1. 两个C文件link1.c和link2.c的内容分别如下

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

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

link2.c: In function ‘main’:
link2.c:4:2: warning: implicit declaration of function ‘printf’ [-Wimplicit-
function-declaration]
    printf("%d\n", *buf);
    ^~~~~~
link2.c:4:2: warning: incompatible implicit declaration of built-in function
‘printf’
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 :使编译器不在系统缺省的头文档目录里面找头文档,一般和-I联合使用,明确限定头文档的位置
- ```
  gcc选项
  -c：只激活预处理,编译,和汇编,也就是只把程序做成obj文档
  -S：只激活预处理和编译，就是指把文档编译成为汇编代码
  -E：只激活预处理,这个不生成文档,您需要把他重定向到一个输出文档里面
  ```
- 执行`gcc -E -nostdinc link2.c`，报出错误

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

  extern int *buf;
  int main() { printf("%d\n", *buf); }
  ```

  可见错误是在预处理阶段报出的。通过执行`gcc -E link2.c > 1.txt`查看预处理文件，发现这是因为预处理器有文件包含功能，会把源程序文件中的包含声明(如`#include<stdio.h>`)展开为程序正文。当用`-nostdinc`显式拒绝在系统缺省的头文档目录里面找头文档时，系统找不到此文件，所以出错。

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。

- 用`ls -l filename`查看文件大小，发现`link-s`为 871696 Byte，`link`为 16760 Byte
- 静态链接：当链接程序时，需要使用的每个库函数的一份拷贝被加入到可执行文件中。静态链接使用静态库进行链接，生成的程序包含程序运行所需要的全部库，可以直接运行。
  ```

动态链接：可执行文件只包含了文件名，让载入器在运行时能够寻找程序所需要的函数库。动态链接使用动态链接库进行链接，生成的程序在执行的时候需要加载所需的动态库才能运行。动态链接生成的程序体积较小，但是必须依赖所需的动态库，否则无法执行。

- 由上所述，并用生成的反汇编码以及符号信息作比较发现，区别在于：
  - 在动态链接里，printf只是以一个形式上的标记

    反汇编码里的调用只是

    ```
    1054:   f2 ff 25 75 2f 00 00    bnd jmpq *0x2f75(%rip)        # 3fd0
    <printf@GLIBC_2.2.5>
    ```

    而符号表里也只是记录了`U printf@@GLIBC_2.2.5`

    并未把printf的具体代码放入文件。
  - 但是在静态链接生成的符号信息里，发现包含了大量诸如`W _IO_fprintf`的符号

    相应在汇编码里，也是有各种带printf的调用

    ```
     402e6b:    e8 f0 dd 00 00          callq  410c60 <___asprintf>

     402e8c:    e8 1f 4f 01 00          callq  417db0 <__fxprintf>
    ```

    这正是因为库函数printf的代码被拷贝到了可执行文件里。

    所以相应的，link-s的反汇编码以及符号信息比link多很多。
- 各自的好处
  - 静态链接
    - 具备了所有执行程序所需要的内容，代码装载速度快，执行速度比动态链接库快
    - 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
    - 运行时函数库不再有联系，便于移植。在动态链接里，链接器把库文件名或路径名植入可执行文件中。所以函数库的路径不能随意移动。否则，当程序调用该函数库的函数时，就会在运行时导致失败。当在一台机器上编译完程序后，把它拿到另一台不同的机器上运行时，也可能出现这种情况。而静态链接则没有这种担忧。
  - 动态链接
    - 动态链接的优点是可执行文件的体积可以非常小，能节省磁盘空间和虚拟内存，因为函数库只有在需要时才被映射到进程中
    - 链接-编辑阶段的时间更短（因为链接器的有些工作被推迟到载入时）
    - 动态链接把程序员自己写的程序与特定的函数库版本分离开，以应用程序二进制接口（Application Binary Interface，ABI）的思想来为程序提供接口，该接口保持稳定，不随时间和操作系统的后续版本发生变化。这样程序可以调用接口所承诺的服务，而不必担心这些功能是怎样提供的或者它们的底层实现是否改变。
    - 所有动态链接到某个特定函数库的可执行文件在运行时共享该函数库的一个单独拷贝。操作系统内核保证映射到内存中的函数库可以被所有使用它们的进程共享。这提供了更好的I/O和交换空间利用率，节省了物理内存，从而提高了系统的整体性能。相反，静态链接的可执行文件，每个文件都将拥有一份函数库的拷贝，非常浪费。
    - 动态链接使得函数库的版本升级更为容易。新的函数库可以随时发布，只要安装到系统中，旧的程序就能够自动获得新版本函数库的优点而无需重新链接(DLL文件与EXE文件独立，只要输出接口不变，更换DLL文件是允许的)。动态链接允许用户在运行时选择需要执行的函数库。这就使为了提高速度 或 提高内存使用效率 或 包含额外的调试信息而创建新版本的函数库是完全可能的，用户可以根据自己的喜好，在程序执行时用一个库文件取代另一个库文件。

相反，静态链接里对程序的更新、部署、发布会有一些麻烦。

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/ccccOpRN.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

- -nostdlib:不连接系统标准启动文件和标准库文件，只把指定的文件传递给连接器。
- _ start是程序的默认进入点，该符号的地址是程序开始时跳转到的地址。其对应的反汇编码如下。查阅资料发现，实际上main函数只是用户代码的入口,它会由系统库去调用,在main函数之前,系统库会做一些初始化工作,比如分配全局变量的内存,初始化堆、线程等，这些是从_ start处开始执行的。用户可以自己实现_ start函数。通常，_start函数由crt0文件提供，后者包含C运行时环境的启动代码。

```
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
```

- 执行命令`gcc -c -nostdlib link2.c link1.c`成功生成两个.o文件，说明错误是在链接阶段发生的。
  - 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
  - 而-nostdlib指定不连接系统标准启动文件，包括crt0，所以会出错。
  - 同时由于没有连接动态库，所以出现错误：对‘printf’未定义的引用

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。

- 32位(树莓派)和64位(本地x86机器)下运行结果都是："Segmentation fault"
- 64位下观察两个文件各自的汇编码发现

```
;link1.s
    .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
    .section    .note.gnu.property,"a"
```

```
;link2.s
movq    buf(%rip), %rax
movl    (%rax), %eax
```

因为每个文件单独编译的原因，所以在链接之前并不会知道彼此源文件里buf的类型。而链接时不会检查名字类型，所以仍能连接成目标程序，仅仅是让不同文件中同一名字的地址相同，如上面汇编码所示，buf在link2.s里只是一个标记。考虑到link1.c里声明的是一个数组，所以link1.s汇编码里把buf作为该数组内容的起始地址标记。然而在执行link2.c中的`*buf`时，会直接取buf+%rip的地址里的内容(即100)存入%rax,然后再把100对应的地址里的内容存入%eax。但是100这个地址不在程序数据区内，是内核态才能访问的，所以会报段错误。

- 32位下观察

```
;link1.s
    .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
    ldr r3, .L3;取得buf的地址
    ldr r3, [r3];取得buf地址处内容100
    ldr r3, [r3];试图取得地址100处的内容
    ......
.L3:
    .word   buf;对应的是link1.s里buf的地址位置
    .word   .LC0
    .size   main, .-main
    .ident  "GCC: (Raspbian 8.3.0-6+rpi1) 8.3.0"
    .section    .note.GNU-stack,"",%progbits
```

可以发现，汇编码虽指令形式不同，但本质是一样的。link1.s里buf作为数组起始地址标记。而link2.s里则把buf视为一个int *，先去buf所在地址取出100作为目标地址，再去访问100地址空间取数，所以出现段错误。

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;
```

```
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？

- 编译是以.c文件为单位的，不会发现文件之间的类型错误。由于数据类型信息未附加在目标文件中，所以连接时不会发现两个文件里变量i的类型不一致，从而可以得到可执行目标程序。
- 运行时不会报错。但输出结果不是65536，而是0。这是由于x86机器是小尾端模式，对于整型数据，是低地址放整数低位，高地址放整数高位。所以short.c里short i取的是long.c里long i的两个低位字节，它们都是0，因此输出结果为0。

  本地运行结果

```
root@LAPTOP-7ME28M81:~/compiler# ./1
0
```

- 在本地机器编译得到部分汇编码如下，可以发现i仍是作为一个地址标记在两个文件间交互的，与数据类型、数据大小无关，short.c里汇编码已经默认把i当做一个short型变量的地址来实现。

```
;short.c
movzwl  i(%rip), %eax
```

```
;long.c
    .globl  i
    .data
    .align 8
    .type   i, @object
    .size   i, 8
i:
    .quad   65536
    .ident  "GCC: (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0"
    .section    .note.GNU-stack,"",@progbits
    .section    .note.gnu.property,"a"
```

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：

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

(a) 编译器是否会报错？若你认为会，则说明理由。

- 不会。因为两个文件是分别编译的，所以不会发现彼此存在的类型错误。

(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。

- 连接器不会报错，因为可重定位代码里没有变量的类型信息。

(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。

- 运行时不会报错

(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。

- 若如x86，机器特点是低地址放整数低位，高地址放整数高位，则结果是258，因为分配给变量j的字节在高地址处，正好作为变量short k的高位字节，即k为`0000,0001  0000,0010`
- 否则结果是513，即k为`0000,0010  0000,0001`

参考文档

- https://www.jianshu.com/p/dbe848e4ad0d
- gcc编译选项