gdb A Quick Start

Sources:

- https://www-users.cs.umn.edu/~kauffman/2021/gdb.html
- https://www.fayewilliams.com/2011/07/13/gdb-conditional-breakpoints/
- https://www.cse.unsw.edu.au/~learn/debugging/modules/all_gdb/

Start and stop `gdb`
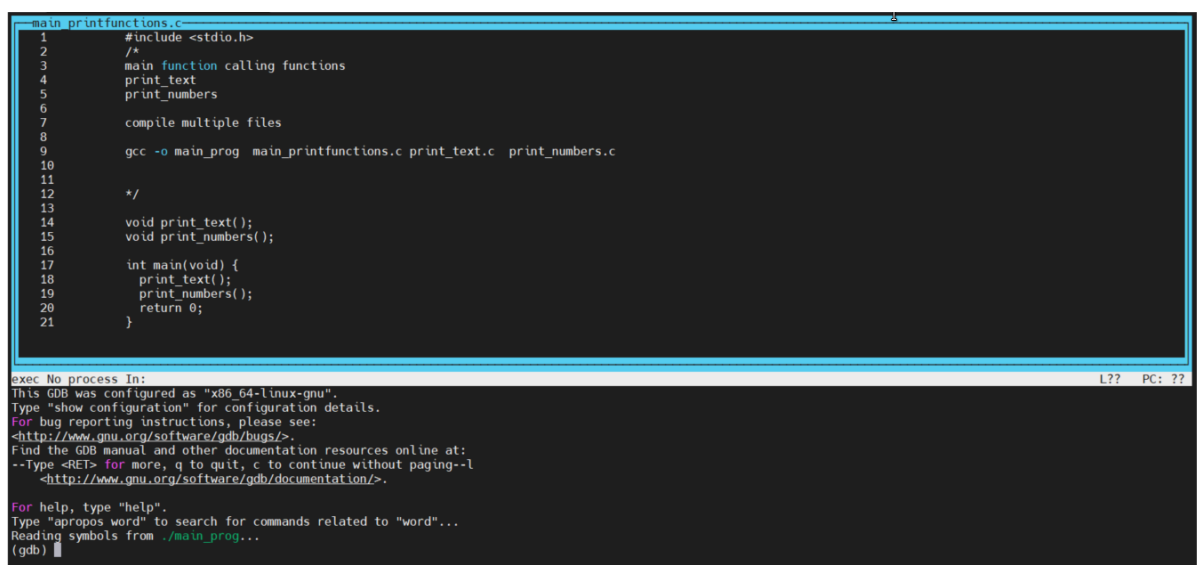
To run your program under `gdb`'s control, type:

```
gdb program_name
```

at the prompt. Do not enter your program's command line arguments (if needed), this can be done in the debugging environment.

If you are a beginner, it is best to start in *Text User Interface (TUI)* mode. It shows

- Commands and history towards the bottom,
- Source code position towards the top

```
gdb -tui program_name
```


The screen will occasionally get "messed up" which can be corrected by pressing `ctrl-L` which will force a redraw of the terminal screen.

To exit GDB, type

```
quit
```

at the GDB prompt.

gdb supports tab autocompletion, start typing a command and it can be completed, or an overview of the existing commands is giving by pushing the tab-key.

gdb also supports *short command names*, use only the first character of a command b[reak]: b instead of break; r[un]: r instead of run; etc.

Setting breakpoints in `gdb`

A *breakpoint* indicates a place that execution will be stopped in a program by the debugger. They can be created in a variety of ways and removed when no longer needed.

Command	Effect	Notes
`b[reak]`	Stop running at the current line	
`b[reak] function name`	Stop running at the beginning of `function name`	
`b[reak] line number`	Stop running at line 100	
`b[reak] main`	Stop running at the beginning of `main`	Easy way to put a break at the beginning of the program
`info breakpoints`	Show all breakpoints with locations	
`disable 2`	Don't stop at breakpoint #2 but keep it there	
`enable 2`	Stop at breakpoint #2 again	
`clear function name`	Remove breakpoint at the beginning of `function name`	
`delete 2`	Remove breakpoint #2	

Conditional breakpoint

In GDB you can specify a condition in the programming language you are debugging and apply it to any breakpoint. Let's stop a loop at the *n*th iteration

```
(gdb) b 16 if i == 99
```

In this case the execution will stop once the counter I reaches 99

To ensure gdb stops execution, use the first line of code inside the loop as the stopping point, not the loop itself.

You can also specify a condition on an existing breakpoint by using the breakpoint number as a reference (can be found using `info breakpoints`)

```
(gdb) cond 3 i == 99
```

Arguments and Running

After loading a program and setting a few breakpoints, typically one runs it. Command line arguments are also often necessary.

Command	Effect	Notes
`set args arguments`	Set command line arguments to arguments	Arguments are separated by a blanc
`show args`	Show the current command line arguments	
`r[un]`	Start running the program from the beginning	Will run to complete unless a breakpoint is set
`r[un] arguments`	Another way of passing the arguments	Ex. run 12 15.5 The numbers 12 and 15 are passed as arguments
`r[un] < file name`	Get input from a file through redirection	
`kill`	Kill the running program	Usually done to re- run the program before hitting a failure
`file program`	Load program and start debugging	

Stepping

When a breakpoint is hit, single steps forward are possible in the debugger to trace which path of execution is taken. Use `step` to move into functions line by line and `next` to stay in the current function stepping over function calls.

Command	Effect	Notes
`s[tep]`	Step forward one line of code	steps inside any functions called within the line.
`s[tep] count`	Step forward *count* lines of code	If a breakpoint is reached before *count* steps, stepping stops right away.
`s[ext]`	Step forward one line of code	Similar to step, but does not go into functions
`n[ext] count`	Step forward *count* lines of code	
`stepi`	Step a single assembly instruction forward	stepi goes into functions
`nexti`	Step an assembly instruction forward over functions	nexti does not go into functions
`c[ontinue]`	Continue running until the next breakpoint	

Printing Values

Inspecting values is often necessary to see what is going on in a program. The debugger can display data in a variety of formats including formats that defy the C type of the variable given.

Command	Effect	Notes
`p[rint] variable`	Print value of variable a which must be in the current function	Formats a according to its data type
`p[rint]/f variable`	Print value according to the format specifier x: hexadecimal o: octal s: string …	
`x address`	Examine memory pointed to by address	

Command History and Screen Management in TUI Mode

A bit of basic editing discipline is useful in gdb. The Text User Interface `-tui` mode alter keys so that arrows don't work normally. However, TUI mode shows the source code position in the upper area of the terminal which most users find to be helpful.

Command/Keystroke	TUI -tui mode	Normal Mode
`Ctrl-l`	Re-draw the screen to remove cruft (do this a LOT)	Clear the screen
`Ctrl-p`	Previous command, repeat for history	Same
`Ctrl-n`	Next command if at a previous	Same
`Ctrl-r`	Interactive search backwards	Same
`Ctrl-b`	Move the cursor back (left) a char	Same
`Ctrl-f`	Move the cursor forward (right) a char	Same
Up / Down arrows	Move the source code window up/down	Previous/Next Commands
Left / Right arrows	Move the source code window left/right	Move cursor left/right
`list`	List source code	Show 10 lines of source code around current execution point