Recursion

- A function that "calls itself"
  - In function definition, call to same function
- Divide and Conquer
  - Basic design technique
  - Break large task into subtasks
- Subtasks could be smaller versions of the original task!
  - When they are task recursion
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 "elegant" solution

Recursion Example: Powers

• Recall predefined function pow():
  result = pow(2.0, 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 calls:
  - power(2, 0);
    \( \rightarrow \) returns 1
  - power(2, 1);
    \( \rightarrow \) returns \( (power(2, 0) \times 2) \);
      \( \rightarrow \) returns 1
    – Value 1 multiplied by 2 & returned to original call
Calling Function power()

- Larger example:
  power(2,3);
  \[ \Rightarrow power(2,2) \times 2 \]
  \[ \Rightarrow power(2,1) \times 2 \]
  \[ \Rightarrow power(2,0) \times 2 \]
  \[ \Rightarrow 1 \]
  - Reaches base case
  - Recursion stops
  - Values "returned back" up stack

---

Tracing Function power():

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

<table>
<thead>
<tr>
<th>Sequence of Recursive Calls</th>
<th>How the Final Value is Computed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( \text{power}(2,0) \times 2 )</td>
<td>( 1 \times 2 = 2 )</td>
</tr>
<tr>
<td>( \text{power}(2,1) \times 2 )</td>
<td>( 2 \times 2 = 4 )</td>
</tr>
<tr>
<td>( \text{power}(2,2) \times 2 )</td>
<td>( 4 \times 2 = 8 )</td>
</tr>
</tbody>
</table>

Start Here

power(2, 3) is 8
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!

Alternate Function Definition

• 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 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

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:
  \[ \text{power}(x, n) \]
  returns:
  \[ \text{power}(x, n - 1) \times x \]
  - 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:
     - 2\text{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 power(x,n-1)*x
     - Plug in values \rightarrow correct