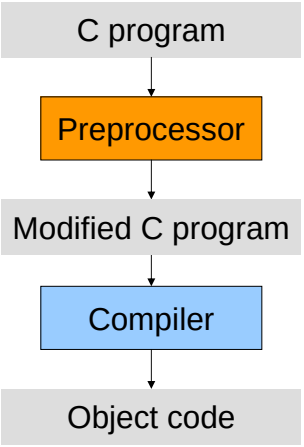
Preprocessing and Macros

CS 217

C Preprocessor

- Invoked automatically by the C compiler
  - **1st pass: invokes C preprocessor**
  - 2nd pass: invokes compiler on the resulting C code
- Manually invoke C preprocessor
  `gcc –E foo.c`



Preprocessor Directives

- Three kinds of directives
  - File inclusion
    - **`#include`**
  - Conditional compilation
    - **`#if, #ifdef, #ifndef, #elif, #else, #endif`**
  - Macros
    - **`#define`**
- Rules
  - Always starts with a line with "#"
  - Can appear anywhere in a program
  - Comments may appear on the same line
  - Takes one line unless explicitly continue
    - `#define MAX_CHARS 300   /* max file name size */`
    - `#define MAX_FILES \`
      `100`

File Inclusion

- Why?
  - Allow a program or a module's implementation to use certain interfaces
- An interface or a header file contains declarations for a module
  - Name of the header file should end in **`.h`**
- User-define header files **`" ... "`**
  `#include "mydefs.h"`
- System header files: **`< ... >`**
  `#include <stdio.h>`

Conditional Compilation

- Why?
  - One source for many platforms or many cases
  - Need to have special source for specific situations
- Conditional compilation
  - **`#ifdef`** *name*
  - **`#ifndef`** *name*
  - **`#if`** *expr*
  - **`#elif`** *expr*
  - **`#else`**
  - **`#endif`**
- Removing macro definitions
  - **`#undef plusone`**

Allows this file To be included Multiple times

```
#ifndef FOO_H
#define FOO_H

#ifdef WINDOWS_OS
#include <windows.h>
#elif LINUX_OS
#include <linux.h>
#endif
.
.
.
#endif
```

Allows selective (OS-specific) compilation

```
gcc -DWINDOWS_OS foo.c
```


Another Example

- Conditionally compile debugging code

```
...
    if (some expr) {
        some code
#ifdef DEBUG
        printf("this path taken\n" );
#endif
    }
    else
        some other code;
```

```
gcc -DDEBUG foo.c
```


Macros

- Provide parameterized text substitution
- Why?
  - The code may be slightly faster
  - No type checking
- Macro definition

```
#define MAXLINE 120
#define lower(c) ((c)-`A'+'a')
```

- Macro replacement

```
char buf[MAXLINE+1];
```

becomes

```
char buf[120+1];

c = lower(buf[i]);
```

becomes

```
c = ((buf[i])-`A'+'a');
```


Macros: Use "(" and ")"

- Always parenthesize macro parameters in definition

```
#define plusone(x) x+1

i = 3*plusone(2);
```

becomes

```
i = 3*2+1
```

```
#define plusone(x) ((x)+1)

i = 3*plusone(2);
```

becomes

```
i = 3*((2)+1)
```

Macros: Careful about Side-Effects

- "++" and "—" operators create side effects
- Always avoid side-effects in parameters passed to macros

```
#define max(a, b) ((a)>(b)?(a):(b))

y = max(i++, j++)
```

becomes

```
y = ((i++)>(j++)?(i++):(j++));
```

- Question
  - **What data type can we use in the macro "max"**

More on Macros: # Operator

- # in the macro converts an argument into a string literal

```
#define PRINT_INT(x) printf( #x "= %d\n", x)
...
PRINT_INT( x * y );
...
```

will become

```
...
printf( "x * y" "= %d\n", x*y);
...
```

- Question
  - We now have **"foo""bar"** in printf, what does this mean?

More on Macros: ## Operator

- You may never need to use this
- ## pastes two tokens into one
- Example

```
#define GENERIC_MAX(type) \
type type##_max(type x, type y) \
    { return x > y ? x : y };

GENERIC_MAX(float)
```

becomes

```
float float_max(float x, float y)
    { return x > y ? x : y };
```


More on Macros: #error

- Let the preprocessor print out an error message

```
#error message
```

- **Example**

```
#if defined(WINDOWS)
...
#elif defined(LINUX)
...
#elif defined(MAC_OS_X)
...
#else
#error no OS specified
#endif
```

Some General Properties

- A macro may contain another macro
  - Preprocessor will rescan to replace
  - How many times does the preprocessor rescan?
- A macro definition is in effect until the end of the file
- A macro may not be defined twice
- You can use “#undef” to undefine a macro

Summary

- Preprocessing allows programmers to modify C source code automatically
  - File inclusion
  - Conditional compilation
  - Macros
- Macros are sometimes useful, but you need to be careful
  - Make sure that you remember the rules
  - Must use parentheses for the arguments
  - Avoid side effects
  - Can improve readability
  - Can reduce the amount of code
  - Overuse can also reduce readability, or introduce opportunities for bugs