解题报告

设计思路：

为了实现面向对象和工厂模式，首先将$pizza$和$plate$视作两个对象，构建$pizza_factory$，将$plate$的制作、$pizza$的制作以及随后的工序在工厂中进行包装完成。因而，在重构代码过程中，我构建了三个头文件`pizza.h` `plate.h`以及`pizza_factory.h`，对main进行分解。

`plate.h`中保留大部分原来代码的设计，没有做太大的改动。

`pizza.h`中主要支持$pizza$的$cook$、cut、 eat、 $put\ on\ plate$ 等操作，另外为了记录每块pizza所在的plate，需要用一个plate的私有成员来记录。但是为了实现工厂模式，特别是对于多种不同类型的pizza的多态支持，我们有两种选择：一是使用template模版方式，但是这个方式不如if-else语句的可扩展性强，因为每一个pizza并不是仅仅名称不一样，其ketchup和cheese属性都是不一样的；因而我们采用继承的方式，保留pizza不作为纯虚类，为的是支持main.cpp中临时创建新的pizza，各种类型的pizza均继承自pizza类，他们依靠构造函数重载实现。另外，因为eat涉及pizza的私有成员，因而将源代码的私有成员改写成保护成员，以免访问error。

`pizza_factory.h`中，为了实现pizza的流水线生产，我们需要有两个成员：`std::vector<Plate* > plates`

与`std::vector<Pizza* > ordered_pizzas`。它们分别记录当前的工序。另外，find_empty_plate作为私有函数，因为我们不希望它在main.cpp中被调用导致factory的私有成员被间接访问。`make_pizza`函数与`make_new_plate`作用都是将pizza和plate的制作封装进工厂。为了防止出现盘子不够的情况，在`make_pizza`的时候特判是否能找到盘子（虽然有点丑）。`del`函数的目的是在外界pizza被eat了之后，在工厂内能够更新工厂内`ordered_pizzas`s的信息。为了实现标准输出，在pizza类和plate的类中析构函数都设计了相应的输出，同时pizza_factory的析构函数顺序设计为先析构pizza再析构plate。

总结架构：pizza为父类，派生出六种具体的pizza子类；plate为一个类；pizza_factory使用"has-a"的设计方式包括pizza和plate，最后主要的生产操作可以在main.cpp中通过操作factory来实现。

Your appreciation is my biggest motivation.