CSI 201
Function Prototypes and Function Calls

1. We're going to be learning how to write our own functions. This is an immensely powerful tool that helps us in both modularity and code re-usability. Because modularity often leads to pieces that are easier to test and debug, this will be a big boon once we completely comprehend them. Let's start with a a few prototypes.

2. `double sqrt(double);`

3. The above is the prototype for the sqrt function. It tells us what kind of thing the sqrt function expects and the kind of thing the sqrt function will return. The first double in that line above is the return value. It is what sqrt will evaluate to. It tells us that assigning the result of a sqrt to a `string` is useless. It must be assigned to a double. The second double in that expression tells us the kind of thing that sqrt needs to passed. What we need to give to the sqrt to call it. In this case, we know that we must give the sqrt a double. In our code, then, we expect to see sqrt calls that involve doubles going into the function and coming out of the function. We can thus use the sqrt function in any place where we could use a double. For example if there was the following code: `double d = 3.3 * 4.0;` We can multiply 3.3 times any double. That means, if we needed to, I can multiply 3.3 times a sqrt.

4. It actually helps if we simply get some practice using it. Let's see if we can find the result of $\sqrt{a * a + b * b}$ where a and b are input by the user. If we can do that, we can find the length of the hypotenuse of a right triangle with shorter sides of length a and b! Try it out:

5. There are plenty of other functions. The pow function takes a base and an exponent and gives the result of doing that mathematics. See if we can find the result of raising 14 to the 3rd power. Output the result. What is the answer?

6. The pow function works this way because of its prototype. It looks like the following bullet:

7. `double pow(double, double);`

8. In this prototype we know that pow takes in two doubles and gives us a double in return. The parameter list is comma separated. The base is also always the first argument to the pow function. Both pow and sqrt are tucked away in the cmath library. Let's look at some other interesting use cases for prototypes.

9. When we seed the random number generator, we pass an integer to srand. That has the following prototype:

10. `void srand(unsigned int);`

11. This means we can give any unsigned integer to this seed function and it will see the random number generator. This is great! The void here means that when we call srand, it doesn't compute some value and bring it back to us. void is really the absence of any return type. The function simply doesn't bring anything back. It does set some internal state or change "the outside world" in some way. But it doesn't, programmatically, give us a value. This means we can't cout an srand or assign an srand to a variable or anything like that. But that's okay with us, because we don't need it to have a return value for it to be useful. Let's look at rand too:

12. `int rand();`

13. Rand is an interesting function because we don't have to give it anything to get an answer. Effectively what we do when we call rand is simply ask it for the next random integer. Every time we call it, we get the next random integer over and over again.

14. It is quite frequent as a computer scientist to run into code that we have not seen before. Thus it is critical to be able to read documentation and glean some information out of it. The function prototype is one such piece of code that can give us hints about how a function is meant to operate. Does it return something? What name must I use to call it? What kinds of things does it expect as parameters? These are critical to understanding how a function operates. Imagine we received a tangent function that took two parameters. What would that mean? Understanding those two inputs will help us uncover the purpose of the tangent function (this does exist in cmath). But even without precisely understanding what the function does, we can make function calls given the prototype information. For example, if the prototype is: `double xyzzy(double, double, double);` we can at least write a proper function call by knowing that it takes in three doubles and returns a double at the end of its computation.

15. Let's look at some function calls again in a complete code snippet:

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

int main() {
        double ui1, ui2, ui3; //these are for user input
        cin >> ui1 >> ui2 >> ui3; //read them in
        //directly output a result with sqrt:
        cout << sqrt(ui1) << endl;
        //since sqrt can be used like a double
        //I can use it in other arithmetic
        cout << 10*sqrt(ui1) << endl;
        //I can use the results of arithmetic to pass to
        //the sqrt function as well:
        cout << sqrt(10*ui1) << endl;
        //let's move on to pow for a few more examples
        //pow computes base raised to exponent power
        // so for 2 ^ 3 we can write:
        cout << pow(2,3) << endl;
        //so far we've just been outputting results, we
        //can store things in variables just as easily
        double my_pow = pow(ui1, ui2);
        //my pow now holds whatever ui1 raised to ui2
        //might happen to be. Let's make things
        //interesting by using this result in another
        //pow function
        double my_new_pow = pow(ui3/20.0, my_pow);
        //this is perfectly fine and even acceptable.
}
```

16. So, sqrt and pow are used a ton in that example. But we've had code that used plenty of other functions too. size, at, push_back, and resize are all functions with vectors. Our main is a function!