Make Lab

本次Lab对应CSAPP中的链接单元，将一步步教大家如何使用Makefile构建一个稍大型的项目，并熟悉编译和链接过程中的一些核心概念~

makelab没有什么需要写的代码，也不像前几个lab一样烧脑，但由于这是第一个与现实操作系统充分接轨的lab，可能需要大量的搜索和学习。lab中所有的操作步骤都会在文档中给出，你只需要按照步骤就可以完成实验啦。

实验报告中需要记录你的操作和观察到的现象，参考我们给出的问题对实验现象作出分析。回答不用每个都太详细，做到清晰准确即可，很多一句话就可以，当然你想详细的话也可以~（没有篇幅加分之类的东西）

Part 0

对于通常的项目结构，根目录下会有三个目录：`include`用于放置头文件，`src`用于放置源代码，`build`用于放置编译结果，这些目录我们已经帮大家建好啦。

> 有些项目还会有：`doc`放置文档，`lib`放置外部依赖库，`scripts`放置脚本，`samples`放置样例……项目结构是根据具体情况灵活设置的，并非硬性规定。

项目根目录一般还会有`Makefile`文件用于构建这个项目，`README.md`文件用于介绍这个项目，`LICENSE`文件用于描述项目著作权相关的法律信息~

我们已经在`Makefile`里为大家准备了一些东西。运行`make`即可输出`hello world`，运行`make clean`即可重置实验，运行`make PART=n`即可进行第`n`部分的实验~

请阅读根目录下的`Makefile`，自行了解这是如何实现的。

> 推荐Makefile教程：Metaprogramming 跟我一起写Makefile 1.0
>
> Makefile是一种基础的构建系统，支持的操作较为单一。在更复杂的项目中，往往会使用一些高阶脚本自动生成Makefile，经典的此类脚本有CMake和Autotools。

Task 0

> 进行本任务前，请先确认项目根目录下存在`build`目录。

运行`make`，记录你观察到的现象。

运行`make clean && make`，记录你观察到的现象。

将`Makefile:4 .PHONY: clean all`改为`.PHONY: clean all $(OUTPUT_DIR)`，再次运行`make`，记录你观察到的现象。

请**根据上面实验的结果**，查阅一些资料，分析`.PHONY`的效果和`make`的工作原理。

我们的`Makefile`中将`all`和`clean`标记为`.PHONY`，你认为这是必须的吗？

在上面的实验中，你可能会看到两个`hello world`。可是我们的代码里只写了一条`echo hello world`，为什么会输出两个`hello world`呢？请你查阅资料，给出让它只输出一个`hello world`的办法。

Task 1

> 进行本任务前，请先确认项目根目录下存在`build`目录，并保持上个任务对于`.PHONY`的更改。

在指令前加上`-`可以让`make`忽略指令中的错误。请将`Makefile:14 mkdir $(OUTPUT_DIR)`改为`-mkdir $(OUTPUT_DIR)`，然后运行`make`看一看效果。

另一种忽略错误的方法是在指令后面加上`|| true`。请将`mkdir`那一行改为`mkdir $(OUTPUT_DIR) || true`，再运行一次`make`，将你观察到的现象与前面的`-mkdir`比较，**分析这两种方法的差异**。你认为哪种方法更好呢？

我们可以看到，在忽略错误之后，将`$(OUTPUT_DIR)`标记为`.PHONY`并不影响`Makefile`的执行，与原先的`Makefile`相比，你认为哪种实现更优呢？

Part 1

现在我们来尝试构建这个项目~

也许你已经学过，对于多文件编译，只要将多个源文件都写到编译命令中即可，如`gcc -o main *.c`。这种做法在每次修改了一个文件后，都需要重新编译所有文件，对于大型项目很不友好。如果每次只要重新编译修改后的部分（称为增量编译），效率就会高很多啦。

一种常见的增量编译方案是先将各个源文件分别编译为目标文件，再将目标文件链接为可执行程序。这样，在修改了一个源文件之后，只要重新编译其对应的目标文件，再与之前已经编译好的其他目标文件链接，即可生成新的可执行程序啦~

Part 1将使用`mk/part1.mk`，其中将`src/main.cpp`、`src/A/A.cpp`、`src/A/some.cpp`和`src/B/B.cpp`组合编译为`build/main`。我们已经为大家写好了这个文件，请自行阅读其中的实现，然后运行`make PART=1 && build/main`看一下效果~

Task 2

> 在进行本任务前，请先确认你已经成功运行了一次`make PART=1`。

请尝试运行`make PART=1`，记录你观察到的现象。

修改一下`src/main.cpp`，再次运行`make PART=1`，记录你观察到的现象。

修改一下`include/shared.h`，再次运行`make PART=1`，记录你观察到的现象。

> 此处修改只是为了作出修改文件的效果。为方便后面实验，请使用换行/空格等无实际意义的修改方式，不要改动文件中的代码。

请**根据上面实验的结果**，结合`mk/part1.mk`的代码和`make`的工作原理，分析我们的增量编译是如何实现的，并探讨如何处理涉及头文件的增量编译。

Task 3

请注释掉`include/shared.h:1 #pragma once`，运行`make clean && make PART=1`，记录你观察到的现象，说明`#pragma once`的效果。

请恢复所做的更改，然后删去`include/shared.h:5 static std::string MassSTR`中的`static`，再运行`make clean && make PART=1`，记录你观察到的现象，借助`objdump -Ct build/*`输出的**符号表信息**，说明`static`的效果。

请恢复所做的更改，然后运行`make PART=1`确保其能正确通过编译。

在上面的实验中，你可能会发现没有`static`标记的同名全局变量会导致链接冲突。但是在`src/main.cpp`和`src/A/A.cpp`中，我们定义了四对同名的"全局变量"`a`、`b`、`c`、`d`，且能正确通过编译。请阅读这四对变量的定义，借助`objdump -Ct build/*`输出的**符号表信息**，查阅一些资料，说明它们能够避免链接冲突的原因，并比较这四种定义方式的差异。

Task 4

与变量类似，同名函数（对于C++，指同函数签名）也会导致链接冲突，且可以通过添加`static`避免。请将`include/shared.h:7 static int LenOfMassSTR()`中的`static`删去，运行`make clean && make PART=1`，观察函数冲突的效果。

另一种避免链接冲突的方法是将函数标记为`inline`。请将前述函数定义改为`inline int LenOfMassSTR()`，运行`make clean && make PART=1`，借助`objdump -Ct build/*`输出的**符号表信息**说明`inline`能够避免链接冲突的原因。你认为这种做法好吗？如果一个函数同时被标记为`static inline`会怎么样？

> 在开启编译优化时，`static` 和 `inline` 的效果可能会有些不同。你可以自行尝试一下，以发现其中存在的未定义行为（Undefined Behavior）。
>
> 开启编译优化的方法是在 `mk/config.mk` 的 `CPPFLAGS` 后面加上 `-O2`。你可以自行了解一下 `Makefile` 中 `CFLAGS`、`CPPFLAGS`、`CXXFLAGS`、`CC`、`CXX` 这些预定义变量的意义和 `%.o` 的默认规则。

对于一些小型的工具函数，如我们的 `LenOfMassSTR`，一般可以直接定义在头文件中。你认为以上提及的定义方式中哪种最优?

> 完成本任务后，请恢复所做的更改。

Part 2

在大型项目中，常常会采用模块化的组织方式。通过将一个模块内编译出的所有目标文件打包为一个库文件，我们可以方便地在模块层面上进行链接，这将比维护一大串的目标文件列表更为简洁，也方便将编译好的库分发出去供其他项目使用。

库文件分为静态链接库和动态链接库两种。静态链接库本质上就是一些目标文件的集合，可以如同目标文件一样参与链接，链接器会自动从中查找所需的目标文件~

Part 2将使用 `mk/part2.mk`，其中将 `src/A/A.cpp`、`src/A/some.cpp` 和 `src/A/notA.cpp` 编译打包为静态链接库 `libA.a`，将 `src/B/B.cpp` 编译打包为静态链接库 `libB.a`。我们已经为大家写好了这个文件，请自行阅读其中的实现，然后运行 `make clean && make PART=2 && build/main` 看一下效果~

Task 5

> 在进行本任务前，请先确认你已经成功运行了一次 `make PART=2`。

也许你已经发现了，`libA.a` 中的 `A.cpp` 和 `notA.cpp` 间存在链接冲突（同名函数 `void A()`），但是他们可以正常编译出静态链接库和可执行程序（虽然运行的结果可能有点怪怪的），这是为什么呢？程序是如何决定执行哪个版本的 `void A()` 的呢？请结合 `objdump -Cat build/libA.a build/main` 输出的**静态链接库头信息和符号表信息**，查阅一些资料，对上述问题作出分析，并讨论这种设计的利弊。

Task 6

请尝试更改 `mk/part2.mk:3 $(OUTPUT): main.o libB.a libA.a` 中各链接对象的顺序（如改为 `libA.a libB.a main.o`），然后运行 `make clean && make PART=2`，记录你观察到的现象。

请尝试更改 `mk/part1.mk:3 $(OUTPUT): A.a.o some.a.o B.b.o main.o` 中各链接对象的顺序，然后运行 `make clean && make PART=1`，记录你观察到的现象。

> 你可以为两个part多尝试几种顺序，也可以改改`mk/part2.mk:6 libA.a: notA.a.o A.a.o some.a.o`中的顺序。

请**根据上面实验的结果**，查阅一些资料，分析链接对象的顺序对链接的影响及其原因。

> 完成本任务后，请恢复所做的更改。

Part 3

恭喜~马上就做完啦！还有一种库文件是动态链接库。与静态链接库不同，动态链接库在程序运行时才被加载，通过动态链接器链接到程序中。

Part 3将使用`mk/part3.mk`，其中将`src/A/A.cpp`和`src/A/some.cpp`编译打包为动态链接库`libA.so`，将`src/B/B.cpp`和`src/A/notA.cpp`编译打包为动态链接库`libB.so`。我们已经为大家写好了这个文件，请自行阅读其中的实现，然后运行`make clean && make PART=3`看一下效果~与之前不同的是，这次你不能直接运行`build/main`，而要先进入`build`目录，然后在其中运行`./main`。

Task 7

> 在进行本任务前，请先确认你已经成功运行了一次`make PART=3`。

如前所述，这次我们编译出的程序需要进入`build`目录再执行。请尝试一下直接在项目根目录下执行`build/main`会发生什么，并结合`objdump -p build/main`输出的**动态链接库信息**分析原因。

请将`mk/part3.mk:6 $(CXX) -o $@ $(addprefix ./, $^)`改为`$(CXX) -o $@ $^`，然后运行`make clean && make PART=3`编译程序，观察`objdump -p build/main`输出的动态链接库信息。现在应该如何运行这个程序？请查阅一些资料，说明系统是如何查找动态链接库的，并设法在不重新编译的情况下让我们的程序运行起来。

> 完成本任务后，请恢复所做的更改。（尤其是，如果你对系统目录做了一些更改，记得改回去）

Task 8

> 在进行本任务前，请先确认你已经成功运行了一次`make PART=3`。

与Part 2类似，我们在Part 3中也制造了一次链接冲突（`libA.so`中的`A.cpp`和`libB.so`中的`notA.cpp`）。但是程序可以编译执行（虽然输出的结果可能还是有点怪怪的），这又是为什么呢？程序又是如何决定执行哪个版本的`void A()`的呢？请结合`objdump -Ct build/main`输出的**符号表信息**，查阅一些资料，对上述问题作出分析。

> 你可以自行了解一下动态链接器的工作原理。

请尝试更改`mk/part3.mk:5 $(OUTPUT): main.o libB.so libA.so`中各链接对象的顺序，然后运行`make clean && make PART=3`，记录你观察到的现象。Task 6中讨论的规律是否对动态链接库也适用?

请恢复所做的更改，然后注释掉`mk/part3.mk:3 CPPFLAGS += -fPIC`（`Makefile`中的注释是在行的开头加一个`#`），运行`make clean && make PART=3`，记录你观察到的现象，查阅一些资料，说明`-fPIC`的作用。

- THE END -

恭喜你做完啦！剩下的时间一定要好好做PJ哦ヾ(≧▽≦*)o