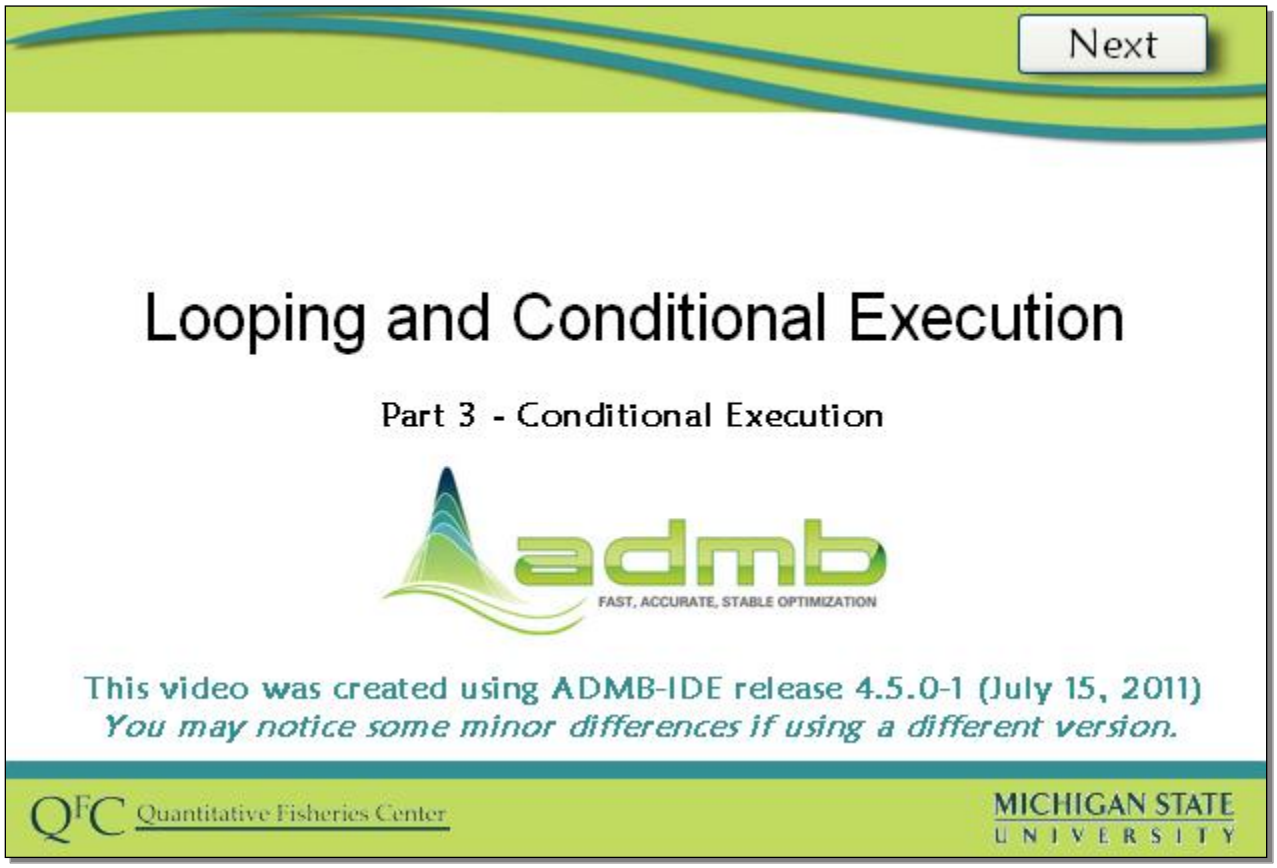
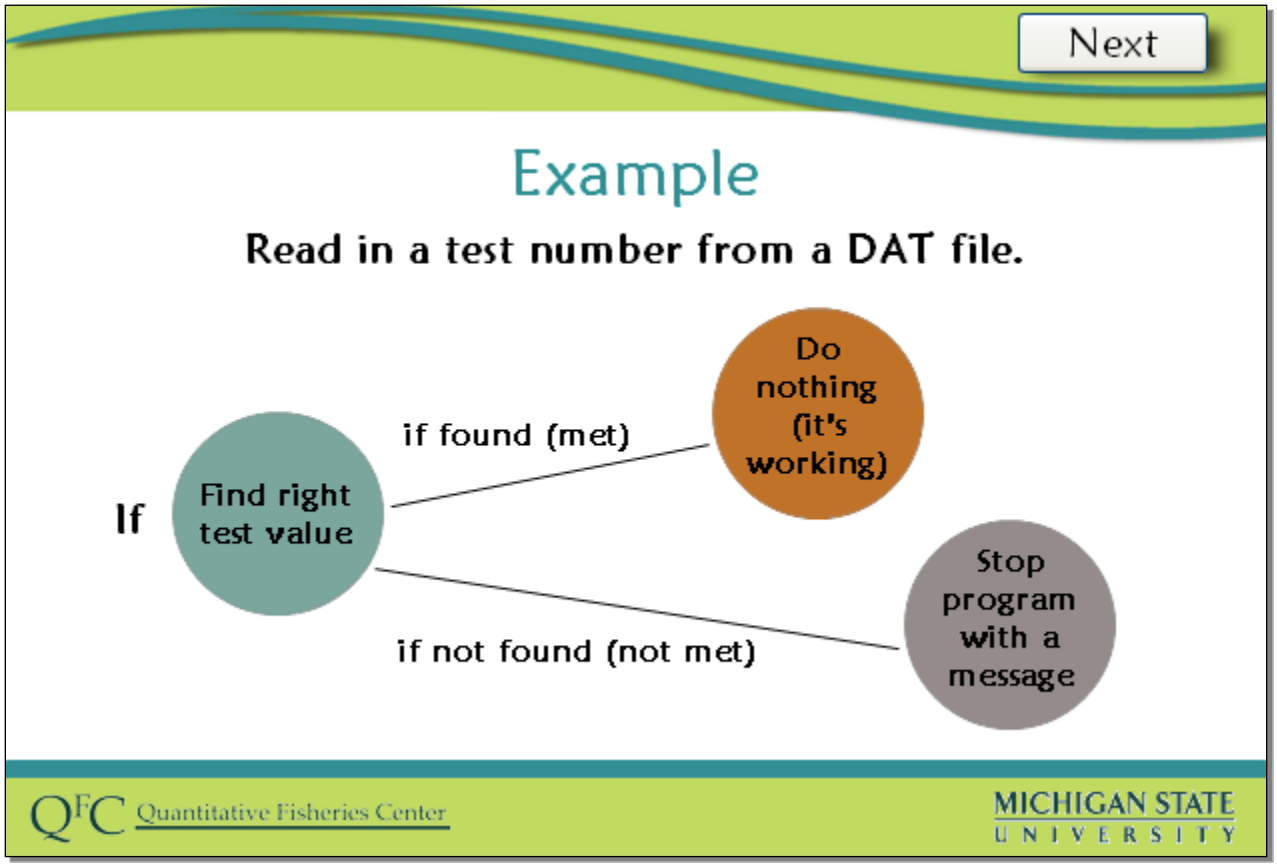
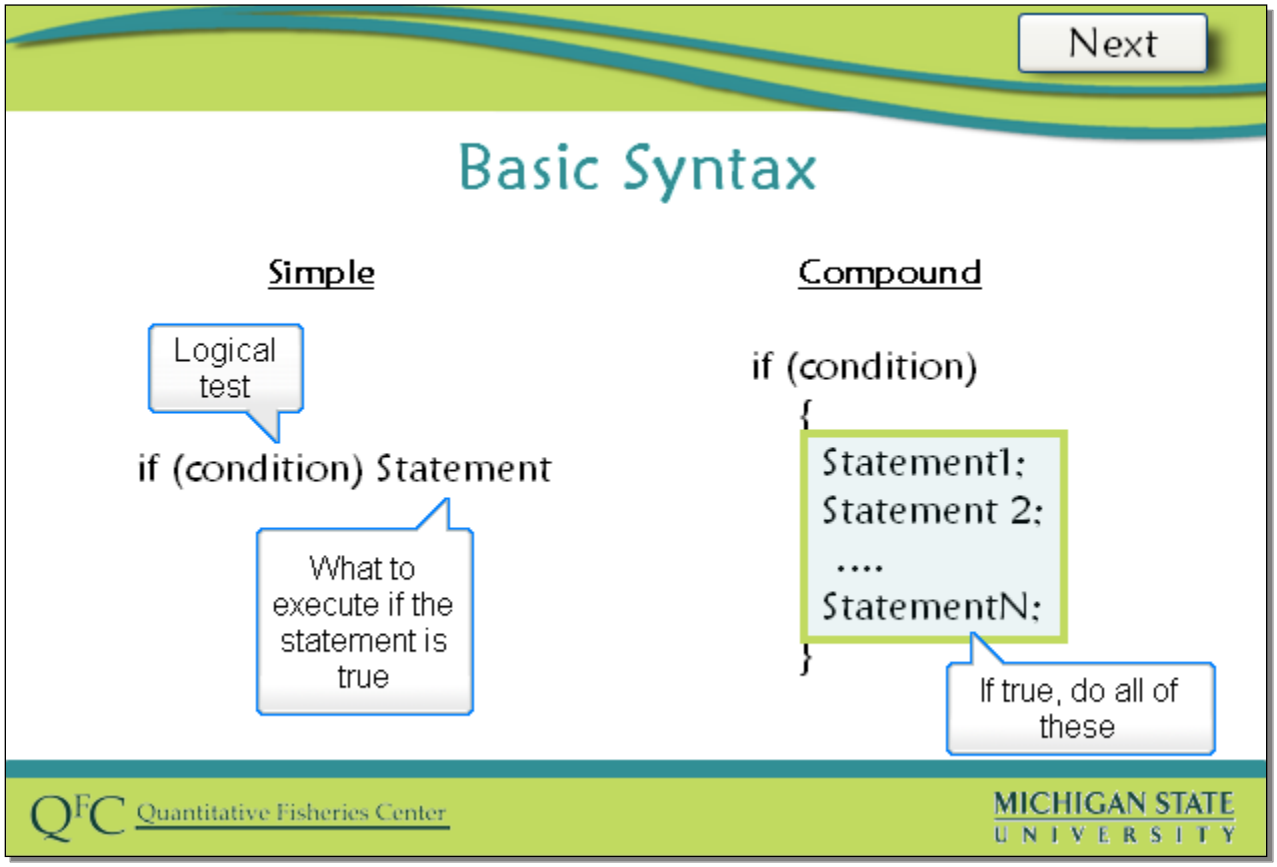


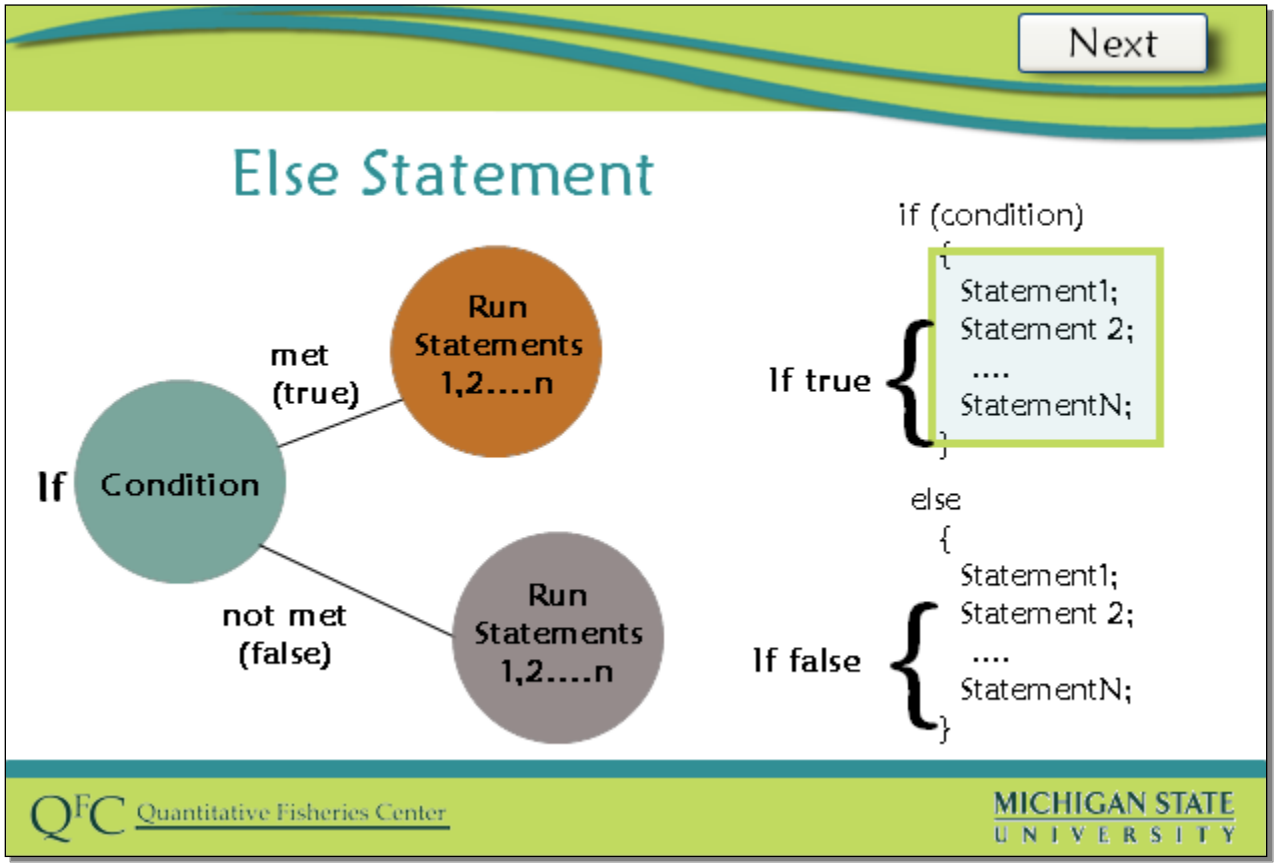
We now turn to the use of conditional statements. Particularly the if statement. Our intent is to control what happens based on a condition that is tested. First determine if you would like a little background on the if statement before we proceed to the coding.



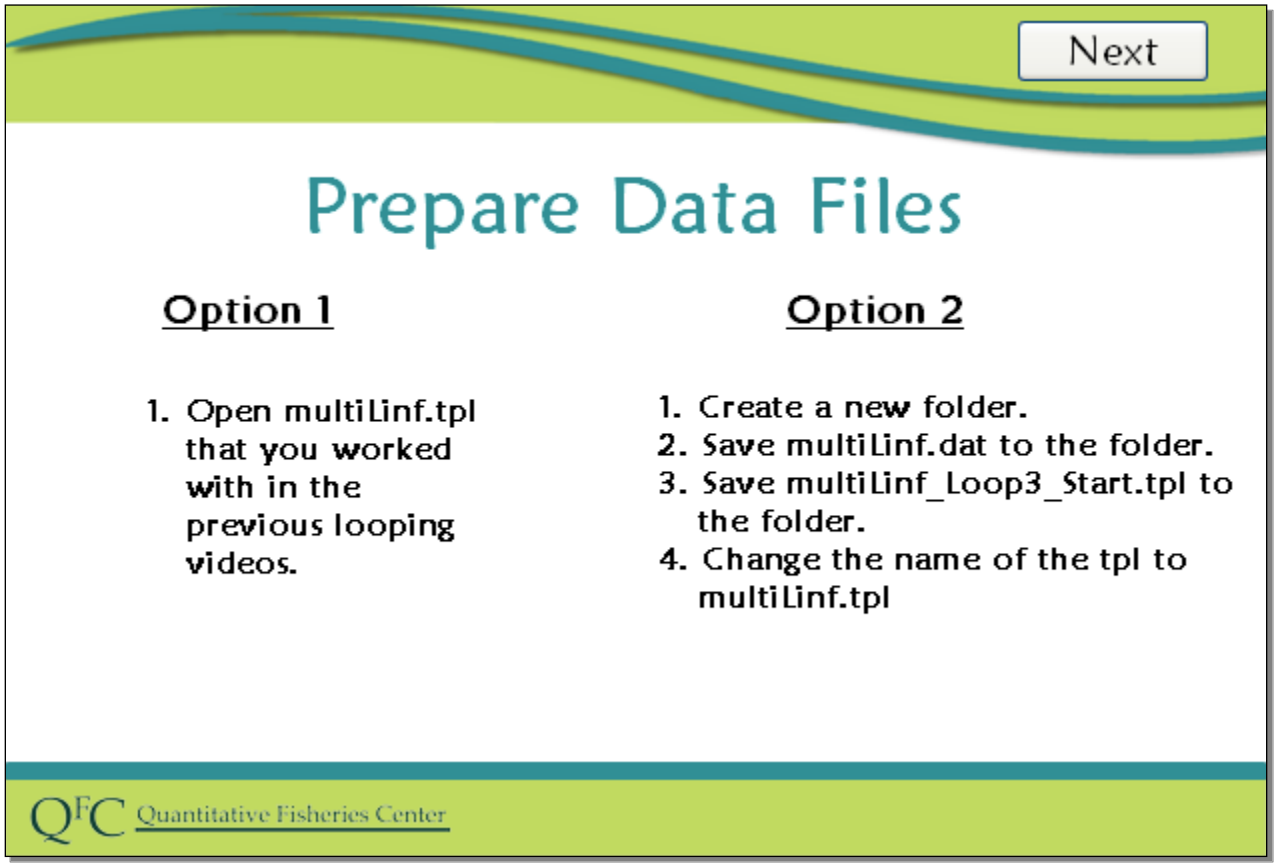
In our first example we will read in a test number from a DAT file and our program will stop with an appropriate message if the right test value is not found. We have advocated checking on a test value as a standard part of building a tpl file. However, even after an initial version is working there is a possibility that you make changes and forget to check that the data still read in correctly before proceeding. Our conditional approach will automate the check for us and not stop the program if things seem ok.



The basic syntax for a simple if statement is shown here. Here condition is a logical test and statement is the statement to be executed if the statement condition is true. Statement can be a compound statement so the if statement could be displayed in this form where there could be multiple statements that get executed when the condition is true.



Optionally you can also specify using an else statement what should happen only if the condition is not true. For example, these statements will run if the condition is true, and the statements after the else will run if the statement is false. The else statement always comes immediately after an if statement.

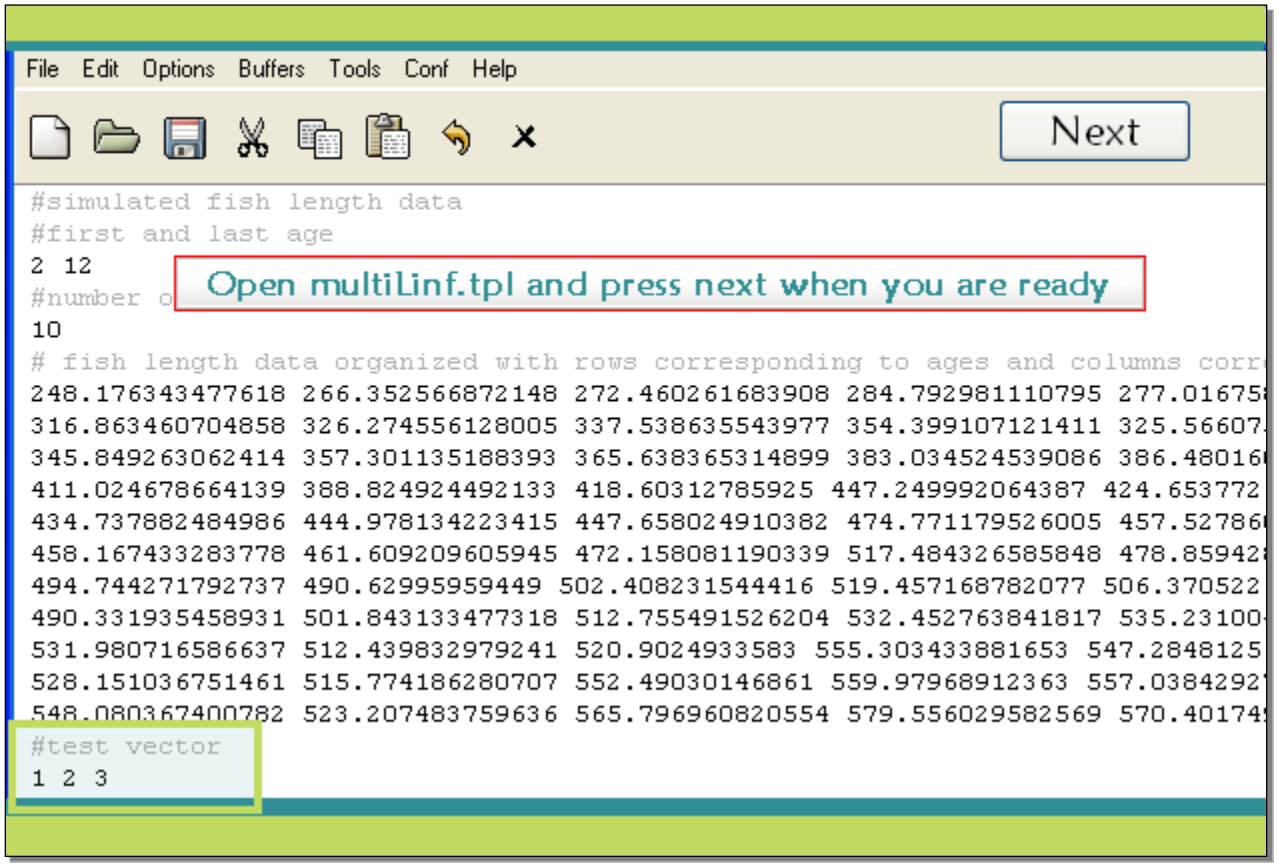


Prepare and open your files.

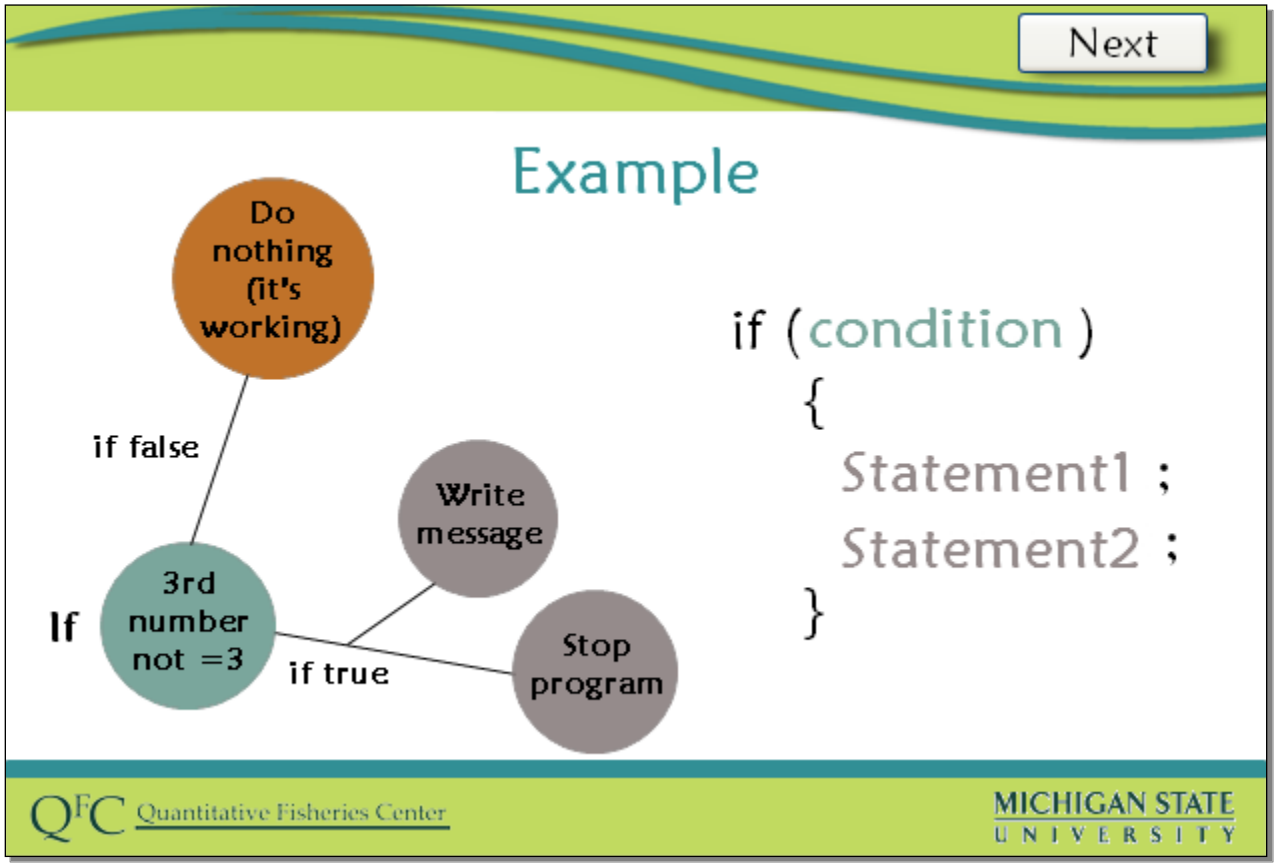
Either open multiLinf.tpl that you worked with in the previous looping videos or:

1. Create a new folder.
2. Save multiLinf.dat to the folder.
3. Save multiLinf_Loop3_Start.tpl to the folder.
4. Change the name of the tpl to multiLinf.tpl

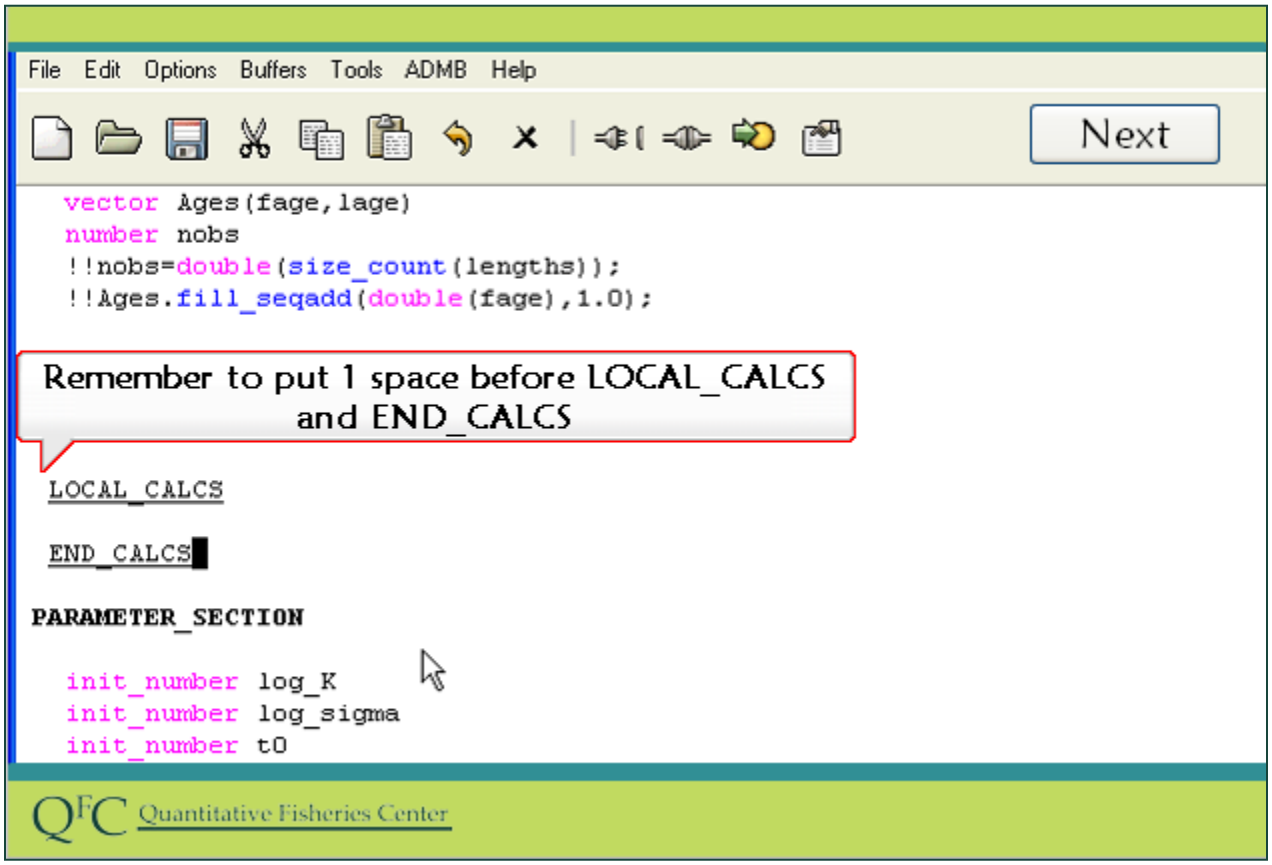
Click next when you are ready.



Now let's try our first if statement. Recall that the dat file for our multiple pond length at age example contains a test vector with three values. Open multiLinf.tpl and press next when ready.



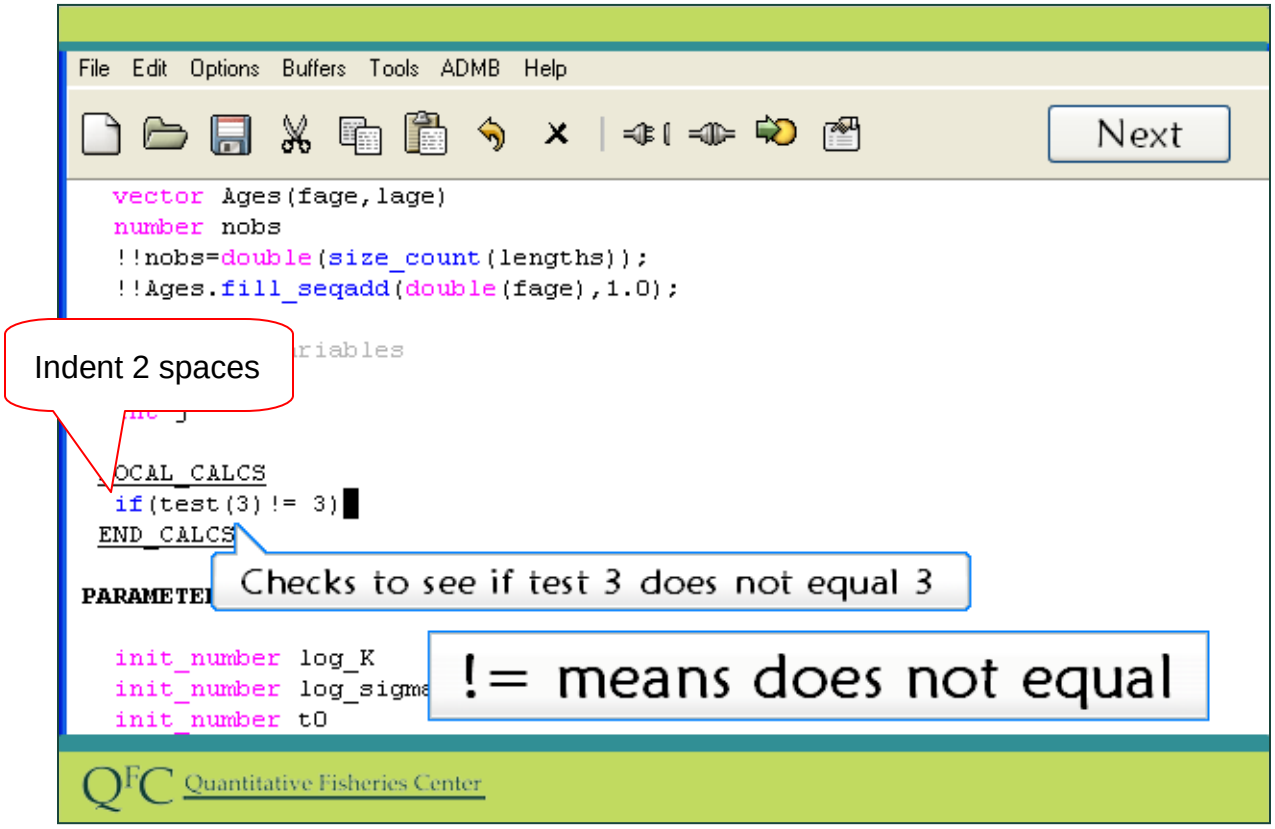
For our first if statement we will check if the third value is not equal to three, and if it isn't we will cause the program to exit. We are going to both write out an error statement and stop the program so our statement will be a compound one.



We will be adding the if statement to the bottom of the data section so it will need to go within local calcs blocks since this is C++ code.

Slide Code:

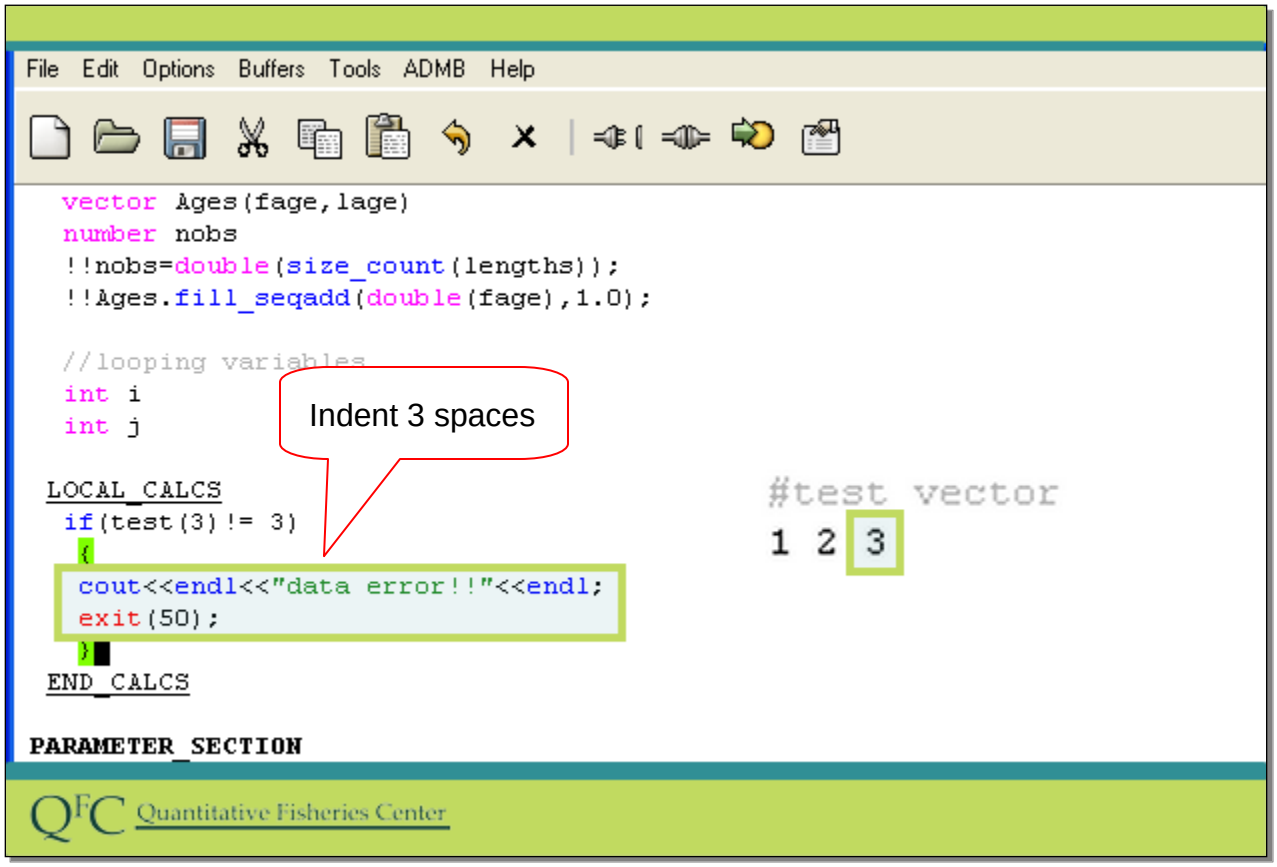
```
LOCAL_CALCS


END_CALCS
```



Following the word if we put our condition within parentheses. Here exclamation followed by an equals sign means not equal to. This should return true if test three does not equal 3 or false if it equals 3.

Slide Code:

```
LOCAL_CALCS
  if (test (3) != 3)
END_CALCS
```



Within the curly braces we put two lines. The cout statement will just write out that there has been a data error.

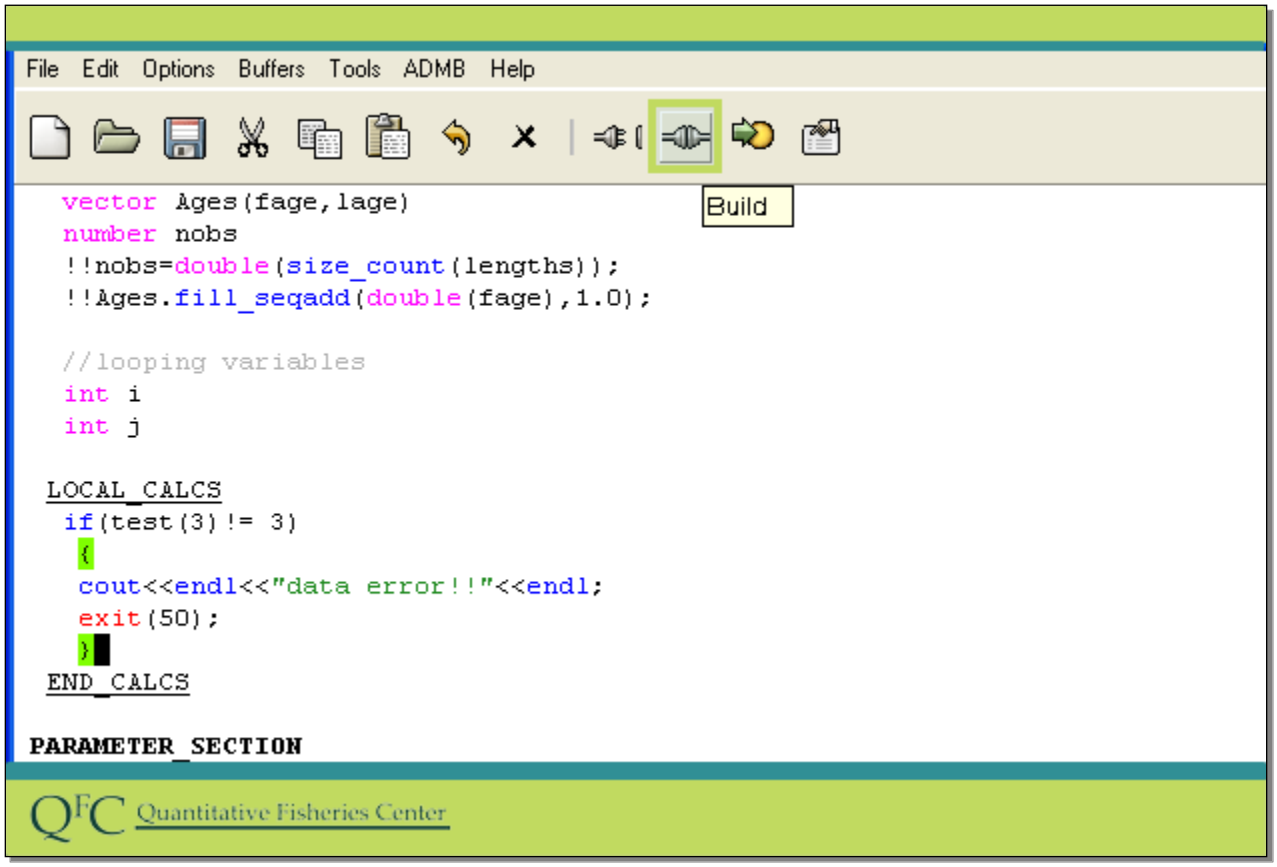
The exit statement will cause the program to quit at that point.

We close the loop.

Remember, these will only happen if our test 3 does not equal 3, which as you remember it does equal 3.

Slide Code:

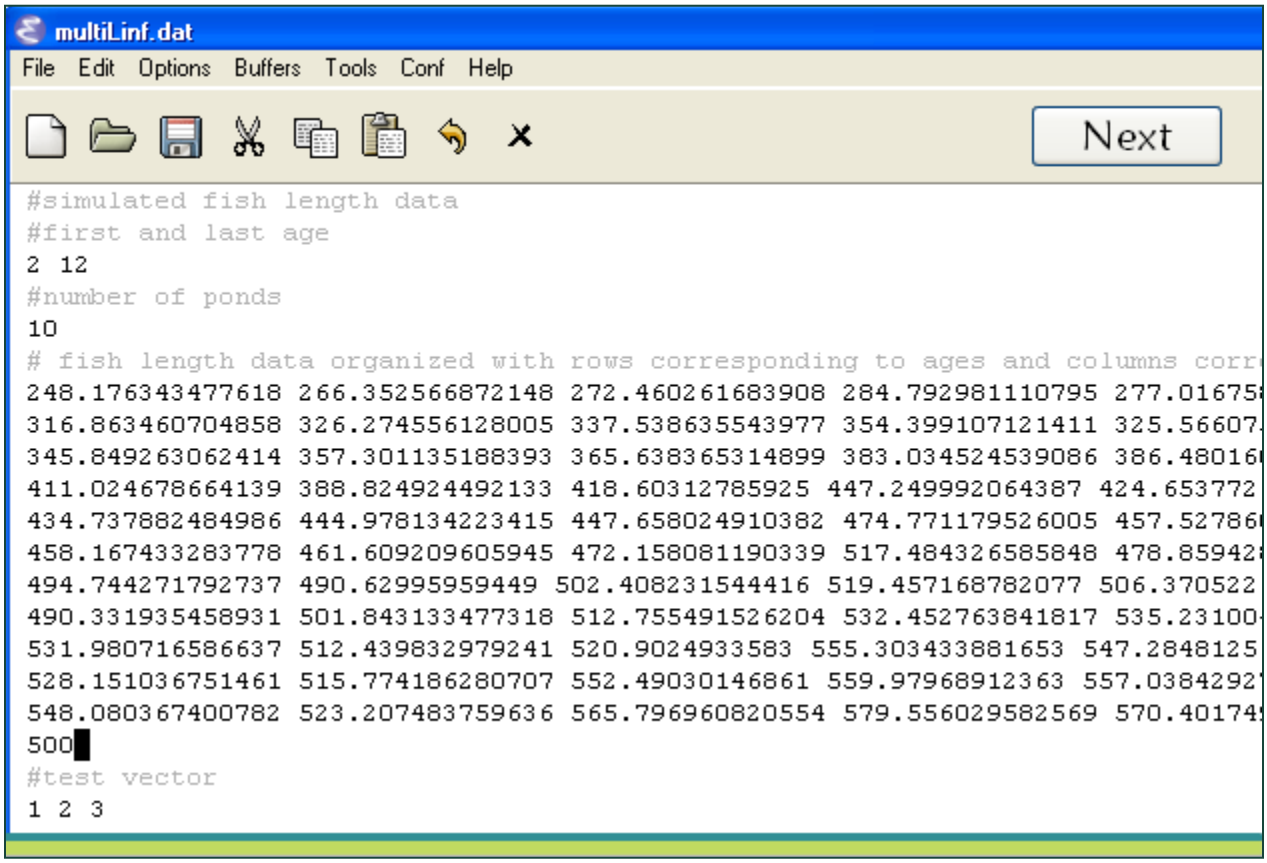
```
{
cout<<endl<< ”data error!!”<<endl;
exit (50);
}
```



Now let's build and run our program. It still runs fine and we see no change because in fact test(3) does equal 3.

Slide Action:

Build and run the program



Now let's open up the dat file and add an extra length before proceeding to the test vector. Here we add 500.

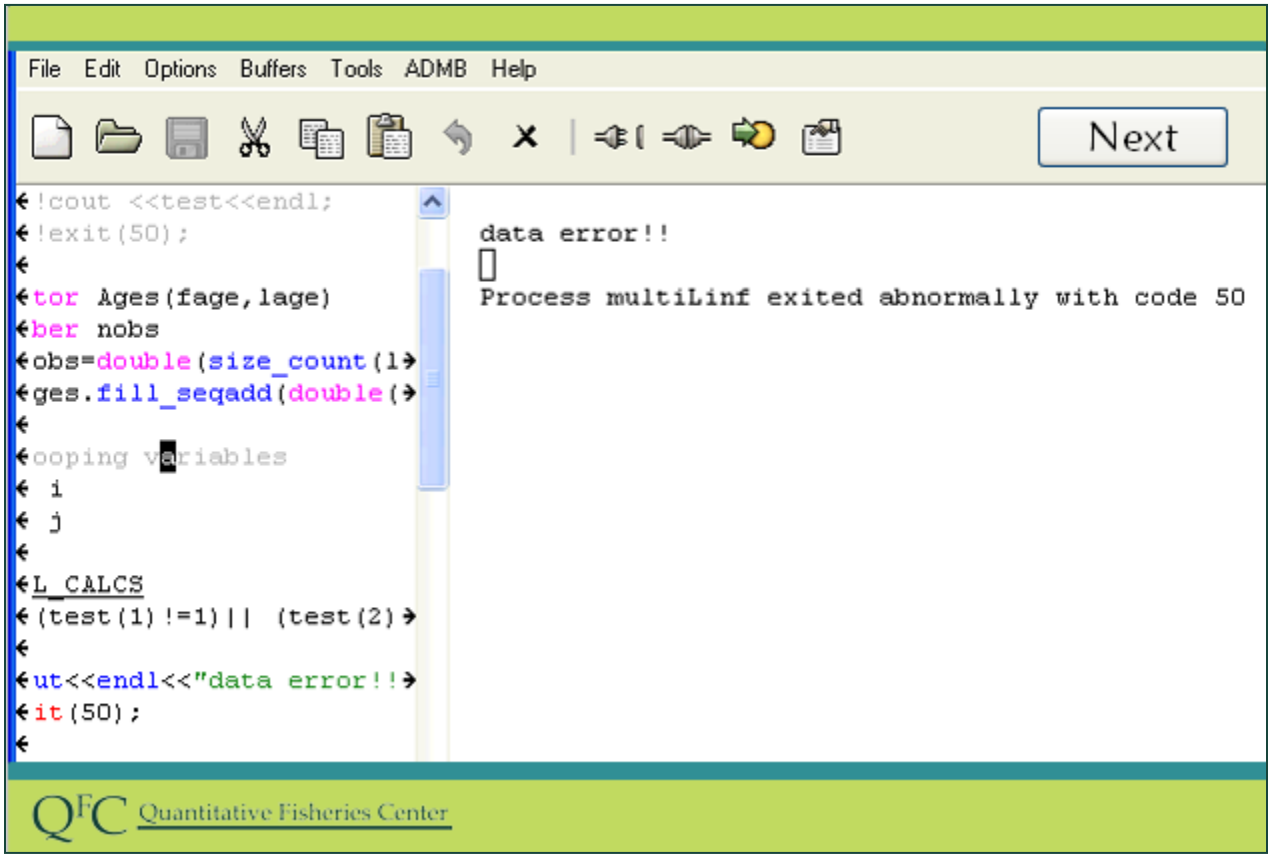
Save the changes to our dat file.

Slide Action:

Open multiLinf.dat

Type 500 in as an additional fish length.

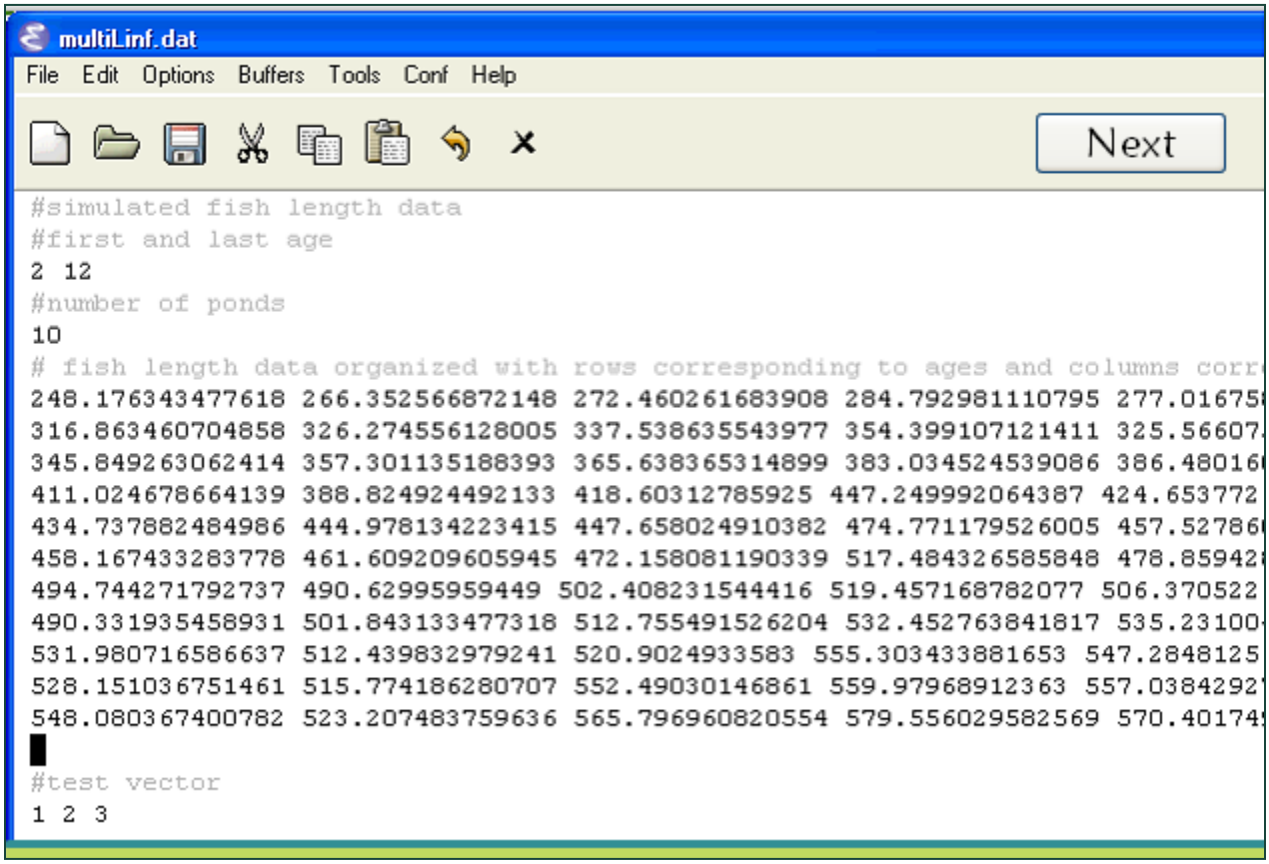
Save changes



Now if we run our program again it prints the error message "data error" that we specified in our cout statement and stops abnormally with exit code 50. It worked.

Slide Action:

Run program

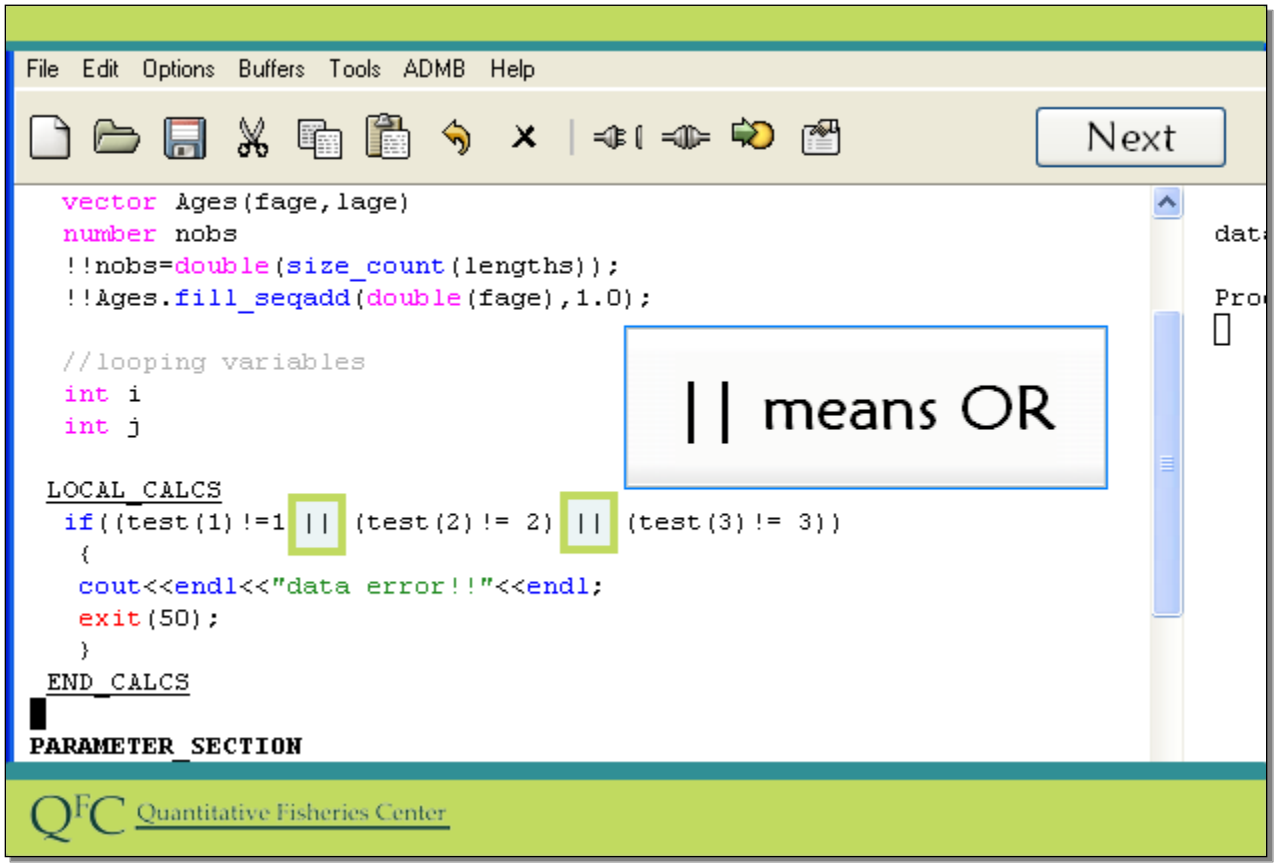


Before we forget let's go back and delete the extra observation and re-save the dat file.

Slide Action:

Remove the new length (500) from the dat file

Re-save the dat file



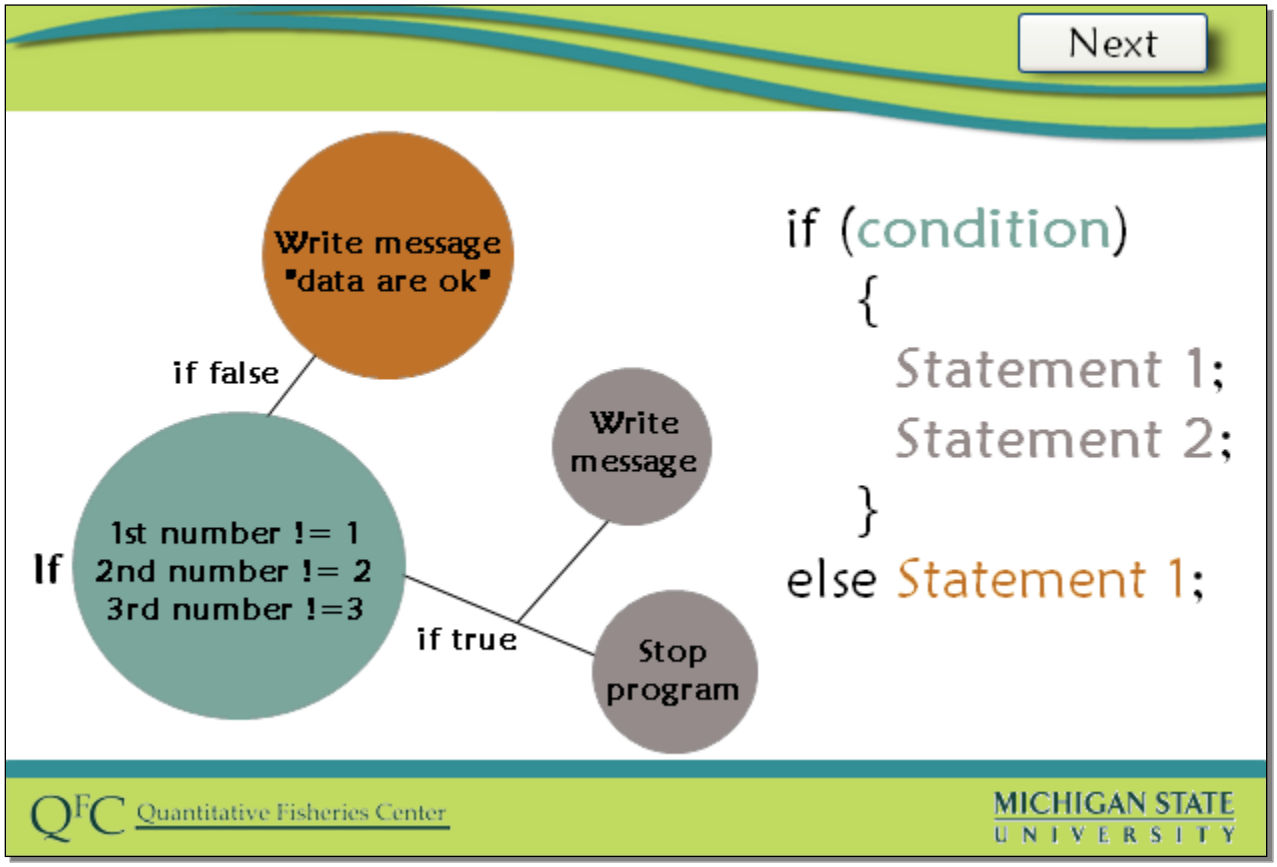
You may have noticed that our if statement is not a really rigorous test because we might get the number 3 as the value by accident if there happens to be two threes in a row and we got the wrong one. Ideally we would want to check that test one equals one, test two equals two and test three equals three and quit if any of them do not.

We can do this because our logical test can be a compound logical evaluation checking if test 1 does not equal one, test 2 does not equal 2 along with the test 3 check.

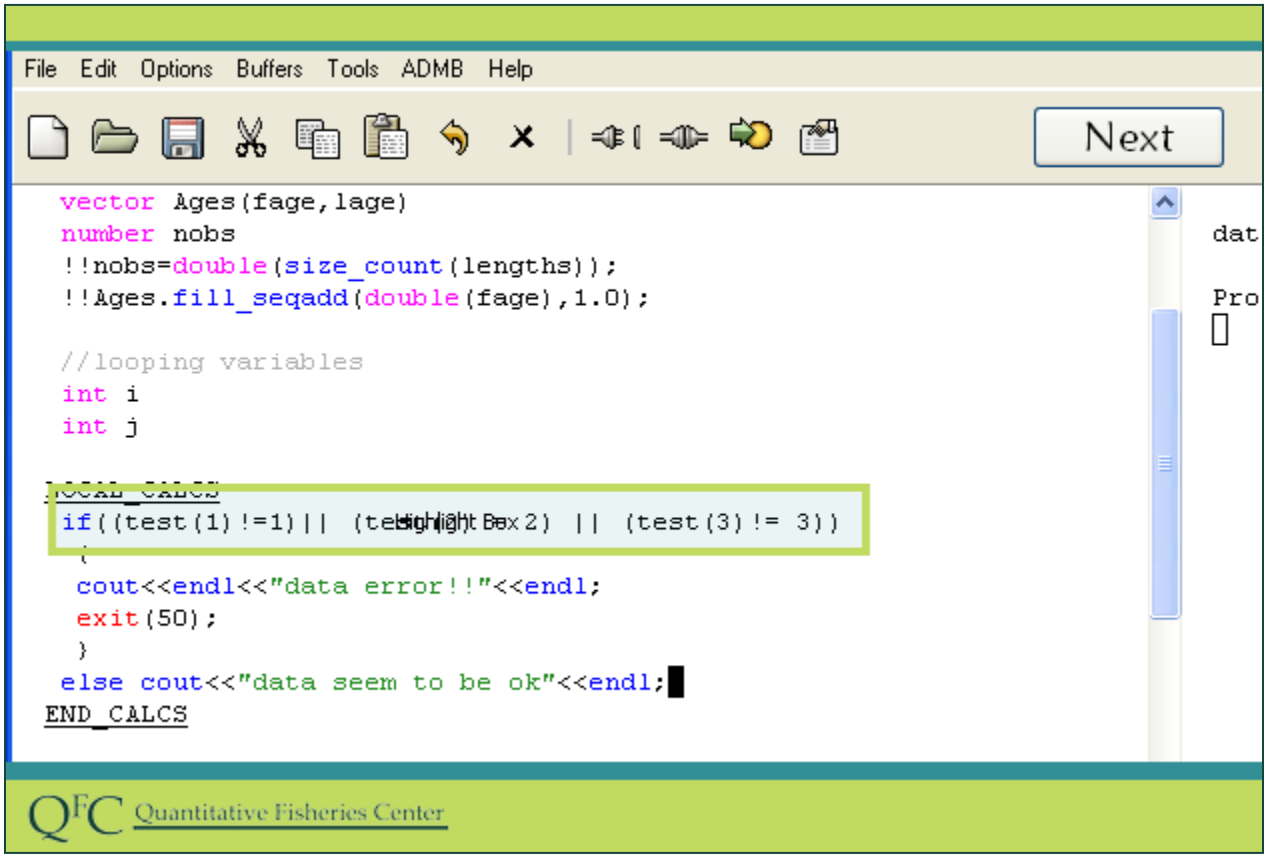
We can use two vertical bars to indicate logical OR in C++ and formally in this case we want to quit if test one is not equal to 1 OR test two is not equal to two OR test 3 is not equal to 3

Slide Code:

```
if ((test (1) != 1  ||  (test (2) != 2  ||  (test (3) != 3))
```



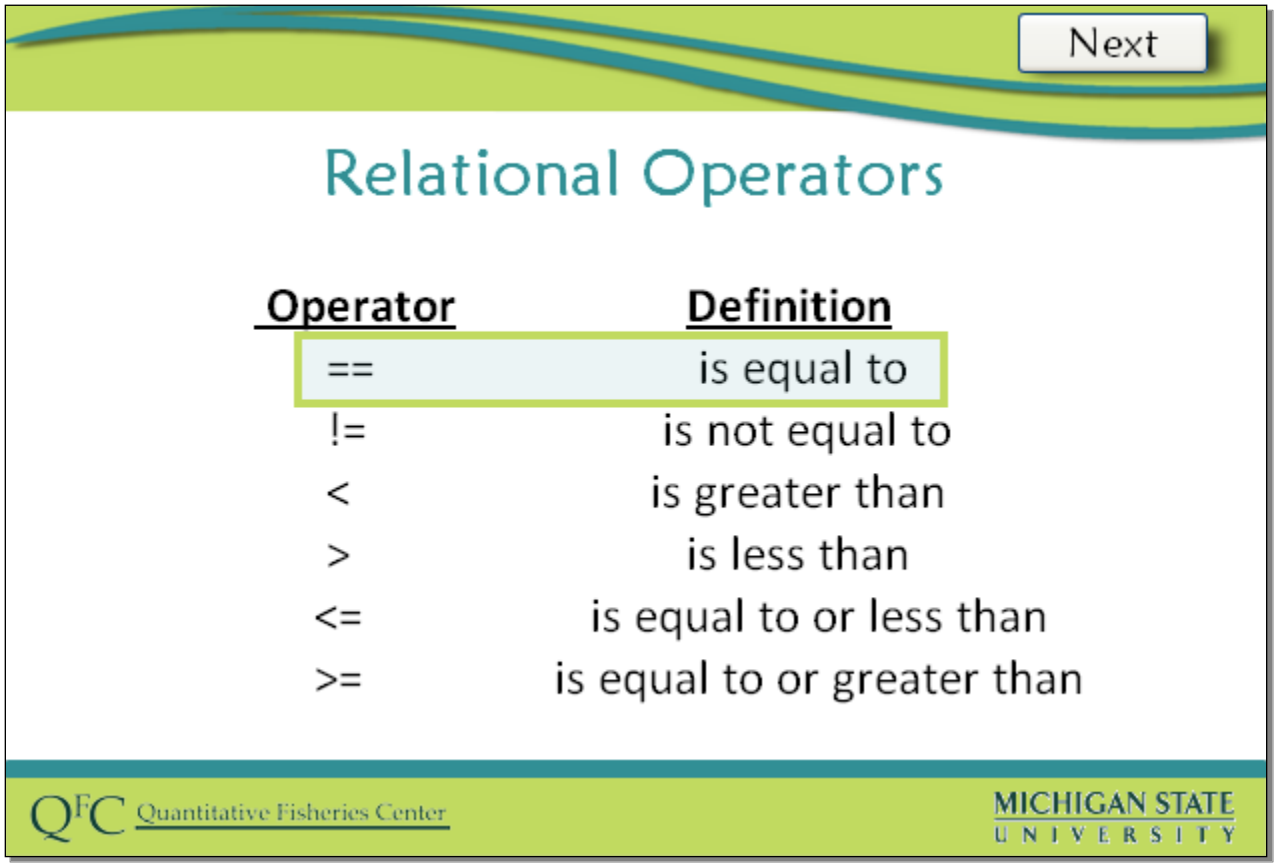
Notice that our if statement currently takes an action if the condition is true. We can write it to take an action if the condition is false. For example, we could have the program write that the data are ok. We can take this alternative action with an else statement that follows immediately after the if statement;



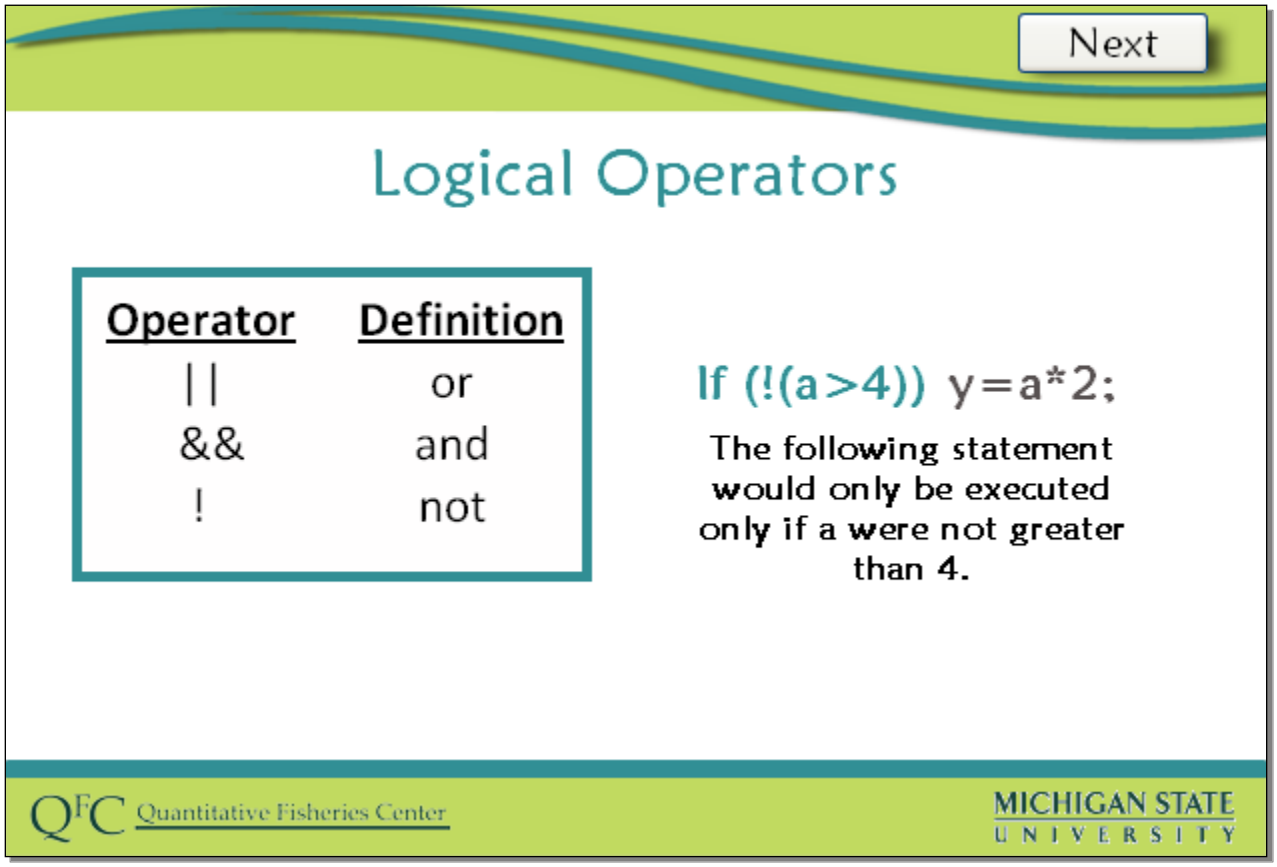
Here we add a message that the data seem to be ok if the tests turn out ok.

Slide Code:

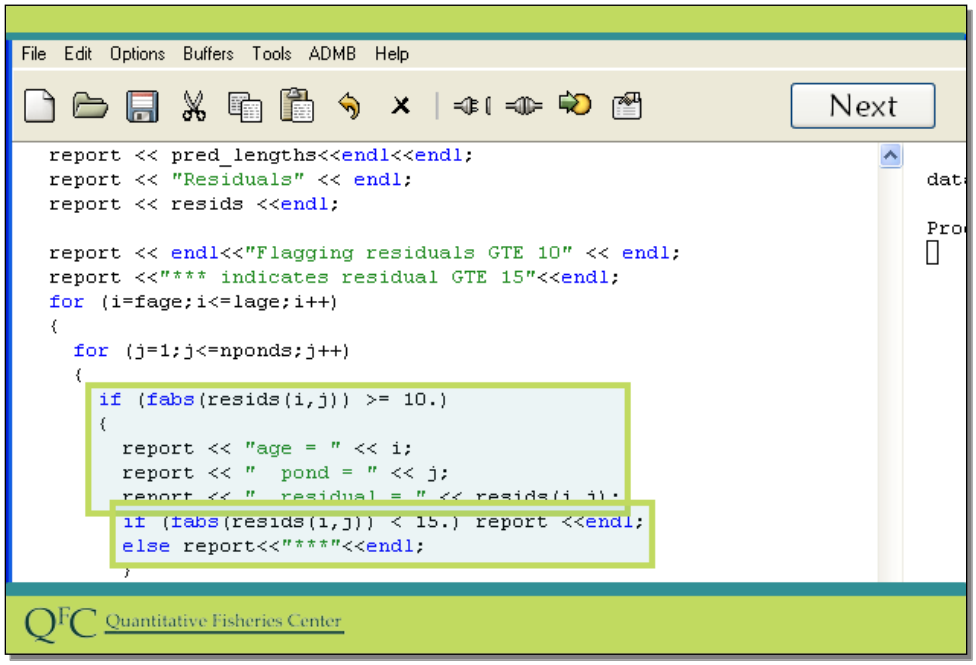
else cout<<”data seem to be ok”<<endl;



In addition to the not equal to relational operator there other relational operators commonly used in if statement conditions including equal to, less than, greater than, less than or equal to, and greater than or equal to. Notice that the relational operator for equal to is two equal sign. If you use just one, your code will set what is on the left of the equal sign to what was on the right rather than evaluating the condition, which can often cause troublesome errors if you intended to evaluate equality rather than do an assignment.



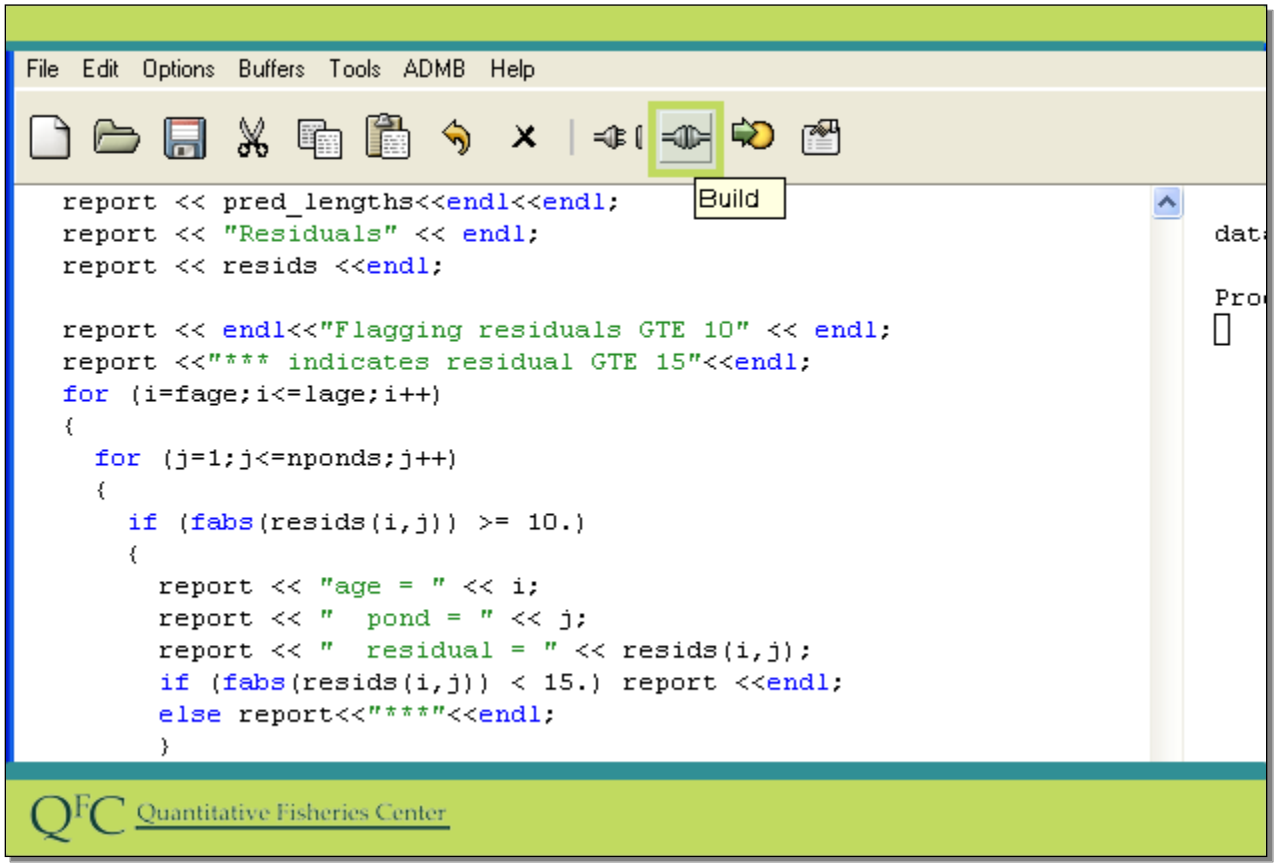
In addition to the OR logical operator, logical AND and logical NOT operators can sometimes be useful. The AND operator requires that each clause connected by the AND are true for the entire statement to be true. The NOT operator takes only one argument which immediately follows the symbol. For example the statement: If (!(a>4) would cause the following statement to be executed only if a were not greater than 4.



We conclude this section on looping and conditional statements by showing one more example in the report section of our application. You can follow along or just watch if you wish. Here we are flagging large residuals and writing them out to the report file. This example illustrates that if statements can be nested within loops and that if statements can also be nested within other if statements.

Slide Code:

```
report << endl<<"Flagging residuals GTE 10" << endl;
report <<"*** indicates residual GTE 15"<<endl;
for (i=fage;i<=lage;i++)
{
 for (j=1;j<=nponds;j++)
 {
  if (fabs(resids(i,j)) >= 10.)
  {
   report << "age = " << i;
   report << "  pond = " << j;
   report << "  residual = " << resids(i,j);
   if (fabs(resids(i,j)) < 15.) report <<endl;
   else report<<"***"<<endl;
   }
  }
 }
```

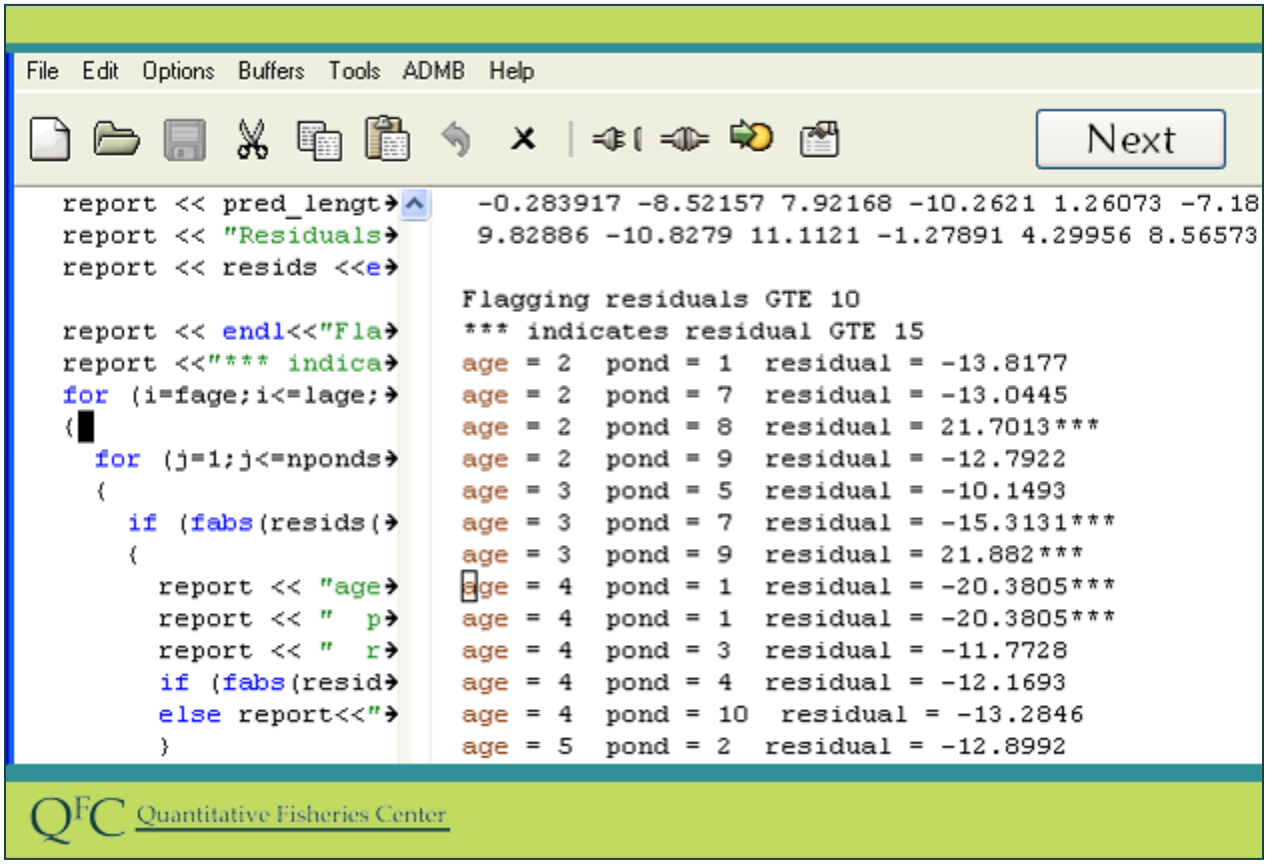


Here we are looping over ages and within ages over ponds. For each pond age combination we check if a residual is larger than or equal to 10. If it is we write it out and then check to see if its less than 15. If it's less than 15 we print a line feed and move on. If its not less than 15 we print three asterisk and a line feed and then move on.

Let's build and run the model with this addition to see what we get.

Slide Action:

Build and run the model if you added the additional code



The results are a list of the residuals of 10 or larger with those less than 15 flagged with three asterisks.

Obviously there is lots more that can be done with loops and conditional statements. Hopefully this video has provided you the basic ideas.