第四章 程序的链接

张宇

一个典型的程序转换处理过程

经典的“ hello.c ”C-源程序

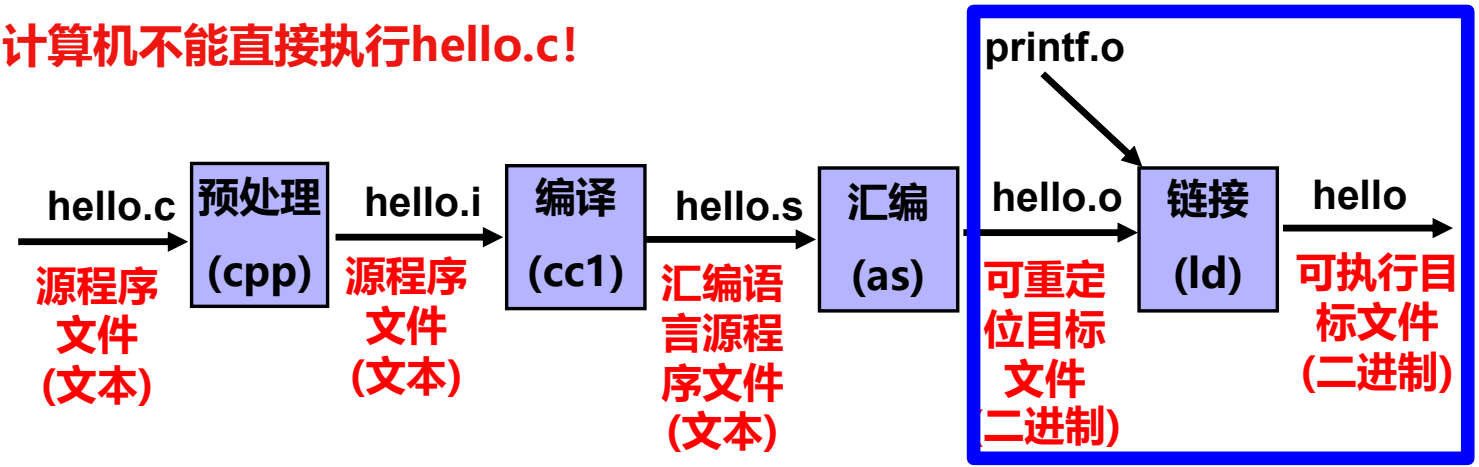
```
#include <stdio.h>

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

hello.c的ASCII文本表示

```
# i n c l u d e <sp> < s t d i o .
35 105 110 99 108 117 100 101 32 60 115 116 100 105 111 46
h > \n \n i n t <sp> m a i n ( ) \n {
104 62 10 10 105 110 116 32 109 97 105 110 40 41 10 123
\n <sp> <sp> <sp> <sp> p r i n t f ( " h e l
10 32 32 32 32 112 114 105 110 116 102 40 34 104 101 108
l o , <sp> w o r l d \ n " ) ; \n }
108 111 44 32 119 111 114 108 100 92 110 34 41 59 10 125
```

功能：输出“hello,world”

计算机不能直接执行hello.c!



可执行文件的链接生成

- **主要教学目标**
  - 掌握链接器是如何工作的
- **包括以下内容**
  - 链接概念（什么是程序链接、链接的意义与过程）
  - 可重定位目标文件格式、可执行目标文件格式
  - 符号、符号表
  - 符号解析和重定位过程
  - 动态链接

程序的链接

分以下三个部分介绍

- 第一讲：目标文件格式
  - 什么是程序链接、链接的意义与过程
  - ELF目标文件格式、可重定位目标文件格式和可执行目标文件格式
- 第二讲：符号解析与重定位
  - 什么是符号表、怎么进行符号解析
  - 与静态库的链接
  - 重定位所需信息和重定位过程
  - 可执行文件的加载
- 第三讲：动态链接
  - 动态链接的特性
  - 程序加载时的动态链接、程序运行时的动态链接
  - 位置无关代码

程序的链接

分以下三个部分介绍

- 第一讲：目标文件格式
  - 什么是程序链接、链接的意义与过程
  - ELF目标文件格式、可重定位目标文件格式和可执行目标文件格式
- 第二讲：符号解析与重定位
  - 什么是符号表、怎么进行符号解析
  - 与静态库的链接
  - 重定位所需信息和重定位过程
  - 可执行文件的加载
- 第三讲：动态链接
  - 动态链接的特性
  - 程序加载时的动态链接、程序运行时的动态链接
  - 位置无关代码

链接：

在将高级语言编译转换成了机器代码之后，还需要最后一步：将所有关联到的可重定位目标文件（包括用到的标准库函数目标文件），按照某种形式组合在一起，生成一个具有统一地址空间、可被加载到存储器直接执行的可执行目标文件

—— 这种**将一个程序的所有关联模块对应的可重定位目标文件结合在一起，以形成一个可执行目标文件的过程**称为链接

- 在早期计算机系统中，链接靠手工完成，
- 在现代计算机系统中，链接由专门的链接程序（linker，链接器）完成

思考：为啥不一次性就把程序所有代码都写在一个文件里面，而是写在多个文件里，再最后费劲地进行链接？

使用链接的好处：**支持模块化**

（1）一个程序可以被划分成多个模块，分别由不同程序员进行编辑、编译，从而缩短程序开发周期

（2）允许将常用的函数用于构建公共函数库（例如：数学库，标准C库），供不同程序员进行重用，以实现共享

（3）可以支持增量编译。例如，如果有部分源程序文件被修改，我们只需重新编译被修改的源程序文件，然后重新链接，提高程序修改后的编译效率

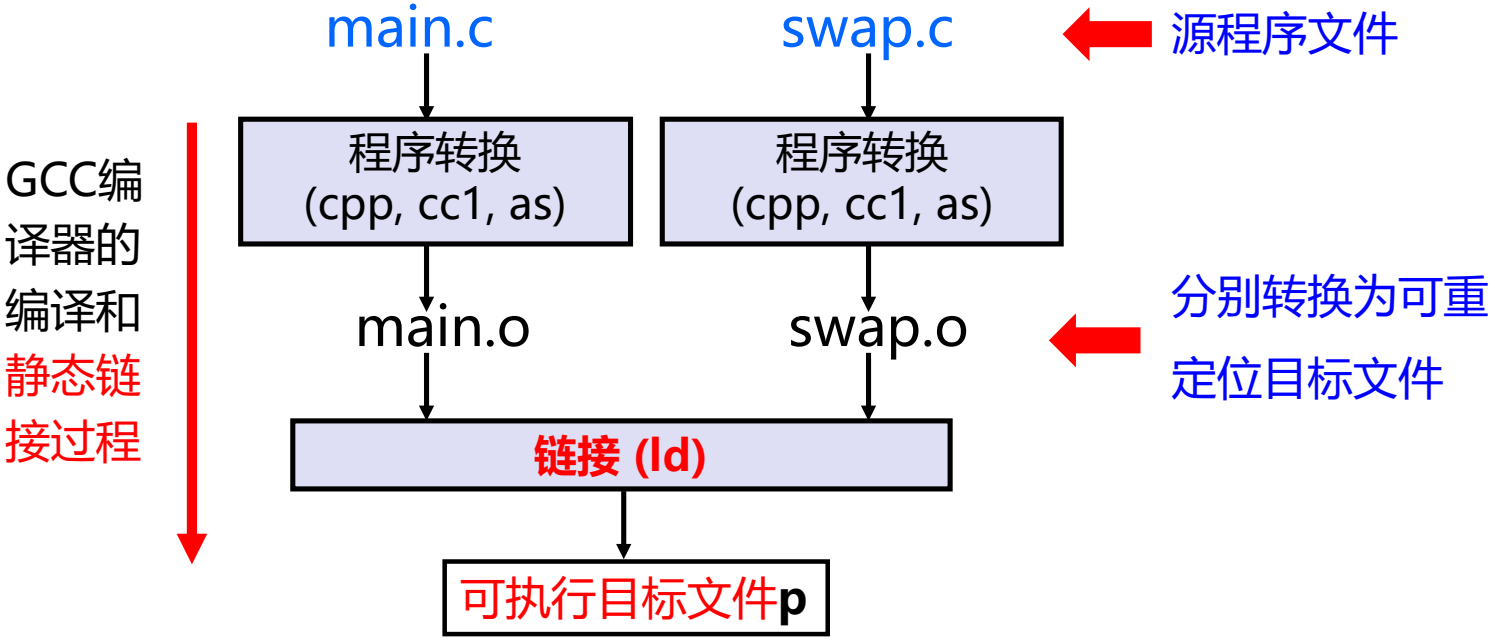
一个C语言程序链接过程举例

main.c

```
int buf[2] = {1, 2};
void swap();
int main( )
{
      swap( );
      return 0;
}
```

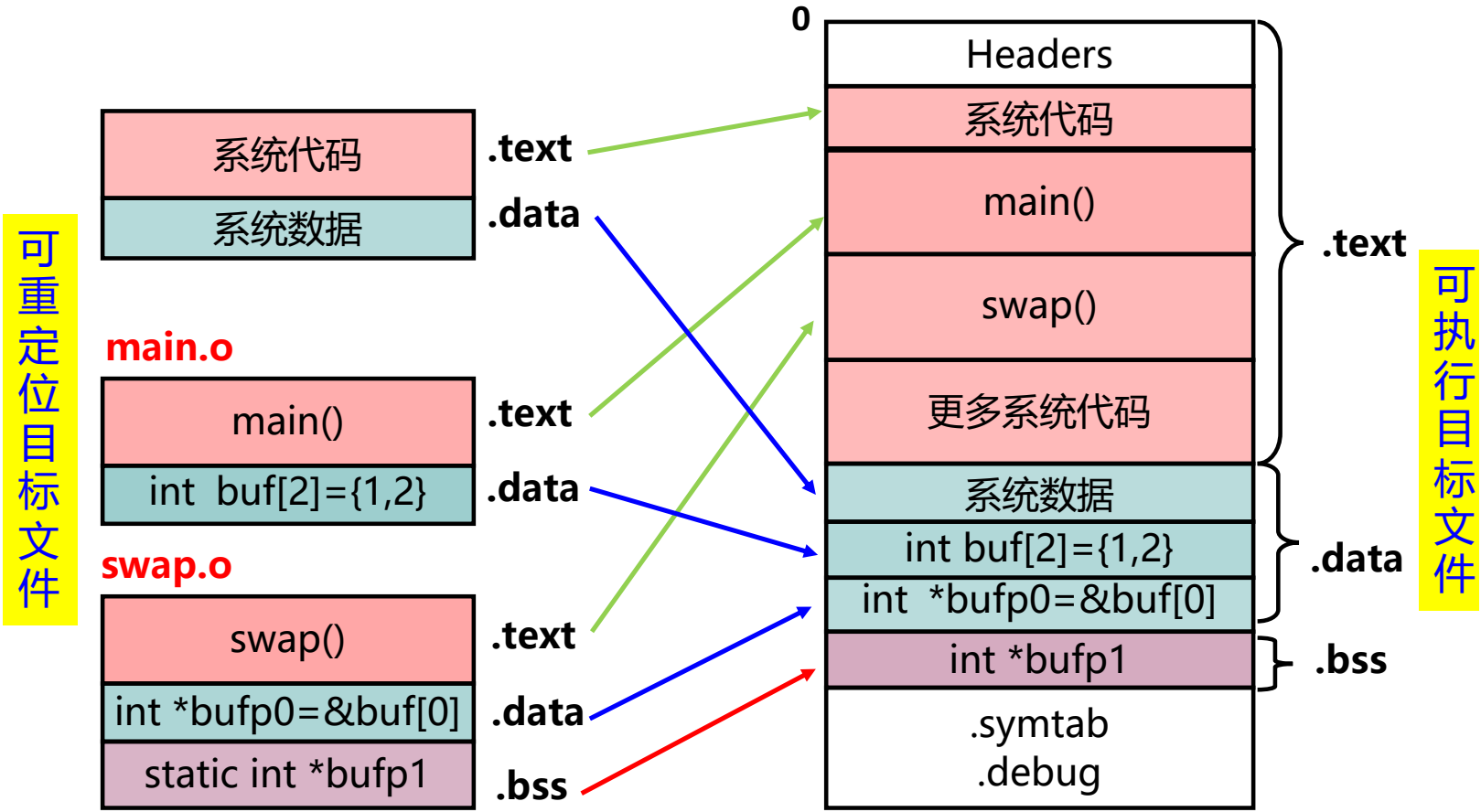
swap.c

```
extern int buf[];
int *bufp0 = &buf[0];
static int *bufp1;
void swap()
{
     int temp;
     bufp1 = &buf[1];
     temp = *bufp0;
     *bufp0 = *bufp1;
     *bufp1 = temp;
}
```



链接过程的本质：

- 将涉及到的所有可重定位目标文件中相同的节合并，形成具有统一虚拟地址空间编址的可执行目标文件。



目标文件

```
/* main.c */
int add(int, int);
int main( )
{
   return add(20, 13);
}
```

可重定位目标文件

objdump -d test.o

```
00000000 <add>:
   0:   55              push  %ebp
   1:   89 e5           mov   %esp, %ebp
   3:   83 ec 10        sub   $0x10, %esp
   6:   8b 45 0c        mov    0xc(%ebp), %eax
   9:   8b 55 08        mov    0x8(%ebp), %edx
   c:   8d 04 02        lea    (%edx,%eax,1), %eax
   f:   89 45 fc        mov    %eax, -0x4(%ebp)
  12:   8b 45 fc        mov    -0x4(%ebp), %eax
  15:   c9              leave
  16:   c3              ret
```

```
/* test.c */
int add(int i, int j)
{
    int x = i + j;
    return x;
}
```

可执行目标文件

objdump -d test

```
080483d4 <add>:
80483d4:   55              push  %ebp
80483d5:   89 e5           mov   %esp, %ebp
80483d7:   83 ec 10        sub   $0x10, %esp
80483da:   8b 45 0c        mov    0xc(%ebp), %eax
80483dd:   8b 55 08        mov    0x8(%ebp), %edx
80483e0:   8d 04 02        lea    (%edx,%eax,1), %eax
80483e3:   89 45 fc        mov    %eax, -0x4(%ebp)
80483e6:   8b 45 fc        mov    -0x4(%ebp), %eax
80483e9:   c9              leave
80483ea:   c3              ret
```

链接操作的步骤

- **Step 1. 符号解析（Symbol resolution）**

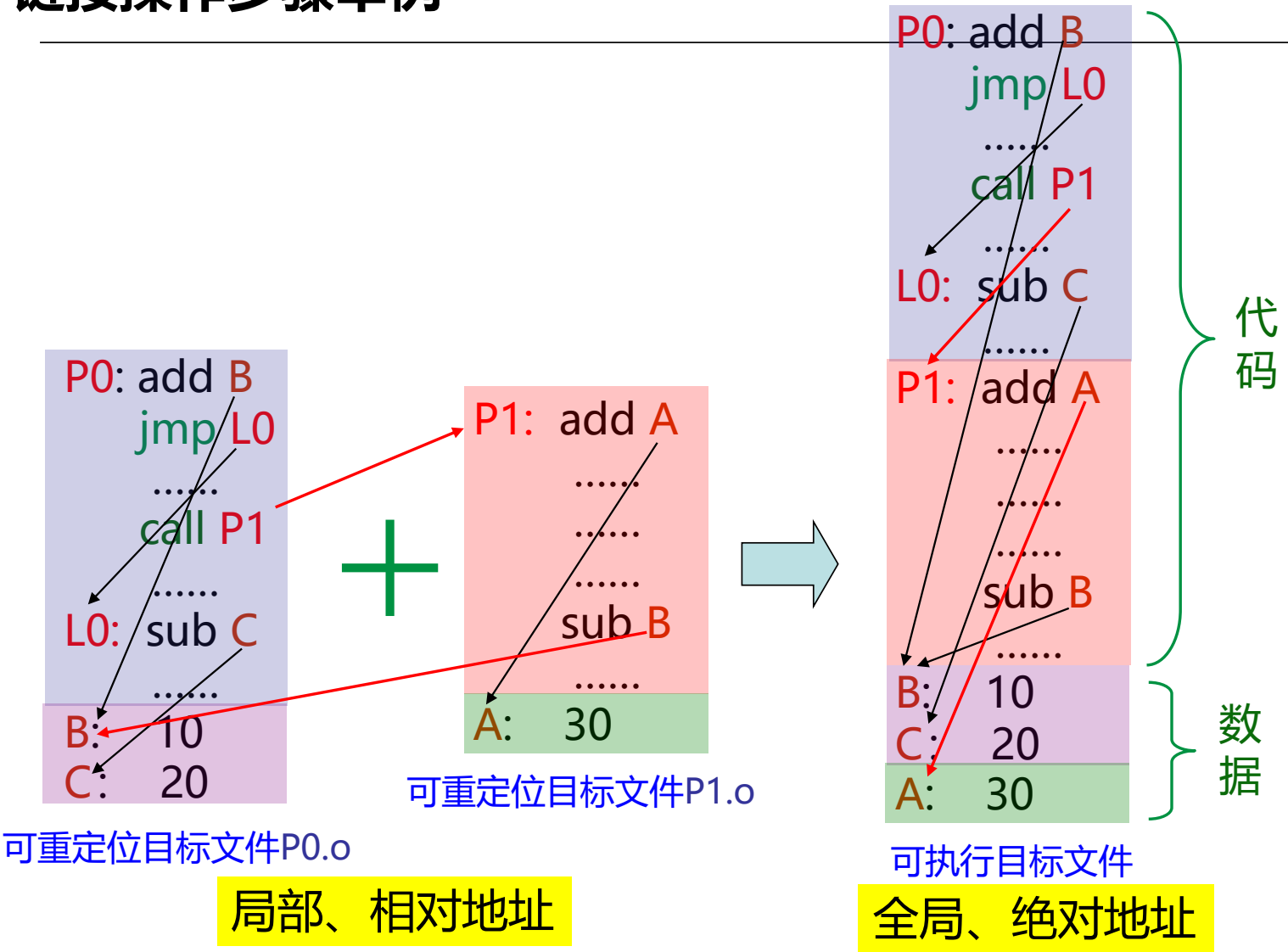
  目的：将每个模块中引用的符号(包括变量和函数) 和某个模块中定义符号建立关联

  - **符号定义和符号的引用举例**
    - void swap() {…} /* 定义符号swap */
    - swap(); /* 引用符号swap */
    - int *xp = &x; /* 定义符号 xp, 引用符号 x */
  - **编译器将定义的符号存放在一个符号表（symbol table）中.**
    - 符号表是一个结构数组
    - 每个表项包含符号名、长度和位置等信息
  - **链接器将每个符号的引用都与其确定的符号定义建立关联**

- **Step 2. 重定位**

  目的：将所有关联的模块合并，并确定运行时每个定义符号在虚拟地址空间中的地址，然后在定义符号的引用处重定位引用符号的地址

链接操作步骤举例


P0: add B
jmp L0
.....
call P1
.....
L0: sub C
.....
B: 10
C: 20
可重定位目标文件P0.o
P1: add A
.....
.....
.....
sub B
.....
A: 30
可重定位目标文件P1.o
局部、相对地址
P0: add B
jmp L0
.....
call P1
.....
L0: sub C
.....
P1: add A
.....
.....
.....
sub B
.....
B: 10
C: 20
A: 30
可执行目标文件
全局、绝对地址
代码
数据

程序的链接

分以下三个部分介绍

- 第一讲：目标文件格式
  - 什么是程序链接、链接的意义与过程
  - 目标文件格式、可重定位目标文件格式和可执行目标文件格式
- 第二讲：符号解析与重定位
  - 什么是符号表、怎么进行符号解析
  - 与静态库的链接
  - 重定位所需信息和重定位过程
  - 可执行文件的加载
- 第三讲：动态链接
  - 动态链接的特性
  - 程序加载时的动态链接、程序运行时的动态链接
  - 位置无关代码

一、目标文件的格式

- **目标文件（Object File）**
  - 源代码经编译或汇编后所生成的机器语言代码称为**目标代码**。
  - 存放目标代码的文件称为**目标文件**。
- **目标文件主要有两类：**
  - **可重定位目标文件**
    - 其代码和数据可以和其他可重定位文件的代码和数据合并为可执行目标文件，其中每个可重定位目标文件的代码和数据的地址都从0开始。
  - **可执行目标文件**
    - 包含的代码和数据可以被直接复制到内存并被执行，其中代码和数据的地址为统一地址空间中的虚拟地址。

目标文件格式：目标文件的结构规范

- 早期，不同的计算机各有自己的独特格式，非标准的，不兼容
- 现在标准的目标文件格式有： COM、 COFF、PE、 ELF等格式
  - COM格式：DOS操作系统使用，简单，文件中仅包含代码和数据，且被加载到固定位置.
  - COFF格式：System V UNIX早期版本使用，文件中不仅包含代码和数据，还包含重定位信息、调试信息、符号表等其他信息，由一组严格定义的数据结构序列组成.
  - PE格式：Windows使用，COFF的变种，称为可移植可执行格式（Portable Executable, 简称PE）
  - **ELF格式**：现代UNIX操作系统（例如Linux）使用，COFF的另一变种，称为可执行可链接格式（Executable and Linkable Format，简称ELF）

ELF目标文件格式

- **ELF目标文件格式有两种视图**
  - **链接视图**：对应“可重定位目标文件”
  - **执行视图**：对应“可执行目标文件 ”

1) 链接视图

➢可重定位目标文件由不同的**节**组成。

➢**节**：section，是 ELF 文件中具有相同特征的最小可处理单位，不同的节描述了目标文件中不同类型的信息及其特征。

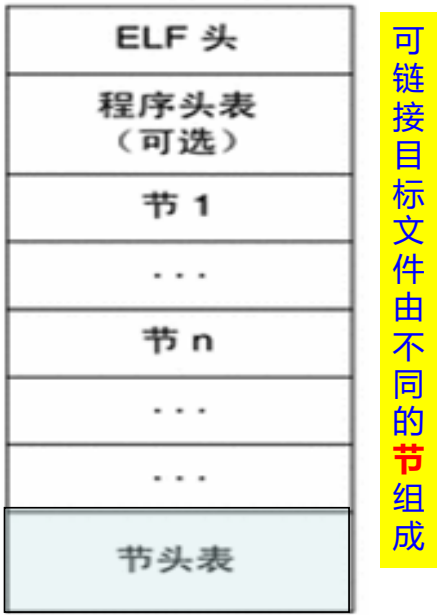
如：

.text: 代码节

.rodata: 只读数据节

.data: 已初始化的数据节

.bss: 未初始化的数据节



链接视图

2) 执行视图

- 可执行目标文件的最基本结构单元还是”节”，但引入逻辑单元“段”
- 段： segment，是具有某种共同的性质的节的集合

  如：可读写数据段是映射到一块连续区域的.data节和.bss节的集合

  如：只读代码段是映射到一块连续区域的.text和.rodata等节的集合

执行视图

程序的链接

分以下三个部分介绍

- 第一讲：目标文件格式
  - 什么是程序链接、链接的意义与过程
  - ELF目标文件格式、可重定位目标文件格式和可执行目标文件格式
- 第二讲：符号解析与重定位
  - 什么是符号表、怎么进行符号解析
  - 与静态库的链接
  - 重定位所需信息和重定位过程
  - 可执行文件的加载
- 第三讲：动态链接
  - 动态链接的特性
  - 程序加载时的动态链接、程序运行时的动态链接
  - 位置无关代码

可重定位目标文件的ELF格式

– 可重定位目标文件由不同的节组成：

①.text：代码

②.data：已初始化数据

③.bss：未初始化数据

④.symtab：符号表

⑤.rel.txt：代码重定位信息

⑥.rel.data：数据重定位信息

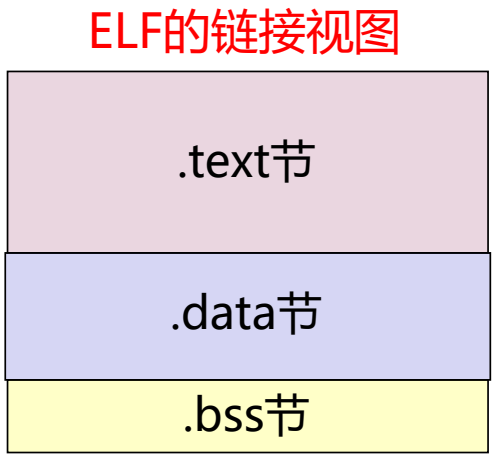
⑦等等

注：Linux中可重定位目标文件的扩展名为.o

Windows中可重定位目标文件的扩展名为.obj

可重定位目标文件ELF格式

```
int x=100;
int y;
void prn(int n)
{
   printf(  "%d\n" ,n);
}

void main( )
{
    static int a=1;
    static int b;
    int i=200;
    int j;
    prn(x+a+i);
}
```



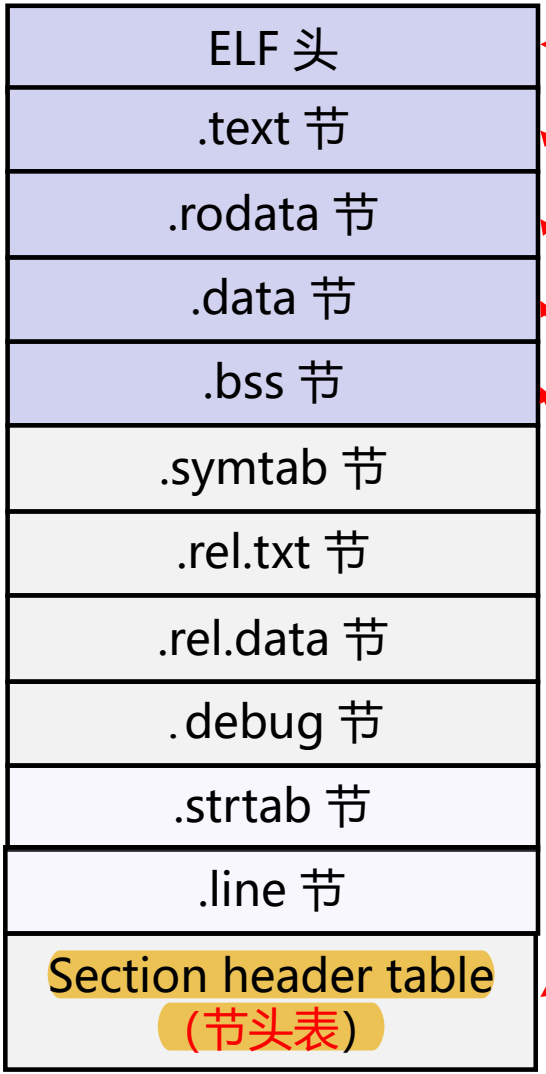
指令映射到代码节.text

已初始化的全局变量或静态局部变量映射到已初始化数据节.data

未初始化的全局变量或静态局部变量映射到未初始化数据节.bss

特别注意：非静态局部变量（例如 i 和 j ），只在运行时，在栈里分配存储空间, 不在.data节或.bss节分配存储空间

可重定位目标文件ELF格式



ELF 头

- ✓ 位于目标文件的起始位置，包含文件结构说明信息

.text 节

- ✓ 编译后的代码部分

.rodata 节

- ✓ 只读数据，如printf 格式串、switch跳转表等

.data 节

- ✓ 已初始化的全局变量和静态局部变量

.bss 节

- ✓ 未初始化的全局变量和静态局部变量，仅是占位符，不占据任何实际磁盘空间。
- ✓ 区分初始化和非初始化是为了空间效率

节头表

- ✓ 由若干个表项组成，每个节都有一个表项与之对应，每个表项描述对应的一个节的信息

ELF头（ELF Header）

描述了本目标文件的一些基本信息，例如

① ELF头的版本和大小

② 数据存放是大端还是小端

③ 目标文件的类型（例如：可重定位目标文件、可执行目标文件）

④ 机器结构类型（例如： IA-32、AMD64 ）

⑤ 控制权转移的起始虚拟地址（只对可执行目标文件有效）

⑥ 程序头表基本信息（只对可执行目标文件有效）：位置、表项数量、表项大小

⑦ 节头表基本信息：位置、表项数量、表项大小等

1）ELF头（ELF Header）

a.out的魔数：01H 07H，PE格式魔数：4DH 5AH

```
#define EI_NIDENT     16
typedef struct {
    unsigned char  e_ident[EI_NIDENT];
    Elf32_Half     e_type;
    Elf32_Half     e_machine;
    Elf32_Word     e_version;
    Elf32_Addr     e_entry;
    Elf32_Off      e_phoff;
    Elf32_Off      e_shoff;
    Elf32_Word     e_flags;
    Elf32_Half     e_ehsize;
    Elf32_Half     e_phentsize;
    Elf32_Half     e_phnum;
    Elf32_Half     e_shentsize;
    Elf32_Half     e_shnum;
    Elf32_Half     e_shstrndx;
} Elf32_Ehdr;
```

魔数：在文件开头，用来确定文件类型、格式的几个字节数据

16字节序列，ELF魔数（7f 45 4c 46） 等

目标文件的类型：可重定位文件/可执行文件/共享库文件/其他类型文件等

指定机器结构类型：IA-32、SPARC V9、AMD64

标识目标文件版本

指定系统将控制权转移的起始虚拟地址（入口点），若没有关联的入口点，则置0，例如：可重定位文件，此字段为0

程序头表在文件中的偏移量（以字节为单位）

节头表在文件中的偏移量（以字节为单位）

处理器特定标志，对IA32而言，此项为0。

说明ELF文件头的大小（以字节为单位）

程序头表中一个表项的大小（以字节为单位），所有表项大小相同

程序头表中表项的数量。 程序头表大小= e_phentsize* e_phnum

节头表中一个表项的大小（以字节为单位），所有表项大小相同

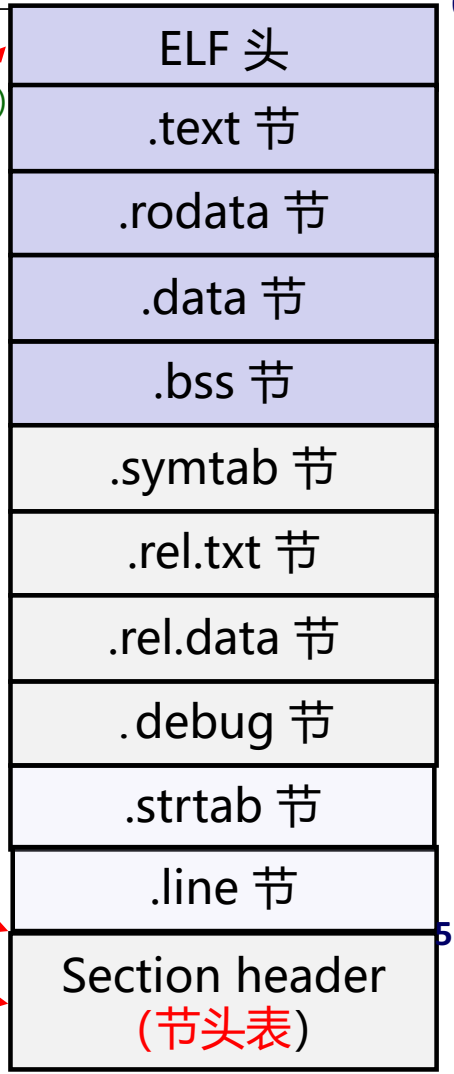
节头表中表项的数量。 节头表大小= e_shentsize* e_shnum

.strtab在节头表中的索引

可重定位目标文件ELF头信息举例

```
$ readelf -h main.o
ELF Header:  ELF文件的魔数（分别对应ascii码表中的delete、E、L、F字符）
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:     ELF32
  Data:      2's complement, little endian
  Version:   1 (current)    ELF头的版本号
  OS/ABI:    UNIX - System V
  ABI Version:   0          可重定位目标文件
  Type:      REL (Relocatable file)
  Machine:   Intel 80386
  Version:   0x1  目标文件版本     因为是可重定位目标文
  Entry point address:  0x0  件，所以程序入口为0
  Start of program headers:  0 (bytes into file)
  Start of section headers:  516 (bytes into file)
  Flags:     0x0
  Size of this header:   52 (bytes)
  Size of program headers:   0 (bytes)
  Number of program headers:   0
  Size of section headers:   40 (bytes)
  Number of section headers:  15
  Section header string table index: 12
```

15个表项，每个表项大小为40B



节头表（Section Header Table）

- 描述每个节的基本信息，例如：节名、在目标文件中的偏移、大小、访问属性、对齐方式等
- **下是32位系统对应的节头表数据结构（每个表项占40B）**

```
typedef struct {
    Elf32_Word      sh_name;
    Elf32_Word      sh_type;
    Elf32_Word      sh_flags;
    Elf32_Addr      sh_addr;
    Elf32_Off       sh_offset;
    Elf32_Word      sh_size;
    Elf32_Word      sh_link;
    Elf32_Word      sh_info;
    Elf32_Word      sh_addralign;
    Elf32_Word      sh_entsize;
} Elf32_Shdr;
```

节名字符串在.strtab中的偏移

节类型：无效/代码或数据/符号/字符串/...

节标志：该节在存储空间中的访问属性

虚拟地址：若可被加载，则对应虚拟地址

在文件中的偏移地址，对.bss节而言则无意义

节在文件中所占的长度

sh_link和sh_info用于与链接相关的节（如.rel.text节、.rel.data节、.symtab节等）

节的对齐要求

节中每个表项的长度，0表示无固定长度表项

可重定位目标文件节头表信息举例

$ readelf -S test.o

There are 11 section headers, starting at offset 0x120:

Section Headers:

文件内偏移 节大小

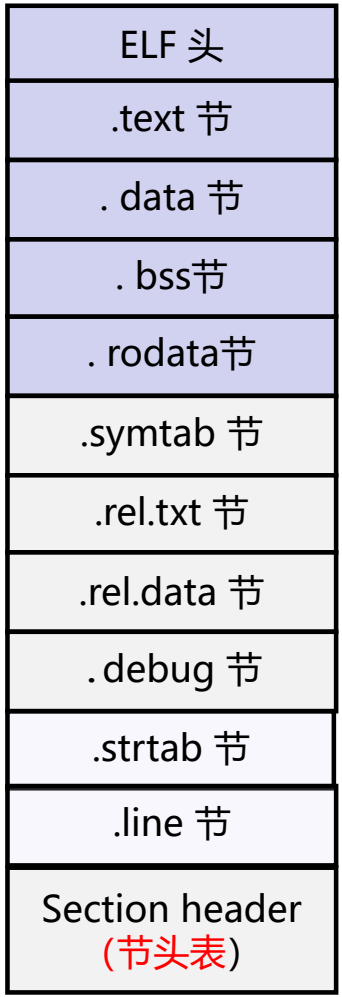
[Nr] Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]	NULL	00000000	000000	000000	00		0	0	0
[1] .text	PROGBITS	00000000	000034	00005b	00	AX	0	0	4
[2] .rel.text	REL	00000000	000498	000028	08		9	1	4
[3] .data	PROGBITS	00000000	000090	00000c	00	WA	0	0	4
[4] .bss	NOBITS	00000000	00009c	00000c	00	WA	0	0	4
[5] .rodata	PROGBITS	00000000	00009c	000004	00	A	0	0	1
[6] .comment	PROGBITS	00000000	0000a0	00002e	00		0	0	1
[7] .note.GNU-stack	PROGBITS	00000000	0000ce	000000	00		0	0	1
[8] .shstrtab	STRTAB	00000000	0000ce	000051	00		0	0	1
[9] .symtab	SYMTAB	00000000	0002d8	000120	10		10	13	4
[10] .strtab	STRTAB	00000000	0003f8	00009e	00		0	0	1

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings)

I (info), L (link order), G (group), x (unknown)

O (extra OS processing required) o (OS specific), p (processor specific)



可重定位目标文件节头表信息举例

$ readelf -S test.o

There are 11 section headers, starting at offset 0x120:

Section Headers:

[Nr] Name	Off	Size	ES	Flg	Lk	Inf	Al
[0]	000000	000000	00		0	0	0
[1] .text	000034	00005b	00	AX	0	0	4
[2] .rel.text	000498	000028	08		9	1	4
[3] .data	000090	00000c	00	WA	0	0	4
[4] .bss	00009c	00000c	00	WA	0	0	4
[5] .rodata	00009c	000004	00	A	0	0	1
[6] .comment	0000a0	00002e	00		0	0	1
[7] .note.GNU-stack	0000ce	000000	00		0	0	1
[8] .shstrtab	0000ce	000051	00		0	0	1
[9] .symtab	0002d8	000120	10		10	13	4
[10] .strtab	0003f8	00009e	00		0	0	1

Key to Flags:

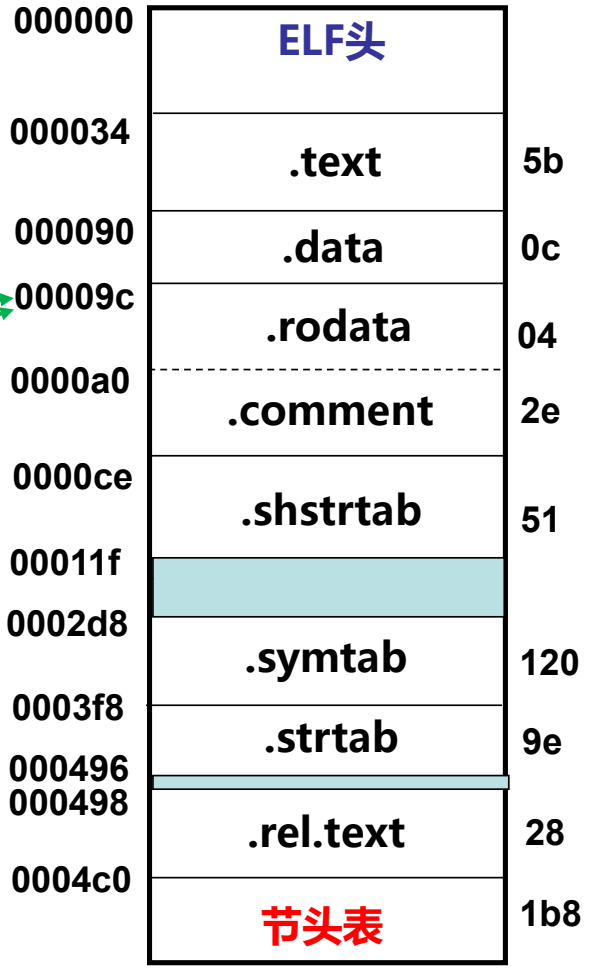
W (write), A (alloc), X (execute), M (merge), S (strings)

I (info), L (link order), G (group), x (unknown)

说明: .text节从文件第0x34(52) 字节开始,占用0x5b(91) 字节

这两地址相同, 说明未初始化数据节.bss在文件中没有占用字节,但是.bss大小显示是0x0c(12)字节,因此在执行时需在主存中给.bss分配12字节

可重定位目标文件test.o的结构



程序的链接

分以下三个部分介绍

- 第一讲：目标文件格式
  - 什么是程序链接、链接的意义与过程
  - ELF目标文件格式、可重定位目标文件格式和可执行目标文件格式
- 第二讲：符号解析与重定位
  - 什么是符号表、怎么进行符号解析
  - 与静态库的链接
  - 重定位所需信息和重定位过程
  - 可执行文件的加载
- 第三讲：动态链接
  - 动态链接的特性
  - 程序加载时的动态链接、程序运行时的动态链接
  - 位置无关代码

可执行目标文件格式ELF格式

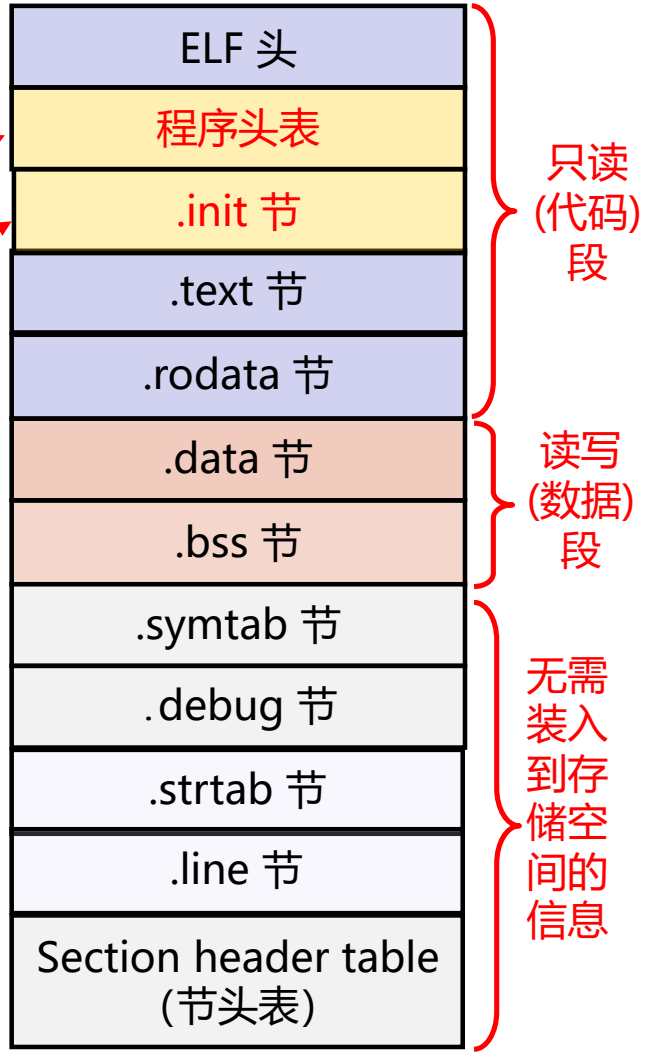
- 包含代码、数据等
- 其中定义的所有变量和函数在统一虚拟地址空间中已有确定地址
- 符号引用处已被重定位，指向相应的已定义符号在统一虚拟地址空间中的地址
- 可被CPU直接执行，指令地址和指令给出的操作数地址都是统一虚拟地址空间中的虚拟地址

为了能执行，在执行视图中，需要将具有相同访问属性的节合并成**段**（Segment），并说明每个段的属性，例如：在可执行目标文件中的位移、大小、在统一虚拟地址空间中的位置、对齐方式、访问属性等。**程序头表就是用来说明这些段信息，也称段头表（segment header table）**

可执行目标文件格式ELF格式

与可重定位目标文件相比，它有如下不同：

- ELF头中的字段e_entry在可执行目标文件中会给出执行程序时第一条指令的地址，而在可重定位文件中，此字段为0；
- 多一个程序头表（段头表），用于说明可执行目标文件的段的组成。
- 多一个.init节，其中定义了一个_init函数，用于可执行目标文件开始执行时的初始化工作；
- 少了两个用于重定位的信息节(即.rel.text节和.rel.data节)；

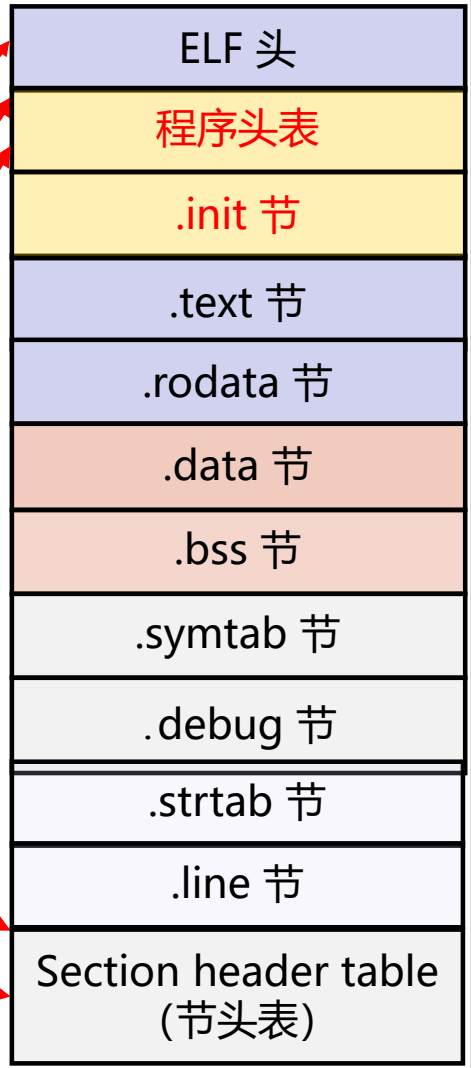


可执行目标文件的ELF头信息举例

```
$ readelf -h main
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:    ELF32
  Data:      2's complement, little endian
  Version:  1 (current)
  OS/ABI:    UNIX - System V
  ABI Version:     0
  Type:    EXEC (Executable file)
  Machine:   Intel 80386
  Version:    0x1
  Entry point address:    0x8048580
  Start of program headers:  52 (bytes into file)
  Start of section headers:    3232 (bytes into file)
  Flags:    0x0
  Size of this header:   52 (bytes)
  Size of program headers:    32 (bytes)
  Number of program headers:   8
  Size of section headers:     40 (bytes)
  Number of section headers:    29
  Section header string table index: 26
```

8个表项，每个表项大小为32B

29个表项，每个表项大小40B



可执行文件程序头表

- 程序头表：描述可执行目标文件中的段与虚拟地址空间的映射关系，每个表项记录一个段的映射等相关信息

程序头表的数据结构

```
typedef struct {
    Elf32_Word    p_type;
    Elf32_Off     p_offset;
    Elf32_Addr    p_vaddr;
    Elf32_Addr    p_paddr;
    Elf32_Word    p_filesz;
    Elf32_Word    p_memsz;
    Elf32_Word    p_flags;
    Elf32_Word    p_align;
} Elf32_Phdr;
```

- p_type：描述存储段的类型或特殊节类型，如：是否是可装入段(PT_LOAD)、是否是特殊的动态节(PT_DYNAMIC)、是否是特殊的解释程序节(PT_INTERP)
- p_offset：指出本段的首字节在文件中的偏移地址
- p_vaddr：指出本段的首字节的虚拟地址
- p_paddr：指出本段的首字节的物理地址，但因为物理地址只有在运行时由操作系统动态确定，所以该信息通常无效。
- p_filesz：指出本段在文件中所占的字节数，可以为0
- p_memsz：指出本段在**存储器**中所占的字节数，可以为0
- p_flags：指出存取权限
- p_align：指出对齐方式

可执行目标文件程序头表信息举例

$ readelf –l main

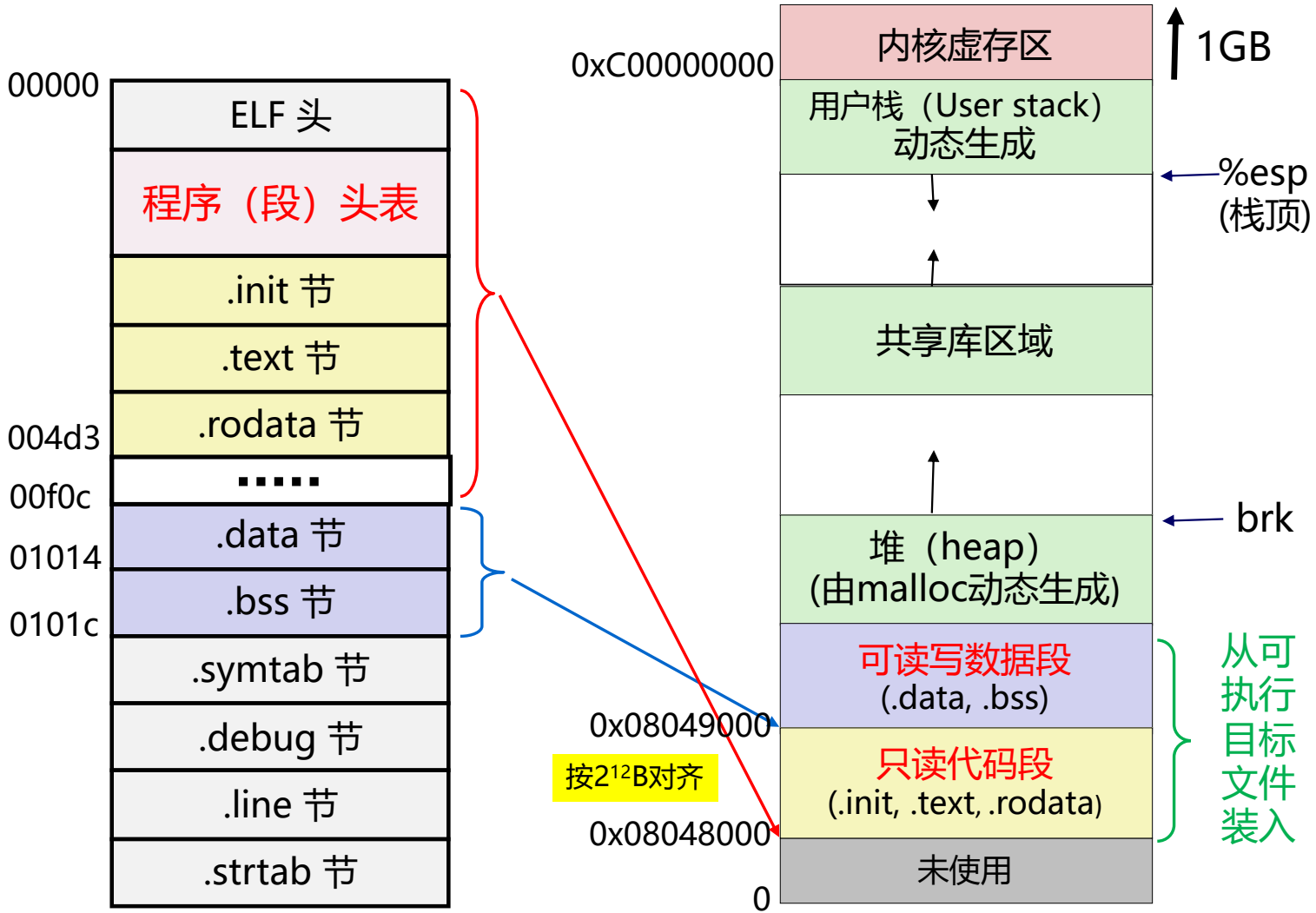
```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00100 0x00100 R E 0x4
  INTERP         0x000134 0x08048134 0x08048134 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x004d4 0x004d4 R E 0x1000
  LOAD           0x000f0c 0x08049f0c 0x08049f0c 0x00108 0x00110 RW  0x1000
  DYNAMIC        0x000f20 0x08049f20 0x08049f20 0x000d0 0x000d0 RW  0x4
  NOTE           0x000148 0x08048148 0x08048148 0x00044 0x00044 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
  GNU_RELRO      0x000f0c 0x08049f0c 0x08049f0c 0x000f4 0x000f4 R   0x1
```

有8个表项，其中两个为可装入段（即Type=LOAD）

第一可装入段：具有只读/执行权限（因为Flg=RE），所以是只读代码段。它的内容在可执行目标文件中从0x000000 (Offset=0x000000）开始到0x004d3（FileSiz= 0x004d4字节）结束。映射到虚拟地址0x08048000 (因为VirtAddr=0x08048000）开始，长度为0x004d4字节(因为MemSiz= 0x004d4字节)的区域，按0x1000 = 2^{12} =4KB对齐

第二可装入段：具有可读写权限（Flg=RW），所以是可读写数据段。它的内容是可执行目标文件中从0x000f0c开始的大小为0x00108字节的内容，映射到虚拟地址从0x08049f0c开始长度为0x00110字节的存储区域，按0x1000=4KB对齐。该存储区域(大小为0x00110=272字节)的前0x00108=264B会用.data节内容初始化，后面272-264=8B对应.bss节，初始化为0

上一页例子具体映射解释图



程序的链接

分以下三个部分介绍

- 第一讲：目标文件格式
  - 什么是程序链接、链接的意义与过程
  - ELF目标文件格式、可重定位目标文件格式和可执行目标文件格式
- 第二讲：符号解析与重定位
  - 什么是符号表、怎么进行符号解析
  - 与静态库的链接
  - 重定位所需信息和重定位过程
  - 可执行文件的加载
- 第三讲：动态链接
  - 动态链接的特性
  - 程序加载时的动态链接、程序运行时的动态链接
  - 位置无关代码

- 链接的第一步：符号解析
- 这个过程需要知道**被定义的所有符号（即函数名和变量名）的相关信息**。这些**相关信息**就存储在**可重定位目标文件**的符号表（即.symtab节）中

包含在符号表中的符号有三种：

1） **Global symbol（全局符号）**：

- 指本模块内部定义并被其他模块引用的符号

  包括：非static的函数名和非static的全局变量名

2）**Local symbol（本地符号）**：

- 指本模块内部定义并仅能由本模块内部引用的符号

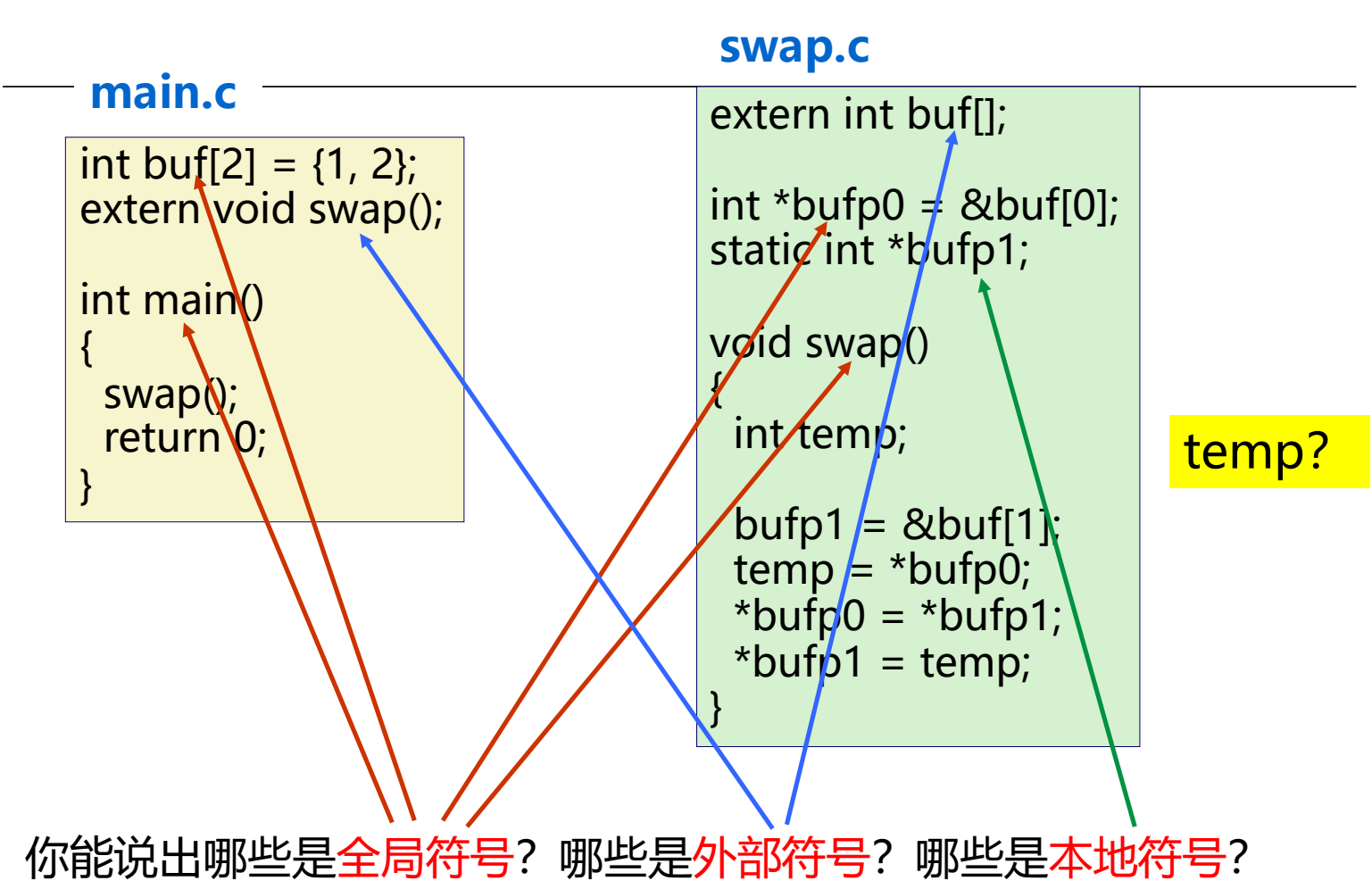
  包括：在本模块内部定义的带static属性的函数名和带static属性的全局变量名

3）**External symbol（外部符号）**：

- 由其他模块定义并被本模块引用的符号

  包括：在其他模块定义的外部函数名和外部变量名

  例如：声明中出现的带extern的函数名和带extern的全部变量名



注意：局部变量（如temp）不包含在符合表中

局部变量是在栈中分配空间进行存储的临时性变量，链接器不关心这种局部变量

可重定位目标文件中的符号表

➢ **.symtab** 节记录符号表信息，它是一个结构数组

符号表每个条目的结构如下：

```
typedef  struct {
        Elf32_Word    st_name;
        Elf32_Addr    st_value;
        Elf32_Word    st_size;
        unsigned char st_info;
        unsigned char  st_other;
        Elf32_Half  st_shndx;
} Elf_Symbol;
```

给出符号在字符串表(.strtab节)中的索引（字节偏移量），指向在字符串表中的一个以null结尾的字符串。

给出符号的位置，**在可重定位目标文件中**，是指符号**所在位置**相对于**所在节**(函数名在.text节中,变量名在.data节或.bss节中)**起始位置**的**字节偏移量**。在可执行目标文件和共享目标文件中是符号所在的虚拟地址

给出符号所表示对象的字节个数，如果符号是函数名，则指**函数所占字节个数**；若符号是变量名，则指**变量所占字节个数**；如果符号表示的内容没有大小或大小未知，则值为0

低四位表示符号类型：可以是**变量(OBJECT)**、**函数(FUNC)**、**未指定(NOTYPE)**、节(SECTION，用于重定位)

高四位表示绑定属性：可以是**本地(LOCAL)**、**全局(GLOBAL)**、弱(WEAK，弱符号)等

指出符号所在节在节头表中的索引。但有三种伪节，在节头表中没有相应的表项，故无法表示索引值，因而用以下特殊的索引值表示：ABS表示该符号不会由于重定位而发生值的改变，即不应该被重定位。 **UNDEF表示未定义符号，即在本模块引用而在其他模块定义的外部符号**；**COMMON表示还未被分配位置的未初始化的变量**(即.bss中的变量)，**对于COMMON类型的符号，st_value字段给出的是对齐要求，st_size给出的是最小长度**。

例：可以命令行**readelf –s main.o** 查看main.o中的符号表

1）main.o中的符号表中最后三个条目（共10个）

Num:	value	Size	Type	Bind	Ot	Ndx	Name
8:	0	8	OBJECT	Global	0	3	buf
9:	0	33	FUNC	Global	0	1	main
10:	0	0	NOTYPE	Global	0	UND	swap

- **buf**是main.o中第3节（.data）中偏移为0的符号，是全局变量，占8B；
- main是main.o中第1节（.text）中偏移为0的符号，是全局函数，占33B；
- swap是main.o中未定义全局符号（在其他模块定义，即外部符号），类型和大小未知。

2）swap.o中的符号表中最后4个条目（共11个） **readelf –s swap.o**

Num:	value	Size	Type	Bind	Ot	Ndx	Name
8:	0	4	OBJECT	Global	0	3	bufp0
9:	0	0	NOTYPE	Global	0	UND	buf
10:	0	39	FUNC	Global	0	1	swap
11:	4	4	OBJECT	Local	0	COM	bufp1

buf: 未定义全局符号（在其他模块定义）

bufp1是未分配位置且未初始化(ndx=COM，即.bss节中变量)的本地变量(Bind=Local，Type=OBJECT)，按4B对齐（因为ndx=COM，value =4）对齐，至少占4B（因为ndx=COM，Size =4）

程序的链接

分以下三个部分介绍

- 第一讲：目标文件格式
  - 什么是程序链接、链接的意义与过程
  - ELF目标文件格式、可重定位目标文件格式和可执行目标文件格式
- 第二讲：符号解析与重定位
  - 什么是符号表、怎么进行符号解析
  - 与静态库的链接
  - 重定位所需信息和重定位过程
  - 可执行文件的加载
- 第三讲：动态链接
  - 动态链接的特性
  - 程序加载时的动态链接、程序运行时的动态链接
  - 位置无关代码

符号解析（Symbol Resolution），也称符号绑定

- 符号区分为定义符号和引用符号
  - **定义符号**：指在定义处的符号。因此，每个定义符号在代码段或数据段中都被分配了存储空间，有相应的地址。对函数名而言指它在代码区的首地址；对变量名而言指它在静态数据区的首地址。
  - **引用符号**：指引用处的符号。
- **符号解析**的目的就是将每个模块中的引用符号与在某个目标模块中的定义符号建立关联。

  有了这个关联，就可在重定位时将**引用符号**的地址重定位为相关联的**定义符号**的地址，从而知道引用符号对应的定义符号的实际地址，也就能够访问引用符号对应的定义符号的实际内容（即函数体或变量值）

符号解析分两种情况：本地符号的符号解析和全局符号的符号解析

- 本地符号在本模块内定义并引用。因为引用符合只要与本模块内唯一的定义符号（如果在一个模块内有多次定义，编译器在编译此模块时就能发现此错误）关联即可，所以解析简单 。
- 全局符号的解析涉及多个模块（这些模块可能被不同人在不同时候编写和编译，使得不同模块中可能多次定义同一个符号），因此较复杂 。
- **我们主要介绍如何进行全局符号的符号解析。**

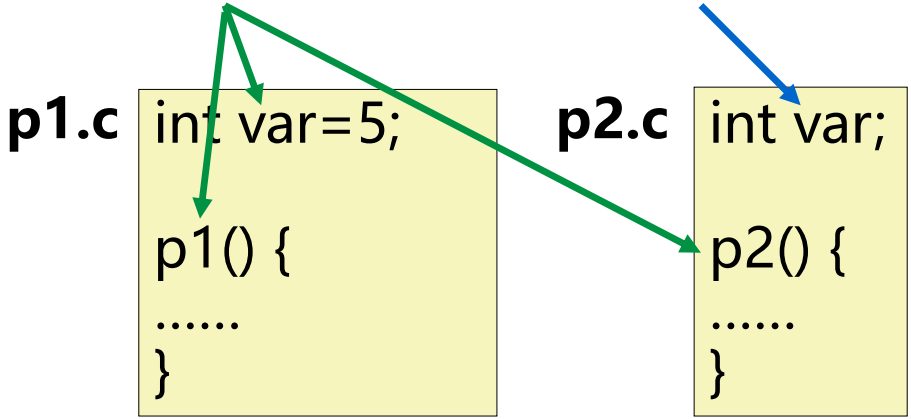
全局符号的符号解析

1. 全局符号的强/弱特性

由于不同模块可能会多次定义相同名字的全局符合，为**尽量**保证程序正确性，编译器在符号解析时会把**全局符号**分成**强符号**和**弱符号**：

- 已定义的函数名和已初始化的全局变量名是强符号
- 未初始化的全局变量名是弱符号

例如，以下符号哪些是强符号？哪些是弱符号？



以下定义符号哪些是强符号定义？哪些是弱符号定义？

main.c

```
int buf[2] = {1, 2};
extern void swap();

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

swap.c

```
int buf[];

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

void swap()
{
  int temp;

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

局部变量

注：局部变量不包含在符号表里

外部符合和本地符号没有强弱之分，因为没必要

2.链接器对符号的解析规则

- 只定义一次的符号，只要建立引用符号和定义符号的关联即可
- 多重定义的符号如何处理?

如:

```
int  x=10;
int  p1(void);
int main()
{
     x=p1();
     return x;
}
```

main.c

```
int  x=20;
int p1()
{
     return x;
}
```

p1.c

- 在符号解析时，任何符号不管有多少处定义，最终只能有一个确定的定义（即每个符号仅能对应一处唯一的存储空间）

链接器对多重定义符号的处理规则

Rule 1: 强符号不能多次定义

也就是说，强符号只能被定义一次，否则链接错误

Rule 2: 若一个符号被说明为一次强符号定义和多次弱符号定义，则按强符号定义为准

此时，弱符号被解析为对其强定义符号的引用

Rule 3: 若有多个弱符号定义，则任选其中一个

多重定义符号的解析举例1

以下程序会发生链接出错吗?

```
int  x=10;
int  p1(void);
int main()
{
    x=p1();
    return x;
}
```

main.c

```
int  x=20;
int p1()
{
    return x;
}
```

p1.c

x有两次强定义，所以，链接器将输出一条出错信息。

```
-------------- Build: Debug in test (compiler: GNU GCC Compiler)---------------
mingw32-gcc.exe -Wall -g  -c e:\test\f3.c -o obj\Debug\f3.o
mingw32-g++.exe  -o bin\Debug\test.exe obj\Debug\f3.o obj\Debug\test.o
obj\Debug\test.o:test.c:(.data+0x0): multiple definition of `x'
obj\Debug\f3.o:f3.c:(.data+0x0): first defined here
collect2.exe: error: ld returned 1 exit status
Process terminated with status 1 (0 minute(s), 2 second(s))
1 error(s), 0 warning(s) (0 minute(s), 2 second(s))
```

多重定义符号的解析举例2

以下程序会发生链接出错吗？ 没错！

```
# include <stdio.h>
int  y=100;
int  z;
void  p1(void);
int main()
{
    z=1000;
    p1( );
    printf( “y=%d, z=%d\n” , y, z);
    return 0;
}
```

main.c

强定义的符号y

弱定义的符号y

以main.o强定义的符号y为准

z的两次定义都是弱定义

任选其一（如果按先后顺序，那么链接器将main.o中的z作为唯一定义符号）

```
int  y;
int  z;
void p1( )
{
    y=200;
    z=2000;
}
```

p1.c

打印结果是什么？

y=200，z=2000

多重定义符号的解析举例3

以下程序会发生链接出错吗?

```
1  #include <stdio.h>
2  int d=100;
3  int x=200;
4  voi
5  int
6  {
7     p
8     printf("  d=%d,x=%d\n  ",d,x);
9     return 0;
10 }
```

main.c

p1.c

```
1  double d;
2
3  void p1()
4  {
```

```
-------------- Build: Debug in test (compiler: GNU GCC Compiler)---------------

mingw32-gcc.exe -Wall -g  -c e:\test\test.c -o obj\Debug\test.o
mingw32-g++.exe  -o bin\Debug\test.exe obj\Debug\f3.o obj\Debug\test.o
Output file is bin\Debug\test.exe with size 29.40 KB
Process terminated with status 0 (0 minute(s), 2 second(s))
0 error(s), 0 warning(s) (0 minute(s), 2 second(s))
```

多重定义符号的解析举例3

以下程序会发生链接出错吗?

```
1  #include <stdio.h>
2  int d=100;
3  int x=200;
4  void p1(void);
5  int main()
6  {
7     p1();
8     printf( "d=%d,x=%d\n" ,d,x);
9     return 0;
10 }
```

main.c

强定义的符号d

弱定义的符号d

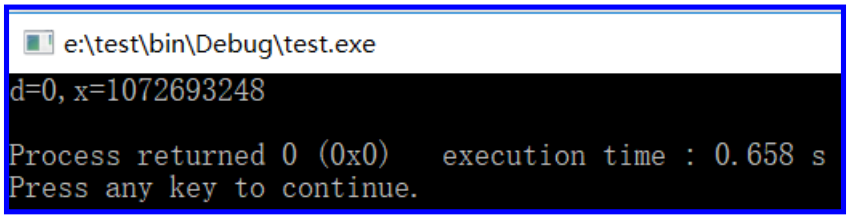
虽然类型不一，但不会报错，仍以main.o强定义的符号d为准

p1.c

```
1  double d;
2
3  void p1()
4  {
5     d=1.0;
6  }
```

就是说，main.o中的d作为d的唯一定义符号，类型为int，并有唯一的存储空间：4字节。但在p1中，d的类型还是double的，赋值按double型处理：8字节。

那么打印结果是什么呢?



多重定义符号的解析举例

main.c

长度后缀：l(双字)、 q(四字)

```
.......
1  int d=100;
2  int x=200;
3  int main()
4  {
5    p1( );
6    printf ( “d=%d, x=%d\n” , d, x );
7    return 0;
8  }
```

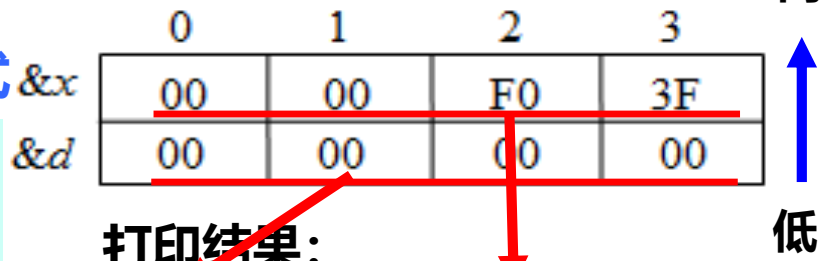
p1.c

```
1  double d;
2
3  void p1( )
4  {
5    d=1.0;
6  }
```

double型数1.0对应的机器数

3FF0 0000 0000 0000H

IA-32是小端方式



该例说明：如果在两个不同模块定义相同变量名，很可能发生意想不到的结果！特别是有弱符号定义时，并且两个重复定义的变量具有不同类型时!

打印结果：

d=0，x=1 072 693 248

Why?

多重定义全局符号的问题

- 多重定义全局变量会造成一些意想不到的错误，而且是**默默发生**的，编译系统不会警告，并会在程序执行很久后才能表现出来，且远离错误引发处。特别是在一个具有成千上万个模块的大型软件中，这类错误很难修正。
- 了解链接器工作原理，**养成良好的编程习惯是非常重要的。**

- **因此，尽量避免使用全局变量**

- **一定需要用的话，就按以下规则使用：**
  - 尽量使用本地变量（static，本地化），这样就没有强弱之分
  - 全局变量要赋初值（强定义化）

程序的链接

分以下三个部分介绍

- 第一讲：目标文件格式
  - 什么是程序链接、链接的意义与过程
  - ELF目标文件格式、可重定位目标文件格式和可执行目标文件格式
- 第二讲：符号解析与重定位
  - 什么是符号表、怎么进行符号解析
  - 与静态库的链接（以与静态库的链接为例解释符合解析过程）
  - 重定位所需信息和重定位过程
  - 可执行文件的加载
- 第三讲：动态链接
  - 动态链接的特性
  - 程序加载时的动态链接、程序运行时的动态链接
  - 位置无关代码

静态共享库

- 在程序开发中，一些相关的常用的**目标模块**（.o）可以打包为一个称之为**静态库文件**的文件（.a文件）也称**存档文件**（archive），供程序员共享使用。

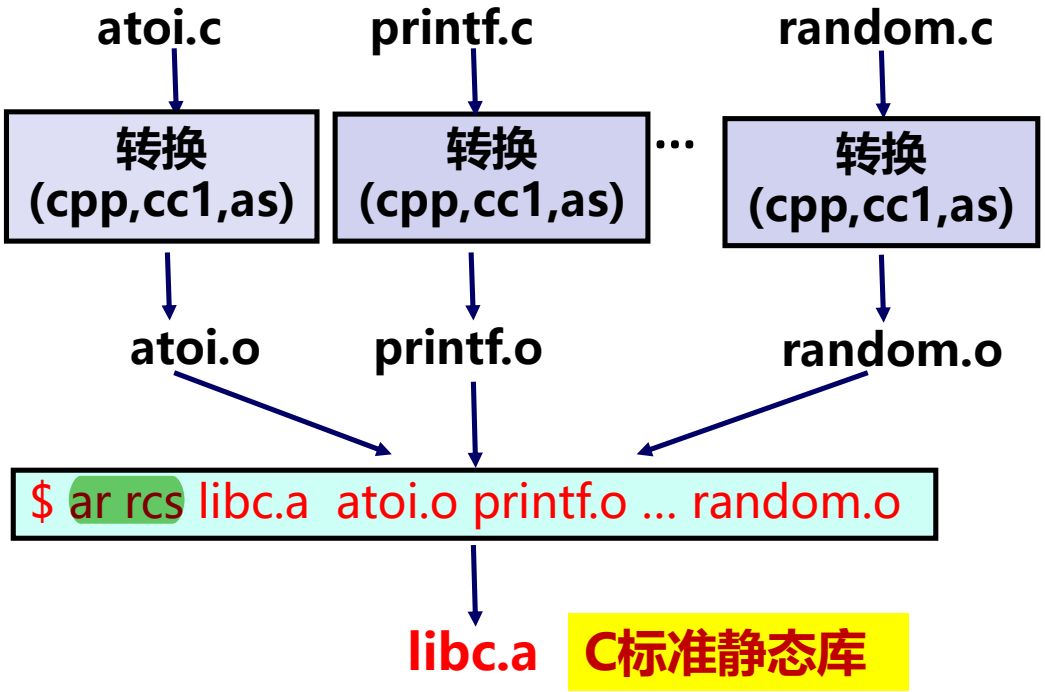
  例如：

  - C语言的标准函数库文件**libc.a**，包含了广泛使用的标准I/O函数、字符处理函数和整数处理函数。
  - C语言的数学函数库文件libm.a，包含sin、cos等数学函数。

- 在构建可执行文件时，如果用到静态库文件中的模块时，只需在链接的时候指定**库文件名**，链接器会自动到库中寻找用到的这些目标模块，并且只把用到的模块从库中拷贝出来，再和其他模块一块链接。

如何创建自己的静态库

- 首先，用”gcc -c”命令将目标模块源文件编译成可重定位目标文件 (.o文件)
- 然后，用ar命令打包.o文件生成.a文件



注意：Archiver（归档器）允许增量更新，只要重新编译修改过的源码，并将其.o文件替换到静态库中既可

静态库的创建和使用示例

例：将myproc1.o和myproc2.o 打包生成mylib.a

myproc1.c

```
# include <stdio.h>
void myfunc1() {
    printf("This is myfunc1!\n");
}
```

myproc2.c

```
# include <stdio.h>
void myfunc2() {
    printf("This is myfunc2\n");
}
```

$ gcc –c myproc1.c myproc2.c

$ ar rcs mylib.a myproc1.o myproc2.o

main.c

```
void myfunc1(viod);
int main()
{
   myfunc1();
   return 0;
}
```

调用关系：main→myfunc1→printf

$ gcc –c main.c

$ gcc –static –o myproc main.o ./mylib.a

libc.a是默认用于静态链接的库文件，可以不用在链接命令中显式给出

问题：如何进行符号解析?

我们以这个例子来看下链接器中符号解析过程

```
$ gcc –c main.c
$ gcc –static –o myproc main.o ./mylib.a
```

libc.a（无需明显指出）

函数调用关系：main→myfunc1→printf

main.c

```
void myfunc1(viod);
int main()
{
   myfunc1();
   return 0;
}
```

维护三个集合：（按命令行指定顺序从左往右扫描需链接的目标文件）

- **E 指将被合并到一起组成可执行目标文件的所有目标文件集合**
- **U 指当前所有未被解析的引用符号的集合**
- **D 指当前所有定义符号的集合**

处理步骤：

1）开始E、U、D都为空

2）链接器首先扫描到main.o，把它加入E，同时把定义符号main加入D，并把main函数中未解析的引用符号myfun1加入U。

3）链接器接着扫描到mylib.a，将U中的所有符号（例如myfunc1）与mylib.a中所有目标模块（myproc1.o和myproc2.o）中定义的符号，依次匹配。若发现在某目标模块（例如myproc1.o）中定义了myfunc1，则将myproc1.o加入E，并将myfunc1从U转移到D。在myproc1函数定义中发现还有未解析符号printf，从而将printf加到U中

我们以这个例子来看下链接器中符号解析过程

4）链接器继续在mylib.a的未丢弃的各模块中不断匹配U中的符号，直到E和U集合不发生变化。此时U中只有一个未解析符号printf，因为模块myproc2.o没有printf的定义，所以没有用到，从而myproc2.o不用被加入E中，而被丢弃

5）再接着，链接器扫描默认的库文件libc.a。发现其中目标模块printf.o定义了printf，于是printf也从U移到D，并将printf.o加入E，同时把printf函数里面定义的所有符号加入D，所有未解析符号加入U（这些未解析的符号都可以在标准库内其他模块中找到，所以在链接器处理完libc.a时，U最终变成空的）

6）最后，链接器合并E中的目标模块并输出最后生成的可执行目标文件

最终解析结果：
E中有main.o、myproc1.o、printf.o及其调用的模块
D中有main、myproc1、printf及其引用符号
U为空

链接器符号解析算法有这样几个要点:

- **链接器符号解析算法:**
  - 按照命令行给出的顺序扫描.o 和.a 文件

    例如： **$ gcc –static –o myproc main.o ./mylib.a libc.a**
  - 扫描期间将当前未解析的引用记录到一个列表U中
  - 每遇到一个新的.o 或 .a 中的模块，都试图用它里面定义的符号来解析U中的符号
  - 如果扫描到最后，U中还有未被解析的符号，则发生链接错误

思考：符号解析是否成功与命令行给出的文件顺序有关吗?

答案：能否正确完成符号解析与命令行给出的文件顺序有关

main.c

```
void myfunc1(viod);
int main()
{
   myfunc1();
   return 0;
}
```

$ gcc –static –o myproc main.o ./mylib.a

所以被链接的模块顺序应按调用顺序指定；否则可能会出现链接错误

main→myfunc1→printf

若命令为：$ gcc –static –o myproc ./mylib.a main.o， 结果会怎样呢?

- 首先，扫描mylib.a，它会根据U中未解析的符号，来匹配mylib.a中哪个.o文件中包含了这个未解析的符号的定义。因为开始U为空，所以其中**两个.o模块都不被加入E中而被丢弃**。
- 然后，扫描main.o，将myfunc1加入U，直到最后，它都不能被解析。

因此，出现链接错误！

因为mylib.a中两个.o模块早已都已被丢弃

正确在命令行给出文件顺序举例

- **假设调用关系如下：**

➢ **func.o → libx.a 和 liby.a 中的函数，libx.a → libz.a 中的函数**

➢ libx.a 和 liby.a 之间、liby.a 和 libz.a 相互独立

则以下几个命令行都是可行的：

– gcc -static –o myfunc func.o libx.a liby.a libz.a

– gcc -static –o myfunc func.o liby.a libx.a libz.a

– gcc -static –o myfunc func.o libx.a libz.a liby.a

也就是说，libx.a必须要在libz.a之前

- **假设调用关系如下：**

➢func.o → libx.a 和 liby.a 中的函数

➢**libx.a → liby.a 同时 liby.a → libx.a**

也就是说，被链接的目标文件可以在命令行中重复出现

- **则以下命令行可行：**

– **gcc -static –o myfunc func.o libx.a liby.a libx.a**

程序的链接

分以下三个部分介绍

- 第一讲：目标文件格式
  - 什么是程序链接、链接的意义与过程
  - ELF目标文件格式、可重定位目标文件格式和可执行目标文件格式
- 第二讲：符号解析与重定位
  - 什么是符号表、怎么进行符号解析
  - 与静态库的链接
  - 重定位所需信息和重定位过程
  - 可执行文件的加载
- 第三讲：动态链接
  - 动态链接的特性
  - 程序加载时的动态链接、程序运行时的动态链接
  - 位置无关代码

三、重定位

链接的第二步：重定位，又分三步

1）首先，合并相同的节

- 在符号解析的基础上，将所有关联的目标模块（即集合E中所有目标模块）中相同的节合并成同一类型的新节

  例如：集合E中所有可重定位目标文件的.text节合并作为可执行目标文件中的.text节

2）然后，对定义符号进行重定位，确定其地址

- 根据每个定义符号相对于以前其所在节的相对位置以及该节在虚拟地址空间中的起始位置，为每个定义符号确定其在统一虚拟地址空间中的存储地址。
- **完成这一步后，每条指令和每个全局变量都可确定全局地址**

三、重定位

3）最后，对引用符号进行重定位

- 链接器对合并后新代码节(.text)和新数据节(.data)中的引用符号进行重定位，使其指向对应的**定义符号**在统一虚拟地址空间中的**首地址**。

那么，链接器怎么知道目标文件中哪些引用符号需要重定位、所引用的是哪个定义符号呢?

- 需要用到.o文件中.rel.data和.rel.text节中保存的重定位信息

汇编器，在遇到引用时，会为每个引用生成一个重定位条目记录重定位信息，并保存到目标文件的.rel.data和.rel.text**节中**

重定位信息记录

- 汇编器为每一处引用生成一个**重定位条目**记录重定位信息
  - ELF中重定位条目格式如下：

```
typedef  struct {
        int  offset;
        int  symbol:24,
             type: 8;
} Elf32_Rel;
```

当前需重定位的位置**相对于其所在节起始位置的字节偏移量**。若重定位的是变量的位置，则所在节是.data节。若重定位的是函数的位置，则所在节是.text节。

高24位，该引用符号所**引用的定义符号在符号表里的索引值**

低8位，该引用符号**重定位类型**， 最基本类型是：R_386_32(绝对地址方式)和R_386_PC32(相对寻址方式)

- 重定位条目和汇编后的机器代码都写在可重定位目标文件中
  - 数据引用的重定位条目在.rel.data节中
  - 指令引用的重定位条目在.rel.text节中

内容就是上述结构的结构数组，每个元素对应一个需要重定位的符号

重定位类型：

➢重定位类型与特定的处理器有关。IA-32处理器的重定位类型有16种之多。

➢最基本的重定位类型是R_386_PC32和R_386_32

1）**R_386_PC32**：指明引用处采用的是相对寻址方式，即

有效地址 = PC的内容+重定位后的32位地址（偏移）

（注意：PC的内容是下一条指令的地址）

（重定位后的32位地址简称为重定位值）

（调用指令call中的转移目标地址就采用相对寻址方式）

2）**R_386_32**：指明引用处采用绝对地址方式，即

有效地址 = 重定位后的32位地址

重定位过程举例

main.c

```
int buf[2] = {1, 2};
extern void swap();

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

swap.c

```
extern int buf[];

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

void swap()
{
  int temp;
  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

注意：局部变量temp分配在栈中，不会在过程外被引用，链接器不会考虑

- 在编译后，各**可重定位目标文件** (即main.o和swap.o) 的**符号表** (存储在**.symtab节**) 中，存储着所有此文件中定义的符号
- 在main.o和swap.o的重定位节(.rel.text、.rel.data)中有重定位信息(给出每个需要重定位的引用符号的引用位置、希望绑定的定义符号、重定位类型)

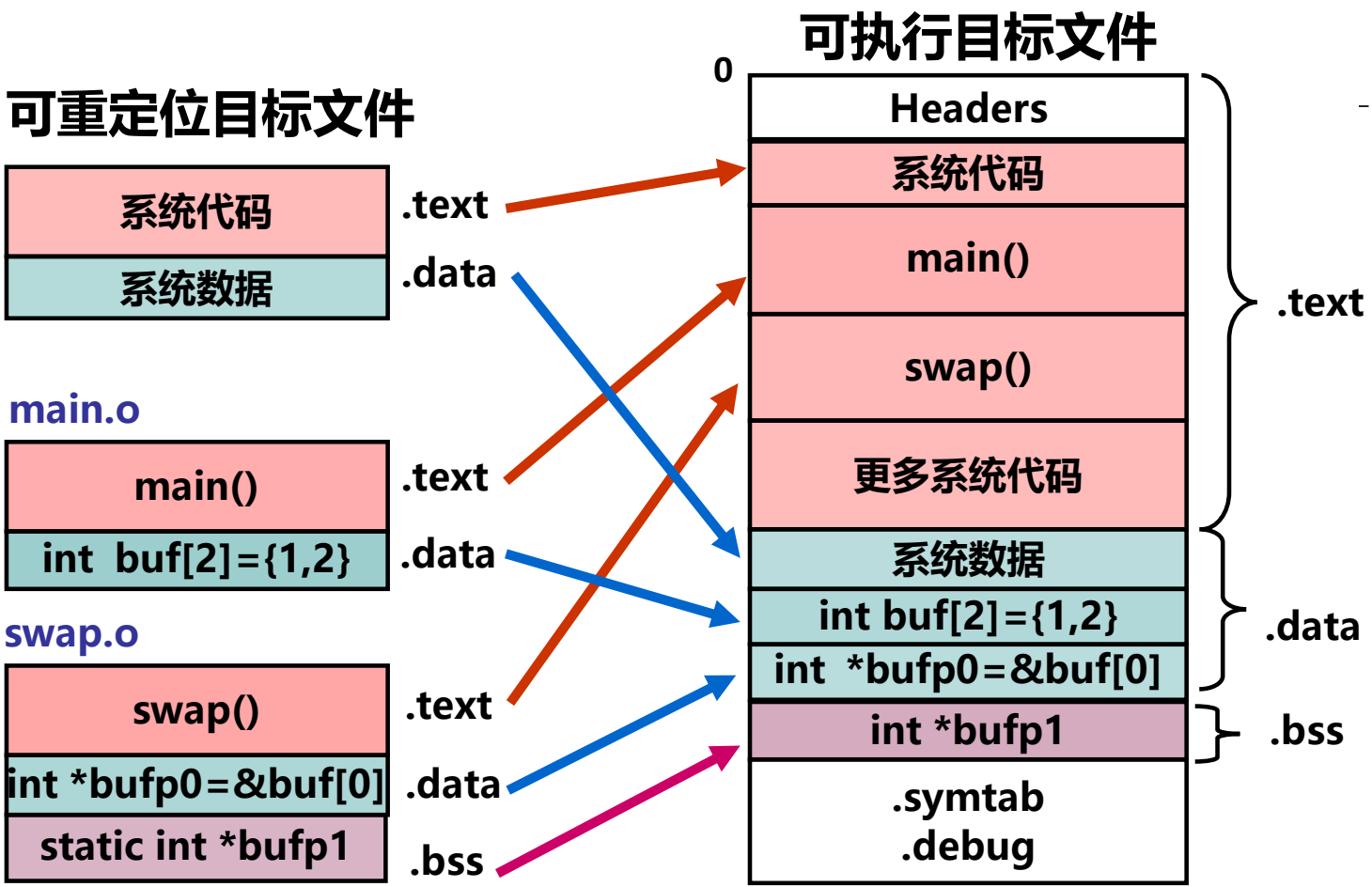
swap.c

```
extern int buf[];

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

void swap()
{
   int temp;
   bufp1 = &buf[1];
   temp = *bufp0;
   *bufp0 = *bufp1;
   *bufp1 = temp;
}
```

main.c

```
int buf[2] = {1, 2};
extern void swap();

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

- 在符号解析后，E中有main.o和swap.o两个模块；D中有所有定义符号
- 在main.o和swap.o的重定位节(.rel.text、.rel.data)中的每个需要重定位的引用符号已经和对应的定义符号绑定 (即已经在main.o和swap.o的重定位节.rel.text和.rel.data中记录绑定的定义符号在符号表中的索引)



接着，就合并集合E中所有可重定位目标文件。此后，所有定义符号都有了全局地址，接着就使用定义符号在统一虚拟地址空间中的首地址对对应的引用符号进行重定位（最基本重定位类型有R_386_PC32和R_386_32）

R_386_PC32的重定位方式：以main.o中swap函数重定位为例

在main.o重定位前

main.c

```
int buf[2] = {1, 2};
extern void swap();

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

main.o

```
Disassembly of section .text:
00000000 <main>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 e4 f0                and    $0xfffffff0,%esp
   6:   e8 fc ff ff ff          call   7 <main+0x7>
                                7: R_386_PC32 swap
   b:   b8 00 00 00 00          mov    $0x0,%eax
  10:   c9                      leave
  11:   c3                      ret
```

main的定义在 .text节中偏移为0处开始，占0x12字节

在main.o的.rel.text节中有重定位条目：

r_offset=0x7,
r_sym=10
r_type=R_386_PC32

readelf -r main.o命令导出来后的提示信息

r_sym=10说明该引用符号swap所引用的定义符号在main.o的符号表里的索引值是10

main.o中的符号表

- main.o中的符号表中最后三个条目

Num:	value	Size	Type	Bind	Ot	Ndx	Name
8:	0	8	Object	Global	0	3	buf
9:	0	18	Func	Global	0	1	main
10:	0	0	Notype	Global	0	UND	swap

swap在main.o的符号表中第10项，是未定义符号，类型和大小未知，说明swap是在main中被引用的由外部模块定义的符号。

➢ 为什么在重定位前，call指令是这样？

注：对于重定位类型是R_386_PC32，偏移地址（重定位值）=转移目标地址-PC

- 小端格式的负数的补码
- 即：**0x ff ff ff fc= -4**
- **为什么是-4?**

main.o

```
Disassembly of section .text:
00000000 <main>:
  0:        55                      push   %ebp
  1:        89 e5                   mov    %esp,%ebp
  3:        83 e4 f0                and    $0xfffffff0,%esp
  6:        e8 fc ff ff ff          call   7 <main+0x7>
                                    7: R_386_PC32 swap
  b:        b8 00 00 00 00          mov    $0x0,%eax
 10:        c9                      leave
 11:        c3                      ret
```

分析：此-4是编译器需重定位的4字节地址的**初始值**（**init**）

含义：在重定位时，**PC（call指令的下一条指令地址）**距离**需要重定位的地址**（即**ADDR(.text) + r_offset**）的偏移是**-4**，其中**ADDR(.text)** 表示**.text节**起始存储地址

R_386_PC32的重定位方式：以main.o中swap函数重定位为例

在main.o和swap.o合并后

假定：

- 在文件合并后，main函数对应的机器代码从**0x8048380**开始，占0x12字节
- swap紧跟在main函数之后，其机器代码首地址按4字节边界对齐

那么swap起始地址为多少?

- 因为main函数在.text节中占0x12字节，所以其结束地址为0x8048380+0x12=0x8048392
- 在4字节边界对齐的情况下，swap函数机器代码的起始地址是**0x8048394**

➢ 在重定位后，mian函数中call指令的机器代码是什么?

分析： 因为重定位类型是R_386_PC32，所以转移目标地址=PC+偏移地址（重定位值）。即重定位值=目标地址-PC，其中PC内容等于下一条指令的地址

在call命令处，PC的内容等于它的下一条指令的地址，即

PC=0x8048380+0x07+4=0x804838b

由于call命令执行时要转向swap函数，所以call命令的转移目标地址就是swap的地址0x8048394

所以，call命令中对应swap的重定位值= 转移目标地址-PC

= 0x8048394-0x804838b

=0x09

main函数对应机器代码首地址是0x8048380

swap函数机器代码首地址是0x8048394

在main.o的rel_text节中有重定位条目：
r_offset=0x7,
r_sym=10
r_type=R_386_PC32

main.o

```
Disassembly of section .text:
00000000 <main>:
  0:    55                      push   %ebp
  1:    89 e5                   mov    %esp,%ebp
  3:    83 e4 f0          and   $0xfffffff0,%esp
  6:    e8 fc ff ff ff    call   7 <main+0x7>
                                  7: R_386_PC32 swap
  b:    b8 00 00 00 00    mov    $0x0,%eax
 10:    c9                      leave
 11:    c3                      ret
```

➢ 在重定位后，mian函数中call指令的机器代码是什么?

重定位值=0x09

```
08048380 <main>:
 8048380:   55                  push  %ebp
 8048381:   89 e5               mov   %esp,%ebp
 8048383:   83 e4 f0            and   $0xfffffff0,%esp
 8048386:   e8 09 00 00 00      call  8048394 <swap>
 804838b:   b8 00 00 00 00      mov   $0x0,%eax
 8048390:   c9                  leave
 8048391:   c3                  ret
 8048392:   90                  nop
 8048393:   90                  nop
```

你能写出该call指令的功能描述吗?

假定每个函数要求4字节边界对齐,故填充两条nop指令

此时R[eip]=0x804838b

1) R[esp]← R[esp]-4
2) M[R[esp]] ←R[eip]

返回地址入栈

3) R[eip] ←R[eip]+0x9

构造跳转地址

所以，重定位后，call指令的机器代码为“e8 09 00 00 00”

IA-32是小端方式

R_386_32的重定位方式：以buf变量重定位为例（在重定位前）

main.c

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

int main()
……
```

经编译转化为main.o后

buf定义在.data节中偏移为0处，占8B，不需要重定位

main.o中.data节内容

```
Disassembly of section .data:

00000000 <buf>:
  0:   01 00 00 00 02 00 00 00
```

swap.c

```
extern int buf[];

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

void swap()
……
```

经编译转化为swap.o后

bufp0定义在.data节中偏移为0处，占4B，初值为0x0

readelf -r swap.o

swap.o中.data节内容

```
Disassembly of section .data:

00000000 <bufp0>:
  0:   00 00 00 00
       0: R_386_32  buf
```

swap.o中重定位节.rel.data中有重定位表项：
r_offset=0x0, r_sym=9, r_type=R_386_32
OBJDUMP工具解释后显示为“0：R_386_32 buf”

r_sym=9说明该引用符号所引用的定义符号buf在swap.o的符号表里的索引值是9

重定位方式是R_386_32

swap.o中的符号表

- swap.o中的符号表中最后4个条目

Num:	value	Size	Type	Bind	Ot	Ndx	Name
8:	0	4	Object	Global	0	3	bufp0
9:	0	0	Notype	Global	0	UND	buf
10:	0	36	Func	Global	0	1	swap
11:	4	4	Object	Local	0	COM	bufp1

buf是swap.o的符号表中第9项，是未定义符号，类型和大小未知， buf是在swap中被引用的由外部模块定义的符号。

R_386_32的重定位方式：以buf变量重定位为例（在重定位后）

- **假定：**在可重定位文件合并后，buf和bufp0都在同一个.data节中，并且连续存放，并且假定buf在可重定位文件合并后的存储地址ADDR(buf)=**0x08049620**
- **那么，在重定位后，bufp0的地址和内容变为什么？**

注：这里基于前面例子中 int *bufp0 = &buf[0];

– buf和bufp0都在.data中，且bufp0紧接在buf后，由于buf占8字节

因此，bufp0的地址为0x08049620+8= **0x08049628**

– 因为是**R_386_32重定位方式(绝对地址)，所以bufp0内容为buf的绝对地址，即0x08049620**

即“20 96 04 08”

（小端方式）

可执行目标文件中.data节的内容

```
Disassembly of section .data:

08049620 <buf>:
 8049620:    01 00 00 00 02 00 00 00

08049628 <bufp0>:
 8049628:    20 96 04 08
```

R_386_32的重定位方式：以swap.o重定位为例（在重定位前）

swap.c

```
extern int buf[];

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

void swap()
{
  int temp;

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

共有6处需要重定位

划红线处：8、c、11、1b、21、2a

```
Disassembly of section .text:
00000000 <swap>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 ec 10                sub    $0x10,%esp
   6:   c7 05 00 00 00 00 04    movl   $0x4,0x0
   d:   00 00 00
                8: R_386_32     .bss
                c: R_386_32     buf
  10:   a1 00 00 00 00          mov    0x0,%eax
                11: R_386_32    bufp0
  15:   8b 00                   mov    (%eax),%eax
  17:   89 45 fc                mov    %eax,-0x4(%ebp)
  1a:   a1 00 00 00 00          mov    0x0,%eax
                1b: R_386_32    bufp0
  1f:   8b 15 00 00 00 00       mov    0x0,%edx
                21: R_386_32    .bss
  25:   8b 12                   mov    (%edx),%edx
  27:   89 10                   mov    %edx,(%eax)
  29:   a1 00 00 00 00          mov    0x0,%eax
                2a: R_386_32    .bss
  2e:   8b 55 fc                mov    -0x4(%ebp),%edx
  31:   89 10                   mov    %edx,(%eax)
  33:   c9                      leave
  34:   c3                      ret
```

第2个元素

假定buf和bufp0在可执行目标文件中地址分别是0x08049620和0x08049628

假定bufp1在可执行文件中地址（此例中是文件合并后.bss节首地址）是0x08049700

&buf[1](c处重定位值）为0x8049620+0x4=0x08049624

8 (bufp1):
00 97 04 08

c (&buf[1]):
24 96 04 08

11 (bufp0):
28 96 04 08

1b (bufp0) :
28 96 04 08

21 (bufp1):
00 97 04 08

2a (bufp1):
00 97 04 08

```
6:    c7 05 00 00 00 00 04   movl   $0x4,0x0
d:    00 00 00
          8: R_386_32        .bss
          c: R_386_32        buf
```

bufp1 = &buf[1];

```
10:   a1 00 00 00 00         mov    0x0,%eax
          11: R_386_32       bufp0
15:   8b 00                  mov    (%eax),%eax
17:   89 45 fc               mov    %eax,-0x4(%ebp)
```

temp = *bufp0;

```
1a:   a1 00 00 00 00         mov    0x0,%eax
          1b: R_386_32       bufp0
1f:   8b 15 00 00 00 00      mov    0x0,%edx
          21: R_386_32       .bss
25:   8b 12                  mov    (%edx),%edx
27:   89 10                  mov    %edx,(%eax)
```

*bufp0 = *bufp1;

```
29:   a1 00 00 00 00         mov    0x0,%eax
          2a: R_386_32       .bss
2e:   8b 55 fc               mov    -0x4(%ebp),%edx
31:   89 10                  mov    %edx,(%eax)
```

*bufp1 =temp;

R_386_32的重定位方式：以swap.o重定位为例（在重定位后）

```
08048394 <swap>:
 8048394:  55                          push  %ebp
 8048395:  89 e5                       mov   %esp,%ebp
 8048397:  83 ec 10                    sub   $0x10,%esp
 804839a:  c7 05 00 97 04 08 24        mov $0x8049624,0x8049700
 80483a1:  96 04 08
 80483a4:  a1 28 96 04 08              mov  0x8049628,%eax
 80483a9:  8b 00                       mov  (%eax),%eax
 80483ab:  89 45 fc                    mov   %eax,-0x4(%ebp)
 80483ae:  a1 28 96 04 08              mov  0x8049628,%eax
 80483b3:  8b 15 00 97 04 08           mov  0x8049700,%edx
 80493b9:  8b 12                       mov  (%edx),%edx
 80493bb:  89 10                       mov  %edx,(%eax)
 80493bd:  a1 00 97 04 08              mov  0x8049700,%eax
 80493c2:   8b 55 fc                   mov  -0x4(%ebp),%edx
 80493c5:   89 10                      mov  %edx,(%eax)
 80493c7:   c9                         leave
 80493c8:   c3                         ret
```

swap.c

```
extern int buf[];

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

void swap()
{
  int temp;

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

仔细观察全局变量和局部变量的访问方式的不同

思考：上述重定位后的地址都是用的绝对地址，难道每次运行都能将这些单元加载到固定的内存单元里吗？

—— 真相：不会，这些是可执行目标文件对应的逻辑地址空间里的虚拟地址。当加载可执行目标文件执行时，数据可能会分配到内存的不同物理地址存放。

程序的链接

分以下三个部分介绍

- 第一讲：目标文件格式
  - 什么是程序链接、链接的意义与过程
  - ELF目标文件格式、可重定位目标文件格式和可执行目标文件格式
- 第二讲：符号解析与重定位
  - 什么是符号表、怎么进行符号解析
  - 与静态库的链接
  - 重定位所需信息和重定位过程
  - 可执行文件的加载
- 第三讲：动态链接
  - 动态链接的特性
  - 程序加载时的动态链接、程序运行时的动态链接
  - 位置无关代码

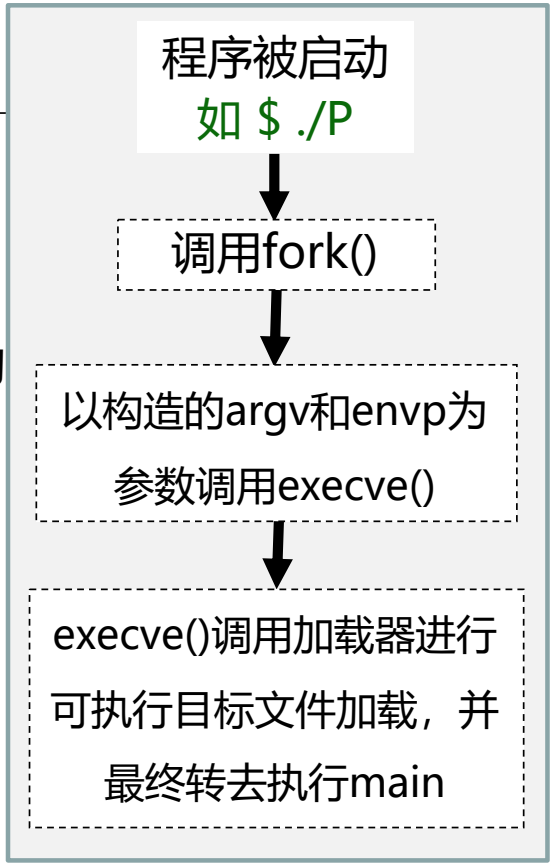
可执行文件的加载

当**启动一个可执行目标文件执行**时：

1）通过**execve**系统调用函数来启动**加载器**;
（linux系统下）

2）加载器(loader)根据可执行文件的**程序头表**中的信息，将**可执行目标文件相关内容（只读代码段**和**可读写数据段）**加载到与虚拟地址空间中

3）加载后，将PC（EIP）设定指向可执行目标文件入口点Entry point (即符号_start处)，最终转向main函数的执行。



_start: __libc_init_first → _init → atexit → main → _exit

可执行文件的加载

$ readelf –l main

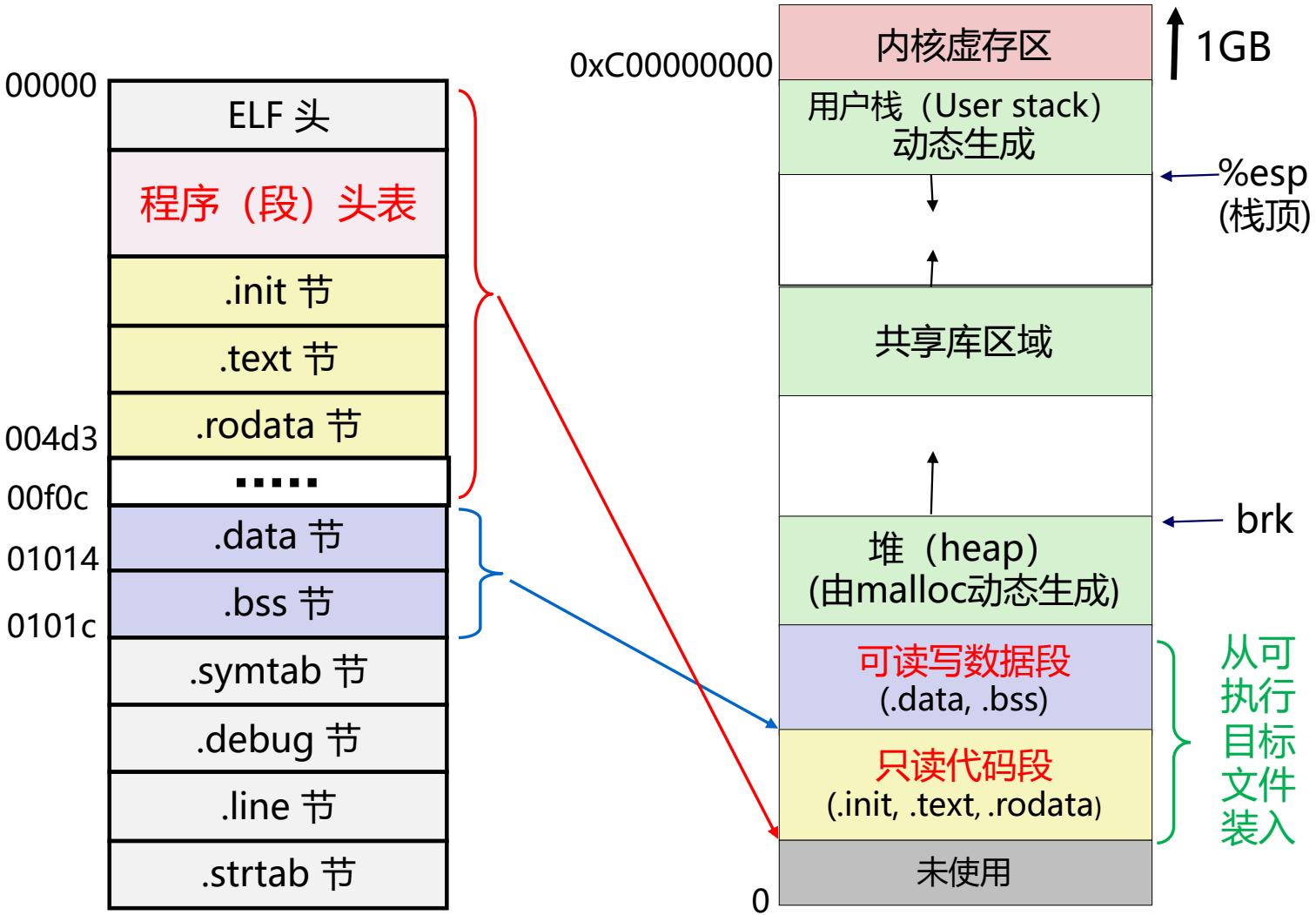
```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00100 0x00100 R E 0x4
  INTERP         0x000134 0x08048134 0x08048134 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x004d4 0x004d4 R E 0x1000
  LOAD           0x000f0c 0x08049f0c 0x08049f0c 0x00108 0x00110 RW  0x1000
  DYNAMIC        0x000f20 0x08049f20 0x08049f20 0x000d0 0x000d0 RW  0x4
  NOTE           0x000148 0x08048148 0x08048148 0x00044 0x00044 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
  GNU_RELRO      0x000f0c 0x08049f0c 0x08049f0c 0x000f4 0x000f4 R   0x1
```

有8个表项，其中两个为可装入段（即Type=LOAD）

第一可装入段：具有只读/执行权限（因为Flg=RE），所以是只读代码段。它的内容在可执行目标文件中从0x000000 (Offset=0x000000）开始到0x004d3（FileSiz= 0x004d4字节）结束。映射到虚拟地址0x08048000 (因为VirtAddr=0x08048000）开始，长度为0x004d4字节(因为MemSiz= 0x004d4字节)的区域，按0x1000 = 2^{12} =4KB对齐

第二可装入段：具有可读写权限（Flg=RW），所以是可读写数据段。它的内容是可执行目标文件中从0x000f0c开始的大小为0x00108字节的内容，映射到虚拟地址从0x08049f0c开始长度为0x00110字节的存储区域，按0x1000=4KB对齐。该存储区域(大小为0x00110=272字节)的前0x00108=264B会用.data节内容初始化，后面272-264=8B对应.bss节，初始化为0

可执行文件的加载

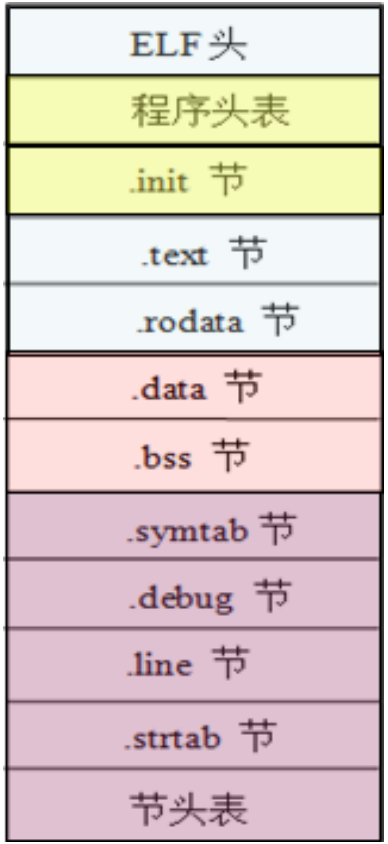

00000
ELF 头
程序（段）头表
.init 节
.text 节
004d3
.rodata 节
00f0c
.....
01014
.data 节
0101c
.bss 节
.symtab 节
.debug 节
.line 节
.strtab 节
0xC0000000
内核虚存区
1GB
用户栈（User stack）
动态生成
%esp
(栈顶)
共享库区域
brk
堆（heap）
(由malloc动态生成)
可读写数据段
(.data, .bss)
只读代码段
(.init, .text, .rodata)
从可执行目标文件装入
未使用
0

可执行文件的加载

$ readelf -h main 可执行目标文件的ELF头

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:    ELF32
  Data:     2's complement, little endian
  Version:  1 (current)
  OS/ABI:    UNIX - System V
  ABI Version:     0
  Type:    EXEC (Executable file)
  Machine:   Intel 80386
  Version:    0x1
  Entry point address:    x8048580
  Start of program headers:  52 (bytes into file)
  Start of section headers:    3232 (bytes into file)
  Flags:    0x0
  Size of this header:    52 (bytes)
  Size of program headers:    32 (bytes)
  Number of program headers:   8
  Size of section headers:     40 (bytes)
  Number of section headers:    29
  Section header string table index: 26
```

可执行目标文件入口点
第一条指令的地址



本章小结

- 链接涉及两种目标文件格式：可重定位目标文件、可执行目标文件
- ELF目标文件格式有链接视图和执行视图两种，前者是可重定位目标格式，后者是可执行目标格式。
  - 链接视图中包含ELF头、各个节以及节头表
  - 执行视图中包含ELF头、各种节组成的段以及程序头表（段头表）
- 链接分为静态链接和动态链接两种
  - 静态链接将多个可重定位目标文件中相同类型的节合并起来，以生成完全链接的可执行目标文件，其中所有引用符号在虚拟地址空间中都有确定的最终地址，因而可以直接被加载执行。
  - 动态链接的可执行目标文件是部分链接的，还有一部分引用符号的地址没有确定，需要根据共享库中定义符号的地址进行重定位，因而需要由动态链接器来加载共享库并重定位可执行目标文件中那部分引用符号
    - 加载时进行共享库的动态链接
    - 执行时进行共享库的动态链接

本章小结

- 链接过程需要完成符号解析和重定位两方面的工作
  - 符号解析的目的就是将引用符号与定义符号关联起来
  - 重定位的目的是分别合并代码节和数据节等，并根据代码节和数据节在虚拟地址空间中的位置，确定每个定义符号的最终地址，然后根据定义符号的最终地址来修改引用符号的地址。
- 在不同目标模块中可能会定义相同的全局符号，因为多个相同全局符号只能分配一个地址，因而链接器需要确定以哪个定义符号为准。
- 编译器根据全局符号的情况确定其为强符号还是弱符号，由链接器根据一套规则来确定哪个符号是唯一的定义符号，如果不了解这些规则，则可能无法理解程序执行的有些结果。
- 加载器在加载可执行目标文件时，实际上只是把可执行目标文件中的只读代码段和可读写数据段通过页表映射到了虚拟地址空间中确定的位置，并没有真正把代码和数据从磁盘装入主存。

本章作业

- 课后习题（P216）：

4、5、7、8、9、10、11

6月21前交作业

总结

- **第一章：计算机系统概述**
  - “存储程序”思想、冯·诺依曼结构及工作机制（冯·诺依曼机组成、IR、PC等各寄存器是什么？指令执行一般包含哪几个阶段？指令怎么执行？）、程序的开发和执行、ISA是什么
- **第二章：数据的机器级表示与处理**
  - 进位计数制、定点\浮点表示、定点数的编码（原码、补码、移码、反码）
  - 带符号整数和无符号整数的表示（真值和机器数怎么转化）、C语言中整数及其相互转换
  - 浮点数的表示（IEEE 754浮点数标准、真值和机器数怎么转化、规格化、精度和范围问题）
  - 数据存储方式：小端\大端方式
  - 数据的基本运算（左移运算和右移运算及溢出判断、位扩展运算和位截断运算等）
- **第三章：程序的转换与机器级表示**
  - 高级语言程序转换为机器代码的过程（预处理—编译—汇编—链接，生成可执行目标程序）
  - 高级语言源程序（过程调用、控制语句）对应的机器级代码是怎样的？（栈帧在过程调用时的变化过程及相关寄存器的变化情况、逆向工程）
  - 复杂数据类型（数组、结构体、联合体）在机器上是如何存储和访问的？
  - 数据的对齐（数据对齐策略和意义）
- **第四章：程序的链接**
  - 编译后生成的可重定位目标文件如何链接成最终的可执行目标文件
  - ✓ ELF目标文件格式（两种视图：链接视图、执行视图，程序头表信息的理解）
  - ✓ 符号表包括：本地符号、外部符号、全局符号（强符号、弱符号），符号表中已初始化变量符号在.data，未初始化变量符号在.bss，函数符号在.text，其中临时变量不在符号表
  - ✓ 符合解析过程（即如何在引用符号和定义符号之间建立关联）、与静态库的链接（与静态库链接时符号解析过程怎么样？如何才能成功链接？）
  - ✓ 重定位方法（最基本的重定位类型：R_386_PC32、R_386_32，分别怎么重定位）

总结

- **各章课后作业**
- **实验：**
  - **数据表示（按要求写相关代码实现给定数据处理功能）**
  - **二进制炸弹（逆向工程：反汇编后生成的AT&T格式代码阅读和理解）**
  - **缓冲区溢出攻击（攻击思路、攻击代码对应AT&T格式怎么写、攻击字符串怎么构造、攻击时栈帧变化情况及相关关键值应该是多少等）**