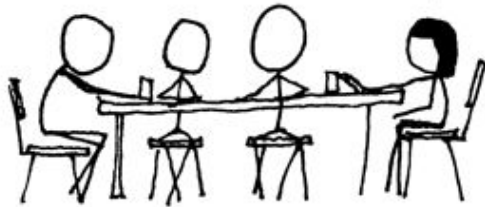


YOUR PARTY ENTERS THE TAVERN.

I GATHER EVERYONE AROUND
A TABLE. I HAVE THE ELVES
START WHITTLING DICE AND
GET OUT SOME PARCHMENT
FOR CHARACTER SHEETS.

HEY, NO RECURSING.



LAB 10 - RECURSION

README

REMINDERS

- Lab 10 due Sunday, December 9th
- Project 5 due TODAY
- Teaching evaluations on Canvas – I value your feedback!
- Final review Sunday 4-6pm – see Piazza announcement

AGENDA

- Recursion/ error handling
- Worksheet
- Lab time
 - Work on lab assignment
 - Work on project
 - Do practice final review

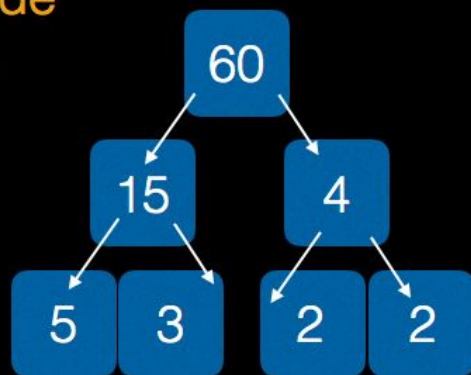
https://github.com/MelGeorge/recursion_demo

RECURSION

- Recall: recursion is whenever a function calls itself.
- What do we need for recursion to work?
 - We'll need a base case - the recursion has to stop somewhere so our function will return an answer. Don't forget this!
 - The problem we're trying to solve must be able to be split into many self-similar pieces.
- Recursion is similar to the idea of mathematical induction. Proofs by induction look like this:
 - Show something works for $n = 1$. Now, assume that it works for some value k . Show that if it works for k , it must work for $k + 1$. Thus, it must work for all $n \geq 1$.
 - When writing recursive functions, we'll use the same idea!

Factorization Pseudocode

```
factorization(int n, set of prime_factors) {  
  // base case  
  if ( n is prime ) {  
    add n to set of prime_factors  
    return  
  }  
  // typical case  
  else {  
    divide n into two smaller pieces  
    factorization(piece_1, prime_factors)  
    factorization(piece_2, prime_factors)  
    return  
  }  
}
```



Let's write a function to do the summation of all natural numbers up to n :

$$\sum_{n=1}^1 n = 1 \quad 1 = 1$$

$$\sum_{n=1}^2 n = 3 \quad 1 + 2 = 3$$

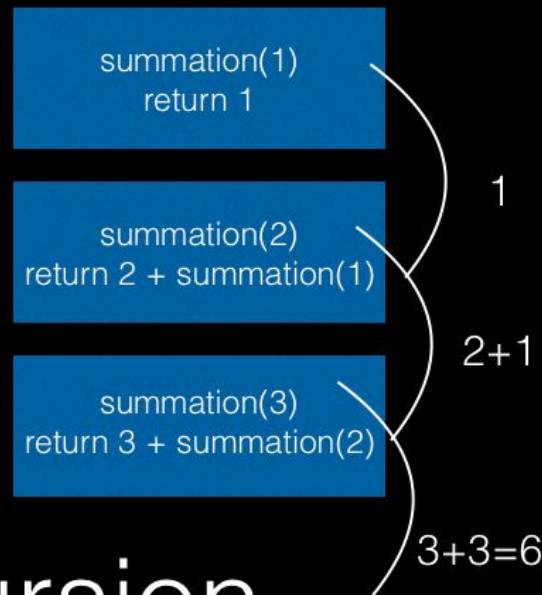
$$\sum_{n=1}^3 n = 6 \quad 1 + 2 + 3 = 6$$

We notice that:

$$\sum_{n=1}^3 n = 6 \quad = \quad \sum_{n=1}^2 n = 3 \quad + \quad 3 \quad = \quad 6$$
$$1 + 2 + 3 \quad \quad \quad 1 + 2$$

```
// Function for determining the summation of all
// natural numbers up to (and including) n
int summation(int n) {
    if(n == 1) {
        return 1;
    }
    else {
        return n + summation(n - 1);
    }
}
```

Memory Complexity: $O(n)$
Time Complexity: $O(n)$



Linear Recursion

A function makes one call to itself every time it runs

But say we don't want to use $O(n)$ space to do a simple summation? Let's try to write summation tail recursively instead.

Tail recursion is when the last line of work performed is the recursive call. That looks something like this:

```
// A tail recursive function to calculate factorial
unsigned factTR(unsigned int n, unsigned int a)
{
    if (n == 0) return a;
    return factTR(n-1, n*a);
}

// A wrapper over factTR
unsigned int fact(unsigned int n)
{
    return factTR(n, 1);
}
```

Idea for a tail recursive version of summation:

Let's keep track of a sum "so far", and keep track of the numbers we still need to add.

Idea for a tail recursive version of summation:
Let's keep track of a sum "so far", and keep track
of the numbers we still need to add. That might
look something like this:

$$\sum_{n=1}^3 n = 6$$

To do $1 + 2 + 3 = 6 \dots$

sum_so_far = 0 we still need to add: 3, 2, 1

now add 3 to sum_so_far

sum_so_far = 3 we still need to add: 2, 1

now add 2 to sum_so_far

sum_so_far = 5 we still need to add: 1

now add 1 to sum_so_far

sum_so_far = 6 we still need to add:

return sum_so_far = 6


```
// Helper function
int summation_helper(int sum_so_far, int nums_to_add) {
    if(nums_to_add == 1) {
        return sum_so_far + 1;
    }
    else {
        return summation_helper(sum_so_far + nums_to_add,
                                nums_to_add - 1);
    }
}
```

Idea: keep adding the next smallest natural number, keeping track of what we've added so far.

```
// A tail recursive implementation of the above
// summation function-- still determines the
// summation of all natural numbers up to
// (and including) n
int summation_tail(int n) {
    return summation_helper(0, n);
}
```

nums_to_add = 1
sum_so_far = 5

~~summation_helper(5, 1)
return 5 + 1~~

nums_to_add = 2
sum_so_far = 3

~~summation_helper(3, 2)
return summation_helper(5, 1)~~

nums_to_add = 3
sum_so_far = 0

~~summation_helper(0, 3)
return summation_helper(3, 2)~~

n = 3

~~summation(3)
return summation_helper(0, 3)~~

6

Tail Call Optimization!

Memory Complexity: $O(1)$ (!!)

Time Complexity: $O(n)$

Tail Recursion

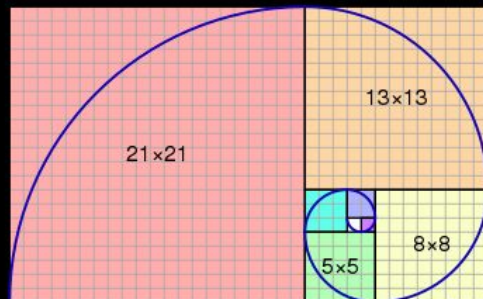
The recursive call is the last item of work to be done

Now let's look at fibonacci numbers:

$$F_0 = 0, F_1 = 1.$$

$$F_n = F_{n-1} + F_{n-2}.$$

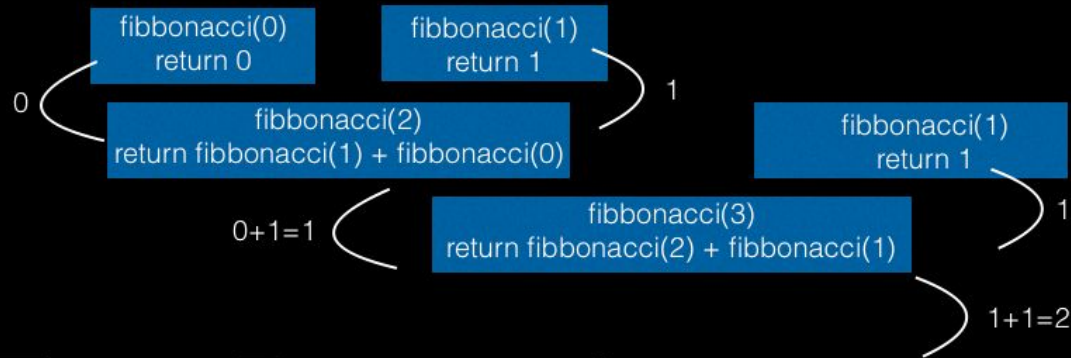
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...



Fibonacci numbers might be fun to calculate recursively, since each fibonacci number is the sum of the previous two. we'll have to use a special type of recursion called tree recursion— where a function calls itself more than once.

```
// Function for determining the nth fibonacci
// number: remember that fibonacci numbers go
// like this: 0 1 1 2 3 5 8 13 21 34 ...
// Where each number is the sum of the previous 2
// Let's define fibonacci(0) = 0, fibonacci(1) = 1
int fibonacci(int n) {
    if(n == 0) return 0;
    if(n == 1) return 1;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Memory Complexity: $O(2^n)$
 Time Complexity: $O(2^n)$



Tree Recursion

A function makes multiple recursive calls to itself at a time

ERROR HANDLING -- EXCEPTIONS

When we are writing code, we often run into situations where we get an unexpected input, or something goes wrong. When this happens, we would like a safe way to let the function caller/ the user know that something went wrong. It's also useful to give them a description of **what** went wrong.

Solution-- exceptions.

```
class error {  
public:  
    error(std::string msg) : message(msg) {};  
    std::string message;  
};  
  
int main() {  
    try {  
        throw error("main threw an error");  
    }  
    catch(error & err) {  
        std::cout << err.message << std::endl;  
    }  
    std::cout << "finishing up main" << std::endl;  
}
```

```
main threw an error  
finishing up main
```

TRY BLOCKS AND CATCH BLOCKS

- We surround code that might throw an error in a “try” block
- We put matching “catch” blocks at the end of our “try” blocks to catch the errors thrown
- We look through the catch blocks one by one (top to bottom) until one matches the type of error thrown-- then we execute the matching catch block code
- Note: polymorphism comes into play here!
- If no matching catch block exists, the error propagates “outward” through the function calls until one is found
- If no matching catch block is ever found, the program crashes

TRY BLOCKS AND CATCH BLOCKS -- EXAMPLE

```
23 class input_error {};  
24  
25 int main(int argc, char *argv[]) {  
26     try {  
27         if(argc != 2) {  
28             throw input_error();  
29         }  
30         if(strcmp(argv[1], "hello")  
31             && strcmp(argv[1], "world")) {  
32             throw input_error();  
33         }  
34     }  
35     catch(input_error & e) {  
36         std::cout << "Usage:" << std::endl;  
37         std::cout << "./main.exe hello" << std::endl;  
38         std::cout << "./main.exe world" << std::endl;  
39     }  
40     catch(...) {  
41         std::cout << "Unknown error" << std::endl;  
42     }  
43 }
```

Say we have a program that can only be run in these two ways:

./main.exe hello

./main.exe world

We can enforce this and handle input errors as is shown here.

Note that

catch(input_error &e)

catches input_errors only,
while

catch(...)

catches any other errors.

What would happen if **strcmp** threw an error?

TRY BLOCKS AND CATCH BLOCKS -- PRACTICE QUESTION

```
5 class one {};  
6 class two : public one {};  
7 class three : public one {};
```

```
9 int divide(int a, int b) {  
10     try {  
11         if(b == 0) {  
12             throw two();  
13         }  
14     }  
15     catch(three &e) {  
16         std::cout << "three caught" << std::endl;  
17         b = 1;  
18     }  
19     std::cout << "finished divide" << std::endl;  
20     return a / b;  
21 }
```

```
23 int main() {  
24     try {  
25         int answer;  
26         answer = divide(1, 2);  
27         answer = divide(1, 0);  
28         answer = divide(4, 2);  
29     }  
30     catch(two &e) {  
31         std::cout << "two caught" << std::endl;  
32     }  
33     catch(one &e) {  
34         std::cout << "one caught" << std::endl;  
35     }  
36     catch(...) {  
37         std::cout << "any error caught" << std::endl;  
38     }  
39     std::cout << "finished main" << std::endl;  
40 }
```

If I have these 3 error classes, the function on the left, and the main function on the right, what is printed?

TRY BLOCKS AND CATCH BLOCKS -- ANSWER

```
5 class one {};  
6 class two : public one {};  
7 class three : public one {};
```

```
9 int divide(int a, int b) {  
10     try {  
11         if(b == 0) {  
12             throw two();  
13         }  
14     }  
15     catch(three &e) {  
16         std::cout << "three caught" << std::endl;  
17         b = 1;  
18     }  
19     std::cout << "finished divide" << std::endl;  
20     return a / b;  
21 }
```

```
23 int main() {  
24     try {  
25         int answer;  
26         answer = divide(1, 2);  
27         answer = divide(1, 0);  
28         answer = divide(4, 2);  
29     }  
30     catch(two &e) {  
31         std::cout << "two caught" << std::endl;  
32     }  
33     catch(one &e) {  
34         std::cout << "one caught" << std::endl;  
35     }  
36     catch(...) {  
37         std::cout << "any error caught" << std::endl;  
38     }  
39     std::cout << "finished main" << std::endl;  
40 }
```


Answer:

```
finished divide  
two caught  
finished main
```


POLYMORPHISM & EXCEPTIONS

```
5 class one {};  
6 class two : public one {};  
7 class three : public one {};
```

What happens if we put the “catch(one &e)” block before the “catch(two &e)” block?

```
23 int main() {  
24     try {  
25         int answer;  
26         answer = divide(1, 2);  
27         answer = divide(1, 0);  
28         answer = divide(4, 2);  
29     }  
30     catch(one &e) {  
31         std::cout << "one caught" << std::endl;  
32     }  
33     catch(two &e) {  Exception of type 'two &' will be caught by earlier handler  
34         std::cout << "two caught" << std::endl;  
35     }  
36     catch(...) {  
37         std::cout << "any error caught" << std::endl;  
38     }  
39     std::cout << "finished main" << std::endl;  
40 }
```

Exceptions display polymorphic behavior--in our previous example, since class “two” inherits from class “one”, the catch block that catches objects of type “one” will also catch objects of type “two”, so this second catch block will never run!

ADVICE FOR REVIEWING FOR THE FINAL

- Go through all of the lecture slides/ redo examples
- If you missed lectures, watch them. If you didn't miss them, DON'T watch them again.
- Make your cheat sheet as you look through the lecture slides
- Redo lab activities & glance at projects 3 - 5
- Do the practice exams we gave you
- Do review session worksheet & maybe attend or watch the recorded review session
- Study with friends and make up questions for each other

NOTICE THAT OUR SAMPLE EXAMS HAVE:

1. Short Answer
2. Dynamic Memory
3. Lists and Templates
4. Iterators and Functors
5. Recursion

GOOD LUCK ON THE EXAM!

Thanks for a great semester!

Study hard!

