Chapter 13
Recursion
Recursion

• A function that "calls itself"
  – In function definition, call to same function

• Divide and Conquer
  – Basic design technique
  – Break large task into subtasks

• Use recursion when subtasks are smaller versions of the original task
Recursive Function Example

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

- Function power():
  
  ```
  result = power(2,3);
  ```

  - Returns 2 raised to power 3

- Can we use recursion for this problem?

  - Can it be divided into subtasks which are smaller versions of the original task?
Function Definition for power()

- int power(int x, int n) {
  if (n < 0) {
    cout << "Illegal argument";
    exit(1);
  }
  if (n == 1)
    return 1;
  return (x * power(x, n-1));
}
Calling Function power()

• 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()

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

Sequence of recursive calls:

1. power(2, 0) * 2
2. power(2, 1) * 2
3. power(2, 2) * 2
4. power(2, 3)

How the final value is computed:

1. 1
2. 1 * 2 is 2
3. 2 * 2 is 4
4. 4 * 2 is 8
5. power(2, 3) is 8

Start Here
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!

```c++
int power(int x, int n) {
    return (x * power(x, n-1));

    if (n < 0) {
        cout << "Illegal argument";
        exit(1);
    }
    if (n == 1)
        return 1;
}
```
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 Vs Iteration

• 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
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"
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:
     - 2\textsuperscript{nd} 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 \( \text{power}(x,\( n-1 \)) \times x \)
     - From math, we know this is correct