CS 161
Intro to CS I
Finish Recursion
What is different when we call after?

• Recursive Solution

```cpp
void draw_rect(int i) {
    if(i>0){ //Base case
        cout << "******" << endl;
        cout << "*         *" << endl;
        cout << "******" << endl;
        draw_rect(--i); //Recursive call
    }
}
```
Example: Factorial

• Definition

\[0! = 1;\]
\[n! = n \times (n-1) \times \ldots \times (n-(n-1)) \times 1 = n \times (n-1)!; n > 0\]
Iterative Factorial

factorial(0) = 1;
factorial(n) = n*n-1*n-2*...*n-(n-1)*1;

long factorial(int n) {
    long fact;
    if(n==0)
        fact=1;
    else
        for(fact=n; n > 1; n--)
            fact=fact*(n-1);
    return fact;
}
Recursive Factorial

factorial(0) = 1;
factorial(n) = n*factorial(n-1);

long factorial(int n) {
    if (n == 0)    // Base case
        return 1;
    else
        return n * factorial(n - 1);    // Recursive call
}
Computing Factorial Iteratively

factorial(4)

factorial(0) = 1;
factorial(n) = n*(n-1)*...*2*1;
Computing Factorial Iteratively

factorial(4) = 4 * 3

factorial(0) = 1;
factorial(n) = n*(n-1)*...*2*1;
Computing Factorial Iteratively

\[
\text{factorial}(4) = 4 \times 3 \\
= 12 \times 2
\]

\[
\text{factorial}(n) = n \times (n-1) \times \ldots \times 2 \times 1;
\]

\[
\text{factorial}(0) = 1;
\]
Computing Factorial Iteratively

factorial(4) = 4 * 3
= 12 * 2
= 24 * 1

factorial(0) = 1;
factorial(n) = n*(n-1)*...*2*1;
Computing Factorial Iteratively

\[
\text{factorial}(n) = n \times (n-1) \times \ldots \times 2 \times 1;
\]

\[
\text{factorial}(0) = 1;
\]

\[
\text{factorial}(4) = 4 \times 3
\]
\[
= 12 \times 2
\]
\[
= 24 \times 1
\]
\[
= 24
\]
Computing Factorial Recursively

\[ \text{factorial}(0) = 1; \]
\[ \text{factorial}(n) = n \times \text{factorial}(n-1); \]
Computing Factorial Recursively

factorial(4) = 4 * factorial(3)

factorial(0) = 1;
factorial(n) = n*factorial(n-1);
Computing Factorial Recursively

\[
\text{factorial}(4) = 4 \times \text{factorial}(3) \\
= 4 \times (3 \times \text{factorial}(2))
\]

\[
\text{factorial}(0) = 1; \\
\text{factorial}(n) = n \times \text{factorial}(n-1);
\]
Computing Factorial Recursively

factorial(4) = 4 * factorial(3)
= 4 * ( 3 * factorial(2))
= 4 * ( 3 * (2 * factorial(1)))

factorial(0) = 1;
factorial(n) = n*factorial(n-1);
Computing Factorial Recursively

\[
\text{factorial}(4) = 4 \times \text{factorial}(3) \\
= 4 \times (3 \times \text{factorial}(2)) \\
= 4 \times (3 \times (2 \times \text{factorial}(1))) \\
= 4 \times (3 \times (2 \times (1 \times \text{factorial}(0))))
\]

\[
\text{factorial}(0) = 1; \\
\text{factorial}(n) = n \times \text{factorial}(n-1);
\]
Computing Factorial Recursively

\[
\text{factorial}(4) = 4 \times \text{factorial}(3) \\
= 4 \times (3 \times \text{factorial}(2)) \\
= 4 \times (3 \times (2 \times \text{factorial}(1))) \\
= 4 \times (3 \times (2 \times (1 \times \text{factorial}(0)))) \\
= 4 \times (3 \times (2 \times (1 \times 1)))
\]

\[
\text{factorial}(0) = 1; \\
\text{factorial}(n) = n \times \text{factorial}(n-1);
\]
Computing Factorial Recursively

factorial(4) = 4 * factorial(3)
  = 4 * (3 * factorial(2))
  = 4 * (3 * (2 * factorial(1)))
  = 4 * (3 * (2 * (1 * factorial(0))))
  = 4 * (3 * (2 * (1 * 1)))
  = 4 * (3 * (2 * 1))

factorial(0) = 1;

factorial(n) = n*factorial(n-1);
Computing Factorial Recursively

\[
\text{factorial}(n) = n \times \text{factorial}(n-1);
\]

\[
\text{factorial}(4) = 4 \times \text{factorial}(3)
= 4 \times (3 \times \text{factorial}(2))
= 4 \times (3 \times (2 \times \text{factorial}(1)))
= 4 \times (3 \times (2 \times (1 \times \text{factorial}(0))))
= 4 \times (3 \times (2 \times (1 \times 1)))
= 4 \times (3 \times (2 \times 1))
= 4 \times (3 \times 2)
\]
Computing Factorial Recursively

factorial(4) = 4 * factorial(3)
= 4 * (3 * factorial(2))
= 4 * ( 3 * (2 * factorial(1)))
= 4 * ( 3 * ( 2 * (1 * factorial(0))))
= 4 * ( 3 * ( 2 * (1 *1)))
= 4 * ( 3 * ( 2 * 1))
= 4 * (3 * 2)
= 4 * 6
Computing Factorial Recursively

factorial(4) = 4 * factorial(3)
  = 4 * (3 * factorial(2))
  = 4 * ( 3 * (2 * factorial(1)))
  = 4 * ( 3 * ( 2 * (1 * factorial(0))))
  = 4 * ( 3 * ( 2 * (1 *1)))
  = 4 * ( 3 * ( 2 * 1))
  = 4 * (3 * 2)
  = 4 * 6
  = 24

factorial(0) = 1;
factorial(n) = n*factorial(n-1);
Differences

• Pros
  – Readability

• Cons
  – Efficiency
  – Memory
Recursive Factorial

factorial(4) --> Executes factorial(4)

Main method

Space Required for factorial(4)

Space Required for factorial(3)

Space Required for factorial(2)

Space Required for factorial(1)

Stack

Main method

Stack
Recursive Factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Executes factorial(3)

Step 1: executes factorial(3)

Step 2: executes factorial(2)

Step 3: executes factorial(1)

Step 4: executes factorial(0)

Step 5: return 1

Step 6: return 1

Step 7: return 2

Step 8: return 6

Space Required for factorial(4)

Space Required for factorial(3)

Space Required for factorial(2)

Space Required for factorial(1)
Recursive Factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Step 1: executes factorial(3)

return 3 * factorial(2)

Step 2: executes factorial(2)

Step 3: executes factorial(1)

Step 4: executes factorial(0)

return 1

Step 5: return 1

Step 6: return 1

Step 7: return 2

Step 8: return 6
Recursive Factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Step 1: executes factorial(3)

return 3 * factorial(2)

Step 2: executes factorial(2)

return 2 * factorial(1)

Step 3: executes factorial(1)

return 1 * factorial(0)

Step 4: executes factorial(0)

return 1

Step 5: return 1

Step 6: return 1

Step 7: return 2

Step 8: return 6

Space Required for factorial(4)

Space Required for factorial(3)

Space Required for factorial(2)

Space Required for factorial(1)

Stack

Main method
Recursive Factorial

factorial(4)

Step 0: executes factorial(4)
return 4 * factorial(3)

Step 1: executes factorial(3)
return 3 * factorial(2)

Step 2: executes factorial(2)
return 2 * factorial(1)

Step 3: executes factorial(1)
return 1 * factorial(0)

Executes factorial(0)

Main method

Space Required for factorial(0)
Space Required for factorial(1)
Space Required for factorial(2)
Space Required for factorial(3)
Space Required for factorial(4)
Main method
Recursive Factorial

return 1
factorial(4)
return 4 * factorial(3)
return 3 * factorial(2)
return 2 * factorial(1)
return 1 * factorial(0)

Step 0: executes factorial(4)
Step 1: executes factorial(3)
Step 2: executes factorial(2)
Step 3: executes factorial(1)
Step 4: executes factorial(0)

Main method
3
Space Required for factorial(3)
Space Required for factorial(2)
Space Required for factorial(1)
Space Required for factorial(0)
Space Required for factorial(4)
Main method
Recursive Factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Step 1: executes factorial(3)

return 3 * factorial(2)

Step 2: executes factorial(2)

return 2 * factorial(1)

Step 3: executes factorial(1)

return 1 * factorial(0)

Step 4: executes factorial(0)

return 1

Step 5: return 1

return 1

return factorial(0)

Space Required for factorial(3)

Space Required for factorial(2)

Space Required for factorial(1)

Space Required for factorial(0)

Main method
Step 0: executes factorial(4)  
return 4 * factorial(3)  
Step 1: executes factorial(3)  
return 3 * factorial(2)  
Step 2: executes factorial(2)  
return 2 * factorial(1)  
Step 3: executes factorial(1)  
return 1 * factorial(0)  
Step 4: executes factorial(0)  
return 1  
Step 5: return 1  
Step 6: return 1  
Main method
Recursive Factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Step 1: executes factorial(3)

return 3 * factorial(2)

Step 2: executes factorial(2)

return 2 * factorial(1)

Step 3: executes factorial(1)

return 1 * factorial(0)

Step 4: executes factorial(0)

return 1

Step 5: return 1

Step 6: return 1

Step 7: return 2

returns factorial(2)
Recursive Factorial

Step 0: executes factorial(4)
return 4 * factorial(3)

Step 1: executes factorial(3)
return 3 * factorial(2)

Step 2: executes factorial(2)
return 2 * factorial(1)

Step 3: executes factorial(1)
return 1 * factorial(0)

Step 4: executes factorial(0)
return 1

Step 5: return 1
Step 6: return 1
Step 7: return 2
Step 8: return 6

Space Required for factorial(4)
Space Required for factorial(3)
Space Required for factorial(2)
Space Required for factorial(1)

Stack
Recursive Factorial

Step 9: return 24

Step 8: return 6

Step 7: return 2

Step 6: return 1

Step 5: return 1

Step 4: executes factorial(0)

return 1

factorial(4)

return 4 * factorial(3)

return 3 * factorial(2)

return 2 * factorial(1)

Step 3: executes factorial(1)

return 1 * factorial(0)

Step 2: executes factorial(2)

return 2 * factorial(1)

Step 1: executes factorial(3)

return 3 * factorial(2)

Step 0: executes factorial(4)

return 4 * factorial(3)

returns factorial(4)
In-class Exercise

• Get into groups of 4 – 5.
• Write your own recursive \texttt{int pwr()} function that takes two integers as arguments and returns the integer result.
  – What does the function prototype look like?
  – Now, write the function definition...
Demo
Revisit Variables vs. Pointers

• Value Semantics
  – Values stored directly
  – Copy of value is passed
  int i, j=2;
i=j;

• Pointer Semantics
  – Address to variable is stored
  – Copy of address is passed
  int *i, j=2;
i=&j;
What if we don’t have the $j$?

- We need to create the address space.
- How do we do this?
  - `new` type;
- For example:
  ```
  int *i;
  i = new int;   //new returns an address
  *i = 10;
  ```
Binky Pointer Video

• Watch the C++ Stanford Binky video: http://cslibrary.stanford.edu/104/

... and make sure you don’t blow binky’s head off in the future😊
Stack vs. Heap

• Static vs. Dynamic
Static vs. Dynamic

- **Static Semantics**
  - Assign address of variable
    ```
    int *i, j=2;
    i=&j;
    ```

- **Dynamic Semantics**
  - Create memory
  - Assign memory to pointer
    ```
    int *i=NULL;
    i=new int;
    *i=2;
    ```
What About Memory Leaks?

• What happens here...
...  
int main () {
    int *i=NULL; //created in main function
    while(1) {
        i = new int;
    }
}
Fixing Memory Leaks...

- What happens here...

```
int main () {
    int *i=NULL; //created in main function
    while(1) {
        i = new int;
        delete i; //free memory that i points to, preventing mem leaks
    }
}
```