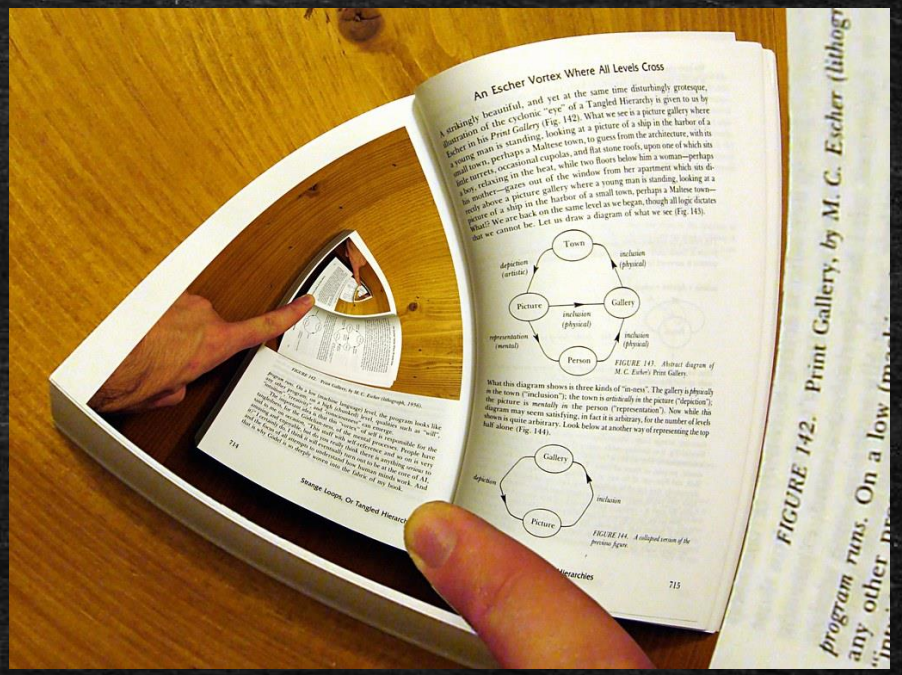
# Introducing Recursion




CS 150 – C++ Programming I
Lecture 10

# Try It Yourself

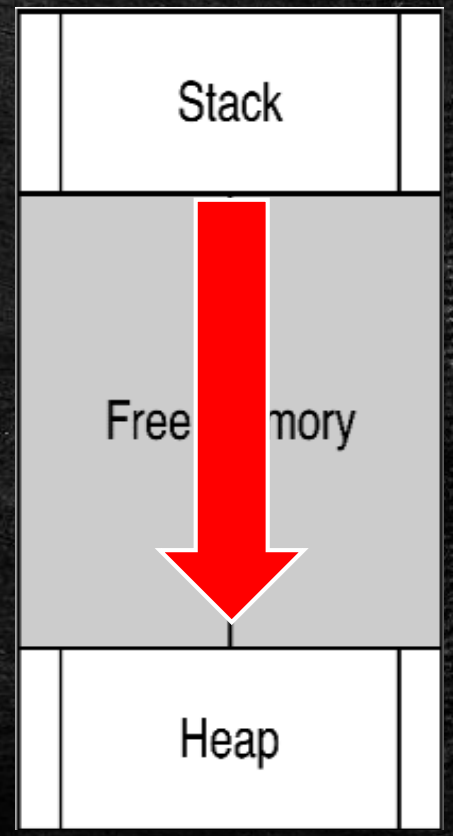
- Programming Exam 4 will ask you to write a function
  - The function will have both input and output parameters
- It will also ask you to write a group of prototypes
  - You will have to deduce the correct prototype based on the way that the function is called
  - This checks your ability to apply the data flow checklists
- Exercise: complete the prototypes along with your instructor.

# What is Recursion?

- Recursive math function: defined in terms of itself
- Factorial function *f!*
  - `3!` is `3 * 2 * 1`, `4!` is `4 * 3 * 2 * 1`, etc.
  - We can generalize so that: `n! -> n * ((n - 1)!)`
  - Try with `n = 2`
    - `2! -> 2 * ((2 - 1)!) ->`
      `(2 – 1)! -> 1 * (1 – 1)!`
- OOPS!!! Definition is circular
  - Fix with the qualification that `0!` is `1`

# Recursion in Programming

- Recursive programming is when a function calls itself
  - Try this example: *elevator.cpp*
- OOPS! Creates an endless loop effect
  - Unlike an endless loop, however, the program eventually crashes
  - Each time `elevator()` is called, a new `int` *`floor`* is placed in an area of memory called the stack
  - Eventually the stack collides with the heap
  - This is called a stack overflow



# The Way of Escape

- Recursive functions need a way to stop!
  - Called the base case. It is similar to a loops bounds
- *if* statement determines when a function calls itself
  - Creates a "loop-like" effect, but it is NOT a loop
- Exercise: stop at the penthouse (floor 25)
  - Any code after *if* appears on "way down"
  - This is called "unwinding" the stack

```
Going up. Floor: 24
Going up. Floor: 25
Going down. Floor: 25
Going down. Floor: 24
Going down. Floor: 23
```

# A factorial Example

- The factorial function:
  - n! -> n * (n-1)!
  - 0! = 1 and 1! = 1

```
int factorial(int n)
{
    if (n < 2) return 1;
    return n * factorial(n - 1);
}
```

The Base Case

The Recursive Case

# How Does it Work?

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

n

0

factorial

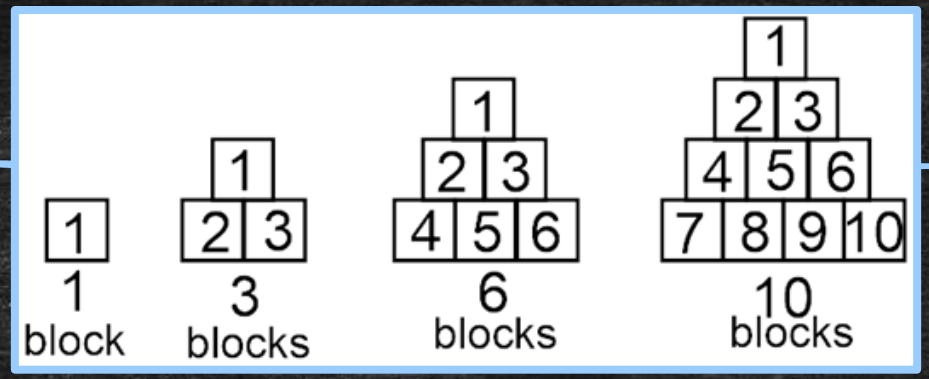
```
Enter n: 5
5! = 120
```

# The Recursive Paradigm

- Simple recursive functions fit the following pattern:

```
if (problem is sufficiently simple) {
   Directly solve the problem.
   Return the solution.
 } else {
   Split the problem up into one or more smaller
      problems with the same structure as the original.
   Solve each of those smaller problems.
   Combine the results to get the overall solution.
   Return the overall solution.
}
```

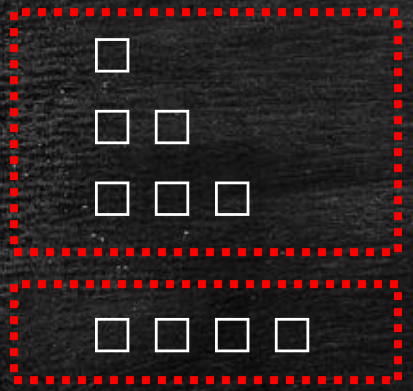
# Example




- How many blocks in a pyramid of height 10?
- A height (or base) of:
  - 1 needs one block
  - 2 needs three blocks
  - 3 needs six blocks
  - 4 needs ten blocks
- These are called triangle numbers

# A Recursive Definition

- Total blocks is base plus blocks in the next-smaller triangle



Triangle of base 4
is equal to 4 +
blocks in Triangle(3)

```
triangle(n) = n + triangle(n - 1)
```

- And the 1st triangle (base 1) = `1`. No blocks? Area of `0`
  - This is the base case or terminator

# Try It Yourself

- Exercise: complete the `triangle()` function
  - Solve it without using loops
  - Step 1: what is the base case?
    - Write an *if* statement that returns that
  - Step 2: what is the recursive case
    - Current *base* + *base* of next case
    - Call with argument that moves closer to base case
- Exercise: complete the `power()` function (no loops)
- Exercise: complete `changeXtoY()` (no loops)

# Efficiency and Helper Functions

- This function is correct, but inefficient

```
bool isPalindrome(const string& str) {
    size_t len = str.size();
    if (len > 1)
        return str[0] == str[len - 1] &&
            isPalindrome(str.substr(1, len - 2));
    return true;
}
```

  - Each call places a new string on the stack from `substr()`
- We can fix this by using a helper or wrapper
  - Pass indexes to helper, instead of substring (p 7)

# Recursion and Efficiency

- The Fibonacci sequence is 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
  - Each term is the sum of the two preceding terms
  - Solve this recursively using a function like this

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

- Unfortunately this is very inefficient
- Exercise: Fix with a helper