Laboratory 4:
First Looks: IF Statement, Debugging tools

Overview: As your knowledge of the C++ language grows, your capacity to write more useful and complex programs also increases. Today we'll explore two facilities for increasing this capacity. First, we'll see how the **`if`** and **if-else** statements work for making decisions. Second, we'll see how we can use the debugging tools in Borland C++ to test our programs.

Objectives: Before you leave lab today, your goals should include: knowing how to use the **`if`** statement, boolean logic and basic decision making in programs; understanding how to design good sets of test data and how to use diagnostic output statements or the debugging tools in Borland C++ during testing.

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 *lab04* in the *labs* directory.

c. Download the files from the class Web site to your new directory. The URL is http://s9000.furman.edu/chealy/cs11/labs/lab04.

2. Inquiring about the IF statement

The **`if`** statement allows the programmer to make decisions. If some expression—called a *boolean expression*—is true, one action can be taken. If, instead, the expression is false, another action can be executed. The **`if`** statement can definitely be a problem for the careless programmer. The following will help you to better understand the **`if`** statement, so that you can avoid some of its more troublesome pitfalls.

a. Open the program *triangle.cpp*. This program expects three numbers in non-decreasing order as its input. It first determines whether the user correctly typed in the numbers. If he or she did not, the program terminates. If the instructions were followed, the program determines whether the numbers correspond to the sides of a triangle, and if they do, what kind of triangle.

b. Look over the program thoroughly to get an understanding of how it works. Try as input the numbers *5 4 9* (in that order). Observe what output is displayed.

c. Now try those numbers in the proper order: *4 5 9*. Again observe what output is displayed.

d. Come up with other test inputs that will cause the various other output messages to be displayed (i.e., inputs that correspond to the various kinds of triangles). Write your test inputs in the space below. Run the program to show that you chose correct input values. √

Now we'll use the **`if`** statement to help compute the value of a standard *boolean function*, that is, a function which has *True* (1) and *False* (0) for its input and output.

e. Open the file named *nand.cpp*. This file offers one solution to the *truth table* below for the *nand* function. Run it on the four possible input combinations to verify its correctness.

p	***Q***	***p NAND q***
False	*False*	*True*
False	*True*	*True*
True	*False*	*True*
True	*True*	*False*

f. Notice that it is only the last combination—*p* and *q* both being true—that causes the *nand* result to be *false*. Restructure your code so that it uses a single **`if`** and a single **`else`**. This can be accomplished by testing only whether or not the combination defined in the last row of the truth table is the one you have as input.

g. Make and run the program. Remember, if you get stuck, ask for help. √

3. Writing your own if statements

Now let's get some practice writing your own **if** statements from scratch. You and your partner should read through the following decision descriptions. Discuss them together if you have questions about how to interpret a decision.

Decision Descriptions

Segment a If *i* divided by *j* is 4, then *i* is set to 100.

Segment b If *i* times *j* is 8, then *i* is set to 50, otherwise *i* is doubled and *j* is set to 60.

Segment c If *i* is less than *j*, then *j* is doubled; if instead *i* is even, then *i* is doubled; otherwise, both *i* and *j* are incremented by 1.

Write down a statement in C++ which would implement each program segment. You may want to use pseudocode or flow charting first (especially on Segment c).

Statement for Segment a

Statement for Segment b

Statement for Segment c

Before you even test your code segments, you should determine some test sets for each segment. Good test sets will test *every* branch of your **if-else** statement. In other words, there should be a test case for every possible decision outcome. Take a few minutes with your partner to fill in the following tables with good test sets for each segment. Also, predict what i and j will hold after each segment is executed using the test cases.

Segment a

Test Values		Results after segment	
i	j	i	j

Segment b

Test Values		Results after segment	
i	j	i	j

Segment c

Test Values		Results after segment	
i	j	i	j

Now, you are ready to implement and test your **if** and/or **if-else** statements.

a. Open the source file *segments.cpp* from your *lab04* folder. This program will allow you to enter test cases for each of your three segments. Scroll through the file and find where you should place your segment code. Enter your three segments and compile until you do not have any syntax errors.

b. Run the program. Notice that you may enter several test cases for each segment by entering 'Y' when asked if there is more testing necessary. Run the program as often as necessary to test and retest until you have successfully matched the predicted outcome for each of your test cases. √

4. Tracking down problems in code

There are three kinds of errors—syntax, semantic and logical. Often, the code you or others write will go through the compiler with no difficulty (i.e., there are no syntax errors) and will even run with no errors reported (i.e., there are no semantic errors). Does this mean that it will work? Well, it will do *exactly* what you told it to do, and not necessarily what you want it to do. Frequently, we'll see programs that *look* correct and that compile just fine, but *do not* run properly (i.e., there are logical errors).

The source of programmer logical errors is often carelessness. Other times it is a lack of understanding about the use of a particular statement that controls the flow of execution in the program, such as the `if` statement we've been practicing with.

Diagnostic print statements can be used effectively to track down logical errors. Borland C++ also provides a powerful **Debugger** with tools to help you find software defects in your programs.

We will go back and work with the file *nand.cpp*, so close any windows for *segments.cpp* and open *nand.cpp.* You will probably have to compile and link the program again. The first tool we will explore is the *Step* feature. *Step*-ping through a program enables you to watch the flow of control in your program.

a. Try the *Step* option by selecting *Step Over* from the **Debug** menu. The first executable line of code in your program is highlighted and the program execution is halted. To execute this line you may select *Step Over* from the **Debug** menu or press the F8 key. The highlighted line is executed and the next executable line in your program is highlighted. Every time you press *F8*, Borland C++ will execute one more line of code.

b. Continue to step through the program. When you get to the *cout*, you must press F8 again to get to the *cin*, so that you'll be able to input. When you get to the **if-else** statement, notice that only one branch of the statement is highlighted (or executed). The other branches are completely ignored. Step all the way to the end of *nand.cpp*. You should now reset your program so that you may run it again. To do this, you can select the *Terminate Program* option from the **Debug** menu. Use the "Page-Up" key or the scroll bars to move to the beginning of *nand.cpp*.

Another feature the debugger offers is the *Watches* option. By using *Watches*, you are able to observe the value of variables at different points in the program.

c. Go up to the **Debug** menu and select *Add Watch*. When the dialog box comes up, enter the variable *p* into the box labeled *Expression* as the one you wish to watch. Then choose *Add Watch* again, and enter the variable *q*.

d. Use the *Step* function (*F8*) to walk again through your program one line at a time. Observe that once program execution begins and even before the variables are initialized in an assignment statement, that there is a value for the variables. The values are the ones left there from some previous use of the computer memory. Also, observe that the *Watch* box shows the values of both *p* and *q* as they change during the program's execution.

e. Reset your program.

A third feature that Borland provides is the *Breakpoint* option. By inserting a breakpoint, it is possible to simply run your program in the usual fashion (i.e., by choosing *Run*). When the program reaches the line where you have inserted a breakpoint, it pauses its execution and awaits your commands. Breakpoints are useful as you don't always want to step through large programs—it is often easier to let them run until they reach the place you want to observe. Let's insert a breakpoint into the *nand.cpp* code.

f. Use your mouse to select a line in the program that contains the first `if` statement. Choose the *Toggle Breakpoint* option from the **Debug** menu. You will see a red bar appear on the line you selected.

g. Now run the program. Observe that execution stops at the breakpoint you inserted. You now have several options:
   — Adding or removing *Watch* variables

— Pressing *F8* to execute the next line of code
— Choosing *Run* to continue the normal execution
— Choosing *Terminate Program* to start over
— Removing the breakpoint

h. Try removing the *Watch* on the variable *q*. To do this, click on the *q* in the Watch Window and press the "Delete" key. Continue the program execution by hitting *F8* or choosing *Run*. Be sure that you can stop a program on a specific line, add a watch on a variable object, and step through your program one line at a time. √

i. Breakpoints are removed by *Toggle*-ing them again.

5. Finishing up

a. Close all windows and exit Borland C++.

b. Exit all applications and shut down your computer.