CSI 201
Functions - Putting things together - more practice

1. We've had a worksheet on creating prototypes and attaching those to function calls. Most recently we've had a worksheet on function definitions.

2. So, for today, let's first take a look at how to put those things together into a fully working piece of code.

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

//function prototypes go here
//variable names are optional in prototypes
double addition(double, double);

int main() {
   double ui1, ui2, ui3; //these are for user input
   cin >> ui1 >> ui2 >> ui3;
   cout << addition(ui1, ui2) << endl;
   cout << addition(ui1, ui3+3) << endl;
   //what values do ui1, ui2 and ui3 have now?
}

//function definitions can go here
double addition(double a, double b) {
   double my_result = a + b;
   return my_result;
}
```

3. This code shows the three pieces of functions. It is important (for your understanding) to truly understand the three pieces involved here. I will discuss three locations in these descriptions. They will be described as: `before main`, `after main` or `inside main`, where main is simply the main function where we've inserted all code. In any cases, the three pieces of functions are:

   (a) `function prototype`: This happens before main. This is a promise to the compiler that some function will eventually be implemented. It describes to the compiler the three parts of a function. It describes the name of the function, the parameter list (what the function takes in), and the return type of the function (what the function puts out). With this in hand, the compiler will be able to tell if any `function calls` match this particular signature. Those calls must have the same name as a function that has been prototyped and pass in the proper types to the function, as the function prototype describes. So, in the example above, `double addition(double,double);` is the function prototype. It gives a promise to the compiler that the addition function will eventually be defined that will take in two doubles and return a double.

(b) `function definition:` In this code, this happens after main. In many introductory books (and like we did last time) this can happen before main without an explicit prototype. This describes both the function signature and what the function will actually do (and return). In our example above, the `addition` function is defined as taking in two doubles. The definition then creates a new double, stores the addition of the passed in doubles and then returns the new double as the result. This means that the function is given any two numbers and then is evaluated to their addition. This is analogous to using `sqrt(a)` in your code in order to find out $\sqrt{a}$. But now I can use: `addition(a, b)` to find out $a+b$. This is certainly over simplistic (or perhaps over complicated), but it is good to start with simple and well understood examples before moving on to harder things.

(c) `function call:` Writing the function is like creating a machine with a button or a feed slot. Imagine a paper shredder. Generally it does nothing. But when you put in a paper (the parameter) it shreds it into tiny pieces (the returned type). The function call is the act of giving the machine something. So, in the above code, when we ask for the machine by name and give it two values, it will use those to compute an answer and give that answer back to us. Sometimes, we might not need to give the machine anything at all. We might just need to look at the machine and tell it to go (for example `rand`). Other times the machine does something to the internal state of the program but doesn't give us anything back. For example, a machine that incinerates your papers requires you to give it the paper, but then, perhaps, doesn't give anything back (let's ignore ash for the moment!). A piece of code that formats output in a pretty way might be tucked into a function. We send that function the values we want it to print pretty and it does the print, but it doesn't bring us back anything.

4. We should motivate why the separation of prototypes and definitions works so well. When we include another library, we're usually including the prototypes of functions rather than the definitions. This allows us to define the definitions elsewhere and further organize and modularize our code. This idea is an advanced one, but it is one that will keep coming up in computer science. For now, let's just say that it is easier to debug and code in this way. We moved to functions for just this reason. It might also even be faster to compile and execute because of some of these choices. Seeing how execution time can benefit (because it usually doesn't) by moving to functions or libraries is a stretch that involves deep discussions of computer memory and memory management. We talk about these things in classes like computer architecture and operating systems. They are a lot of fun so they are whole heartedly recommended to those who are interested. In any case, our examples today will highlight and help us to see this separation, so let's get to a few.

5. You've been asked to find the area of a regular pentagon given user input. You have no idea how to find the area of a pentagon, but that's okay because someone has provided a function that takes in the length of one side of a regular pentagon and returns the area.

That function has the following prototype: `double regularPentagonArea(double);` Given this function, complete your task.

6. How is it that we can write this code without knowing how things work? Is this useful or no?

7. It turns out we must do this all the time in computer science. Digging down to the lowest level is not always useful or helpful (although sometimes it can be). Just like driving a car by directly manipulating the engine is generally not useful. Instead, we use exposed functionality to operate our vehicles. In this way, it is not even necessary to know specifically how the engine works. As long as the exposed functionality works and it is connected to the internal pieces correctly, knowing only how to use those pieces is fine. We see that when we use `sqrt` or in our last example with `regularPentagonArea`.

8. Let's practice this one or two more times. Imagine we have a function for the acceleration of a particle given a single parameter. The parameter can change from 0 to 100, but no more or less. The function prototype is `double acc(double);`. Write some code to find the value of the parameter that gives the smallest possible acceleration over all possible input parameters.

9. We've been given a function to find the speed of a particle given two double parameters that can change from 0 to 100. The function prototype is: `double speed(double, double);`. Write some code to find the values of the two parameters that give the maximum speed of all possible input parameter values.

10. Did you consider only integer values of 0 to 100? How might be change this to search deeper?

11. Imagine we're given a function to compute Δv. The prototype might be: `double deltaV(double, double, double);` Imagine the third paramter can have a value between 0 and whatever the second parameter might be. Write some code to read in user input for the first two parameters. Then find the maximum Δv and the value of the third parameter that creates this maximum.