C Preprocessor (continued)

COM S 113

February 15, 1999

Announcements

Should have read just about all of K&R through Chapter 7

All previous assignments & quizzes graded

Assignment 4 available, due in a week

Simple Macro Definitions

```
#define NULL 0
#define EOF (-1)
#define GET getc(stdin)
#define begin { /* allows compound statements */
#define end }   /* like: while (e) begin ... end */
```

Comments in replacement strings are legal—they just become part of the replacement. What problems could occur?

Parameterized Macro Definitions

```
#define getchar() getc(stdin)
#define putchar(x) putc(x, stdout)
#define MAX(x,y) ((x)>(y)?(x):(y))
#define MIN(x,y) ((x)>(y)?(y):(x))
#define UPPER(c) ((c)-'a'+'A')
  /* assumes c is lowercase letter */
#define LOWER(c) ((c)-'A'+'a')
```

Example Effect of Parameterized Macro

```
while ((c = getchar()) != EOF)
  putchar(UPPER(c));
```

is transformed into

```
while ((c = getc(stdin)) != (-1))
  putc(((c)-'a'+'A'), stdout);
```

Macro Parameters as String Constants

Suppose we want a macro `PRINTI` with one parameter that when called with `max`, it expands to

```
printf("max = %d\n", max)
```

Macro Parameters as String Constants (cont.)

```
#define PRINTI(x) printf("x = %d\n", x)
```

means that `PRINTI(max)` will expand to

```
printf("x = %d\n", max)
```

since the preprocessor does not scan string constants for replacement

Macro Parameters as String Constants (cont.)

```
#define PRINTI(x) printf(x "= %d\n", x)
```

means that `PRINTI(max)` will expand to

```
printf(max "= %d\n", max)
```

which also is not what we want

Macro Pararmeters as String Constants (cont.)

Solution is to use `#` operator; using `#x` in a replacement string expands to the value of the parameter enclosed in double quotes

```
#define PRINTI(x) printf(#x " = %d\n", x)
```

Then `PRINTI(max)` expands to

```
printf("max" " = %d\n", max)
```

Pasting Items in Macro Definitions

Parameters adjacent to `##` operator are substituted, the `##` is removed, and result is rescanned

```
#define RedApple 3
#define GreenApple 5
#define Apple(x) x##Apple
```

So `Apple(Red)` expands to `RedApple` which is rescanned to produce `3`

Rescanning

After macro invocation replaced by corresponding body, the result is rescanned for further macro invocations

However, a macro name mentioned within its own body is *not* expanded

So `#define sizeof (int) sizeof` is legal and does not produce infinite recursion

Removing Macro Definitions

`#undef` *identifier*

undefines the macro definition of *identifier*

An identifier defined as a macro must be undefined before it can be redefined *unless the redefinition is* identical *to the original definition*

Predefined Macros

`__LINE__`	Current line number in source file
`__FILE__`	Source file name
`__DATE__`	Date of compilation in form `"Mmm dd yyyy"`
`__TIME__`	Time of compilation in form `"hh:mm:ss"`
`__STDC__`	Value of 1 means ANSI-conforming

These can't be removed with `#undef`

File Inclusion

Actually three forms of `#include`:

```
#include "fname"
#include <fname>
#include sequence-of-chars
```

In third form, macro substitutions are performed on sequence of characters, and result must match one of the first two forms

Conditional Compilation

Selective compilation of portions of programs, such as only those portions necessary for a particular system

Advantages:

1. It provides a compile-time parameterization facility, so you can generate programs that have different kinds of structures

2. Greater storage efficiency because extraneous code is not included

3. Greater time efficiency because decisions made at compile time

Conditional Compilation (continued)

`#if` *constant-expression*

lines for true case

`#endif`

constant-expression must be an integral constant expression that does not use `sizeof` or a cast or an enumeration constant

Conditional Compilation (continued)

It can use the `defined` operator (same syntax as `sizeof`), which returns true if its operand is currently defined as a macro

```
#if !defined(MAX_STK_SIZE)
#define MAX_STK_SIZE 128
#endif
```

Conditional Compilation (continued)

The preprocessor also supports `#else` and `#elif`, which behave as you'd expect

```
#if defined(u370)
#define BUFSIZ 4096
#elif defined(vax) || defined(u3b)
#define BUFSIZ 1024
#endif
```

Conditional Compilation (continued)

```
typedef struct {
#if defined(vax) || defined(u3b)
   int _cnt;              unsigned char *_ptr;
#else
   unsigned char *_ptr; int _cnt;
#endif
   unsigned char *_base;
   char _flag, _file;
} FILE;
```

Conditional Compilation (continued)

Also provided are two special forms of `#if`:

`#ifdef` *identifier*

`#ifndef` *identifier*

These behave like the following:

`#if defined` *identifier*

`#if !defined` *identifier*

Using Conditional Compilation for Disabling Large Blocks of Code

```
#if 0
  lots of code here
  that can even contain /* comments */
#endif
```

Error Directive

`#error` *token-sequence*

Causes the preprocessor to write a message that includes the token sequence

```
#if !(defined ABC || defined DEF)
#error "You must define either ABC or DEF!"
#endif
```

Error Directive (continued)

Since the `#error` directive requires a *token-sequence*, the following is *not* allowed:

```
#error What's going on?
```

The single quote is treated as starting a character constant, and since it is never closed, this is illegal

Similarly, using unmatched double quotes is an error

Null Directive

A line containing just the character # is ignored.

Sample Implementation of `assert` macro

```
#ifdef  NDEBUG
#define assert(e) ((void)0)
#else
#define assert(e) (void)((e) || \
         (__assert(#e, __FILE__, __LINE__), 0))
#endif
```