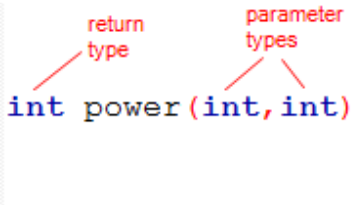
Computer Science 3-4
Problem Set #10: Finally…Functions!

One of things you may have noticed about C++ so far is that we use quite a lot of built in functions to do things. For instance, the math functions *sqrt(x)* and *pow(x,y)* simplify finding the square root of a number (do you even know how to calculate a square root by hand?) and raising a number to a power. What is great about programming is that we can make our own functions and use them over and over in our program, as long as we define them in a particular way.

Fundamentally there are two parts to making our function: *function declaration* (sometimes called the *function prototype*) and *function definition*. They work just like they sound…declaration is just outlining that we have some function and later on in the program we actually define the code that makes the function work. To declare a function, we need to do three things: decide the *return type* of the function (if it even returns a value at all), give it a clever name and decide if/how many parameters it accepts (and what kinds). Take a quick look at the declaration of a simple power function:



This is just letting the compiler know that there is going to be a new function called power, it is going to take in two integers as parameters and then return a single integer wherever the function was called. Now let's look at the actual definition of the function that would hold all the "guts:"

```
int power(int base, int exponent)    // we give names to our parameters
{                                    // these are "local" to this function only
    int total = 1;                   // another local variable
    if(exponent != 0)
        for(int i = 0; i < exponent; i++)    // some repeated multiplication
            total*=base;

    return total;                    // return our calculated value

}
```

We can see in the first line of the function, we've actually given names to integer variables we are going to use. It is important to realize that these names will only be used inside this function. The fancy name for this is *encapsulation* but it really just means that the variables are local to the function and are hidden from the rest of the. The *return* command does exactly what it sounds like; it sends the value of your variable back to the place you called the function.

You can only return one item from a function, it acts just like a break statement…if you get to the *return* it sends a value back and doesn't look at the rest of your function, so it can be a handy trick to put these inside of if statements if you have specific actions for specific cases. We must make sure that what we return matches the return type we specified when we declared the function (here we have a return type of *int* and we are returning total, which is an *int* so everything is kosher).

Suppose we wanted to make a function that didn't actually return a value, it just performed some task (like printing some text to the screen). Then we'd declare our function with type *void* (i.e. don't expect anything to be returned).

This is really a topic that is easier with an example, so take a look at this program that uses custom made functions (if you check the site this code is available for download) and make sure you watch the video where I go over this topic:

```
1   #include <iostream>
2   #include <cstdlib>
3   #include <time.h>
4   #include <string>
5   using namespace std;
6
7   int rollDice();           // These are the functions we are making
8                             // we need to declare them BEFORE the main loop
9   void winMessage(string);    // void means we aren't returning a value, int
10                            // means we are returning an integer, the (string)
11                            // lets us know that we are going to send a variable
12                            // of type string to the function
13
14  int main()
15  {
16
17  int player1Roll;          // We need variables to hold the player rolls
18  int player2Roll;
19  string player1Name;
20  string player2Name;
21
22  srand(time(NULL));        // Set the random seed to current time
23
24  cout << "Player 1 Enter Your Name: ";     // players enter their names
25  getline(cin, player1Name);       // this lets us get a first and last name
26                                           // because it takes in the whole line of text
27
28  cout << "Player 2 Enter Your Name: ";
29  getline(cin, player2Name);
30
```

```
31  cout << player1Name << ", press any key to roll the dice!"<< endl;
32  system("PAUSE");                    // This pauses for a key press
33  player1Roll=rollDice();             // we use our function to generate a value
34                                      // for each player, it is defined below
35  cout << player2Name << ", press any key to roll the dice!"<< endl;
36  system("PAUSE");
37  player2Roll = rollDice();
38  cout << endl << endl;               //create a bit of space before the winner message
39  if(player1Roll>player2Roll)      // Check to see if player 1 is a winner!
40      winMessage(player1Name);     // Here is a call to our other function
41
42      else if (player1Roll == player2Roll)      // Check for a tie
43              cout << "You Tie!"<<endl;
44          else
45              winMessage(player2Name);      // if neither a tie or player 1, then player 2 wins
46
47
48      return 0;
49  }
50
51  // this is the end of the main loop, we define our functions down here!
52  //Our functions look a lot like the main loop, the start off with a declaration and everything that they do
53  // is written inside of curly braces
54
55   int rollDice()             // int at the start of our function lets us return a single value
56   {
57   // note that any variables we create exist only in the function
58   // and we can't use any variables we define in the main
59
60   int die1 = rand()%6+1;         // roll the first die
61   int die2 = rand()%6+1;        // roll the second die
62
63   int sum = die1 + die2;        // find the sum
64   cout << "You've rolled a " << die1 << " and a " << die2 << endl;  // let the user know their roll
65   return sum;          // we return the sum value so we can use it in our main program
66
67   }
68
69   // this function takes a variable, the string, which is the player name and prints out a message
70   // using that name
71   // a function can take any number of variables, but can only return one variable
72   // this doesn't prevent us from printing messages to the screen or doing calculations though
73
74   void winMessage(string player)     // void means this function does something, but returns nothing
75   {
76   cout<< player << " you have emerged victorious!" << endl;
77
78   }
79
```

So before you move on, take a second to make sure you understand everything that is going on in this program. The main loop should be fairly straightforward, with the exception of the calls to our homemade functions. We've seen two types of functions here, one that returns a value "rollDice" and one that just prints text to the screen "winMessage" that also takes a variable. We had to declare the functions we were going to make at the top of our program (this is so the compiler knows to look for new functions so we don't get an error).

So the main reason that we want to make functions is that it helps us reuse code very effectively and keeps us from cluttering up our *main* loop with large blocks of repetitive code. Think back on the tic-tac-toe game…if you wrote a function that took in the values of the spots (or better yet took in an array that held the spots) and then drew the board, many of you would have saved quite a few lines of code. Similarly a checkValidMove(move,spot) function would have let you only write those if/else statements one time, then call on your function repeatedly! Later on we'll talk about a way to make libraries where you save your favorite functions so you can use them in all your programs!

Here are a few practice problems:

1. **Find the Smallest Number! -** Write a program that inputs three integers and passes them to a function that returns the smallest number.

2. **Prime Time –** Write a function that checks to see if a number is prime and returns either true or false (i.e. make your function a type *bool* and actually return a true or false value).

3. **Prime Factorization –** Write a function that will print out the prime factorization of any integer. You might find the work you did on the GCD useful and what you did on #2.

4. **Temp Changer –** Write two functions: one that converts from Fahrenheit to Celsius and one that converts from Celsius to Fahrenheit. Use the *setprecision* function to truncate the values to two decimal places. Use these to write a program that prints charts showing the Fahrenheit equivalents of all Celsius temps from 0 to 100 and all Celsius equivalent to Fahrenheit temps from 32 to 212…organize your output so it looks nice! Here's a link to more on *setprecision* for you:

   http://www.cplusplus.com/reference/iostream/manipulators/setprecision/

5. **More Rectangles! –** write a function that takes in three arguments: width, height and a character and then draws a rectangle using the specified character. For example, makeRectangle(4,3,%) should draw:

   %%%%
   %%%%
   %%%%

   Put this inside of a program that asks the user to input these three things, draws the shape and repeats until they enter -1 as the width.

6. **Reverser –** Write a function that takes in an integer value of any length and returns the number with its digits reversed (i.e. 1954 would be 4591).

7. **Perfect Numbers –** An integer is said to be a perfect number if the sum of its divisors, including 1 (but not the number itself), is equal to the number. For example, 6 is a perfect number since 1 + 2 + 3 = 6. Write a function called that checks to see if a number is a perfect number. Use this function in a program that determines and prints all the perfect numbers between 1 and 1000. Print the divisors of each perfect number to confirm that the number is indeed perfect!

8. **Holiday Challenge (a musical interlude) x2 –** if we add #include<windows.h> we can gain access to two fun little functions called *Beep()* and *Sleep()*. Beep takes two arguments, a frequency (in Hz) and a duration (in ms). So to play an "A" for a half second would be Beep(440,500). Sleep just takes a time in ms and acts like a rest (the system will not do anything for the specified amount of time), so a half second rest is Sleep(500). Use this information to write a synthesized version of your favorite holiday tune (you might find it useful to google the note frequencies if you aren't that musically inclined). Writing functions that play choruses might be useful (or you can get fancy and write your song as a function so you can play it in a different key or switch octaves).