

Recursion in C++

CS 16: Solving Problems with Computers I
Lecture #16

Ziad Matni
Dept. of Computer Science, UCSB

Lecture Outline

- Linked Lists: solution to homework #13
- Recursion in C++

Translating search to C++

```
#include <iostream>
using namespace std;

struct Node
{
    int data;
    Node *link;
};

typedef Node* NodePtr;
NodePtr search(NodePtr head, int target);

int main()
{
    ...
    ...
    someptr = search(head, 6);
    ...
    return 0; }
```

```
NodePtr search(NodePtr head, int target)
{
    NodePtr here = head;

    if (here == NULL)
        return NULL;
    else
    {
        //go thru the linked list and look for target
        while ((here->data != target) &&
                (here->link != NULL))
            here = here->link;

        //the while loop stopped b/c it either
        // found target or it found nothing
        if (here->data == target)
            return here;
        else
            return NULL;
    }
}
```

Other Functions We Might Create for LLs...

- Insert node at the head
 - Print out all the values in the LL
 - Search the LL for a target
 - Insert node *at the end* of LL
 - Insert node *anywhere* in the LL
 - Delete a node according to some target value criteria
 - Sort an LL according to some target value criteria
- etc...*

A child couldn't sleep,
so her mother told a story about a little frog,
who couldn't sleep,
so the frog's mother told a story about a little bear,
who couldn't sleep,
so bear's mother told a story about a little weasel
...who fell asleep.
...and the little bear fell asleep;
...and the little frog fell asleep;
...and the child fell asleep.

Recursive Functions

- **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

Example: The Factorial Function

Recall: $x! = 1 * 2 * 3 \dots * x$

You could code this out as either:

- A 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) (then what?!?)



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, ...

- I note that I can write each number in the sequence as:

$$a_n = a_{n-1} + 5 \quad (n \text{ being the position})$$

For example: $a_4 = a_3 + 5$

$$= (a_2 + 5) + 5$$

$$= ((a_1 + 5) + 5) + 5 \quad \leftarrow \text{At this point, I need to designate } a_1 \text{ as } 5$$

$$= (5 + 5 + 5 + 5) = 20$$

The Base Case

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

- If we assume that we start the sequence at $n = 1$... (an arbitrary value)
... then we could devise an algorithm for $a(n)$ like this:

1. If $n = 1$, then **return 5** to $a(n)$

The **BASE** case

2. Otherwise, **return $a(n-1) + 5$**

The **RECURSION** (i.e. the function calling itself)

- I'll **need to know** what that base case is, otherwise I risk not ending my recursion (or not making sense of it)

Case Study: Vertical Numbers

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

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

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

Case Study: Vertical Numbers

Analysis:

- Take a decimal 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

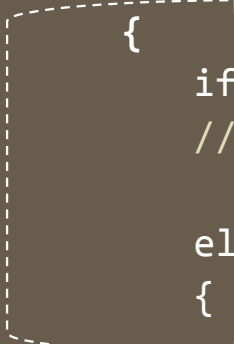
- *Simplest case* (what do we call that again???)
If n is 1 digit long, just write the number
- *More typical case:*
 - 1) Output all but the last digit vertically (recursion!)
 - 2) Write the last (least significant) digit (base case!)
 - Step 1 is a smaller version of the original task - The recursive case
 - Step 2 is the simplest case - The base case

Case Study: Vertical Numbers

The write_vertical algorithm:

```
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-least-significant-digit-removed);
        cout << the least-significant digit of n << endl;
    }
}
```



Case Study: Vertical Numbers

```
void write_vertical( int n )
{
    if (n < 10) cout << n << endl;
    else
    {
        write_vertical
            (n-without-last-digit);
        cout << LSD << endl;
    }
}
```

- **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, $124 \% 10 = 4$
- **How might we combine these in the previous function?**

Case Study: Vertical Numbers

The `write_vertical` function in C++

```
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 / 10);
        cout << (n % 10) << endl;
    }
}
```


Example Run

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

```
graph TD; A[write_vertical(543)] -- 5 --> B[cout << 3 << endl;]; A -- 1 --> C[write_vertical(54)]; C -- 4 --> D[cout << 4 << endl;]; C -- 2 --> E[write_vertical(5)]; E -- 3 --> F[cout << 5 << endl;];
```

Diagram illustrating the recursive calls for writing the number 543 vertically:

- write_vertical(543) → cout << 3 << endl; (5)
- write_vertical(54) → cout << 4 << endl; (4)
- write_vertical(5) → cout << 5 << endl; (3)

stdout:

5
4
3

“Infinite” Recursion

- A function that never reaches a base case, in theory, *will run forever*
 - Why “in theory”?

- 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)**,

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)**,

...etc...

“Infinite” Recursion

- In **practice**, the computer will often run out of *resources* (i.e. memory usually) and the program will *terminate abnormally*
 - This can happen even in non-infinite recursion situations!
(can you think of a case where this could happen?)
- So... remember that computers are machines, not Math Gods and design your (recursive) functions with that in mind!

Stacks for Recursion

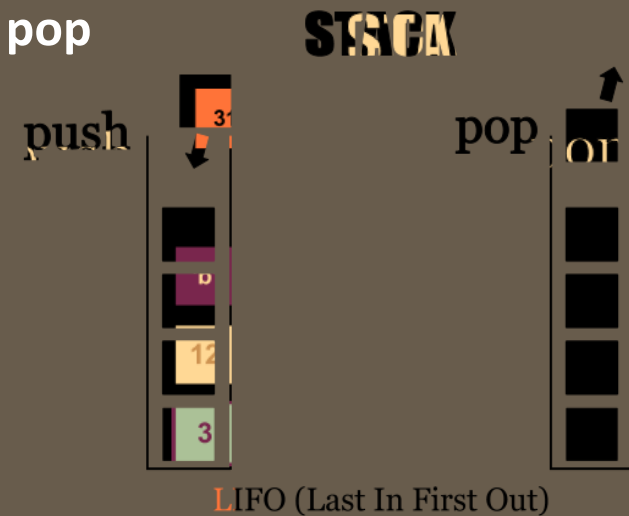


- Computers use a memory structure called a **stack** to keep track of recursion
- **Stack**: a **computer memory structure** analogous to a **stack of paper**
 - Start at zero: no papers, just knowledge of where to start (via a “stack pointer”)
 - To place data on the stack: write it on a piece of paper and place it on **top** of the stack
 - To **insert more** information on the stack: use a new sheet of paper, write the information, and place it on the **top** of the stack
 - Keep going... until you don't...
 - To **retrieve** information: you can only take the top sheet of paper
 - Then throw it away when it you're done “reading” it
 - If you want access to any paper farther down, go thru the stack to get to it

LIFO



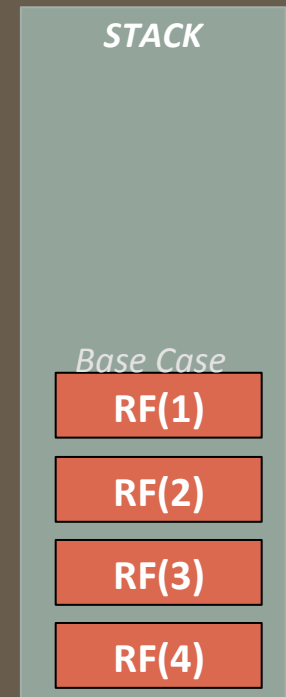
- This scheme of handling sequential data in a stack is called:
Last In-First Out (LIFO)
- When we put data in a LIFO, we call it a **push**
- When we pull data out of a LIFO, we call it a **pop**
- The other common scheme in data organization is FIFO (First In-First Out) aka *queue*



Stacks & Making the Recursive Call

When execution of a function definition reaches a **recursive call**...

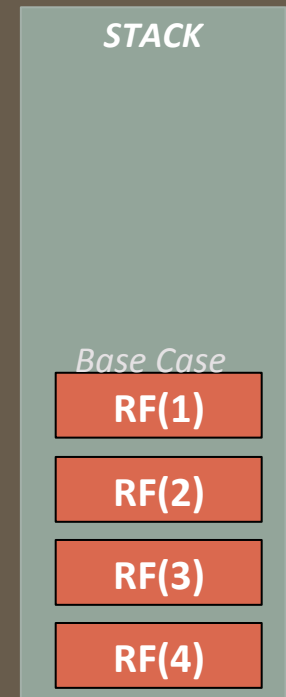
1. Execution is paused
2. Data is then saved in a new place in the stack on top
 - Remember, this is part of **computer memory**
3. Then, a *new* place in memory is “prepared” for the recursive call
 - a) A new function definition is written, arguments are plugged into parameters
 - b) Execution of the recursive call begins
4. New data is saved on top of the stack
5. Repeat until you get to the base case



Stacks & Ending Recursive Calls

When a recursive function call gets to the **base case**...

1. The computer retrieves the top memory unit of the stack
2. It resumes computation based on the information on the sheet
3. When that computation ends, that memory unit is “discarded”
4. The mem. unit on the stack is retrieved so that processing can resume
5. The process continues until the stack is back to it original status



Stack Overflow



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

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 earlier...

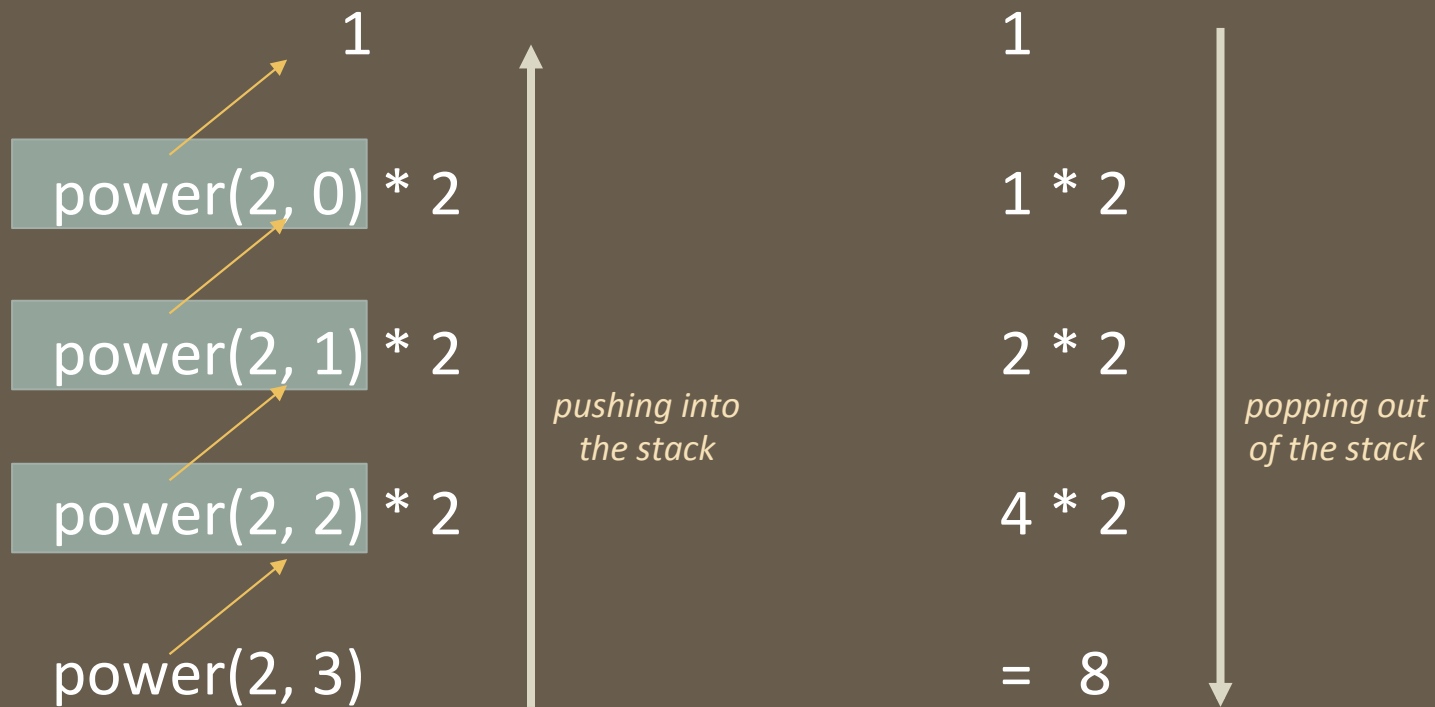
Program Example: A Powers Function

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

- Let it return an integer, **2³**, 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**
 - What is the base/stopping case?
*It's when **n = 0***
 - What should happen then?
*power() should return **1***

Tracing *power(2, 3)*


Demo!



```
int power(int x, int n)
{
    // Before you do a base-case, you should take care of
    // “illegal” operations...
    if (n < 0)
    {
        cout << “Cannot use negative powers in this function!\n”;
        exit(1);
    }

    if (n > 0)
        return ( power(x, n - 1) * x );

    else // i.e. if n == 0
        return (1);
}
```



Stopping case

Recursion versus Iteration

- Any task that can be accomplished using recursion *can also be done without recursion (using loops)*
- A **non-recursive** version of a repeating function is called an *iterative-version*

Recursion versus Iteration

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

Recursive Version

```
int power(int x, int n)
{
    int p = 1;
    for (int k = 1; k <= n; k ++ )
        p *= x;

    return(p);
}
```

Iterative Version

- A **recursive** version of a function...
 - Usually runs a little slower, takes up more memory
 - BUT it uses code that is *easier to write and understand*

YOUR TO-DOs

- ☐ Turn in Lab 9 on **Monday**
- ☐ Do HW14 by **Tuesday**
- ☐ New (AND LAST) lab next week: **Lab 10**

- ☐ Visit Prof's and TAs' office hours if you need help!

</LECTURE>