2.6 — Why functions are useful, and how to use them effectively

Now that we've covered what functions are and some of their basic capabilities, let's take a closer look at why they're useful.

Why use functions?

New programmers often ask, "Can't we just put all the code inside the *main* function?" For simple programs, you absolutely can. However, functions provide a number of benefits that make them extremely useful in programs of non-trivial length or complexity.

- Organization -- As programs grow in complexity, having all the code live inside the main() function becomes increasingly complicated. A function is almost like a mini-program that we can write separately from the main program, without having to think about the rest of the program while we write it. This allows us to reduce a complicated program into smaller, more manageable chunks, which reduces the overall complexity of our program.
- Reusability -- Once a function is written, it can be called multiple times from within the program. This avoids duplicated code ("Don't Repeat Yourself") and minimizes the probability of copy/paste errors. Functions can also be shared with other programs, reducing the amount of code that has to be written from scratch (and retested) each time.
- Testing -- Because functions reduce code redundancy, there's less code to test in the first place. Also because functions are self-contained, once we've tested a function to ensure it works, we don't need to test it again unless we change it. This reduces the amount of code we have to test at one time, making it much easier to find bugs (or avoid them in the first place).
- Extensibility -- When we need to extend our program to handle a case it didn't handle before, functions allow us to make the change in one place and have that change take effect every time the function is called.
- Abstraction -- In order to use a function, you only need to know its name, inputs, outputs, and where it lives. You don't need to know how it works, or what other code it's dependent upon to use it. This lowers the amount of knowledge required to use other people's code (including everything in the standard library).

Effectively using functions

One of the biggest challenges new programmers encounter (besides learning the language) is understanding when and how to use functions effectively. Here are a few basic guidelines for writing functions:

- Groups of statements that appear more than once in a program should generally be made into a function. For example, if we're reading input from the user multiple times in the same way, that's a great candidate for a function. If we output something in the same way in multiple places, that's also a great candidate for a function.
- Code that has a well-defined set of inputs and outputs is a good candidate for a function, (particularly if it is complicated). For example, if we have a list of items that we want to sort, the code to do the sorting would make a great function, even if it's only done once. The input is the unsorted list, and the output is the sorted list. Another good prospective function would be code that simulates the roll of a 6-sided dice. Your current program might only use that in one place, but if you turn it into a function, it's ready to be reused if you later extend your program or in a future program.
- A function should generally perform one (and only one) task.
- When a function becomes too long, too complicated, or hard to understand, it can be split into multiple sub-functions. This is called **refactoring**. We talk more about refactoring in lesson 3.10 -- Finding issues before they become problems.

Typically, when learning C++, you will write a lot of programs that involve 3 subtasks:

1. Reading inputs from the user
2. Calculating a value from the inputs
3. Printing the calculated value

For trivial programs (e.g. less than 20 lines of code), some or all of these can be done in function *main*. However, for longer programs (or just for practice) each of these is a good candidate for an individual function.

New programmers often combine calculating a value and printing the calculated value into a single function. However, this violates the "one task" rule of thumb for functions. A function that calculates a value should return the value to the caller and let the caller decide what to do with the calculated value (such as call another function to print the value).

Next lesson
2.7Forward declarations and definitions
Back to table of contents
Previous lesson
2.5Introduction to local scope