CS-2303, System Programming Concepts, C-term 2017

Lab 5 (10 points)
February 14 & 15, 2017

Laboratory Assignment #5 — Consolidating Your Debugging Skills

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

Objective

To continue to develop your debugging skills in *Eclipse*.

Introduction

From the previous Laboratory Assignments, you should now know 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.

In this laboratory, we will learn how to track down a segmentation fault. It was originally intended to also introduce *watchpoints* in this lab. However, these have proven to be too fragile to be usable in this course.

Getting Started

Sign the attendance sheet.

Download and unzip the same set of files that you used for Lab #4 from the following URL:–

http://web.cs.wpi.edu/~cs2303/c17/Resources_C17/Lab4_C++_example.zip

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 puts the input arguments into a linked list in numerical order and keeps a running total of the values in the list.

Start *Eclipse* and create (or reopen) a *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
```

Segmentation Faults

Introduce a deliberate segmentation fault into this program by editing **SortedList.cpp**, line 49. The line currently reads

```
while ((q -> next) && (q -> next -> payload < itemValue))
```

Comment out the test for a **null** value of **q->next**, so that line reads

```
while (/*(q -> next) &&*/ (q -> next -> payload < itemValue))
```

This is likely to cause a segmentation fault because there are many combinations of input arguments that cause this loop to reach the end of the list, as indicated by a **null** value in the **next** field. Also, add the following line at the start of **main()** so that you know that your program has started:–

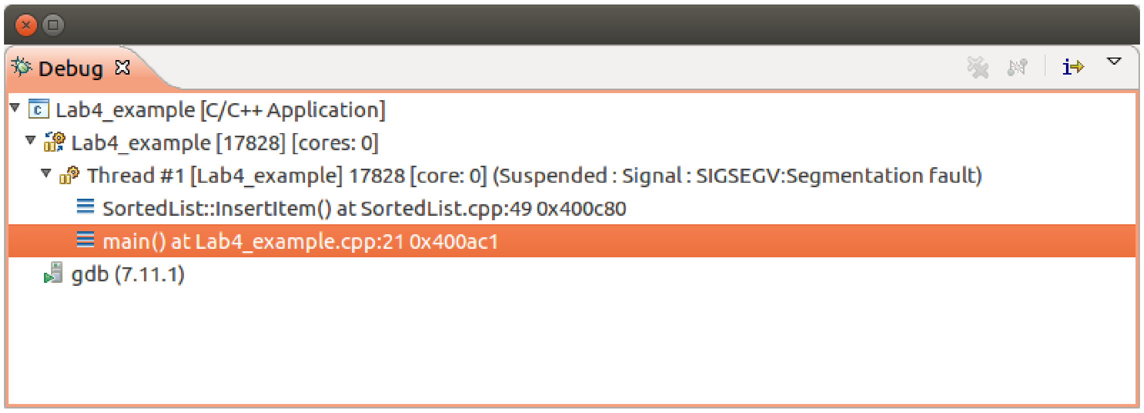
```
std::cout << "Starting test program." << std::endl;
```

Clean and build the project.

If you invoke the ***Run*** command in *Eclipse*, *Eclipse* swallows the output and terminates the program, without telling you anything. Try this.

If, however, you invoke the ***Debug*** menu command in *Eclipse*, the Console tab will display the fact that it received a Segmentation Fault and will display the line.

Cancel the dialog box and try debugging in *Eclipse*. Remove or disable all breakpoints and select ***Run > Debug***. Resume the program and let it take the segmentation fault. The debug tab should now look something like the following:–



```
Lab4_example [C/C++ Application]
  Lab4_example [17828] [cores: 0]
    Thread #1 [Lab4_example] 17828 [core: 0] (Suspended : Signal : SIGSEGV:Segmentation fault)
      SortedList::InsertItem() at SortedList.cpp:49 0x400c80
      main() at Lab4_example.cpp:21 0x400ac1
  gdb (7.11.1)
```

Figure 1

The top element in the call stack is, of course, the deliberate error that you introduced. Click on it, and it will take you to the source line (49) of **`SortedList.cpp`** that you just edited. However, click on the next function in the call stack, and you will see where the program was executing when it tripped over the fault. For the example list of input arguments above, you can see that it failed on the second argument — i.e., **`i == 2`**. Try it with different arguments to see the failure at different points.

This is a general technique for tracking down segmentation faults. Let the program run, and then reverse engineer the fault from the call stack.

Getting credit for this part of the Lab

Select different command line arguments so that the segmentation fault occurs at some argument past the second one. Capture screen shots of the ***Debug*** tab (resembling Figure 1) and also of the ***Variables*** tab, showing the iteration and input that caused the fault. Submit your screen shots to *Canvas* assignment *Lab 5*.

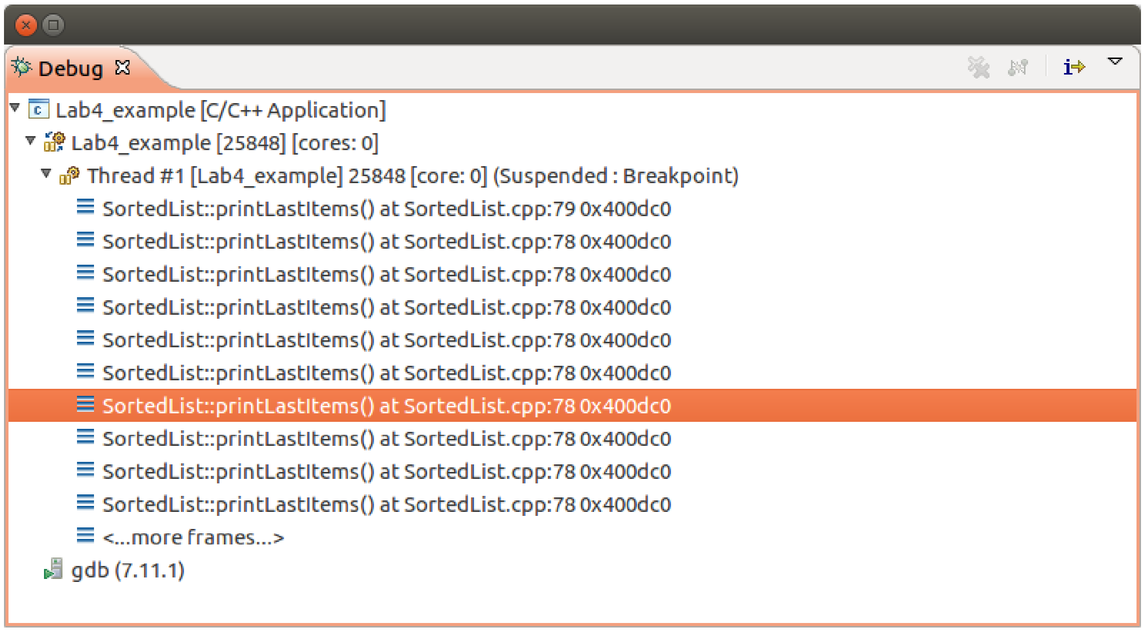
More about the Call Stack

For this part of the assignment, remove the **`/*`** and **`*/`** of the comment that you inserted on line 49 of **`SortedList.cpp`**. Clean, rebuild, and run your program to make sure that it operates as originally intended, without the (artificially induced) segmentation fault.

Next, clear all breakpoints, and then set a new breakpoint at line 79 of **`SortedList.cpp`**. This line is in the (recursive) function **`printLastItems()`**, at the line

```
std::cout << p->payload << " ";
```

Run the program to this breakpoint. You will see that it has stopped at the last of a list of recursive calls to **`printLastItems()`**, as shown in Figure 2.



```
Lab4_example [C/C++ Application]
  Lab4_example [25848] [cores: 0]
    Thread #1 [Lab4_example] 25848 [core: 0] (Suspended : Breakpoint)
      SortedList::printLastItems() at SortedList.cpp:79 0x400dc0
      SortedList::printLastItems() at SortedList.cpp:78 0x400dc0
      SortedList::printLastItems() at SortedList.cpp:78 0x400dc0
      SortedList::printLastItems() at SortedList.cpp:78 0x400dc0
      SortedList::printLastItems() at SortedList.cpp:78 0x400dc0
      SortedList::printLastItems() at SortedList.cpp:78 0x400dc0
      SortedList::printLastItems() at SortedList.cpp:78 0x400dc0
      SortedList::printLastItems() at SortedList.cpp:78 0x400dc0
      SortedList::printLastItems() at SortedList.cpp:78 0x400dc0
      SortedList::printLastItems() at SortedList.cpp:78 0x400dc0
      <...more frames...>
  gdb (7.11.1)
```

Figure 2

Select one of these recursive calls and note the value of **`p`** and **`p-> payload`**. Repeat this step up and down the *Call stack*.

Finally, display the properties of the breakpoint at line 79, and set it to ignore eight occurrences of that breakpoint before stopping the program. Select the most recent call of **`printListItems()`** and display its variables. Take another screenshot and submit to *Canvas* under the assignment *Lab5*.

Lab quiz in two weeks

One of the purposes of this lab exercise is to prepare you for the *Lab Quiz* in two weeks' time. You will have to display your skill at debugging a program, taking breakpoints, displaying its variables, and showing the call stack.