



# 程序的链接

王晶

[jwang@ruc.edu.cn](mailto:jwang@ruc.edu.cn), 信息楼124

2024年12月



# 为什么需要链接器？

- **原因 1：模块化（Modularity）**

- 程序可以被编写为多个较小的源文件的集合，而不是一个单一的庞大结构
- 可以构建常用函数的库（稍后会详细讨论）
  - 例如，数学库、标准 C 库



# 为什么需要链接器？（续）

- 原因 2：效率

- 时间：分离编译

- 更改一个源文件，编译，然后重新链接
    - 无需重新编译其他源文件
    - 可以并行编译多个文件

- 空间：库

- 常用函数可以被聚合到单个文件中...
    - 选项 1：静态链接（Static Linking）
      - 可执行文件和运行时内存镜像只包含它们实际使用的库代码
    - 选项 2：动态链接（Dynamic linking）
      - 可执行文件不包含库代码
      - 在执行过程中，库代码的单个副本可以在所有执行进程之间共享



# 为什么要学习链接器？

- 帮助你构建大型程序
  - 链接错误（如缺失模块、库或不兼容的库）可能令人困惑和沮丧
- 帮助你避免危险的错误
  - 链接器对符号引用的解析决策可能会静默影响程序的正确性
- 帮助你理解语言作用域规则是如何实现的
  - 全局名称与局部名称，static 的真正含义
- 使你能够利用共享库



# 示例 C 程序

```
int sum(int *a, int n);

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

int main(int argc, char** argv)
{
    int val = sum(array, 2);
    return val;
}
```

*main.c*

```
int sum(int *a, int n)
{
    int i, s = 0;

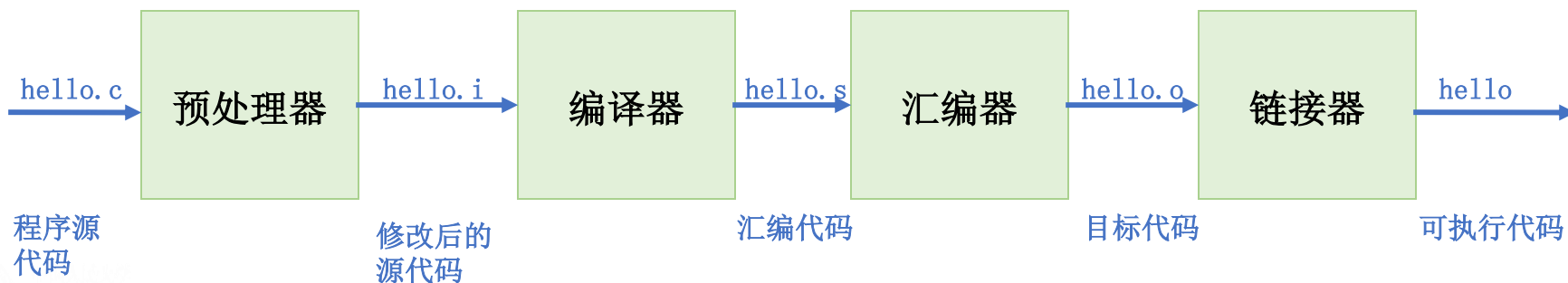
    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

*sum.c*



# 编译器驱动程序（以 GCC 为例）

- GCC 是编译工具链中的编译器驱动程序
- GCC 调用多个其他编译阶段
  - cpp, 预处理器
  - cc1, 编译器
  - as/gas, 汇编器
  - ld, 链接器
- 每个阶段的功能和输出是什么？





# 预处理器

- 首先，gcc 编译器驱动程序调用 cpp 生成展开的源代码
  - 预处理器只进行文本替换 / gcc 带有 "-E" 选项
  - 将 C 源文件转换为另一个 C 源文件
  - 展开 "#" 指令

```
#include <stdio.h>
#define FOO 4

int main() {
    printf("hello, world %d\n", FOO);
}
```

```
...
extern int printf (const char
*__restrict __format, ...);
...

int main() {
    printf("hello, world %d\n", 4);
}
```



# 编译器

- 接下来，gcc 调用 cc1 生成汇编代码
  - 将高级 C 代码翻译成汇编语言

```
// ...  
extern int printf (const char  
*__restrict __format, ...);  
// ...  
int main() {  
    printf("hello, world %d\n", 4);  
}
```

```
.section      .rodata  
.LC0:  
    .string "hello, world %d\n"  
  
.text  
main:  
    pushq    %rbp  
    movq     %rsp, %rbp  
    movl     $4, %esi  
    movl     $.LC0, %edi  
    movl     $0, %eax  
    call     printf  
    popq     %rbp  
    ret
```





# 汇编器

- 此外，gcc 调用 gas 来生成目标代码
  - 将汇编代码转换为二进制目标代码

```
# readelf -a hello | grep rodata
[10] .rodata          PROGBITS          0000000000495d40  00095d40

# readelf -a hello | grep -E "GLOBAL.* main"
1591: 0000000000401190   31 FUNC          GLOBAL DEFAULT   6 main

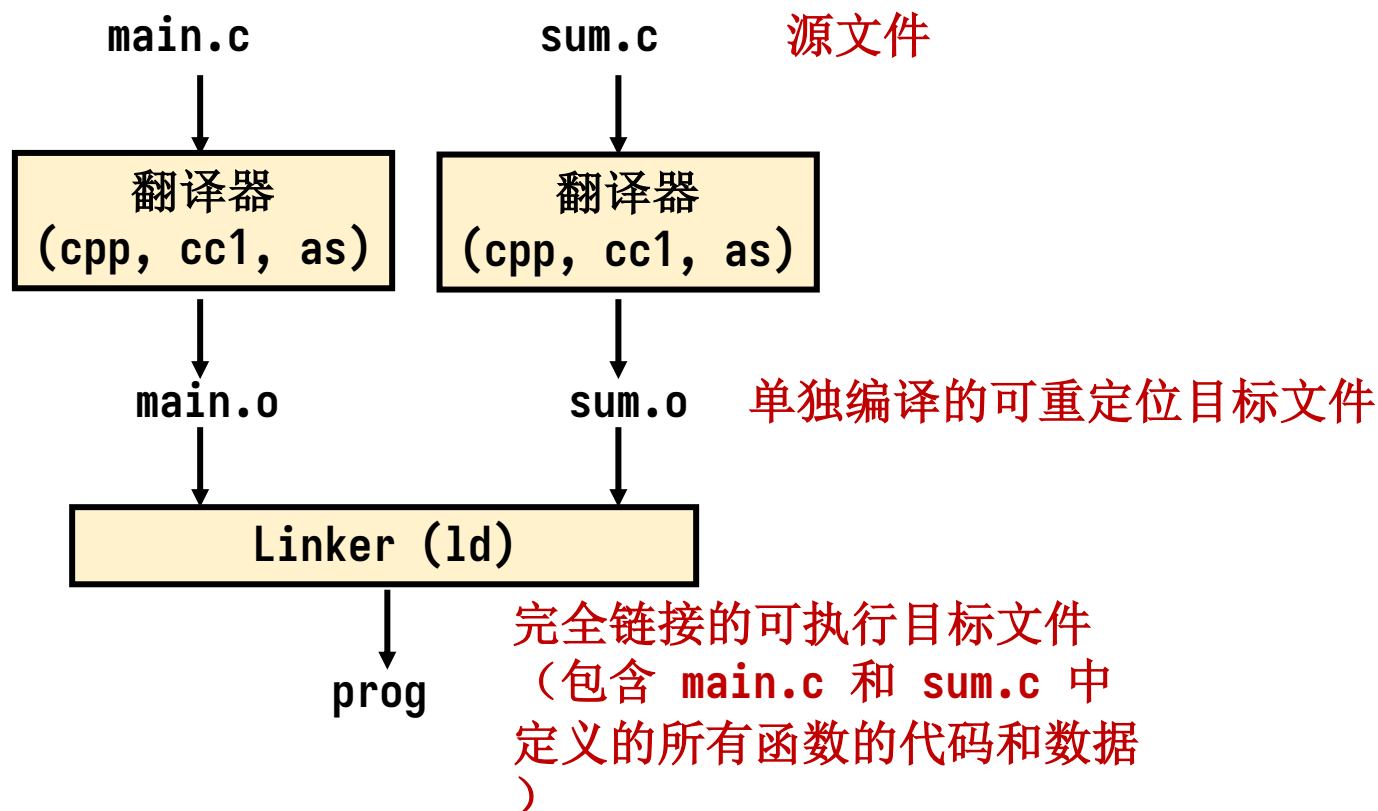
# readelf -x .rodata hello
Hex dump of section '.rodata':
0x00495d40 01000200 68656c6c 6f2c2077 6f726c64 ....hello, world
0x00495d50 2025640a 00464154 414c3a20 6b65726e %d..FATAL: kern

# objdump -d hello
0000000000401190 <main>:
401190:    55                push    %rbp
401191:    48 89 e5          mov     %rsp,%rbp
401194:    be 04 00 00 00    mov     $0x4,%esi
401199:    bf 44 5d 49 00    mov     $0x495d44,%edi
40119e:    b8 00 00 00 00    mov     $0x0,%eax
4011a3:    e8 d8 0e 00 00    callq   402080 <_IO_printf>
4011a8:    b8 00 00 00 00    mov     $0x0,%eax
4011ad:    5d                pop     %rbp
4011ae:    c3                retq
4011af:    90                nop
```



# 链接

- 程序使用编译器驱动程序进行翻译和链接：
  - `linux> gcc -Og -o prog main.c sum.c`
  - `linux> ./prog`





# 链接器的作用是什么？

## • 步骤 1：符号解析

- 程序定义和引用符号（全局变量和函数）：

- `void swap() {...}`      `/* 定义符号 swap */`
- `swap();`                      `/* 引用符号 swap */`
- `int *xp = &x;`              `/* 定义符号 xp, 引用 x */`

- 符号定义存储在目标文件的符号表中（由汇编器生成）
  - 符号表是一个条目数组
  - 每个条目包括符号的名称、大小和位置
- 在符号解析步骤中，链接器将每个符号引用与一个确切的符号定义关联起来



# C 程序中的符号示例

```
int sum(int *a, int n);

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

int main(int argc, char** argv)
{
    int val = sum(array, 2);
    return val;
}
```

引用

*main.c*

定义

```
int sum(int *a, int n)
{
    int i, s = 0;

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

*sum.c*

```
# gcc -c -o main.o main.c
# gcc -c -o sum.o sum.c
# nm main.o
0000000000000000 D array
0000000000000000 T main
                   U sum

# nm sum.o
0000000000000000 T sum
```

您也可以尝试：  
**objdump -t main.o**  
**objdump -t sum.o**



# 链接器的作用是什么？（续）

- 步骤 2：重定位

- 合并分散的代码和数据段为单一的段
- 将符号从它们在 .o 文件中的相对位置重定位到可执行文件中的最终绝对内存位置
- 更新所有对这些符号的引用，以反映它们的新位置
- 让我们更详细地看看这两个步骤.....



# 三种目标文件（模块）

- 可执行目标文件（.out 文件）
  - 包含可以直接复制到内存中并执行的代码和数据形式
- 可重定位目标文件（.o 文件）
  - 包含可以与其他可重定位目标文件组合形成可执行目标文件的代码和数据形式
  - 每个 .o 文件精确对应一个源（.c）文件
  - 每个 .c 文件对应一个，用于组成可执行文件
- 共享目标文件（.so 文件）
  - 一种特殊的可重定位目标文件，可以在加载时或运行时被动态加载到内存并链接
  - 在 Windows 系统中被称为动态链接库（DLLs）



# Linux 下的三种目标文件

```
# file sum.o main.o
```

## Relocatable object file (.o file)

```
sum.o:  ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

```
main.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

```
# file main
```

## Executable object file (a.out file)

```
main: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses shared  
libs), for GNU/Linux 2.6.24, BuildID[sha1]=0x34c39011eac6fd0ebae938e4087e788b28a4f6dd, not  
stripped
```

```
# ldd main
```

```
linux-vdso.so.1 => (0x00007fff9dbfe000)
```

```
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f4bef587000)
```

```
/lib64/ld-linux-x86-64.so.2 (0x00007f4bef956000)
```

```
# file /lib/x86_64-linux-gnu/libc.so.6
```

```
/lib/x86_64-linux-gnu/libc.so.6: symbolic link to `libc-2.15.so'
```

```
# file /lib/x86_64-linux-gnu/libc-2.15.so
```

## Shared object file (.so file)

```
/lib/x86_64-linux-gnu/libc-2.15.so: ELF 64-bit LSB shared object, x86-64, version 1  
(SYSV), dynamically linked (uses shared libs),
```

```
BuildID[sha1]=0x760efc6878e468a84b60e307a5bad802cbe2a480, for GNU/Linux 2.6.24, stripped
```



# 可执行与可链接格式（ELF）

- 目标文件的标准二进制格式
- 一种统一格式，适用于：
  - 可重定位目标文件（.o）
  - 可执行目标文件（a.out）
  - 共享目标文件（.so）
- 通用名称：ELF 二进制文件
- 首次出现于 System V Release 4 Unix，约 1989 年
- Linux 于约 1995 年切换到 ELF，BSD 后来于约 1998-2000 年间切换





# ELF 目标文件格式

- **ELF 头**

- 字长、字节顺序、文件类型（.o, exec, .so）、机器类型等

- **段头表**

- 页面大小、虚拟地址内存段（节）、段大小

- **.text 节**

- 代码

- **.rodata 节**

- 只读数据：跳转表、字符串常量等

- **.data 节**

- 已初始化的全局变量

- **.bss 节**

- 未初始化的全局变量
- "Block Started by Symbol"(由符号开始的块)
- "Better Save Space"(更好地节省空间)
- 有节头但不占用空间

|                  |
|------------------|
| ELF 头            |
| 段头表<br>(可执行文件必需) |
| .text 节          |
| .rodata 节        |
| .data 节          |
| .bss 节           |
| .symtab 节        |
| .rel.txt 节       |
| .rel.data 节      |
| .debug 节         |
| 节头表              |



# ELF 目标文件格式（续）

- **.symtab 节**
  - 符号表
  - 过程和静态变量名称
  - 节名称和位置
- **.rel.text 节**
  - .text 节的重定位信息
  - 需要在可执行文件中修改的指令地址
  - 修改指令
- **.rel.data 节**
  - .data 节的重定位信息
  - 需要在合并的可执行文件中修改的指针数据地址
- **.debug 节**
  - 用于符号调试的信息（gcc -g）
- **节头表**
  - 每个节的偏移量和大小

|                  |
|------------------|
| ELF 头            |
| 段头表<br>(可执行文件必需) |
| .text 节          |
| .rodata 节        |
| .data 节          |
| .bss 节           |
| .symtab 节        |
| .rel.txt 节       |
| .rel.data 节      |
| .debug 节         |
| 节头表              |



# 链接符号

- 全局符号

- 由模块  $m$  定义的符号，可以被其他模块引用
- 例如：非 *static* 的 C 函数和非 *static* 的全局变量

- 外部符号

- 由模块  $m$  引用，但由其他模块定义的全局符号

- 局部符号

- 由模块  $m$  独立定义和引用的符号
- 例如：使用 *static* 属性定义的 C 函数和全局变量
- 局部链接符号不是局部程序变量



# 第一步：符号解析

...在这里定义

```
int sum(int *a, int n);  
int array[2] = {1, 2};  
  
int main(int argc, char **argv)  
{  
    int val = sum(array, 2);  
    return val;  
}
```

main.c

定义一个全局符号

链接器不知 val

引用全局符号

引用全局符号

...在这里定义

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

sum.c

链接器不知 i 或 s

链接器如何解析重复符号定义（例如 **sum** 和 **array**）？



# 符号识别

- 下列名称中的哪些会出现在 `symbols.o` 的符号表中？

`symbols.c`:

```
int incr = 1;
static int foo(int a) {
    int b = a + incr;
    return b;
}

int main(int argc,
          char* argv[]) {
    printf("%d\n", foo(5));
    return 0;
}
```

名称:

- `incr`
- `foo`
- `a`
- `b`
- `argc`
- `argv`
- `main`
- `printf`
- `"%d\n"`

可以使用 `readelf` 工具找到这些信息:

```
linux> readelf -s symbols.o
```



# 解析全局符号

- 当无法在任何输入模块中找到引用符号的定义时，链接器会打印错误信息并终止

```
void foo(void);  
int main() {  
    foo();  
    return 0;  
}  
  
linkerror.c
```

```
# gcc -Wall -Og -S linkerror.c  
# as -o linkerror.o linkerror.s
```

**编译器和汇编器运行正常**

```
# gcc -Wall -Og -o linkerror linkerror.o  
linkerror.o: In function `main':  
linkerror.c:(.text+0x5): undefined reference to `foo'  
collect2: error: ld returned 1 exit status
```

**当无法解析对 foo 函数的引用时，链接器终止**



# 局部符号

- 局部非静态 C 变量与局部静态 C 变量
  - 局部非静态 C 变量：存储在堆栈上
  - 局部静态 C 变量：存储在 .bss 或 .data 中

```
static int x = 15;
```

```
int f() {  
    static int x = 17;  
    return x++;  
}
```

```
int g() {  
    static int x = 19;  
    return x += 14;  
}
```

```
int h() {  
    return x += 27;  
}  
  
static-local.c
```

编译器为每个 `x` 的定义在 `.data` 中分配空间

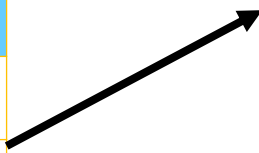
在符号表中为局部符号创建唯一名称，例如 `x`、`x.1721` 和 `x.1724`



# 名称改编

- C++/Java 中的重载

| 名称      | 引用             |
|---------|----------------|
| "A"     | <class A>      |
| "print" | <overload set> |



| 签名           | 引用        |
|--------------|-----------|
| void(int)    | <print 1> |
| void(char)   | <print 2> |
| void(String) | <print 3> |

- 名称改编将函数的签名（参数和返回类型）编码成文本形式。

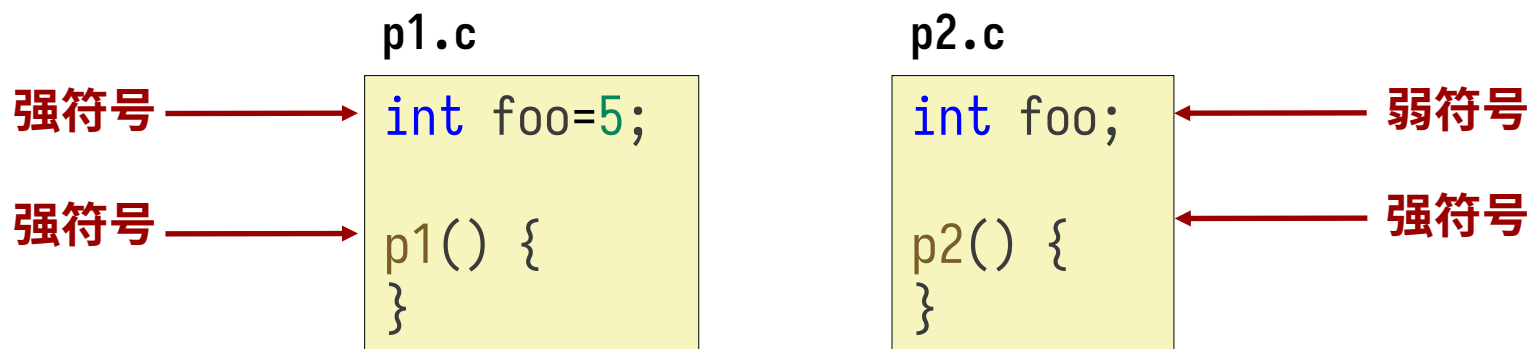
```
void print(int i, float f)
=> "_Z5printf"          (g++)
=> "?print@@YAXHM@Z"   (msvc++)
void print(float f, int i)
=> "_Z5printfi"         (g++)
=> "?print@@YAXMH@Z"    (msvc++)
```





# 链接器如何解决重复的符号名称

- 程序符号分为强（strong）或弱（weak）两种
  - 强符号：过程和已初始化的全局变量
  - 弱符号：未初始化的全局变量
    - 或者使用 `extern` 说明符声明的变量
- 编译器输出此类信息，汇编器将其隐式编码在 ELF 文件的符号表中。





# 链接器的符号规则

- 规则 1：不允许存在多个强符号
  - 每个项目只能定义一次
  - 否则：链接器报错
- 规则 2：给定一个强符号和多个弱符号时，选择强符号
  - 对弱符号的引用会解析为强符号
- 规则 3：如果存在多个弱符号，任选其中一个
  - 可以使用 `gcc -fno-common` 来覆盖这一行为



# 如果出错了会怎样？

```
int x=7;  
p1() {}
```

```
extern int x;  
p2() {}
```

正确的程序。  
x、p1、p2 各只有一个定义。

```
int x=7;  
p1() {}
```

```
int x=0;  
p1() {}
```

链接错误：x 和 p1 有两个定义。

```
int x;  
p1() {}
```

```
int x;  
p2() {}
```

依赖编译器。  
可能被视为 x 的一个或两个定义。

```
int x=7;  
int y=5;  
p1() {}
```

```
extern double x;  
p2() {}
```

未定义行为。不会产生链接错误。  
在 p2 中对 x 的写入可能会覆盖 y！

```
char p1[]  
= 0xC3;
```

```
extern void p1();  
p2() { p1(); }
```

未定义行为。不会产生链接错误。  
调用 p1 可能会导致程序崩溃！

链接器检查一个符号是否有两个定义，不检查引用的类型。



# 链接器谜题

```
int x;  
p1() {}
```

```
p1() {}
```

链接时错误：两个强符号（p1）

```
int x;  
p1() {}
```

```
int x;  
p2() {}
```

对 x 的引用将指向同一个未初始化的 int  
这真的是你想要的吗？

```
int x;  
int y;  
p1() {}
```

```
double x;  
p2() {}
```

在 p2 中对 x 的写入可能会覆盖 y!  
非常危险！

```
int x=7;  
p1() {}
```

```
int x;  
p2() {}
```

对 x 的引用将指向同一个已初始化的变量。

重要：链接器不进行类型检查。



# 练习题

## Module1

```
int main()  
{}
```

```
void main  
{}
```

```
int x;  
void main()  
{  
}
```

## Module2

```
int main  
int p2() {}
```

```
int main=1;  
int p2() {}
```

```
double x =1.0;  
int p2() {}
```

REF(main.1)-> REF( )  
REF(main.2)-> REF( )

REF(main.1)-> REF( )  
REF(main.2)-> REF( )

REF(x.1)-> REF( )  
REF(x.2)-> REF( )



# 类型不匹配示例

```
extern long int x;  
  
int main(int argc, char*  
argv[])  
{  
    printf("%ld\n", x);  
    return 0;  
}
```

*mismatch-main.c*

```
double x = 3.14;
```

*mismatch-variable.c*

- 编译时没有任何错误或警告
- 打印结果是什么？

```
bash > ./mismatch  
4614253070214989087
```



# 检测类型不匹配示例

```
extern long int x; mismatch.h
```

```
#include "mismatch.h"

int main(int argc, char* argv[])
{
    printf("%ld\n", x);
    return 0;
}
```

*mismatch-main.c*

```
#include "mismatch.h"

double x = 3.14;
```

*mismatch-variable.c*

- 现在我们从编译器（而不是链接器）得到一个错误：  
mismatch-variable.c:3:8: conflicting types for 'x'  
mismatch.h:1:17: previous declaration of 'x'



# 避免类型不匹配的规则

- 尽量避免使用全局变量
- 尽量使用 `static`
- 将所有非 `static` 的声明放在头文件中
  - 确保将头文件包含在所有相关的文件中
  - 包括定义这些符号的文件
- 在头文件中的声明上始终加上 `extern`
  - 对于函数声明来说这可能多余但无害
  - 避免没有 `extern` 的全局变量出现奇怪的行为
- 当函数不接受参数时，始终写上 `(void)`
  - `extern void no_args(void);`
  - 省略 `void` 意味着“我不声明这个函数的参数列表。”
    - 这会关闭参数类型检查！





# 在 .h 文件中使用 extern (#1)

c1.c

```
#include "global.h"

int f() { return g + 1; }
```

global.h

```
extern int g;
int f();
```

c2.c

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

int g = 0;

int main(int argc, char argv[])
{
    int t = f();
    printf("Calling f yields %d\n", t);
    return 0;
}
```



# 在 .h 文件中使用 extern (#2)

c1.c

```
#include "global.h"

int f() { return g + 1; }
```

```
extern int g;
static int init = 0;
```

global.h

```
#else
extern int g;
static int init = 0;
```

c2.c

```
#define INITIALIZE
#include "global.h"
#include <stdio.h>

int main(int argc, char* argv)
{
    if (init)
        int t = f();
    printf("Calling f yields %d\n", t);
    return 0;
}
```

```
int g = 23;
static int init = 1;
```



# 符号表 (Symbol Table)

- 符号表由汇编器构建，使用由编译器导出到汇编语言 .s 文件中的符号
- ELF 符号表包含在 .symbol 节中。它包含一个条目数组

source code of glibc/elf/elf.h

```
529 typedef struct
530 {
531     Elf64_Word st_name;      /* Symbol name(string tbl index) */
532     unsigned char st_info;   /* Symbol type and binding */
533     unsigned char st_other;  /* Symbol visibility */
534     Elf64_Section st_shndx;  /* Section index */
535     Elf64_Addr st_value;     /* Symbol value */
536     Elf64_Xword st_size;     /* Symbol size */
537 } Elf64_Sym;
```



# 符号表

|                  |
|------------------|
| ELF 头            |
| 段头表<br>(可执行文件必需) |
| .text 节          |
| .rodata 节        |
| .data 节          |
| .bss 节           |
| .symtab 节        |
| .rel.txt 节       |
| .rel.data 节      |
| .debug 节         |
| 节头表              |

0



# COMMON 节与 .bss 节的对比

- COMMON 和 .bss 之间的区别很微妙
- 现代版本的 GCC 使用以下约定将可重定位目标文件中的符号分配给 COMMON 和 .bss

| 变量类型   | 全局变量   | 静态变量  |
|--------|--------|-------|
| 未初始化   | COMMON | .bss  |
| 初始化为零  | .bss   | .bss  |
| 初始化为非零 | .data  | .data |

- 在可重定位目标文件中，变量可能位于 COMMON 或 .bss 中，但在可执行文件中都位于 .bss 中



# 符号表条目

全局

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

外部

```
int main()  
{  
    swap();  
    return 0;  
}
```

*m.c*

外部

链接器对 temp  
变量一无所知

全局

外部

局部

```
extern int buf[];
```

全局

```
int* bufp0 = &buf[0];  
static int* bufp1;
```

全局

```
void swap()  
{  
    int temp;
```

外部

```
    bufp1 = &buf[1];  
    temp = *bufp0;  
    *bufp0 = *bufp1;  
    *bufp1 = temp;  
}
```

*swap.c*



# 练习题

swap.o

| 符号    | .symtab条目 | 符号类型 | 定义模块 | 节 |
|-------|-----------|------|------|---|
| buf   |           |      |      |   |
| bufp0 |           |      |      |   |
| bufp1 |           |      |      |   |
| swap  |           |      |      |   |
| temp  |           |      |      |   |

.symtab条目： 是否在swap.o的.symtab中有符号表条目，有（是），无（否）

符号类型： 局部、全局、外部

模块： swap.o和m.o

节： .text, .data, .bss, COMMON



# swap.o

| 符号    | .symtab条目 | 符号类型 | 定义模块   | 节     |
|-------|-----------|------|--------|-------|
| buf   | 是         | 外部   | m.o    | .data |
| bufp0 | 是         | 全局   | swap.o | .data |
| bufp1 | 是         | 全局   | swap.o | .bss  |
| swap  | 是         | 全局   | swap.o | .text |
| temp  | 否         |      |        |       |

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

int main()
{
    swap();
    return 0;
}
```

*m.c*

```
extern int buf[];
int* bufp0 = &buf[0];
static int* bufp1;
void swap()
{
    int temp;

    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
```

*swap.c*





# 符号表条目

```
# objdump -r -d -t m.o | head -n 15
m.o:      file format elf64-x86-64
SYMBOL TABLE:
0000000000000000 1      df *ABS* 0000000000000000 m.c
0000000000000000 1      d  .text 0000000000000000 .text
0000000000000000 1      d  .data 0000000000000000 .data
0000000000000000 1      d  .bss  0000000000000000 .bss
0000000000000000 1      d  .note.GNU-stack0000000000000000 .note.GNU-stack
0000000000000000 1      d  .eh_frame 0000000000000000 .eh_frame
0000000000000000 1      d  .comment 0000000000000000 .comment
0000000000000000 g      0  .data 0000000000000008 buf
0000000000000000 g      F  .text 0000000000000015 main
0000000000000000      *UND* 0000000000000000 swap
```

```
# readelf -s m.o
Symbol table '.symtab' contains 11 entries:
Num:      Value              Size Type      Bind      Vis      Ndx Name
  0: 0000000000000000          0 NOTYPE    LOCAL    DEFAULT  UND
  1: 0000000000000000          0 FILE      LOCAL    DEFAULT  ABS m.c
  2: 0000000000000000          0 SECTION   LOCAL    DEFAULT    1
  3: 0000000000000000          0 SECTION   LOCAL    DEFAULT    3
  4: 0000000000000000          0 SECTION   LOCAL    DEFAULT    4
  5: 0000000000000000          0 SECTION   LOCAL    DEFAULT    6
  6: 0000000000000000          0 SECTION   LOCAL    DEFAULT    7
  7: 0000000000000000          0 SECTION   LOCAL    DEFAULT    5
  8: 0000000000000000          8 OBJECT    GLOBAL    DEFAULT    3 buf
  9: 0000000000000000         21 FUNC      GLOBAL    DEFAULT    1 main
 10: 0000000000000000          0 NOTYPE    GLOBAL    DEFAULT  UND swap
```



# 符号表条目

```
# objdump -r -d -t swap.o | head -n 15
```

```
swap.o:      file format elf64-x86-64
```

```
SYMBOL TABLE:
```

```
0000000000000000 1      df *ABS*  0000000000000000 swap.c
0000000000000000 1      d  .text  0000000000000000 .text
0000000000000000 1      d  .data  0000000000000000 .data
0000000000000000 1      d  .bss   0000000000000000 .bss
0000000000000000 1      d  .note.GNU-stack 0000000000000000 .note.GNU-stack
0000000000000000 1      d  .eh_frame 0000000000000000 .eh_frame
0000000000000000 1      d  .comment 0000000000000000 .comment
0000000000000000 g      0  .data  0000000000000008 bufp0
0000000000000000      *UND* 0000000000000000 buf
0000000000000008      0 *COM* 0000000000000008 bufp1
0000000000000000 g      F  .text  000000000000003c swap
```

```
# readelf -s swap.o | tail -n 11
```

|     |                  |    |         |        |         |     |        |
|-----|------------------|----|---------|--------|---------|-----|--------|
| 1:  | 0000000000000000 | 0  | FILE    | LOCAL  | DEFAULT | ABS | swap.c |
| 2:  | 0000000000000000 | 0  | SECTION | LOCAL  | DEFAULT | 1   |        |
| 3:  | 0000000000000000 | 0  | SECTION | LOCAL  | DEFAULT | 3   |        |
| 4:  | 0000000000000000 | 0  | SECTION | LOCAL  | DEFAULT | 5   |        |
| 5:  | 0000000000000000 | 0  | SECTION | LOCAL  | DEFAULT | 7   |        |
| 6:  | 0000000000000000 | 0  | SECTION | LOCAL  | DEFAULT | 8   |        |
| 7:  | 0000000000000000 | 0  | SECTION | LOCAL  | DEFAULT | 6   |        |
| 8:  | 0000000000000000 | 8  | OBJECT  | GLOBAL | DEFAULT | 3   | bufp0  |
| 9:  | 0000000000000000 | 0  | NOTYPE  | GLOBAL | DEFAULT | UND | buf    |
| 10: | 0000000000000008 | 8  | OBJECT  | GLOBAL | DEFAULT | COM | bufp1  |
| 11: | 0000000000000000 | 60 | FUNC    | GLOBAL | DEFAULT | 1   | swap   |



# 链接示例

```
int sum(int* a, int n);

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

int main(int argc, char** argv)
{
    int val = sum(array, 2);
    return val;
}                                     main.c
```

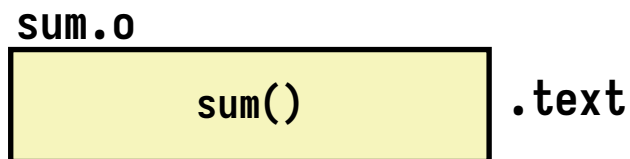
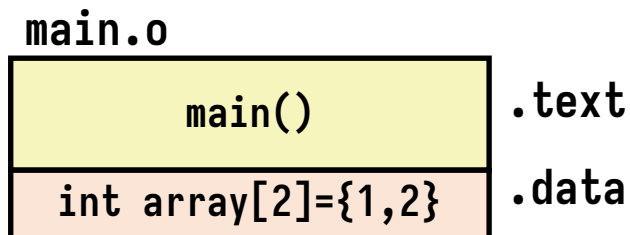
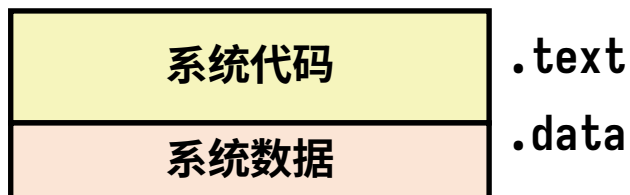
```
int sum(int* a, int n)
{
    int i, s = 0;

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

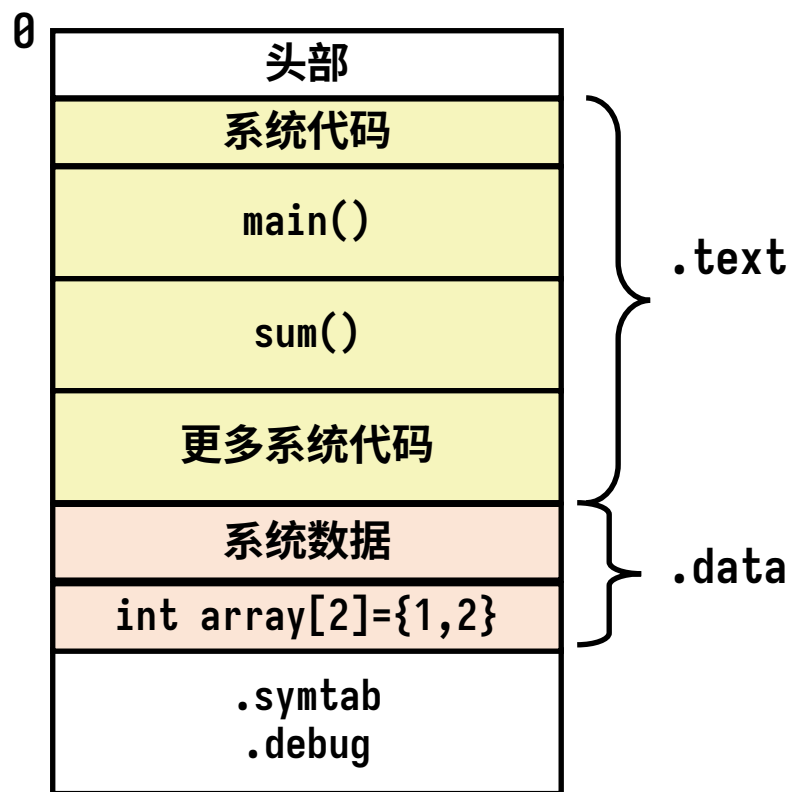


# 步骤 2: 重定位

## 可重定位目标文件



## 可执行目标文件





# 静态链接中的两步重定位

- 重定位段和符号定义
  - 将同类型的所有段**合并**到一个新的聚合段中
  - 分配运行时内存地址给：
    - 新的聚合段
    - 输入模块定义的每个段
    - 输入模块定义的每个符号
- 重定位段内的符号引用
  - 修改代码和数据段中的每个符号引用，使其指向正确的运行时地址
  - 依赖于可重定位模块中称为**重定位条目（relocation entries）**的数据结构



# 重定位条目

- 重定位条目从引用位置未知的引用中生成。

```
/* 带加数的重定位表条目（在 SHT_RELA 类型的段中）*/
```

```
660 typedef struct
661 {
662     Elf64_Addr r_offset;    /* Address */
663     Elf64_XWord r_info;    /* Relocation type and symbol index */
664     Elf64_Sxword r_addend; /* Addend */
665 } Elf64_Rela;
673 #define ELF64_R_SYM(i) ((i) >> 32)
674 #define ELF64_R_TYPE(i) ((i) & 0xffffffff)
```

- **ELF\_64\_R\_SYM** 标识引用应该指向的符号
- **ELF\_64\_R\_TYPE** 告诉链接器如何修改新的引用
- **r\_addend** 是用于某种重定位中偏移调整的常量



# 两种最基本的重定位类型

- R\_X86\_64\_PC32
  - 重定位使用 32 位 PC 相对地址（PC 相对引用）的引用。
- R\_X86\_64\_32/R\_X86\_64\_32S
  - 重定位使用 32 位绝对地址（绝对引用）的引用。

```
for each section s {
    foreach relocation entry r {
        refptr = s + r.offset; /* ptr to reference to be relocated */

        /* Relocate a PC-relative reference */
        if (r.type == R_X86_64_PC32) {
            refaddr = ADDR(s) + r.offset; /* ref's run-time address */
            *refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr);
        }

        /* Relocate an absolute reference */
        if (r.type == R_X86_64_32)
            *refptr = (unsigned) (ADDR(r.symbol) + r.addend);
    }
}
```



# 重定位条目

```
int array[2] = { 1, 2 };
int main(int argc, char** argv) {
    int val = sum(array, 2);
    return val;
}
```

main.c

# readelf -r main.o  
Relocation section '.rela.text' at offset 0x560 contains 2 entries:

| Offset       | Info         | Type          | Sym.Value        | Sym.Name+Addend |
|--------------|--------------|---------------|------------------|-----------------|
| 000000000015 | 00080000000a | R_X86_64_32   | 0000000000000000 | array + 0       |
| 00000000001a | 000a00000002 | R_X86_64_PC32 | 0000000000000000 | sum - 4         |

Relocation section '.rela.eh\_frame' at offset 0x590 contains 1 entries:

| Offset       | Info         | Type          | Sym.Value        | Sym.Name+Addend |
|--------------|--------------|---------------|------------------|-----------------|
| 000000000020 | 000200000002 | R_X86_64_PC32 | 0000000000000000 | .text + 0       |

偏移

类型

符号名称和加数

共有 3 个符号需要重定位





# 重定位条目（在 main.o 中）

```
int array[2] = { 1, 2 };
int main(int argc, char** argv) {
    int val = sum(array, 2);
    return val;
}
main.c
```

```
# readelf -r main.o
```

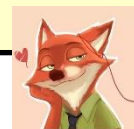
```
Relocation section '.rela.text' at offset 0x560 contains 2 entries:
```

| Offset         | Info          | Type          | Sym.Value        | Sym.Name+Addend |
|----------------|---------------|---------------|------------------|-----------------|
| 00000000000015 | 000800000000a | R_X86_64_32   | 0000000000000000 | array + 0       |
| 0000000000001a | 000a000000002 | R_X86_64_PC32 | 0000000000000000 | sum - 4         |

亲爱的链接器：

请在偏移量 0x15 处修补 .rela.text 段。按照以下步骤填入一个 32 位的值。当你确定了 .data 的地址时，计算 [array 的地址] + [加数，等于 0]，并将结果放置在指定位置。

此致，  
汇编器





# 重定位条目（在 main.o 中）

```
int array[2] = { 1, 2 };
int main(int argc, char** argv) {
    int val = sum(array, 2);
    return val;
}
main.c
```

```
# readelf -r main.o
```

```
Relocation section '.rela.text' at offset 0x560 contains 2 entries:
```

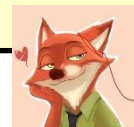
| Offset        | Info         | Type          | Sym.Value        | Sym.Name+Addend |
|---------------|--------------|---------------|------------------|-----------------|
| 0000000000015 | 00080000000a | R_X86_64_32   | 0000000000000000 | array + 0       |
| 000000000001a | 000a00000002 | R_X86_64_PC32 | 0000000000000000 | sum - 4         |

亲爱的链接器：

请在偏移量 0x1a 处修补 .rela.text 段。按照以下步骤填入一个 32 位的"PC 相对"值。当你确定了 sum 的地址时，计算 [sum 的地址] + [加数，等于 -4] - [段的地址 + 偏移量]，并将结果放置在指定位置。

此致，

汇编器





# 重定位条目（在 sum.o 中）

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

```
# readelf -r sum.o
Relocation section '.rela.eh_frame' at offset 0x4f8 contains 1 entries:
```

| Offset         | Info          | Type          | Sym.Value        | Sym.Name+Addend |
|----------------|---------------|---------------|------------------|-----------------|
| 00000000000020 | 0002000000002 | R_X86_64_PC32 | 0000000000000000 | .text + 0       |

偏移

类型

符号名称和加  
数

共有 1 个符号需要重定位 (.text)



# main.o 的原始目标文件

```
int array[2] = { 1, 2 };
int main(int argc, char** argv) {
    int val = sum(array, 2);
    return val;
}
```

*main.c*

0000000000000000 <main>:

Source: objdump -r -d main.o

|     |                       |        |                  |                    |
|-----|-----------------------|--------|------------------|--------------------|
| 0:  | 55                    | push   | %rbp             |                    |
| 1:  | 48 89 e5              | mov    | %rsp,%rbp        |                    |
| 4:  | 48 83 ec 20           | sub    | \$0x20,%rsp      |                    |
| 8:  | 89 7d ec              | mov    | %edi,-0x14(%rbp) |                    |
| b:  | 48 89 75 e0           | mov    | %rsi,-0x20(%rbp) |                    |
| f:  | be 02 00 00 00        | mov    | \$0x2,%esi       |                    |
| 14: | bf 00 00 00 00        | mov    | \$0x0,%edi       | # %edi = &array    |
| 15: | R_X86_64_32 array     |        |                  | # Relocation entry |
| 19: | e8 00 00 00 00        | callq  | 1e <main+0x1e>   | # sum()            |
| 1a: | R_X86_64_PC32 sum-0x4 |        |                  | # Relocation entry |
| 1e: | 89 45 fc              | mov    | %eax,-0x4(%rbp)  |                    |
| 21: | 8b 45 fc              | mov    | -0x4(%rbp),%eax  |                    |
| 24: | c9                    | leaveq |                  |                    |
| 25: | c3                    | retq   |                  |                    |



# sum.o 的原始目标文件

```
int sum(int *a, int n){
```

0000000000000000 <sum>:

|     |                      |      |                   |
|-----|----------------------|------|-------------------|
| 0:  | 55                   | push | %rbp              |
| 1:  | 48 89 e5             | mov  | %rsp,%rbp         |
| 4:  | 48 89 7d e8          | mov  | %rdi,-0x18(%rbp)  |
| 8:  | 89 75 e4             | mov  | %esi,-0x1c(%rbp)  |
| b:  | c7 45 fc 00 00 00 00 | movl | \$0x0,-0x4(%rbp)  |
| 12: | c7 45 f8 00 00 00 00 | movl | \$0x0,-0x8(%rbp)  |
| 19: | eb 1d                | jmp  | 38 <sum+0x38>     |
| 1b: | 8b 45 f8             | mov  | -0x8(%rbp),%eax   |
| 1e: | 48 98                | cltq |                   |
| 20: | 48 8d 14 85 00 00 00 | lea  | 0x0(,%rax,4),%rdx |
| 27: | 00                   |      |                   |
| 28: | 48 8b 45 e8          | mov  | -0x18(%rbp),%rax  |
| 2c: | 48 01 d0             | add  | %rdx,%rax         |
| 2f: | 8b 00                | mov  | (%rax),%eax       |
| 31: | 01 45 fc             | add  | %eax,-0x4(%rbp)   |
| 34: | 83 45 f8 01          | addl | \$0x1,-0x8(%rbp)  |
| 38: | 8b 45 f8             | mov  | -0x8(%rbp),%eax   |
| 3b: | 3b 45 e4             | cmp  | -0x1c(%rbp),%eax  |
| 3e: | 7c db                | j1   | 1b <sum+0x1b>     |
| 40: | 8b 45 fc             | mov  | -0x4(%rbp),%eax   |
| 43: | 5d                   | pop  | %rbp              |
| 44: | c3                   | retq |                   |



# 链接准备的一些要点

- 链接脚本可用于配置你的链接过程
- 可以指定各节的起始地址：
  - .text 节从 0xbabe00 开始
  - .data 节从 0xcafe00 开始
- 使用以下命令：
- `gcc a.lds -o m main.o sum.o`

## SECTIONS

```
{  
    .text 0x00BABE00:  
    {  
        *(.text)  
    }  
    . = ALIGN(0);  
    .data 0x00CAFE00:  
    {  
        *(.data)  
    }  
}
```

*a.lds*



**.text=0xbabe00**

```
0000000000000000 <main>:
0: 55
1: 48 89 e5
4: 48 83 ec 20
8: 89 7d ec
b: 48 89 75 e0
f: be 02 00 00 00
14: bf 00 00 00 00
19: e8 00 00 00 00
1e: 89 45 fc
21: 8b 45 fc
24: c9
25: c3
```

*main.o*

```
0000000000000000 <sum>:
0: 55
1: 48 89 e5
4: 48 89 7d e8
8: 89 75 e4
b: c7 45 fc 00 00 00 00
12: c7 45 f8 00 00 00 00
19: eb 1d
1b: 8b 45 f8
1e: 48 98
20: 48 8d 14 85 00 00 00
27: 00
28: 48 8b 45 e8
2c: 48 01 d0
2f: 8b 00
31: 01 45 fc
34: 83 45 f8 01
38: 8b 45 f8
3b: 3b 45 e4
3e: 7c db
40: 8b 45 fc
43: 5d
44: c3
```

*sum.o*

```
000000000000babe00 <_start>:
...
0000000000babf18 <main>:
babf18: 55
babf19: 48 89 e5
babf1c: 48 83 ec 20
babf20: 89 7d ec
babf23: 48 89 75 e0
babf27: be 02 00 00 00
babf2c: bf 10 fe ca 01
babf31: e8 0a 00 00 00
babf36: 89 45 fc
babf39: 8b 45 fc
babf3c: c9
babf3d: c3
0000000000babf40 <sum>:
babf40: 55
babf41: 48 89 e5
babf44: 48 89 7d e8
babf48: 89 75 e4
babf4b: c7 45 fc 00 00 00 00
babf52: c7 45 f8 00 00 00 00
babf59: eb 1d
babf5b: 8b 45 f8
babf5e: 48 98
babf60: 48 8d 14 85 00 00 00
babf67: 00
babf68: 48 8b 45 e8
babf6c: 48 01 d0
babf6f: 8b 00
babf71: 01 45 fc
babf74: 83 45 f8 01
babf78: 8b 45 f8
babf7b: 3b 45 e4
babf7e: 7c db
babf80: 8b 45 fc
babf83: 5d
babf84: c3
```

*executable*

```
Disassembly of section .data:
0000000000cafe00 <__data_start>:
...
0000000000cafe10 <array>:
cafe10: 01 00
cafe12: 00 00
cafe14: 00 00
```

**.data=0xcafe00**

**addr\_of\_array=0xcafe10**  
**Using the value 0xcafe10 to modify the content here**

**addr\_of\_main=0xbabf18**  
**addr\_of\_sum=0xbabf40**  
**offset = 0x1a**  
**addend = -4**

**refptr**  
**= 0xbabf18 + 0x1a**  
**= 0xbabf32**

**\*refptr(content)**  
**= 0xbabf40 - 4 - 0xbabf32**  
**= 0x0a**

4为常量？



0000000000babe00 <\_start>:

...

0000000000babf18 <main>:

babf18: 55  
babf19: 48 89 e5  
babf1c: 48 83 ec 20  
babf20: 89 7d ec  
babf23: 48 89 75 e0  
babf27: be 02 00 00 00  
babf2c: bf 10 fe ca 00  
babf31: e8 0a 00 00 00  
babf36: 89 45 fc  
babf39: 8b 45 fc  
babf3c: c9  
babf3d: c3

0000000000babf40 <sum>:

babf40: 55  
babf41: 48 89 e5  
babf44: 48 89 7d e8  
babf48: 89 75 e4  
babf4b: c7 45 fc 00 00 00 00  
babf52: c7 45 f8 00 00 00 00  
babf59: eb 1d  
babf5b: 8b 45 f8  
babf5e: 48 98  
babf60: 48 8d 14 85 00 00 00  
babf67: 00  
babf68: 48 8b 45 e8  
babf6c: 48 01 d0  
babf6f: 8b 00  
babf71: 01 45 fc  
babf74: 83 45 f8 01  
babf78: 8b 45 f8  
babf7b: 3b 45 e4  
babf7e: 7c db  
babf80: 8b 45 fc  
babf83: 5d  
babf84: c3

它因链接顺序而不同

0000000000babe00 <\_start>:

...

0000000000babf18 <sum>:

babf18: 55  
babf19: 48 89 e5  
babf1c: 48 89 7d e8  
babf20: 89 75 e4  
babf23: c7 45 fc 00 00 00 00  
babf2a: c7 45 f8 00 00 00 00  
babf31: eb 1d  
babf33: 8b 45 f8  
babf36: 48 98  
babf38: 48 8d 14 85 00 00 00  
babf3f: 00  
babf40: 48 8b 45 e8  
babf44: 48 01 d0  
babf47: 8b 00  
babf49: 01 45 fc  
babf4c: 83 45 f8 01  
babf50: 8b 45 f8  
babf53: 3b 45 e4  
babf56: 7c db  
babf58: 8b 45 fc  
babf5b: 5d  
babf5c: c3  
babf5d: 00 00

0000000000babf60 <main>:

babf60: 55  
babf61: 48 89 e5  
babf64: 48 83 ec 20  
babf68: 89 7d ec  
babf6b: 48 89 75 e0  
babf6f: be 02 00 00 00  
babf74: bf 10 fe ca 00  
babf79: e8 0a ff ff ff  
babf7e: 89 45 fc  
babf81: 8b 45 fc  
babf84: c9  
babf85: c3





# 库：打包一组函数

- 如何打包程序员常用的函数？
  - 数学运算、输入/输出、内存管理、字符串操作等
- 鉴于目前的链接器框架，这种做法会很尴尬：
  - 选项 1：将所有函数放入单个源文件
    - 程序员将**大型**目标文件链接到他们的程序中
    - 空间和时间效率低下
  - 选项 2：将每个函数放入单独的源文件
    - 程序员需要显式地将适当的二进制文件链接到他们的程序中
    - 效率更高，但对程序员来说比较繁琐

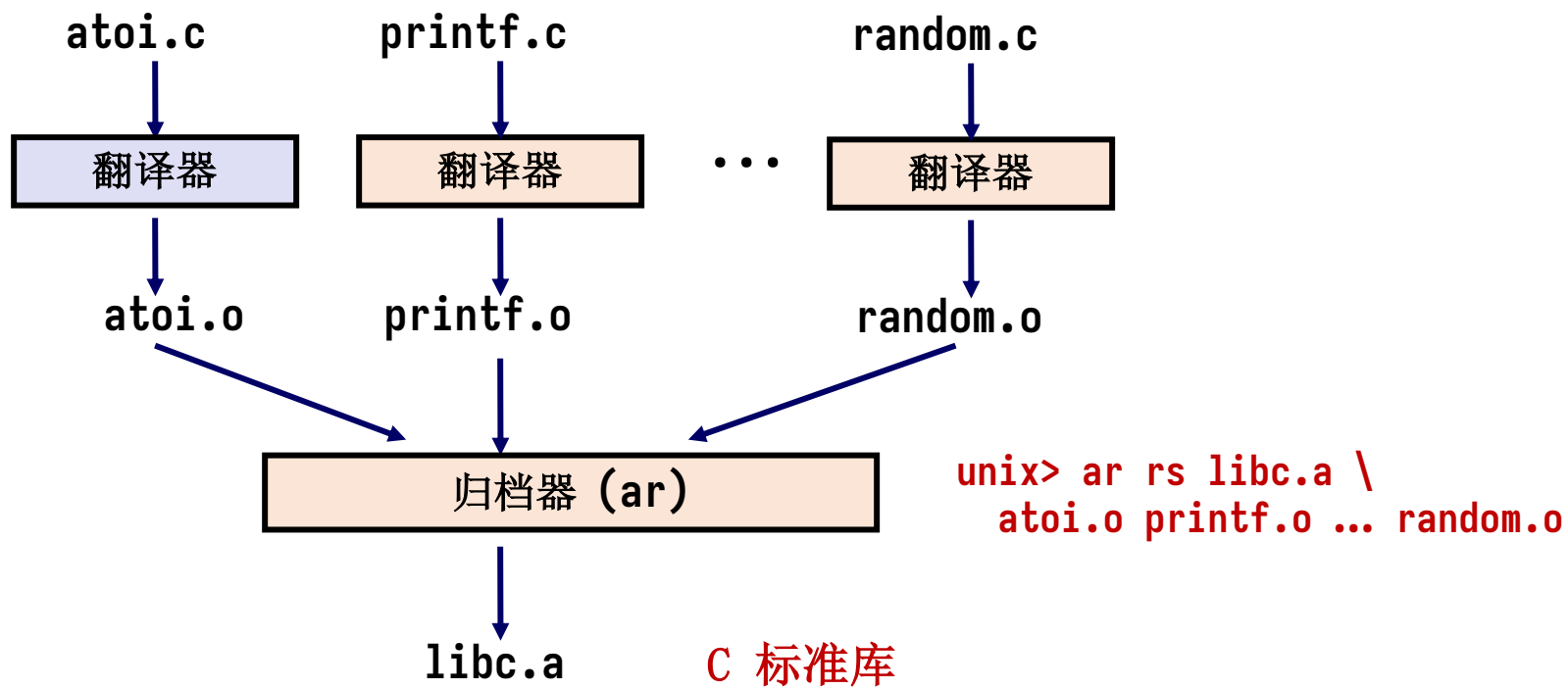


# 解决方案：库

- 静态库（.a 归档文件）
  - **合并**：将相关的可重定位目标文件**连接**成一个带索引的单一文件（称为归档文件）。
  - **解析**：增强链接器，使其尝试通过在一个或多个归档文件中查找符号来**解析未解析的外部引用**。
  - **链接**：如果归档成员文件解析了引用，则将其**链接**到可执行文件中。



# 创建静态库



- 归档器允许进行增量更新。
- 重新编译发生变化的函数，并在归档中替换相应的 .o 文件。



# 常用库

- **libc.a (C 标准库)**
  - 4.6 MB 的归档文件，包含 1496 个目标文件。
  - 功能包括：输入/输出、内存分配、信号处理、字符串处理、日期和时间、随机数生成、整数数学运算
- **libm.a (C 数学库)**
  - 2 MB 的归档文件，包含 444 个目标文件。
  - 功能：浮点数学运算（如 sin、cos、tan、log、exp、sqrt 等）

```
% ar -t /usr/lib/libc.a | sort
...
fork.o
...
fprintf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
fseek.o
fstab.o
...
```

```
% ar -t /usr/lib/libm.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
e_asinl.o
...
```



# 与静态库链接

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

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

int main(int argc, char** argv)
{
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n",
        z[0], z[1]);
    return 0;
}
main2.c
```

## libvector.a

```
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*

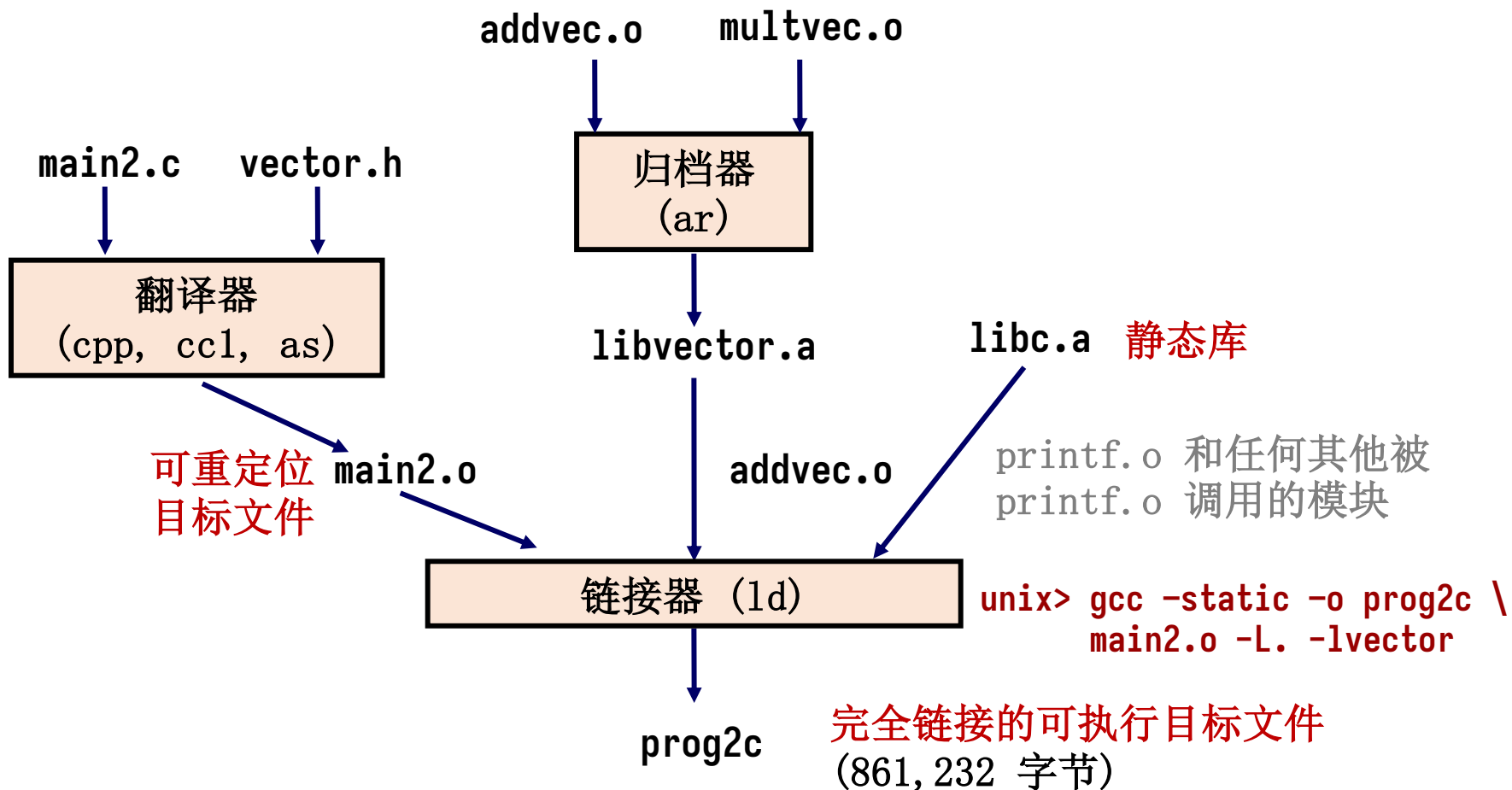
```
void multvec(int *x, int *y,
             int *z, int n)
{
    int i;

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

*multvec.c*



# 与静态库链接



"c" 表示"编译时"



# 使用静态库

- 链接器解析外部引用的算法：
  - 扫描：按命令行顺序扫描 .o 文件和 .a 文件。
  - 建立表：在扫描过程中，保持一个当前未解析引用的列表。
  - 尝试解析：当遇到每个新的 .o 或 .a 文件（obj）时，尝试将列表中的每个未解析引用与 obj 中定义的符号进行解析。
  - 表不空则报错：如果扫描结束时未解析列表中仍有条目，则报错。
- 问题：

```
unix> gcc -static -o prog2c -L. -lvector main2.o  
main2.o: In function `main':  
main2.c:(.text+0x19): undefined reference to `addvec'  
collect2: error: ld returned 1 exit status
```



# 现代解决方案：共享库

- 静态库有以下缺点：
  - 冗余：存储的可执行文件中存在重复（每个函数都需要 libc）
  - 运行的可执行文件中存在重复
  - 修改难：系统库的小型 bug 修复需要每个应用程序显式重新链接
    - 用 glibc 重新构建所有内容？
- 现代解决方案：共享库
  - 包含代码和数据的目标文件，可以动态地加载并链接到应用程序中，可以在加载时或运行时进行
  - 也称为：动态链接库、DLL、.so 文件

不见兔子不撒鹰





# 动态链接共享库

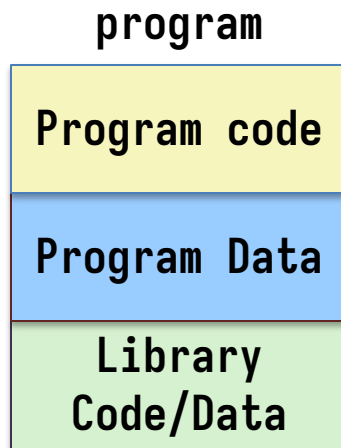
- 动态链接可以在可执行文件首次加载并运行时发生（加载时链接）。
  - Linux 常见情况，由动态链接器（ld-linux.so）自动处理。
  - 标准 C 库（libc.so）通常动态链接。
- 动态链接也可以在程序开始运行后发生（运行时链接）。
  - 在 Linux 中，这是通过调用 **dlopen()** 接口完成的。
    - 用于分发软件。
    - 高性能 Web 服务器。
    - 运行时库插入。
- 共享库例程可以被多个进程共享。
  - 当我们学习虚拟内存时会有更多相关内容
  - 可重入，多进程共享



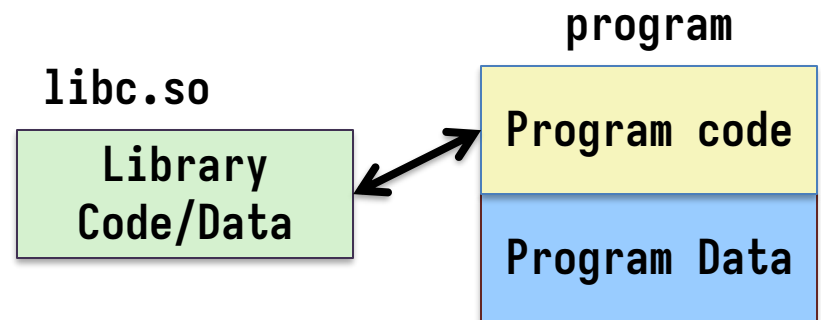
# 静态链接 vs 动态链接

- **静态链接**：在编译时，所需的代码和数据被复制到可执行文件中
- **动态链接**：在运行时，所需的代码和数据被链接到可执行文件中

静态链接：  
：



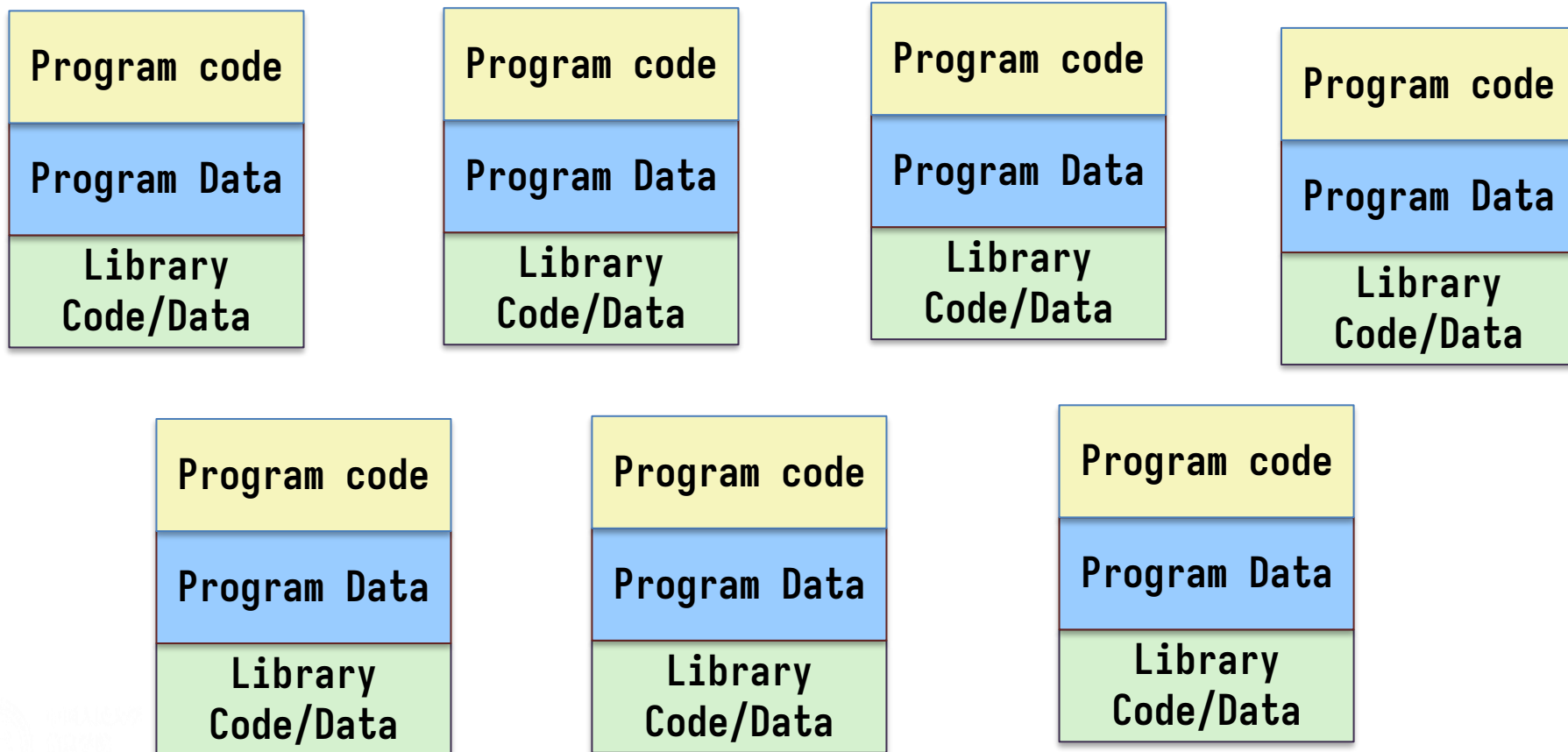
动态链接：





# 静态链接

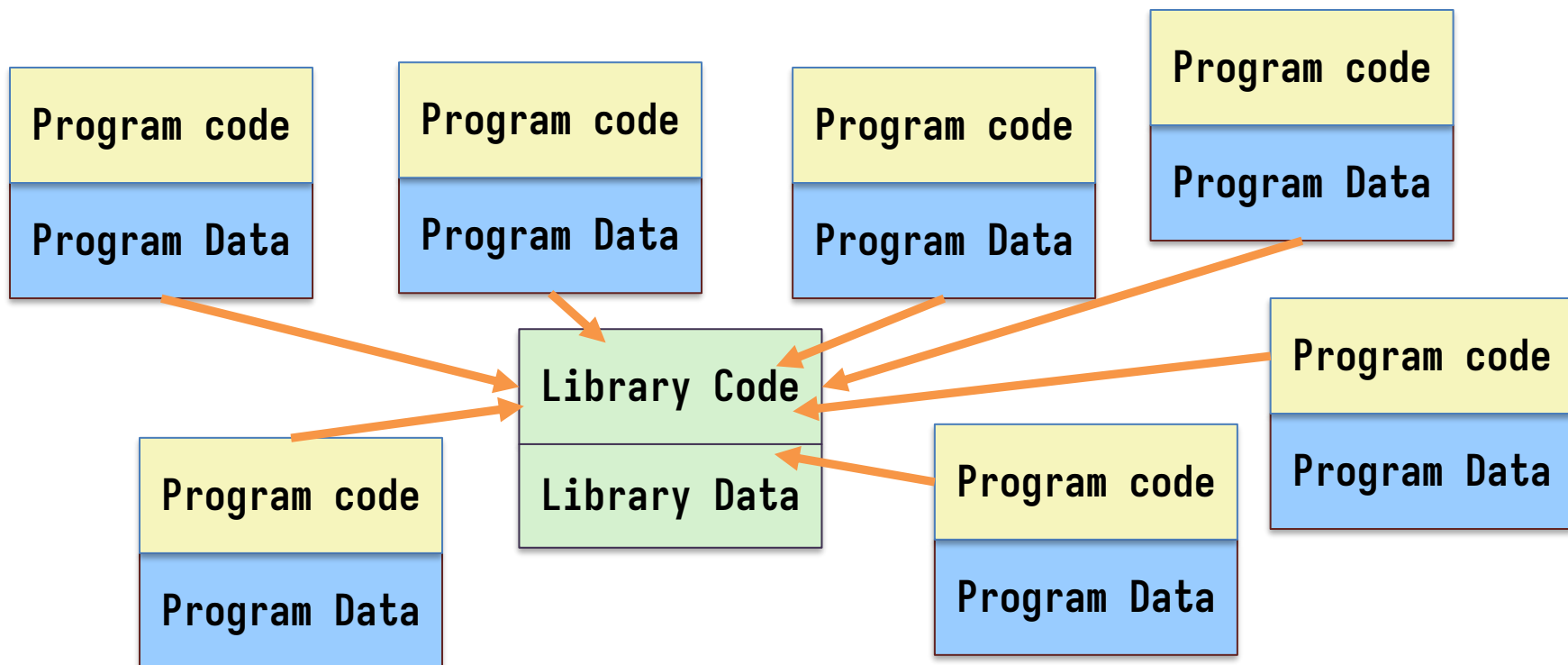
- 对于多个程序，可执行文件的大小会发生什么变化？





# 动态链接

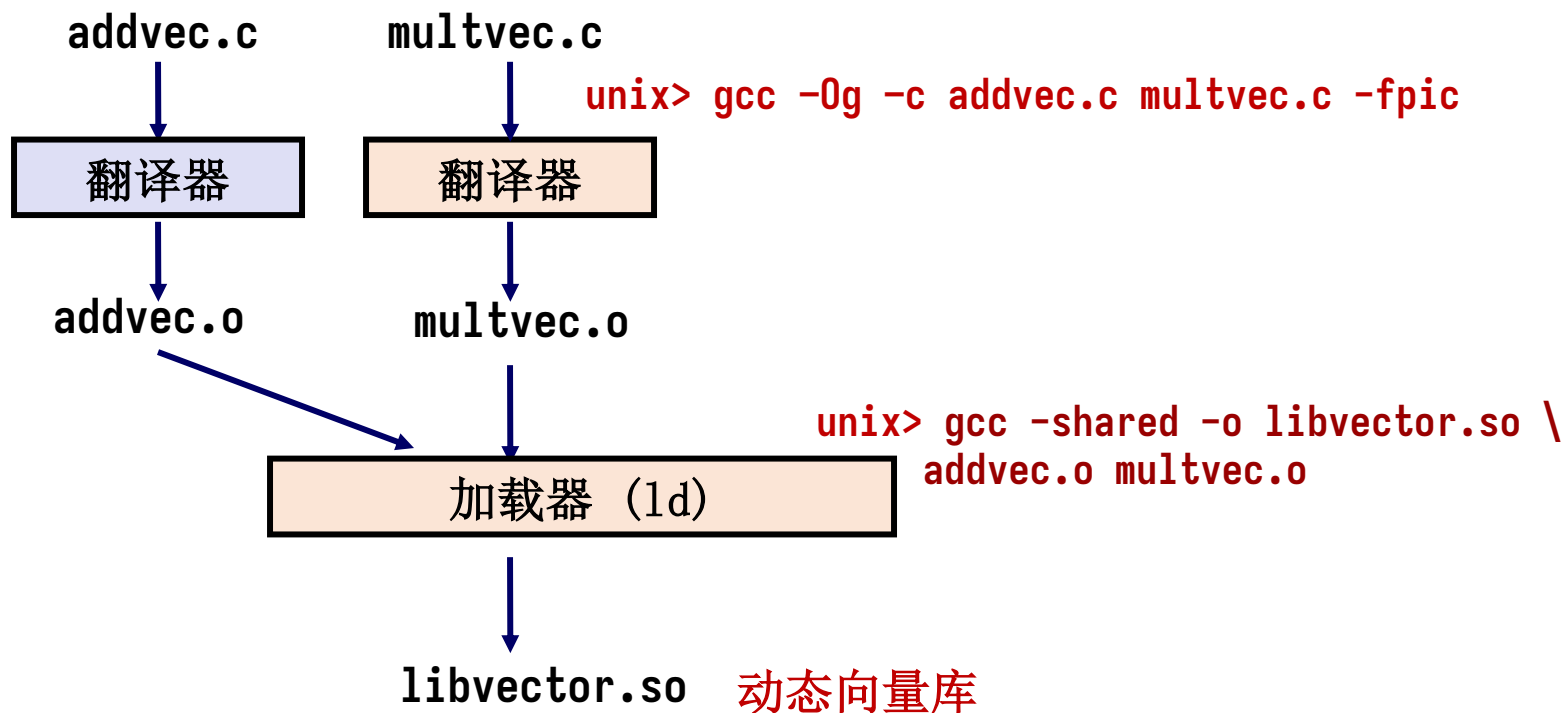
- 对于多个程序，可执行文件的大小会发生什么变化？



- 库的全局数据如何处理？**写时复制**（Copy on Write）

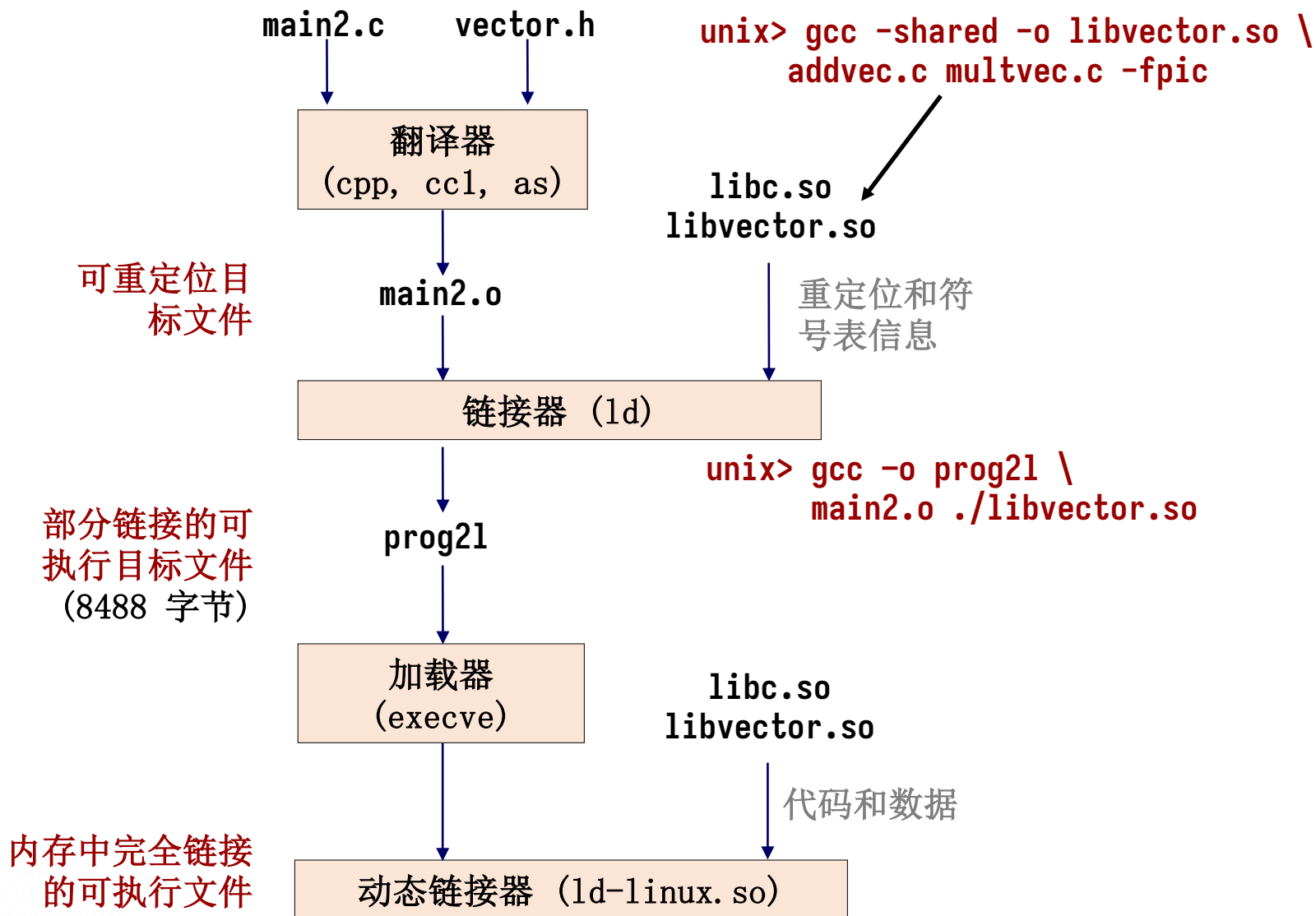


# 动态库示例





# 加载时动态链接





# ELF 文件的并行视图

- 程序头表/段（Segments）用于构建进程映像（执行程序）；可重定位文件不需要它
- 链接过程中使用的文件必须有节头表（Sections）；其他目标文件可能有也可能没有。

| ELF 头    |
|----------|
| 程序头表（可选） |
| 节 1      |
| ...      |
| 节 n      |
| ...      |
| 节头表（必需）  |

链接视图

| ELF 头    |
|----------|
| 程序头表（必需） |
| 段 1      |
| 段 2      |
| 段 3      |
| ...      |
| 节头表（可选）  |

执行视图



# 程序头

- ELF 可执行文件易于加载到内存中，连续的块映射到连续的内存段。这由程序头表描述

```
# objdump -x m
```

```
.....  
LOAD off 0x000000000001abe00 vaddr 0x0000000000babe00 paddr 0x0000000000babe00 align 2**12  
      filesz 0x000000000000032c memsz 0x000000000000032c flags r-x  
LOAD off 0x000000000002afe00 vaddr 0x0000000000cafe00 paddr 0x0000000000cafe00 align 2**12  
      filesz 0x0000000000000018 memsz 0x0000000000000020 flags rw-
```

off: 目标文件中的偏移量

vaddr/paddr: 内存地址

align: 对齐要求

filesz: 目标文件中的段大小

memsz: 内存中的段大小

flags: 运行时权限

- .text 从 0x00babe00 开始，.data 从 0x00cafe00 开始，正如我们之前在链接脚本中指定的那样
- 根据标志，.text 是可执行的，而 .data 是可写的





# 加载可执行目标文件

可执行目标文件

|               |
|---------------|
| ELF 头         |
| 程序头表（可执行文件必需） |
| .init 节       |
| .text 节       |
| .rodata 节     |
| .data 节       |
| .bss 节        |
| .symtab       |
| .debug        |
| .line         |
| .strtab       |
| 节头表（可重定位文件必需） |

0

0x00cafe00

0x00babe00

0

