C Programming under Linux

P2T Course, Semester 1, 2004–5
Linux Lecture 7

Dr Graeme A Stewart

Room 615, Department of Physics and Astronomy

University of Glasgow

`graeme@physics.gla.ac.uk`

Summary

- The GNU Debugger
- Display Data Debugger
- Programing Tips
- The End of the Beginning

`http://www.physics.gla.ac.uk/p2t/`

`$Id: linux-lecture-07.tex,v 1.2 2003/12/02 12:30:12 graeme Exp $`

Debugging C Code

- C programs are hard to write correctly. Programs will often require debugging simply to get them to compile. Fortunately this stage of debugging is quite easy – syntactic errors are straightforward to correct.
- Once a program actually compiles (and warnings have been examined to ensure they don’t correspond to actual errors) debugging in earnest begins – now you must verify that the program behaves correctly.
- This stage of debugging is much harder than the syntactic debugging at compile time. There are infinitely many programs which compile and run properly, but do not give the correct answer. (Satellites have been lost because control programs had an extra minus sign in a mathematical formula!)

GCC Debugging Flag

- To help with debugging `gcc` has a debugging flag, `-g`, which inserts extra symbol information into the compiled binary which allows it to be tagged closely to the originating compiled code.
- Note, the `-g` flag does not make the program run more slowly (it is larger though). It is also possible to have debugging and optimisation, `-O` or `-O2`, flags together.
- Usually you can adapt a `Makefile` to easily switch debugging on an off:

```
CFLAGS=-O2 -Wall
CFLAGS+= -g     # <- Comment this line in and out as required
```

or

```
CFLAGS=-O2 -Wall
ifdef DEBUG
  CFLAGS+= -g      # Call make using "DEBUG=1 make"
endif              # to switch on debugging
```

Using the Debugger

The GNU debugger is called `gdb`. It's a command line debugger, started with `gdb PROGRAM`:

```
indigo:~/solar/mhd_wind/model$ gdb sym
GNU gdb 6.0-debian
Copyright 2003 Free Software Foundation, Inc.
(gdb)
```

What does the debugger do? From `man gdb`:

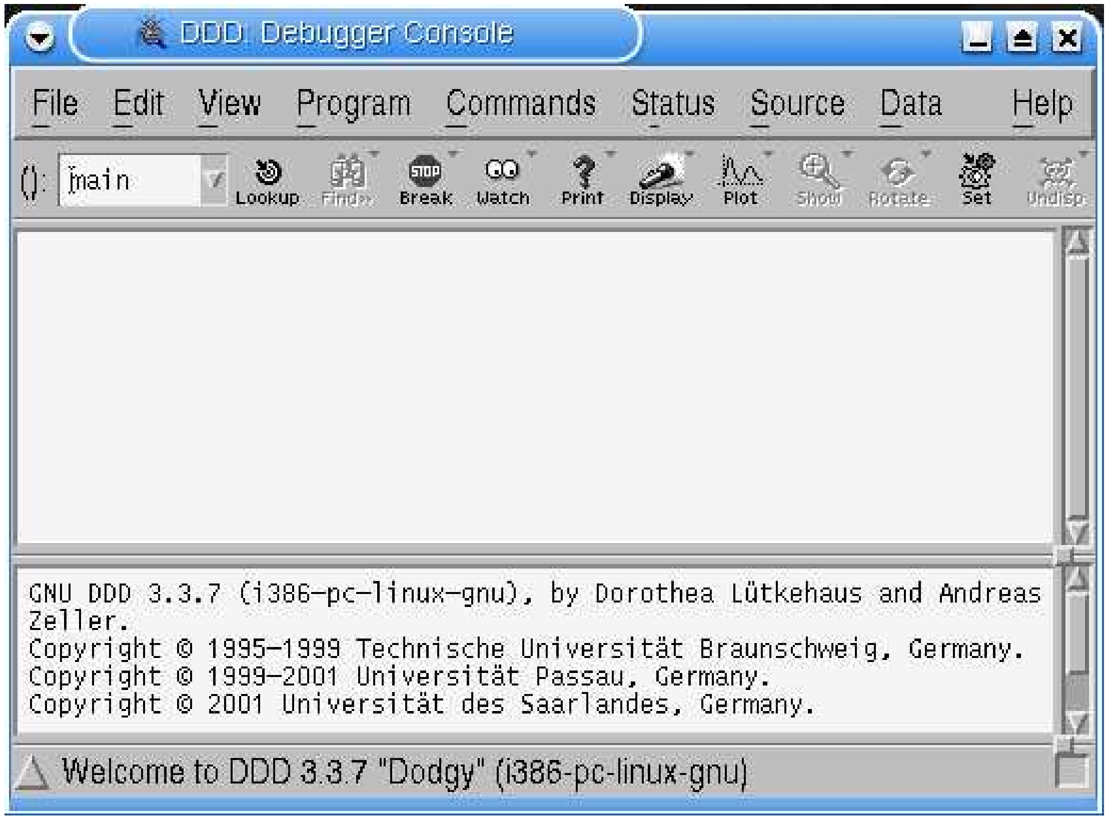
```
GDB  can  do four main kinds of things (plus other things in support of
these) to help you catch bugs in the act:

  -   Start your program, specifying anything that  might  affect  its
      behavior.

  -   Make your program stop on specified conditions.

  -   Examine what has happened, when your program has stopped.

  -   Change  things  in your program, so you can experiment with cor-
      recting the effects of one bug and go on to learn about another.
```

Command Line Debugging

- All of the power of `gdb` is avaliable on the command line. This does, however, have a pretty steep learning curve (which we don't have time to scale in these lectures. . .).
- As ever, the info node for `gdb` is the source of much wisdom about the debugger – `info gdb` also contains tutorial information to help you get started: look at the `sample session`.
- If you do want to use `gdb` as a command line debugger, then running it in an XEmacs window, using `M-x gdb` can be useful – XEmacs understands a lot of `gdb` output and can follow execution through your source files.

Display Data Debugger

- Fortunately there is a nice graphical interface to gdb, called the *Display Data Debugger*, or ddd.
- It can be invoked through the KDE menu (under ‘Development’), or from the command line as ddd.

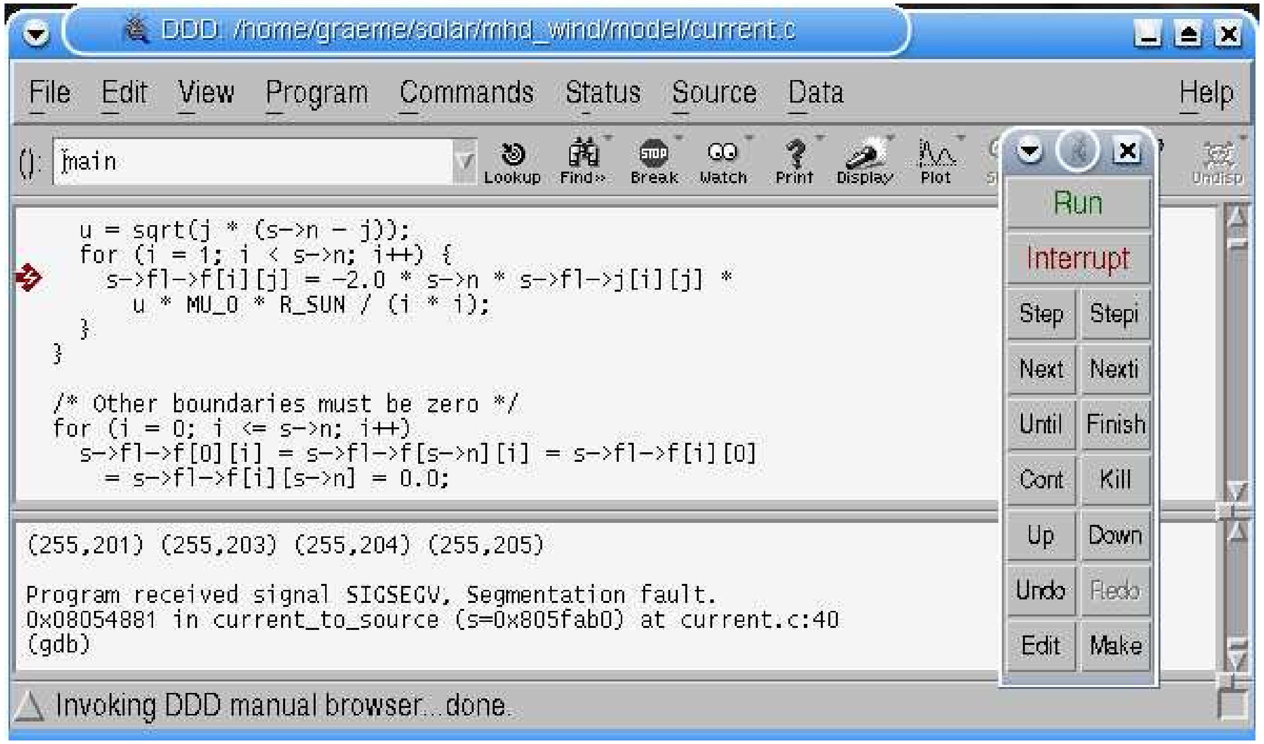

DDD: Debugger Console
File Edit View Program Commands Status Source Data Help
(): main
Lookup Find» Break Watch Print Display Plot Show Rotate Set Undisp

```
GNU DDD 3.3.7 (i386-pc-linux-gnu), by Dorothea Lütkehaus and Andreas
Zeller.
Copyright © 1995-1999 Technische Universität Braunschweig, Germany.
Copyright © 1999-2001 Universität Passau, Germany.
Copyright © 2001 Universität des Saarlandes, Germany.
```

Welcome to DDD 3.3.7 "Dodgy" (i386-pc-linux-gnu)

Loading and Running a Program

- Load a into ddd using the `File->Open Program` menu.
- To run the program in the debugger, use `Program->Run` – arguments can be added, then the program will be run. In the gdb window `STDOUT` and `STDERR` will be shown, and any `STDIN` can be entered.
- If the program crashes ddd will stop and display the line that crashed the code:



```
    u = sqrt(j * (s->n - j));
    for (i = 1; i < s->n; i++) {
      s->fl->f[i][j] = -2.0 * s->n * s->fl->j[i][j] *
        u * MU_0 * R_SUN / (i * i);
    }
  }

  /* Other boundaries must be zero */
  for (i = 0; i <= s->n; i++)
    s->fl->f[0][i] = s->fl->f[s->n][i] = s->fl->f[i][0]
      = s->fl->f[i][s->n] = 0.0;
```

```
(255,201) (255,203) (255,204) (255,205)

Program received signal SIGSEGV, Segmentation fault.
0x08054881 in current_to_source (s=0x805fab0) at current.c:40
(gdb)
```

Invoking DDD manual browser...done.

Examining The Code Status

- Hovering the mouse over a variable in the source window, will display its value. Right click to view more options – particularly `Print` and `Display` options for variables are very useful.
- The `Status->Backtrace` option will show the sequence of function calls made before the program crashed:

  ```
  (gdb) bt
  #0  0x08054881 in current_to_source (s=0x805fab0) at current.c:40
  #1  0x080490e8 in main (argc=2, argv=0xbffffbf4) at sym.c:176
  ```

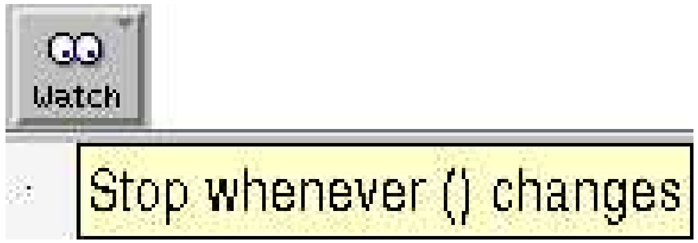
  Note that the menu option actually passes the `gdb` command down to the running `gdb` process – which is a good way of learning the command line option, `bt` in this case.
- The `Status->Up/Down` will move up and down the program's stack – in and out of the calling functions.
- The value of variables when the program crashed should give valuable information about why the program failed.

Setting Breakpoints

- Sometimes analysing a crash doesn’t tell you why a program failed.
- It’s possible to stop the execution of a program at a particular place – this is called setting a *breakpoint*.
- To do this in `ddd` select the line of code to break at, and press the break button: STOP Break.
- When the program reaches the breakpoint execution will stop.
- The value of variables can be examined, and the program’s execution can be *stepped*, one source line at a time through the `Program` menu.
- The difference between `Step` and `Next` is that `Step` enters subroutines, `Next` skips over them.
- To run the program normally again after a break has been reached, use `Continue`.

Setting Watchpoints

- If you want to examine a particular variable, rather than a source line, a *watchpoint* can be set:



- This will stop the program's execution when the value of the variable changes.
- As with breakpoints, once the program has stopped it can be `Stepped`, `Nexted` or `Continued`.

Core File Analysis

- When a program crashes in unix it will dump a file called `core` – this is an image of the file's memory core as it crashed. (Note, a core file will only be produced if the shell's `ulimit -c` is less than the core file size – use `ulimit -c unlimited` to always get a core file.)
- A core file can be loaded into `ddd` by selecting `File->Open Core Dump`. (The program file itself has to be loaded first.)
- Once a core file has been loaded, the crash can be analysed normally – the stack can be traced and variable values examined.

Advanced DDD Use

- We’ve only scratched the surface of the debugging facilities that `gdb` and `ddd` offer (although often that surface is enough!).
- For further help, use the `Help->DDD Reference` option, and references in the `man` page.

Top Programing Tips

It was a tradition in the temple that the Great Zen Programing Master (aka ‘Keeper of the Kernel’, and the only programmer ever to sucessfully dereference a `NULL` pointer), would gather the novices around him of a summer’s evening. There they would sit and watch the stream trickle and the rabbits hop around the meadow.

Frankly, they were terribly bored by this, and would chaff to return to the keyboard to continue to wrestle with pointers. The Zen Master would smile and watch them shuffle.

Sometimes he would sit for hours and say nothing. Sometimes he would talk of redirection operators and subshell execution. Sometimes he would discuss his favourite recipes for rabbit stew. Sometimes he told them of ways to improve their `C` code.

One of the smarter novices had a Palm Pilot, and would scribble down the Master’s thoughts with that terrible stylus (which always mixed up ‘s’ and ‘c’).

Here follows some of the things he wrote (though the Master hacked his Palm Pilot and deleted the recipe for rabbit stew, which is now lost to mortals). . .

Do You Really Need To Scanf?

- Notice how terse and efficient most unix utilities are.
- You should aim to write programs in the same style.
- So this means *do not* read input from the keyboard unless it's really necessary. Don't write code like this:

```
$ myfoo
Please input number of electrons: 1000
Is field linear(1) or log(2): 2
Ionization threshold: 0.155
Time steps: 10000
Output file: 1klog.dat
[....]
```

- It makes your program slow, tedious to use and impossible to script.

Don’t Fix Parameters In Code

- Even worse than asking for parameters is having them fixed in the code:

```
int main() {
  long int n_elec=10000, t_steps=1000;
  int field_type=2;
  float ionization=0.155;
  char *output="1klog.dat";
```

- If you do this you have to recompile to run the program with new options!
- And again, it cannot be easily scripted.
- By all means have sensible *default* values, but do have a means of changing them.

Specify Options on the Command Line

- Code that works like this is good:

  ```
  $ myfoo 10000 2 10000 1klog.dat
  [...]
  ```

  Here the arguments are given in a set order.

- This then becomes very easy to script:

  ```
  $ for ne in 100 1000 10000 100000; do
  >   myfoo $ne 2 10000 log-$ne.dat
  > done
  ```

- Another possibility is just to read the options from a file that is specified on the command line: `myfoo run1.options`

- If other people are going to use your code, it's probably worth using command line switches, rather than a fixed argument set:

  ```
  $ myfoo --numelec=10000 --field=2 --tsteps=10000 --output=1klog.dat
  [...]
  ```

- Scanning these options takes a little more work, but the library functions `getopt()` and `getopt_long()` will help you.

Write A Logfile

- If your program produces output you might be looking at months later, don't rely on remembering the parameters you passed.
- Write a logfile which shows how the code was called:

  ```
  snprintf(logfile, FILE_NAME_SIZE-1, "%s.log", file_name_root);
  if ((LOGFILE=fopen(logfile, "w")) == NULL) {
     fprintf(stderr, "Failed to open logfile %s\n", logfile);
     exit(EXIT_FAILURE);
  }
  fprintf(LOGFILE, "Options: %ld %d %ld %s\n", n_elec, field, tsteps, outfil
  ```

  It's a good idea to record exactly the set of command line options used, so that runs can be reproduced exactly.
- If your program's not of this type, still consider a logfile if, say, a debugging flag is set.

Long Programs Should Save State

- If your program takes more than a few hours (days?) to run you cannot rely on the system being up until your program finishes.
- Power failures, system shutdown and even code bugs might cause your program to terminate unexpectedly early.
- Your program should save its state – writing out the current values of all variables in such a way as the program can easily be restared from the saved state.
- You may want to get your program to trap signals. e.g. `SIGHUP` to exit gracefully; `SIGUSR1` to dump state immediately, etc. Read the `libc` info node on signals to get started.

Never Write Numbers in ASCII

- Always read and write numbers in *binary* form.
- Use the `fwrite` and `fread` functions.
- Writing floats and doubles in ASCII is bad because:
  - it's slow
  - it loses precision in the conversion
- If you have to analyse your final data with a data visualisation tool which doesn't like binary format numbers (e.g. `gnuplot`), consider writing out the binary numbers
  - alongside an ASCII file
  - and having a separate Binary->ASCII converter program

Use CVS To Store Your Source Code

- When editing code, extending and experimenting, it's easy to make changes which break the program.
- If you undo these changes and the program is still broken then you've made some other changes you have forgotten, or your program was broken to begin with, you just didn't realise it.
- Both of these scenarios are frustratingly difficult to recover from.
- CVS, *Concurrent Version Systems*, is a way of keeping past revisions of code in a form that can be easily recovered.
  - Known working versions can be tagged and restored.
  - Differences with current code can be easily found.
  - It makes having two versions of the code (on your desktop and laptop) easy to work with as it can merge changes.
  - See `info cvs` for help on getting started.

Profile To Improve Code Speed

- If you need to make a slow program run faster compile with the `-pg` option.
- This compiles it with *profiling* code.
- When the program runs it writes a file called `gmon.out`, which describes what fraction of time the code spent in each function or on each code line.
- The *profiler* `gprof` will examine this file after the run has finished.
- `gprof` will let you know where the program spends most of its time, so that you can concentrate on optimising these pieces of code.

Write Regression Tests

- When you write a function, write a test program which uses the function in certain known ways – and checks that the function returns the correct values.
- You'll have much more confidence in your code if you do this.
- It'll save a lot of time in tracking down bugs.
- If you can regression test the whole code, do so, but make sure you have component testing as well.

The End of the Beginning

- Unix *is* friendly! It's just picky about who its friends are. . .
- `UNIX` is a four letter word, `vi` is a two letter abreviation, `Linux` is a five letter abomination!
- 'We all know Linux is great. . . it does infinite loops in 5 seconds.' (Linus Torvalds about the superiority of Linux on the Amterdam Linux Symposium)
- Now I know someone out there is going to claim, 'Well then, UNIX is intuitive, because you only need to learn 5000 commands, and then everything else follows from that! Har har har!' (Andy Bates in comp.os.linux.misc)
- It's now the GNU Emacs of all terminal emulators. (Linus Torvalds, regarding the fact that Linux started off as a terminal emulator.)

Have fun with Linux!

Copyright

All these notes are Copyright (c) 2003, Graeme Andrew Stewart.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

A copy of the license is here:

http://www.physics.gla.ac.uk/p2t/fdl.txt