Debugging tools – a short guide

ARCQP 2016/2017

September 18, 2016

Contents

1 Introduction

There are several methods for debugging a program, but debugging should be avoided. In oder to avoid debugging one must:

1. Think very well the algorithms used in the programs.

2. Use the warning facilities of the compilers, and correct all the warnings after understanding the real source of those warnings.

To help finding errors in programs, one can use static analysis tools that make a deeper source checking, warning of possible problems. One of those tools is `splint`[1]. To analyze a program one uses the command line "`splint prog.c`".

To detect issues with dynamic memory use or pointer troubles, other type of tools is needed. The `valgrind`[2] tool analyses the run-time behavior of an executable program, checking it's memory use (among other things). The typical command line would be "`valgrind ./prog`" to analyze the memory behavior of `prog`.

Both `splint` and `valgrind` are very useful, because besides showing the origin or cause of already detected errors, allow the detection of errors that a simple program execution will never show.

[1] `http://www.splint.org/`
[2] `http://valgrind.org/`

2 Strategies and additional references

An excellent (and fun) book about debugging strategies is (Agans, 2002), but it talks about both hardware and software bugs and not only software bugs. In (Grötker et al., 2008) one can find an edited shortlist of the book's tips:

1. Understand the requirements and the system – Understand how C works, how a program runs and the use of the development tools; perhaps the program is right and you are wrong or you are doing a wrong interpretation of the requirements
2. Cause a failure – To know how and when errors appear is fundamental
3. Simplify the test case – Know the contributing error factors
4. Read the right error message – The issue could be in an unread (previous) error message (or warning)
5. Stop thinking and look – Before inventing theories look at what is happening, get facts instead of interpretations
6. Divide to conquer – Break the program/system into parts to contain the error into one of those parts
7. Use the right tool – Know the tools available, and how to use them
8. Change only one thing at a time – In order to know what has really changed
9. Keep an audit trail – In paper or in a file, because you will not remember everything
10. Check the plug – Start by checking the basic and fundamental stuff
11. Get a different viewpoint – Ask somebody for help, but tell them facts, not theories
12. Check if the error has been corrected – If the error has only disappeared, it has not been corrected
13. Build a regression test for it – This will avoid a new appearance of the error

There are many books about debugging, but some works cover only a specific platform, others are purely theoretical or are only focused on how to use a specific tool. In (Zeller, 2009) one can find a deep and serious analysis of how to debug a program, while in (Matloff/Salzman, 2008) the focus is centered on the practical use of tools. In (Grötker et al., 2008) there is a wide coverage of the subject, from debugging strategies to a list of useful tools.

3 Using gdb

`gdb` is the most common used debugger in Unix based systems and in many microprocessor based systems. An illusory disadvantage of `gdb` is the text based interface, using a simple command line. To side step that, several programs and development environments use `gdb`, *hiding* it behind a more beginner friendly interface. As examples, we have with GUI interfaces `ddd`, `kdbg` and the IDE's Eclipse and Netbeans.

But the command line use is available in systems where there is no graphical interface, the use of a graphical interface is computationally heavy or, we want to use `gdb` features unavailable in the graphic interface.

Preparing the programs for debugging

The programs to debug must be compiled with the `gcc`'s `-g` option order to include in the executable the required information about the symbols in the source code. Problems may arise with C code when the `-g` option is specified in the existing `Makefile` but due to `Makefile` errors, the desired rules are not triggered, and `make` uses it's default rules (without debug info) for building the executable.

In assembly, if a source code file has not the different section's correctly defined, the program may run but `gdb` will not be able to debug it.

In order to remove the debugging information from an executable file, resulting in a smaller file size one can use the `strip` command.

Normal Use

`gdb` takes as a parameter the name of the program to run, or if we start it without parameters,one can load a programm using the command `file`.

run runs the program

- Examples:

  `run`

quit quits `gdb`

- Examples:

  `quit`

next executes the next line (does not go into functions–*step over*)

- Examples:

  `next`

  `next` *n*

step executes the next step (goes inside functions–*step into*)

- Examples:

  `step`

  `step` *n*

finish finish the function where we are

- Examples:

  `finish`

continue continues executing up to the next *breakpoint* or until the program ends, or continues ignoring *n* times the current *breakpoint*

- Examples:

  `continue`

  `continue 3`

list lists the source code

- Examples:

```
list
list 3,10
```

break places a *breakpoint* in a line, in a function or an address

- Examples:

```
break main
break fich.c:12
break *pointer
```

watch places a watchpoint in a variable or uses an expression as a watchpoint

- Examples:

```
watch x
watch x>2
watch (*p)==y
```

delete/enable/disable deletes/enables/disables a *breakpoint* or *watchpoint*

- Examples:

```
delete 1
enable 2
disable 2
```

print prints the value of a variable or the result of a function

- Examples:

```
print x
print sizeof(int)
print my_func(3,3)
print $eax
```

display activates the display of variables in each execution step

- Examples:

```
display x
```

undisplay disables the display of variables

- Examples:

```
undisplay n
```

set change the value of a variable/register

- Examples:

```
set x=2
set $eax=4
```

backtrace shows the active *frames* of a program (with the local variables for each one)

- Examples:

  ```
  backtrace
  backtrace full
  ```

info shows information about different items

- Examples:

  ```
  info frame
  info reg
  info locals
  ```

Tips:

In the use of `gdb` there is a series of small details that might help or infuriate you.

- One cannot run a program step by step if the program is not running already. To do that, first place a *breakpoint* in a line on the beginning of the program, do `run` and when the program hits the *breakpoint*, one can do `step` or `next`.
- *Watchpoints* might be more useful than *breakpoints*, because they allow stopping a program when data changes. They allow data centered debugging, when we have no idea where is the wrong code, but we know which variables have the wrong values.
- The `print` command is useful for printing the value of variables but also to call functions.
- In order to print to use the right format, one can use the type casting facilities of the C language. Simple examples:

  ```
  print (char) 65
  print (int) 'a'
  ```

Practical examples

`gdb`

Now are several examples of the commands/keys needed to debug an hypothetic program (`prog`).

1. Command line: `gdb prog`
2. Place a *breakpoint* on the`main` function: `b main`
3. Run the program: `r`
4. Step by step: `s`

DDD

1. Command line: `ddd prog&`
2. Place a *breakpoint* on the `main` function: *Right-click* with the mouse on `main` in the source code and choose *break*.
3. Run the program: Menu Program » Run or F2
4. Step by step: Menu Program » Step or F5

Light user interface

In `gdb` there is a mode with text based user interface (tui), very useful to see at the same time the registers of the processor and the source code of the program:

1. First increase to the maximum the size of the terminal
2. Invoke `gdb` with : `gdb -tui prog`

   Or if `gdb` has already started: `tui enable`

   In alternative one can use Ctrl + x Ctrl + a to alternate between modes
3. `layout regs` – show the processor's registers (nothing when the command is done)
4. `b main` – place a *breakpoint* on the `main` function
5. `r` – shortcut for *run* – stops on the *breakpoint*
6. `s` – shortcut for *step* – executes the program step by step

If the terminal window becomes garbled, one can refresh it with Ctrl + L.

References

Agans, David J. – *Debugging: The Nine Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems*. New York: AMACOM–American Management Association, 2002 ⟨URL: `http://www.debuggingrules.com`⟩. ISBN 0–8144–7168–4

Grötker, Thorsten et al. – *The Developer's Guide to Debugging*. New York: Springer, 2008 ⟨URL: `http://www.debugging-guide.com/`⟩. ISBN 978–1–4020–5539–3

Matloff, Norman; Salzman, Peter Jay – *The Art of Debugging: with GDB, DDD and Eclipse*. San Francisco: No Starch Press, 2008 ⟨URL: `http://www.nostarch.com/debugging.htm`⟩. ISBN 1–59327–002–X

Stallman, Richard; Pesch, Roland; Shebs, Stan – *Debugging with GDB*. 9th edition. Boston, MA, USA: Free Software Foundation, 2010 ⟨URL: `http://sourceware.org/gdb/onlinedocs/`⟩. ISBN 1–882114–77–9

Zeller, Andreas – *Why programs fail: A guide to systematic debugging*. 2nd edition. Burlington, Massachusetts: Morgan Kaufmann Publishers, 2009. ISBN 978–0–12–374515–6