**1**

1. How to enable built-in debugging in gcc?

Use –g flag to compile.

1. What is the meaning of GDB?

The GNU Project Debugger.

1. Compile the C program from homework 4 with debugging enabled.

gcc –g –o cthread cthread.c

**2**

1. Find the homepage of the GDB project.

<https://www.gnu.org/software/gdb/>

2. What languages are supported by GDB?

Ada, C, C++, Objective-C, Free Pascal, Fortan, Go…

3. What are the following GDB commands doing:

• backtrace: Print backtrace of all stack frames, or innermost COUNT frames. With a negative argument, print outermost –COUNT frames. Use of the ‘full’ qualifier also prints the values of the local variables. Use of the ‘no-filters’ qualifier prohibits frame filters from executing on this backtrace.

• where: Same as backtrace

• finish: Execute until selected stack frame returns. Usage: finish Upon return, the value returned is printed and put in the value history.

• delete: Delete some breakpoints or auto-display expressins. Arguments are breakpoint numbers with spaces in between. To delete all breakpoints, give no argument. Also a prefix command for deletion of other GDB objects. The “unset” command is also an alias for “delete”.

List of delete subcommands:

delete bookmark – Delete a bookmark from the bookmark list

delete breakpoints – Delete some breakpoints or auto-display expressions

delete checkpoints – Delete a checkpoint(experimental)

delete display – Cancel some expressions to be displayed when program stops

delete mem – Delete memory region

delete tracepoints – Delete specified trace points

delete tvariable – Delete one or more trace state variables

• info breakpoints: status of specified breakpoints (all user-settable breakpoints if no augument). The “Type” colume indicates one of:

breakpoint - normal breakpoint

watchpoint - watchpoint

The “Disp” column contains one of “keep”, “del”, or “dis” to indicate the disposition of the breakpoint after it gets hit. “dis” means that the breakpoint will be disabled. The “Address” and “What” columns indicate the address and file/line number respectively.

Convenience variable “$\_” and default examine address for “x” are set to the address of the last breakpoint listed unless the command is prefixed with “server “.

1. Search the documentation and explain how to use conditional breakpoints. Watch this youtube video and answer the following questions.

|  |  |  |
| --- | --- | --- |
| 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 99th iteration (I’m debugging C/C++, so my conditions are written in C/C++):  (gdb) b Message.cpp:112 if i == 99   |  | | --- | | You can also specify a condition on an existing breakpoint by using the breakpoint number as a reference:  (gdb) cond 3 i == 99 | | And remove a condition from a breakpoint using:  (gdb) cond 3  Breakpoint 3 now unconditional. |   (gdb) cond 2 \*p == 'r'  //stop if the char\* pointer p points to the letter 'r'  Reference: <https://www.fayewilliams.com/2011/07/13/gdb-conditional-breakpoints/> |
|  |

1. What is -tui option for GDB?

It enables the TUI(Text User Interface) mode. The GDB Text User Interface (TUI) is a terminal interface which uses the curses library to show the source file, the assembly output, the program registers and GDB commands in separate text windows. The TUI mode is supported only on platforms where a suitable version of the curses library is available.

1. What is the “reverse step” in GDB and how to enable it. Provide the key steps and commands.

GDB can allow you to “rewind” the program by running it backward.

reverse-continue [*ignore-count*]

rc [*ignore-count*]

Beginning at the point where your program last stopped, start executing in reverse. Reverse execution will stop for breakpoints and synchronous exceptions (signals), just like normal execution. Behavior of asynchronous signals depends on the target environment.

reverse-step [*count*]

Run the program backward until control reaches the start of a different source line; then stop it, and return control to GDB.

Like the step command, reverse-step will only stop at the beginning of a source line. It “un-executes” the previously executed source line. If the previous source line included calls to debuggable functions, reverse-step will step (backward) into the called function, stopping at the beginning of the *last* statement in the called function (typically a return statement).

Also, as with the step command, if non-debuggable functions are called, reverse-step will run thru them backward without stopping.

reverse-step [*count*]

Run the program backward until control reaches the start of a different source line; then stop it, and return control to GDB.

Like the step command, reverse-step will only stop at the beginning of a source line. It “un-executes” the previously executed source line. If the previous source line included calls to debuggable functions, reverse-step will step (backward) into the called function, stopping at the beginning of the *last* statement in the called function (typically a return statement).

Also, as with the step command, if non-debuggable functions are called, reverse-step will run thru them backward without stopping.

set exec-direction

Set the direction of target execution.

Reference: <https://sourceware.org/gdb/onlinedocs/gdb/Reverse-Execution.html>