Lab 4 (10 points)  
 September 20, 2017

(The day after International Talk Like a Pirate Day)

Laboratory Assignment #4 —  
More Debugging in Eclipse CDT



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

Due: at 11:59 pm on the day of your lab session

### Objective

To learn additional techniques for the *Eclipse C/C++* debugger, particularly in *C++*.

### Introduction

You should now have some basic skills surrounding the use of *Eclipse* and its debugger. Programming Assignment #4 requires to implement a linked list to support an event-driven simulation. This lab illustrates how you can examine the structure of a linked list.

In addition, we will learn how to examine the call stack, how to set conditional breakpoints, how to change a data value while debugging a running program, and how to watch an expression.

### Getting Started

Sign the attendance sheet.

Create a directory to hold the files for Lab 4. Download the file Lab4\_C++\_example.zip from Canvas. Unzip it into a directory; do not use a directory under your eclipse-workspace directory; Eclipse does not like importing from inside its own workspace.

Your directory should now contain a makefile, a single header file SortedList.h, and two *C++* source code files, Lab4\_example.cpp and SortedList.cpp. This program is a little bit like the Sample.c from Lab 3, but instead of putting its input arguments into an array, it puts them into a linked list in numerical order.[[1]](#footnote-1) It also keeps a running total of the values in the list. There are no intentional bugs in this program.

Start *Eclipse* and create a new *C++* project containing copies of these files. Build the project, and set up the launch configuration with a sequence of random numbers, for example:–

123 456 789 -1001 2002 -3003 45 0 23 32767 -100000000

This program simply reads its arguments from the command line, inserts them one-at-a-time into a linked list, preserving numerical order, and prints them on the standard output.

### Overview of the Code and its Classes

If you study the code, you will see that Lab4\_example.cpp contains a the main() function and a global object S of class SortedList. The class itself is defined in SortedList.h, and its methods are implemented in SortedList.cpp. The body of the function main() is a for-loop that converts the command-line arguments to numeric values and calls S.InsertItem() to insert them into a linked list in numerical order. Then main()prints out the list in revere order, and it prints the sum of the items, which it obtains from the getSum() method of S.

In the Project Explorer, click on the small triangle next to SortedList.h to expand its contents. You will see something like the Figure 1 below.

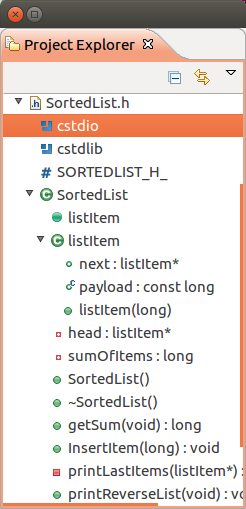


Figure 1

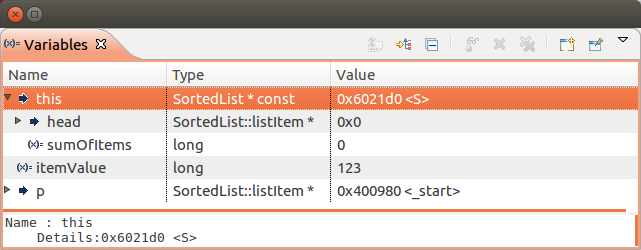
This shows a #define line (SORTEDLIST\_H\_), two included header files (<cstdio> and <cstdlib>), and the items declared in this file. The principal definition is a *C++* class named SortedList, indicated by a “C” in a green circle. This is the Class View of the project. Notice that this class contains a member class named listItem, indicated by another “C” in a green circle. The Class View is an important representation of any project in object-oriented languages such as *C++* and *Java*. In these languages, your thought processes are organized more around the classes, their members, and their relationships, rather than what the executing code does.

In case you did not already know, one of the differences between C++ and Java is that C++ puts class definitions in header files along with the function prototypes, but puts the bodies of the functions in .c files. Java puts everything in .java files.

### Linked List

Let us first watch the linked list being built up. Set a breakpoint at the beginning of InsertItem() (line 34) in SortedList.cpp, and invoke Run > Debug to start the program in the debugger with the argument list above. Click the Resume button to let it run to the breakpoint.

* Look at the Variables tab in the upper right. Notice that the values of the local variables of InsertItem() are displayed, along with the virtual pointer this. Click on the tiny triangle to the left of the identifier this, and you will see the values of members of the class object. So far, both head and sumOfItems have values of zero (NULL).



Figure

Resume the program by clicking the Resume button.[[2]](#footnote-2) This takes you through the next iteration of the first for-loop of main() and to the next call to InsertItem(). Resume several more times until the list has accumulated several values. Now, look at the Variables tab and expand the value of head, the value of next in the listItem object pointed to by head, the value of next in the listItem object pointed to by the previous next, etc. You should see something resembling Figure 3:–

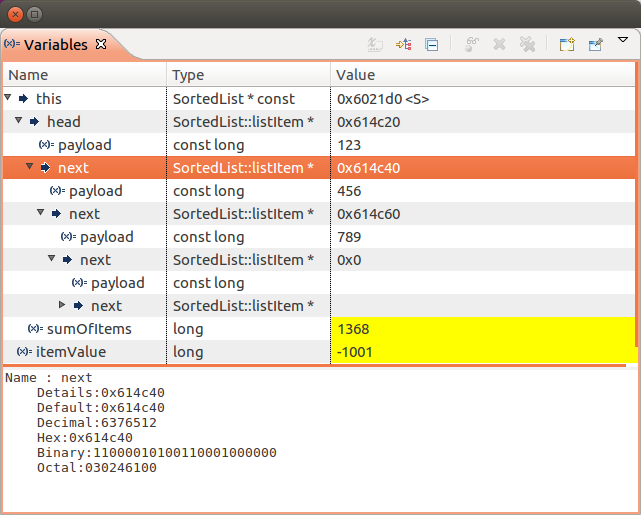


Figure 3

Notice also at the bottom the information panel that tells you everything you need to know about the variable or member selected above (in this case, the next member of the first item in the list).

Finally, notice that in this example, the parameter itemValue of the next item to be inserted into the list has a value less than any previous member. Click Resume one more time, and you will get something resembling Figure 3, which shows the new sort order:–

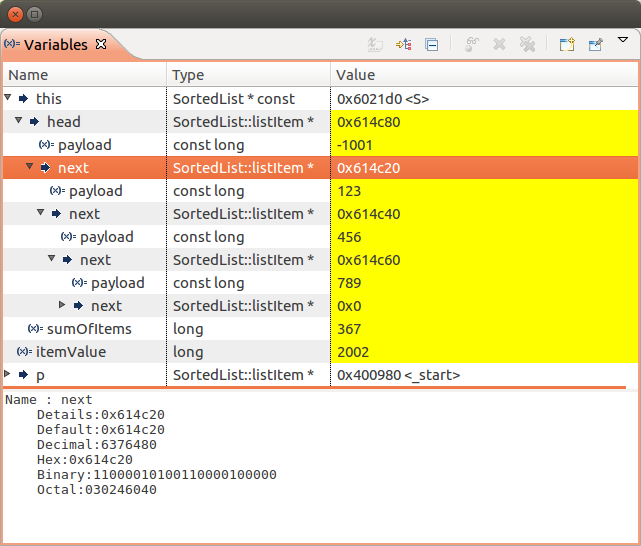


Figure 4

By this means, you have been able to inspect the contents of the linked list. If the list had been longer, you would have had to scroll the view vertically or horizontally or both.

You can also apply this technique to other kinds of data structures such as trees, queues, etc. For example, if you have an array of pointers in your program, you can follow any pointer of the array by clicking on the little arrow to the left. If, on the other hand, the pointer is really meant to be interpreted as the beginning of an array, you should use Display as array… from the previous laboratory session.

**A little trick:** If that you cannot remember where something was declared, select its identifier in an expression or statement, right click, and select Declarations > project from the popup menu. The Search tab of the Console group at the bottom will show you the file name and line number of the declaration. Double-clicking on that information will take you directly to the declaration. The search operation will also leave a small yellow arrow next to the declaration in both the .cpp file and the .h file.[[3]](#footnote-3)

### Other Debugging Techniques

To continue this lab, click Reset Perspective… in the Window menu. If the perspective no longer has all of the tabs that you wish to see, click Show View in the Window menu and select the *view* (i.e., the tab) that you wish to see. Also, terminate your debugging session by selecting the Terminate button (i.e., the red square) in the top row of the *Eclipse* window.

The following exercises will illustrate other debugging techniques.

#### Running to a specific line of code

You learned how to set breakpoints in Lab 3. A useful kind of breakpoint is the *temporary breakpoint* — i.e., a breakpoint that is to be used only once. In this case, you run the program, and when it reaches any breakpoint, the temporary breakpoint is deleted. In *Eclipse*, this is invoked by the menu command Run to Line.

Try this in Lab4\_example.cpp. Start the debugger by invoking Run > Debug. Disable all breakpoints in the Breakpoints tab, and then select Line 26 — i.e., return EXIT\_SUCCESS. Right-click to bring up the popup menu, and select Run to Line (about two-thirds of the way down the menu). The program will run to this line and take the temporary breakpoint. This is just like a normal breakpoint, and you can do all of the same things as a normal breakpoint. However, it is an unconditional breakpoint, and it evaporates as soon as you get there.

The temporary breakpoint also evaporates if you take another breakpoint on the way. To see this, re-enable other breakpoints — e.g., the one at SortedList.cpp, line 34. Start the debugger by invoking Run > Debug. Select Line 26 of again Lab4\_example.cpp and invoke Run To Line. Note that it stops at your recently re-enabled breakpoint. Now disable your breakpoints and click the Resume button. Notice that the program goes all the way to completion. It does not stop at the temporary breakpoint you had previously requested.

#### Displaying Global Variables

Last week, you learned how to view the values of the variables of a function in the Variables tab. However, you do not see the global variables there. Since these are not within the scope of any function, the display parser of the debugger does not find them whenever the debugger stops at a breakpoint.

To see the global variables, you need to use the Expressions tab. To expose the Expressions tab, select Window > Show View > Expressions starting in the top menu. Next, invoke Run > Debug. Before resuming the program, click the Expressions tab in the upper right tab group, and select Add new expression. Add the expression S — i.e., the name of the global SortedList item declared on Line 15 of Lab4\_example.cpp. This is shown in Figure 5 below.

At this point during the execution of the program, S.head is NULL and the sumOfItems member has been initialized to zero. As the program executes and you stop at future breakpoints, you will see that S builds up the data structure of this program.[[4]](#footnote-4)

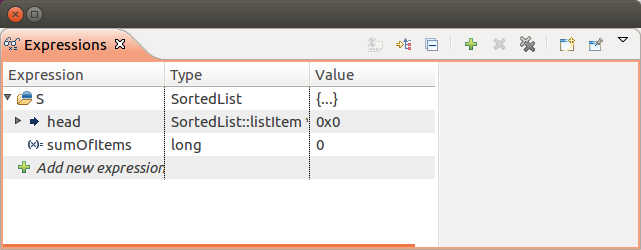


Figure 5

#### Changing data values

Occasionally, you find that you are in the middle of a debugging session when you really wish that a certain variable had a different value in order that you can make progress without starting over. This is easy — you simply type the new values into the *Variables* view. To see how this is done, re-run you program in the debugger, stopping in SortedList::InsertItem() after receiving four inputs. In the *Expressions* view, edit the first few items by typing new values in the Value column for each payload. For example, Figure 6 shows the same items of Figure 4 edited to new values, right in the dialog box. Running the program to completion shows that the new values are printed as part of the output. Try this with your program.

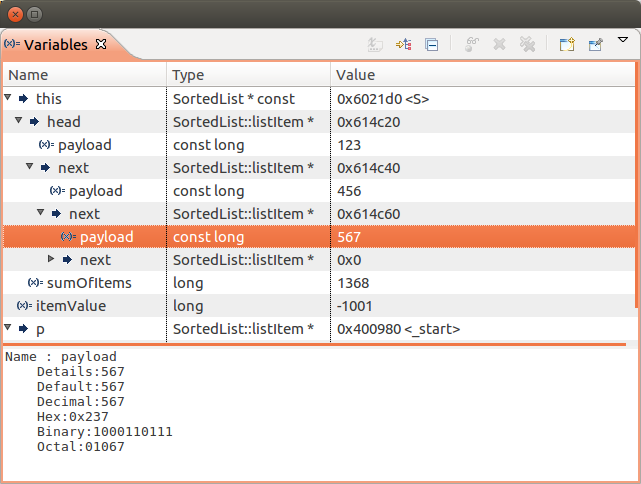


Figure 6

The program will run to completion and print out a list of the newly assigned values. However, it will end with an error, because the getSum() method of the SortedList class does a check that the running total maintained by the method is the actual sum of the element in the list.

#### Calling a function from the debugger

In serious debugging situations, you may find yourself needing to carry out a non-trivial computation on your data *while you are in the debugger*. This may necessitate writing some “support” functions in your program, just to help you debug. These support functions won’t actually be called at run time, but you can get the debugger to execute them, simply by making them part of an expression in the *Expressions* view.

To try this, open the Expressions view and click +Add new expression. For the expression, type

S.getSum(head)

Now run the program in the Debugger, taking breakpoints as before at the InsertItem() method of the SortedList class (line 34). Watch the value of this expression in the Expressions view as you take each breakpoint. Notice that it adds up the current contents of the list, which changes with each input. This computation is carried out by the debugger, not by your program, but the debugger is calling the method in the context of the variables of your program.

### Note about include files

In previous terms, several students had the problem that *Eclipse* could not find the *C++* include files. The environment variables were set up correctly, but it did not help. We stumbled on the solution of creating a whole new workspace. This seemed to solve it. Apparently, *Eclipse* left some toxic settings in the old workspace that were preventing it from telling gcc/g++ where to find the include files.

### Getting Credit for this Lab

Spend the rest of your lab session using these techniques to debug the assignment you are currently working on, PA3.

To get credit for this Lab, set a conditional breakpoint to stop the PA3 program after it has processed some words from the input file, perhaps a dozen or so. Make sure that the contents of the struct representing the root node in the binary tree is showing in the pane which displays variables; single-step from your conditional breakpoint if necessary.

Take a screenshot of that display and submit it to *Canvas* under the Assignment *Lab 4.*

1. This is called an *insertion sort*. [↑](#footnote-ref-1)
2. I.e., the green arrow in the list of debug icons from last week’s lab. [↑](#footnote-ref-2)
3. You can clear these little yellow arrows by right-clicking in the Console tab and selected Remove Selected Matches or Remove All Matches. [↑](#footnote-ref-3)
4. This is a minor inconvenience, because you need to switch between two tabs to both automatic and global variables. You can modify the perspective by dragging the Expressions tab to another part of the Debug Perspective in order to see both Expressions and Variables at the same time. [↑](#footnote-ref-4)