Introduction

This is a report on an attempt at developing a reusable module and a complementary terminal user interface wrapepr that makes debugging educational C programs more intuitive and accessible. While integrated coding environments (IDEs) offer debugging tooling, not every developer would wish to use an IDE for various reasons like resource limitations, vendor lock-in, learning curve, additional and unnecessary complexity, and lack of customisation. Moreover, it can be said that students should reach for a text editor and the command line over an IDE to study how compilation and code execution works, improving their understanding of what is happening "under the hood". IDEs also provide many "comfort" features which may be threats to the learning experience for students as they block the student from performing analysis and critical thinking themselves. That said, it would be unwise to not provide a restricted, simplified, and curated collection of tooling for educational purposes.

While students are encouraged to explore, analyse, and think on their own, it is important to recognise the importance of lending a helping hand after they have sufficiently exhausted their options. Outside of IDEs, there is a limited set of tooling that help with debugging programs written in the C language. Specifically, the only main options are WinDBG, LLDB, and GDB, all of which allows the inspection of the internals of another (potentially buggy) program while it executes. However, the first option, WinDBG, is unfortunately exclusive to the Windows operating system and native code compiled by MSVC. The other two, LLDB and GDB, maintained by LLVM and GNU, respectively, are run and interacted with through the terminal. While it is important to learn how to run commands in the terminal, having "knowing the terminal" as a prerequisite for debugging is not ideal for students learning a programming language and shell scripting simultaneously. Interacting with GDB and LLDB is through the terminal but also using a bespoke language, making the bar to entry even higher. It can be said that there is a gap in educational tooling for debugging programs that is also intuitive to use.

While it is important to understand the other method of debugging, such as displaying signals, flags, and values to the terminal and pair programming, it can be said that the line-by-line and breakpoint debugging workflow offered by GDB and LLDB is more intuitive, clean, and efficient.

Existing Work and Areas of Development

While the debuggers have been around for decades, there are few solutions that aim to bridge their learning curves as, historically speaking, systems languages like C were reserved for experienced and talented prgorammers. That said, an analysis of existing work reveals that each solution is lacking in a combination of extensibility, intuition, and complexity.

The solution `gdbgui` is a browser-based frontend to GDB that exposes a large set

of features, including thread and assembly instruction inspection. It consists of a frontend-backend pair where the the backend creates a new GDB process instance for each client connected to it via the frontend using websocket API. It can be said, however, the complexity of its features, the complexity of the architecture, and its resource overhead is overkill, overwhelming, and unnecessarily unintuitive in the scope of debugging simple, single-file C programs written by students.

The solution `pygdbmi` is a Python module that allows the management of and interaction with GDB processes through Python code, but parts of its implementation are problematic. Specifically, manner in which the module communicates with GDB processes is through synchronous polling, which leads to inefficient performance and interfaces pooly with the GDB's out-of-band class of messages.

The solution `lldbg` is a lightweight native GUI for LLDB that aims to function intuitively so that "using `lldbg`" is not a skill individuals have to learn, require as little configuration as possible, and be a graphical drop-in replacement for LLDB. However, it is in an alpha-stage that is about 60% usable, and discourages users from using it except for contributory purposes.

While existing work in this gap can be found, they all fall short when it comes to assisting novices in debugging relatively simple programs with as little friction and complexity as possible and through mediums they already use (i.e., text editor and terminal).

Solution

We believe that GDB is superior than LLDB when it comes to programmatic, non-human interfacing as both are designed to be human-readable, but only GDB offers a "machine interface" (GDB MI) that produces normalised and machine-oriented output.

In the interest of maintainability and extensibility, our solution has been divided into two parts.

1. A programming language library module that abstracts over managing and interacting with GDB sub-processes
2. A terminal user interface (TUI) frontend that provides exposes to the programming language module graphically to the user.

Library module

The library code is contained within a single class named `GDB`, is divided between `BaseGDB` and `GDB`. The former lays an internal-centric foundational layer of functionality to facilitate the instantiation, termination, and messaging with GDB sub-processes, while the latter exposes to the client code functionalities of interest such as running, stepping, and memory inspection.

GDB sub-proceeses are created using the standard library function `asyncio.create_subprocess_exec`, which returns an `asyncio.Process`, which is the medium we can use to send commands to GDB, read output from it, ultimately terminate it.

When it comes to GDB MI output, there are, broadly speaking, two categories that are differentiated between one another using the first character of the line. GDB MI "result records" are outputs that begin with a caret ("^") that are immediate responses to commands sent to GDB MI, and can be either of "^done", "^running", and "^erorr". It is important to note that the result record of a command does not imply the completion of a command, it is more of an indicator that GDB MI is ready to accept the next command. The other kind of output are "async records" or "out-of-band records" that are used by GDB MI to notify the client of additional changes that have occurred, such as hitting a breakpoint. Out-of-band records are used by GDB MI to return the final results of long-running commands such as "next" while result records are used by GDB MI to acknowledge it is ready for the next command, even if "next" is still being executed.

Unlike GDB and LLDB, the output of GDB MI is well-structured and well-defined. There exists a specification in the following format that defines the structure of every class of output.

```
result →
    variable "=" value

variable →
    string

value →
    const | tuple | list

const →
c-string

tuple →
    "{}" | "{" result ( "," result )* "}"

list →
    "[]" | "[" value ( "," value )* "]" | "[" result ( "," result )* "]"
```

Most classes of GDB MI output are variations that all encode the bulk of their information in a structure similar to JSON. The specification is shown directly above, and the key differences between it and JSON are the fact that the keys in a dictionary/object are *not* quoted, key-value pairs are separated by an equals sign ("=") and not a colon (":"), and strings are represented as c-strings that are encoded slightly differently to JSON strings.

This means that much of GDB MI output is can be parsed into JSON-like structures using a single method, and so `BaseGDB` contains a function `parse_result` that does this job. It attempts to use simple regular expressions to map the GDB MI output to a valid JSON string, and then uses the standard library `json.loads` to convert it into Python data structures.

A caveat of this approach is that GDB MI result arrays can contain key-value pairs, such as the following.

```
[frame={level="0"},frame={level="1"}]
```

However, it can be found that in every array containg key-value pairs, all its keys are identical, so they carry trivial information. The GDB MI output can still be mapped to JSON by eliminating the redundant keys. For example, the above example could be mapped to the following.

```
[{level="0"},{level="1"}]
```

This reduction is achieved rebuilding the string character by character and maintaining a stack of braces, and eliminating keys when the most recent brace is an opening curly brace.

The task of reading output from the GDB MI needs to be done carefully because of the presence of both synchronous and asynchronous output packets sent by GDB MI. That is, it is difficult to predict the number of messages sent by GDB for a given input and so it would be unwise and would go against the design of the GDB MI to hard-code the number of packets of output expected from the MI for each specific command. Thus, we use the concurrency model of the standard `asyncio` module to create a long-running thread that asynchronously waits on the MI's output. To separate synchonous (result) output and asynchronous (out-of-bad) output, we have a separate `asyncio.Queue` for each category of output, and the long-running thread submits each message it receives down the matching queue.

To support the upper-level methods, `BaseGDB` exposes the method `run_command` which accepts any string and submits it as a command to the MI asynchronously, and then collects the next packet from the synchonous queue. The asynchonous queue is left uncollected unless the client code explicitly collects it, whereas each synchronous packet is directly matched to a command.

To prevent vulnerabilities arising from target code impersonating as GDB, we separate the standard output and standard error of the inferior program (the program we are trying to debug) out into a separate channel. This is done using GDB's `tty` feature which redirects the inferior program's streams to a separate teletypewriter. Since obtaining a real teletypewriter is infeasible, we use the standard library module `os.openpty` that creates a pseudo-terminal and redirect the inferior program to it. This allows us to separate input and output to the inferior program and input and output to GDB, eliminating a prime attack vector. Reading from the inferior process is done asynchronously using `os.read` on an executor in a default thread pool, andd is present to the

client code through an asynchronous iterator. The MI's asynchonous packets are also presented in a similar manner, allowing access to both in an "Pythonic" manner. That is, instead of the typical "event-handler" approach to events, we can asynchronously iterate over each event, patiently awaiting until the next event arises, as the following code shows.

```
async for message in gdb.out_of_band_messages():
    print(f"Got message: '{message}'")
```

`GDB` uses the `async with` statement to implicitly manage the construction and destruction logic, eliminating the chance for client code to accidently forget to destruct `GDB` when they are done. The difference between the two approaches are as followsl

```
# Standard Approach:
gdb = await GDB().init()  # create an instance
await gdb.do_something1()
await gdb.do_something2()
await gdb.do_something3()
# here we forget to destruct gdb,
# potentially leaving a dangling process

# Context Approach:
async with GDB() as gdb:  # create an instnace
    await gdb.do_something1()
    await gdb.do_something2()
    await gdb.do_something3()
# here gdb is automatically destructed
# as we leave the `async with` block
```

The intermediate-level abstraction `GDB` layers over `BaseGDB` by providing methods useful for debugging C programs. The methods that are one-to-one equivalents of the MI's commands are as follows.

- `GDB.functions` returns a list of strings which are the names of all the user-defined functions in the inferior process. The underlying MI instruction is `-symbol-info-functions` which is applied as shown in the following example.

```
(gdb)
-symbol-info-functions
^done,symbols={debug=[{filename="target.c",fullname="/home/tosha/fluoresce/target.c",symbols
(gdb)
```

- `GDB.breakpoint` sets a breakpoint at the start of a given function and returns a unique breakpoint number. It is intended to be used with `GDB.functions` to set breakpoints at every user-defined command to step into user-defined functions (like `main`) while stepping over standard library functions (like `printf` and `malloc`). The underlying MI command is

`-break-insert` which is applied as shown in the following example.

```
(gdb)
-break-insert main
^done,bkpt={number="1",type="breakpoint",disp="keep",enabled="y",addr="0x000000000000133f",f
(gdb)
-break-insert createPerson
^done,bkpt={number="2",type="breakpoint",disp="keep",enabled="y",addr="0x0000000000001197",f
(gdb)
```

- `GDB.run` starts the inferior process, which will run until it exits or reaches a breakpoint set by `GDB.breakpoint`. The underlying MI command is `-exec-run` which is applied as shown in the following example.

```
(gdb)
-exec-run
=thread-group-started,id="i1",pid="446166"
=thread-created,id="1",group-id="i1"
=breakpoint-modified,bkpt={number="1",type="breakpoint",disp="keep",enabled="y",addr="0x0000
=breakpoint-modified,bkpt={number="2",type="breakpoint",disp="keep",enabled="y",addr="0x0000
=library-loaded,id="/lib64/ld-linux-x86-64.so.2",target-name="/lib64/ld-linux-x86-64.so.2",h
^running
*running,thread-id="all"
(gdb)
=library-loaded,id="/lib/x86_64-linux-gnu/libc.so.6",target-name="/lib/x86_64-linux-gnu/libc
~"[Thread debugging using libthread_db enabled]\n"
~"Using host libthread_db library \"/lib/x86_64-linux-gnu/libthread_db.so.1\".\n"
=breakpoint-modified,bkpt={number="1",type="breakpoint",disp="keep",enabled="y",addr="0x0000
~"\n"
~"Breakpoint 1, main () at target.c:62\n"
~"62\t    struct Person *head = NULL; // Initialize the linked list head to NULL\n"
*stopped,reason="breakpoint-hit",disp="keep",bkptno="1",frame={addr="0x000055555555533f",fun
(gdb)
```

- `GDB.next` will execute the next line of code in the inferior process. If the line is a standard library function call, it should be stepped over. If the next line is a function call and a breakpoint has been set for that function, GDB will step into it. The underlying MI command is `-exec-next` which is applied as shown in the following example.

```
(gdb)
-exec-next
^running
*running,thread-id="all"
(gdb)
*stopped,reason="end-stepping-range",frame={addr="0x000055555555534f",func="main",args=[],fi
(gdb)
```

- `GDB.frames` returns a list of strings, representing all the function names

of all the stack frames at the current state of paused execution. The frames are sorted by increasing age, so the first frame is the latest function call and the last frame is always `main`. The underlying MI command is `-stack-list-frames` which is applied as shown in the following example.

```
(gdb)
-stack-list-frames
^done,stack=[frame={level="0",addr="0x000055555555534f",func="main",file="target.c",fullname
(gdb)
```

- `GDB.variables` return a dictionary representing all the local variables and arguments for a given stack frame. If a stack frame is not specified, it defaults to the most recent frame of idnex zero. The dictionary contains key-value pairs of each variable's name to their value. The underlying MI command is `-stack-list-variables --thread 1 --frame <frame_index> --all-values` which is applied as shown in the following example. A caveat of this MI command is that it shows all the variables in the given frame, even if they haven't been yet defined and initialised in the C code. That is, it will list variables that have not yet been defined.

```
(gdb)
-stack-list-variables --thread 1 --frame 0 --all-values
^done,variables=[{name="head",value="0x0"},{name="pi",value="32767"},{name="pi_ref",value="(
(gdb)
```

There are also currently two functions that are composed of multiple MI commands which return a view of the inferior program's heap and stack at the current execution. They are as follows.

- `GDB.variable_info` returns more details about a given variable in a given stack frame. This is separate to `GDB.variables` as the former aims to be a one-to-one mapping of commands shipped by MI, while the latter focuses on retrieving information regardless. Specifically, it returns a tuple of four elements, representing `(var_type, var_value, var_address, var_children)`. The first three should be intuitive for any programmer, but the fourth, `var_children` is arbitrarily defined by GDB. For example, pointers, structs, struct pointers, and arrays are children since they contain references to other variables or spaces. The first two values in the tuple are obtained by creating a "GDB variable", inspecting it, and then deleting it using the MI commands `-var-create`, `-var-info-type`, `-var-list-children`, `-var-delete`. The last two values are more straightforward and obtained by the MI command `-data-evaluate-expression`. These commands are also sandwitched by frame switches using the MI command `-stack-select-frame` since these commands only operate on the "selected" frame. These instructions are applied as shown in the following example.

```
(gdb)
-stack-select-frame 0
```

```
^done
(gdb)
-var-create VARI * pi
^done,name="VARI",numchild="0",value="32767",type="int",thread-id="1",has_more="0"
(gdb)
-var-info-type VARI
^done,type="int"
(gdb)
-var-list-children VARI
^done,numchild="0",has_more="0"
(gdb)
-var-delete VARI
^done,ndeleted="1"
(gdb)
-data-evaluate-expression pi
^done,value="32767"
(gdb)
-data-evaluate-expression &pi
^done,value="0x7fffffffdbbc"
(gdb)
-stack-select-frame 0
^done
(gdb)
```

- `GDB.traverse` is the highest level method which encapsulates the task of traversing the inferior program's memory space starting from all available frame varaibles. Reusing other `GDB` methods, it iterates over each variable in each stack frame, collects information about each frame variable, and creates a mapping from memory addresses to values. It is effectively a BFS starting from the frame variables descending down the stack and heap tree, akin to the strategies used by garbage collectors to deallocate circular references. The final result is presented in a tuple of two dictionaries. The first dictionary maps local frame variables to metadata containing the variable's address, type signature, and name. The second dictionary is effectively a memory map of all allocations (both stack and heap), detailing the type and value at the allocation. An example of the return value is provided below.

```
({(0, 'freeList'): [Variable(name='head',
                             address='0x7fffffffdb78',
                             type='struct Person *'),
                    Variable(name='current',
                             address='0x7fffffffdb70',
                             type='struct Person *'),
                    Variable(name='temp',
                             address='0x7fffffffdb68',
                             type='struct Person *')],
```

```
 (1, 'main'): [Variable(name='head',
                        address='0x7fffffffdbc0',
                        type='struct Person *'),
               Variable(name='pi', address='0x7fffffffdbbc', type='int'),
               Variable(name='pi_ref', address='0x7fffffffdbb0', type='int *'),
               Variable(name='person1',
                        address='0x7fffffffdba8',
                        type='struct Person *'),
               Variable(name='person2',
                        address='0x7fffffffdba0',
                        type='struct Person *'),
               Variable(name='person3',
                        address='0x7fffffffdb98',
                        type='struct Person *')]},
{('0x7fffffffdb78', 'struct Person *'): Chunk(type='struct Person *',
                                              value='0x555555559810'),
 ('0x7fffffffdb70', 'struct Person *'): Chunk(type='struct Person *',
                                              value='0x555555559760'),
 ('0x7fffffffdb68', 'struct Person *'): Chunk(type='struct Person *',
                                              value='0x555555559810'),
 ('0x555555559810', 'struct Person'): Chunk(type='struct Person', value=None),
 ('0x555555559760', 'struct Person'): Chunk(type='struct Person',
                                            value={'name': 'Bob',
                                                   'age': 34,
                                                   'address': {'city': 'Los '
                                                                       'Angeles',
                                                               'state': 'CA'},
                                                   'next': '0x5555555596b0'}),
 ('0x555555559810', 'char [50]'): Chunk(type='char [50]', value=None),
 ('0x555555559844', 'int'): Chunk(type='int', value=23),
 ('0x555555559848', 'struct Address'): Chunk(type='struct Address',
                                             value={'city': 'Chicago',
                                                    'state': 'IL'}),
 ('0x5555555598b0', 'struct Person *'): Chunk(type='struct Person *',
                                              value='0x555555559760'),
 ('0x555555559760', 'char [50]'): Chunk(type='char [50]', value='Bob'),
 ('0x555555559794', 'int'): Chunk(type='int', value=34),
 ('0x555555559798', 'struct Address'): Chunk(type='struct Address',
                                             value={'city': 'Los Angeles',
                                                    'state': 'CA'}),
 ('0x555555559800', 'struct Person *'): Chunk(type='struct Person *',
                                              value='0x5555555596b0'),
 ('0x555555559848', 'char [50]'): Chunk(type='char [50]', value='Chicago'),
 ('0x55555555987a', 'char [50]'): Chunk(type='char [50]', value='IL'),
 ('0x555555559798', 'char [50]'): Chunk(type='char [50]', value='Los Angeles'),
 ('0x5555555597ca', 'char [50]'): Chunk(type='char [50]', value='CA'),
```

```
  ('0x5555555596b0', 'struct Person'): Chunk(type='struct Person',
                                             value={'name': 'Alice',
                                                    'age': 28,
                                                    'address': {'city': 'New '
                                                                        'York',
                                                                'state': 'NY'},
                                                    'next': '0x0'}),
  ('0x5555555596b0', 'char [50]'): Chunk(type='char [50]', value='Alice'),
  ('0x5555555596e4', 'int'): Chunk(type='int', value=28),
  ('0x5555555596e8', 'struct Address'): Chunk(type='struct Address',
                                              value={'city': 'New York',
                                                     'state': 'NY'}),
  ('0x555555559750', 'struct Person *'): Chunk(type='struct Person *',
                                               value='0x0'),
  ('0x5555555596e8', 'char [50]'): Chunk(type='char [50]', value='New York'),
  ('0x55555555971a', 'char [50]'): Chunk(type='char [50]', value='NY'),
  ('0x7fffffffdbc0', 'struct Person *'): Chunk(type='struct Person *',
                                               value='0x555555559810'),
  ('0x7fffffffdbbc', 'int'): Chunk(type='int', value=3),
  ('0x7fffffffdbb0', 'int *'): Chunk(type='int *', value='0x7fffffffdbbc'),
  ('0x7fffffffdba8', 'struct Person *'): Chunk(type='struct Person *',
                                               value='0x5555555596b0'),
  ('0x7fffffffdba0', 'struct Person *'): Chunk(type='struct Person *',
                                               value='0x555555559760'),
  ('0x7fffffffdb98', 'struct Person *'): Chunk(type='struct Person *',
                                               value='0x555555559810')})
```

The `GDB` module is designed entirely to faciliate interaction with GDB through a programming language, and so a typical usage of this module would be as shown in the following example.

```
async def log_reader(gdb: GDB):
    # Async coroutine that awaits for each out-of-band message
    # indefinitely, terminated when the event loop itself terminates
    async for message in gdb.out_of_band_messages():
        print(spark(message, [Misc.faint, Foreground.blue]))

async def output_reader(gdb: GDB):
    # Async coroutine that awaits for bytes produced by the
    # inferior process indefinitely, terminated when the event
    # loop itself terminates
    async for output in gdb.target_output():
        print(spark(f">>> {output}", [Misc.bold, Foreground.green]))

async with GDB("target.c") as gdb:
    # Register the indefinite coroutines with the `asyncio`
    # event loop
```

```
create_task(log_reader(gdb))
create_task(output_reader(gdb))

functions = await gdb.functions()  # get a list of user function
print(f"found functions: {functions}")
for function in functions:
    # break at each function to step into them
    print(f"breakpoint no. {await gdb.breakpoint(function)} added")
pp(await gdb.run())  # begin executing the inferior process

try:
    while True:
        await gdb.next()  # execute the next line of code
        # pp(await gdb.frames())  # optionally print the list of frames
        # pp(await gdb.variables())  # optionally print variables
        res = await gdb.traverse()  # collect the memory information
        print(spark(f"Info at report!", [Foreground.bright_yellow]))
        pp(res)  # print the information
except AssertionError as e:
    if "No registers." not in str(e):
        raise e
```

Note that `spark` is a mini module used to add color and other formatting to the terminal using ANSI codes.

Terminal User Interface Module

The user-facing component of this project is build using `Textual`, a Rapid Application Development framework for Python that aimes to enable developers to create sophisticated user interfaces with a simple Python API that can be run in the terminal as well as a web browser. It prides itself on its ability to be run remotely, integrated with the CLI, run on any platform, and enable developers to create interfaces in a fraction of the usual duration, and its minimal resource footprint. The Textual framework is architected quite similarly to the web library react, including a virtual DOM-style tree of components, and asynchronous event-handler functions.

Our TUI application aims to be as simple as possible with a single button and three panels. The first panel contains a view of the source code of the inferior process, the second panel contains text displaying the memory space produced by `GDB.traverse`, and the third panel contains the inferior program's output. Had we more time, the second panel would have been a visualisation instead of text. The single button steps through the program. In the name of simplicity, the frontend automatically sets breakpoints at `main` and all other user-defined functions and sets the inferior program to begin executing. Should an individual need more control, they should use the library module instead.

Conclusion

This project set out to improve the learning experience of novice programmers who chose to start their journey with the C language by bridging the gap in tooling for intuitive, plug-and-play, and uncomplicated debugging. The existing work done by other developers are indeed impressive but each lack a key component which this project aims to have.

Limitations and Next Steps

It can be said that the core components of this project has been established, but there are many areas in which time can be poured into. Some of the next steps that could be taken are as follows.

- Since GDB shows all variables of a function (or frame) even if they haven't been declared or initialized, we inadvertently attempt to access uninitialised values which yield garbage and destroy our lexers. We have temporarily dealt wiht uninitialised garbage using the compile flag `-ftrivial-auto-var-init=zero` but have no way to deal with deallocated garbage (i.e., reading heap memory after it has been freed), and so anticipate a proper fix using a custom-built parser instead of a monkey-patched JSON parser.
- We are yet to polish the corners. Most notably, we need proper handling of expected exceptions like the inferior program exiting normally, since it currently throws an unhandled AssertionError, and proper handling of other errors to shut down gracefully ant not result in a zombie-like state where multiple interrupt signals are required to terminate the code.
- We should also question the value of initialising `GDB` with the `async with` statement, as building the TUI has led to a crossways where it is not possible to organically initialise `GDB`. Speicifcally, the `async with` statuement must be run in an `asyncio` event loop, but `Textual` crashes unless it creates its own event loop, so `GDB` has to be created with too little scope to be useful. The current suboptimal solution is to manually call the dunder methods `__aenter__` and `__aexit__` which should be called by the Python interpreter.
- We should question the necessity of real-time stepping and information feedback. That is, referencing `rr`, we should consdier the value of executing the inferior process all at once while recording information at each step of execution, and then present that information to the user. This could potentially improve the user experience as it frontloads the loading time to the start of the debugging process, improving the latency between stepping, and enabling the user to step "backwards" in execution.

References

Existing Work

- “gdbgui”: https://www.gdbgui.com
- “pygdbmi”: https://github.com/cs01/pygdbmi
- “pygdbmi async issue”: https://github.com/cs01/pygdbmi/issues/41
- “lldbg”: https://github.com/zmeadows/lldbg

Solution

- “GDB MI documentation”: https://sourceware.org/gdb/current/onlinedocs/gdb.html/GDB_002fMI.html
- “Textual documentation”: https://textual.textualize.io
- “rr main page”: https://rr-project.org