The Command Line

Matthew Bender

CMSC Command Line Workshop

Octover 30

Section 1

Development from the Command Line

gcc

GCC (The **G**NU **C**ompiler **C**ollection, formerly the **G**NU **C** **C**ompiler) was released in 1987, and could originally only compile C, although today it can compile languages such as C++, Fortran, Go, and others.

`$ gcc [.c source files]` will compile the given files, and produce an executable names `a.out`
`$ gcc [.c source files] -o progname` will produce an executable names `progname`
`$ gcc -S prog.c` will produce an assembly file `prog.s`

Useful `gcc` flags

`-Wall -Wextra -Werror` are 3 flags that should always be added to `gcc` (create an alias!) - they will warn you about a lot of things that could go wrong with your code when you compile.
`-g` : turn on debugging info. When you are debugging with `gdb`, you want to compile with this flag.
`-O, -O2, -O3` - turn on size optimizations (each is a higher level than the last)
`-Os` - optimize for executable space

Examining Output

Don't inspect your code output by hand! You can check it against the expected output from the command line.
The `cmp` command will compare 2 files quickly. However, it doesn't produce useful output.
The `diff` command compares files and points out the lines and characters that differ:

```
$ diff expected_output.txt actual_output.txt

< Average:  87.6
---
> Average:  87.600000
```

The < means that line was from the first file, and the > means that line was from the second. The student forgot to truncate the float when printing!

Section 2

Debugging

Debugging with `gdb`

`gdb` (The **G**NU **D**e**b**ugger) is one of the most useful tools for developing. To use it, add the `-g` flag when compiling:

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

Then start the debugger:

```
$ gdb prog
```

You will see a prompt:

```
(gdb)
```

This is `gdb` waiting for you to enter commands

Debugging with `gdb`

There are two ways to run the program - all at once or step by step.

To go step by step, enter `(gdb) start arg1 arg2 argn` - where each `arg` is an argument to your program

`gdb` will print each line of code for you as you step through.
To step to the next line, enter `(gdb) next` (or `n`) for short.

To step to the next line, or into a function, enter `(gdb) step` (or `s`) for short.

To run all at once, replace `start` with `run` above.
This will run your code until it completes or runs into an error, like a DB0 or segfault, or it encouters a breakpoint.

Breakpoints

Breakpoints are ways to stop your code running when it reaches a certain line.
Create one with `(gdb) break <linenum>` or `(gdb) break <funcname>`
To continue running after hitting a breakpoint, enter `(gdb) continue` (or `c` for short)
Or, use `next` or `step` if you want to step through the program instead of run it.

You can also set conditional breakpoints, which only pause if a certain condition is met (like if `x > 5` or `strcmp(s, "Hello") == 0`).
Another useful thing is to set a **watchpoint** on a variable, which pauses if it is read or wrote to.

Printing Information

Enter `(gdb) print x` to view the value of the `x` variable - the variable must be in scope.
You can print full C expressions in the debugger as well:

```
(gdb) print strlen(str)
(gdb) print (x << 3) * y
```

You can view all local variables with `(gdb) info locals`
You can view all arguments to the current function with `(gdb) info args`

The stack

If you are unsure where you are, run `(gdb) frame` to view the current line.
Even more helpful is `(gdb) backtrace` (or just `bt`) to view the whole stack trace.
An easy way to debug segfaults is to run the program until the fault happens, then run `(gdb) backtrace` to see what function calls with what values got you to the fault and what line it happened on.

If you want to exit the current function, run `(gdb) up` to go up a stack frame. Run `(gdb) down` to go back down to where you were.

Valgrind

The other most important tool you have for debugging is Valgrind.
Valgrind is more useful for catching memory errors.
Some examples of the errors it will catch:

- Catching memory leaks (when you don't free a block after you malloc it)
- Reading or writing to memory after is has been free'd
- Reading or writing beyond allocated blocks (like arrays or malloc'ed blocks)
- Freeing memory that was not malloc'ed
- Reading from uninitialized variables

You should also compile with `-g` to get better output from Valgrind.