CMPSC 60: Week 4 Discussion

What is compiling

- gcc steps
  - preprocessing – you know what this does
  - compilation – produces assembly code
  - assembly – creates an object file (.o)
  - linking – links .o files to create executable

gcc non-optional options

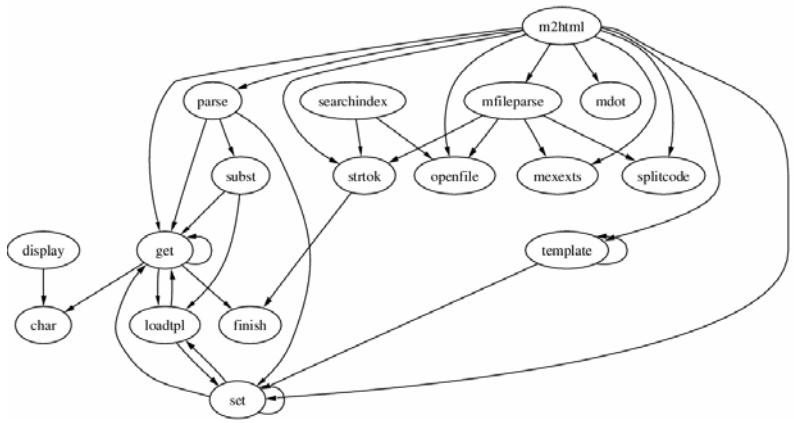
- -c – compile but don’t link
- -o – executable file name
- -l<library> – link with library
- -L<dir> – non-standard directory for libraries

gcc optional options

- -Wall – print lots of warnings
- -W – print fewer (but sometimes different) warnings
- -pedantic – strict ANSI C standard
- -g – generate symbol table for debugging
- -O<1-3> - compile with optimizations

Why we need Makefiles

- Source files can be dependent on each other in very complicated ways.
- Saves time of having to remember what to recompile, or recompile everything



Makefile syntax

- Target – the name of a task – can be compiling a file, or a number of other things
- Dependency – a target will often have dependencies – other targets that need to be done before it
- Up to date – a file is more recent than all the files it depends on

Make tips

- Goto cd ~/jwither/cs60/week4 for makefile examples
- In a given target like:

```
Build:
    gcc -o exe input.c helper.c
```

The space before the command (gcc) is a single tab character – spaces don't cut it

Dependencies

```
stimulate: stimulate.o inputs.o ouputs.o
    gcc -o stimulate stimulate.o inputs.o ouputs.o
```

The things after the : on the first line are the dependencies. In this case to build the program stimulate we need to have an up to date stimulate.o, inputs.o, outputs.o Otherwise those targets need to be executed first. Dependencies can also be files, in which case it just checks to see what's up to date i.e.

```
stimulate.o: stimulate.c
    gcc -c stimulate.c
```

Abbreviations

- $@ - full target name
- $* - target name without suffix

```
stimulate: stimulate.o inputs.o ouputs.o
    gcc -o stimulate stimulate.o inputs.o ouputs.o
```

Becomes

```
stimulate: $*.o inputs.o ouputs.o
    gcc -o $@ $*.o inputs.o ouputs.o
```

Macros

- MACRONAME = macro
- With the rule:

```
OBJECTS = stimulate.o inputs.o outputs.o
stimulate: stimulate.o inputs.o ouputs.o
    gcc -o stimulate stimulate.o inputs.o ouputs.o
```

Becomes

```
stimulate: $(OBJECTS)
    gcc -o $@ $(OBJECTS)
```

Suffix rules

- Rules to make commands shorter and more general – can be easily reused
- Read pages 175-176 of Gnu Toolkit book

How to run make

- Type make
  - Executes first target defined – so make the first rule the rule to build your executable project
- Other targets can be executed as well by typing: make <target>

Other handy make targets

Target to build your executable – make sure to have this as the first rule

```
all: <whatever you need>
```

Target to get rid of all old files – can be useful to make sure everything is up to date

```
clean:
   rm -f *.o *~ #*# core*
```

Make a brand new executable

```
fresh: clean all
```

Makedepend

- Handy program that can be used to figure out dependencies for you

```
depend:
    makedepend -- $(INCLUDES) -- $(SOURCES)
```

Look at example makefiles for more

Write your own Makefile

- Compile options: -g –Wall –pedantic –W
- Needed libraries: -lm
- Main file main.c and helper file that final executable depend on (helper.c)

Good solution

```
CC        =     /usr/bin/gcc
CFLAGS    =      -g -Wall -pedantic -W
LIBS      =      -lm

APP       = basic
OBJS      = main.o helper.o

all:    $(APP)
$(APP): $(OBJS)
        $(CC) $(CFLAGS) -o $(APP) $(OBJS) $(LIBS)
main.o: $*.c
         gcc $(CFLAGS) -c $*.c
helper.o: $*.c
         gcc $(CFLAGS) -c $*.c
clean:
        rm -f *.o *~* $(APP)
```

Also works (because of default suffix rules)

```
CC        =     /usr/bin/gcc
CFLAGS    =      -g -Wall -pedantic -W
LIBS      =      -lm

APP       = basic
OBJS      = main.o helper.o

all:    $(APP)
$(APP): $(OBJS)
        $(CC) $(CFLAGS) -o $@ $(OBJS) $(LIBS)
main.o:
helper.o:
clean:
        rm -f *.o *~* $(APP)
```