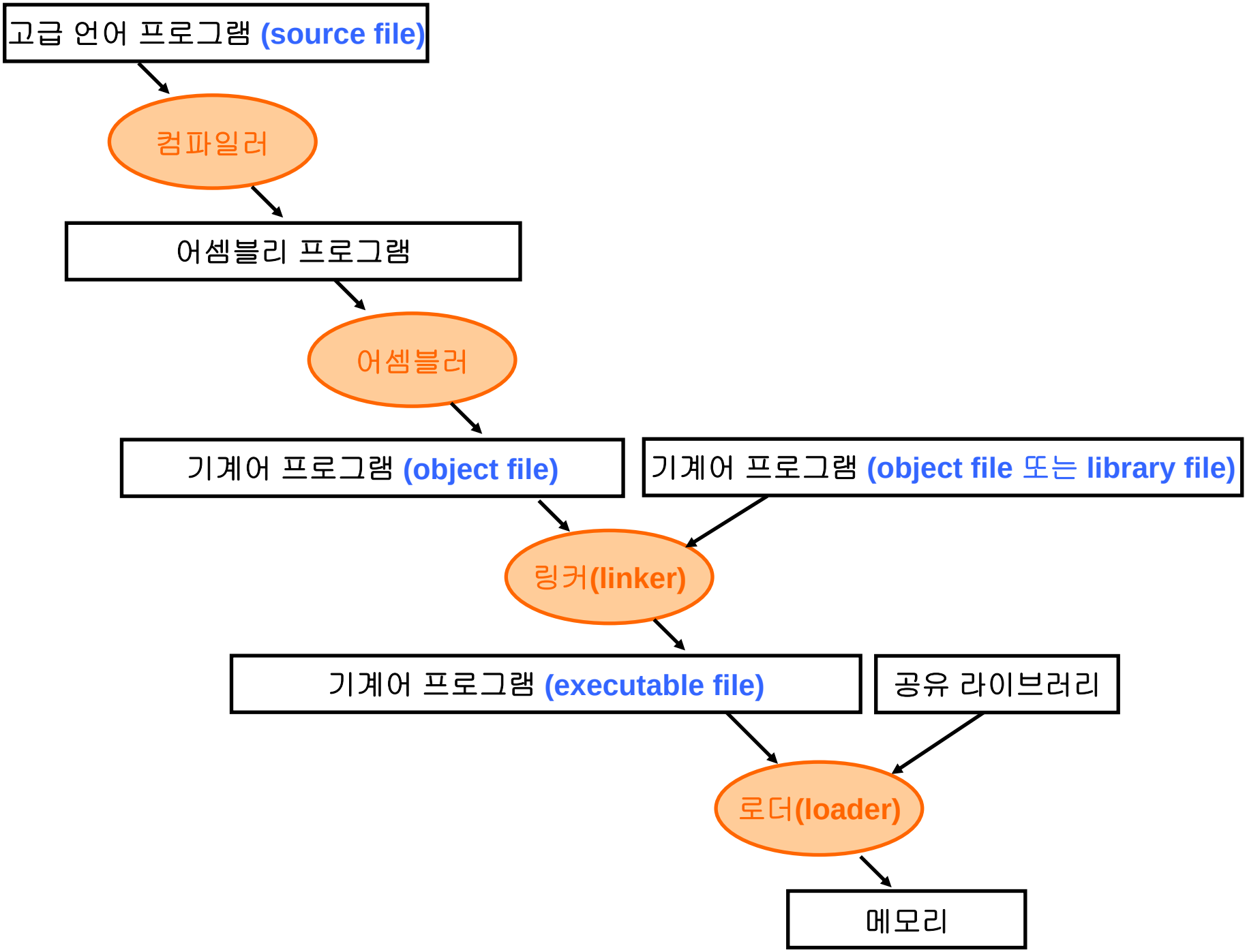
System Software and Program Execution


고급 언어 프로그램 (source file)
컴파일러
어셈블리 프로그램
어셈블러
기계어 프로그램 (object file)
기계어 프로그램 (object file 또는 library file)
링커(linker)
기계어 프로그램 (executable file)
공유 라이브러리
로더(loader)
메모리

Program Execution과 관련한 System Software

고급언어 프로그램 (시스템의 종류에 무관)
→ 어셈블리 프로그램 (시스템에 따라 다름)

어셈블리언어 프로그램 (예: MIPS)
→ 기계어 프로그램 (object file)

여러 object file 및 library file을 하나의 executable file로 만듦

executable file 또는 object file을 메모리로 올림

* 상용 컴파일러 소프트웨어(gcc, visual c)가 컴파일러, 어셈블러, 링커 등을 내포하고 있어 소스 파일에서 실행 파일 변환이 한번에 이루어짐
* 로더 기능은 유닉스, 윈도우 등의 운영체제에 포함되어 수행

Programming Language

고급 언어(High level language)

- 인간이 이해하기 가장 쉬운 언어, machine independent
- machine language /assembly language로 변환시켜주는 소프트웨어 필요 (compiler, interpreter)
- (예) BASIC, C, Java, C++, FORTRAN, COBOL, PASCAL...

어셈블리 언어(Assembly language)

- 0과1의 조합을 상징적인 코드로 변환하여 인간의 이해도를 향상
- Machine language와 1대1 매핑, machine dependent

기계어(Machine language)

- 컴퓨터가 직접 이해가능한 언어, machine dependent
- 0과1의 조합

Programming Language

High level language program

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Assembly language program

```
lw$to,0($2)
lw$t1,4($2)
sw$t1,0($2)
sw$t0,4($2)
```

Machine language program

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

Separate Compile

File “one.c”

```
int a = 3, b = 4;
static int c = 5;
extern int func(int, int);
void main( )
{   int i = 1, j = 2;
    c = c + func(i, j);
     printf("%d \n", c);
}
```

File “two.c”

```
int c = 6;
extern int a, b;
int func(int x, int y)
{
    a++; b++;
    return(x + y + a + b + c);
}
```

Separate Compile

a.c

```
int  x;

x = 0;
```

b.c

```
extern  int  x;

x++;
```

c.c

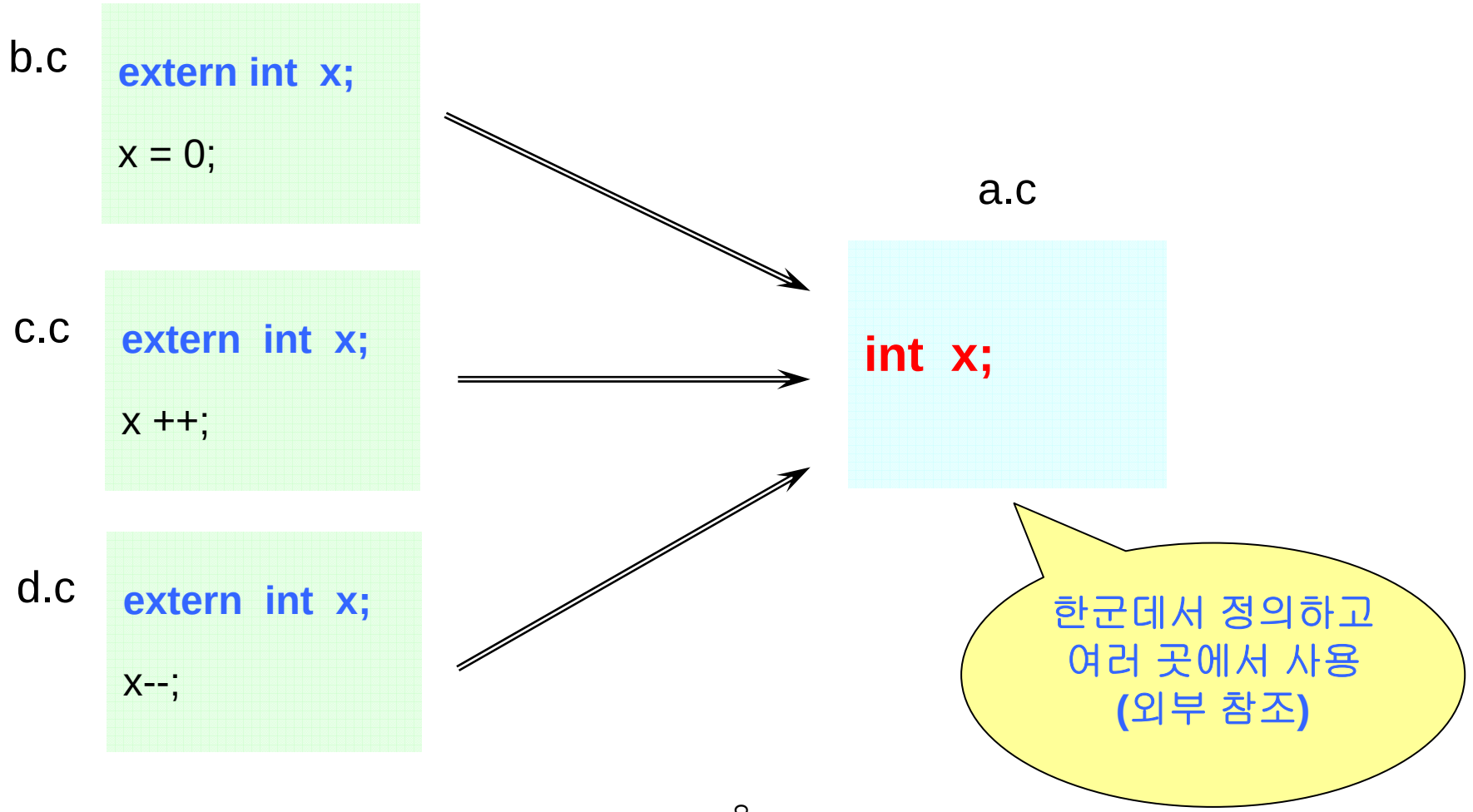
```
extern  int  x;

x--;
```

- Separate compile:
  symbol*의 위치 불명확, type만 파악
  → "object file"
  (*.o or *.obj)

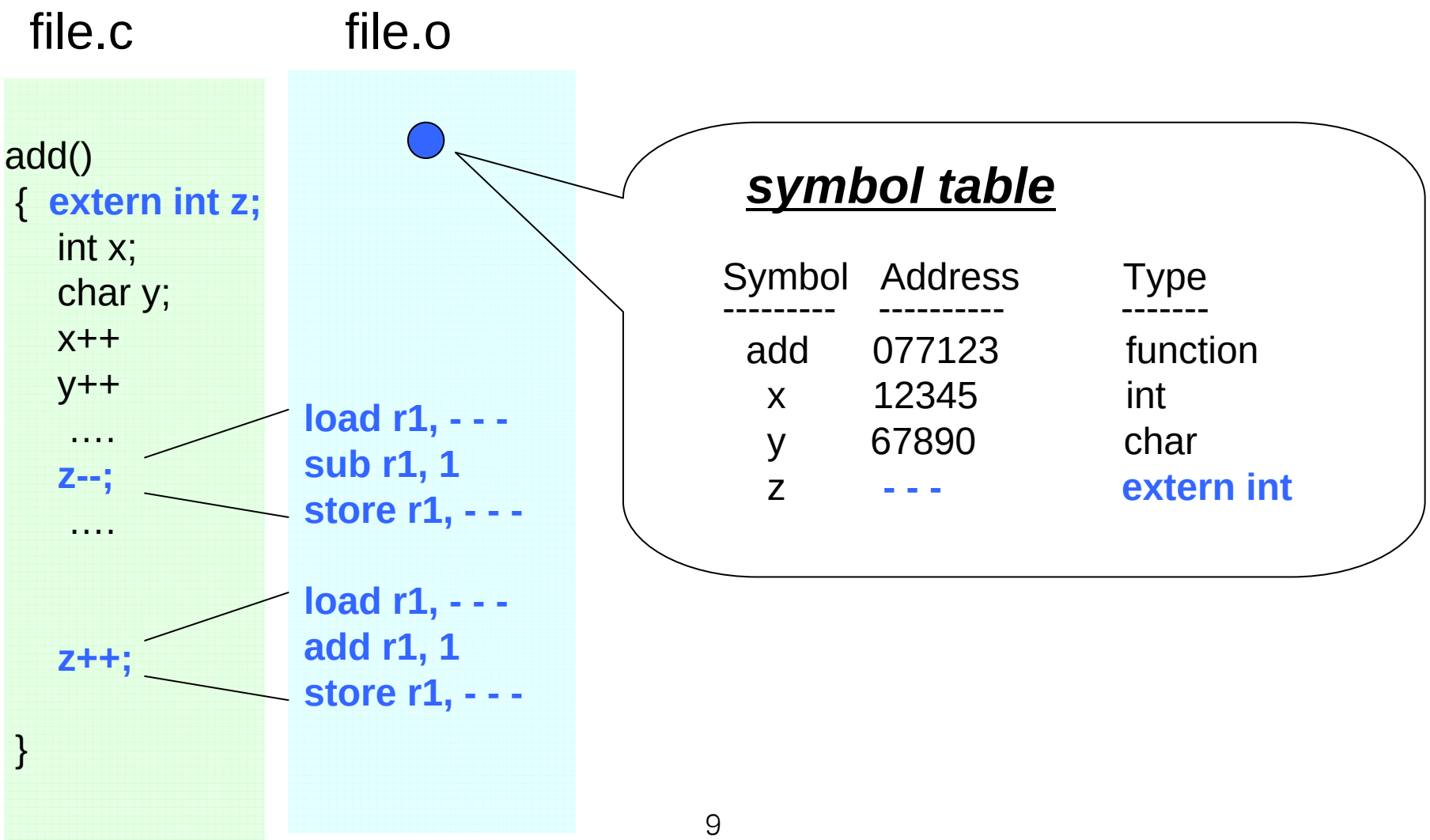
- Linking:
  symbol의 위치 명확
  → "executable file"
  (a.out or *.exe)

* symbol이란 변수, 함수 등의 이름을 말함

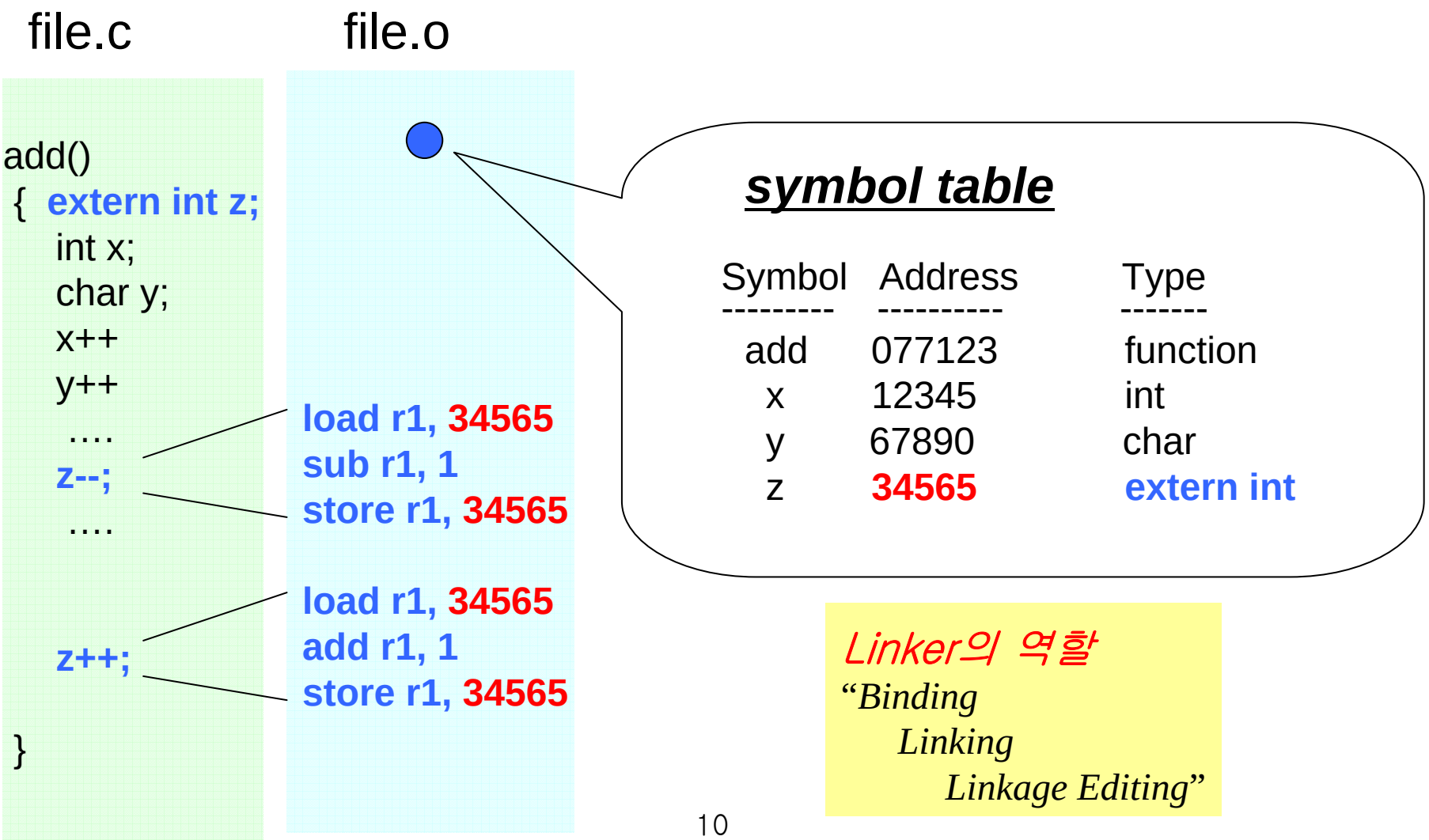
Separate Compile



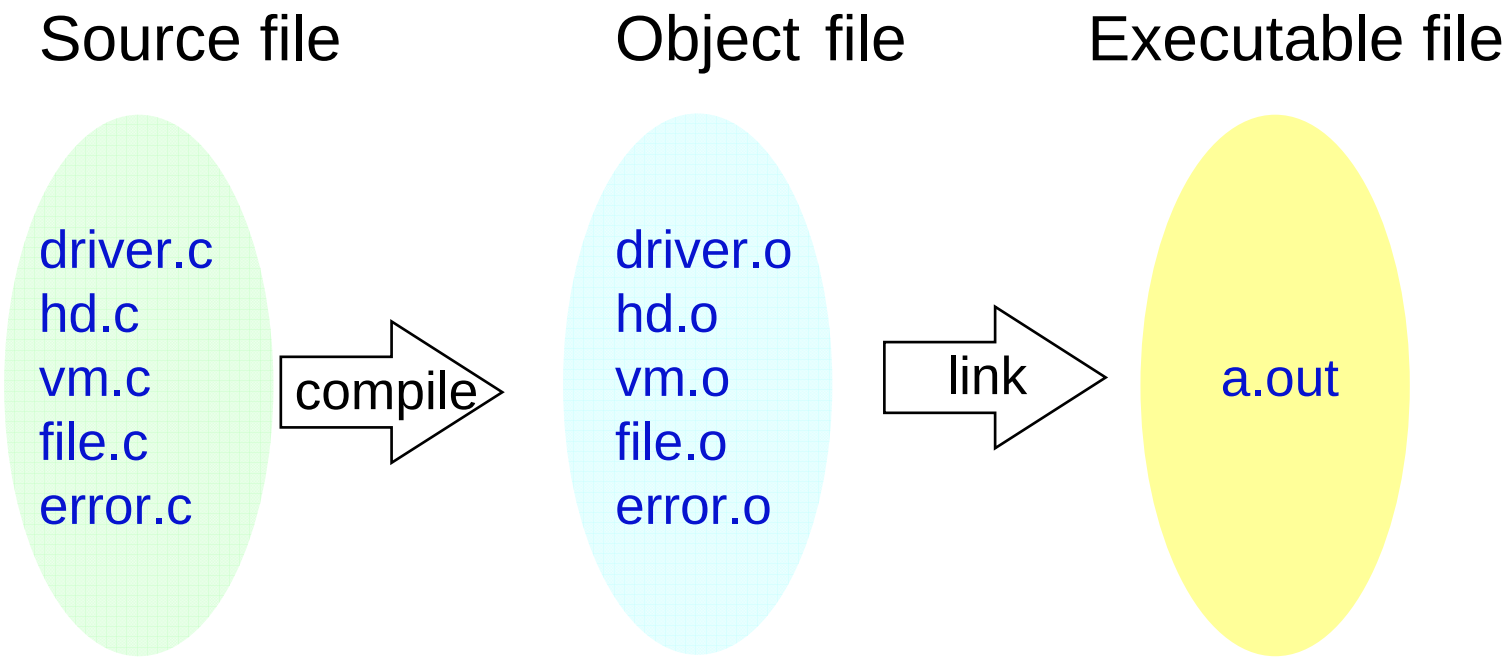
Symbol Table



Symbol Table



Compile and Linking



What if there are so many extern's?

a.c

```
int  x;
x = 0;
```

b.c

```
extern  int  x;
x ++;
```

c.c

```
extern  int  x;
x--;
```

Suppose we change int x in a.c

Q: int x; → char x;

A: Search all files

Change all **extern int x → extern char x;** manually !

Q: What if 1,000 files and 100,000 variables?

Header File

x.h
```
extern int  x;
```

a.c
```
int  x;

x = 0;
```

b.c
```
# include “x.h”

x ++;
```

c.c
```
# include “x.h”

x--;
```

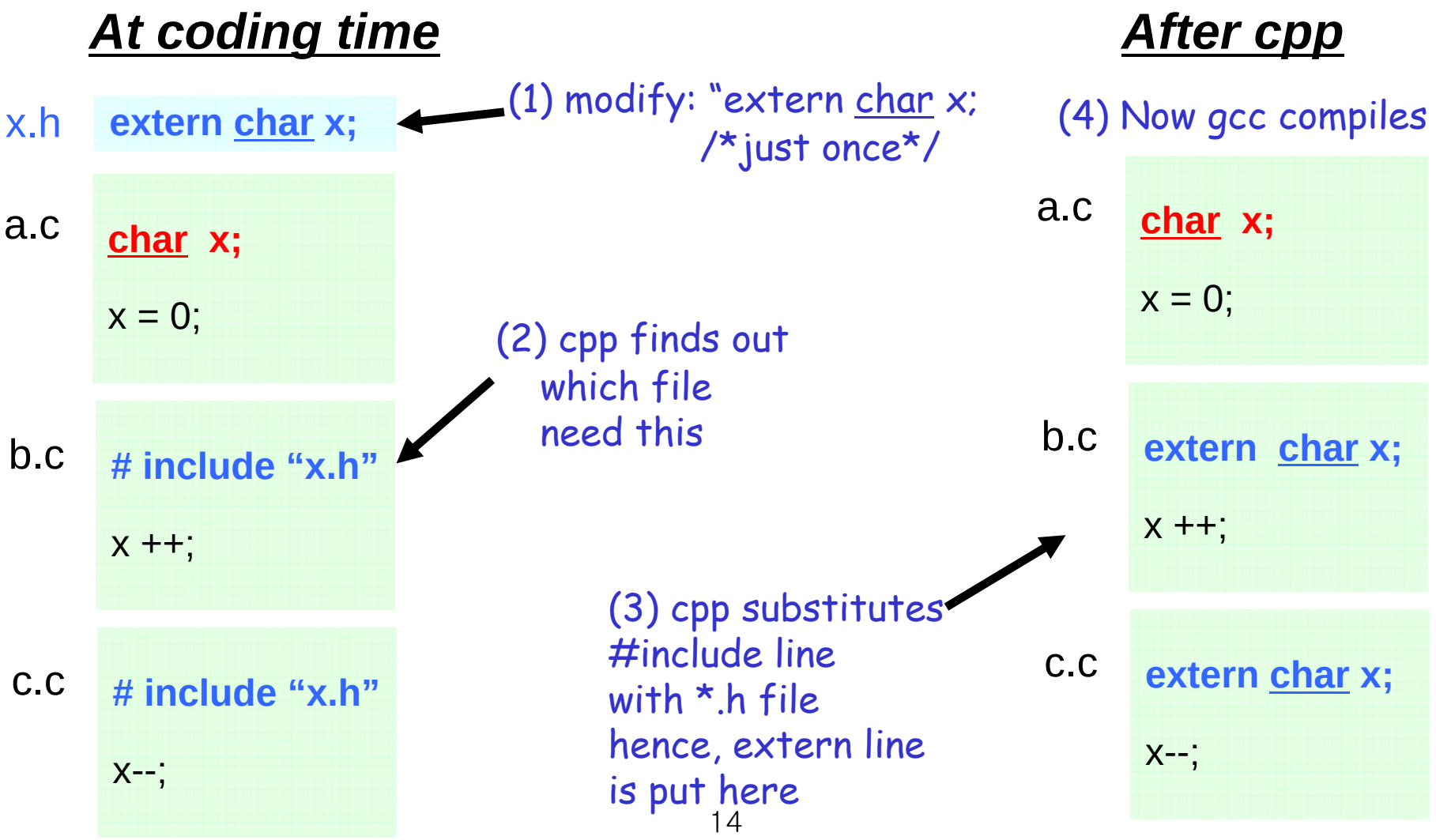
(1) Let b.c c.c say “#include” not “extern”
“We reference x”

(2) Let header file (x.h) have all extern’s
“Change x? Change here once”

(3) cpp* substitutes (just before gcc)
#include “x.h” → extern

* cpp: c preprocesspr (전처리기)

C preprocessor



Library

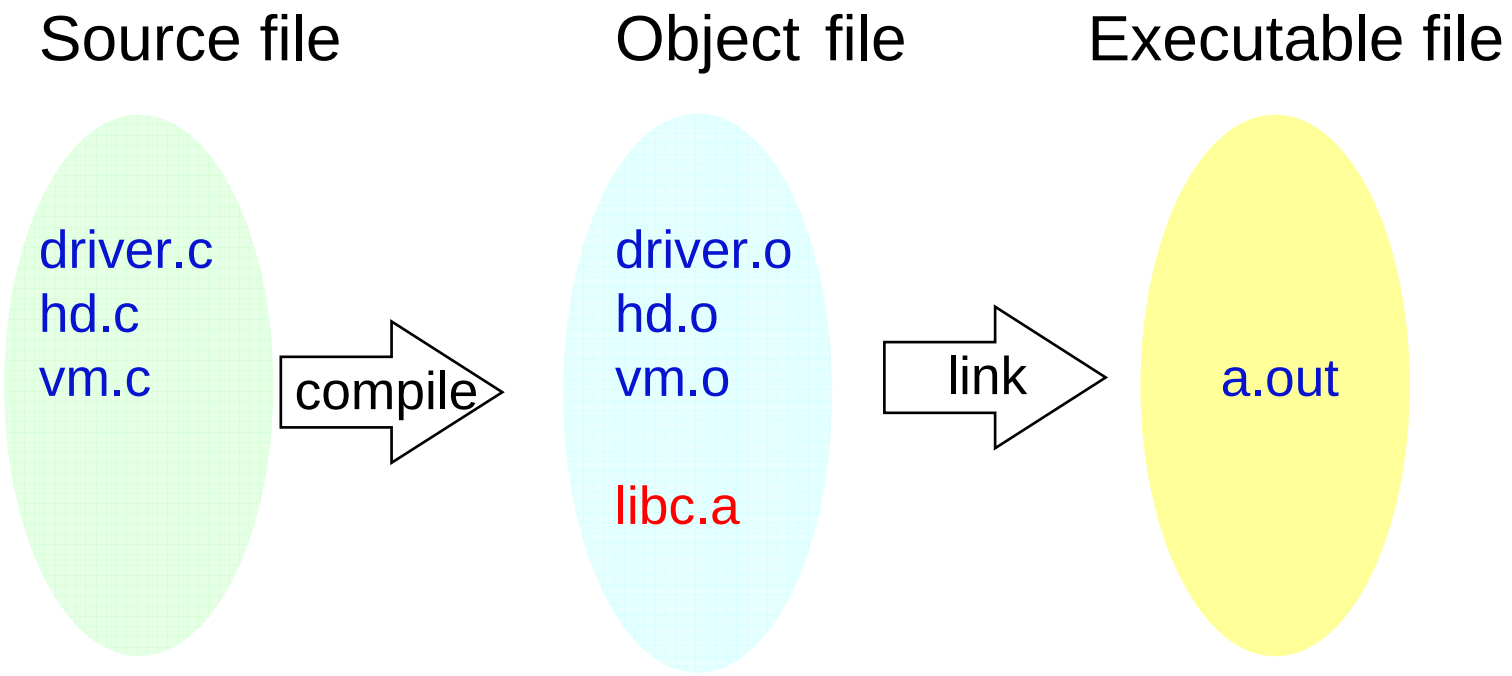
- Standard library header files
  - <…> /usr/include
  - <sys/…> /usr/include/sys
- 헤더 파일에는 라이브러리 함수의 선언만 있고 실제 함수의 정의는 라이브러리 파일에 있는 것이 보통임

(예)

```
#include <stdio.h>
main()
{
    printf();
}
```

함수 선언인 **extern printf();**가 있어서 함수의 **argument** 및 **return type**을 알려줌 **(**실제 정의는 라이브러리에 존재**)**

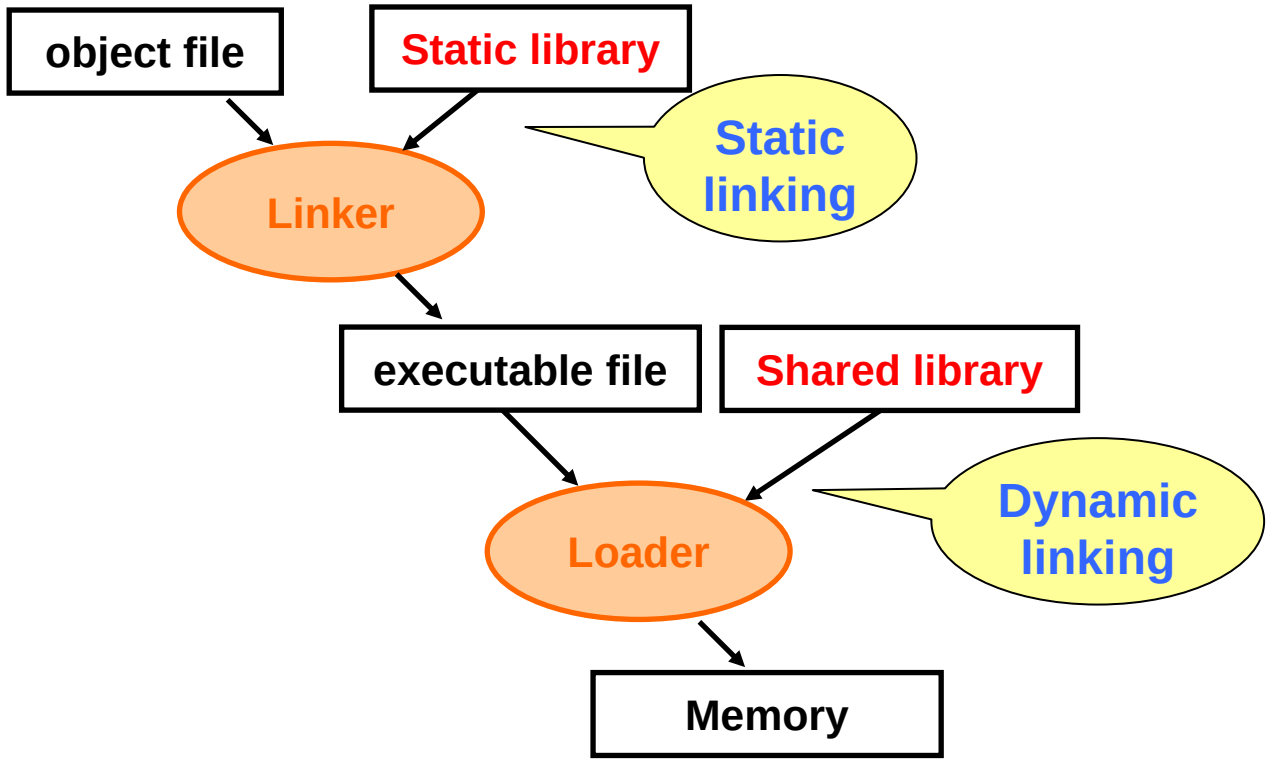
Compile and Linking (Revisited)



Static Linking vs. Dynamic Linking

- Static linking
  - 라이브러리가 프로그램의 실행 파일 코드에 포함됨 (***static library***)
  - 실행 파일의 크기가 커짐
  - 동일한 라이브러리를 각각의 프로세스가 메모리에 올리므로 메모리 낭비 (예: printf 함수의 라이브러리 코드)
- Dynamic linking
  - 라이브러리가 실행시 link됨 (***shared library***)
  - 실행파일에는 라이브러리 자체가 포함되는 것이 아니라 라이브러리의 위치를 찾기 위한 간단한 코드만 둠
  - 라이브러리가 이미 메모리에 있으면 그 루틴의 주소로 가고 없으면 디스크에서 읽어옴

Static Linking vs. Dynamic Linking

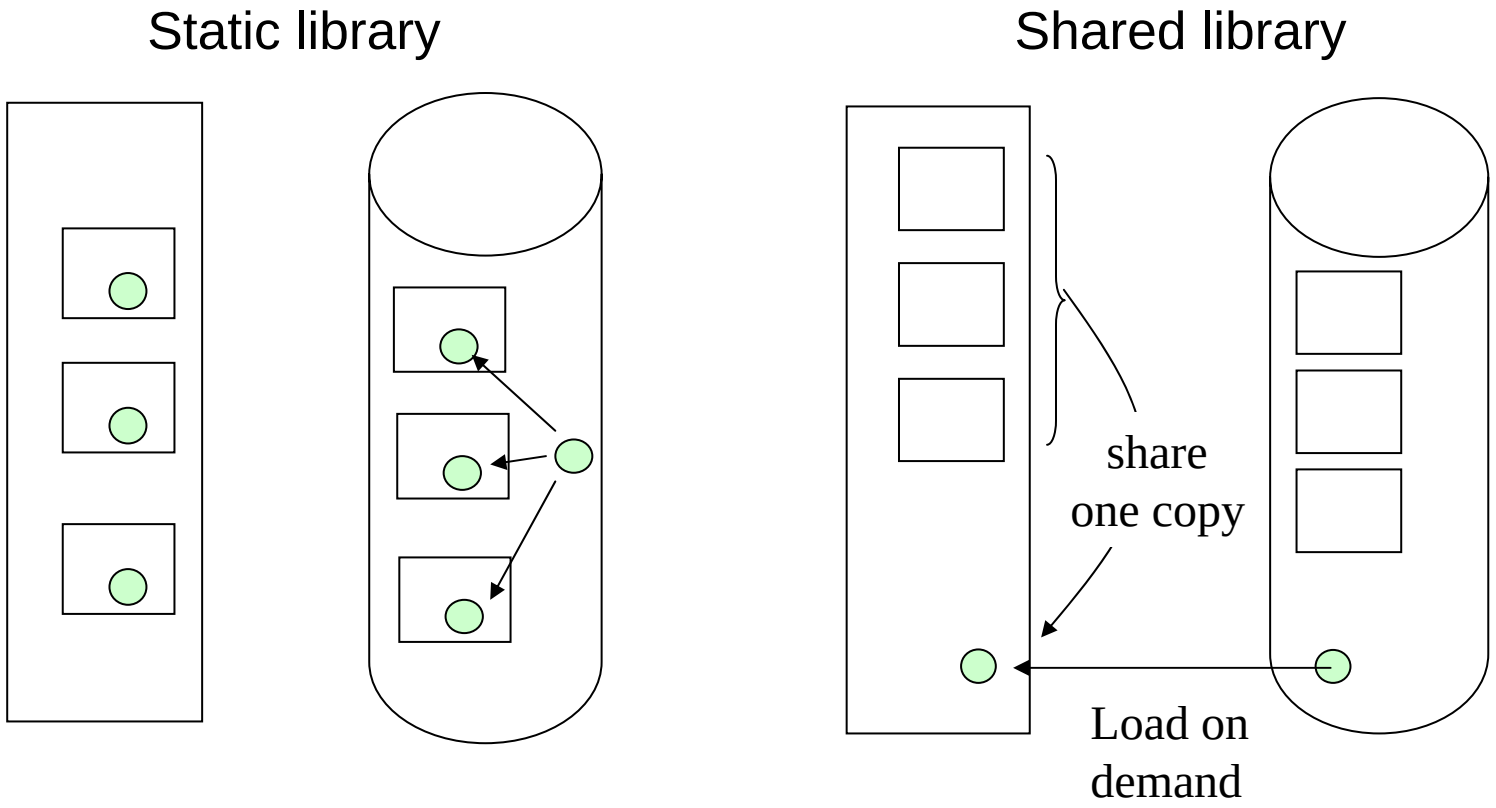


Static Library vs. Shared Library

- Static library
  - functions are copied into a.out during gcc time
  - binding: compile time

- Shared library
  - functions are not copied during gcc time (Just map)
  - During the run, load on demand from library (shared)
  - binding: run time

Static Library vs. Shared Library



Shared Library

- Advantage of shared library
  - Many processes share one library function in memory
    - (예) many a.out's calls printf() --- one copy shared
  - less memory space
- Disadvantage of shared library
  - Cannot give a.out to others who do not have library!
    - Do they have shared library files? Same version?
  - Slower (for lib calls, bind and load at run time)
  - heavy overhead in initial bookkeeping during gcc