CS1110 Introduction to Systematic Programming

Practical Class 2 Week 4
Tracing Program Execution with GDB

In this practical we will be using the Gnu Run-time Debugger `gdb` via its graphical user interface `gdbtk` to trace the execution of an Ada program.

First copy the files

```
factors04.adb     savings08.adb
```

to your own `Ada` directory from the directory

```
/usr/local/staffstore/cs1110/unit-programs/
```

The file `factors04.adb` contains the example factorisation program from Unit 4 and the file `savings08.adb` contains the procedurised version of the savings program from Unit 8.

Change directory to your `Ada` sub-directory then compile the factorisation program for use with the debugger by supplying the option `-g` on the UNIX command-line:

```
gnatmake -g factors04.adb
```

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

Start the debugger with the UNIX command (from a terminal window)[1]

```
gdbtk factors04
```

Do not start gdbtk in the background[2] as program I/O from the Ada program being debugged will be directed to the terminal window.

After a short delay a number of windows will appear: a **Source window** where the source code of the program will be displayed, a **Commands window** where `gdb` commands can be entered and where `gdb` output appears[3]. The source window also contains a number of buttons for the most frequently used debug commands and a number of menus. Arrange the windows on the screen so that all are visible.

The program `factors04` is now ready to be run under debugger control. Do this by clicking on the **Start** button in the **Source window**. Do NOT use the **Run** button as this is will run the program 'at full speed' as no breakpoints have been set to allow the debugger to gain control.

The program `factors04` is launched and the debugger gains control just as the main Ada procedure is about to start. The Source window displays the source file `factors04.adb` with the first executable statement in highlighted in blue. 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.

Program I/O

As `Put` and other output procedures are executed, their output appears in the terminal window. Similarly, as input procedures (such as `Get`) are executed, you must type the required data into the terminal window and press `<Return>`.

Arrange the **Source**, **Command** and **Terminal** windows on the screen so that you can see them all.

[1] Don't use the Emacs Ada menu command **Debug** because this currently start a different debugger gvd. The version of gvd on PC-Solaris machines does not currently work properly.
[2] That is don't terminate the command with an ampersand &.
[3] Do NOT close the Commands Window as this may cause problems.

Single-stepping
The simplest way to use the debugger is to 'single-step' the program from the start (that is execute it one line at a time), checking that the program is behaving as you expect. There are two single-step commands **Step** and **Next** both available on buttons in the **Source** Window.

These commands behave in the same way for most Ada commands: the instruction is executed and the debugger regains control. However they behave differently for **procedure** calls:

Step steps into the procedure body allowing the procedure body to be executed step by step. **Next** steps over the procedure regarding it as a single step and the debugger regains control when execution of the procedure is complete.

Step is useful if you suspect that there may be a bug in the procedure whereas **Next** is useful if you are confident that a procedure is bug-free. For calls to standard Ada library procedures **Step** and **Next** both step over the procedure call[4]. This is sensible since you can be reasonably confident that the standard libraries are bug free.

After you have executed the step

```
      Put(Item => "Type in a number greater than 1 (0 or 1 to quit): ");
```

notice how the output appears in the terminal window. When you step past the command

```
      Get(Item => Number);
```

you will need to type a suitable value (12 say) in the terminal window and (of course) press `<Return>`.

Inspecting the Value of Variables
When the debugger is in control the value of program variables may be inspected. To see the value of a variable:

in the **Source Window, double-click on the variable**

The **Expressions Window** will appear with the variable's value displayed in a sub-window.

Display the variables `Number`, `TrialDivisor` and `NumOfDivisors` by doubling clicking on each variable in turn in the source Window. In the **Expressions Window**, arrange the boxes for each variable so that the contents of each are visible. 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.

Step through the program until its terminates normally.

Setting Breakpoints and using Continue
Sometimes it can be tedious to single-step all the way through a lengthy program. If the program contains a large loop, thousands of instructions may have to be executed. We would like to be able to run the sections of the program at 'full speed'[5] and then have the debugger gain control when a certain program step is reached. To do this we need to set a breakpoint at the required point in the program.

To set a breakpoint at a particular point in a program click on a **line number** at the appropriate point in the **left-hand edge of the Source Window**. A **red cross** will appear next to the line number indicating that a breakpoint is set. To clear a breakpoint simply click on the line number which will change back to its original state[6].

---

[4] unless the library has been compiled with the debug option `-g`.
[5] Actually programs run somewhat slower when under debugger control, but still at a very fast speed in human terms.
[6] You can also create and delete breakpoints by using the **Create** and **Delete** buttons in the **Breakpoints** window.

When a program is idle with the debugger in control it can be resumed at 'full speed' by clicking on the **Continue** button in the Source window. The program will run at 'full speed' until it terminates or until a **Breakpoint** is encountered, at which time the debugger will regain control and then program variables can be inspected.

Start the program again by clicking the **Start** button and set a breakpoint at the start of the inner `WHILE` loop (line 31). Then run the program at 'full speed' by clicking on the **Continue** button, enter a number in the terminal window when prompted to do so and then note how the program stops at the breakpoint and displays the current values of the program variables. Clear the breakpoint by clicking on line 31 again and click **Step** to advance past line 31 and then click on line 31 to reset the breakpoint. You can then single-step through the factorisation loop by clicking repeated on **Step** watching the values of the variables change as program execution proceeds. Alternatively you can run the loop at 'full speed' by clicking on **Continue**, the debugger will gain control again when the breakpoint at line 31 is reached again.

To proceed past the breakpoint a second time, clear the breakpoint by clicking on line 31 again and click **Step** to advance past line 31 and then click on line 31 to reset the breakpoint. Continue in this way until the program terminates. Of course if you don't reset the breakpoint and click **Continue** the program will run at 'full-speed' until it terminates.

Important Note It <u>should be possible</u> to simply **Step** or **Continue** past a breakpoint without clearing and then resetting it. However with the current version installed, if you attempt to continue or step past a breakpoint that is set, the debugger will crash and not respond to further button clicks -- this is a bug which should disappear when a newer version of `gdbtk` is installed. In the meantime always remember to clear the breakpoint, single-step past it and then reset it (if necessary).

Quit the debugger by selecting **Quit** from the file menu.

Tracing Programs containing Procedures

Compile the program `savings08.adb` for use with the debugger (i.e. use the option `-g` with `gnatmake`) and then start `gdbtk`:

```
gdbtk savings08
```

Click **Start** and then step over the `Introduction` procedure by clicking **Next**. Note how the code for the procedure is executed at full-speed with output appearing in the terminal window. Display the main program variables `TotalSavings`, `MonthlySavings` and `PlanLength` in the Expression window by double-clicking on each variable in turn (you will have to scroll the source window to line 100 or so first). Note how these variables have 'junk' values as they have not yet been initialised.

Now click **Step** to step into the code for the procedure `GetPlanInfo`. Note how the code for this procedure is displayed.

The Procedure Call Stack

During program execution the main Ada procedure may call a procedure `a` which in turn may call another procedure `b`, procedure `b` in turn can call a third procedure `c` and so on. The list of active procedure calls, that is those that have been called but which have not yet returned, is called the **call-stack**. At any stage you can inspect the call stack by:

> opening the **Stacks Window** by selecting **Stacks** from the **Window** menu of any GDBTK window. Do this now!

Suppose that (as above) we have single-stepped through the `savings08` program so that the main procedure `Savings` has called procedure `GetPlanInfo`. The stacks window shows information of the form[7]:

```
Lev   Lang  Location             Function
0     ada   savings08.adb:45     savings.getplaninfo
1     ada   savings08.adb:108    savings
```

[7]Note `gdb` displays all identifiers in lower case.

```
2      ada    b~savings08.adb:170     main
```

indicating that the procedure `getplaninfo` is paused at line 45 in the file `savings08.adb` and that `getplaninfo` was called from procedure `savings` (at line 108 in the file `savings08.adb`) which in turn was called from the procedure `main` (at line 170 of the bind file `b~savings08.adb`).

The procedure which is at the bottom[8] of the call stack (`getplaninfo` in this case) is highlighted and the source window will display the source code of the `getplaninfo` procedure.

Each level in the call-stack is called a **frame**. If we want to display the source code of the main procedure `savings` we can click on the frame of `savings` and the source code window will show the source code of `savings` with the call to `getplaninfo` highlighted. Similarly if we click in the frame of `main`, the source code window will show source code of the bind file with the call to `main` highlighted (usually the bind file is of little interest to the Ada programmer).

Suppose we want to display a local variable or formal parameter of `getplaninfo` (`LumpSum`, say). If we attempt this when in the frame of `savings` or the bind file, GDBTK will respond that there is no variable `LumpSum` in the current context. To get GDBTK to display `LumpSum` we must move into the frame of `getplaninfo`.

To summarise a local variable or formal parameter can only be displayed when GDB is in that variable's frame, that is when that local variable is **in scope**. However to display the value of a main program variable, we can be in the frame of `savings` (or `getplaninfo` or of any other procedure) as the variable is in scope throughout the program.

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 and the corresponding box in the **Expressions Window** will be blank.

We can also navigate up and down the call-stack using the **Up**, **Down** and **Bottom** buttons in the **Source** Window.

Display the values of the formal parameters of procedure `GetPlanInfo` by double-clicking on each of these variables in the Source window, note that these variables also have 'junk' contents as they have not been initialised. Then single-step through the code of `GetPlanInfo` using the **Step** button and enter values in the terminal window when prompted. Note that as the procedure `GetPlanInfo` returns the values of the formal parameters are copied out to the corresponding actual parameters and that the boxes corresponding the formal parameters become blank to indicate that these parameters no longer exist once the procedure has 'returned'. Note it takes two clicks of the **Step** button to return from the procedure: one click for the return and one click for the copy out of parameters.

Continue stepping through the program and display the local variables and parameters of each procedure in turn. Note also how the call stack changes as procedures are called and return.

Continue single-stepping until the program terminates normally or alternatively use continue to complete execution of the program at full-speed. Then quit the debugger.

At any time you can quit by selecting **Exit** from the **File** menu of any GDBTK window. If a program is currently being run under the control of the debugger the program will ask for confirmation before quitting.

[8] The call stack is displayed with the bottom of the stack at the top!!