oop期末复习

1. 打开文件 #include<fstream>

#include<fstream> #include<iostream>ofstream fout, ifstream fin;

Ofstream fout;

fout.open(“data.txt”,ios::in)

注意！！cin读完一行后自动换行。

输出使用ofstream类；输入使用ifstream类。

7f314eceead4dc38ae3df60abe5f840打开文件的类型如下：

Ios::in,ios::out,ios::app可以多个联合使用；

ios::in只读，ios::out清除写，ios::app追加写。可以同时使用多个打开类型，只需要用|（或）连接即可。

1. STL容器

序列：

array定长容器，单向链表forward\_list，list表示双向列表，deque双向队列（先进先出）

需要对映引用#include<array>、#include<forward\_list>、#include<deque>

#Include<list>

联合（可能默认是有顺序的）：set.insert(),map[a]=b,

set（元素含有只能唯一），map（键值对，一个键对应于一个值），multiset（保证序列有序且可以含有重复元素），multimap（按照关键字排序，不能直接修改容器中的关键字，key值可以重复）

multimap的使用---如何遍历重复key值不同的对映value：

https://blog.csdn.net/itlilyer/article/details/120703723

需要引用#include<set>，#include<map>

无顺序的联合：

Unordered\_set，unordered\_map，unordered\_multiset，unordered\_multimap，

需要应用库函数同上

接配器：

stack（栈），queue（队列，线性结构，纯先进先出），priority\_queue（优先队列，也可以称为“堆”，即二叉树，最大最小为root）

Top(),.pop(),.push()

1. Vector

Auto x = aaa.begin(),x!=aaa.end(),x++

最常用的vector函数为push\_back()，注意迭代器的使用，使用auto指针接收；

注意front()与end()是元素不是对映的迭代器；

.front()函数返回第一个元素；.back()返回最后一个元素；.at()基本功能同[]重载，但是更为安全（有越界检测机制）。

其中可以用begin与end获得对映的迭代器，区别于cbegin与cend函数，begin与end（非const类型，可以被修改）函数返回迭代器可以被修改，后者返回的为const类型。

.size()返回当前所含有的元素个数，.capacity()返回容器可以容纳的大小--->注意vector的两倍扩展原则;.empty()判断是否为空；reverse()重新设置大小；

.empty()判断是否为空，.size()获得容器当前含有元素的个数，.capacity()返回容器总容积大小，.reserve()（可能是重排函数，reserve（a.begin(),b.end()）以迭代器为参数）重新分配容器的总容量大小，不改变size函数的返回值，并未改变分配的内存空间；.resize()改变了已分配的内存空间。

Resize使其size变小时，仅改变size函数的大小，且容器会被截断，而不会改变capacity的大小；若使其size变大，则size与capacity均变化为改变值，容器新设置位会被置0；

.clear()清空容器，只是改变了size，对映的capacity没有改变；.insert(position，number(可以没有)往往在set的插入中使用)；data(可以是一段范围的插入))在指定位置插入元素；.erase(first\_position,last\_position)容器删除指定位置的元素或范围内元素，位置参数输入需要一个对映的迭代器，会使当前迭代器失效，并返回下一个迭代器的位置参数。注意，erase()函数的操作对象是对映的迭代器，其中，删除当前迭代器并指向下一个迭代器；

Size--->实际占用的内存，capacity--->分配的空间

1. map

map的key值访问可以使用[]重载，只有key value对映才有效；

[]为键，等号后内容为值；

.count函数check key对映的value是否存在，

.contains函数check对映的key是否存在；

.count(key值)函数，若key值对映的value存在，则返回1；

.contains(key值)函数，查询对应的key值是否存在，存在即返回1。

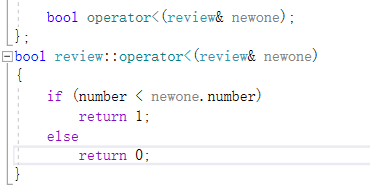
1. 使用typedef简化过长的变量名；

Typedef n name；

1. 引用/结构体可以使用 . 来访问class中属性，指针需要用 -> 来访问。

指针使用->，结构体/类等使用. ；

1. 运算符的重载格式如下：

注意返回值，类归属，operator以及符号的位置与使用。

返回值不同不能当作重载；注意注意，；class默认为private，因而public需要显示的给出；

1. stack存放local变量；heap存放动态变量与内存申请的变量；其余存放代码数据，如全局变量（注意使用extern，需要注明类型与名称），静态变量（只在当前文件中可以使用），静态局部变量仅在某一函数中可见。

Extern

只需要同名include .h1文件即可，在.cpp文件的定义中，需要包含需要的.h文件与库文件，对于模板，只能在.h文件中定义；对于namespace，需要在.cpp函数的实现前，加上namespace的限定；保守一点，在实现namespace的函数定义时，显示使用std限定；

1. New与delete需要一一对应，new数组指针需要用delete[]删除（需要删除整个内存分配序列，而不只是序列起始处），delete空指针是安全的（注意注意，delete空指针是可以的，好像对空指针用了两次delete也没有产生报错QAQ），但同一个指针不能被delete两次。
2. 引用定义时必须被初始化（只是一个重命名）。不是新建立一个对象，而是为已存在变量取一个别名。

需要注意引用的权限，及类型的转变不能放大；一经绑定，变量绑定对象不能更改（不能指向其他变量）。

绑定对象一定要为一确定数值（变量即可？不为数字，表达式也不行）。

可以对指针进行引用，如int\*& a，对指针a进行引用（引用指针）。

普通引用不能接收函数的返回值（函数返回值是临时变化，会直接被释放，引用直接失效）。

1. const的使用

const int \*a，int const \*a等价，无法通过\*a来修改a指向的变量的数值，但可以修改a的指向。Int\* const a，不能改变a的指向，但可以改变a指向的数值。

在使用指针时也需注意，int类型的地址不能赋予const int的指针对象（两者为不同的类型）。

若函数返回值为const int时，可以使用int变量进行接收。

1. Class

在类函数函数名（以及参数列表后） + const表示函数体捏不可以修改class的成员（成员数据）；在类函数声明和定义处均需要加上const说明。

Public:，private:关键词；

注意（::a）为一种全局作用域符号，可以在类中使用同名的全局变量；函数的使用同理，不用全局作用域符号便于定义类内递归函数。

1. 多文件引用

Class、函数头声明/定义在.h文件（还有全局变量的迁移声明），在每个使用其的.cpp文件中都需要进行对映.h文件的引入。

注意头文件重复定义的保护措施

#program once

#ifndef XXXX #define XXXX #endif

#ifndef xxxx #define xxx #endif

一般而言，一个头文件仅定义一个class类，且.h文件的命名与.cpp文件对映（相等）。

1. Encapsulation 封装、abstraction抽象、modularization模块化（的思想）

属性名（属性值），......{函数体}；

1. 构造函数与析构函数

构造函数与类同名，不需要添加返回值信息，其中存在list初始化手段。

如review(int a,string b):number(a),name(b){}，其中使用“：”来介导列表初始化赋值，其中()前表示类属性，如number、name等，()中内容表示初始化的具体数值，如a，b，列表初始化结果即为number = a，name = b

初始化列表的实际赋值顺序与类属性定义顺序一致，与列表陈列顺序无关（特别注意）。使用初始化构造可以减少一次默认构造与赋值。

定义了一个构造函数或析构函数后，默认的构造函数与析构函数将不在存在，因而可以使用 类名（）=default的方法，使默认构造/析构函数依然存在；--->告诉编译器，这是一个默认生成的函数版本；注意参数列表不要一致！！！

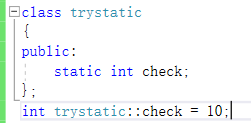
析构函数，函数名为 ~+类名，一般在对象释放时会自发调用。

析构函数没有任何参数，且一般需要额外处理new分配的内存（析构函数不会主动delete new分配的属性占用空间）

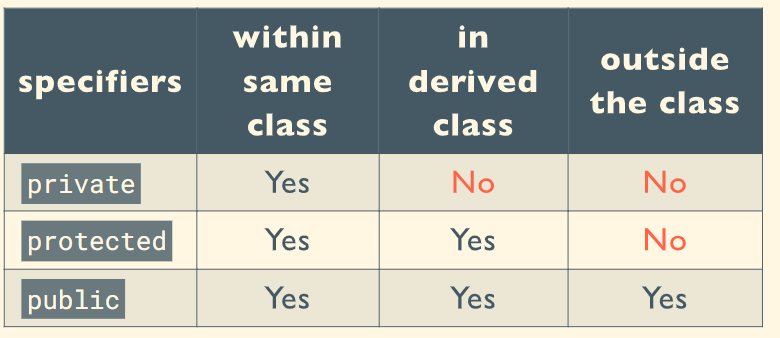
构造函数与析构函数存在默认生成机制。

1. This指针，类函数的隐藏参数。
2. 类中const属性可以在构造函数的初始化列表中被赋值（初始化），也可以直接在声明时初始化（合理的）；对于static const属性而言，每个类只有一个对应的参数。对于类内static变量而言，类中只是声明，需要在类外定义具体的数值（所有class对象共用一个）。

特别注意，类中static变量需要类外初始化；需要类名的限定；所有类实例共用；

1. Inline关键字：对于inline关键字声明的函数（直接在函数头返回值前添加即可），在编译时编译器会把函数对应的代码贴到程序执行的对应位置，可以在一定程度上减少时间消耗。一般只适用于较短的函数（2~3行）。
2. 特别注意，在class声明{}中定义的类函数，默认为inline类型（qqq）。
3. Inheritance继承，composition嵌套（类定义中嵌套其他类元素）。通常把嵌入的对象作为private对象。

在继承关系中，子类（derived class）继承父类（base class）的成员，函数与接口；继承存有多种方式（表示函数/成员可以的使用范围）；

Class A::public B{}子类A继承父类B；

往往是基（父）类最先创立，再创立子类；在析构函数的调用时则刚好相反。

子类使用基类的函数时，需要限定基类的函数作用域。

注意name hiding的问题，如果在子类中重新定义了基类中的同名函数，则所有同名的基类函数不可再被访问。

基类指针可以指向new派生类指向的地址，仅当多态函数被声明为virtual时，才可以通过基类指针或引用使用派生类多态函数。

对于基类函数的重写，函数名与参数列表（特别注意）均需要一样，返回值可以不一样（但返回值之间需要可以协变，如基类引用/指针等）。

1. Struct默认使用public关键词，class默认使用private关键词。
2. Friend友元函数，在类定义中用friend + 函数声明定义，一般接受类引用为参数，只需要像一般函数一样实现即可；其中友元函数可以使用类的private变量。
3. 继承之间的关系：如果B继承了A，在A可以使用的地方B也可以使用。在继承关系中，基类的析构函数一定要用关键词virtual声明。
4. 将派生类视为基类的对象，该对象仅在引用或指针上有效。派生类可以截断转变为基类（向上构型）。如果是直接派生类赋值给基类（类型本身/引用/指针同理），基类只能显示调用自己对映的成员（“重写函数”可以按照派生类函数的执行惯例）；除非同名函数使用virtual并重写（注意两个条件都要满足），不然调用基类的函数。

非virtual函数，不管引用/指针赋值，还是直接赋值，只调用自己当前类的函数；

1. 在虚函数的使用中，可以在函数名称声明后，加上final关键词，表示该虚函数不可再被重写；也可以使用override关键词，表示检查该函数是否重写了一个虚函数（override关键词添加在重写函数之后），若没有则报错；如果直接添加“=0”，则表示该函数为纯虚函数（派生类必须重写，对应于抽象类）；对于虚函数而言派生类可以不重写。
2. Polymorphism多态性，ellipse椭圆，interface接口；派生类可以作为基类返回。
3. 静态绑定（static binding）和动态绑定（dynamic binding）；静态绑定指编译期就直接关联了函数调用的地址。动态绑定只有当运行期时才能基于对象类型进行函数内存地址的绑定。C++使用虚函数和函数重写时才会发生动态绑定。
4. 虚函数有对映的指针指向虚函数表，如果对于virtual关键词定义的函数在派生类中没有重写，则派生类的虚函数表会记录基类中对映的同名函数虚函数地址；若重写，则会指向重写后的函数。
5. 如果只是派生类对象对基类对象进行赋值，基类对象的虚函数表不会发生改变，且派生类会被截断再填入基类对象中。（即虚函数的调用仅关注于原始对象的类型（非指针与引用））。
6. 对于已经重写的基类虚函数，我们仍然可以通过加上作用域的方式去调用它。
7. 特别注意，若虚函数发生了重载，则子类在实现时，需要重写所有重载的虚函数。若重写，则所有虚函数可用；若只重写了一部分重载的虚函数，则其余未重写的虚函数被hidden。可以用using 类::函数的形式，强行声明可以使用指定类的虚函数。

重载需要重写所有虚函数，不然未重写的直接被hidden；

特别注意，构造函数不能被声明为virtual虚函数。

1. Cohesion凝聚力；如果一个函数只解决一个简单的问题，则我们称它为是高凝聚性力的。Loose coupling松散耦合性，使不同类/函数之间的耦合性下降。
2. Oop的三大原则：understandable，flexible and maintainable。（易于理解，易于变化（比较灵活），易于维护）。（solid原则）

指针拷贝构造的单独考虑；

1. 拷贝构造函数。当类对象作为函数参数传入时，实参会对传入的值进行隐式的拷贝构造。

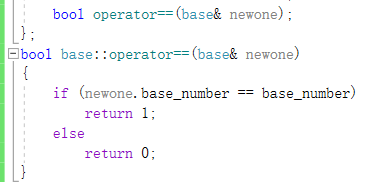
在拷贝构造时，需要特别考虑指针的拷贝，不能直接简单的赋值（会导致不同对象的指针指向同一个区域），需要先使用new获得新的地址空间，再通过使用strcpy等函数进行数值的拷贝，因而在析构函数使用delete时需特别注意。

1. 拷贝构造函数样式同构造函数，没有返回值，函数名同类名。如果声明了拷贝构造函数，则编译器不会再生成默认构造函数，需要用户自己定义。（=default，生成默认的对映函数，=delete依然表示函数已存在，会影响构造函数的默认生成）
2. 拷贝构造函数被调用时间点：初始化赋值时；当一个对象作为实参传入函数时；当一个函数返回类对象时（非引用）。对于一些构造对象返回，编译器会进行对应的优化，从而减少拷贝构造的过程。
3. 若不想使用拷贝构造函数，可以在拷贝构造函数声明后加上“ = delete”的关键词；或把拷贝构造函数放置于private:部分下（可以不定义函数主体）。
4. Static关键词可以用于class对象上，表示对象最多只能被初始化一次。
5. 对于不同类对象析构函数的调用，编译器使用类似于栈（LIFO）策略。
6. Static定义的函数，没有默认的this参数，只能访问static成员，也不能被override。

特别注意第17条；且类中static成员需要在.cpp文件中定义与初始化，初始化时不需要再写static关键字。可以使用“类名::静态变量名”的方式访问类中声明的静态变量；对于类中静态成员函数，则可以使用“类名::静态成员函数名”的方式进行调用。

1. 操作符重载，其中“.”“.\*”“::”，“?:”不可以被重载。

重载”==”操作符函数实例，需要返回值，直接以“operator+操作符”作为函数名。

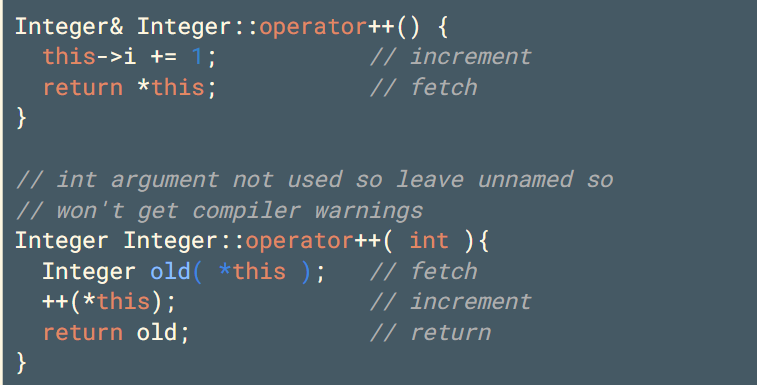


重载的操作符需要存在于一般的算数计算中，比如\*\*表示乘方的操作符便不可以重载。

重载运算符可以作为类内成员函数，也可以在类外单独声明（建议声明为friend，便于调用类内成员），关注函数的参数列表即可。

注意重载“+”操作符如x + y表示x.operator+(y)；重载“-”取负，“!”取反时不需要函数参数（单目操作符）。

特别注意，为了区分前置自增操作符和后置自增操作符；对于前置自增操作符而言，operator++()函数不需要任何参数；而后置自增操作符的函数定义时，需要在operator++(int)函数定义中增加参数int只要类型即可；（先返回后自增），注意！！！只有int占位 没有具体的实参。

前后置自增实例。

在重载==、!=等符号时，可以对函数使用const（函数参数表后），对参数输入也使用const，来确保判断过程不发生意外修改。

重定义operator[]时，只能使用成员函数的方法。

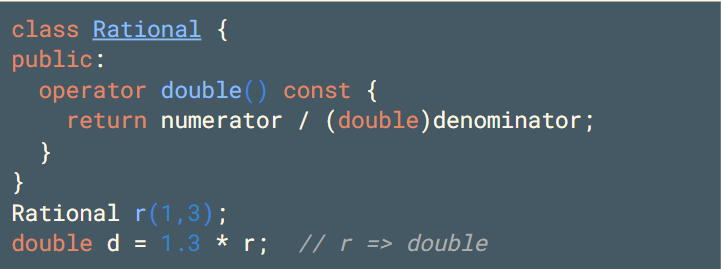
对于=操作符的重载，需要返回当前类型的引用值（注意），便于实现连续赋值操作。如果只是使用类名作为返回值，则在函数返回时，会根据\*this创造一临时变量（与\*this指向同一内存区域），赋值完成后临时变量删除，会导致原有类的内存被释放（可能存在指针时会有明显弊端）。

也可以重载()，使其表现得像一个函数一般。

对于类的隐式转变，可以使用explicit关键字使隐式转变无效，只能显式调用函数进行赋值操作。好像只允许对构造函数使用

可以重载如operator double()此类函数，必要时编译器会隐式调用，没有参数，返回类型无需声明，自动与函数中类型名一致。例如：

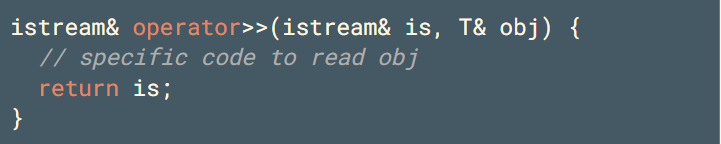
Operator double(){xxxx;}注意没有返回值；



如果想在类外定义函数，则用“类名::operator double()”即可。

隐式调用可能会出现很多问题，因而可以考虑调皮编程必须显示使用（ + explicit关键字）。

1. C++函数调用的匹配顺序
2. 寻找完全匹配的函数；
3. 寻找函数模板，进行实例化；
4. 通过类型转换可产生参数匹配的函数。
5. 对输出输入流进行重载

在参数列表中，需要增加istream/ostream 对象（in/out）；最后需要将对映的输入输出流作为返回值，并指定为引用类型（便于连续操作）。

get()函数，获得下一个字符，若为EOF则没有字符返回。

get(char\* buf, int limit, char delim = ‘n’) 读入字符至buf，最多读取limit个字符或者遇到 ’\n’ 字符；不会读入分隔符。

getline()与get类似，但会处理掉分隔符（如换行符）。

gcount()返回刚刚读入的字符数量。

putback(char c)将字符c塞入stream流中。

peek()获得stream中下一字符，但不将其从stream流中取出。

put(char c)输出一个单独字符。

flush()输出stream中所有的内容。

1. 若想对输出的内容进行格式控制：

需要#include<iomanip>

#include<iomanip><iomanip>

cout<<hex;表示将输出转化为十六进制形式；

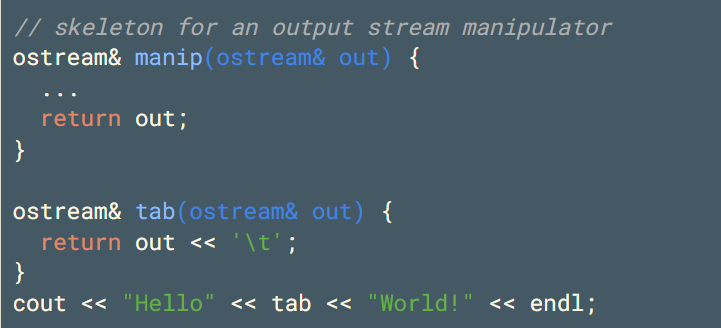
setw(number)只对最近邻输出有效，算上输出内容一共占位number格数；若输出内容占位小于number，则用空格填补；若格数大于number，则全部输出；

setprecision(n)，将输出设置为仅显示n位小数（全局控制，默认输出为6位）；

setfill(c)，将一般填充字符设置为字符c（全局控制）；

setiosflags(ios::fixed)，用固定的浮点显示，其中ios::scientific/ios::left/ios::right表示输出用指数表示/左对齐/右对齐；

可以自己定义一些输出格式的控制符，更具灵活性。

在一般设置格式时使用setiosflags(flags)、对于流输出时，使用setf(flags)设置流输出格式；

常见的flags有：ios::doc/hex/oct对应于十进制，十六进制，八进制；ios::showbase输出不同的进制数时，输出其前缀，如0x；ios::uppercase，输出进制数时使用大写字母；ios::showpos对正数，输出前缀“+”......

1. template模板

template<class T><typename T>

参数的默认输入，直接在参数列表中根值即可；附初值的参数必须从右至左依次增加。

在模板定义中template<class T>与template<typename T>完全等价；

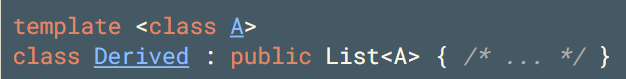
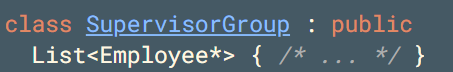
Instantiation实例化，模板实例化时，参数类型的隐式转换也被忽略（限制）；

在使用模板函数时，也可以显示的提供需求类型“函数名<需求变量类型>”即可；

模板可以嵌套，在模板定义中可以定义常数：

3d973a8c6e38f670afb761c8497560f也可以使用“ int bounds”，相当于限制部分模板的类型需求（non-type parameters类型参数）；

继承含模板类也需要用模板，也可以直接继承实例化的模板类；



注意，使用模板实现的函数需要放在头文件中（实现）；

奇异递归模板，基类为模板类，使用派生类的类型进行模板实例化；这样做可以让基类访问的派生类的成员，可以理解为基类变成了派生类本身。

1. 迭代器

一般容器，可以使用begin()，end()来访问对映的迭代器，而对于迭代器而言只需要自增操作便可遍历所有容器中元素（的迭代器）。

find(x.begin(),x.end(),target)表示在一个迭代器范围内寻找目标元素，若未找到则返回-1；找到则返回相应的位置。

迭代器指向类型获得需要额外考虑。

在迭代器中作typedef，可以通过value\_type记录模板的参数类型；在后续期望返回模板的实例化类型时，使用value\_type即可。注意，返回值前需要加typename引导；

为了使容器和原生指针类型可以被等效处理，引入iterator\_traits类型萃取器：

typedef typename I::value\_type value\_type(本身是一个模板）

模板特殊化：1.全特殊值，对于模板特殊的类型参数（全部模板参数均有限制），给出特殊的定义函数/类；2.部分特殊化，保留部分模板参数类型，给出特殊定义类与函数。

实现过程在class类名后+“<具体参数类型>”即可（模板部分需要作对映的修改）。看模板处有没有typename来判断是全特化还是部分特化。

迭代器的具体实现还是有点东西的，很依赖与typename与iterator\_traits

迭代器也分为顺序访问、双向访问、随机访问......

定义类模板后，在类函数的具体实现时，同样需要模板使用；

构造函数也可以在类外定义；



1. 异常处理

在发生异常的地方 throw<<something>>(比如直接throw a)即可，如果exception一直没有被捕获，它便一直向上传递；

使用try{需要检测的代码运行部分}catch(可能捕获的异常throw类型及对象){捕获异常后对映的执行操作}，异常经过捕获处理后，便不会再向上传递。

也可以把上式作为中途处理，但只需要throw关键词即可，异常依然会继续向上传递。

catch(...)表示抓取所有类型的异常。

try block 类似于case结构：

try{}catch{}catch{}，catch从上至下一一check匹配，如果通过类型转换可以符合，catch直接生效，因而catch存放的顺序十分重要。

Catch匹配：

1. 完全匹配；
2. 可以进行基类转换的匹配；
3. 所有类型匹配。

如果直接在构造函数中throw异常，会导致析构函数不被调用，因而会造成内存泄露的问题。需要手动完成析构函数再throw或者进行包装。

包装即把成员装入其他类中，在本类中使用其他类作为成员，因而成员类会先于类构造，因而在当前类构造函数中若发生异常，成员类也会调用相应的析构函数，不会造成内存的泄漏。

使用智能指针，在构造函数throw异常时，其会根据变量的状态自发调用析构以死亡。使用智能指针时，需要#include<memory>再使用unique\_ptr.

析构函数中不能throw异常，会出现问题。

一般使用引用去catch throw出来的异常变量。（使用指针catch也可以，但不要忘记delete对映的指针）

如果抛出的异常没有被对映的catch捕获，则会触发std::terminate()直接中断程序。

一些其他的主题

类型转换：

Static\_cast：真实的转换类型，只check变量的表面类型，不check具体二进制表达；派生类允许被父类强转，普通类不行；（可以下行，memory本身内容不变且不检查）

static\_cast < type-id > ( expression ) 类型（不用指针的形式） + 变量名；

不同类（没有继承关系）之间进行转化，也会报错；有继承关系，则上行下行均不检验；

Reinterpret\_cast：具体二进制不发生改变，指向统一区域声明的指针类型不同（二进制读取方式不同）；

Int\* a = reinterpret\_cast<int\*> (ab); 二进制读取的方式不同；

Const\_cast：真实获得const类型的地址，可以改变const变量对映地址的数值，但不能改变const变量的值（与编译器宏优化有一定的关系）；

const\_cast<int \*>(p); 需要使用 地址 + 变量名的形式；const 指针不会被宏优化，只有const变量会被优化；

Dynamic\_cast：只有基类与派生类之间可以发生关系（基于实际的指针）；依赖于虚函数virtual的引导（只有在virtual下才可以实现），没有virtual会产生报错；

基类中至少有一个virtual函数

**dynamic\_cast**<D\*>(pb); 同样需要用到指针，以实际类型作为参考的标准；

多重继承

在class A：后增添多个属性即可；

继承双方的成员变量及函数，符合各自的继承规则；如果有两个同名属性，则不能对其调用，但会两个同名属性均会保留；

使用虚基类则会在很大程度上缓解冲突的问题；增加virtual关键词的继承，等价于只有基类产生了一份数据变量，两个虚继承只是同时指向了目标数据的位置，因而在真正派生类中只会有一份数据变量。

可以在使用时指明继承来源；

添加virtual关键字--->虚继承，可以使不同继承类之间完成协议，共用一份虚基类；

在不同namespace尽量避免使用同名函数；

Namespace尽量在.h文件中定义；

具体函数的实现需要在前添加特定命名空间的限定，或同样放入namespace中；

只用自定义namespace时，需要引用对映的头文件，并且在使用函数时增加领域限定；

可以使用using namespace名::函数名的形式，只需声明一次就可像一般函数使用（选择需要使用的函数进行声明）；

尽量不要using namespace 直接引入全部空间，极易引起函数重名的问题；

Namespace可以嵌套，对需要的空间/空间函数可以做一整合（可以附加额外的函数）；

Namespace A = B，可以将B命名空间重命名为A；

同一命名空间可以在不同.h文件中被定义（附加新的函数）；

Unique\_ptr<A\*> xxx; --->为一指针；注意<>中为指针！！

A x;

Std::Shared\_ptr<A> xxx = make\_shared<A> x ，表示xxx的指向与x一致；use\_count可以用于查看当前使用同一内存个数；不能用一个原始指针初始化多个shared\_ptr；

注意<>中为类型即可；

Weak\_ptr; 即使指针被释放，依然可以查看变量；