CSC 325 Lab 2 Supplement

C programming and debugging with GDB

Table of Contents

1. Goals for this week:

1. Read user input from "standard in" (`stdin`).
2. Learn some basic `gdb` debugging commands. You should start using GDB to help with debugging your Lab 2 code. Over the course of the semester we will revisit using `gdb`, introducing more commands and features.
3. Introduce and get started on Lab 2.

2. Input in C

First, let's look at an example of reading input values into a C program.

2.1. Reading user input (from stdin)

The `scanf` function from the C stdio library can be used to read in different type values entered by the user (this is known as reading input from "standard in", or `stdin`).

Note that `scanf` needs to know the memory location of where to put the values read in, and we use the & (ampersand) operator on a variable name to pass this location to `scanf`. We'll talk **much** more about what that ampersand means as we build up our C programming skills in future assignments.

String format codes

For both `printf` and `scanf`, the following formatting codes (also known as "conversion codes" or "conversion characters) are likely to be most useful to you:

`%d` (signed) integer

- `%u` unsigned integer
- `%x` hexadecimal (lower case)
- `%X` hexadecimal (upper case)
- `%c` character
- `%f` floating point value

3. GDB intro

Next, we'll look at the GNU Debuggger (GDB), whose command is `gdb`. GDB helps programmers debug C and C++ programs.

Over the course of the semester, we'll explore `gdb` features in more depth, but today we'll cover just a few basics so that you can start using `gdb` to help you debug your C lab assignments.

To use the debugger, you usually want to compile your C program with the `-g` flag to add debugging information to the `a.out` file (this allows `gdb` to map binary machine code to C program code that the programmer understands).

```
$ gcc -g -o testprog testprog.c
```

The Makefile already has this rule for us, so let's just run `make`.

3.1. Running GDB

Next, we will run the executable file inside the GDB debugger:

```
$ gdb ./testprog
```

The first thing we get is the GDB prompt. At this point `testprog` has not yet started running.

3.2. Example GDB session

We usually begin a debugging session by setting a break point at `main` before starting the program running in GDB. A breakpoint tells GDB to grab control at a certain point in the execution, in this case right before the first instruction in main is executed:

```
(gdb) break main
Breakpoint 1 at 0x1155: file testprog.c, line 20.
```

Next, we will enter the run command at the GDB prompt to tell GDB to start running our program:

```
(gdb) run
Starting program: /home/degoodj/Lab2/testprog
```

```
Breakpoint 1, main () at testprog.c:20
20      x = 10;
```

The run command will start your program running, and GDB will only gain control again when a breakpoint is hit.

There are a few other primary GDB commands we will learn today. The first is the `list` command that displays the C source code around the point where we are in the execution:

```
(gdb) list
```

`list` with a line number lists the source code around that line:

```
(gdb) list 30
```

The `next` command (or just `n` as a shortcut) tells GDB to execute the next instruction and then grab control again:

```
(gdb) next    # execute the x = 10 line we stopped on when entering main
21      y = 8;
(gdb) next    # execute y = 8 and display the next line to run
22      z = y / x;
```

The `print` command can be used to print out the value of a program variable or expression:

```
(gdb) print x
$1 = 10
```

`cont` tells GDB to let the program continue running. Since we have no more breakpoints, it will run until termination.

```
(gdb) cont
Continuing.
x = 10 y = 8 z = 0.00
x = 10 y = 8 z = 53.00
[Inferior 1 (process NUM) exited normally]
(gdb)
```

Now let's add a breakpoint in the function `mystery`, and rerun the program:

```
(gdb) break mystery
Breakpoint 2 at 0x555555555206: file testprog.c, line 36.
```

The `run` command starts the program's execution over from the beginning. When re-run, the breakpoint at the beginning of the `main` function will be hit first (and `list` displays the code around the breakpoint).

```
(gdb) run
Starting program: /home/degoodj/Lab2/testprog

Breakpoint 1, main () at testprog.c:20
20      x = 10;
```

```
(gdb) list 25
```

Let's set a breakpoint at line 25, right before the call to mystery. Next, type `cont` to continue execution from breakpoint in main:

```
(gdb) break 25
Breakpoint 3 at 0x5555555551a6: file testprog.c, line 25.
(gdb) cont
```

The program continues running until it reaches the breakpoint we just set at line 25 (Breakpoint 3). We can examine the program's execution state at line 25 by printing out the argument values before the call to `mystery` (using `print`), and then type `cont` to continue the program's execution:

```
Continuing.

x = 10 y = 8 z = 0.00

Breakpoint 3, main () at testprog.c:25
25      z = mystery(x, y);

(gdb) print x
(gdb) print y
(gdb) cont
```

After continuing, the breakpoint in `mystery` is hit next (Breakpoint 2), let's step through some of the `mystery` function's execution, and print out some of its parameters and locals.

We can use the `print` command to print out expressions in the program, so let's print out the values of the arguments passed to mystery, and type cont to run until the next break point is hit:

```
(gdb) print a        # print out the value of the variable a
(gdb) print (a - 4)  # print out the value of the expression (a - 4)
(gdb) list
```

The `where` or `bt` command list the call stack:

```
(gdb) where
```

When you're done using `gdb`, type the command `quit`.

```
(gdb) quit
$
```

3.3. GDB and command line arguments

If you use GDB to help you debug a program that expect command line arguments, you'll need to pass the program's arguments to the `run` command:

```
$ gdb ./testfile
(gdb) break main
(gdb) run values1
```

In general, for programs with command line arguments, simply list the arguments after the `run` command, for example to run with 3 command line arguments (6, 4, and hello), do the following:

```
(gdb) break main
(gdb) run 6 4 hello
```

We'll learn more about C and GDB over the course of the semester, but these GDB basics are enough to start using GDB to help you debug your C programs.

4. Lab 2 Intro

Lets talk through the next Lab 2 Assignment, where you will implement a C program that, among other things, uses arrays, command line arguments, and reads values in from a file.

5. Handy Resources

- Chapter 1 and Chapter 2 on C programming in course textbook
- Chapter 3 on gdb