# ECE2800J

## Programming and Introductory Data Structures

### Function Call Mechanism

**Learning Objectives:**

Understand function call mechanism

# Last time

- Recursion
- Function Pointers

# Recursion

- Recursion is a nice way to solve problems
  - "Recursive" just means "refers to itself".
  - There is (at least) one "trivial" base or "stopping" case.
  - All other cases can be solved by first solving one smaller case, and then combining the solution with a simple step.
- Example: calculate factorial $n!$

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & n > 0 \end{cases}$$

```
int factorial (int n) {
// REQUIRES: n >= 0
// EFFECTS:  computes n!
  if (n == 0) return 1; // base case
  else return n*factorial(n-1); // recursive step
}
```

# Function Pointers

## The basic format

- How do you define a **variable** that points to a function that takes two integers, and returns an integer?

- Here's how:

```
int   (*foo)(int, int);
```

- You read this from "inside out". In other words:

```
foo                              "foo"
(*foo)                           "is a pointer"
(*foo)(          );              "to a function"
(*foo)(int, int);                "that takes two integers"
int   (*foo)(int, int);          "and returns an integer"
```

# Outline

- Recursion
- Function Pointers
- Function Call Mechanism

# Call Stacks

## How a function call really works

- When we call a function, the program does following steps:

1. Evaluate the actual arguments to the function (order is not guaranteed).

Example: y = add(4-1, 5);

2. Create an "**activation record**" (sometimes called a "**stack frame**") to hold the function's formal parameters and local variables.
   - When call function `int add(int a, int b)`, system creates an activation record:

a, b (formal), result (local)

3. Copy the actuals' values to the formals' storage space.

a=3
b=5

4. Evaluate the function in its local scope.
5. Replace the function call with the result.

y=8

6. Destroy the activation record.

# Call Stacks

## How a function call really works

- It is typical to have multiple function calls. How the activation records are maintained?
  - Answer: stored as a **stack**.
- Stack: a set of objects which is modified as **last in first out**. Example: a stack of plates in a cafeteria
  - Each time you clean a plate, you add it to the top of the stack
  - Each time a new plate is needed, the one at the top is taken **first**

# Call Stacks

## How a function call really works

- When a function f() is called, its **activation record** is added to the "top" of the stack.
- When the function f() returns, its **activation record** is removed from the "top" of the stack.
- In the meantime, f() may have called other functions.
  - These functions create corresponding activation records.
  - These functions must return (and destroy their corresponding activation records) before f() can return.

# Call Stacks

## Example

- When a function is called, its **activation record** is added to the "top" of the stack.
- When that function returns, its **activation record** is removed from the "top" of the stack.

| double add(double a, double b): a = 1, b = 0, result = 0 |
| --- |
| double sin(double x): x = 1, result = 0 |
| int main(): x = 1, sinResult = 0 |

- Note: "top" is placed in quotes, because in reality, stack of activation records grows **down** rather than **up**.

# Call Stacks

## Example

```
int plus_one(int x) {
  return (x+1);
}

int plus_two(int x) {
  return (1 + plus_one(x));
}

int main() {
  int result = 0;

  result = plus_two(0);
  cout << result;
  return 0;
}
```

# Call Stacks

## Example

```
int plus_one(int x) {
  return (x+1);
}

int plus_two(int x) {
  return (1 + plus_one(x));
}

int main() {
  int result = 0;

  result = plus_two(0);
  cout << result;
  return 0;
}
```

Main starts out with an activation record with room only for the local "result":

main:

| result: 0 |
|---|

# Call Stacks

## Example

```
int plus_one(int x) {
  return (x+1);
}

int plus_two(int x) {
  return (1 + plus_one(x));
}

int main() {
  int result = 0;

  result = plus_two(0);
  cout << result;
  return 0;
}
```

Then, main calls plus_two, passing the literal value "0":

main:

| result: 0 |
|---|

plus_two:

| x: 0 |
|---|

# Call Stacks

## Example

```
int plus_one(int x) {
  return (x+1);
}

int plus_two(int x) {
  return (1 + plus_one(x));
}

int main() {
  int result = 0;

  result = plus_two(0);
  cout << result;
  return 0;
}
```

Which in turn calls plus_one:

main:

| result: 0 |
|---|

plus_two:

| x: 0 |
|---|

plus_one:

| x: 0 |
|---|

# Call Stacks

## Example

```
int plus_one(int x) {
  return (x+1);
}

int plus_two(int x) {
  return (1 + plus_one(x));
}

int main() {
  int result = 0;

  result = plus_two(0);
  cout << result;
  return 0;
}
```

plus_one adds one to x, returning the value 1:

main:

| result: 0 |
|---|

plus_two:

| x: 0 |
|---|

1

plus_one:

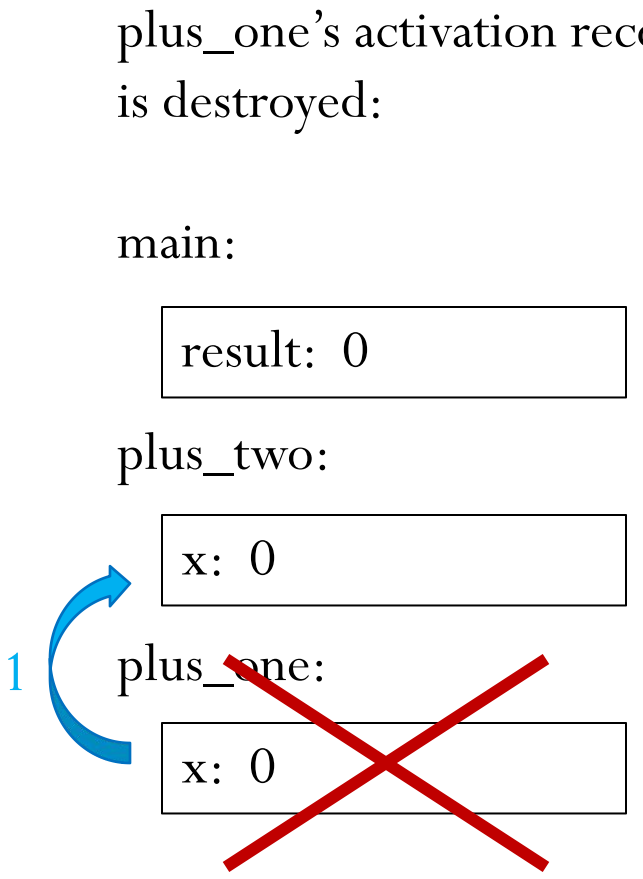
| x: 0 |
|---|

# Call Stacks

## Example

```
int plus_one(int x) {
  return (x+1);
}

int plus_two(int x) {
  return (1 + plus_one(x));
}

int main() {
  int result = 0;

  result = plus_two(0);
  cout << result;
  return 0;
}
```

plus_one's activation record is destroyed:

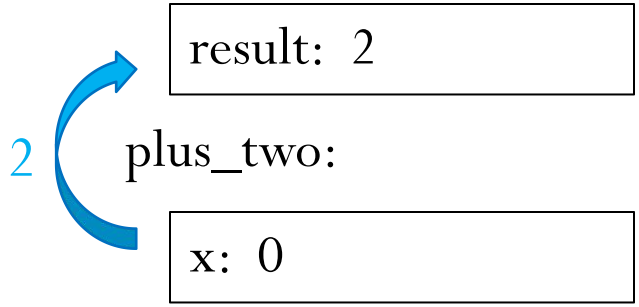


# Call Stacks

## Example

```
int plus_one(int x) {
  return (x+1);
}

int plus_two(int x) {
  return (1 + plus_one(x));
}

int main() {
  int result = 0;

  result = plus_two(0);
  cout << result;
  return 0;
}
```

plus_two adds one to the result, and returns the value 2:

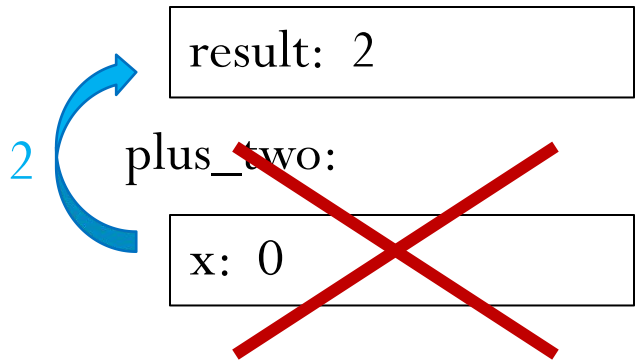


# Call Stacks

## Example

```
int plus_one(int x) {
  return (x+1);
}

int plus_two(int x) {
  return (1 + plus_one(x));
}

int main() {
  int result = 0;

  result = plus_two(0);
  cout << result;
  return 0;
}
```

plus_two's activation record is destroyed:



# Call Stacks

## Example

```
int plus_one(int x) {
  return (x+1);
}

int plus_two(int x) {
  return (1 + plus_one(x));
}

int main() {
  int result = 0;

  result = plus_two(0);
  cout << result;
  return 0;
}
```

main then prints the result:

**2**

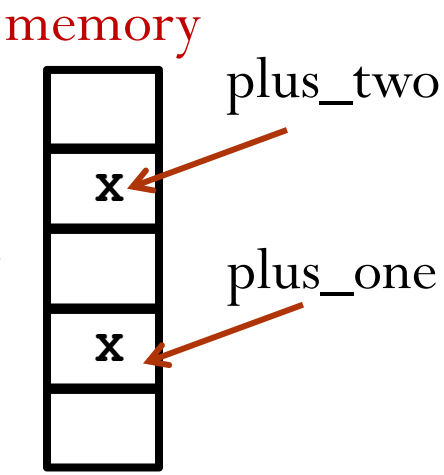
main:

| result: 2 |
|---|

# Call Stacks

## Example: Some things to note

- Even though `plus_one` and `plus_two` both have formal parameters called "x", there is no problem.
  - These two x's are at different locations in memory.
  - `plus_one` cannot see `plus_two`'s x.
  - Instead, the **value** of `plus_two`'s x is passed to `plus_one`, and stored in `plus_one`'s x.

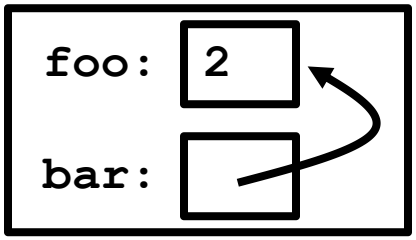


# Call Stack

## Example: Using Pointers

```
void add_one(int *x) {
   *x = *x + 1;
}

int main() {
   int foo = 2;
   int *bar = &foo;
   add_one(bar);
   return 0;
}
```

Activation record of main:



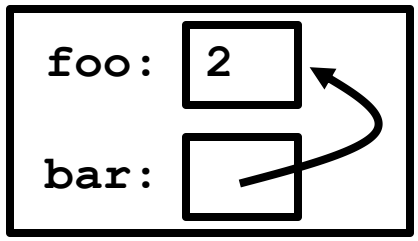
# Call Stack

## Example: Using Pointers

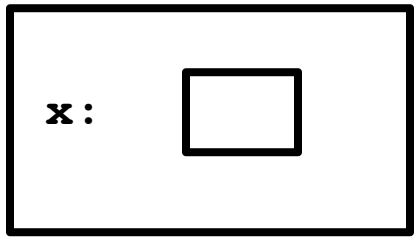
```
void add_one(int *x) {
   *x = *x + 1;
}

int main() {
   int foo = 2;
   int *bar = &foo;
   add_one(bar);
   return 0;
}
```

Main calls add_one, creating an activation record for add_one

main:




add_one:

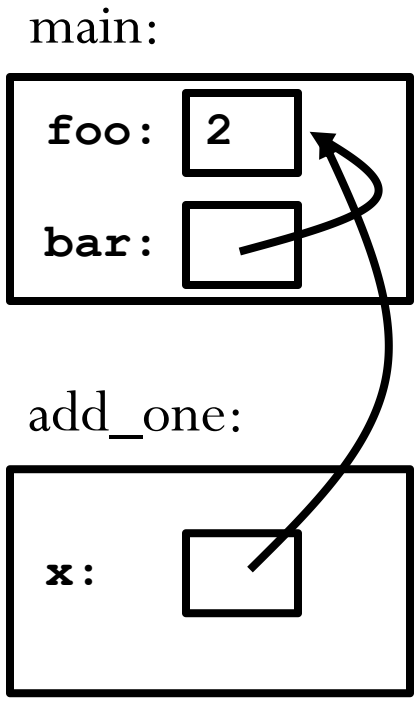


# Call Stack

## Example: Using Pointers

```
void add_one(int *x) {
   *x = *x + 1;
}

int main() {
   int foo = 2;
   int *bar = &foo;
   add_one(bar);
   return 0;
}
```

Copy the value of bar to add_one's formal parameter x.



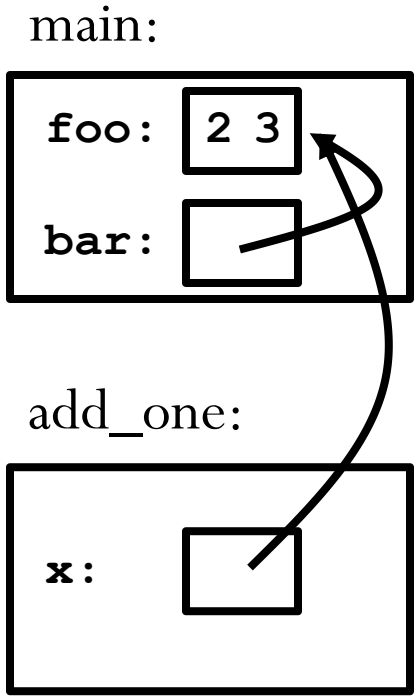

Both x and bar point to foo.

# Call Stack

## Example: Using Pointers

```
void add_one(int *x) {
   *x = *x + 1;
}

int main() {
   int foo = 2;
   int *bar = &foo;
   add_one(bar);
   return 0;
}
```

add_one adds 1 to the object pointed to by x.

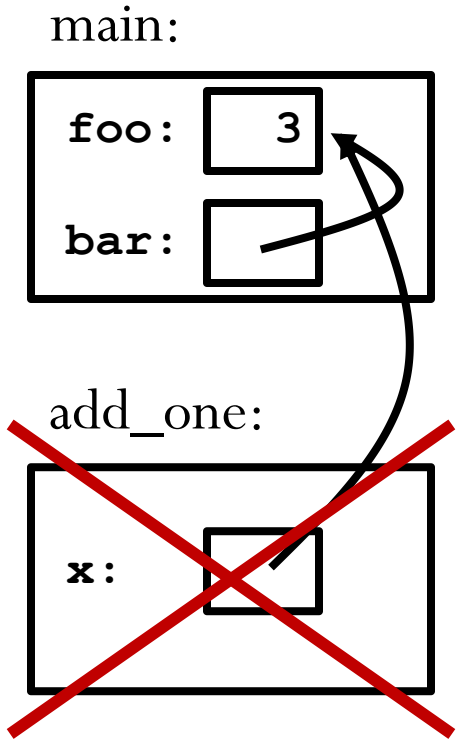


# Call Stack

## Example: Using Pointers

```
void add_one(int *x) {
   *x = *x + 1;
}

int main() {
   int foo = 2;
   int *bar = &foo;
   add_one(bar);
   return 0;
}
```

add_one's activation record is destroyed.



# Call Stack

## Example: Recursion


- Suppose we call our function as follows:

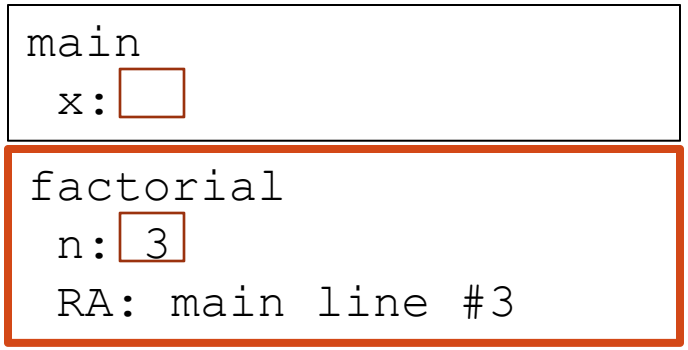
```
   void main()
1. {
2.  int x;
3.  x = factorial(3);
4. }
```

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

# Call Stack

## Example: Recursion

- main() calls factorial with an argument 3.
- We evaluate the actual argument, create an activation record, and copy the actual value to the formal.




**RA = "Return Address"**

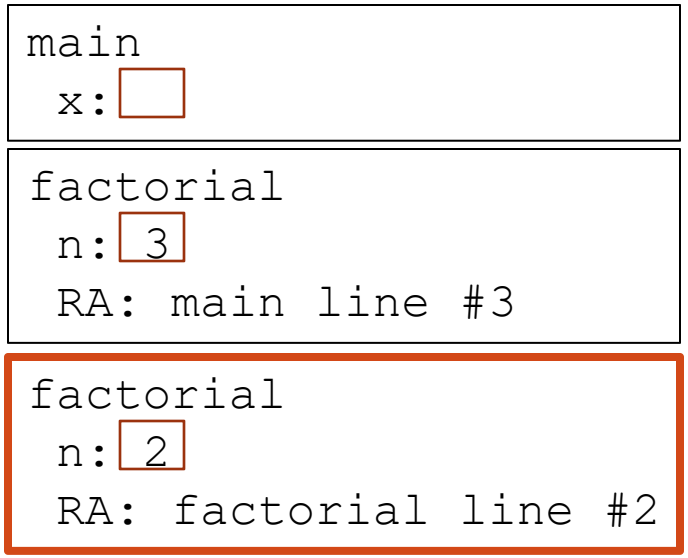
```
   int factorial (int n) {
1.  if (n == 0) return 1;
2.  else return n*factorial(n-1);
   }
```

# Call Stack

## Example: Recursion

- Now we evaluate the body of factorial:
  - n is not zero, so we evaluate the **else** arm of the if statement:

    return 3 * factorial(2)
  - So, factorial must call factorial. We will create a **new** activation record for a **new** instance of factorial.




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

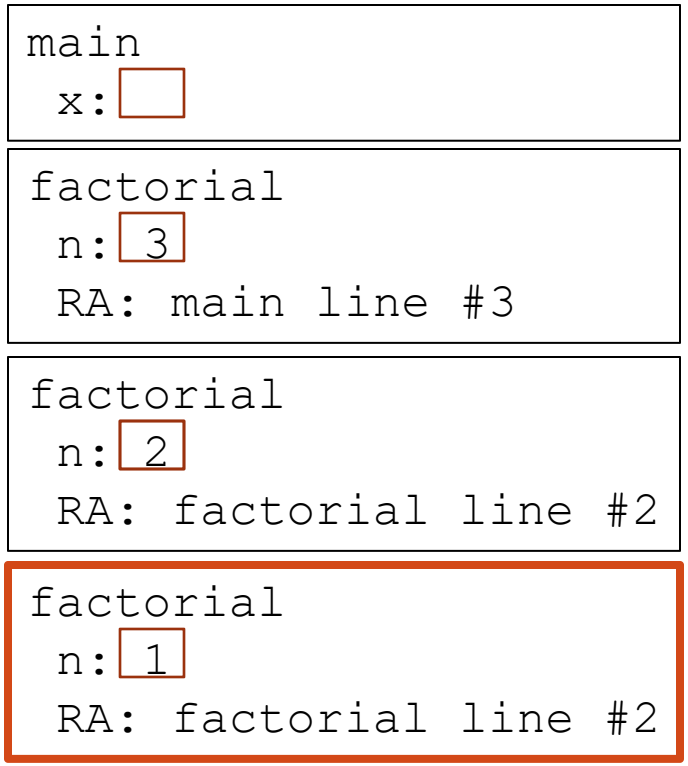
# Call Stack

## Example: Recursion

- Again, n is not zero, so we evaluate the **else** arm again:

  return 2 * factorial(1)

- This creates a new activation record for factorial




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

# Call Stack

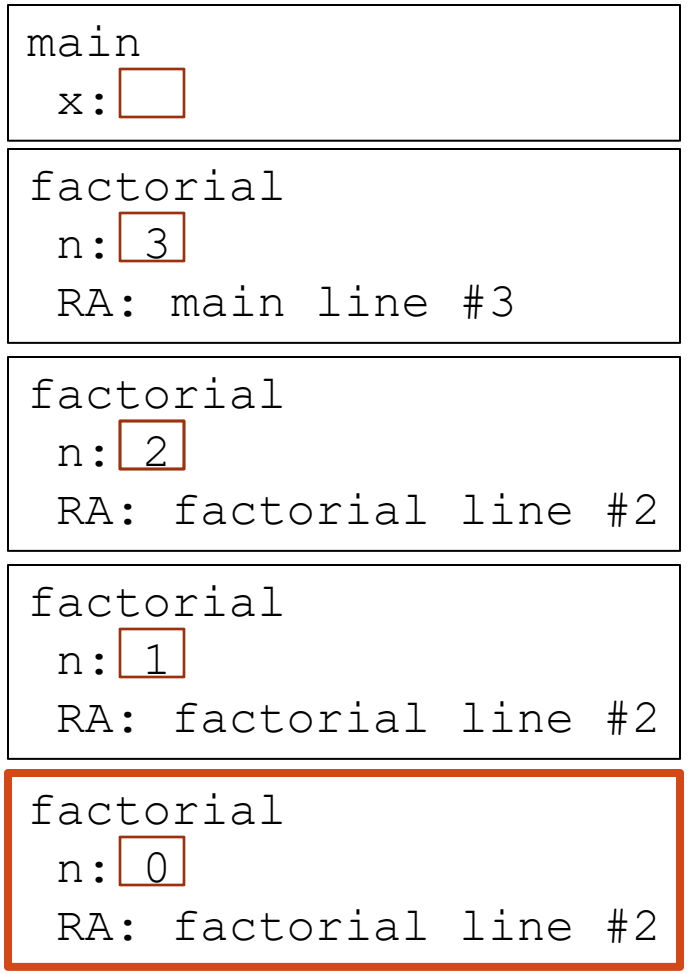
## Example: Recursion

- And again, we evaluate the **else** arm:

  return 1*factorial(0)

- This creates a new activation record for factorial

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

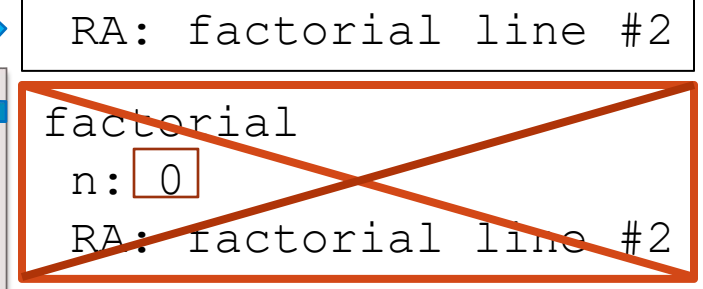


# Call Stack

## Example: Recursion

- In evaluating factorial(0), n is zero, so we evaluate the **if** arm rather than **else** arm.
- Return the value "1"
- Popping the most recent activation record off the stack.

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



# Call Stack

## Example: Recursion

- In factorial(1), we called factorial(0) as follows:

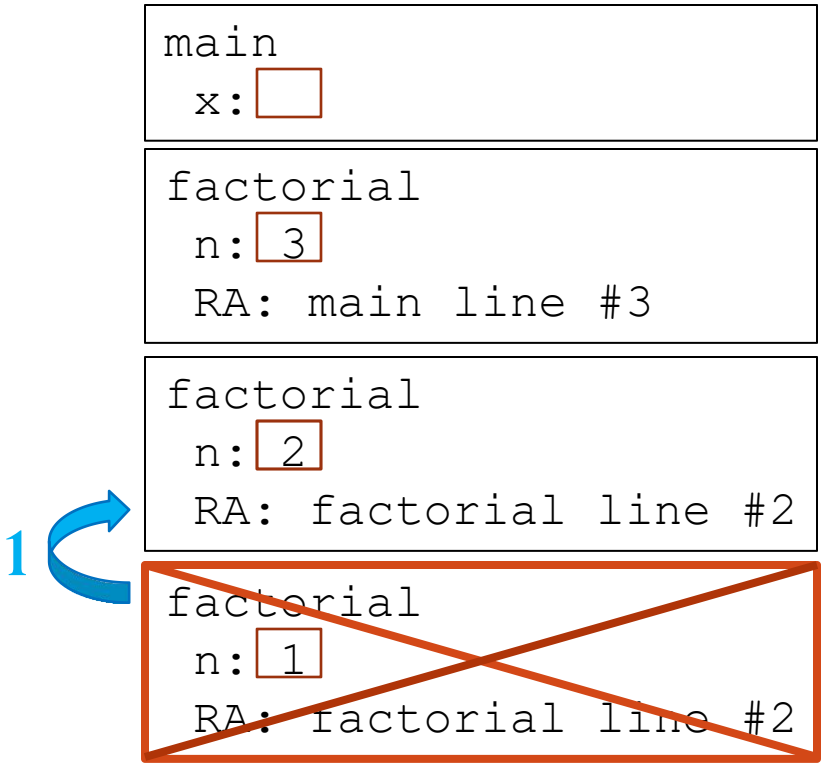
  return 1 * factorial(0)

- Now we know the value of factorial(0), so we complete factorial(1):

  return 1 * 1 => return 1;

  from factorial(1)

- This pops another activation record off the stack



# Call Stack

## Example: Recursion

- Now it allows us to complete evaluating factorial(2):

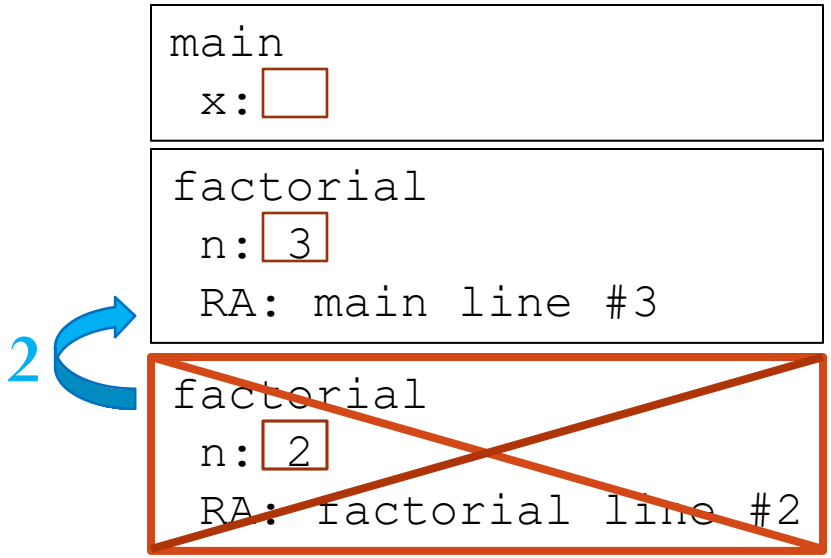
  return 2 * factorial(1) =>

  return 2 * 1 =>

  return 2

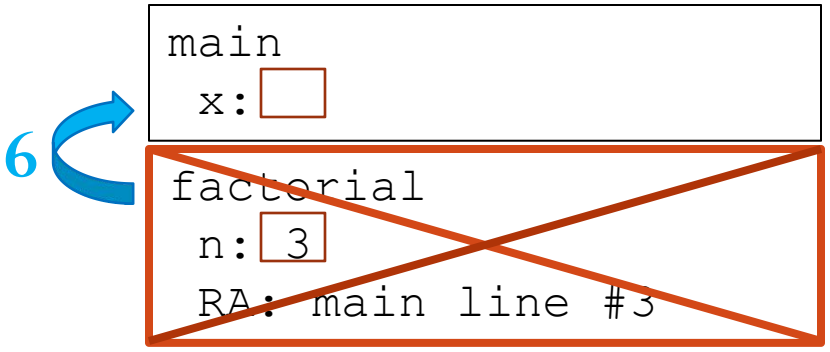
  from factorial(2)

- Now pop off another activation record.



# Call Stack

## Example: Recursion




- Now we can complete evaluating factorial(3):

  return 3 * factorial(2) =>

  return 3 * 2 =>

  return 6

- That is the correct answer.

- Don't forget that last pop!

# Which Statements Are True?

Select all the correct answers.

- **A.** The number of recursive calls of factorial can be as high as we want.
- **B.** The number of calls of factorial could be just 1.
- **C.** We can change the function factorial so that the number of calls of factorial could be **reduced by** 1 in general case.
- **D.** None of the above.

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

# Exercise: Fibonacci

- The Fibonacci Sequence is the series of numbers:
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …
- The next number is found by adding up the two numbers before it
- Write a recursive function to calculate the n-th Fibonacci number.
- int fib(int n);