

## 作业 2：排序程序的编辑、编译和调试

17307130178 宁晨然

### 一、概述

熟悉开发环境、掌握开发和调试的基本过程以及工具。主要工作在于理解程序编译链接的过程，学会使用 gcc 进行基本的查看和调试。

### 二、实验过程

#### (一)、完成代码

1.BubbleSort.h:

```
#include <stdio.h>
void BubbleSort(int s[],int n);
```

2.BubbleSort.c:

```
#include "BubbleSort.h"
void BubbleSort(int s[],int n){
    int i,j;
    for(i=0;i<n-1;i++){
        for(j=0;j<n-1-i;j++){
            if(s[j]>s[j+1]){
                int temp = s[j+1];
                s[j+1] = s[j];
                s[j] = temp;
            }
        }
    }
}
```

3.add.h:

```
#include <stdio.h>
int add(int s[], int n);
```

4.add.c:

```
#include "add.h"
int add(int s[],int n){
    int sum = 0,i;
    for(i=0;i<n;i++){
        sum += s[i];
    }
    return sum;
}
```

5.printResult.h:

```
#include <stdio.h>
void BubbleSort(int s[],int n);
```

6.printResult.c:

```
#include "printResult.h"
void printResult(int s[], int n, char *str)
{
```

```

    printf("%s", str);
    int i;
    for (i = 0; i < n; i++)
        printf("%5d", s[i]);
    printf("\n");
}
7.main.c:
#include "add.h"
#include "BubbleSort.h"
#include "printResult.h"
#include <time.h>
#define bool char
#define true 1
#define false 0
#define LENGTH 10
int main()
{
    int a[LENGTH], i;
    int b[LENGTH];
    int randValue = 0;
    srand(time(NULL));
    for (i = 0; i < LENGTH; i++)
    {
        randValue = 1 + (int)rand() % LENGTH;
        a[i] = randValue;
        b[i] = a[i];
    }
    printResult(a, LENGTH, "\nrandom array: ");
    bool flag = true;
    while (flag)
    {
        printf("\n1.Bubble Sort\n2.sum\n3.print result\n4.exit");
        printf("\nchoose a number:");
        int number = 0;
        scanf("%d", &number);
        int sum = 0;
        switch (number)
        {
            case 1:
                BubbleSort(a, LENGTH);
                break;
            case 2:
                sum = add(a, LENGTH);
                printf("\nResult of sum: %d\n", sum);

```

```

        break;
    case 3:
        printResult(b, LENGTH, "\noriginal array:\t");
        printResult(a, LENGTH, "\nsorted array:\t");
        break;
    case 4:
        flag = false;
        break;

    default:
        printf("\nPlease choose a correct number and continue!");
        break;
}
printf("\nDone!\n\n");
}
return 0;
}

```

## (二)、程序处理

### 1. 可执行文件

使用 `gcc -o main main.c BubbleSort.c add.c printResult.c` 代码可以直接生成 main 的可执行文件。

```

chty627@ubuntu:~/Documents/homework2$ gcc -o main main.c BubbleSort.c add.c printResult.c

```



### 2. 可重定位文件，再链接

①可重定位：使用代码 `gcc -c main.c BubbleSort.c add.c printResult.c` 可以先生成.o 文件（可重定位文件），发现此时无法打开该文件，用 `vim` 进入查看也都是乱码，因为此时生成的文件是机器代码，如果需要查看需要用 `objdump` 反汇编成汇编语言后查看。

```

chty627@ubuntu:~/Documents/homework2$ gcc -c main.c BubbleSort.c add.c printResult.c

```

②链接：使用教案中的代码，发现会出现 Segmentation fault (core dumped) 的段错误，发现去掉 `-e main` 即可成功生成。此处助教说用 `-e_start main` 可以成功，但在我的电脑上没有成功，应该是 `gcc` 设置默认不相同。

先使用 `gcc -v main.o` 可以查看链接所需要的库。最后结合前面 `ld` 的代码：

```

ld -o main main.o add.o BubbleSort.o printResult.o --sysroot=/ --build-id --eh-frame-hdr -m elf_x86_64 --hash-style=gnu --as-needed -dynamic-linker /lib64/ld-linux-x86-64.so.2 -z relro /usr/lib/gcc/x86_64-linux-gnu/4.8/../../../../x86_64-linux-gnu/crt1.o /usr/lib/gcc/x86_64-linux-gnu/4.8/../../../../x86_64-linux-gnu/crti.o /usr/lib/gcc/x86_64-linux-gnu/4.8/crtbegin.o -L/usr/lib/gcc/x86_64-linux-gnu/4.8 -L/usr/lib/gcc/x86_64-linux-gnu/4.8/../../../../x86_64-linux-gnu -L/usr/lib/gcc/x86_64-linux-gnu/4.8/../../../../lib -L/lib/x86_64-linux-gnu -L/lib/../../lib -L/usr/lib/x86_64-linux-gnu -L/usr/lib/../../lib -L/usr/lib/gcc/x86_64-linux-gnu/4.8/../../../../ -lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed /usr/lib/gcc/x86_64-linux-gnu/4.8/crtend.o /usr/lib/gcc/x86_64-linux-gnu/4.8/../../../../x86_64-linux-gnu/crtn.o

```

### 3. objdump 的使用

该命令以一种可阅读的格式让程序员更多了解二进制文件的附带信息，其中的操作很多，此处我简单尝试了几个重要的命令：

① objdump -f main.o: 查看 main.o 的文件 ELF 头信息

```
chty627@ubuntu:~/Documents/homework2$ objdump -f main.o
main.o:      file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x0000000000000000
```

main.o 为可重定位目标文件格式，32 位系统对应的数据结构占 52 字节；上图可知程序起始位置为 0x0，对应 e\_entry（指定系统将控制权转移到的起始虚拟地址，入口点），可重定位文件为 0。图中还可以看出这个信息与 readelf -h main.o 得到的 elf 信息略有不同。

② objdump -d main.o: 反汇编 main.o 中需要执行指令的 section

```
chty627@ubuntu:~/Documents/homework2$ objdump -d main.o
main.o:      file format elf64-x86-64

Disassembly of section .text:
0: 55                push    %rbp
1: 48 89 e5          mov     %rsp,%rbp
4: 48 83 c4 80       add     $0xffffffffffff80,%rsp
8: c7 45 98 00 00 00 movl    $0x0,-0x68(%rbp)
f: bf 00 00 00 00    mov     $0x0,%edi
14: e8 00 00 00 00    callq   19 <main+0x19>
19: 48 89 c7          mov     %rax,%rdi
1c: b8 00 00 00 00    mov     $0x0,%eax
21: e8 00 00 00 00    callq   26 <main+0x26>
26: c7 45 94 00 00 00 movl    $0x0,-0x6c(%rbp)
```

起始位置从 0x0 开始，下面对应的是 .text（目标代码部分）的 section 的汇编语言和机器指令语言。

③ objdump -h main.o: 显示 section 的头信息（节头表）

```
chty627@ubuntu:~/Documents/homework2$ objdump -h main.o
main.o:      file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text          000001a6  0000000000000000  0000000000000000  00000040 2**0
CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data           00000000  0000000000000000  0000000000000000  000001e6 2**0
CONTENTS, ALLOC, LOAD, DATA
  2 .bss            00000000  0000000000000000  0000000000000000  000001e6 2**0
ALLOC
  3 .rodata         000000be  0000000000000000  0000000000000000  000001e8 2**3
CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .comment        00000024  0000000000000000  0000000000000000  000002a6 2**0
CONTENTS, READONLY
  5 .note.GNU-stack 00000000  0000000000000000  0000000000000000  000002ca 2**0
CONTENTS, READONLY
  6 .eh_frame       00000038  0000000000000000  0000000000000000  000002d0 2**3
CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
```

④ objdump -x main.o: 显示全部的头信息

⑤ objdump -s main.o: 显示全部头信息并且和它的对照十六进制代码

```
chty627@ubuntu:~/Documents/homework2$ objdump -s main.o
main.o:      file format elf64-x86-64

Contents of section .text:
0000 554889e5 4883c480 c7459800 000000bf  UH..H....E.....
0010 00000000 e8000000 004889c7 b8000000  ....H.....
0020 00e80000 0000c745 94000000 00eb54b8  ....E.....T...
0030 00000000 e8000000 0089c1ba 67666666  ....gfff.....
0040 89c8f7ea c1fa0289 c8c1f81f 29c289d0  .........)....
0050 c1e00201 d001c029 c189ca8d 42018945  .......)....B..E
0060 988b4594 48988b55 98895485 a08b4594  ..E..H..U..T...E.
0070 48988b54 85a08b45 94489889 5485d083  H..T...E..H..T...
0080 45940183 7d94097e a6488d45 a0ba0000  E...}...~.H.E....
0090 0000be0a 00000048 89c7e800 000000c6  ....H.....
00a0 458f01e9 ed000000 bf000000 00b80000  E.....
00b0 00000000 000000bf 00000000 b8000000  ....H.....
```

⑥objdump -S main.o:输出 C 源代码和反汇编出来的指令对照的格式

```
chty627@ubuntu:~/Documents/homework2$ objdump -S main.o
main.o:      file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
0:  55                      push    %rbp
1:  48 89 e5                mov     %rsp,%rbp
4:  48 83 c4 80             add     $0xffffffffffff80,%rsp
8:  c7 45 98 00 00 00 00    movl   $0x0,-0x68(%rbp)
f:  bf 00 00 00 00         mov     $0x0,%edi
14: e8 00 00 00 00         callq  19 <main+0x19>
```

## 4. GDB 的使用

我安装的有 pwndbg 的工具，比自带的 gdb 的调试更方便，处理的信息更多。

```
chty627@ubuntu:~/Documents/homework2$ gcc -g -o main main.c BubbleSort.c add.c printResult.c
chty627@ubuntu:~/Documents/homework2$ gdb main
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
```

①disassemble main:先反汇编后得到函数地址，方便查看地址。

```
pwndbg> disassemble main
Dump of assembler code for function main:
0x00000000004006cd <+0>:      push    rbp
0x00000000004006ce <+1>:      mov     rbp, rsp
0x00000000004006d1 <+4>:      add     rsp, 0xffffffffffff80
0x00000000004006d5 <+8>:      mov     DWORD PTR [rbp-0x68], 0x0
0x00000000004006dc <+15>:     mov     edi, 0x0
0x00000000004006e1 <+20>:     call   0x4005a0 <time@plt>
```

②break \*:设置断点，程序运行后可以在此截断

```
pwndbg> break *0x000000000040076c
Breakpoint 1 at 0x40076c: file main.c, line 25.
```

③info r:达到断点后，查看此时程序中寄存器的状态

```
Breakpoint *0x000000000040076c
pwndbg> info r
rax      0xa      10
rbx      0x0      0
rcx      0x7ffff7af4154  140737348845908
rdx      0x7ffff7dd18c0  140737351850176
rsi      0xa      10
```

④si N:单步执行，此时程序下移一行机器代码 eip 更新

```
pwndbg> si 1
26      while (flag)
```

⑤x N EXPR:扫描内存

```
pwndbg> x $rsp
0x7ffff7fd00: 0x00000000
```

⑥bt:打印栈帧链

```
pwndbg> bt
#0  main () at main.c:26
#1  0x00007ffff7a05b97 in __libc_start_main (main=0x4006cd <main>, argc=1, argv=
0x7ffff7fd00, init=<optimized out>, fini=<optimized out>, rtld_fini=<optimized
out>, stack_end=0x7ffff7fd00) at ../csu/libc-start.c:310
#2  0x00000000004005fa in _start ()
```

⑦c:继续执行

gdb 的使用很方便，极其易于调试，使用 pwndbg 后还可以直接查看栈、源码、汇编代码，调试更加方便。

```
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA

[ REGISTERS ]
RAX 0xa
RBX 0x0
RCX 0x7ffff7fd00 (utils.c:10) ← cmp rax, -0x1000 /* 'H' */
RDX 0x7ffff7dd18c0 (.io_stdio.c:1) ← 0x0
RDI 0x1
RSI 0xa
R8 0xa
R9 0x7ffff7fd4c0 ← 0x7ffff7fd4c0
R10 0x3
R11 0x246
R12 0x4006cd (.cabi) ← xor ebp, ebp
R13 0x7ffff7fd00 ← 0x1
R14 0x0
R15 0x0
RBP 0x7ffff7fd00 → 0x4006cd (.io_stdio.c:1) ← push r15
RSP 0x7ffff7fd00 ← 0x0
RIP 0x40076c (.cabi) ← mov byte ptr [rbp - 0x71], 1

[ DISASM ]
→ 0x40076c <main+159> mov byte ptr [rbp - 0x71], 1
0x400770 <main+163> jmp main+405 <main+163>
0x400802 <main+405> cmp byte ptr [rbp - 0x71], 0
0x400806 <main+409> jne main+168 <main+168>
0x400775 <main+168> mov edi, 0x400ab0
0x40077a <main+173> mov eax, 0
0x40077f <main+178> call printf@plt <main+178>
0x400784 <main+183> mov edi, 0x400ae3
0x400789 <main+188> mov eax, 0
0x40078e <main+193> call printf@plt <main+193>
0x400793 <main+198> mov dword ptr [rbp - 0x70], 0
```

5.readelf -h main.o 的功能

```
chty627@ubuntu:~/Documents/homework2$ readelf -h main.o
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                                ELF64
  Data:                                      2's complement, little endian
  Version:                               1 (current)
  OS/ABI:                                UNIX - System V
  ABI Version:                           0
  Type:                                  REL (Relocatable file)
  Machine:                               Advanced Micro Devices X86-64
  Version:                               0x1
  Entry point address:                   0x0
  Start of program headers:              0 (bytes into file)
  Start of section headers:             1992 (bytes into file)
  Flags:                                  0x0
  Size of this header:                   64 (bytes)
  Size of program headers:               0 (bytes)
  Number of program headers:             0
  Size of section headers:               64 (bytes)
  Number of section headers:             13
  Section header string table index:    12
```

- ①魔数：表示为 ELF 文件，64 位格式，小端方式存放，版本为 1，文件类型为 REL（可重定位文件）
- ②入口点：可重定位文件，入口点为 0x0
- ③无程序头表；节头表离文件起始处的偏移为 1992 字节
- ④表项数 13 个，字符串表在节头表的索引为 12
- ⑤每个表项大小占 64 字节

所以 readelf -h main.o 的功能即，读取 main.o 可重定位文件的 elf 头信息，并且解析。可以用 c 程序实现这个过程。下图为 test.c 的运行结果。

```
chty627@ubuntu:~/Documents/homework2$ ./test
file total size is:2824 bytes
Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
Class: ELF64
Data: 2's complement, little endian
Version: 1 (current)
OS/ABI: UNIX -System V
ABI Version: 0
Type: 1 (REL)
Machine: 62 (Advanced Micro Devices X86-64)
Version: 0x1
Entry point address: 00
Start of program headers: 0(bytes into file)
Start of section headers: 1992(bytes into file)
Flags: 0
Size of this header: 64(bytes)
Size of program headers: 0(bytes)
Number of program headers: 0
Size of section headers: 64(bytes)
Number of section headers: 13
Section header string table index: 12
```

首先根据 readelf -h main.o 的结果可知，此时的 elf 是 64 位，占用 64 字节，和书上的略有不同，在网上查询可知 64 的 struct 结构如下，与 32 略有不同。

Name	Size	Alignment	Purpose
Elf64_Addr	8	8	Unsigned program address
Elf64_Off	8	8	Unsigned file offset
Elf64_Half	2	2	Unsigned medium integer
Elf64_Word	4	4	Unsigned integer
Elf64_Sword	4	4	Signed integer
Elf64_Xword	8	8	Unsigned long integer
Elf64_Sxword	8	8	Signed long integer
unsigned char	1	1	Unsigned small integer

```
typedef struct elf64_hdr {
    unsigned char    e_ident[EI_NIDENT];
    Elf64_Half e_type;
    Elf64_Half e_machine;
    Elf64_Word e_version;
    Elf64_Addr e_entry;
    Elf64_Off e_phoff;
    Elf64_Off e_shoff;
    Elf64_Word e_flags;
    Elf64_Half e_ehsize;
    Elf64_Half e_phentsize;
    Elf64_Half e_phnum;
    Elf64_Half e_shentsize;
    Elf64_Half e_shnum;
    Elf64_Half e_shstrndx;
} Elf64_Ehdr;
```

由 struct 可知结构的组成部分，现在只需要把 elf 读入进来即可。fileopen 打开 main.o 文件，查找到 ELF\_header 所在位置（其实就是从 0 开始的 64 字节），然后填装进预设好的 struct 当中，根据 printf 把其中的信息读取出来，即可实现类似 readelf 的功能。值得注意的是，readelf -h 中前面有五行代码是对魔数的解析，通过直接输出魔数不能达到效果，所以猜测机内应该存有相关的定义，此处的魔数只是对定义数组的取值，但是我没有找到这个定义的文件。例如 x86\_64 中常见的 OS/ABI:UNIX - Linux / UNIX - System V/ UNIX - GNU 但是此处机器使用的是其中的第二个，应该有数组取得信息的过程。

过程中又遇到空指针的问题，特别是在构造 struct Elf\_header 的指针时，应当使用代码 Elf64\_Ehdr\* Elf\_header = (Elf64\_Ehdr\*)malloc(sizeof(Elf64\_Ehdr));避免出现空指针无法操作的报错现象。

test.c 源码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define EI_NIDENT 16

/* Unsigned. */
typedef unsigned char      uint8_t;
typedef unsigned short int uint16_t;
#ifdef __uint32_t_defined
typedef unsigned int       uint32_t;
# define __uint32_t_defined
#endif
#if __WORDSIZE == 64
typedef unsigned long int  uint64_t;
#else
__extension__
typedef unsigned long long int  uint64_t;
#endif

typedef uint16_t Elf64_Half;
typedef uint32_t Elf64_Word;
typedef uint64_t Elf64_Addr;
typedef uint64_t Elf64_Off;

typedef struct elf64_hdr {
    unsigned char  e_ident[EI_NIDENT];
    Elf64_Half e_type;
    Elf64_Half e_machine;
    Elf64_Word e_version;
    Elf64_Addr e_entry;
    Elf64_Off e_phoff;
    Elf64_Off e_shoff;
```

```

    Elf64_Word e_flags;
    Elf64_Half e_ehsize;
    Elf64_Half e_phentsize;
    Elf64_Half e_phnum;
    Elf64_Half e_shentsize;
    Elf64_Half e_shnum;
    Elf64_Half e_shstrndx;
} Elf64_Ehdr;

void elf_head(FILE *file )
{
    FILE *file_elf = file;
    int i = 0;
    int MagNum = 0;
    unsigned long file_size = 0;
    Elf64_Ehdr* Elf_header = (Elf64_Ehdr*)malloc(sizeof(Elf64_Ehdr));
    //开空间, 构造 struct Elf64_Ehdr
    fseek(file_elf,0,SEEK_END);
    file_size = ftell(file_elf);
    fseek(file_elf,0,SEEK_SET);
    printf("file total size is:%ld bytes\n",file_size);
    fread(Elf_header,sizeof(Elf64_Ehdr),1,file_elf);    //读入

    printf("Magic:\t");
    for(MagNum=0; MagNum<16; MagNum++)
    {
        printf("%02x ",Elf_header->e_ident[MagNum]);
    }
    //下面的信息都是由魔数得出, 猜测机内有一个数组存放该信息, 但为了输出我直接用
    printf 输出原始信息和解释信息
    printf("\nClass:                \tELF64\n");
    printf("Data:                \t2's complement, little endian\n");
    printf("Version:                \t%x\n",Elf_header->e_ident[6]);
    printf("OS/ABI:                \tUNIX -System V\n");
    printf("ABI Version:            \t0\n");
    //下面格式略有不同
    printf("Type:                \t%d (REL)\n",Elf_header->e_type);
    printf("Machine:            \t%d (Advanced Micro Devices X86-64)\n",Elf_header->e_machine);
    printf("Version:            \t%x\n",Elf_header->e_version);

    printf("Entry point address:        \t%#02x\n",(unsigned
int)(Elf_header->e_entry));

```



```

        printf("Start of program headers:           \t%d(bytes into
file)\n", (int)Elf_header->e_phoff);
        printf("Start of section headers:           \t%d(bytes into
file)\n", (int)Elf_header->e_shoff);
        printf("Flags:                             \t%x\n", Elf_header->e_flags);
        printf("Size of this header:
\t%d(bytes)\n", Elf_header->e_ehsize);
        printf("Size of program headers:
\t%d(bytes)\n", Elf_header->e_phentsize);
        printf("Number of program headers:         \t%d\n", Elf_header->e_phnum);
        printf("Size of section headers:
\t%d(bytes)\n", Elf_header->e_shentsize);
        printf("Number of section headers:         \t%d\n", Elf_header->e_shnum);
        printf("Section header string table index:\t%d\n", Elf_header->e_shstrndx);
        return;
    }

int main()
{
    char *path = "main.o";
    FILE *file_elf = NULL;
    file_elf = fopen(path, "rb");
    if(file_elf == NULL){
        printf("Path error!\n");
        return 0;
    }
    elf_head(file_elf);
    fclose(file_elf);
    return 0;
}

```

### 三、实验问题

(1) 分析同一个源程序在不同机器上生成的可执行目标代码是否相同。

**提示：**从多个方面（如 ISA、OS 和编译器）来分析。

不相同。

同一个源程序例如 hello.c 文件，代码都相同，但是由于不同机器的硬件/OS/编译器不同会导致生成的可执行目标代码不同。首先对于不同的硬件，例如 CPU 不同，则一条指令的执行过程不同、CPU 的指令集构架 (ISA) 定义会把机器码数值翻译成指令，相当于同一道菜的不同菜谱具有不同的操作过程。

其次对于操作系统 OS 不同，例如 win32/win64/ios/linux 的大小端不同、数据存储方式不同（例如 64 位和 32 位的数据大小不同），则对应的指令也不同，例如 32 位和 64 位的符号扩展操作不同。最后编译器也会起到很大的作用，编译器不同，所生成的可重定位代码不同（机器译码更加不同），编译器的环境配置也会影响代码的编译过程，例如 ld 链接过

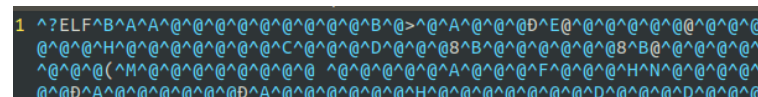
程，不同编译器不同。

综上所述，同一个源程序在不同机器上生成的可执行目标代码不同。

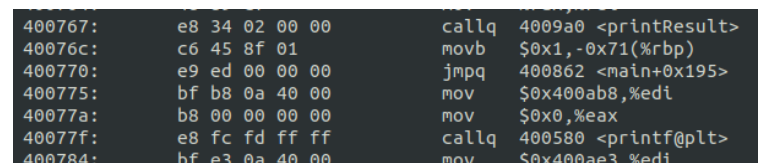
**(2) 你能在可执行目标文件中找出函数 printf ()对应的机器代码段吗？能的话，请标示出来。**

不能。

因为源程序中的 printf()函数在可执行文件中已经完全翻译成机器语言，而机器语言中肯定会含有 printf()函数的调用，但是我们并不清楚机器语言从哪一段开始翻译、翻译过程如何。用 vim 打开 main 可执行目标文件，显示为乱码。



但是我们可以通过 `objdump -d main` 将源码反汇编成汇编语言查看 printf()的位置，下图是其中一个时候调用了 printf@plt 函数：



**(3) 为什么源程序文件的内容和可执行目标文件的内容完全不同？**

打开可执行目标文件，可以发现可执行文件由于被机器语言翻译后，无法用字符的形式显示出来，所以用 vim main 打开后全是乱码。但是源程序文件，是可以直接编写、直接修改操作的，用于程序员来修改、操作、查看，适用于给人看的各种语言代码。可执行目标文件是给机器操作使用的机器语言，并不是给人来看的，如果需要了解机器操作的过程，可以反汇编成汇编语言后查看，可以了解机器操作的具体方式。

## 四、实验感受

本次实验的理解内容居多，操作很多都学过，更多的实现细节在了解程序链接、编译的过程。本次学习到了很多关于程序在机器内部链接调用、编译的整个过程，更加深入了解到计算机如何成功运行我们写好的 c 语言程序的过程。也更加学会使用 gdb 进行调试，用 linux 代码查看各种程序运行的详情。特别是使用 c 语言实现 readelf 的过程，需要对 elf header 进行深入了解后，才能使用 c 语言写出程序，这个过程相对艰辛。

[reference]

1.ELF 文件解析：

<https://blog.csdn.net/feglass/article/details/51469511>