Getting the most out of GDB

Greg Law

Chris Croft-White

undo™

Disclaimer: random bunch of stuff

Learnt along the way, talking to customers

Lots I don’t know, lots inevitably missing

please help me improve these slides!

Most of this is about knowing what you don’t know

`info gdb` is quite a useful manual

GDB - more than you knew

GDB may not be intuitive but it is very powerful

Easy to use, just not so easy to learn

GDB - more than you know

GDB may not be intuitive but it is very powerful

Easy to use, just not so easy to learn

TUI: Text User Interface

TUI top tips

ctrl-x-a: toggle to/from TUI mode (or `layout src`)

ctrl-l: refresh the screen

ctrl-p / ctrl-n: prev, next, commands

ctrl-x-2: second window; cycle through

GDB has Python!

Single line commands:

```
(gdb) python my_python_function()
```

Interactive:

```
(gdb) python-interactive
>>> my_python_function()
>>> ...
```

Scripts:

```
(gdb) source my_python_script.py
```

GDB has Python!

Full Python interpreter with access to standard modules

(Unless your gdb installation is messed up!)

The **`gdb`** python module gives most access to gdb

`(gdb) python gdb.execute()`	to do gdb commands
`(gdb) python gdb.parse_and_eval()`	to get data from inferior
`(gdb) python help(gdb)`	to see online help

Custom Prompts

Default:

```
(gdb)
```

Static - use this to identify a particular GDB session:

```
set prompt <prompt>
show prompt
```

Dynamic - use Python to generate the prompt:

```
def my_prompt_hook(current_prompt):
    my_prompt = < … arbitrary code … >
    return my_prompt

gdb.prompt_hook = my_prompt_hook  # GDB will call this for every prompt.
```

Python Pretty Printers

Format data better, highlight interesting values, hide boring ones:

```
class MyPrinter:
    def __init__(self, val):
        self.val = val
    def to_string(self):
        return self.val['member']

import gdb.printing
pp = gdb.printing.RegexpCollectionPrettyPrinter('mystruct')
pp.add_printer('mystruct', '^mystruct$', MyPrinter)
gdb.printing.register_pretty_printer(gdb.current_objfile(), pp)
```

.gdbinit

My ~/.gdbinit is nice and simple:

```
set history save on
set print pretty on
set pagination off
set confirm off
```

If you're funky, it's easy for weird stuff to happen.

Hint: have a project gdbinit with lots of stuff in it, and source that.

There are many gdbinit files.

Remote debugging

Debug over serial/sockets to a remote server

Start

```
$> gdbserver localhost:2000 ./a.out
```

Then connect from a gdb with e.g

```
(gdb) target remote localhost:2000
```

Breakpoints

`break foo`	stop at function foo
`tbreak`	temporary breakpoint at foo
`rbreak`	break on regular expression match
`break foo thread 3`	stop at foo only in thread 3
`break foo if bar > 10`	stop at foo only if bar > 10

`delete [n]`	delete breakpoint number n
`disable [n]`	disable breakpoint number n
`enable [n]`	enable breakpoint number n

undo

Watchpoints

`watch foo`	stop when foo is modified
`watch -l foo`	watch location
`rwatch foo`	stop when foo is read
`watch foo thread 3`	stop when thread 3 modifies foo
`watch foo if foo > 10`	stop when foo is > 10

`delete` / **`disable`** / **`enable`** work for watchpoints too!

Catchpoints

Catchpoints are like breakpoints but catch certain events, such as C++ exceptions

e.g. **`catch catch`** to stop when C++ exceptions are caught

e.g. **`catch syscall nanosleep`** to stop at nanosleep system call

e.g. **`catch syscall 100`** to stop at system call number 100

But watch out for the confusing command names…

`delete` / **`disable`** / **`enable`** work for catchpoints too!

Multiprocess Debugging

Debug multiple ‘inferiors’ simultaneously

Add new inferiors

Follow fork/exec

Multiprocess Debugging

```
set follow-fork-mode child|parent
set detach-on-fork off
info inferiors
inferior N
set follow-exec-mode new|same
add-inferior -copies <count> -exec <name>
remove-inferior N
clone-inferior
print $_inferior
```

undo

Non-stop mode

Other threads continue while you're at the prompt

```
set non-stop on
start
continue -a
```

Make sure you set pagination off otherwise bad stuff happens!

thread apply

```
thread apply 1-4 print $sp
thread apply all backtrace
thread apply all backtrace full
```

calling inferior functions

`call foo()` will call foo in your inferior

But beware, print may well do too, e.g.

```
print foo()
print foo+bar  (if C++)
print errno
```

And beware, below will call **`strcpy()`** *and* **`malloc()`**!

```
call strcpy(buffer, "Hello, world!\n")
```

Time Travel Debugging - how did that happen?

GDB inbuilt reversible debugging:Works well, but is **very** slow

Time-Travel Debugging - how did that happen?

GDB inbuilt reversible debugging:Works well, but is **very** slow

GDB in-build ‘record btrace’: Uses Intel branch trace or processor trace.

- Only on certain CPUs
- Not really reversible, no data

rr: Very good at what it does, though can be limited features/platform support

UDB/LiveRecorder: perfect :-)

Dynamic Printf

Use dprintf to put printf's in your code without recompiling, e.g.

```
dprintf mutex.c:100,"m is %p m->magic is %u\n",m,m->magic
```

control how the printfs happen:

```
set dprintf-style gdb|call|agent
set dprintf-function fprintf
set dprintf-channel mylog
```

More Python

Create your own commands

```
class my_command( gdb.Command):
    '''doc string'''
    def __init__( self):
        gdb.Command.__init__( self, 'my-command', gdb.COMMAND_NONE)
    def invoke( self, args, from_tty):
         do_bunch_of_python()
my_command()
```

Yet More Python

Hook certain kinds of events

```
def stop_handler(ev):
    print('stop event!')
    if isinstance(ev, gdb.SignalEvent):
        print('its a signal: ' + ev.stop_signal)

gdb.events.stop.connect(stop_handler)
```

Other cool things...

`display`	show the value of expression every time you stop
`advance foo`	like tbreak, but one-shot and stops on stack frame exit
`until`	like `next` but doesn't loop
`command`	list of commands to be executed when breakpoint hit
`silent`	special command to suppress output on breakpoint hit
`save breakpoints`	save a list of breakpoints to a script
`save history`	save history of executed gdb commands
`info line foo.c:42`	show PC for line
`info line * $pc`	show line begin/end for current program counter

And finally...

gcc's -g and -O are orthogonal; gcc -Og is optimised but doesn't mess up debug

gdbWatchPoint resource

undo.io/resources/gdb-watchpoint/

Search tutorials SEARCH

12 Tutorials

Save Time Debugging with Time Travel Debugging

GDB tutorial | 1 min read

Debugging with pretty printers in GDB - part 3

GDB tutorial | 8 min read

Debugging with pretty printers in GDB - part 2

GDB tutorial | 7 min read

How Linux C++ Debuggers Really Work

GDB tutorial | 3 min read

gdbWatchPoint: your resource for everything GDB

GDB tips & tricks to make your debugging life easier - straight from GDB guru Greg Law