HW9

1

1

1. 两个C文件link1.c和link2.c的内容分别如下

```
1.      int buf[1] ={100};
```

和

```
1.      #include <stdio.h>
2.      extern int *buf;
3.      int main() { printf("%d\n", *buf); }
```

1）执行 `gcc -c -nostdinc link2.c` 时会输出如下错误：

```
1. link2.c:1:19: error: no include path in which to search for stdio.h
2.  #include <stdio.h>
3.                    ^
4. link2.c: In function ‘main’:
5. link2.c:4:2: warning: implicit declaration of function ‘printf’ [-Wimplicit-function-declaration]
6.   printf("%d\n", *buf);
7.   ^~~~~~
8. link2.c:4:2: warning: incompatible implicit declaration of built-in function ‘printf’
9. link2.c:4:2: note: include ‘<stdio.h>’ or provide a declaration of ‘printf’
```

但是执行 `gcc -c link2.c` 时会正确生成link2.o文件，请问 `-nostdinc` 的作用是什么，使用该选项产生的error信息是什么阶段报出的错误？（提示：你可以通过使用-E、-S等选项来帮助你分析）

- `-nostdinc` 的作用：不在标准系统目录中搜索头文件。只有通过 `-I`，`-iquote`，`-isystem` 和/或 `-dirafter` 选项显式指定的目录（可能还有当前文件目录）才被搜索
- `gcc` 选项

  `-E`：只激活预处理

  `-S`：只激活预处理和编译，把文件编译成为汇编代码

  `-c`：只激活预处理,编译和汇编，生成obj文件
- 预处理阶段。这个阶段中，以 `#` 开头的行都将被预处理器当做预处理命令来解释。由于无法找到头文件 `stdio.h`，故无法对 `#include<stdio.h>` 进行宏展开

```
link2.c:1:19: error: no include path in which to search for stdio.h
    1 | #include <stdio.h>
      |                   ^
```

2

2）分别执行

`gcc -o link-s -static link2.c link1.c`

或

`gcc -o link link2.c link1.c`

将得到可执行文件 `link-s` 和 `link`，比较文件大小，并用 `objdump -dS <可执行文件名>` 和 `nm <可执行文件名>` 查看可执行文件 `link-s` 和 `link` 的反汇编代码和符号信息，请说明它们的区别和各自的好处。

- 文件大小：静态的 `link-s` 大小为852KB，动态的 `link` 大小只有17KB
- `link-s` 的反汇编码比 `link` 的反汇编码要长很多，主要是包含了库文件中实现 `printf` 的部分，其符号信息也比 `link` 多了很多，主要多了库文件中关于 `printf` 的符号信息

- `link`的反汇编码则较为短小，且符号名也较少，不包含库文件中`printf`部分，只有如下部分

```
0000000000001050 <printf@plt>:
    1050:   f3 0f 1e fa             endbr64
    1054:   f2 ff 25 75 2f 00 00    bnd jmpq *0x2f75(%rip)        # 3fd0
<printf@GLIBC_2.2.5>
    105b:   0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
```

符号表里也只有如下符号：

```
                 U printf@@GLIBC_2.2.5
```

- 上述都是由于二者链接方式不同导致的。静态链接是由链接器在链接时将库的内容加入到可执行程序中。链接器是一个独立程序，将一个或多个库或目标文件（先前由编译器或汇编器生成）链接到一块生成可执行程序。而动态链接（`Dynamic Linking`），把链接这个过程推迟到了运行时再进行，在可执行文件装载时或运行时，由操作系统的装载程序加载库。
- 使用静态链接库的好处：
    - 运行时与函数库不再有联系，便于移植。只需保证在开发者的计算机中有正确的`.lib`文件，在以二进制形式发布程序时不需考虑在用户的计算机上`.lib`文件是否存在及版本问题。
    - 代码装载速度快，执行速度略比动态链接库快
- 使用动态链接库的好处：
    - 节省空间和资源，动态库在程序运行时才被载入
    - 不同的应用程序如果调用相同的库，那么在内存里只需要有一份该共享库的实例，节省空间，避免内存中驻留冗余的具有相同功能的代码。不同编程语言编写的程序只要按照函数调用约定就可以调用同一个DLL函数
    - 动态库在程序运行时才被载入，这也解决了静态库对程序更新 、部署、发布带来的麻烦。用户只需要动态增量更新库即可
    - 适用于大规模的软件开发，使开发过程独立、耦合度小，便于不同开发者和开发组织之间进行开发和测试
    - DLL文件与EXE文件独立，只要输出接口不变（即名称、参数、返回值类型和调用约定不变），更换DLL文件不会对EXE文件造成任何影响，因而极大地提高了可维护性和可扩展性

3

3）执行 `gcc -o link -nostdlib link2.c link1.c` 时，会输出如下错误信息：

```
/usr/bin/ld: 警告：无法找到项目符号 _start；缺省为 000000000040017c
/tmp/cccC0pRN.o：在函数‘main’中：
link2.c:(.text+0x1a)：对‘printf’未定义的引用
collect2: error: ld returned 1 exit status
```

请解释为什么会使用_start（提示：可以结合上一题中得到的link的反汇编代码来理解），这些错误是在什么阶段发生的

- `_start` 是程序真正的起始点，该符号的地址是程序开始时跳转到的地址
    - 通常，名为 `_start` 的函数由一个名为 `crt0` 的文件（采用 `crt0.o` 的目标文件形式）提供，它包含用于C运行时环境的启动代码。它设置一些东西，填充参数数组argv，计算参数个数，然后调用main函数。
    - 该文件常采用汇编语言编写，链接器自动的将它包括入它所生成的所有可执行文件中。
    - `crt0` 包含大多数运行时库的基本部分。因此，它进行的确切工作依赖于程序的编译器、操作系统和C标准库实现
    - 在 `link` 的反汇编码中也可以找到对应的 `_start` 部分

```
0000000000001060 <_start>:
    1060:   f3 0f 1e fa             endbr64
    1064:   31 ed                   xor    %ebp,%ebp
    1066:   49 89 d1                mov    %rdx,%r9
    1069:   5e                      pop    %rsi
    106a:   48 89 e2                mov    %rsp,%rdx
    106d:   48 83 e4 f0             and    $0xfffffffffffffff0,%rsp
    1071:   50                      push   %rax
    1072:   54                      push   %rsp
    1073:   4c 8d 05 76 01 00 00    lea    0x176(%rip),%r8        # 11f0 <__libc_csu_fini>
    107a:   48 8d 0d ff 00 00 00    lea    0xff(%rip),%rcx        # 1180 <__libc_csu_init>
    1081:   48 8d 3d c1 00 00 00    lea    0xc1(%rip),%rdi        # 1149 <main>
    1088:   ff 15 52 2f 00 00       callq  *0x2f52(%rip)        # 3fe0 <__libc_start_main@GLIBC_2.2.5>
    108e:   f4                      hlt
    108f:   90                      nop
```

- 这些错误在连接阶段产生
    - `gcc` 命令默认会调用 `ld /usr/lib/crt1.o /usr/lib/crti.o ... -lc -dynamic-linker /lib/ld-linux.so.2` 这样的连接命令
    - `-nostdlib` 表示链接时不使用标准的系统启动文件或库，只有用户指定的文件会被连接

- 由此可知上述命令不会将需要的库目标文件`crt0.o`连接，故会报Warning：无法找到项目符号`_start`
- 同时由于没有连接动态库，所以会出现错误"对printf未定义的引用"

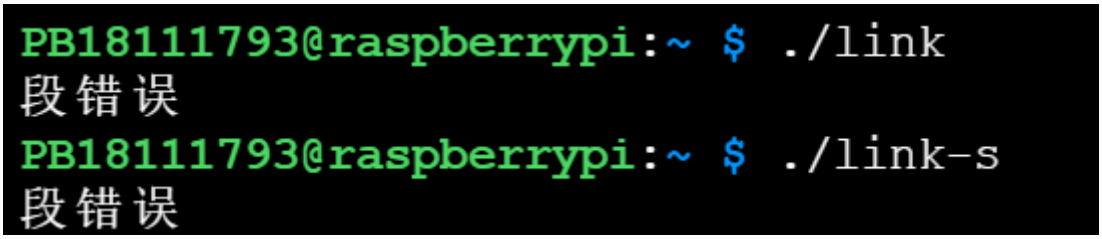
4

4）执行第2）小题得到的可执行文件，会有什么样的运行结果？请在32位和64位系统分别进行实验，再进行分析说明。

- 在64位系统中，结果如下

```
root@LAPTOP-HRJHHKLT:/mnt/d/mywork/compiler_lab/hw9/1# ./link
Segmentation fault
root@LAPTOP-HRJHHKLT:/mnt/d/mywork/compiler_lab/hw9/1# ./link-s
Segmentation fault
```

- 在32位系统中，结果如下



- 可以看到结果均为段错误(segmentation fault)
- C中每个文件单独编译，连接前不会知道源文件中buf的类型。连接时也不会进行类型检查，故可生成目标程序。
- 首先`link1.c`中`buf`是一个长度为1的一维数组，对应的内存单元值为100
- 在`link2.c`中`buf`是一个指向整型的指针
- 在连接后，`link.c`中的`printf`访问了`*buf`，相当于访问内存地址为100的内存单元中的值。而地址100对应的是内核态空间（不管是32位系统还是64位系统），用户态的代码是没有权限进行访问的，故会出现段错误，这从32位和64位的汇编代码中也可看出

  64位（只取了其中的关键部分）

  link1.s

```
    .file   "link1.c"
    .text
    .globl  buf
    .data
    .align 4
    .type   buf, @object
    .size   buf, 4
buf:
    .long   100
    .ident  "GCC: (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0"
    .section    .note.GNU-stack,"",@progbits
```

link2.s

```
    movq    buf(%rip), %rax
    movl    (%rax), %eax
```

link1.c中将buf声明为一维数组，buf对应该数组内容的起始地址。执行link2.c中的`*buf`时，会将100（`buf(%rip)`）存入`rax`，再将地址100中的值放入`eax`，访问内核态地址，出现段错误

32位

link1.s

```
        .file   "link1.c"
        .text
        .global buf
        .data
        .align  2
        .type   buf, %object
        .size   buf, 4
buf:
        .word   100
        .ident  "GCC: (Raspbian 8.3.0-6+rpi1) 8.3.0"
        .section        .note.GNU-stack,"",%progbits
```

link2.s

```
        .file   "link2.c"
        .text
        .section        .rodata
        .align  2
.LC0:
        .ascii  "%d\012\000"
        .text
        .align  2
        .global main
        .arch armv6
        .syntax unified
        .arm
        .fpu vfp
        .type   main, %function
main:
        @ args = 0, pretend = 0, frame = 0
        @ frame_needed = 1, uses_anonymous_args = 0
```

```
        push    {fp, lr}
        add     fp, sp, #4
        ldr     r3, .L3 #r3=buf数组的首地址
        ldr     r3, [r3] #r3=100
        ldr     r3, [r3] #r3=[100]
        mov     r1, r3
        ldr     r0, .L3+4
        bl      printf
        mov     r3, #0
        mov     r0, r3
        pop     {fp, pc}
.L4:
        .align  2
.L3:
        .word   buf
        .word   .LC0
        .size   main, .-main
        .ident  "GCC: (Raspbian 8.3.0-6+rpi1) 8.3.0"
        .section        .note.GNU-stack,"",%progbits
```

和64位原理相同，解释在注释中给出

2

2. 教材11.13 两个C文件long.c和short.c的内容分别是

```
long i = 32768□2;
```

和

```
extern short i;
main() { printf("%d\n", i); }
```

在X86/Linux系统上，用cc long.c short.c命令编译这两个文件，能否得到可执行目标程序？若能得到目标程序，运行时是否报错？若不报错，则运行结果输出的值是否为65536？若不等于65536，原因是什么？

- 能得到可执行目标程序。`cc`指令用于文件的编译和连接
    - **编译**是以`.c`文件为单位的，不会发现两个文件之间的变量`i`的类型错误。
    - 数据的类型信息没有附加在目标文件中，故**连接**时也不会发现两个文件中变量`i`类型不一致
- 运行时不报错
- 运行输出的值为0
    - 首先测试C中不同类型在存储中所占的字节数

```c
#include<stdio.h>

int main(){
    printf("%ld\n", sizeof(char));
    printf("%ld\n", sizeof(short));
    printf("%ld\n", sizeof(int));
    printf("%ld\n", sizeof(long));
}
```

输入如下

```
root@LAPTOP-HRJHHKLT:/mnt/d/mywork/compiler_lab/hw9# ./size
1
2
4
8
```

- 原因：x86机器是小尾端模式，对于整形数据，低地址放低位，高地址放高位，这可以通过如下程序测试

```c
#include<stdio.h>

int main(){
    short int x;
    char x0,x1;
    x=0x1122;
    x0=((char*)&x)[0]; //低地址单元
    x1=((char*)&x)[1]; //高地址单元
    printf("x0 = %x  x1 = %x\n", x0, x1);
    return 0;
}
```

输入如下

```
root@LAPTOP-HRJHHKLT:/mnt/d/mywork/compiler_lab/hw9# ./temp
x0 = 22  x1 = 11
```

由此可知x的低位放在了低地址单元，高位放在了高地址单元，故为**小尾端模式**

- 在`short.c`中，变量`short i`取的是`long .c`中同名变量`long i`低位的两个字节（65536的16进制表示为0x10000），故实际`i`的值为`0x0000`（共2字节），即为0

3

3. 教材11.14 下面左右两边分别是两个C程序文件file1.c和file2.c的内容，用命令cc file1.c file2.c对这两个文件进行编译和连接。请回答：

(a) 编译器是否会报错？若你认为会，则说明理由。

(b) 若编译器不报错，连接器是否会报错？若你认为会，则说明理由。

(c) 若上面2步都不报错，则运行时是否会报错？若你认为会，则说明理由。

(d) 若上面3步都不报错，则运行输出的结果是什么？说明理由。

```
1.    char k = 2;            |        #include <stdio.h>
2.    char j = 1;            |        extern short k;
3.                           |        main(){
4.                           |           printf("%d\n", k);
5.                           |        }
```

1. 编译器不报错。两个文件是分开独立编译的，编译器不会发现其中的类型错误
2. 连接器不会报错。可重定位代码中没有变量的类型信息
3. 运行时不会报错
4. 输出的结果为258

- 有第二大问可知，本地机器是小尾端模式
- 在`file1.c`中，char 类型的变量占1个字节，k 对应的1字节内存为`0000 0010`（二进制表示，以下同理）
- j 紧接着 i 存放，对应内存`0000 0001`，且 j 对应的地址比k要高
- 在`file2.c`中，short 类型的k占两个字节，再根据小尾端模式可知，对应的k为`0000 0001 0000 0010`，即为258（$2^8+2=258$）