Using gdb, C++, and Emacs*

Jeffrey D. Oldham

2000 Mar 22

Contents

1 Introduction

A *debugger* is a tool permitting your program to run in "slow motion" so you can examine how your program runs or why it crashes. With a debugger, one can

- Run your program.
- Make your program stop on specified conditions.
- Examine your program when it is stopped.

- Change things in your program.

Since the Gnu debugger `gdb` works nicely with `emacs` and `Xemacs` so we will be describing it. We assume you are running either of these editors.

2 Obtaining More Information About `gdb`

`gdb` has many more features than can be described here. Other sources of information include

- using the `help` command inside `gdb`.
- a short introduction to `gdb`, `make`, `gcc`, and `emacs` for Stanford students
- `emacs` info pages, available inside `emacs` by typing `C-h i` and then `m gdb`.
- the online gdb manual, a definitive source of information

3 A Short Example of Using `gdb`

3.1 Compiling Your Program

The debugger needs more information than the compiler usually places in the executable so compile your program using the `-g` compiler flag. For example,

```
g++ -g -Wall -pedantic factorial.cc -o factorial
```

Here is the `factorial.cc` program.

3.2 Running Your Program

To run a program named `factorial` using `gdb` inside `emacs`, type `M-x gdb`. (The `M-x` is Meta-x, probably Alt-x or ESC-x on your keyboard.) Enter `factorial` when queried for the executable's name. A `gdb` buffer will appear.

To starting running `factorial`, type

```
run
```

The program instantly dies. We could read the source code to find the problem, but let's use the debugger instead.

To stop execution when the program enters the `main` function, type

```
break main
```

Then, type `run` to start running the program again. Execution halts when starting the `main` function. `Emacs` splits the screen in half, displaying the program line that is being executed.

To execute the next line, type the

`next`

command. Hitting return will repeat the last command. Hitting return a few more times indicates we are missing a command-line argument, and the program will be aborted.

To specify command-line arguments, type them after `run`. For example,

`run 4`

Repeatedly invoking `next` demonstrates how the program runs. If we wish to watch how the `factorial` function works, we should

`step`

into the function when the debugger reaches the function call. `step` steps into function calls while `next` moves to the next line. Continuing `stepp`ing, we can watch each recursive call of the `factorial` function.

Starting each function call adds a *frame* to the stack while finishing a function call removes a frame from the stack. To inspect the stack, i.e., to see a listing of all the frames, try

`backtrace`

Another way to inspect calls to `factorial`, is to set a breakpoint at its beginning. One could type `break factorial`, but instead move the cursor the function's first line and type C-x *space*. `gdb` will indicate a breakpoint has been set. Then, start running the program again. To continue execution until reaching the next breakpoint, type

`continue`

To finish the the execution of the current function, type `finish`.

3.3 Another Short Example

When dealing with pointers and dynamic memory, segmentation faults are frequent occurrences. `gdb` will automatically stop execution when one occurs. Try running this program. To determine its state when it halts, use the `backtrace` command. To move up and down in the stack, use the `up` and `down` commands. Some stack frames are for functions automatically called by the compiler or by other functions.

In this particular case, we learn the segmentation fault occurs in the destructor, but it is hard to tell why. Solving this problem requires thought beyond what the debugger can reveal, but we can watch `obj1`'s and `obj2`'s values change as the program executes. To print the values of `*(obj1.b)` or `*(obj2.b)`, use

```
print *(obj1.b)
```

If we forget the type of `obj1`, we can learn that by typing

```
ptype obj1
```

Somewhere `obj2`'s value changes from `true` to `false`. To learn that, we can ask `gdb` to display values of expressions after each command. For example,

```
display *(obj1.b)
```

After passing `obj2`'s declaration, we can do the same for it.

```
display *(obj2.b)
```

Thus, we learn that its value changes somewhere near the `set` call.[1]

Enjoy!

[1] An alternative to the `display` command is the `watch` command.