# Recursion

Computer Science Department
University of Central Florida

*COP 3502 – Computer Science I*

# Recursion

- What is Recursion?
  - Powerful, problem-solving strategy
    - "yeah, that tells us a whole lot"
    - </sacrasm_off>
  - In plain English:
    - Recursion: the process a procedure goes through, when one of the steps of the procedure involves rerunning the entire procedure
      - Example: say that some procedure has 4 steps
      - The 3rd step instructs you to run the entire procedure again
      - Each time you get to the third step, you have to start anew
      - This goes on, potentially, infinitely
      - And this is an example of Recursion

# Recursion: Ex of Thinking Recursively

**Strategy for processing nested dolls:**

INITIATE FUNCTION “Open All Dolls”

if there is only one doll

(1) you’re done! Play with the doll.

else

(1) open the outer doll

(2) Process the inner nest in the same way

*This part is the “recursion”*

# Recursion

- What is Recursion?
  - From the programming perspective:
    - A **recursive** function is one that contains a call to its own self
    - Example: we know that we are allowed to call function B from within function A
    - Also, you are allowed to call function A from within function A!
      - This is recursion
  - Note:
    - This could go on for infinity as function A keeps calling function A
      - So we must have a way to exit the function!

# Recursion Example w/o terminate

- Example of recursion without a terminating condition.  Just keeps going and going and…

```
#include <stdio.h>

void print();  // This is just a cheesy function that prints something

int main() {
        print();  // Here we call the cheesy function
        system("PAUSE");

        return 0;
}

void print() {

        printf(“Example of recursion WITHOUT a stopping case.\n");
        print();  // And here is the recursive function call
}
```

# Recursion

- What is Recursion?
  - From the programming perspective:
  - Recursion solves large problems by **reducing** them to **smaller** problems of the **<u>same form</u>**
  - Again, recursion is a function that invokes itself
    - Basically **<u>splits</u>** a problem into <u>one or more SIMPLER versions of itself</u>
    - And we must have a way of stopping the recursion
    - So the function must have some sort of calls or conditional statements that can actually terminate the function

# Recursion

- Programming example:
  - Let us write a program that counts down from 10 and then prints BLAST OFF!
  - How would we do this iteratively?

```
#include <stdio.h>

int main(void) {
     int i;
     for (i = 10; i > 0; --i)
          printf("%d!   ", i);
     printf("\nBLAST OFF!\n");
}
```

  - This program prints:
    - 10! 9! 8! 7! 6! 5! 4! 3! 2! 1!
      BLAST OFF!

# Recursion

- How do we do this recursively?
  - We need a function that we will call
    - And this function will then call itself
      - until the stopping case

```
#include <stdio.h>

void   count_down(int n);

int main(void) {
     count_down(10);
     return 0;
}
```

**Here’s the Count Down Function**

```
void count_down(int n){
     if (n>0) {
          printf(“%d!    ”, i);
          count_down(n-1);
     }
     else
          printf(“\nBLAST OFF!\n”);
}
```

- Once again, this program prints:
  - 10!  9!  8!  7!  6!  5!  4!  3!  2!  1!  BLAST OFF!

# Recursion

- Program Details:
  - So what's going on here in this program?
    - The first line of the main program calls the function count_down, with 10 as the input
      - Think of this as starting a new "mini" program
    - When count_down(10) runs, what happens?
      - Execution flows into the first IF statement
        - Cause 10 is surely greater than 0.
      - After printing "10!", the function count_down then CALLS ITSELF with count_down(9)
        - Think of this as starting another "mini" program
      - Again, execution flows into the first IF statement
        - Cause 9 is surely greater than 0.
      - This new, mini program then prints "9!" and calls itself with count_down(8)

# Recursion

- Program Details:
  - So what's going on here in this program?
    - This continues until we get to the mini program called count_down(1)
      - This mini program will print "1!"
        - Cuz, again, 1 is greater than 0
      - And then it calls count_down(0)
    - What happens now?
      - Execution does NOT flow into the IF statement
        - 0 is NOT greater than 0
      - So execution goes into the ELSE statement
        - BLAST OFF! is printed
        - This mini program has finished
        - AND all the other function calls have finished
        - Control returns to the main program and the program ends.

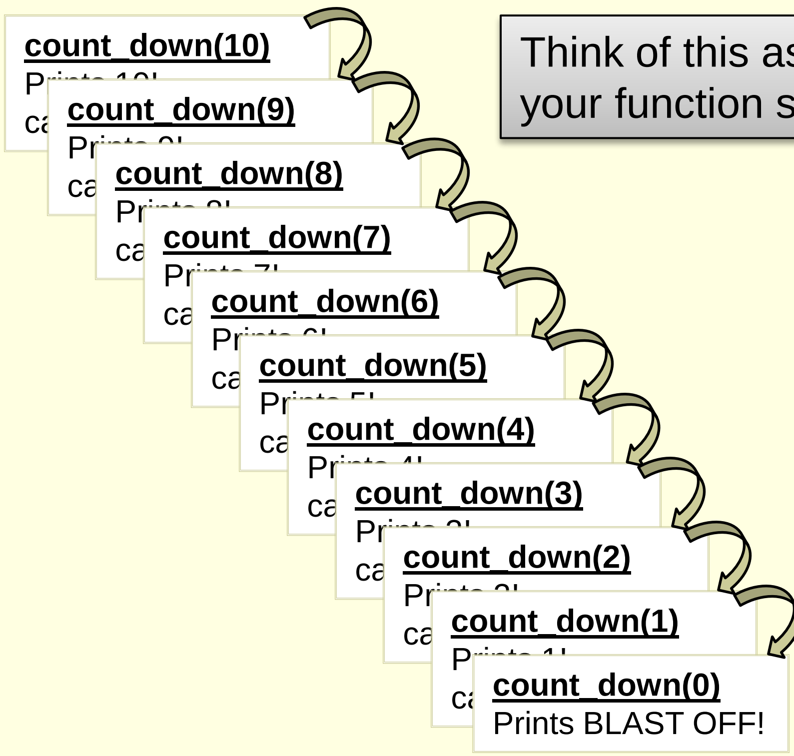
# Recursion

- Here's what's going on…in pictures

```
#include <stdio.h>

void   count_down(int n);

int main(void) {
     count_down(10);
     return 0;
}
```




Think of this as your function stack

- The Output:
  - 10!  9!   8!   7!   6!   5!   4!   3!   2!   1!
    BLAST OFF!

# Recursion - Factorial

- Count Down program
  - Not the most enlightening
    - But it gives us an idea of how recursion works
    - Let's look at another example
- Example:  Compute Factorial of a Number
  - What is a factorial?
    - 4! = 4 * 3 * 2 * 1 = 24
    - In general, we can say:
    - n! = n * (n-1) * (n-2) * … * 2 * 1
    - Also, 0! = 1
      - (just accept it!)

# Recursion - Factorial

- Example:  Compute Factorial of a Number
  - Typical iterative solution

```
int fact(int n)
{
   int p, j;
   p = 1;
   for ( j=n; j>=1; j--)
        p = p* j;
   return ( p );
}
```

Straightforward Result:
ex:  n=3

p = 1*3     // p = 3
p = 3*2     // p = 6
p = 6*1     // p = 6

# Recursion - Factorial

- Example: Compute Factorial of a Number
  - Recursive Solution
    - How do we come up with a recursive solution to this?
      - **<u>This is really the hardest part</u>**
    - You MUST figure out how you can think of the problem in a recursive manner.
      - Ask yourself: how can we rewrite this problem so that it is defined recursively?
    - Remember, we said that recursion:
      - solves large problems by **reducing** them to **smaller** problems of the **<u>same form</u>**

# Recursion - Factorial

- Example:  Compute Factorial of a Number
  - Recursive Solution
    - Mathematically, factorial is already defined recursively
      - **Note that each factorial is related to a factorial of the next smaller integer**
    - 4! = 4*3*2*1  =  4 * (4-1)!  =  4 * (3!)
    - Right?
    - Another example:
    - 10!  =  10*9*8*7*6*5*4*3*2*1
    - 10!  =  10*(9!)

This is clear right?
Since 9! clearly is equal to
9*8*7*6*5*4*3*2*1

# Recursion - Factorial

- Example: Compute Factorial of a Number
  - Recursive Solution
    - Mathematically, factorial is already defined recursively
      - **Note that each factorial is related to a factorial of the next smaller integer**
    - Now we can say, in general, that:
    - n! = n * (n-1)!
    - But we need something else
      - We need a stopping case, or this will just go on and on and on
      - NOT good!
    - We let 0! = 1

- So in “math terms”, we say
  - n! = 1 if n = 0
  - n! = n * (n-1)! if n > 0

# Recursion - Factorial

- How do we do this recursively?
  - We need a function that we will call
    - And this function will then call itself (recursively)
      - until the stopping case (n = 0)

```
#include <stdio.h>

int Fact(int n);
int main(void) {
    int factorial = Fact(10);
    printf("%d\n", factorial);
    return 0;
}
```

**Here's the Fact Function**

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

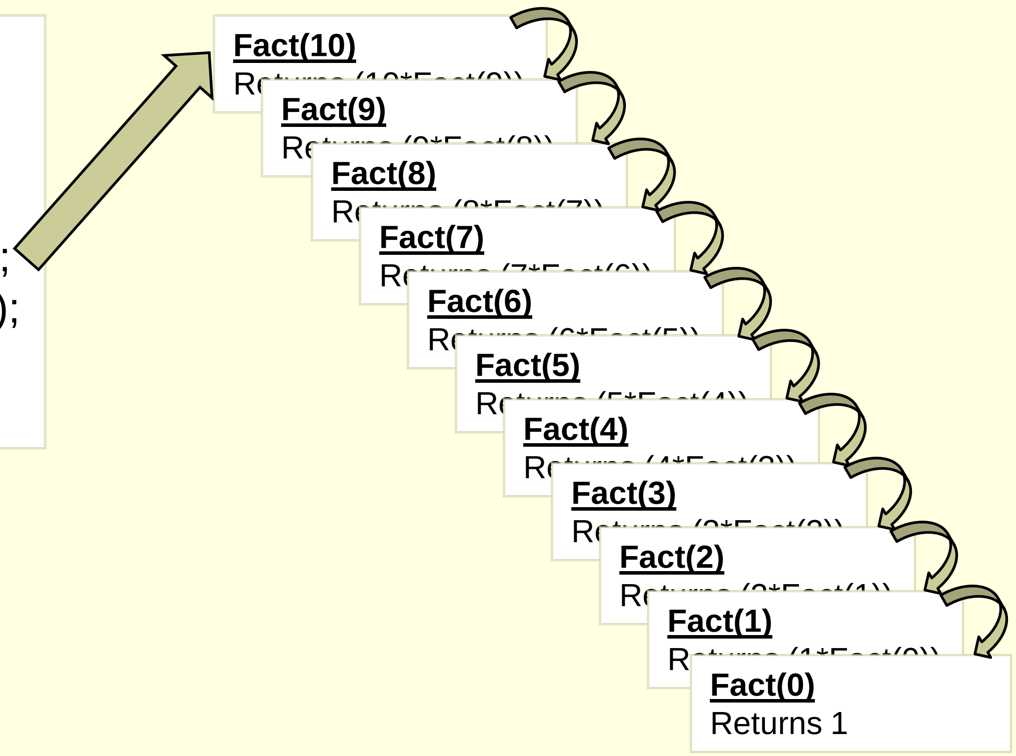
    - This program prints the result of 10*9*8*7*6*5*4*3*2*1:
      - 3628800

# Recursion - Factorial

- Here's what's going on…in pictures

```
#include <stdio.h>

int Fact(int n);
int main(void) {
     int factorial = Fact(10);
     printf("%d\n", factorial);
     return 0;
}
```

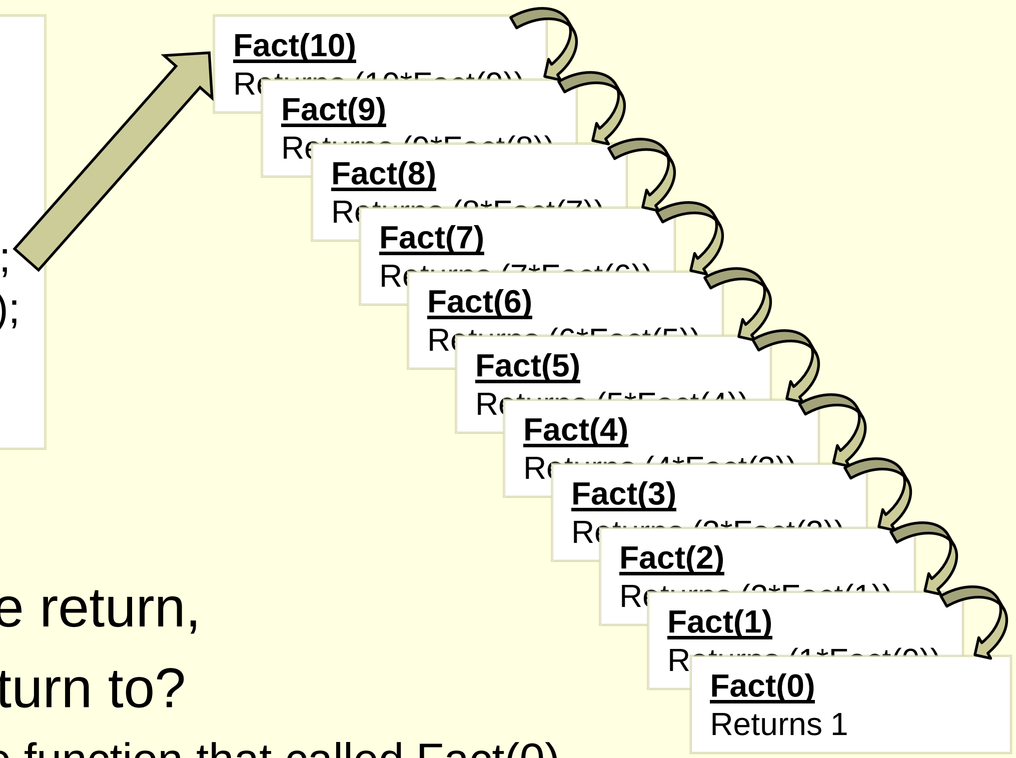


# Recursion - Factorial

- Here’s what’s going on…in pictures

```
#include <stdio.h>

int Fact(int n);
int main(void) {
    int factorial = Fact(10);
    printf(“%d\n”, factorial);
    return 0;
}
```



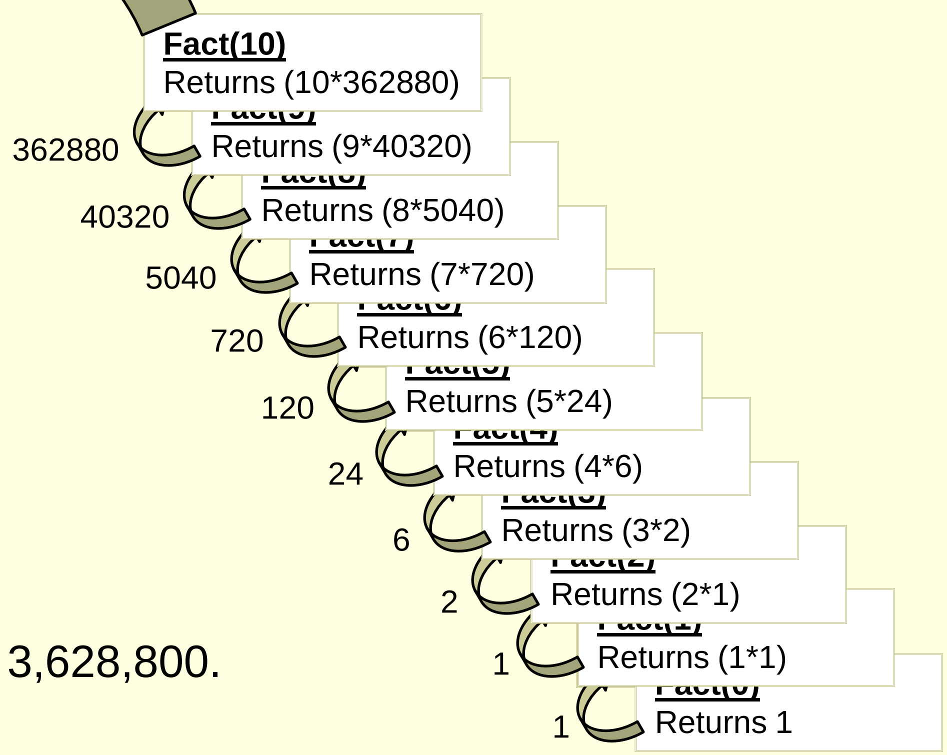

- So now when we return,
- Where do we return to?
  - We return to the function that called Fact(0)
  - We return the value, 1, into the spot that called Fact(0)

# Recursion - Factorial

- Here's what's going on…in pictures

```
#include <stdio.h>

void   Fact(int n);
int main(void) {
      int factorial = Fact(10);
      printf("%d\n", factorial);
      return 0;
}
```




- Now factorial has the value 3,628,800.

# Brief Interlude:  Human Stupidity

# Recursion

- Recursive functions
  - Are functions that calls themselves
  - Can only solve a base case
  - If not base case, the function breaks the problem into a slightly smaller, slightly simpler, problem that resembles the original problem and
    - Launches a new copy of itself to work on the smaller problem, slowly converging towards the base case
    - When computing a value, often makes a call to itself inside the **`return`** statement
    - Eventually the base case gets solved and then that value works its way back up to solve the whole problem

# Recursion

- So why use recursion?
  - Elegant solution to complex problems
    - *"To iterate is human, to recurse divine."*

      -L. Peter Deutsch
    - Yeah, we're dorks
      - Comes with the territory
      - Get over it
  - Some solutions are naturally recursive
  - Sometimes these involve writing less code and are clearer to read

# Recursion

- On the flipside, why NOT use recursion…
  - Every problem that can be solved recursively can be solved with iteration.
  - Recursive calls take up both memory and CPU time
    - Exponential Complexity – calling the Fib function uses $2^n$ function calls.
  - Trade off of High Performance vs. Good Software Engineering.

# Recursion - Fibonacci

- Fibonacci Sequence
  - Some programs are just more naturally written recursively
    - Fibonacci is one such example
  - What is the Fibonacci sequence?
    - The first two terms of the sequence are 1
    - Each of the following terms is the sum of the two previous terms
      - **1 1 2 3 5 8 13 21 34 55 89 144 …**
  - So how can we define this Fibonacci sequence:
    - Base (stopping) cases:

      fib(1) = 1

      fib(2) = 1
    - fib(n) = fib(n-1) + fib(n-2), for n > 2
      - So, fib(7), referring to the seventh Fibonacci number, which we see from the sequence above is 13, can be found by adding fib(6) + fib(5).

# Recursion - Fibonacci

- So how do we code this up recursively?
  - We need a function that we will call
    - And this function will then call itself
      - until the stopping cases (n = 1 or n = 2)

```
#include <stdio.h>

int fib(int n);
int main(void) {
    int FibNum= fib(10);
    printf("%d\n", FibNum);
    return 0;
}
```

**Here's the fib function**

```
int fib(int n) {
    if (n <= 2)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}
```

    - This program prints out the $10^{th}$ fibonacci number:
      - 55

# Recursion - Fibonacci

- Fibonacci Sequence:
  - So what was the point of this example?
    - Showed how recursive programming can truly be easier
    - Recursive solutions are often more elegant
      - Although not necessarily faster
    - And recursive solutions are often the obvious choice based on the given function definitions
  - Now that you semi-understand recursion:
    - Check out Google's search result for recursion:
      - www.google.com
      - Type in "recursion"
      - ya get it???

# Recursion

**WASN'T THAT FASCINATING!**

# Daily Demotivator

# Recursion

Computer Science Department
University of Central Florida

*COP 3502 – Computer Science I*