# Multi-File C Programs

**main.c**

```
int sum(int n);

int array[2] = {1, 2};

int main()  {
  int val = sum(2);
  return val;
}
```

**sum.c**

```
extern int array[];

int sum(int n) {
  int i, s = 0;
  for (i = 0; i < n; i++)
    s += array[i];
  return s;
}
```

```
$ gcc -O2 -o sum main.c sum.c
```

# Multi-File C Programs

**main.c**

```
int sum(int n);

int array[2] = {1, 2};

int main()  {
  int val = sum(2);
  return val;
}
```

**sum.c**

```
extern int array[];

int sum(int n) {
  int i, s = 0;
  for (i = 0; i < n; i++)
    s += array[i];
  return s;
}
```

```
$ gcc -O2 -o sum main.c sum.c
```

**cpp + cc1 + as + ld**

# Multi-File C Programs

**main.c**

```
int sum(int n);

int array[2] = {1, 2};

int main()  {
  int val = sum(2);
  return val;
}
```

**sum.c**

```
extern int array[];

int sum(int n) {
  int i, s = 0;
  for (i = 0; i < n; i++)
    s += array[i];
  return s;
}
```

```
$ gcc -O2 -c main.c
$ gcc -O2 -c sum.c
$ gcc -O2 -o sum main.o sum.o
```

# Multi-File C Programs

**main.c**

```
int sum(int n);

int array[2] = {1, 2};

int main()  {
  int val = sum(2);
  return val;
}
```

**sum.c**

```
extern int array[];

int sum(int n) {
  int i, s = 0;
  for (i = 0; i < n; i++)
    s += array[i];
  return s;
```

**cpp + cc1 + as**

```
$ gcc -O2 -c main.c
$ gcc -O2 -c sum.c
$ gcc -O2 -o sum main.o sum.o
```

# Multi-File C Programs

**main.c**

```
int sum(int n);

int array[2] = {1, 2};

int main()  {
  int val = sum(2);
  return val;
}
```

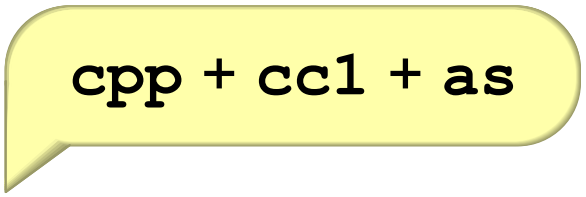
**sum.c**

```
extern int array[];

int sum(int n) {
  int i, s = 0;
  for (i = 0; i < n; i++)
    s += array[i];
  return s;
}
```

```
$ gcc -O2 -c main.c
$ gcc -O2 -c sum.c
$ gcc -O2 -o sum main.o sum.o
```

**ld**

# Why Compile and Link Separately?

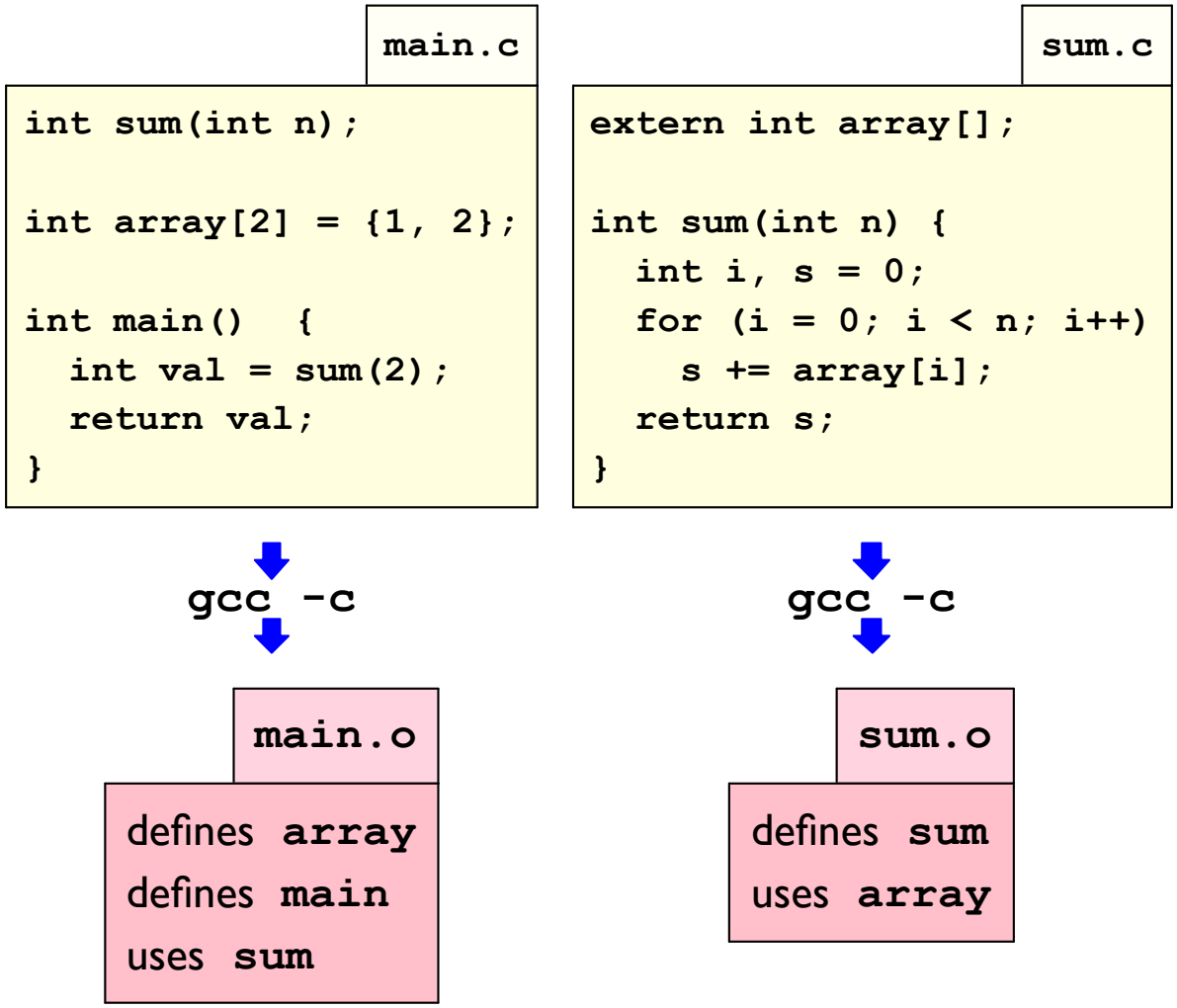



```
$ gcc -O2 -c main.c
$ gcc -O2 -c sum.c
$ gcc -O2 -o sum main.o sum.o
```

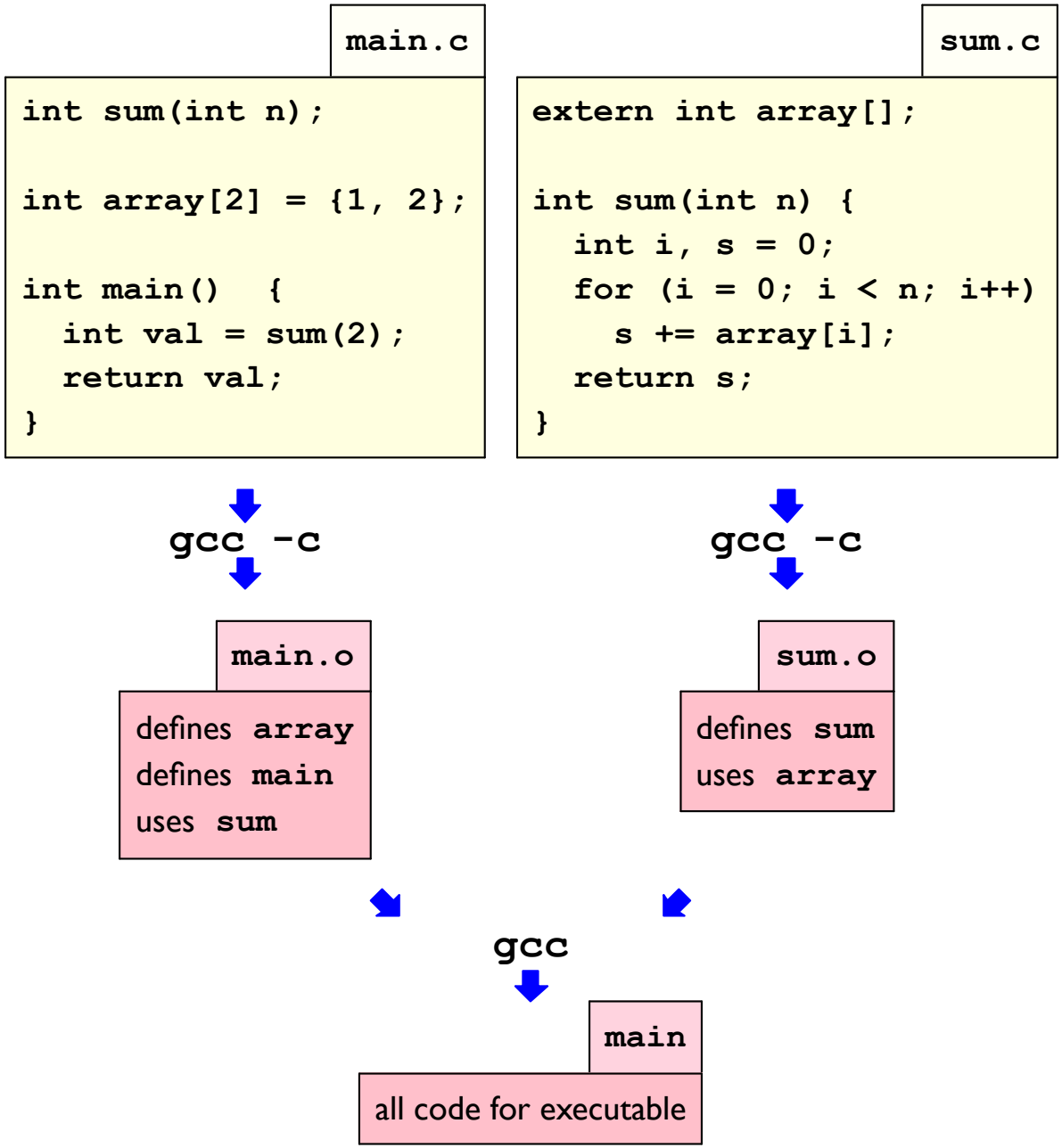
The `cc1` step tends to be the long one

- *Faster:* Don't re-compile if nothing changed
- *Faster:* One `sum.o` with different main programs
- *Smaller:* Sharing `.o` leads to in-memory code sharing

# Separate Compilation and Linking



# Separate Compilation and Linking



# Using Definitions from Other Files

Function:

```
int sum(int n);
```

Data:

```
int array[];
```

# Using Definitions from Other Files

Function:

```
extern int sum(int n);
```

Data:

```
extern int array[];
```

**`extern`** is optional, but especially good practice for data

# Providing Definitions to Other Files

Function:

```
int sum(int n) {
  ....
}
```

Data:

```
int array[2] = {1, 2};
```

# Providing Definitions to Other Files

Function:

```
int sum(int n) {
  ....
}
```

Data:

```
int array[2];
```

initialization is optional, but especially good practice for data

# Declaring and Defining

It's ok to both declare and define:

```
extern int a[];
int sum(int n);

....

int a[2] = {1, 2};

int sum(int n) {
    ....
}
```

Declaration and definion must be consistent

# Weak Symbols

```
int array[2];
```

Definition or use declaration?

... Depends on linking

Data without initialization or **`extern`** ⇒ **weak** symbol

- Only **weak** ⇒ *weak definition used*
- **Strong** + **weak** ⇒ *strong definition used*
- **Strong** + **strong** ⇒ *error*
- **Weak** + **weak** ⇒ *silent choice!*

# Strong and Weak Symbols

**main.c**

```
int sum(int n);

int array[2] = {1, 2};

int main()  {
  int val = sum(2);
  return val;
}
```

**sum.c**

```
int array[];

int sum(int n) {
  int i, s = 0;
  for (i = 0; i < n; i++)
    s += array[i];
  return s;
}
```

⇒ link with **`array`** from **`main.c`**

# Strong and Weak Symbols

**main.c**

```
int sum(int n);

int array[2];

int main()  {
  int val = sum(2);
  return val;
}
```

**sum.c**

```
int array[2] = {1, 2};

int sum(int n) {
  int i, s = 0;
  for (i = 0; i < n; i++)
    s += array[i];
  return s;
}
```

⇒ link with **`array`** from **`sum.c`**

# Strong and Weak Symbols

**main.c**

```
int sum(int n);

int array[2] = {1, 2};

int main()  {
  int val = sum(2);
  return val;
}
```

**sum.c**

```
int array[2] = {1, 2};

int sum(int n) {
  int i, s = 0;
  for (i = 0; i < n; i++)
    s += array[i];
  return s;
}
```

⇒ *error*

# Strong and Weak Symbols

**main.c**

```
int sum(int n);

int array[2];

int main()  {
  int val = sum(2);
  return val;
}
```

**sum.c**

```
int array[];

int sum(int n) {
  int i, s = 0;
  for (i = 0; i < n; i++)
    s += array[i];
  return s;
}
```

⇒ link with either **`array`**!

Make multiple definitions an error with
**`gcc -fno-common`**

# Consistency of Definitions

**main.c**

```
double sum(int n);

double array[2] = {1, 2};

int main()  {
  double val = sum(2) > 0.0;
  return val;
}
```

**sum.c**

```
extern int array[];

int sum(int n) {
  int i, s = 0;
  for (i = 0; i < n; i++)
    s += array[i];
  return s;
}
```

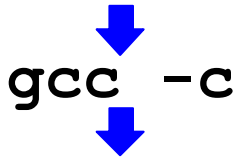
# Consistency of Definitions

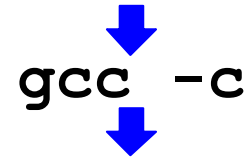
**main.c**

```
double sum(int n);

double array[2] = {1, 2};

int main()  {
  double val = sum(2) > 0.0;
  return val;
}
```

**sum.c**

```
extern int array[];

int sum(int n) {
  int i, s = 0;
  for (i = 0; i < n; i++)
    s += array[i];
  return s;
}
```




**main.o**

defines **array**
defines **main**
uses **sum**

**sum.o**

defines **sum**
uses **array**

# Consistency of Definitions

**main.c**

```
double sum(int n);

double array[2] = {1, 2};

int main()  {
  double val = sum(2) > 0.0;
  return val;
}
```

**sum.c**

```
extern int array[];

int sum(int n) {
  int i, s = 0;
  for (i = 0; i < n; i++)
    s += array[i];
  return s;
}
```

"Random" result due to **sum** mismatch

# Ensuring Consistency

**sum.h**

```
extern int array[];
int sum(int n);
```

**main.c**

```
#include "sum.h"

int array[2] = {1, 2};

int main()  {
  int val = sum(2);
  return val;
}
```

**sum.c**

```
#include "sum.h"

int sum(int n) {
  int i, s = 0;
  for (i = 0; i < n; i++)
    s += array[i];
  return s;
}
```

# Ensuring Consistency

main.c

```
#include "sum.h"

int array[2] = {1, 2};

int main()  {
  int val = sum(2);
  return val;
}
```

sum.c

```
#include "sum.h"

int sum(int n) {
  int i, s = 0;
  for (i = 0; i < n; i++)
    s += array[i];
  return s;
}
```



main.c

```
extern int array[];
int sum(int n);

int array[2] = {1, 2};

int main()  {
  int val = sum(2);
  return val;
}
```

sum.c

```
extern int array[];
int sum(int n);

int sum(int n) {
  int i, s = 0;
  for (i = 0; i < n; i++)
    s += array[i];
  return s;
}
```

# Ensuring Consistency

main.c

```
#include "sum.h"

int array[2] = {1, 2};

int main()  {
  int val = sum(2);
  return val;
}
```

sum.c

```
#include "sum.h"

int sum(int n) {
  int i, s = 0;
  for (i = 0; i < n; i++)
    s += array[i];
  return s;
}
```



gcc -E

main.c

```
extern int array[];
int sum(int n);

int array[2] = {1, 2};

int main()  {
  int val = sum(2);
  return val;
}
```

sum.c

```
extern int array[];
int sum(int n);

int sum(int n) {
  int i, s = 0;
  for (i = 0; i < n; i++)
    s += array[i];
  return s;
}
```

# Non-Exported Definitions

**main.c**

```
int sum(int a[], int n);

int array[2] = {1, 2};

int main()  {
  int val = sum(array, 2);
  return val;
}
```

**sum.c**

```
int sum(int a[], int n) {
  int i, s = 0;
  for (i = 0; i < n; i++)
    s += a[i];
  return s;
}
```

# Non-Exported Definitions

**main.c**

```
int sum(int a[], int n);

static int array[2] = {1, 2};

int main()  {
  int val = sum(array, 2);
  return val;
}
```

**sum.c**

```
int sum(int a[], int n) {
  int i, s = 0;
  for (i = 0; i < n; i++)
    s += a[i];
  return s;
}
```

Use **`static`** to make a definition file-local

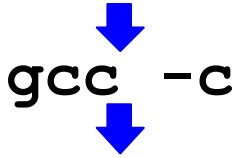
# Non-Exported Definitions

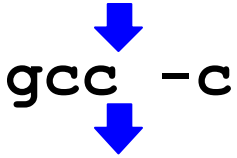
**main.c**

```
int sum(int a[], int n);

static int array[2] = {1, 2};

int main()  {
  int val = sum(array, 2);
  return val;
}
```

**sum.c**

```
int sum(int a[], int n) {
  int i, s = 0;
  for (i = 0; i < n; i++)
    s += a[i];
  return s;
}
```




**main.o**

defines **main**

uses **sum**

**sum.o**

defines **sum**

# Non-Exported Definitions

**main.c**

```
int sum(int a[], int n);

static int array[2] = {1, 2};

int main()  {
  int val = sum(array, 2);
  return val;
}
```

**sum.c**

```
static double array[3];

int sum(int a[], int n) {
  int i, s = 0;
  for (i = 0; i < n; i++)
    s += a[i];
  return s;
}
```

File-local means that two **`arrays`** don't conflict

Use **`static`** for all local data and functions

# Nested `static`

```
#include <stdio.h>


int sum(int a, int b) {
   static int counter = 0;
   printf("called %d times\n", ++counter);
   return a+b;
}

int main() {
   return sum(1, 2) + sum(3, 4) + sum(5, 6) + sum(7, 8);
}
```

Copy

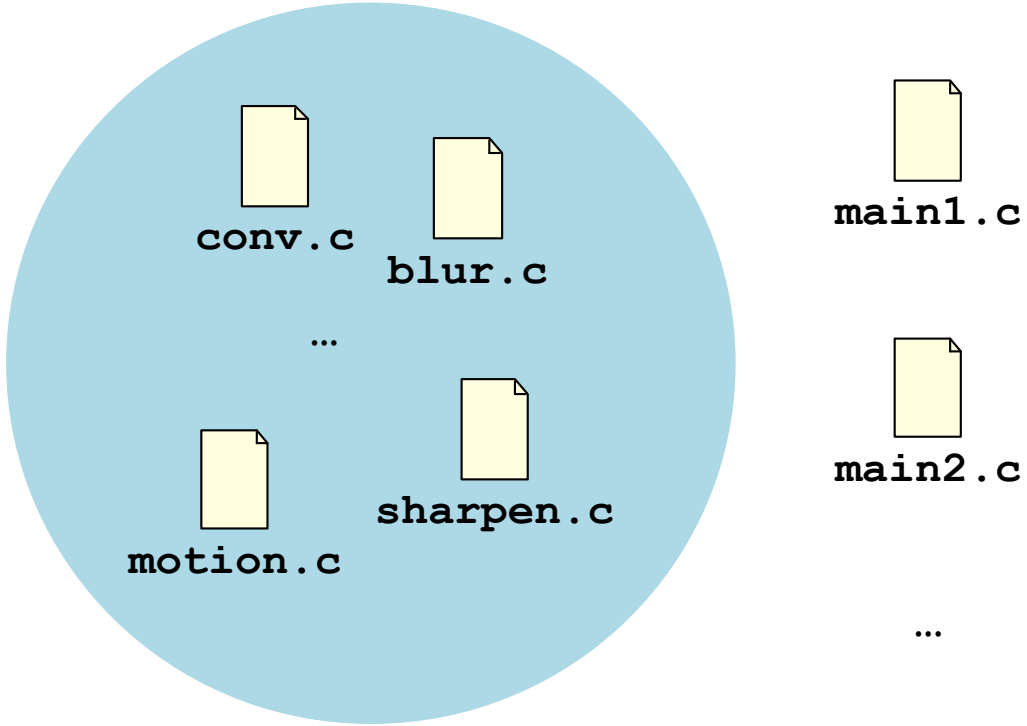
Nested **`static`** restricts a "global" to local block

# Nested `static`

```
#include <stdio.h>
static int counter_4279 = 0;

int sum(int a, int b) {

   printf("called %d times\n", ++counter_4279);
   return a+b;
}

int main() {
   return sum(1, 2) + sum(3, 4) + sum(5, 6) + sum(7, 8);
}
```

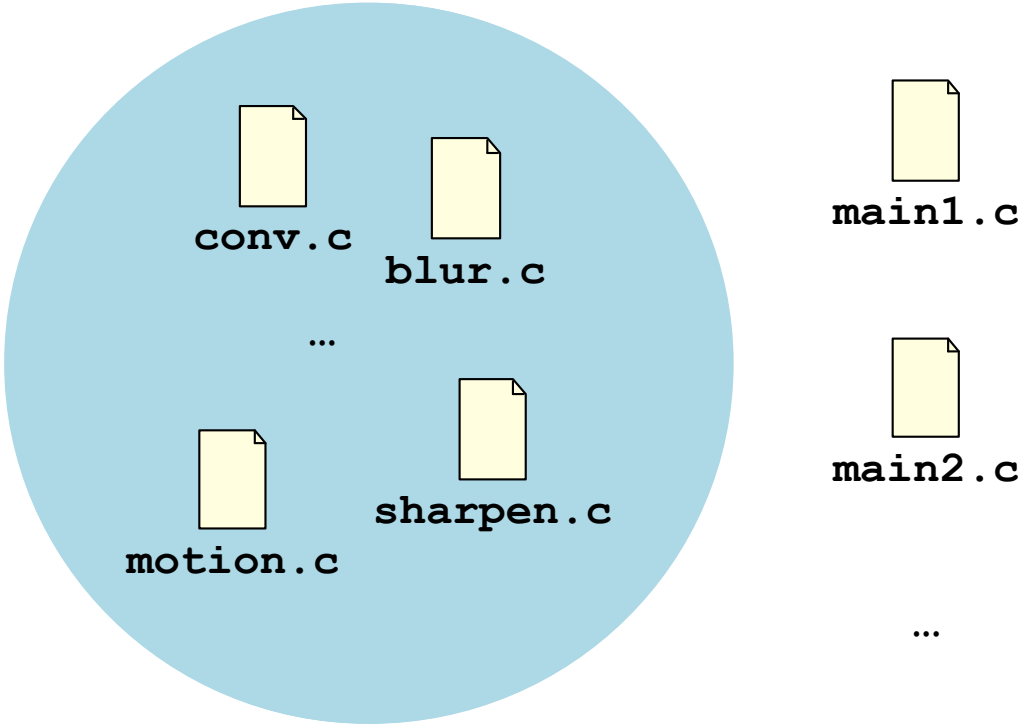
Copy

# Libraries



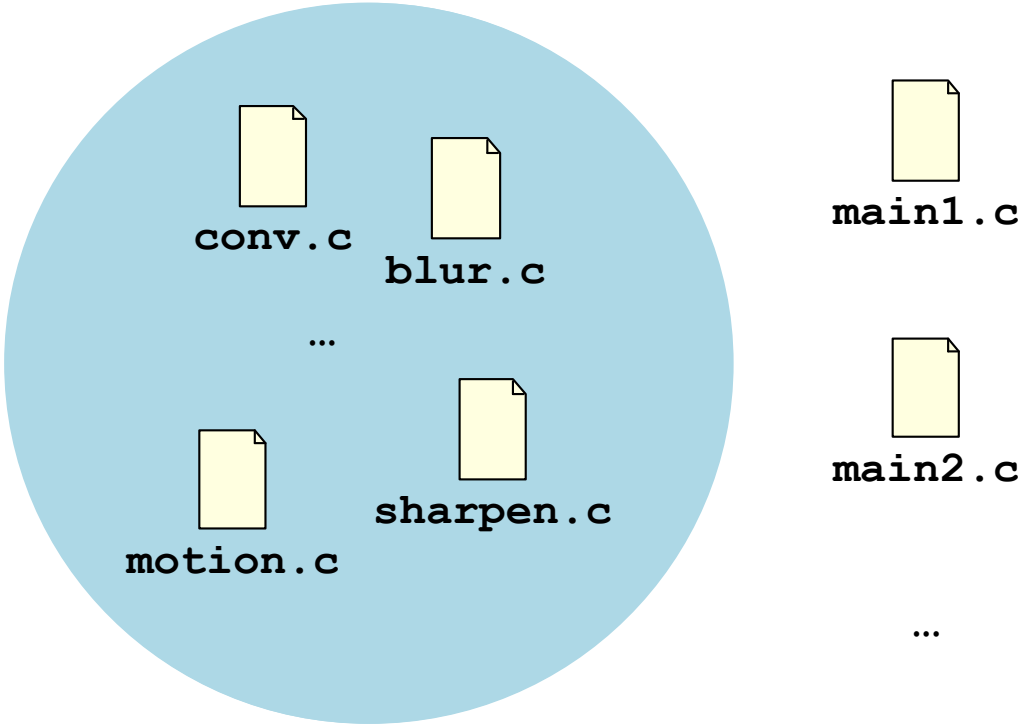

```
gcc -c conv.o
gcc -c blur.o
...

gcc -o main1 main1.o conv.o blur.o motion.o sharpen.o ...
```

# Libraries




```
gcc -c conv.o
gcc -c blur.o
...

ar -ruv image.a conv.o blur.o motion.o sharpen.o ...
gcc -o main1 main.c image.a
```

# Libraries




```
gcc -c conv.o
g
.
ar -ruv image.a conv.o blur.o motion.o sharpen.o ...
gcc -o main1 main.c image.a
```

create a ***library*** (specifically: a ***static library***)

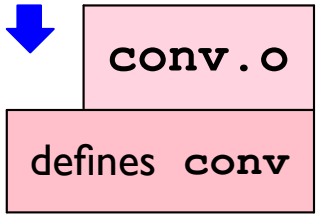
# Libraries

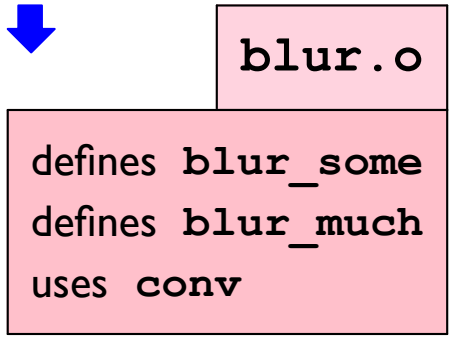
**conv.c**

```
int conv(img_t *i, mat_t *op) {
  ...
}
```

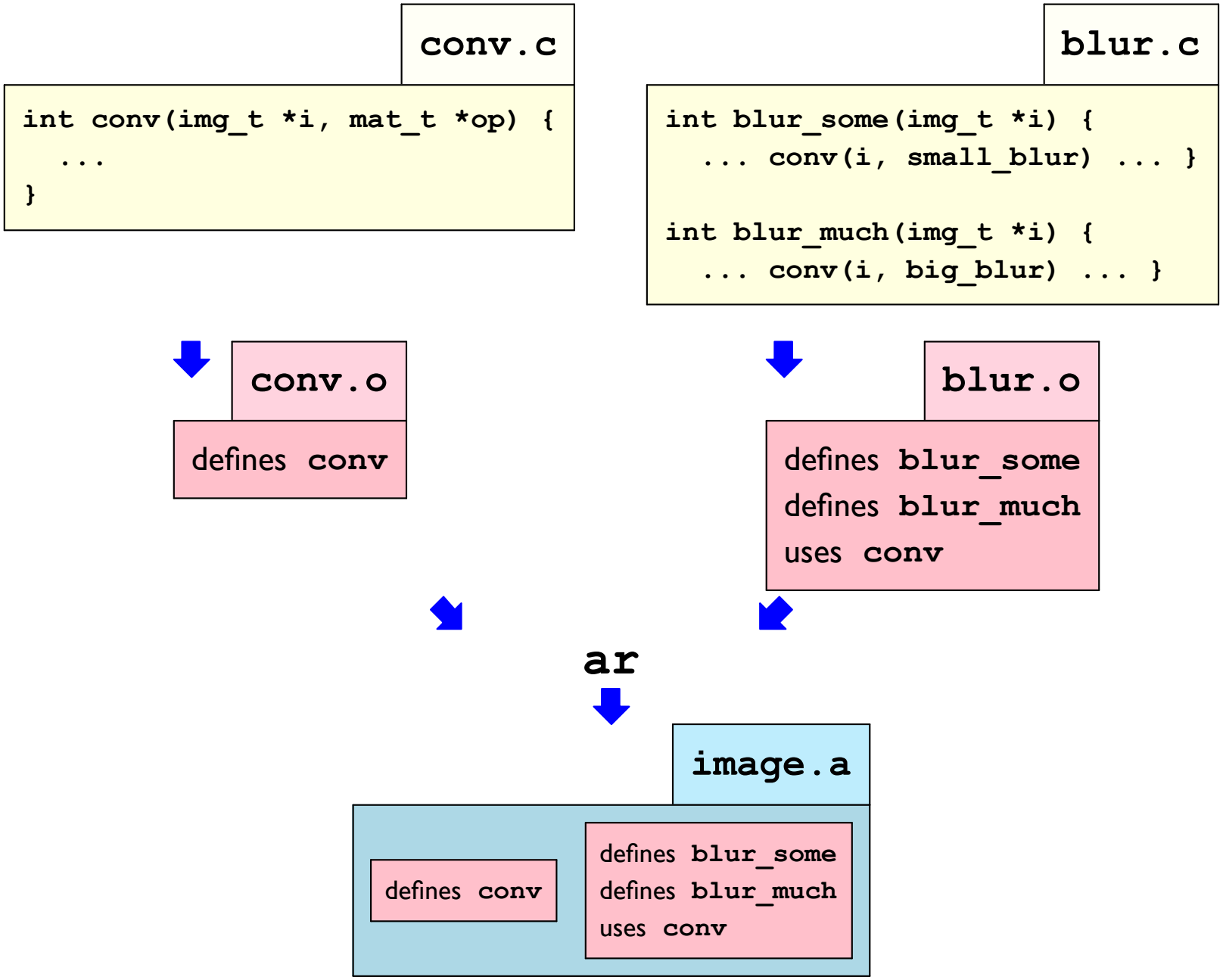
**blur.c**

```
int blur_some(img_t *i) {
  ... conv(i, small_blur) ... }

int blur_much(img_t *i) {
  ... conv(i, big_blur) ... }
```

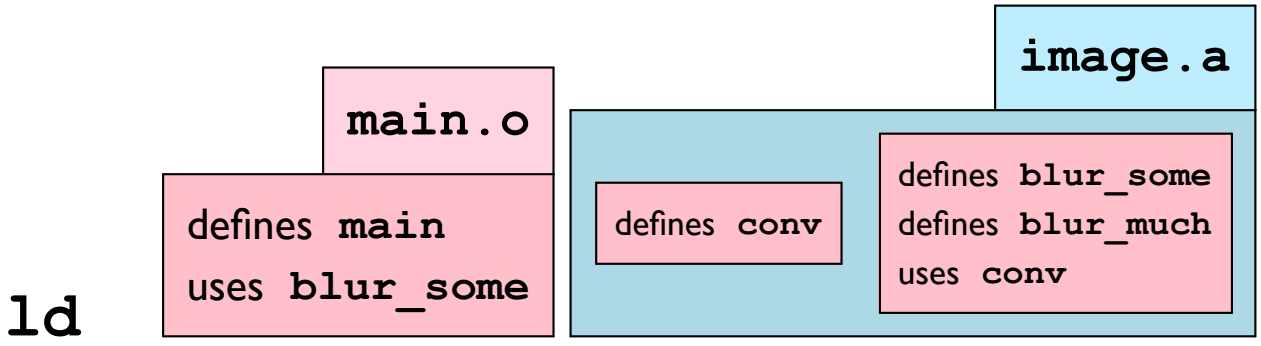


# Libraries

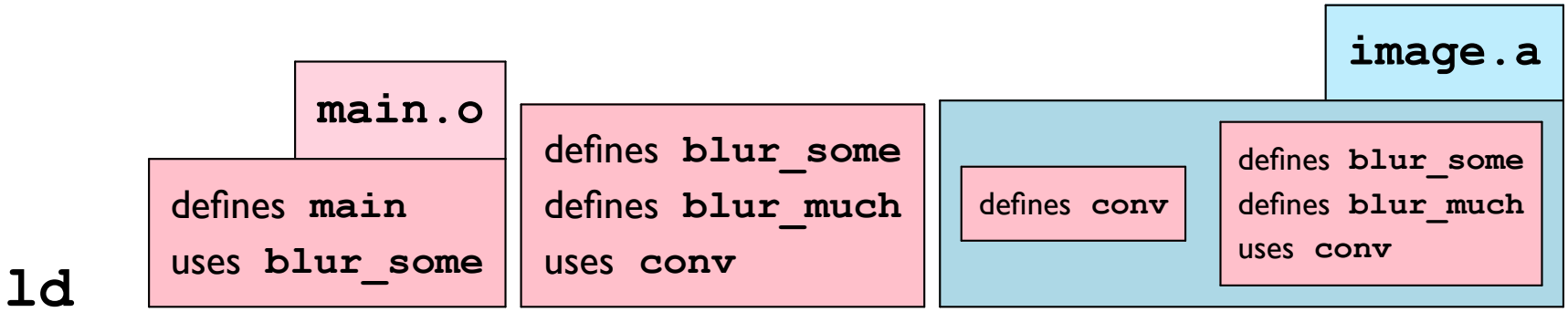

conv.c
int conv(img_t *i, mat_t *op) {
...
}
blur.c
int blur_some(img_t *i) {
... conv(i, small_blur) ... }
int blur_much(img_t *i) {
... conv(i, big_blur) ... }
conv.o
defines conv
blur.o
defines blur_some
defines blur_much
uses conv
ar
image.a
defines conv
defines blur_some
defines blur_much
uses conv

# Linking with Static Libraries

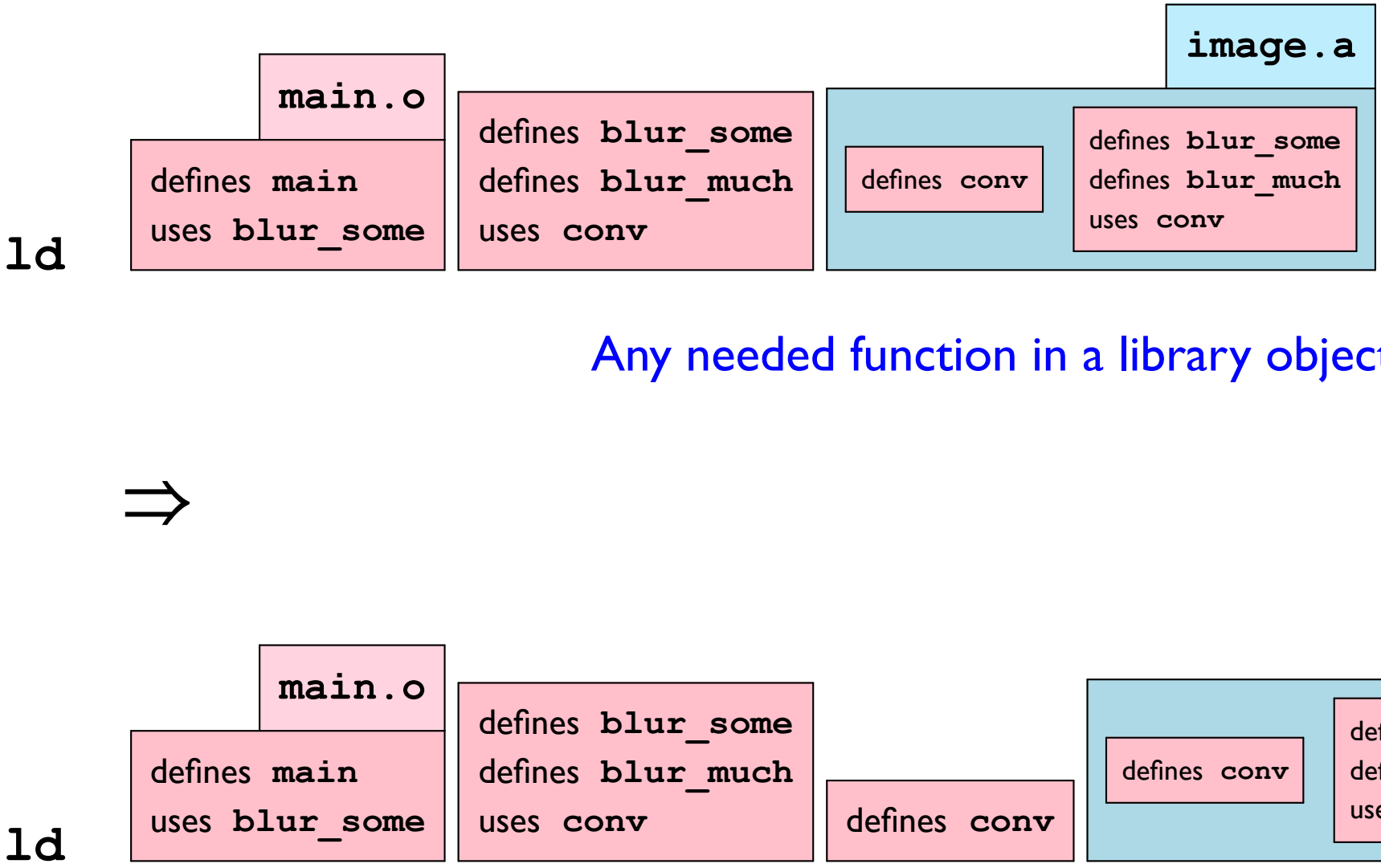



Any needed function in a library object?

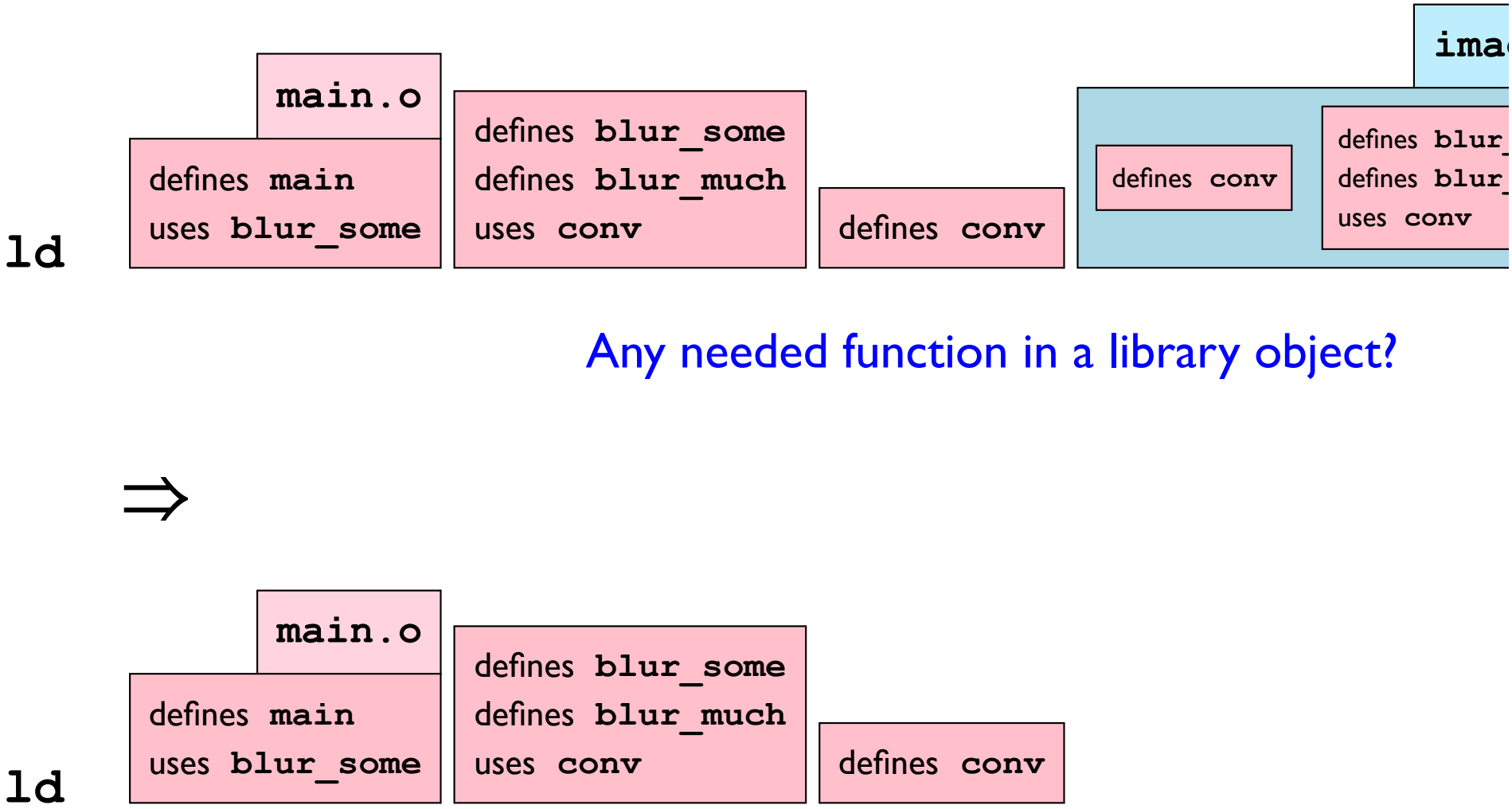
⇒



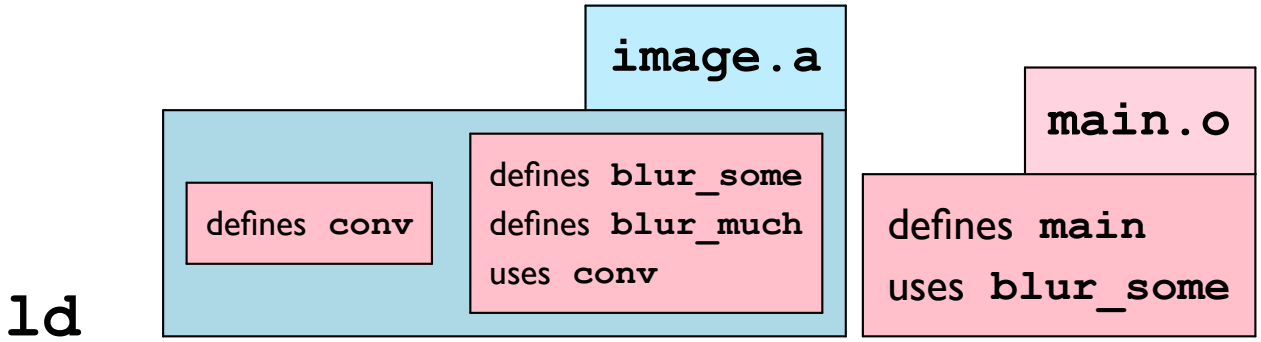
# Linking with Static Libraries



# Linking with Static Libraries

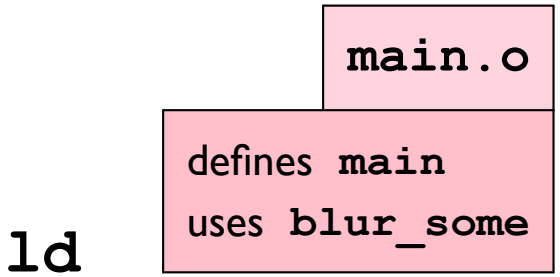


# Static Library Order



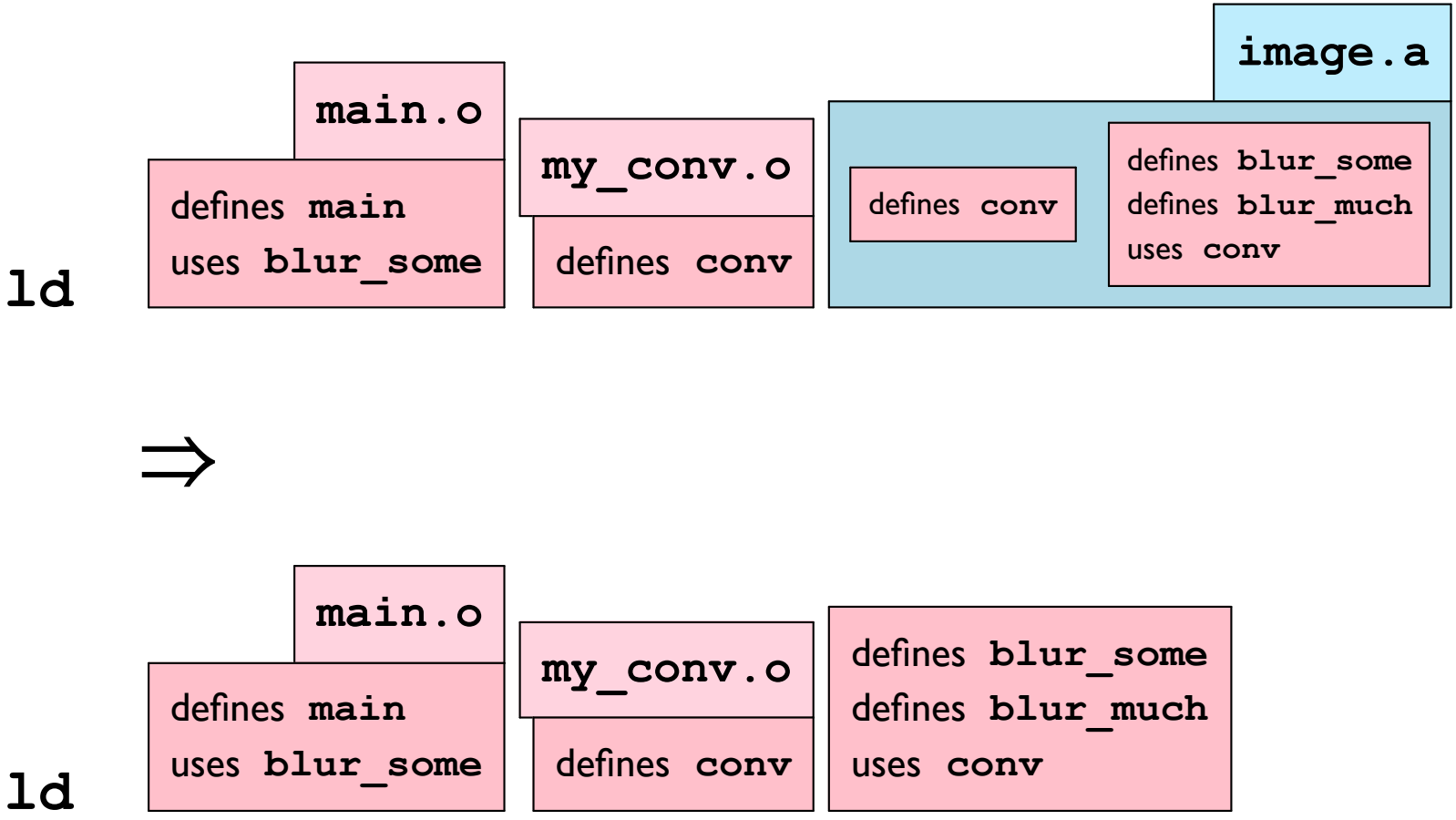

Any needed function in a library object?

⇒



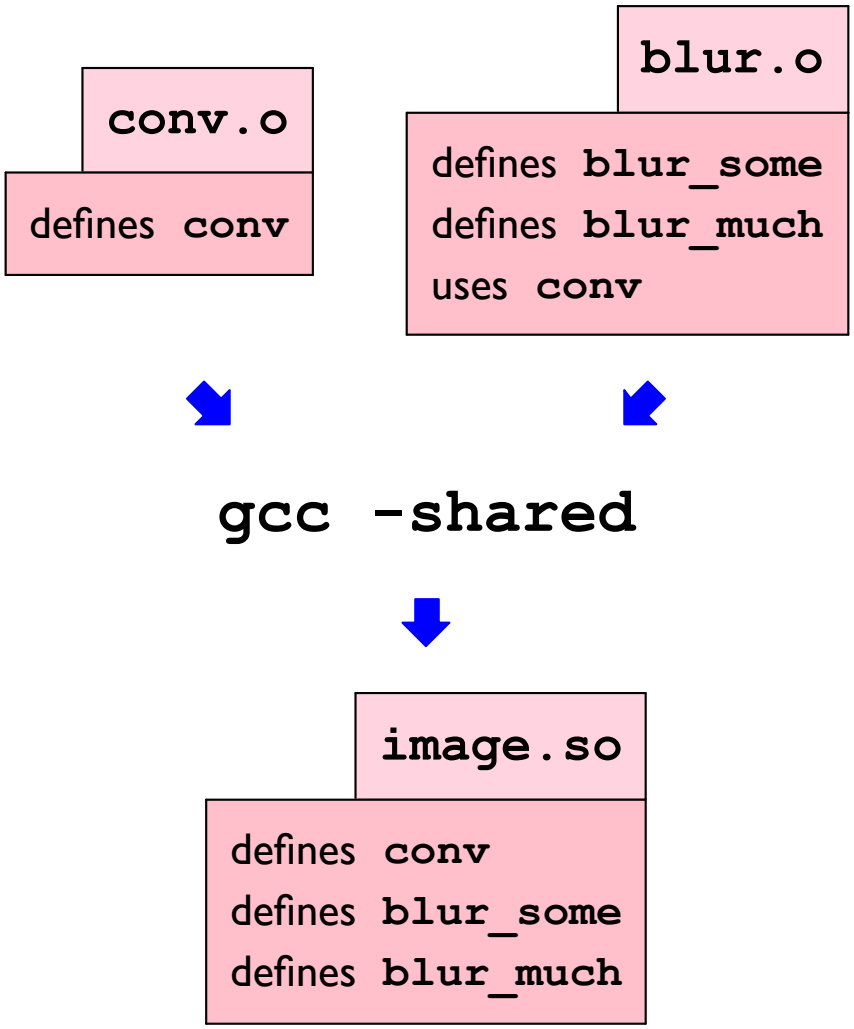

*error: no definition of* **`blur_some`**

# Exploiting Order to Replace Functions



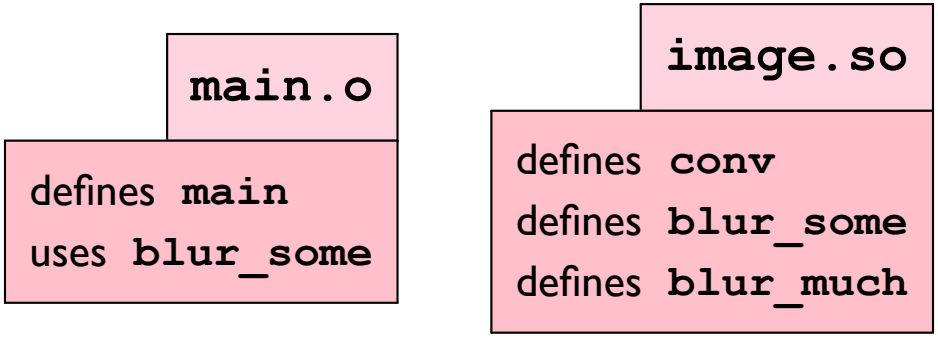
# Creating a Shared Library

A ***shared library*** is more like an object than a static library



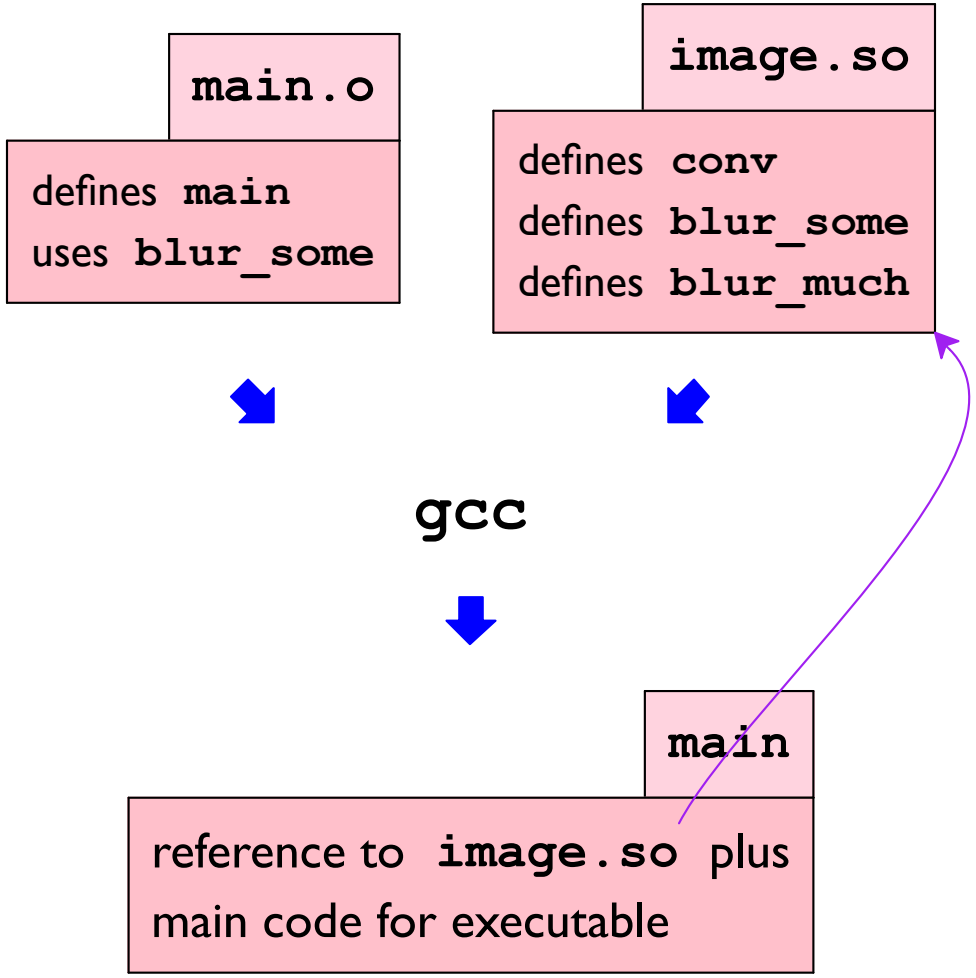
# Using a Shared Library

Linking with a shared library retains a reference to the library



# Using a Shared Library

Linking with a shared library retains a reference to the library



# Linking to Installed Libraries

Supplying

**`-l`*`name`***

to **`gcc`** is equivalent to supplying

***`/path/to/`*`lib`*`name``*`.so`**

or

***`/path/to/`*`lib`*`name``*`.a`**

**`-lc  =  /lib64/libc.so`**

# Linking to Installed Libraries

Supplying

**`-l`*`name`***

to **`gcc`** is equivalent to supplying

***`/path/to/`*`lib`*`name``*`.so`**

or

***`/path/to/`*`lib`*`name``*`.a`**

**`-lssl3`  =  `*`/usr/lib64/libssl3.so`**

# Linking to Installed Libraries

Supplying

**`-l`*`name`***

to **`gcc`** is equivalent to supplying

***`/path/to/`*`lib`*`name``*`.so`**

or

***`/path/to/`*`lib`*`name``*`.a`**

**`-lbsd-compat = /usr/lib64/libbsd-compat.a`**

# Linking to Installed Libraries

Supplying

**`-l`*`name`***

to **`gcc`** is equivalent to supplying

***`/path/to/`*`lib`*`name``*`.so`**

or

***`/path/to/`*`lib`*`name``*`.a`**

**`gcc`** adds **`-lc`** automatically, which is why calling **`printf`** works

# Using Shared Libraries via Reflection

**`dlopen`** and **`dlsym`** load and access a shared library at run time

```
void *dlopen(const char *filename, int flag);
```

Returns a handle for use with **`dlsym`**

```
void *dlsym(void *handle, const char *symbol);
```

Returns function or variable address for **`symbol`**

# Case Study: Library Interpositioning

***Interposition*** causes existing calls of some function **`f`** to be redirected to an alternative implementation, **`alt_f`**

Example: interpose on **`malloc`** and **`free`** to record a trace

**`int.c`**

```
#include <stdio.h>
#include <stdlib.h>

int main() {
  int *p = malloc(32);
  free(p);
  return 0;
}
```

Copy

# Case Study: Library Interpositioning

***Interposition*** causes existing calls of some function **`f`** to be redirected to an alternative implementation, **`alt_f`**

Example: interpose on **`malloc`** and **`free`** to record a trace

**`int.c`**

```
#include <stdio.h>
#include <stdlib.h>

int main() {
  int *p = malloc(32);
  free(p);
  return 0;
}
```

Copy

Three approaches:

- Compile time via **`cpp`**
- Link time via **`ld`**
- Run time via **`dlsym`**

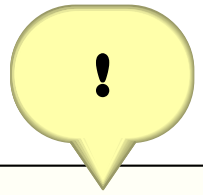
# Compile-Time Interposition

mymalloc.c

```
#include <stdio.h>
#include <stdlib.h>

void *mymalloc(size_t size) {
  void *ptr = malloc(size);
  printf("malloc(%d)=%p\n",
         (int)size, ptr);
  return ptr;
}

void myfree(void *ptr) {
  free(ptr);
  printf("free(%p)\n", ptr);
}
```

Copy

# Compile-Time Interposition




**mymalloc.c**

```
#include <stdio.h>
#include <stdlib.h>

void *mymalloc(size_t size) {
  void *ptr = malloc(size);
  printf("malloc(%d)=%p\n",
         (int)size, ptr);
  return ptr;
}

void myfree(void *ptr) {
  free(ptr);
  printf("free(%p)\n", ptr);
}
```

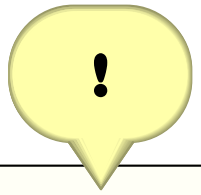
Copy

**stdlib.h**

```
#define malloc(sz) mymalloc(sz)
#define free(ptr)  myfree(ptr)

void *mymalloc(size_t sz);
void myfree(void *ptr);
```

Copy

# Compile-Time Interposition




**mymalloc.c**

```
#include <stdio.h>
#include <stdlib.h>

void *mymalloc(size_t size) {
  void *ptr = malloc(size);
  printf("malloc(%d)=%p\n",
         (int)size, ptr);
  return ptr;
}

void myfree(void *ptr) {
  free(ptr);
  printf("free(%p)\n", ptr);
}
```

Copy

**stdlib.h**

```
#define malloc(sz) mymalloc(sz)
#define free(ptr)  myfree(ptr)

void *mymalloc(size_t sz);
void myfree(void *ptr);
```

Copy

```
gcc -c mymalloc.c

gcc -I. int.c mymalloc.o
```

# Compile-Time Interposition

**mymalloc.c**

```
#include <stdio.h>
#include <stdlib.h>

void *mymalloc(size_t size) {
  void *ptr = malloc(size);
  printf("malloc(%d)=%p\n",
         (int)size, ptr);
  return ptr;
}

void myfree(void *ptr) {
  free(ptr);
  printf("free(%p)\n", p
}
```

Copy

**stdlib.h**

```
#define malloc(sz) mymalloc(sz)
#define free(ptr)  myfree(ptr)

void *mymalloc(size_t sz);
void myfree(void *ptr);
```

Copy

```
gcc -c mymalloc.c

gcc -I. int.c mymalloc.o
```

Causes `gcc` to find replacement `stdlib.h`

# Link-Time Interposition

wmalloc.c

```
#include <stdio.h>

void *__real_malloc(size_t size);
void __real_free(void *ptr);

void *__wrap_malloc(size_t size) {
  void *ptr = __real_malloc(size);
  printf("malloc(%d)=%p\n",
         (int)size, ptr);
  return ptr;
}

void __wrap_free(void *ptr) {
  __real_free(ptr);
  printf("free(%p)\n", ptr);
}
```

Copy

# Link-Time Interposition

wmalloc.c

```
#include <stdio.h>

void *__real_malloc(size_t size);
void __real_free(void *ptr);

void *__wrap_malloc(size_t size) {
  void *ptr = __real_malloc(size);
  printf("malloc(%d)=%p\n",
         (int)size, ptr);
  return ptr;
}

void __wrap_free(void *ptr) {
  __real_free(ptr);
  printf("free(%p)\n", ptr);
}
```

Copy

```
gcc -c wmalloc.c

gcc -c int.c

gcc -Wl,--wrap,malloc
    -Wl,--wrap,free
    int.o wmalloc.o
```

# Link-Time Interposition

**wmalloc.c**

```
#include <stdio.h>

void *__real_malloc(size_t size);
void __real_free(void *ptr);

void *__wrap_malloc(size_t size) {
  void *ptr = __real_malloc(size);
  printf("malloc(%d)=%p\n",
         (int)size, ptr);
  return ptr;
}

void __wrap_free(void *ptr) {
  __real_free(ptr);
  printf("free(%p)\n", ptr);
}
```

Copy

```
gcc -c wmalloc.c

gcc -c int.c

gcc -Wl,--wrap,malloc
    -Wl,--wrap,free
    int.o wmalloc.o
```

**-Wl** ⇒ pass flag to **ld**

# Link-Time Interposition

wmalloc.c

```
#include <stdio.h>

void *__real_malloc(size_t size);
void __real_free(void *ptr);

void *__wrap_malloc(size_t size) {
  void *ptr = __real_malloc(size);
  printf("malloc(%d)=%p\n",
         (int)size, ptr);
  return ptr;
}

void __wrap_free(void *ptr) {
  __real_free(ptr);
  printf("free(%p)\n", ptr);
}
```

Copy

```
gcc -c wmalloc.c

gcc -c int.c

gcc -Wl,--wrap,malloc
    -Wl,--wrap,free
    int.o wmalloc.o
```

**--wrap,free:**

**free → __wrap_free**

**__real_free → free**

# Link+Run-Time Interposition

rmalloc.c

```
#define _GNU_SOURCE
#include <stdio.h>
#include <dlfcn.h>

void *malloc(size_t size) {
  void *(*mallocp)(size_t size) = dlsym(RTLD_NEXT, "malloc");
  char *ptr = mallocp(size);
  printf("malloc(%d) = %p\n", (int)size, ptr);
  return ptr;
}

void free(void *ptr) {
  void (*freep)(void *) = dlsym(RTLD_NEXT, "free");
  if (!ptr) return;
  freep(ptr);
  printf("free(%p)\n", ptr);
}
```

Copy

**gcc int.c rmalloc.c -ldl**

# Run-Time Interposition

Since **`libc.so`** is dynamically linked:

```
gcc -o int int.c
gcc -o rmalloc.so -shared -fpic rmalloc.c -ldl
env LD_PRELOAD=./rmalloc.so ./int
```

Setting **`LD_PRELOAD`** to **`./rmalloc.so`** causes **`rmalloc.so`** to be consulted before **`libc.so`**