CS1110 Introduction to Systematic Programming

Practical Class 1 Week 5
Debugging with GDB

In the last practical class we used the Ada run-time debugger `gdbtk` to trace the execution of an Ada program and to view how the contents of the program variables changed as execution proceeded. Of course the debugger is normally used when a program contains logical errors, that is when it compiles and runs but fails to produce the correct results or indeed any results at all.

In this practical we will be using the `gdbtk` to debug several simple Ada programs.

First copy the files

```
dbex1.adb   dbex2.adb   dbex3.adb
```

to your own Ada directory from the directory

```
/usr/local/staffstore/cs1110/debug-exercises/
```

The first file `dbex1.adb` contains a buggy version of a program to compute factorials. Note the factorial of a number N (is denoted by N!)is the product of all the numbers between one and the number itself

```
N! = N * (N-1) * (N-2) * .....4 * 3 * 2 * 1
```

For example 4! = 4*3*2*1 = 24, and 5! = 5*4* 3*2*1 = 120 etc..

Compile the program for use with the debugger by supplying the option `-g` on the UNIX command-line:

```
gnatmake -g dbex1.adb
```

or use **Build** from the **Ada** menu (which supplies the option `-g` automatically).

Then run this program in the normal way and observe that it gives erroneous results.

Start the debugger with the command

```
gdbtk  dbex1
```

Arrange the **Source window** , the **Commands window** and the **Terminal window**on the screen so that all are visible.

Now start the program `dbex1` under debugger control by clicking on the **Start** button in the **Source window**.

The program `dbex1` is launched and the debugger gains control just as the main Ada procedure is about to start.

The Source window displays the source file `dbex1.adb` with the first executable statement highlighted in green. Recall that as the program is executed under the control of the debugger the highlighting moves through the source file and always indicates to the **next** line of Ada code to be executed.

Display the contents of the variables `N`, `I` and `Product`by doubling clicking on each variable in turn. The **Expressions Window** will appear with each variable's value displayed in a sub-window. Arrange these so that you can see the value of each variable and then drag the Expressions Window so that it does not overlap the other windows.

Now single-step through the program noting how the variables change as execution proceeds. Recall that program output appears in the **Terminal window** and when the program requires

interactive input from the keyboard you must type the required data into the terminal window and press <Return>.

Try to deduce what is wrong with the program. There are in fact two bugs[1] -- one very obvious -- the other slightly less so! If necessary, you can restart the program from the beginning by clicking on **Start** again and then enter a different value for the variable N.

Exercise
Compile dbex2.adb for use with the debugger. This is a buggy version of the GCD program whichg appeared in an earlier practical sheet (Lab 2.1). Now start dbex2 under the control of the debugger.

Single step through the program and enter two non-zero values for the variables First and Second (say 25 and 16 whose correct greatest common divisor is 1). Try to deduce what is wrong with the algorithm. Quit the debugger by selecting **Quit** from the **File** menu in any debugger window. Correct the bug (two assignments statements need swapping) using Emacs, recompile the program after saving the changes made and then run the program again under the control of the debugger and trace what happens this time. After several loop iterations the correct result should be obtained.

However the program is still not compeletly correct. It crashes if 0 is entered as the value of the variable Second. Single step the program and enter a zero value for the variable Second (and any old value for the variable First. Single-step the program keeping a watchful eye on the Command Buffer and see when a error occurs. Can you see the problem? Using Emacs modify the program in such a way as to avoid the division by zero problem (remember that calculating a remainder involves doing a division). Compile and test the modified program.

Important Note
When you modify a program and recompile, always quit the debugger and restart it to debug the modified program. If you don't you will still be debugging the old executable program -- this can become very confusing particularly if the modified source of the program may be displayed in the source window!

Setting Breakpoints on Exceptions
To allow the debugger to gain control whenever a run-time error occurs (in Ada, run-time errors are called **exceptions**), we need to set a Breakpoint on All Exceptions. To do this open the **Breakpoints** window from the **Windows** menu in the **Source** window, now click on the **Create** button, then select **Exception** and **All** in the window that appears[2].

It is now safe to run a buggy program at full speed using **Continue**, the debugger will gain control as the error occurs and will display the offending command.. **In future you should always** set a Breakpoint on All Exceptions. when debugging (do this immediately after clicking on **Start** to start the program).

Exercise
Finally try to debug the program dbex3.adb. This is a buggy version of procedurised version of the factorisation program in Unit 8. This exercise is intended to give you some practice at debugging programs containeing procedures.

Note for most inputs the program will go into an infinite loop and appear to 'hang'. If you are running the program normally, you can kill the program by pressing Control-C[3]. If you are running it at full speed under the debugger quit the debugger and then restart it.

---

[1] Even if you can spot the bugs just by looking at the source code, use the debugger to single-step the program and display the value of program variables as execution proceeeds -- not all bugs are so obvoius and so you will need to become familiar with the debugger.
[2] You can also create and delete normal breakpoints by using the **Create** and **Delete** buttons in the **Breakpoints** window.
[3] Press the Control and C keys together.

Single-step through the loop in `ProduceFactors` a few times, inspecting the values of relevant variables such as `TrialDivisor` and `NumOfDivisors`, and see if you can deduce the cause of the problem. If not, set breakpoints at strategic places in `ProduceFactors` and then restart the program (using **Start** and **Continue**) and then single step from the breakpoint and try to find the cause of the errors.

Remember always to clear a breakpoint by clicking on it before attempting to Step or Continue past it.

Some Hints on Debugging Programs containing Procedures

Step and Next
Recall that **Next** steps over procedures as if they were single instructions whereas **Step** steps through procedure bodies line by line.

The Procedure Call Stack
Always display the Procedure Call Stack by selecting **Stacks** from the **Window** menu.

Recall that a local variable or formal parameter can only be displayed when GDBTK is in that variable's frame, that is when the variable is **in scope**.

We can navigate up and down the call-stack by clicking on a level in the **Call Stack** window or using the **Up**, **Down** and **Bottom** buttons in the **Source** Window.

Even with the use of the call stack, it is only possible to inspect the value of local variables and formal parameters of a procedure whilst that procedure is active, that is when the procedure has been called, but has not yet returned. The reason for this should be obvious: storage for local variables and formal parameter is allocated only when the procedure is called and that this storage is released as the procedure returns. Hence the formal parameters, the local variables and the call frame of a procedure do not even exist when the procedure is not active.

Exercise
Practise navigating up and down the call stack and displaying the values of local variables and formal parameters of the various procedures in `dbex3.adb`.

'Magic' Return from Procedures
Occasionally you may accidentally step **into** a procedure when you meant to step **over** it. It can be very inconvenient to step through the procedure body, particularly if it contains a large loop which is repeated many times. To cater for this situation GDBTK provides the command **Finish** on a button in the Source Window. This causes the procedure to be run 'at full speed' until it returns, at which point the debugger regains control.