CS 161
Intro to CS I

Finish Recursion/Begin Memory Model
Odds and Ends

• Assignment 5 posted
• Assignment 4 demo this week

1 credit hour of lecture/course is 3 hours outside the course.

Poor planning on your part does not constitute an emergency on mine!

KISS!
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;  // Base case
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;
factorial(4) = $4 \times 3$

= $12 \times 2$

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

factorial(4) = 4 \times 3
= 12 \times 2
= 24 \times 1

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

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

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

factorial(4)

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

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

factorial(4) = 4 * factorial(3)
Computing Factorial Recursively

factorial(4) = 4 * factorial(3)
= 4 * (3 * factorial(2))
factorial(0) = 1;
factorial(n) = n*factorial(n-1);

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

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

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

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

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))))
Computing Factorial Recursively

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

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

You have to have shallow depth but too many recursive calls.
Recursive Factorial

factorial(4)

Executes factorial(4)

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)

Step 5: return 1

Step 6: return 1

Step 7: return 2

Step 8: return 6

Step 9: return 24
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

Step 9: return 24

Main method

Space Required for factorial(4)

Stack

Main method
Recursive Factorial

factorial(4)

return 4 * factorial(3)

Step 0: executes factorial(4)

step 1: executes factorial(3)

return 3 * factorial(2)

Executes factorial(2)

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)

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)

Step 4: executes factorial(0)

Step 5: return 1

Step 6: return 1

Step 7: return 2

Step 8: return 6

Step 9: return 24

Executes factorial(1)

Space Required for factorial(4)

Space Required for factorial(3)

Space Required for factorial(2)

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)

Step 4: executes factorial(0)

Step 5: return 1

Step 6: return 1

Step 7: return 2

Step 8: return 6

Step 9: return 24

Executes factorial(o)
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

returns 1

Stack

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

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 5: return 1
Step 4: executes factorial(0)
return 1
Recursive Factorial

factorial(4)
  \[ \text{Step 0: executes factorial(4)} \]
  \[ \text{return } 4 \times \text{factorial(3)} \]
  \[ \text{Step 1: executes factorial(3)} \]
  \[ \text{return } 3 \times \text{factorial(2)} \]
  \[ \text{Step 2: executes factorial(2)} \]
  \[ \text{return } 2 \times \text{factorial(1)} \]
  \[ \text{Step 3: executes factorial(1)} \]
  \[ \text{return } 1 \times \text{factorial(0)} \]
  \[ \text{Step 4: executes factorial(0)} \]
  \[ \text{return 1} \]
  \[ \text{Step 5: return 1} \]
  \[ \text{Step 6: return 1} \]
  \[ \text{returns factorial(1)} \]

Space Required for factorial(4)
Space Required for factorial(3)
Space Required for factorial(2)
Space Required for factorial(1)
Space Required for factorial(0)
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

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)
returns factorial(3)

Step 5: return 1

Step 6: return 1

Step 7: return 2

Step 8: return 6

Stack

Space Required for factorial(4)

Main method
Recursive Factorial

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)
Step 5: return 1
Step 6: return 1
Step 7: return 2
Step 8: return 6
Step 9: return 24

returns factorial(4)
In-class Exercise #4

• Get into groups of 4 – 5.

• Write your own recursive \textit{int} \textit{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...
```cpp
#include <iostream>

using namespace std;

int pwr(int base, int exp) {
    int result = 1;
    for (int i = 0; i < exp; i++) {
        result = result * base;
    }
    return result;
}

int pwr_r(int base, int exp) {
    if (exp == 0) // base case
        return 1;
    else
        return base * pwr_r(base, exp - 1); // get us closer to base case
}

int main() {
    cout << pwr(2, 10000000) << endl;
    cout << pwr_r(2, 10000000) << endl; // too deep of recursion, blow stack
    return 0;
}
```
Stack vs. Heap

- Static vs. Dynamic

Created/memory is known at compile time. Created at runtime. You control stack. Nameless.

Initial item has a name.
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...

```c
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;
        cout<<i; //free memory that i points to, preventing mem leaks
        delete i;
    }
}