Part0

Task0

Q：运行 make 观察到的现象。
A：两行输出
echo hello world
hello world

Q：运行 make clean && make 观察到的现象。
A：四行输出
rm -rf /mnt/c/Users/10848/Desktop/makelab/makelab/build
mkdir /mnt/c/Users/10848/Desktop/makelab/makelab/build
echo hello world
hello world

Q：将 clean all 改为 clean all $(OUTPUT_DIR)后，执行 make。
A：会报错，显示文件已存在。输出三行：
mkdir /mnt/c/Users/10848/Desktop/makelab/makelab/build
mkdir: cannot create directory '/mnt/c/Users/10848/Desktop/makelab/makelab/build': File exists
make: *** [Makefile:14: /mnt/c/Users/10848/Desktop/makelab/makelab/build] Error 1

Q：.PHONY 的效果以及 make 的工作原理。
A：（1）.PHONY 表示后面跟的是一个伪文件，类似于一个标签，实际上不会对这个文件进行操作。（2）make 工作原理：make 打开名为 Makefile(or makefile)的文件，然后找到第一个目标作为最终目标；如果该文件不存在或者之后的文件修改时间比该文件更晚，就会执行这些依赖文件；若依赖文件有同样问题，会“递归”地处理，直到顺利完成，或者找不到文件就报错。make 后可以接参数，命令。

Q：此处的 all 和 clean 标记为.PHONY 的必要与否。
A：这里表示 clean all 不代表一个文件名，make 时不会做什么；可以避免文件同名；这里还是有必要的吧，用 all 一口气生成多个文件，显式写出更好（参考前面那个链接的内容）。

Q：为什么会输出两个 hello world，以及如何只生成一个。
A：（1）生成两个是因为 echo 的存在，echo 会打印 echo 这条命令以及要输出内容。
（2）要只生成一个，只需要在 echo 前加@即可。

Task1

Q：在指令前加 - 的输出效果。
A：与上次不同，文件成功输出了 hello world（1 个或 2 个视有没有加@而定）；并且在 error 后有说明：Error 1 (ignored)。

Q：在指令后加 || true 的输出效果。
A：正常输出 hello world；没有显示的 error，只提示 file exists；并且在 mkdir 后多了 || true。

Q：两种解决方法哪种更好。
A：都正常打印了信息，但我觉得第一种更好，因为看着更清楚，显然这里是有一个错的，只是被 ignored 了。

Q：更改.PHONY 前后哪种更好。
A：我觉得更改后好，因为这样可以多次 make，都能输出 hello world；如果不更改，只有第一次 make 能输出 hello world.

Part1

Task2

Q：运行 make PART=1 的效果。
A：会正确打印 mkdir 的路径，但是会显示 file exists；以及并不能像之前执行 make PART=1 && build main 一样输出测试成功的字样；会显示 'main' is up to date. ，即 main 文件已是最新的，无需再次 make 了。

Q：修改 src/main.cpp 后执行 make PART=1 的效果。
A：前几行与上一问题一样，但在提示进入 build 后，会输出
cp /mnt/c/Users/10848/Desktop/makelab/makelab/src/main.cpp main.cpp
g++ -I/mnt/c/Users/10848/Desktop/makelab/makelab/include -c -o main.o main.cpp
（中间还有几行 warning，说 std::cout,std::endl 有问题~）
g++ -o main A.a.o some.a.o B.b.o main.o
rm main.cpp

Q：修改 include/shared.h 后执行 make PART=1 的效果。
A：与 Task2 一开始运行 make PART=1 的效果是一样的。

Q：分析增量编译是怎么作用的，以及如何处理头文件的增量编译。
A：（1）增量编译的作用：如 A.o 依赖于 B.o C.o，若 B.o 依赖的 B.c 发生了更改（而 C.o 依赖的 C.c 没有改变），就只需要重新编译 B.o，然后重新编译 A.o。
（2）如何处理头文件的增量编译：一是删除依赖这个头文件的.o 文件（但是不是太繁琐了），或者用依赖项列表，显式写出头文件的类型。

Task3

Q：注释掉#pragma once 的现象，以及分析 pragma once 的效果。
A：（1）注释掉之后运行会报错，有重复定义的行为，并且标出了是在哪里定义过的。
error: redefinition of ‘std::string MassSTR’
error: redefinition of ‘int LenOfMassSTR()’
（2）pragma once 的效果：告诉系统，不要重复包含，也就不会造成 redefinition 的错误了。

Q：去掉 MassSTR 前面的 static，执行 make clean && make PART=1 的现象；并借助 objdump -Ct build/* 分析 static 的作用。
（1）A：现象即会报错：（在 some.a.o B.b.o main.o 都会提示）
multiple definition of `MassSTR[abi:cxx11]'; A.a.o:(.bss+0x0): first defined here

（2）static 的作用：表明能使用的范围仅针对本文件内，与其他文件无关；多个文件定义同名的 static 变量就不冲突了；但是如果没有，会在符号表内看到，在所有文件里这个变量的信息是一模一样的。

Q：分析 4 种规避链接冲突的方式，比较其差异。
A：（1）可以看到，在 main.cpp 中，int a,b,c,d; 都只是定义，没有初始化（都是弱符号）；（2）a 用__attribute__((common))，利用了 common 块，用于表示不同弱符号所需空间以最大的为准；b 用__attribute__((weak))，声明了 b 是弱符号，如果包含的其他文件有定义就用其他文件的，否则采用自己文件的；c 用 extern 关键字，声明 c 是外部变量；d 直接初始化，是强符号。
（3）差异：a 的方式比较“包容”，可以允许多种类型的定义，再以所需最大空间来分配，很好储存了弱符号；b 的方式比较“保险”，先看其他文件，在看自己；c 的方式，表明了这个文件是与其他文件有依赖关系的，单拎出来可能是有问题的；d 的方式比较“霸道”，如果同时存在了两个强符号就报错了。

Task4

Q：删去 LenOfMassSTR()前的 static，观察效果。
A：报错：multiple definition of `LenOfMassSTR()'; A.a.o:A.a.cpp:(.text+0x0): first defined here

Q：借助符号表分析 inline 避免连接冲突的原因；若函数为 static inline 会怎么样？
A：(1)inline 能将函数当作弱符号来处理，全局保存一份，“就地替换”。
(2)static inline 的函数与 static 单独修饰一样，只能在该文件内被使用，但因为有 inline 执行更快。

Q：针对小型工具函数，以上提及的哪种方式更优。
A：因为其短小，展开不会造成太大后果，可以考虑先加上 inline，提高效率（OOP 是这么教的）；再根据我们的需求，若这个函数是公用的工具函数，可不加 static；若这个是这个文件私有的（like 类内的私有函数），那我们就要加上 static

PART2

Task5

Q：为什么不冲突，执行哪个版本的 A()函数，并讨论利弊。
A：这里执行输出“我不是 A 哒”。不冲突的原因是在链接时，根据先后顺序查找，先在 notA 里找到了 A 这个符号，那之后就不会再去寻找了。好处是能避免一些错误，坏处是可能达到意想不到（甚至相反）的效果，如这里的输出。

Task6

Q：调换顺序，看效果。
A：(1)改变 main.0 libB.a libA.a 的顺序，会报错 undefined reference to `A()' / reference to `B()';
(2)改变 A.a.o some.a.o B.b.o main.o 的顺序，并不会报错。

Q：分析链接对象的顺序对链接的影响及其原因。
A：如果是有包含.a 文件要注意依赖顺序，而 gcc 会将全部.o 文件加入到链接过程。

PART3

Task7

Q：直接在根目录 build/main 会发生什么：

A：会显示没有这个路径；可以发现会显示需要 libstdc++.so.6 libgcc_s.so.1 libc.so.，而这在根目录是没有的。

Q：更改后如何运行，系统如何查找动态链接库。

A：(1)可临时使用 export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:. 将当前目录加到动态链接库查找的目录中去。

(2)首先查看程序的.dynamic 段是否包含一个叫 DT_RPATH 的项；查找是否存在环境变量 LD_LIBRARY_PATH；查看库高速缓存文件/etc/ld.so.conf；默认路径/lib 和/usr/lib。

Task8

Q：动态链接库对同名函数的处理 && 改变链接顺序的影响。

A：和 Task6 相同，这里对于同名函数先找到谁就用谁，之后不会再寻找同名符号；改变链接顺序仍会报错，undefined reference。

Q：-fPIC 的作用探究。

A：没有这行代码，之后 make clean && make PART=3 会提示报错，final link failed: bad value。查阅后可以知道，-fPIC 的作用是告诉编译器产生位置无关代码，让代码可以被加载到内存的任意位置，这是共享库需要的。