# Shell Commands and Makefiles

## EECS 348 Lab 3 — 2/13/2025

Harlan Williams

Electrical Engineering and Computer Science
University of Kansas

# Useful shell commands

In the filename argument, `*` can be used as a wildcard to match files of that form, *i.e.*, `*.c` means any file ending in a `.c`

- `ls`—Prints all files in current directory
  - `ls -a` to print all files, including hidden ones (ones that begin with a '.')
  - `ls -l` to print all files, their permissions, timestamps, and size
- `cd <dir>`—Change directory
- `mkdir <dir>`—Create a new folder
- `cp <src> <dest>`—Copies the file `<src>` to `<dest>`
- `mv <src> <dest>`—Moves a file to `<dest>`. If the destination in is the same directory this is equivalent to renaming.
- `rm <file>`—*Permanently* deletes a file
  - `rm -r <dir>`—*Permanently* deletes an entire directory

# Python

Prints “fizz” if a number is divisible by 3, “buzz” if a number is divisible by 5, and “fizzbuzz” if divisible by both

```
def fizzbuzz(n):
    for i in range(1, n+1):
        print(f"{i}:", end="")

        if i % 15 == 0:
            print("fizzbuzz", end="")
        elif i % 3 == 0:
            print("fizz", end="")
        elif i % 5 == 0:
            print("buzz", end="")

        print()
```

# C

Prints “fizz” if a number is divisible by 3, “buzz” if a number is divisible by 5, and “fizzbuzz” if divisible by both

```
void fizzbuzz(int n) {
    for (int i = 1; i <= n; i++) {
        printf("%d:", i);
        if (i % 15 == 0) {
            printf("fizzbuzz");
        } else if (i % 3 == 0) {
            printf("fizz");
        } else if (i % 5 == 0) {
            printf("buzz");
        }
        printf("\n");
    }
}
```

# Key differences

- Blocks in C are defined by brackets {} rather than by indentation
- Statements must end with a semicolon ;
- Variables and functions must have a type (`int`, `void`)

# Compiling C programs

C programs have to be compiled before they can be ran—unlike Python programs which are just interpreted.

A compiler is just another program. `gcc` is what is on the lab computers and is the default on most Linux distributions.

Example usage: `gcc main.c` will by default create an executable named `a.out` from the code in `main.c`. This can then be run by typing `./a.out` in the terminal.

# Makefiles

Code in larger C programs is often split across multiple headers and source code files, making building the final artifact more complicated

Makefiles are a way to automate the build process in a reproducible way.

A makefile is just a file named ‘Makefile’ which contains variable definitions and rules for what commands to run to build each object.

To run a makefile, run `make` on the terminal, or optionally `make <rule>` to run a specific rule

# Makefile structure

Each rule defines a sequence of commands to run in order to complete a task (building, cleaning, *etc.*) A variable can be substituted into a command with the syntax `$(VAR)`. There are also implicit variables that refer to the rule name (`$@`) or the prerequisites (`$^`)

```
VAR := definition

rule: prerequisites
    command1
    command2

rule2: prerequisites
    command1
```

# Makefile example

```
CC := gcc
CFLAGS := -Wall
OBJS := main.o file1.o file2.o

program: $(OBJS)
    $(CC) $(OBJS) -o $@     # '$@' stands for the
                            # rule name
%.o: %.c
    $(CC) $(CFLAGS) -c $< # '$<' stands for the
                          # first prerequisite
```

The % is a wildcard that matches files with a certain name, *e.g.*, `%.c` matches files ending in a `.c`.

# Lost? See...

- `man <cmd>` for help about a specific command
- www.gnu.org/software/make/manual/html_node/index.html
- www.makefiletutorial.com/
- There are many C tutorials out there for help with learning the language