exact

Stefano Cozzini

CRS – OGS Udine – 26 Novembre 2019

The art (?) of debugging with gdb debuggers

Debugging

The process of identifying the cause of an error and correcting it

Once you have identified what the defects are, you must

- find the cause
- remove the defect from your code

A lot of programmers don't know how to debug!

20 to 1 difference in the time it takes experienced programmers to find and fix the same set of errors

Debugging (2)

- Debugging doesn't improve the quality of your software
  - It's a way to remove defects
- You get better results by doing the design process properly
  - Requirements analysis
  - Good design
  - High quality programming practices
- Debugging is a last resort

Errors are Opportunities

- Learn from the program you're working on
  - Errors mean there's something you don't understand about the program
  - If you knew it perfectly, it wouldn't have an error, You would have fixed it already
- Learn about the kind of mistakes you make
  - If you wrote the program, you inserted the error
  - Once you find a mistake, ask:
    - Why did you make it?
    - How could you have found it more quickly?
    - How could you have prevented it?
    - Are there other similar mistakes in your code? Can you correct them now, instead of waiting to find them later?

Errors are Opportunities

- Learn about the quality of your code from the point of view of someone who has read it
  - Look critically at your code
  - Is it easy to read?
  - How could it be better?
- Use your discoveries to improve the next code you write
  - Learn about how you solve problems
  - Is your approach to debugging productive?
  - Do you need to improve it?
  - You can spend hours and hours (even days, months) on debugging
  - Taking some time to think about how you are debugging is not going to take much more time
- Learn about how you fix errors
  - Do you apply goto band-aids?
  - Do you fix special cases?
  - Do you make systemic corrections?

Devil's guide to Debugging

- Find the error by guessing
- Scatter print statements randomly throughout the program
- Examine the output to see where the error is
- If you can't find it that way, try changing things until something seems to work
- Don't keep track of what you changed
- Don't backup the original
- Don't waste time trying to understand the problem
- It's likely that the problem is trivial, and you don't need to understand it completely to fix it
- Fix the error with the most obvious fix
- It's usually good just to fix the specific problem you see, rather than wasting a lot of time making some big ambitious correction that is going to affect the whole program

Debugging tools..

- Source code comparator/ revision system (git!)
  - helps you find where you changed the code
  - look up the vimdiff/tkdiff program
- Compiler warning messages
  - Set the compiler warning level to the highest level, and fix the code so that it doesn't produce any warnings!
  - Treat warnings as errors:
    - Gcc flags -Wall
  - Enable all the warnings which the authors of cc believe are worthwhile. Despite the name, it will not enable all the warnings cc is capable of.

Debugging tools (2)

- Extended syntax and logic checking
- GCC flags:
  - --ansi : Turn off most, but not all, of the non-ANSI C features provided by cc. Despite the name, it does not guarantee strictly that your code will comply to the standard.
  - -pedantic: Turn off all cc's non-ANSI C features.
- Execution Profiler
  - Programmer errors can cause bad performance as well as bad output
  - Identify routines that take up a disproportionate amount of execution time

Debuggers

- The purpose of a debugger:
  - allow you to follow the execution of a program
  - help you to understand what a program was doing at the moment it crashes.
- To use debugger you have to generate debugging info at compilation time:
  - compile with -g option
  - all the debugging info are stored in the *.o files
- Lots of debuggers:
  - graphical debuggers:
  - user friendly
  - text debuggers:
    - not user friendly but almost always available

What a debugger should do

- Start your program, specifying anything that might affect its behavior
- Make your program stop at specified conditions
- Examine what has happened, when your program has stopped
- Change things in your program during execution, so you can experiment with correcting the effects of one bug and go on to learn about other

What is not able to do

- Even though GDB can help you in finding out memory leakage related bugs, but it is not a tool to detect memory leakages.
- GDB cannot be used for programs that compile with errors and it does not help in fixing those errors.

Gdb

GDB A GNU source level debugger

  - portable
  - efficient
  - it has some GUI

- To use it:
  - interactive way: just launch the program and then load the executable
  - Postmortem: analyze what went wrong by means of the core dump file (a snapshot of the memory when program crash)
  - core files are automatically produced by all unless this feature is disabled by the user.
    - Check ulimit -c value

Generating symbol table for GDB

In order to include debugging information in your code, you need to compile it with –g family options (read the gcc manual for complete info):

`-g`

produce dbg info in O.S. native format

`-ggdb`

produce gdb specific extended info, as much as possible

`-glevel`

default level is 2. 0 amounts to no info, 1 is minimal, 3 includes extra information (for instance, macros expansion) – this allows macro expansion; add -gdwarf-n in case, where possibile, where n is the maximum allowed (4)

`-ggdblevel`

you can combine the two to maximize the amount of useful info generated

remember: `-fno-omit-frame-pointer`, especially if you are using `-Ox`

Stopping execution

- `break`
  - A breakpoint makes your program stop whenever a certain point in the program is reached. in the program.
  - `break FUNCTION`
    - Set a breakpoint at entry to function FUNCTION.
  - `break LINENUM`
    - Set a breakpoint at line LINENUM in the current source file.
  - `break FILENAME:LINENUM`
  - Set a breakpoint at line LINENUM in source file FILENAME.
  - `break ... if COND`
    - Set a breakpoint with condition COND;

Stopping execution (2)

- `watch`
  - `watch EXPR`
  - Set a watchpoint for an expression. GDB will break when EXPR is written into by the program and its value changes.
- `info`
  - Print a table of all breakpoints and watchpoints set

Running/resuming execution (1)

- `run`
  - Use the `run' command to start your program under GDB. You must first specify the program name with an argument to GDB, or by using the `file' or `exec-file' command.
- `step`
  - Continue running your program until control reaches a different source line, then stop it and return control to GDB. This command is abbreviated s.
- `step [COUNT] (shorthand s)`
  - Continue running as in `step', but do so COUNT times. If a breakpoint is reached, or a signal not related to stepping occurs before COUNT steps, stepping stops right away.

Running/resuming execution (2)

- `next [COUNT] (shorthand n)`
  - Continue to the next source line in the current (innermost) stack frame. This is similar to `step', but function calls that appear within the line of code are executed without stopping. Execution stops when control reaches a different line of code at the original stack level that was executing when you gave the `next' command. This command is abbreviated `n'.
- `continue [IGNORE-COUNT] (shorthand c)`
  - Resume program execution, at the address where your program last stopped; any breakpoints set at that address are bypassed. The optional argument IGNORE-COUNT allows you to specify a further number of times to ignore a breakpoint at this location.
- `finish`
  - Continue running until just after function in the selected stack frame returns. Print the returned value (if any).

Examining data and source code

- `print [EXP] (shorthand p)`
  - EXP is an expression (in the source language). By default the value of EXP is printed in a format appropriate to its data type; If you omit EXP, GDB displays the last value again.
- `display EXP (shorthand disp)`
  - Add the expression EXP to the list of expressions to display each time your program stops. `display' does not repeat if you press RET again after using it.
- `list (shorthand l)`
  - To print lines from a source file, use the `list' command (abbreviated `l'). By default, ten lines are printed.
  - `list LINENUM :`
    - Print lines centered around line number LINENUM in the current source file.
  - `list FUNCTION :` Print lines centered around the beginning of function FUNCTION.

Examining the stack

When your program has stopped, the first thing you need to know is where it stopped and how it got there.

Each time your program performs a function call, information about the call is generated. That information includes the location of the call in your program, the arguments of the call, and the local variables of the function being called. The information is saved in a block of data called a "stack frame". The stack frames are allocated in a region of memory called the "call stack".

`backtrace` (shorthand bt)

Print a backtrace of the entire stack: one line per frame for all frames in the stack.

Valgrind [http://valgrind.org/]

- Valgrind is an instrumentation framework for building dynamic analysis tools.
- The Valgrind distribution currently includes six production-quality tools..
    - A memory error detector, (memcheck)
    - two thread error detectors,
    - A cache and branch-prediction profiler,
    - A call-graph generating cache
    - A branch-prediction profiler
    - A heap profiler.

Valgrind

Valgrind basically runs your code in a virtual ‚sandobx' where a synthetic CPU (the same you have) is simulated and executes an instrumented code. There are various Valgrind based tools for debugging and profiling purposes.

•Memcheck is a memory error detector correctness

•Cachegrind is a cache and branch-prediction profiler velocity

•Callgrind is a call-graph generating cache profiler. It has some overlap with Cachegrind

KCacheGrind is a very useful GUI

Memcheck

- Detects memory-management problem and is aimed primarily at C and C++ programs.
- When a program is run under Memcheck's supervision, all reads and writes of memory are checked, and calls to malloc/new/free/delete are intercepted. As a result, Memcheck can detect if your program:
    - Accesses memory it shouldn't (areas not yet allocated, areas that have been freed, areas past the end of heap blocks, inaccessible areas of the stack).
    - Uses uninitialised values in dangerous ways.
    - Leaks memory.
    - Does bad frees of heap blocks (double frees, mismatched frees).
    - Passes overlapping source and destination memory blocks to memcpy() and related functions.
- Memcheck reports these errors as soon as they occur, giving the source line number at which it occurred, and also a stack trace of the functions called to reach that line.
- Memcheck runs programs about 10--30x slower than normal.

Using Valgrind

- COMPILING

  -
    ```
    gcc –g –fno-omit-frame-pointer my_prog.c –o my_prog
    ```
- RUNNING

  -
    ```
    Valgrind –tool=callgrind –callgrind-out-file=$CALLGRIND_OUT –dump-instr=yes –coolect-jumps=yes –cache-sim=yes –branch-sim=yes < --I1=… > < --D1=…> my_prog <args>
    ```
- ANALYZING

  -
    ```
    kcachegrind $CALLGRIND_OUT
    ```

Hands on session: please follow the tutorial

Final citation

- « Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it. »

(Brian Kernighan, The Elements of Programming Style)