Next time: LCR 3.

Functions

Consider functions with no parameters and void return.
These are like abbreviations for a sequence of actions or commands.

```
void <function name> () {
  <statements> // the body of the function
}
```

When you call the function, your program jumps to the beginning of the function, does the statements in it, and then "returns to the caller." It resumes execution right after where it was called.

A function of this kind is like setting up an office to do some well-defined task. For example, a business might have to file taxes once a year, and it might have a special office for doing this. When we want to do the taxes, we knock on the door ("calling" them), they do the work to file the taxes, and when they are done they ring a bell or do something else to indicate they are done. But they don't have any information to give us except the notification they are done. When we hear the bell, we resume doing what we were doing before we called them.

Consider a void function with parameters. These are inputs that the function needs to do its job. Maybe the business has an office that pays bills, but I need to tell it which bill to pay. So I put the bill to be paid in the office's In Box, they pay the bill, and ring a bell when they are done.

We can also have functions that might or might not have parameters, but also

return some result. So in this case we're thinking of an office that does something and returns a result. Suppose we have an office that looks up a word in a dictionary. So when I get to a word I don't understand, I put it in the In Box of the "Lookup Office." Someone in the office looks up the word n the dictionary, writes the definition on a slip of paper, puts the paper in their Out Box, and rings the bell. I get the answer out of their Out Box and resume what I was doing,

The type of the return value goes in front of the function name when you define the function:

```
<return type> <function name> ( <parameters> ) {
  <statements>
  // put something in Out Box and ring the bell
  return <something of the appropriate type>;
}
```

A common operation is to print a prompt and then read in something supplied by the user. Remember the Abstraction Principle: Factor out the common pattern and give it a name. So I'm going to call this function getDouble (<prompt>).

The function header (normally the first line) tells you about all the inputs and outputs to the function (i.e., In and Out Boxes). In this case:

```
  double getDouble (string prompt)
```

Any variable declared within a function is considered local to that function, in other words, it is visible only within the function. For example "input" is only visible within getDouble. (The same rule applies to parameter names: they are local to the function.) We say that these names have local scope.

If a variable (or some other name) is declared outside of all functions, it is visible throughout the whole program and is said to have global scope.

Notice that we can compose functions, that is, use functions as the arguments to other functions. [Just like we can compose operators. For example, we can compose + and * in an expression like (3 * 6) + 1.]

Each function is a self-contained task or job, a module. It has its input parameters, it has its output (which may be void), and it can have local variables to help it do its job.

Upper level functions call lower level functions, and they call even lower level ones, until you get to functions and operations that are either part of C++ (e.g., the +, -, *, /) or are defined in standard libraries (sqrt, pow, sin, log, …).

You can also start from the given functions and build up more complex ones to do more complicated tasks. Bottom-up development vs. top-down development.