

## Lab Exercises wk05 – More Debugging with NetBeans

### Required Reading

*None*

**Instructions** – PLEASE READ (notice **bold** and underlined phrases)

**Lab Exercise has three parts:**

- A. Lab Demo – Watch Demo, reproduce it, show your TA
  - B. Exercises – Work on the four lab debug exercises
  - C. Submission – There Four submissions this week
1. **You are to work on these lab exercises as individuals. Each student must reproduce the demo and show the TA that they have completed Part A.**
  2. **Submission deadline: Friday Feb 5<sup>th</sup> at 10:30am**
  3. **The exercises are presented in sequence so that you gradually advance with the material.**
  4. **Before you leave the CSIL labs, make sure that a TA looks at your work in order to receive your attendance and lab active participation marks.**
  5. **Lab05 Intended learning outcomes**

By the completion of the demo, students should be able to use NetBeans to:

- Single step through programs and methods
- Set breakpoints and view program state and internal variables
- Set Command Line parameters through the NetBeans IDE

## A. Lab Demo – Presented by TAs, and repeated by Students

Since the projector did not work in Week 04, the TA will repeat the Lab04 demo this week. The TA will only present a quick overview of this material. Students must observe the overview presented by TA, then complete Part A on their own. Past A is a repeat of Part A from Week 04; if you already completed Part A from last week, show your TA your working program and move on to Part B.

Student must have the TA check off demo program completion by end of tutorial for full marks. Marks will be awarded for attendance but only partial marks for incomplete work at the discretion of the TA.

Each student must create a running project based on the demo for full marks.

## Debugging Code with NetBeans

In the Lab04 demo, you learned how to debug code using NetBeans. NetBeans is a powerful IDE that allows you to monitor what your program and methods are doing, how they affect variables and state, and use it to locate bugs. This is a far more powerful and efficient debugging technique than adding `cout` statements throughout your code.

In this Demo, we will be using a C++ program that formats text strings as either left justified, right justified, or centered. You will need the following file which is available on the CMPT 135 Course website called:

- `Lab04Main.cpp` (same file as last week)

To get started,

### 1. Create Project `Lab04Justify`


Create a project called `Lab04Justify` and import the source file for `Lab04Main.cpp` from the Class website.

- Download `Lab04Main.cpp` and place it in your Downloads directory, or some other temporary location.
- Start NetBeans. Make a new project called **Lab04Justify**.
- Click on 'Source Files' under the `Lab04Justify` project to reveal `main.cpp`. Right-click on it, and select 'Rename', Rename it to `Lab04Main.cpp`.
- Right click on '**Lab04Justify**' project in the Package Explorer view and select 'Properties' from the menu. Select "C++ Properties" and ensure that the "C++ Standard" is set to **C++11**. If it is not already set that way, select it from the menu on the right hand side of that option.
- From the Ubuntu desktop, open the Filing Cabinet and go to your

Downloads directory. Double click on file Lab04Main.cpp. By default, it will be opened by GEdit. Click on the window, then press <ctrl-a> to select the entire file, and <ctrl-c> to copy the entire file to your clipboard.

- Click on your NetBeans window, and double-click on Lab04Main.cpp to open it in NetBean's main edit window. Click on the window to select it. Press <ctrl-a><ctrl-v> to over-write the existing file with the contents of Lab04Main.cpp from your Downloads directory.
- You should now have the contents from your downloaded Lab04Main.cpp in your Lab04Justify NetBeans project.
- **Verify this by clicking on the Green Arrow to "Run Project".** Your program should compile and run. The output should indicate that it is using default parameters with Width 20 and Justification set to Left. However, there is a bug in the program and should then report "**Error: No text specified. Exiting...**".
- The program is *supposed* to accept a set of command line parameters and justify the text according to the displayed Usage statement. If no text is specified, it is supposed to Left-Justify the default text about the quick brown fox.
- Instead of using the defaults, it is reporting an error. Your job is to debug this code and fix it!

## 2. Open a Debug Perspective

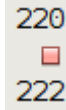
- Up to now, we have been using the NetBeans **Run** configuration. In this configuration, we can run and test our program but we cannot set breakpoints or debug our program very easily.
- The Debug configuration is oriented towards debugging.
- Enter this configuration by either hitting the "Debug Project"  button beside the green "Run" button, or go to the top level menu bar, and select "Debug Project" from the Debug menu, or simply hit <ctrl-F5>.
- When you enter this mode, nothing much may appear different but it is using a different configuration. When you enter this mode, it will likely recompile your code and run it.
- The Output window at the bottom will still contain the output from the program that just ran but now we will be able to step through the running program and display variables at different points in the execution. This will allow us to find the bug that is causing our program to report an error.

## 3. Executing a Program One Line at a Time



**A basic ability of any good IDE is the ability to control the execution of a program — to be able to stop the program when and where you want and view its current state. The simplest example of that (from a user**



perspective) is line-by-line execution. This is called **Single Stepping** through a program,.

- a. In the Edit window, scroll through Lab04Main.cpp until you find `main()` around line 221. Click on the line number beside where






`main()` starts and a red square  should appear on line 221. This indicates that you have just set a breakpoint at the start of `main()`. Do the same thing on the first IF statement after `main` around line 224. You should have two breakpoints set now.

- b. Now when you hit the "Debug Project"  button, the program will stop at the first breakpoint at the start of `main()`. Because it stopped, there is no output yet produced. The program is stopped at this breakpoint waiting for you.
- c. Look at the line under `main()` where `Parms` is declared. Notice that there is a green arrow  where its line number used to be. This indicates that this is where the program will resume once you tell it to continue running.
- d. When you are at a break point, you can display the values of any variables that are in scope, or set more breakpoints, or tell the debugger to continue until it hits the next breakpoint, or reaches the end of the program.
- e. In bottom window, besides the output window, you will notice there are three new tabs: the **Variables** Tab, the **Call Stack** Tab, and the **Breakpoints** Tab. You may also notice that at the very bottom there is a little orange ball bouncing back and forth: this is telling you that you are in Debug Mode and that your program is waiting for you. Go to the Window menu at the top, select Debugging, and you will see a list of all the various debug windows you can use. We will stick to these three for now.
- f. Click on the **Variables** window. Notice that the Variables window shows three variables: `parms`, `argc`, `argv`. At this point in the program, these are the only variables in scope.
- g. This window displays the values of `argc` and `argv` which have well defined values at this point. `Parms` has not yet been initialized by our program and so contains uninitialized values.
- h. `Argc` is set to one because only one command line argument was passed in: the name of the program itself. `Argv` contains a pointer to a pointer to the name (because it is an array of pointers to character arrays - but you do not need to know these details yet!).
- i. Click on the **Call Stack** window. Notice that it only contains one line

- containing `main()` and its two parameters (`argc=1`, `argv=0x7ffff...`). When `main()` calls `ParseCmdParms`, it too will appear in this window along with its parameters. This window can be very useful for understanding how a function was called when it had a bug.
- j. Now click on the **Breakpoints** window. This contains the set of breakpoints where you have told your program to stop. At this point, we have set two: both in `Lab04Main.cpp` at lines 221 and 224. Notice that the green arrow is indicated on the first breakpoint. That is where the program is stopped.
  - k. Hit the Continue button  (or F5 button) to continue execution until it hits the next breakpoint. Notice that the IF statement on line 224 is now highlighted in green and the green arrow in the Breakpoints menu has moved to the second breakpoint.
  - l. If we hit Continue again, the program would run to the end because there are no more breakpoints. But breakpoints are not the only way to debug. We can also single step through the program execution.
  - m. Let's single step into the `ParseCmdParms` routine by hitting the StepInto  button (or F7 button). This tells the debugger to execute the next step in the program. If that step is a procedure call or function call, it will step into the routine and stop, waiting for the next command. You may notice that before it enters `ParseCmdParms`, it first shows the declaration of struct `Parms` which it needs as a parameter for the call to `ParseCmdParms`. Hit the StepInto button a several times until you reach the switch statement.
  - n. Now look at the **Call Stack** window once again. It still contains `main()`, but now `ParseCmdParms` has been pushed onto the stack along with its parameters and their values.
  - o. Click on the **Variables** window. Now you can view all the variables that are in scope within `ParseCmdParms`. `Argc` is still one, `Argc` is unchanged, but `parms` has not been initialized thanks to the three statements before the switch statements.
  - p. Review the switch statement and the value of `argc` which is used by the switch statement. Can you predict where StepInto will go next? Hit StepInto and see if you are correct.
  - q. The debugger went to the `Case` statement where `argc == 1` and is about to do a `cout` statement.
  - r. We could StepInto each of those `cout` statements, but that would not be interesting. Instead, click on the line number associated

with the return statement to create a new breakpoint there.

- s. Now hit the Continue button and the debugger will skip past all those cout statements and stop at the return statement, just before ParseCmdParms is about to return to main.
  - t. In the Variable Windows, notice that value of parms. It has a width of 20, it is specifying Left justification, and its inputText is set to something but it is unclear what. Click on the dots icon  on the right, and NetBeans will pop up a new window to show you the rest of the contents of parms. Can you see what inputText is set to now? You should see the string value near the end of this display. No need for you to understand the gory details, just notice that the field inputText is set to "The quick brown fox jumped over the lazy dog".
  - u. Close the window and hit the StepOut  button (or the <ctrl-F7> button). The StepOut button continues running the program until it completes the current function and returns to the caller where it then stops once again, waiting for the next debug command. In this case, it will continue running until it returns to main() following the call to ParseCmdParms.
  - v. Now look at the Variables window and check what the current value of parms is now. Does it still contain the values that it had within the ParseCmdParms procedure? If not, then why not?
  - w. To exit debugging mode, terminate the current debug session by hitting the Finish Debugger Session  Button (or the <shift-F5> button).
  - x. You can remove individual breakpoints by click on the red square where the breakpoint is, and it will be deleted and the line number redisplayed.
  - y. To delete all breakpoints, go to the BreakPoint window and right-click to get the breakpoint menu. From there, you can select Delete-All to delete all breakpoints, or Disable-All to disable the breakpoints for now (but you can reenale them later).
  - z. Once you have deleted (or disabled) all the breakpoints, you can hit the Continue button to complete the execution of the program to the end.
4. Fix Lab04Main.cpp so that ParseCmdParms works and show the TA
- a. Fix ParseCmdsParms so that parms is set correctly when the function returns. Try to solve this on your own, or with your partner. You can ask your TA for help if you get stuck.
  - b. When you have fixed the program, it should display the following in

**the output window when you run it:**

```
Using default parameters:
```

```
Width=20 Justification=Left Text="The quick brown fox jumped over the lazy dog"
```

```
The quick brown fox  
jumped over the lazy  
dog
```

```
RUN FINISHED; exit value 0; real time: 0ms; user: 0ms; system: 0ms
```

- **Show your TA this output for partial Participation Marks. You must also get Command Line Processing working for full Participation Marks. Keep going!**

**5. Command Line Arguments**

Many programs accept command line arguments when they are invoked. For example, `Lab04Main.cpp` justifies the same text string every time in exactly the same way. We might instead want to use the command line to pass our own text to be formatted by this program. One might want to specify the width and justification type as left, right, or center.

- a. **Compile and test `Lab04Main.cpp` from the command line.**

To call `Lab04Main.cpp` from the Command Line, you need to build it from the command line. Using a terminal window, go to your NetBeans folder with your `Lab04Main.cpp` program (using the `cd` command) and compile it using the following command:

```
g++ -std=c++11 Lab04Main.cpp -o justify
```

Once compiled, you can invoke it from the command line. Here is one example:

```
justify 20 C "This is the text to fill and justify."
```

This would produce the following output:

```
parameters:  
Width=20 Justification=Center Text="This is the text to fill and justify. "  
  
This is the text to  
fill and justify.
```

- b. **Testing Command Line Parameters using NetBeans**

To set the command line parameters using NetBeans, select the Project `Lab04` from the project explorer window, and right-click and select properties.

From the Properties window, select the "Run" category. You will see a menu of Configuration Options. At the top you can select either the "Release" configuration or the "Debug" Configuration. Select "Debug" for running with the debugger (the other option is the configuration used when you hit the RUN button instead of the DEBUG button).



The first Configuration option under "General" is called "Run Command". This is where you set the command line parameters within NetBeans.

The Run Command starts with "\${OUTPUT\_PATH}". Click on this, and add your command line parameters after this string. Set this configuration Option as follows: (all on one line)

"\${OUTPUT\_PATH}" 35 Right Everything should be made as simple as possible, but no simpler. - Albert Einstein

Run Lab04Main.cpp from NetBeans with the Run Command set as above. The program should produce the following output:

```
parameters:
Width=35 Justification=Right Text="Everything should be made as simple as possible,
but no simpler. - Albert Einstein "

Everything should be made as simple
as possible, but no simpler. -
Albert Einstein

RUN FINISHED; exit value 0; real time: 0ms; user: 0ms; system: 0ms
```

- c. Since PART A is a repeat of week 04, there is no need to submit your working Lab04Main.cpp to CourSys.

### Check in with the TA

Demonstrate to the TA that you are able to pass command line arguments to Lab04Main and have it right justify your text as shown above, **then Continue to Part B if time remains in your lab session. Otherwise you will have to complete it later.**



## B. Lab Exercises – To be completed by Students Individually

Now that you know how to use the NetBeans debugger, you will find that debugging your programs will be much easier. It might even be fun. In this lab exercise, we will debug some programs in order to get practice using the NetBeans debugger.

The exercises do not need to be completed by the end of the lab for full marks, but for full participation marks you must be able to demonstrate to the TA each of the following:


1. Demonstrate breakpoint and single-step debugging of Lab05Buggy
2. Show that you have found and fixed the bug in Lab05Buggy.cpp
3. Demonstrate breakpoint and single-step debugging of the student's own RandomAvg.cpp program from Assignment #1

Students must work and complete the exercises individually.

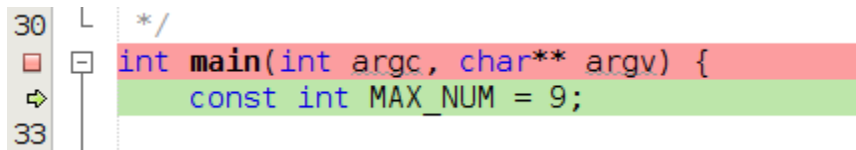
1. Create a new C++ project called Lab05Buggy
  - a. Create a new C++ application program called Lab05Buggy
  - b. Go to the course website "Labs" page and open the file named Lab05Buggy.cpp, replace all the contents of main.cpp with the contents of this file using cut and paste.
  - c. Rename main.cpp to Lab05Buggy.cpp
  - d. The Lab05Buggy.cpp main method uses a `for` loop to display the numbers from 0 to 9 and compute their average to three decimal places. Run the program to see if it works.
  - e. You should see the following traceback in the Output window at the bottom that shows that the program had a run-time error. Notice that it is reporting a floating point exception:

0 1 2 3 4 5 6 7 8 9

RUN FINISHED; Floating point exception; core dumped;


2. Use the NetBeans Debugger to *single step* through the program to find the bug and fix it.
  - a. To begin single stepping, we need to tell NetBeans that we want control as soon as the program begins execution. Set a breakpoint on main at line 31 by click on the line number. It should turn into a red square .
  - b. Select Debug Project (or <ctrl-F5>) from the debug menu to start debugging.
  - c. Notice that while the breakpoint is set on line 31, code execution stops on line 32 as indicated by the green shown below. That is because this

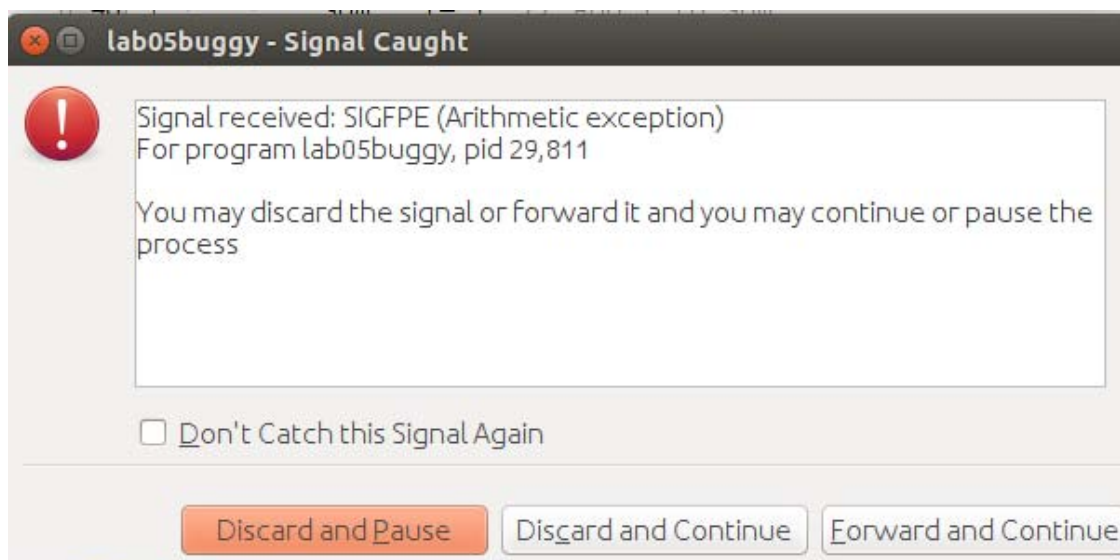
is the first executable line of code in the main program. Breakpoints can only stop where there is code. If no code exists, NetBeans will stop at the next place where code exists: in this case, that is line 32.



The screenshot shows a code editor with a breakpoint (red square) set at line 32. The code is as follows:

```
30  L  */
31  int main(int argc, char** argv) {
32  const int MAX_NUM = 9;
33
```

- d. Click on the Variables tab to view the list of variables in scope. Notice variables sum and count. Notice that they are set to zero by default.
- e. Notice also that the FOR loop variable `i` is not currently in the list of variables. That is because it is not in scope. It is local to the FOR loop.
- f. Single Step through the program by hitting the  'Step Over' button.
- g. Continue Single Stepping until you enter the FOR loop. Notice that `i` is now in scope as well.
- h. Single step until the FOR loop exits once `i == 10`. Watch the values of the variables `i`, `sum` and `count` after each time through the loop.
- i. Continue single stepping until you get to line 52 where `avg` is about to be computed. Record the values of `i`, `sum`, and `count`.
- j. Single step again until you get the exception reported once again.
- k. Notice that in debug mode, when an exception occurs a new window pops up:



- l. This window reports that an exception signal was caught by the debugger. The signal SIGFPE stands for a Floating Point Exception. Given what you recorded in step i above, what went wrong?

- m. The Signal Caught window allows you debug exception errors, or ignore them and continue running. Hit Forward and Continue, then hit single-step again and notice you get the same Floating Point Exception as before.
- n. What did you notice about the variables `i`, `sum`, and `count` as you watched the program run. Did all three get updated through each time through the for loop?
- o. Did you spot the bug? If not, try it again and look at the statements about to be executed at each step. Understand what each line is supposed to do and what local variables *should* get modified.
- p. Before you execute each line, look what variable should be modified. Look at its current value in the Variables window, and then press Step Over, to execute the statement. Was the variable changed as you expected? If not, figure out why.
- q. There are three statements that get executed in the loop before the code encounters a run time error on line 32. The statements are:
- `for (int i=0; i<=MAX_NUM; i++) {` - Line 39: FOR loop adds 1 to i
  - `sum += i; // Add i to sum` - Line 40: Should Add i to sum
  - `count *= 1; // Add 1 to count` - Line 41: Should Add 1 to count
  - `cout << i << " ";` - Line 42: Should output i
  - `}`
- r. One of these four lines is causing the problem on line 52. Keep single stepping through the code until you are about to execute line 52.
- s. Before you execute line 52, check that values of `sum` and `count`. The sum of the numbers from 0 to 9 should be 45. Does it have the right value? If not, then line 40 may be where the bug is. If it does equal 45, line 40 is not likely the cause of the problem.
- t. Now check `count`. After going through the loop 10 times, `count` should equal 10. Does it have the right value? If it does not equal 10, then it may be the problem. Look at line 41 which modifies `count` in the loop. Can you spot the bug?
- u. Once you figure out what the bug is. Edit the code to fix it, then single step through the program again to see if the code is now working as expected.
- v. The computed average of these numbers should be reported as 4.500. Is this what the program now displays? If not, then line 52 miscalculated avg. Click on line 52 to set a break point there.
- w. Clear the breakpoint at the start of main by clicking on red dot on line 31. Stop the current debug session and restart a new one. Execution will not stop on line 52 just before calculated avg.

- x. Look at the values of `sum` and `count`. Why would line 52 get this division wrong and return 4 instead of the correct value of 4.5?
- y. Inspect line 51 and the declarations of `sum` and `count`. Why is it computing an `int` division instead of a floating point one? Fix the bug by changing the calculation into a floating point operation by changing the declaration of `sum` from `int` to `double`.
- z. Retest the fixed `Lab05Buggy.cpp` program. Now that you have fixed `Lab05Buggy.cpp` so it does not get an exception, run it by hitting the run button. Does it now compute the correct average?

### 3. `Lab05Buggy2`

- a. Create a new Project called `Lab05Buggy2`
- b. Paste the `Lab05Buggy2.cpp` source code from the CourSys Labs page.
- c. The intent of this program is that it displays each character of `testStr` on a separate line. The default value of this string is "Hello!" which is 6 characters long.
- d. Run the program and notice that this program does not crash, but it displays two extra characters. Why?
- e. You used the Eclipse Debugger to find the bug in `Lab05Buggy.cpp`. This time we will use Trace statements instead. To do tracing, most people simply adding calls that send text to `cout` while debugging, and then remove these statements once their program is working.

The problem with this method is that once you make another change, you will need to add in those Trace statements all over again, and then remove them as well. That is not efficient.

There is a better way. `Lab05Buggy2` has Trace statements already in place but they are inactive: nothing gets printed when you run it. Why? Look at line 24 where the following boolean is defined:

```
#define Want_DebugStatements false
```

Using a `#define'd` boolean is an excellent way of controlling Trace statements so you do not need to constantly add them in, then comment them out, and so forth. By setting this to `true`, the trace statements get executed and displayed. Setting it back to `false` results in no trace statements getting executed or printed.

- f. Good Trace statements can do more than print information; they can also test assertions and detect errors. Edit line 24 and change the value of this boolean to `true`. Run the program now to see the Trace output is displayed as it runs.

The Trace statements have not only provided Trace information, they have detected the bug. Fix the bug by changing line 37 so that it uses

**"testStr.length()" instead of "sizeof(testStr)".**

- g. **Rerun the program to verify that it no longer has a run-time error and that the Trace statements no longer detect an error.**
- h. **Set the `Want_DebugStatements` boolean back to false and rerun the program to verify that it is working correctly.**
- i. **If you use Trace statements, consider using this convention of having a boolean control whether the Trace is displayed or not. It will save you a lot of time during your debugging sessions.**
- j. **So what was the bug on line 37?**
- k. **"testStr.length()" is invokes the `length()` method for the `testStr` string object and returns the number of characters in the string. Many new C++ programmers mistakenly use `"sizeof(testStr)"` - this returns the number of bytes needed for the **POINTER** to the `testStr` object. Pointers are 8 bytes (64 bits), so in this case `sizeof` returns 8 regardless of how many characters are in the string. Tip: Use the `.length()` for strings.**

#### 4. Lab05Buggy3

- a. Create a new Project called Lab05Buggy3
- b. Paste in the `Lab05Buggy3.cpp` source code from CourSys Labs page.
- c. This program does a simple computation using doubles that results in the value 10.0 and stores the value in the double variable `x`. It then compares `X` with the constant 10.0 using the helper function `equalDoubles()` to verify that everything worked as expected.
- d. Run this program and you will see that the wrong output is displayed. This program is using doubles to do some very simple math and comparisons. One would expect it to work, but it doesn't!
- e. Use the Eclipse Debugger or Trace statements to determine what is wrong with the floating point comparison within the `if` statement and fix the bug.
- f. For a hint, read about comparing Doubles at Stack Exchange here:  
<http://stackoverflow.com/questions/18971533/c-comparison-of-two-double-values-not-working-properly>
- g. Fix the helper function to compare two doubles accord to the solution provided by this website. The helper function should return true if the two doubles are within a tolerance of Epsilon.
- h. Rerun the code to verify that the `if` statement works correctly and submit the working program into CourSys.

#### 5. Lab05AnimalData

- a. **Create a new Project called Lab05AnimalData**
- b. **Paste in `Lab05AnimalData.cpp` source code from the CourSys.**



- c. This copy of `AnimalData` is based on our partially completed `AnimalData` class. The class now declares settings and getters for private member variables `AnimalType` and `Sound`.

```
29 class AnimalData {
30 public:
31     AnimalData();
32
33     string getAnimalType();
34     string getSound    ();
35     void  setAnimalType(string type );
36     void  setSound     (string sound);
37
38 private:
39     string AnimalType;    // Type of Animal
40     string sound;        // What sound does it make
41
42     char   carnny_omni_herby; // What does it eat
43     int    NumLimbs;         // How many limbs does it have
44     int    size;            // Size where 1 is small, 100 large
45     double weight;         // Estimated avg weight in KGs
46 };
47
```

- d. Member functions have defined for these getters and setters but they are currently just stubs.
- e. Review functions `InputAnimal` and `OutputAnimal`. They worked last Friday when `AnimalType` and `sound` were public. Now that these member variables are properly encapsulated as private member variables, these functions need to use getters and setters to work.
- f. Run the program and notice that the program gets aborted with a core dump and an invalid pointer exception. That's not good. Let's use NetBeans debugger to figure out why.
- g. Set a breakpoint at the start of `main` on line 147. Type <ctrl-F5> to run the debugger. It will stop at the breakpoint on line 150 since this is where `main`'s executable code starts.
- h. While at the breakpoint, click on the Variables tab at the bottom of the NetBeans window. Notice that there are three `AnimalData` object variables in scope, These match the three declared within `main`: `bear`, `bee`, and `Lion`.
- i. In the variables window, click on `bear` to view its members variables. Click on its `AnimalType`, then click on `_M_p` which is where the characters for the string are stored. Notice that it contains no valid characters. This is because the default constructor has not been called yet for `bear`.
- j. Now hit the Step Over button to allow `bear`'s default constructor to be



called. Look at `_M_p` again and you will see that it points to the characters "Worm" which is the default `AnimalType` string set by the default constructor.

- k. Continue stepping to the program until you get to the first call to the `OutputAnimal` function on line 156. We are about to call `OutputAnimal` with object `bear`. This is where the program will crash.
- l. Hit the Step-Into button  (F7) until you step into the call to function `OutputAnimal`.
- m. Within `OutputAnimal`, there is only one local variable in scope: `animal`
- n. Click on the Variables window to view the contents of `animal`. Check its `AnimalType _M_p` value. Is it still worm?
- o. So the values of `Bear` have been passed into `OutputAnimal` as a copy parameters into the object "animal". Looks good so far.
- p. Step-Into the call to `getAnimalType()` on line 92. Now the only local variable in scope is called "this". This represents the object whose member function is being invoked. Explore it through the Variables window and verify that `AnimalType._M_dataplus._M_p` points to the string "Worm" as we expected.
- q. Notice that `getAnimalType` contains no code. *That's the problem!* It is supposed to return a string value by returning the private member variable `AnimalType`. Since it does not do this, the caller will get an invalid pointer to a random piece of memory. Verify this by hitting the Step-Out button  to return to the invoking function. Continue to single step through the code until you hit the Signal Caught pop-up window. Hit Forward and Continue and you will see random characters displayed on your screen plus an "invalid pointer" error.
- r. You will need to implement these four getter and setting functions to fix this bug. To get you started, here is the implementation for `getAnimalType`:

```
string AnimalData::getAnimalType(void) {
    return AnimalType;
}
```

- s. Implement the four incomplete getters and setters and validate that it now runs as shown in class.

```
value of bear after being initialized through the default constructor:
Worm says Nothing

Now lets use InputAnimal to initialize bear:
Animal Type : Bear
Animal sound: Grrr!!
Bear says Grrr!!
```



### C. Lab Exercise Submission – To be completed by Students

Students are responsible for submitting the requested work files by the stated deadline for full marks. Since Lab Exercise solutions may be discussed in class following the submission deadline, late submissions will NOT be accepted. It is the student's responsibility to submit on time. If you do not have access to CSIL or issues with the computers in CSIL, please contact the CSIL Help Desk at [helpdesk@cs.sfu.ca](mailto:helpdesk@cs.sfu.ca)

Students must work on these exercises individually and submit the set of files to CourSys. No group should be created for this submission.

1. You must submit your final version of the following file before the deadline. Students must ensure that all submitted code compiles and is properly commented and formatted for readability:
  - [Lab05AnimalData.cpp](#)
  - [Lab05Buggy.cpp](#)
  - [Lab05Buggy2.cpp](#)
  - [Lab05Buggy3.cpp](#)
2. Files are to be submitted into CourSys under Lab05. Do NOT create a group name for this submission, submit individually.