LIBERATING THE DEBUGGING EXPERIENCE WITH THE GDB PYTHON API

JEFF TRULL

27 SEPTEMBER 2018

INTRODUCTION

ABOUT ME

- Hardware (microprocessors) -> CAD -> C++ Consultant.
- Organizer of Emacs SF Meetup https://www.meetup.com/Emacs-SF/
  - BoF 8AM Friday
- Available for Projects

(end of advertisement)

GENERAL THEME OF TALK

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

OUTLINE

- (Very) Basic Python
- Four Applications
  - Making stack traces more friendly
  - Better stepping through code
  - Finding memory leaks
  - Visualizing algorithms
- Summary and Conclusion

JUST ENOUGH PYTHON

BASIC PYTHON

Whitespace sensitive

Indentation is used to indicate blocks:

```
if foo > 3:
    bar = "large"
else:
    bar = "small"
```

Classes

```
class Derived(Base):

    def __init__(self, x):
        super(Derived, self).__init__()  # base class "initializer"
        self.x = x                       # this->x = x

    def method(self):                    # possibly an override
        print('x is %d'%x)
```

```
// the same class in C++:
struct Derived : Base
{
    Derived(int x) : x_{x} {}

    void method() { std::cout << "x is " << x_ << "\n"; }

    int x_;
};
```

PYTHON IN GDB

Accessing Python

- Everything starts by typing "python"

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

- You can do multiple lines

```
(gdb) python
>import gdb
>print(gdb.VERSION)
>end
8.1.0.20180409-git
```

- You can also load and execute your own scripts:

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

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

gdb API

Everything we can do on the command line can be done with `gdb.execute()`

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

You can capture the output too:

```
(gdb) python result = gdb.execute('p foo', to_string = True)
(gdb) python print(type(result))
<class 'str'>
(gdb) python foo = int(result)
(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
(gdb) python print(result.type)
int
(gdb) python foo = int(result)
(gdb) python print(foo)
42
```

APPLICATIONS

IMPROVING STACK TRACES

BACKTRACE CAN BE CONFUSING

- Exposes many library internals
- Verbose function signatures
  - default arguments like allocators cause types to be repeated
- Users may prefer a more concise version

GOALS

- Shrink verbose frames with common-sense substitutions
  - `std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >`
- Eliminate intermediate frames inside libraries

TOOLS FROM THE API

- Frame Decorator to change how each frame is displayed
- Frame Filter to remove library internal calls

DECORATORS

You can change the appearance of any frame. Here we obfuscate the function name:

```python
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. Here we remove everything inside Boost:

```python
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
```

SOLVING BACKTRACES

Decorator

Use simple regexes to replace common types with their aliases (minus default arguments)

Filter

Eliminate all but the first call in a sequence of `std::` library frames

DEMO

The Code

```cpp
// (broken) attempt at lexicographic sort

using Strings = std::vector<std::string>;

std::vector<Strings> data{
    {"Frodo", "Sam", "Smeagol"},
    {"Foo", "Bar", "Baz"},
    {"Monoid", "Endofunctor", "Monad"}};

std::sort(data.begin(), data.end(),
          [](Strings const & a, Strings const & b) {
              if (a[0] < b[0]) {
                  return true;
              } else {
                  return a[1] < b[1];
              }
          });
```

Let's Try It

BETTER STEPPING

STEP TO USER CODE

- Often we supply our own code to a library for its use
  - From simple callback to full objects
- We don't want to have to step through the library code
- We don't want to set breakpoints at every possible callee

Solution: "Step to User Code"

TOOLS FROM THE API

`gdb.Breakpoint` will help us mimic single-stepping by creating temporary breakpoints

Breakpoints 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_
```

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!

PYTHON MODULE: LIBCLANG

libClang's Python bindings

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

PUTTING IT TOGETHER

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
```

gdb to libClang

libClang lets us interrogate the AST (Abstract Syntax Tree) representing our program:

```
# 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
```

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

Let's Try It

Back to our previous example

FINDING LEAKS

APPLICATION: REFERENCE LOOPS

- Excessive use of `std::shared_ptr<T>` can lead to memory leaks
- Can we automate the process of finding reference loops?

TOOL: VALGRIND

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

Starting the valgrind gdbserver

```
$ valgrind --vgdb=yes --vgdb-error=0 ./leak
==29620== Memcheck, a memory error detector
==29620== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et a
==29620== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyrigh
==29620== Command: ./leak
==29620==
==29620== (action at startup) vgdb me ...
==29620==
==29620== TO DEBUG THIS PROCESS USING GDB: start GDB like this
==29620==   /path/to/gdb ./leak
==29620== and then give GDB the following command
==29620==   target remote | /usr/lib/valgrind/../../bin/vgdb --pid=2962
==29620== --pid is optional if only one valgrind process is running
```

Starting gdb

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

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

Monitor commands

Valgrind adds special "monitor" commands to gdb:

leak_check
: Main leak detection command

block_list
: Gets details about leaked blocks

who_points_at
: Finds pointers to given blocks

It does not provide Python API for this, so we will manually parse `monitor` output.

PYTHON MODULE: GRAPH_TOOL

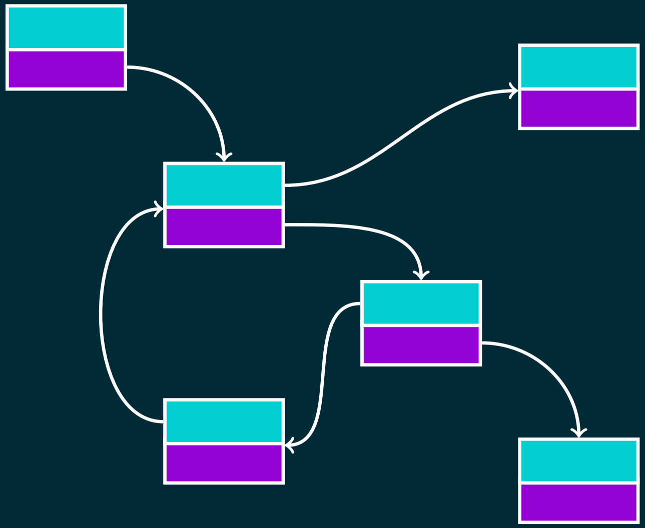
- Blocks of allocated memory contain pointers to other blocks. You can visualize this as a directed graph.
- Reference loops are loops within this directed graph. We can use a graph library module to help us find them.

Using graph_tool

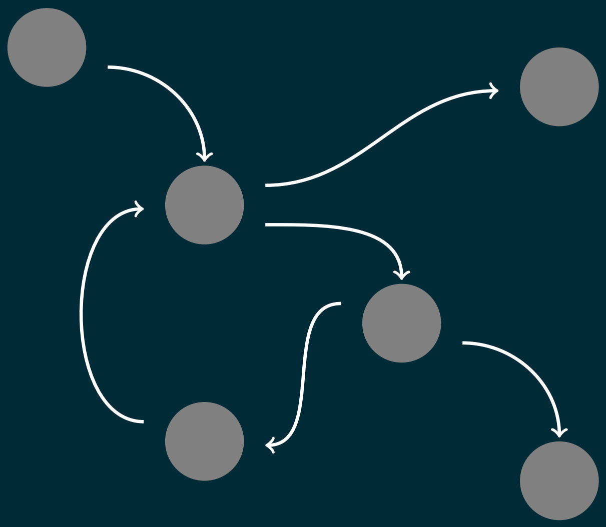
The `graph_tool` author bound `Boost.Graph` into Python and added some extra features. We will use it like this:

- construct a minimal graph by starting with the source node from the leak report
- run a depth-first search, growing the graph as we discover more pointers
- loop found when we encounter a vertex for the second time
- vertex "predecessors" (a map from each vertex to the previous) let you trace the loop

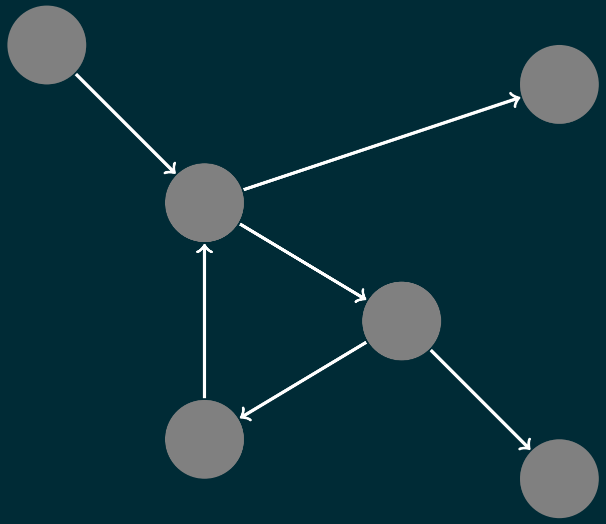
DFS example



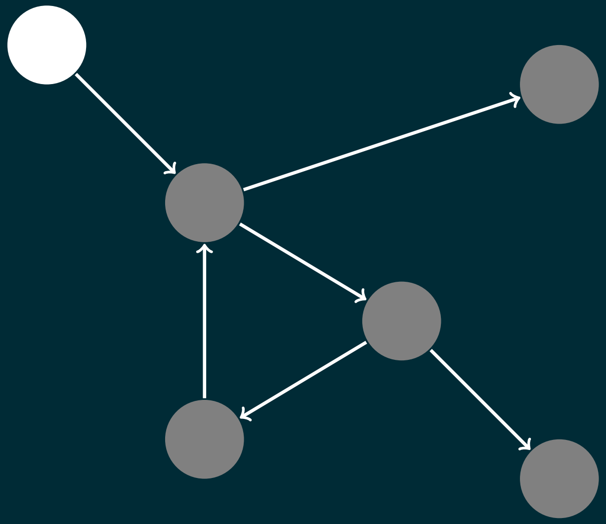
DFS example



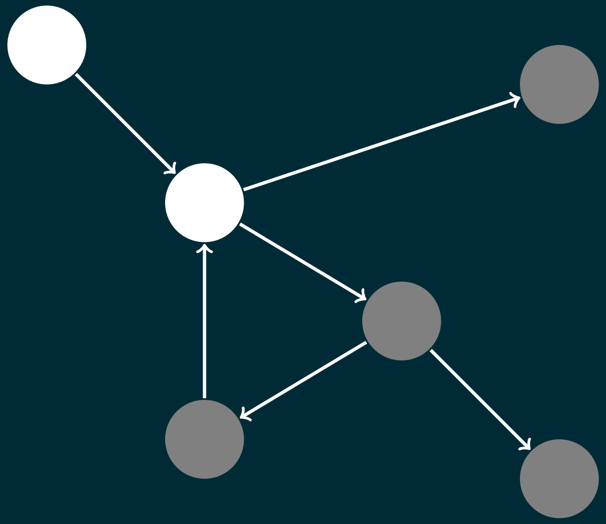
DFS example



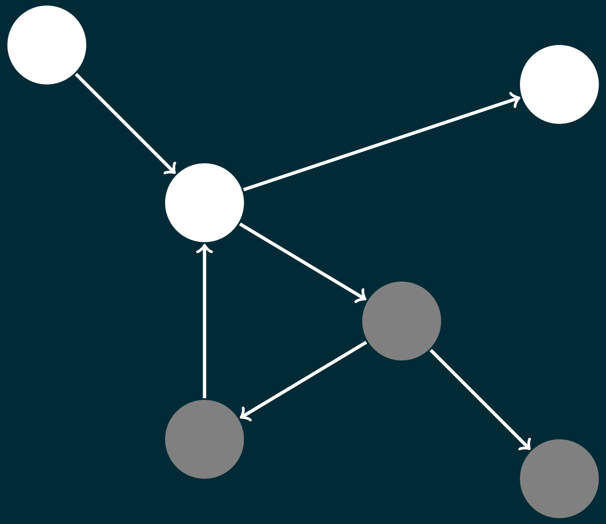
DFS example



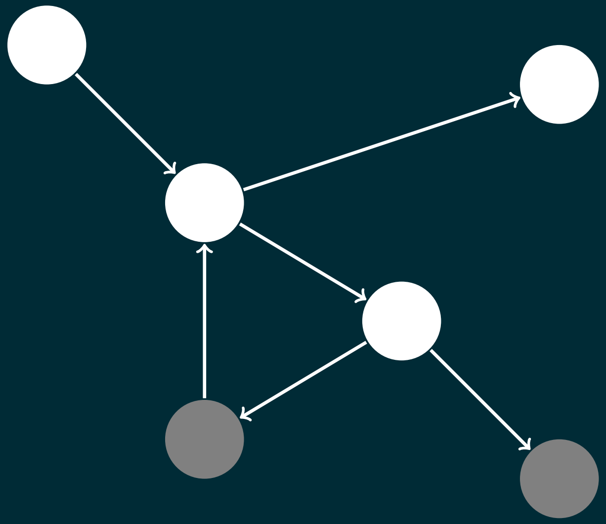
DFS example



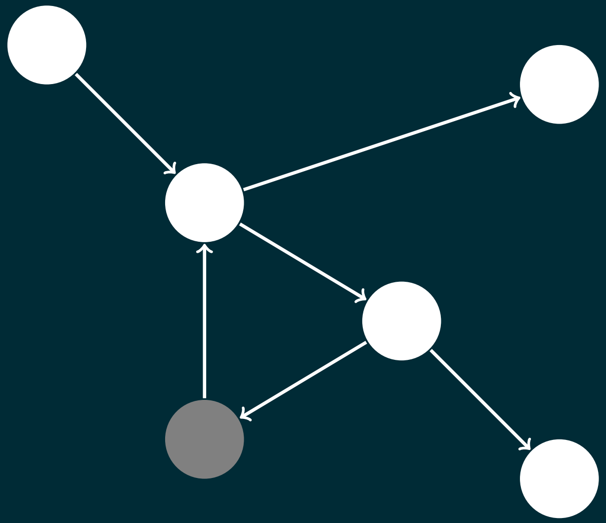
DFS example



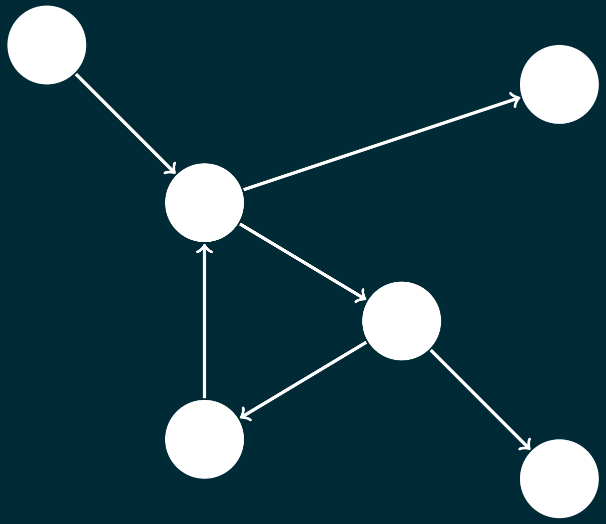
DFS example



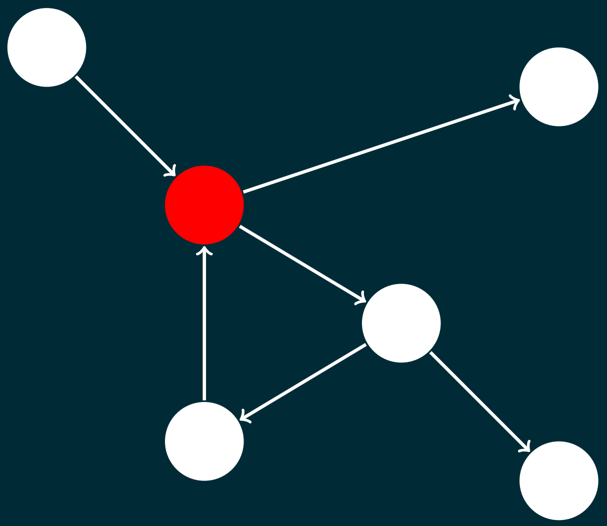
DFS example



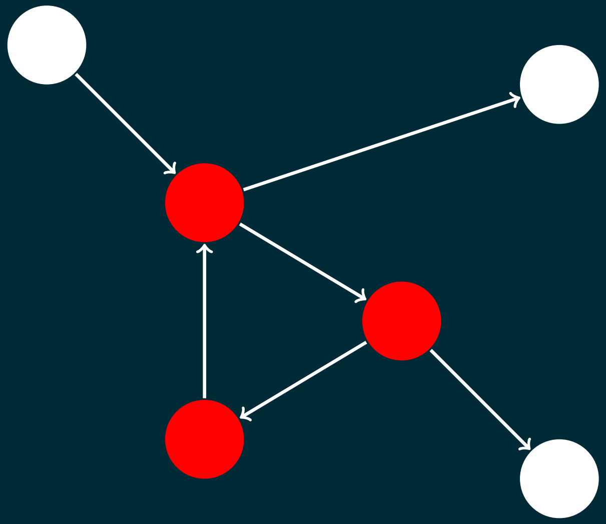
DFS example



DFS example



DFS example



DEMO

Test Case: Task List

```
struct TaskList
{
    template<typename F>
    void add(F f) {
        tasks_.push_back(move(f));
    }

    void doOne() {
        if (!tasks_.empty()) {
            auto f = tasks_.front();
            tasks_.pop_front();
            f();
        }
    }

private:
    deque<function<void()>> tasks_;
};
```

Test Case: Adding Tasks

```
int main() {
    auto tasks = make_shared<TaskList>();

    tasks->add([tasks]() {
            cout << "task 1\n";
            // queue another one
            tasks->add([]() {
                    cout << "task 2\n";
                });
        });
}
```

Let's Try It

VISUALIZING ALGORITHMS

UNDERSTANDING THE OPERATION OF KEY CODE

Every large codebase seems to have a few critical algorithms or data structures.

- Diagnosing bugs tends to require referring to their operation
- Often there's special debug settings (via `#ifdef`), or logging
- Typical use is to painstakingly review/grep through the logs to understand what is happening

Why not build some visualization tooling?

GOAL

Build a graphical display of an algorithm in action

- We will use `std::sort` on a vector

TOOLS FROM THE API

We will use mainly breakpoints, for driving display updates

PYTHON MODULE: PYQT5

- This is exactly what you would guess it was
- Surprisingly easy to use. Maybe easier than the C++ version :)
- Usage is pretty obvious if you know Qt

GENERAL APPROACH

- A special wrapper class for the value type
- Breakpointing C++ special member functions and `swap` free function
- Running PyQt and gdb in separate threads
  - Communication via thread-safe Queue

Instrumenting Value Class

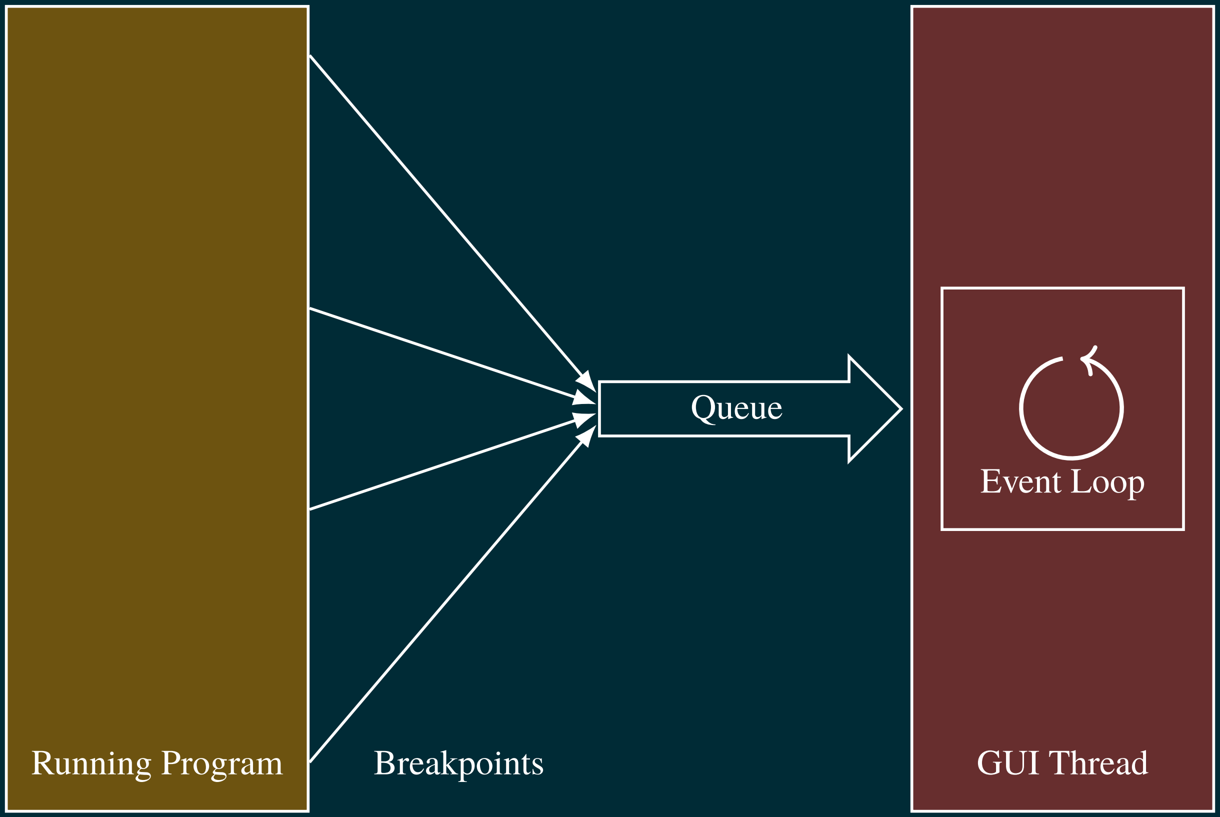
```
struct int_wrapper_t
{
    int_wrapper_t() : v_(0) {}
    int_wrapper_t(int v) : v_(v) {}

    // 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)
{
    // disable move ctor/assign breakpoints
    std::swap(a, b);
    // re-enable move ctor/assign breakpoints
}
```

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.

A Separate Thread for the GUI



DEMO

The Code

```
// randomly shuffle the sequence 1 to N
std::iota(A.begin(), A.end(), 1);
std::shuffle(A.begin(), A.end(),
             std::mt19937{std::random_device{}()});

// then sort it
std::sort(A.begin(), A.end());   # <- animated
```

Let's Try It

WRAPPING UP

INVESTING IN DEBUG TOOLING PAYS OFF

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

If you name a script `objfile-gdb.py`, where `objfile` is an executable or library, gdb will load it for you.

PYTHON IS A GAME CHANGER

- Largely because of its vast ecosystem
- Take the cross product of:
  - Anything we can measure or detect in the program
  - Some Python visualization or analysis module

There are endless possibilities!

LET'S MAKE SOME TOOLS!

QUESTIONS

Code from this presentation:

https://github.com/jefftrull/gdb_python_api

RESOURCES

MORE INFORMATION

- 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/15qOKBh9F
  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

TIPS AND TRICKS

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:

```
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
```

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")
```