设计模式：创建型模式

1. 单例模式

举例：设计一个计数器类，将函数实现为静态(static)方法

```
class Counter {
    private:
        static int count = 0;
    public:
        static void addCount() {
            count += 1;
        }
        static int getCount() {
            return count;
        }
};

Counter::addCount();
cout << Counter::getCount() << endl;
```

如果我们有不同的Counter，想在基类中把addCount()变成虚函数，这样可以吗？其实是不可以的。因为static函数一定不能是虚的！这是因为**静态方法只与类相关而不与实例相关，调用时必须知道类名。**

所谓单例，就是只能构造一份实例的类：

```
class Counter {
    Counter(const Counter &) = delete;  // 显式删除拷贝构造函数
    void operator =(const Counter &) = delete; //显式删除赋值符
    int count;
    Counter() { count = 0; } //私有化构造函数
    static Counter _instance; // 全局唯一的实例，私有静态
public:
    static Counter &instance() { //返回唯一的那个实例，公有静态
        return _instance;
    }
    void addCount() { count += 1; }
    int getCount() { return count; }
};
```

调用单例：

```
Counter Counter::_instance; // 定义类中的静态成员，单例在此被初始化
int main() {
    Counter &c = Counter::instance(); // 由于删去了拷贝构造函数，必须存为引用
    c.addCount();
    cout << c.getCount() << endl;
}
```

单例模式的陷阱：

```
Counter &c = Counter::instance();
delete &c;    // 可以成功执行!
c.addCount(); // 运行时错误
```

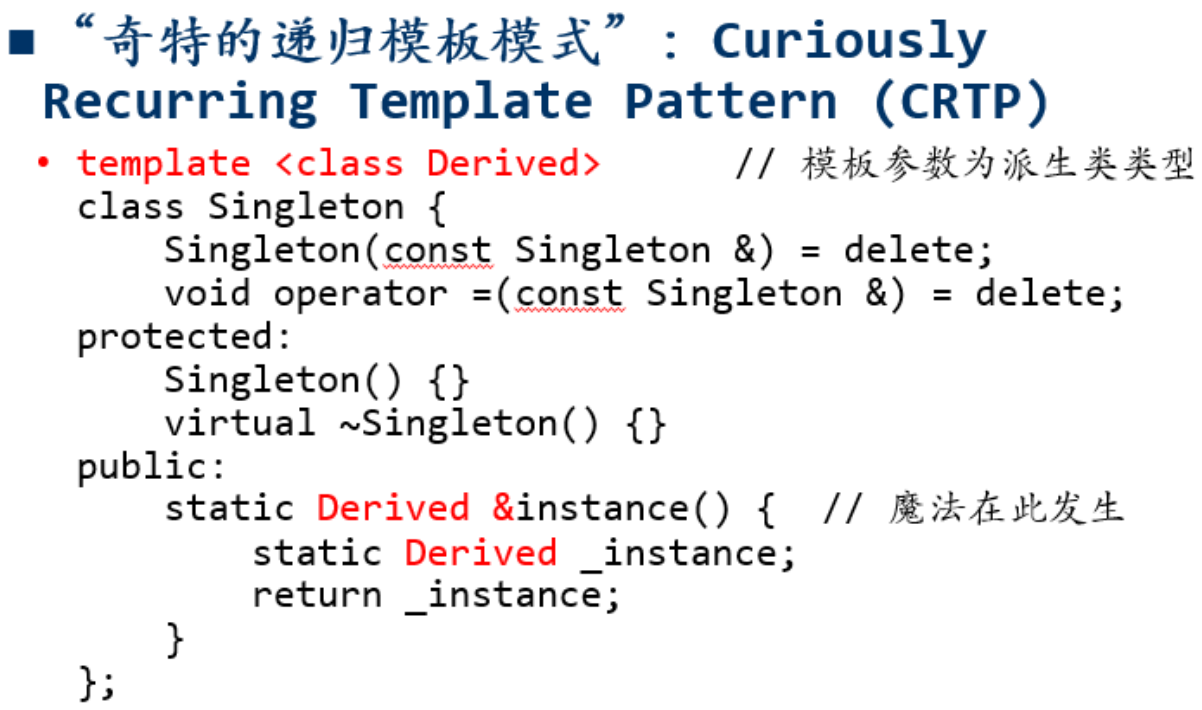
故应当把析构函数也设为private。

惰性初始化：能否在使用时再构造单例实例？

```
class Counter {
    // ...
 public:
    static Counter &instance() {
        static Counter _instance;
        return _instance;
    }
    // ...
 };
```

在第一次调用instance方法时才会构造单例。

如何让基类知道派生类的类别呢？可以使用"奇特的递归模板模式"：



CRTP是实现多态的另一种方式。不过与虚函数不同，本质上实现的还是编译期多态。

2. 工厂模式

在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用一个共同的接口来指向新创建的对象。

2.1 包装复杂的构造逻辑

当构造逻辑过于复杂，或者有必要进行分离时，可以把工厂方法放在单独的类中：

```
class TeaFactory {
 public:
     void setMilk(int amount) { ... }
     void setSugar(int amount) { ... }
     Tea *createTea(string type) { //在工厂中创建
         Tea *tea = nullptr;
         if (type == “GreenTea”)
             tea = new GreenTea;
         else if (type == “BlackTea”)
             tea = new Blacktea;
         if (milkAmount > 0) tea->addMilk(...);
         if (sugarAmount > 0) tea->addSugar(...);
         // ...
     }
 };
```

2.2 为重载的构造函数提供描述性名称

一些类可能具有多个重载的构造函数，可以改写为工厂方法以使用描述性的名称：

```
class Complex {  // 复数类
   float real, imag;  // 实部、虚部
   Complex(float real, float imag) { ... }
 public:
   // 笛卡尔坐标，写成成员函数的形式
   static Complex fromCartesian(float x, float y) {
     return Complex(x, y);
   }
   // 极坐标
   static Complex fromPolar(float r, float a) {
     return Complex(r * cos(a), r * sin(a));
   }
 };
```

2.3 对象构造需要用到当前函数体无法访问的信息；需要集中管理被构造对象的生命周期

例如Box2D中的工厂模式：

构造b2Body必须使用工厂方法，其构造函数为private：

```
b2World *world = new b2World(...);
b2BodyDef bodyDef(...);
b2Body *body = world->createBody(&bodyDef);
```

原因有二：

1.b2Body在构造时需要用到b2World的**私有成员。**

2.Box2D需要频繁申请并释放小块内存，为高效处理，Box2D自己实现了小块内存分配器b2BlockAllocator并手动管理内存。通过工厂方法，可以**避免用户误用系统的内存分配机制。**（工厂统一分配）

3. 抽象工厂模式

工厂方法的目的是构造单个类的对象，如果我们要构造的是多个类的对象该怎么办呢?

抽象工厂模式（Abstract Factory Pattern）是围绕一个**超级工厂**创建其他工厂。该超级工厂又称为其他工厂的工厂。在抽象工厂模式中，接口是负责创建一个相关对象的工厂，不需要显式指定它们的类。每个生成的工厂都能按照工厂模式提供对象。

一个例子：一个编译器分为Lexer和Parser，Generator三个阶段，需要针对C++和Java做判断。写成如下形式不好的地方就是每增加一个语言就要对所有函数都增加一个If。

```
class Compiler {
    string type;
    public:
        Compiler(string type) {this->type = type;}
        LexResult *lex(Code *input) {
            Lexer *lexer;
            if (type == “cpp”) lexer = new CppLexer;
            else if (type == “java”) lexer = new JavaLexer;
            return lexer->lex(input);
        }
        ParseResult *parse(LexResult *input) {
            Parser *parser;
            if (type == “cpp”)  parser = new CppParser;
            else if (type == “java”)  parser = new JavaParser;
            return parser->parse(input);
        };
};
```

设计一个基类，抽象同一语言所需的所有步骤：

```
class AbstractFactory {  //抽象工厂
 public:
     virtual Lexer *createLexer();
     virtual Parser *createParser();
     virtual Generator *createGenerator();
 };

class CppFactory : public AbstractFactory { //实际的CPP工厂
 public:
     Lexer *createLexer() {
         return new CppLexer; }
     Parser *createParser() {
         return new CppParser; }
```

```
        Generator *createGenerator() {
            return new CppGenerator; }
    };

    class JavaFactory : public AbstractFactory ... //实际的Java工厂
```

S