Functions
As you probably know from your other programming courses, a key part of any modern programming language is the ability to create separate functions in your code.

A function is a named piece of code that performs a specific task. Sometimes functions are called “methods”, “procedures”, or “subroutines” (like in LC-3).

As in just about every other language that you've seen, a function takes some input and produces some output. The exact definitions of “input” and “output” are things we'll talk about later.

Why write functions? There are a lot of software engineering-y answers to that:
- it makes it easier to think about how to write your program (by solving little problems and then putting those together to solve bigger problems)
- it makes it easier to find a bug when something goes wrong
- it makes it easier to change one part of the program without having to change everything else

Let's say we want to write a function that determines which of two ints is bigger. This is admittedly a pretty trivial task and may not justify having its own function, but it's okay for our purposes here.

Here's how that might look in C:

```c
#include <stdio.h>
#include <stdlib.h>

int find_max(int a, int b) {
    int bigger;
    if (a > b) {
        bigger = a;
    } else {
        bigger = b;
    }
    return bigger;
}

int main() {
    printf("Enter the first number: ");
    scanf("%d", &num1);
    printf("Enter the second number: ");
    scanf("%d", &num2);
    int max = find_max(num1, num2);
    printf("The max is %d\n", max);
    return 0;
}
```

```c
/*
 * Function to determine the maximum of two numbers.
 */
```
There are three important parts of this program related to our find_max function:

- On line 4, we have the function declaration. This is sometimes called the function header or function prototype. It tells the compiler that we're going to have a function called “find_max” which takes two ints as its inputs, and returns an int as its output. Just like a variable, with a function, you have to declare it before you can use it.

- On line 14, we have the function invocation. This is where we call the function. The variables num1 and num2 are the inputs, or arguments to this function call.

- From lines 22-31 we have the function definition. This is where the code for the function goes. Note that on line 22, we give names to the inputs (a and b). These are called the parameters of the function. And on line 30, we have the return statement, which “sends back” the value of the variable bigger.

What if a function doesn't take any inputs? Or doesn't return any output? In those cases, the parameters or return type would be void. For instance, you could write a function like this:

```c
void print_hello(void) {
    printf("hello world!\n");
    return;
}
```

In this case, the return statement on line 3 doesn't take any argument; it just returns. However, all of this is a bit redundant; after all, doesn't “void” just mean “nothing”? So you could alternatively write the code like this:

```c
print_hello() {
    printf("hello world!\n");
}
```

Either one is acceptable. I kind of like the second one more, but maybe that's just me...

Scope

How can functions share variables? In the find_max example above, why doesn't main just access the variable bigger, instead of setting max to the return value of find_max? The issue here is related to “scope”.

The scope of a variable is where it “lives”, i.e. the part of the code in which it can legally be accessed.
Broadly speaking, a variable can either have:

- **global scope**, in which it can be accessed by any function in the file
- **local scope**, in which it can be accessed only in the function in which it is defined

Let's see an example. Note that there is no “main” function here; that's okay, it's just an example:

```c
#include <stdio.h>
#include <stdlib.h>

int x;

int one(int a, int b) {
    int y = 3 + x;
    return a + b + y;
}

int two (int a) {
    x = 7;
    y = 11;
    return a*2;
}
```

On line 4, we have declared `x` as a global variable. How do we know it's global? Because it's declared outside of any function. This means that it can be used by functions `one` (where it is read on line 7) and `two` (where it is modified on line 12).

Recall above that we mentioned function “inputs” and “outputs”. The input not only includes the function's parameters, but also any variable to which the function has access, i.e. global variables. In this case, the global variable `x` is also considered input to `one`, since its value is read. On the other hand, the output of a function not only includes its return value, but any global variables that are changed by it. Here, the function `two` modifies `x`, which is considered part of the output; this is referred to as a **side effect** of the function `two`.

Note that there is a bug in this code! And the bug is related to the scope of the variable `y` that is declared on line 7. The general rule of thumb is that the scope of a variable extends from its declaration to the closing curly brace that matches the open brace closest to the declaration (got all that?). In this case, the scope of `y` goes from line 7 (its declaration) to line 9 (the closing curly brace that matches the open brace on line 6), i.e. the definition of function `one`.

Where is the bug? On line 13. The compiler will complain that the variable `y` has not been declared, because it is “out of scope” there. Note that changing line 13 to

```c
    int y = 11;
```

would declare a different variable called `y` in function `two`, i.e. it would not be the same as the one used in function `one`. 

```c
```
Can a local variable have the same name as a global variable? Unfortunately, yes. I say “unfortunately” because this can cause a lot of confusion when reading your code.

For instance:

```c
#include <stdio.h>
#include <stdlib.h>

int x = 1;

int fun(int a, int b) {
    int x = a + b;
    printf("x is %d\n", x);
    return a + b;
}
```

In this case, we have a global variable called `x` declared on line 4, and a local variable called `x` declared on line 7. So what gets printed on line 8? The compiler assumes that, if you declared a local variable with the same name as a global variable, you must mean to use that one. So on line 8, the local variable `x` (which has the value `a + b`) will be printed.

Scope can be a tricky thing, and it's best to use distinct variable names whenever possible. Keep in mind the rule about the scope being related to closing curly braces. Let's look at an example. Here's a function that's attempting to return the absolute value of the difference between two ints:

```c
int abs_diff (int x, int y) {
    if (x > y) {
        int diff = x - y;
    } else {
        diff = y - x;
    }
    return diff;
}
```

Here, the scope of the variable `diff` is only from its declaration (line 4) to the closing curly brace (line 5), so clearly it's out of scope when you try to use it on lines 7 and 10; the compiler will complains about this.

So perhaps you try to do this:

```c
int abs_diff (int x, int y) {
    if (x > y) {
        int diff = x - y;
    }
    return diff;
}
```
5 }  
6 else {  
7 int diff = y - x;  
8 }  
9 return diff;  
10 }

Nope, still not right. First of all, the variable diff declared on line 4 is not the same as the one declared on line 7. Since they have different scopes, they will be considered different by the compiler. Regardless, neither is in scope when you get to line 10.

So, you try to fix it by doing this:

1 int abs_diff (int x, int y) {  
2     int diff = 0;  
3     if (x > y) {  
4         int diff = x - y;  
5     }  
6     else {  
7         int diff = y - x;  
8     }  
9     return diff;  
10 }

Ah ha! This actually compiles. Now we're getting somewhere, right? Sadly, no.

By declaring a variable called diff on line 6, you're creating a new variable and the compiler will think that's the one in which you want to store \( x - y \). So when you get to line 12, the diff there is not the one declared on line 6 (which is no longer in scope), but rather is referring to the one declared on line 3, which is still 0. Oops.

A way to think about it is that, in the example above, you've basically written this code:

1 int abs_diff (int x, int y) {  
2     int diff = 0;  
3     if (x > y) {  
4         int diff1 = x - y;  
5     }  
6     else {  
7         int diff2 = y - x;  
8     }  
9 }

Now it's clear that what's used on line 12 is different from what's declared on lines 6 and 9. Put simply, if you're declaring the same variable more than once in the same function, then you're probably doing something wrong.

Of course, the correct implementation is:

```c
int abs_diff (int x, int y) {
    int diff = 0;
    if (x > y) {
        diff = x - y;
    }
    else {
        diff = y - x;
    }
    return diff;
}
```

**Call-by-value**

One last important aspect of using functions in C is to understand that, when you use a variable as an argument to a function, its *value* is passed. This is known as a “call-by-value” approach (as opposed to “call-by-reference”).

Here's an example:

```c
#include <stdlib.h>
#include <stdio.h>

int swap(int, int);

main() {
    int a = 5, b = 9;
    swap(a, b);
    printf("now a is %d\n", a);
}

int swap(int x, int y) {
```
Let's start by looking at the function `swap`. As you can tell, it swaps the values of `x` and `y` (so that `x` has the value of `y` when the function ends, and vice versa).

So what gets printed on line 11? Is a 5? Or is it 9?

It's still 5. Why? Because when `swap` is called, the parameter `x` takes on the value of `a`, which is 5. The parameter `x` is not an alias of `a` (i.e., it's not another name for `a`), it's a totally different variable that just so happens to have the same value of `a`. When we change `x`, though, we don't change `a`. So `a` is still 5 when we get back to line 11.

Why, then, is C a call-by-value language? We'll answer that next time....