CSE 303
Lecture 13b

The C preprocessor

reading: *Programming in C* Ch. 13

slides created by Marty Stepp

http://www.cs.washington.edu/303/

C preprocessor

- **preprocessor** : Part of the C compilation process; recognizes special # statements, modifies your source code before it is compiled

function	description
`#include <`***filename***`>`	insert a library file's contents into this file
`#include "`***filename***`"`	insert a user file's contents into this file
`#define` ***name [value]***	create a preprocessor symbol ("variable")
`#if` ***test***	`if` statement
`#else`	`else` statement
`#elif` ***test***	`else if` statement
`#endif`	terminates an `if` or `if/else` statement
`#ifdef` ***name***	`if` statement; `true` if ***name*** is defined
`#ifndef` ***name***	`if` statement; `true` if ***name*** is *not* defined
`#undef` ***name***	deletes the given symbol name

Constants

- The preprocessor can be used to create constants:

```
#define NUM_STUDENTS  100
#define DAYS_PER_WEEK 7
...

double grades[NUM_STUDENTS];
int six_weeks = DAYS_PER_WEEK * 6;   // 42
printf("Course over in %d days", six_weeks);
```

  - When the preprocessor runs before compilation, 7 is literally inserted into the code wherever DAYS_PER_WEEK is seen
    - the name DAYS_PER_WEEK does not exist in the eventual program

```
int six_weeks = 7 * 6;   // 42
```

Debugging code

- The preprocessor is often used to include optional debug code:

```
#define DEBUG
...

#ifdef DEBUG
    // debug-only code
    printf("Size of stack = %d\n", stack_size);
    printf("Top of stack  = %p\n", stack);
#endif
    stack = stack->next;    // normal code
```

  - How is this different from declaring a `bool/int` named DEBUG?

Advanced definitions

- #define can be used to dialect the C language:

```
#define AND    &&
#define EQUALS ==
#define DEREF  ->
...

Point p1 = (Point*) malloc(sizeof(Point));
p1 DEREF x = 10;
p1 DEREF y = 10;
if (p1 DEREF x EQUALS p1 DEREF y AND p1 DEREF y > 0) {
    p1 DEREF x++;
}
```

    - Warning: Evil may result.

Preprocessor macros

- `#define` can accept arguments to create a *macro*.
  - sort of like a function, but injected inline before compilation

```
#define SQUARED(x)    x * x
#define ODD(x)        x % 2 != 0
...

int a = 3;
int b = SQUARED(a);
if (ODD(b)) {
    printf("%d is an odd number.\n", b);
}
```

  - The above literally converts the code to the following and compiles:

```
int b = a * a;
if (b % 2 != 0) { ...
```

Subtleties

- the preprocessor is dumb; it just replaces tokens with tokens

```
#define foo 42
int food = foo;          // int food = 42;       ok
int foo = foo + foo;     // int 42 = 42 + 42;    bad
```

- preprocessor macros can do a few things functions cannot:

```
#define NEW(t)   (t*) calloc(1, sizeof(t))
...
Node* list = NEW(Node);
```

Caution with macros

- since macros are injected directly, strange things can happen if you pass them complex values

```
#define ODD(x)       x % 2 != 0
...
if (ODD(1 + 1)) {
    printf("It is odd.\n");    // prints!
}
```

  - The above literally converts the code to the following and compiles:

```
if (1 + 1 % 2 != 0) {
```

  - Fix: *Always* surround macro parameters in parentheses.

```
#define ODD(x)      (x) % 2 != 0
```

Running the preprocessor

- to run *only* the preprocessor, use the `-E` argument to `gcc`:

```
$ gcc -E example.c
int main(void) {
    if ((1 + 1) % 2 != 0) {
        printf("It is odd.\n");
    }
    return 0;
}
```

  - outputs the result of preprocessing `example.c` to standard-out; rarely used in practice, but can be useful for debugging / learning

- to define a preprocessor variable, use the `-D` ***variable*** argument:

```
$ gcc -D DEBUG -o example example.c
```