

Recursion

Learning Objectives

- Recursive void Functions
 - Tracing recursive calls
 - Infinite recursion, overflows
- Recursive Functions that Return a Value
 - Power function
- Thinking Recursively
 - Recursive design techniques
 - Binary search

Introduction to Recursion

- A function that "calls itself"
 - Said to be *recursive*
 - In function definition, call to same function
- C++ allows recursion
 - As do most high-level languages
 - Can be useful programming technique
 - Has limitations

Recursive void Functions

- Divide and Conquer
 - Basic design technique
 - Break large task into subtasks
- Subtasks could be smaller versions of the original task!
 - When they are → recursion

Recursive void Function Example

- Consider task:
- Search list for a value
 - Subtask 1: search 1st half of list
 - Subtask 2: search 2nd half of list
- Subtasks are smaller versions of original task!
- When this occurs, recursive function can be used.
 - Usually results in "elegant" solution

Recursive void Function: Vertical Numbers

- Task: display digits of number vertically, one per line
- Example call:
`writeVertical(1234);`
Produces output:
1
2
3
4

Vertical Numbers: Recursive Definition

- Break problem into two cases
- Simple/base case: if $n < 10$
 - Simply write number n to screen
- Recursive case: if $n \geq 10$, two subtasks:
 - 1- Output all digits except last digit
 - 2- Output last digit
- Example: argument 1234:
 - 1st subtask displays 1, 2, 3 vertically
 - 2nd subtask displays 4

writeVertical Function Definition

- Given previous cases:

```
void writeVertical(int n)
{
    if (n < 10)                                //Base case
        cout << n << endl;
    else
    {                                            //Recursive step
        writeVertical(n/10);
        cout << (n%10) << endl;
    }
}
```


writeVertical Trace

- Example call:
writeVertical(123);
→ writeVertical(12); (123/10)
→ writeVertical(1); (12/10)
→ cout << 1 << endl;
cout << 2 << endl;
cout << 3 << endl;
- Arrows indicate task function performs
- Notice 1st two calls call again (recursive)
- Last call (1) displays and "ends"

Recursion—A Closer Look

- Computer tracks recursive calls
 - Stops current function
 - Must know results of new recursive call before proceeding
 - Saves all information needed for current call
 - To be used later
 - Proceeds with evaluation of new recursive call
 - When THAT call is complete, returns to "outer" computation

Recursion Big Picture

- Outline of successful recursive function:
 - One or more cases where function accomplishes it's task by:
 - Making one or more recursive calls to solve smaller versions of original task
 - Called "recursive case(s)"
 - One or more cases where function accomplishes it's task without recursive calls
 - Called "base case(s)" or stopping case(s)

Infinite Recursion

- Base case MUST eventually be entered
- If it doesn't → infinite recursion
 - Recursive calls never end!
- Recall writeVertical example:
 - Base case happened when down to 1-digit number
 - That's when recursion stopped

Infinite Recursion Example

- Consider alternate function definition:

```
void newWriteVertical(int n)
{
    newWriteVertical(n/10);
    cout << (n%10) << endl;
}
```
- Seems "reasonable" enough
- Missing "base case"!
- Recursion never stops

Stacks for Recursion

- A stack
 - Specialized memory structure
 - Like stack of paper
 - Place new on top
 - Remove when needed from top
 - Called "last-in/first-out" memory structure
- Recursion uses stacks
 - Each recursive call placed on stack
 - When one completes, last call is removed from stack

Stack Overflow

- Size of stack limited
 - Memory is finite
- Long chain of recursive calls continually adds to stack
 - All are added before base case causes removals
- If stack attempts to grow beyond limit:
 - Stack overflow error
- Infinite recursion always causes this

Recursion Versus Iteration

- Recursion not always "necessary"
- Not even allowed in some languages
- Any task accomplished with recursion can also be done without it
 - Nonrecursive: called iterative, using loops
- Recursive:
 - Runs slower, uses more storage
 - Elegant solution; less coding

Recursive Functions that Return a Value

- Recursion not limited to void functions
- Can return value of any type
- Same technique, outline:
 1. One+ cases where value returned is computed by recursive calls
 - Should be "smaller" sub-problems
 2. One+ cases where value returned computed without recursive calls
 - Base case

Return a Value

Recursion Example: Powers

- Recall predefined function `pow()`:
`result = pow(2.0,3.0);`
 - Returns 2 raised to power 3 (8.0)
 - Takes two double arguments
 - Returns double value
- Let's write recursively
 - For simple example

Function Definition for power()

```
■ int power(int x, int n)
{
    if (n<0)
    {
        cout << "Illegal argument";
        exit(1);
    }
    if (n>0)
        return (power(x, n-1)*x);
    else
        return (1);
}
```

Calling Function power()

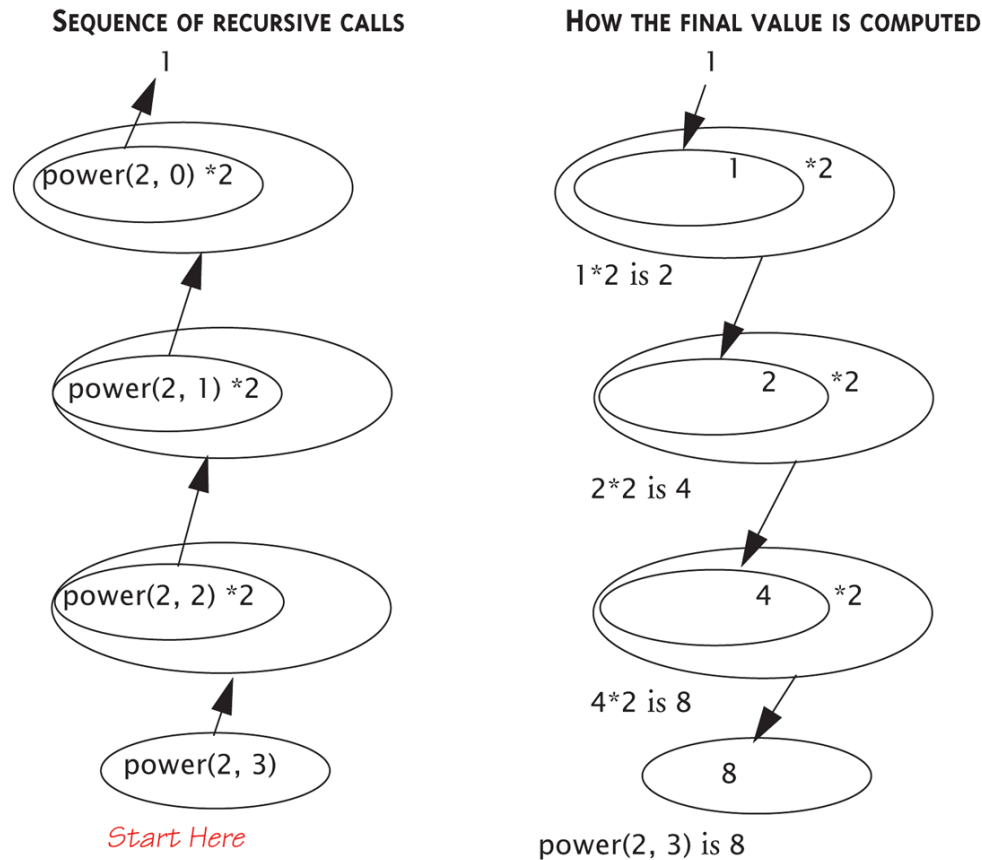
- Example calls:
- `power(2, 0);`
→ returns 1
- `power(2, 1);`
→ returns (`power(2, 0) * 2`);
→ returns 1
 - Value 1 multiplied by 2 & returned to original call

Calling Function power()

- Larger example:
power(2,3);
→ power(2,2)*2
 → power(2,1)*2
 → power(2,0)*2
 → 1
 - Reaches base case
 - Recursion stops
 - Values "returned back" up stack

Tracing Function `power()`: Evaluating the Recursive Function Call `power(2,3)`

Display 13.4 Evaluating the Recursive Function Call `power(2,3)`



Thinking Recursively

- Ignore details
 - Forget how stack works
 - Forget the suspended computations
 - Yes, this is an "abstraction" principle!
 - And encapsulation principle!
- Let computer do "bookkeeping"
 - Programmer just think "big picture"

Thinking Recursively: power

- Consider power() again
- Recursive definition of power:
power(x, n)

returns:

$\text{power}(x, n - 1) * x$

- Just ensure "formula" correct
- And ensure base case will be met

Recursive Design Techniques

- Don't trace entire recursive sequence!
- Just check 3 properties:
 1. No infinite recursion
 2. Stopping cases return correct values
 3. Recursive cases return correct values

Recursive Design Check: power()

- Check power() against 3 properties:
 1. No infinite recursion:
 - 2nd argument decreases by 1 each call
 - Eventually must get to base case of 1
 2. Stopping case returns correct value:
 - power(x,0) is base case
 - Returns 1, which is correct for x^0
 3. Recursive calls correct:
 - For $n > 1$, power(x,n) returns power(x,n-1)*x
 - Plug in values → correct

Binary Search

- Recursive function to search array
 - Determines IF item is in list, and if so:
 - Where in list it is
- Assumes array is sorted
- Breaks list in half
 - Determines if item in 1st or 2nd half
 - Then searches again just that half
 - Recursively (of course)!

Pseudocode for Binary Search

Display 13.5 Pseudocode for Binary Search

```
int a[Some_Size_Value];
```

ALGORITHM TO SEARCH a[first] THROUGH a[last]

```
//Precondition:
```

```
//a[first] <= a[first + 1] <= a[first + 2] <= ... <= a[last]
```

TO LOCATE THE VALUE KEY:

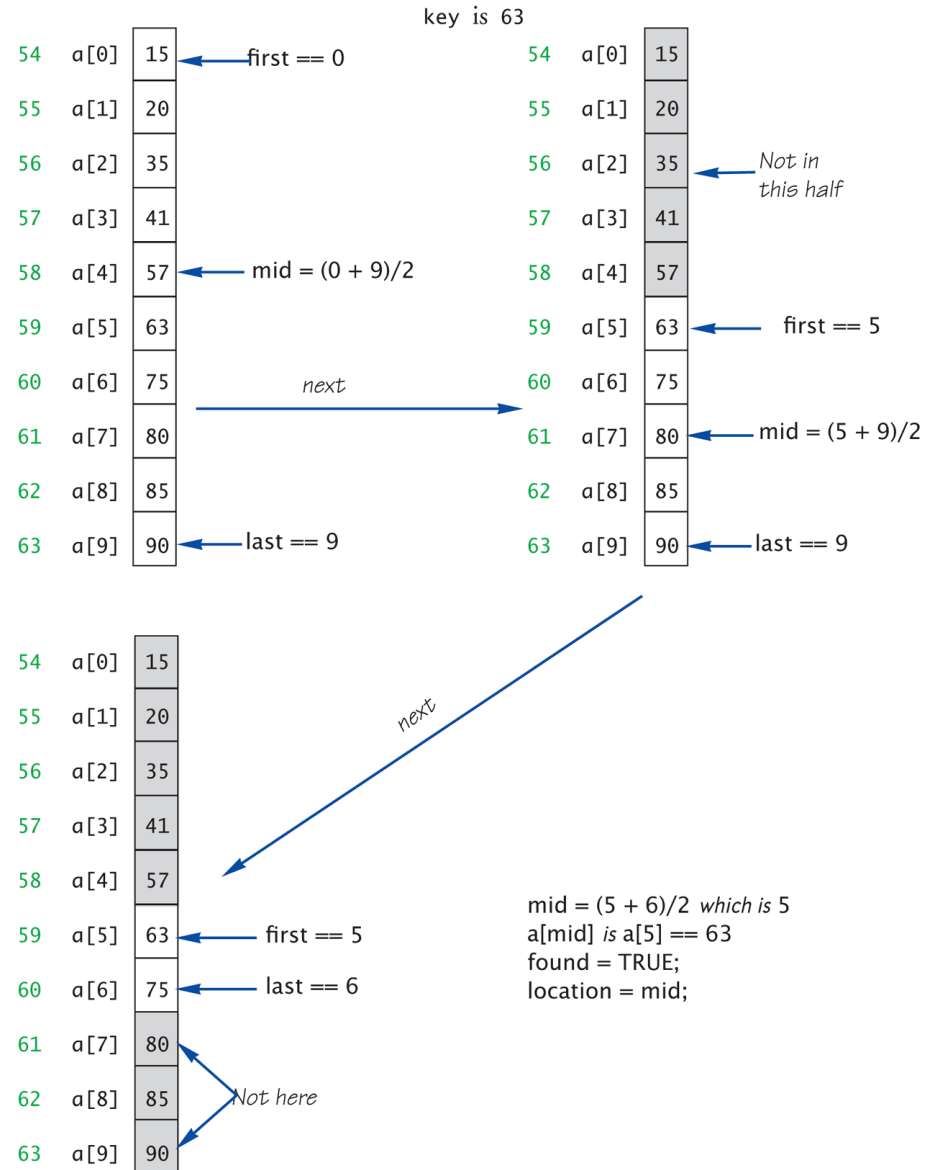
```
if (first > last) //A stopping case
    found = false;
else
{
    mid = approximate midpoint between first and last;
    if (key == a[mid]) //A stopping case
    {
        found = false;
        location = mid;
    }
    else if key < a[mid] //A case with recursion
        search a[first] through a[mid - 1];
    else if key > a[mid] //A case with recursion
        search a[mid + 1] through a[last];
}
```

Checking the Recursion

- Check binary search against criteria:
 1. No infinite recursion:
 - Each call increases first or decreases last
 - Eventually first will be greater than last
 2. Stopping cases perform correct action:
 - If $\text{first} > \text{last} \rightarrow$ no elements between them, so key can't be there!
 - If $\text{key} == a[\text{mid}] \rightarrow$ correctly found!
 3. Recursive calls perform correct action
 - If $\text{key} < a[\text{mid}] \rightarrow$ key in 1st half – correct call
 - If $\text{key} > a[\text{mid}] \rightarrow$ key in 2nd half – correct call

Execution of Binary Search: Execution of the Function search

Display 13.7 Execution of the Function search



Efficiency of Binary Search

- Extremely fast
 - Compared with sequential search
- Half of array eliminated at start!
 - Then a quarter, then 1/8, etc.
 - Essentially eliminate half with each call
- Example:
Array of 100 elements:
 - Binary search never needs more than 7 compares!
 - Logarithmic efficiency ($\log n$)

Recursive Solutions

- Notice binary search algorithm actually solves "more general" problem
 - Original goal: design function to search an entire array
 - Our function: allows search of any interval of array
 - By specifying bounds *first* and *last*
- Very common when designing recursive functions

Summary

- Reduce problem into smaller instances of same problem -> recursive solution
- Recursive algorithm has two cases:
 - Base/stopping case
 - Recursive case
- Ensure no infinite recursion
- Use criteria to determine recursion correct
 - Three essential properties
- Typically solves "more general" problem