What is a Function?

A function is a subprogram that completes a specific task.

In fact, main is really just a function. It's special in that it IS the only function that is directly executed.

main's specific task is to run the entire program.

However, in writing a program you may find it useful to use a function. Once a function is written (we'll talk about how to do this a bit later), we can use it. In fact, if someone else has written a function for us, we can use it as well!

All you have to do to use a function is adhere to the rules of CALLING that function. When we are calling a function, an analogy that might simplify our work is the black box analogy. You can think of a function as some sort of device you don't necessarily understand the inner workings of, such as a cell phone, the inside of which, is essentially a black box to you since you don't understand all the complex math and the materials that goes into transforming sound energy into electrical energy and then sending that signel through the air, etc. BUT, most of you can use a cell phone perfectly well.

All you have to do is follow the directions: For example, if you want to make a call, dial in the 10 digit number and then hit the send button.

In order to use a function, the rules we will have to follow are as follows:

1) Give the function information it needs to finish its task.
2) Use the information the function returns properly.

Pre and Post-conditions

I will talk about these more as we see examples of functions, but I want to at least give you an idea of what pre and post-conditions are, and why they are important.

One analogy that I often use to describe a function and its relationship to an program is a boss/employee analogy.

Consider that you are the boss of a company. Now, as we all know, the boss does NOT do all the work. (In fact, many would argue that they do little compared to others involved in the company!) However, the boss must have a good idea of the goal of the company, and also have a "game plan" for achieving that goal.

Certainly, there is too much work for the boss to do all by himself. However, he has employees!!! Thus, the boss's job is to find a way to utilize these employees so that he can achieve the company's goal.

This sounds easy enough, but there is some difficulty here. First of all, it seems extremely unlikely that each employee will understand the "big picture" the way the boss needs to. It would be difficult to understand the "big picture" AND be able to carry out specific tasks with the attention to detail that the employee is expected to observe. Thus, the boss can not assume that the employee understands what the overall goal of the company is and how his/her specific task relates to that goal. (Granted this should never be true in real life, but it aids my analogy, so play along!!! It will make more sense when I talk about the black box analogy.)

Also, the boss needs some way of conveying to the employee what he/she needs to do, so that the boss can be guaranteed that the employee is doing the right job, without having to spend extra time to check up on the employee. This means that there must be a way to specify the employee's job without ambiguity.

Consider this situation: Let's say the employee has some training and can do certain tasks. What the employee can do is tell the boss his capabilities, ie. what tasks he is capable of performing. In particular, the employee can tell the boss that "if these conditions hold..., then I will guarantee that I will do this.... for you."

In essence, one analogy we can use is that the employee has a contract. In that contract, the boss must meet certain requirements, (ie. provide health care, proper working conditions, etc.), and in return, the employee must provide something(build chairs, fix the assembly line when its broken, etc.).

This contract is made up of a pre-condition and a post-condition. This is the same the specifications for how a procedure or function works. Each procedure or function you write MUST have a list of pre-conditions and post-conditions. The contract is that IF the function's pre-conditions have been met by the function caller, THEN the function will correctly execute some task, as specified by the post-condition.

First, we will talk about how to call a function. In calling a function, we MUST adhere to the pre-conditions if we are to expect the function to work properly. If we don't, there is no guarantee as to what the function will do.

How to Call a Function

In some sense, you already know how to call a function. We've been calling pre-written functions in C (printf, scanf, sqrt, abs, etc.) for a while now. But, we haven't focused on the rules for doing so. The rules for calling a function you or someone else has written are the same, but now we'll carefully examine those rules in preparation for writing our own functions.

For the time being, let's assume that someone else has written functions for us, and if we decide we can use them, then we must properly integrate them into our program.

Now, consider the following function header, preconditions and postconditions:

```
// Precondition: The function takes in a single character as a
//                parameter. The character MUST BE one of
//                the following: 'A', 'B','C','D', or 'F'.
// Postcondition:The function will return a numerical value
//                in between 0 and 4 corresponding to the
//                parameter passed to it.
int Comp_Grade(char grade);
```

The function header follows the following syntax:

<return type> <function name>(<parameter list>);

The parameter list is a set of items separated by commas. A single item in a parameter list is of the form:

<type of parameter> <formal parameter name>

So, for the specific example above, we see that the function Comp_Grade returns a int value and it takes in exactly one char parameter.

It's very important to realize that the syntax for MAKING a function call is DIFFERENT than the function header looks!!!

In particular, to make a valid call to this function, we would have to do it as follows:

Comp_Grade(<some expression that evaluates to a char>)

In general, a valid call to a function is as follows:

<function name>(<list of actual parameters>)

The list of actual parameters is a set of expressions separated by commas, where each expression evaluates to the corresponding type listed on the function header.

I have now used the terms "formal paramter" and "actual parameter" without defining them. Let me do so now:

***Formal parameters*: These are the ones listed in the actual function definition, they are essentially local VARIABLES.**

***Actual parameters*: These are the EXPRESSIONS/VALUES that the calling function passes to the function being called.**

Key differences between a function CALL and function HEADER:

1) The return type is NOT written in the function CALL.
2) The types of the actual parameters are NOT written in the function CALL.

For this particular function, this entire expression evaluates to a int value. Thus, for this type of function call to be syntactically valid AND be useful, we must put it in a place where we would place a int. So, now, we will look at a program that makes a call to the Comp_Grade function. In particular, the program will calculate a student's GPA.

```
// The program calculates a student's GPA, by reading
// in information about all of their grades.
#include <stdio.h>
int main() {
        // Initialize variables used to compute GPA
        int total_points = 0, total_hours = 0, c_hours;
        char ans = 'y', my_grade;

        // Loop runs until student has entered all of their grades.
        while ((ans == 'y') || (ans == 'Y')) {

                // Get grade information for one class.
                printf("Enter your letter grade.\n");
                scanf("%c", &my_grade);
                printf("Enter the # of credit hours for that class.\n");
                scanf("%d", &c_hours);

                // Recompute points and hours based on new class
                total_points += c_hours*Comp_Grade(my_grade);
                total_hours += c_hours;

                // Check to see if student has another grade to enter.
                printf("Do you have another grade to enter?")
                scanf("%c", &ans)
        }
        // Print out final GPA.
        printf("Your GPA is %lf", (double)total_points/total_hours);
}
```

Formal and actual parameters

I'll define these again:

***Formal parameters*: These are the ones listed in the actual function definition, they are identifiers.**

***Actual parameters*: These are the VALUES that the calling function passes to the function being called.**

Rules that must be followed for a legal function call

1) **The number of actual parameters must equal the number of formal parameters**
2) **The types of each actual parameter must match the types of each formal parameter, in order.**
3) **Each actual parameter must be associated with the corresponding formal parameter in the function definition.**
4) **Actual parameter may be any expression if they are pass by value parameters.**

Thus, in our example, the variable my_grade used in the program was the actual parameter passed to the function Comp_Grade.

And, when we look at the function header for Comp_Grade, we find that the formal parameter used was grade.

As far as calling the function Comp_Grade, the important thing to realize is that we could have passed it any actual parameter that evaluated to a character. So, the following is syntactically correct:

```
printf("An A is worth %d points.\n",Comp_Grade('A'));
```

Class Exercise

Consider the following function headers, with pre and post-conditions:

```
// Pre-condition:  The value of the parameter passed in must be
//                 positive and in some type of units.
// Post-condition: The function will return the volume of the
//                 sphere with the given radius in units^3.
double Sphere_Volume(double radius);

// Pre-condition:  The value of both parameters must be
//                 positive. The unit used for volume must
//                 be the same used in the density.
// Post-condition: The function will return the mass of the
//                 object in the units used in the density.
double Mass(double den, double volume);
```

Use these to write an program to compute the mass of a snowman. Read in from the user the three radii (in centimeters) of the three "snowballs" used to create the snowman. You will also have to use the following constant, which is in gm/cm^3.

```
#define SNOW_DENSITY  0.1
```

```
#include <stdio.h>

#define     SNOW_DENSITY    0.1

// This program computes the mass of a snowman, based on
// information about the snowman's dimensions.
int main() {

    double r1, r2, r3, total_volume;

     // Read in size of each snowball.
    printf("Enter the size of each snowball in succession.\n");
    scanf("%lf%lf%lf", &r1, &r2, &r3);

    // Compute volume of snowman
    total_volume = Sphere_Volume(r1) + Sphere_Volume(r2) +
                        Sphere_Volume(r3);

    // Print out mass of snowman.
    printf(“Snowman mass = ”);
    printf("%lf\n", Mass(SNOW_DENSITY, total_volume));

}
```