Chapter 1 - Preliminaries

Advantages of GDB

- faster startup
- enables remote SSH debugging
- can manage multiple debugging sessions at once
- may need to debug GUI program without a GUI
- many additional features

General steps

- compile

```
gcc -g -Wall -o output_exe input_file.c
```

- *codesign GDB - https://www.ics.uci.edu/~pattis/common/handouts/macmingweclipse/allexperimental/mac-gdb-install.html
- gdb exe -tui
- *call `directory` once started to add directory of exe and debug symbols
- commands

```
run <prog args>
print <var name -> in scope> (p)
break <line num> (b)
condition <break point num> <condition -> var=val etc>
break <line num> if <condition>
run (r)
next (n)
list (l)
continue (c)
disp (d)
step (s)
bt -> backtrace
set <var = val>
call <function>
clear <break point number>
```

- stay in gdb while recompiling
- can use a startup file with gdb commands (~/.gdbinit)

```
gdb -command=z x
```

Chapter 2 - Stopping to Take a Look Around

- symbolic debugger relies on debug symbols to link machine instructions to source code

Pause

- breakpoint - pause at a location
- watchpoint - pause when a memory location (or related expression) changes value
- catchpoint - pause when an event occurs

Breakpoints

- assigns numbers to breakpoints
- list breakpoints:

```
info breakpoints
```

- remove breakpoints:

```
delete <breakpoint num>
```

- to create:

```
break <function>
break <line_number>
break <filename:line_number>
break <filename:function>
break <+offset or -offset> - ahead or behind current execution point
break <*address> - used for programs without debug info

tbreak -> temporary breakpoint
hbreak -> hardware assisted breakpoint
thbreak -> temporary hardware assisted breakpoint
rbreak -> sets breakpoints at start of any function matching a regular expression

catch -> triggered by things like thrown exceptions, caught exceptions, signals, c
         loading and unloading of libraries, and other events
```

- working with multiple files (best to create Makefile):

```
gcc -g3 -Wall -Wextra -c main.c swapper.c
gcc -o swap main.o swapper.o
```

- gdb has an idea of a focus, starts on file with main

- list code in a function

```
list <func name>
```

- breakpoints may move between edits and recompiles

Deleting breakpoints

```
delete <breakpoint_list>
delete -> deletes all
clear -> clears the breakpoint at the next instruction (use when stopped at a brea
clear <function>
clear <filename:function>
clear <line_number>
clear <filename:line_number>
```

Disabling breakpoints

```
disable <break num(s)>
enable <break num(s)>
```

Info about breakpoints

```
info breakpoints
Identifier - num
Type - breakpoint, catchpoint, watchpoint
Disposition - keep, del, dis (keep, delete after next encounter, discard after nex
Enable status - enabled or disabled
Address
Location - line in source code and file
```

Continuing execution

- next and step

```
n
s
```

- continue execution

```
continue
continue <num> -> ignores next n breakpoints
```

- continue to end of current stack frame

```
finish (fin)
```

- continue execution through current loop

```
until (u)
until <breakpoint arguments>
```

Conditional Breakpoints

- see pg 82 for more examples

```
break <break args> if <condition>
```

- can use int returning functions also
- to use non-int returning functions

```
set $p = (double (*) (double)) cos
ptype $p
p cos(3.14159265)
p $p(3.14159265)
```

Breakpoint Command Lists

- used to execute sets of commands after a breakpoint is reached

```
commands <breakpoint number>
<commands>
end
```

- any valid gdb expression, including library functions or functions within the executable can be used

Watchpoints

- pause when an expression changes value

```
watch <var> -> i.e. watch i
watch <expression> -> i.e. watch (i | j > 12) && i > 24 && strlen(name) > 6
```

- displayed with info as a hardware watchpoint

Expressions

- can be:
    - gdb convenience variables
    - any in-scope var from the running program
    - any kind of string, numerical, or character constant
    - pre-processor macros (must be compiled to include pre-processor) info
    - conditionals, function calls, casts, and operators defined by the language you're using
- NOTE: the manual says compiling with preprocessor debug info is not possible but can be done with `-g3`

Chapter 3 - Inspecting and Setting Variables

- print

```
p <var> -> can include member/function access, index access, pointer dereference,
```

- display -> automatically print an item at each pause in execution

```
disp <var>
```

- also helpful to use the commands command with printf, etc
- use the call command within the commands function

```
e.g.
commands <break num>
<command>
call function(args)
end
```

- printing a stack array

```
p <array>
```

- will not work with dynamic arrays, which can be solved two ways:
    - p <*pointer@number_of_elements>
    - p (int [25]) *x -> print with a specific cast

C++ considerations

- most commands work the same but with different output
- cannot inspect class types with any built ins but can review structure

```
ptype <var>
```

Examining memory

```
x <location>
```

Advanced print and display options

```
p/x <var> -> print hex
dis disp 1 -> temporarily disables display items
enable disp 1 ->
undisp 1 -> delete a display item
```

Setting variables

```
set x = 12
set args <args> -> sets command line arguments
```

GDB's variables

- $ and $ - value history variables
- $ - convenience variables
- names can be almost anything except $ and $
- can access registers with $

Chapter 4 - When a Program Crashes

Memory Management

- most common crash -> illegal memory access
    - seg fault on unix-like platforms
    - general protection fault on Windows

    - occurs when hardware supports virtual memory and it must be in use

- VM layout

    - unix generalization
        - .text
            - instructions generated from code
            - includes statically linked code
        - .data
            - global and static variables
        - .bss
            - uninitialized global and static variables
        - heap
        - unused
        - stack
        - env
        - *dynamically linked code is included somewhere depending on platform
    - can view process memory layout on linux by viewing /proc//maps file

- memory pages

    - defaults to 4096 on Pentium (older)
    - OS maintains a page table -> each process entries with
        - physical location of page on disk
        - permissions of page
    - no partial pages allocated

- page table accesses for running processes

    - global static variable access requires r/w access of data section
    - local variable access requires stack r/w access
    - function enter/exit requires stack access
    - malloc/new requires heap access
    - machine instructions require text section access

- TODO: more here

Core files

- gcc -g -W -Wall .c -o

- objdump -s core-file

- gdb or gdb -c

- gdb --args ./crash -p param1 -o param2

- creation

  - contains description of program's state when it died
    - contents of stack (or stacks for each thread)
    - CPU register contents
    - values of statically allocated and global variables

- overriding shell suppression

  - bash `ulimit -c unlimited` or size (`ulimit -c` to check)
  - tcsh/csh `limit coredumpsize 1000000`

- don't leave gdb while making code changes during debugging

Chapter 5 - Debugging in a Multiple-Activities Context

Client/Server Network Programs

- very complex debugging topic
- not mentioned but can use remote debugging

Debugging Threaded Code

```
compile with -lpthread -lm (link pthread and math libraries)
```

- interrupt with ctrl-C
- inspect threads

```
info threads
```

- switch threads

```
thread <thread num>
```

- break in a thread

```
break <break info> thread <thread num>
```

Debugging Parallel Applications

- shared memory and message passing
- attach to a running process

```
gdb <proc name> <proc num>
```

- info after attach

```
bt (backtrace)
```

- backtrace options

```
// stop with ctrl-c
bt <n> -> print innermost n frames
bt full -> print values of local variables
bt no-filters
bt no-filters full -> no python frame filters
// see https://sourceware.org/gdb/onlinedocs/gdb/Backtrace.html for more
```

- print current stack frame

```
frame (f)

info frame
info frame <address>
info args
info locals
```

Shared-Memory Systems

- OpenMP is popular and often uses threads
- TODO: more info and research here -> software-distributed shared memory (p 170)
- OpenMP is essentially C with OpenMP directives
- uses the omni compiler to compile (http://www.hpcc.jp/Omni/)
- hooks in with preprocessor directives and uses OpenMP library code

- after that use many of the same thread debug techniques

Chapter 6 - Special Topics

Common issues

- phantom line numbers in syntax error messages
- missing libraries

```
// compile with
-l<lib name> -> same as lib88.a

LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/Debug/z
export LD_LIBRARY_PATH

// may need to set pkgconfig path
// in C or TC shell
setenv PKG_CONFIG_PATH /usr/lib/pkgconfig:/usr/local/lib/pkgconfig
```

- debugging gui programs
    - curses ->

Chapter 7 - Other Tools

- use strace:
    - logs all system calls made by a program
- use ltrace:
    - logs all library calls made by the program
- splint -> static C secure code checking linter
- EFence -> dynamic memory checker
- GNU MALLOC_CHECK_ environment variable -> set 0 - 3
- GNU mcheck/mtrace function -> include mcheck.h, call mcheck/mtrace before heap functions

Additional Notes

- prefer clang static analysis tools
- can use PVS studio on Linux

- for secure coding:
    - http://www.cert.org/secure-coding/tools/index.cfm
        - clang thread safety analysis
        - compiler-enforced buffer overflow elimination
        - rosecheckers
        - secure coding validation suite
        - As-If Infinitely Ranged Integer Model (checks integer overflow)
- valgrind
- memcheck
- https://github.com/DynamoRIO/dynamorio - suite of tools

Chapter - Using GDB/DDD/Eclipse for other languages

- can use gdb with
    - Java
    - Perl
    - Python
    - SWIG code
    - assembly -> TODO: more here