Program Structure

"to get a deeper understanding of the language"

Deep C - a 3 day course
Jon Jagger & Olve Maudal

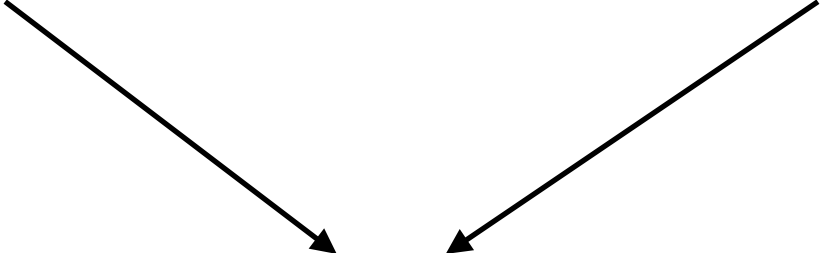# forward declaration

- #include is the most obvious code reflection of coupling

when is a #include required?

when is a #include _not_ required?

```
#include "wibble.h"
```

```
struct wibble;
```

```
#ifndef WIBBLE_INCLUDED
#define WIBBLE_INCLUDED

...

struct wibble
{
    ...
};

#endif
```

forward declaration

- which of 1,2,3,4,5,6 _won't_ compile?

```
struct wibble;

struct data_member
{
   struct wibble   value;     // 1
   struct wibble * pointer;   // 2
};

struct wibble   global_value;    // 3
struct wibble * global_pointer; // 4

extern struct wibble   ext_global_value;    // 5
extern struct wibble * ext_global_pointer; // 6
```

data declarations/definitions

forward declaration

- 1 and 3 won't compile

```
struct wibble;

struct data_member
{
    struct wibble   value;     // 1  ⬅──────────────── ✘
    struct wibble * pointer;   // 2
};

struct wibble   global_value;    // 3  ⬅─────────────── ✘
struct wibble * global_pointer; // 4

extern struct wibble   ext_global_value;    // 5
extern struct wibble * ext_global_pointer; // 6
```

data declarations/definitions

forward declaration

- which of 7,8,9,10 _won't_ compile?

```
struct wibble;

struct wibble   return_value(void);        //  7
struct wibble * return_pointer(void);      //  8

void parameter_value(struct wibble w);     //  9
void parameter_pointer(struct wibble * p); // 10
```

function declarations

forward declaration

- they all compile!

```
struct wibble;

struct wibble   return_value(void);          //  7
struct wibble * return_pointer(void);        //  8

void parameter_value(struct wibble w);       //  9
void parameter_pointer(struct wibble * p);   // 10
```

function declarations

forward declaration

- which of 11,12,13,14 *won't* compile?

```
struct wibble;

struct wibble return_value(void)          // 11
{ ... }

void parameter_value(struct wibble w)     // 12
{ ... }

struct wibble * return_pointer(void)      // 13
{ ... }

void parameter_pointer(struct wibble * p) // 14
{ ... }
```

function definition '*signatures*'

forward declaration

- 11,12 won't compile

```
struct wibble;

struct wibble return_value(void)          // 11
{ ... }

void parameter_value(struct wibble w)     // 12
{ ... }

struct wibble * return_pointer(void)      // 13
{ ... }

void parameter_pointer(struct wibble * p) // 14
{ ... }
```

function definition '*signatures*'

forward declaration

- which of 15,16,17 _won't_ compile*

```
struct wibble;

void pass_pointer(struct wibble * p) // 15
{
    pass(p);
}


void arrow_pointer(struct wibble * p) // 16
{
    arrow(p->member);
}


void deref_pointer(struct wibble * p) // 17
{
    deref(*p);
}
```

function definition _bodies_

* ignore pass(),arrow(),deref() not being prototyped

forward declaration

- 16 and 17 *won't* compile

```
struct wibble;

void pass_pointer(struct wibble * p) // 15
{
    pass(p);
}


void arrow_pointer(struct wibble * p) // 16
{
    arrow(p->member);
}


void deref_pointer(struct wibble * p) // 17
{
    deref(*p);
}
```
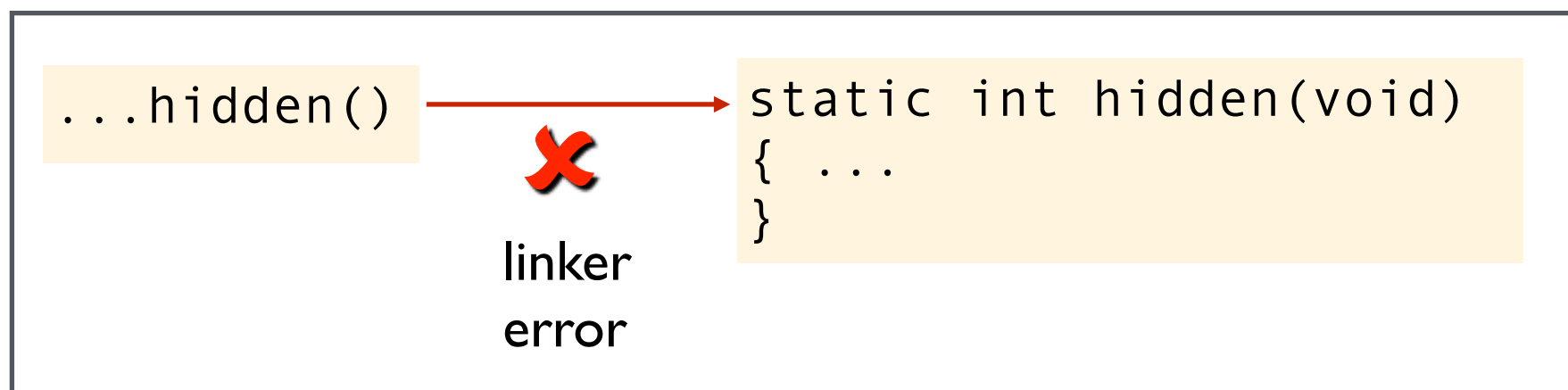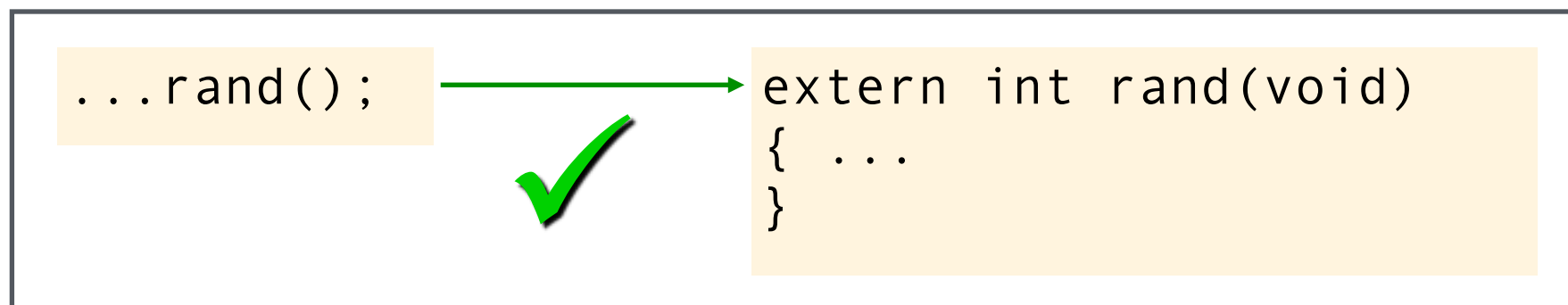
function definition *bodies*

linking

• a linker links the use of an identifier in one file with its definition in another file
• an identifier is made available to the linker by giving it external linkage (the default) or using the `extern` keyword
• an identifier is hidden from the linker by giving it internal linkage using the `static` keyword

```
...rand();          ──────✔──────▶    extern int rand(void)
                                       { ...
                                       }
```

```
...hidden()         ──────✘──────▶    static int hidden(void)
                                       { ...
                                       }
           linker
           error
```

external linkage pattern

• if a function definition has external linkage it *should* have been previously prototyped (in a header file)

eg.h
```
...
int eg(const char * s);  ←——————— function prototype
...
```

eg.c
```
#include "eg.h"

int eg(const char * s)
{    ...                    ✓
}
```

Using **-Wmissing-prototypes** detects
function definitions with external linkage
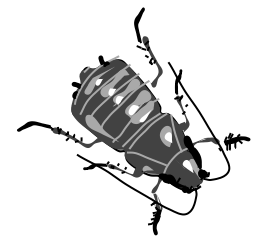but _no_ prior function prototype

```
...                                                    eg.h
```

```
#include "eg.h"                                        eg.c

int eg(const char * s)  ✖
{    ...
}
```

```
$ gcc ... -Werror -Wmissing-prototypes eg.c
error: no previous prototype for 'eg'
$
```

data linkage

- without a storage class specifier or an initializer a data definition is tentative (external) – and can be repeated!
- at link time the duplicates collapse into one!
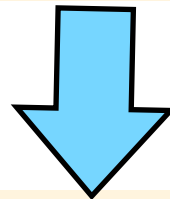- this is confusing and *not* compatible with C++
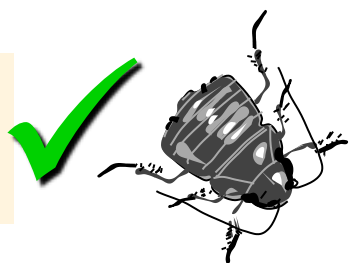
fubar.h

?
```
int v;
```

snafu.h

?
```
int v;
```

```
#include "fubar.h"
#include "snafu.h"
```

```
int v; // external, tentative definition
int v; // external, tentative definition
```
✔

not an error in C :-(

duplicate definition error in C++ :-)

data linkage recommendation

- extern data *declarations:* use extern keyword, do *not* initialize

multiple declarations

```
extern int v;

extern int v;
```
✓

- extern data *definitions:* do *not* use extern keyword, *do* initialize

multiple definitions

```
int v = 42;

int v = 42;
```
✗

spot the problem

snafu.h

```
#ifndef SNAFU_INCLUDED
#define SNAFU_INCLUDED

#include <stddef.h>

int snafu(size_t);

#endif
```

wibble.h

```
#ifndef WIBBLE_INCLUDED
#define WIBBLE_INCLUDED

int wibble(const char *);
void wobble(size_t);

#endif
```

snafu.c

```
#include "snafu.h"
...
```

wibble.c

```
#include "snafu.h"
#include "wibble.h"
...
```
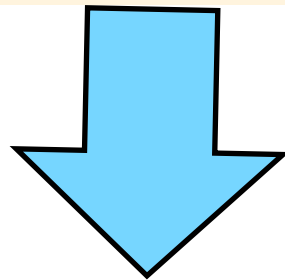
<stddef.h> contains a typedef for size_t

spot the problem

- wibble.c depends on the order of its #includes

wibble.c
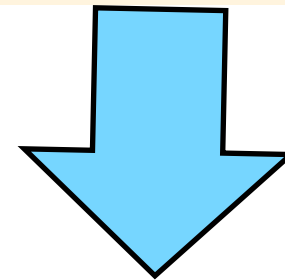```
#include "snafu.h"
#include "wibble.h"
...
```

```
typedef ... size_t;

int snafu(size_t);

int wibble(const char *);
void wobble(size_t);
...
```
✓

wibble.c
```
#include "wibble.h"
#include "snafu.h"
...
```

```
int wibble(const char *);
void wobble(size_t);

typedef ... size_t;

int snafu(size_t);
...
```
✗

# recommendation

- each source file should could #include it's *own* header first
  - easy to automate a test for this
- consider checking each individual header file compiles! (-x c)
  - as part of the build

```
#ifndef WIBBLE_INCLUDED
#define WIBBLE_INCLUDED

#include <stddef.h> // size_t

int wibble(const char *);
void wobble(size_t);

#endif
```
wibble.h

```
#include "wibble.h"
#include "snafu.h"
...
```
wibble.c

static linking
dynamic linking

position independent code
 show assembler fragments that show different op codes

optimization
different levels
implication for testing
why you should always compile with optimisation on - because it forces the compilers
span of attention to increase and thus it can see and diagnose more. Example? Does
Olve know one off the top of his head?

# summary