

**CSC112**                      **Fall 2010**  
**Fundamentals of Computer Science**  
**Lab 7**

In this lab we will not be writing any new programs, but will be taking a look at how to use a symbolic debugger. In addition to looking at a debugger (gdb), you should continue your work on Labs 5 and 6.

A symbolic debugger is a program to help you identify errors in your program. Symbolic debuggers are especially helpful for mysterious program failures such as “segmentation fault” and other run-time errors. Once you become familiar with using a symbolic debugger, you may find it *very* helpful for locating errors in your projects for Labs 5 and 6.

A symbolic debugger reads your executable code (native machine instructions which are contained in the file named **a.out** by default), and steps through (executes) your program in a controlled manner.

The debugger can correlate positions in the executable code to lines of C++ source code through a special table which is included in the executable code file. If an error occurs (say, a segmentation fault), the symbolic debugger can show you the line of your C++ source code which caused the error. You can also set a “breakpoint” in your program. Execution will pause at the breakpoint, and allow you to examine the variables in your program at exactly that point in the execution of your program.

Data objects in your C++ source code are reduced to addresses in the executable code. There is a table in the executable program which also enables the debugger to correlate machine addresses to the variables which you have declared in your C++ source code. You can refer to variables **by name** in a running machine code program. Sweet.

But, there is one more important fact: the table(s) that enable the debugger to correlate executable code to C++ source code are not included in the executable code by default. A compiler option is necessary to instruct the compiler to include debugging tables in the compiler’s output file. If our C++ source program is named “**lab7\_example.cc**”, then we need to use the following command to compile it for use with the debugger.

```
g++ -g lab7_example.cc
```

The symbolic debugger for Ubuntu is called **gdb**. You can use **gdb** from the command line, but it is much more convenient to use it through a graphical interface named **ddd**. Your default install of Ubuntu probably does not include **ddd**. To get **ddd**, open a terminal and use:

```
sudo apt-get install ddd
```

After installing **ddd**, proceed with the following steps.

1. Download the C++ program named “lab7\_example.cc” from

```
http://mehune.opt.wfu.edu/CSC112
```

2. Compile the C++ program using “g++ -g lab7\_example.cc”

3. Run `ddd` on the executable. I.e., “`ddd a.out`”. After a few seconds, two windows will open. The larger window is divided into two sub-windows. The upper sub-window shows your source code, and the lower sub-window shows a command line session with the underlying symbolic debugger (`gdb`). A second, much smaller window contains a control panel with action-buttons such as “Run”, “Interrupt”, “Step”, etc.
4. For our first experiment, click on the “Run” button. The program will halt with a **Segmentation fault**.
  - (a) On what line of code does the segmentation fault occur ?
  - (b) What are the values of the local variables (all local variables) where the fault occurs ? *Hint: Place the cursor in the lower window at the (`gdb`) prompt and use the `print` command. For example, `print i` will print the value of the variable `i`.*
5. For our next experiment, you will set a breakpoint to pause the execution at a desired point. Place your cursor directly in front of the line of code in the main program that reads:

$$r[j] = f(a,k) \quad ;$$

and hold down the right mouse button. A pop-up menu should appear. Select “Set Breakpoint”. At this point you should see a small red “Stop” sign in front of the statement. Go to the **Program** menu at the top of `ddd` and select “Run Again”. You may notice that there are keyboard shortcuts for everything in the menus. As you get more experienced with `ddd`, feel free to try the shortcuts.

- (a) Using the (`gdb`) command line session, display the values of the local variables `j` and `k` at the breakpoint and make a note of them.
- (b) Use the “Cont” button to continue execution. Once again, find the values of `j` and `k`, and make a note of them.
- (c) Repeat step 5b until the segmentation fault occurs. *Hint: This will not take very long.*
- (d) What were the values of `j` and `k` just before the segmentation fault occurs ?

**Last question: Why does this program cause a segmentation fault ?**

**Turn in:** Write a few sentences answering the questions in the steps above. Write your answers in a text file named `lab7.txt`. Upload the file “`lab7.txt`” to your account on telesto.