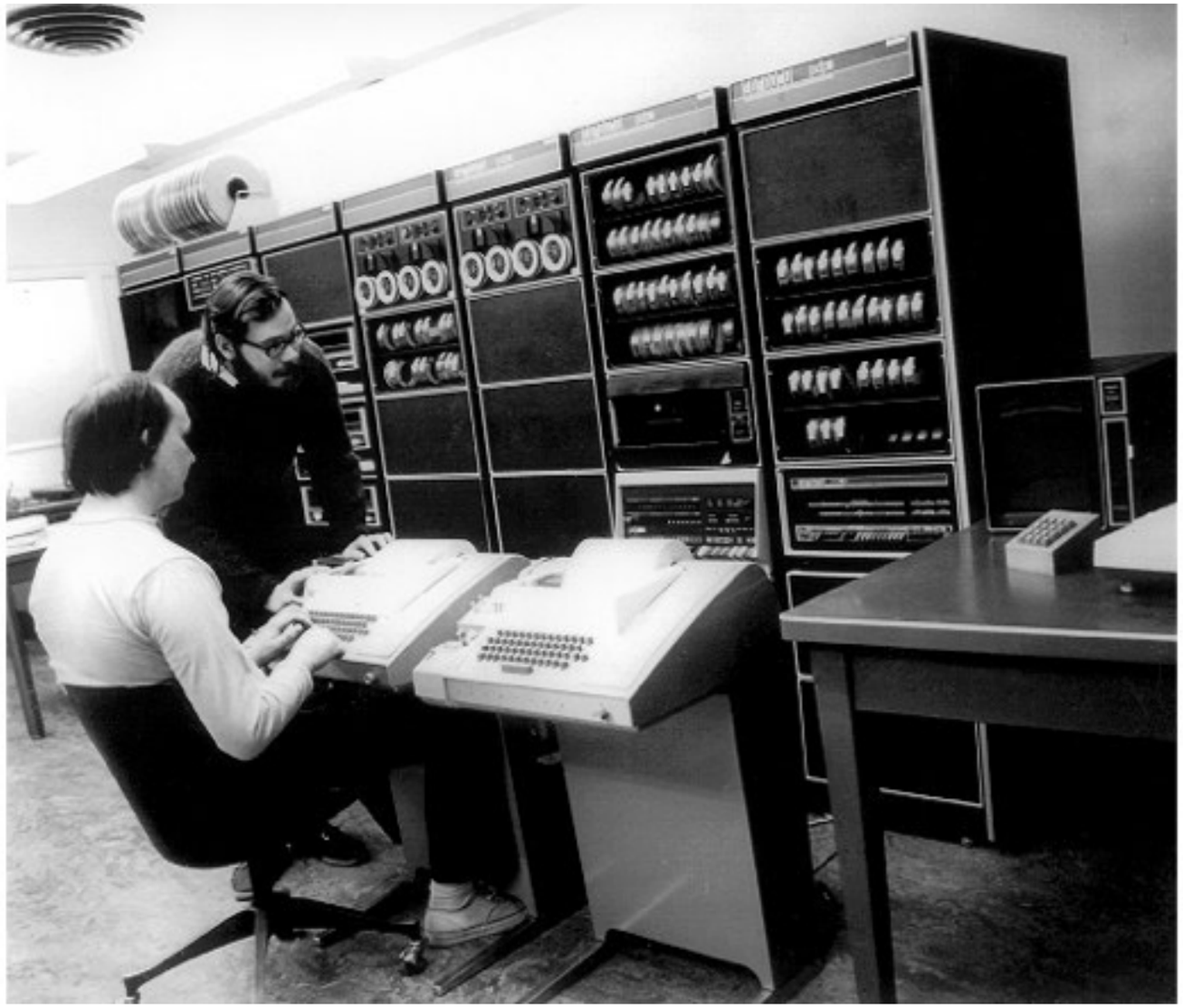
Program Structure

"to get a deeper understanding of the language"



Deep C - a 3 day course

Jon Jagger & Olve Maudal

forward declaration

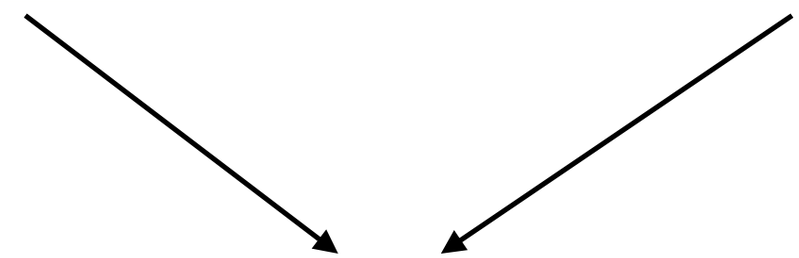
- #include is the most obvious code reflection of coupling

when is a #include required?

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

when is a #include *not* required?

```
struct wibble;
```



```
#ifndef WIBBLE_INCLUDED
#define WIBBLE_INCLUDED

...

struct wibble
{
    ...
};

#endif
```

forward declaration

- which of 1,2,3,4,5,6 *<u>won't</u>* 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 <u>*won't*</u> 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*

forward declaration

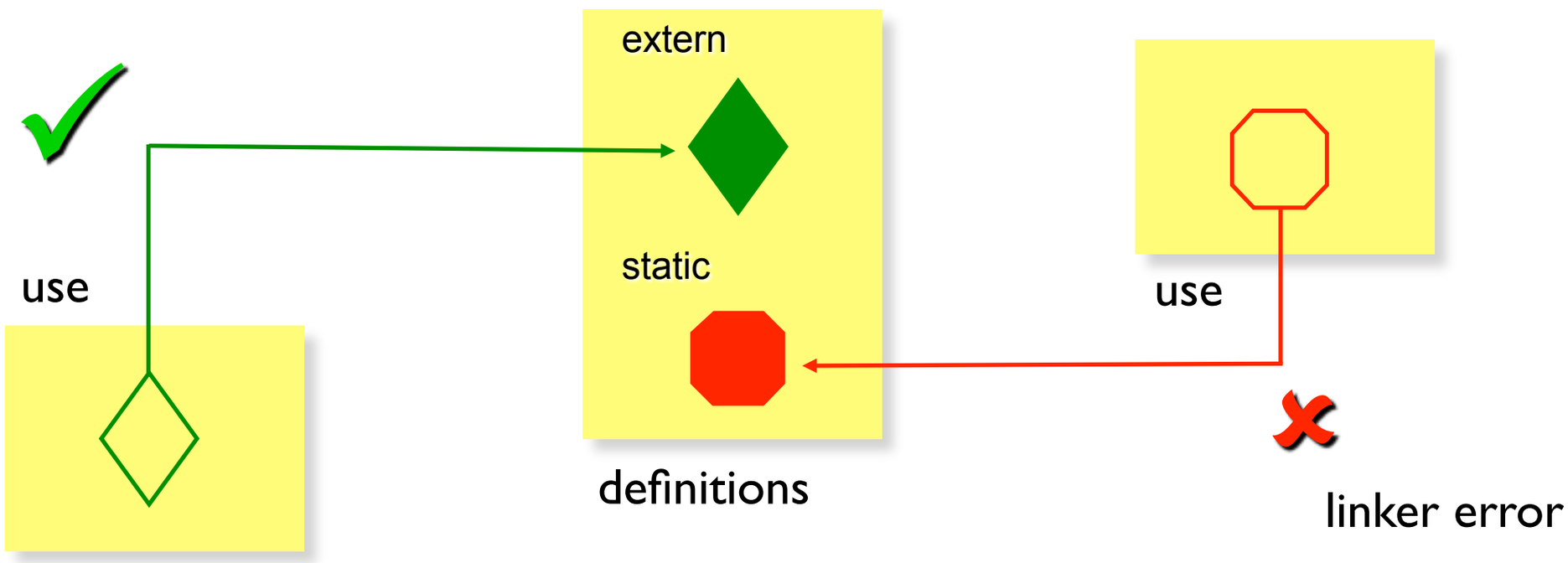
- 16 and 17 *won't* compile

```
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



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)
{    ...
}
```

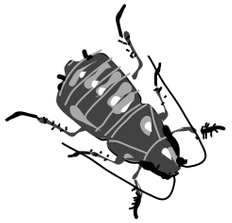
Using -Wmissing-prototypes detects
function definitions with external linkage
but *no* prior function prototype

```
...
```

eg.h

```
#include "eg.h"

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

eg.c



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

If the function should have external linkage then adding a function prototype...

eg.h

```
...
int eg(const char * s);
...
```

eg.c

```
#include "eg.h"

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

```
$ gcc ... -Werror -Wmissing-prototypes eg.c
$
```

If the function should have internal linkage then make it so!

```
...
```

eg.h

```
#include "eg.h"

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

eg.c

```
$ gcc ... -Werror -Wmissing-prototypes eg.c
$
```

data linkage

- without a storage class or an initializer a data definition is tentative (external) – and can be repeated!
- this is confusing and *not* compatible with C++

ok in C, duplicate definition errors in C++

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

recommendation: extern data *declarations*

- use explicit extern keyword, do not initialize

recommendation: extern data *definitions*

- do not use extern keyword, do initialize

multiple declarations ok

```
extern int v;
extern int v;
```

single definition with initializer

```
int v = 42;
```

ensure header files are self-contained
  include own header first
  compile headers as part of build?

ensure header files are idempotent
  macro guards

static linking
dynamic linking

position independent code

optimization

seams
-compile time
-link time
-runtime

summary