Programming Abstractions

CS106B

Cynthia Bailey Lee
Julie Zelenski

Today's Topics

Recursion!

- Functions calling functions

Next time:

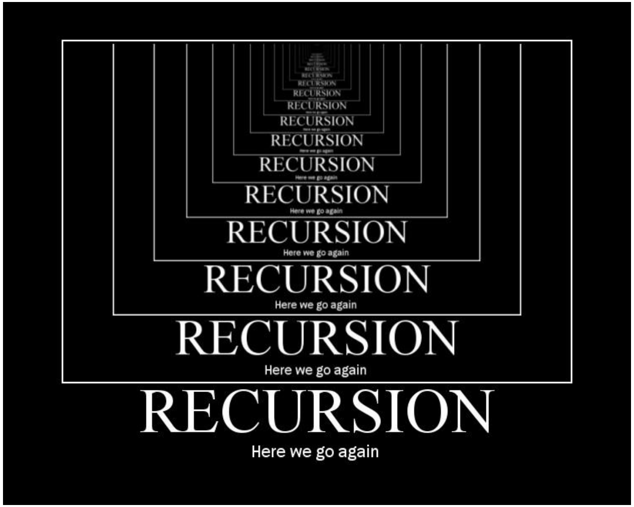
- More recursion! It's Recursion Week!
  - Like Shark Week, but more nerdy

Announcements:

- Today is Indigenous Peoples Day
  - Indigenous students in our class who wish to take the day off for reflection and observance are encouraged to do so.
  - We encourage others to set aside some time today to learn about the accomplishments, hardships, and current issue advocacy of Indigenous people locally and around the world.

Recursion!

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



Factorial!

$$\boldsymbol{n}! = \boldsymbol{n}(\boldsymbol{n}-\mathbf{1})(\boldsymbol{n}-\mathbf{2})(\boldsymbol{n}-\mathbf{3})(\boldsymbol{n}-\mathbf{4})\ldots(\mathbf{3})(\mathbf{2})(\mathbf{1})$$

This could be a really long expression!

Recursion is a technique for tackling large or complicated problems by just “eating” one “bite” of the problem at a time.

Factorial!

$$\boldsymbol{n}! = \boldsymbol{n}(\boldsymbol{n}-\mathbf{1})(\boldsymbol{n}-\mathbf{2})(\boldsymbol{n}-\mathbf{3})(\boldsymbol{n}-\mathbf{4})\ldots(\mathbf{2})(\mathbf{1})$$

An alternate mathematical formulation:

$$n! = \begin{cases} 1 & if\ n = 1 \\ n(\boldsymbol{n}-\mathbf{1})! & otherwise \end{cases}$$

Translated to code

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * someFunctionThatKnowsFactorialOfNMinus1();
    }
}
```

Factorial!

$$\boldsymbol{n}! = \boldsymbol{n}(\boldsymbol{n}-\mathbf{1})(\boldsymbol{n}-\mathbf{2})(\boldsymbol{n}-\mathbf{3})(\boldsymbol{n}-\mathbf{4})\ldots(\mathbf{2})(\mathbf{1})$$

An alternate mathematical formulation:

$$n! = \begin{cases} 1 & if\ n = 1 \\ n(\boldsymbol{n}-\mathbf{1})! & otherwise \end{cases}$$

Translated to code

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

The recursive function pattern

Always two parts:

Base case:

- This problem is so tiny, it's hardly a problem anymore! Just give answer.

Recursive case:

- This problem is still a bit large, let's (1) bite off just one piece, and (2) delegate the remaining work to recursion.

Translated to code

```
int factorial(int n) {
    if (n == 1) {  // Easy! Return trivial answer
        return 1;
    } else {  // Not easy enough to finish yet!
       return n * factorial(n - 1);
    }
}
```

The recursive function pattern

Recursive case:

- This problem is still a bit large, let’s **(1) bite off just one piece**, and (2) delegate the remaining work to recursion.

```
int factorial(int n) {
    if (n == 1) {  // Easy! Return trivial answer
        return 1;
    } else {  // Not easy enough to finish yet!
        return n * factorial(n - 1);
    }
}
```

Do <u>one</u> of the many, many multiplications required for factorial.

The recursive function pattern

Recursive case:

- This problem is still a bit large, let's (1) bite off just one piece, and **(2) delegate the remaining work to recursion**.

```
int factorial(int n) {
    if (n == 1) {  // Easy! Return trivial answer
        return 1;
    } else {  // Not easy enough to finish yet!
        return n * factorial(n - 1);
    }
}
```

Do <u>one</u> of the many, many multiplications required for factorial.

Delegate all the other multiplications to the recursive call.

Digging deeper in the recursion

Looking at how recursion works “under the hood”

Factorial!

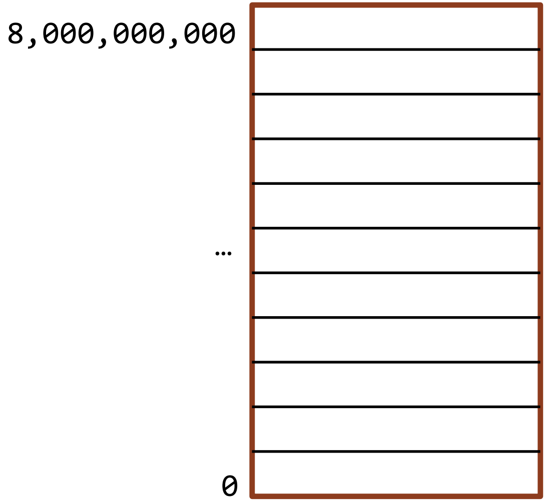
```
int factorial(int n) {
    cout << n << endl;  // **Added for this question**
    if (n == 1) {  // Easy! Return trivial answer
        return 1;
    } else {  // Not easy enough to finish yet!
        return n * factorial(n - 1);
    }
}
```

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

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

How does this look in memory?
A little background…

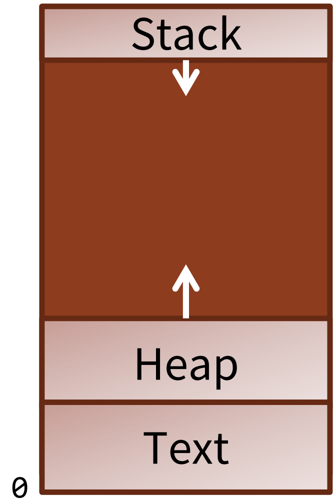
- A computer’s memory is like a <u>giant</u> Vector/array, and like a Vector, we start counting at index `0`.
- We typically draw memory vertically (rather than horizontally like a Vector), with index `0` at the bottom.
- A typical laptop’s memory has billions of these indexed slots (one byte each)



* Take CS107 to learn much more!!

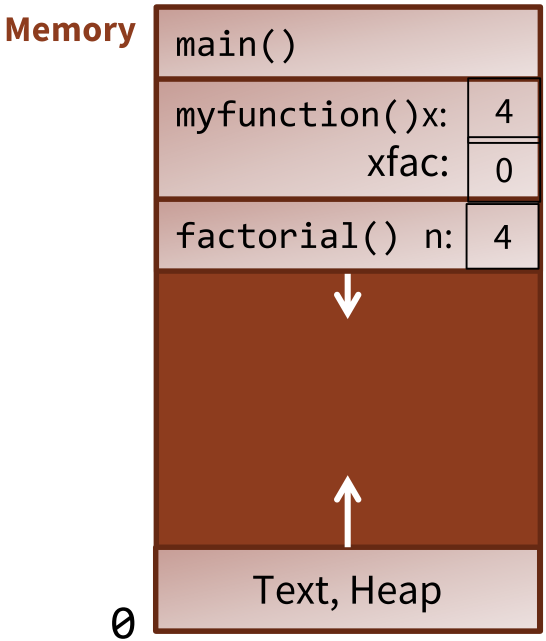
How does this look in memory? A little background…

- Broadly speaking, we divide memory into regions:
  - **Text:** the program's own code (needs to be in memory so it can run!)
  - **Heap:** we'll learn about this later in CS106B!
  - **Stack:** this is where local variables for each function are stored.



* Take CS107 to learn much more!!

How does this look in memory?

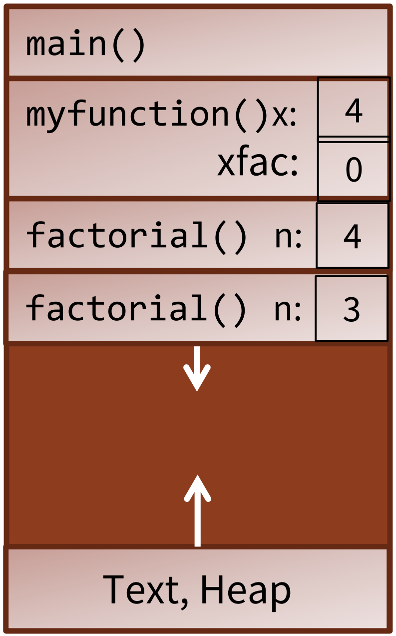


Recursive code

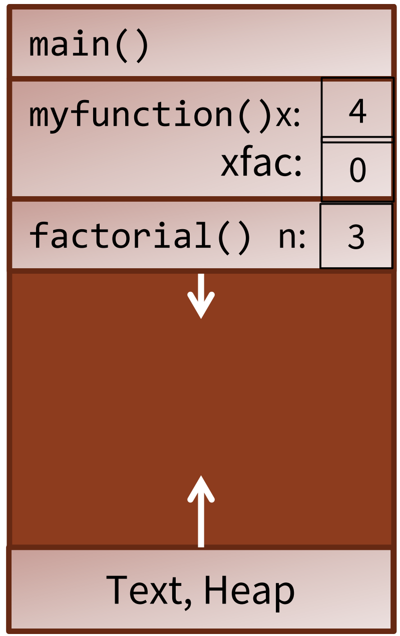
```
int factorial(int n) {
     cout << n << endl;
     if (n == 1) return 1;
     else return n * factorial(n – 1);
}

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

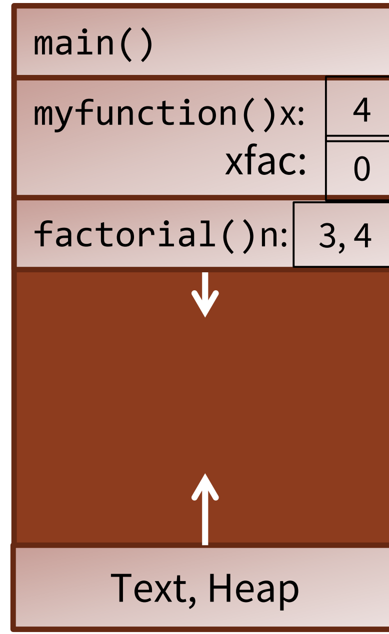
Memory (A)



Memory (B)



Memory (C)



(D) Other/none of the above

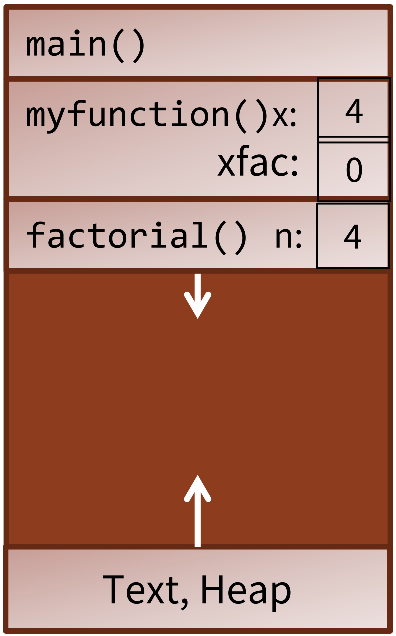
Fun fact:
The “stack” part of memory is a stack

Function **call** = **push** a stack frame

Function **return** = **pop** a stack frame

* Take CS107 to learn much more!!

The “stack” part of memory is a stack

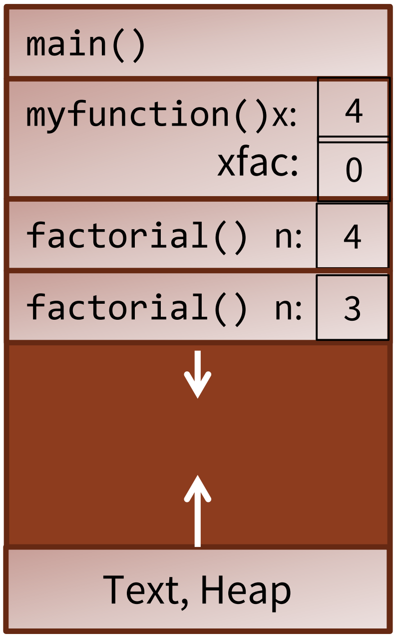


Recursive code

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

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

The “stack” part of memory is a stack

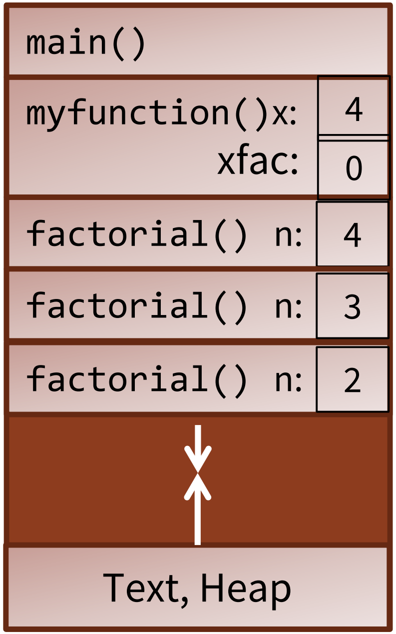


Recursive code

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

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

The “stack” part of memory is a stack



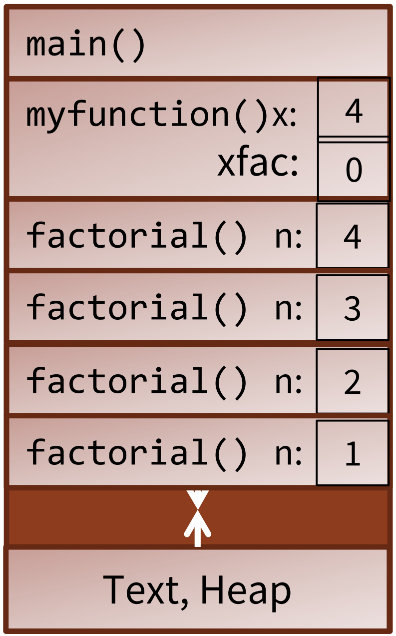
Recursive code

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

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

Answer: 3rd thing printed is 2

The “stack” part of memory is a stack



Recursive code

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

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

Factorial!

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

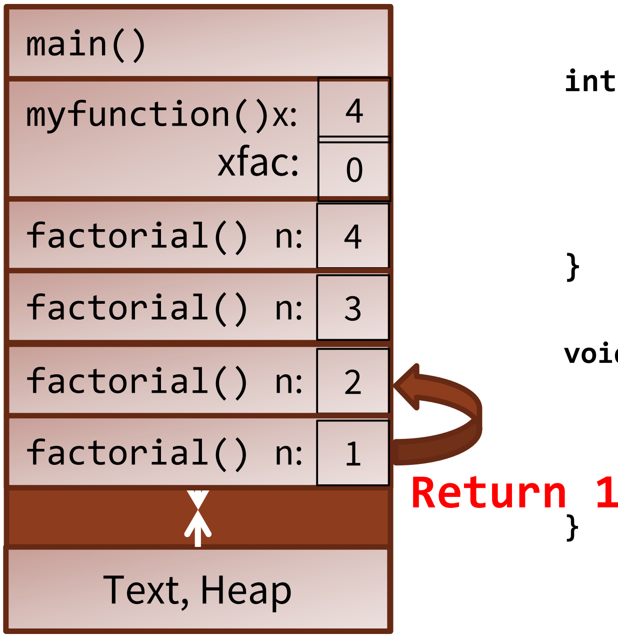
A. 4
B. 6
C. 10
D. 24
E. Other/none/more than one

Recursive code

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

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

The “stack” part of memory is a stack

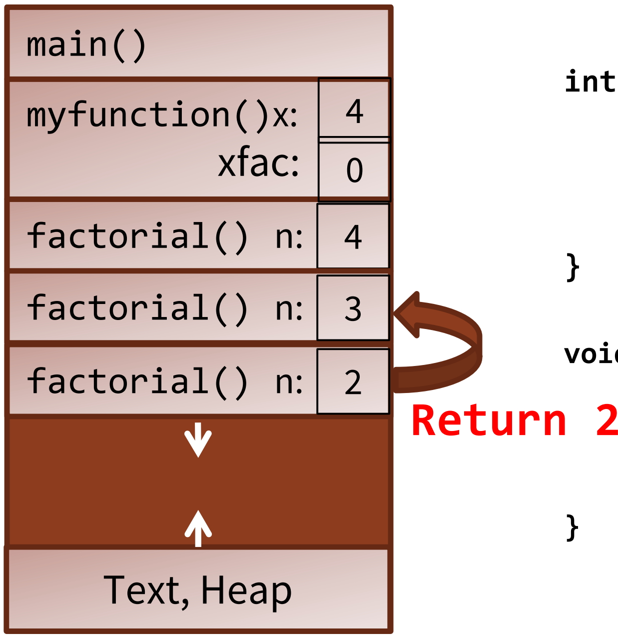


Recursive code

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

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

The “stack” part of memory is a stack

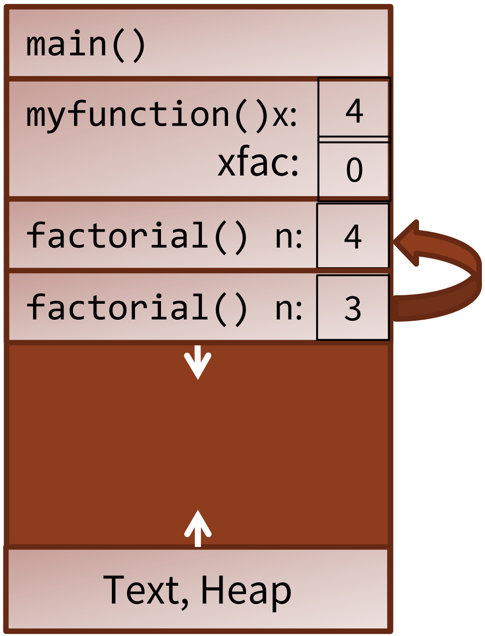


Recursive code

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

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

The "stack" part of memory is a stack



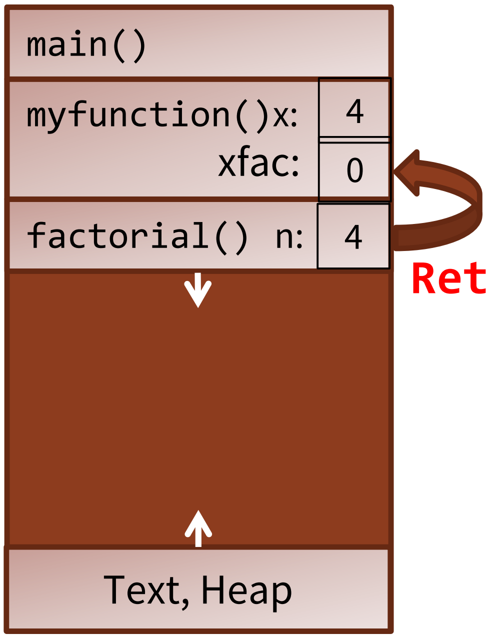
Recursive code

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

Return 6

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

The "stack" part of memory is a stack



Recursive code

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

Return 24

Answer: 4th thing returned is 24

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

Factorial!

Iterative version

```
int factorial(int n) {
    int f = 1;
    while (n > 1) {
        f = f * n;
        n = n - 1;
    }
    return f;
}
```

Recursive version

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

NOTE: sometimes **iterative can be much faster** because it doesn't have to push and pop stack frames. Method calls have overhead in terms of space *and* time (to set up and tear down).