Lab 3 (10 points)  
 September 13, 2017  
Due: at 11:59 pm on day of lab session

Laboratory Assignment #3 —  
Debugging in Eclipse CDT[[1]](#footnote-1)



**CS-2303, System Programming Concepts, A-term 2017**

### Objective

To learn to use the *Eclipse C/C++* debugger.

### Introduction

One of the most powerful aspects of an Integrated Development Environment is the integration of a debugger with the graphical user interface. This makes the “edit-compile-debug” loop very easy and quick. You can easily set breakpoints by pointing and clicking. These let you get a good look at what your program is doing at key points during its execution.

*Eclipse CDT* includes a graphical interface to gdb, the *GNU Debugger*, which is one of the most powerful and flexible debuggers in the *C/C++* development world. Unfortunately, the command-line interface of gdb has a steep learning curve and requires more keystrokes than the intuitive *Eclipse* interface.

In this Laboratory Assignment, you will learn how to use the basic debugging features of *Eclipse*.

### Getting Started

Please sign the attendance sheet.

Login to your virtual machine, and download and unzip the file Lab3\_Sample.zip from Canvas.

The result should be directory named Lab3\_Sample containing a makefile and a single source code file Sample.c.

Start *Eclipse* and create a new *C* *Makefile project* for these two files. Build the project.

### Launch Configurations

In order to run or debug the program with command line arguments, you need to set up *launch configurations.* A *launch configuration* is information about the arguments, environment, and other information needed to run and/or debug the program. There are two kinds of *launch configurations:–*

* *Debug configurations*
* *Run configurations*

When you right-click on an *Eclipse* project and select Run As or Debug As you are given the option of running it as a Local C/C++ Application, just as if you had typed the name of the program in a command shell with no arguments. Or you can select from the Run Configurations … or Debug Configurations … submenus, respectively. These let you provide arguments and other settings needed to run or debug the application properly.

For this lab assignment, you need to set up a Debug Configuration for the *Lab3\_Sample* project that you just set created. Right click on the project name and select Debug As and then select Debug Configurations … from the submenu to bring up a window similar to the following:–

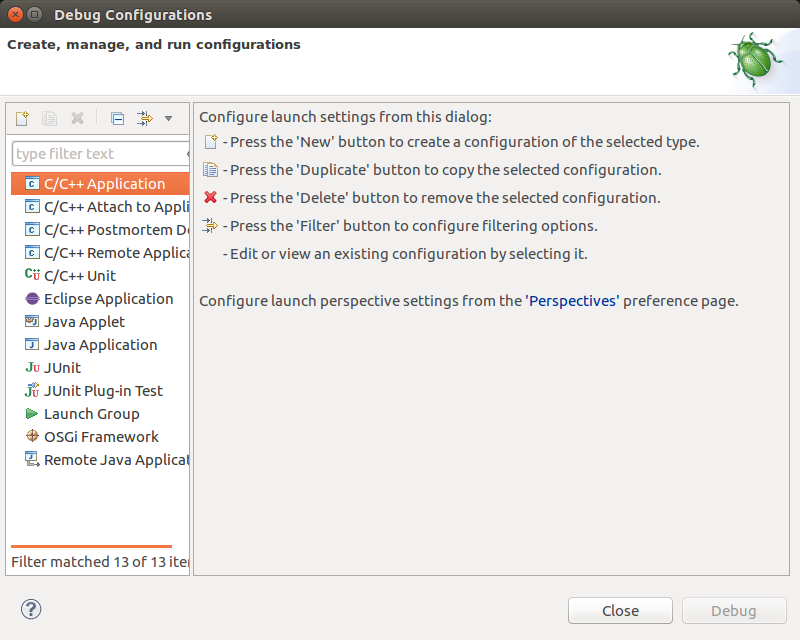


Figure 1

The list of things in the left panel may vary depending upon what is already in your *Eclipse* workspace. Click the New button (circled in red in Figure 1) to create a new configuration for the current *C* or *C++* application in the Project Explorer panel. This will bring up a template for a new launch setting, shown below in Figure 2. Note: If the New button is grayed-out, try right-clicking on C/C++ Application and then picking New.

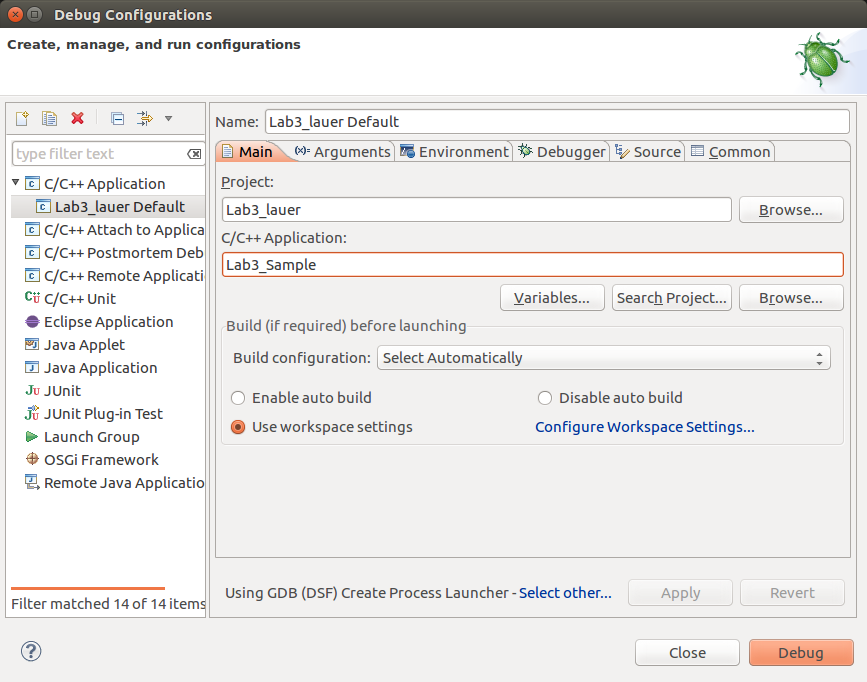


Figure 2

Note that the project name has been filled in (in this case *Lab3\_lauer*) and has the name of the application program (*Lab3\_sample*). Select the Arguments tab, to reveal a window in which you can type the command-line arguments for this program. Type the following arguments:–

9 8 7 6 5 4 3 2 1 0

The result should look like this:–

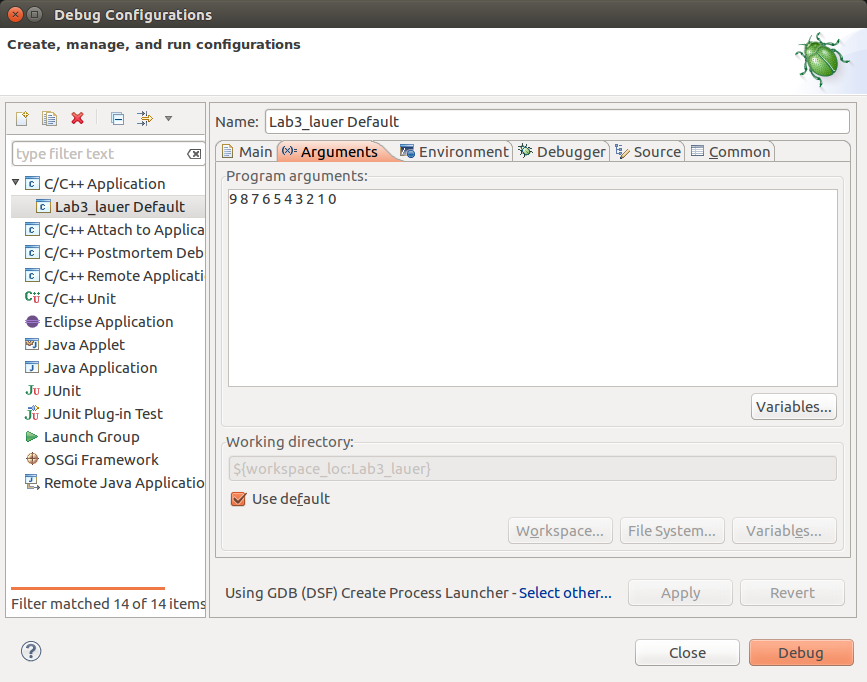


Figure 3

This is equivalent to the command line

./Lab3\_Sample 9 8 7 6 5 4 3 2 1 0

(where Lab3\_Sample is the name of the program in the box in Figure 2).

This program simply reads its arguments from the command line, sorts them using a well-known sorting algorithm called *Shell Sort*, and prints them in order on the standard output.

Build and run the project. If it works properly, it should print the line

0 1 2 3 4 5 6 7 8 9

Run it several more times. Vary the number of arguments and their values and convince yourself that it is still running properly. (You can run the program with different arguments simply by selecting Run Configurations… from the Run menu, and then by selecting the Arguments tab.) Enter the arguments you want and click on the Run or Debug button. The results will be displayed in the Console tab at the bottom of the *Eclipse* window.

This program has a bug in it — a rookie error that may be hard to spot and that may or may not show itself.[[2]](#footnote-2) It is possible that you could run many tests and never see the results of the bug, or you may have already seen it. A test case that sometimes brings out the bug is one of the following sets of arguments:–

**100000000 9 8 7 6 5 4 3 2 1 0  
 100000000 8 7 6 5 4 3 2 1 0**

Another way to bring out the bug is to run the program Lab3\_Sample from a Linux command shell. Run the program from the command prompt with various sets of arguments.

A third, more reliable, way to bring out the bug is to export the project to a CCC Linux system (or to any other Linux system). Build the program with the makefile and run it with a several argument lists. In some cases, it will run correctly, but in some cases the output is sorted and contains the right number of results, but at least one of the inputs is replaced by a bogus number. In a few cases, the program gets a segmentation fault or other fatal error.

### Debugging

To debug your program in *Eclipse*, make sure that it was built with the **–g** switch. This tells the compiler to include extra information in the object file so that a debugger can find its way around the compiled code. In this example, the –g switch is already part of the CFLAGS variable in the makefile.

Open the project file Sample.c in a source code tab. Notice that this program has two functions — shell\_sort() and main(). Right click at the extreme left margin of the source code to bring up a menu. Select Show line numbers to cause line numbers to display in this tab.

Next, set a breakpoint on the line where the variable a is initialized in main() — i.e., line 31. To set the breakpoint, right click again in the left margin and select Toggle Breakpoint. A tiny blue dot will appear next to the line number. This tells the debugger to suspend execution when it reaches this point in the program — specifically, it should pause *before* executing any code on that line.

You can start the debugger and simultaneously load the program to run in debugging mode by invoking the Debug Configurations… command under the Run menu. *Eclipse* now switches to the Debug perspective. (*Eclipse* may display a dialog box asking if you want to switch to the Debug perspective*.* Say Yes to this dialog.) In this arrangement of windows, a Debug tab appears in the upper left corner, along with a group of tabs in the upper right (i.e., Variables, Breakpoints, Expressions, Registers*,* and Modules), a group of tabs in the left middle containing your open source code tabs, a tab on the right called Outline, and a group of console tabs at the bottom. A picture of the Debug perspective is shown in Figure 4.[[3]](#footnote-3)

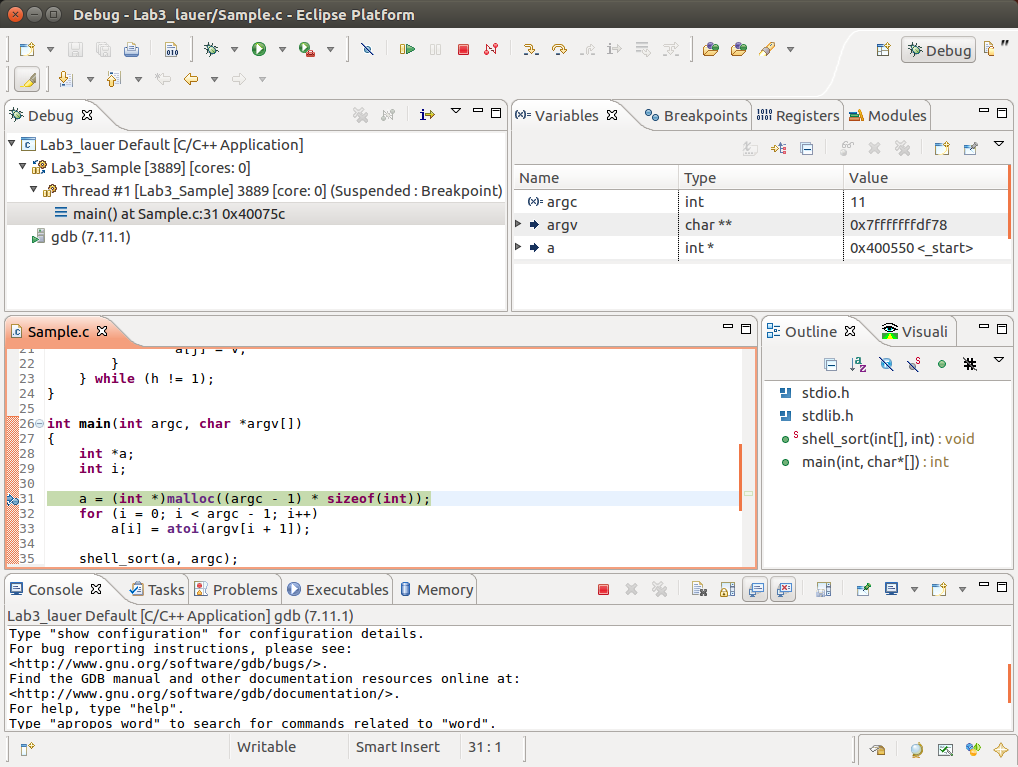


Figure 4

At this point, the debugger is waiting to allow your program to execute its first instruction — i.e., before the first executable line in main(), which happens to be where you set the breakpoint.

In the left margin of the code of the Sample.c tab will be a small green arrow indicating the point of execution. In the upper right window, the Variables tab shows the names and values of the variables in the current scope at the point of execution. As you can see, argc contains the number of arguments that you supplied to the program (including the program name itself) and argv is a pointer. Both a and i have garbage values because they have not yet been initialized.

You can also hover over an instance of any variable in the code, and a popup box will appear with the value of that variable. This also works for expressions. For example, select the expression

**argc – 1**

in the argument list of the call to malloc() on line 31. Now, hover over this expression, to bring up the popup box shown in Figure 5. In this case, the value is 11, because the program had been called with twelve arguments.

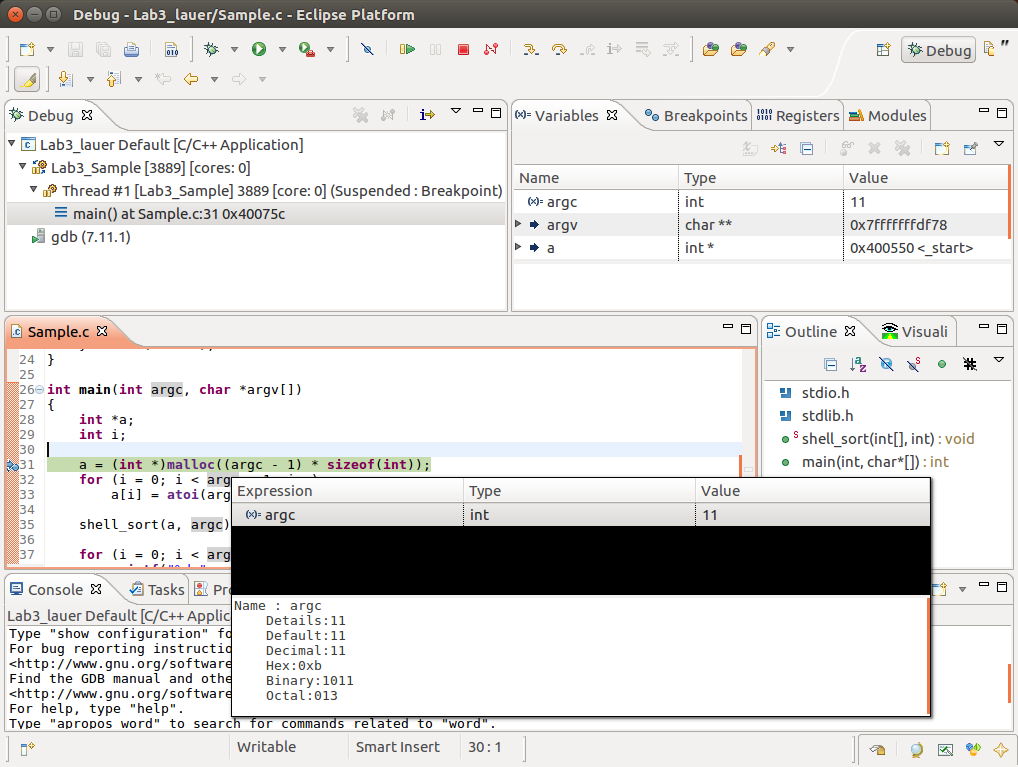


Figure 5

#### Stepping through your program

At the top of Figure 4 is a row of buttons for common debugging actions. This row is shown here in magnified form. Allow the cursor to hover over each icon to see its name.

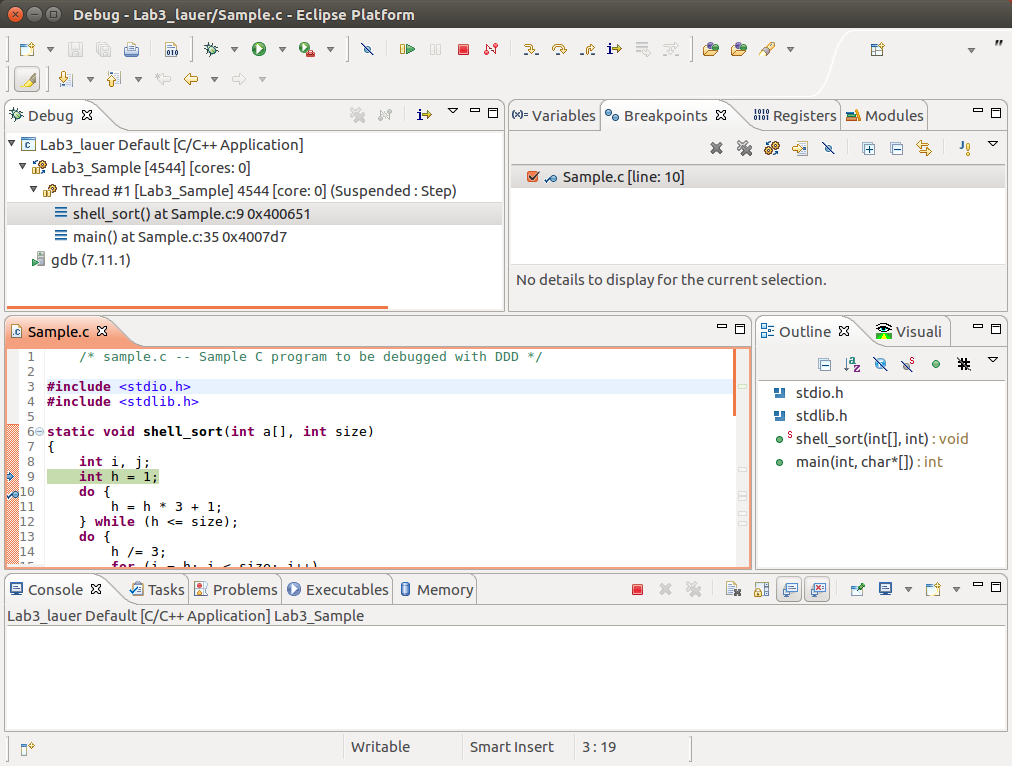
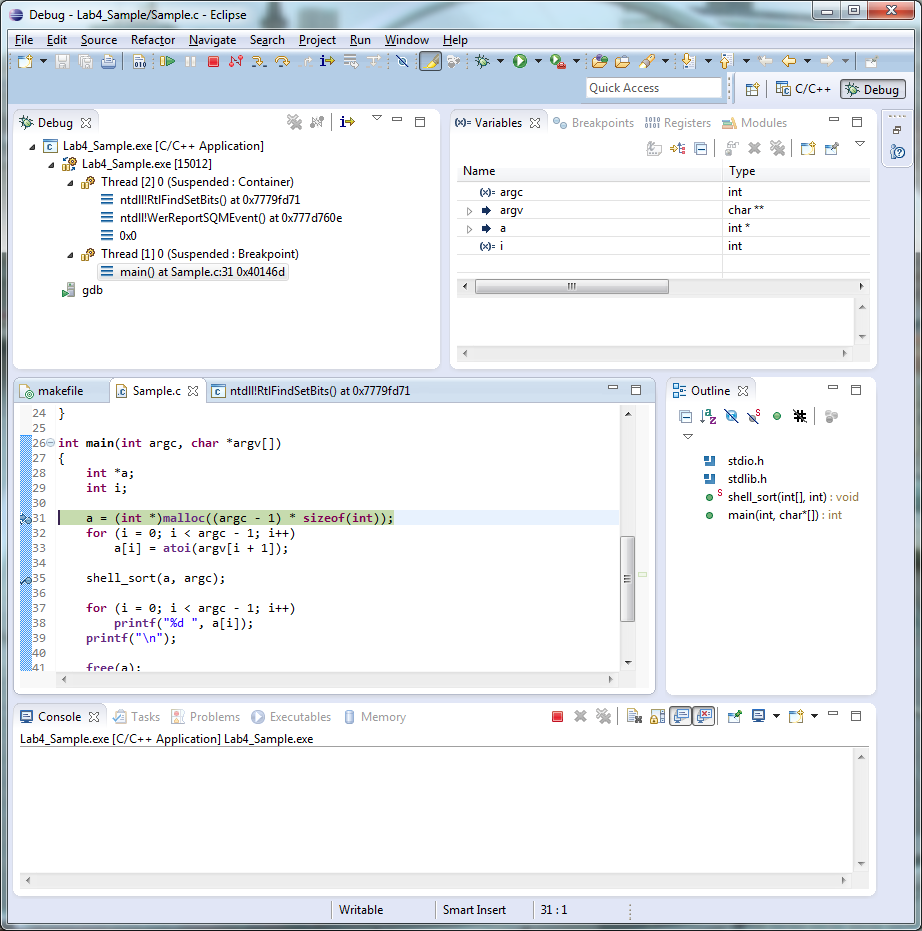


Figure 6

Three very useful debugger commands are Step into, Step over, and Step out, represented by the three yellow arrows shown in the oval in Figure 6.[[4]](#footnote-4) Click on Step over. You will see that the point of execution has advanced to line 32 and that the variable a now has a valid pointer value.[[5]](#footnote-5)

Click on Step over twice more to enter the loop that initializes the array a. Examine the values of the argument of atoi (i.e., argv[i+1]) and the resulting assignment to the array element a[i]. They should be the same value, and that value should be the value of the first argument on your list in the Run Configurations… or Debug Configurations… menu command.

Next, set a breakpoint on line 35, the call to the shell\_sort() function. Click the Resume button (i.e., , just right of the center of Figure 6). The program will finish the for-loop and suspend execution just prior to the call to shell\_sort(). To see what the arguments to shell\_sort are, go to the Variables tab in the upper-right window. So far, the debugger thinks that a is a pointer. However, *you* know that it is an array. To see the values of the array, right-click on the name a (in the Variables tab) and select Display as Array…. A new dialog box pops up asking you to enter the starting index and the length of the array. Enter zero for the starting index, and enter the value of argc for the length, because this is what the program supplied to shell\_sort() on line 35.

Click on the small arrow to the left of a in the Variables tab. This will expand a and show the values of the individual elements of a. In addition, you can click on the line representing a itself, and in the lower panel of the Variables tab, it will show the entire array. Figure 7 shows an example.

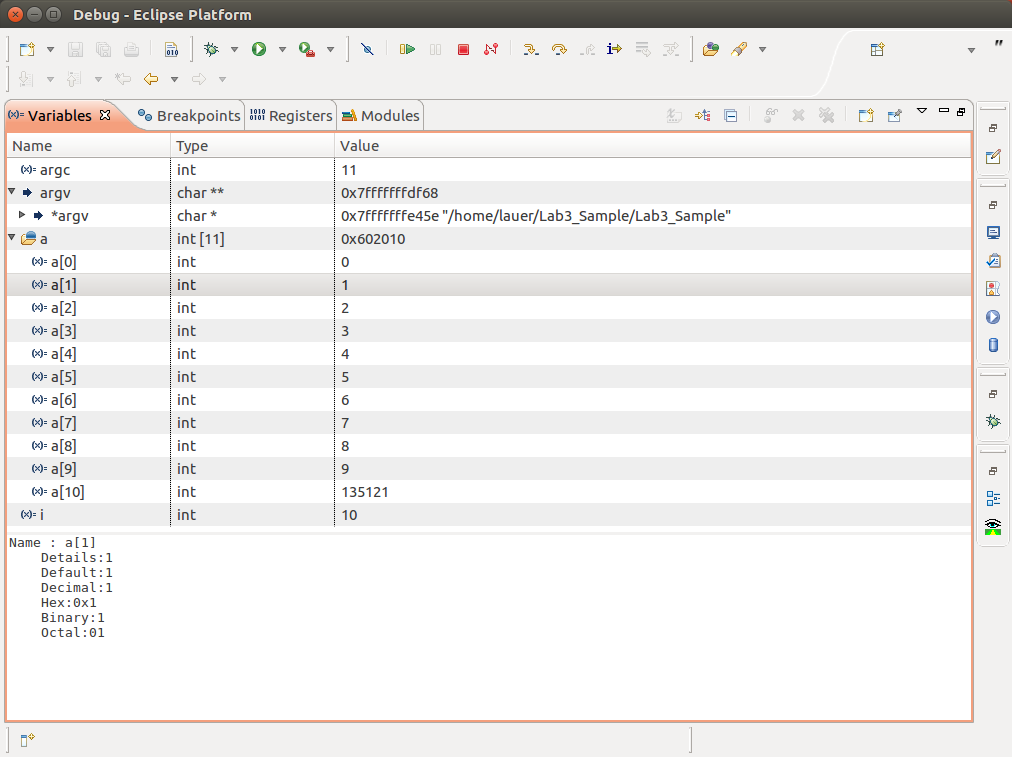


Figure 7

Now the problem becomes apparent:– there are one too many values in the array a, and the last value is a garbage value. The shell\_sort() function correctly sorts this (erroneous) list and puts the garbage value in numerical order along with the rest of the array, and then the for-loop of lines 37-39 prints out the sorted array, but of the correct length. There are three possibilities:–

* The garbage value is larger than any of your input values. In this case, it sorts to its original position at the end of the extended array, the print loop prints out the sorted array (not including the garbage value), and the program appears to work correctly.
* The garbage value is smaller than at least one of your input values. In this case, it sorts into the appropriate place in the array, and the largest of your input values is left off the end of the output. Also, the largest value does no damage to anything else, so the program appears to work almost correctly.
* The nastiest case is that garbage value is smaller than at least one element of the array, and the largest of your inputs overwrites some crucial data that just happens to be stored at the end of the array. This crucial data could be control information for malloc(), which in many implementations is stored between the allocated storage. If this control information is overwritten in a damaging way, malloc() could go haywire, perhaps causing a segmentation fault.

Because of these three possibilities, the error may or may not have made itself known when you were testing your program from a command prompt or a Linux shell.

If you now click the Resume button and let the program run to completion, the debugger *may* discover that The Heap has been corrupted and exit with the following warning:–

warning: HEAP[Lab3\_Sample.exe]:  
warning: Heap block at 005118B0 modified at 005118E8 past requested size of 30

To finish this part of the lab, fix the “error” in the program, rebuild the project, and convince yourself that it works correctly and completes without any warnings about Heap corruption. In particular, show that the value beyond the end of the array is not corrupted.

#### Managing breakpoints

There are many other things that you can do with breakpoints. Click in the Breakpoints tab in the upper right group of the Debug Perspective. You should see something resembling the following:–

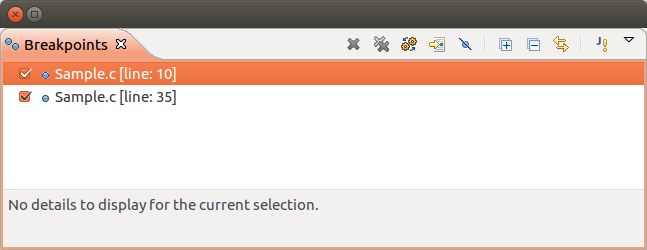


Figure 8

A breakpoint can be enabled or disabled, simply by checking or unchecking the box next to it. If the breakpoint is disabled, the debugger will not stop at that particular line. You can also enable or disable a breakpoint by right-clicking in the margin of the source window next to the line of the breakpoint. Try this now, and notice that the check box in the Breakpoint tab changes accordingly. In addition, you can enable or disable a breakpoint by holding down *shift* and double-clicking in the margin.

The ability to enable and disable breakpoints is very handy, because you can set any combination of them in a serious debugging situation without having to delete them and re-create them. You can also see at a glance which breakpoints are present and which are enabled. This is enormously helpful in keeping track of a complex debugging session.

To remove a breakpoint completely, right-click on it in the Breakpoints tab and select Remove. Alternatively, you can select the line of source code, right-click in the margin next to it, and select Toggle Breakpoint. You can also remove *all* breakpoints in your program by selecting Remove All from the same menu.

Note that *Eclipse* (and most other debuggers) preserve breakpoints across debugging sessions. This means that you can suspend a debugging session, exit *Eclipse*, and return later to resume debugging the same program with the same breakpoints.

#### Conditional breakpoints

You can dynamically control whether a breakpoint is taken or not by making it a *conditional* breakpoint. To see how to do this, Terminate your previous debugging session by clicking on the red square which is two icons to the left of the oval in Figure 6. (Alternatively, you may right-click on the name of your program in the Debug tab in the upper left of the Debug Perspective.) Set a new breakpoint on line 33 of Sample.c — i.e.,

a[i] = atoi(argv[i + 1]);

In the Breakpoint tab, right-click on this breakpoint and select Breakpoint Properties … . This brings up the dialog box of Figure 9, which lets you control under what circumstances a breakpoint is actually taken. Select Common in the left of this dialog box.

Set the value of the Ignore count field to 5 and click Run > Debug to start the program in the debugger. Click Resume, and it will run to the breakpoint, but it will only stop on the fifth time it encounters it. You can verify this by looking at the Variables tab. The value of i should be 5 (you may have to scroll the tab to see the variable).

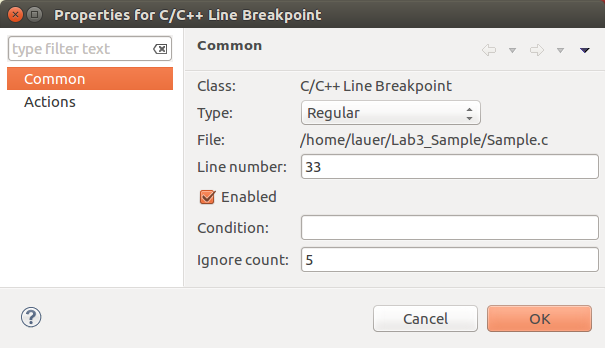


Figure 9

You can also stop on the value of a variable or expression. To see this, set the Ignore count in Figure 9 to zero, and set the condition to

i >= 5 && i <= 8

Invoke Run > Debug, and click Resume to start the program in the debugger. It should stop in the for-loop of main with a value of 5 for i. Verify this in the Variables tab. Click Resume again, and it will stop when i has a value of 6, and then again for values of 7 and 8. However, when you click Resume after the value of 8, the program will continue to the end, because the breakpoint condition is no longer satisfied.

You can put an arbitrary *C* expression in the condition field. If it evaluates to zero, the breakpoint is not taken. If it evaluates to a non-zero value, the breakpoint is taken. Experiment with this. For example, you might want to stop on the condition

atoi(argv[i+1]) == 9

This is a perfectly legal expression in the context of the breakpoint.

#### The Call Stack

Another important feature of the debugger is to display the *call stack* — i.e., the nested set of function calls from the current breakpoint back to the call of main. This is displayed in the Debug tab in the upper left of the Debug Perspective. To see an example, remove all breakpoints, and then set a conditional breakpoint on line 18 with a condition of v == 9 — i.e., at this line:–

for (j = i; j >= h && a[j - h] > v; j -= h)

Run the program to the breakpoint. The Debug tab should show something like Figure 10. The first line shows the program you are debugging (Lab4\_Sample.exe), and the second line shows the *process number* of the running program (5345, in this figure). This is the operating system’s identification of a particular instance of an executing program. Below this is a line labeled Thread#1. This represents your program, and the two lines below it show that it is at a breakpoint in shell\_sort() at line 18 and that shell\_sort() was called from main() at line 35.

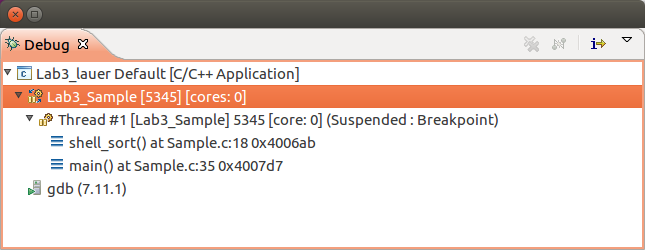


Figure 10

Click on the shell\_sort() line in Figure 10. The Variables tab will show the shell\_sort() variables at the moment of the breakpoint. Now, click on main() of Figure 10, and the Variables tab switches to show the values of the variables in main(). By this means, you can navigate up and down the call stack to see what conditions were present in each of the calling functions at the time of the breakpoint.

Finally, Terminate the debugging session by selecting Terminate from the Run menu, and the Debug tab will contain two lines denoting the carcass of the debugged program. You can remove this by right-clicking on it and selecting Remove all Terminated. Alternatively, you can re-launch the program being debugged by selecting that option in the popup menu.

### Getting Credit for this Lab

To get full credit for this lab, you need to capture two screenshots of (a portion of) the array representing the Game of Life board in Programming Assignment #3. Capture one screenshot of the array *immediately before* playing a generation, and capture another screenshot *immediately after* playing that generation. Select a non-trivial portion of the game board and convince yourself that the game has played at least that move correctly.

Note: An easy way to capture a screenshot of any tab in *Eclipse* is to do the following:– (a) Drag the tab from its place in the *Eclipse* window to the desktop. It now becomes an independent window on the desktop. (b) Press ALT and PrintScreen*[[6]](#footnote-6)* together on the keyboard to make a copy of that window. (c) Paste that copy into a document, which you can later submit. (d) Invoke the Reset Perspective… command from the Window menu to return the tab to its original place in the *Eclipse* window.

If this technique does not work, you can ask VirtualBox to take the screenshot. There is a menu choice Machine->Take Snapshot. Just be aware that this stores the screenshot on the drive of the host PC, not the drive of the VM.

Once you have captured both screenshots, submit them to *Canvas* under the assignment *Lab3.*

If you still have time left, spend it using the debugging tools of this lab to debug your Game of Life.

### Quick Reference Guide — Debug view toolbar options

The table below is copied from the *Eclipse CDT* help system. It lists the toolbar icons displayed in the *Debug* view. Note that *breakpoints* supersede the Step Over and Step Out commands. This means that if you attempt to step over a function call, and if that function contains a breakpoint, the breakpoint will be taken. Likewise, if you attempt to step out of a function in order to return to its caller, and if that function encounters a breakpoint, that breakpoint will be taken.

|  |  |  |
| --- | --- | --- |
| **Icons** | **Action** | **Description** |
| Remove all Terminated launches icon | **Remove All Terminated Launches** | Clears all terminated processes in Debug view |
|  | **Connect to a Process** | Enables the selection of a running process to debug. |
| Restart icon | **Restart** | Starts a new debug session for the selected process |
| Resume icon | **Resume** | Select the Resume command to resume execution of the currently suspended debug target. |
| Suspend icon | **Suspend** | Select the Suspend command to halt execution of the currently selected thread in a debug target. |
| Terminate icon | **Terminate** | Ends the selected debug session and/or process. The impact of this action depends on the type of the item selected in the Debug view. |
| Disconnect icon | **Disconnect** | Detaches the debugger from the selected process (useful for debugging attached processes). |
| Step Into icon | **Step Into** | Select to execute the current line, including any routines, and proceed to the next statement. |
| Step Over icon | **Step Over** | Select to execute the current line, following execution inside a routine. |
| Step Return icon | **Step Return** | Select to continue execution to the end of the current routine, then follow execution to the routine's caller. |
| Drop to Frame icon | **Drop to Frame** | Select the Drop to Frame command to re-enter the selected stack frame in the Debug view. |
| Instruction Stepping Mode icon | **Instruction Stepping Mode** | Activate to enable instruction stepping mode to examine a program as it steps into disassembled code. |
| Use Step Filters icon | **Use Step Filters** | Select the Use Step Filters command to change whether step filters should be used in the Debug view. |
|  | **Menu** | Use the Debug view menu to:   * change the layout of debug view (tree vs. breadcrumb) * alternate showing of full or no paths on source files * control how views are managed * activate or deactivate the child elements shown in the Debug view |

1. The sample program and the debugging sequence of this lesson were adapted from Chapter 1 of *Debugging with DDD*, version 3.3, by Andreas Zeller. *DDD* is the *Data Display Debugger*, also a graphical interface to *gdb*, but independent of any Integrated Development Environment. It is available on the CCC Linux systems, and (probably) the course VM. It was an excellent graphical user interface (GUI) for a debugger. Unfortunately, it has not been maintained for nearly a decade. [↑](#footnote-ref-1)
2. If you should happen to spot the problem in the code, please continue to work through this laboratory lesson anyway, just to get a feel for tools and actions of debugging in *Eclipse*. [↑](#footnote-ref-2)
3. If your Debug perspective does not resemble this, execute the Reset Perspective… command in the Window menu. This can be used whenever you get your windows and tabs so mixed up that they become hard to use. [↑](#footnote-ref-3)
4. All of the icons in this row are explained in more detail in the table at the end of this document. [↑](#footnote-ref-4)
5. You really do not want to click Step into at this point, because that could take you into the malloc() function, which is very large and contains a lot of executable code but no source code. It would not be useful to spend any time looking around in it. If you do accidentally end up in a system call such as malloc(), simply click Step out to return to your program. [↑](#footnote-ref-5)
6. **ALT-PrintScreen** is equivalent to calling **gnome-screenshot** from a command shell. It automatically selects the active window. [↑](#footnote-ref-6)