



Continue

Creating your first AD Model Builder application

Part 2 - Coding the template and complete it through reading the data in.



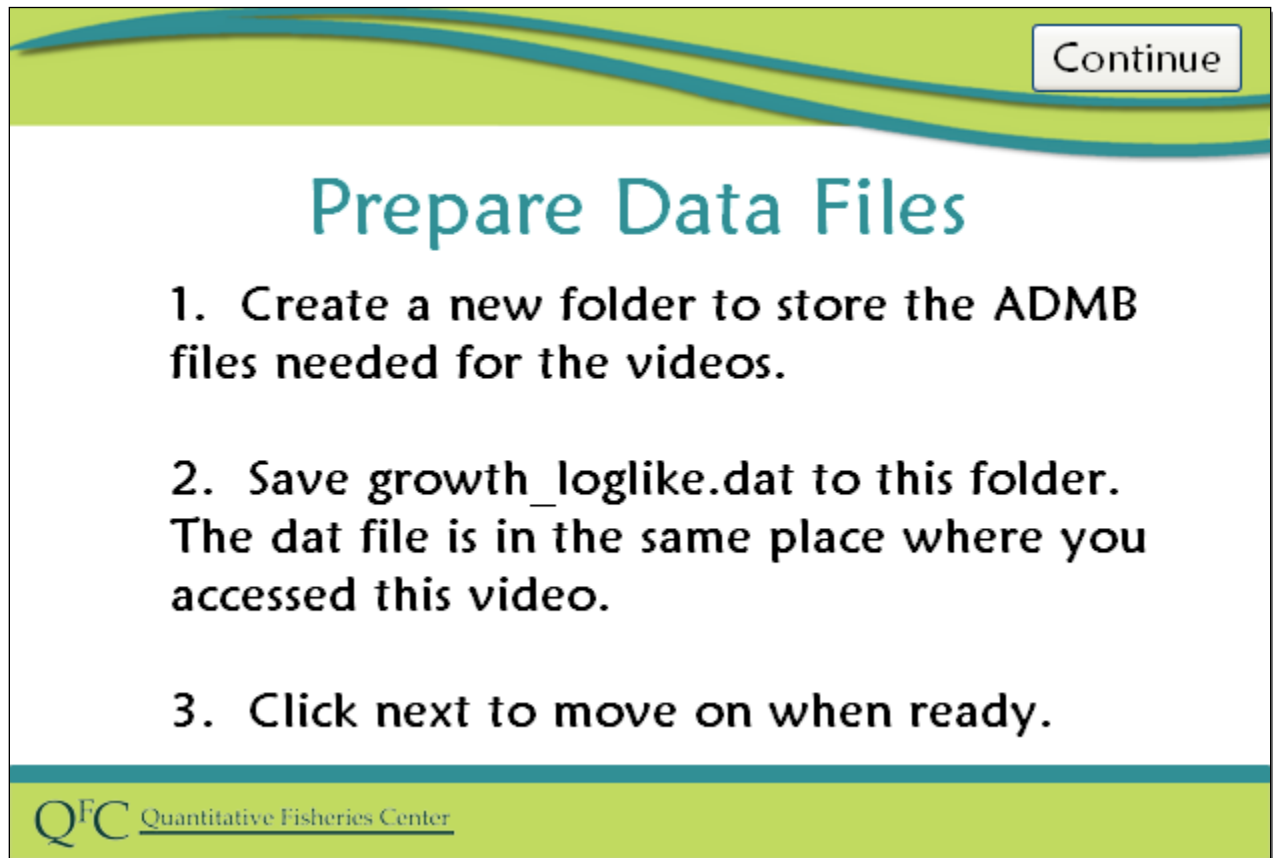
This video was created using ADMB-IDE release 4.4.0-2 (January, 2011)
You may notice some minor differences if using a different version.

 Quantitative Fisheries Center

MICHIGAN STATE
UNIVERSITY

This is the second in a series of videos stepping you through a process for developing an admb application. The first in the series introduced you to the data and model we will be fitting to the data. In this video we will begin coding the template or “t-p-l” file and complete it through the stage of reading the data in.

Prepare the Data



Continue

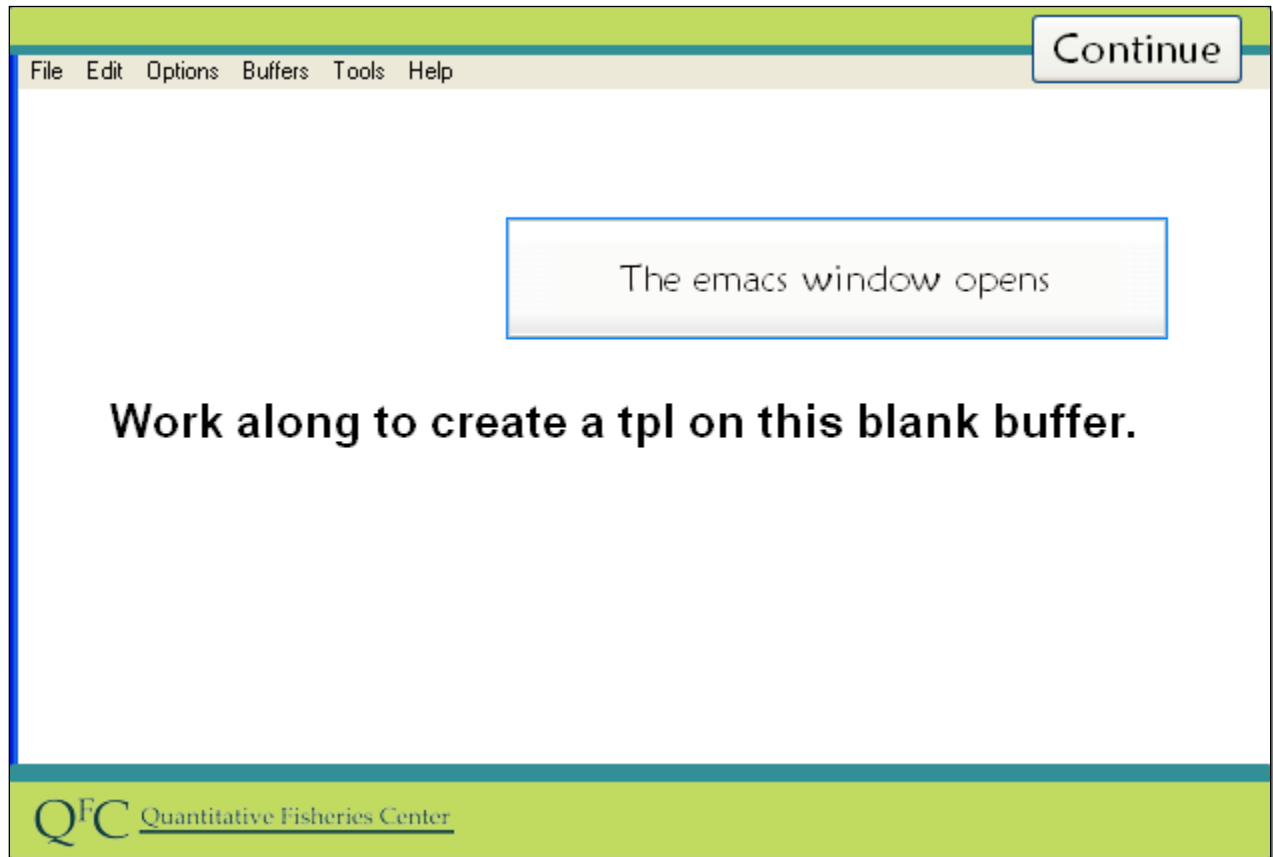
Prepare Data Files

1. Create a new folder to store the ADMB files needed for the videos.
2. Save `growth_loglike.dat` to this folder. The dat file is in the same place where you accessed this video.
3. Click next to move on when ready.

QFC Quantitative Fisheries Center

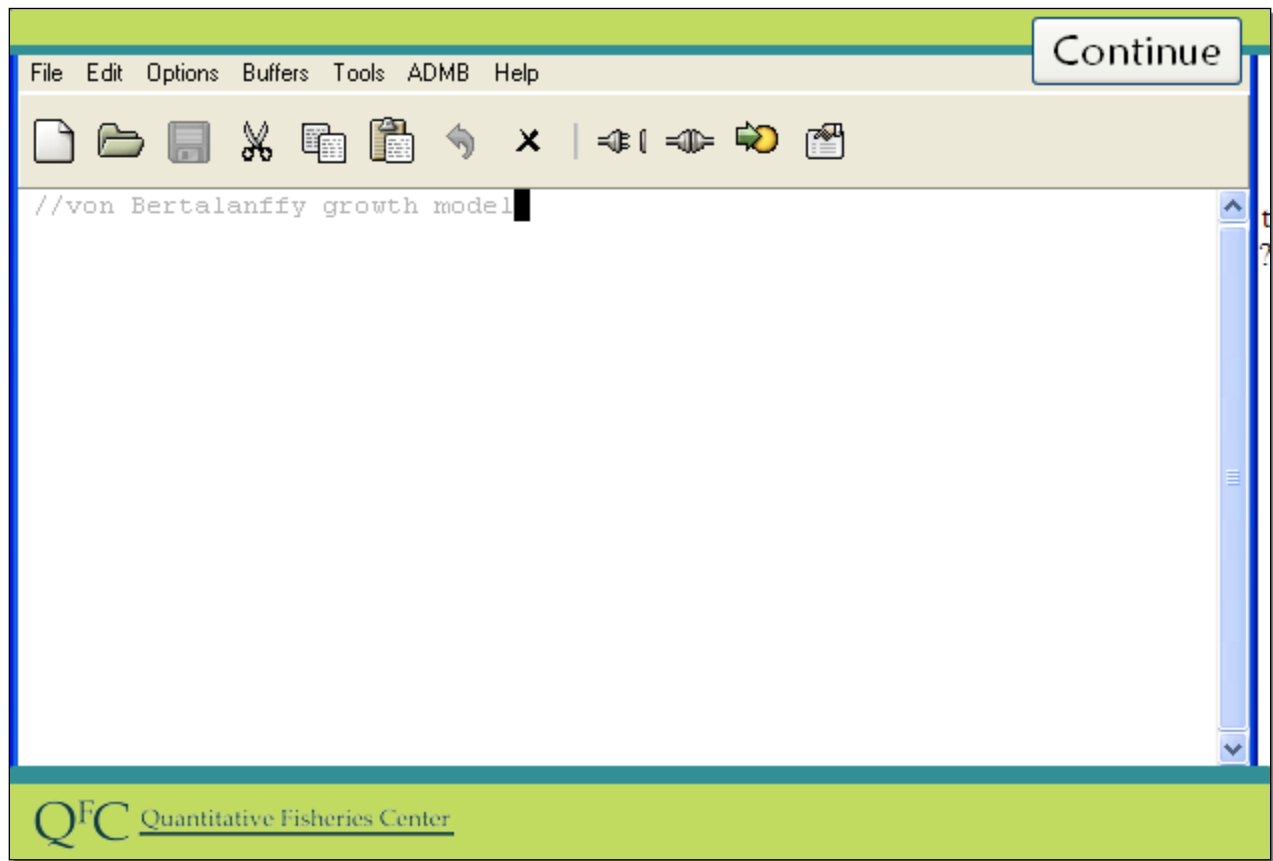
1. Create a new folder.
2. Save `growth_loglike.dat` to this folder.

Open ADMB



This set of videos is designed to guide you as you build a tpl of your own.

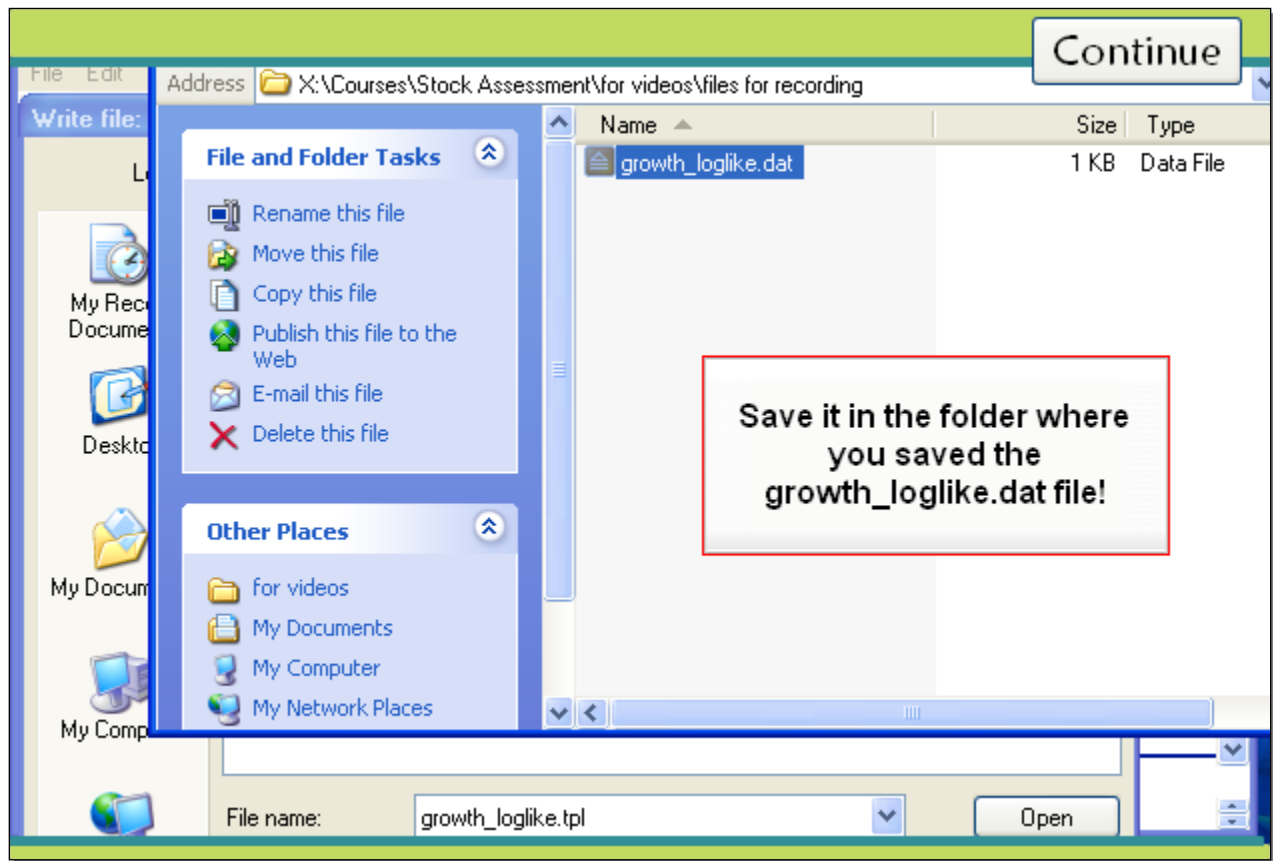
As a starting point let's create the minimal admb template file that will compile and run without errors. First I will open the admb-IDE emacs interface to display a new buffer. We recommend you work along with us. We will work with this new blank buffer. A buffer is just what admb calls an open window that may or may not correspond to a file saved on disk.



First I will add a comment line. Notice that in the tpl file a comment starts with two slashes. This is different than the pound sign used in the data file. You will get errors either by using the slashes in the data files or using the pound sign here in the tpl.

Slide Code:

//von Bertalanffy growth model

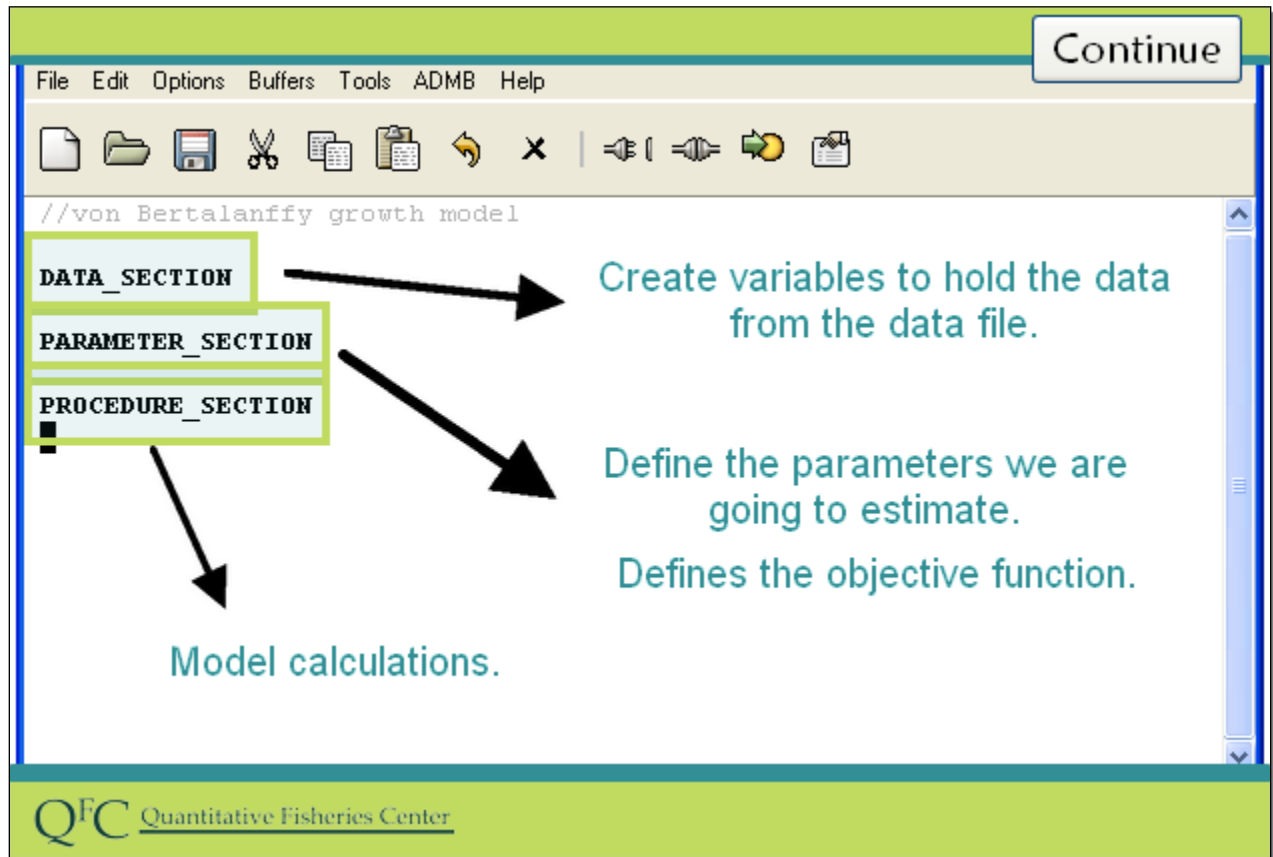


Now save the file.

- Go to file.
- You need to use “save as” because this file has not been saved before.
- I am calling this growth_loglike.tpl.
- Notice the name matches the name of the data file and the extent is what admb expects for a template.

I will save it in the same directory that the dat file is saved because this is what admb expects by default. You will next have a chance to check your understanding by answering a quick question before we move on.

Add Sections



Now I am going to add some more to my tpl file. Every functioning admb program has at least three sections: a data section, a parameter section, and a procedure section. So first we add these sections but leave them empty for now. Section names start at the far left margin with no leading blanks. They are all capitals and spelled just as I spell them here. When you get them right they will become bold if you use the admb-IDE interface.

The data section will be where we will create variables to hold data read in from the dat file. We can also create other variables here if their values do not depend upon the parameters. For now we leave this section empty. The parameter section is where we define the parameters we are going to estimate and anything calculated from them. This includes the special objective function value variable, which is what your admb program will attempt to minimize when it is running. For now we will leave this section empty. The procedure section is where all the action is. Based on your parameters and model calculations are done to determine the objective function value. For example in our example we will generate predicted values of length given age and compare these with observed data.

Slide Code:

DATA_SECTION

PARAMETER_SECTION

PROCEDURE_SECTION

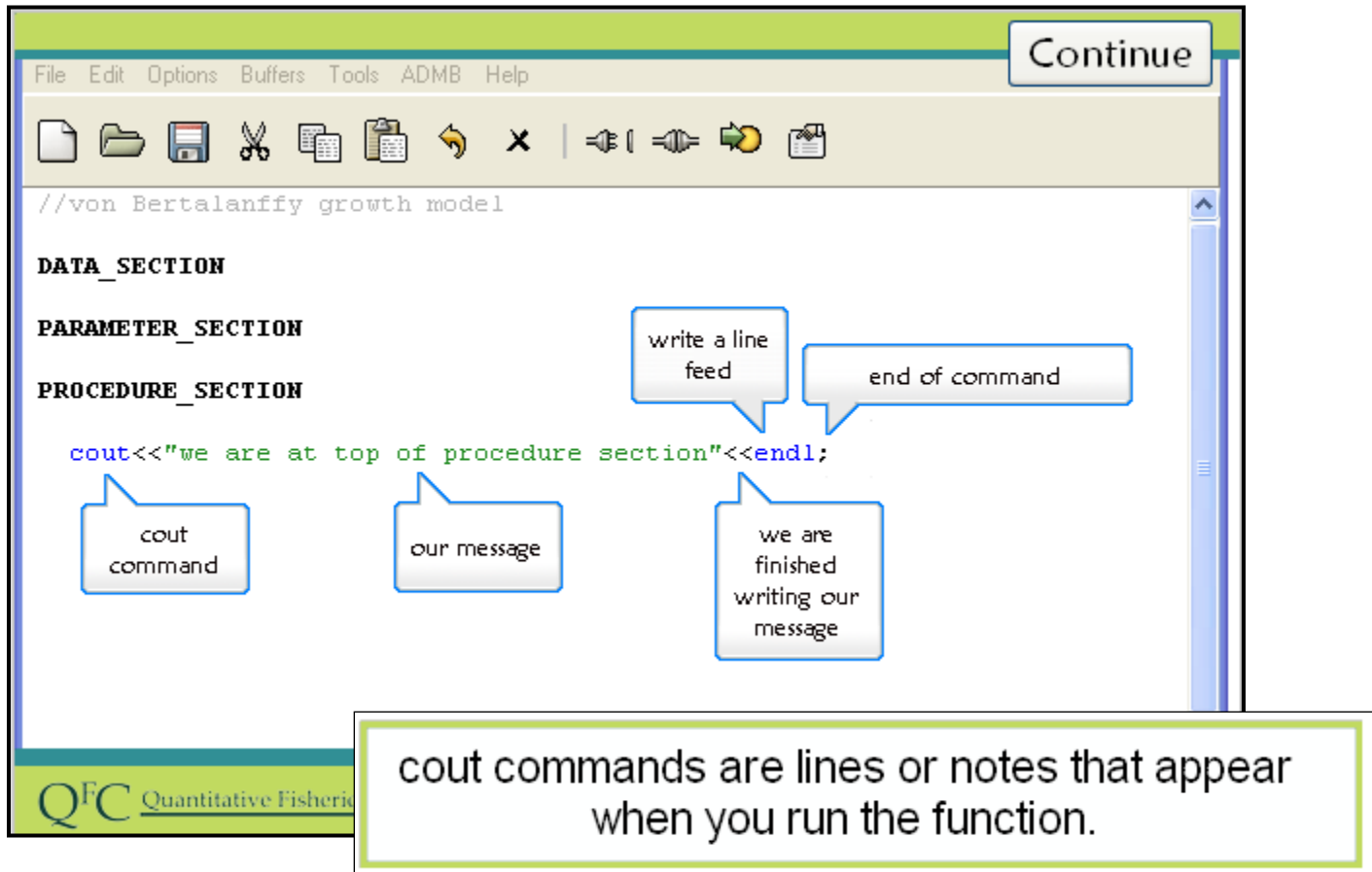
Section Name Rules

Do not indent section names!

Type in all capitals.

Verify that it is bold.

Cout and Exit



The screenshot shows the ADMB software interface with a menu bar (File, Edit, Options, Buffers, Tools, ADMB, Help) and a toolbar. The main window displays the following code:

```
//von Bertalanffy growth model  
  
DATA_SECTION  
  
PARAMETER_SECTION  
  
PROCEDURE_SECTION  
  
    cout<<"we are at top of procedure section"<<endl;
```

Annotations in the image:

- cout command**: Points to the `cout` keyword.
- our message**: Points to the string `"we are at top of procedure section"`.
- write a line feed**: Points to the `endl` manipulator.
- end of command**: Points to the semicolon `;`.

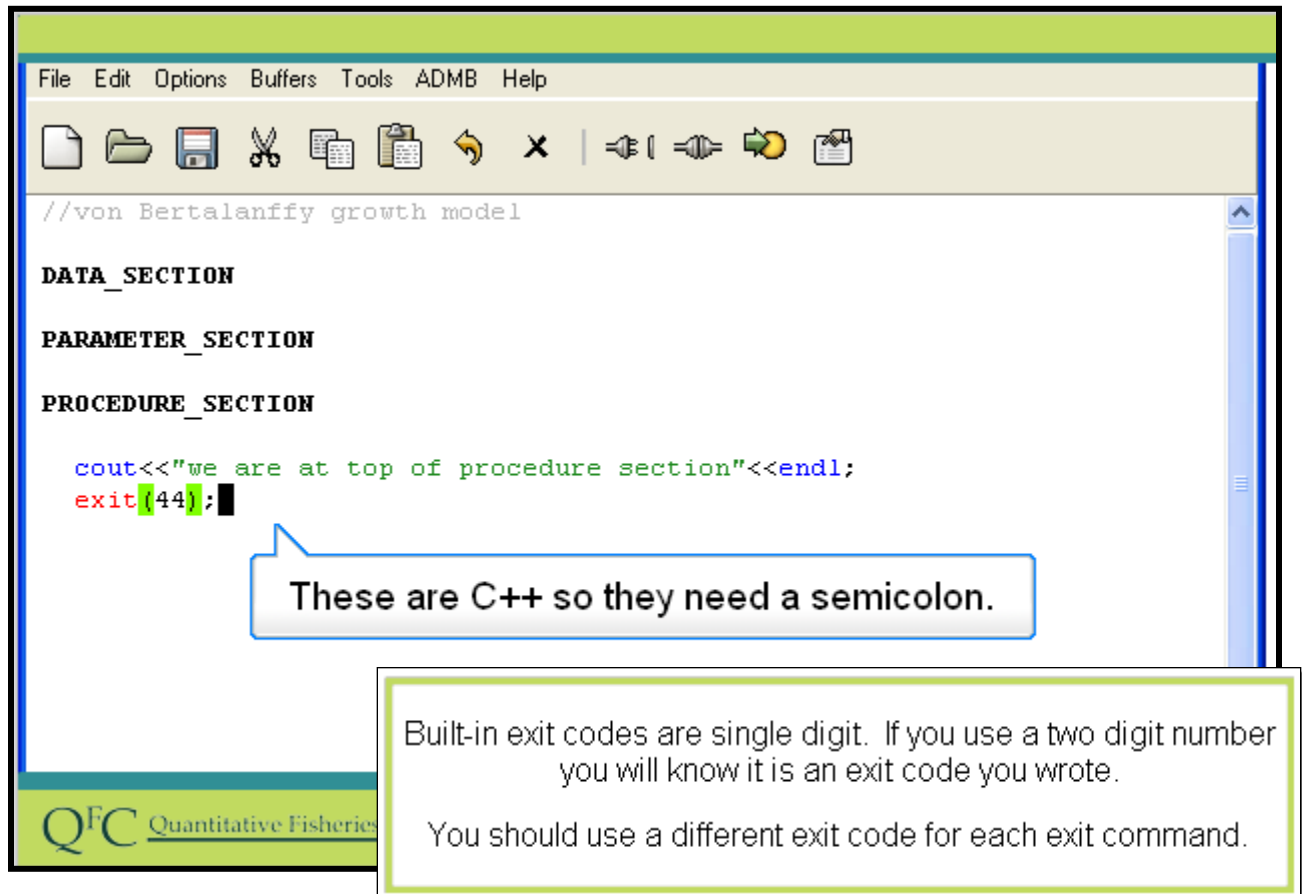
A green box at the bottom of the slide contains the text: **cout commands are lines or notes that appear when you run the function.**

Right now we are not going to add any code for our actual model to the procedure section. Instead we add two lines. 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.

Slide Code:

```
cout<< "we are at top of procedure section" <<endl; (*****Don't forget to indent)
```



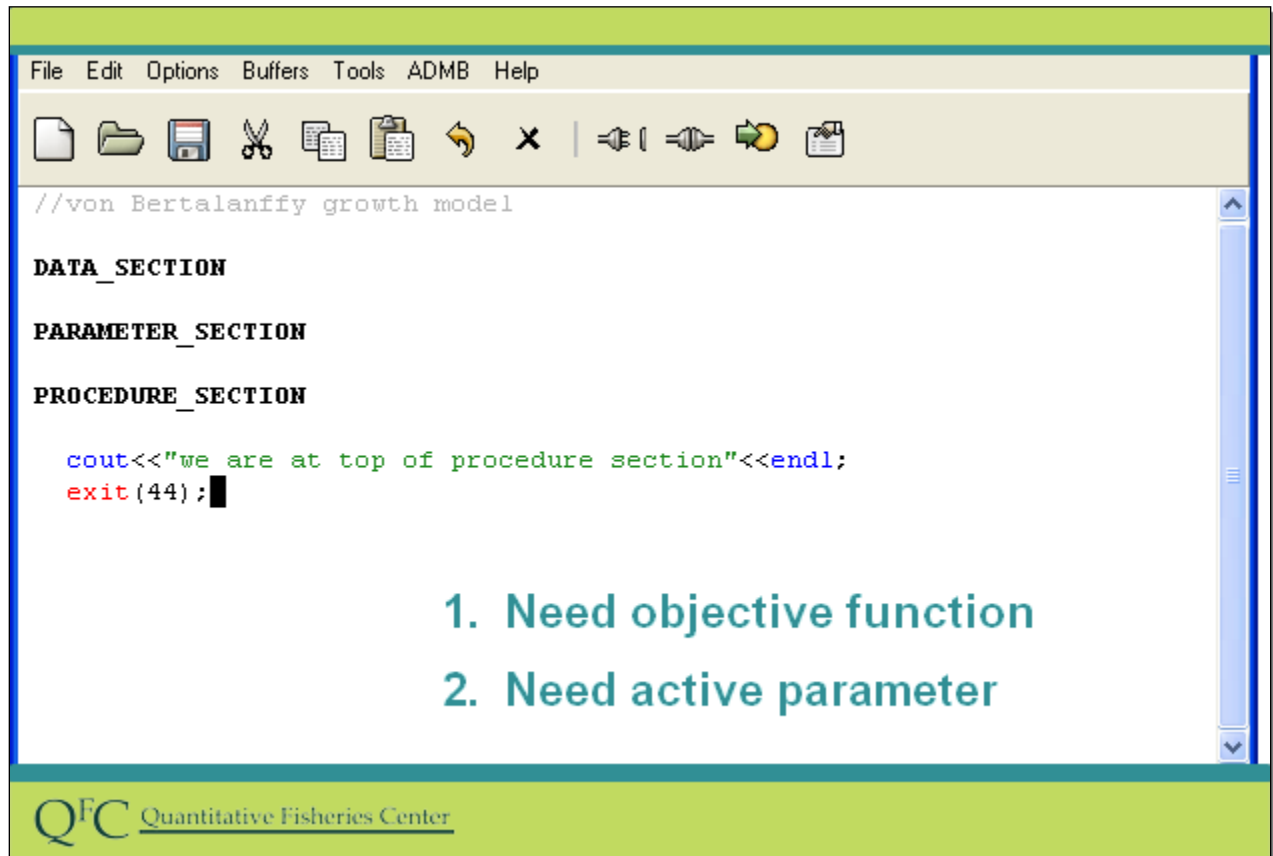
The next line uses the exit command. Exit commands tell ADMB to stop running the code.

The number "44" within the braces is an exit code. I just chose a two digit number somewhat arbitrarily. Built-in exit codes for admdb 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.

Slide Code:

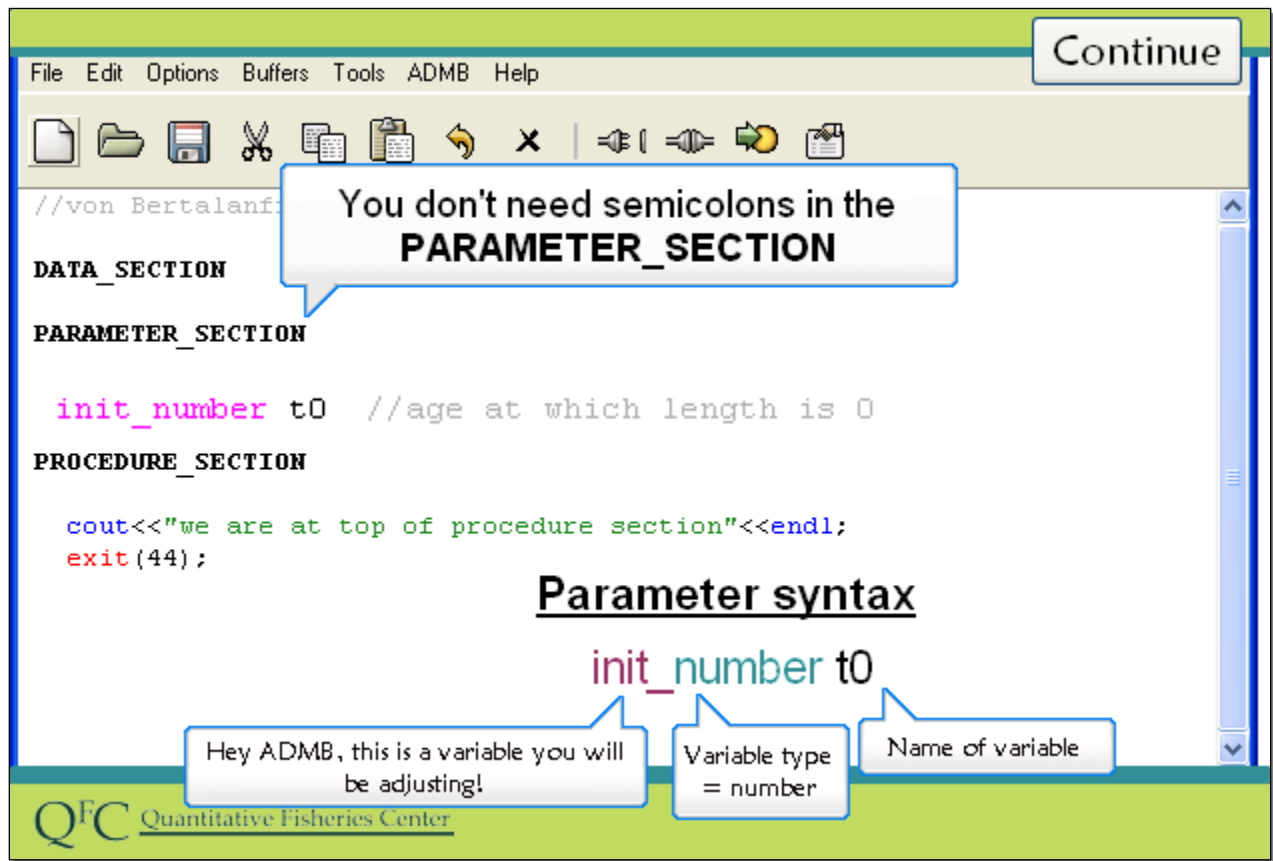
```
exit (44);
```


Objective Function



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`.

We are not yet ready to test our program because all `admb` programs must have an objective function value in order to compile. In addition, they need to have at least one active parameter in order for any code in the Procedure section to be executed without a runtime error.

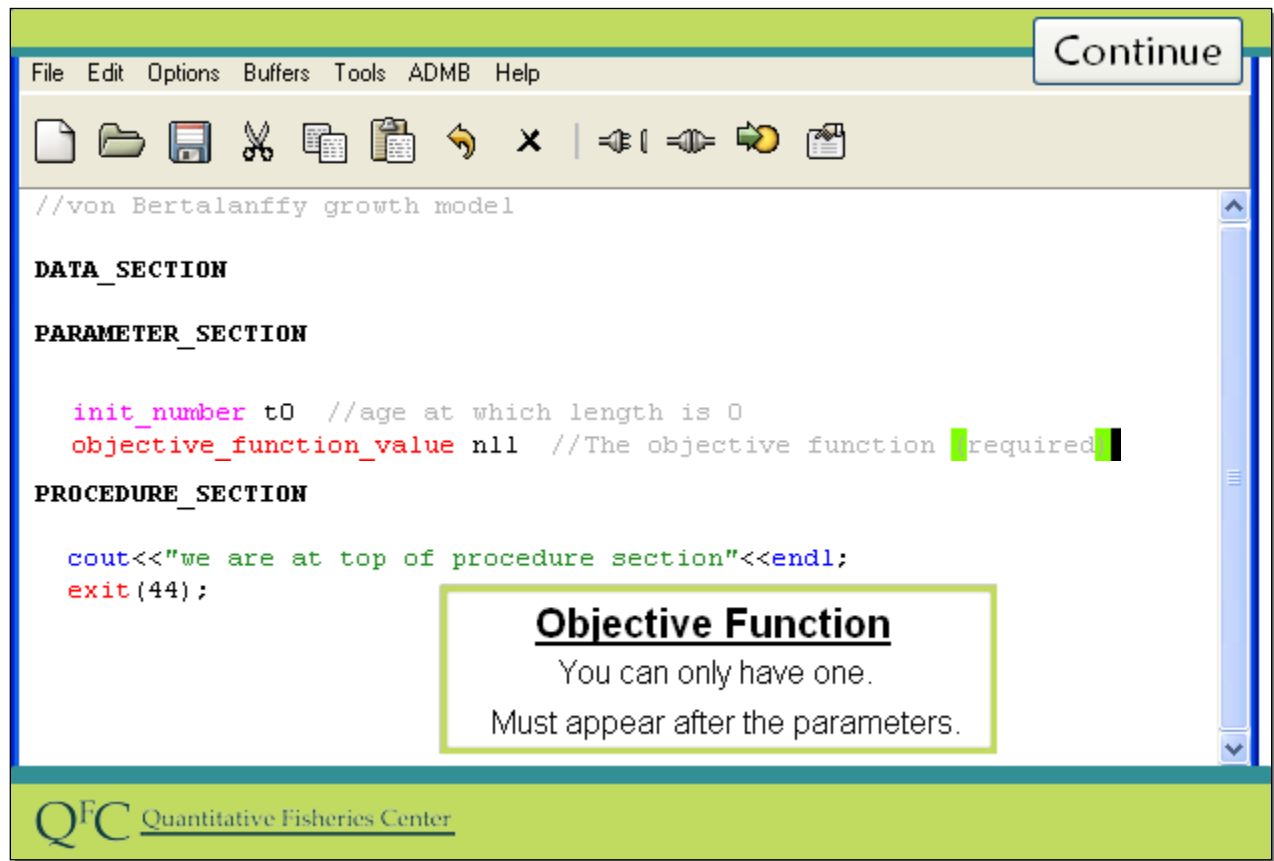


By active parameter we mean a parameter that will be estimated when we reach that stage. We must first add these definitions to the parameter section.

We can define a single parameter with this line in the parameter section. The text `init_number` tells our admb program what kind of variable we are creating and this is followed by the variable name, In this case `t0`. In the parameter section “init” at the start when we specify the type of a variable tells admb that this is one of the parameters it will be estimating. During the iterative fitting process these will be adjusted following a quasi-Newton automated procedure. Notice that we end this line with a comment. It is legal to include a comment on the same line as active code as well as on its own line. Everything after the double slashes is ignored when the program runs. Also notice that there is no semi-colon at the end of this command. Lines you write here get translated into real C++ code when your template is automatically converted to a C++ program. Because you are not writing actual C++ code in this section, semi-colons are not needed. However, like most other lines within sections we indent this one with two or more spaces.

Slide Code:

```
init_number t0 // age at which length is 0
```



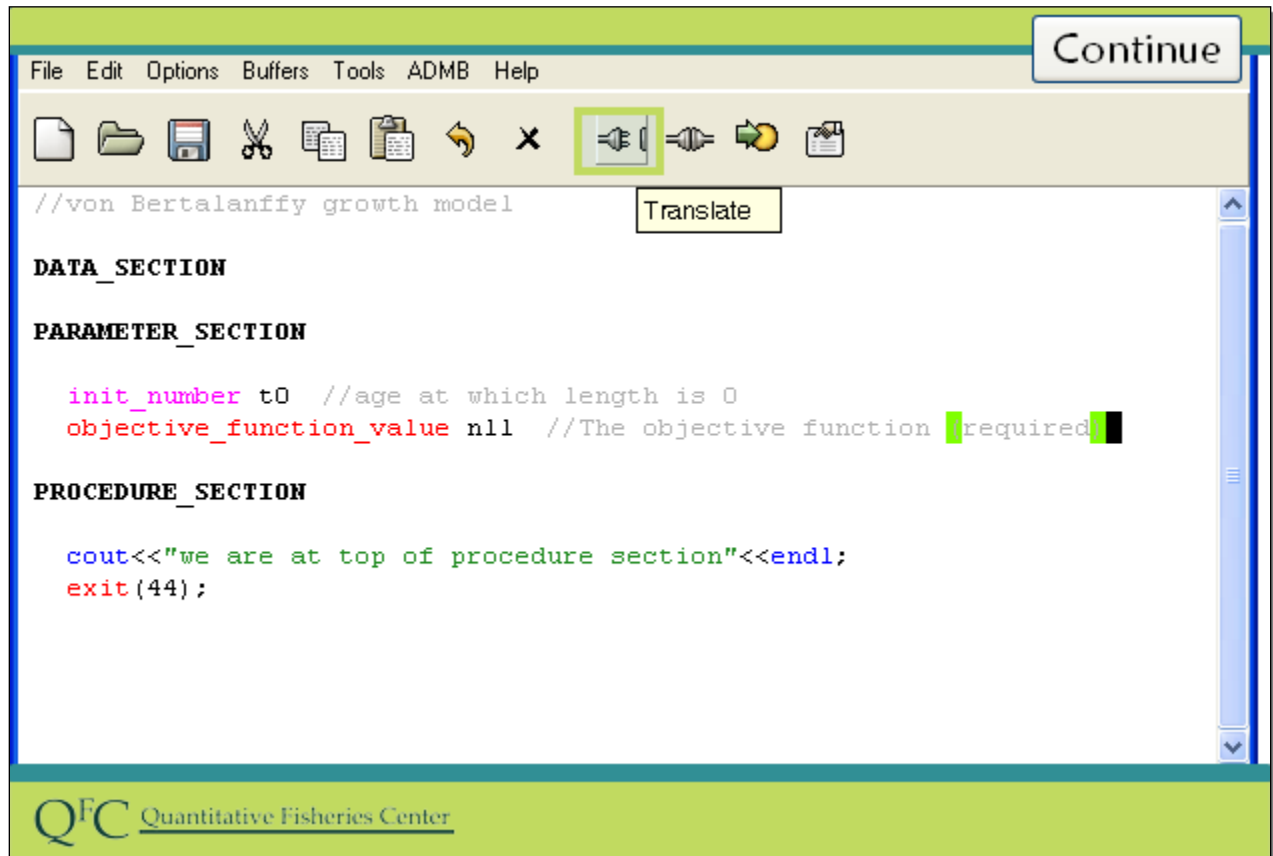
The second line we add creates the variable `nll` as a variable of the special type `objective-function-value`. This tells `admb` that this will hold the quantity you want to minimize by adjusting the estimated parameters. We chose “`nll`” for this variable because we will want to minimize the negative log-likelihood. You can define one and only one variable of `objective-function-value` type and it must come after all the definitions for estimated parameters.

Our template is now ready for an initial test. We should be able to create a working `admb` program. Of course it really will not do anything other than write out some text and exit before it realizes that the objective function does not respond to changes in the one parameter we have defined!

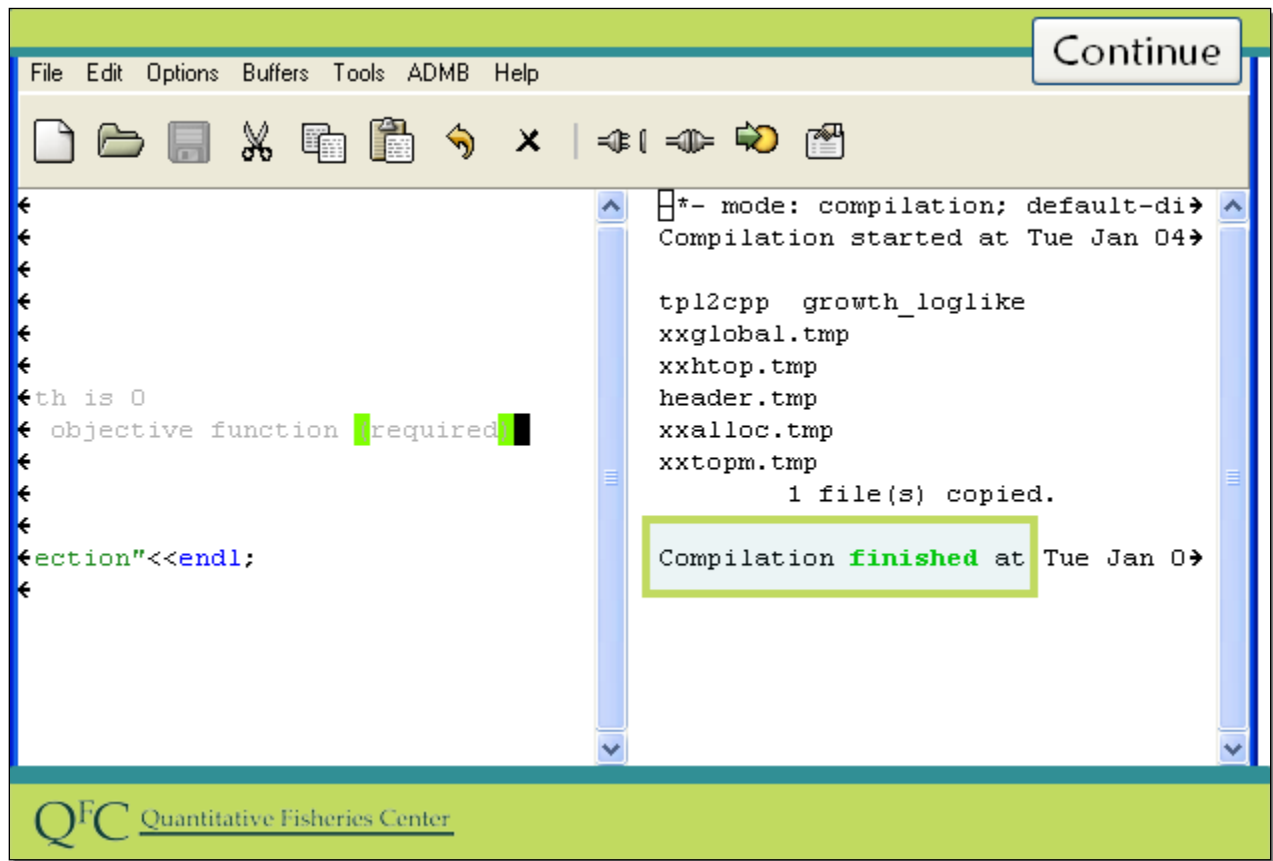
Slide Code:

```
objective_function_value nll // The objective function (required)
```

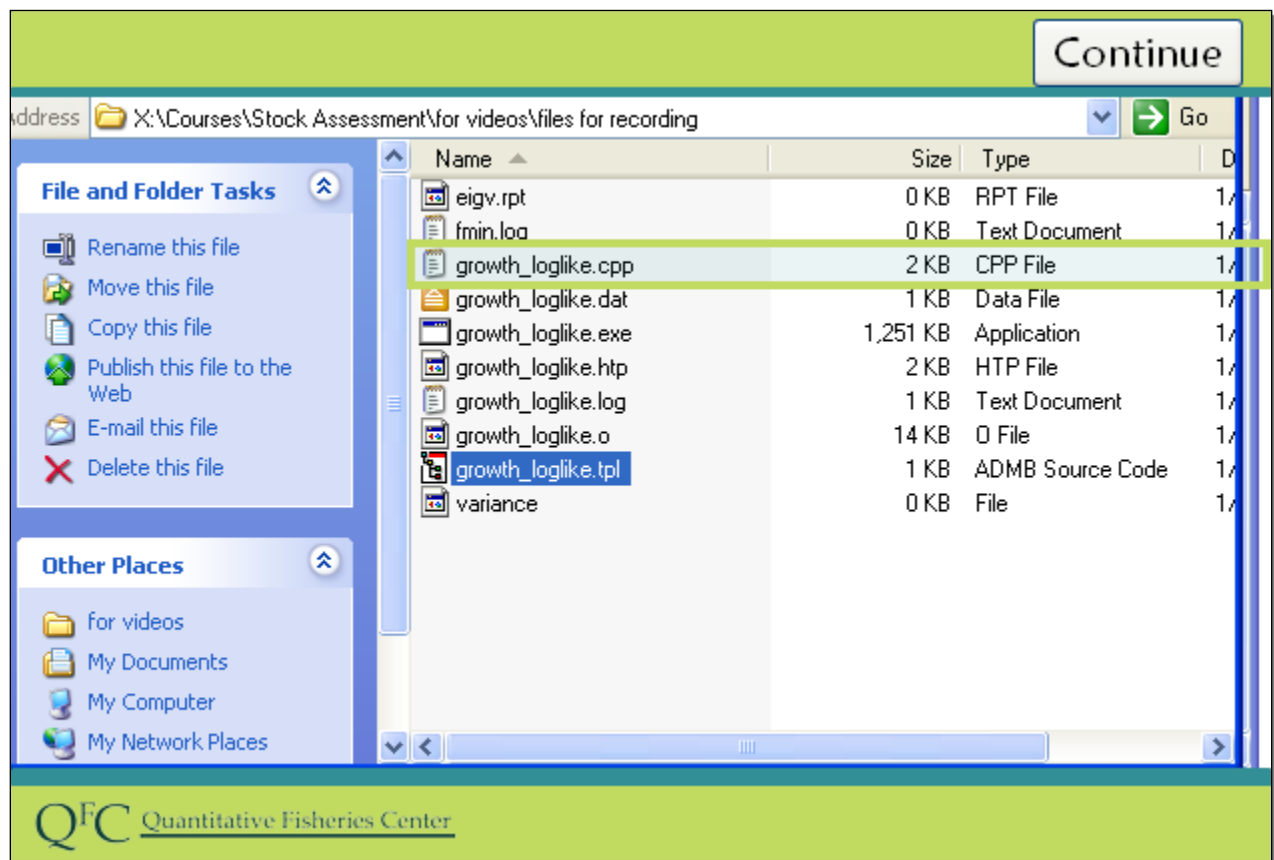
Translate, Compile and Link



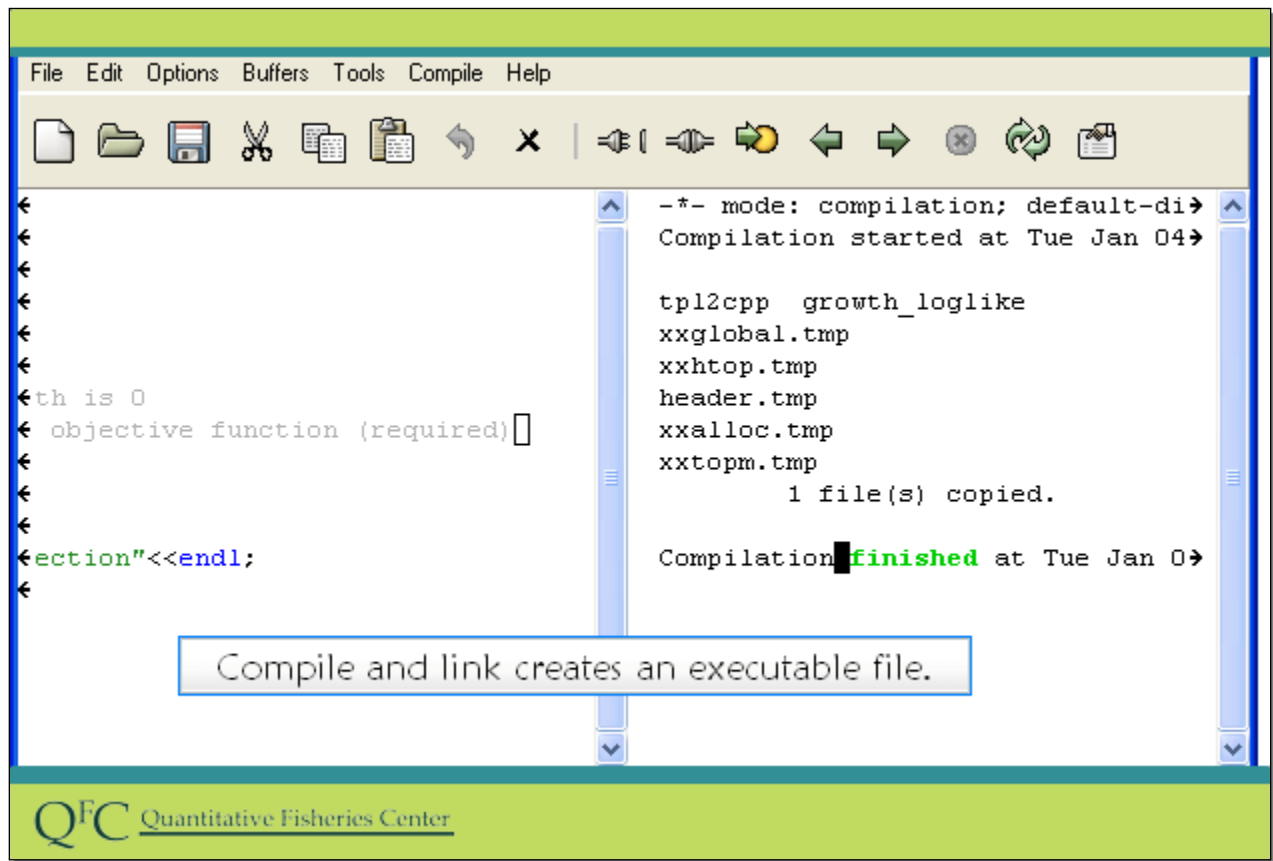
Our next step is to translate the tpl to real c++ code. We can accomplish this by clicking on the translate button.



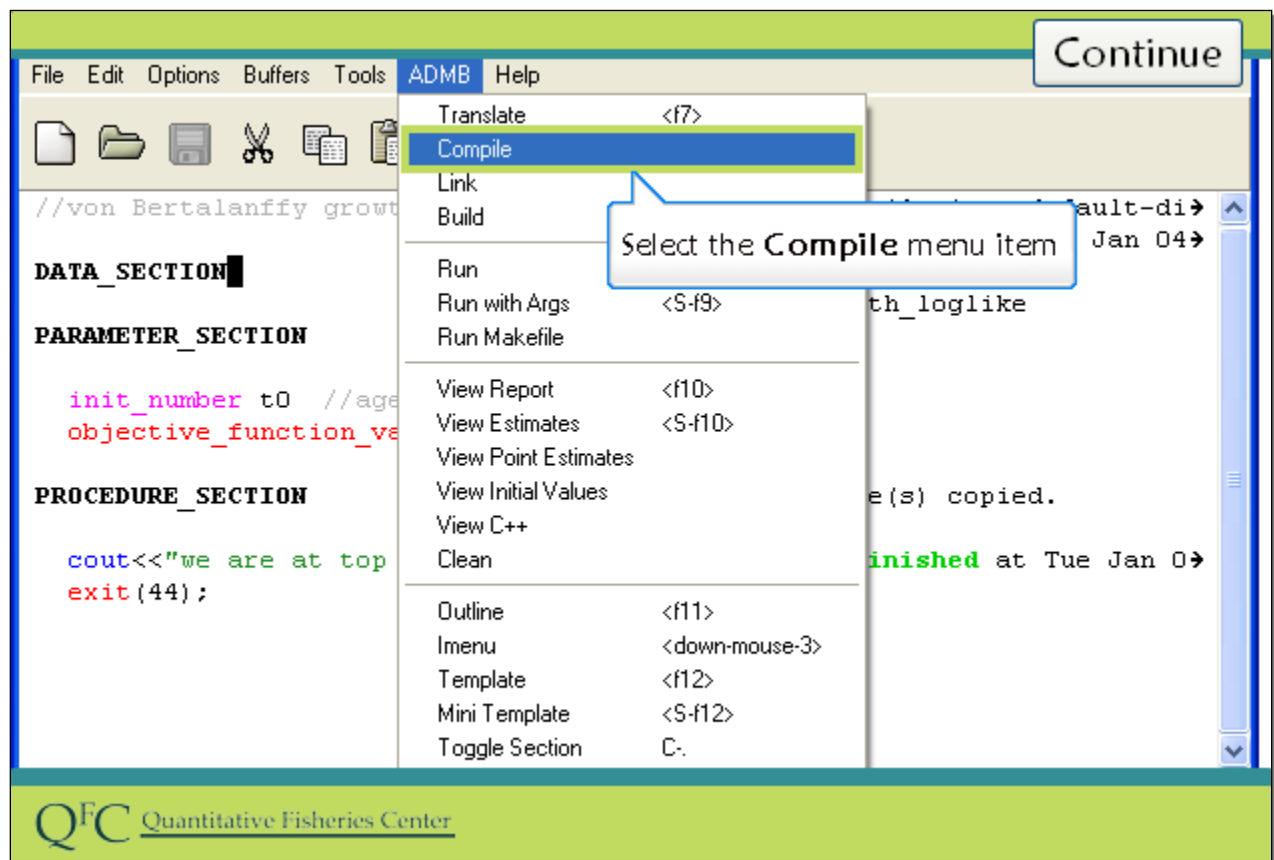
We should see a bunch of messages in a new buffer ending with “compilation finished”.



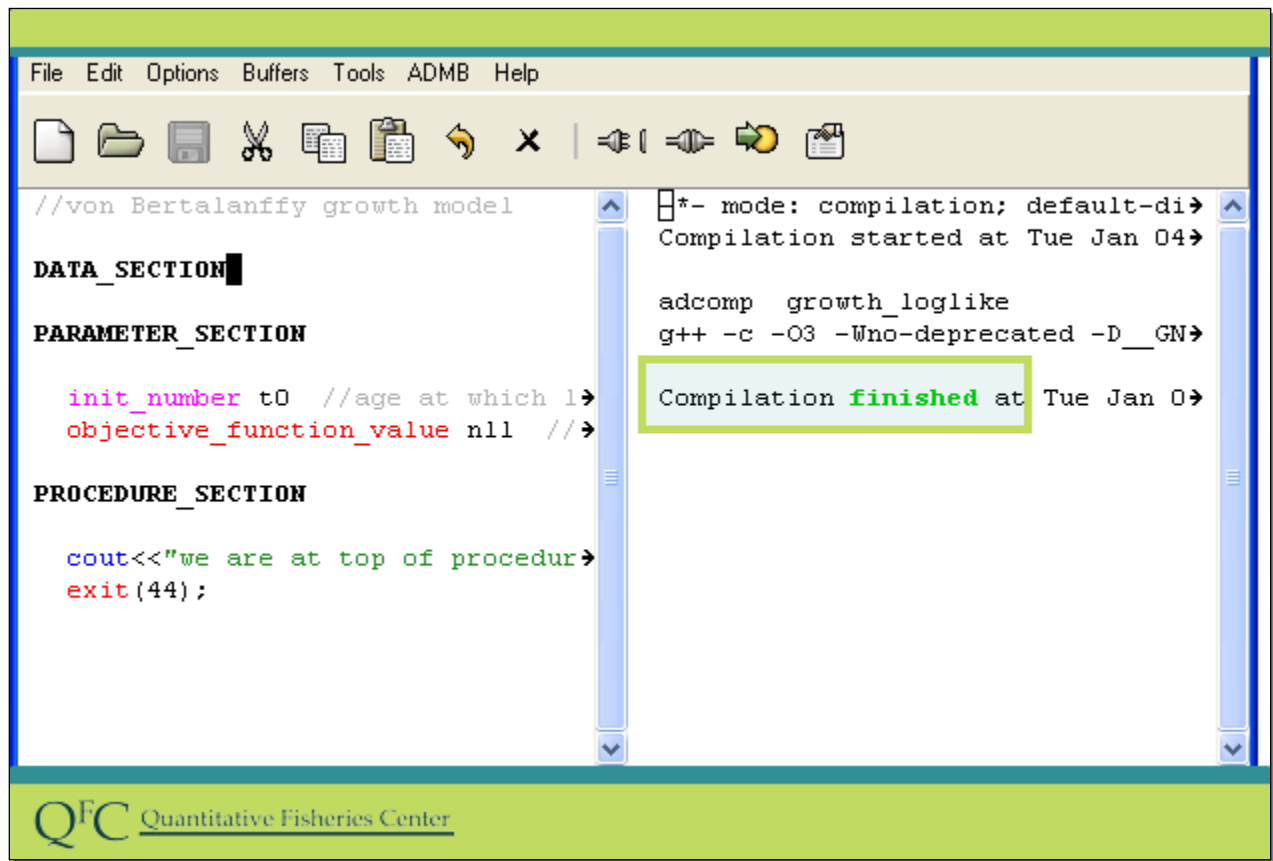
If we were to look in our directory where the tpl is saved we would see that growth_loglike.cpp now exists. We next compile and link the c++ code to create our working program.



For most purposes we can think of the compile and link steps together as a single operation that creates an executable file `growth_loglike.exe` based on our C++ code.



To compile click on the admb tab and then on “compile”.



The screenshot shows the ADMB software interface. The left pane displays a C++ code file named `//von Bertalanffy growth model`. The code is organized into sections: **DATA_SECTION**, **PARAMETER_SECTION**, and **PROCEDURE_SECTION**. The **PARAMETER_SECTION** contains two parameters: `init_number` (type `t0`, comment `//age at which l`) and `objective_function_value` (type `nll`, comment `//`). The **PROCEDURE_SECTION** contains a `cout` statement and an `exit(44);` statement. The right pane shows the compilation output. It starts with `*- mode: compilation; default-di` and `Compilation started at Tue Jan 04`. The compiler command is `adcomp growth_loglike` followed by `g++ -c -O3 -Wno-deprecated -D__GN`. The output ends with `Compilation finished at Tue Jan 0`, which is highlighted with a green box. The bottom of the window features the Quantitative Fisheries Center logo and name.

```
//von Bertalanffy growth model

DATA_SECTION

PARAMETER_SECTION

    init_number t0 //age at which l
    objective_function_value nll //

PROCEDURE_SECTION

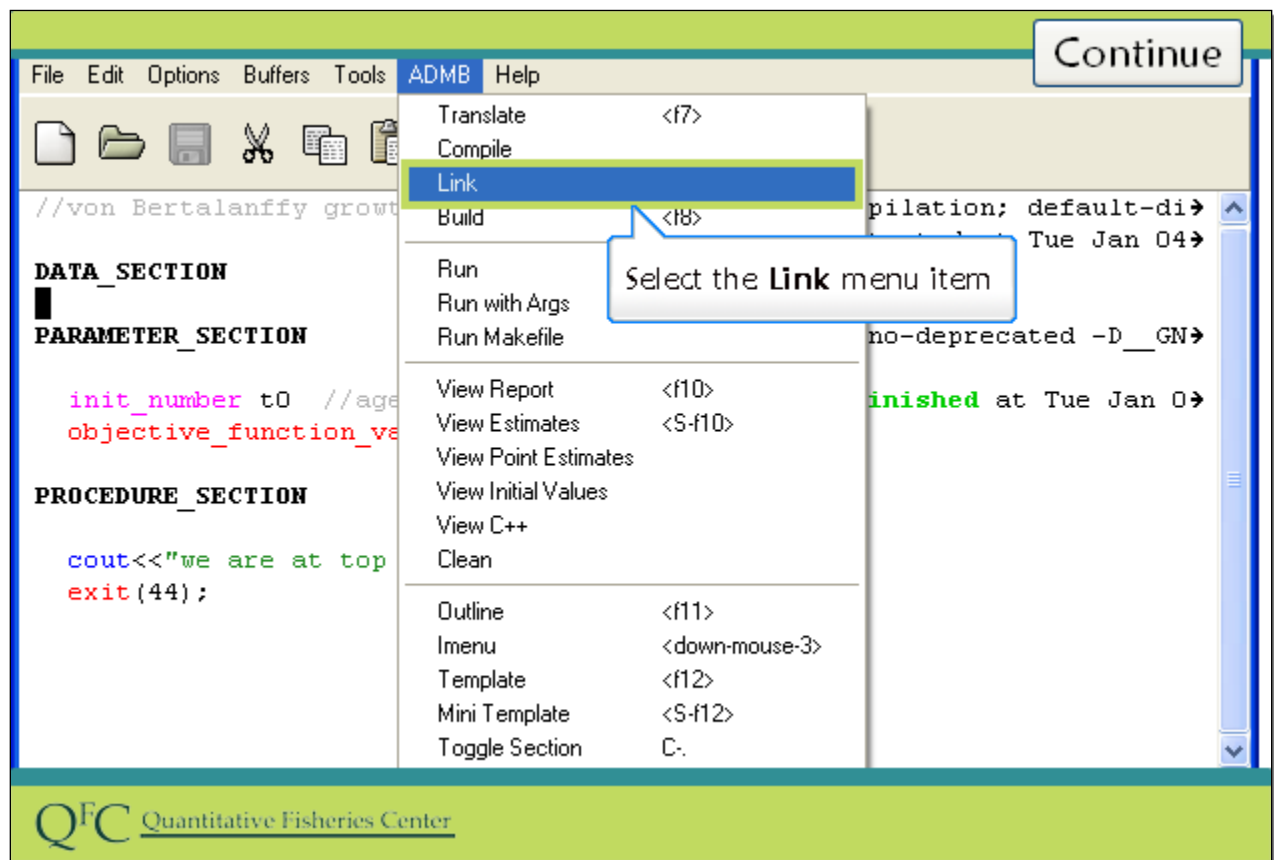
    cout<<"we are at top of procedur
    exit(44);
```

```
*- mode: compilation; default-di
Compilation started at Tue Jan 04

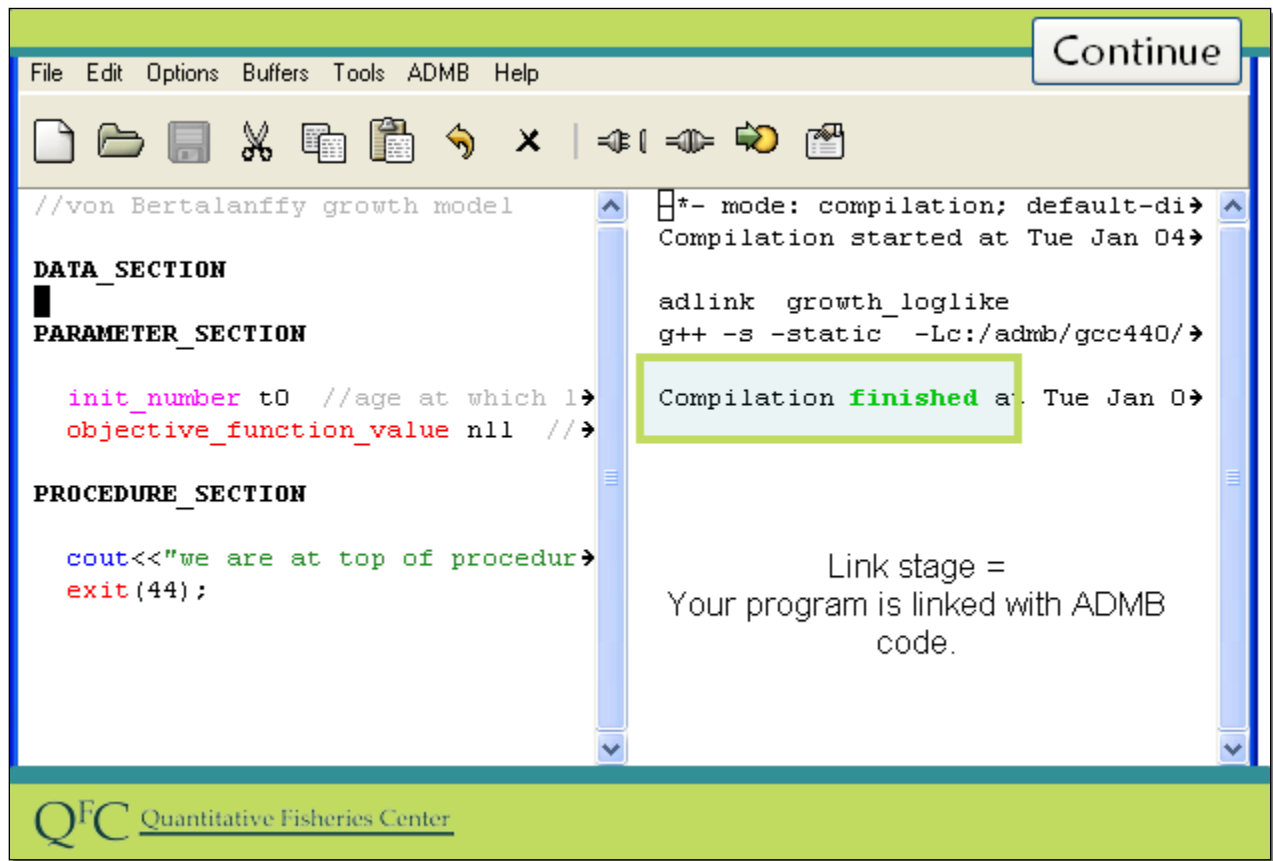
adcomp growth_loglike
g++ -c -O3 -Wno-deprecated -D__GN

Compilation finished at Tue Jan 0
```

You should see messages going to a new buffer ending with “compilation finished”.

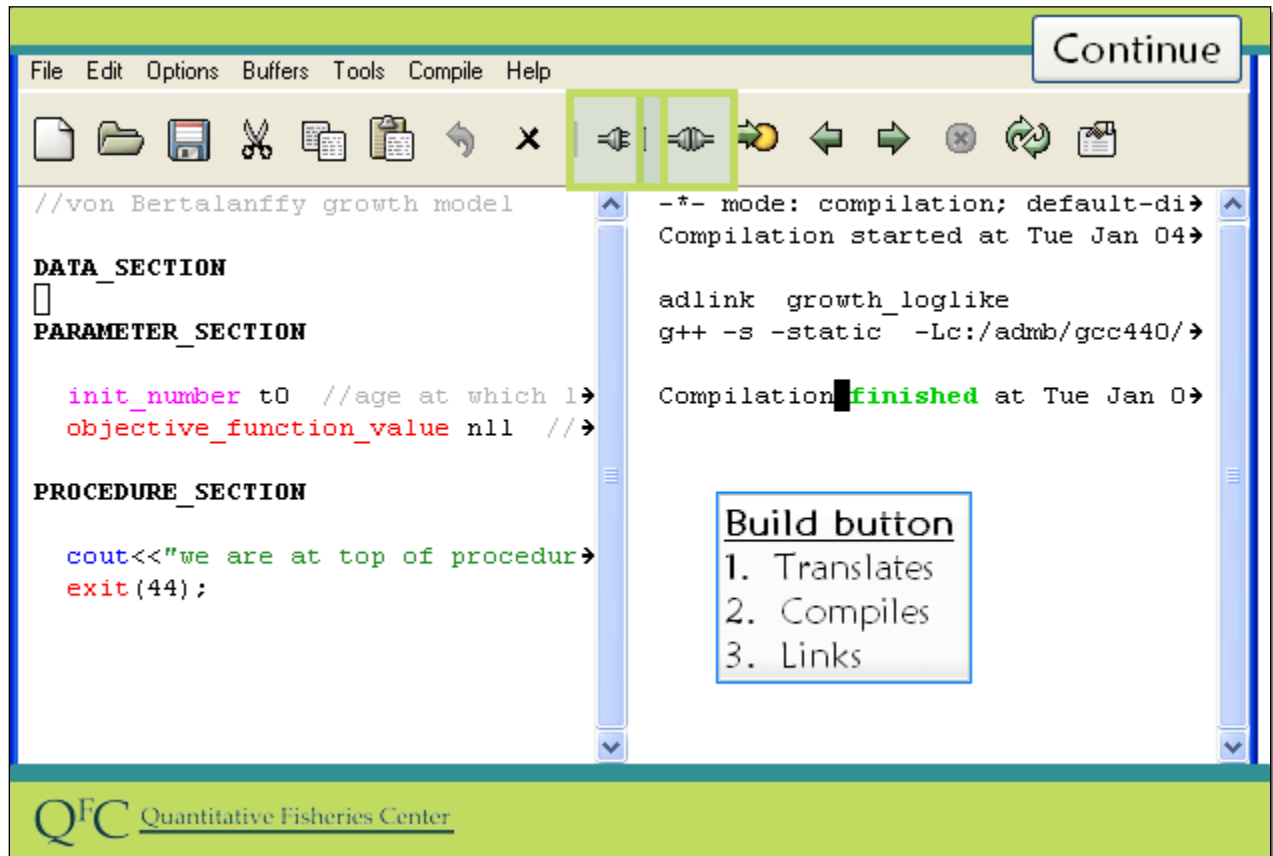


Now go to the admb tab again and select "link".

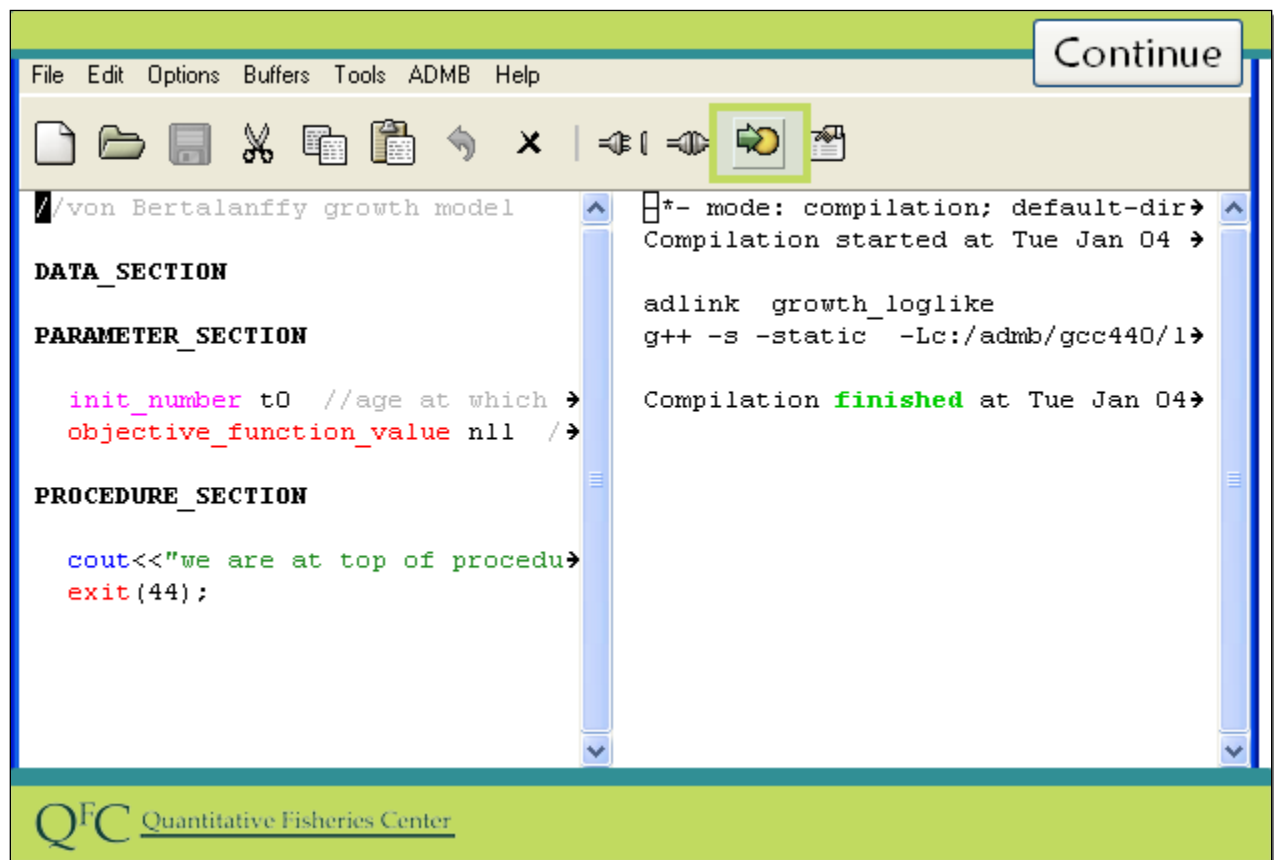


Again you will see messages ending with “compilation finished.” What actually happens in the link stage is that code that was pre-compiled into libraries as part of your admb installation is merged with your program. This is useful to know about in case you get errors at the link stage. This usually means that the compiler is having trouble finding the admb libraries, and this would mean there is a problem with your admb installation.

Build and Run

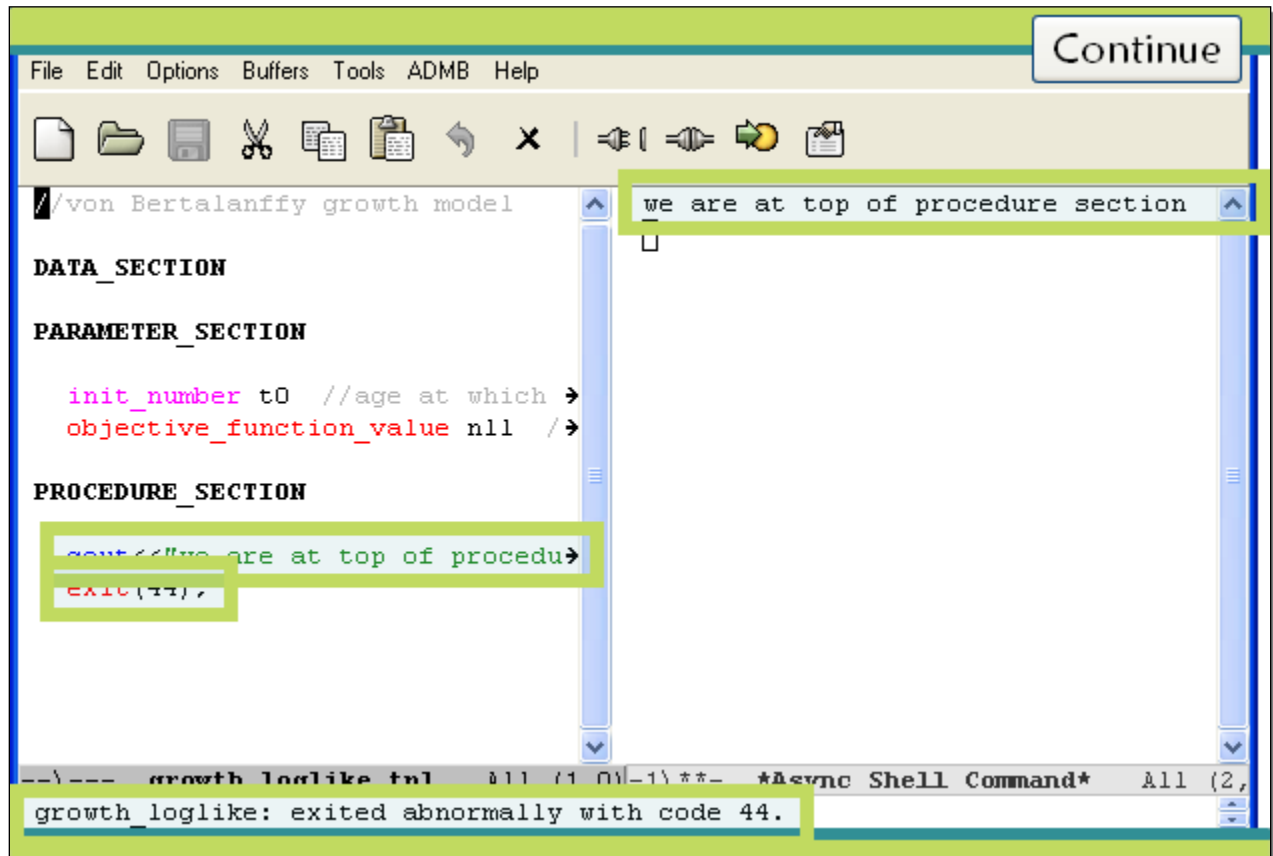


As you saw above you can use a button to translate a template to c plus-plus code but you used the drop down menu to compile and link. There is also a build button that simultaneously translates, compiles, and links. This can be convenient but be careful if you do this to check messages and make sure that the translate stage worked. If it did not than compile and link may have worked with an earlier version of the c-plus-plus code!



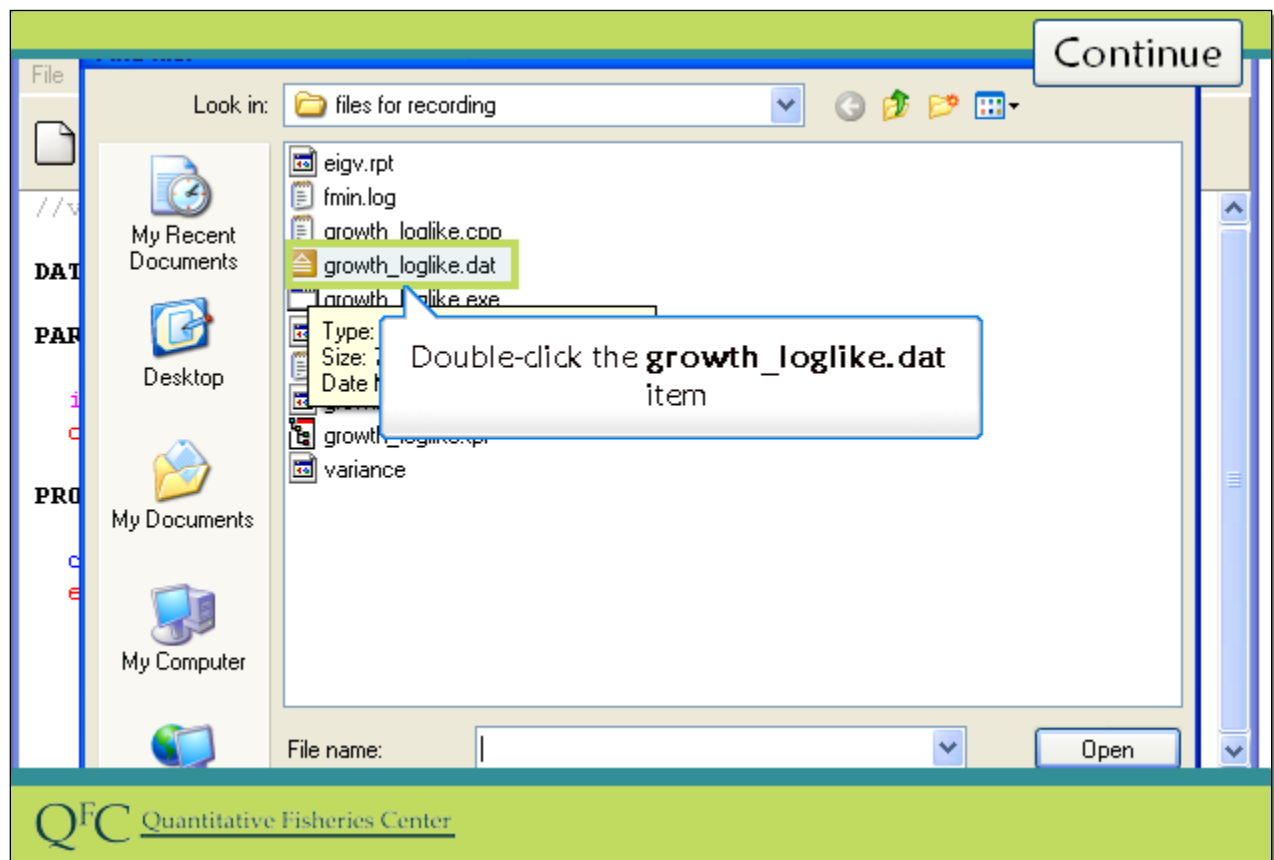
OK now for the moment of truth. We will run our program. We can do this by clicking on the run button.

Output

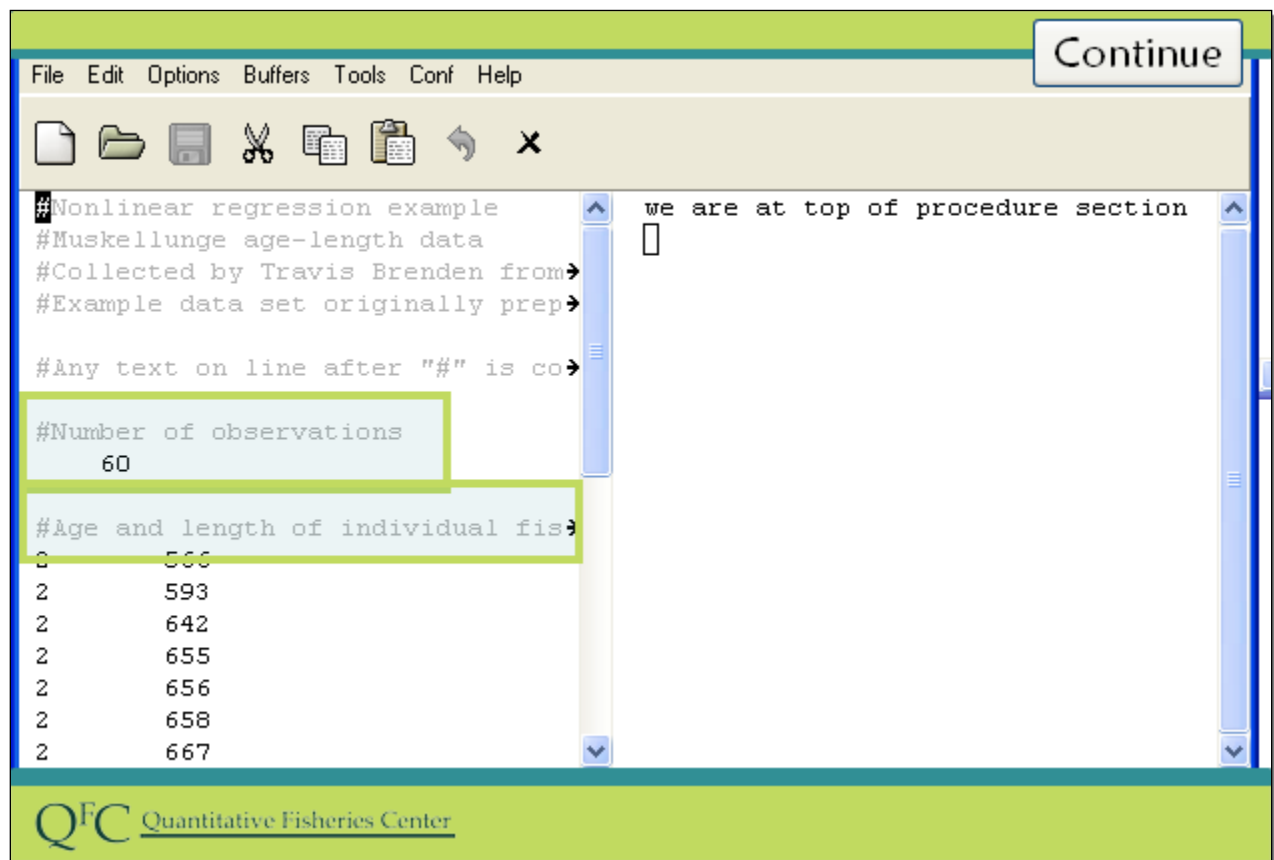


When you do this you should see the text from your cout command get written to the screen in one buffer. And in another window, likely a small one at the bottom of your screen, you will have the message: growth_loglike: exited abnormally with code 44.

So our program worked, wrote our test text out and quit where we wanted it to. So now we need to go back and get it to actually read our data in! First let us do a quick review of our dat file.



Use file, to open the growth_loglike.dat file.



```
#Nonlinear regression example
#Muskellunge age-length data
#Collected by Travis Brenden from
#Example data set originally prepared by Travis Brenden

#Any text on line after "#" is considered a comment

#Number of observations
60

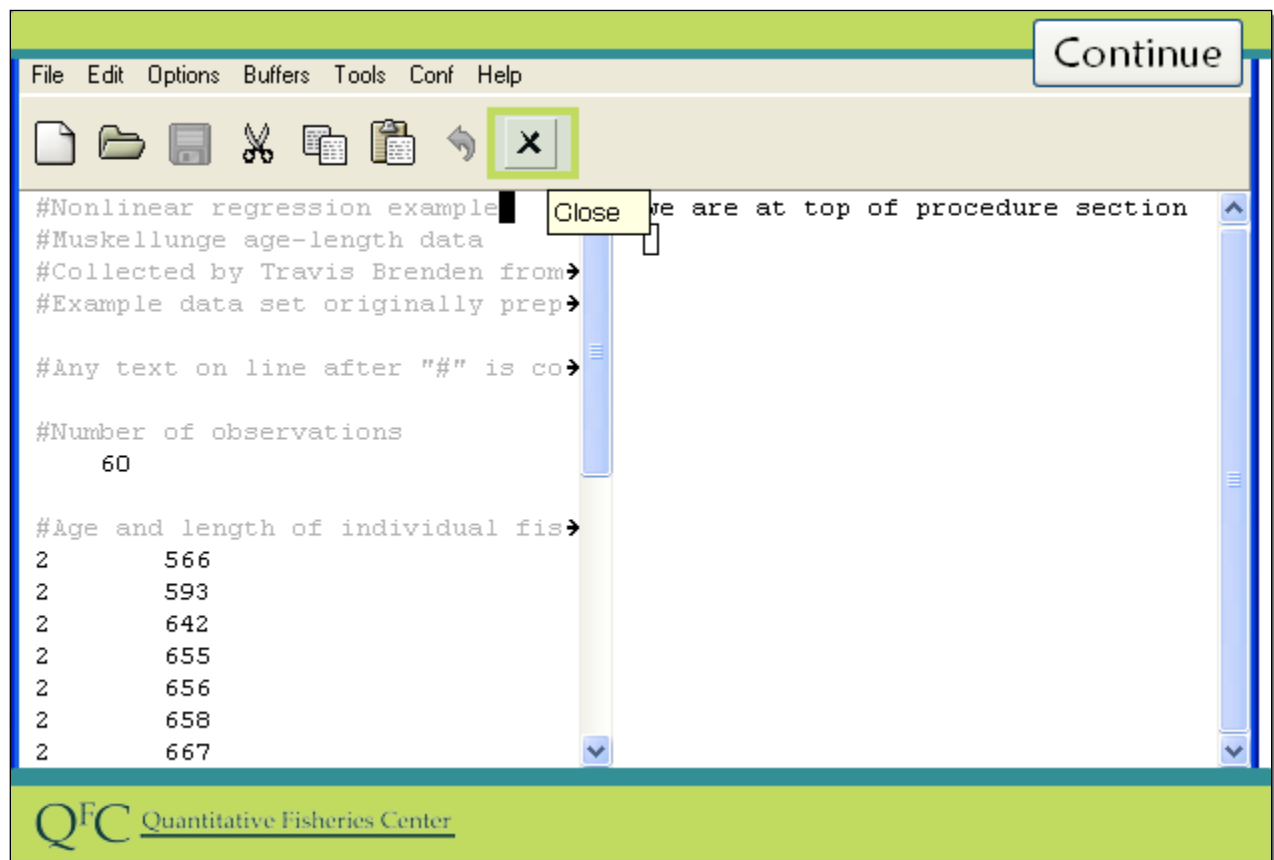
#Age and length of individual fish
0      565
2      593
2      642
2      655
2      656
2      658
2      667
```

we are at top of procedure section

QFC Quantitative Fisheries Center

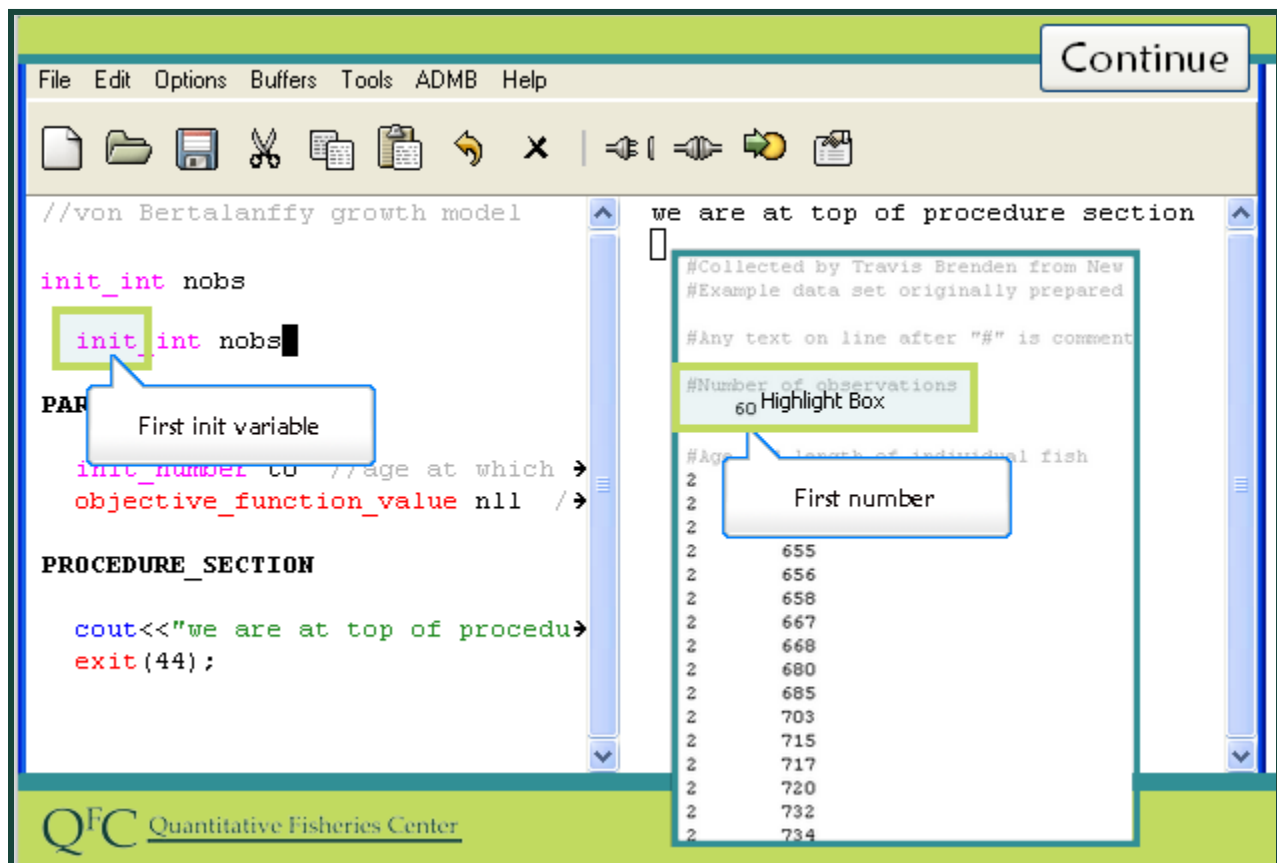
This refreshes our memory that the first thing we read in is the number of observations, followed by a table of ages and lengths where each row is an observed fish and the first column is the age and the second the length.

At the end we have a vector of three test values.



We return to our tpl buffer and go to the data section.

Read in Data and Test Vec, Add Cout and Exit Statements



First we add a line that creates a variable called `nobs` and tells our `admb` program to read this from the `dat` file. Here `init` has a different meaning than in the parameter section. It defines variable types that will be read from the `dat` file. `init_int` means create an integer type variable and read it from the `dat` file. This is the first `init` variable in the data section so the first number will be read which will be 60, the number of observations. Like the parameter section the code you write in the data section gets translated into real C++ code so you do not need a semi-colon after this line.

Slide Code:

```
init_int nobs
```

On one line for easier typing
`init_matrix vonBdata(1,nobs,1,2)`

The number of rows is equal to the number of observations in our dat file

Two colums in the data (Age and Length)

are at top of procedure section

`vonBdata ≠ vonbdata`

`vonBdata(5,2)`

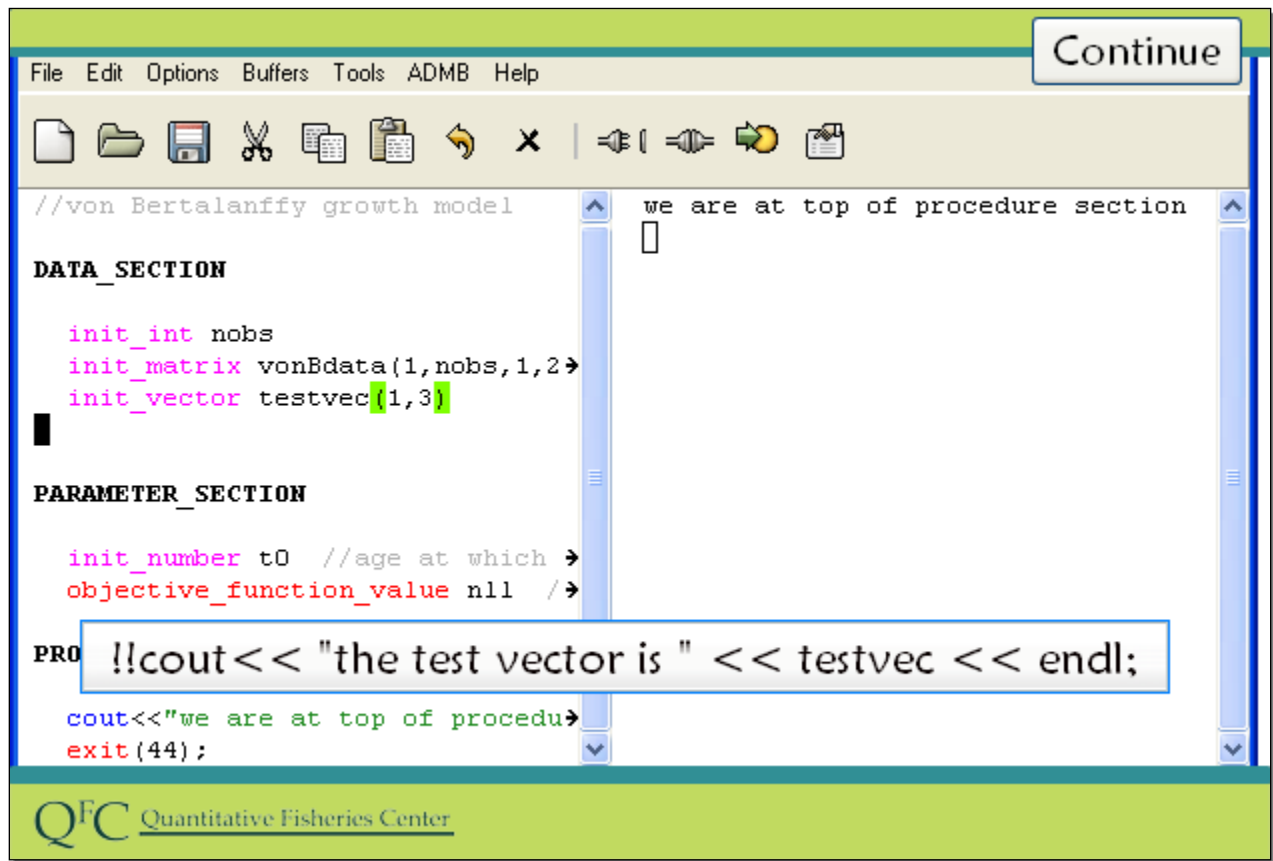
#Age and length of individual fish	
2	566
2	593
2	642
2	655
2	656
2	658
2	667
2	668
2	680
2	685
2	703
2	715
2	717
2	720
2	732
2	734

Now we are ready to read in the table of ages and lengths. It is convenient to read this into a single matrix variable. Later on we can extract the ages and lengths into separate vectors. We read in the matrix of data with this line:

We have created a matrix variable `vonBdata`. Keep in mind that `admb` programs are case sensitive with respect to variable names so when you refer to this variable the `B` will need to be capitalized. The four numbers within the matrix define the minimum and maximum row indices, followed by the minimum and maximum column indices for the matrix `vonBdata`. Often we define the minimum index for rows or columns as 1. Later on if we refer to say `vonBdata(5,2)` then the `admb` program will know we are talking about the 5th row and second column of the matrix.

Slide Code:

```
init_matrix vonBdata(1,nobs,1,2)
```



The screenshot shows the ADMB software interface. The menu bar includes File, Edit, Options, Buffers, Tools, ADMB, and Help. The toolbar contains icons for file operations and execution. The main window displays a C++ code file for a von Bertalanffy growth model. The code is organized into sections: DATA_SECTION, PARAMETER_SECTION, and a procedure section. A callout box highlights the following line of code:

```
!!cout<< "the test vector is " << testvec << endl;
```

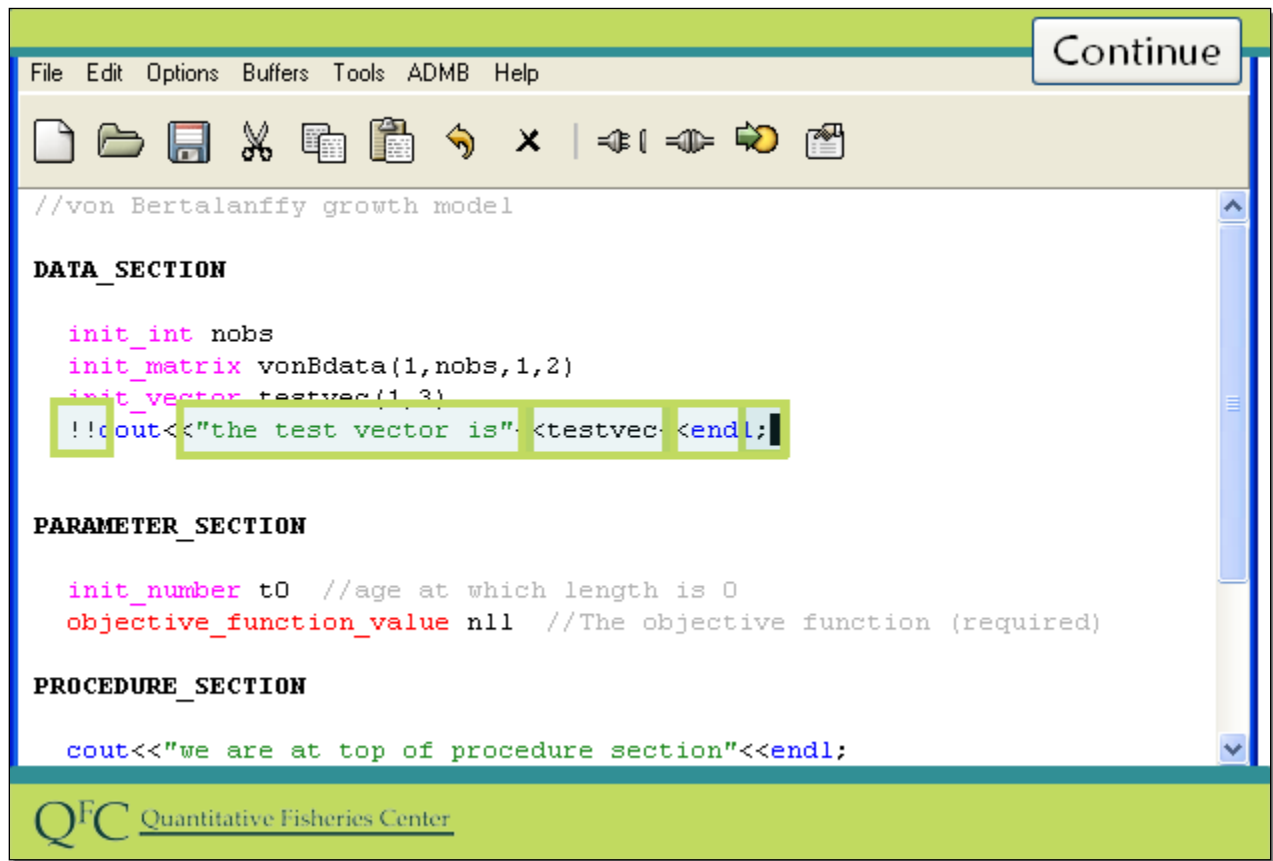
The bottom of the window features the Quantitative Fisheries Center logo and name.

We end reading in our data by reading in the test vector. We can do this by adding this line:
So now to test if things seem to work ok we need to write out the test vector using the cout command.

Slide Code:

```
init_vector testvec(1,3)
```

```
!!cout<< "the test vector is " << testvec << endl;
```



```
//von Bertalanffy growth model

DATA_SECTION

    init_int nobs
    init_matrix vonBdata(1,nobs,1,2)
    init_vector testvec(1,3)
    !!cout<<"the test vector is"<<testvec<<endl;

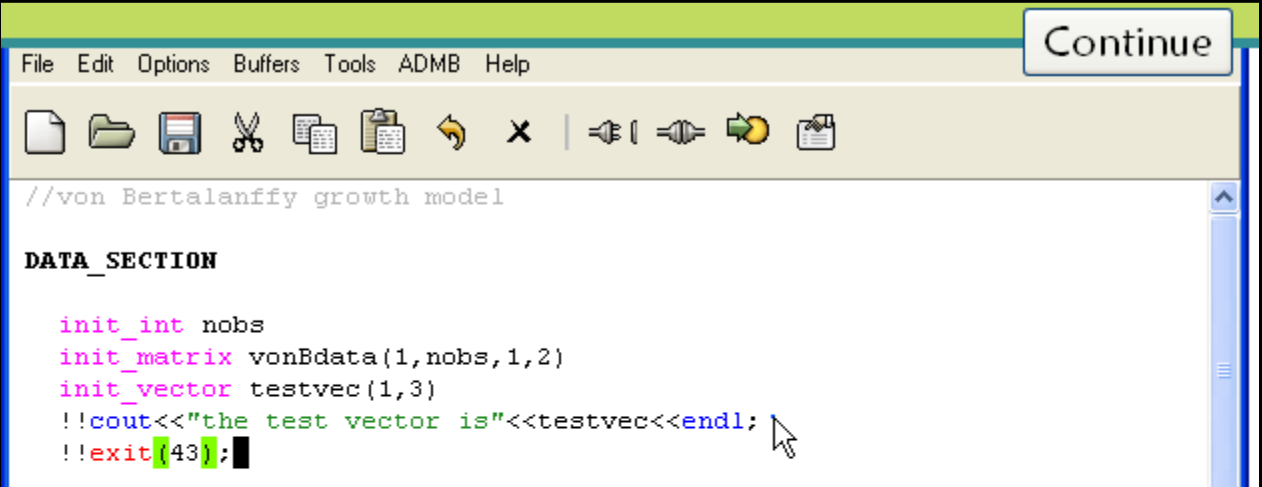
PARAMETER_SECTION

    init_number t0 //age at which length is 0
    objective_function_value nll //The objective function (required)

PROCEDURE_SECTION

    cout<<"we are at top of procedure section"<<endl;
```

The syntax is just like the cout command we used in the procedure section, but now we start the line with two exclamation marks. These tell our admB program to not try to translate this line into C++ code, using parsing rules it knows about for the data section. Instead, this line gets added directly to the resulting C++ code, as is. Because what follows the exclamation marks is assumed to already be C++ code and is not translated, it has to end with a semi-colon. That is a rule for C++ commands. When executed, our cout command will first write out the text in quotes, then the actual value of “testvec” and then a line feed.



File Edit Options Buffers Tools ADMB Help

Continue

```
//von Bertalanffy growth model

DATA_SECTION

init_int nobs
init_matrix vonBdata(1,nobs,1,2)
init_vector testvec(1,3)
!!cout<<"the test vector is"<<testvec<<endl;
!!exit(43);

PARAMETER_SECTION

init_number
objective

PROCEDURE_SECTION
```

Normally the admB translation program assumes that lines in the data section start with admB key words. cout is a C++ command and not an admB key word. !! signals the translate program that the line is already C++ code and not something it should try to translate into C++.

It is great that the translation program can convert simple commands starting with key words into more complex C++ code. For example, the key word init_vector in the data section tells the translate program to write code to create the vector and read its values from a file. However, being able to also include lines of C++ code in the data section gives you additional capabilities.

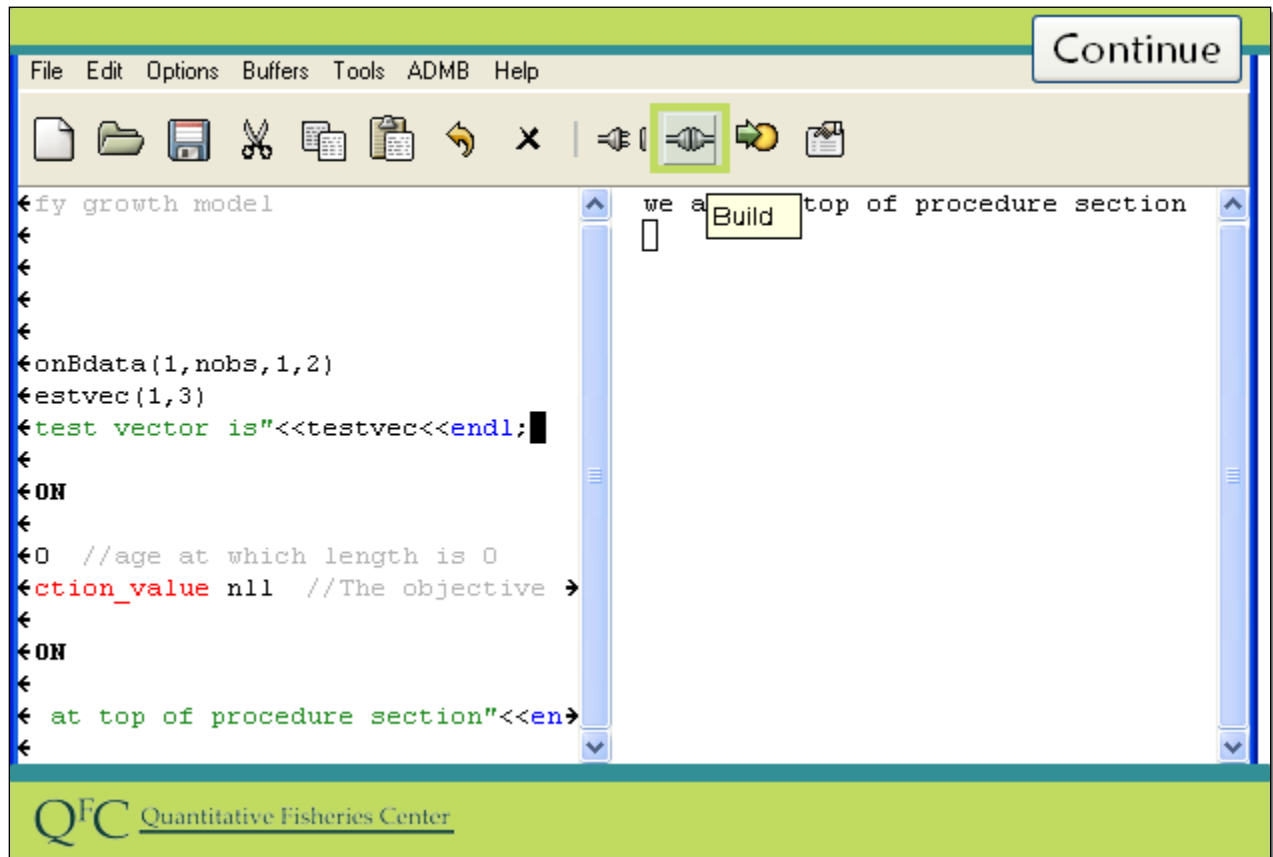
QFC Quantitative Fisheries Center

We follow this line with an exit command. Again notice we start it with two exclamation marks and end it with a semi-colon. We use a different exit code so we know our program stopped here.

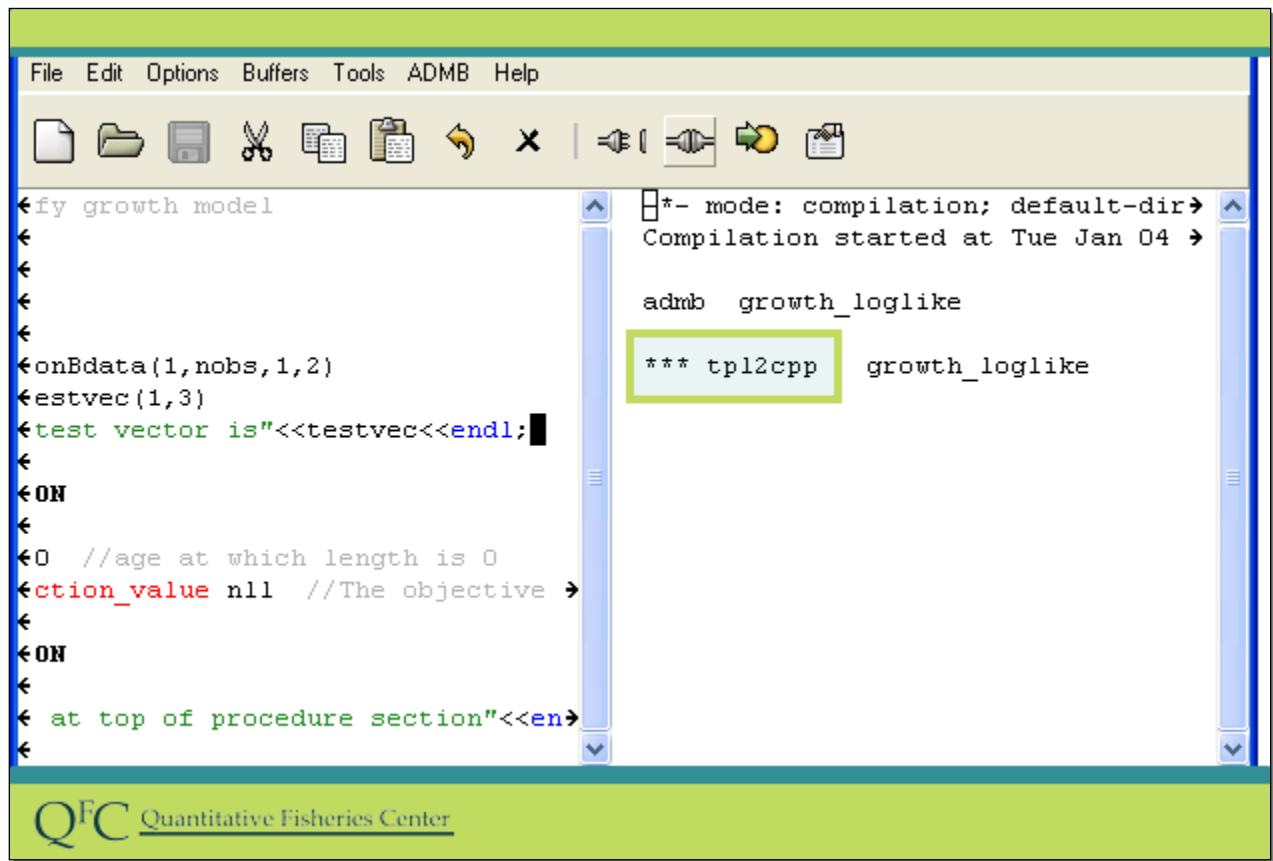
Slide Code:

```
!!exit(43);
```

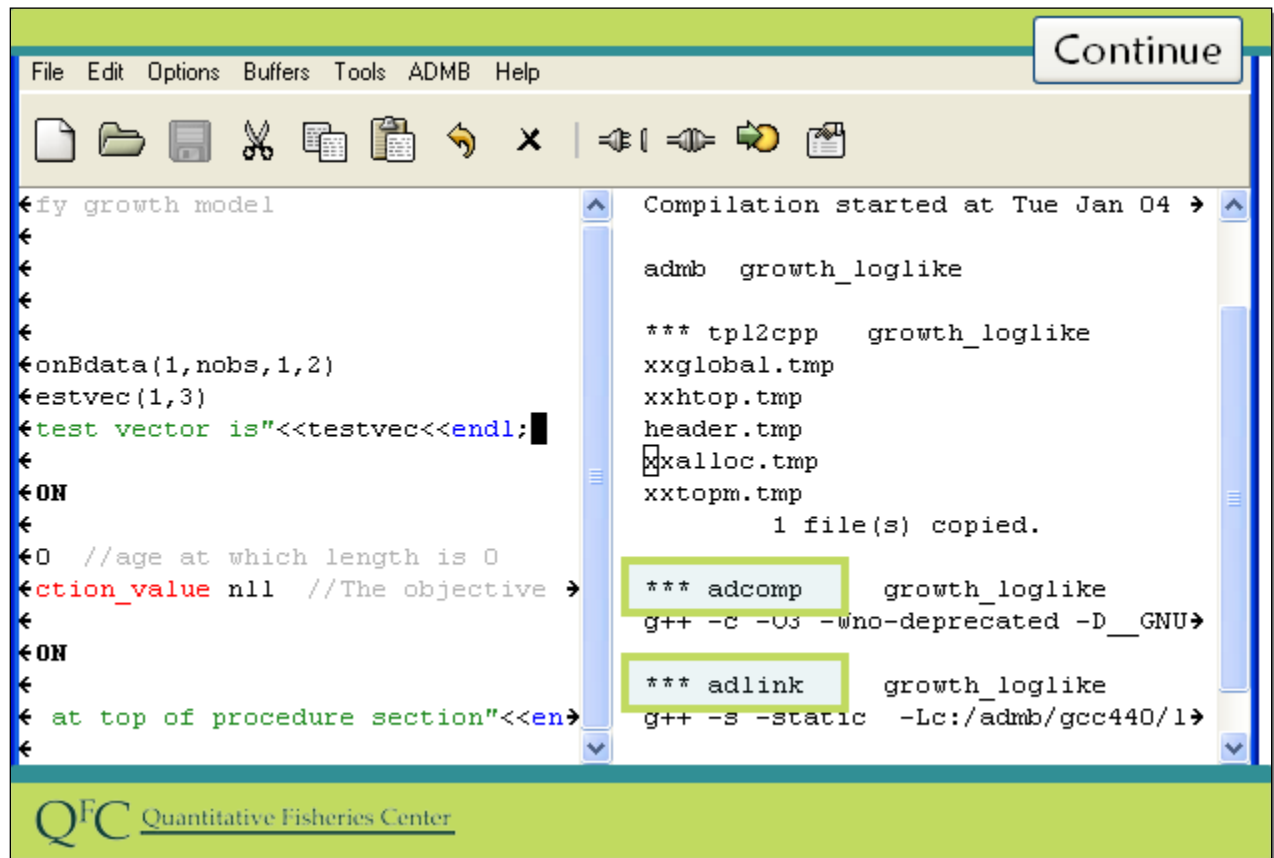
Run Again with Test Vec and Data



Now we are ready to test if our program is working right. This time we will translate, compile, and link in one step using the build button.



When you do this messages for all three steps get sent to a buffer sequentially. The translate messages start with: *** tpl2cpp,



The screenshot shows the ADMB software interface. The top menu bar includes File, Edit, Options, Buffers, Tools, ADMB, and Help. A toolbar with various icons is located below the menu. The main window is split into two panes. The left pane displays a C++ code file named 'growth_model'. The code includes comments and function calls like 'onBdata', 'estvec', and 'test vector is'. The right pane shows the compilation output, starting with 'Compilation started at Tue Jan 04'. It lists the files being compiled: 'admb growth_loglike', '*** tpl2cpp growth_loglike', 'xxglobal.tmp', 'xxhtop.tmp', 'header.tmp', 'xalloc.tmp', and 'xxtopm.tmp'. It then reports '1 file(s) copied.' and shows the compilation commands: '*** adcomp growth_loglike' followed by 'g++ -C -O3 -Wno-deprecated -D__GNU__' and '*** adlink growth_loglike' followed by 'g++ -s -static -Lc:/admb/gcc440/1'.

```
File Edit Options Buffers Tools ADMB Help
Continue

<fy growth_model
<
<
<
<onBdata(1,nobs,1,2)
<estvec(1,3)
<test vector is"<<testvec<<endl;
<
<ON
<
<0 //age at which length is 0
<ction_value nll //The objective
<
<ON
<
< at top of procedure section"<<en>
<
```

```
Compilation started at Tue Jan 04

admb growth_loglike

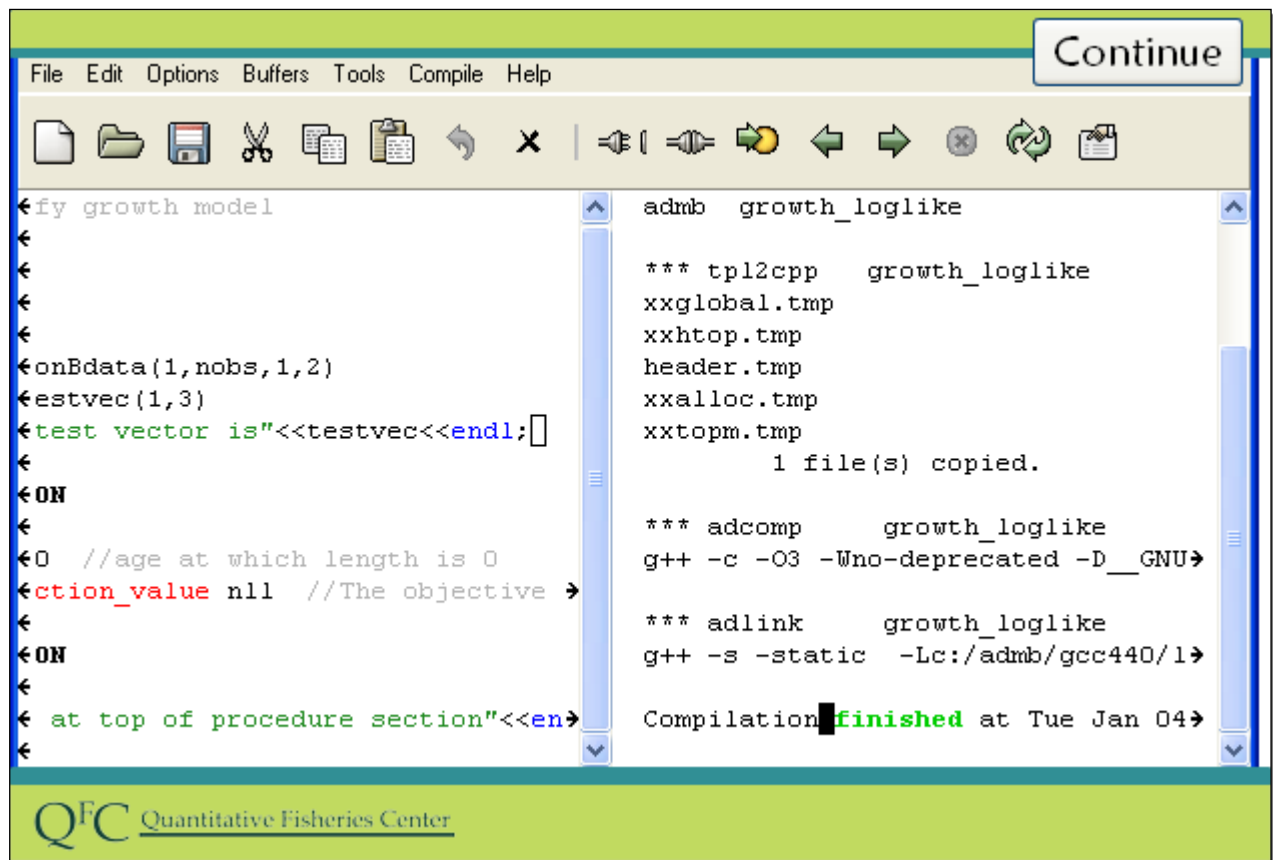
*** tpl2cpp growth_loglike
xxglobal.tmp
xxhtop.tmp
header.tmp
xalloc.tmp
xxtopm.tmp
1 file(s) copied.

*** adcomp growth_loglike
g++ -C -O3 -Wno-deprecated -D__GNU__

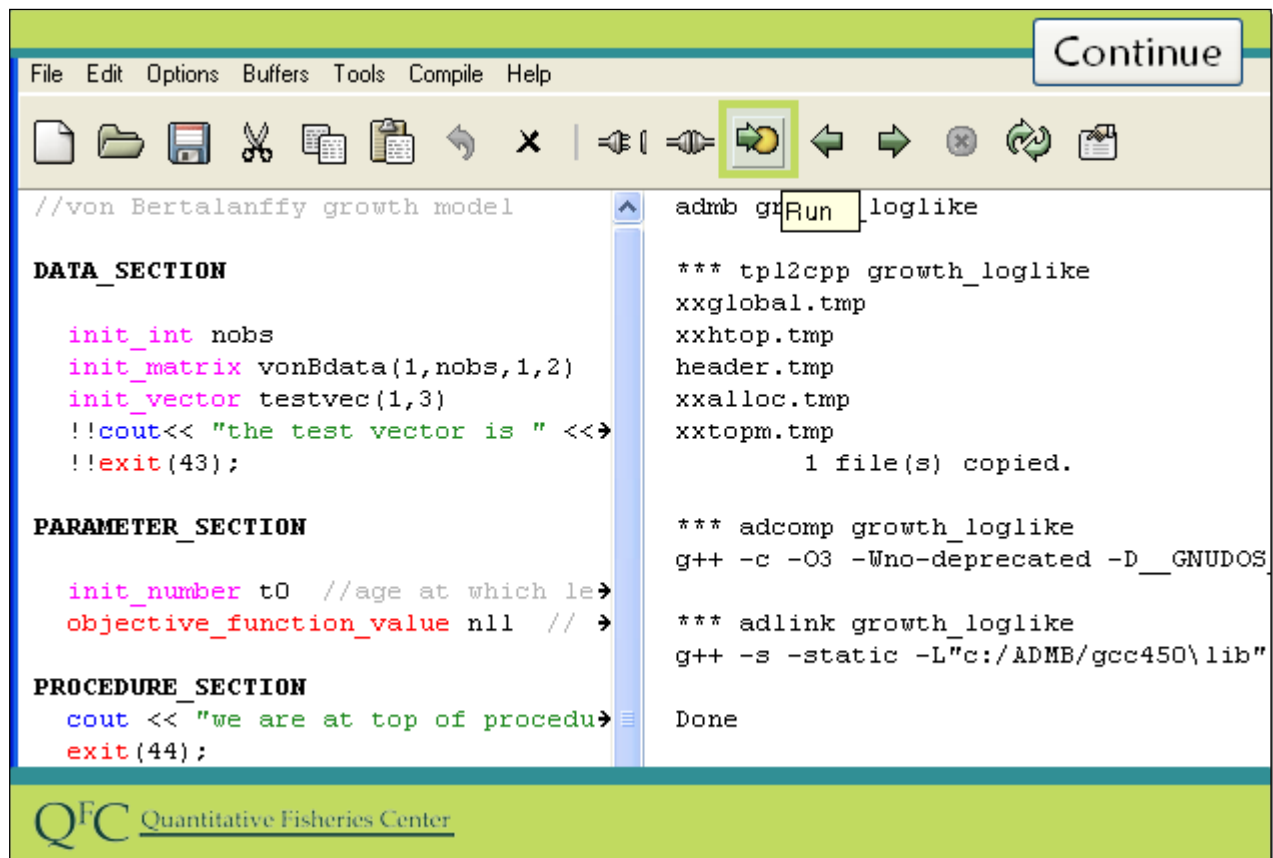
*** adlink growth_loglike
g++ -s -static -Lc:/admb/gcc440/1
```

Q^{FC} Quantitative Fisheries Center

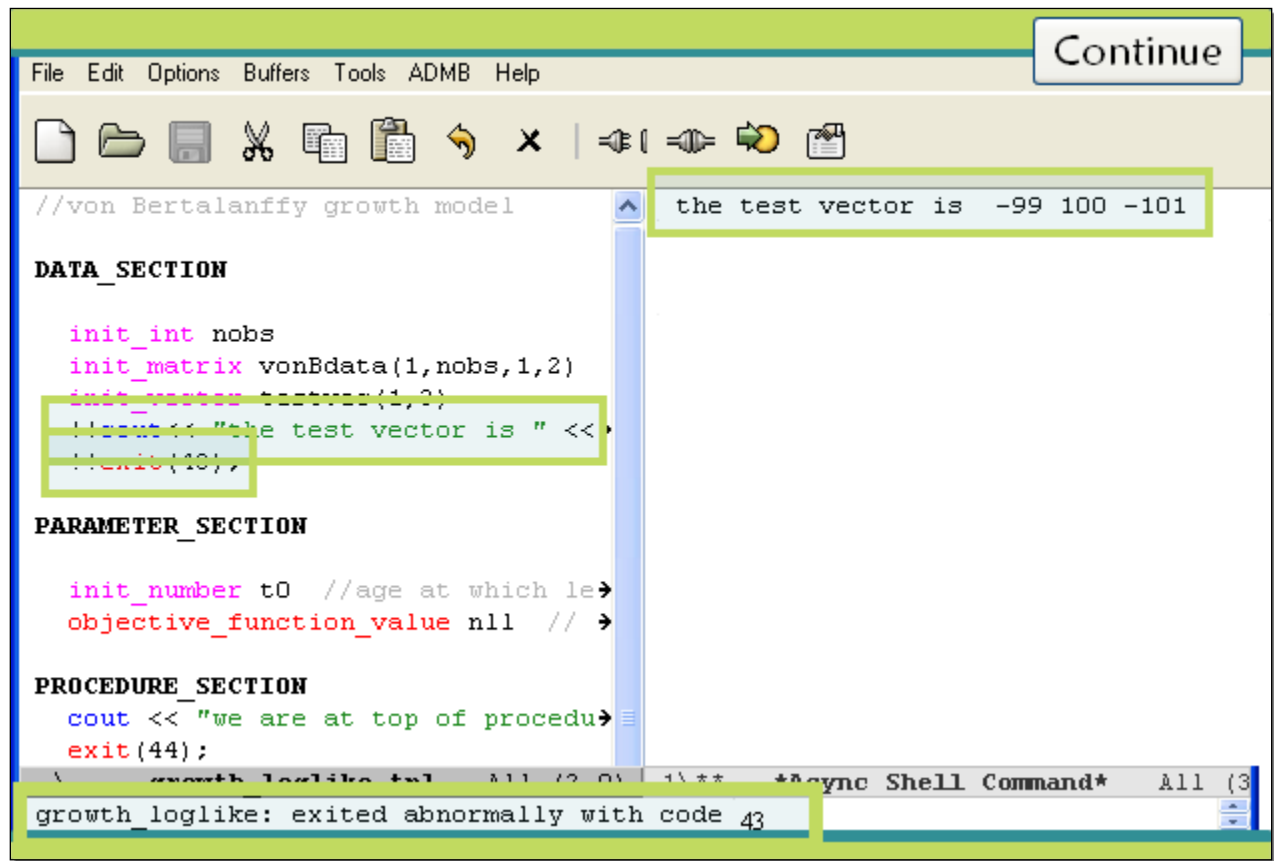
the compile messages with *** adcomp and the link messages with *** adlink.



Be sure that there are no translate error messages or you will be just be compiling and linking your previous version!



Now we can run the new version.



The screenshot shows the ADMB software interface. The main window displays a C++ code file named `//von Bertalanffy growth model`. The code is organized into three sections: **DATA_SECTION**, **PARAMETER_SECTION**, and **PROCEDURE_SECTION**. In the **DATA_SECTION**, the following lines are highlighted with green boxes:

```
init_int nobs
init_matrix vonBdata(1,nobs,1,2)
init_vector testvec(1,2)
cout << "the test vector is " <<
exit(43);
```

In the **PROCEDURE_SECTION**, the following line is highlighted with a green box:

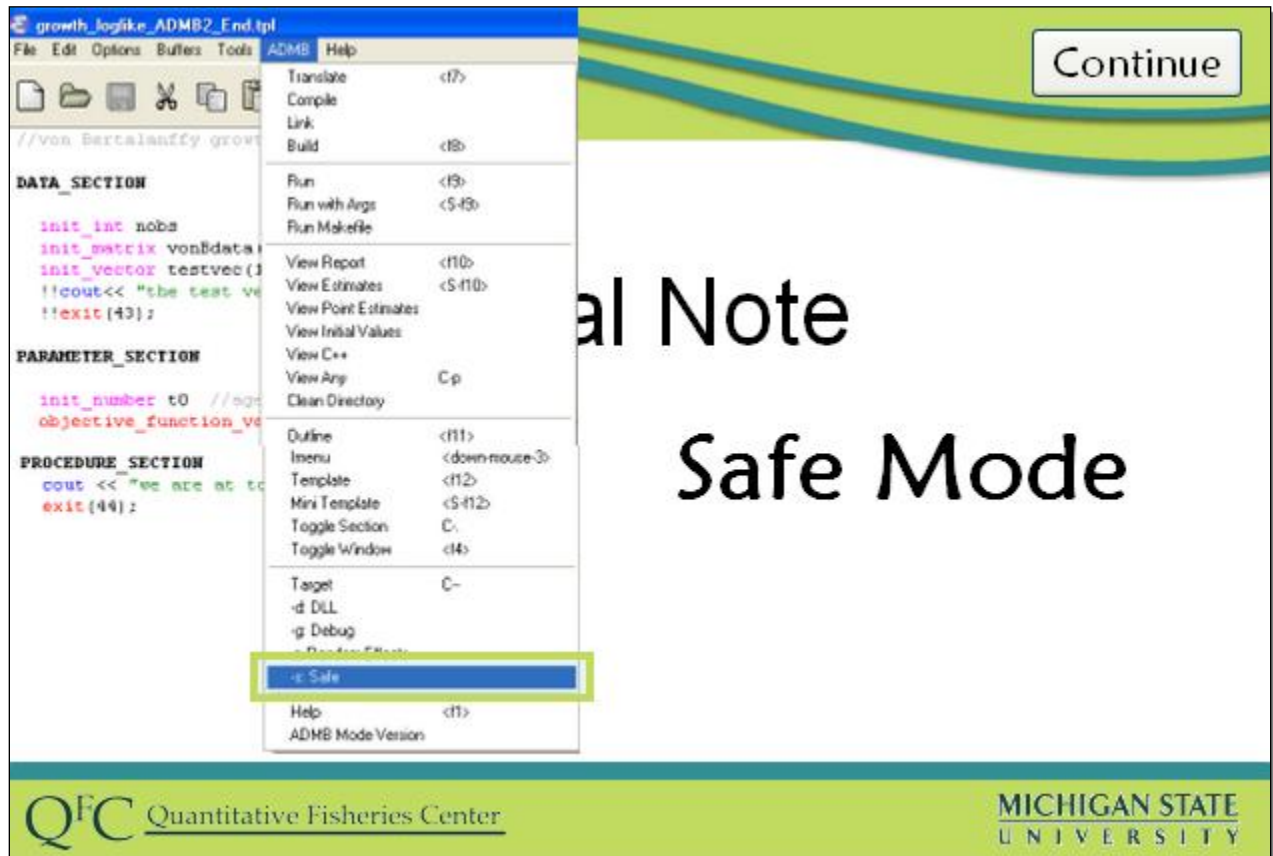
```
cout << "we are at top of procedu";
```

The output window on the right shows the text: `the test vector is -99 100 -101`. The status bar at the bottom indicates: `growth_loglike: exited abnormally with code 43`.

As we should, we get our test numbers of minus 99, 100 and minus 101 printed to a buffer and our exit code of 43 to another.

So it looks like this worked. We know our program read the right number of observations and ended in the right place in the dat file. This is not fool proof, however. For example we could read different objects in the wrong order and if we read the right number of numbers then we will get the right test vector. In this case we only read the number of observations and the matrix of data and we use the number of observations in reading the matrix so we really cannot do this in the wrong order without getting an error. But in more complicated programs it is often useful to cout the various data we are reading in (or portions of it if they are long) to make sure things are read in correctly.

Safe Mode



As a side note - By default the version of ADMB-IDE we are using the so called-optimized ADMB libraries. In these videos we use these libraries except in the video where we demonstrate the use of the safe mode at the end of these series of videos. Some experienced ADMB users advocate always using the safe mode except when working with debugged programs that really need the added speed of the optimized mode. You can switch to the safe mode by clicking on dash s, safe under the target section of the admdb menu before you build your program - - that is, before you translate, compile, and link. You can learn more about safe mode by viewing the ADMB - Safe Mode video. The benefits of using safe mode will be clearer and the video on this mode will be more understandable after you have learned about loops, which is covered in a later video series.