

CMSC 216
Introduction to Computer Systems
Lecture 14
C Preprocessor, cont. and
Assembly Language

Larry Herman & Alan Sussman

Administrivia

- Project 4 posted today, due Monday, April 2
- Project 3 late deadline today
- Exam solutions posted today
 - you have until Tuesday after break to request regrade
 - check solutions before asking for a regrade
- Start reading Bryant and O'Hallaron Section 4.1 (Y86 subset) and Chapter 3, for more info on IA-32 instruction set architecture

Chapter 14, Reek

THE PREPROCESSOR

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 ...

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

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 f const-expr1 is false and const-expr2 is true
#else
    code here is included by the preprocessor if f 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*

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...

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?

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;
```

File inclusion example, cont.

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

Can't define a type twice!

```
/* 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;  
}
```

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

Chapters 3 and 4.1, Bryant and O'Hallaron

ASSEMBLY LANGUAGE

Assembly language

- The CPU uses machine language to perform all its operations
- Assembly is a much more readable translation of machine language, and it is what we work with if we need to see what the computer is doing
- Many different kinds of assembly languages; we'll focus on the Y86 language defined in the text (with minor modifications)

An example of Y86 assembly

- The summation of all integers from 1 to 1000:

```
irmovl $0,%eax    # sum = 0
irmovl $1,%ecx    # num = 1
Loop: addl %ecx,%eax # sum += num
irmovl $1,%edx    # tmp = 1
addl %edx,%ecx    # num++
irmovl $1000,%edx # lim = 1000
subl %ecx,%edx    # if lim - num >= 0
jge Loop          # loop again

wrint %eax        # printf("%d", sum)
irmovl $10,%edx   # ch = '\n'
wrch %edx         # printf("%c", ch)
halt
```

Assembly language concepts

- Registers
 - Fast-access locations that the CPU uses, rather than storing all variables in memory
 - In the code we'll be examining, these are named by three-letter codes, and begin with %
 - Y86 has 8 registers, listed on pg. 168 in Figure 3.2
- Memory
 - Program text is here, along with any data which cannot fit in the registers
 - Since there are usually many different pieces of data involved in a program, and a limited number of registers, program variables are stored in memory when not in immediate use

Assembly language instructions

- These typically have an instruction name, a list of registers, and sometimes have a constant value too
- Examples:

```
irmovl $1000,%edx
subl %ecx,%edx
jge Loop
```
- Specify which operation to perform, which registers to act upon, and any values that might be needed

Assembling assembly

- Machine code (pure numbers) is generated by translating each instruction into binary numbers that the CPU uses
- This process is called "assembling"; conversely, we can take assembled code and disassemble it into (mostly) human readable assembly language

Labels

- When the assembler encounters a label (e.g., **Loop:**), the address of the labeled instruction is used by the assembler to replace all references to that label in the program
- Labels do not need to be declared before use
- In Y86, instructions have variable lengths, and are not aligned; most take 6 bytes, but some use fewer

Label translation

<code>irmovl \$0,%eax</code>	<code>0x00: irmovl \$0,%eax</code>
<code>irmovl \$1,%ecx</code>	<code>0x06: irmovl \$1,%ecx</code>
<code>Loop: addl %ecx,%eax</code>	<code>0x0c: addl %ecx,%eax</code>
<code>irmovl \$1,%edx</code>	<code>0x0e: irmovl \$1,%edx</code>
<code>addl %edx,%ecx</code>	<code>0x14: addl %edx,%ecx</code>
<code>irmovl \$1000,%edx</code>	<code>0x16: irmovl \$1000,%edx</code>
<code>subl %ecx,%edx</code>	<code>0x1c: subl %ecx,%edx</code>
<code>jge Loop</code>	<code>0x1e: jge 0x0c</code>
<code>wrint %eax</code>	<code>0x23: wrint %eax</code>
<code>irmovl \$10,%edx</code>	<code>0x25: irmovl \$10,%edx</code>
<code>wrch %edx</code>	<code>0x2b: wrch %edx</code>
<code>halt</code>	<code>0x2d: halt</code>

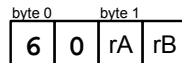
Instruction encoding

- In any assembly language, each instruction name has a numeric opcode associated with it; Y86 opcodes use one byte each
- Registers also have a numeric mapping, as do labels (as we saw last time)
- These numbers are used to encode each instruction into a number

Encoding examples

- In Y86, the **addl** instruction has the opcode of **0x60**
- The register names **%edx** and **%ecx** map to numbers 2 and 1, respectively
- The encoding for an **addl** instruction is specified by the Y86 instruction set as following this format (pgs. 339, 340):

– **addl** rA, rB



- So what would the numeric encoding of **addl %edx, %ecx** be?
– **0x6021**

A full assembly

- This is the result of running the Y86 assembler on the example assembler source code:

```

0x000: 308000000000 |      irmovl $0,%eax    # sum = 0
0x006: 308101000000 |      irmovl $1,%ecx    # num = 1
0x00c: 6010          | Loop: addl  %ecx,%eax   # sum += num
0x00e: 308201000000 |      irmovl $1,%edx    # tmp = 1
0x014: 6021          |      addl  %edx,%ecx    # num++
0x016: 3082e8030000 |      irmovl $1000,%edx # lim = 1000
0x01c: 6112          |      subl  %ecx,%edx    # if lim - num >= 0
0x01e: 750c000000 |      jge   Loop        # loop again
|
0x023: f308          |      wrint %eax        # printf("%d", sum)
0x025: 30820a000000 |      irmovl $10,%edx   # ch = '\n'
0x02b: f128          |      wrch  %edx        # printf("%c", ch)
0x02d: 10           |      halt
    
```