school of computing, informatics, & decision systems engineering

CSE 100, CIDSE, Fall B 2013
Lab 6: Functions

Pre-Lab: Please read the pre-lab before you start the Lab.

In this lab session, you will learn:

- when to use functions
- how to pseudocode functions
- how to call functions
- what the basic parts of function syntax are
- how to write custom functions

Earlier when we learned to write out algorithms, we noticed that some plans repeat steps over again. We were able to use loops to make repeating code easier to program. Similarly, functions allow us to write code that performs a specific task and can be called multiple times. This makes it possible to split up and organize code in a more efficient manner.

When writing your plan, you may notice similar steps being repeated. For example, in this pseudocode:

Get name of first puppy
Get breed of first puppy
Calculate cuteness factor of first puppy
Get name of second puppy
Get breed of second puppy
Calculate cuteness factor of second puppy
Compare cuteness factors
Output which puppy is cuter

The parts to get the name, breed, and cuteness of each puppy are very similar. In fact, the only real difference between them is that one is the "first" and one is the "second". It would be nice if we could shorten the pseudocode to look like this:

Get information for first puppy
Get information for second puppy
Compare cuteness factors
Output which puppy is cuter

With the magic of functions, now we can! ***Functions*** are subprograms (separated, independent sections of code) that perform a specific task. A function is like a machine in a factory: you put raw materials in the machine, it cranks some gears, blows some whistles, and hopefully doesn't catch fire, and out the other end comes a finished part. Similarly, when you call a function, you pass some values into it, it performs some computations, and returns a value. We'll cover the details of how that works later.

There are two parts to using functions: writing them and calling them. Writing them is more complex, so we'll address calling them first. Here are some examples of calling functions:

```
finalDistance = abs(initialDistance);
printf("I've got a lovely bunch of coconuts!");
system("PAUSE");
myScore = pow(yourScore, 2);
```

The first example is the function *abs*, which takes a number and returns its absolute value. To call it, you take a variable (*finalDistance*, in this case) and set it equal to *abs*. This means that whatever value is returned by *abs* will be assigned to that variable. Then, inside the parentheses, you give the name of a variable or variables that you want to have serve as the input (*initialDistance*, in this case). Each thing you put in between the parentheses is called an ***argument*** or a ***parameter*** to the function. Most functions get passed at least one argument, and many get passed more. For instance, the function *pow* takes its first argument *(yourScore*, in the example) and raises it to the power of the second argument (2, in the example). So in the example, if *yourScore* were equal to 10, then *pow* would set *myScore* equal to 100.

C++ already comes with a bunch of functions built-in. ***Predefined functions*** are bits of code already included with the language that help with common programming tasks, such as mathematical computation, string manipulation, and file input & output. Some of the more commonly used predefined functions in C++ come from the *cmath* library. This includes a bunch of mathematical functions that end up being used a lot by programmers *(abs* and *pow* in the examples above are two such functions). Since they've already been written, all you have to do is include them in your code, and you can start using them. Here's what the include statement would look like for *cmath*:

```
#include <cmath>
```

Now, if you want to make your own functions, you'll have to write them yourself. Before writing your own functions, it's a good idea to figure out what that function will do. The easiest way to do this is to write pseudocode for the function. Here's an example of pseudocode for a function that would get & calculate information for our puppy program (mentioned earlier):

getPuppyInfo
    get name of puppy
    get age of puppy
    calculate cuteness factor
    return cuteness factor

After we figure out what the function will do, we need to figure out parameters the function will need to have. In this case, we lucked out, and *getPuppyInfo* doesn't need any parameters. So, to call *getPuppyInfo*, we might do this:

```
puppy1cuteFactor = getPuppyInfo();
puppy2cuteFactor = getPuppyInfo();
```

Note that you still need the parentheses, even though nothing is being passed to *getPuppyInfo*. Also, note that we can call the same function twice and assign the returned result to different variables.

To actually write a function, we need six magical items. Here's what an implementation of the function *abs* might look like when coded, we'll use it for reference:

```
int abs(int value)
{
   if(value < 0)
   {
      value *= -1;
   }
   return value;
}
```

The six things we need are:

1. **The return type**
   The return type comes right before the name of the function, and tells C++ what kind of value (if any) will be coming out of the function when it's done. In our case, *abs* will return an int. Any C++ type can be the return type for a function (int, string, double, etc.). In addition, a special type, void, can be the return type. This tells C++ that no value will be returned when the function is done. Most normal functions return some kind of value, so don't worry about void for now.
2. **The function name**
   Captain Obvious says, "The function name is the name your code will use to call it!" Function names are a lot like variables: they must be unique, they can't be "reserved words" (like return, main, namespace, etc.), they have to start with a letter, and they can have letters, numbers, and the underscore (_) in them. Also like variables, the more descriptive a function's name, the better. So a function named *calcSalesTax* is much more useful than a function called *cst*.
3. **The argument(s)**
   The arguments, or parameters, of the function are the variables or values that get passed into the function when it starts. Each argument for a function is a pair of a type and a variable name (in *abs*, this is *int value*). The name of the variable only works inside the function, so as long as it's different from your other variable names, you can call it whatever you want. Multiple arguments are separated by commas (so the arguments for the *pow* function might look like this: *double base, double exponent*). Make sure to put the argument list in the parentheses!

These three things form what's called the ***function prototype***. They are also known as the ***header*** of the function, and they tell C++ in a nutshell what the function will do. If you end the argument list with a semicolon, you can skip the body of the function (described below) and code that later on in your program. Check out Chapter 6 in your book for more info on function prototypes. Briefly, were we to write the *abs* function in prototyped style, here's what it might look like in a program (ellipses indicate "snipped" portions of code):

```
...
using namespace std;
...
int abs(int value);
...
int main()
{
   ...
}
...
int abs(int value)
{
   if(value < 0)
   {
      value *= -1;
   }
   return value;
}
```

This allows us to tell C++ early on in our code that we have a function called *abs*, but it helps keep the code looking clean by moving the nitty-gritty of the function to the end of the .cpp file. If you don't want to use function prototypes, you'd just code the functions as you declare them (so in the above example, you'd replace the "*int abs(int value);*" line with the complete *abs* function definition & code that's after *int main*.

Speaking of code, there are three more things we need to finish the function:

4. **The braces**
   Like *switch* statements and most *do while* loops, a function needs to have all of its code inside curly braces ({ }). The basic explanation is that the braces group together the code so that it can be properly tied to the function.
5. **The code**
   Captain Obvious says, "Every function must have some code in it, here's where the code goes!"
6. **The return statement**
   Every function (even ones with return type *void*) must end with a return statement. This is the statement that says, "the function is done now, go back to the main program." If there is a return type, this is where you pass that value back to whatever called the function (in the case of *abs*, we used "*return value;*" to pass back the results of our calculation). If the return type is *void*, the return statement is just "*return;*". After this line, no more code from the function will get run, so make sure to put it at the end of your function.

And there's your function! Your programs can have as many functions as you want, and you can have functions call other functions as part of their code (you won't need to do that here, though). For added insanity, you can even have a function call itself as part of its code (this is called "recursion"). However, if you mess recursion up, you can easily end up in an infinite loop, and infinite loops cause puppies to be kicked, so don't play around with recursion until later.

Now that you've learned all this cool stuff about functions, take another look at *main* in one of your programs. Guess what, *main* is a function too! Hopefully now some of the syntax and conventions for *main* make a little more sense (e.g., "why do we need that silly *return 0;* at the end?").

Lab 6: Submission Required

One of the problems that have been discussed in video lectures is to write a simple C++ program to determine the area and circumference of a circle of a given radius. In this lab you will be rewriting the program again but this time you will be writing some functions to take care of the user input and math. Specifically, your main function should look like the following:

```
int main()
{
      double radius;           //the radius of the circle
      double area;             //the area of the circle
      double circumference;    //the circumference of the circle

      //get the value of the radius from the user
      radius = getRadius();

      //determine the area and circumference
      area = findArea(radius);
      circumference = findCircumference(radius);

      //output the results
      cout << "A circle of radius " << radius << " has an area of: " << area <<endl;
      cout << "and a circumference of: "<< circumference << endl;

      system("PAUSE");
      return 0;
}
```

Your job will be to write these three functions:

1) getRadius
   This function should ask the user to enter the value for the radius and return that value.
2) findArea
   This function should take the radius as a parameter and calculate and return the area.
   (Hint: area = 3.14159*radius*radius)
3) findCircumference
   This function should take the radius as a parameter and calculate and return the circumference.
   (Hint: circumference = 2*3.14159*radius)

OPTIONAL: Use prototypes for your functions. Place the prototypes before *main* (but after the *using namespace std;* line) and place the actual functions after the *main* function.

A sample run of your program should look like this (bold bracketed text stands for user input):

```
Enter the radius of the circle: [3.5]
A circle of radius 3.5 has an area of: 38.4845
and a circumference of: 21.9911
Press any key to continue . . .
```

Submission Instructions

Save your program as Lab6.cpp and submit it.