CS 11 Data Structures and Algorithms

Lesson 8: Functions 3

Skip to Main Content

Section 8.1: Value-Returning Functions

In lesson 7 we saw two different ways in which to call functions. In most cases when we called a function we used the function call as if it were a statement -- in other words, we placed the function call on a line all by itself, followed by a semi-colon, like this:

```
drawHorizontalLine(width);
```

This makes sense because, as with other statements, when we call `drawHorizontalLine` we are giving the computer an instruction, telling it to do something. In at least one case, the case of the `rand()` function, we used the function call as if it were an expression, like this:

```
num1 = rand() % 101;
```

This makes sense because in this case we are not giving the computer an instruction, we are asking the computer to determine a value, which is what expressions do.

The first type of function, the type used as a statement, is called a **void** function. The second type of function, the type used as an expression, is called a **value-returning** function. Although we have called both types of functions, we have only defined our own void functions. We have not yet seen how to define our own value-returning functions. That is the task before us.

Before moving on, though, let me point out that most students learning about functions do not have any trouble learning how to define value-returning functions. The big stumbling block is usually understanding how to call them. So let me belabor the point just a bit more by using C++'s built in `pow` function as an example. If I want to print out the value of 2 raised to the power 3, I would write it like this:

```
cout << pow(2,3) << endl;
```

The mechanism here is the same as if I had written

```
cout << 7 * 5 << endl;
```

You can think of it like this. In both cases, C++ goes off and evaluates the expression. (In the first case the expression is `pow(2,3)`. In the second case the expression is `7 * 5`). When the expression is evaluated, C++ replaces the expression with the result. So after the expression has been evaluated in the first example, C++ sees the `cout` statement as

```
cout << 8 << endl;
```

(since 2 raised to the power 3 is 8). After the expression in the second example has been evaluated, C++ sees the `cout` statement as

```
cout << 35 << endl;
```

A mistake that many students make is to think that before you can use a value-returning function as an expression you have to call it first as a void function to "initialize" it. So I often see this in code:

```
pow(2,8);
cout << pow(2,8) << endl;
```

The insidious thing about this error is that your program will still work. C++ essentially ignores the first call to `pow`. However, code like this demonstrates a lack of understanding about how value-returning functions work, and will be penalized.

Defining a value-returning function is almost exactly the same as defining a void function. There are just two differences. First, you have to indicate the data-type of the value that is going to be returned. Second, you have to actually return the value.

As an example, let's design our own `pow` function. Unlike C++'s built in `pow` function, ours will deal only with non-negative integers. The very first word we write in our function definition is different than it would be if we were writing a void function. Since our function will be returning an int, we will write the word int instead of void in front of the function name. So our function header will look like this:

```
int pow(int base, int power)
```

It would be an interesting exercise to develop the algorithm for calculating the result of this function; however, in the interest of time and space, I will simply provide the code for you. You should study it to make sure you understand why it works. The only other new thing in the function that follows is the statement

```
return result;
```

This is the second difference between defining a void function and defining a value-returning function: a value-returning function must have a return statement which tells C++ what value the function will return. This is the value that will be substituted in the place of the function call in the calling function. Here is the final version of our `pow` function.

```
int pow(int base, int power)
{
    int result = 1;

    for (int count = 0; count < power; count++){
        result *= base;
    }

    return result;
}
```

There is one operator in this function which may be new to you. The `*=` operator is simply a shorthand notation for "multiply and assign". So the statement

```
result *= base;
```

is equivalent to the statement

```
result = result * base;
```

There are two ways to explain this operator. One way is to simply point out the above equivalency. Whenever you see the `*=` operator in code, simply replace it in your mind with the equivalent assignment statement. Another way is to describe the concept behind the operator. The idea is that `result *= base;` means "multiply the value stored in `result` by `base`". There is also a += operator for "add and assign", a -= operator for "subtract and assign", and a /= operator for "divide and assign."

Section 8.2: When to Use Value-Returning Functions

Value-returning functions are never required to complete a programming task. We could easily write all of our code without them. We can always use the pass-by-reference mechanism to communicate values back to the calling function instead of making the function a value-returning function. On the other hand, we could also write all of our code without void functions, using only value-returning functions. So why do we have two techniques for communicating values back to the calling function?

The answer is that having two options for how to call a function often allows us to write cleaner code. For example, if we didn't have value-returning functions, then instead of

```
cout << pow(2,3) << endl;
```

we would have to write

```
pow(2,3,answer);
cout << answer << endl;
```

Programmers have different preferences about when to use value-returning functions and when to use void functions. My own preference is to use value-returning functions only when:

a. there is exactly one value being communicated to the calling function, and
b. there is no input or output occurring in the value-returning function.

Some programmers prefer a less restricted usage. They prefer to use value-returning functions **any time** a value is being communicated back to the calling function. The rule for this class is:

> You **must** use a value-returning function if there is exactly one value being communicated to the calling function, and there is no input or output occurring in the function. You **may** use a value-returning function **any time** a value is being communicated back to the calling function.

Section 8.3: Where to Put the Return Statement

The issue of where to put the `return` statement in your function is also debatable. I used to tell my students to always put their `return` statements ONLY as the very last statement in their functions, and this approach does have some merit. I could argue that putting a `return` in the middle of a function violates the "single entry -- single exit" rule (see the Style Conventions section of the syllabus, under "Miscellaneous"). Code is more predictable if you can always assume that the first statement in a function is the first one to be executed and the last statement is the last to be executed. However, it turns out that allowing `return` statements in other places in a function can often lead to much cleaner code. Here's my recommendation: try writing the function with only one `return` statement (at the bottom), and then write it with multiple `return` statements, and choose the one that seems cleanest and clearest. Don't use a `return` statement in the middle of a function if it is just a desperate attempt to get out of a tight situation. Restructure your code instead.

Let's do another value-returning function example where we will use multiple `return` statements. At the same time we will illustrate another concept: many value-returning statements return the `bool` data-type. This type of value-returning function would typically be called inside an `if` or `while` condition. For example, C++ has a built in value-returning function named `isdigit`. It takes a single parameter, a character, and returns true if the character is a digit, false otherwise. It might be called like this:

```
cout << "enter a character: ";
cin >> ch;
if (isdigit(ch)){
    cout << "you entered a digit.";
}
```

It is very common for these `bool` value returning functions to be named "issomething". Other examples are `isalpha`, `islower`, `ispunct`, and `isspace`. Let's write our own `bool` value-returning function named `isLeapYear`. You may use this function in your Calendar program (project 2). The requirements given in the text for this program allow you to assume that any year that is divisible by 4 is a leap year. I'm going to require you to determine with complete accuracy whether or not a year is a leap year. So you'll want to pay close attention to this discussion.

First we need to know what is a leap year and what isn't. Here's the rule.

- If a year is not divisible by 4, it is NOT a leap year.
- If a year is divisible by 4, it IS a leap year
    - EXCEPT, if a year is divisible by 100, it is NOT a leap year,
        - EXCEPT, if a year is divisible by 400, it IS a leap year.

Notice that we lived through a pretty special year, since 2000 is divisible by 400. Was the year 2000 a leap year?

When we write our code for this function, it will actually turn out to be easier if we turn the rule above around and start with what we know for sure. We know for sure that if a year is divisible by 400, it is a leap year. After that, once we have taken care of the "divisible by 400" case, we then know for sure that if the year is divisible by 100, then it is NOT a leap year. And so on. Here's a first shot at how we would code that:

```
bool isLeapYear(int year)
{
    if (year % 400 == 0){
        return true;
    }

    if (year % 100 == 0){
        return false;
    }

    if (year % 4 == 0){
```

```
            return true;
        }

        return false;
    }
```

Notice, however, that the code in gray above is equivalent to simply

```
return year % 4 == 0;
```

If you aren't sure you believe that, check what gets returned if year is divisible by 4, and then check what gets returned if year is not divisible by 4. You will find that both code segments return the same thing in any situation, and are therefore equivalent. So, making this simplification, our code becomes:

```
    bool isLeapYear(int year)
    {
        if (year % 400 == 0){
            return true;
        }

        if (year % 100 == 0){
            return false;
        }

        return year % 4 == 0;
    }
```

Notice that although we only want to execute the second `if` statement if the first condition is false, we don't need to use `else` here, because when C++ hits the `return` statement the function immediately ends.

In my in-person classes I spend quite a bit of time this week developing a large program interactively with the students. I write a program that adds, subtracts, multiplies, and divides roman numbers. We assume that the value of roman numbers is determined by simply adding the value of the digits together. For example, the value of IX is 11. This type of exercise is hard to reproduce in the online setting; however, here is the **final product**. My suggestion is that you try to write the program on your own, then peek at the solution when you get stuck. Only peek at enough of the solution to get you back on track (for example, just the main function), and then try again. When you finish, compare your solution to my solution. You will gain a lot more from this exercise than you will from just staring at the final product.

Here's an example of the output that the program should produce:

```
enter the first number: VII
The first number is 7
enter the second number: IX
The second number is 11
Enter the desired arithmetic operation: +
The sum of VII and XI is XVIII(18)
end of program.
```