Step Debugging with GDB

CS14 - S26

Debugging

The process of investigating and editing existing code to meet some expected execution.

Many forms of bugs, so there are many forms of debugging:

- Memory usage errors → memcheckers, e.g. Valgrind
- Inefficient memory/compute usage → profilers, e.g. perf, strace
- Protocol failures → verbose logs, dump files, e.g. tcpdump, -v option
- Algorithmic errors → tracing, step debugging, e.g. **GDB**

Tracing

In your mind, or on paper, going through each operation to imagine if what the code does is what you expect. Can be a painstaking process.

Students often add print statements to investigate if values match what they expect at different points in the code.

(In production code, this kind of output is sent to log files, or controlled by verbosity options)

The idea is to compare expectation vs observation.

Step Debugging

Trace by actually executing the code line by line. At each line, allow investigation of values to see if they match expectations.

Common features:

- Execute single line of code: “step”
- Expand function calls, or treat them as single lines: “step in”
- Complete a function call: “finish”/”step out”
- Jump between stack frames/function calls
- Execute up to a specific line of code: “breakpoint”
- Print/track specific variables

Step Debugging

Most languages have a dedicated step debugging tool:

- GDB/LLDB: C, C++, Rust
- JDB: Java
- PDB: Python
- ocamldebug: Ocaml
- etc.

GDB - Gnu Debugger

Debugger for Assembly, C, C++, Rust

Compile with the `-g` flag to embed debugging info in executable.

Start `gdb` by calling it with name of executable (can also be attached to a running program by PID).

Use the `set args` command to set command line arguments if necessary.

```
$ gcc -g -o my_prog my_prog.c

$ gdb my_prog

...

(gdb) set args arg1 arg2
```

`break <func/line>`	Set a breakpoint
`run`	Start program running from the beginning
`set args <arg1> <arg2> ...`	Set the command line arguments that the program will use
`cont`	Continue execution of the program until it hits a breakpoint. Pauses just before the line of the breakpoint executes
`next`	Execute the next line of C code, treating function calls as a single line
`step`	Execute the next line of C code; if the next line contains a function call, step into the function and pause
`finish`	Execute through the end of the current function call
`print <var>`	Print the value of a variable once
`display <var>`	Repeatedly print the value of a variable after each line of execution
`where`	Print the call stack
`frame <frame num>`	Move into the context of a specific stack frame

Example 1 - segfaulter.c

Copy example programs from

```
/home/ckazer/public/cs14/inclass/gdb_examples
```

Compile with:

```
gcc -g -o segfaulter segfaulter.c
```

Skim through the code.
What do you *expect* to happen?
What do you *observe* happens when you try to run the executable?

GDB Text User Interface Mode (TUI Mode)

GDB has a builtin ncurses environment that is a little more interactive.

- `layout src` – add a split window that shows the source code
- `layout asm` – add a split window that shows the source assembly
- `focus <cmd/src/asm>` – change arrow key control between different splits

Caveat: stdout/stderr output conflicts with the TUI. This output will be displayed near the bottom of the screen, mixed in with TUI elements.

Use CTRL-L (the clear shortcut) to refresh the TUI.

Example 2 - badprog.c

Compile with:

```
gcc -g -o badprog badprog.c
```

Skim through the code.
What do you *expect* to happen?
What do you *observe* happens when you try to run the executable?

Example 3 - revflip.c

Compile with:

```
gcc -g -o revflip revflip.c
```

Skim through the code.
What do you *expect* to happen?
What do you *observe* happens when you try to run the executable?

Circular Array

Idea: Suppose we have limited space to store an infinite amount of information, but old information isn’t important.

Use a fixed capacity array, but whenever we get a new piece of information, overwrite the oldest element with the new info. Just go from front to back, and wrap around.

Could be used to implement terminal history, where oldest commands are evicted first.

Circular Array

Keep track of:

- **Array**
- **Capacity** - the number of elements the array can hold
- **Filled** - the number of slots in the array that are holding elements
- **Next** - the index of the array that the next element should be put into. This will wrap around to 0 once it reaches the same value as capacity.

Exercise: Write an insert function for a circular array

Circular Array - Insert

```
insert(element, array, capacity, filled, next):
    array[next] = element
    if filled < capacity:
        filled += 1
    next += 1
    next = next % capacity
```

Resizing Array

Idea: Suppose we have to insert an uncertain number of elements into an array.

Arrays in C/C++ are a fixed size. We want an array that automatically doubles in capacity (the maximum number of elements it can hold) whenever it becomes full.

Useful as part of an ArrayList implementation; take CS35 for more info!

Resizing Array

Keep track of:

- **Array**
- **Capacity** - the number of elements the array can hold
- **Size** - the number of elements currently held in the array

Exercise: Write insert and double_capacity functions for a resizing array

Resizing Array – Insert

```
insert(element, array, size, capacity):
    array[size] = element
    size += 1
    if size == capacity:
        double_capacity(array, capacity)
```

Resizing Array – Double Capacity

```
double_capacity(array, capacity):
    old_array = array
    new_array = [2 * capacity]
    for each element in old_array:
        copy element to new_array
    free old_array
    array = new_array
    capacity = 2 * capacity
```

GDB Interfaces

- GDB command line (default)
- GDB TUI
- VScode GDB integration
- GDB Dashboard