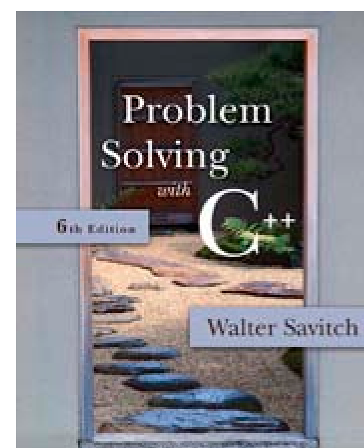
ECE244

Wael Aboulsaadat

Recursion

Acknowledgment: these slides are partially based on slides by; Prof. Schmidt from Drexel U., Prof. Shewchuk from UC Berkely, Kruse & Ryba Data Structure and Program Design in C++, Prof. Savitch Problem Solving in C++ and others



Recursive Functions for Tasks

- A recursive function contains a call to itself
- When breaking a task into subtasks, it may be that the subtask is a smaller example of the same task
  - Searching an array could be divided into searching the first and second halves of the array
  - Searching each half is a smaller version of searching the whole array
  - Tasks like this can be solved with recursive functions

A Closer Look at Recursion

- Recursive calls are tracked by
  - Temporarily stopping execution at the recursive call
    - The result of the call is needed before proceeding
  - Saving information to continue execution later
  - Evaluating the recursive call
  - Resuming the stopped execution

How Recursion Ends

- Eventually one of the recursive calls must not depend on another recursive call
- Recursive functions are defined as
  - One or more cases where the task is accomplished by using recursive calls to do a smaller version of the task
  - One or more cases where the task is accomplished without the use of any recursive calls
    - These are called base cases or stopping cases

"Infinite" Recursion

- A function that never reaches a base case, in theory, will run forever
  - In practice, the computer will run out of resources and the program will terminate abnormally

Example: Infinite Recursion

- Function write_vertical, without the base case

```
void new_write_vertical(int n)
  {
        new_write_vertical (n /10);
        cout << n % 10 << endl;
   }
```

will eventually call write_vertical(0), which will call write_vertical(0),which will call write_vertical(0), which will call write_vertical(0), which will call write_vertical(0), which will call write_vertical(0), which will call write_vertical (0), …

Stack Overflow

- Because each recursive call causes values to be placed on the stack
  - infinite recursion can force the stack to grow beyond its limits to accommodate all the activation frames required
  - The result is a stack overflow
  - A stack overflow causes abnormal termination of the program

Recursion versus Iteration

- Any task that can be accomplished using recursion can also be done without recursion
  - A nonrecursive version of a function typically contains a loop or loops
  - A non-recursive version of a function is usually called an iterative-version
  - A recursive version of a function
    - Usually runs slower
    - Uses more storage
    - <u>Often use code that is easier to write and understand</u>

Recursive Functions for Values

- Recursive functions can also return values
- The technique to design a recursive function that returns a value is basically the same as what you have already seen
  - One or more cases in which the value returned is computed in terms of calls to the same function with (usually) smaller arguments
  - One or more cases in which the value returned is computed without any recursive calls (base case)

Program Example:
A Powers Function

$2^3 = 8$

$2 * 2 * 2$

$9^2 = 81$

Program Example: A Powers Function

- To define a new power function that returns an int, such that

  int y = power(2,3);

  places 2^3 in y

  - Use this definition:

    xn = xn-1 * x

  - Translating the right side to C++ gives:

    power(x, n-1) * x

  - The base case: n = = 0 and power should return 1

Tracing power(2,1)

- int power(2, 1)
```
{
    …
    if (n > 0)
        return ( power(2, 1-1) * 2);
    else
        return (1);
}
```

Call to power(2,0)

resume

1

return 2

Function Ends

Tracing power(2,0)

```
int power(2, 0)
{
      …
      if (n > 0)
            return ( power(2, 0-1) * 2);
      else
            return (1);
}
```

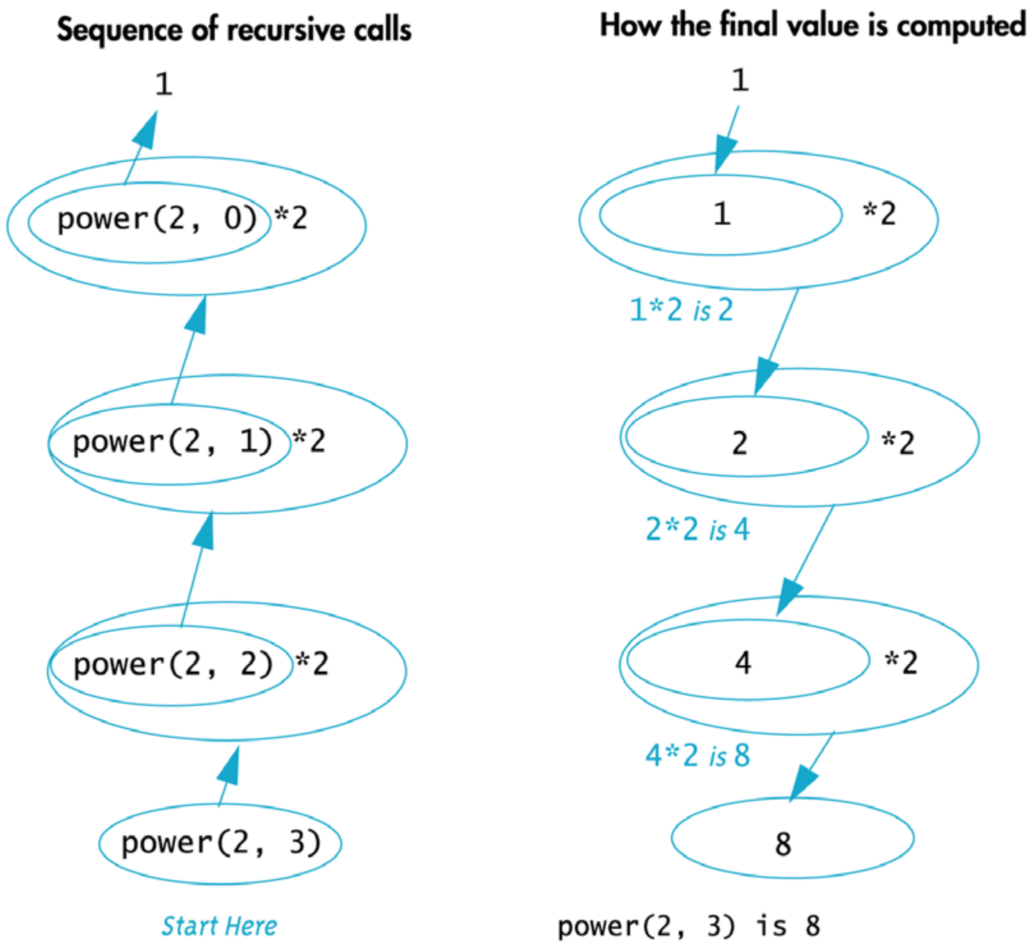
Function call ends

1 is returned

Tracing power(2, 3)

- Power(2, 3) results in more recursive calls:
  - power(2, 3) is power(2, 2) * 2
  - Power(2, 2) is power(2, 1) * 2
  - Power(2, 1) is power(2, 0) * 2
  - Power (2, 0) is 1 (stopping case)

Evaluating the Recursive Function Call power(2, 3)



The Recursive Function power

```
//Program to demonstrate the recursive function power.
#include <iostream>
#include <cstdlib>
using namespace std;

int power(int x, int n);
//Precondition: n >= 0.
//Returns x to the power n.

int main( )
{
    for (int n = 0; n < 4; n++)
        cout << "3 to the power " << n
             << " is " << power(3, n) << endl;

    return 0;
}

//uses iostream and cstdlib:
int power(int x, int n)
{
    if (n < 0)
    {
        cout << "Illegal argument to power.\n";
        exit(1);
    }

    if (n > 0)
        return ( power(x, n - 1)*x );
    else // n == 0
        return (1);
}
```

Sample Dialogue

```
3 to the power 0 is 1
3 to the power 1 is 3
3 to the power 2 is 9
3 to the power 3 is 27
```

how to approach recursion?

1. **Strategy:**
   - Rewrite the problem definition in a recursive way..
2. **Header:**
   - What info needed as input and output?
   - Write the function header.
   - Use a noun phrase for the function name
3. **Spec:**
   - Write a method specification in terms of the parameters and return value.
   - Include preconditions
4. **Base cases:**
   1. When is the answer so simple that we know it without recursing?
   2. What is the answer in these base cases(s)?
   3. Write code for the base case(s)
5. **Recursive Cases:**
   1. Describe the answer in the other case(s) in terms of the answer on smaller inputs
   2. Simplify if possible
   3. Write code for the recursive case(s)

Factorial using Recursion

$$N! = 1 * 2 * \ldots * N$$

```
int Factorial(int n) {
  int Product = 1,
      Scan    = 2;

  while ( Scan <=  n ) {
    Product  =  Product  *  Scan ;
    Scan = Scan + 1 ;
  }
  return (Product) ;
}
```

Factorial using Recursion

$$N! = 1 * 2 * \ \dots \ * N$$

```
int Factorial(int n) {
  int Product = 1,
      Scan    = 2;

  while ( Scan <=  n ) {
    Product   =   Product   *   Scan ;
    Scan = Scan + 1 ;
  }
  return (Product)  ;
}
```

```
int Factorial(int n ) {
  if ( n > 1 )
    return( n * Factorial (n-1) );
  else
    return(1);
}
```

Factorial using Recursion

$$N! = 1 * 2 * \dots * N$$