CMSC 216
Introduction to Computer Systems
Lecture 13
Make, cont. and C Preprocessor

Larry Herman & Alan Sussman

Notes

- Project 2 style grading still not done
- Project 3 due today
  - questions?
- Project 4 posted soon
  - will extend Project 3, so get that one working!
- Exam #1
  - Mean: 81 Median: 83 25%: 74 75%: 91
  - solutions posted soon
  - requests for regrades only after checking against solutions, and by Tuesday after spring break
  - questions?
- Read Reek, Ch. 14: Preprocessor

(Not in the textbooks)

MAKE

Make's operation

- Make constructs a dependency tree based on the makefile
- A target is out of date when any one of its (direct or indirect) dependencies is newer than itself
- If a target is out of date, make recreates it by performing the action specified in that target's action
  - this process is performed bottom to top in the dependency tree
- Make ensures minimum compilation, if your makefile and dependencies are correct
- Will the following rule make **`publicX`**?

```
publicX: publicX.c chess.c chess.h
        gcc -o publicX publicX.c chess.c
```

- Yes, but is it desirable? We rebuild the entire program no matter which source file is changed with this rule.

Makefile macros

- A macro is a makefile variable
- A macro is defined by: *variable* = *value*

```
CC = gcc
CFLAGS = -ansi -pedantic-errors -Wall -Werror

file.o: file.c file.h
        $(CC) $(CFLAGS) -c file.c
```

  - Last rule has the effect as if it were written:

```
file.o: file.c file.h
        gcc -ansi -pedantic-errors -Wall -Werror -c file.c
```

- Note that the above options for **`CFLAGS`** only affect the way that code is compiled, not linking (so they should be omitted for rules that just do linking)

Additional features of make

- Implicit rules
  - Make understands conventions
    - **`xxx.o`** is built from **`xxx.c`** using the action "**`$(CC) $(CFLAGS) -c xxx.c`**"
  - Has lots of rules it knows about
    - With a simple, single-file program (e.g., **`helloworld.c`**), you can build your program with a command like this:

```
make helloworld
```

  - These are useful, but there's a lot to know about using them correctly

More make features

- Actions can be any shell commands, even multiple commands, not just compilation/ linking

```
clean:
        @echo "Removing object, emacs backup, and core files."
        rm -f *.o *~ core core.*
```

- Preceding an action line with an at sign (`@`) keeps make from echoing the command when make is run
- A long line in a makefile can be continued to the next line by ending it with a backslash

A more complex makefile

```
# comments start with a pound sign and last until the end of the line

CC = gcc
CFLAGS = -ansi -pedantic-errors -Wall -Werror
PROGS = int_count table

all: $(PROGS)

int_count: int_count.o
        $(CC) -o int_count int_count.o

table: table.o main.o hash_table.o
        $(CC) -o table table.o main.o hash_table.o

table.o: table.c table.h
hash_table.o: hashtable.c table.h
main.o: main.c table.h
int_count.o: int_count.c table.h

clean:
        rm -f *.o $(PROGS)
```

macro definitions

builds the two top-level programs

note no `$(CFLAGS)` in these

use implicit rules

not part of the main tree - built by typing "make clean"

More about header files

- Note that header files should contain only declarations, with no executable code
- Header files **are not directly compiled** (for example, there were no rules to compile any `.h` files in the previous example); they just get included in source files, which are compiled
- Header files should contain declarations that the compiler needs to see to be able to compile executable code
  - examples include function prototypes, typedefs, and `extern` global variables

Chapter 14, Reek

THE PREPROCESSOR

Compiling a program

- Source files (`*.c`) compiled into object files (`*.o`)
  1. Source file is first preprocessed
  2. Preprocessed file is then compiled to assembly
     1. scanner/parser
     2. type checker
     3. code generator
     4. optimizer
  3. Assembler converts assembly code to object code
- Object files are converted into an executable
  - Done by the linker, which resolves symbols
    - match function use to its definition
    - global variable declaration and external use

Preprocessor predefined symbols

- `__FILE__` : a string, the filename in which the symbol is found
- `__LINE__` : an integer, the line on which the symbol is found
- `__DATE__` : a string, date of compilation
- `__TIME__` : a string, time of compilation
- `__STDC__` : 1 or undefined, whether or not compiler is ANSI-compliant
- Note that each is preceded and followed by two underscores
- Examples of their values are available in table 14.1 (pg. 383) of Reek

The `#define` directive

- Syntax: **`#define`** *name lots of text*
- Substitutes *lots of text* for *name* in the rest of the source file
- Most commonly used for constants
- By convention, we use all capital letters for our constants - makes it easier for humans to recognize them

Macros

- **`#define`** can also be used to define macros, where parameters are substituted as well as straight text substitution
- Syntax: **`#define`** *name(parameter-list) text*
- Example:

```
#define SUM(a, b) a + b
...
x = SUM(2, 6);
/* becomes "x = 2 + 6;" */
```


Macros, cont.

- We can also use a backslash at the end of a line to extend a macro onto a second line:

```
#define MIN(a, b) a < b \
  ? a : b
```


Typical uses for macros

- **`#define SQUARE(x) ((x) * (x))`**

```
s = SQUARE(5);
```

becomes

```
s = ((5) * (5));
```

```
s = SQUARE(a + 1);
```

becomes

```
s = ((a + 1) * (a + 1));
```

Typical uses for macros

- ```
  #define MAX(a, b) \
      ((a) > (b) ? (a) : (b))
  ```

```
x = 4;
y = 9;
z = MAX(x, y);
```

becomes

```
z = ((x) > (y) ? (x) : (y));
```


Why we use all those parentheses

- Remember, substitution is essentially just find-and-replace, like in a text editor:

```
#define BAD_SQUARE1(x)  x * x
#define BAD_SQUARE2(x) (x * x)
#define BAD_SUM(x, y) (x) + (y)

s = BAD_SQUARE1(a + 1);
/* s = a + 1 * a + 1; */
s = BAD_SQUARE2(a + 1);
/* s = (a + 1 * a + 1); */
s = BAD_SUM(a + 1, b + 2) * 3;
/* s = (a + 1) + (b + 2) * 3; */
```

- This is why we use parentheses around both the macro body and macro arguments!

`#define` substitution

1. The macro arguments are checked for occurrences of **`#define`**d symbols, and values are replaced as appropriate
2. Uses of preprocessor symbols are replaced by their **`#define`**d values
3. The now modified program is rescanned for **`#define`**d symbols, and if any are found, the process begins again at step 1

But string literals are not scanned for replacement

```
#define MAX 10
printf("The value of MAX is %d.\n", MAX);
```

Output: **`The value of MAX is 10.`**

Macros aren't functions

- Macros ...
  - ... are textually replaced, which can be faster than functions for short things
  - ... can't be recursive
  - ... can't be type-checked by the preprocessor
  - ... can result in difficult to find bugs when using complex macros - where is the error in the source code?
  - ... can have types as arguments:

```
#define PRINT_SIZE(type) \
    printf("%d\n", (int) sizeof(type))
```
```