一、一些需要注意的点

this指针：非静态数据、非静态函数成员（构造、析构）、虚函数、纯虚函数都有

静态函数成员通常在类里以static说明或定义，它没有this参数

静态函数成员的参数表后不能出现const、volatile、const volatile等修饰符。

虚基类最早被构造，最晚被析构。仅一次。

缺省参数在参数表最右边。

C++中，friend、static、virtual为什么不能同时用 注：static和friend可以同时

在类外定义静态的东西前面不再加static

友元关系不能传递、互换

friend static virtual 总结

在声明普通友元时，也可同时定义函数体(自动内联)内联的友员函数的存储类默认为static，作用域局限于当前代码文件

构造函数不能虚，析构函数可以

派生类中重定义虚函数可以不写virtual（无限传递），但函数原型必须相同。但如果基类的析构函数定义为虚析构函数，则派生类的析构函数就会自动成为虚析构函数（原型不同），涉及分配动态内存的类，一定要这样做。

虚函数从最底层向上找，除非明确说明调用A::f()

对象数组指针p应用delete []p释放

当用常量对象、类型为&&的返回对象作为实参调用函数时，优先调用的函数是带有&&参数的函数

纯虚函数：没有函数题的虚函数。使用时一句声明virtual void f1( )=0;

抽象类：定义了新纯虚函数的类、继承了纯虚函数但没定义函数体的类。主要用于定义派生类共有的数据和函数成员，常作为基类。不能有对象、被传参或返回，可以有指针。

虚函数可以作为友元。纯虚函数也可以（最好别）。

用union声明的类既不能作派生类的基类，也不能作任何基类的派生类

union既不能派生又不能继承

一个函数若不是返回左值引用，则返回值都是右值。

谁能引用右值：const的左值引用和右值引用。

右值引用类型的变量本身是左值，不能再被右值引用。

A::b 只能恢复到基类的访问权限

二、改错常见错误

数据成员定义为virtual

const类型成员变量和引用类型变量要在参数表处进行初始化

构建子类的时候要构造基类，说明基类的构造方法

析构函数不能定义为保护和私有，不然无法进行调用

本类的符号重载函数不能定义成友元

static，friend，virtual不能同时使用

成员指针不能指向物理地址

访问权限问题

返回值类型要匹配

不能用 void定义非指针类型的数据成员

不能将构造函数声明为虚函数

static类型变量不能在类的体内进行初始化

构造函数参数个数应当匹配

不存在普通赋值函数的重载，=必须重载为双目

构造函数不能定义为不带this参数的静态函数成员

int operator++(double);必须用int作为第一个参数给后置运算

构造函数和析构函数的this不能说明为c

三、编程题的注意事项

所有成员函数头要包括T：：

返回值类型为T&的函数必须用return \*this返回

返回值类型为T的含糊是必须定义局部对象或者使用常量对象COMPLEX

COMPLEX::operator - ( ) const{ return COMPLEX(-real, -imag); }

const不能漏掉，凡尾部部带const的函数不能改变当前对象的值。

STACK::STACK(**int** m) :q(m), QUEUE(m) {};

//初始化栈：最多存放2m-2个元素

STACK::STACK(**const** STACK& s) :QUEUE(s), q(s.q) {};                 //用栈s深拷贝初始化栈

STACK::STACK(STACK&& s)**noexcept** :q((QUEUE&&)s.q), QUEUE((QUEUE&&)s) {

};//同样，用栈s移动拷贝初始化栈

**int**  STACK::size()**const** **noexcept** {

**int** m = 0; m = q.size();//m是q里面存的数目

**int** n = 0; n = QUEUE::size();//n是继承来的队列，存的数目

    m = m + n;//两个队列元素个数之和

**return** m;

};

//和上面的一摸一样

STACK::**operator** **int**() **const** **noexcept** {

**int** n = 0, m = 0;

    n = q.**operator** **int**();

    m = QUEUE::**operator** **int**();

    n = m + n;

**return** n;

};

//入栈

STACK& STACK::**operator**<<(**int** e) {

**int** n = QUEUE::**operator** **int**();

**int** m = QUEUE::size();

    //求出两个队列的值

**if** (n < m - 1) {

        //如果栈不满，那么就可以入栈，就是继承

        QUEUE::**operator**<<(e);

**return** (\***this**);

    }

    //第一个队列满了

    //判断第二个队列是不是为满

    n = q.**operator** **int**();

    m = q.size();

    //满：报错

**if** (n == m - 1) {

**throw** "STACK is full!";

    }

    //不满：进栈

**else** {

        q.**operator**<<(e);

    }

**return** (\***this**);

};

STACK& STACK::**operator**>>(**int**& e) {

**int** n = QUEUE::**operator** **int**();

**int** m = q.**operator** **int**();

**int** size = QUEUE::size();

    //求第一个队列的元素个数，如果第一个队列元素都是没了。那就是空栈

**if** (n == 0) { **throw** "STACK is empty!"; }

**int** i;

**int**\* elem = **NULL**;

**if** (m) {//queue满

        elem = **new** **int**[m];//给elem赋值

**for** (i = 0; i < m - 1; i++) {

            q.**operator**>>(elem[i]);//第二个队列的每一个元素都出队列

        }

        q.**operator**>>(e);//最后一个队列元素出栈

**for** (i = 0; i < m - 1; i++) {

            q.**operator**<<(elem[i]);//重新入队

        }

    }

**else** {//queue 不满

        elem = **new** **int**[n];//给elem赋值

**for** (i = 0; i < n - 1; i++) {

            QUEUE::**operator**>>(elem[i]);//就看第一个队列的元素

        }

        QUEUE::**operator**>>(e);//最后一个元素出队

**for** (i = 0; i < n - 1; i++) {

            QUEUE::**operator**<<(elem[i]);//同理

        }

    }

**return** (\***this**);

};

STACK& STACK::**operator**=(**const** STACK& s)    //深拷贝赋值并返回被赋值栈

{

    //std::cout << "STACK& STACK::operator=(const STACK& s)" << std::endl;

**if** (**this** == &s) {

**return** \***this**;

    }

    q = s.q;

    QUEUE::**operator**= (s);

**return** \***this**;

}

STACK& STACK::**operator**=(STACK&& s)**noexcept**//移动赋值并返回被赋值栈

{

**if** (**this** == &s) {

**return** \***this**;

    }

    q = (QUEUE&&)s.q;

    QUEUE::**operator**=((STACK&&)s);

**return** \***this**;

}

//利用队列的print进行操作

**char**\* STACK::print(**char**\* b)**const** **noexcept** {

    QUEUE::print(b);

    q.print(b);

**return** b;

};

STACK::~STACK()**noexcept** {

    q.~QUEUE();

    QUEUE::~QUEUE();

};

QUEUE::QUEUE(**int** m) :max(0), elems(**NULL**)

{

    \*(**int**\*\*)&elems = **new** **int**[m];

    \*(**int**\*)&max = m;

    head = tail = 0;

}

QUEUE::QUEUE(**const** QUEUE& q) :max(0), elems(**NULL**)//用q深拷贝初始化队列

{

**if** (**this** == &q) **return**;

    \*(**int**\*\*)&elems = **new** **int**[q.max];

    \*(**int**\*)&max = q.max;

    memcpy(elems, q.elems, q.max \* 4);

    head = q.head;

    tail = q.tail;

}

QUEUE::QUEUE(QUEUE&& q)**noexcept** :max(0), elems(**NULL**)

{

**if** (**this** == &q) **return**;

    \*(**int**\*\*)&elems = q.elems;

    \*(**int**\*)&max = q.max;

    head = q.head;

    tail = q.tail;

    \*(**int**\*\*)&q.elems = **nullptr**;

    \*(**int**\*)&q.max = 0;

    q.head = q.tail = 0;

}

QUEUE& QUEUE::**operator**=(**const** QUEUE& q)    //深拷贝赋值并返回被赋值队列

{

**if** (**this** == &q) **return** \***this**;

**if** (!q.elems || !elems) **return** \***this**;

**if** (q.head == q.tail)

    {

        //throw("QUEUE is empty!");

        tail = q.tail;

        head = q.head;

        \*(**int**\*)&max = q.max;

**return** \***this**;

    }

    memcpy(elems, q.elems, q.max \* 4);

    \*(**int**\*)&max = q.max;

    head = q.head;

    tail = q.tail;

**return** \***this**;

}

QUEUE& QUEUE::**operator**=(QUEUE&& q)**noexcept**  //移动赋值并返回被赋值队列

{

**if** (**this** == &q) {

**return** \***this**;

    }

    free(elems);

    \*(**int**\*\*)&(elems) = q.elems;

    \*(**int**\*)&(max) = q.max;

    tail = q.tail;

    head = q.head;

    \*(**int**\*\*)&q.elems = **nullptr**;

    \*(**int**\*)&q.max = 0;

    q.tail = q.head = 0;

**return** \***this**;

}

QUEUE::~QUEUE()        //销毁当前队列

{

**if** (elems == **nullptr**) {

**return**;

    }

**delete**  elems;

    \*(**int**\*\*)&elems = **nullptr**;

    \*(**int**\*)&max = 0;

    head = tail = 0;

}

QUEUE& QUEUE::**operator**<<(**int** e)      //将e入队列尾部，并返回当前队列

{

    //std::cout << "QUEUE& QUEUE::operator<<(int e)" << std::endl;

**if** (!elems) **return** \***this**;

**if** (head == (tail + 1) % max)

    {

**throw**("QUEUE is full!");

**return** \***this**;

    }

    elems[tail % max] = e;

    tail = (tail + 1) % max;

**return** \***this**;

}

QUEUE& QUEUE::**operator**>>(**int**& e)     //从队首出元素到e，并返回当前队列

{

**if** (!elems) **return** \***this**;

**if** (head == tail)

    {

        //throw("QUEUE is empty!");

**return** \***this**;

    }

    e = elems[head % max];

    head = (head + 1) % max;

**return** \***this**;

}

]QUEUE::**operator** **int**() **const** **noexcept**    //返回队列的实际元素个数

{

**if** (!max) {

**return** 0;

    }

**else** {

**return** (max + tail - head) % max;

    }

}

**int** QUEUE::size() **const** **noexcept**        //返回队列申请的最大元素个数max

{

**return** max;

}

**char**\* QUEUE::print(**char**\* s) **const** **noexcept**   //打印队列至s并返回s

{

**if** (!elems)

    {

**return** s;

    }

**if** (head == tail)

    {

        //throw("QUEUE is empty!");

**return** s;

    }

**char**\* str = **nullptr**;

    str = **new** **char**[max];

    //strcpy(s, "\0");

**for** (**int** i = head; i != tail; i = (i + 1 + max) % max)

    {

        sprintf(str, "%d", elems[i]);

        strcat(s, str);

        strcat(s, ",");

    }

**return** s;

}

1. **作业五答案**

1. 如果有 3 个异常处理 hi 过程：catch(…)、catch(const void \*)、catch(int \*)，应如何摆放它们的

位置？catch(int \*)可以放在 catch(const void \*)后面而不影响其捕获异常吗 ？

**答案：**(1) 根据下面的顺序摆放：catch(int \*)、catch(const void \*)、catch(…)。

(2) 将 catch(int \*)放在 catch(const void \*)后面，将导致 catch(int \*)失效。

2. int x, 显式转换(int)x 的结果是左值还是右值？

**答案：**右值。

3. 对于全局变量 const int x = 0，能够使用 \*const\_cast<int \*>(&x) = 3 修改 x 的值吗？用 \*(int \*)(&x) = 3 可以修改 x 的值吗？

**答案：**都不能。非类内定义的简单类型变量即使转换为可写左值，也不能修改其值 (受到页面保 护机制的保护）。

4. 说明 static\_cast 和 const\_cast 的区别。

**答案：**static\_cast 和 const\_cast 同 C 语言的强制类型转换用法基本相同，但 static\_cast 不能修改源

类型中的 const 和 volitale 属性，const\_cast 则可以。

5. 对于 Lambda 表达式：auto f = [x] (int y) -> int { return ++x + y; }，解释 [x] (int y) 中的 x 和 y

的意义，并说明为什么 f 名义上是一个函数，但实际上是一个对象。

**答案：**[x]表示捕捉 lambda 函数体外变量 x 的值，并根据这个值创建匿名类的实例成员变量 x。(int y)表示匿名类的( )运算符的重载函数 operator( )(…)的调用参数，即 int operator( )(int y)。