Make!

CSC230: C and Software Tools

N.C. State Department of Computer Science

Some examples adapted from http://mrbook.org/tutorials/make/

Computer Science
NC STATE UNIVERSITY

How have we been compiling?

- Using the compiler directly with one source file:

```
gcc -Wall -std=c99 -o coolapp coolapp.c
```

- Problems:
  - All that typing
  - What if your app has more than one source file? (covered later)
  - If you need libraries, optimization, etc….more typing.
  - *Danger!!*

```
gcc -o coolapp.c coolapp.c
```
→ *Destroys your code!*

A solution descends!

What ‘make’ does

- When you run “make”, it looks for a file called “Makefile”, and executes it
- Can use an alternate Makefile:

```
make -f Makefile2
```

- Makefiles tell ‘make’ how to build your app

What's in a Makefile?

- The basic Makefile is composed of:

  target: *dependencies*
  [tab] system-command

- A dumb Makefile of our earlier example:

```
all:
	gcc -Wall -std=c99 -o coolapp coolapp.c
```

```
$ make
gcc -Wall -std=c99 -o coolapp coolapp.c
```

- Note:
  - Make runs the first target by default ("all" here)
  - There were no dependencies, so it just runs the command

Three things that make ‘make’ cool

- Thing 1: Make only makes if it has to
- Thing 2: Variables add flexibility
- Thing 3: Program compilation can be broken up
  (To be covered later)
- Bonus thing: Makefiles are **required** to get full credit in your homework!

Thing 1: Make only makes if it has to

- Better example:

```
coolapp: coolapp.c
	gcc -Wall -std=c99 -o coolapp coolapp.c
```

```
$ make
gcc -Wall -std=c99 -o coolapp coolapp.c
$ make
make: `coolapp' is up to date.
```

- It knew that coolapp didn't need to be recompiled, because coolapp.c didn't change!
  - If the timestamp on the target is newer than all the dependencies, then skip this command
- Saves work, saves time!
  - Large builds can take HOURS!!!

Thing 2: Variables add flexibility

- Simple Makefile template*:

```
# Makefile for coolapp by Tyler Bletsch
CC=gcc
FLAGS=-Wall -std=c99
SRC=coolapp.c
EXE=coolapp


all: $(EXE)

clean:
	rm $(EXE)

$(EXE): $(SRC)
	$(CC) $(FLAGS) -o $@ $^
```

Variable CC represents our compiler

Compiler flags

List of source files (just one here)

The executable we're building

Default rule: make the executable

Optional but commonly used target: used to delete the build when "make clean" is typed.

To make the EXE, compile all these SRC files with this CC compiler using these FLAGS.

"The target"

"The dependencies"

* You'll outgrow this one when we get to multi-file apps.

Thing 2: Variables add flexibility

- Simple Makefile template*:

```
# Makefile for coolapp by Tyler Bletsch
CC=gcc
FLAGS=-Wall -std=c99
SRC=coolapp.c
EXE=coolapp


all: $(EXE)

clean:
	rm $(EXE)

$(EXE): $(SRC)
	$(CC) $(FLAGS) -o $@ $<
```

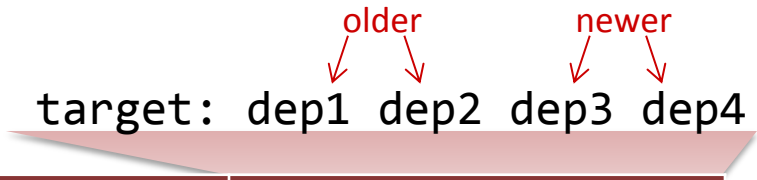
```
$ make
gcc -Wall -std=c99 -o coolapp coolapp.c
$ make
make: Nothing to be done for `all'.
$ make clean
rm coolapp
$ make
gcc -Wall -std=c99 -o coolapp coolapp.c
$
```

* You'll outgrow this one when we get to multi-file apps.

Thing 3: Breaking up compilation

- We'll get to multi-file programs later. If you're curious, the content is at the end of this deck.

Automatic variables



Var	Meaning	Example value
$@	The file name of the *target* of the rule.	`target`
$<	The name of the *first prerequisite*.	`dep1`
$?	The names of *all the prerequisites that are newer* than the target, with spaces between them.	`dep3 dep4`
$^	The names of *all the prerequisites*, with spaces between them.	`dep1 dep2 dep3 dep4`

Suffix rules

- It's common to convert one file type to another.
  - "Convert" can mean "compile"…
- Example:

```
# Compile any requested .c file to human-readable assembly code (.s file)
.c.s:
	gcc -S $< -o $@

# Compile any requested .c file to "object code" (compiled but not linked, .o file)
.c.o:
	gcc -c $< -o $@
```

```
$ make x.s
gcc -S x.c -o x.s
$ make x.o
gcc -c x.c -o x.o
```

Make is for more than just C

- I use make to prepare the PDFs of these slides!

```
SRCS=$(wildcard *.pptx)

PDFS=$(SRCS:.pptx=.pdf)

all: $(PDFS)

.pptx.pdf:
	cscript ppt2pdf.vbs $<

.SUFFIXES : .pptx .pdf
```

Advanced command that expands wildcards (also considered bad practice for C programs). Don't use for your homework.

Turn "x.pptx y.pptx …" to "x.pdf y.pdf …"

Build all the PDFs for these PPTX's

A recipe to convert a .pptx file to .pdf using an incredibly ugly Vbscript I found

Tell make that "pptx" and "pdf" are extensions it can handle with an extension-based rule like the above.

JUST TELL ME WHAT I NEED TO DO TO GET CREDIT ON THE HOMEWORK

- Take this Makefile template and replace the stuff in red. Build your app by typing “make”.

```
# Makefile for PROJECT-NAME by AUTHOR
CC=gcc
FLAGS=-Wall -std=c99
SRC=MY-SOURCE-FILE.c
EXE=MY-EXECUTABLE


all: $(EXE)

clean:
	rm $(EXE)

$(EXE): $(SRC)
	$(CC) $(FLAGS) -o $@ $^
```

WHAT IF THE HOMEWORK HAS MULTIPLE EXECUTABLES?

- Then do this:

```
# Makefile for PROJECT-NAME by AUTHOR
CC=gcc
FLAGS=-Wall -std=c99
SRC1=MY-SOURCE-FILE1.c
EXE1=MY-EXECUTABLE1
SRC2=MY-SOURCE-FILE2.c
EXE2=MY-EXECUTABLE2
SRC3=MY-SOURCE-FILE3.c
EXE3=MY-EXECUTABLE3
# add/delete SRC/EXE pairs as needed


all: $(EXE1) $(EXE2) $(EXE3)

clean:
          rm $(EXE1) $(EXE2) $(EXE3)

$(EXE1): $(SRC1)
          $(CC) $(FLAGS) -o $@ $^

$(EXE2): $(SRC2)
          $(CC) $(FLAGS) -o $@ $^

$(EXE3): $(SRC3)
          $(CC) $(FLAGS) -o $@ $^

# add/delete rules as needed
```

Exercise 07a

Makefiles

- Write a hello-world program “hello.c”
- Write a Makefile for it
- Build it by typing “make”
- Submit *just the Makefile*.
  - Set the code type in IDEOne to “Text”.

Reminder: Go to course web page for link to exercise form.
Paste code into ideone.com and submit the link.

BACKUP

Thing 3: Breaking up compilation

- More advanced Makefile template:

```
# Advanced Makefile for coolapp by Tyler Bletsch
CC=gcc
CFLAGS=-c -Wall -std=c99
LDFLAGS=
SRC=main.c support.c
OBJ=$(SRC:.c=.o)
EXE=coolapp



all: $(EXE)

clean:
	rm -f $(OBJ) $(EXE)

$(EXE): $(OBJ)
	$(CC) $(LDFLAGS) $(OBJ) -o $@

.cpp.o:
	$(CC) $(CFLAGS) $< -o $@
```

```
$ make clean
rm -f main.o support.o coolapp
$ make
gcc -c -Wall -std=c99   -c -o main.o main.c
gcc -c -Wall -std=c99   -c -o support.o support.c
gcc  main.o support.o -o coolapp
$ make
make: Nothing to be done for `all'.
$ vim support.c
$ make
gcc -c -Wall -std=c99   -c -o support.o support.c
gcc  main.o support.o -o coolapp
```