

CMSC 216
Introduction to Computer Systems
Lecture 27
C Preprocessor

Jan Plane & Alan Sussman
{jplane, als}@cs.umd.edu

Administrivia

- Project 6 due Friday
 - questions?
- Project grade weights
 - total of 35% of your grade – each 6.2%, except Project 3 is 4%
- Practice final exam posted, with answers
- Final exam is on Mon., Dec. 13, 4:00-6:00PM
 - in BRB 1101
- Please do course evaluation, at www.CourseEvalUM.umd.edu

CMSC 216 - Wood, Sussman, Herman

2

Chapter 14, Reek

THE PREPROCESSOR

CMSC 216 - Wood, Sussman, Herman

3

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

CMSC 216 - Wood, Sussman, Herman

4

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

CMSC 216 - Wood, Sussman, Herman

5

The `#define` directive

- Syntax: `#define name lots of text`
- Substitutes lots of text for name throughout 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

CMSC 216 - Wood, Sussman, Herman

6

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;" */
```

CMSC 216 - Wood, Sussman, Herman

7

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
```

CMSC 216 - Wood, Sussman, Herman

8

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));`

CMSC 216 - Wood, Sussman, Herman

9

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));`

CMSC 216 - Wood, Sussman, Herman

10

Why we use all those parentheses

- Remember, substitution is essentially just find-and-replace, like in a text editor:
`#define SQUARE1(x) x * x`
`#define SQUARE2(x) (x * x)`
`#define BAD_SUM(x, y) (x) + (y)`

`s = SQUARE1(a + 1);`
`/* s = a + 1 * a + 1; */`
`s = 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!

CMSC 216 - Wood, Sussman, Herman

11

#define substitution

1. The macro arguments are checked for occurrences of **#defined** symbols, and values are replaced as appropriate
2. Uses of preprocessor symbols are replaced by their **#defined** values
3. The now modified program is rescanned for **#defined** 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.

CMSC 216 - Wood, Sussman, Herman

12

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))
```

CMSC 216 - Wood, Sussman, Herman

13

Other issues with macros

- What if macro arguments have side effects?


```
#define MIN(x, y) ((x) < (y) ? (x) : (y))
m = MIN(++a, --b);
/* m = ((++a) < (--b) ? (++a) : (--b)); */
```
- The solution? Don't use an expression with side effects as a macro argument.
- But how do you know if it's a macro? Use the naming convention we discussed and it becomes much more clear...

CMSC 216 - Wood, Sussman, Herman

14

Other preprocessor capabilities

- **#undef name**
 - needed to undefine symbols in order to assign a new value to them
 - you can undefine undefined symbols
- The compiler can define symbols at compile time:


```
gcc -D SIZE=300 -c file.c
```

 - defines **SIZE** to the value 300 while compiling the file, just as if **#define SIZE 300** appeared in the program

CMSC 216 - Wood, Sussman, Herman

15

Conditional compilation

```
#if const-expr1
  code here is included by the preprocessor iff const-expr1 is true
#elif const-expr2
  code here is included by the preprocessor if const-expr1 is false and const-expr2 is true
#else
  code here is included by the preprocessor if const-expr1 is false and const-expr2 is false
#endif
```

- This facility is useful if code must be different on different systems, but overuse tends to make code hard to read
- One type of *const-expr* is **defined(macro-name)**, which has the value 1 if *macro-name* is defined, and 0 if not
- The preprocessor directive **#error string** stops preprocessing (and compilation) at that point, and prints *string*

CMSC 216 - Wood, Sussman, Herman

16

File inclusion

- `#include "file.h"`
- What if `file.h` `#includes` another file?
 - That's fine, preprocessor can handle it
- What if two header files `#include` the same third header file?
 - Also okay, unless a program `#includes` both of the first two header files...

CMSC 216 - Wood, Sussman, Herman

17

File inclusion example

```
main.c:
#include <stdio.h>
#include "some_struct.h"
#include "another_struct.h"

int main() {
    Some_struct ss;
    Another_struct as;
    /* some code here */
    return 0;
}
```

```
another_struct.h:
#include "common_struct.h"

typedef struct {
    float x;
    Common_struct y;
} Another_struct;

Another_struct g(void);
```

```
some_struct.h:
#include "common_struct.h"

typedef struct {
    int a;
    char b;
    float c;
} Some_struct;

void f(Some_struct s, Common_struct t);
```

```
common_struct.h:
typedef struct {
    float x;
    double y;
} Common_struct;
```

How many times is `Common_struct` defined in `main.c` after preprocessing?

CMSC 216 - Wood, Sussman, Herman

18

File inclusion example, cont.

```
main.c:
#include <stdio.h>
#include "common_struct.h"

typedef struct {
    int a;
    char b;
    float c;
} Some_struct;

void f(Some_struct s, Common_struct t);
#include "common_struct.h"

typedef struct {
    float x;
    Common_struct y;
} Another_struct;

Another_struct g(void);

int main() {
    Some_struct ss;
    Another_struct as;
    /* some code here */
    return 0;
}
```

```
common_struct.h:
typedef struct {
    float x;
    double y;
} Common_struct;
```

CMSC 216 - Wood, Sussman, Herman

19

File inclusion example, cont.

```
main.c:
#include <stdio.h>
typedef struct {
    float x;
    double y;
} Common_struct;

/* other code from some_struct.h */
typedef struct {
    float x;
    double y;
} Common_struct;

/* other code from another_struct.h */

int main() {
    Some_struct ss;
    Another_struct as;
    /* some code here */
    return 0;
}
```

Can't define a type twice!

CMSC 216 - Wood, Sussman, Herman

20

File inclusion, cont.

- We can solve this problem using conditional compilation inside `common_struct.h`:

```
#if ! defined(COMMON_STRUCT_H)
#define COMMON_STRUCT_H
(previous contents of common_struct.h here)
#endif
```

- This solution is seen in many header files