

Program Structure

“to get a deeper understanding of the language”



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

forward declaration

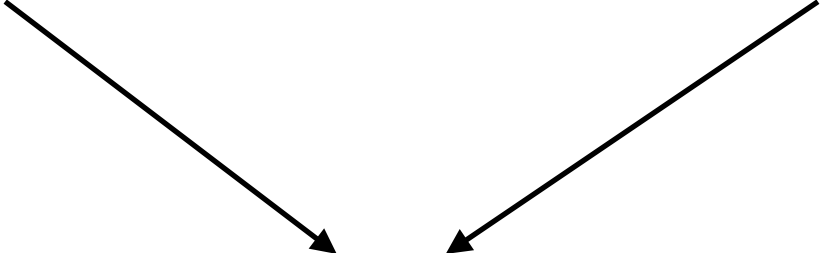
- `#include` is the most obvious code reflection of code coupling

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 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 prototypes

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 prototypes

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);
}

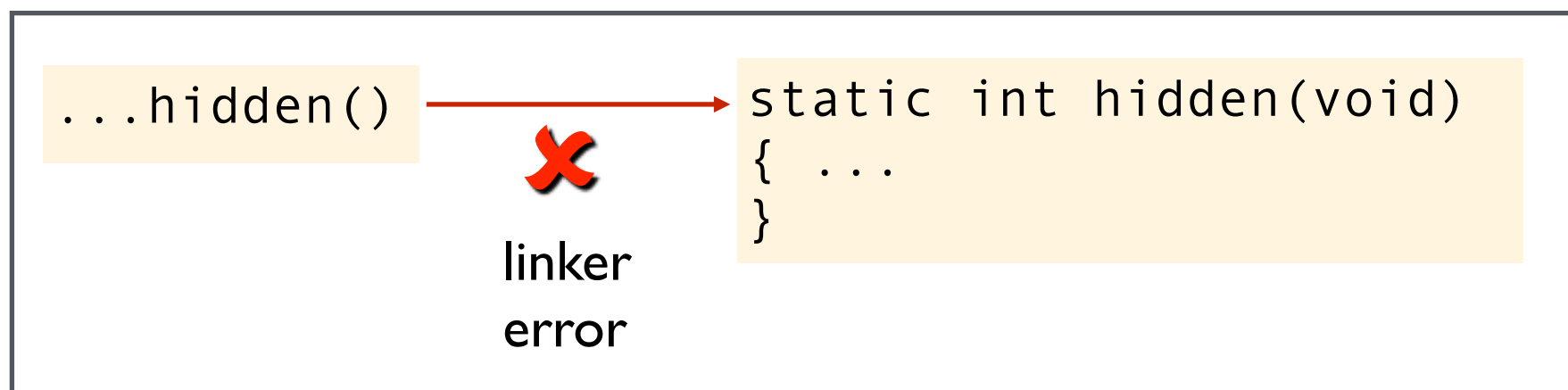
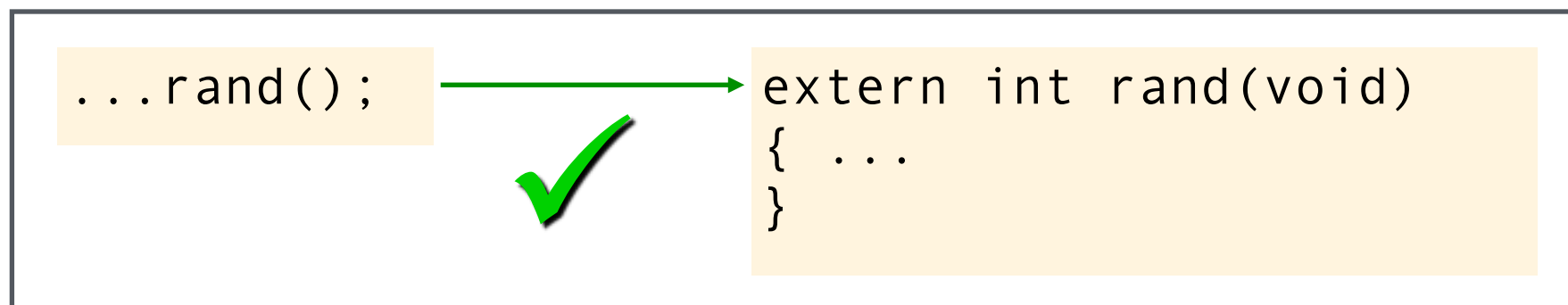
void arrow_pointer(struct wibble * p) // 16
{
    ✗ → arrow(p->member);
}

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

function definition bodies

linking

- a linker links the use of an identifier with its definition
- 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




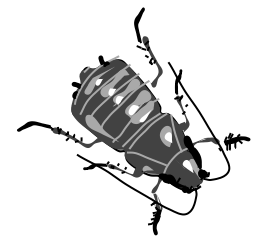
```
...
```

eg.h

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

eg.c

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



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

If the function should have external linkage then add a function prototype to the header...



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



eg.h

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

eg.c

```
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 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++

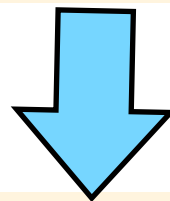
fubar.c

? int v;

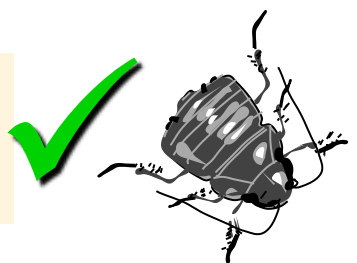
snafu.c

? int v;

```
gcc fubar.c snafu.c
```



```
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 = 0;
```

```
int v = 0;
```



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"
...
```



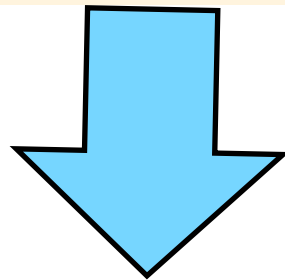
<stddef.h> contains a typedef for size_t

spot the problem

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

wibble.c

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

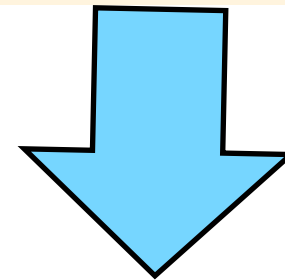


```
typedef ... size_t;  
  
int snafu(size_t);  
  
int wibble(const char *);  
void wobble(size_t);  
...
```

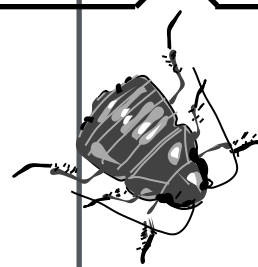


wibble.c

```
#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
```

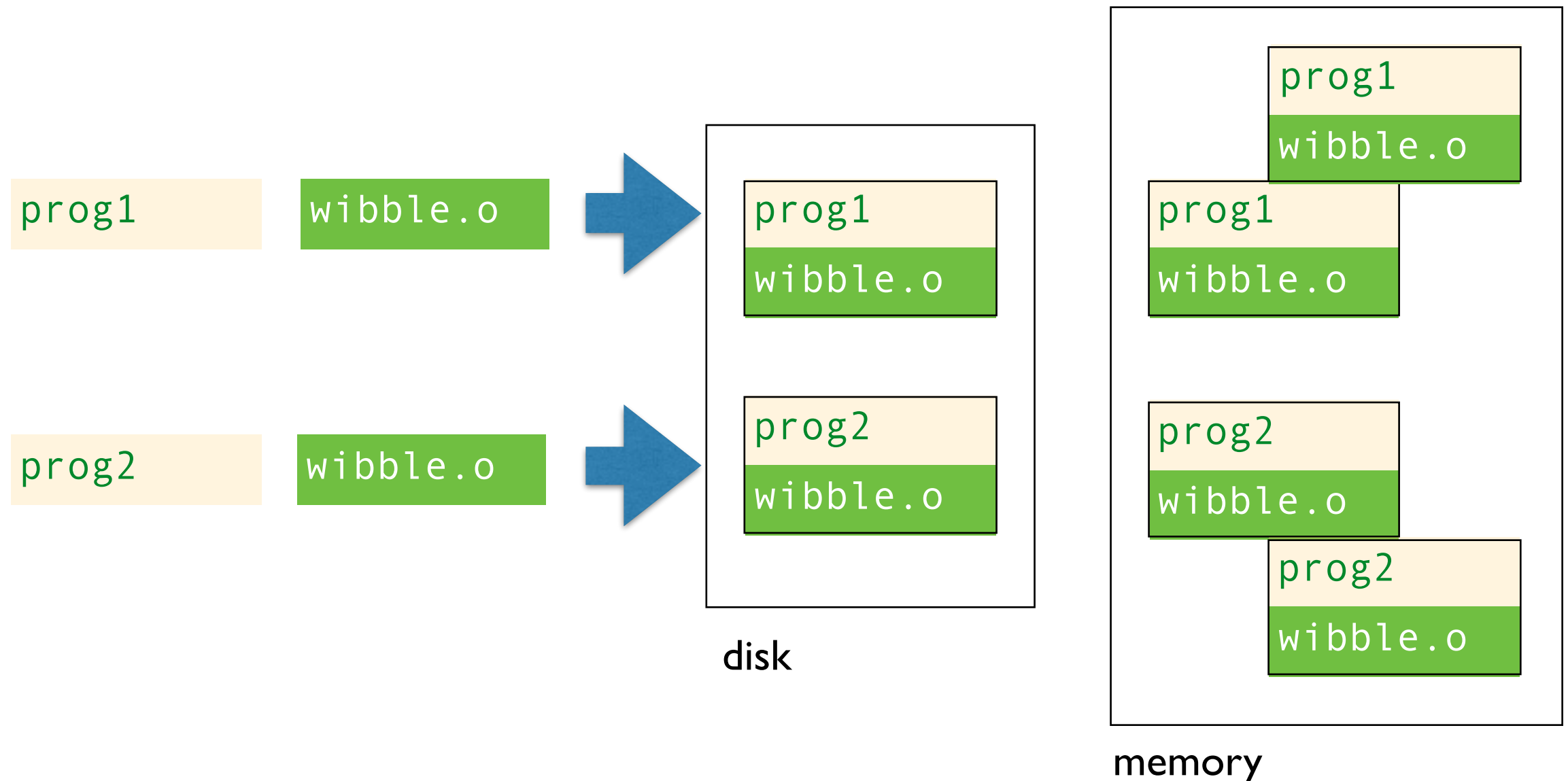
wibble.h

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

wibble.c

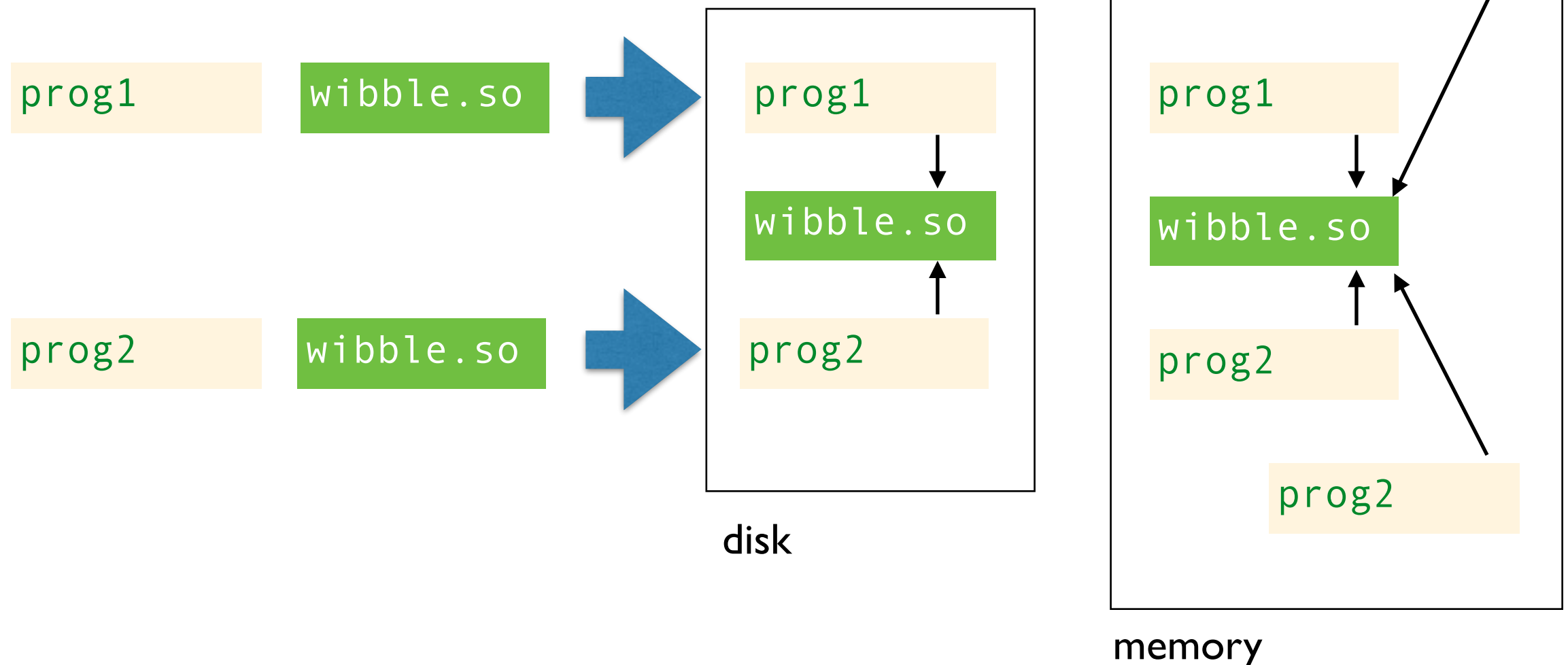
static linking

- static libraries have their code directly embedded
- static libraries are not shared
- don't need static library to run executable
- simplest option when you need to distribute the executable
- to fix a bug you have to relink every executable



dynamic linking

- dynamic libraries do not have their code embedded directly
- dynamic libraries is shared and loaded at load-time
- dynamic library has to exist to run executable
- reduced disk & memory footprint
- to fix a bug you only have to replace the dynamic library



dynamic linking

- compile .c files with -fPIC option (Position Independent Code)

```
gcc $(CFLAGS) -fPIC wibble.c
```

- convert .o files into .so files using -shared option

```
gcc -shared wibble.o -o libwibble.so
```

- build executable telling gcc where shared libraries live

```
gcc -L/sandbox ... -o test -lwibble
```

- run the executable telling the os where to look for new .so files

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/sandbox  
$ ./test  
All tests passed
```

optimization

-O, -O0

no optimization; make debugging produce expected results; the default

-O1

moderate optimization; tries to reduce code and size and execution time without increasing compilation time significantly;

-O2

full optimization minus space-time optimizations; increases compilation time

-O3

-O2 plus aggressive inlining of subprograms - may increase program size
attempts to vectorize loops

-Os

optimize to reduce size (code and data)

-Og

enable optimizations that do not interfere with debugging

optimization

Requesting greater optimization forces the compiler to increase its 'span of attention'.
This helps it detect more warnings.
You should compile with optimisation on.



```
int n;  
scanf("%d", &n);
```

```
$ gcc -Wall -Wextra -O0 ...  
$
```

```
$ gcc -Wall -Wextra -O2 ...  
'scanf' ... [-Werror=unused-result]
```

summary

- forward declarations help reduce coupling
- -Wmissing-prototypes for sensible linkage patterns
- avoid tentative data declarations
- every header file should compile in its own right
- static linking and dynamic linking
- switch optimization on by default