Recursion

CS 16: Solving Problems with Computers I
Lecture #17

Ziad Matni
Dept. of Computer Science, UCSB

Lecture Outline

- Recursion in C++
  - **Chapter 14**

Administrative

- 2 MORE CLASSES TO GO! ☺

M	T	W	Th	F
5/29	5/30 **LECTURE 15** HW13 due	5/31 Lab8 issued	6/1 **LECTURE 16** HW14 due Lab7 due	6/2
6/5	**6/6** **LECTURE 17** HW15 due	**6/7**	**6/8** **REVIEW** HW16 due Lab8 due	**6/9** *Last day of the quarter*

Administrative

- Homework #15 is due today
- New homework #16 (**LAST ONE!**) issued:
  Due Thursday 6/8
- Lab #8 due Thursday 6/8

Recursive Functions for Tasks

- **Recursive: (adj.) Repeating unto itself**
- **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

- For example: **Searching an array**
  - Could be divided into searching the 1st, then 2nd halves of array
  - Searching each half is a smaller version of searching the whole array

Example: The Factorial Function

Recall: x! = 1 * 2 * 3 … * x

You could code this out as either ***(the following is pseudocode)****:*

- A for-loop:

```
(for k=1; k < x; k++) { factorial *= k; }
```

- Or a recursion/repetition:

```
factorial(x) = x * factorial(x-1)
             = x * (x-1) * factorial (x-2)
             = etc…
               until you get to factorial(1)
```

Example: Recursive Formulas

- Recall from Math, that you can create a recursive formula from a sequence

Example:

- Consider the arithmetic sequence:

5, 10, 15, 20, 25, 30, …

- If I call $a_1 = 5$, then I can write the formula as:

$$\mathbf{a_n = a_{n-1} + 5}$$

Starting Point (aka Base Case)

$$a_n = a_{n-1} + 5$$

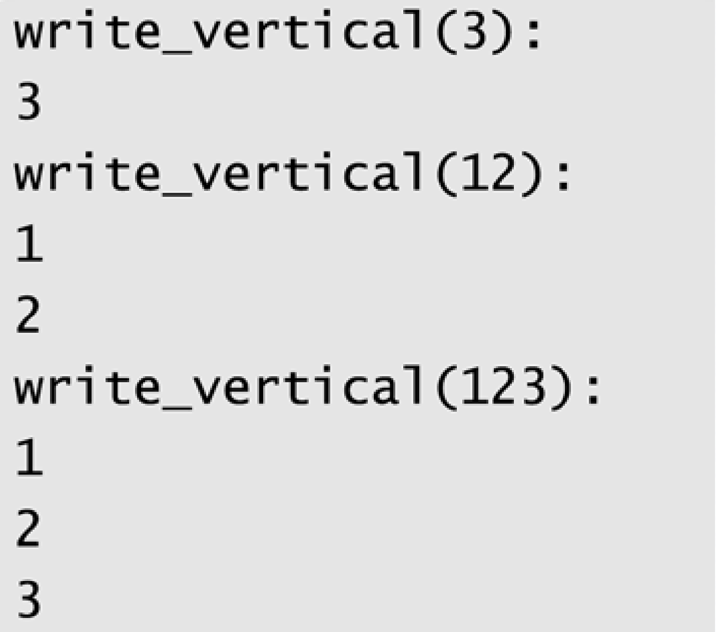
- If we start with n = 1…
  - An arbitrary value
- … then we could devise an algorithm like this:

1. If n = 1, then **return 5** to a(n)
   - This is the *base-case*
2. Otherwise, **return a(n-1) + 5**
   - This is the *recursion* (i.e. calling itself)

- Example: n = 3
  - ***According to [2]***: a(n) = a(3) = a(2) + 5 = (a(1) + 5) + 5
  - ***According to [1]***: Since a(1) = 5 , then a(3) = (5 + 5) + 5 = **15**

Case Study: Vertical Numbers

- Problem Definition:
  Write a function that takes an integer number and prints it out one digit at a time vertically :



```
write_vertical(3):
3
write_vertical(12):
1
2
write_vertical(123):
1
2
3
```

```
void write_vertical( int n );
//Precondition:  n >= 0
//Postcondition: n is written to the screen vertically
//        with each digit on a separate line
```

Case Study: Vertical Numbers

Analysis:

- Take a number, like 543.
- How do I separate the digits from each other?
  - So that I can print out **5**, then **4**, then **3**?

- Hint: Note that 543 = 500 + 40 + 3

Case Study: Vertical Numbers

Algorithm design

- *Simplest case*:
  If **n** is 1 digit long, just write the number

- *More typical case*:
  1) Output all but the last digit vertically (recursion!)
  2) Write the last digit
  - *Step 1 is a smaller version of the original task*
    - *The recursive case*
  - *Step 2 is the simplest case*
    - *The base case*

Case Study: Vertical Numbers

*The **write_vertical** algorithm (in pseudocode):*

```
void write_vertical( int n ) {

if (n < 10)  cout << n << endl;
 // n < 10 means n is only one digit


else  // n is two or more digits long
  {
        write_vertical(n with the last digit removed);
        cout << the last digit of n << endl;
  }


}
```

Case Study: Vertical Numbers

- Note that: **n / 10 (integer division)** returns **n**
  with ***just*** *the least-significant digit removed*
  - So, for example, **85 / 10 = 8** *or* **124 / 10** = **12**

- Whereas: **n % 10** returns
  the *least-significant digit of* ***n***
  - In this example, **124 % 10 = 4**

A Closer Look at Recursion

- The function **write_vertical** uses recursion
  - It simply calls itself with a different argument

- If you want to ***track*** a recursive call:
  - Temporarily stop the execution ***at*** the recursive call
  - Show or save the result of the call before proceeding
  - Evaluate the recursive call
  - Resume the stopped execution

How Recursion Ends

- Recursive functions have to stop eventually
  - One of the recursive calls must not depend on another recursive call
- Usually, that’s the last recursive call
  - What ends recursion is the **base case**
    - Also called **stopping case**

“Infinite” Recursion

- A function that never reaches a base case, in theory, *will run forever*
- In practice, the computer will often run out of resources (i.e. memory usually) and the program will terminate abnormally

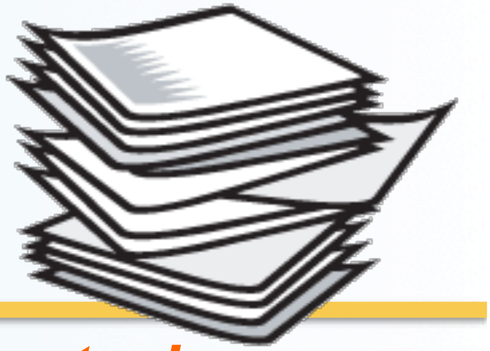
Example: Infinite Recursion

- What if we wrote the function **write_vertical**, *without the base case*

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

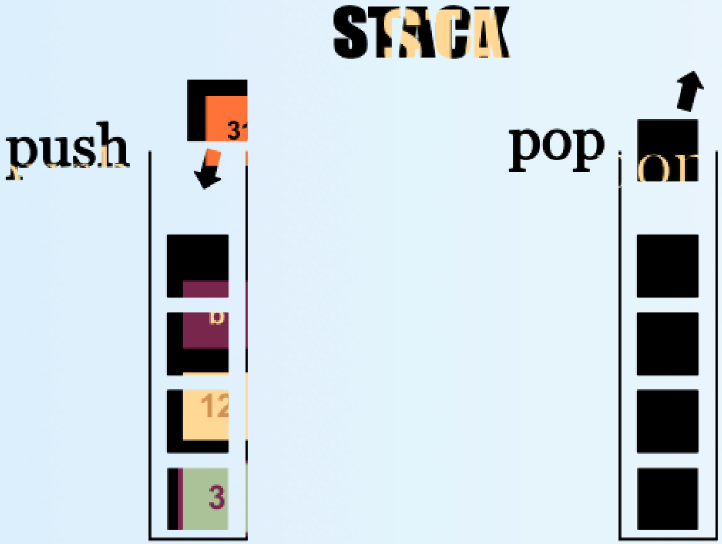
- Will *eventually* call **write_vertical(0)**,
  which will call **write_vertical(0)**,
  which will call **write_vertical(0)**,
  which will call **write_vertical(0)**, …etc…

Stacks for Recursion



- Computers use a memory structure called a ***stack*** to keep track of recursion
- **Stack**:
  a memory structure analogous to a ***stack of paper***
  - To place information on the stack, write it on a piece of paper and place it on **top** of the stack
  - To **insert *more*** information on the stack, use a clean sheet of paper, write the information, and place it on the **top** of the stack
  - To **retrieve** information, only the top sheet of paper can be read, and then thrown away when it is no longer needed

LIFO



LIFO (Last In First Out)

- This scheme of handling sequential data in a stack is called:

  Last In-First Out (**LIFO**)

- The other common scheme in CS data organization is FIFO (First In-First Out)

Stacks & Making the Recursive Call

When execution of a function def. reaches a recursive call

1. Execution is halted (paused)
2. Then, data is saved in a new place in the stack
    - It's **computer memory**, but think of it as a "clean sheet of paper"
3. The "sheet of paper" is placed *on top of the stack*
4. Then a *new* sheet is used for the recursive call
    a) A new function definition is written, and arguments are plugged into parameters
    b) Execution of the recursive call begins
5. And it goes on…

Stacks & Ending Recursive Calls

- When a recursive function call is able to complete its computation with *no* recursive calls:
- The computer retrieves the **top** “sheet of paper” from the stack
  - Resumes computation based on the information on the sheet
- When that computation ends, that sheet of paper is “discarded”
- The next sheet of paper on the stack is retrieved so that processing can resume
- The process continues until no sheets remain in the stack

Activation Frames

- Instead of “paper”, think “memory”…
- Portions of computer memory are used for the stack
  - The contents of these portions of memory is called an ***activation frame***
- Because each recursive call causes an **activation frame** to be placed on the stack
  - Infinite recursions can force the stack to grow **beyond** its limits

Stack Overflow

- Infinite recursions can force the stack to grow **beyond** its limits
- The result of this erroneous operation is called a ***stack overflow***
  - This causes abnormal termination of the program

Image from stackoverflow.com

Recursion versus Iteration

Algorithmic Truism:

- Any task that can be accomplished using recursion can also be done without recursion

- A non-recursive version typically contains loop(s) because you need to create the repetition in the process

- A non-recursive version of a repeating function is called an ***iterative-version***

- A **recursive** version of a function…
  - Usually runs slower, uses more storage
  - BUT it uses code that is *easier to write and understand*

Recursive Functions for Values

Recursive Functions for *Values*

- Recursive functions don't have to be **void** types
  - They can also return values

- The technique to design a recursive function that returns a value is basically the same as what we described…
  - One or more cases in which the value returned is computed in terms of calls to the same function with (usually) smaller arguments (i.e. recursive call)
  - One or more cases in which the value returned is computed without any recursive calls (i.e. base case)

Program Example: A Powers Function

Example: Define a new **power** function (not the one in <cmath>)

- Let it return an integer, 2^3 ,when we call the function as:
  int y = power(2,3);
  - Use the following definition:
    $x_n = x_{n-1} * x$ *i.e.* $2^3 = 2^2 * 2$
    - Note that this only works if n is a positive number
  - Translating the right side of that equation into C++ gives:
    power(x, n-1) * x
  - The base/stopping case:
    when n is 0, then power() should return 1

```
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

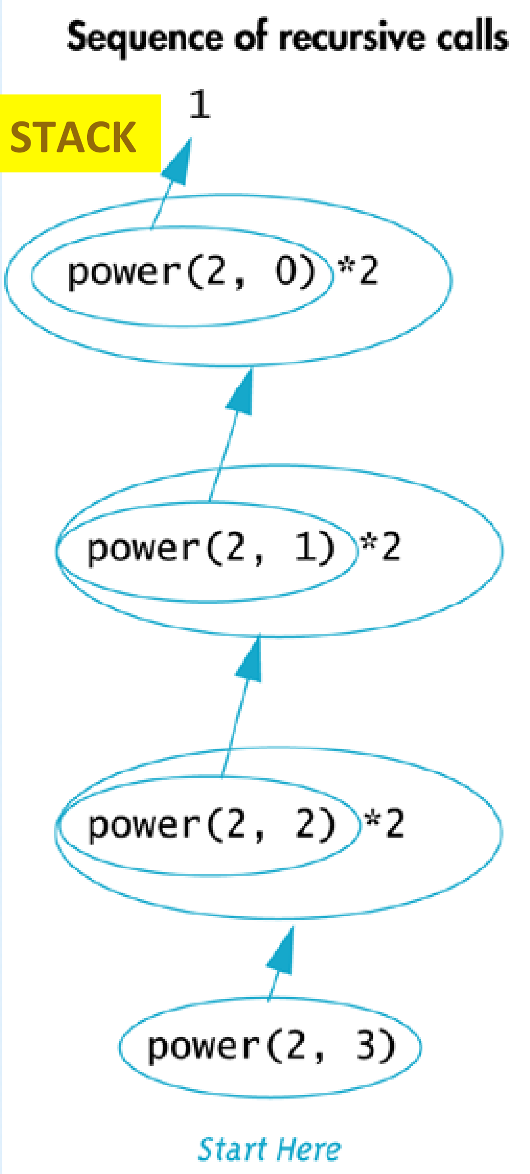
```
3 to the power 0 is 1
3 to the power 1 is 3
3 to the power 2 is 9
3 to the power 3 is 27
```

Stopping case

Tracing *power(2, 3)*

- **power(2, 3)** results in the following 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)

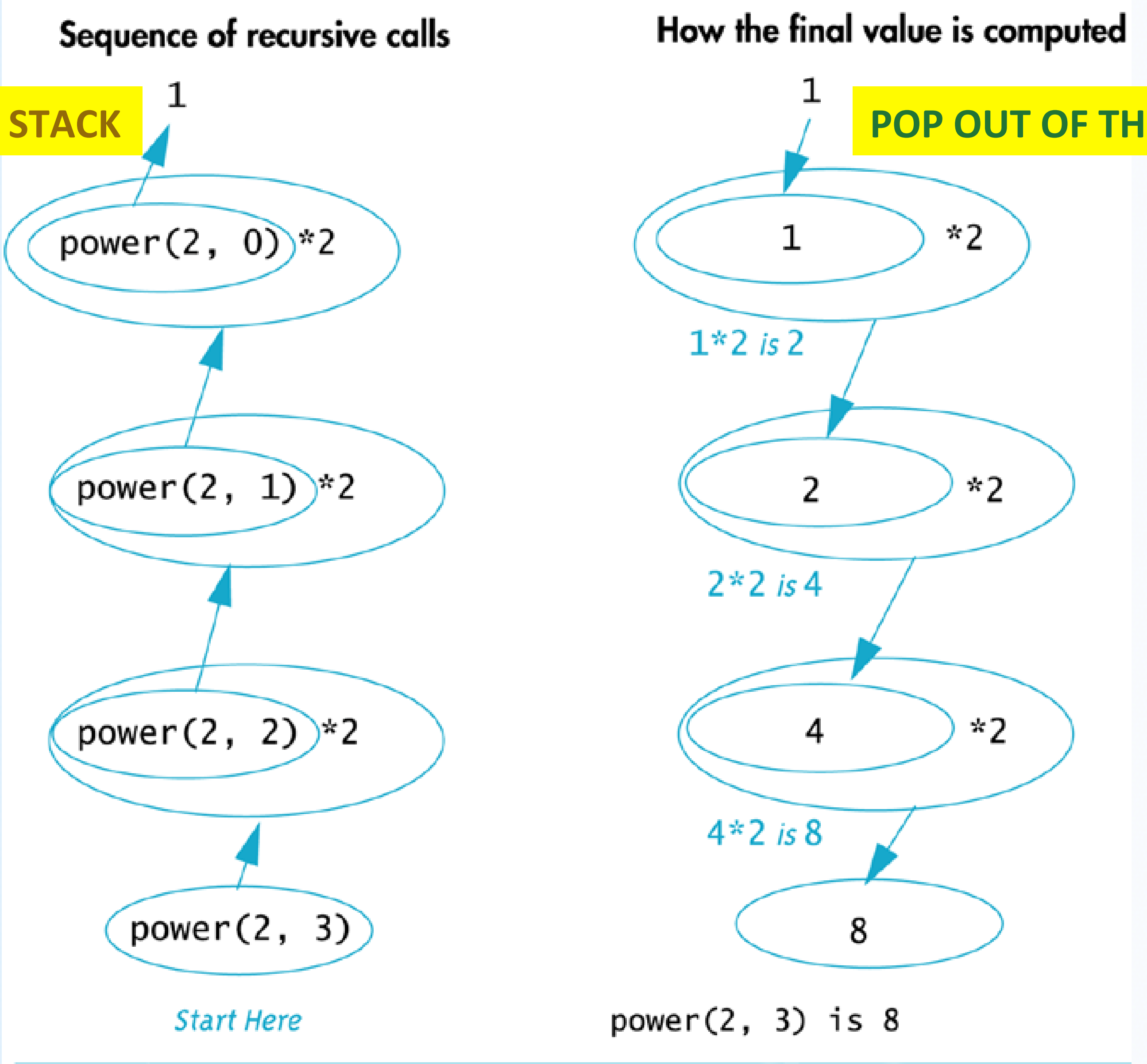
PUSH INTO THE STACK



Sequence of recursive calls

PUSH INTO THE STACK

POP OUT OF THE STACK



Thinking Recursively

- When designing a recursive function, you do not need to trace out the entire sequence of calls

- Instead just check the following:
  - That there is no infinite recursion:
    i.e. that, eventually, a stopping case is reached
  - That each stopping case returns the correct value
  - That the final value returned is the correct value

</LECTURE>