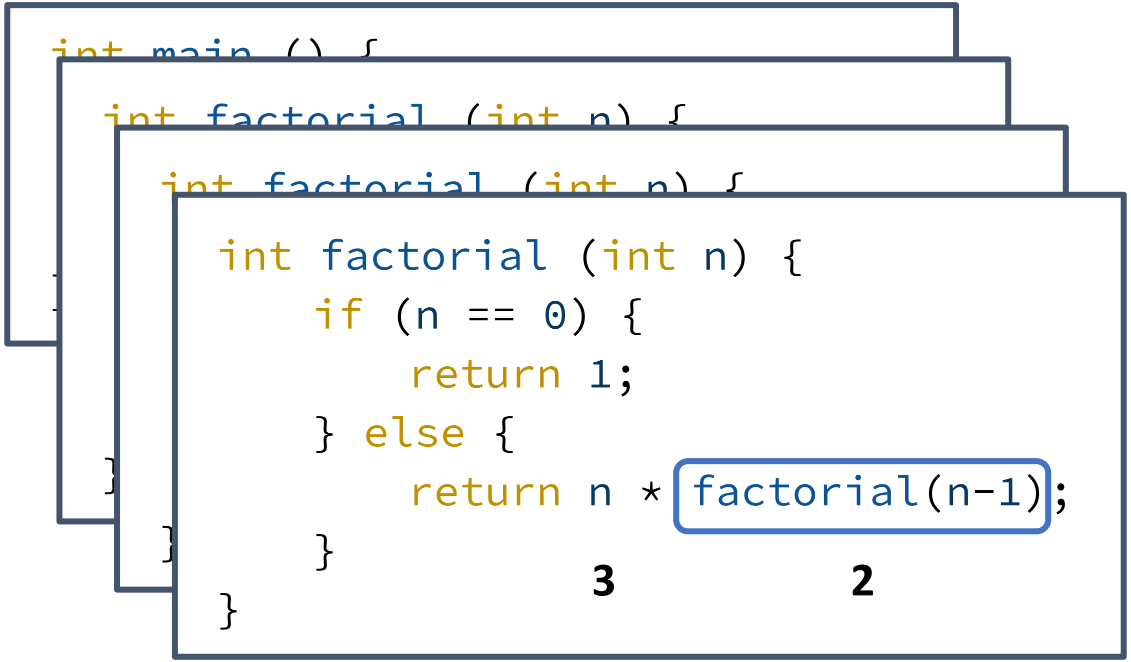
Recursion in action

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

3



Recursion in action

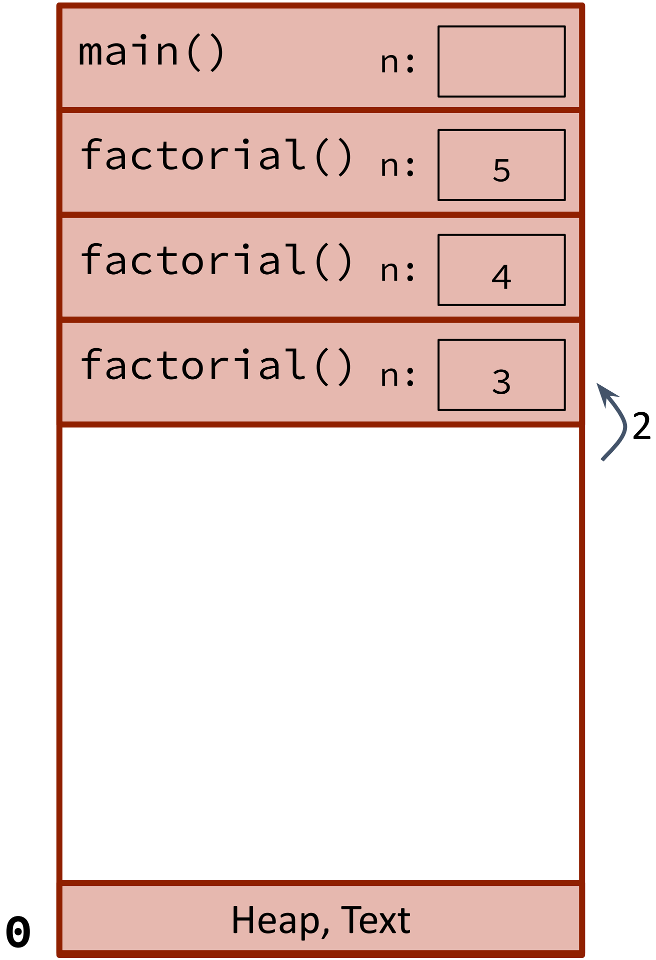


```
int main () {
```

```
int factorial (int n) {
```

```
int factorial (int n) {
```

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



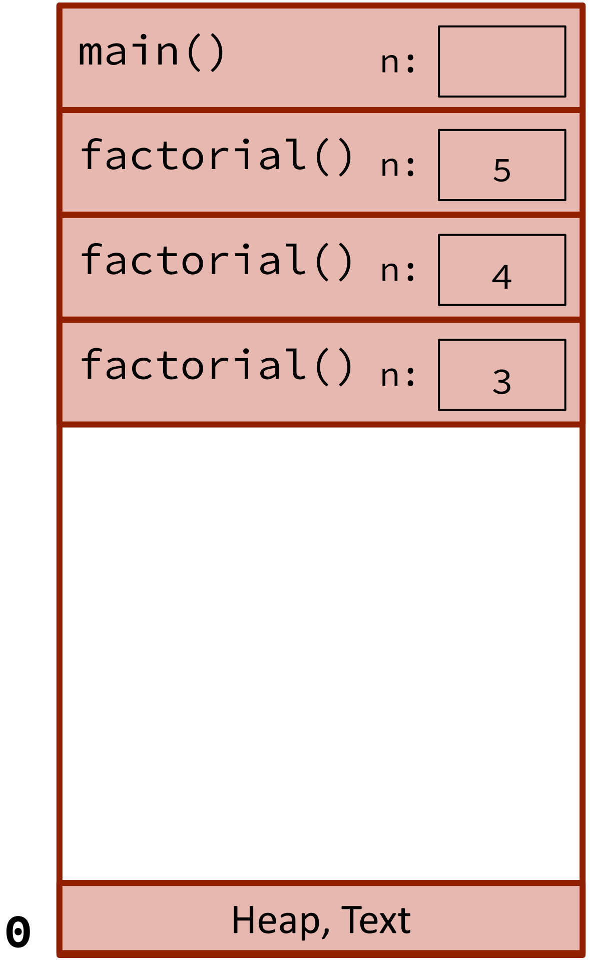
main()	n:
factorial()	n: 5
factorial()	n: 4
factorial()	n: 3

Recursion in action

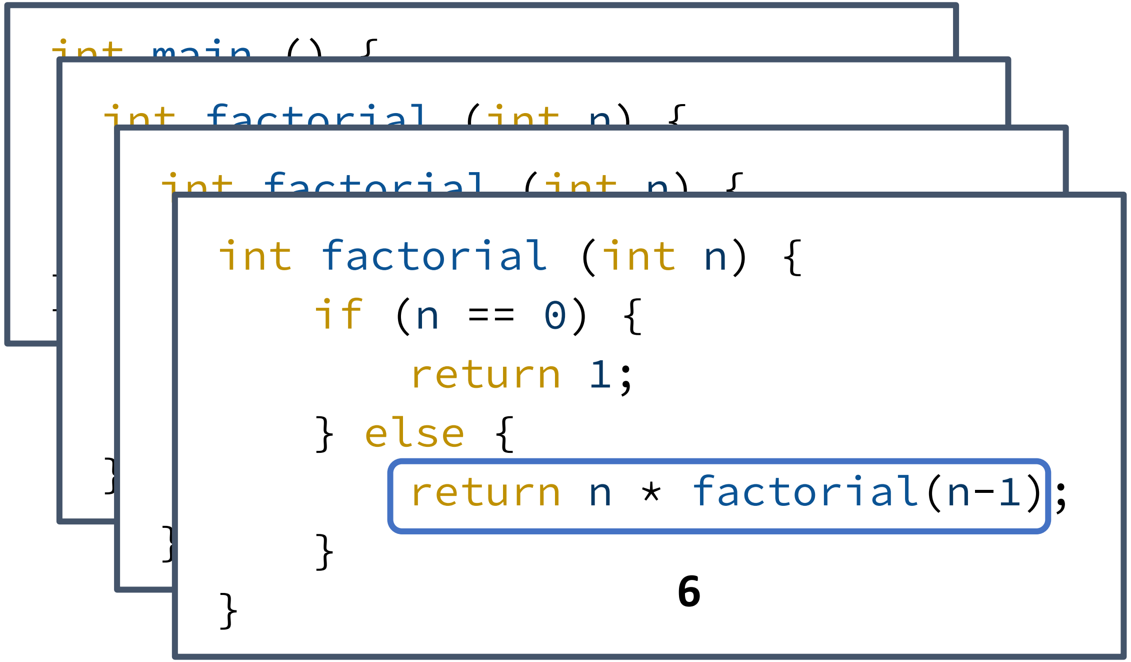
```
int main () {
  int factorial (int n) {
    int factorial (int n) {
```

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

3 x 2

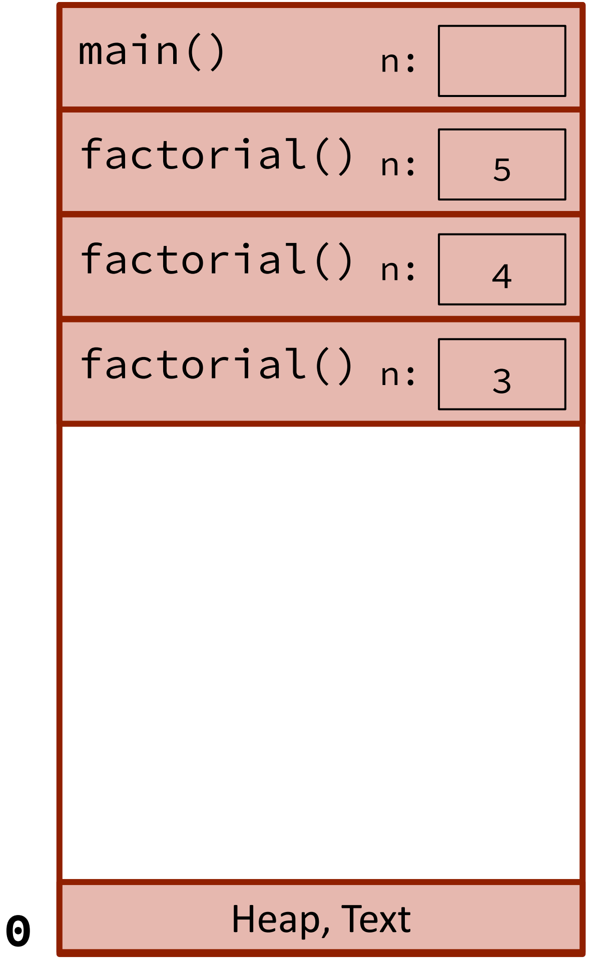


Recursion in action



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

6



main() n:

factorial() n: 5

factorial() n: 4

factorial() n: 3

Heap, Text

0

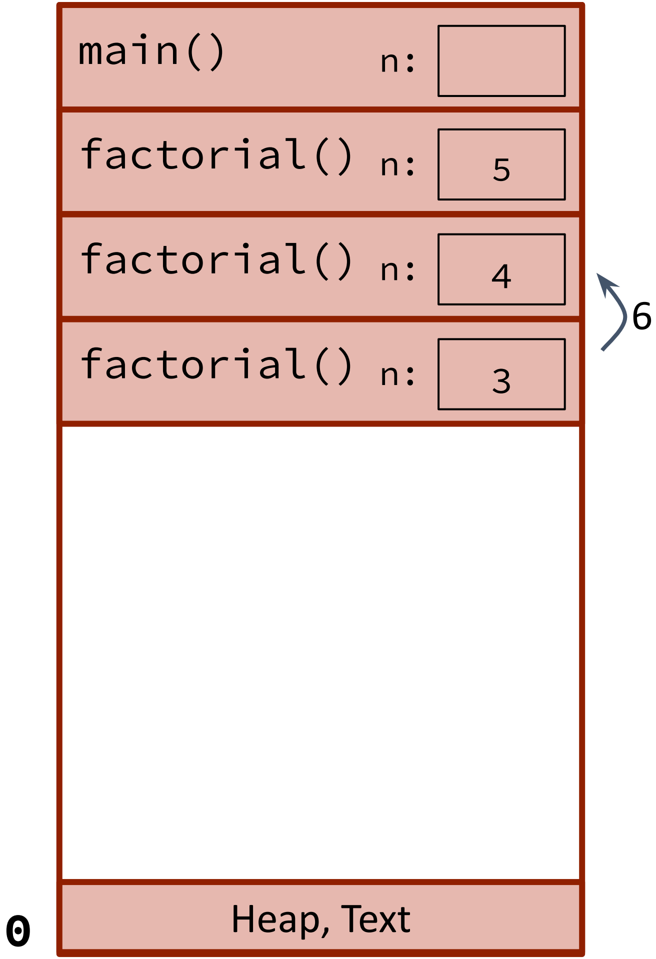
Recursion in action

```
int main () {
```

```
int factorial (int n) {
```

```
int factorial (int n) {
```

```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
                    6
}
```



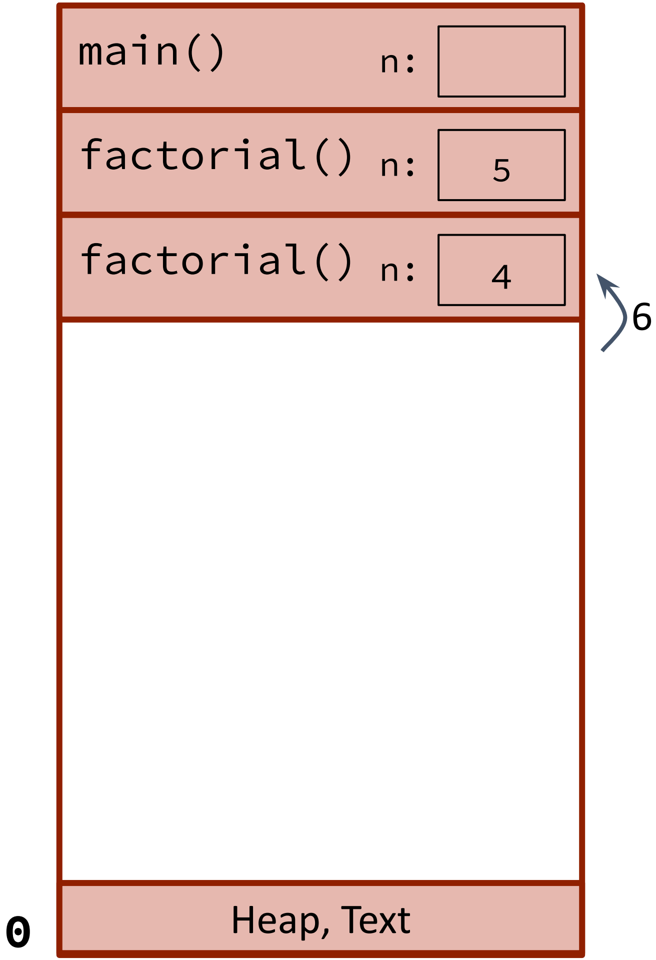
Recursion in action

```
int main () {
```

```
int factorial (int n) {
```

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

4



Stanford University

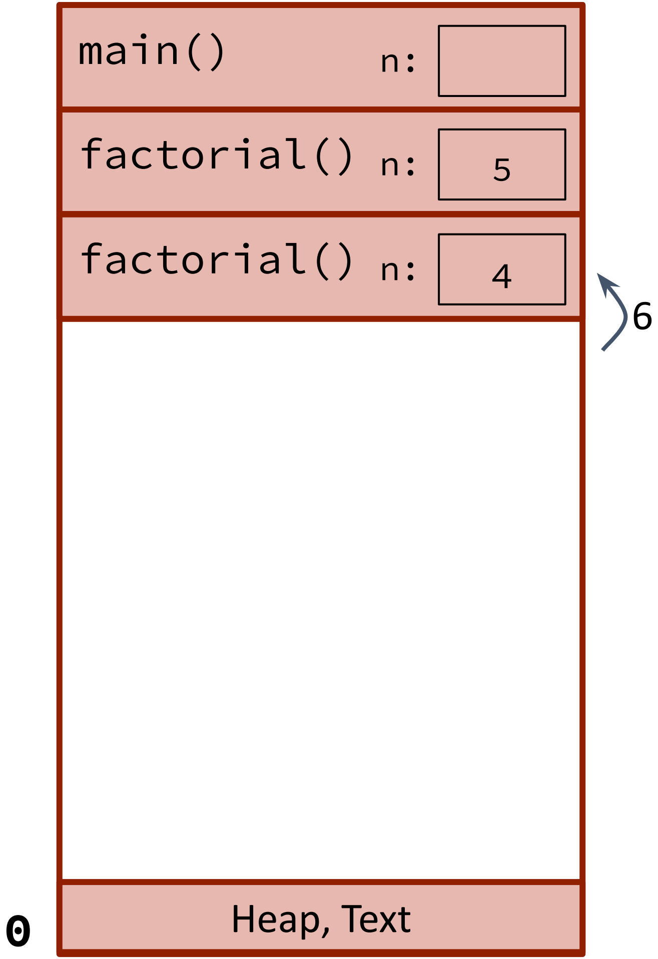
Recursion in action

```
int main () {
```

```
int factorial (int n) {
```

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

4 6

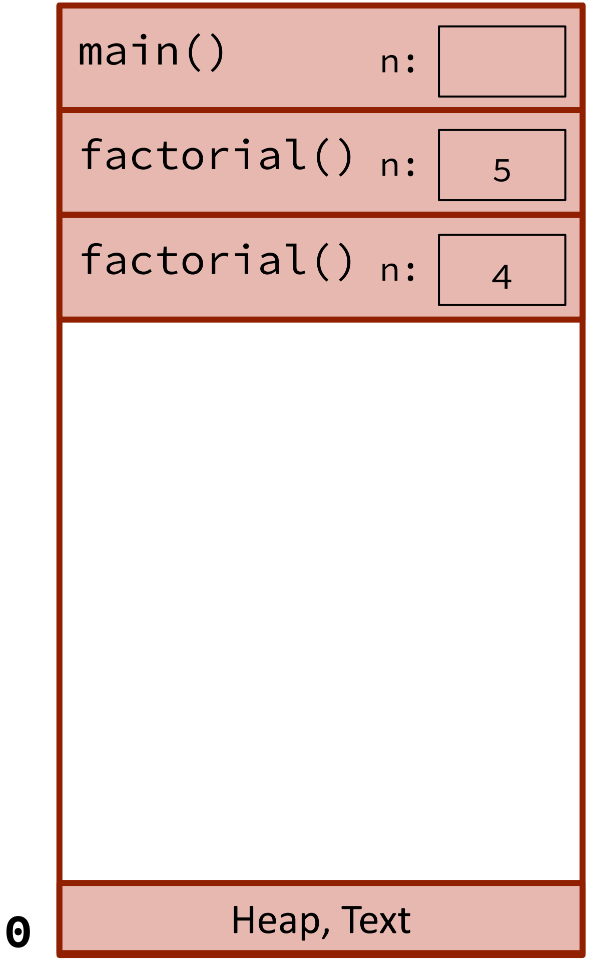


Recursion in action

```
int main () {
```

```
int factorial (int n) {
```

```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
                 4     x    6
}
```



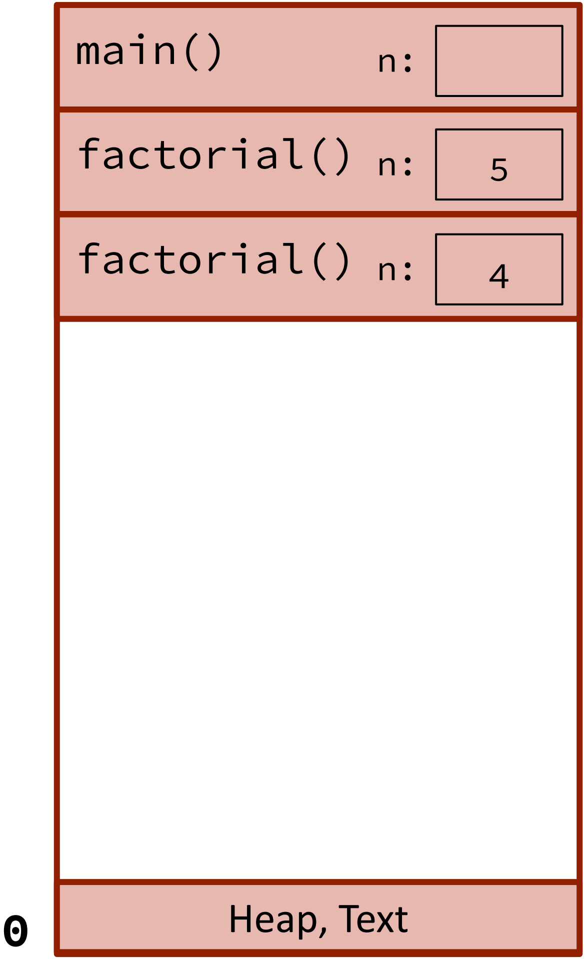
Recursion in action

```
int main () {
```

```
int factorial (int n) {
```

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

24


main() n:

factorial() n: 5

factorial() n: 4

0

Heap, Text

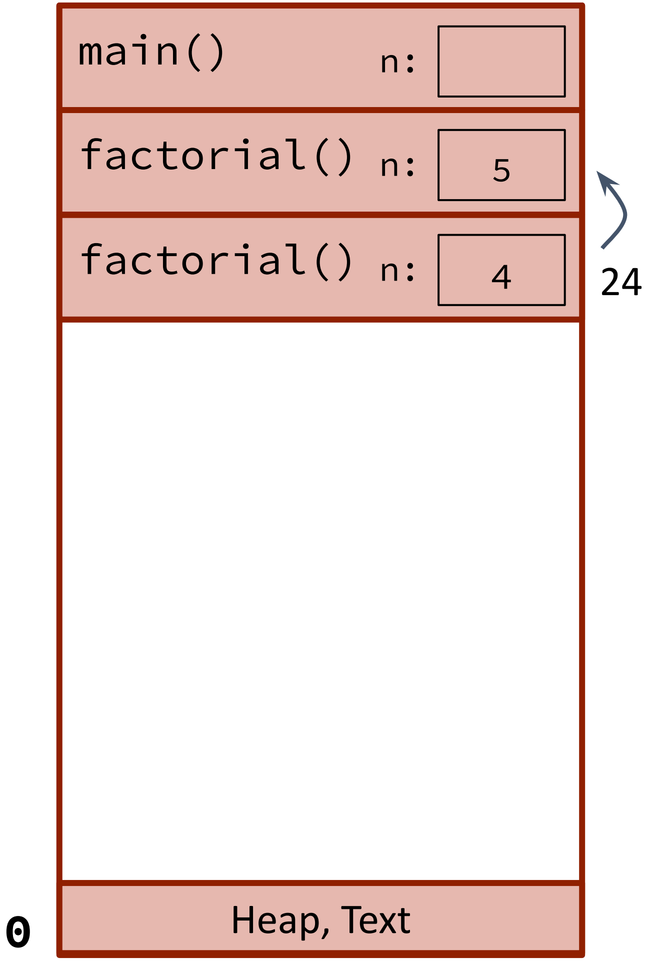
Recursion in action

```
int main () {
```

```
int factorial (int n) {
```

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

24



main() n:

factorial() n: 5

factorial() n: 4

24

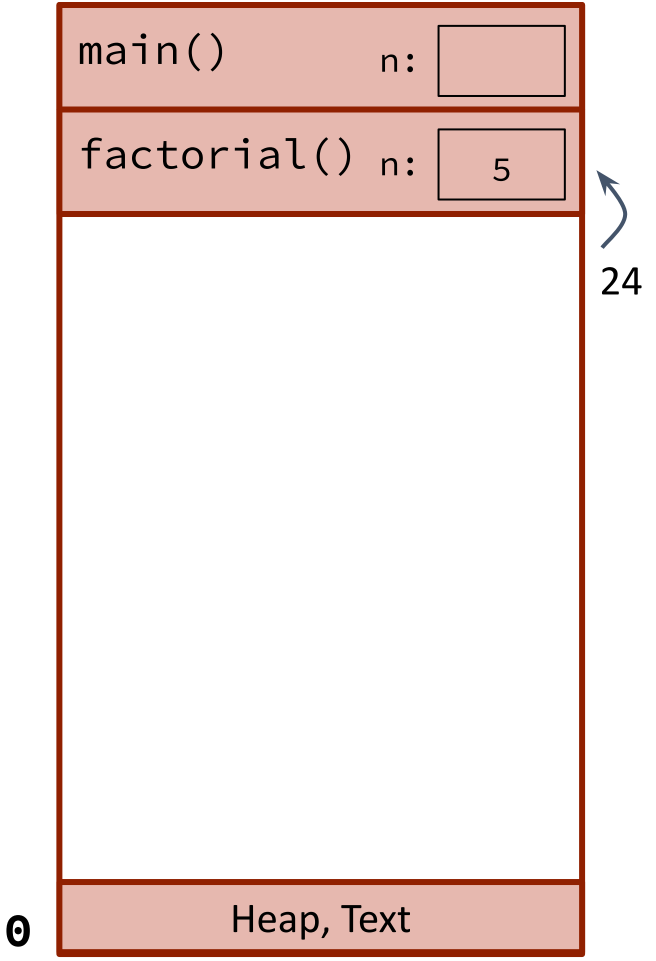
Heap, Text

0

Recursion in action

```
int main () {
```

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

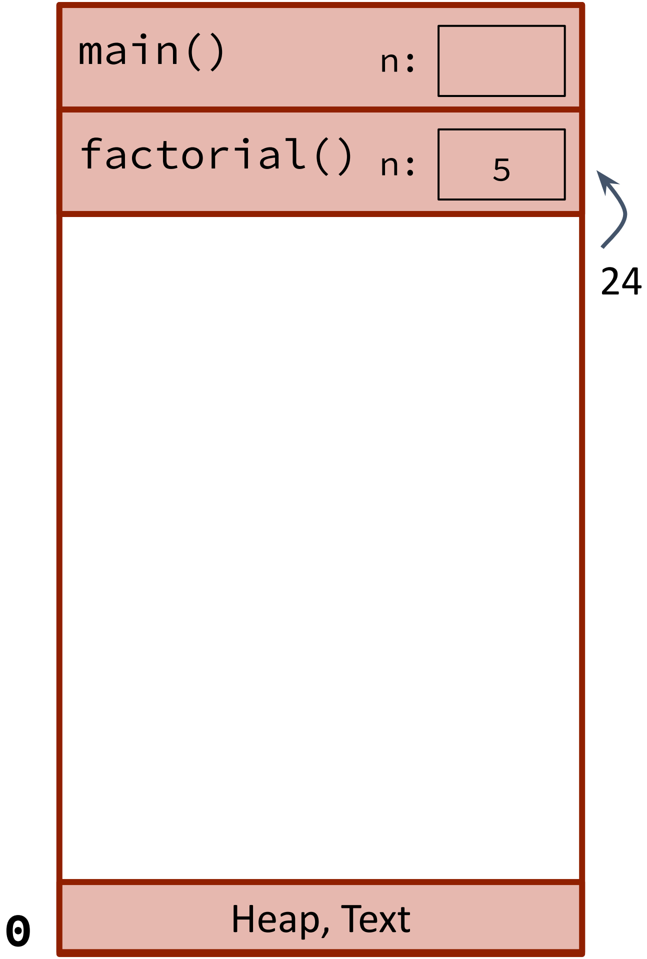


Recursion in action

```
int main () {
```

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

5 **24**

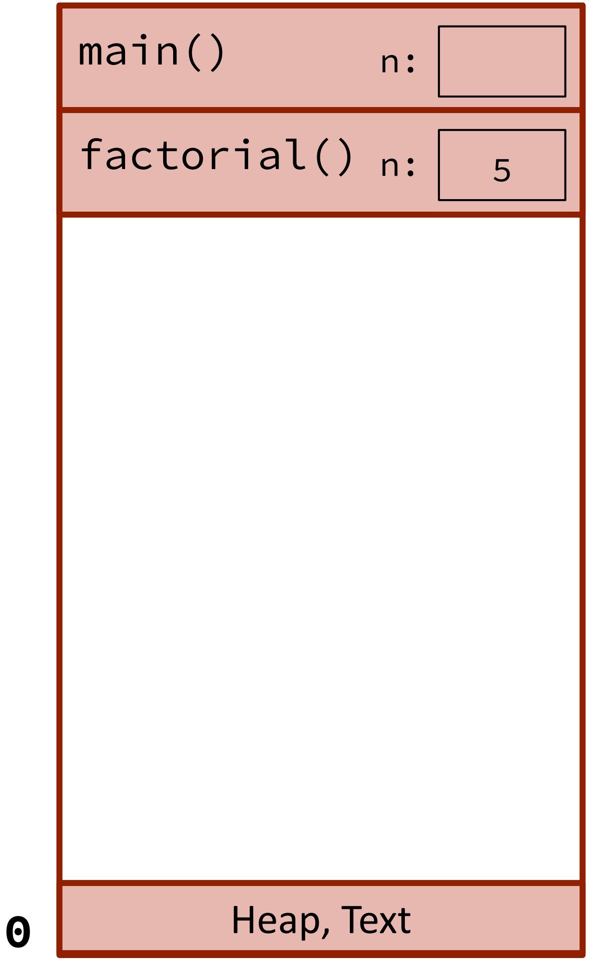


Recursion in action

```
int main () {
```

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

5 x 24

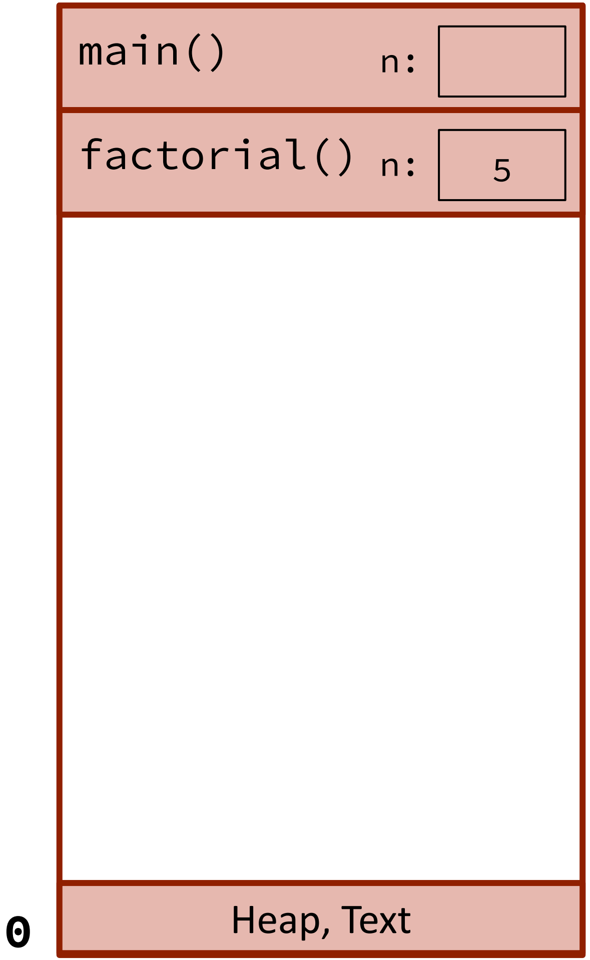


Recursion in action

```
int main () {
```

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

120



main() n:

factorial() n: 5

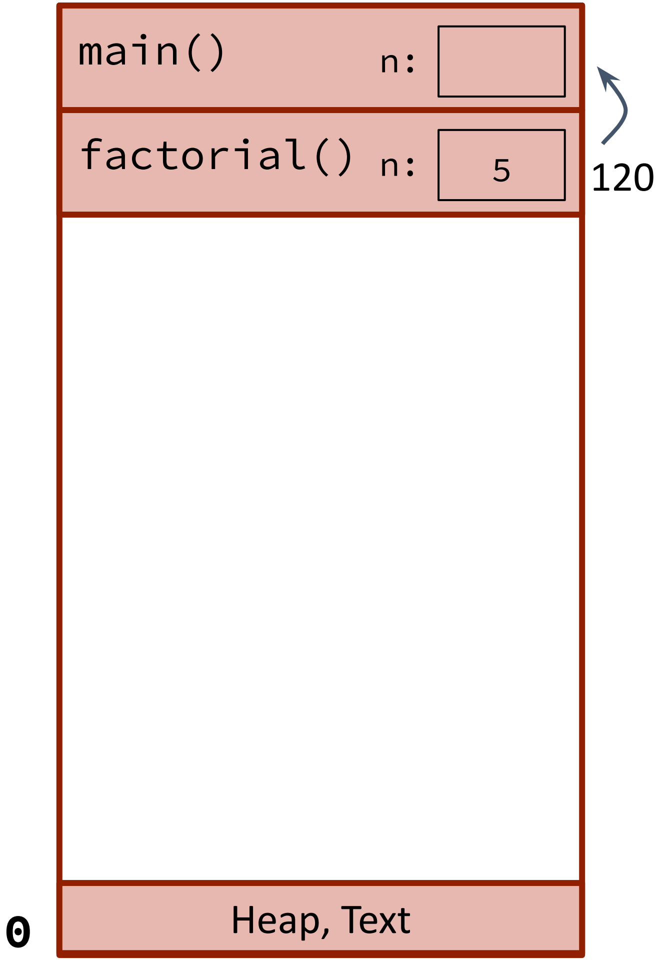
Heap, Text

0

Recursion in action

```
int main () {
```

```
int factorial (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
                  120
}
```



main() n:

factorial() n: 5

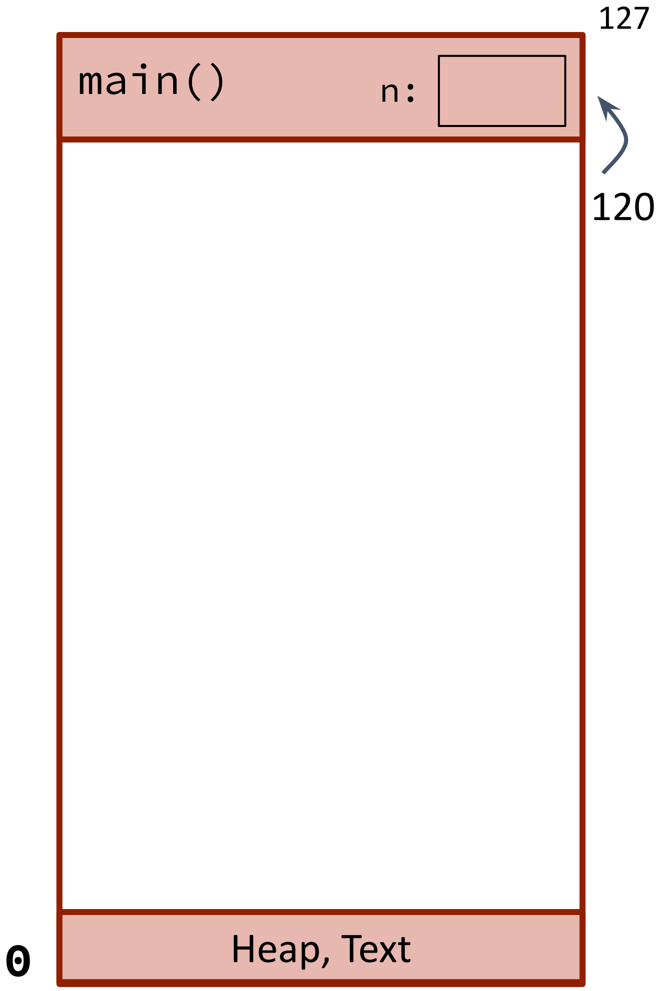
120

0

Heap, Text

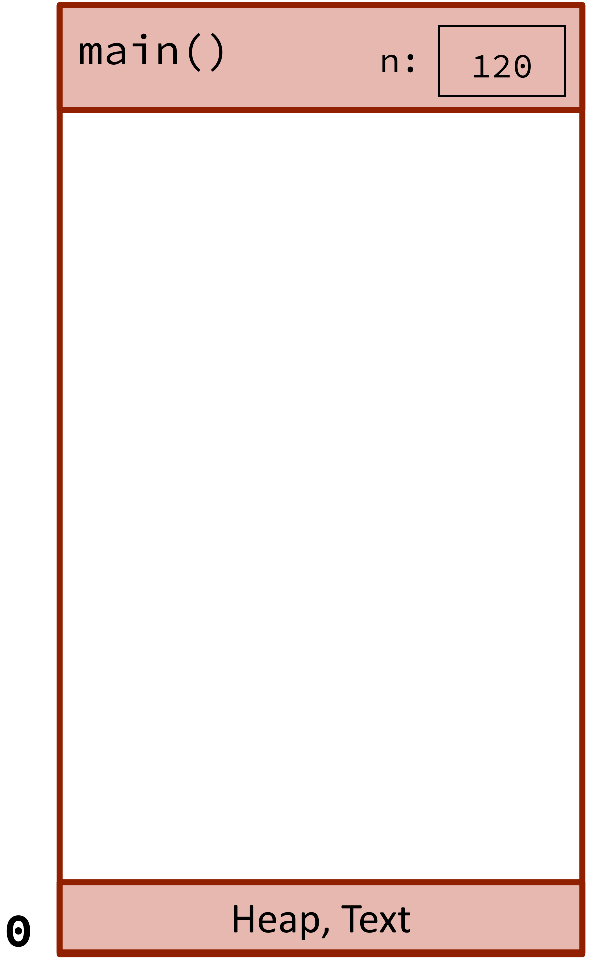
Recursion in action

```
int main () {
    int n = factorial(5);
    cout << “5! = ” << n << endl;
    return 0;
}
```



Recursion in action

```
int main () {
    int n = factorial(5);
    cout << “5! = ” << n << endl;
    return 0;
}
```



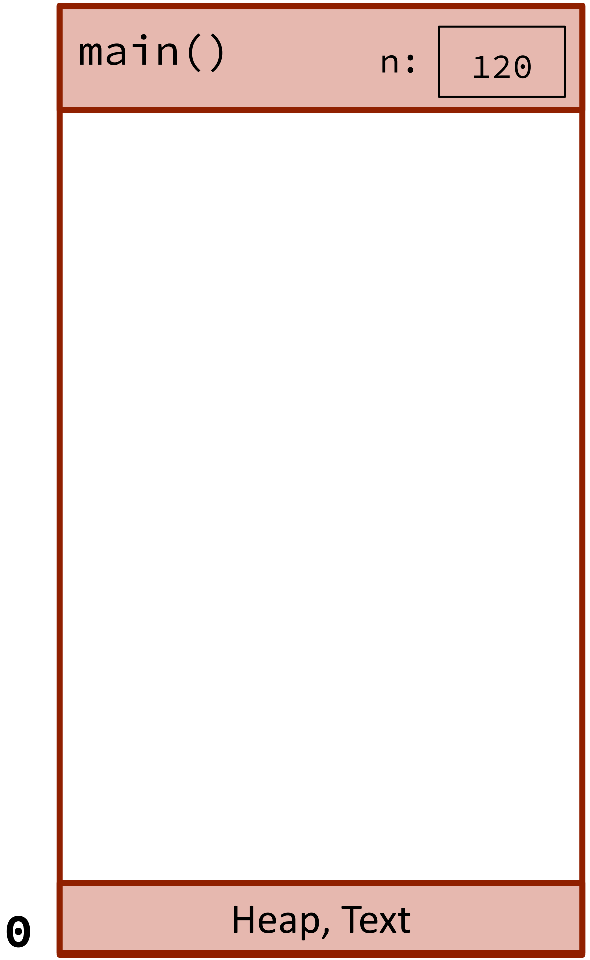
main() n: 120

Heap, Text

0

Recursion in action

```
int main () {
    int n = factorial(5);
    cout << “5! = ” << n << endl;
    return 0;
}
```



main() n: 120

Heap, Text

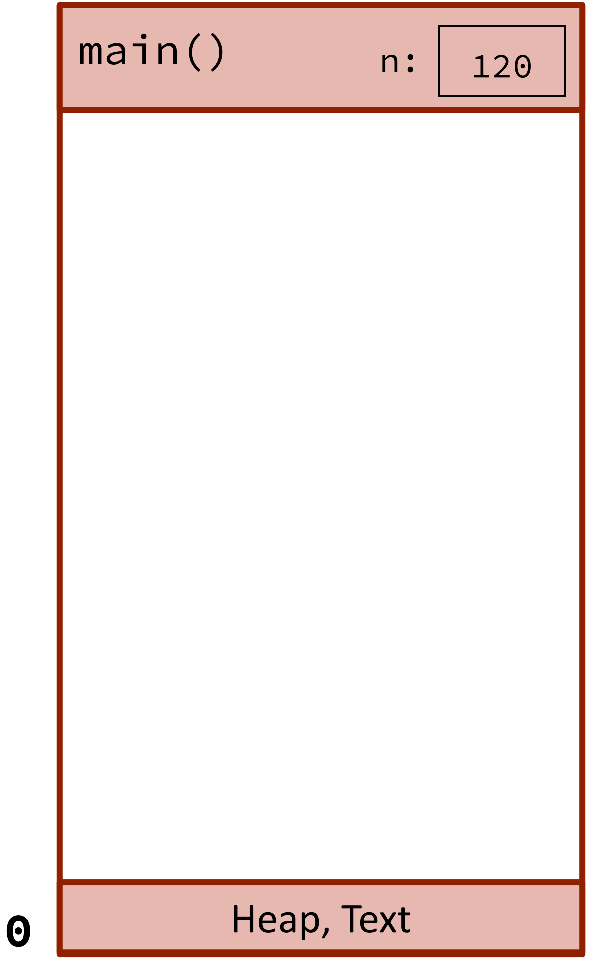
0

Recursion in action

```
int main () {
    int n = factorial(5);
    cout << “5! = ” << n << endl;
    return 0;
}
```

Console:

```
5! = 120
```

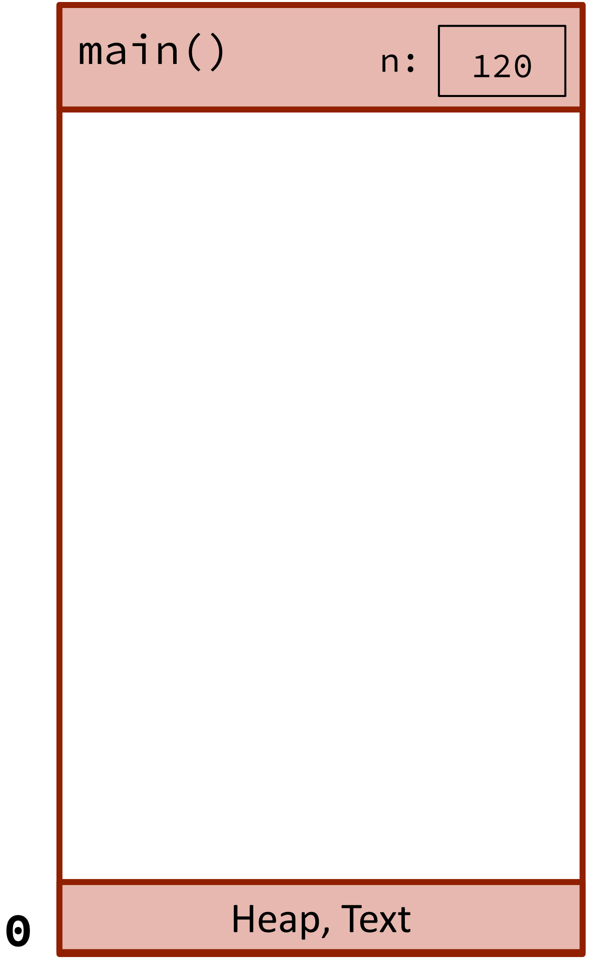


Recursion in action

```
int main () {
    int n = factorial(5);
    cout << “5! = ” << n << endl;
    return 0;
}
```

Console:

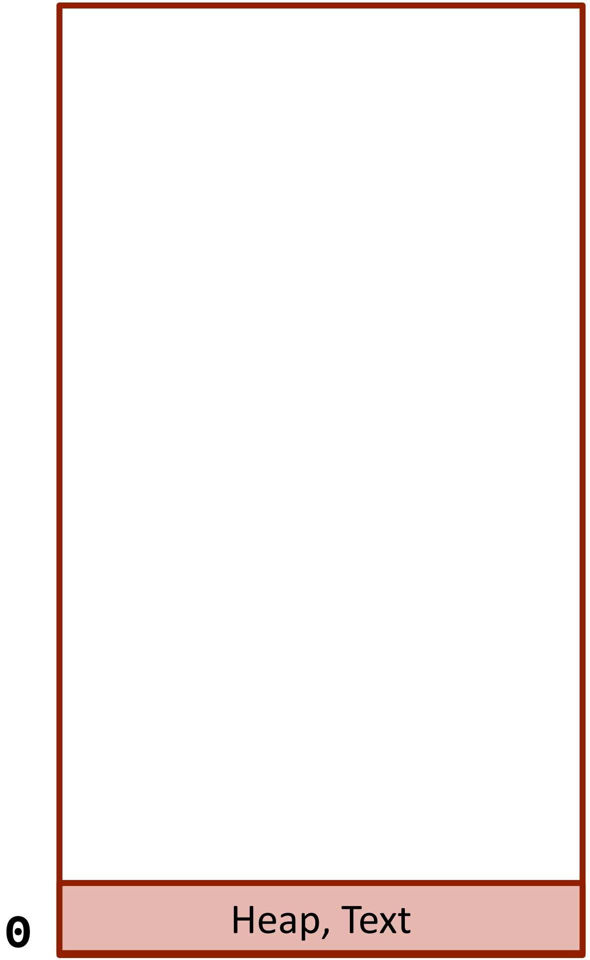
```
5! = 120
```



Recursion in action

Console:

```
5! = 120
```



Recursive vs Iterative Methods

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

```
int factorialIterative (int n) {
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result = result * i;
    }
    return result;
}
```

Recursive vs Iterative Methods

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

n = 5, time = 5.823 ms

```
int factorialIterative (int n) {
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result = result * i;
    }
    return result;
}
```

n = 5, time = 5.485 ms

Recursive vs Iterative Methods

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

n = 5, time = 5.823 ms

n = 100,000, time = 8.703 ms

```
int factorialIterative (int n) {
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result = result * i;
    }
    return result;
}
```

n = 5, time = 5.485 ms

n = 100,000, time = 5.589 ms

Recursive vs Iterative Methods

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

```
int factorialIterative (int n) {
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result = result * i;
    }
    return result;
}
```

n = 5, time = 5.823 ms

n = 100,000, time = 8.703 ms

n = 1,000,000, “segmentation fault”

n = 5, time = 5.485 ms

n = 100,000, time = 5.589 ms

n = 1,000,000, time = 7.501 ms

What is recursion?

- In programming, it means that the function calls itself
- Every time the function is called, the problem becomes a little smaller

```
void recurse() {
    recurse();
}
```

*never ever write code like this

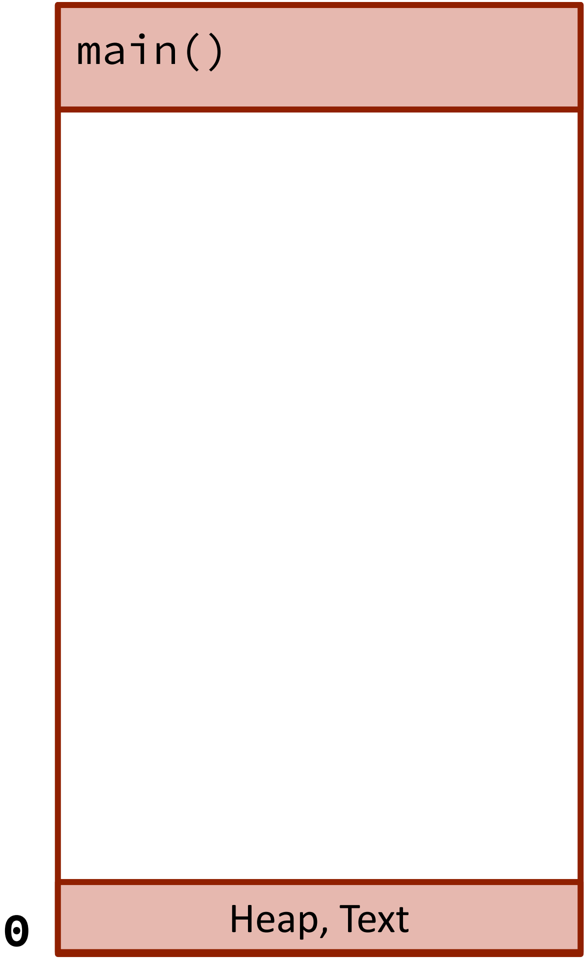
What is recursion?

```
void recurse() {
    recurse();
}
```

1. Your code must have a case for all valid inputs.
2. You must have a base case that does not make recursive calls.
3. When you make a recursive call it should be to a simpler instance of the same problem, and make progress towards the base case.

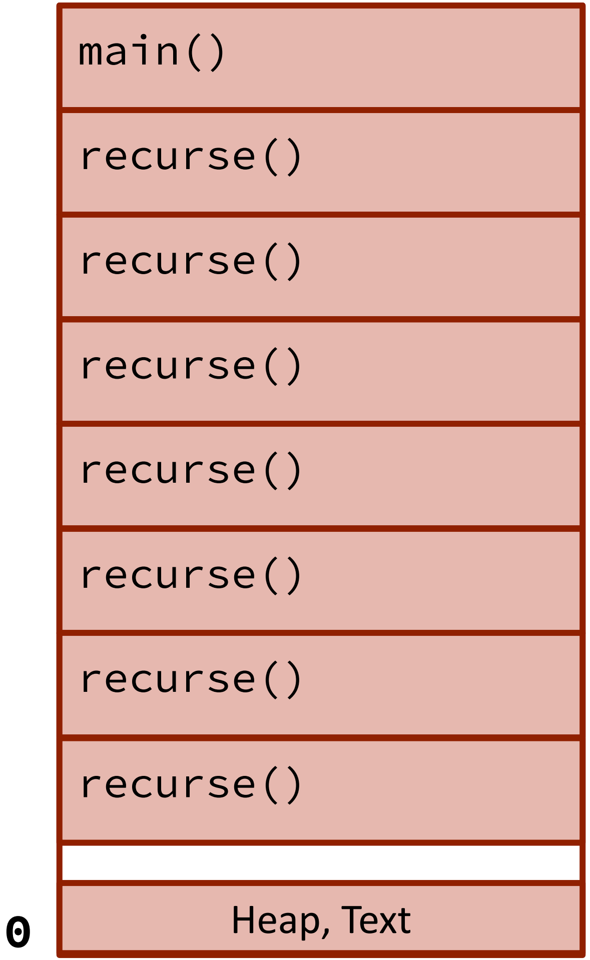
What is recursion?

```
void recurse() {
    recurse();
}
```



What is recursion?

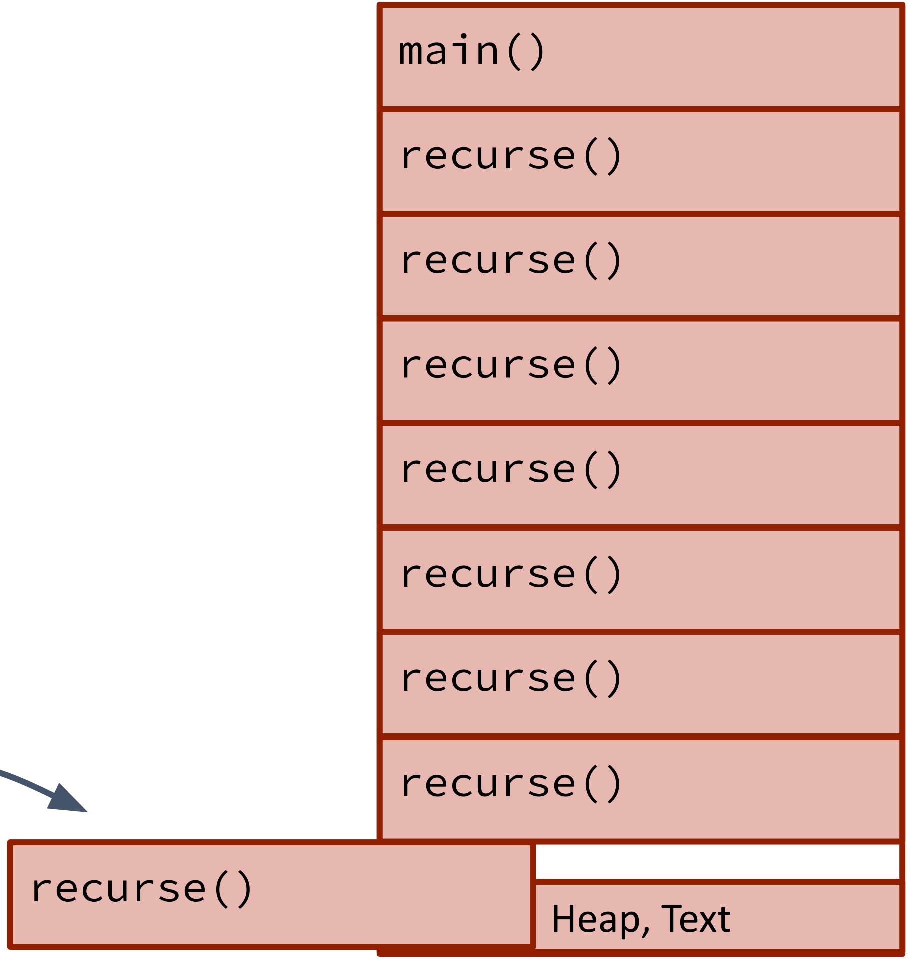
```
void recurse() {
    recurse();
}
```



What is recursion?

```
void recurse() {
    recurse();
}
```

Stack Overflow!



Reverse a String

Reversing strings

Suppose we want to reverse strings like in the following examples:

“dog” → “god”

“stressed” → “desserts”

“racecar” → “racecar”

“yo” → “oy”

“a” → “a”

Approaching recursive problems

- Look for self-similarity.
- Try out an example.
  - Work through a simple example and then increase the complexity.
  - Think about what information needs to be “stored” at each step in the recursive case (like the current value of `n` in each `factorial` stack frame).
- Ask yourself:
  - What is the base case? (What is the simplest case?)
  - What is the recursive case? (What pattern of self-similarity do you see?)

Reversing strings

Look for self-similarity: “stressed” → “desserts”

Reversing strings

Look for self-similarity: “stressed” → “desserts”

- What’s the first step you would take to reverse “stressed”?

Reversing strings

Look for self-similarity: “stressed” → “desserts”

- Take the ‘s’ and put it at the end of the string

Reversing strings

Look for self-similarity: “stressed” → “desserts”

- Take the ‘s’ and put it at the end of the string
- Then reverse “tressed”

Reversing strings

Look for self-similarity: “stressed” → “desserts”

- Take the ‘s’ and put it at the end of the string
- Then reverse “tressed”
    - Take the ‘t’ and put it at the end of the string
    - Then reverse “ressed”

Reversing strings

Look for self-similarity: “stressed” → “desserts”

- Take the ‘s’ and put it at the end of the string
- Then reverse “tressed”
    - Take the ‘t’ and put it at the end of the string
    - Then reverse “ressed”
        - Take the ‘r’ and put it at the end of the string
        - Then reverse “essed”

Reversing strings

Look for self-similarity: “stressed” → “desserts”

- Take the ‘s’ and put it at the end of the string
- Then reverse “tressed”
    - Take the ‘t’ and put it at the end of the string
    - Then reverse “ressed”
        - Take the ‘r’ and put it at the end of the string
        - Then reverse “essed”
            - …
                - Take the ‘d’ and put it at the end of the string
                - Then reverse “” → get “”

Reversing strings

Look for self-similarity: “stressed” → “desserts”

- Take the ‘s’ and put it at the end of the string
- Then reverse “tressed”
    - Take the ‘t’ and put it at the end of the string
    - Then reverse “ressed”
        - Take the ‘r’ and put it at the end of the string
        - Then reverse “essed”
            - …
                - Take the ‘d’ and put it at the end of the string
                    - **Base Case**: reverse “” → get “”

Reversing strings

Look for self-similarity: **“stressed” → “desserts”**

- Take the ‘s’ and put it at the end of the string
- Then reverse “tressed”
    - Take the ‘t’ and put it at the end of the string
    - Then reverse “ressed”
        - Take the ‘r’ and put it at the end of the string
        - Then reverse “essed”
            - …
                - Take the ‘d’ and put it at the end of the string
                    - **Base Case**: reverse “” → get “”

Reversing strings

Look for self-similarity: **reverseString("stressed") → "desserts"**

- Take the 's' and put it at the end of the string
- Then reverse "tressed"
    - Take the 't' and put it at the end of the string
    - Then reverse "ressed"
        - Take the 'r' and put it at the end of the string
        - Then reverse "essed"
            - …
                - Take the 'd' and put it at the end of the string
                    - **Base Case**: reverse "" → get ""

Reversing strings

Look for self-similarity: reverseString(“stressed”) → “desserts”

- **Take the ‘s’ and put it at the end of the string**
- **Then reverse “tressed”**
    - Take the ‘t’ and put it at the end of the string
    - Then reverse “ressed”
        - Take the ‘r’ and put it at the end of the string
        - Then reverse “essed”
            - …
                - Take the ‘d’ and put it at the end of the string
                    - **Base Case**: reverse “” → get “”

Reversing strings

Look for self-similarity: reverseString(“stressed”) → “desserts”

- **reverseString(“stressed”) = reverseString(“tressed”) + ‘s’**
    - Take the ‘t’ and put it at the end of the string
    - Then reverse “ressed”
        - Take the ‘r’ and put it at the end of the string
        - Then reverse “essed”
            - …
                - Take the ‘d’ and put it at the end of the string
                    - **Base Case**: reverse “” → get “”

Reversing strings

Look for self-similarity: reverseString(“stressed”) → “desserts”

- reverseString(“stressed”) = reverseString(“tressed”) + ‘s’
  - **Take the ‘t’ and put it at the end of the string**
  - **Then reverse “ressed”**
    - Take the ‘r’ and put it at the end of the string
    - Then reverse “essed”
      - …
        - Take the ‘d’ and put it at the end of the string
          - **Base Case**: reverse “” → get “”

Reversing strings

Look for self-similarity: reverseString(“stressed”) → “desserts”

- reverseString(“stressed”) = reverseString(“tressed”) + ‘s’
    - **reverseString(“tressed”) = reverseString(“ressed”) + ‘t’**
        - Take the ‘r’ and put it at the end of the string
        - Then reverse “essed”
            - …
                - Take the ‘d’ and put it at the end of the string
                    - **Base Case**: reverse “” → get “”

Reversing strings

Look for self-similarity: reverseString(“stressed”) → “desserts”

- reverseString(“stressed”) = reverseString(“tressed”) + ‘s’
    - reverseString(“tressed”) = reverseString(“ressed”) + ‘t’
        - **Take the ‘r’ and put it at the end of the string**
        - **Then reverse “essed”**
            - …
                - Take the ‘d’ and put it at the end of the string
                    - **Base Case**: reverse “” → get “”

Reversing strings

Look for self-similarity: reverseString(“stressed”) → “desserts”

- reverseString(“stressed”) = reverseString(“tressed”) + ‘s’
    - reverseString(“tressed”) = reverseString(“ressed”) + ‘t’
        - **reverseString(“ressed”) = reverseString(“essed”) + ‘r’**
            - …
                - Take the ‘d’ and put it at the end of the string
                    - **Base Case**: reverse “” → get “”

Reversing strings

Look for self-similarity: reverseString(“stressed”) → “desserts”

- reverseString(“stressed”) = reverseString(“tressed”) + ‘s’
  - reverseString(“tressed”) = reverseString(“ressed”) + ‘t’
    - reverseString(“ressed”) = reverseString(“essed”) + ‘r’
      - …
        - **Take the ‘d’ and put it at the end of the string**
          - **Base Case**: reverse “” → get “”

Reversing strings

Look for self-similarity: reverseString(“stressed”) → “desserts”

- reverseString(“stressed”) = reverseString(“tressed”) + ‘s’
    - reverseString(“tressed”) = reverseString(“ressed”) + ‘t’
        - reverseString(“ressed”) = reverseString(“essed”) + ‘r’
            - …
                - **reverseString(“d”) = reverseString(“”) + ‘d’**
                    - **Base Case**: reverse “” → get “”

Reversing strings

Look for self-similarity: reverseString(“stressed”) → “desserts”

- reverseString(“stressed”) = reverseString(“tressed”) + ‘s’
  - reverseString(“tressed”) = reverseString(“ressed”) + ‘t’
    - reverseString(“ressed”) = reverseString(“essed”) + ‘r’
      - …
        - reverseString(“d”) = reverseString(“”) + ‘d’
          - **Base Case**: reverse “” → get “”

Reversing strings

- **Recursive Case:**

  reverseString(str) = reverseString(str w/o first letter) + first letter

- **Base Case:**

  reverseString(“”) = “”

Reversing strings

- **Recursive Case:**

  reverseString(str) = reverseString(str w/o first letter) + first letter

  or

  reverseString(str) = last letter + reverseString(str w/o last letter)

- **Base Case:**

  reverseString(“”) = “”

Let’s Code it Up!

Recap

- Recursion is a problem-solving technique in which tasks are completed by reducing them into r**epeated, smaller tasks of the same form**
    - A recursive operation (function) is defined in terms of itself (i.e. it calls itself)

Recap

- Recursion is a problem-solving technique in which tasks are completed by reducing them into r**epeated, smaller tasks of the same form**
- Recursion has two main parts: **base case** and **recursive case**
  - Base case: Simplest form of the problem that has a direct answer
  - Recursive case: The step where you break the problem into a smaller, self-similar task

Recap

- Recursion is a problem-solving technique in which tasks are completed by reducing them into r**epeated, smaller tasks of the same form**
- Recursion has two main parts: **base case** and **recursive case**
- The solution will get built up as you come back up the **call stack**.
  - The base case will define the “base” of the solution you’re building up.
  - Each previous recursive call contributes a little bit to the final solution.
  - The initial call to your recursive function is what will return the completely constructed answer.

Recap

- Recursion is a problem-solving technique in which tasks are completed by reducing them into r**epeated, smaller tasks of the same form**
- Recursion has two main parts: **base case** and **recursive case**
- The solution will get built up as you come back up the **call stack**.
- When solving problems recursively, look for **self-similarity** and think about what information is getting stored in each stack frame.

Midterm Logistics

- Monday, July 17 from 7-9pm in Hewlett Teaching Center, Room 200
  - Students with exam accommodations will get an email from us
- This exam is on paper, using pen/pencil.
- The exam is closed-book and closed-device.
  - Provide you with a reference sheet on Stanford library functions.
  - Allow you to bring your own notes sheet (one page, front and back, 8-1/2" x 11", where you have written/printed/drawn whatever information you would like to have handy during the exam)
- All information is here

Midterm Logistics

- Coverage: Material up to and including Lecture 10, Assignment 2, and Section 3 (not testing stuff only in the textbook)
- Format:
  - Write a function or a few lines of code
  - Trace through code and analyze its behavior
  - Write response to a short answer question
- Practice:
  - 2 full length practice exams with solutions
  - Section problems
  - Review session on Thursday and Friday

Midterm

- Evaluate your problem-solving skills and conceptual understanding of the material, not your ability to use perfect syntax
- Most points awarded for valid approach to solving the problem, fewer points for the minute details of executing your plan
- Not taking off points for
  - Missing braces around clearly indented blocks of code
  - Missing semicolons
  - Missing `#include`
- Give partial credit for meaningful pseudocode