Recursion

- Recursive Thinking
- Recursion
- Activation Records
- Recursion vs Iteration
- Recursion Types
- Recursive Problems
  - Multibase Conversion
  - Container Classes

Recursive Thinking

- Occurs when problem can be viewed as smaller sub problems solved using the same algorithm
  - divide and conquer
  - partitioning stops when simpler problem can be solved

$$2^3 = 2 * 2^2$$
$$\downarrow$$
$$2^2 = 2 * 2^1$$
$$\downarrow$$
$$2^1 = 2$$

Recursion

- Recursive algorithm defined as
  - recursive call → current value defined in terms of previous value
  - stopping case → evaluated for given values

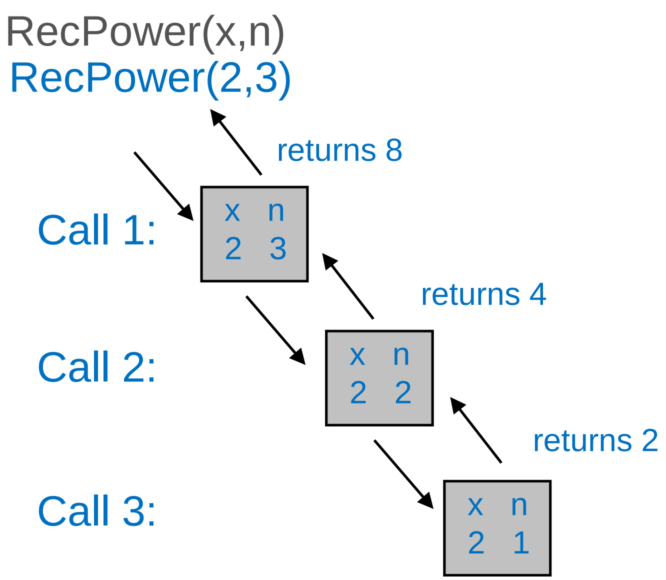
aka
base case

$$X^n = \begin{cases} X * X^{(n-1)} & n > 1 \\ X & n = 1 \end{cases}$$

Recursion

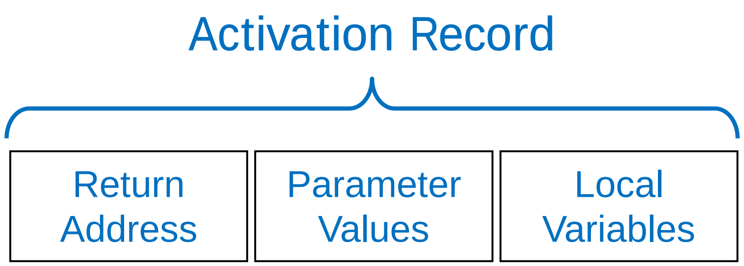
- Recursive call occurs when function calls itself
  - *new* set of local variables for each call
  - should have terminating condition

```
int RecPower(int x, int n) {
   if (n == 1) // Base case
     return x;
   else         // Recursive call
     return x * RecPower(x, n - 1);
}
```



Recursion

- Run-time stack keeps collection of activation records
  - contains
    - function return location
    - parameter values
    - local variables



Activation Records

- Recursive function makes repeated calls to itself using modified parameter list for each call
  - Activation records pushed on call stack until terminating condition is reached

Return Address Call 2	Parameters x = 2, n = 1	Local Variables
Return Address Call 1	Parameters x = 2, n = 2	Local Variables
Return Address RecPower(2,3)	Parameters x = 2, n = 3	Local Variables

Activation Records

RecPower(x,n)
RecPower(2,3)

Call 1:

x	n
2	3

Return Address	Parameters	Local Variables
RecPower(2,3)	x = 2, n = 3	

Activation Records

RecPower(x,n)
RecPower(2,3)

Call 1:

x	n
2	3

Call 2:

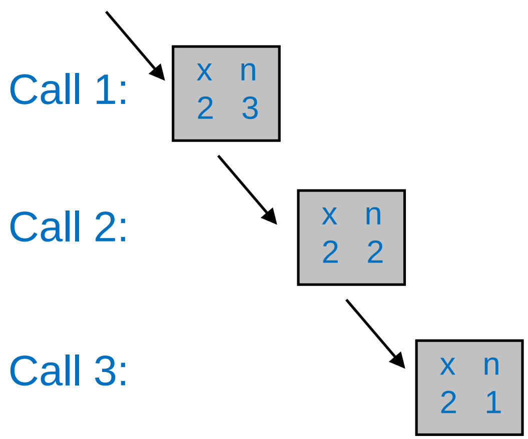
x	n
2	2

Return Address	Parameters	Local Variables
Call 1	x = 2, n = 2	

Return Address	Parameters	Local Variables
RecPower(2,3)	x = 2, n = 3	

Activation Records

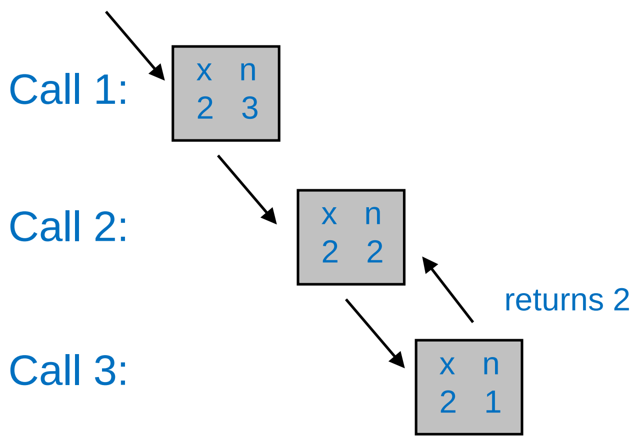
RecPower(x,n)
RecPower(2,3)



Return Address	Parameters	Local Variables
Call 2	x = 2, n = 1	
Return Address	**Parameters**	**Local Variables**
Call 1	x = 2, n = 2	
Return Address	**Parameters**	**Local Variables**
RecPower(2,3)	x = 2, n = 3	

Activation Records

RecPower(x,n)
RecPower(2,3)

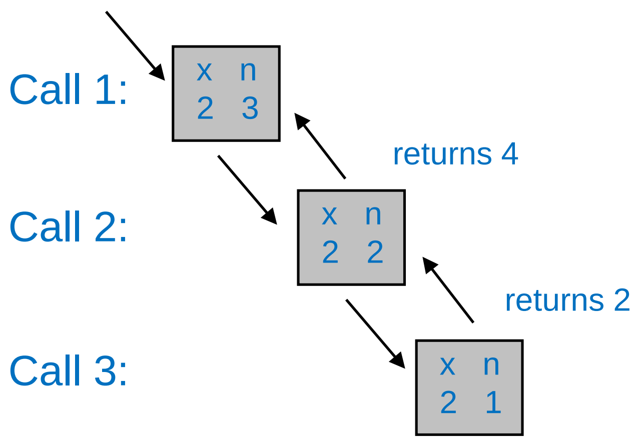


Return Address	Parameters	Local Variables
Call 1	x = 2, n = 2	

Return Address	Parameters	Local Variables
RecPower(2,3)	x = 2, n = 3	

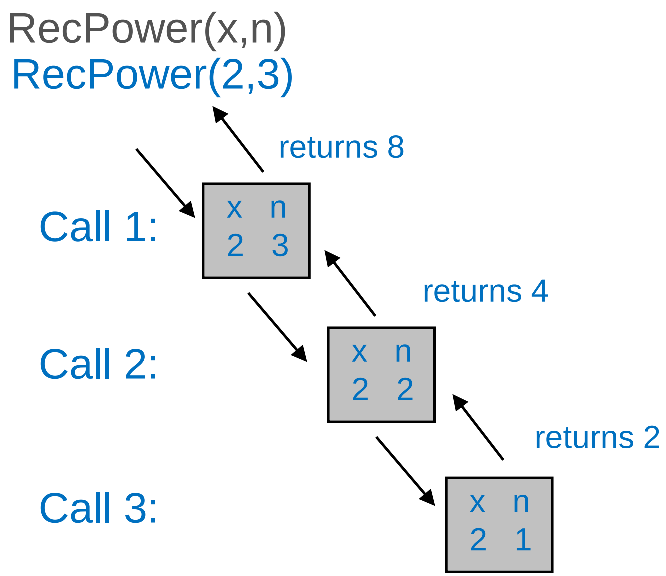
Activation Records

RecPower(x,n)
RecPower(2,3)



Return Address	Parameters	Local Variables
RecPower(2,3)	x = 2, n = 3	

Activation Records



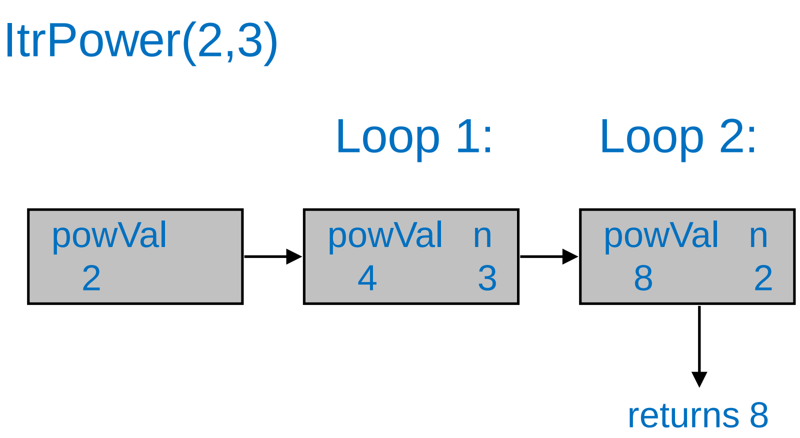
Recursive Thinking

- Best for problems where subtask is simpler version of problem
  - Must lead to stopping case or the potential of infinite recursion exists!
- Recursion *may* simplify algorithm design and coding; however, it *may* decrease efficiency with a large number of recursive calls

Recursion vs Iteration

- Iterative solution eliminates repeated function calling and parameter passing

```
int ItrPower(int x, int n) {
   int powVal;
   // Iterative
   for (powVal = x; n > 1; --n)
     powVal *= x;
   return powVal;
}
```



Recursion Types

- **tail recursion** → last action of function is to make recursive call:
  - activation records not used for significant computation
  - no statements executed after return from recursive call
  - *may be more efficient to use iteration*
- **indirect recursion** → involves a function call to a second function, that then calls the first function

Multibase Conversion

- Iterative approach using stacks
- Recursive approach

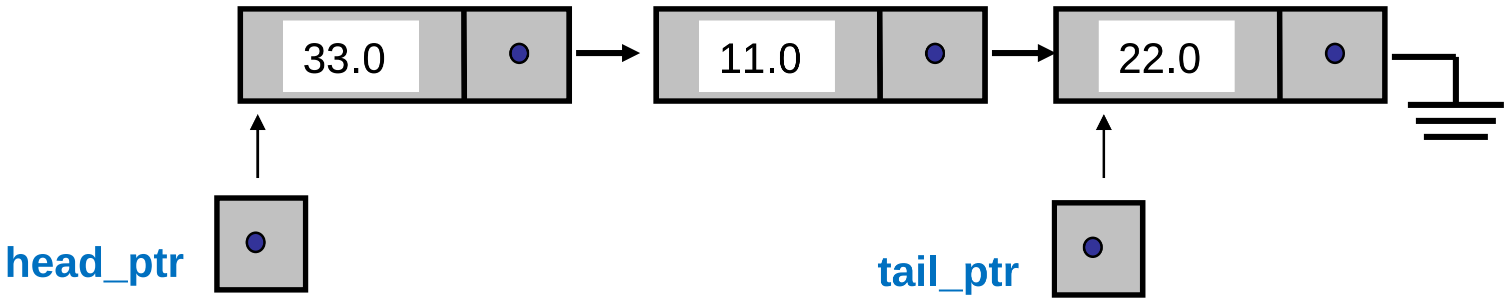
$$n_b = \begin{cases} \left(\frac{n}{b}\right)_b & n > 0 \\ \text{display } n\%b & \\ & n = 0 \end{cases}$$

```
void dispNumInBase(int num, int base) {
  string digitChar = "0123456789ABCDEF";

  if (num > 0) {
    // recursive call
    dispNumInBase(num / base, base);

    // output the remainder character
    cout << digitChar[num % base];
  }
}
```

Recursion Issues

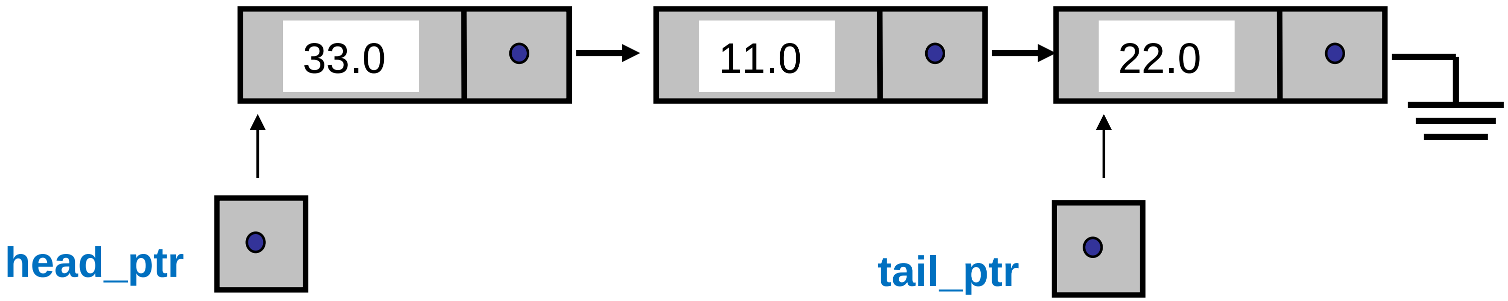
- *Could* degrade runtime performance with function call overhead
  - *may exceed available stack space*
- Useful when processing *some* collection classes
  - linked list
  - trees

Recursion and Linked Lists




```
void display_list(const node* head_ptr) {
   // iterative approach
   for (const node* cursor = head_ptr;
        cursor != nullptr;
        cursor = cursor->link())
     cout << cursor->data() << endl;
   return;
}
```

Recursion and Linked Lists



Process List	Process Node	!EndOfList	
	Process *Sub* List		
	return	EndOfList	

```
void display_list(const node* head_ptr) {
   if (head_ptr != nullptr) {
      cout << head_ptr->data() << endl;
      // recursive call
      display_list(head_ptr->link());
   }
}
```

Recursive Problems

- Factorial

Recursive Factorial Animation

$$\text{factorial}(n) = \begin{cases} 1 & n = 0 \\ n * \text{factorial}(n-1) & n >= 1 \end{cases}$$

- Tower of Hanoi
- Fractals
- Mazes