Preprocessing

Class 16

The Preprocessor

- eventually C++ will completely eliminate the preprocessor
- but it is an integral part of C
- there are 12 directives in standard C, but we will only look at some of them

```
#include        #if          #ifdef
                #elif
#define         #else        #ifndef
                #endif
```

#include

- the compiler replaces this line with the contents of the named file
- if the named file contains a #include directive, that file is recursively included at its location

- the filename must be enclosed in either angle brackets or double quotes

```
#include <foo.h>
#include "foo.h"
```

- filenames in angle brackets are searched for in standard system locations, usually /usr/include and its subdirectories
- filenames in quotes are searched for in the current directory
- can also add to the include search path with -I compiler option

```
$ clang -I /foo/path ...
```

#define

- creates an identifier and a replacement string for that identifier
- throughout the file, before compilation begins, every occurrence of the identifier as a token occurs, it is replaced by the replacement string
- the most common use is to create "constants"
- for example, this:

```
#define LEFT 1
#define RIGHT 0

printf("%u %u %u", LEFT, RIGHT, LEFT+1);
```

becomes

```
printf("%u %u %u", 1, 0, 1+1);
```

#define

- no substitution occurs if the identifier is within a quoted string

```
#define LEFT 1

printf("LEFT");
```

- no substitution occurs if the identifier is not a token

```
#define LEFT 1

unsigned x = LEFTfoo + 1;
```

#define

- a second main use is to create a function-like substitution

```
1 #define ABS(a) (a) < 0 ? -(a) : (a)
2
3 printf("abs value of -1 and 1: %u %u\n", ABS(-1), ABS(1));
```

- the parentheses in the replacement string are critically important
- without them:

```
1 #define ABS(a) a < 0 ? -a : a
2
3 ABS(10-20)
```

would give:
`10-20 < 0 ? -10-20 : 10-20`
which is clearly incorrect

#if – #endif

- the primary use we will make of this construct is to comment out blocks of code during debugging and testing
- normal comments cannot be nested

```
1  /* comment out the following 3 lines
2  int x = 5;
3  int y = 10; /* y is the flux capacitor value */
4  int z = 15;
5  end of commented-out section */
```

- the comment on line 1 ends at the end of line 3
- there is a syntax error on line 5

#if – #endif

- instead, do:

```
1 #if 0
2 int x = 5;
3 int y = 10; /* y is the flux capacitor value */
4 int z = 15;
5 #endif
```

#ifdef – #endif

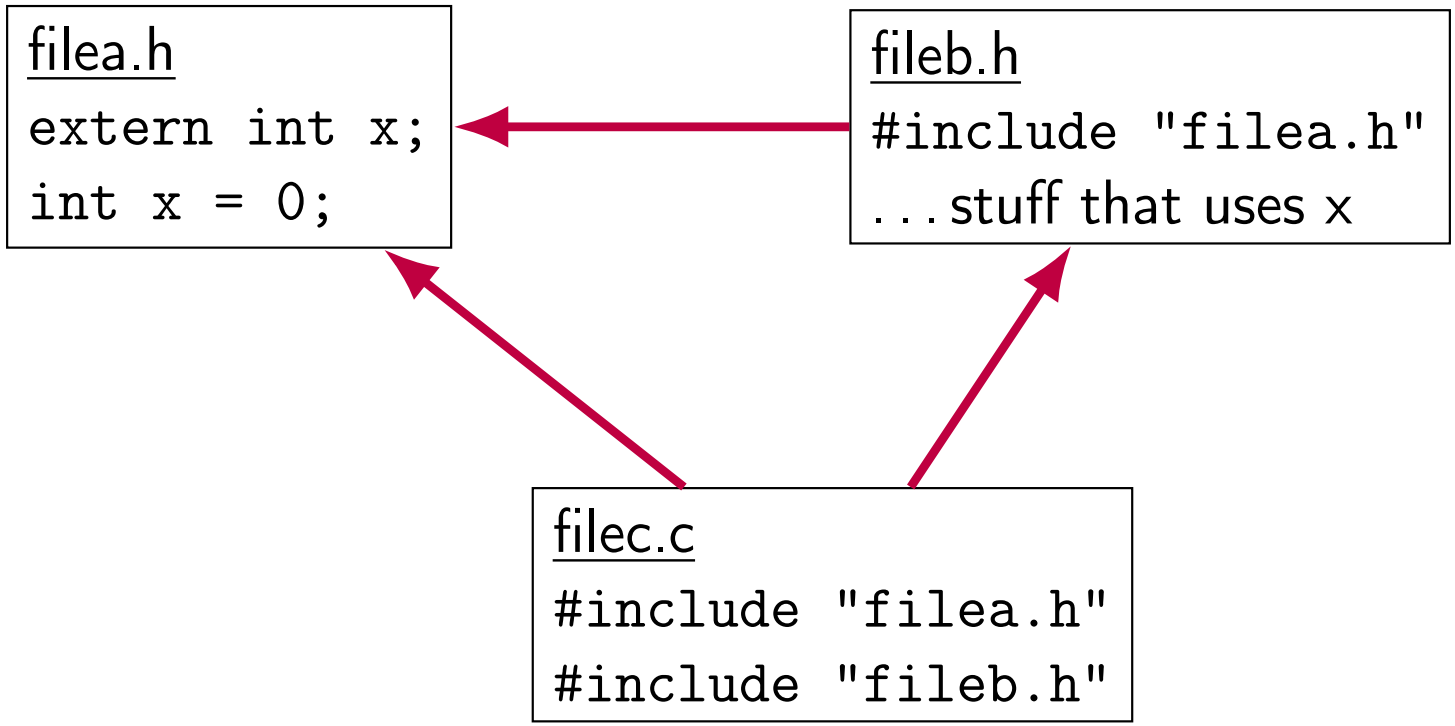
- used for conditional compilation
- often used for debugging
- this is also the third use of #define

```
1  #define DEBUG
2  ...
3
4  foo = bar() + bam() / qux(foobar);
5  #ifdef DEBUG
6    printf("foo: %d\n", foo);
7  #endif
```

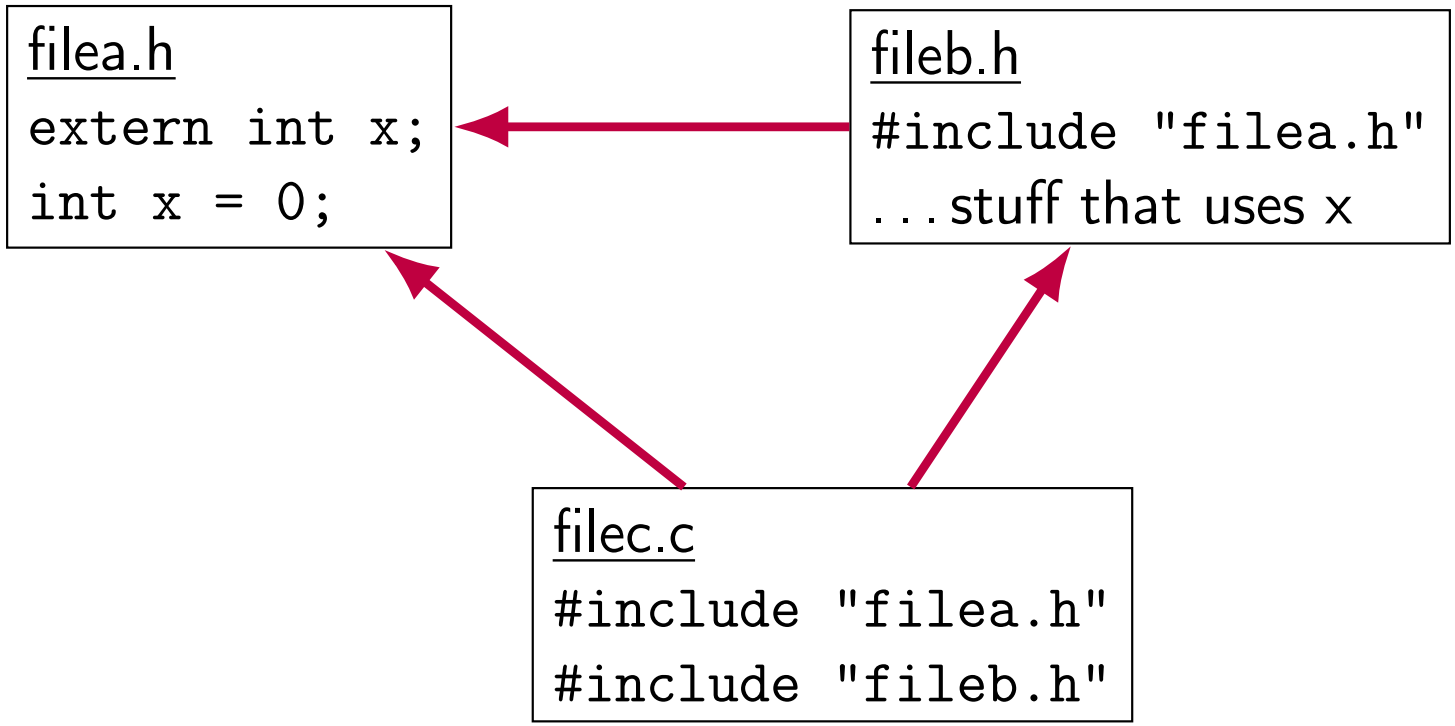
- during debugging and testing, line 1 is present and debug messages are printed
- for production, comment out line 1, and debug messages are suppressed with only a single line change

#ifndef – #endif

- this is a bit more subtle
- .h files are used for declarations and definitions
- some are specific to one .c file; others are project-wide
- extremely common to have multiple .h files included in a project
- each provides some stuff
- consider the following arrangement

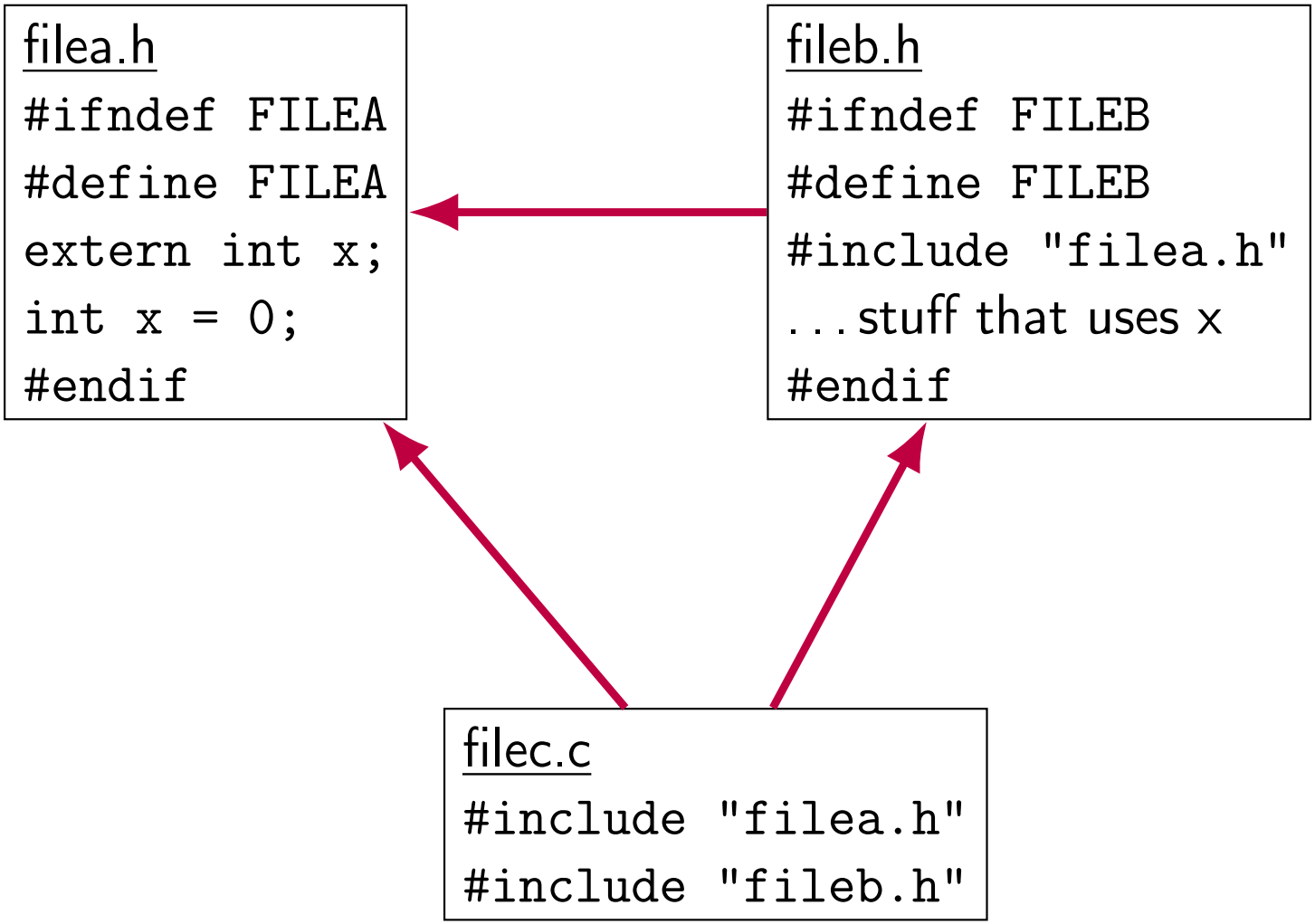


- what happens?



- what happens?
- duplicate definition error
- in filec.c, x is defined twice

Solution


filea.h
#ifndef FILEA
#define FILEA
extern int x;
int x = 0;
#endif
fileb.h
#ifndef FILEB
#define FILEB
#include "filea.h"
...stuff that uses x
#endif
filec.c
#include "filea.h"
#include "fileb.h"

#ifndef – #endif

- the fourth use of #define
- every .h file uses #ifndef #define – #endif pattern

Viewing Results of Preprocessing

- add the -E switch to clang
- the normal clang command:

```
$ clang -pedantic-errors -Weverything -std=c89 -o foo foo.c
```

- to view preprocessor:

```
$ clang -pedantic-errors -Weverything -std=c89 -E foo.c
```

- output with #includes is normally huge, so pipe it into less or redirect into a file