CSCI 104

Rafael Ferreira da Silva

rafsilva@isi.edu

Slides adapted from: Mark Redekopp and David Kempe

STREAMS REVIEW

Kinds of Streams

- I/O streams
  - Keyboard (`cin`) and monitor (`cout`)
- File streams – Contents of file are the stream of data
  - `#include <fstream>` and `#include <iostream>`
  - ifstream and ofstream objects
- String streams
  - `#include <sstream>` and `#include <iostream>`
  - sstream objects
- Streams support appropriate << or >> operators as well as `.fail()`, `.getline()`, `.get()`, `.eof()` member functions

C++ Stream Input

- cin, ifstreams, and stringstreams can be used to accept data from the user
  - int x;
  - cout << "Enter a number: ";
  - cin >> x;
- What if the user does not enter a valid number?
  - Check `cin.fail( )` to see if the read worked
- What if the user enters multiple values?
  - >> reads up until the first piece of whitespace
  - cin.getline() can read a max number of chars until it hits a delimeter ***but only works for C-strings (character arrays)***

```
cin.getline(buf, 80) // reads everything through a '\n'
                     // stopping after 80 chars if no '\n'
cin.getline(buf, 80, ';') // reads everything through a ';'
                          // stopping after 80 chars if no ';'
```

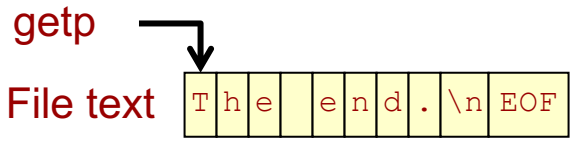
  - The <string> header defines a getline(...) method that will read an entire line (including whitespace):

```
string x;
getline(cin,x,';'); // reads everything through a ';'
```

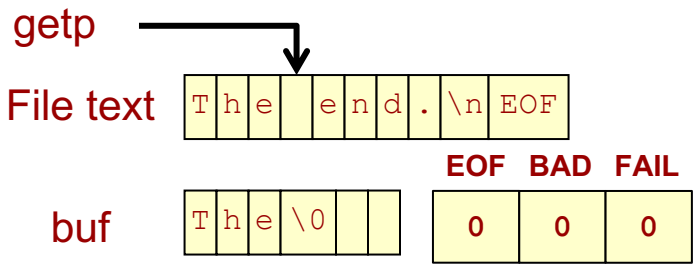
When Does It Fail

- For files & string streams the stream doesn't fail until you read PAST the EOF

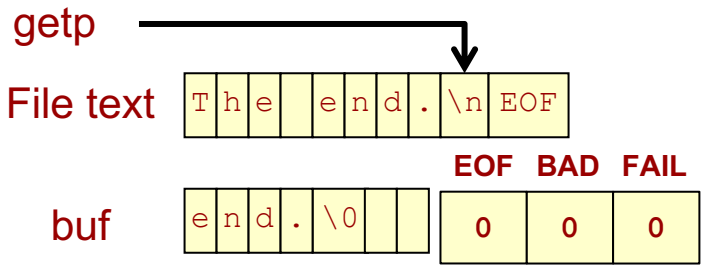
```
char buf[40];
ifstream inf(argv[1]);
```



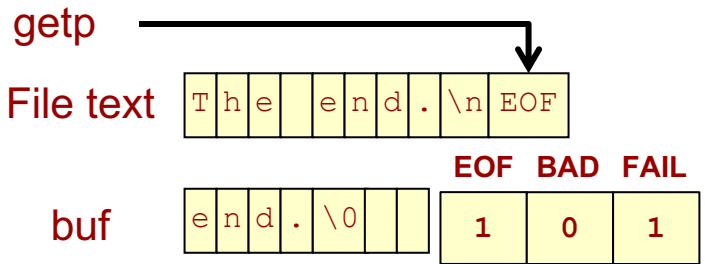

```
inf >> buf;
```




```
inf >> buf;
```




```
inf >> buf;
```



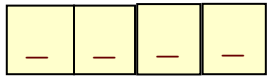
Which Option?

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
 vector<int> nums;
 ifstream ifile("data.txt");
 int x;
 while( !ifile.fail() ){
   ifile >> x;
   nums.push_back(x);
 }
 ...
}
```

data.txt

```
7 8 EOF
```

nums



```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
 vector<int> nums;
 ifstream ifile("data.txt");
 int x;
 while( 1 ){
   ifile >> x;
   if(ifile.fail()) break;
   nums.push_back(x);
 }
 ...
}
```

Need to check for failure after you extract but before you store/use

```
int x;
while( ifile >> x ){
  nums.push_back(x);
}
...
```

A stream returns itself after extraction

A stream can be used as a bool (returns true if it hasn't failed)

Choices

Where is my data?

Keyboard (use _____)

File (use _____)

String (use _____)

Do I know how many items to read?

Yes, n items Use _____

No, arbitrary Use _____

Choices

Where is my data?

- Keyboard (use iostream [cin])
- File (use ifstream)
- String (use stringstream)

Do I know how many items to read?

- Yes, n items
  Use for(i=0;i<n;i++)
- No, arbitrary
  Use while(cin >> temp) or while(getline(cin,temp))

Choices

What type of data?

Text

Integers/ Doubles

Is it delimited?

Yes

No

Yes

Choices

What type of data?

Text
(getline or >>)
getline ALWAYS returns text

Ints/Doubles
(Use >> b/c it converts text to the given type)

Is it delimited?

Yes at newlines
Use getline()

No, stop on any whitespace…use >>

Yes at special chars (';' or ',')
Use getline with 3rd input parameter (delimeter parameter)

getline() and stringstreams

- Imagine a file has a certain format where you know related data is on a single line of text but aren't sure how many data items will be on that line
- Can we use >>?
  - No it doesn't differentiate between different whitespace (i.e. a ' ' and a '\n' look the same to >> and it will skip over them)
- We can use getline() to get the whole line, then a stringstream with >> to parse out the pieces

```
int num_lines = 0;
int total_words = 0;

ifstream myfile(argv[1]);

string myline;
while( getline(myfile, myline) ){

   stringstream ss(myline);

   string word;
   while( ss >> word )
     {  total_words++; }
   num_lines++;
}

double avg =
   (double) total_words / num_lines;

cout << "Avg. words per line: ";
cout << avg << endl;
```

```
The fox jumped over the log.
The bear ate some honey.
The CS student solved a hard problem.
```

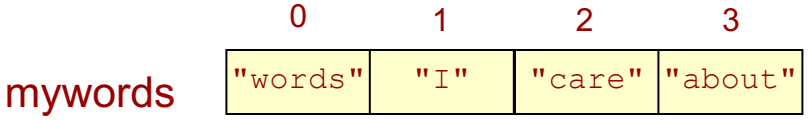
Using Delimiters

- Imagine a file has a certain format where you know related data is on a single line of text but aren't sure how many data items will be on that line
- Can we use >>?
  - No it doesn't differentiate between different whitespace (i.e. a ' ' and a '\n' look the same to >> and it will skip over them)
- We can use getline() to get the whole line, then a stringstream with >> to parse out the pieces

Text file:

```
garbage stuff (words I care about) junk
```

```
vector<string> mywords;

ifstream myfile(argv[1]);

string myline;
getline(myfile, myline, '(');
// gets "garbage stuff "
// and throws away '('

getline(myfile, myline, ')' );
// gets "words I care about"
// and throws away ')'`

stringstream ss(myline);
string word;
while( ss >> word ) {
  mywords.push_back(word);
}
```



Choosing an I/O Strategy

- Is my data delimited by particular characters?
  - Yes, stop on newlines: Use getline()
  - Yes, stop on other character: User getline() with optional 3rd character
  - No, Use >> to skip all whitespaces and convert to a different data type (int, double, etc.)
- If "yes" above, do I need to break data into smaller pieces (vs. just wanting one large string)
  - Yes, create a stringstream and extract using >>
  - No, just keep the string returned by getline()
- Is the number of items you need to read known as a constant or a variable read in earlier?
  - Yes, Use a loop and extract (>>) values placing them in array or vector
  - No, Loop while extraction doesn't fail placing them in vector

Remember: getline() always gives text/string.
To convert to other types it is easiest to use >>

RECURSION

Recursion

- Problem in which the solution can be expressed in terms of itself (usually a smaller instance/input of the same problem) ***and a base/terminating case***
- Input to the problem must be categorized as a:
  - Base case: Solution known beforehand or easily computable (no recursion needed)
  - Recursive case: Solution can be described using solutions to smaller problems of the same type
    - Keeping putting in terms of something smaller until we reach the base case
- Factorial: n! = n * (n-1) * (n-2) * … * 2 * 1
  - n! = n * (n-1)!
  - **Base case**: n = 1
  - **Recursive case**: n > 1 => n*(n-1)!

Recursive Functions

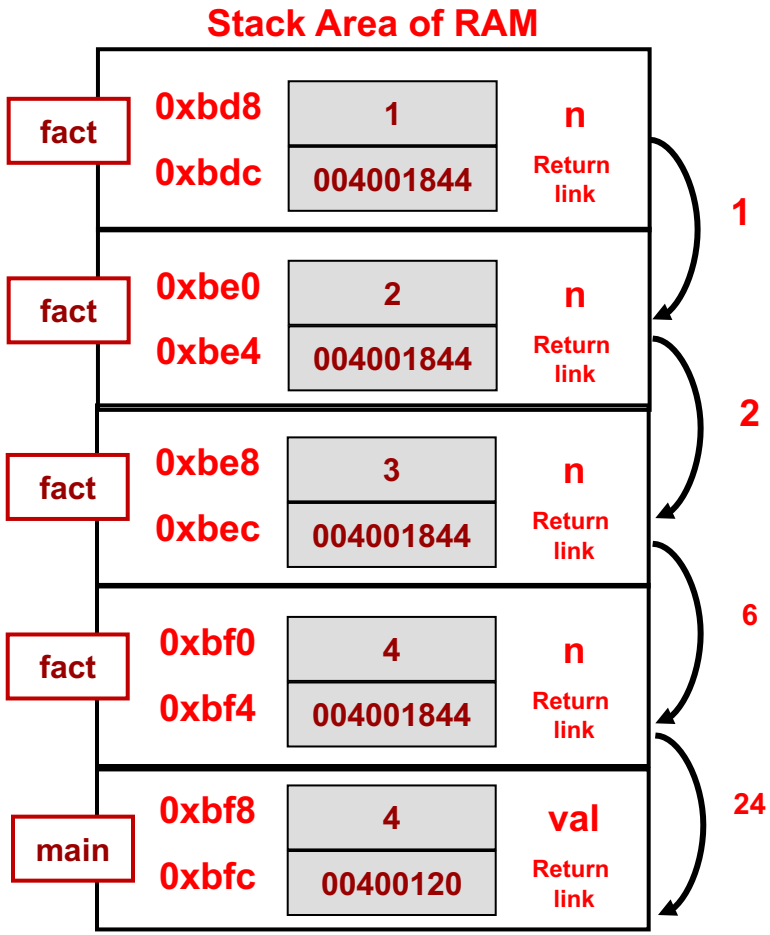
- Recall the system stack essentially provides separate areas of memory for each 'instance' of a function
- Thus each local variable and actual parameter of a function has its own value within that particular function instance's memory space

C Code:

```
int fact(int n)
{
  if(n == 1){
     // base case
     return 1;
  }
  else {
     // recursive case
     return n * fact(n-1);

  }
}
```

Recursion & the Stack

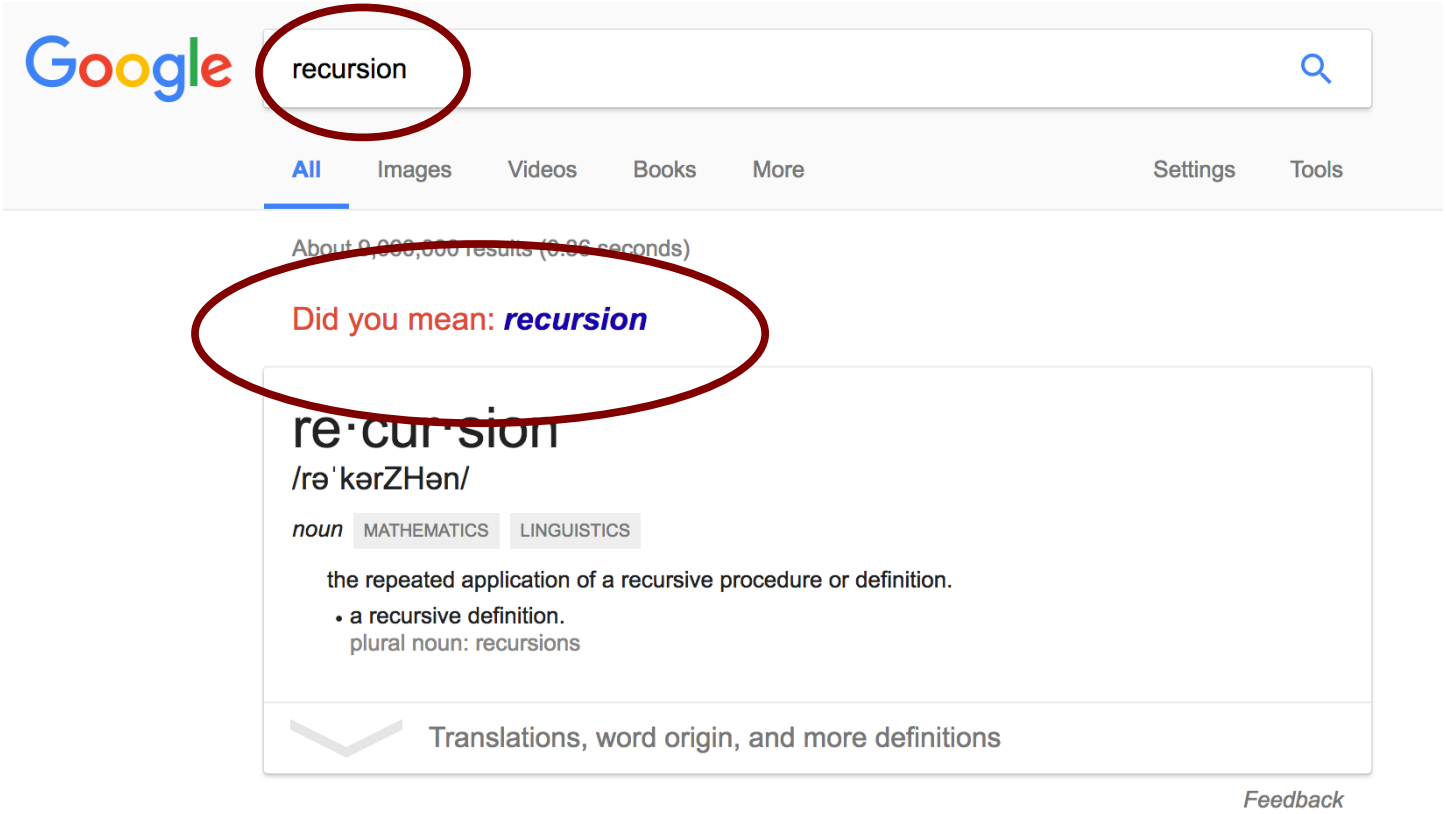
- Must return back through the each call




```
int fact(int n)
{
  if(n == 1){
     // base case
     return 1;
  }
  else {
     // recursive case
     return n * fact(n-1);

  }
}


int main()
{
  int val = 4;
  cout << fact(val) << endl;
}
```

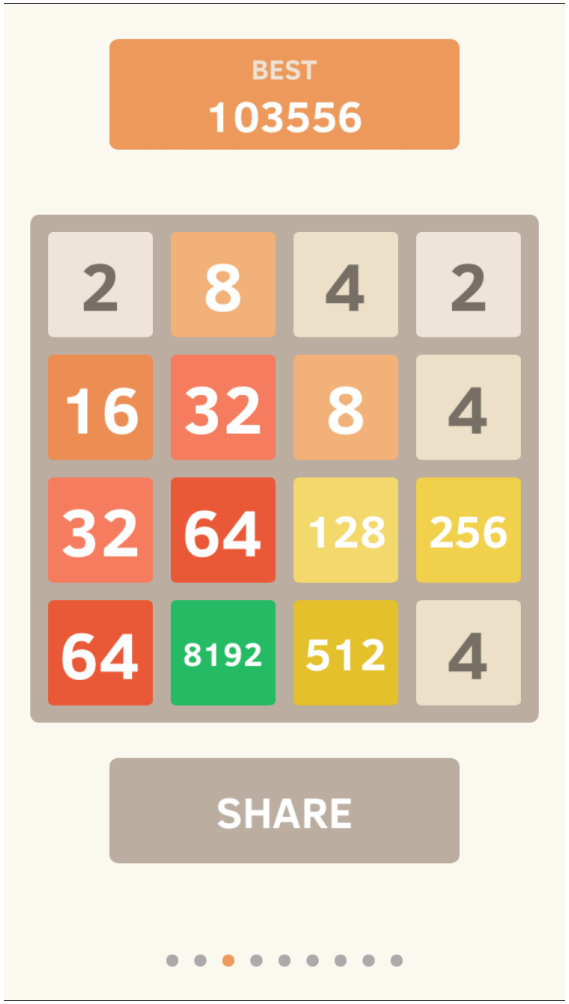
Recursion

- **Google is in on the joke too...**



Recursion

- **2048!**
  - To obtain the 2048 tile
    - Two 1024 tiles are required
    - for which, four 512 tiles are required
    - for which, eight 256 tiles are required
    - for which, 16 128 tiles are required
    - for which, 32 64 tiles are required
    - for which, 64 32 tiles are required
    - for which, 128 16 tiles are required
    - for which, 256 eight tiles are required
    - for which, 512 four tiles are required
    - for which, 1024 two tiles are required



Recursive Functions

- Many loop/iteration based approaches can be defined recursively as well

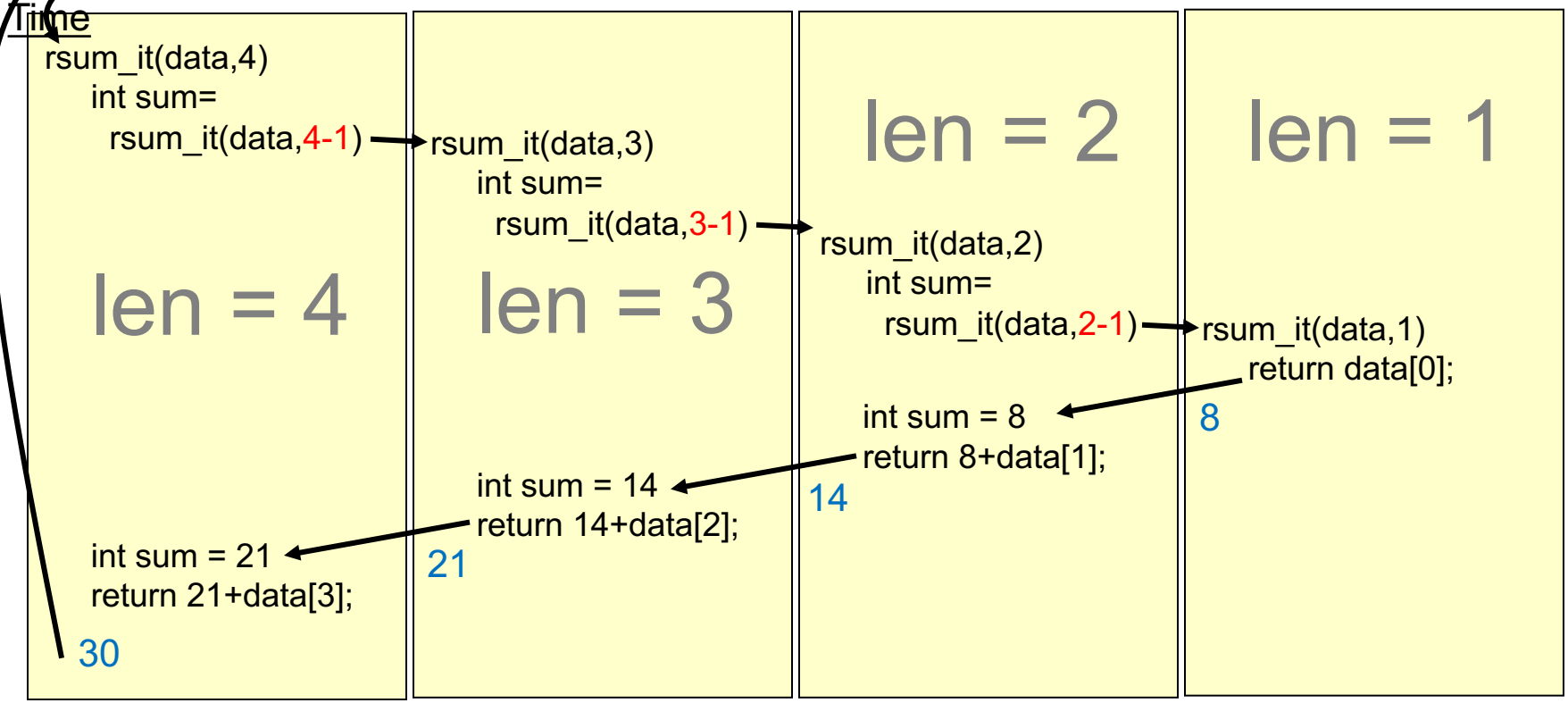
C Code:

```
int main()
{
  int data[4] = {8, 6, 7, 9};
  int size=4;
  int sum1 = isum_it(data, size);
  int sum2 = rsum_it(data, size);
}

int isum_it(int data[], int len)
{
  int sum = data[0];
  for(int i=1; i < len; i++){
    sum += data[i];
  }
}

int rsum_it(int data[], int len)
{
  if(len == 1)
    return data[0];
  else
    int sum = rsum_it(data, len-1);
    return sum + data[len-1];
}
```

Recursive Call Timeline

```
int main(){
  int data[4] = {8, 6, 7, 9};
  int size=4;
  int sum2 = rsum_it(data, size);
  ..
}
```

```
int rsum_it(int data[], int len)
{
  if(len == 1)
    return data[0];
  else
    int sum = rsum_it(data, len-1);
    return sum + data[len-1];
}
```



Each instance of rsum_it has its own len argument and sum variable

Every instance of a function has its own copy of local variables

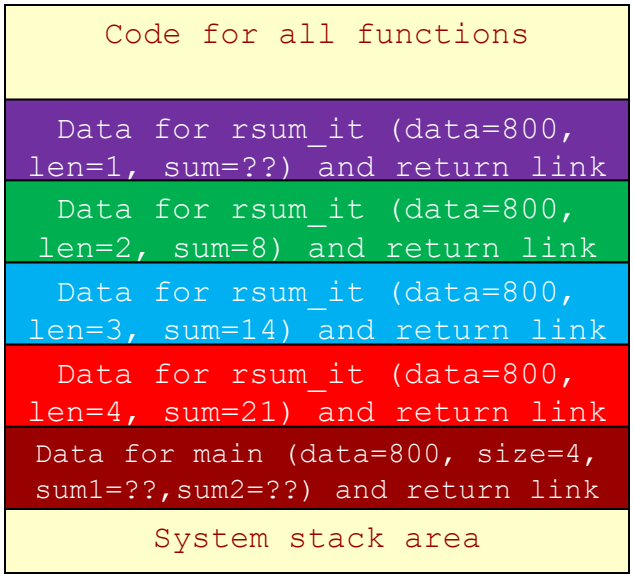
System Stack & Recursion

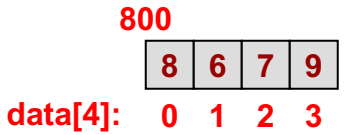
- The system stack makes recursion possible by providing separate memory storage for the local variables of each running instance of the function

```
int main()
{
  int data[4] = {8, 6, 7, 9};
  int size=4;
  int sum2 = rsum_it(data, size);
}

int rsum_it(int data[], int len)
{
  if(len == 1)
    return data[0];
  else
    int sum =
         rsum_it(data, len-1);
    return sum + data[len-1];
}
```

System Memory

(RAM)





HELPER FUNCTIONS

Exercise

- Write a recursive routine to find the maximum element of an array containing POSITIVE integers.

```
int data[4] = {8, 9, 7, 6};
```

- Primary signature:

int max(int data, int len);*

- For recursion we usually need some parameter to tell use which item we are responsible for…thus the signature needs to change. We can make a helper function.
- The client uses the original:

int max(int data, int len);*

- But it just calls:

int max(int data, int len, int curr);*

Exercise – Helper Function

- Head recursion
- Tail recursion

```
int data[4] = {8, 9, 7, 6};
```

```
// The client only wants this
int max(int* data, int len);

// But to do the job we need this
int max(int* data, int len, int curr);
```

```
// The client only wants this
int max(int* data, int len);

// But to do the job we need this
void max(int* data, int len, int curr, int& mx);
```

```
int max(int* data, int len)
{ return max(data, len, 0);
}

int max(int* data, int len, int curr)
{
  if(curr == len) return 0;
  else {
    int prevmax = max(data, len, curr+1);
    if(data[curr] > prevmax)
      return data[curr];
    else
      return prevmax;
}
```

```
int max(int* data, int len)
{ int mymax = 0;
  max(data, len, 0, mymax);
  return mymax;
}

void max(int* data, int len, int curr, int& mx)
{
  if(curr == len) return;
  else {
    if(data[curr] > mx)
      mx = data[curr];
    max(data, len, curr+1, mx);
}
```

Exercise

- We can also formulate things w/o the helper function in this case…

```
int data[4] = {8, 6, 9, 7};
```

```
int max(int* data, int len)
{
  if(len == 1) return data[0];
  else {
    int prevmax = max(data, len-1);
    if(data[len-1] > prevmax)
      return data[len-1];
    else
      return prevmax;
  }
}
```

GENERATING ALL COMBINATIONS

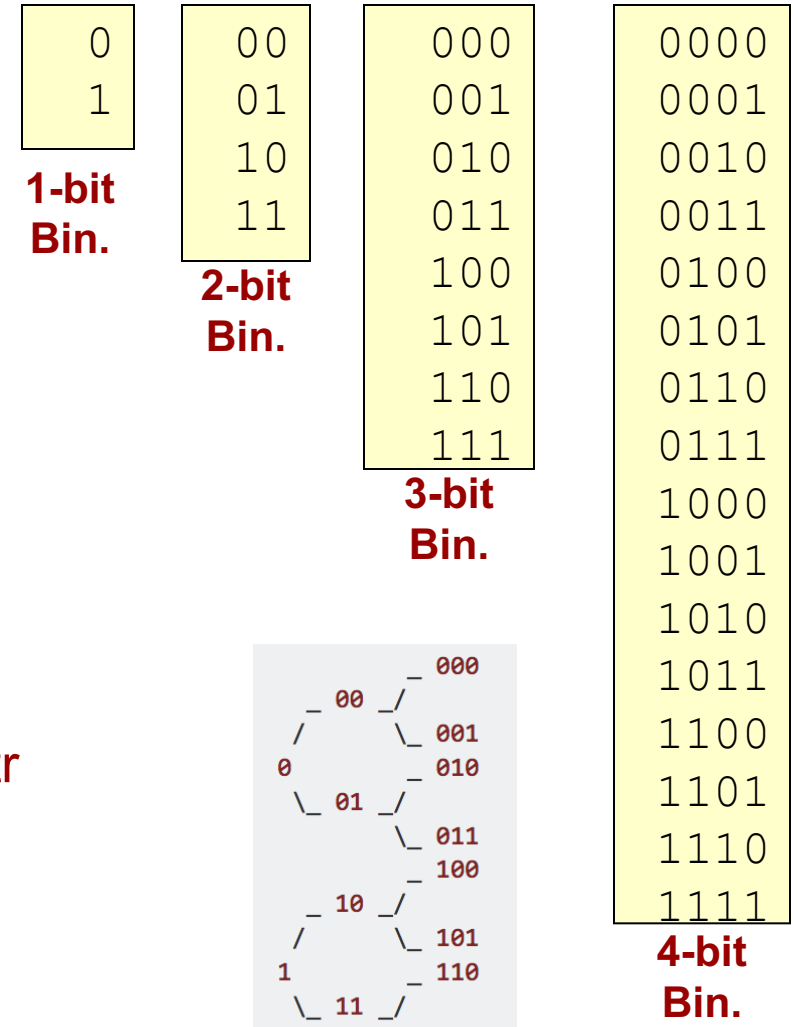
Recursion's Power

- The power of recursion often comes when each function instance makes ***multiple*** recursive calls
- As you will see this often leads to exponential number of "combinations" being generated/explored in an easy fashion

Binary Combinations

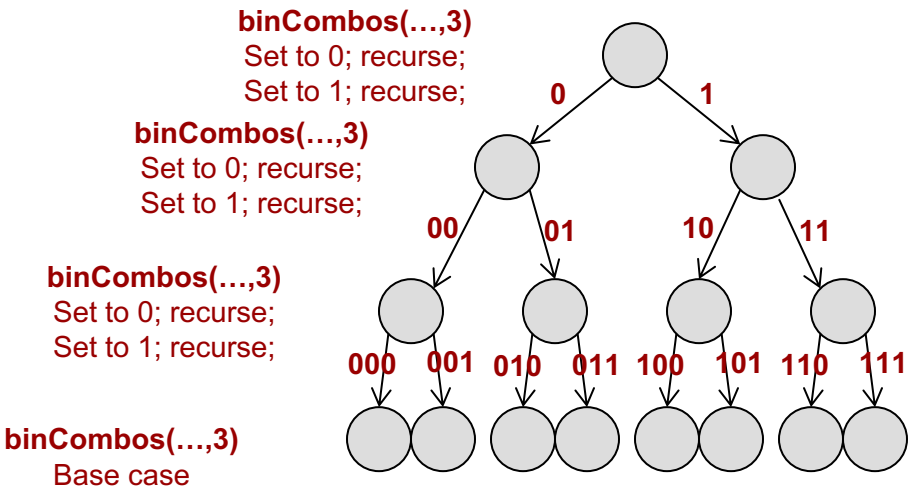
- If you are given the value, n, and a string with n characters could you generate all the combinations of n-bit binary?
- Do so recursively!

Exercise: bin_combo_str



Recursion and DFS

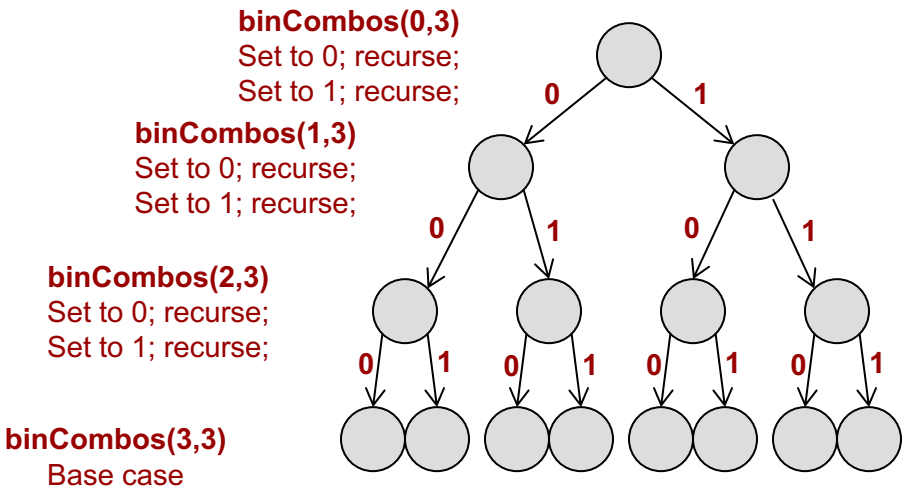
- Recursion forms a kind of Depth-First Search




```
// user interface
void binCombos(int len)
{
  binCombos("", len);
}
// helper-function
void binCombos(string prefix,
               int len)
{
  if(prefix.length() == len )
    cout << prefix << endl;
  else {
    // recurse
    binCombos(prefix+"0", len);
    // recurse
    binCombos(prefix+"1", len);
  }
}
```

Recursion and DFS (w/ C-Strings)

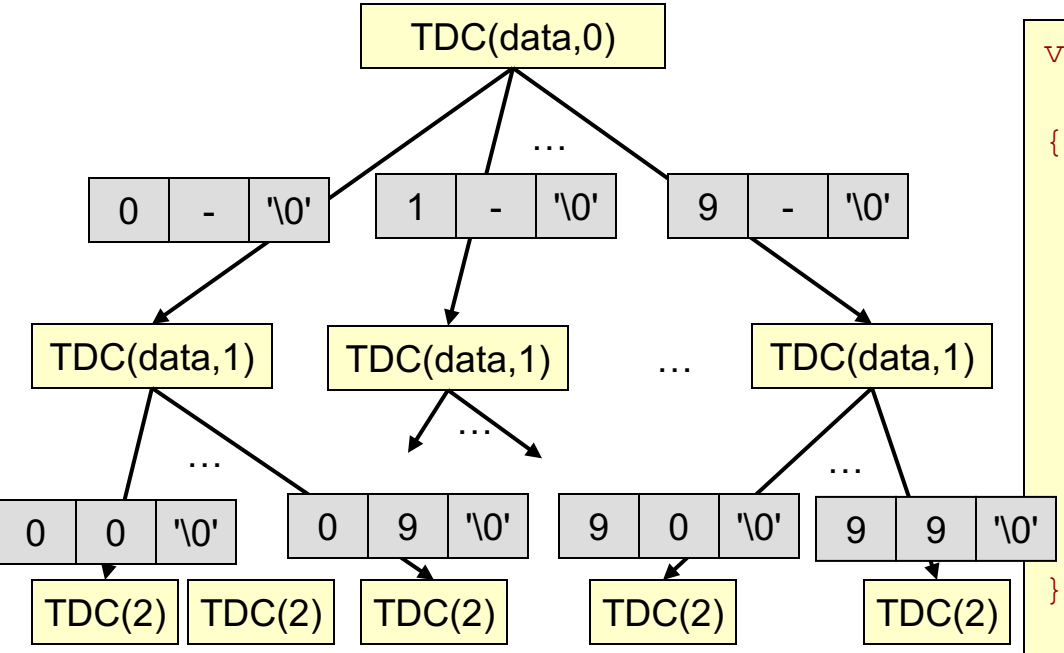
- Recursion forms a kind of Depth-First Search




```
void binCombos(char* data,
               int curr,
               int len)
{
  if(curr == len )
    data[curr] = '\0';
  else {
    // set to 0
    data[curr] = '0';
    // recurse
    binCombos(data, curr+1, len);
    // set to 1
    data[curr] = '1';
    // recurse
    binCombos(data, curr+1, len);
  }
}
```

Generating All Combinations

- Recursion offers a simple way to generate all combinations of N items from a set of options, S
  - Example: Generate all 2-digit decimal numbers (N=2, S={0,1,…,9})




```
void TwoDigCombos(char data[3],
                  int curr)
{
  if(curr == 2 )
    cout << data;
  else {
    for(int i=0; i < 10; i++){
      // set to i
      data[curr] = '0'+i;
      // recurse
      TwoDigCombos(data, curr+1);
  }
}
```

Recursion and Combinations

- Recursion provides an elegant way of generating all **n**-length combinations of a set of values, S.
  - Ex. Generate all length-**n** combinations of the letters in the set **S**={'U','S','C'} (i.e. for n=2: UU, US, UC, SU, SS, SC, CU, CS, CC)
- General approach:
  - Need some kind of **array/vector/string** to store partial answer as it is being built
  - Each recursive call is only responsible for one of the **n** "places" (say location, **i**)
  - The function will iteratively (loop) try each option in **S** by setting location i to the current option, then recurse to handle all remaining locations (i+1 to n)
    - Remember you are responsible for only one location
  - Upon return, try another option value and recurse again
  - Base case can stop when all n locations are set (i.e. recurse off the end)
  - Recursive case returns after trying all options

Exercises

- bin_combos_str
- Zero_sum
- Prime_products_print
- Prime_products
- basen_combos
- all_letter_combos

Another Exercise

- Generate all string combinations of length n from a given list (vector) of characters

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

void all_combos(vector<char>& letters, int n) {

}

int main() {
   vector<char> letters;
   letters.push_back('U');
   letters.push_back('S');
   letters.push_back('C');

   all_combos(letters, 2);

   all_combos(letters, 4);

   return 0;
}
```

RECURSIVE DEFINITIONS

Recursive Definitions

- N = Non-Negative Integers and is defined as:
  - The number 0 [Base]
  - n + 1 where n is some non-negative integer [Recursive]
- String
  - Empty string, ε
  - String concatenated with a character (e.g. 'a'-'z')
- Palindrome (string that reads the same forward as backwards)
  - Example: dad, peep, level
  - Defined as:
    - Empty string [Base]
    - Single character [Base]
    - xPx where x is a character and P is a Palindrome [Recursive]
- Recursive definitions are often used in defining grammars for languages and parsers (i.e. your compiler)

C++ Grammar

- Languages have rules governing their syntax and meaning
- These rules are referred to as its grammar
- Programming languages also have grammars that code must meet to be compiled
  - Compilers use this grammar to check for syntax and other compile-time errors
  - Grammars often expressed as “productions/rules”
- ANSI C Grammar Reference:
  - http://www.lysator.liu.se/c/ANSI-C-grammar-y.html#declaration

Simple Paragraph Grammar

Substitution	Rule
subject	"I" | "You" | "We"
verb	"run" | "walk" | "exercise" | "eat" | "play" | "sleep"
sentence	subject verb '.'
sentence_list	sentence | sentence_list sentence
paragraph	[TAB = \t] sentence_list [Newline = \n]

Example:

I run. You walk. We exercise.

subject verb. subject verb. subject verb.

sentence sentence sentence
sentence_list sentence sentence
sentence_list sentence
sentence_list
paragraph

Example:

I eat You sleep

Subject verb subject verb

Error

C++ Grammar

Rule	Expansion
expr	constant | variable_id | function_call | assign_statement | '(' expr ')' | expr binary_op expr | unary_op expr
assign_statement	variable_id '=' expr
expr_statement	';' | expr ';'

Example:

```
5 * (9 + max);
```

*expr * (expr + expr);*
*expr * (expr);*
*expr * expr;*
expr;
expr_statement

Example:

```
x + 9 = 5;
```

expr + expr = expr;
expr = expr;

NO SUBSTITUTION
Compile Error!

C++ Grammar

Rule	Substitution
statement	expr_statement | compound_statement | if (expr) statement | while (expr) statement …
compound_statement	'{' statement_list '}'
statement_list	statement | statement_list statement

Example:

```
while(x > 0) { doit(); x = x-2; }
while(expr) { expr; assign_statement; }
while(expr) { expr; expr; }
while(expr) { expr_statement expr_statement }
while(expr) { statement statement }
while(expr) { statement_list statement }
while(expr) { statement_list }
while(expr) compound_statement
while(expr) statement
statement
```

Example:

```
while(x > 0)
  x--;
  x = x + 5;
```

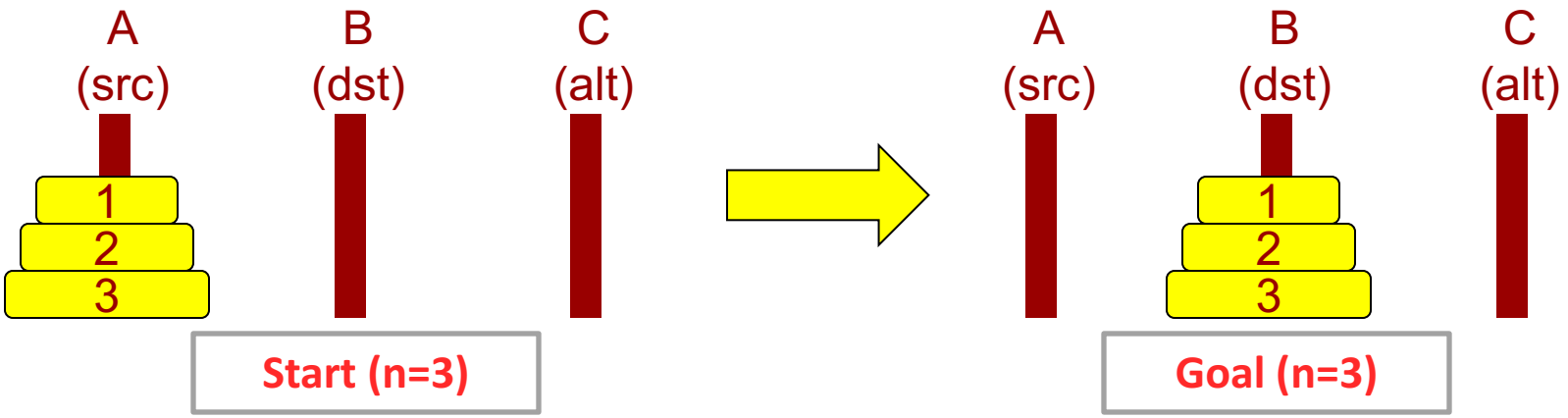
```
while(expr)
  statement
  statement
```

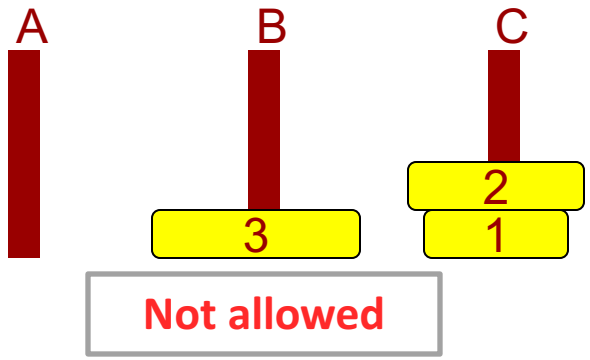
```
statement
statement
```

MORE EXAMPLES

Towers of Hanoi Problem

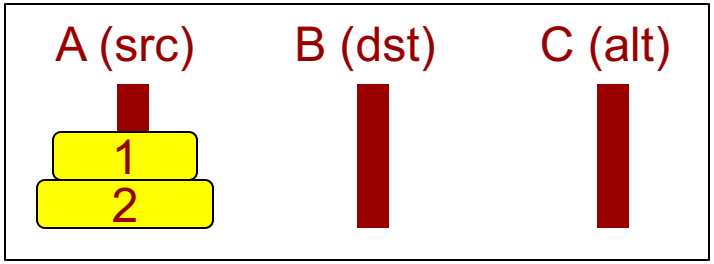
- Problem Statements: Move n discs from source pole to destination pole (with help of a 3rd alternate pole)
  - Cannot place a larger disc on top of a smaller disc
  - Can only move one disc at a time



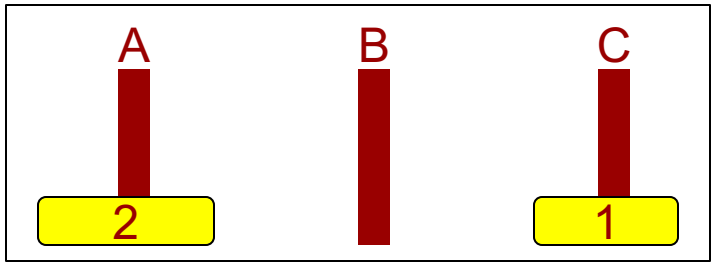


Observation 1

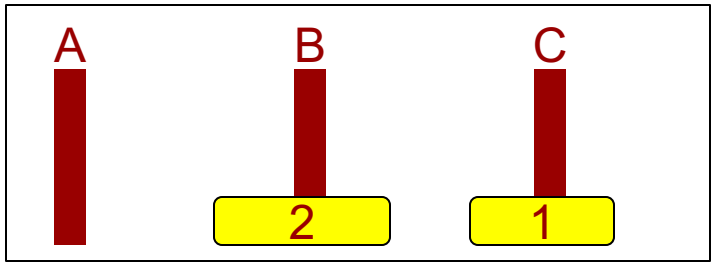
- Observation 1: Disc 1 (smallest) can always be moved
- Solve the n=2 case:



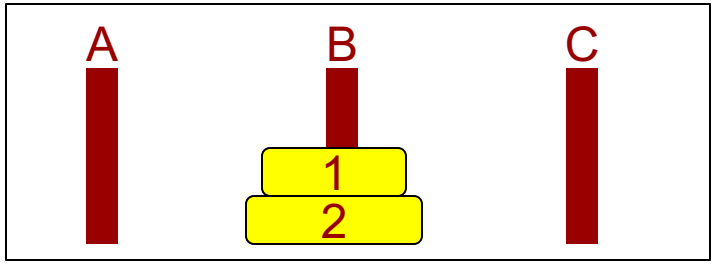
Start



Move 1 from src to alt



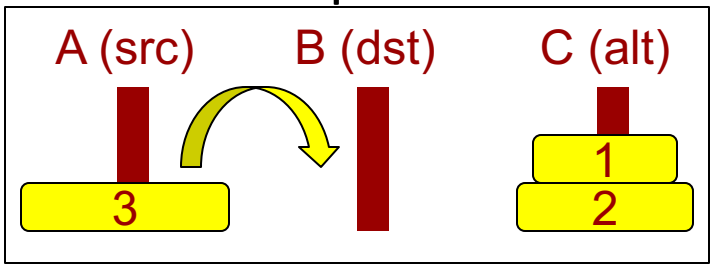
Move 2 from src to dst

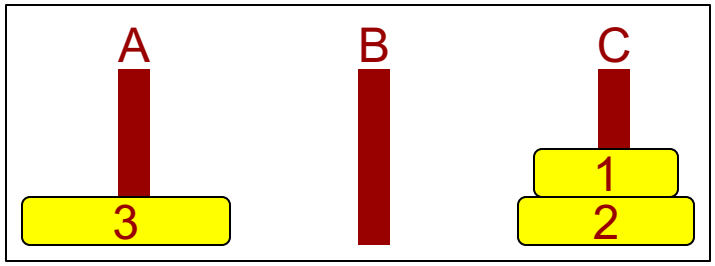


Move 1 from alt to dst

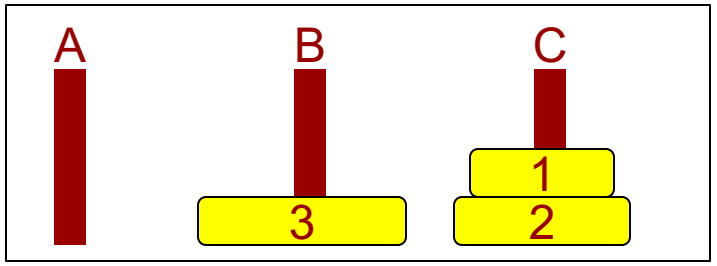
Observation 2

- Observation 2: If there is only one disc on the src pole and the dest pole can receive it the problem is trivial

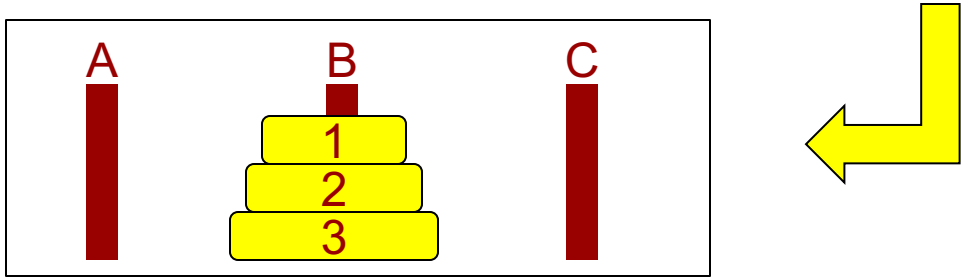




Move n-1 discs from src to alt



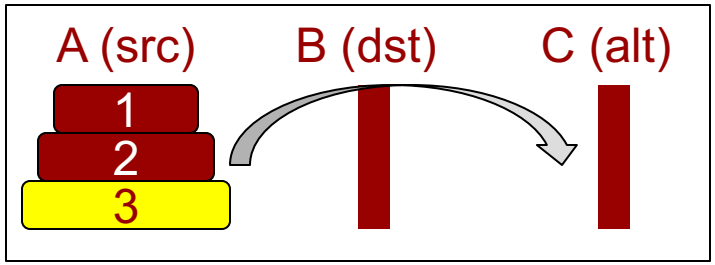
Move disc n from src to dst



Move n-1 discs from alt to dst

Recursive solution

- But to move n-1 discs from src to alt is really a smaller version of the same problem with
  - n => n-1
  - src=>src
  - alt =>dst
  - dst=>alt



- Towers(n,src,dst,alt)
  - Base Case: n==1 // Observation 1: Disc 1 always movable
    - Move disc 1 from src to dst
  - Recursive Case: // Observation 2: Move of n-1 discs to alt & back
    - Towers(n-1,src,alt,dst)
    - Move disc n from src to dst
    - Towers(n-1,alt,dst,src)

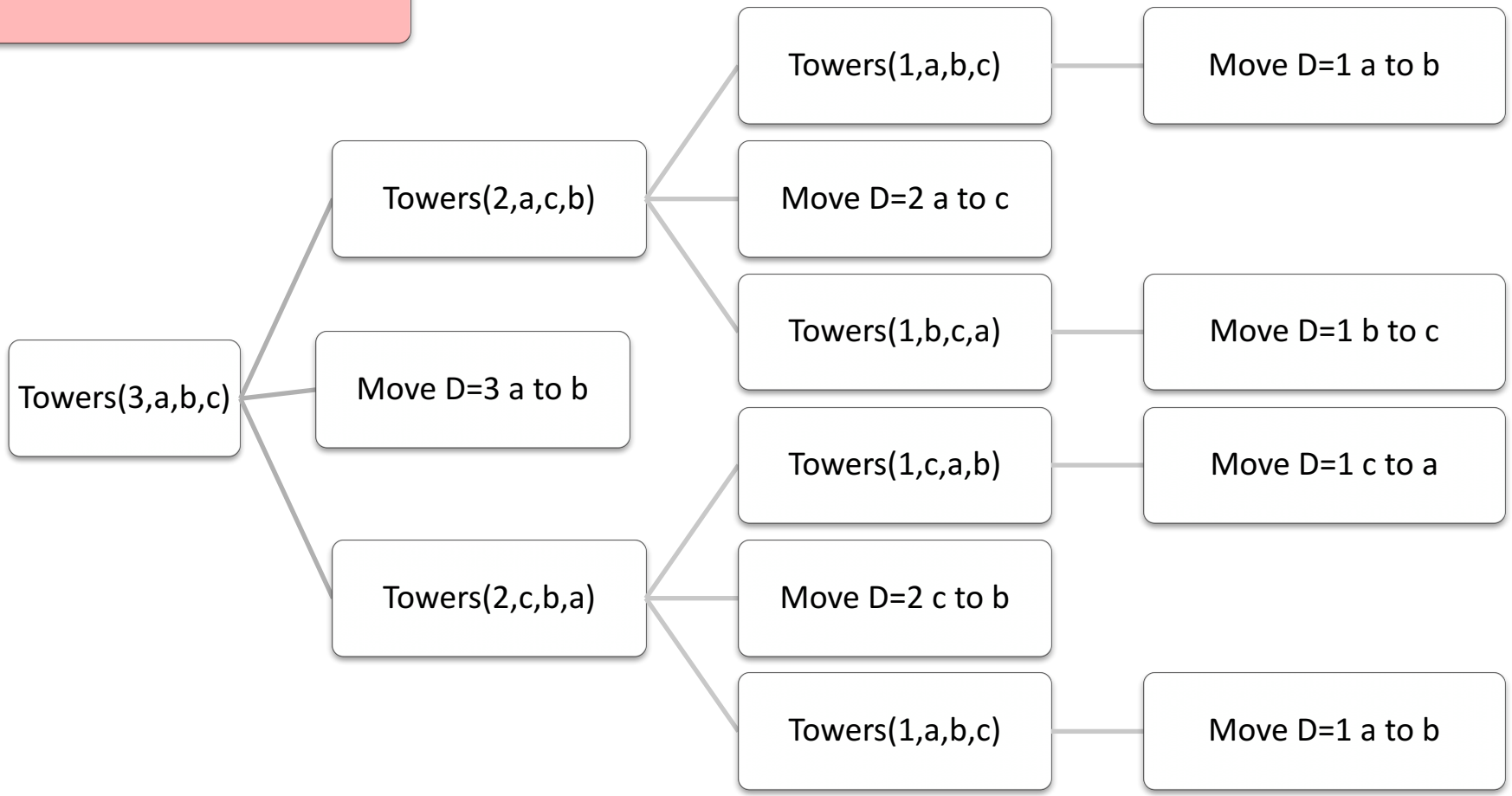
Exercise

- Implement the Towers of Hanoi code
  - $ wget http://ee.usc.edu/~redekopp/cs104/hanoi.cpp
  - Just print out "move disc=x from y to z" rather than trying to "move" data values
    - Move disc 1 from a to b
    - Move disc 2 from a to c
    - Move disc 1 from b to c
    - Move disc 3 from a to b
    - Move disc 1 from c to a
    - Move disc 2 from c to b
    - Move disc 1 from a to b

Recursive Box Diagram

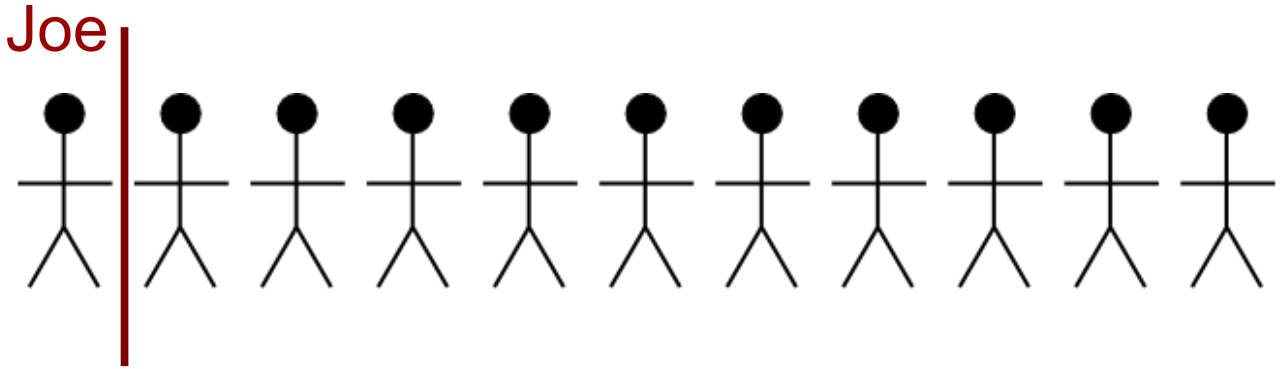
Towers Function Prototype

Towers(disc,src,dst,alt)



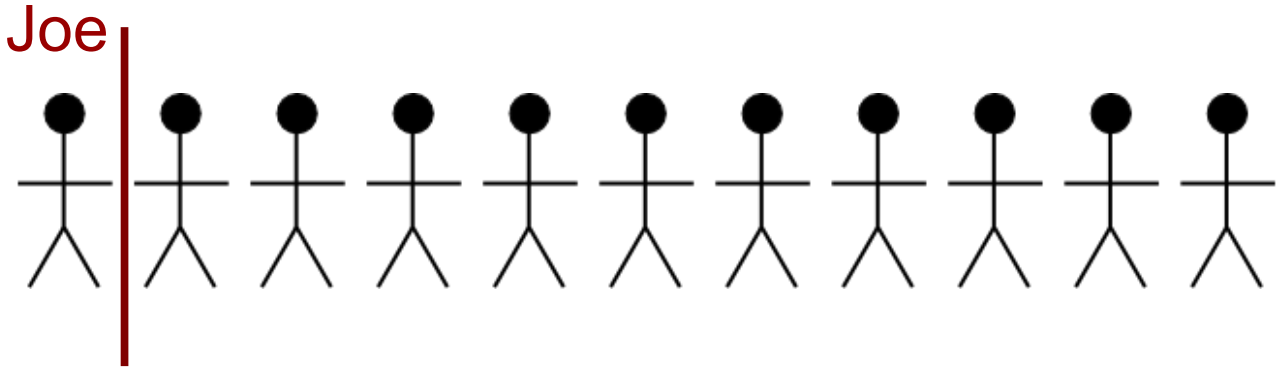
Combinatorics Examples

- Given n things, how can you choose k of them?
  - Written as C(n,k)
- How do we solve the problem?
  - Pick one person and single them out
    - Groups that contain Joe => _____
    - Groups that don't contain Joe => _____
  - Total number of solutions: _____
  - What are base cases?



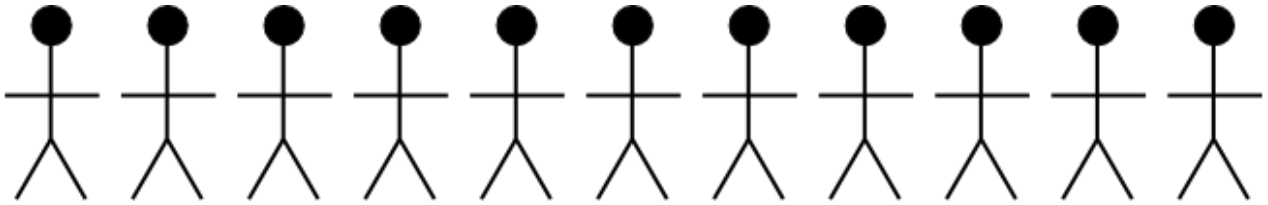
Combinatorics Examples

- Given n things, how can you choose k of them?
  - Written as C(n,k)
- How do we solve the problem?
  - Pick one person and single them out
    - Groups that contain Joe => C(n-1, k-1)
    - Groups that don't contain Joe => C(n-1, k)
  - Total number of solutions: C(n-1,k-1) + C(n-1,k)
  - What are base cases?



Combinatorics Examples

- You're going to Disneyland and you're trying to pick 4 people from your dorm to go with you
- Given n things, how can you choose k of them?
  - Written as C(n,k)
  - Analytical solution: $C(n,k) = n! / [k! * (n-k)!]$
- How do we solve the problem?



Recursive Solution

- Sometimes recursion can yield an incredibly simple solution to a very complex problem
- Need some base cases
  - C(n,0) = 1
  - C(n,n) = 1

```
int C(int n, int k)
{
if(k == 0 || k == n)
  return 1;
else
  return C(n-1,k-1) + C(n-1,k);
}
```