Laboratory 3: *Modules May Be FUNctions!*

Overview: A good program design breaks a large problem into several small subproblems. These subproblems are then implemented in modules which are easy to identify in the program. When the module can be logically separated from the program, we can encapsulate the module implementation into a function. This modularity and encapsulation is what we strive for as good C++ programmers.

Objectives: We will spend some time today introducing the terminology necessary when implementing functions. Before you leave lab today, you should be familiar with the terms function prototype, function implementation, and function call or invocation. You should know how to write function prototypes and know where they may be placed in your source code. You should recognize function implementations and know how to write function calls. You should understand these terms in the context of programmer-defined functions, standard library functions, and member functions.

1. Setting up for the lab

Please perform the following activities as you wait for class to formally begin:

a. Boot up the machine and log on to the CS Department server, as described in Lab 1.

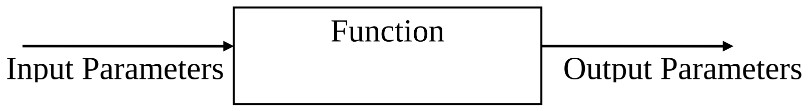
b. Put your lab diskette in the *A:* drive and make a subdirectory *LAB03* in the *LABS* directory.

c. Download the files from the class Web site to your new directory. Note that the URL is http://s9000.furman.edu/chealy/cs11/lab03.

2. Terminology

We have been discussing functions in class, but you may not recall all the terminology yet.. You must become comfortable with the terminology of functions so that you can communicate with other programmers effectively.

We have seen that a function is a module that has been physically separated from the rest of the program. It encapsulates the details of a process that are not necessary in the main program. A function can also receive input and produce values to output. We can conceptualize a function like a "black box" where values go in and values come out but we don't necessarily know how the output values are produced. For example:

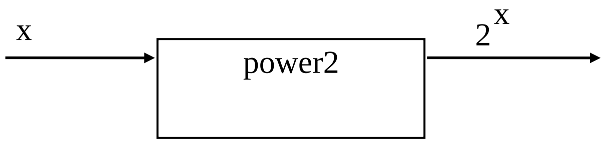


Prototypes - A prototype defines the interface for the function. The interface includes the function name, input parameters (i.e. the data you want to send to the function), and output parameters (i.e. the data you want returned from the function). You should always have a clear understanding of the interface before implementing the function. Let's look at some prototypes.

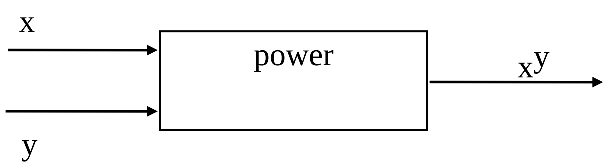
a. Start up Borland C++ and open the file *power.cpp*. Read through the code to try to understand what it is going to do.

b. Only the function prototypes must appear before the function is used in a program. You should notice that the details of the function are placed AFTER the main program rather than before it as we have been doing. Study the prototypes carefully for **power2** and **power** and compare them to the following function diagrams.

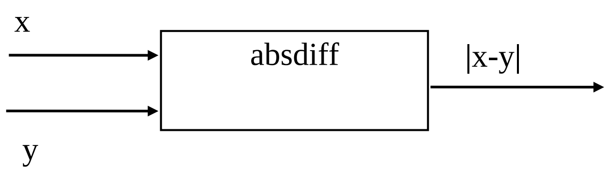
    The function **power2** is to return 2^x for a given integer x.



    The function **power** is to return x^y for given integers x and y.



    The function **absdiff** is to return the absolute difference between real numbers x and y (i.e. |x-y|).



c. Place prototypes for the above functions **power**, **power2** and **absdiff** before the main() statement in your program.

Function Implementations - A function implementation provides the formal parameter names and the instructions to execute when the function is called. The details of what the function is to do are in the implementation. If you look at the bottom of the source file *MATH1.CPP*, you will see function implementations for the three functions above. We will work on writing function implementations later. For now, simply look through these implementations.

Function Calls - A function call executes the function from within another function. This is called invoking a function. The function call must specify the actual parameters or values that are sent to the function. The order of the actual parameters must match the order of the formal parameters in the interface. However, notice that the actual parameter names used in the function call do not have to be the same as the formal parameter names used in the function implementation.

c. Add function calls for the above functions in the main program—**power**, **power2** and **absdiff**. Be sure to use the appropriate actual parameter names in the function call. Compile, run, and test your program. Don't forget to save it too. √

3. More Practice with Function Design

Not all functions are called directly from main(). You will see this in our next program that solves a quadratic equation.

a. Open the file *quad.cpp*, and look at the overall structure of the program to see what it is trying to do. Note that this program is not quite complete yet.

b. Complete the prototypes for the functions **discriminant**, **root1** and **root2**. Pay close attention to what parameters need to be used by these functions. (Hint: Look at the comments immediately above that describe the meaning of these 3 functions.)

c. Inside the main function, complete the statements that assign values to r1 and r2. You will need to pass the appropriate actual parameters to the functions.

d. At the bottom of the program are incomplete implementations of the three functions. Use the formulas given in the comments at the top of the program to finish writing these programs. Note that you will need to call discriminant() from both root1() and root2(). This is a good example of a nest of function calls: main() calls quad() which in turn calls root1() and root2(), and both these functions call discriminant().

e. Compile and run your program. Test it with input values 1, 5.5 and 6. If you don't get the results you expect, check the function calls and implementations again. √

4. Testing Functions Using a Driver

Function Testing

Remember that testing our programs is an important part of the Software Engineering Methodology we have been using. We have seen the importance of creating good test data which tests all conditions in our programs. Now we need to see how functions impact this testing process. As our problems become larger, our testing becomes more cumbersome because we have more code to test. However, if we have broken the code into modules and placed those modules into functions, we can individually test the functions very easily. Once we are convinced that the individual functions work properly, we can then put them together to test our program.

We can use a *test driver* to test a function. A test driver is simply a small program which does nothing but call a function many times. The inputs to the function are supplied by the user before the function is called, and the outputs from the function are displayed on the screen so that they can be verified. When a function needs to be tested, the function implementation is simply copied into the source file of the test driver, the driver is run to test the function, and then the function is copied back to where it is needed in your program. Let's practice this technique on the quad() function.

a. Copy the program *quad.cpp* into a new file called *quad2.cpp*.

b. Modify the main() function as follows. Take out the declarations for a, b and c and the code that gets the values of a, b and c from the user. In the function call to quad(), replace the variable parameters (a, b, c) with (1, 5.5, 6).

c. Compile and run the program. You should notice that the solutions to the equation are the same as we previously obtained with quad.cpp when the coefficients where entered manually.

d. Immediately after the call to quad(), add four more invocations to quad() so that we can solve more quadratic equations. Use the following sets of values for (a, b, c): (2, 0, -18), (9, 24, 16), (1, 7.6, 0) and (5, -32, 45.15).
e. Compile and run the program again, and notice that there are five solution outputs. √

5. Practice with Nested Calls

In a large program that contains many functions and function calls, it is important to know exactly which functions are called when. Because functions can be called from more than one place in the program, it is easy to get lost in the program if you are not aware of the nesting of function calls. In this last section, we will practice with an ugly-looking abstract program to get a handle on tracing through a program that has many similar function calls.

a. Examine the program *func.cpp* given on the next page. Trace through the program by hand in order to determine what you think the output of the program should be.
b. Open the file func.cpp in Borland C++. Compile and run the program.

c. Is the output of the program the same as what you predicted? If not, see where the first discrepancy is. If you have trouble figuring out the sequence of events, call me over.
d. Draw a picture that shows, throughout this program, which functions are called from which other functions. (In lecture we refered to this as a “call graph”.) Hint: when you are looking at the four functions a-d, notice that one of these functions is never called from the other three. Which one is it? √