IMPROVING DEBUGGABILITY WITH GDB'S PYTHON API

JEFF TRULL

6 MAY 2018

Created: 2018-05-09 Wed 18:38

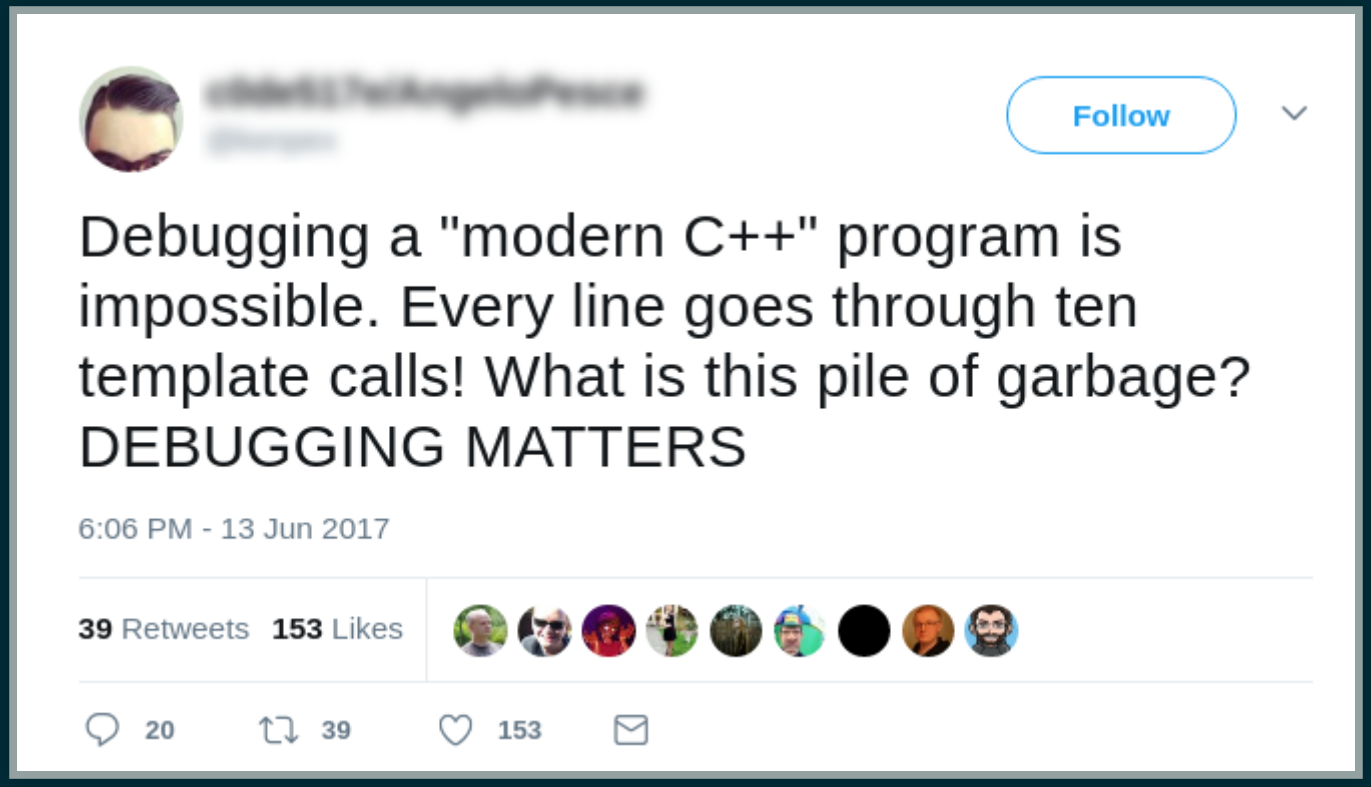
INTRODUCTION

DEBUGGABILITY: A BARRIER TO ADOPTION

LIBRARY INTERNALS ARE INTIMIDATING

Templated libraries can have complex implementations.
Debugging makes them visible.

CONFUSION LEADS TO FRUSTRATION



DEMO (PAINFUL)

SOURCE (INPUT)

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

SOURCE (ALGORITHM)

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

THE EXPERIENCE

IMPROVING THINGS WITH THE PYTHON API

EASING SINGLE STEPPING

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

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
```

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
```

CLEANING UP BACKTRACE

STACK FRAME DECORATORS

Subclass and override gdb's frame display functions

```
class Rot13Decorator(gdb.FrameDecorator.FrameDecorator):
    # boilerplate omitted

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

STACK FRAME FILTERS

- Wraps an iterator over stack frames
- Can remove stack frames or create hierarchy from them

```
class BoostFilter:
    # boilerplate omitted...

    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)
```

PUTTING IT TOGETHER

What do we need to make the user happier?

- A filter that removes all but one of every sequence of library frames
- A decorator that cleans up the ugly type names
- Stepping only into user code

DEMO (HAPPY)

THE EXPERIENCE

INVESTING IN DEBUG TOOLING PAYS OFF

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/15qOKBh9FD xAHXZSJDS5_aoZk0Caz12FL_f294/edit#slide=id.p
- Tom Tromey's utilities: https://github.com/tromey/gdb- helpers