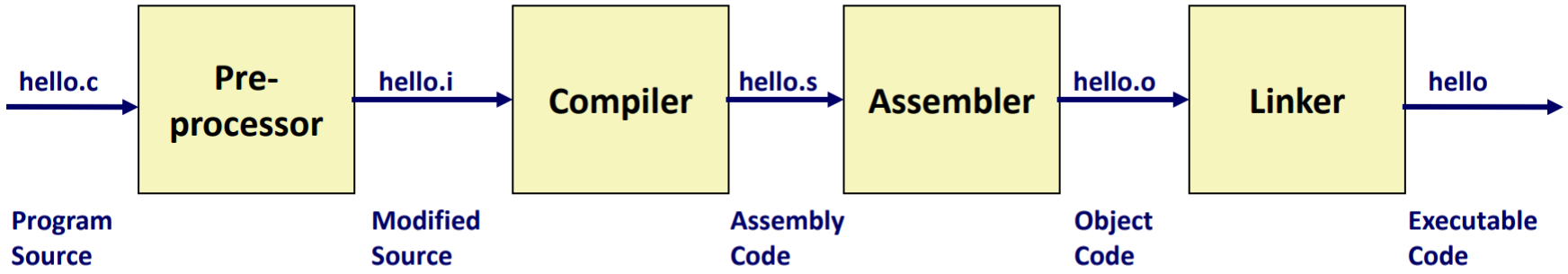
Linking

俞子杰

使用链接的原因

- 模块化
- 高效

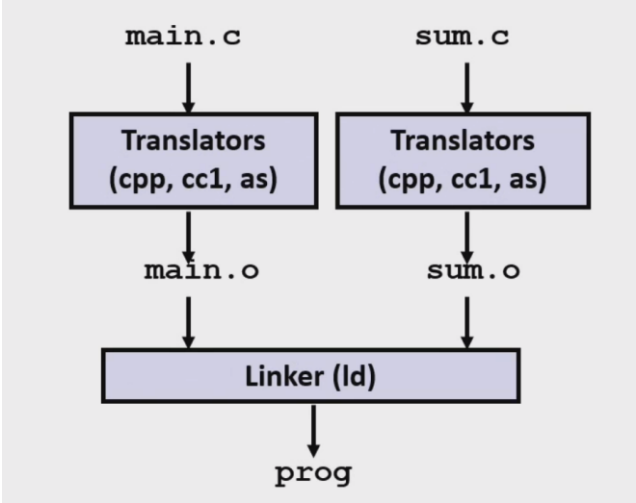
编译驱动




```
#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);
}
```

可重定向
目标程序



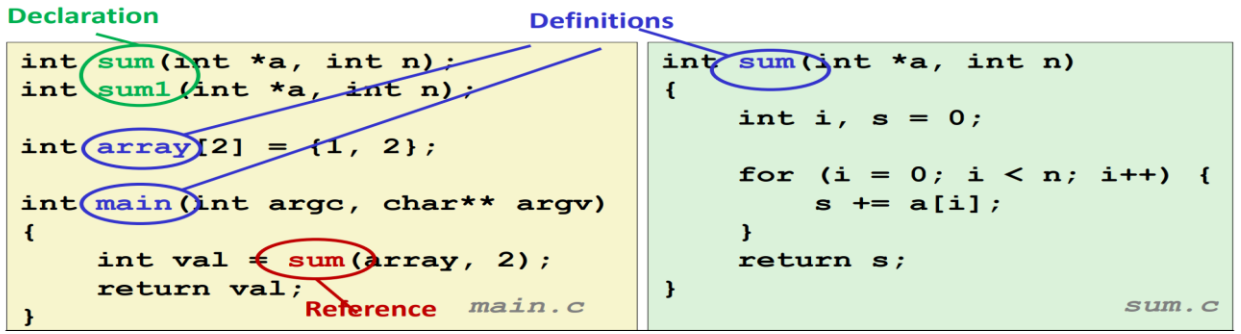
目标文件（模块）

- 可重定位目标文件(.o)
- 可执行目标文件(a.out)
- 共享库(.so)

静态链接

- 1、符号解析
- 2、重定位

声明/定义



- 当定义一个变量的时候，就包含了对该变量声明的过程，同时在内存张申请了一块内存空间。
- 如果在多个文件中使用相同的变量，为了避免重复定义，就必须将声明和定义分离开来。
- 定义是创建与名字关联的实体。声明是让名字为程序所知，当一个文件想要使用其他文件定义的某个变量，则必须包含对那个文件的声明。

- 函数和变量的声明不会分配内存, 但是定义会分配相应的内存空间
- 函数和变量的声明可以有很多次, 但是定义最多只能有一次
- 函数的声明和定义方式默认都是 extern 的, 即函数默认是全局的

static/extern

- 对于函数：
  - 默认extern，全局可见
  - static，仅当前文件可见
- 对于变量：
  - 全局变量
    - 定义/声明时，不加static为全局可见，加static为仅当前文件可见，定义时不可加extern（会警告）
    - extern为外部变量声明，若使用该声明说明已有此全局变量在其他文件定义，从而可以使用这一变量
  - 局部变量（函数内）
    - 作用域为函数内，在栈中
    - 可以使用static，但是作用域仍为函数内，只是不在栈中，作为静态数据

.h头文件

- 如果在.h中定义static全局变量x，则如果该头文件被一个文件包含，则该文件可以使用x
- 但是如果有两个文件同时包含该头文件，并使用x，虽然x会有一样的初始值，但是实际上存储在不同地址
- 可以理解成#include “.h文件”就是把头文件展开放入代码中

重复包含头文件

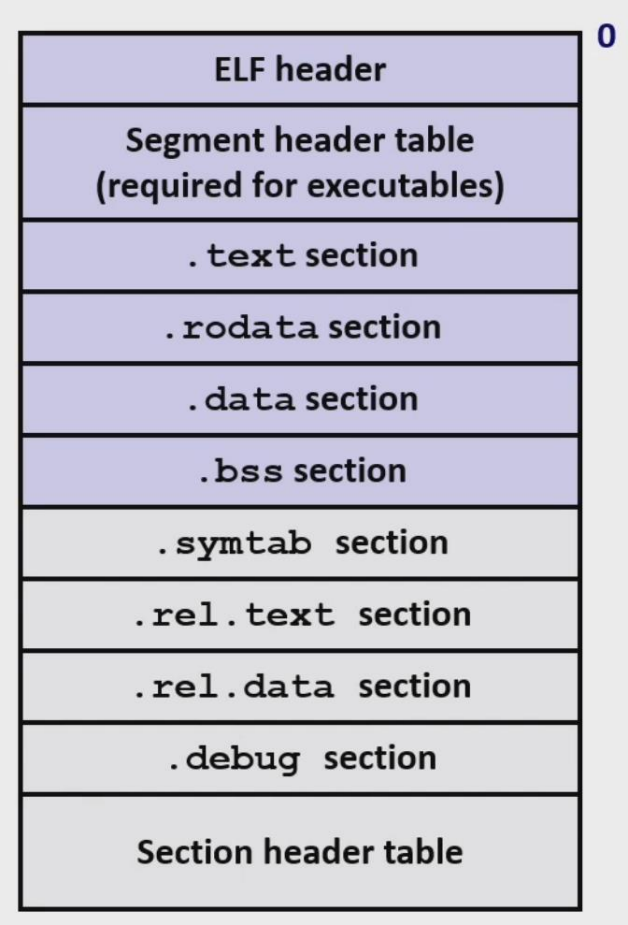
- 链接时发现不同.c文件包含相同的头文件会报错
- （重复包含stdio.h等不会报错）
- 在头文件中使用#ifndef
- 或者使用#pragma once

```
#ifndef _ADD_H_
#define _ADD_H_

int add(int, int);

#endif
```

ELF：（目标文件）可执行可链接格式

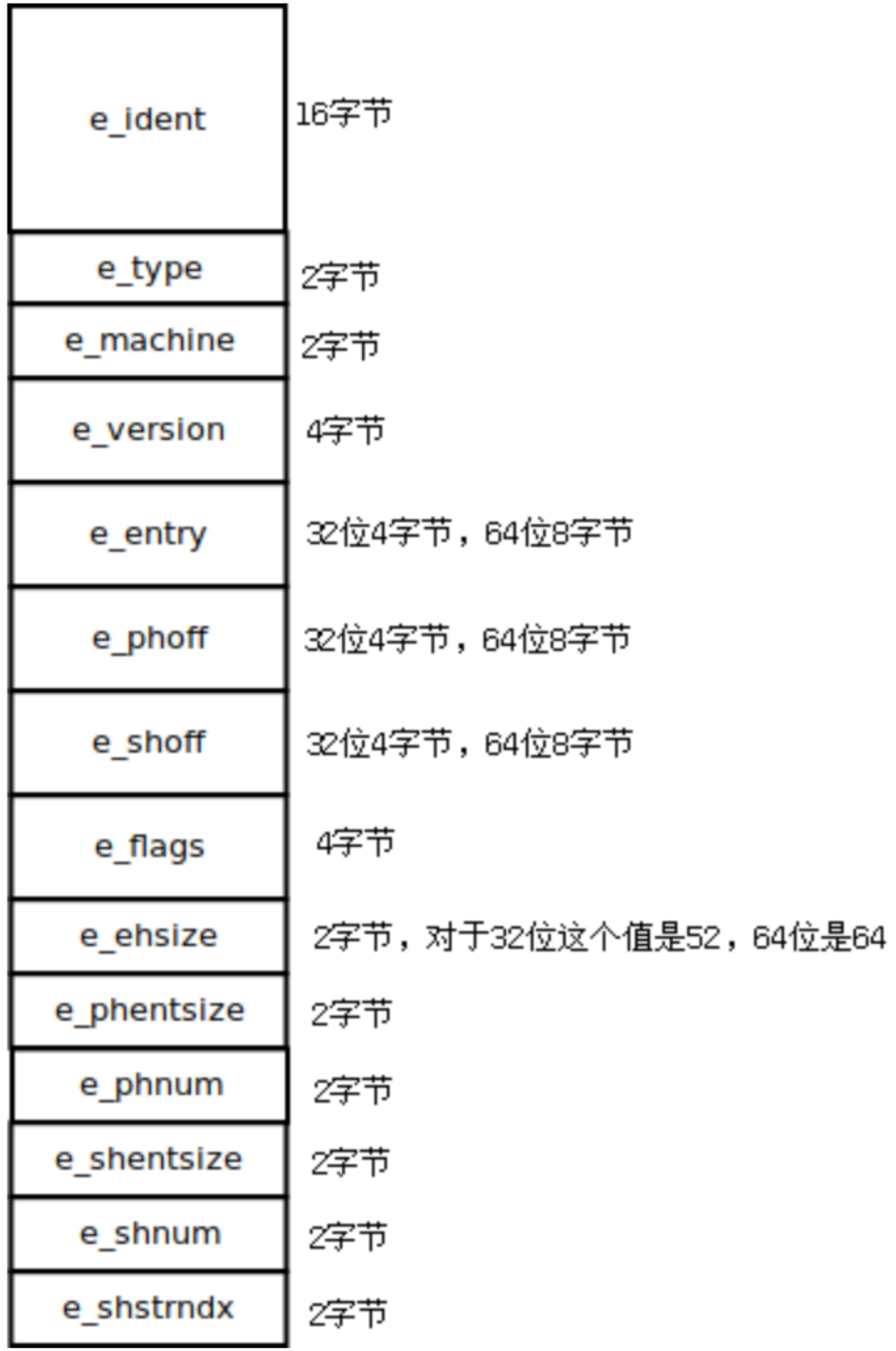


segment：段（执行）
section：节（链接）

- ELF header：下一页
- Segment header table：段头部表，执行时用
- .text：代码（函数）
- .rodata：read only data，如跳转表，字符串常量（printf）
- .data：已初始化的全局、static变量
- .bss：未初始化的static变量，初始化为0的全局、static
  - COMMON：未初始化的全局变量（伪节）（可执行没有，进入.bss）
- .symtab：符号表
- .rel.text：在重定位时，需要更改的.text节中的位置，如调用外部函数、引用全局变量的指令
- .rel.data：与上一个相似，重定位时需要更改的.data中的内容，如已初始化为外部定义函数地址、全局变量地址的全局变量
- .debug：调试符号表

ELF header

- e_ident：
  - 前四个字节为magic number，用于检查文件是否损坏（7F,'E','L','F')
  - 第五个：ELF文件是32还是64位的
  - 第六个：大端法还是小端法
  - 第七个：版本号（1）
  - 其他填充为0
- 其他包括ELF header的大小、目标文件的类型（可重定位、可执行、共享）、机器类型、节头部表的文件偏移、节头部表中条目的大小和数量



符号

- 符号与函数、全局变量、静态变量对应
- 符号表中三种符号：
  - 全局符号：本模块定义的全局变量、extern函数
  - 外部符号：其他模块定义的全局变量，extern声明全局变量
  - 局部符号：静态变量和函数（包括静态局部变量）

```
1   int f()
2   {
3       static int x = 0;
4       return x;
5   }
6
7   int g()
8   {
9       static int x = 1;
10      return x;
11  }
```

在这种情况中，编译器向汇编器输出两个不同名字的局部链接器符号。比如，它可以用 x.1 表示函数 f 中的定义，而用 x.2 表示函数 g 中的定义。

符号表（.symtab）

code/link/elfstructs.c

```
1  typedef struct {
2      int   name;      /* String table offset */
3      char  type:4,    /* Function or data (4 bits) */
4            binding:4; /* Local or global (4 bits) */
5      char  reserved;  /* Unused */
6      short section;   /* Section header index */
7      long  value;     /* Section offset or absolute address */
8      long  size;      /* Object size in bytes */
9  } Elf64_Symbol;
```

code/link/elfstructs.c

图 7-4 ELF 符号表条目。type 和 binding 字段每个都是 4 位

- binding：局部或全局
- Ndx：如果是数字则对应各个section（如main在.text中，array在.data中）
- 三个伪节：ABS（不重定位）、UNDEF（未定义的，外部符号）、COMMON（只在可重定位中有，可执行目标文件没有）
- 符号表包含的是函数、全局变量、静态变量；不包含非静态局部变量（栈中）

```
Num:    Value          Size Type    Bind   Vis      Ndx Name
  8: 0000000000000000    24 FUNC    GLOBAL DEFAULT    1 main
  9: 0000000000000000     8 OBJECT  GLOBAL DEFAULT    3 array
 10: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND sum
```

多重意义的全局符号

- 强弱符号:
  - 函数、已初始化的全局变量：强
  - 未初始化的全局变量：弱
- 规则:
  - 1、不能有多个同名强符号
  - 2、一个强符号和多个弱符号重名：选择强符号
  - 3、多个弱符号重名：任选一个

符号解析时COMMON的作用

- 遇到一个弱全局符号x，并不知道是否有其他的模块定义了x
- 此时先将x扔到COMMON中，让链接器决定

本题基于下列 m.c 及 swap.c 文件所编译生成的 m.o 和 swap.o，编译和运行在 Linux/X86-64 下使用 GCC 完成，编译过程未加优化选项。

```
//m.c
void myswap();
void func();
char buf[2] = {1, 2};
void *bufp0;
void *bufp1;
char temp;
int main(){
    func();
    ......//略去题目无关代码
    myswap(temp);
    ......//略去题目无关代码
    return 0;
}
```

```
//swap.c
extern int buf[];
char *bufp0;
......//略去题目无关代码
char *bufp1 = &buf[1];
void myswap(char temp){
    static int count = 0;
    ......//略去题目无关代码
    bufp0 = &buf[0];
    temp = *bufp0;
    ......//略去题目无关代码
    *bufp0 = *bufp1;
    *bufp1 = temp;
    ......//略去题目无关代码
    count++;
}
void func(){
    ......//略去题目无关代码
}
```

1）对于每个 swap.o 中定义和引用的符号，请用“是”或“否”指出它是否在模块 swap.o 的.symtab 节中存在符号表条目。如果存在条目，则请指出定义该符号的模块（swap.o 或 m.o）、符号类型（局部、全局或外部）以及该符号在所属的模块（“即该符号在该模块中被定义”）中所处的节；如果不存在条目，则请将该行后继空白处标记为“/”。

符号	.symtab 有条目?	符号类型	定义符号的模块	节
buf				
bufp0				
count				
func				
temp				

符号	.symtab 有条目?	符号类型	定义符号的模块	节
buf	有	外部	m.o	.data
bufp0	有	全局	swap.o	COMMON
count	有	局部	swap.o	.bss
func	有	全局	swap.o	.text
temp	无	/	/	/

有以下三个 c 文件 hd.h f1.c f2.c。使用 gcc -c f1.c f2.c; gcc f1.o f2.o 编译后得到可执行文件 a.out。回答以下问题。Part A 中涉及的符号所对应的变量已在代码中加粗。**本大题无需理解代码的含义。**

f1.c

```
#include "hd.h"
#include <stdio.h>
const int total = 1 << 30;
static int count = 0;
static Point pnt;
int iter;
int main() {
  for (iter = 0; iter < total; ++iter) {
    rand_point(&pnt);
    count += if_inside(&pnt);
  }
  printf("Integral on [0,1] is %lf.\n",
         1.0 * count / total);
}
```

hd.h

```
typedef struct {
  double x;
  double y;
} Point;
void rand_point(Point *);
int if_inside(Point *);
```

f2.c

```
#include "hd.h"
#include <stdlib.h>
#include <time.h>
void rand_point(Point *ptr) {
  static int seed = 0;
  if (!seed) {
    srand((unsigned)time(NULL));
    seed = 1;
  }
  ptr->x = 1.0 * rand() / RAND_MAX;
  ptr->y = 1.0 * rand() / RAND_MAX;
}
int if_inside(Point *p) {
  return 1 / (1 + p->x) >= p->y;
}
```

Part A. **（每个符号 1 分，共 5 分）** 请说明以下符号是否**在 a.out 的符号表**中。如果是，请进一步指出符号定义所在的节，可能的选择有.text、.data、.bss、.rodata、COM、UNDEF、ABS。

符号名	iter	pnt	Point	total	seed
在符号表中？**（填是/否）**					
定义所在节					

Part A

符号名	iter	pnt	Point	total	seed
是否在符号表中	是	是	否	是	否
定义所在节	**.bss**	**.bss**		**.rodata**	

```
u22000131124@icsdancer:~/tests$ readelf -s f2.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 f2.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    4 .bss
     4: 0000000000000000     4 OBJECT  LOCAL  DEFAULT    4 seed.0
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5 .rodata
     6: 0000000000000000   123 FUNC    GLOBAL DEFAULT    1 rand_point
     7: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND time
     8: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND srand
     9: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND rand
    10: 000000000000007b    65 FUNC    GLOBAL DEFAULT    1 if_inside
```

u2200013124 > tests > C f2.c > if_inside(Point *)

```c
#include "hd.h"
#include <stdlib.h>
#include <time.h>
static int seed=0;
void rand_point(Point *ptr) {
static int seed = 0;
if (!seed) {
srand((unsigned)time(NULL));
seed = 1;
}
ptr->x = 1.0 * rand() / RAND_MAX;
ptr->y = 1.0 * rand() / RAND_MAX;
}

int if_inside(Point *p) {
static int seed = 0;
return 1 / (1 + p->x) >= p->y;
}

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

bash - tests

```
     4: 00000000000010f0     0 FUNC    LOCAL  DEFAULT   16 deregister_tm_clones
     5: 0000000000001120     0 FUNC    LOCAL  DEFAULT   16 register_tm_clones
     6: 0000000000001160     0 FUNC    LOCAL  DEFAULT   16 __do_global_dtors_aux
     7: 0000000000004010     1 OBJECT  LOCAL  DEFAULT   26 completed.0
     8: 0000000000003da8     0 OBJECT  LOCAL  DEFAULT   22 __do_global_dtor[...]
     9: 00000000000011a0     0 FUNC    LOCAL  DEFAULT   16 frame_dummy
    10: 0000000000003da0     0 OBJECT  LOCAL  DEFAULT   21 __frame_dummy_in[...]
    11: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS f1.c
    12: 0000000000004024     4 OBJECT  LOCAL  DEFAULT   26 count
    13: 0000000000004030    16 OBJECT  LOCAL  DEFAULT   26 pnt
    14: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS f2.c
    15: 0000000000004040     4 OBJECT  LOCAL  DEFAULT   26 seed
    16: 0000000000004044     4 OBJECT  LOCAL  DEFAULT   26 seed.1
    17: 0000000000004048     4 OBJECT  LOCAL  DEFAULT   26 seed.0
    18: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
    19: 0000000000002168     0 OBJECT  LOCAL  DEFAULT   20 __FRAME_END__
    20: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS
    21: 0000000000003db0     0 OBJECT  LOCAL  DEFAULT   23 _DYNAMIC
    22: 0000000000002038     0 NOTYPE  LOCAL  DEFAULT   19 __GNU_EH_FRAME_HDR
    23: 0000000000003fa0     0 OBJECT  LOCAL  DEFAULT   24 _GLOBAL_OFFSET_TABLE_
    24: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_mai[...]
    25: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterT[...]
    26: 0000000000004000     0 NOTYPE  WEAK   DEFAULT   25 data_start
    27: 0000000000004010     0 NOTYPE  GLOBAL DEFAULT   25 _edata
```

函数、变量类型	.symtab	符号类型	节	强弱
extern函数（声明）	√	外部	.text/UNDEF	弱
extern函数（定义）	√	全局	.text	强
static函数	√	局部	.text	-
全局变量（定义）	√	全局	.data（初始化）	强
			COMMON（未初始化）	弱
			.bss（初始化为0）	强
extern全局变量	√	外部	UNDEF（本文件初始化报错）(考虑所有文件时看定义处，同上）	弱
static全局变量	√	局部	.data（初始化）	-
			.bss（未初始化、0）	-
static局部变量	√	局部	.data（初始化）	-
			.bss（未初始化、0）	-
非静态局部变量	×	-	-	-

*本表为可重定位目标文件，若为可执行文件，则COMMON进入.bss

与静态库链接

- 可作为链接器的输入，链接器只复制其中被引用的模块
- 以存档文件形式保存
- 解析引用：
  - E：合并形成可执行文件的文件集合
  - U：未解析符号集合
  - D：前面文件已定义符号集合
  - 对于输入文件f：
    - f为目标文件：加入E，更新U、D
    - f为存档文件：对其中每一个成员m都进行与U中符号的匹配，如果m中有定义，则将U中该元素转至D
    - 结束后U非空，则输出错误并停止；否则合并E中文件

静态库与目标文件先后顺序

练习题 7.3 a和b表示当前目录中的目标模块或者静态库，而a→b表示a依赖于b，也就是说b定义了一个被a引用的符号。对于下面每种场景，请给出最小的命令行(即一个含有最少数量的目标文件和库参数的命令)，使得静态链接器能解析所有的符号引用。

A. p.o → libx.a

B. p.o → libx.a → liby.a

C. p.o → libx.a → liby.a 且 liby.a → libx.a → p.o

重定位

- 1、将相同类型的节合并成同一个节，并将运行时内存地址赋值给新的节，使程序的每条指令和每个全局变量都有唯一的运行时内存地址。
- 2、通过重定位条目修改.text和.data(.rel.text/.rel.data)

重定位条目

code/link/elfstructs.c

```
1  typedef struct {
2      long offset;    /* Offset of the reference to relocate */
3      long type:32,   /* Relocation type */
4           symbol:32; /* Symbol table index */
5      long addend;    /* Constant part of relocation expression */
6  } Elf64_Rela;
```

code/link/elfstructs.c

- offset:需要修改的位置的偏移（对于当前节）
- type：
  - R_X86_64_PC32：重定位一个使用32位PC相对地址的引用
  - R_X86_64_32：重定位一个使用32位绝对地址的引用
- symbol：修改之后指向的符号地址
- addend：常数，计算地址的偏移量

重定位算法

code/link/main-relo.d

```
1  0000000000000000 <main>:
2     0:   48 83 ec 08              sub    $0x8,%rsp
3     4:   be 02 00 00 00           mov    $0x2,%esi
4     9:   bf 00 00 00 00           mov    $0x0,%edi        %edi = &array
5                        a: R_X86_64_32 array              Relocation entry
6     e:   e8 00 00 00 00           callq  13 <main+0x13>   sum()
7                        f: R_X86_64_PC32 sum-0x4          Relocation entry
8    13:   48 83 c4 08              add    $0x8,%rsp
9    17:   c3                       retq
```

code/link/main-relo.d

- 对于该算法，我们已知重定位条目，以及所有节、所引用全局符号的重定位后的地址

- 需要修改的地址：s+r.offset；地址的内存地址：ADDR(s)+r.offset
  - s为所在节，ADDR()为取地址
- 相对情况下，如call，跳转目标到当前距离为PC相减，但是当前与下一条PC相距4，所以加上r.addend=-4

```
1  foreach section s {
2      foreach relocation entry r {
3          refptr = s + r.offset;  /* ptr to reference to be relocated */
4
5          /* Relocate a PC-relative reference */
6          if (r.type == R_X86_64_PC32) {
7              refaddr = ADDR(s) + r.offset; /* ref's run-time address */
8              *refptr = (unsigned) (ADDR(r.symbol) + r.addend - refaddr);
9          }
10
11         /* Relocate an absolute reference */
12         if (r.type == R_X86_64_32)
13             *refptr = (unsigned) (ADDR(r.symbol) + r.addend);
14     }
15 }
```

10. 对如下两个 C 程序，用 gcc 生成对应的 .o 模块，链接在一起得到 a.out 可执行程序。则下列说法正确的是：

```c
// main.c
#include <stdio.h>
static int a;
int main() {
  int *func();
  printf("%ld\n", func() - &a);
  return 0;
}
```

```c
// util.c
int a = 0;
int *func() {
  return &a;
}
```

A 在 main.o 中，符号 a 位于.COMMON 伪节。

B 在 util.o 中，符号 a 位于.COMMON 伪节。

C 无论怎样链接和运行 a.out，输出的结果都一样，但必不为 0。

D 以上说法都不正确。

有以下三个 c 文件 hd.h f1.c f2.c。使用 gcc -c f1.c f2.c; gcc f1.o f2.o 编译后得到可执行文件 a.out。回答以下问题。Part A 中涉及的符号所对应的变量已在代码中加粗。**本大题无需理解代码的含义。**

f1.c

```
#include "hd.h"
#include <stdio.h>
const int total = 1 << 30;
static int count = 0;
static Point pnt;
int iter;
int main() {
  for (iter = 0; iter < total; ++iter) {
    rand_point(&pnt);
    count += if_inside(&pnt);
  }
  printf("Integral on [0,1] is %lf.\n",
         1.0 * count / total);
}
```

hd.h

```
typedef struct {
  double x;
  double y;
} Point;
void rand_point(Point *);
int if_inside(Point *);
```

f2.c

```
#include "hd.h"
#include <stdlib.h>
#include <time.h>
void rand_point(Point *ptr) {
  static int seed = 0;
  if (!seed) {
    srand((unsigned)time(NULL));
    seed = 1;
  }
  ptr->x = 1.0 * rand() / RAND_MAX;
  ptr->y = 1.0 * rand() / RAND_MAX;
}
int if_inside(Point *p) {
  return 1 / (1 + p->x) >= p->y;
}
```

Part B. **(每空 1 分，共 3 分)** 使用 objdump -dx f1.o f2.o 看到如下几条代码。**这里可以将重定位类型 R_X86_64_PLT32 和 R_X86_64_PC32 同等看待。**

```
# objdump 重定位条目格式:
#            OFFSET: TYPE              VALUE
# e.g.           18: R_X86_64_PLT32    rand_point-0x4
# 所有数值均以十六进制表示
# f1.o
0000000000000000 <main>:
... # 省略无关代码
  17: e8 00 00 00 00.         callq  1c <main+0x1c>
            18: R_X86_64_PLT32      rand_point-0x4
  1c: 48 8d 3d 00 00 00 00    lea 0x0(%rip),%rdi
            1f: R_X86_64_PC32       .bss+0xc
  23: e8 00 00 00 00          callq  28 <main+0x28>
            24: R_X86_64_PLT32      if_inside-0x4
  28: 89 c2                   mov %eax,%edx
... # 省略无关代码
# f2.o
0000000000000000 <rand_point>:
... # 省略无关代码
0000000000000074 <if_inside>:
... # 省略无关代码
```

据此可以确定 <main+0x1f> 处的重定位条目是针对符号_____(**填写符号名，不要填写 .bss 这个节名**)的重定位。同时该符号定义的位置在 f1.o 中相对于 .bss 节的偏移量是 0x_____。

现已知 a.out 文件中 <main+0x17> 行变成

```
11a1: e8 69 00 00 00          callq  <rand_point>
```

那么 a.out 中 <main+0x23> 行将变成

```
11ad: e8 __________           callq  <if_inside>
```

pnt; 10; d1 00 00 00