

# Incomplete Types

- 6.2.5 Types

- ◆ para 1 - Types are partitioned into

- object types (types that fully describe objects),
- function types (types that describe functions),
- incomplete types (types that describe objects but lack information needed to determine their sizes).

- 6.2.5 Types

- ◆ para 19 – The void type ... is an incomplete type that cannot be completed.
- ◆ para 22 – An array of unknown size is an incomplete type.
- ◆ para 22 – A structure or union type of unknown content is an incomplete type.

- 6.7.5.3 Function declarators

- para 14 – An empty list in a function declarator that is part of a definition of that function specifies that the function has no parameters.

```
int f(void) { ... } ← equivalent
int f()    { ... } ←
```



- para 14 – The empty list in a function declarator that is not part of a definition of that function specifies that no information about the number or types of the parameters is supplied.

```
int f(void); ← not equivalent
int f(); ←
```



void

- when do you get a diagnostic?
  - ◆ gcc clo.c
  - ◆ gcc -Wall clo.c
  - ◆ gcc -Wmissing-prototypes clo.c

```
int f();

int main(void)
{
    return f(4, 2);
}

int f(int value)
{
    return value;
}
```

clo.c



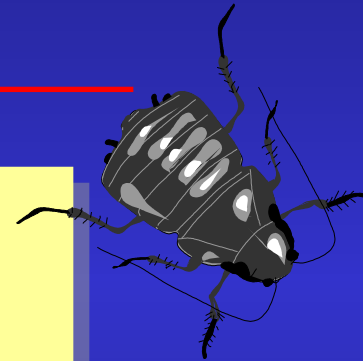
- when do you get a diagnostic?
  - ◆ gcc clo.c → NO
  - ◆ gcc -Wall clo.c → NO! why not?
  - ◆ gcc -Wmissing-prototypes clo.c → YES

```
int f();

int main(void)
{
    return f(4, 2);
}

int f(int value)
{
    return value;
}
```

clo.c

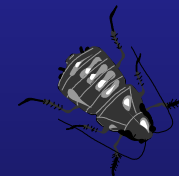


- why do you still get a diagnostic?
  - ◆ gcc -Wmissing-prototypes clo.c  
→ warning: no previous prototype for 'f'

```
int f(int value)
{
    return value;
}

int main(void)
{
    return f(42);
}
```

clo.c



```
static int f(int value)
{
    return value;
}

int main(void)
{
    return f(42);
}
```

clo.c

typically  
via a #include

```
int f(int value);

int f(int value)
{
    return value;
}

int main(void)
{
    return f(42);
}
```

clo.c

- 6.2.5 Types

- ◆ para 22 – An array of unknown size is an incomplete type.



```
int a1[42];  
extern int a2[42];  
typedef int a3[42];
```

these are  
object types



```
int a1[];  
extern int a2[];  
typedef int a3[];
```

these are  
incomplete types

unknown size

- are these fragments conforming or not?

```
typedef struct wibble wibble;
```

```
    wibble w1[42];  
extern    wibble w2[42];  
typedef   wibble w3[42];
```

```
    wibble w1[];  
extern    wibble w2[];  
typedef   wibble w3[];
```



- **neither are conforming!**

```
typedef struct wibble wibble;
```

```
    wibble w1[42];  
extern  wibble w2[42];  
typedef wibble w3[42];
```



```
    wibble w1[];  
extern  wibble w2[];  
typedef wibble w3[];
```



Incomplete arrays can be incomplete in their size  
*but not* in their element type

- **6.7.5.2 Array declarators**
  - ◆ para 1 – the element type shall not be an incomplete type
- are these fragments conforming or not?

```
typedef struct wibble wibble;  
  
void f(wibble x[4]);  
void f(wibble x[]);  
void f(wibble * ptr);
```

**exercise**



```
typedef struct wibble wibble;
```

```
void f(wibble x[4]);
```



```
void f(wibble x[]);
```



```
void f(wibble * ptr);
```



- are these code fragments conforming?
- if not, why not?

```
int table[] =  
{  
    1, 2, 3,  
    4, 5, 6  
};
```

```
int table[][] =  
{  
    { 1, 2, 3 },  
    { 4, 5, 6 }  
};
```





```
int table[] =  
{  
    1, 2, 3,  
    4, 5, 6  
};
```

this is conforming



```
int table[][] =  
{  
    { 1, 2, 3 },  
    { 4, 5, 6 }  
};
```

this is not conforming



- 6.7.5.3 Function declarators
  - ◆ para 12 - If the function declarator is not part of a definition of that function, parameters may have incomplete types...

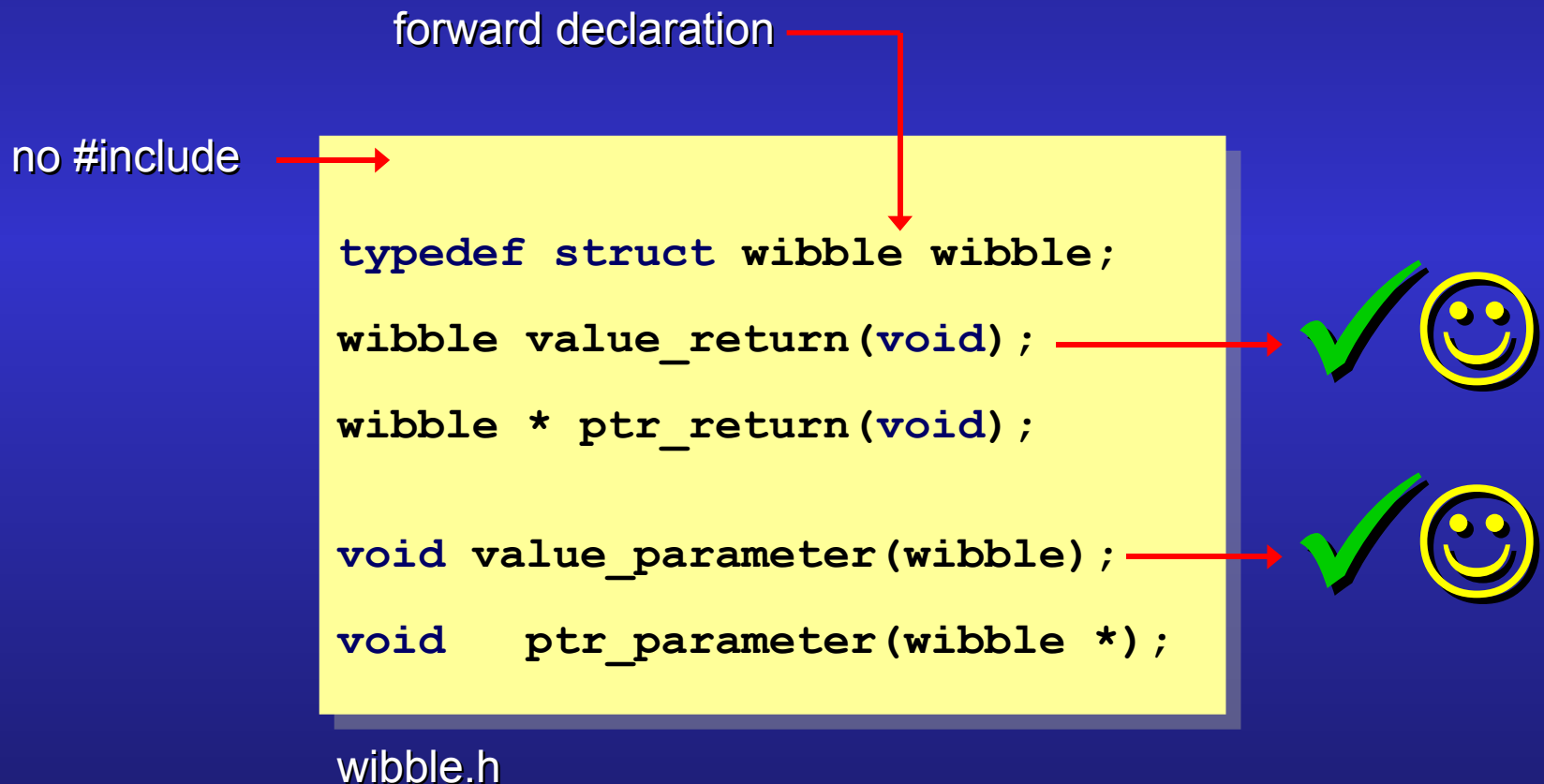
forward declaration

no #include

```
typedef struct wibble wibble;
wibble value_return(void);
wibble * ptr_return(void);

void value_parameter(wibble);
void ptr_parameter(wibble *);
```

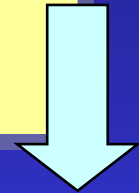
wibble.h



- manage dependencies aggressively
  - ◆ both physically and logically

→ `#include "wibble.h"`

```
void many_includes(wibble * ptr);  
void are_unnecessary(wibble value);
```



```
void many_includes(struct wibble * ptr);  
void are_unnecessary(struct wibble value);
```



→ `typedef struct wibble wibble;`

```
void many_includes(wibble * ptr);  
void are_unnecessary(wibble value);
```

