Debugging

```
segmentation fault (core dumped)
```

Overview

- `printf` debugging
- Logging
- GDB
- Checking memory with Valgrind

`printf` debugging

- Intuitive: just print stuff out at certain points
  - What if you're done littering your file with debug prints?
- With the power of preprocessors, we can turn it on and off!

```
#ifdef DEBUG_PRINT
printf("This be a debug message\n");
#endif // DEBUG_PRINT
```

```
#ifdef DEBUG_PRINT
  // this is known as a "variadic macro"
  #define dbgprintf(fmt, ...) printf("DEBUG: " fmt, ##__VA_ARGS__)
#else
  #define dbgprintf(fmt, ...)
#endif
// ...
dbgprintf("This be a debug message\n");
```

Logging

- An extension on printing
- Provide different verbosity/logging levels
- Set your verbosity level to increase/decrease the amount of logging
  - More logging uses more resources
- Log to standard output, standard error, or to some file

Common logging levels

- Fatal
  - "We can't continue, I shall die now"
- Error
  - "Something went wrong"
- Warning
  - "Something weird *might* be going on"
- Info
  - "Hey a cool thing happened"
- Debug
  - "A thing happened, here's some details"
- Trace/Verbose
  - "Here's everything that's going on"

GDB (GNU Debugger)

- Debugging tool that lets you look around during execution
- Once again, this tool is pretty deep: look at the GDB manual for details
- A neat video
- We'll go over some big overarching concepts and features
  - Interface
  - Breakpoints and watchpoints
  - Stack frames
- If you want to follow along, you can install the `gdb` package on Ubuntu or use the course server
- Example file: https://www.eecs.umich.edu/courses/eecs201/fa2022/files/examples/debug/func.cpp

Interface

- Invoking: `$ gdb [options] [executable file] [core file]`
  - `$ gdb ./myapp`, `$ gdb myapp`
- Hitting return/enter without anything will repeat the previous command
- Entering incomplete commands (such as a single) letter will run a command if there is no ambiguity:
  - `r` -> `run`
  - `n` -> `next`
  - `b` -> `break`
- Also has an approximation of a windowing interface in "Text User Interface" (TUI) mode
  - `tui enable`, `tui disable`
  - C-x s ('C' being control): single-key mode (e.g. hitting 'n' will execute "next")
  - C-x o: switch window focus

Commands

- `run [arguments] [file redirects]`
- `next [count]`: step *over* functions, "next line"
- `step [count]`: step *into* functions
- `finish`: step *out* of current function
- `print <expression>`: print expression (e.g. variables)
- `list [location]`: list source code
- `break <location>`: set breakpoint
- `watch <expression>`: set watchpoint
- `info breakpoints`, `info watchpoints`: list break/watchpoints
- `where`, `backtrace`, `bt`: list stack frames
- `frame <stack frame>`: change stack frame

Breakpoints

- Stop at a certain location in the program
- Can be conditional!
- `info breakpoints`, `info break`, `info b` will list breakpoints
- Examples:
  - `break 20`
  - `break main.cpp:21`
  - `break main.cpp:21 if argc == 4`
  - `break coolfunction`

Watchpoints

- Stop when an expression changes
- `info watchpoints`, `info watch` will list watchpoints
- Examples:
  - `watch somevar`
  - `watch a + b`
- `disable <number>`: disable a break/watchpoint
- `delete <number>`: delete a break/watchpoint

(Catchpoint: stop when an event such as a C++ exception occurs)

Stack frames

- A *stack frame* holds all information local to a particular function call
  - Local variables
  - Arguments
  - (Return address)
- Function calls will push a frames on the *stack*
- Function returns will pop the frame off the *stack*
  - `where`, `backtrace` (`bt`) can show us the current stack frames
  - `frame <number>` can have us switch to a stack frame so we can look at its variables

Valgrind

- General dynamic analysis tool
  - Valgrind manual
- Most known for its Memcheck tool for checking memory accesses (which we'll be focusing on)
  - Memcheck manual entry
- Super useful at finding things like:
  - Memory leaks
  - Use-after-frees
  - Invalid reads
  - Use of uninitialized variables
- Easy to invoke:
  - `$ valgrind ./myapplication`
  - `$ valgrind --leak-check=full ./myapplication`

Valgrind
Playing with it

- Example file: https://www.eecs.umich.edu/courses/eecs201/fa2022/files/examples/debug/badmem.cpp

Closing thoughts

- Ultimately use the right tool for the job
- GDB doesn't work particularly well in complex systems
  - Logging can help out here, but it does incur some overhead
- Valgrind can seriously slow down your program