



Next

Looping and Conditional Execution

Part 1 - Looping in the INITIALIZATION_SECTION

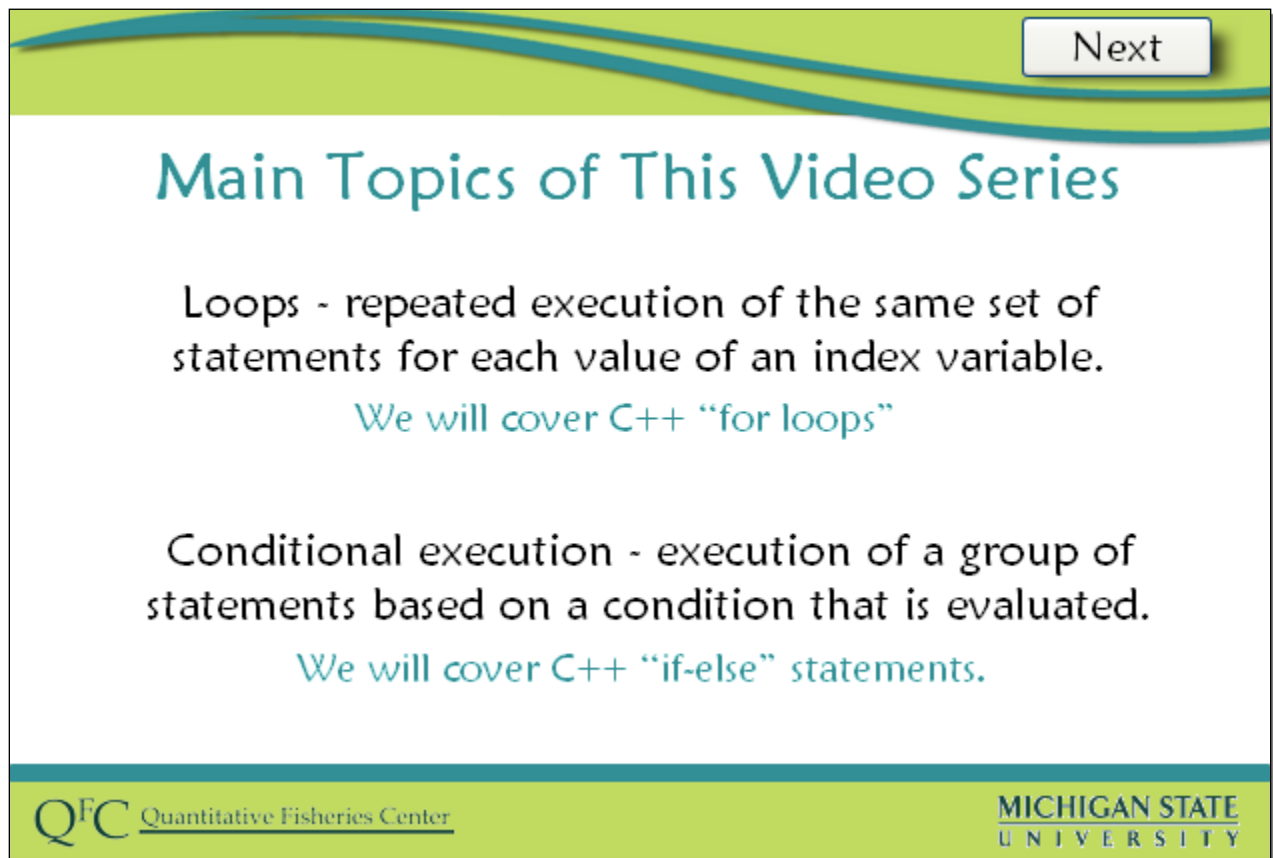


This video was created using ADMB-IDE release 4.5.0-1 (July 15, 2011)
You may notice some minor differences if using a different version.

 Quantitative Fisheries Center

MICHIGAN STATE
UNIVERSITY

For the very simple model we have worked with so far we have not had to deal with how to execute tasks repeatedly in a loop or how to execute tasks depending upon a condition.



Next

Main Topics of This Video Series

Loops - repeated execution of the same set of statements for each value of an index variable.
We will cover C++ “for loops”

Conditional execution - execution of a group of statements based on a condition that is evaluated.
We will cover C++ “if-else” statements.

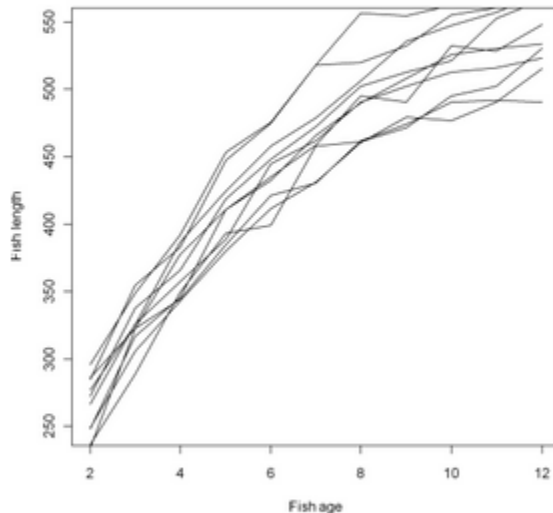
QFC Quantitative Fisheries Center

MICHIGAN STATE
UNIVERSITY

Loops could be useful if we repeatedly want to do the same calculation for each pond in a collection of ponds or each year in a sequence of years. These are often referred to as iterative calculations. There are various ways to do iterative calculations in C++ but here we will only cover what are known as “for loops.” Often in admb loops can be avoided because functions can be applied to an entire vector at once as we have seen in previous examples. This however is not always possible. Conditional statements are executed depending on how a condition is evaluated. For example we might want to stop our program if we read in bad data, or we might want to print out in our report section residuals if they are larger than some fixed number. While a range of different ways to execute statements conditionally are available here we will only cover “if-else” statements. Looping and conditional statements are just standard C++ computer programming so you can use pretty much any standard book on the programming language to go beyond what we cover in this video. The part 1 and part 2 videos cover loops, the part 3 video covers conditional execution. This video, part 1, introduces a loop in its simplest form, with part 2 providing additional information on looping.

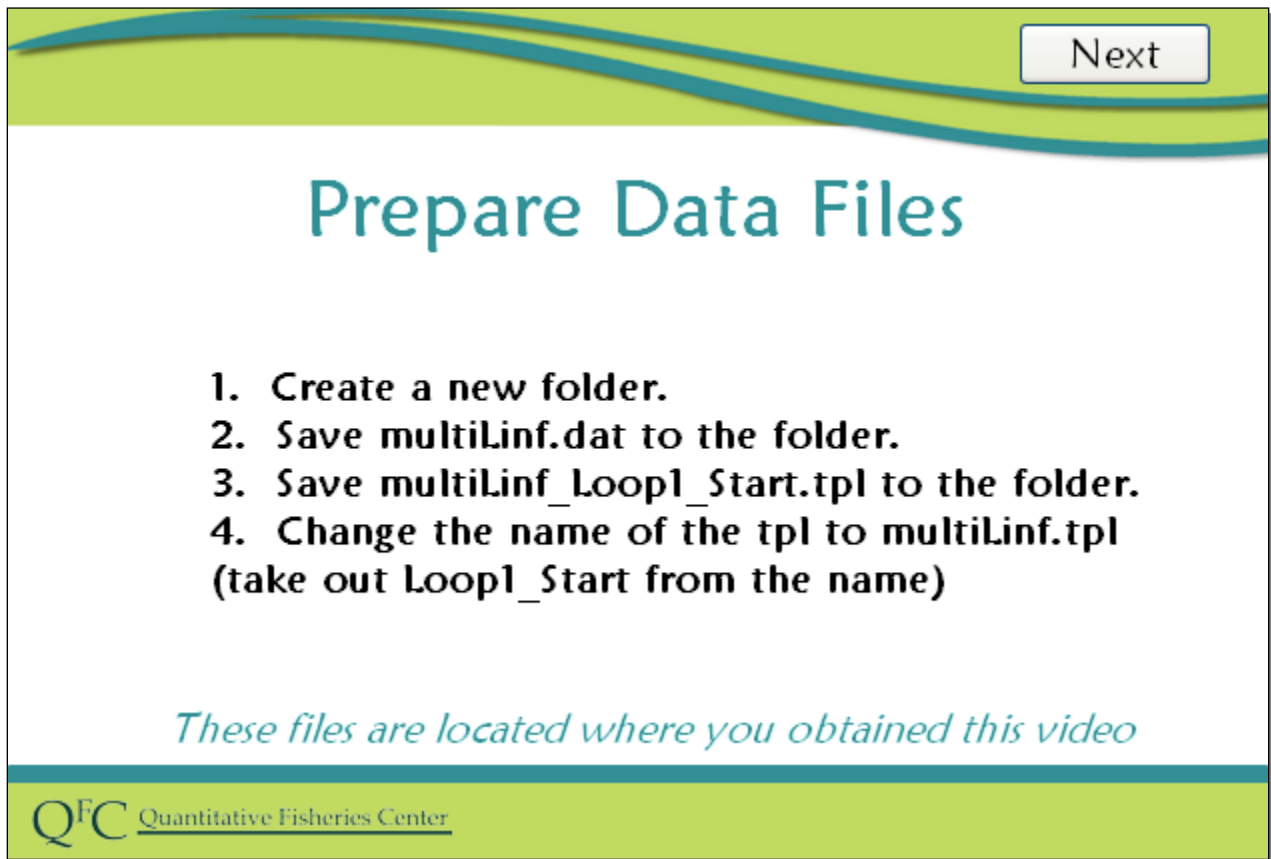
[Next](#)

The Model



Length at age data from 10
ponds

For illustrating loops we will stay with the length at age model we have used in previous videos, but now we will be applying it in a more complex fashion to a more complex data set. For this video we will be using an artificial data set simulated in a computer. The simulated data set consists of length at age data from 10 ponds. For this simple example we produced a perhaps unrealistically balanced data set with exactly one observed length for each age, ages 2 through 12 for each of 10 ponds. The data were generated assuming that asymptotic length varied among the ponds but that the other growth parameters were the same.



Next

Prepare Data Files

1. Create a new folder.
2. Save multiLinf.dat to the folder.
3. Save multiLinf_Loop1_Start.tpl to the folder.
4. Change the name of the tpl to multiLinf.tpl (take out Loop1_Start from the name)

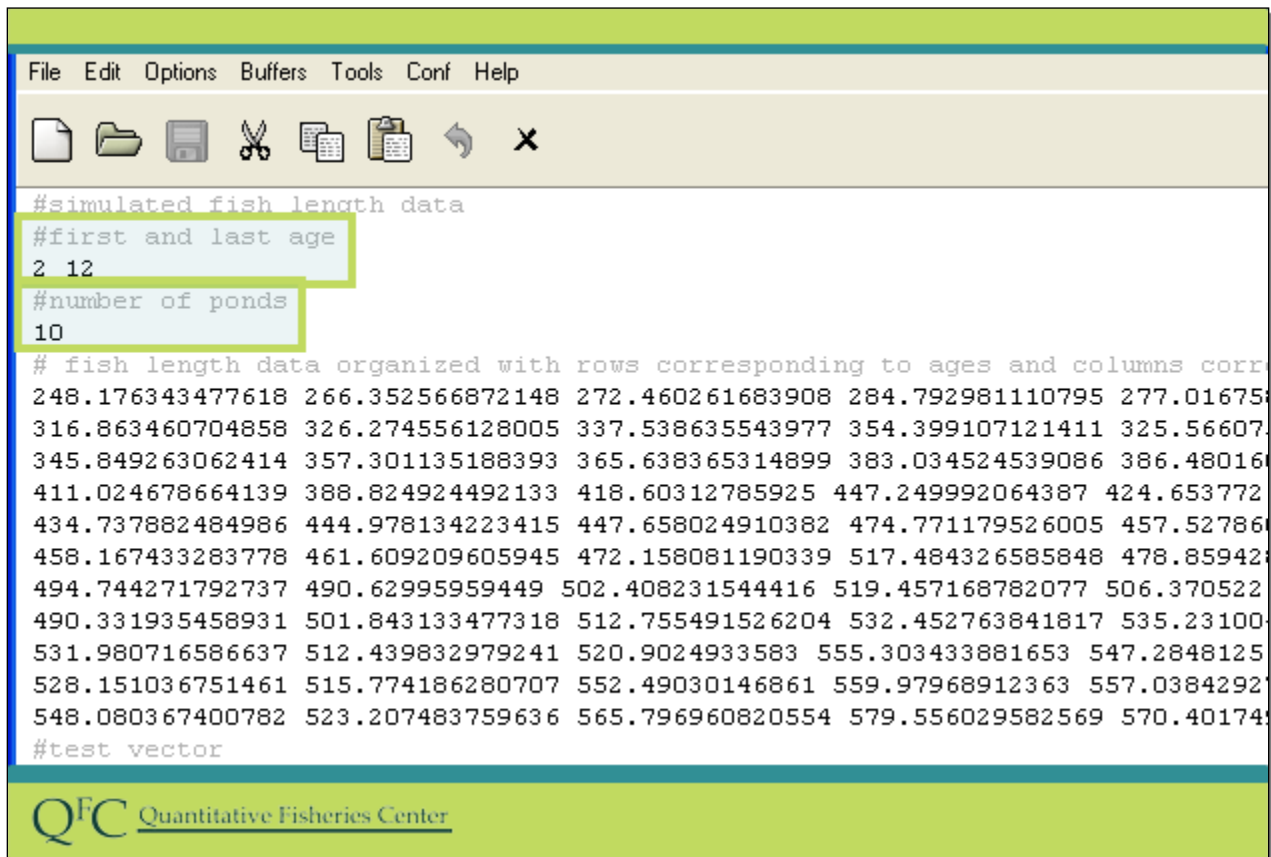
These files are located where you obtained this video

QFC Quantitative Fisheries Center

Prepare and open your files. Click next when you are ready.

Slide Action:

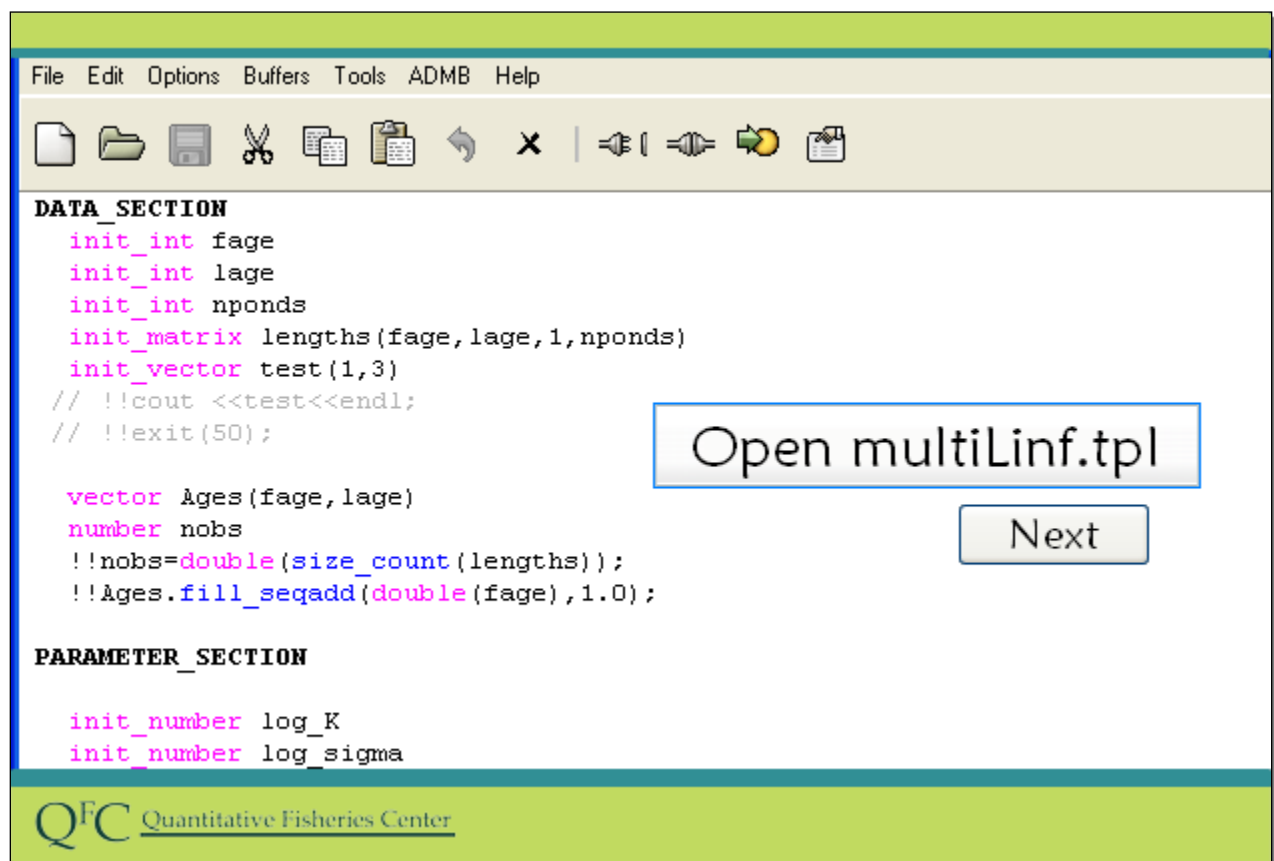
1. Create a new folder.
2. Save multiLinf.dat to the folder.
3. Save multiLinf_Loop1_Start.tpl to the folder.
4. Change the name of the tpl to multiLinf.tpl (take out Loop1_Start from the name)
5. Open multiLinf.tpl



```
File Edit Options Buffers Tools Conf Help
#simulated fish length data
#first and last age
2 12
#number of ponds
10
# fish length data organized with rows corresponding to ages and columns corresponding to ponds
248.176343477618 266.352566872148 272.460261683908 284.792981110795 277.016751
316.863460704858 326.274556128005 337.538635543977 354.399107121411 325.566071
345.849263062414 357.301135188393 365.638365314899 383.034524539086 386.480161
411.024678664139 388.824924492133 418.60312785925 447.249992064387 424.653772
434.737882484986 444.978134223415 447.658024910382 474.771179526005 457.527861
458.167433283778 461.609209605945 472.158081190339 517.484326585848 478.859421
494.744271792737 490.62995959449 502.408231544416 519.457168782077 506.370522
490.331935458931 501.843133477318 512.755491526204 532.452763841817 535.231001
531.980716586637 512.439832979241 520.9024933583 555.303433881653 547.2848125
528.151036751461 515.774186280707 552.49030146861 559.97968912363 557.03842921
548.080367400782 523.207483759636 565.796960820554 579.556029582569 570.401741
#test vector
```

QFC Quantitative Fisheries Center

Here is our dat file. These data are arranged in a dat file with one row for each age and columns corresponding to ponds. Our dat file includes the first and last ages and the number of ponds. So, the first thing our tpl needs to do is to read the data in.



First open the multiLinf.tpl file. Click next when you are ready to continue.

Slide Action:

Open multiLinf.tpl if you haven't already.

```

File Edit Options Buffers Tools ADMB Help

[Icons] [Next]

DATA_SECTION
  init_int fage
  init_int lage
  init_int nponds
  init_matrix lengths(fage, lage, 1, nponds)
  init_vector test(1,3)
// !!cout <<test<<endl;
// !!exit(50);

vector Ages(fage, lage)
number nobs
!!nobs=double(size_count(lengths))
!!Ages.fill_seqadd(double(fage))

PARAMETER_SECTION

init_number log_K
init_number log_sigma
  
```

First age Last age Number of ponds

```

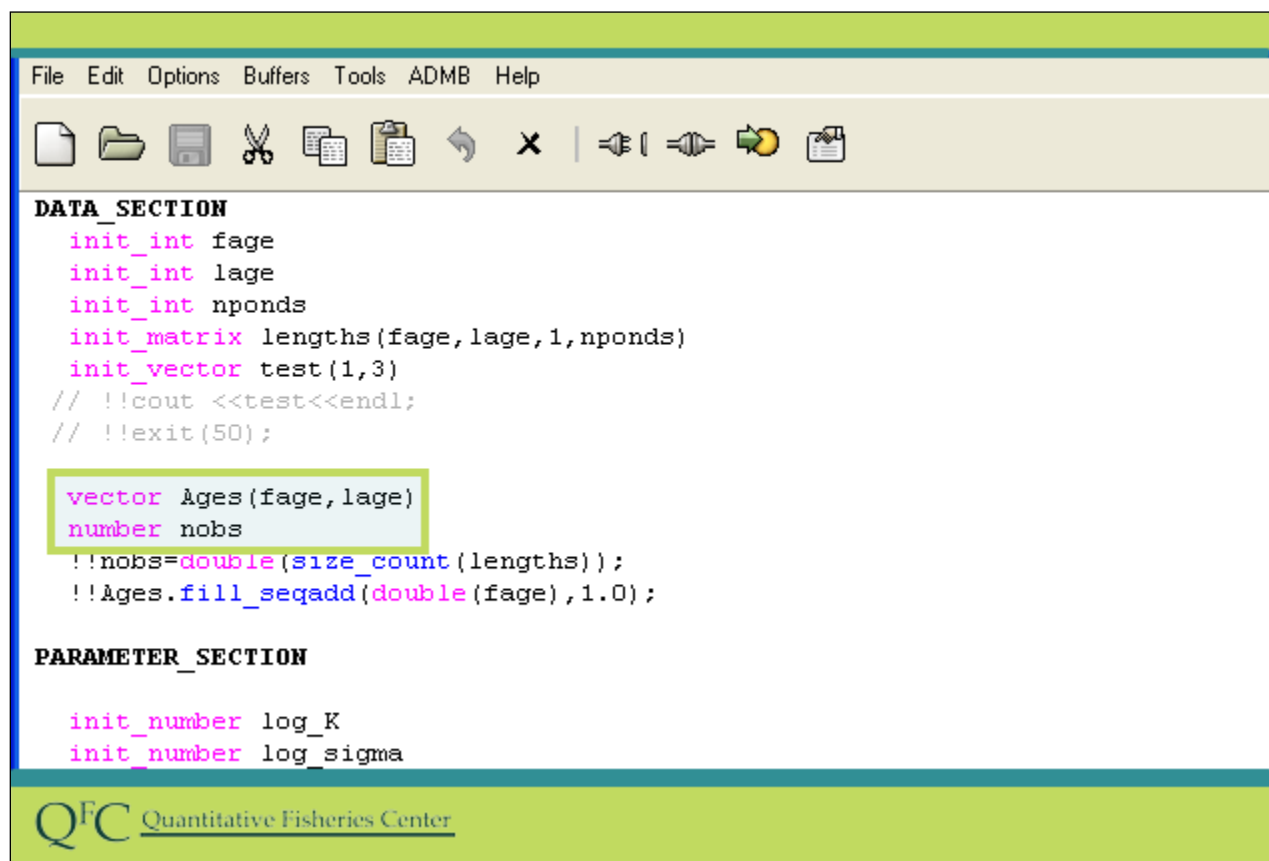
multifit.dat
File Edit Options Buffers Tools Cont Help

[Icons]

#simulated fish length data
#first and last age
2 12
#number of ponds
10
# fish length data organized with rows corresponding to ages and columns corresponding to ponds
248.176343477618 266.352566872148 272.460261683908 284.792981110795 277.016751
316.863460704858 326.274556128005 337.538635543977 354.399107121411 325.56607
345.049263062414 357.301135108193 365.638365314899 383.034524539086 386.48016
411.024678664139 388.824924492133 418.60312785925 447.249992064387 424.653772
434.737882484986 444.978134223415 447.658024910382 474.771179526005 457.52786
458.167433283778 461.609209605945 472.158081190339 517.484326585848 478.05942
494.744271792737 490.62995959449 502.408231544416 519.457168782077 506.370522
490.331935458931 501.843133477318 512.755491526204 532.452763841817 535.23100
531.980716586637 512.439832979241 520.9024933583 555.303433881653 547.2848125
528.151036751461 515.774186280707 552.49030146861 559.97968912363 557.0384292
548.080367400782 523.207483759636 565.796960820554 579.556029582569 570.40174
  
```

Q^{FC} Quantitative Fisheries Center

Recall the data in our dat file are arranged with one row for each age and columns corresponding to ponds. So the first thing our tpl needs to do is to read the data in. We read the length data in as a matrix with rows for ages and columns for ponds.



```
File Edit Options Buffers Tools ADMB Help

[Icons: File Explorer, Save, Cut, Copy, Paste, Undo, Redo, Print, Run, etc.]

DATA_SECTION
  init_int fage
  init_int lage
  init_int nponds
  init_matrix lengths(fage, lage, 1, nponds)
  init_vector test(1, 3)
  // !!cout <<test<<endl;
  // !!exit(50);

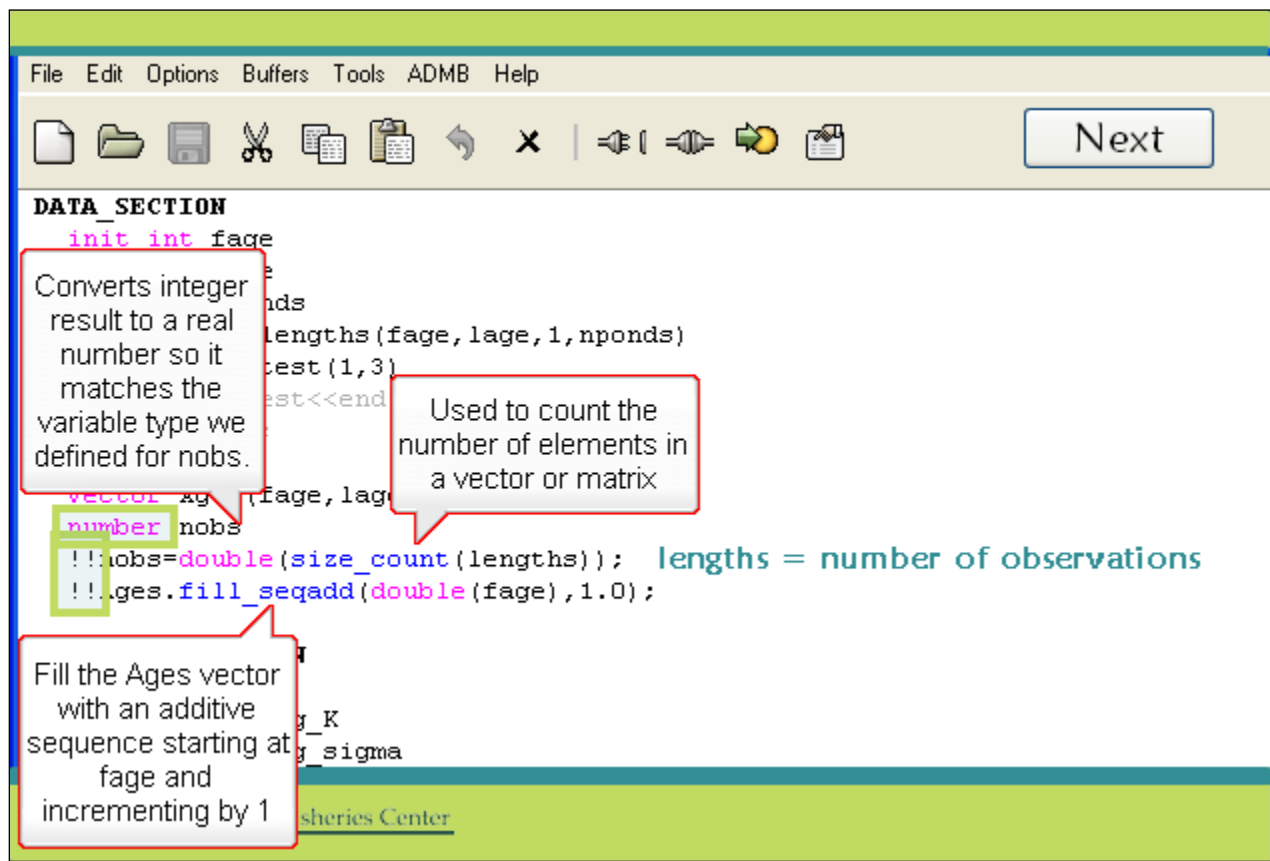
  vector Ages(fage, lage)
  number nobs
  !!nobs=double(size_count(lengths));
  !!Ages.fill_seqadd(double(fage), 1.0);

PARAMETER_SECTION

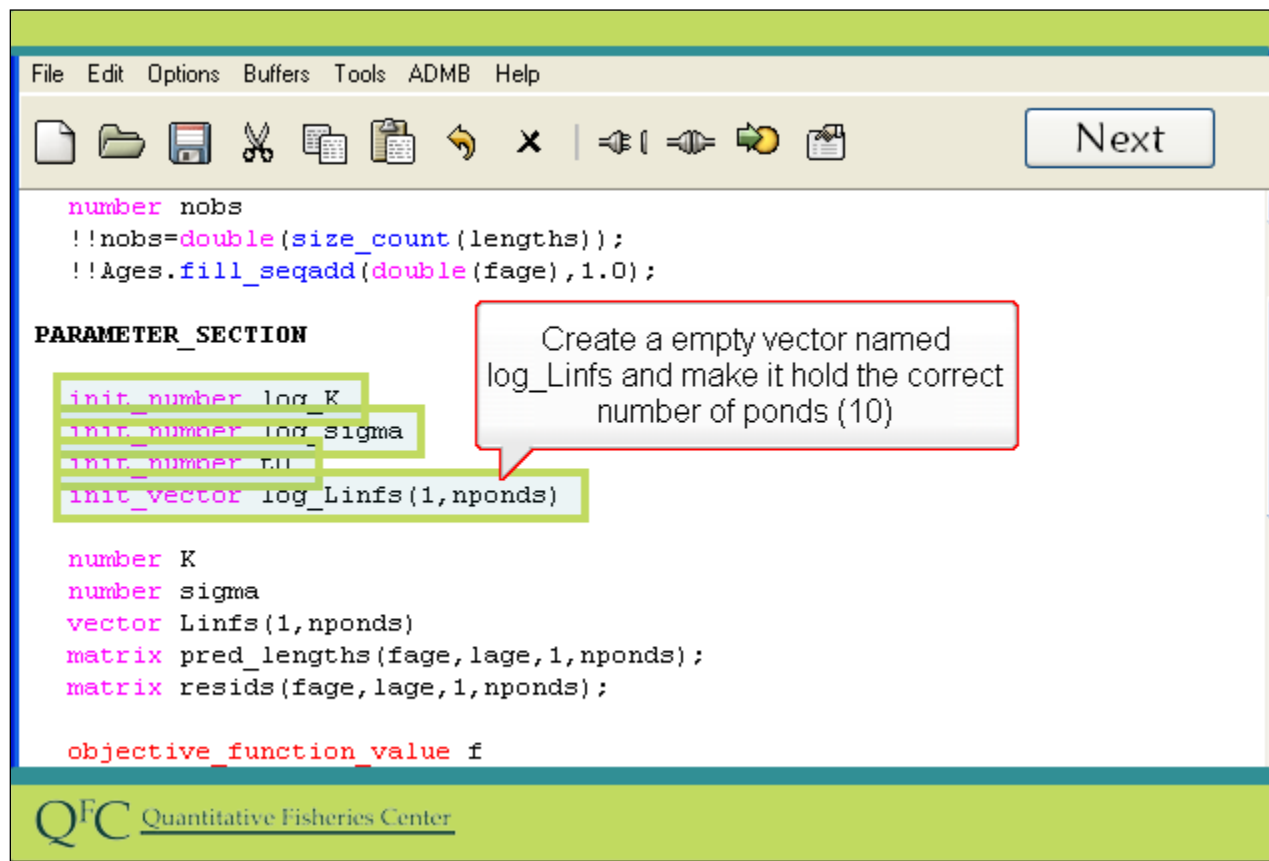
  init_number log_K
  init_number log_sigma

QFC Quantitative Fisheries Center
```

For this example we will need a vector of ages from the first to last age and the total number of observations but did not read them in directly from the data file. We define them here.



Remember that in the tpl file we start actual c plus plus code with double exclamation marks. We can use the function `size_count` to count how many elements are in a matrix or vector. In this case the number of elements in the matrix `lengths` is the number of observations. The function `double` converts the integer result to a real number so it matches the variable type we defined for `nobs`. Here we could have defined `nobs` as an integer as we did in previous videos and things would have worked ok without having to use `double`. However, in this program `nobs` is only used in the calculation of the objective function, which produces a real number result. In such cases it is best to define `nobs` using `number` rather than `int`. Plus we got a chance to show you the function `double`. We did not read in the sequence of ages we need from the dat file so we use `fill_seqadd` to fill the Ages vector with an additive sequence starting at the `fage` and incrementing by a value of 1. This function was covered in a previous video and is also explained in the `admb` user manual.



```
File Edit Options Buffers Tools ADMB Help

number nobs
!!nobs=double(size_count(lengths));
!!Ages.fill_seqadd(double(fage), 1.0);

PARAMETER_SECTION

init_number log_K
init_number log_sigma
init_number t0
init_vector log_Linfs(1, nponds)

number K
number sigma
vector Linfs(1, nponds)
matrix pred_lengths(fage, lage, 1, nponds);
matrix resids(fage, lage, 1, nponds);

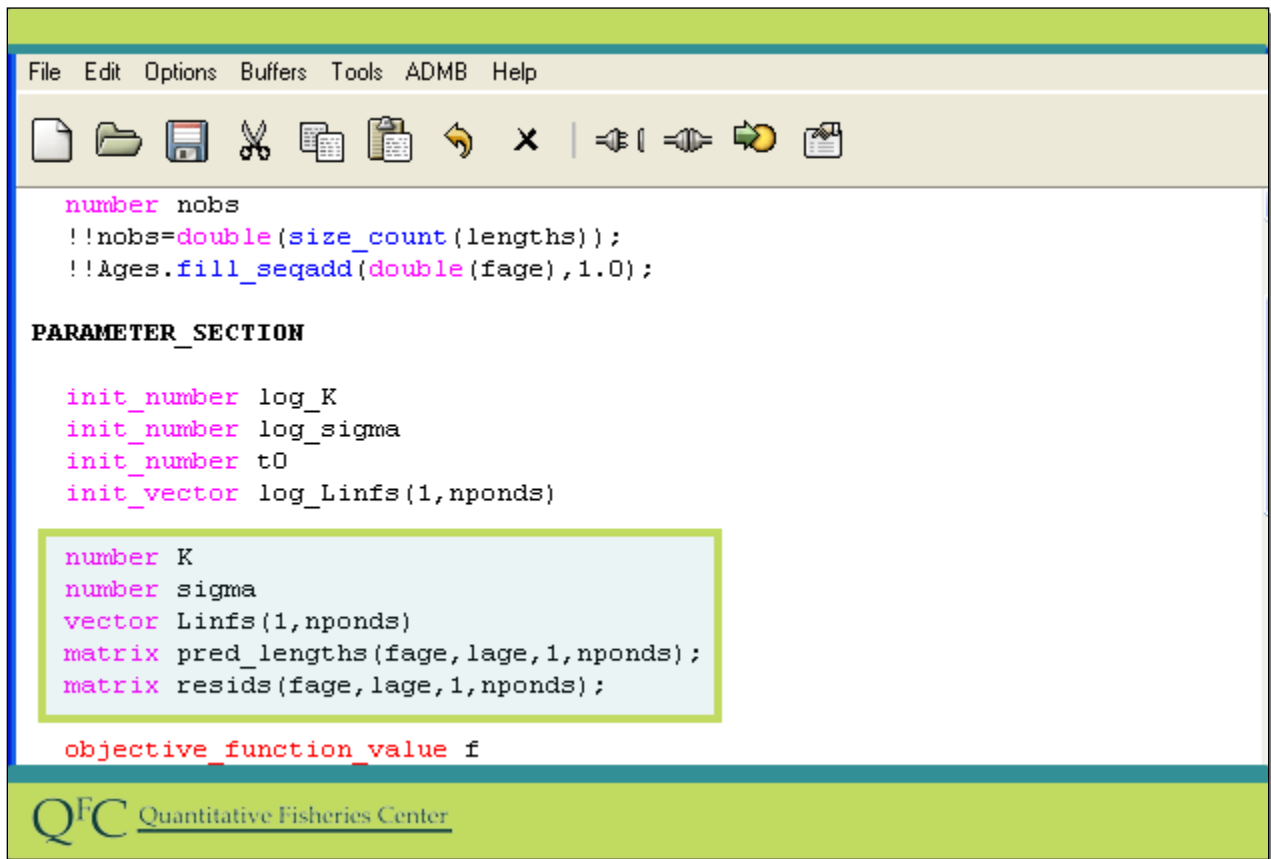
objective_function_value f
```

Next

Create a empty vector named log_Linfs and make it hold the correct number of ponds (10)

QFC Quantitative Fisheries Center

Our parameter section defines the log of the Brody growth coefficient K , the log of the residual standard deviation σ , and t_0 just as we did in our simple growth model example. The change for this example is we now define a parameter vector of log asymptotic lengths, with one element for each pond.



```
File Edit Options Buffers Tools ADMB Help

number nobs
!!nobs=double(size_count(lengths));
!!Ages.fill_seqadd(double(fage),1.0);

PARAMETER_SECTION

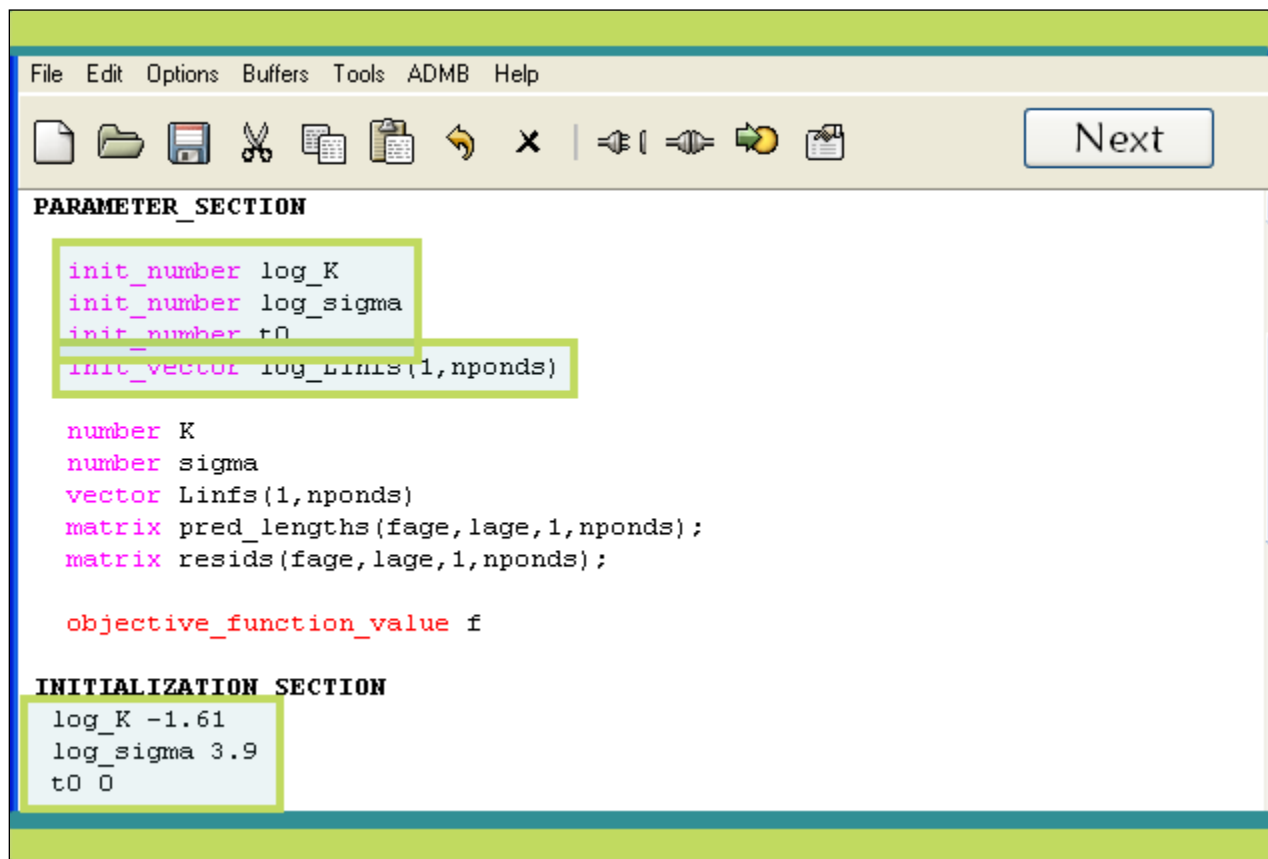
init_number log_K
init_number log_sigma
init_number t0
init_vector log_Linfs(1,nponds)

number K
number sigma
vector Linfs(1,nponds)
matrix pred_lengths(fage,lage,1,nponds);
matrix resids(fage,lage,1,nponds);

objective_function_value f
```

QFC Quantitative Fisheries Center

Our parameter section also defines the calculated quantities that depend upon the parameters – here the back transforms of the log-scale parameters, the predicted lengths, and the residuals.



```
File Edit Options Buffers Tools ADMB Help

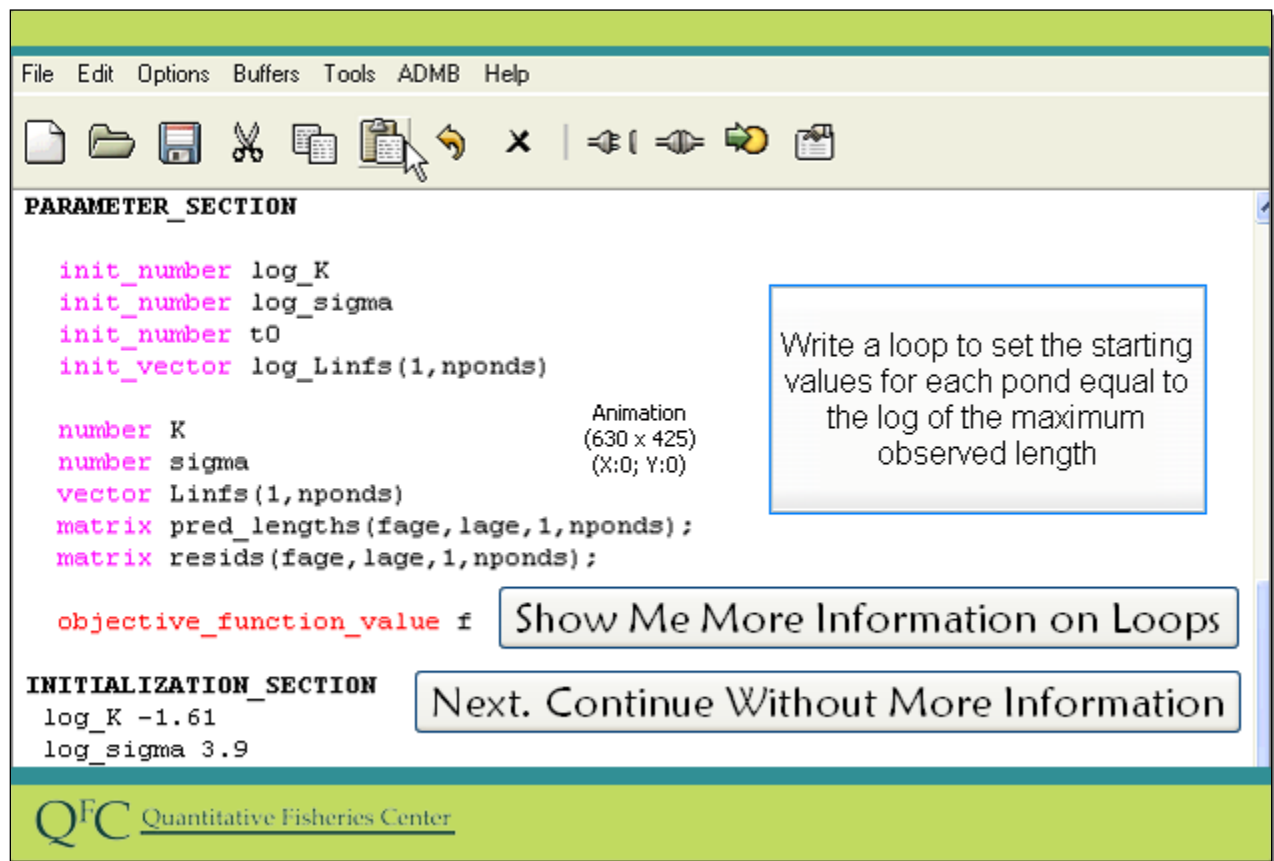
init_number log_K
init_number log_sigma
init_number t0
init_vector log_Linfs(1,nponds)

number K
number sigma
vector Linfs(1,nponds)
matrix pred_lengths(fage,lage,1,nponds);
matrix resids(fage,lage,1,nponds);

objective_function_value f

INITIALIZATION_SECTION
log_K -1.61
log_sigma 3.9
t0 0
```


Notice that in the Initialization section we have defined starting values for log_K, log_sigma, and t naught but not for log_Linfs.



We will use a loop in a preliminary calcs section to set the starting value of these parameters for each pond equal to the log of the maximum observed length.

Next

What is a Loop?



Repeat Procedure

minimator
(186 x 186)
(X:69; Y:127)

2

3

4

5

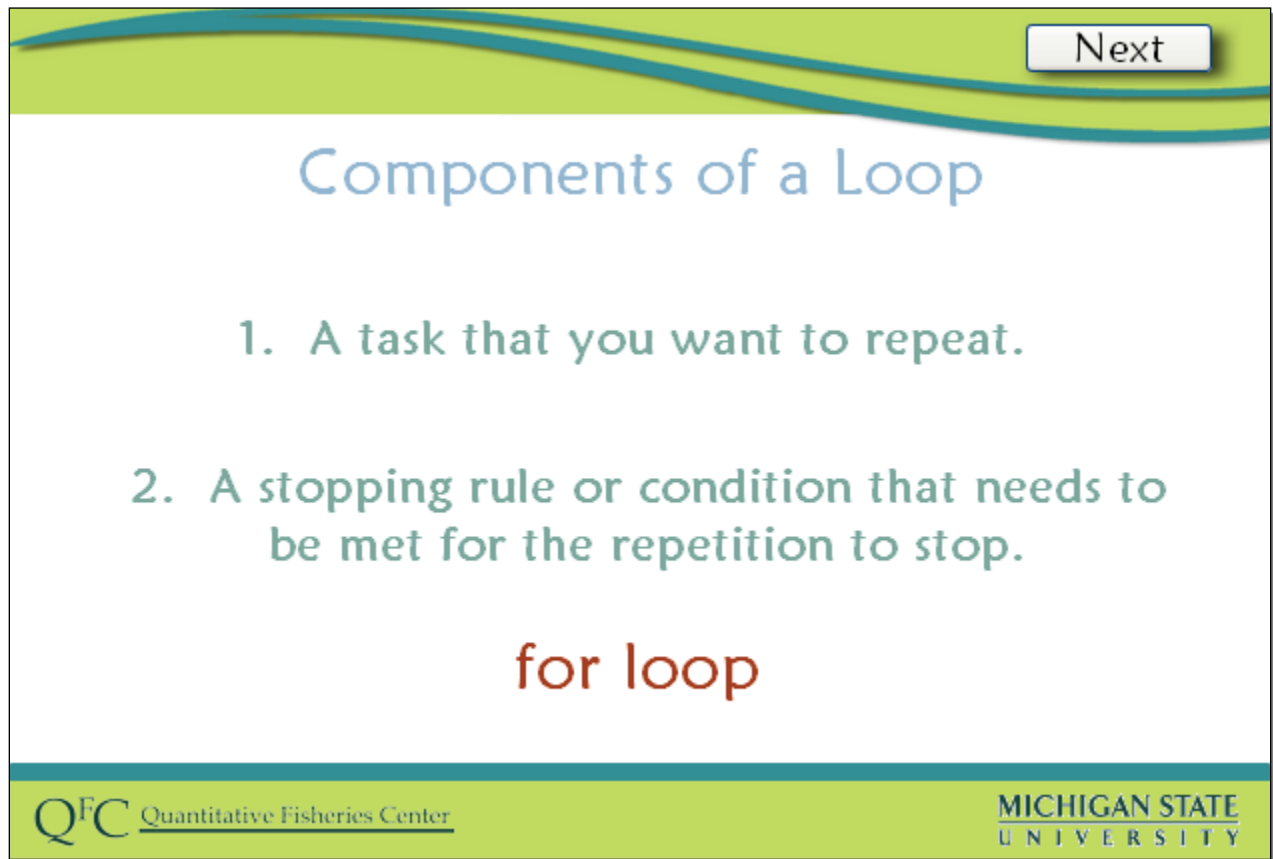
6

You know the task: you want to square each of the 5 numbers

QFC Quantitative Fisheries Center

MICHIGAN STATE UNIVERSITY

During data analysis and programming in general, you will often come across problems that require you to repeat a procedure many times. Performing a repetitive task can be time-consuming and increase the likelihood of making errors. A loop (or “looping”) is a way to complete repetitive tasks in a quick, efficient manner. As a simple example, assume that you are interested in squaring a sequence of five numbers (i.e., raising each number to the second power). You can efficiently complete this task using a “for loop” to repeat the same operation over and over again.



Next

Components of a Loop

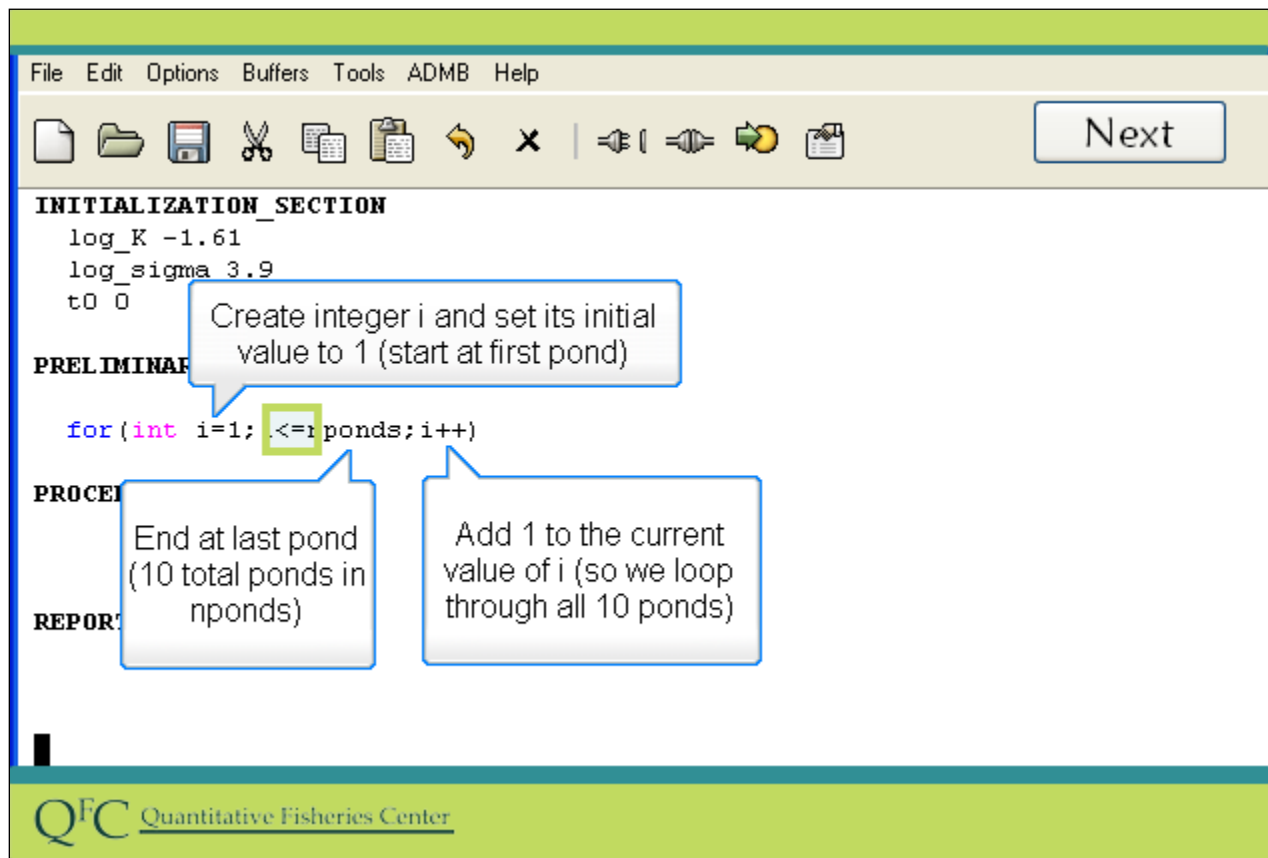
1. A task that you want to repeat.
2. A stopping rule or condition that needs to be met for the repetition to stop.

for loop

QFC Quantitative Fisheries Center

MICHIGAN STATE
UNIVERSITY

Now back to looping. Based on the examination of the simple example of squaring numbers, a loop can more generally be thought of as composed of two main components. 1. A task that you want to repeat. In this example, you want to raise a sequence of numbers to the second power. 2. A stopping rule or condition that needs to be met for the repetition to stop. In this example you will stop looping and squaring numbers after the last number is squared. Thus, a loop in its simplest form is a task to repeat combined with a rule or condition that causes the task to stop being executed. We will focus most of our attention on what is called a “for loop”.

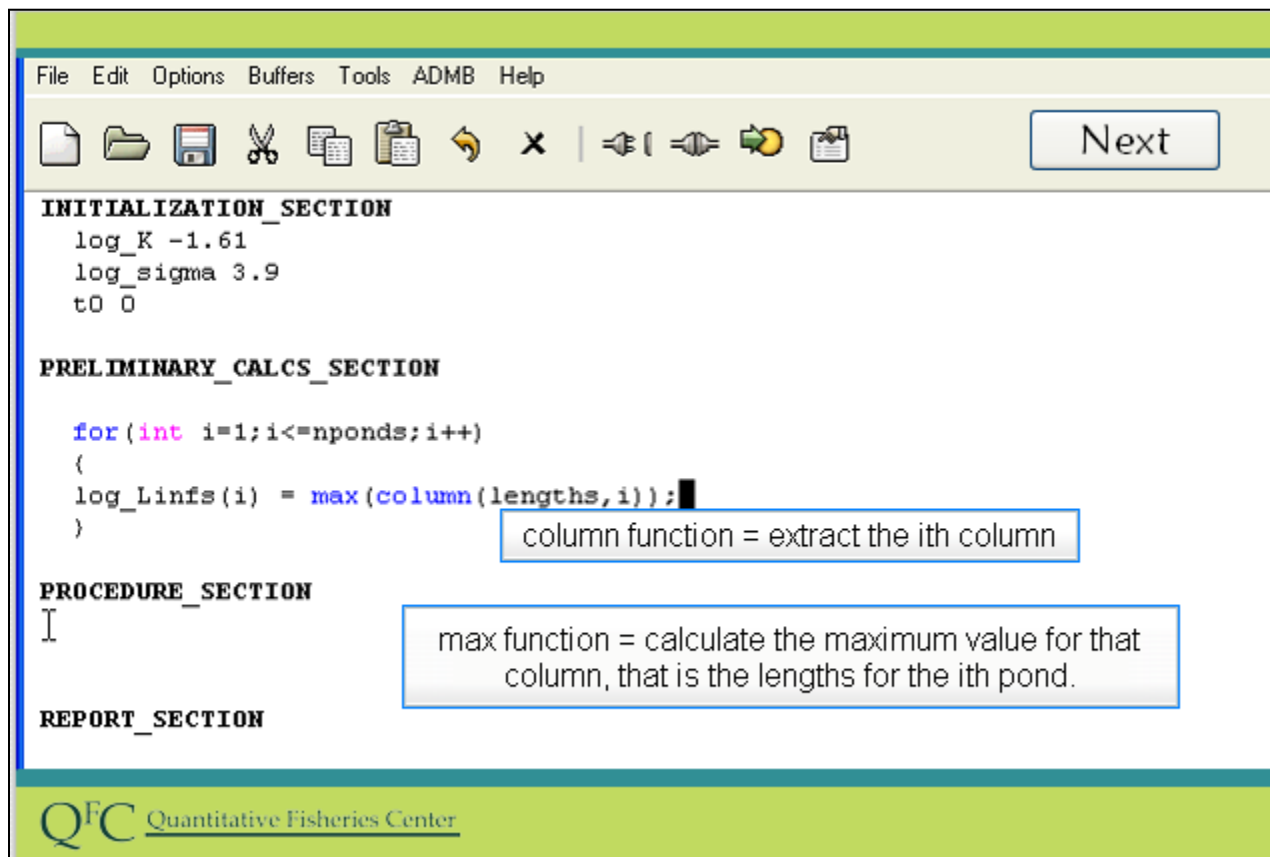


We start with the statement “for” followed by information within parentheses describing what value “i” starts at, what value “i” ends at, and how “i” changes each time through the loop:

Our loop will repeat calculations for an integer variable “i” representing which pond we are considering for values of i going from 1 for the first pond to nponds, that is 10, for the last. The term “int i=1” says create the integer i and set its initial value to 1. We follow that by a semicolon and then specify the ending condition. Here our iterative calculations will repeat as long as “i” is less than or equal to the number of ponds. We then have another semicolon and i++. i++ means add 1 to the current value of i. It is short hand for i=i+1, which would work too but is rarely seen. This short hand is in fact the source of the name for the C++ programming language.

Slide Code:

```
for(int i=1; i<=nponds;i++)
```

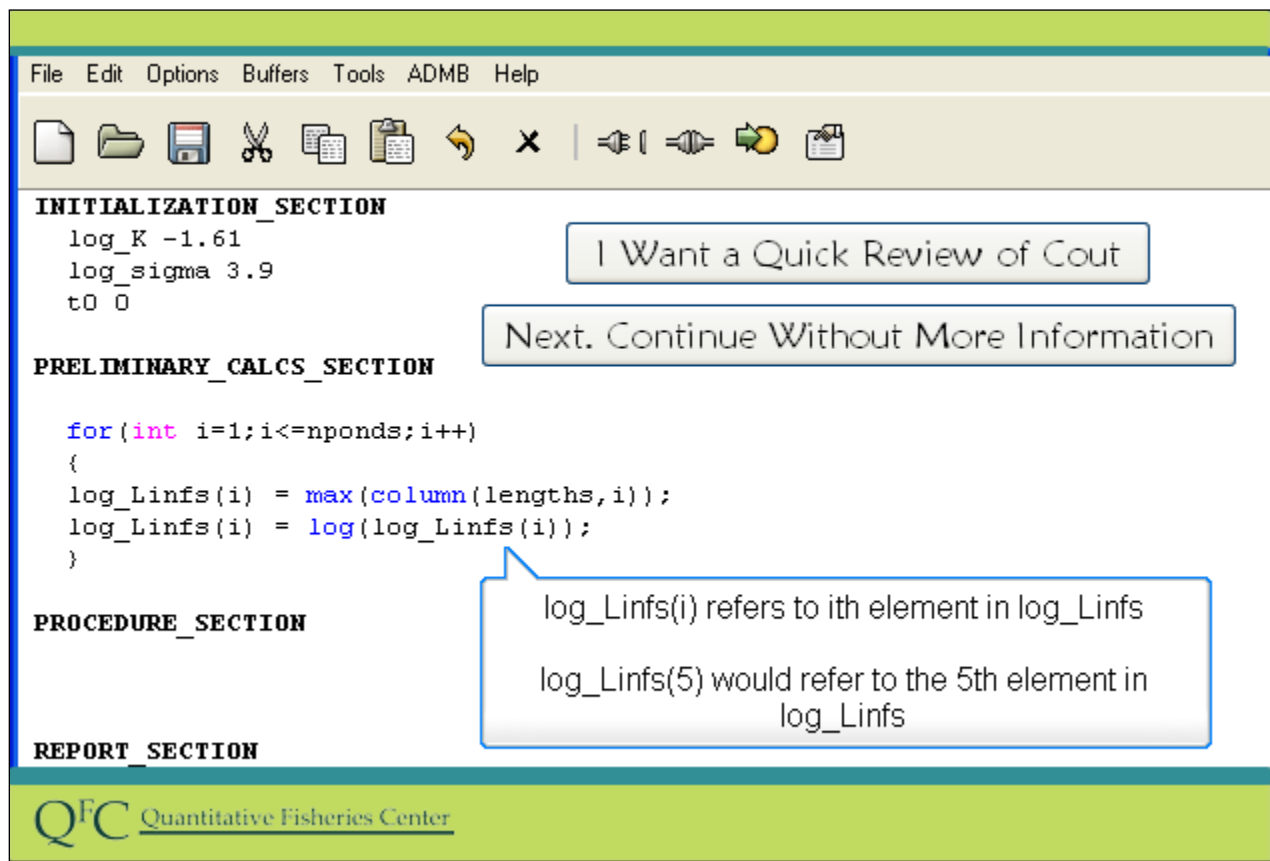



Next we add opening and closing curly braces. All the statements we put within the curly braces will be repeated for each value of i . In this case that means for i equals 1, 2, 3 and so on up to 10.

Now we add the actual lines we want calculated for each value of i between the braces. We first will set the i th value of \log_Linfs equal to the maximum value in the i th column of the lengths matrix. As we have seen in previous videos we extract the i th column using the column function, and then use the max function to calculate the maximum value for that column, that is the lengths for the i th pond.

Slide Code:

```
for(int i=1; i<=nponds;i++)
{
log_Linfs(i) = max(column(lengths,i));
}
```

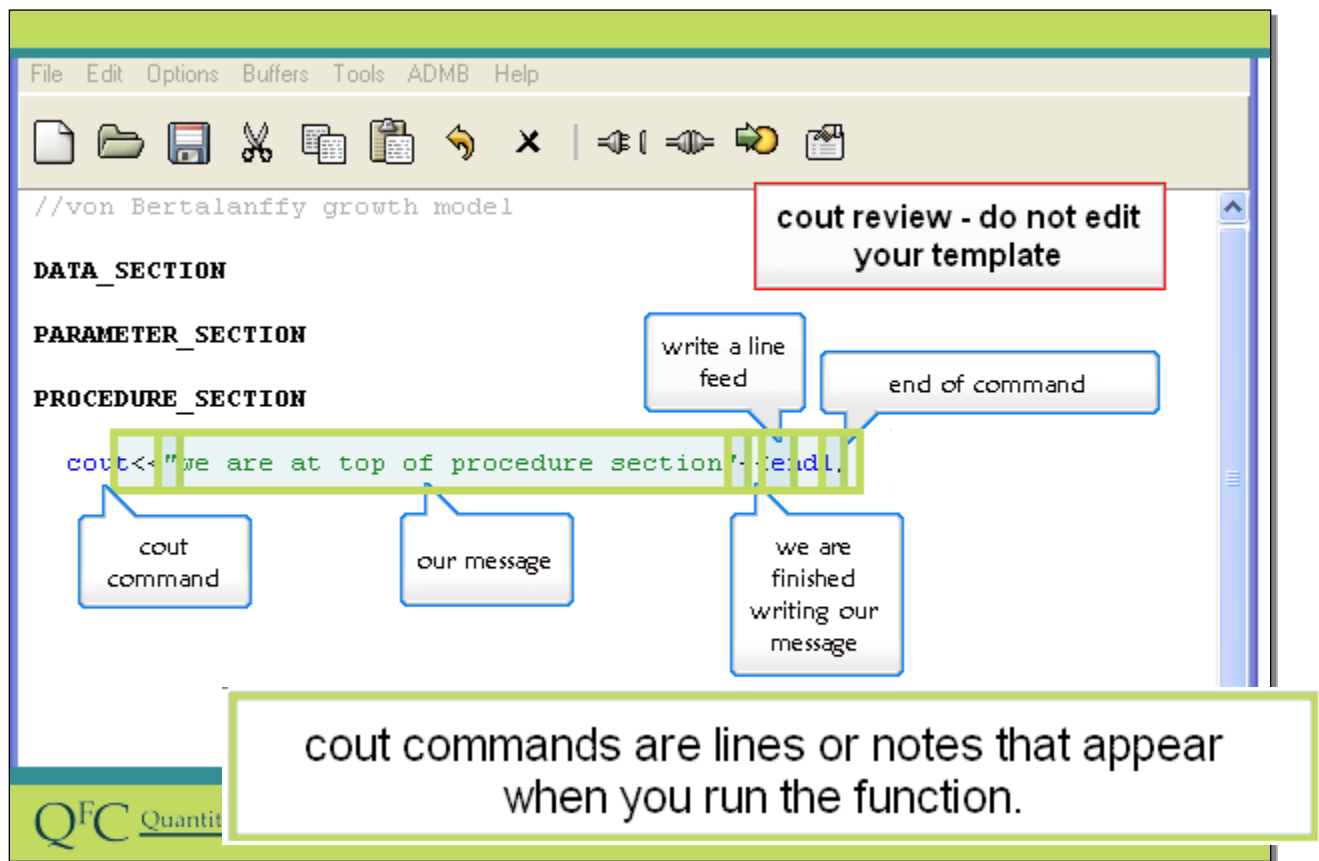


We really want the log of the maximum so we take the log of this result in the next line.

Note that when we want to refer to just a single element of a vector we simply use the vector name, followed by parentheses with the element number inside. We used `i` inside `log_Linfs` to refer to the *i*th element `log_Linfs` followed by parentheses. If we used five inside as in "`log_Linfs(5)`" this would refer to the 5th element. Next we will use `cout`. Click the I want a quick review of `cout` if you would like a review otherwise click next to move on to coding.

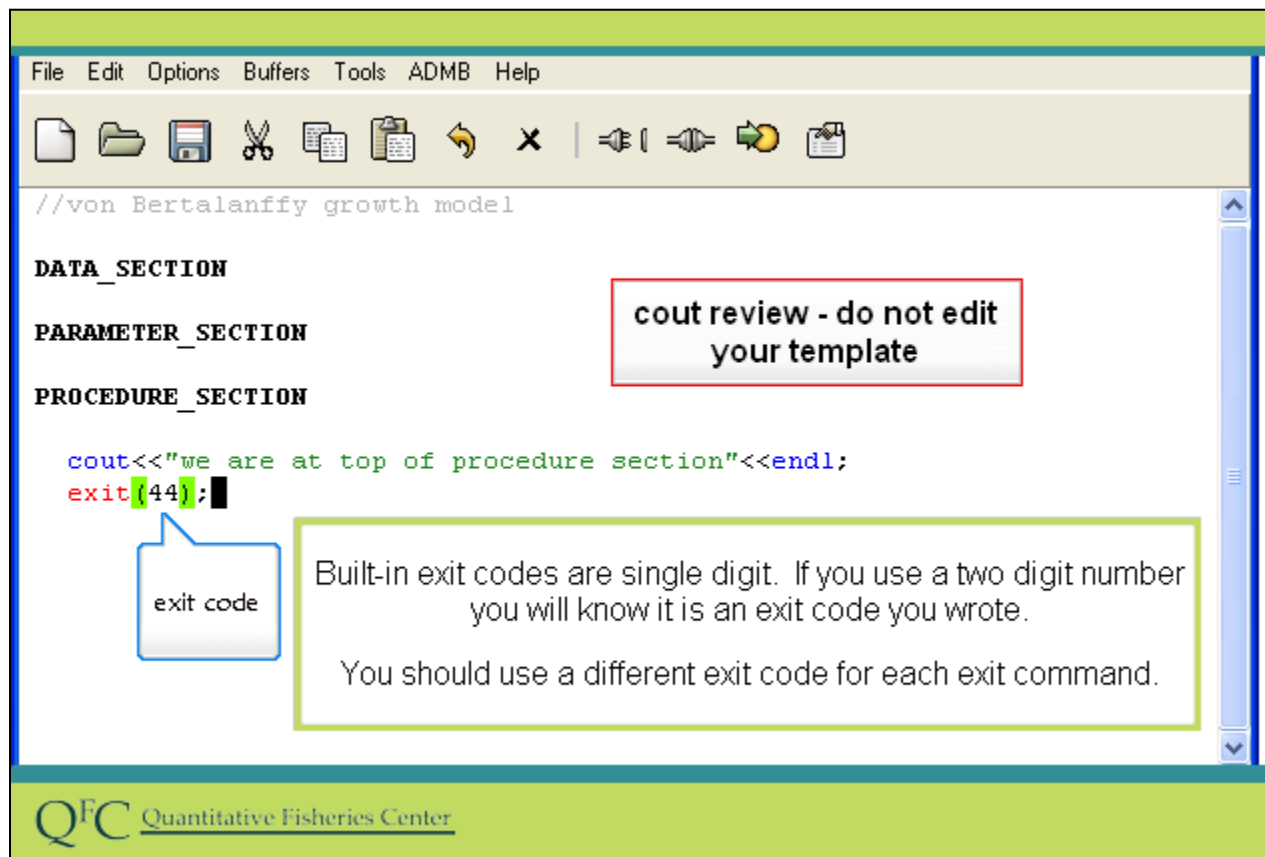
Slide Code:

```
for(int i=1; i<=nponds;i++)  
{  
log_Linfs(i) = max(column(lengths,i));  
log_Linfs(i) = log(log_Linfs (i));  
}
```



The first is a cout command. Notice that we indent this line with two spaces. Usually you will need to indent lines two spaces or more within sections. If you do not indent them or indent them one space admb interprets this to mean that the lines are of a special type and if they are not this will cause errors. Cout commands will produce lines of output you will be able to see when your program is running.

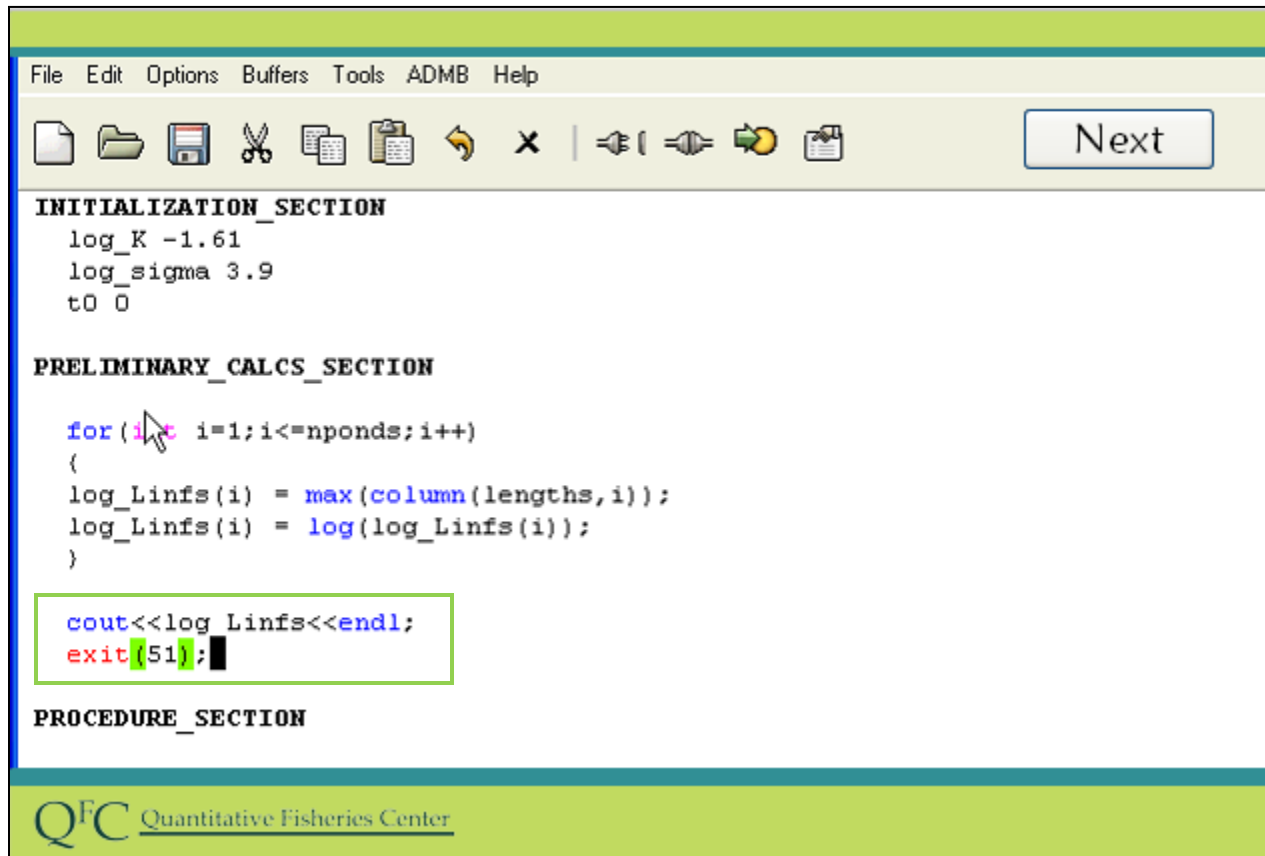
We follow cout with two left pointing arrows. Then we include some text we want written out to our screen inside quotes. We follow this with a second set of arrows indicating we are done writing the first thing out. The line ends with "endl" telling our program to write a line feed and a semi-colon indicating the end of a command. In the Procedure section all lines end with a semi-colon. This is because all our lines of code in this section will end up being used as C++ lines in our final program which is created automatically from our template file. C++ expects all lines to end with a semi-colon so all lines in this section need to end with one.



The number "44" within the braces is an exit code. I just chose a two digit number somewhat arbitrarily. Built-in exit codes for admb programs are typically single digit ones so when we get this exit code we will know the program got to our exit command. If you have multiple exit commands in your program and you want to know for sure which exit command was reached when the program exited you should use a different exit code for each. Now, before moving on, another question.

So our program has at this point been set up to simply get to the start of the procedure section and write out some text and then quit. Notice that cout and exit are standard C++ commands that could be used in a C++ program not related to admb. Technically the cout command writes to the standard output device, unless you deliberately make a change and will usually be a visible buffer when using ADMB-IDE.

THIS IS JUST A REVIEW, DO NOT EDIT YOUR TEMPLATE



```
File Edit Options Buffers Tools ADMB Help

log_K -1.61
log_sigma 3.9
t0 0

INITIALIZATION_SECTION

PRELIMINARY_CALCS_SECTION

for(i=1;i<=nponds;i++)
{
    log_Linfs(i) = max(column(lengths,i));
    log_Linfs(i) = log(log_Linfs(i));
}

cout<<log_Linfs<<endl;
exit(51);

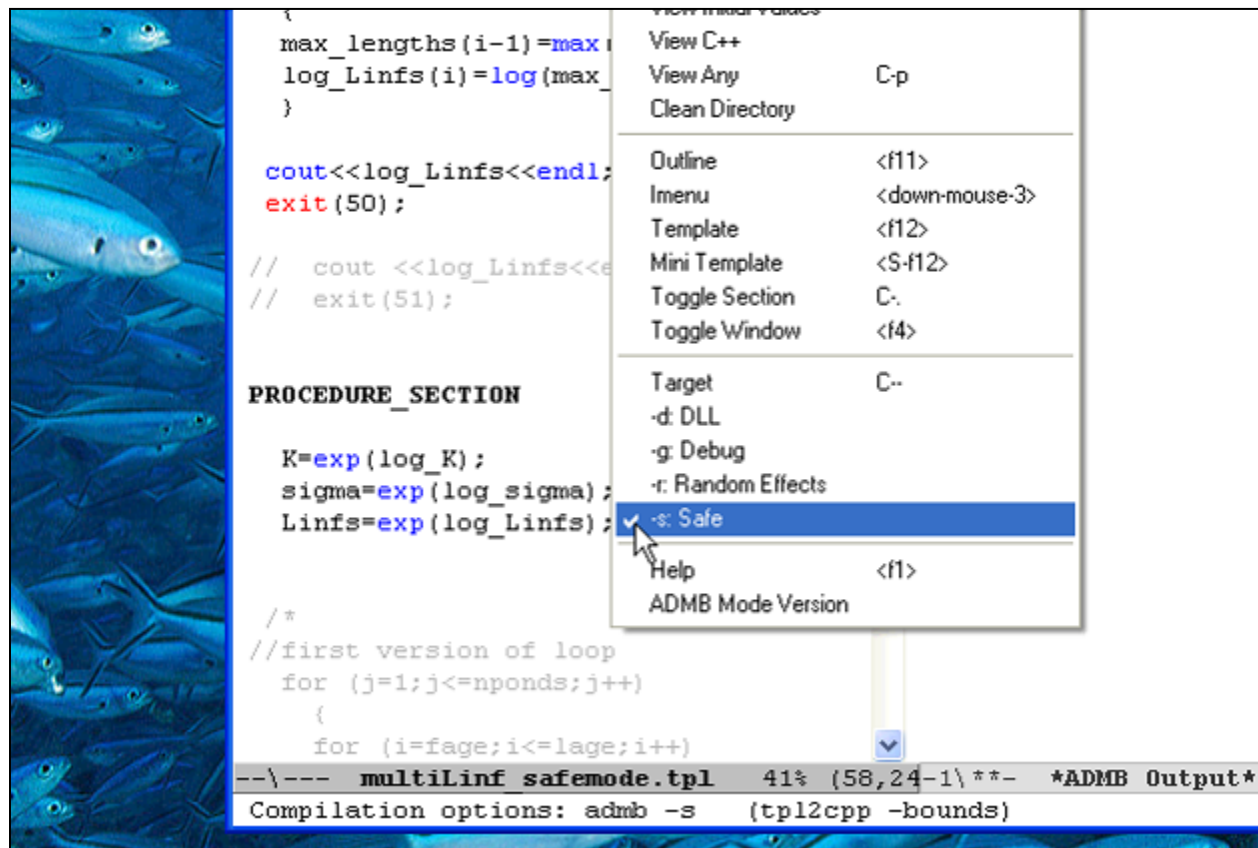
PROCEDURE_SECTION
```

Q^{FC} Quantitative Fisheries Center

We add a cout command and an exit command after the loop to check and see if our loop worked right.

Slide Code:

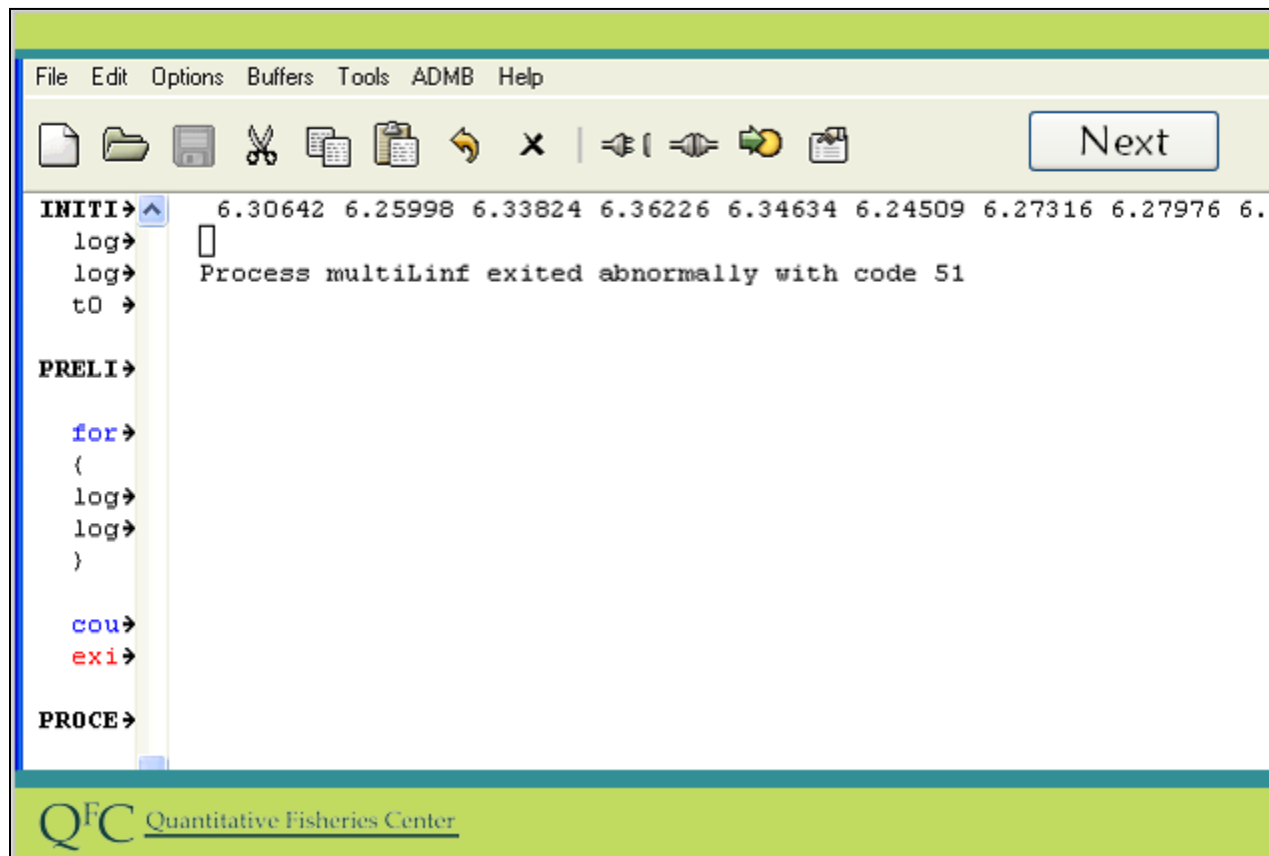
```
cout<<log_Linfs<<endl;
exit(51);
```



As mentioned in earlier videos, some experienced admb users recommend always using safe mode -- that is the safe libraries, until a program is debugged..., but the optimized libraries are used by default. In these videos we are using the optimized libraries, but if you wish, you can switch to safe mode from the admb menu. Go to the target section and select es, safe mode. You can double check you are in safe mode by making sure there is a check next to safe mode in the ADMB menu. We remind you about safe mode at this point because safe mode can be particularly useful in finding some nasty bugs associated with loops. We encourage you to view our video on the safe mode when you complete these videos on loops.

Slide Action:

Go to the ADMB menu and select --s: Safe. You know you are in safe mode if you have a check next to it.



```
File Edit Options Buffers Tools ADMB Help
[Icons] [Next]
INITI→ 6.30642 6.25998 6.33824 6.36226 6.34634 6.24509 6.27316 6.27976 6.
log→
log→ Process multiLinf exited abnormally with code 51
t0 →

PRELI→

for→
{
log→
log→
}

cou→
exi→

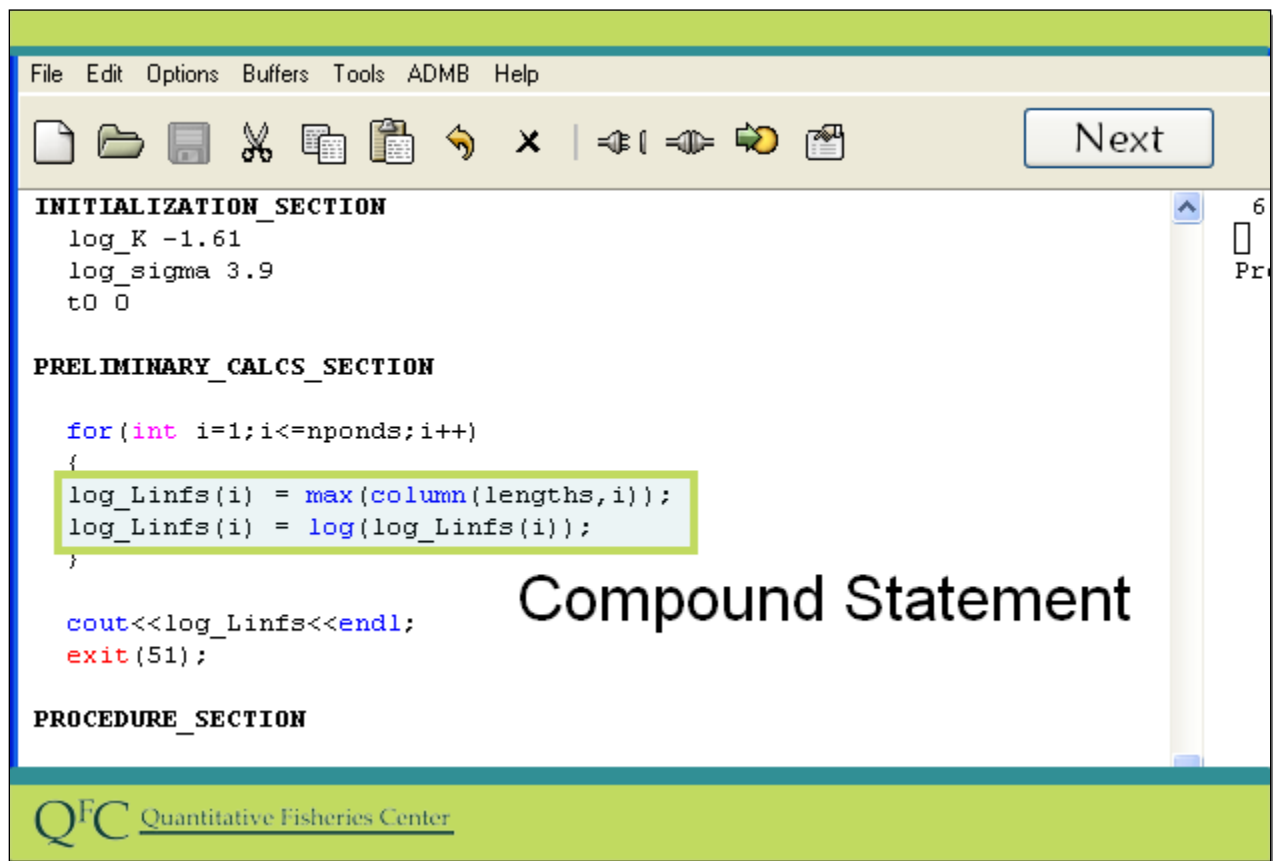
PROCE→

QFC Quantitative Fisheries Center
```

We build our program and run it and indeed the result is that `log_Linf` is taking different starting values for each pond equal to the log of the maximum observed length. Also notice that it exited abnormally with exit code 51 that we just entered.

Slide Action:

Build and run program.



The screenshot shows a C++ IDE window with a menu bar (File, Edit, Options, Buffers, Tools, ADMB, Help) and a toolbar. The code is organized into sections: **INITIALIZATION_SECTION** with variables `log_K`, `log_sigma`, and `t0`; **PRELIMINARY_CALCS_SECTION** containing a `for` loop; and **PROCEDURE_SECTION**. Inside the `for` loop, two lines are highlighted with a green box: `log_Linfs(i) = max(column(lengths,i));` and `log_Linfs(i) = log(log_Linfs(i));`. A large text overlay "Compound Statement" points to these two lines. The IDE also features a "Next" button and a line number indicator on the right.

```
File Edit Options Buffers Tools ADMB Help

log_K -1.61
log_sigma 3.9
t0 0

INITIALIZATION_SECTION

PRELIMINARY_CALCS_SECTION

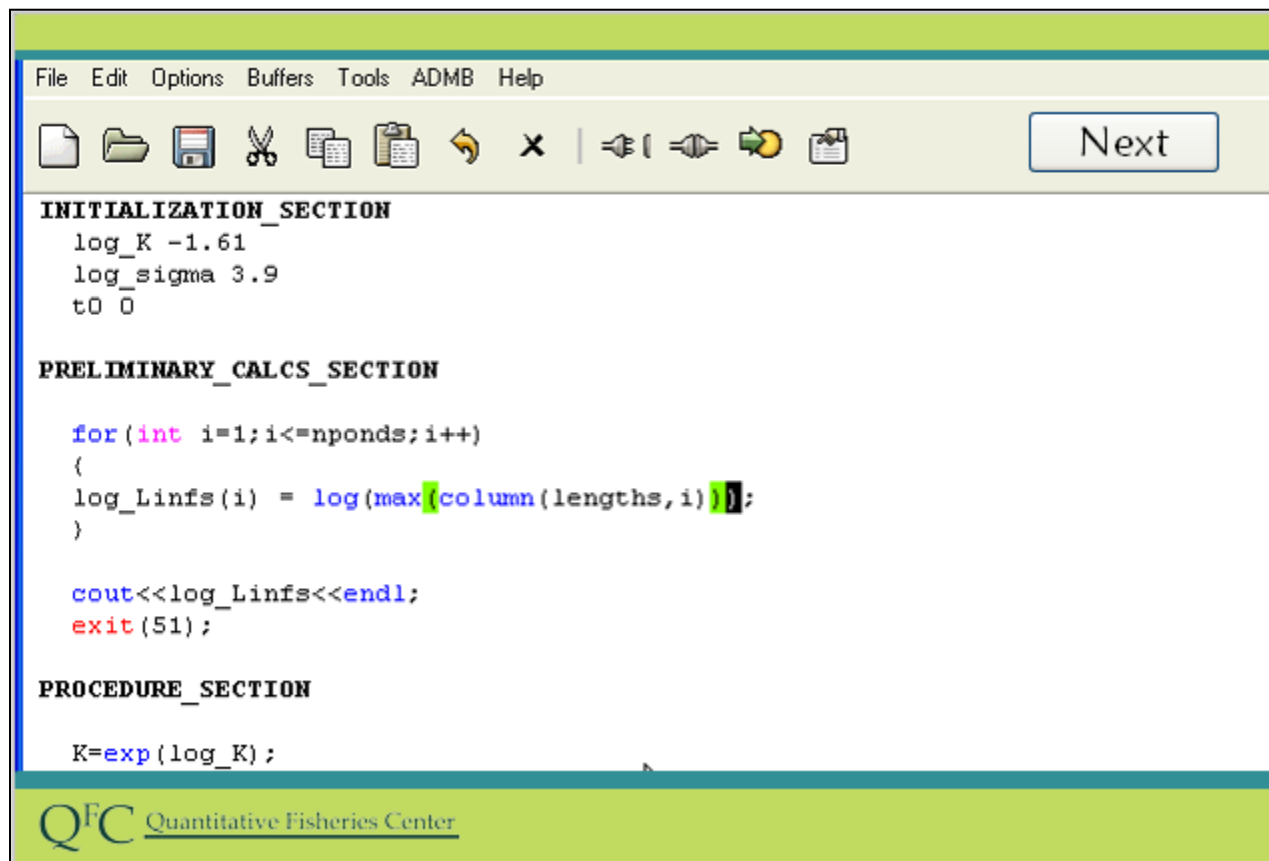
for(int i=1;i<=nponds;i++)
{
    log_Linfs(i) = max(column(lengths,i));
    log_Linfs(i) = log(log_Linfs(i));
}

cout<<log_Linfs<<endl;
exit(51);

PROCEDURE_SECTION
```

Compound Statement

Given this construction of a “for loop” we can put as many executable statements ending with semi-colons as we want within the curly braces. In c++ the material within the braces are considered a compound statement.



```
File Edit Options Buffers Tools ADBM Help

log_K -1.61
log_sigma 3.9
t0 0

INITIALIZATION_SECTION

PRELIMINARY_CALC_SECTION

for(int i=1;i<=nponds;i++)
{
log_Linfs(i) = log(max(column(lengths,i)));
}

cout<<log_Linfs<<endl;
exit(51);

PROCEDURE_SECTION

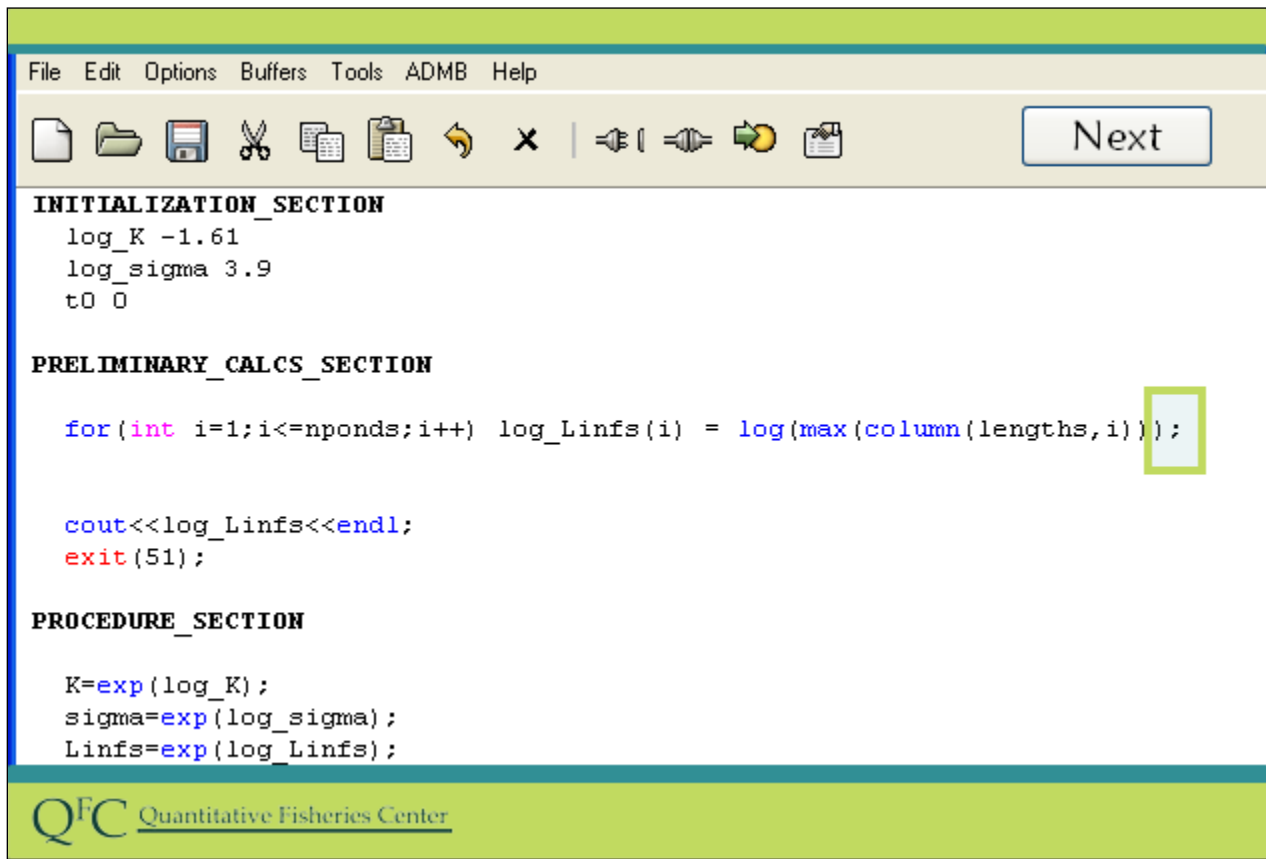
K=exp(log_K);
```

QFC Quantitative Fisheries Center

Of course we really did not need two statements here. We could delete the second line within the braces and just put parentheses around the right hand side of the first statement and then take the log of it. Make sure you add the additional closing parenthesis.

Slide Code:

```
for(int i=1; i<=nponds;i++)
{
log_Linfs(i) = log(max(column(lengths,i)));
log_Linfs(i) = log(log_Linfs(i));
}
```



```
File Edit Options Buffers Tools ADBM Help

[Icons] [Next]

INITIALIZATION_SECTION
log_K -1.61
log_sigma 3.9
t0 0

PRELIMINARY_CALCS_SECTION

for(int i=1;i<=nponds;i++) log_Linfs(i) = log(max(column(lengths,i)));

cout<<log_Linfs<<endl;
exit(51);

PROCEDURE_SECTION

K=exp(log_K);
sigma=exp(log_sigma);
Linfs=exp(log_Linfs);

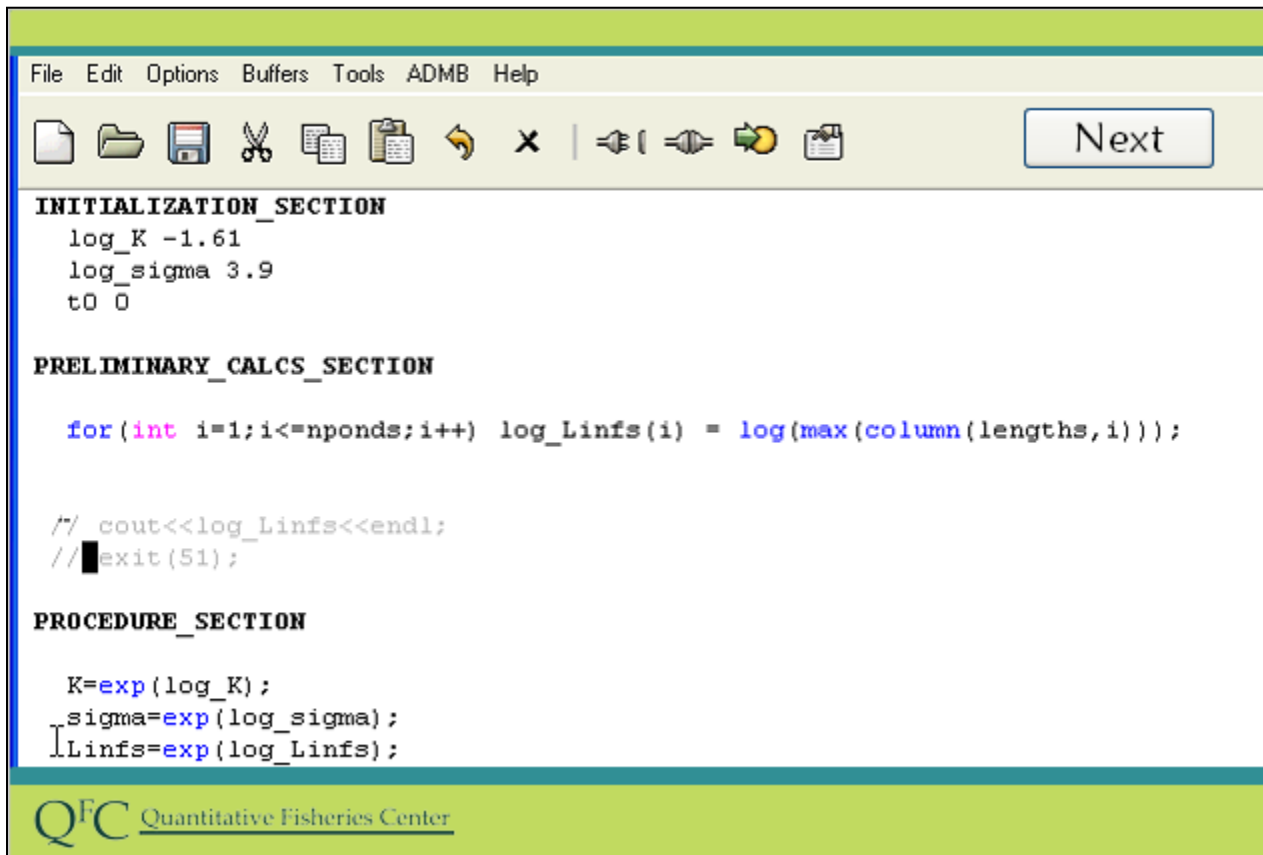
QFC Quantitative Fisheries Center
```

This will work fine. But notice that our compound statement is actually just a single statement. When our for loop is executing just a single statement we could get rid of the curly braces.

Notice that our single statement ends with a semi-colon indicating the end of the statement to be executed in the for loop. Previously the curly braces indicated start and end of the compound statement being executed in the for loop, and each individual statement within the compound statement ended with its own semi-colon. It is not uncommon to see for loops that use curly braces but have only a single statement and this is fine. Some programmers always put the curly braces in before they start writing what they want done in the loop before figuring out they can write it as a single statement.

Slide Action:

Remove both curly braces and move line up to make it one line



```
File Edit Options Buffers Tools ADMB Help

INITIALIZATION_SECTION
    log_K -1.61
    log_sigma 3.9
    t0 0

PRELIMINARY_CALCS_SECTION

    for(int i=1;i<=nponds;i++) log_Linfs(i) = log(max(column(lengths,i)));

    /* cout<<log_Linfs<<endl;
    //exit(51);

PROCEDURE_SECTION

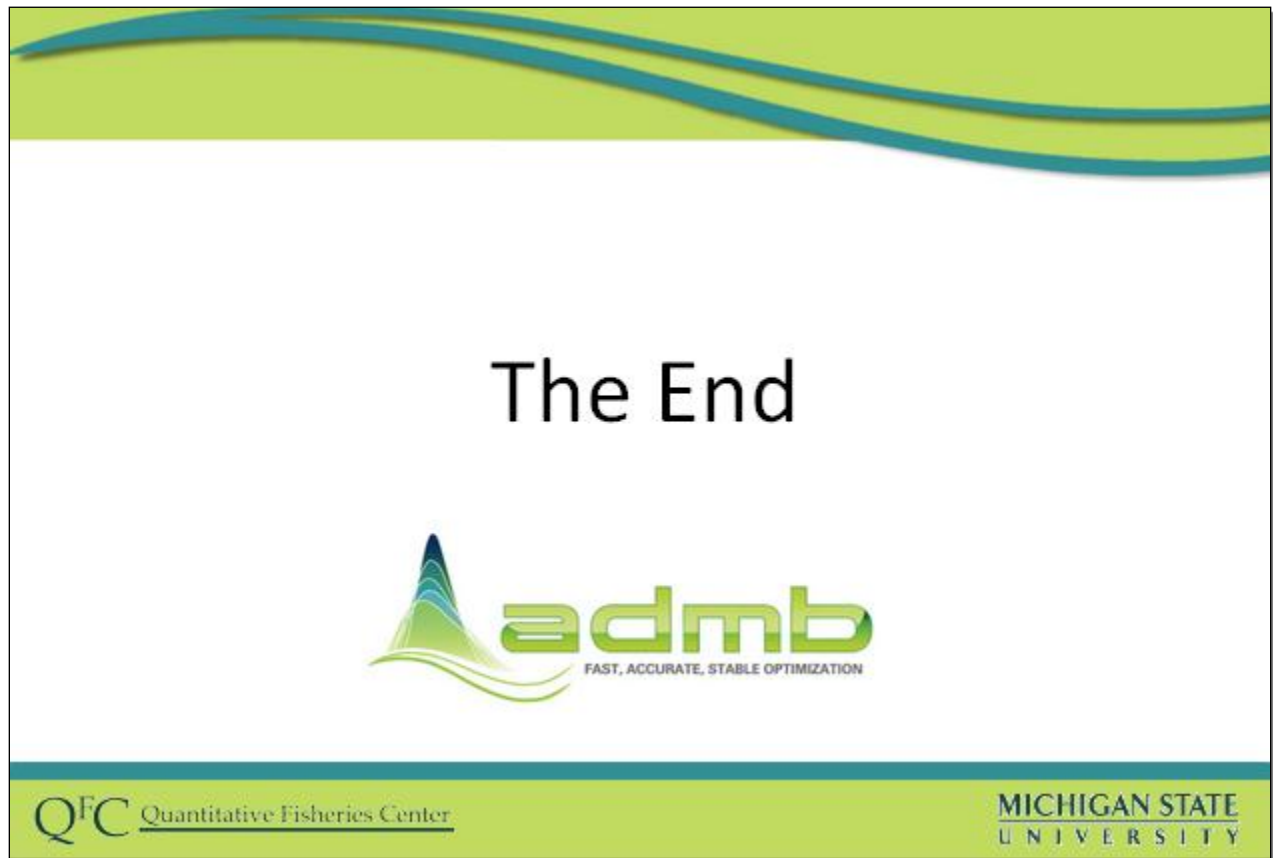
    K=exp(log_K);
    sigma=exp(log_sigma);
    Linfs=exp(log_Linfs);
```

QFC Quantitative Fisheries Center

We now need to comment out our cout and exit statements so next time we run it it won't stop here.

Slide Code:

```
// cout<<log_Linfs<<endl;
// exit(51);
```



This concludes our first video on looping.