[Back to Resources]

GDB

by Anonymous

To launch your program using gdb, run the following command:

```
gdb [program name]
```

To run your program with optional command line arguments:

```
(gdb) run [arguments]
```

Alternatively, you can do this in one line with the following command:

```
gdb --args ./program_name [optional] [args] [here]
```

This allows you to simply type

```
(gdb) run
```

to start your program.

Note Throughout the lab, we'll use the notation

```
(gdb) command...
```

to indicate that the `command` should be run from within GDB.

Tip GDB will provide several helpful features. First, it will output similar debugging information as Valgrind upon errors such as segmentation faults. Second, and more important, it allows you to stop the program execution, move around, and view the state of the running program at any point in time.

To do that, we will use the following common commands (see more details in the slides). We'll also define the abbreviations of these commands, so you don't have to type the full names of these commands when you want to use them.

- **Walking through your code.**
  - `break [file:line number]`
    - *Example usage*: `break skipList.cpp:40`
    - Create a breakpoint at the specified line. This will stop the program's execution when it is being ran. (See `run`).
    - When your program is stopped (by a previous use of `break`) in a certain file, `break n` will create a breakpoint at line `n` in that same file.
    - **Note**: There are other variations on how to use `break` here. One variation is breaking at a function belonging to a class. *Example*: `break SkipList::insert`.
    - Abbreviation: `b`. *Example usage*: `b skipList.cpp:40`
  - `clear [file:line number]`
    - Removes a breakpoint designated by `break`.
  - `run (arguments)`
    - Runs the program, starting from the main function.
    - Abbreviation: `r`.
  - `list`
    - Shows the next few lines where the program is stopped.
  - `layout src`
    - Shows an updating window with your source code and the current line of execution.
    - Usually easier than type `list` every line or referring back to your open code
  - `next`
    - Continues to the next line executed. This does not enter any functions. (See `step` for this).
    - Abbreviation: `n`.
  - `step`

        - Continues to the next line executed. Unlike `next`, this will *step* into any proceeding functions
        - Abbreviation: `s`.
    - `finish`
        - Steps out of a function.
        - Abbreviation: `fin`.
    - `continue`
        - Continues the execution of the program after it's already started to run. `continue` is usually used after you hit a breakpoint.
        - Abbreviation: `c`.
- **Viewing the state of your code.**
    - `info args`
        - Shows the current arguments to the function.
        - If you are stopped within a class's function, the `this` variable will appear.
    - `info locals`
        - Shows the local variables in the current function.
    - `print [variable]`
        - Prints the value of a variable or expression. *Example*: `print foo(5)`
        - The functionality of `print` is usually superseded by `info locals` if you are looking to print local variables. But if you want to view object member variables, `print` is the way to go.
        - *Example*: `print list->head`. Or `print *integer_ptr`.
        - Abbreviation: `p`.
    - `display [variable]`
        - Display the value of a variable or expression every time you iterate through the code. Unlike `print`, `display` is persistent. *Example*: `display foo(5)`
        - *Example*: `display list->head`. Or `display *integer_ptr`.
    - `backtrace`
        - Shows the call stack of your program
        - The list of which function has called the function you are in, recursively
    - `frame [n]`
        - Used to go to the frame numbers as seen in backtrace
- **Other useful commands.**
    - `ctrl-l` (clears the screen)
    - `ctrl-a` (moves cursor to beginning of prompt)
    - `ctrl-e` (moves cursor to end of prompt)
    - `ctrl-o` (lets you switch between `layout` window and gdb prompt)
- **Add C++ STL Support For libc++**
    - If you are a Mac user, you are probably going to prefer using `lldb` over `gdb` to print the STL data types.
    - If you are an On Your Own Machine user in general, you will probably prefer installing `lldb` over using the pretty printer for `gdb`.
    - The following instructions will help you be able to print C++ STL structures like vectors nicely in `gdb`.
        1. Make a directory for `gdb` pretty printers with the command `mkdir -p ~/gdb_printers/python`
        2. Clone the pretty printer source code. `git clone https://github.com/koutheir/libcxx-pretty-printers ~/libcxx-pretty-printers`
        3. Move the pretty printer folder inside the repo into your `~/gdb_printers/python` directory using `mv ~/libcxx-pretty-printers/src/libcxx ~/gdb_printers/python`
        4. Remove the `~/libcxx-pretty-printers` directory. `rm -rf ~/libcxx-pretty-printers`
        5. Now, lets setup the `~/.gdbinit` file to load the `gdb` pretty printer.
            - If you do not have a `~/.gdbinit` file, run the following command.

            ```
            echo "python\nimport sys\nsys.path.insert(0, '$HOME/gdb_printers/python')\nfrom libcxx.v1.printers import register_libcxx_printers\nregister_libcxx_printers (None)\nend\n" > ~/.gdbinit
            ```

            - If you do have pretty printers setup in your `~/.gdbinit`, add the following lines before the end statement.

            ```
            from libcxx.v1.printers import register_libcxx_printers
            register_libcxx_printers (None)
            ```