LIBERATING THE DEBUGGING EXPERIENCE WITH THE GDB PYTHON API

JEFF TRULL

8 AUGUST 2018

Created: 2018-08-09 Thu 11:33

INTRODUCTION

ABOUT ME

- Hardware Guy (microprocessors) -> CAD -> C++ Consultant.
- Organizer of Emacs SF Meetup https://www.meetup.com/Emacs-SF/
- Train Nut
- Available for Projects

GENERAL THEME OF TALK

- Tools are a force multiplier
- gdb is amazing, Python is amazing, their cross product is 🔥

GETTING STARTED

BASICS

ACCESSING PYTHON

- Everything starts by typing "python"

```
(gdb) python print(list(reversed([3, 2, 1])))
[1, 2, 3]
```

- You can also load and execute your own scripts:

```
(gdb) python import myscript
```

- Your script needs to be in the Python search path...

PYTHON SEARCH PATHS

- Two possibilities:
  - external

    ```
    PYTHONPATH=/path/to/my/python/libs gdb ...
    ```

  - internal

    ```
    (gdb) python import sys
    (gdb) python sys.path.insert(0, "/path/to/my/python/libs")
    ```

GDB API

Everything we can do in the CLI can be done with `gdb.execute()`

```
gdb.execute('backtrace')
```

You can capture the output too:

```
(gdb) python foo = int(gdb.execute('p foo', to_string = True))
(gdb) python print(foo)
42
```

EXPRESSIONS

It's better to use gdb APIs where possible, though.

A better way to access a variable

```
result = gdb.parse_and_eval('foo')
```

You now have a `gdb.Value` object you can cast to int or string for Python, or do more interesting things:

```
(gdb) python print(type(result))
<class 'gdb.Value'>
(gdb) python print(result.address)
0x7fffffffdd80
```

FRAMES

Sometimes a variable you want is not currently visible, but is elsewhere on the stack:

```
caller_frame = gdb.selected_frame().older()
result = caller_frame.read_var('bar')
```

You now have a `gdb.Value` from the caller's frame.

BREAKPOINTS AND WATCHPOINTS

CREATING THROUGH THE API

```
bp = gdb.Breakpoint('main.cpp:29')
wp = gdb.Breakpoint('foo', gdb.BP_WATCHPOINT)
```

Now you can manipulate your breakpoint:

```
bp.enabled = False       # temporary disable
bp.condition = 'foo > 3'
bp.commands = 'shell google-chrome https://www.youtube.com/watch?v=Vhh_
```

(Writable commands are a soon-to-be-released feature)

FINISH BREAKPOINTS

```
bp = FinishBreakpoint()
```

- Activated on any exit from the current frame (like "finish" command)
- But you can enable/disable, add conditions, etc. etc.
- Functionality not available from the CLI!

PRACTICALITIES

DEBUGGING YOUR DEBUGGING CODE

GETTING YOUR PYTHON VERSION

```
(gdb) python import sys
(gdb) python print(sys.version)
```

gdb can be built with Python 2 or 3…

RELOADING CODE

```
(gdb) python from importlib import reload
(gdb) python reload(gdb_util.vgleaks)
```

SETTING BREAKPOINTS

Edit the code:

```python
import pdb;pdb.set_trace()
```

Maybe there is a better way?

PRINTING A BACKTRACE

```
import pdb, traceback, sys
...
try:
    thing_that_may_throw()
except:
    extype, value, tb = sys.exc_info()
    traceback.print_exc()
    pdb.post_mortem(tb)
```

Thanks, Stack Overflow

PRETTY PRINTERS

A topic in themselves, but they can get in the way when you're scripting gdb:

```
(gdb) info pretty
global pretty-printers:
  ...
  objfile /usr/lib/x86_64-linux-gnu/libstdc++.so.6 pretty-printers:
  libstdc++-v6
    ...
    std::tuple
    std::unique_ptr
    std::unordered_map
    ...
```

DISABLING A PRETTY PRINTER

```
(gdb) disable pretty /usr/lib/x86_64-linux-gnu/libstdc\\+\\+.so.6
    libstdc\\+\\+-v6;std::tuple
1 printer disabled
163 of 164 printers enabled
```

COMMANDS

A fundamental feature for debug tooling

- Useful for boxing up repetitive or tedious CLI work
- Entry point for Python functionality

DEFINING A COMMAND

```python
import gdb
import re
class StepThroughBoost(gdb.Command):
    """Steps forward until we are not in a Boost library"""

    def __init__(self):
        super(StepThroughBoost, self).__init__("step-through-boost",
                                               gdb.COMMAND_RUNNING)

    def invoke(self, arg, from_tty):
        frame = gdb.selected_frame()
        while re.match('boost::', frame.name()):
            gdb.execute('step', to_string=True)
            frame = gdb.selected_frame()

StepThroughBoost()   # registers command
```

DEMO

```cpp
#include <iostream>
#include <boost/math/common_factor_rt.hpp>

int main()
{
    using namespace boost::math;
    int result = gcd_evaluator<int>()(50, 125);
    std::cout << "GCD of 50 and 125 is " << result << "\n";
}
```

FRAMES

The Python API gives you a lot of control over how backtraces are presented

DECORATORS

You can change the appearance of any frame

```
class Rot13Decorator(gdb.FrameDecorator.FrameDecorator):
    def __init__(self, fobj):
        super(Rot13Decorator, self).__init__(fobj)

    def function(self):
        name = self.inferior_frame().name()
        return codecs.getencoder('rot13')(name)[0]
```

FILTERING

You can remove frames you don't want to see

```
class BoostFilter:
    def __init__(self):
        # set required attributes
        self.name = 'BoostFilter'
        self.enabled = True
        self.priority = 0

        # register with current program space
        gdb.current_progspace().frame_filters[self.name] = self

    def filter(self, frame_iter):
        # compose new iterator that excludes Boost function frames
        f_iter = filter(lambda f : re.match(r"^boost::", f.function()),
                        frame_iter)
        # wrap that in our decorator
        return imap(Rot13Decorator, f_iter)

BoostFilter()  # Register filter
```

DEMO

- Our simple example
- Something a little more interesting

ADDING SEMANTIC INFO WITH LIBCLANG

- gdb has a limited understanding of your code.
- We can augment this with the Python bindings of libclang to make useful tools.

CREATING A COMPILATION DATABASE

- Usually called `compile_commands.json`
- All the flags, include paths, macro definitions etc.
- Gives libclang what it needs to understand your code
- You can generate it in CMake:

```
set( CMAKE_EXPORT_COMPILE_COMMANDS ON )
```

- There's also a tool called "Bear" (**Build EAR**) that listens to (instruments exec calls from) your build tool

USING LIBCLANG

GDB TO LIBCLANG

Getting the current statement's location from gdb:

```
frame = gdb.selected_frame()
line = frame.find_sal().line
fname = frame.find_sal().symtab.filename
```

Getting an AST cursor from libClang:

```
# setup omitted...
loc = cindex.SourceLocation.from_position(translation_unit,
                                          translation_unit.get_file(fna
                                          line, 1)
cur = cindex.Cursor.from_location(translation_unit, loc)
# interrogate cursor to get semantic info
```

APPLICATION: IMPROVED SINGLE STEP

DEMO

(No Fun)

WE ONLY WANT TO STOP IN USER CODE

gdb lacks semantic information

SOLUTION: LIBCLANG'S PYTHON BINDINGS

- find the current statement
- identify calls, objects with methods, and lambdas within it
- Use a regex to remove calls to library code
- use gdb to set temporary breakpoints on what remains

FAKING SINGLE STEP WITH BREAKPOINTS

```
# for each stopping point:
bp = gdb.Breakpoint('%s:%d'%(fname, line),
                    internal=True) # add temporary breakpoint
gdb.execute('continue')
bp.delete()                        # remove temporary breakpoint
```

DEMO

(Much Better)

VALGRIND

VALGRIND'S GDB INTERFACE

BASICS

Valgrind can act as a `gdbserver` instance, as if it were a remote session on an embedded system:

```
$ valgrind --vgdb-error=0 ./a.out
```

You run gdb in a different shell to connect to it:

```
gdb ./leak -ex='target remote | /usr/lib/valgrind/../../bin/vgdb'
```

MONITOR COMMANDS

All of valgrind's tools expose some special commands via the gdbserver "monitor". For leak checking, `memcheck` supplies these:

leak_check
Scans allocated blocks for pointers, reports sets of unreachable blocks that were not deallocated.

block_list
For a given "loss record" found by `leak_check`, lists the unreachable blocks

who_points_at
Finds matching pointers in allocated blocks

DEMO

(Monitor Usage)

BUILDING ON TOP OF VALGRIND-GDB

SEARCHING FOR LEAK ORIGIN

Simple (though slow) idea: run code until a leak appears

```
class StepToLeak(gdb.Command):
    """Step until valgrind reports a leak"""

    def __init__ (self):
        super (StepToLeak, self).__init__ ("stepl", gdb.COMMAND_RUNNIN

    def invoke(self, arg, from_tty):
        result = gdb.execute('monitor leak full', False, True)
        while result.find('are definitely lost in loss record') is -1:
            try:
                gdb.execute('step', to_string = True)  # QUIETLY step
            except gdb.error:
                print('error while stepping')
                break
            result = gdb.execute('monitor leak full', False, True)
        print('loss report:\n%s'%result)
        print('leak first noticed at:\n')
        gdb.execute('bt')
```

FINDING POINTER LOOPS

The monitor commands Valgrind gives us are enough to find shared_ptr loops, but it's a tedious manual process.

Solution: treat pointers as edges in a directed graph and find loops

GRAPH_TOOL LIBRARY

Someone bound `Boost.Graph` into Python and added some extra features. We will use it like this:

- construct a minimal graph by starting with the source node from the leak report
- run a depth-first search, adding edges and vertices as we discover more pointers
- manual parsing of `monitor` command output

DEPTH-FIRST SEARCH WITH VISITOR

```
from graph_tool.search import DFSVisitor
class LoopFindVisitor(DFSVisitor):
    # setup omitted

    def discover_vertex(self, u):
        # having arrived here for the first time, we need to add any o
        # add bogus additional vertex and edge
        ...

    def tree_edge(self, e):
        # this is where we update the predecessor map
        ...

    def back_edge(self, e):
        # the money method. This is where we detect loops
        ...
```

TEST CASE

```
struct Person : std::enable_shared_from_this<Person> {
    Person(std::string name) : name_(std::move(name)) {}
    std::shared_ptr<Person> manager() { return manager_; }

    std::shared_ptr<Person> create_employee(std::string name) {
        employees_.emplace_back(new Person(std::move(name), shared_fro
        return employees_.back();
    }
    std::string name() const { return name_; }

private:
    Person(std::string name, std::shared_ptr<Person> manager)
        : manager_(std::move(manager)), name_(std::move(name)) {}

    std::shared_ptr<Person>              manager_;
    std::vector<std::shared_ptr<Person>> employees_;
    std::string                          name_;
};
```

DEMO

- stepping until leaks appear
- loop finding

PYQT

QT BOUND INTO PYTHON

- Surprisingly easy to use. Maybe easier than the C++ version :)
- Usage is pretty obvious if you know Qt
- Enables visualization tools

VISUALIZING ALGORITHMS

ENABLING FEATURES

- Breakpointing C++ special member functions and `swap` free function
- Running PyQt and gdb in separate threads

INSTRUMENTING VALUE CLASS

```cpp
struct int_wrapper_t
{
    int_wrapper_t() : v_(0) {}
    int_wrapper_t(int v) : v_(v) {}

    // std::sort uses swap, move, and move assignment
    // our custom swap is below
    int_wrapper_t(int_wrapper_t && other);
    int_wrapper_t& operator=(int_wrapper_t && other);

    // so I don't have to write operator< or operator<<
    operator int() const { return v_; }

private:
    int v_;
};
```

INSTRUMENTING SWAP

```
void swap(int_wrapper_t & a, int_wrapper_t & b)
{
    std::swap(a, b);
}
```

For this one I also had to use a **finish breakpoint** to bracket the call to `std::swap` or we would count the moves inside it.

DEMO

WRAPPING UP

INVESTING IN DEBUG TOOLING PAYS OFF

- For teams of more than a few people a part time tool engineer makes sense
- There's usually one or two key data structures you constantly look at to understand what's happening
- Sometimes there are categories of bugs that come up more frequently

RESOURCES

MORE INFORMATION

- Code from this presentation: https://github.com/jefftrull/gdb_python_api
- Blog with more detail: http://jefftrull.github.io/

LINKS

- Greg Law's 2016 CppCon talk on gdb features: https://channel9.msdn.com/Events/CPP/CppCon-2016/CppCon-2016-Greg-Law-GDB-A-Lot-More-Tha Knew
- Michael Krasnyk lightning talk: https://www.youtube.com/watch?v=QtTYXE1wSVs
- Scott Tsai "Programmatic Debugging with gdb and Pyt https://docs.google.com/presentation/d/15qOKBh9FI xAHXZSJDS5_aoZk0Caz12FL_f294/edit#slide=id.p

LINKS

- Tom Tromey's utilities: https://github.com/tromey/gdb-helpers
- pwndbg - gdb library based on reverse engineering https://github.com/pwndbg/pwndbg