CSCI 104

Lab 2: GDB

What is a debugger?

- A **debugger** is a tool used to inspect a program while it is running.
- It runs through a program and pauses when it hits a **breakpoint**.
- Different commands can evaluate variables, call functions, and examine the call stack.
- You can then resume your program, or step through it line-by-line.

Identify the Issue

The most common two ways your code will terminate (besides a successful execution) is by:

1. **Segmentation Fault, a.k.a. Segfault (SIGSEGV)** — When a program tries to read or write outside the memory that is allocated for it, or to write memory that can only be read.
2. **Abort (SIGABRT)** — An error detected by the program itself.

These will automatically trigger the debugger to break, so you don't have to.

Identify the Issue

The other issues could be:

- **Infinite loop or recursion:** caused by faulty logic or base case.
- **Logic:** 2 + 2 == 5? code is correct but the logic is faulty.
- **Translation error:** the logic is correct but it was just coded incorrectly.

Isolate the Problem

Hypothetically, you could look through every line of code and try to find the bug. This is TEDIOUS. You can be efficient by isolating the line, or at least section, in which the problem is occurring.

- **Carefully place breakpoints.** Break at the start of the area where the problem might be occurring. If you have to step through a lot of code, you might miss something important.
- **Print important variables.** Use the print GDB command to check the values of any suspicious variables. Calling print on a struct or class will print out all the member variables.
- **Use cerr statements.** To localize a fault to a specific area of the code or trace the flow of an entire program, it's invaluable to just print out important values and messages throughout your program. Remember to use cerr rather than cout so your output is guaranteed to be flushed to the terminal before the program terminates.

GDB Cheat Sheet

https://bytes.usc.edu/cs104/wiki/gdb/

Running the Program

- `run/r [arguments]`
  - runs the program with the given arguments.

Setting Breakpoints

- **`break/b [file.cpp:line number]`**
  - puts a breakpoint at the given line number in the given file. Note that if you only have one file, just **`break [line number]`** will suffice.
- **`break/b [function name]`**
  - places a breakpoint at the start of the given function.

Clearing Breakpoints

- **`clear [file.cpp:line number]`**
  - clears a breakpoint at the given line number in the given file. Note that if you only have one file, just clear `[line number]` will suffice.
- **`clear [function name]`**
  - removes the breakpoint on the given function. Note: function breakpoints will not work on functions that take strings as arguments (it's complicated) on your course VM due to an incompatibility between GCC and GDB. However, this will work properly on newer systems.

Changing the Layout

- `layout next`
  - From the beginning of GDB, entering ‘layout next’ once the program is running will show you source code around your current location in the program. This view can be helpful to those who are new to gdb, and especially helpful when working with source code you are not familiar with. Repeating ‘layout next’ shows your program in assembly language.
- `layout prev`
  - Takes you back to the previous layout mode.

Function Call Stack

- `bt`
  - shows the function call stack, every function that you've run through since the line you've arrived at.
- `frame [number]`
  - goes to the selected frame in the call stack.
- `where`
  - displays the current line and the function stack of calls that got you there.

Stepping through the Program

- **`continue/c`**
  - continues the program after being stopped by a breakpoint.
- **`next/n`**
  - executes the current source line and moves it to the next one. **Will skip over any function calls.** i.e. using **`next`** on the line `x = getValue(y)` would move you to the next line in the current function call, ignoring what happens in `getValue(y)`. Useful for when you're testing out a function and you KNOW your helper functions work.
- **`step/s`**
  - executes the current source line and moves it to the next one. **Will step INTO any function calls.** i.e. using **`step`** on the line `x = getValue(y)` would move you into the `getValue(y)` function.

Printing Variables

- `print/p [variable]`
  - prints out the variable value. If you pass it a class/struct instance, it will print all the data members in the class.
- `display/d [variable]`
  - is like print, but reprints the information after every instruction.

Finishing Up

- `finish/f`
  - executes the rest of the current function. Will step OUT of the current function.
- `quit`
  - exits GDB.