CSE251: System Programming

10. Linking (2)

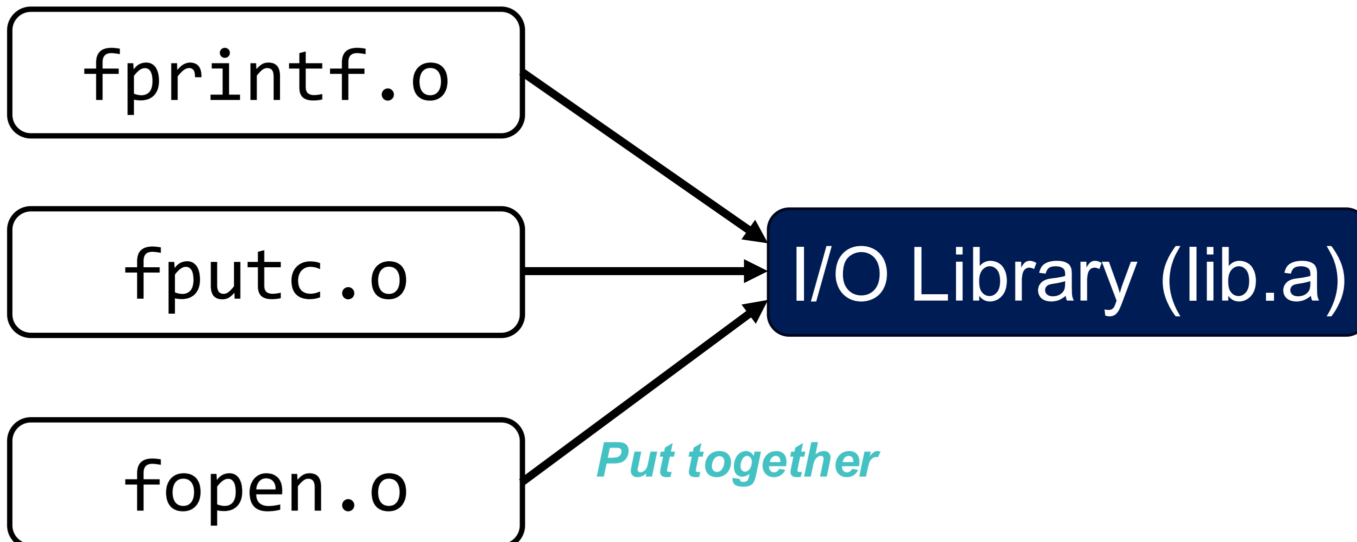
Seongil Wi

Department of Computer Science and Engineering

UNIST
ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY
2007

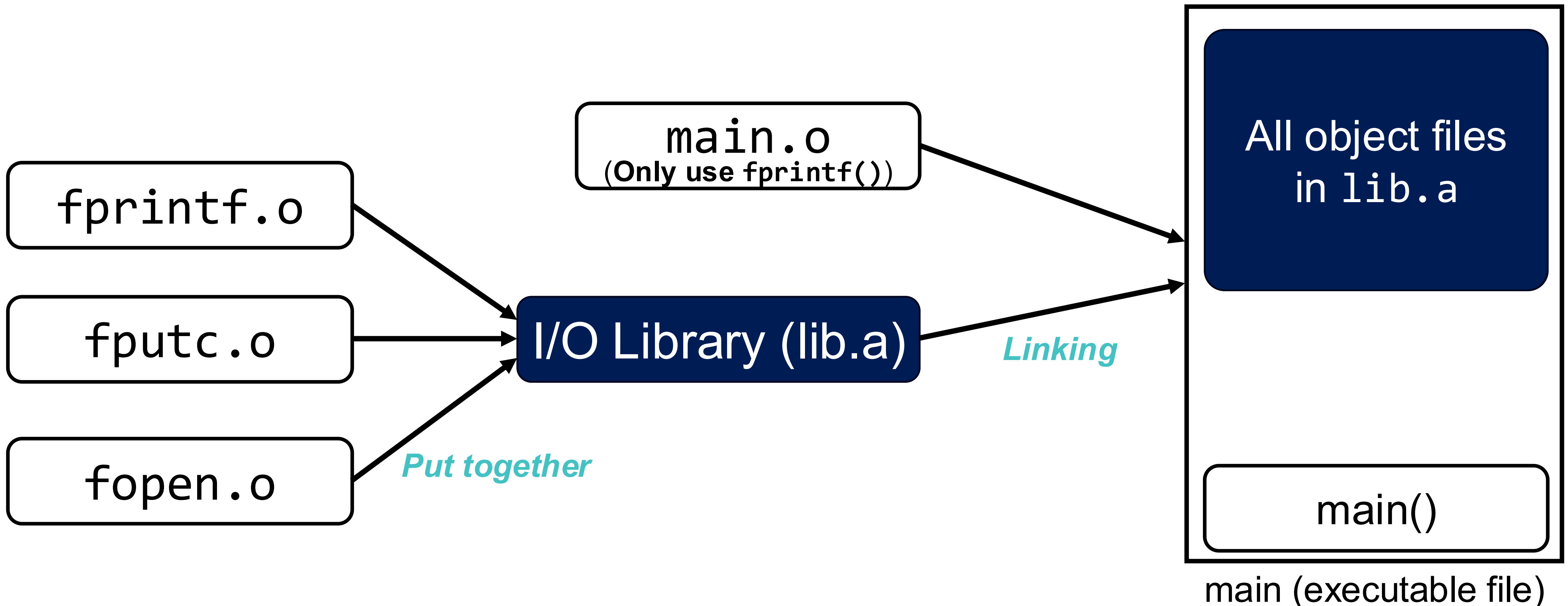
Packaging Commonly Used Functions

- **How to package functions commonly used by programmers?**
  - Math, I/O, memory management, string manipulation, etc.



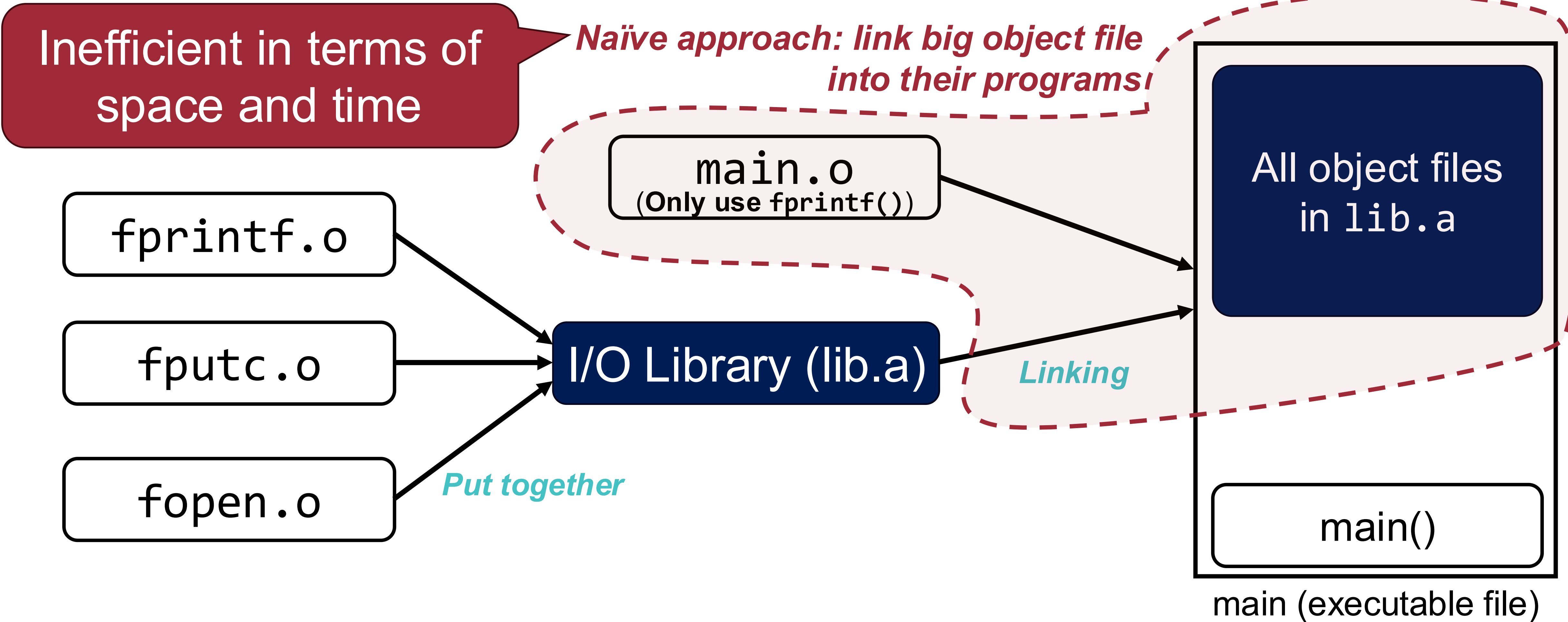
Packaging Commonly Used Functions

- **How to package functions commonly used by programmers?**
  - Math, I/O, memory management, string manipulation, etc.



Packaging Commonly Used Functions

- **How to package functions commonly used by programmers?**
  - Math, I/O, memory management, string manipulation, etc.



Packaging Commonly Used Functions

- **How to package functions commonly used by programmers?**
  - Math, I/O, memory management, string manipulation, etc.

- **Solution #1**: Static linking (static libraries, `.a` file)

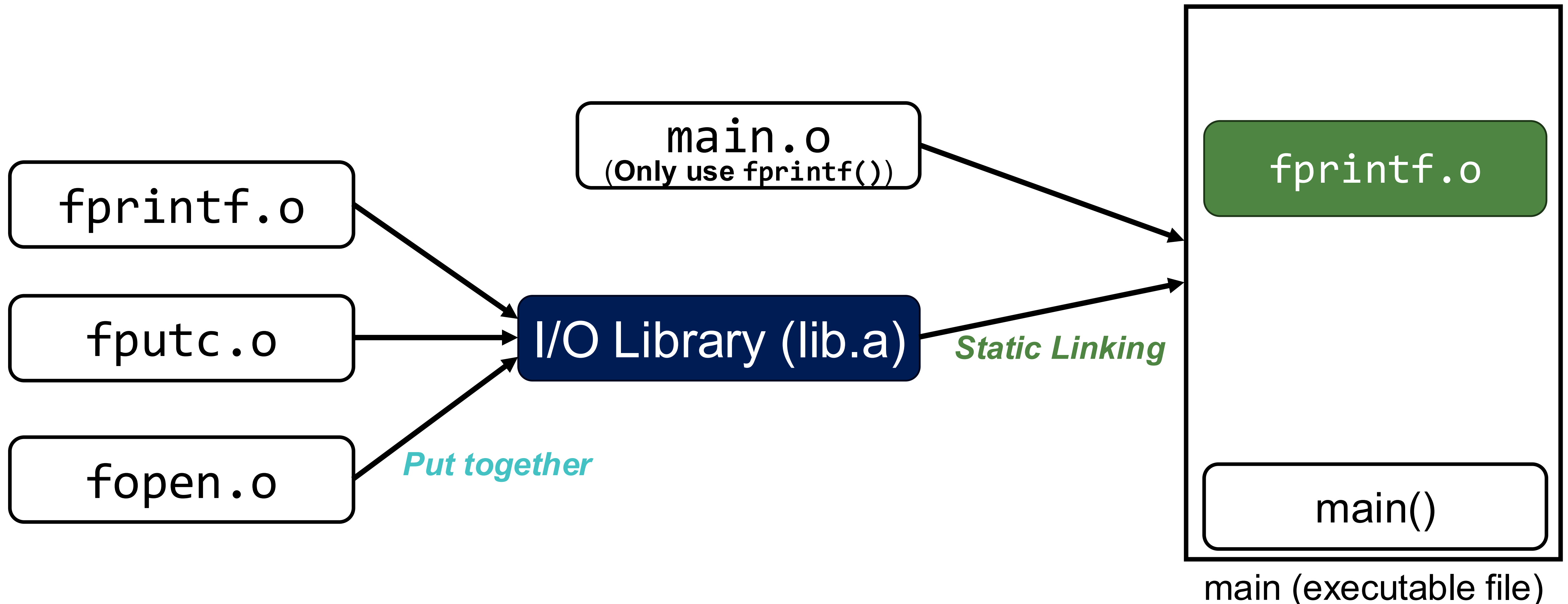
- **Solution #2**: Dynamic linking (shared libraries, `.so` file)

Static Linking

Old-fashioned Solution

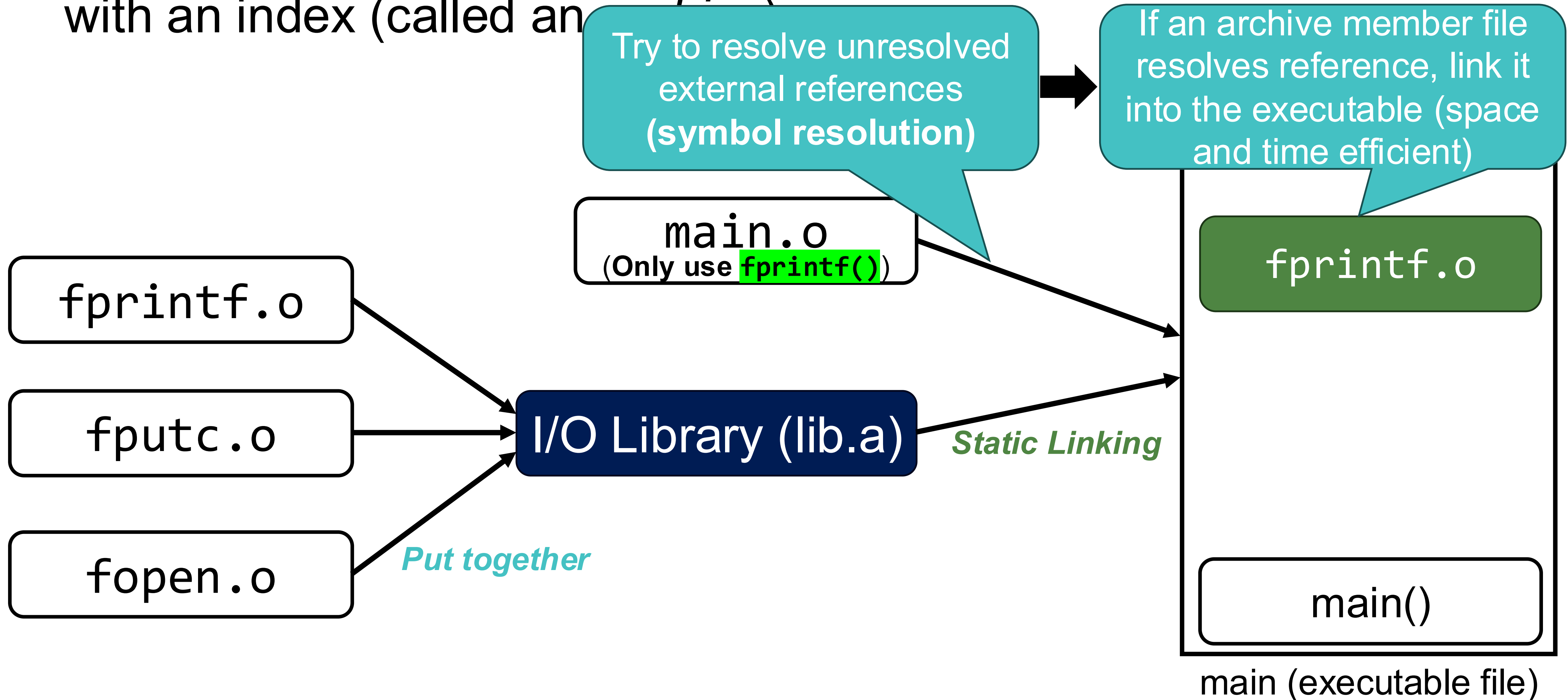
Static Linking (Old-fashioned Solution)

- Concatenate **related relocatable object files** into a single file with an index (called an *archive*)



Static Linking (Old-fashioned Solution)

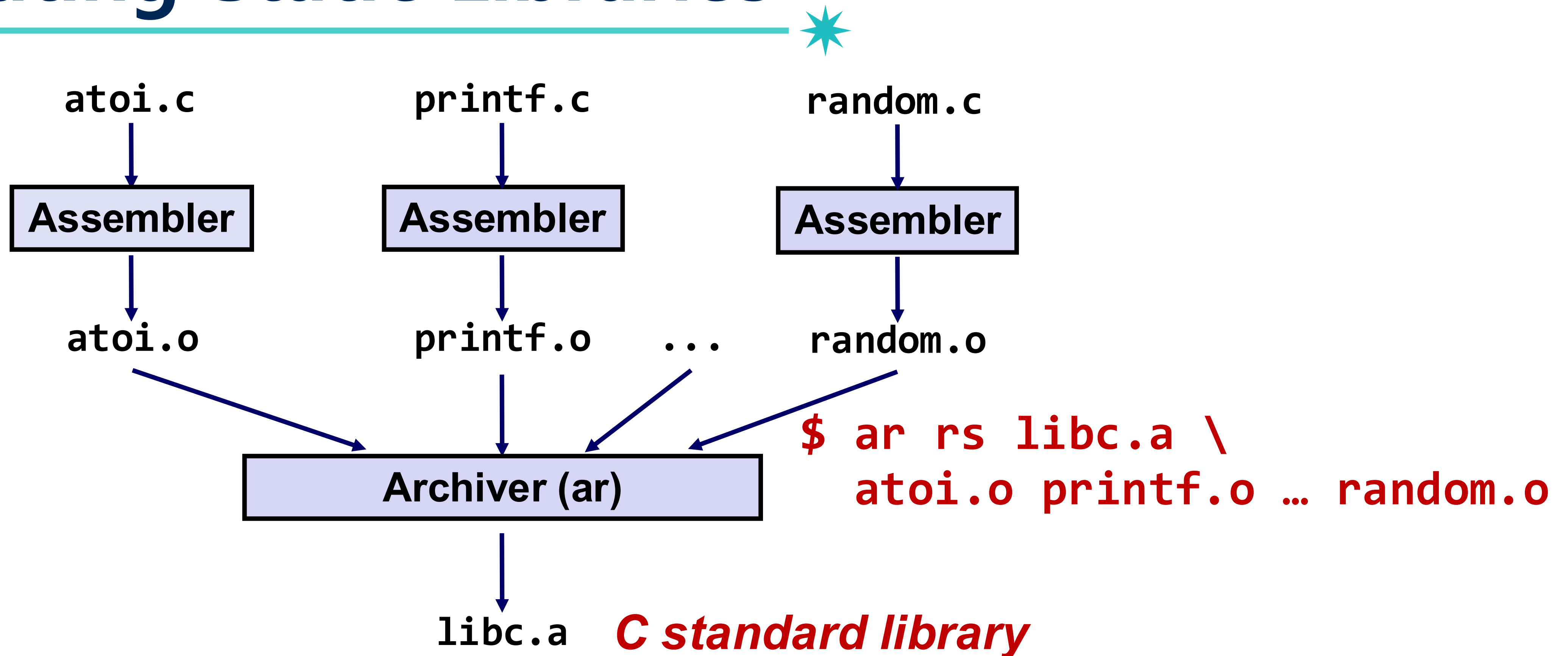
- Concatenate **related relocatable object files** into a single file with an index (called an



Static Linking (Old-fashioned Solution)

- Concatenate **related relocatable object files** into a single file with an index (called an *archive*)
  - Enhance linker so that it tries to resolve unresolved external references **by looking for the symbols** in one or more archives
  - If an archive member file resolves reference, link it into the executable

Creating Static Libraries



- Archiver allows incremental updates
- Recompile function that changes and replace .o file in archive

Commonly Used Libraries

- **`libc.a` (the C standard library)**
  - 4.6 MB archive of 1,496 object files
  - I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math
- **`libm.a` (the C math library)**
  - 2 MB archive of 444 object files
  - Floating point math (sin, cos, tan, log, exp, sqrt, …)

```
/lib/x86_64-linux-gnu$ ar -t libc.a | sort
…
fork.o
…
fprintf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
fseek.o
…
```

```
/lib/x86_64-linux-gnu$ ar -t libm-2.36.a | sort
…
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acoshl.o
e_acosl.o
e_asin.o
e_asinf.o
…
```

Linking with Static Libraries

```c
#include <stdio.h>
#include "vector.h"

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main() {
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n”,
                   z[0], z[1]);
    return 0;
}
```

main.c

libvector.a

```c
void addvec(int *x, int *y, int *z, int n) {
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];
}
```

addvec.c

```c
void multvec(int *x, int *y, int *z, int n) {
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] * y[i];
}
```

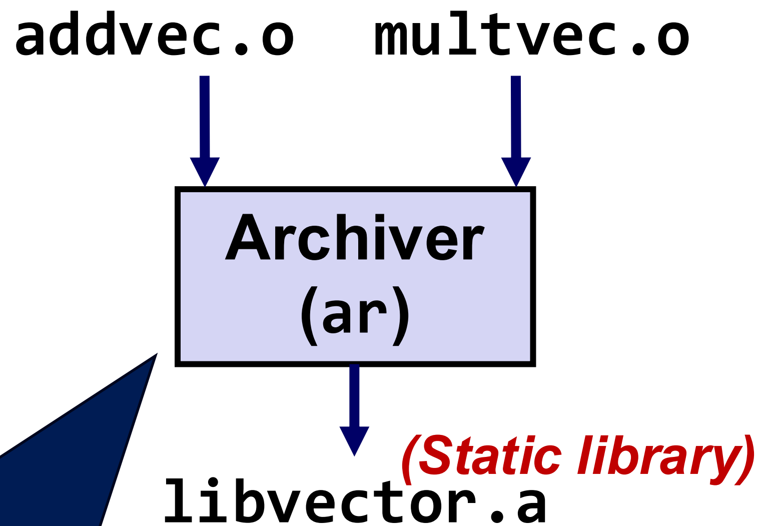
multvec.c

Linking with Static Libr

```
$ gcc -c addvec.c -o addvec.o
$ gcc -c multvec.c -o multvec.o
```

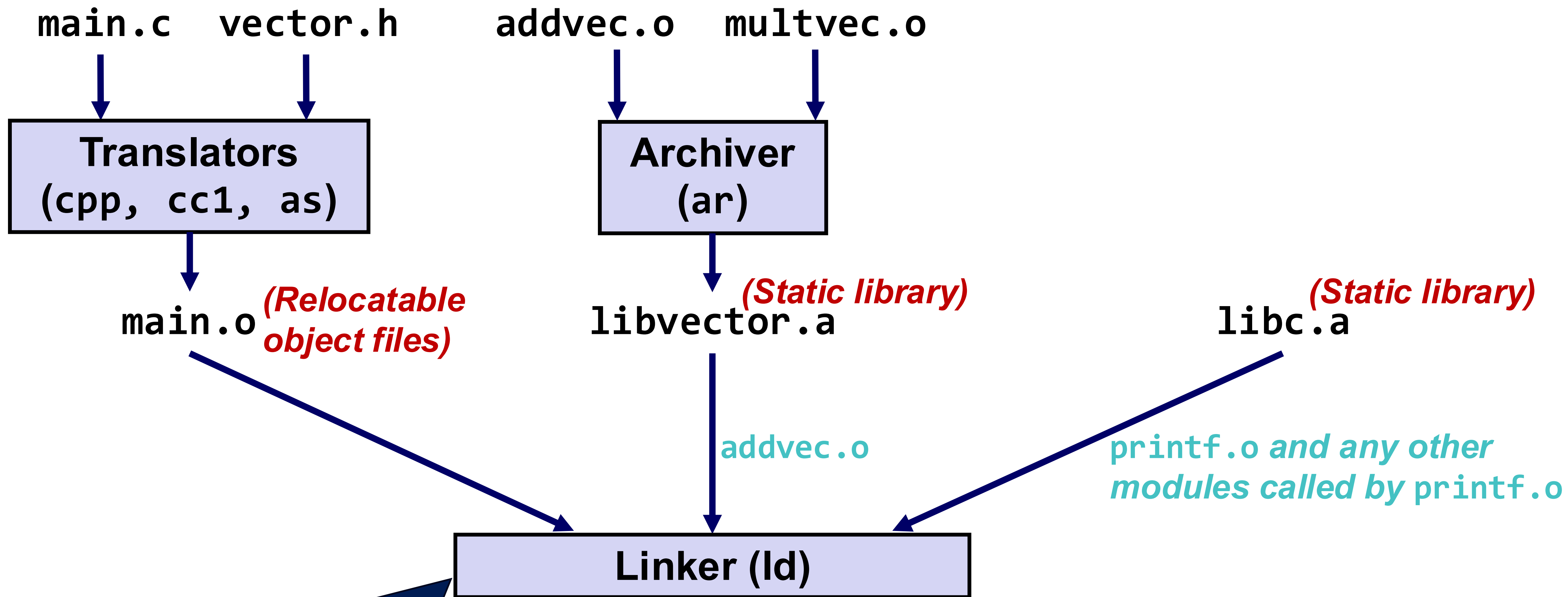
addvec.o **multvec.o**

Linking with Static Libraries



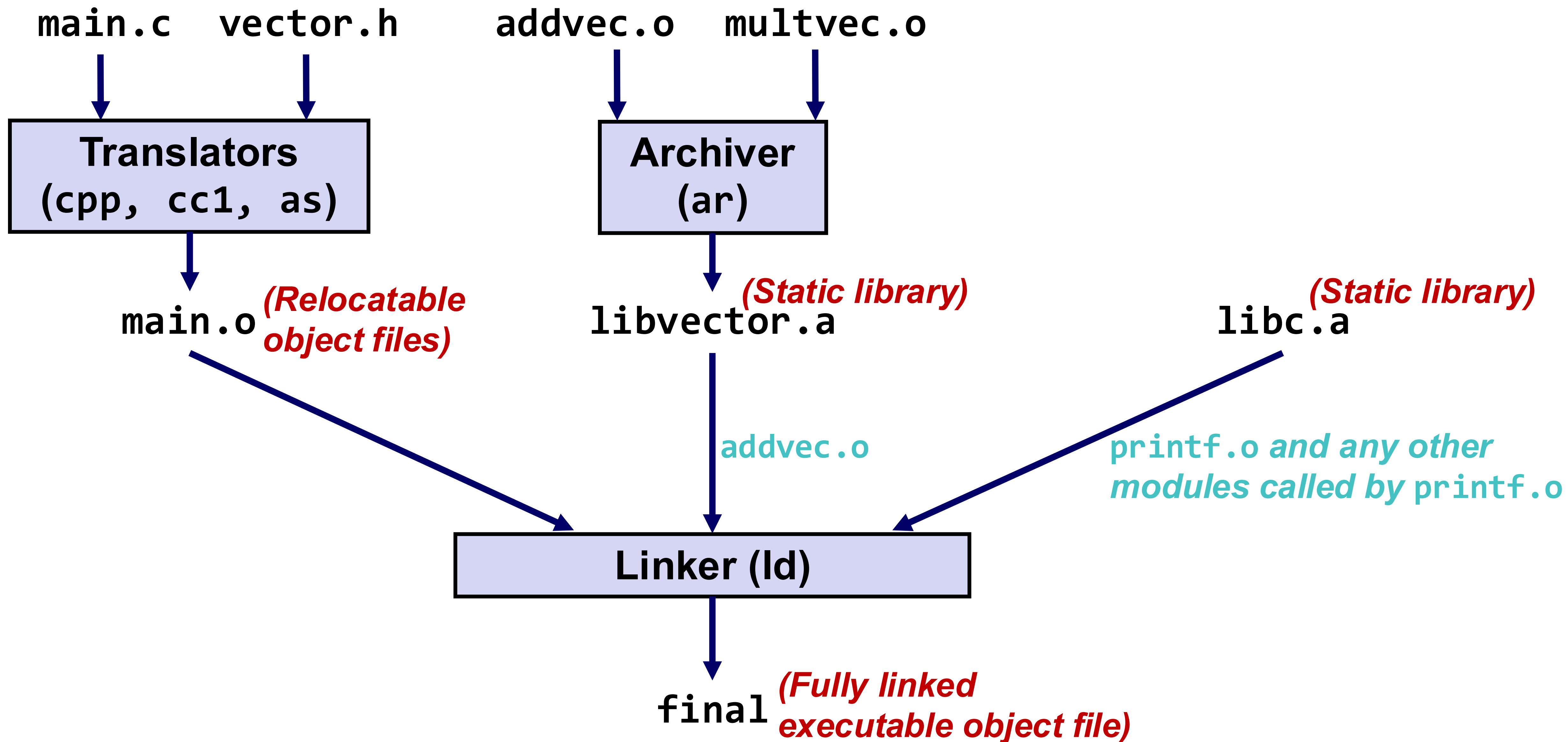

```
$ ar rs libvector.a addvec.o multvec.o
```

Linking with Static Libraries




```
$ gcc -static -o final main.o -L. -lvector
```

Linking with Static Libraries



Using Static Libraries

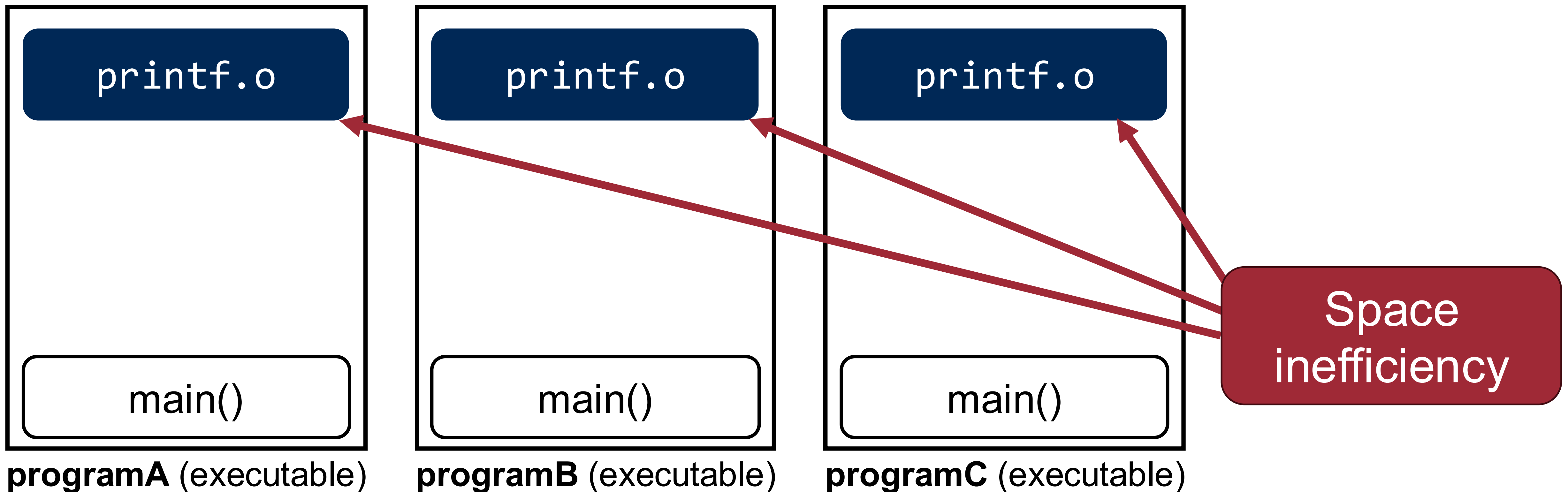
- **Linker's algorithm for resolving external references:**
  1. Scan `.o` files and `.a` files in the command line order
  2. During the scan, keep a list of the current unresolved references (symbols)
  3. As each new `.o` or `.a` file (i.e., ***obj***) is encountered, try to resolve each unresolved reference in the list against the symbols defined in ***obj***
  4. If any entries in the unresolved list at end of scan, then error

- **Problem:**
  - Command line order matters!
  - Moral: put libraries at the end of the command line

```
$ gcc -static -o final main.o -L. -lvector
$ gcc -static -o final -L. -lvector main.o
/usr/bin/ld: main.o: in function `main':
(.text+0x37): undefined reference to `addvec'
collect2: error: ld returned 1 exit status
```

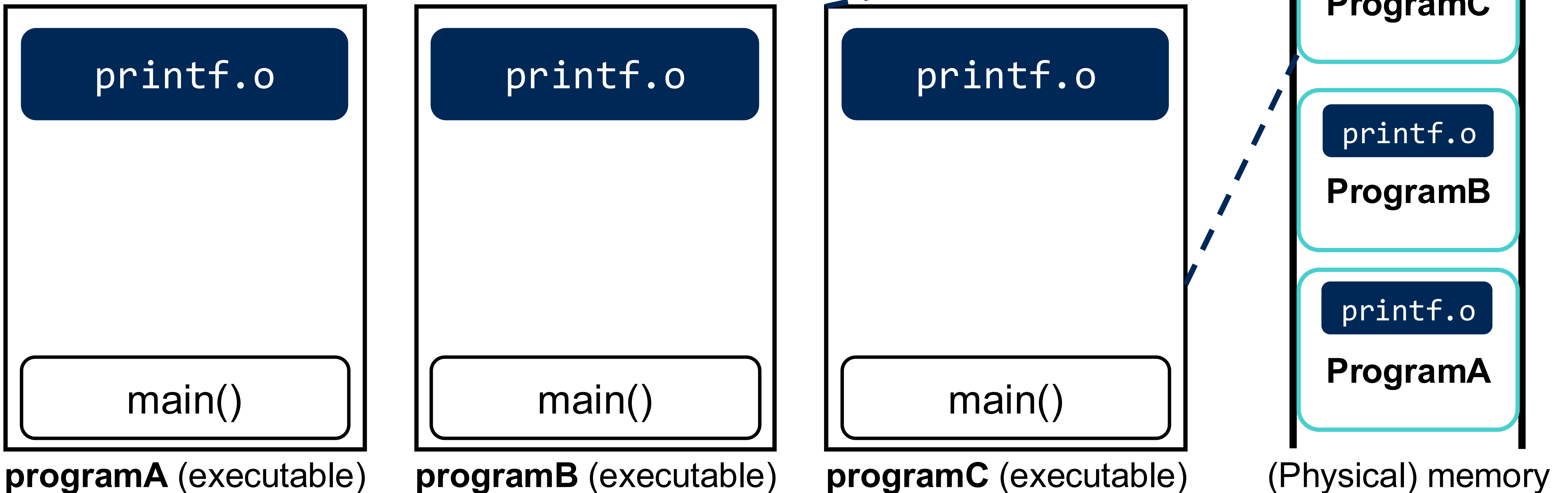
Disadvantages of Static Linking

- Static libraries have the following disadvantages:
  - Duplication in the stored executables (every function needs libc)



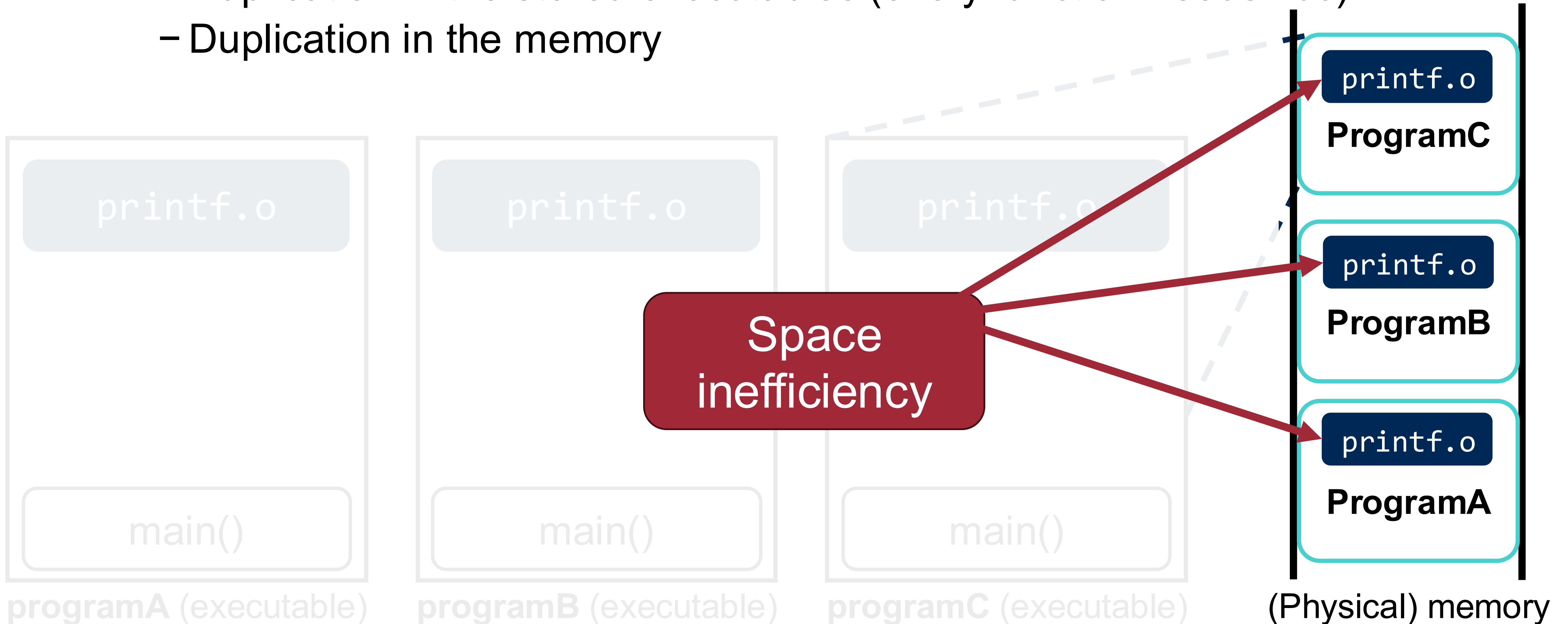
Disadvantages of Static Linking

- Static libraries have the following disadvantages:
  - Duplication in the stored executables (every function needs libc)
  - Duplication in the memory



Disadvantages of Static Linking

- Static libraries have the following disadvantages:
  - Duplication in the stored executables (every function needs libc)
  - Duplication in the memory



Disadvantages of Static Linking

- Static libraries have the following disadvantages:
  - Duplication in the stored executables (every function needs libc)
  - Duplication in the memory
  - Update of libraries require each application to explicitly relink

How to resolve these problems?

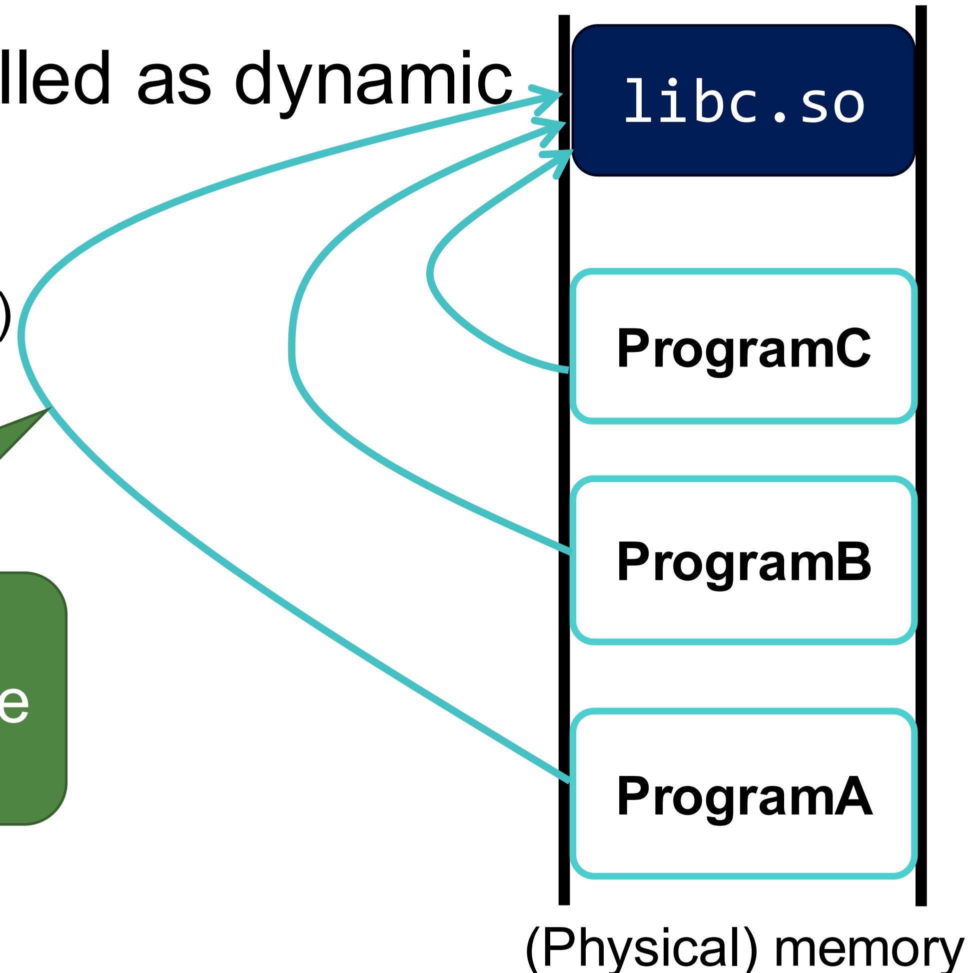
Dynamic Linking!

Dynamic Linking

Modern Solution

Dynamic Linking (Modern Solution)

- Object files are loaded and linked into an application ***dynamically***, at either load-time or run-time
- Realized with shared libraries, also called as dynamic link libraries
  - `.so` files on Linux (shared object)
  - `.dll` files on Windows (dynamically linked library)



- Save space!
- No *relinking* is needed when the library is updated!

Linking with Dynamic Libraries

```
#include <stdio.h>
#include "vector.h"

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main() {
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n”,
                  z[0], z[1]);
    return 0;
}
```

main.c

libvector.so

```
void addvec(int *x, int *y, int *z, int n) {
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];
}
```

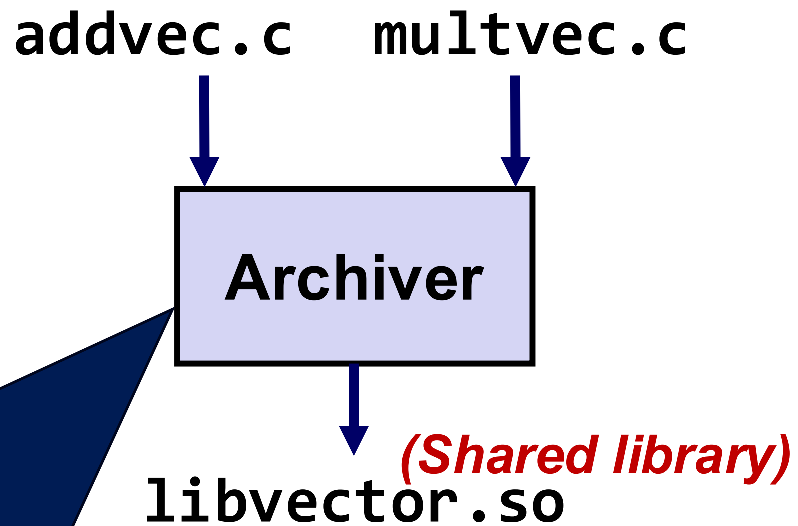
addvec.c

```
void multvec(int *x, int *y, int *z, int n) {
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] * y[i];
}
```

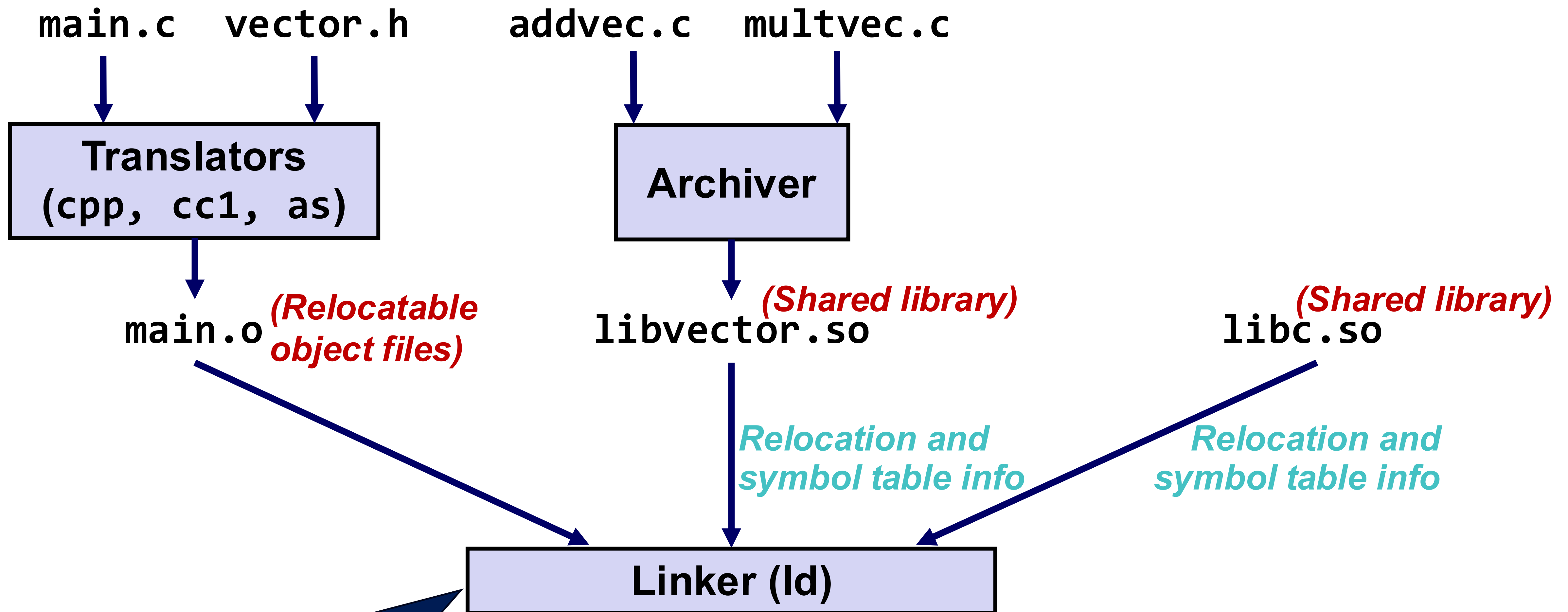
multvec.c

Linking with Shared Libraries

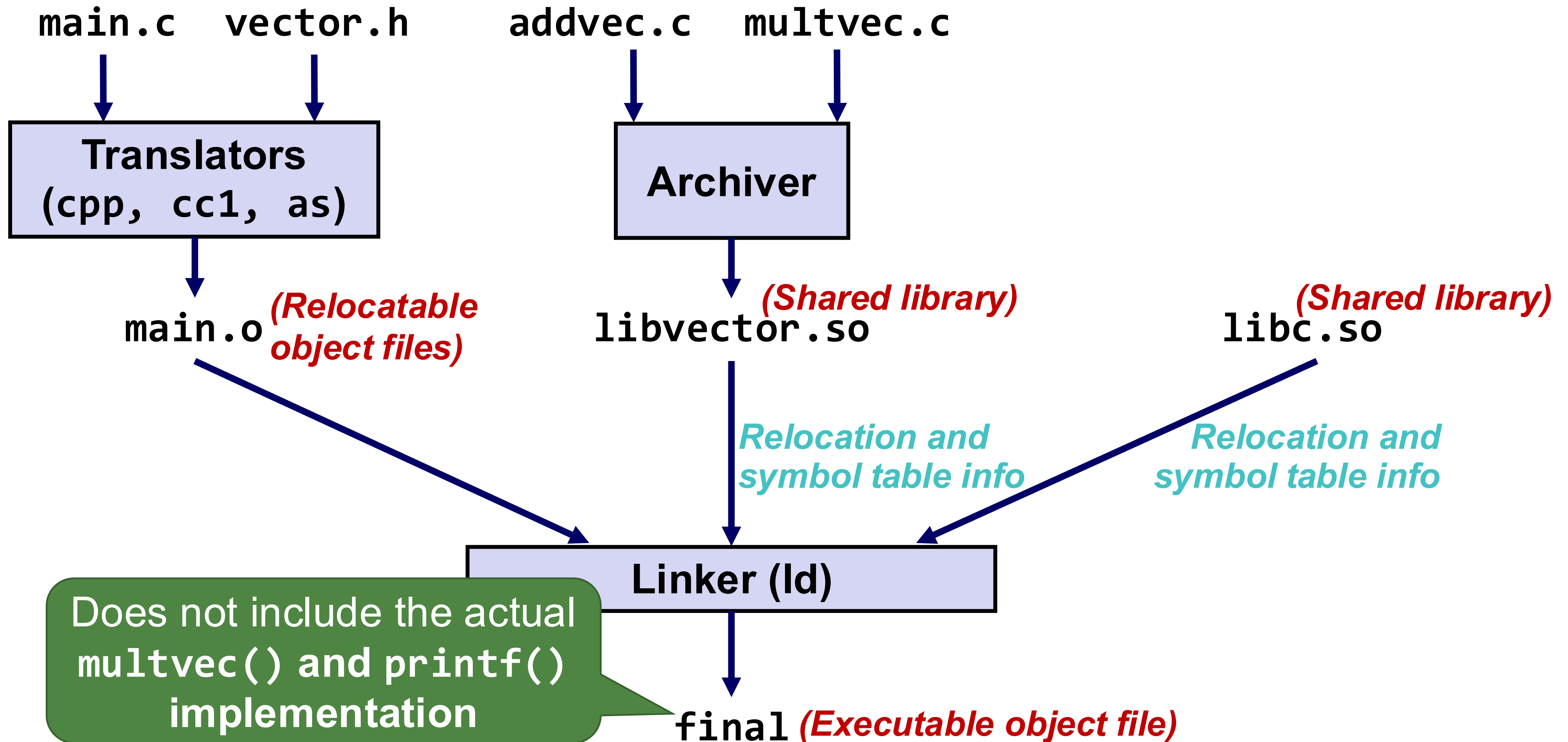



```
$ gcc -shared -o libvector.so addvec.c multvec.c -fpic
```

Linking with Shared Libraries



Linking with Shared Libraries



What the Linker Does Do

- Linker creates an executable that can be linked with the library at load time
  - Records that the program depends on `libc.so` and `libvector.so` (In `.dynamic` section)

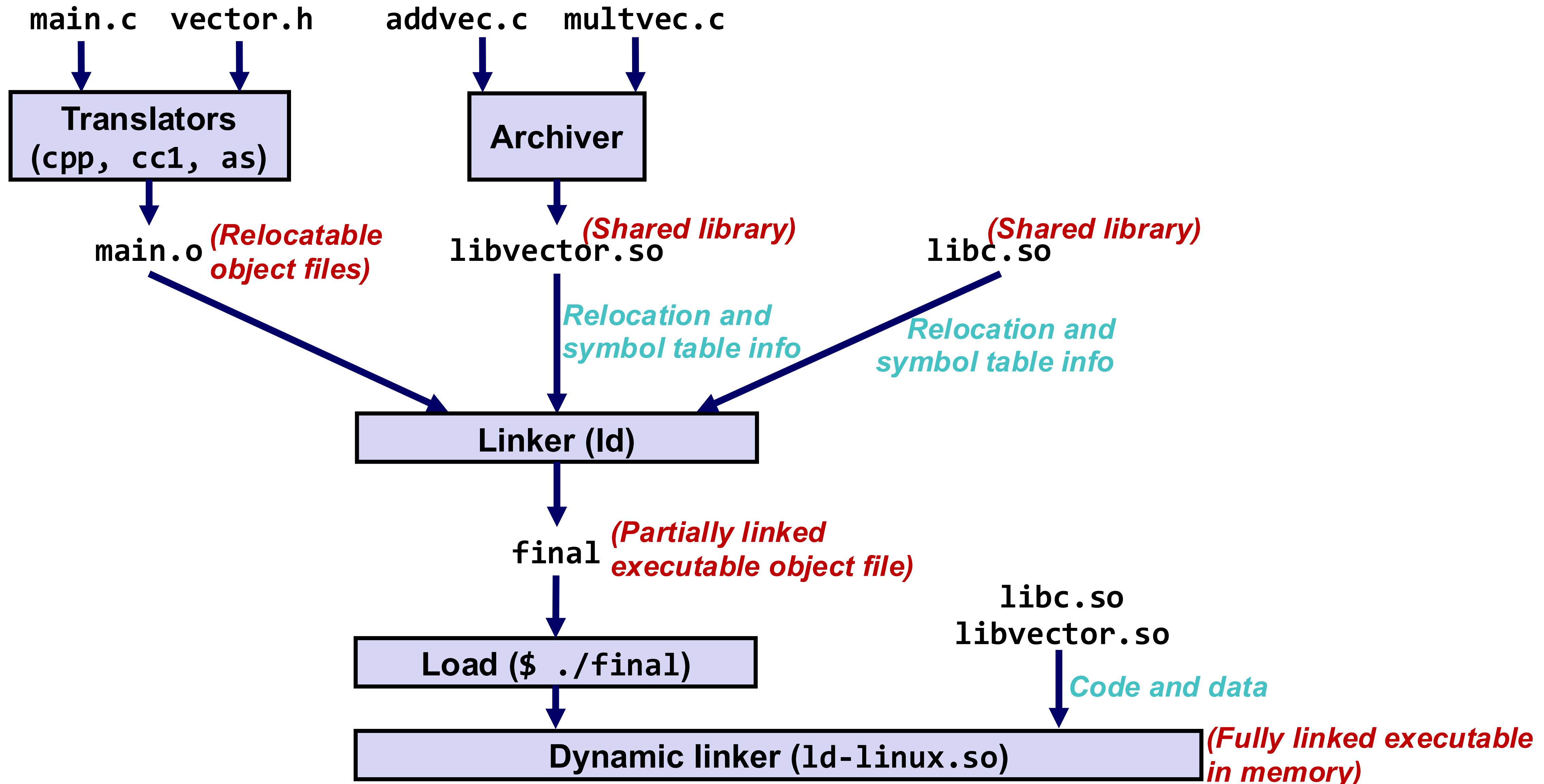
```
$ ldd ./final
linux-vdso.so.1 (0x00007ffff7fc5000)
libvector.so => ./libvector.so (0x00007ffff7fb5000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ffff7dac000)
/lib64/ld-linux-x86-64.so.2 (0x00007ffff7fc7000)
```

  - Creates a .interp section with the location of the dynamic linker (`ld-linux.so`)
  - Copies relocation and symbol table information from library
  - For external function calls such as `printf` and `addvec`, it typically creates PLT/GOT

PLT (Procedure Linkage Table)

GOT (Global Offset Table)

Linking with Shared Libraries



Revisit: What Happens when You Run a Program?

1. The kernel reads the ELF executable and checks its headers
2. The kernel maps (loadable) segments into the program's memory address part
3. The kernel also sets up stack (`args`, env vars) and registers (`rip`, `rsp`, …)
4. If the executable is dynamically linked, **the kernel also loads the dynamic linker (`ld-linux.so`) specified in the ELF file (in `.interp` section)**
5. ...

Dynamic Library Function Call

- What really happens during the library function call?
- When a program calls a library function like `open()`, `rip` does not directly transfer to the library
  - Instead, it first moves to a small code snippet called **PLT**
  - This code snippet uses a function pointer in a table called **GOT**

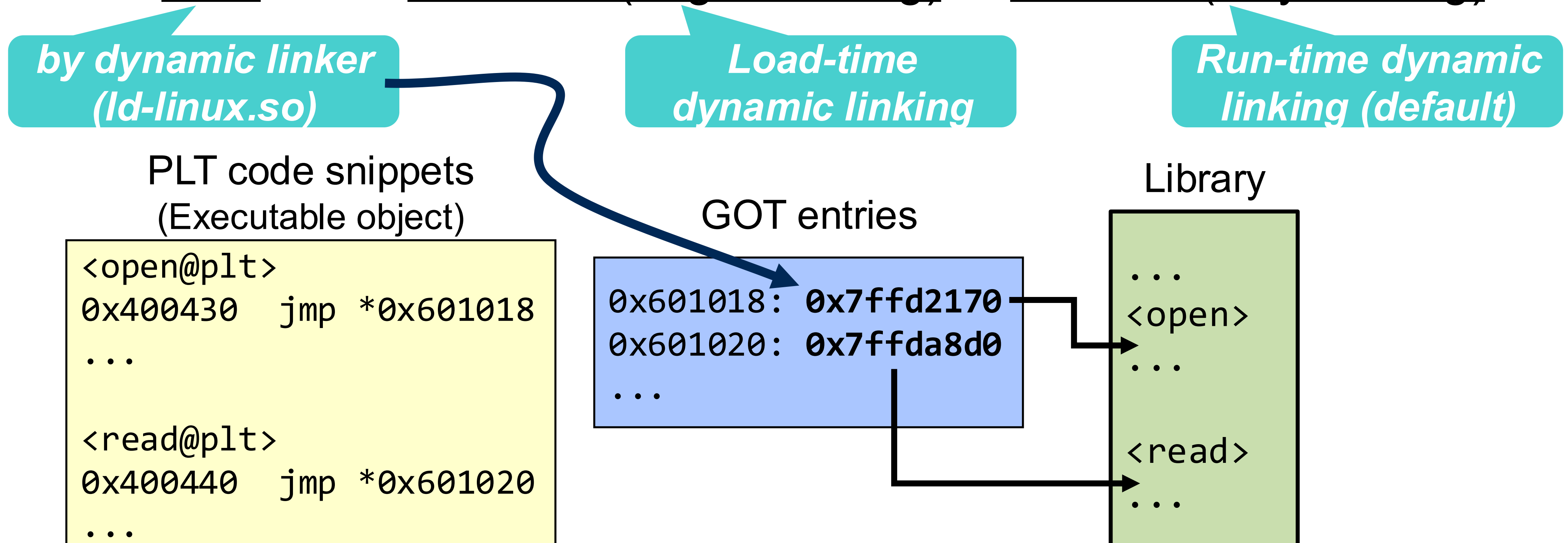
```
int main(void) {
   ...
   open(...);
   ...
}
```

```
<main>
...
0x400579:  call 0x400430 <open@plt>
...

<open@plt>
0x400430  jmp *0x601018  # GOT entry
...
```

PLT and GOT

- In other words, you can think that compiler and linker implicitly generate some function pointer table and fill it
  - To enable your program to call a function in library
  - Each library function called by your program has its PLT+GOT
  - GOT is filled either load-time (eager binding) or run-time (lazy binding)



Revisit: What Happens when You Run a Program?

1. The kernel reads the ELF executable and checks its headers
2. The kernel maps (loadable) segments into the program's memory address part
3. The kernel also sets up stack (`args`, env vars) and registers (`rip`, `rsp`, ...)
4. If the executable is dynamically linked, **the kernel also loads the dynamic linker (`ld-linux.so`) specified in the ELF file (in `.interp` section)**
   - Load shared library (`.so` files)
   - Symbol resolution
   - Relocation (Update GOT entries)

   Load-time dynamic linking
5. The kernel (or **dynamic linker**) transfers control the entry point of the binary: `_start`

Dynamic Linking

- Common case for Linux, handled automatically by the dynamic linker (`ld-linux.so`)
- Standard C library (`libc.so`) usually dynamically linked

Dynamic Linking Pros/Cons

- Pro:
  - Saves disk and memory space
  - No relinking needed when libraries are updated
  - Physical memory sharing of code pages
- Cons:
  - Possible library version mismatch issues
  - Performance cost from GOT/PLT indirection

Question?