

# Make

Beginner to Hardcore in 50 Minutes

Jeff Shantz

x@y

x = jshantz4, y = csd.uwo.ca

October 21, 2009

# Outline

- 1 Motivation
  - Why we need makefiles
- 2 Make
  - Beginning Make
  - Intermediate Make
  - Advanced Make
- 3 Questions

# Compiling Large Programs

- As your programs grow in size and complexity, so too will the number of files you are required to manage
- Compiling a program manually each time a change is made to one or more files can be very tedious
- Fortunately, we have programs like `make` to do the work for us

But first... an example.

## convertc.c

```
1 #include <stdio.h>
2 #include "inputHelper.h"
3 #include "temperatureDefs.h"
4
5 int main()
6 {
7     int tempC; /* Temperature in Celsius */
8     int tempF; /* Temperature in Fahrenheit */
9
10    /* Get the Celsius value from the user */
11    tempC = promptInteger("Temperature (in C): ",
12                          "Please enter a value between 1 and 100",
13                          1,
14                          100);
15
16    /* Convert it to Fahrenheit */
17    tempF = (FAHRENHEIT_COEFFICIENT * tempC) + FAHRENHEIT_MODIFIER;
18
19
20    /* Report the conversion */
21    printf("%d C is approximately equal to %d F\n", tempC, tempF);
22 }
```

## convertf.c

```
1 #include <stdio.h>
2 #include "inputHelper.h"
3 #include "temperatureDefs.h"
4
5 int main()
6 {
7     int tempF; /* Temperature in Fahrenheit */
8     int tempC; /* Temperature in Celsius */
9
10    /* Get the Fahrenheit value from the user */
11    tempF = promptInteger("Temperature (in F): ",
12                          "Please enter a value between 1 and 100",
13                          1,
14                          100);
15
16    /* Convert it to Celsius */
17    tempC = CELSIUS_COEFFICIENT * (tempF + CELSIUS_MODIFIER);
18
19
20    /* Report the conversion */
21    printf("%d F is approximately equal to %d C\n", tempF, tempC);
22 }
```

# Compiling convertc and convertf manually

```
1 gcc -c inputHelper.c
2 gcc -o convertc convertc.c inputHelper.o
3 gcc -o convertf convertf.c inputHelper.o
```

## Dependencies:

`inputHelper.c`     `inputHelper.h`, `boolean.h`

`convertc.c`        `inputHelper.h`, `temperatureDefs.h`

`convertf.c`        `inputHelper.h`, `temperatureDefs.h`

- What if we need to change `inputHelper.h`?
- Need to keep track of all files that depend on it

# Enter make

## make

- A utility for building programs/libraries from source code
- Keeps track of changes made to source files
- Only recompiles those files that have been changed

## makefile

- Text file containing rules to build all program components
- Rules consist of targets, dependencies, and command(s)
- The first rule in a makefile is the *default* rule

# Rules

## Rule Syntax

```
target: dependency1 dependency2 ... dependencyn  
      → commands  
      → ⋮
```

- **Important:**

- → is the [Tab] character. Use tabs and not spaces!
- A blank line is required after the last command in the rule
- You will get strange errors if you do not follow these rules

# Makefile 1

## Makefile 1

```
all:          convertc convertf

convertc:     convertc.o inputHelper.o
              gcc -o convertc convertc.o inputHelper.o

convertc.o:   convertc.c inputHelper.h temperatureDefs.h
              gcc -c convertc.c

convertf:     convertf.o inputHelper.o
              gcc -o convertf convertf.o inputHelper.o

convertf.o:   convertf.c inputHelper.h temperatureDefs.h
              gcc -c convertf.c

inputHelper.o: inputHelper.c inputHelper.h boolean.h
              gcc -c inputHelper.c
```

# Running Make

## Running the default rule

```
$ make
```

## Running a specific rule

```
$ make inputHelper.o
```

## Specifying the name of the makefile to run

```
$ make -f MyMakeFile
```

- You should name your file `makefile` or `Makefile`
- Otherwise, you will need to use the `-f` option
- Everyone uses `Makefile`, though, so you should too

# Makefile Conventions

- The first rule in a makefile should generally be named `all`
  - This should build all programs/libraries to be built
- Each makefile should contain a rule named `clean`
  - This should remove intermediate files created by compilation
- A `realclean` target is also nice
  - This should call `clean` and then also remove all executable files

## Makefile Conventions

```
all:          convertc convertf

clean:
    -rm *.o

realclean: clean
    -rm convertc convertf
```

# Error Handling

- By default, if a command fails, `make` will halt
- You can instruct `make` to continue after errors occur by preceding the command with a hyphen
- This was seen in `clean` and `realclean` in the last example:

## Error Handling

```
clean:
```

```
    -rm *.o
```

```
realclean: clean
```

```
    -rm convertc convertf
```

- Here, `*.o`, `convertc`, and `convertf` might not exist
- We don't want `make` to stop running because of this

# Comments

- A comment in a makefile is any line that begins with a # character
- A header comment at the top should describe the makefile, its major targets, etc.
- Comments should also be used to describe each rule

## Comments

```
#####  
# Makefile  
# Author: Jeff Shantz  
#  
# To build, run 'make all'  
# To remove object files, run 'make clean'  
#####
```

# Forcing Recompilation

- Sometimes we want to force the recompilation of one or more files
- `make` only recompiles files that have been changed since the last compilation
- To fool `make` into recompiling a file, use the `touch` command

## Forcing Recompilation

```
$ touch convert.c
```

```
$ make convertc
```

# Disabling Command Echoing

- By default, `make` echoes each command to the standard output before executing it
- We can override this by preceding the command with `@`

## Disabling Command Echoing

```
convertc:    convertc.o inputHelper.o
              @echo Building convertc...
              @echo =====
              gcc -o $@ convertc.o inputHelper.o
```

# Macros

- We can use macros (similar to variables) to shorten our makefiles, and reduce duplication and typos

## Defining Macros

```
CC = gcc  
CFLAGS = -c
```

## Using Macros

```
$(CC) $(CFLAGS) main.c
```

## Equivalent Command

```
gcc -c main.c
```

# Using Macros

## The Old Way

```
convertc.o: convertc.c inputHelper.h temperatureDefs.h  
    gcc -c convertc.c
```

```
convertf.o: convertf.c inputHelper.h temperatureDefs.h  
    gcc -c convertf.c
```

## The New Way

```
CC = gcc
```

```
CFLAGS = -c
```

```
HEADERS = inputHelper.h temperatureDefs.h
```

```
convertc.o: convertc.c $(HEADERS)  
    $(CC) $(CFLAGS) convertc.c
```

```
convertf.o: convertf.c $(HEADERS)  
    $(CC) $(CFLAGS) convertf.c
```

# Makefile 2

## Makefile 2

```
CC = gcc
CFLAGS = -c -Wall
LFLAGS = -o
HEADERS = inputHelper.h temperatureDefs.h
CONVERTC_OBJECTS = convertc.o inputHelper.o
CONVERTF_OBJECTS = convertf.o inputHelper.o

all:          convertc convertf

convertc:    $(CONVERTC_OBJECTS)
             $(CC) $(LFLAGS) convertc $(CONVERTC_OBJECTS)

convertc.o:  convertc.c $(HEADERS)
             $(CC) $(CFLAGS) convertc.c
```

# Makefile 2 continued...

## Makefile 2 continued...

```
convertf: $(CONVERTF_OBJECTS)
           $(CC) $(LFLAGS) convertf $(CONVERTF_OBJECTS)

convertf.o: convertf.c $(HEADERS)
            $(CC) $(CFLAGS) convertf.c

inputHelper.o: inputHelper.c inputHelper.h boolean.h
              $(CC) $(CFLAGS) inputHelper.c

clean:
       -rm *.o

realclean: clean
          -rm convertc convertf
```

# Special Macros

`$$` - The name of the current target

```
convertc:    convertc.o inputHelper.o
             gcc -o $$ convertc.o inputHelper.o
```

`$$?` - List of out-of-date dependencies for the current target

```
helloworld.o:  hello.c inputHelper.c
              gcc -c $$? -o $$
```

`$$^` - List of all dependencies for the current target

```
convertc:    convertc.o inputHelper.o
             gcc -o $$ $$^
```

# Special Macros continued...

`$<` -The first dependency of the current target

```
convertc.o: convertc.c inputHelper.h temperatureDefs.h  
gcc -c $<
```

# Inference Rules

- We can generalize our makefiles and drastically reduce their size using inference rules
- It is best to see an example to understand inference rules:

## Inference rule to compile all C files to object files

```
%.o: %.c
```

```
gcc -c $< -o $@
```

This rule says:

*“For each C file `file.c` in the current directory, compile it to an object file by executing `gcc -c file.c -o file.o`”*

# Makefile 3

## Makefile 3

```
CC = gcc
CFLAGS = -c -Wall
LFLAGS = -o
HEADERS = inputHelper.h temperatureDefs.h boolean.h
CONVERTC_OBJECTS = convertc.o inputHelper.o
CONVERTF_OBJECTS = convertf.o inputHelper.o

all:      convertc convertf

convertc: $(CONVERTC_OBJECTS)
          $(CC) $(LFLAGS) $@ $^

convertf: $(CONVERTF_OBJECTS)
          $(CC) $(LFLAGS) $@ $^
```

# Makefile 3 continued...

## Makefile 3 continued...

```
%.o: %.c $(HEADERS)
    $(CC) $(CFLAGS) $< -o $@

clean:
    -rm *.o

realclean: clean
    -rm convertc convertf
```

What are the two problems with this makefile?

**Questions?**