Functions

Functions are a way to organize chunks of your code together. A function is a group of statements about a related subtask that is bundled together by a name. For example, we might have a *getLetter* function that asks the user for a lowercase letter, or an *averageNumbers* function that averages a list of numbers. Functions are important because they allow your code to be modular --- you write a function once, and you can use that function over and over.

We have already seen some functions. main() is the most important function in a program, because it contains all the code necessary to start the program and is responsible for calling other functions. Let's take a second to review the setup of main:

```
int main(){
      //code goes here
}
```

We begin by declaring the *return type* of main to be an integer. This tells the computer that we intend for main to send back an integer when it is finished. Usually, this is accomplished through a *return* statement, and, by convention, main returns a 1 if there was an error and 0 if the program completes successfully. We haven't been including a return statement and, for main (and, really, only main!), it isn't absolutely necessary. We'll talk more about return types for other functions soon.

We then write the name of the function, main, and follow it with a set of parentheses. These parentheses are for parameters. Parameters, things like numbers or variables, that you send to a function so that it can use them in its code. For our purposes, they are empty for main, but we'll see them in action in other functions soon. Finally we have a set of curly braces, and we put the code that belongs to main inside the curly braces.

Let's try writing a very simple function that prints the greeting "Hello World!". We will call (run) this function from main. It won't take any parameters or return anything, so the return type of this function will be *void*. Here's how we would set it up:

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

void printGreeting(){
        Cout << "Hello World!\n";
}

int main(){
        printGreeting();
}
```

Here, we made the *printGreeting* function exactly like we made main. We declared its return type to be void, named it printGreeting, and put no parameters inside the parentheses. Inside the curly braces, we wrote the same code we used for our very first C++ program.

What's interesting is how we call, or run, the function. Inside of main, we call the function by naming it and then putting the arguments, or values to be assigned to the parameters, we want to send it inside of parentheses. We don't want to send it any arguments since it doesn't accept any parameters (more on this in a bit), so we leave the parentheses blank. As always, we end the line with a semi-colon.

The way that this program works is the following: the computer begins running the program inside the main function. When it gets to the *printGreeting* function, it "jumps" up to where we defined the *printGreeting* function and starts executing that code. When it gets to the end of the printGreeting function, it jumps back down to the main function exactly where it left off and continues executing the rest of the code in main.

Let's look at a slightly more complicated function that uses parameters. In this example, we're going to change the *printGreeting* function so that it prints out "Hello *name*", where *name* is some name that the user enters. *printGreeting* will be sent the name to print out from main. So, the general layout of this program will be: start in main, get a name from the user, send that name to *printGreeting*, print out the hello message, go back to main. It sounds complicated, but it's really pretty easy.

When a function takes a parameter, we need to tell the computer what kind of parameter to expect. So, since we want to give *printGreeting* a name, the parameter will be a string. We also want to give the computer the name of the parameter to expect, which we can use inside the *printGreeting* function (but only inside this function). So we will call it *name*. So, the *printGreeting* function will look like this, with the name variable used in the *cout* function:

```
void printGreeting(string name){
        Cout << "Hello " << name << "!\n";
}
```

Now the computer knows that anything that calls the *printGreeting* function will need to provide a string argument that will be called *name* inside the function. So, in order to call the function, we need to get some input from the user and provide that input when we call the *printGreeting* function from main:

```
int main(){
        string userName;
        cout << "Enter a name: ";
        cin >> userName;
        printGreeting(userName);
}
```

Here, we got a string *userName* from the user and sent it to the *printGreeting* function by placing it inside the parentheses.

Before we go any further, we need to have a discussion about the difference between arguments and parameters. Arguments are what you *send* to a function, and parameters are what a function expects to *receive*. When the *printGreeting* function is called from main, the argument *userName's* value is copied to the parameter *name* in *printGreeting*. While the computer is executing the code in *printGreeting*, it can't use the variable *userName*, because that variable is only defined (available) in main. So that's why we have a parameter *name*- the value of *userName* gets copied to *name* so that the program has access to that data. When the computer finished executing *printGreeting* and returns to main, it can again use the variable *userName*, but it can no longer use *name*. This is what is known as the scope of a variable, which means where the variable is defined and can be used.

The reason that functions are so great is that we can reuse them and give them different parameters. For example, without changing the *printGreeting* function at all, we can use it to print greetings to three different people:

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

void printGreeting(string name){
```

```
        cout << "Hello " << name << "!\n";
}

int main(){
        printGreeting("Alison");
        printGreeting("Brian");
        printGreeting("Clifford the Big Red Dog");
}
```

And we would get the following output:

```
Hello Alison!
Hello Brian!
Hello Clifford the Big Red Dog!
```

That's pretty convenient. We can also use functions to do computation for us and return the answers back to main. For example, let's imagine that we want to write a function called *squareANum* that calculates the square of a number. We know from math class that the square of a number is that number multiplied by itself. So, let's write a function that takes a floating point number and squares it. After we figure out the square of the number (which we'll call *numSquared*), we'll need a way to return our answer back to the function that called it. To do this, we simply say:

```
return numSquared;
```

This is called the return statement, and it is always the **last** line in a function. Since we know that we are squaring floats, we will be returning a float, so the return type of this function will be a float as well. Now we're ready to write our function:

```
float squareANum(float num){
        float numSquared;

        numSquared = num * num;
        return numSquared;
}
```

We write the function as we normally would, and then tell the computer to return *numSquared* to the function that called it. Since we are returning something to main, we need to set up a variable to store the value that we are returning. This part of programming is like playing a game of catch. You can imagine that *squareANum* and main are playing a game of catch, and *squareANum* is throwing a ball named *numSquared* at main. If main doesn't catch the ball by having a variable ready to catch it, the ball will fall in the grass and be lost. In programming, if a function doesn't have a variable in place to receive a returned value, that value is gone forever. To prevent this from happening, we'll set our function call to *squareANum* equal to the variable *theNumSquared*. Here's the whole program:

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

float squareANum(float num){
        float numSquared;

        numSquared = num * num;
```

```
        return numSquared;
}

int main(){
        float numToSquare, theSquaredNumber;

        numToSquare = 5.5;
        //catch the returned value
        theSquaredNumber = squareANum(numToSquare);
        cout << numToSquare << " squared equals ";
        cout << theSquaredNumber << ".\n";
}
```

Note that you must either:

a) write a function's code earlier in the program text than it is called

b) include a prototype of that function at the top of the file.

A function prototype is just the return type, name, and parameter definition followed by a semicolon. Function prototypes usually appear before the first function and after the include files.

Challenge 1:

Write a program similar to the one that triples a number and adds 5 to it.

Finally, we can write functions that take multiple parameters of different kinds. For example, we can modify the *printGreeting* function so that it takes both a string name and an integer age, and prints out a message about that person's age:

```
#include <iostream>
#include <string>

using namespace std;

void printGreeting(string name, int age){
        cout << "Hello " << name << "!\n";
        cout << "You are " << age << " years old.\n";
}


int main(){
        printGreeting("Ronald McDonald", 49);
}
```

We just have to remember to send the function all parameters it expects as arguments when we call it. Although we can have different numbers and types of parameters, a function can only return one thing, so think carefully about what exactly you want to get back from a function before you write it.

Challenge 2:

Modify the *printGreeting* function to also take a string *birthdayMonth* parameter, and print out a message that tells them how old they will be the next time it is that month. For the Ronald McDonald example, the output might be, "You will be 50 next January".

Source