·char \* strcpy ( char \* destination, const char \* source );

·char \* strncpy ( char \* destination, const char \* source, size\_t num ); str3[5] = '\0'; ！！//复制字符串

·char \* strcat ( char \* destination, const char \* source );

前是目标数组，后是要添加到后面的源数组

·int strcmp ( const char \* str1, const char \* str2 );//相等

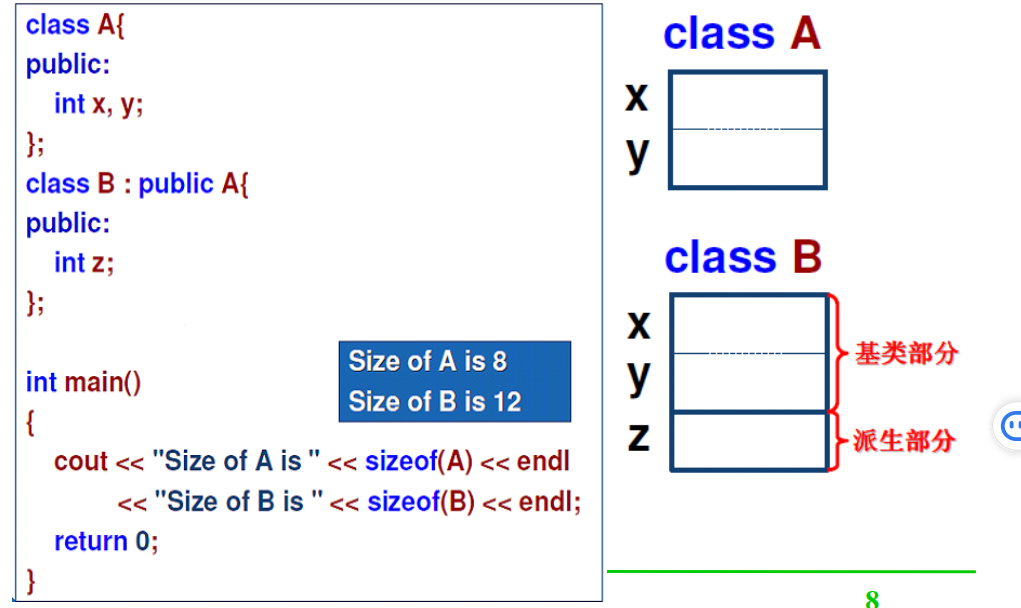
·const char \* strchr ( const char \* str, int character );// 找某个字符第一次出现的位置

·const char \* strstr ( const char \* str1, const char \* str2 );

·void \* memset ( void \* ptr, int value, size\_t num );

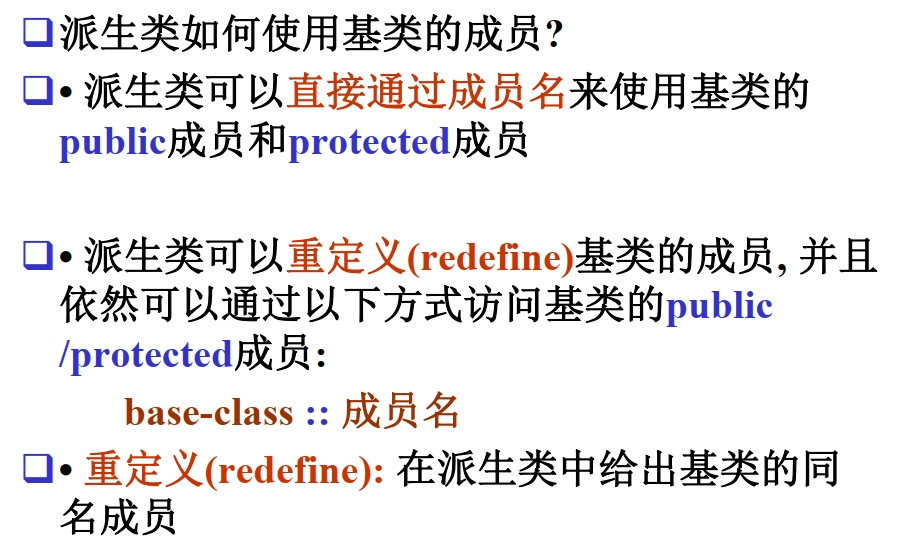
ptr指向要修改的内存块的起始地址，value是要修改成什么值，num是修改多少个。

·size\_t strlen ( const char \* str );

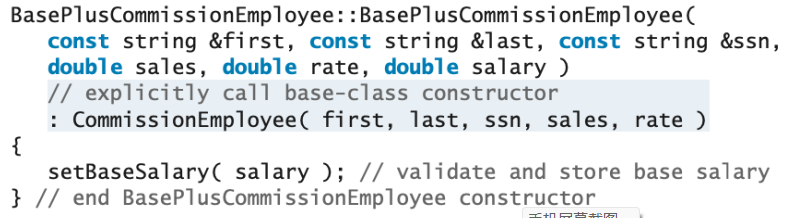


基类的protected成员只能被基类的成员和友元

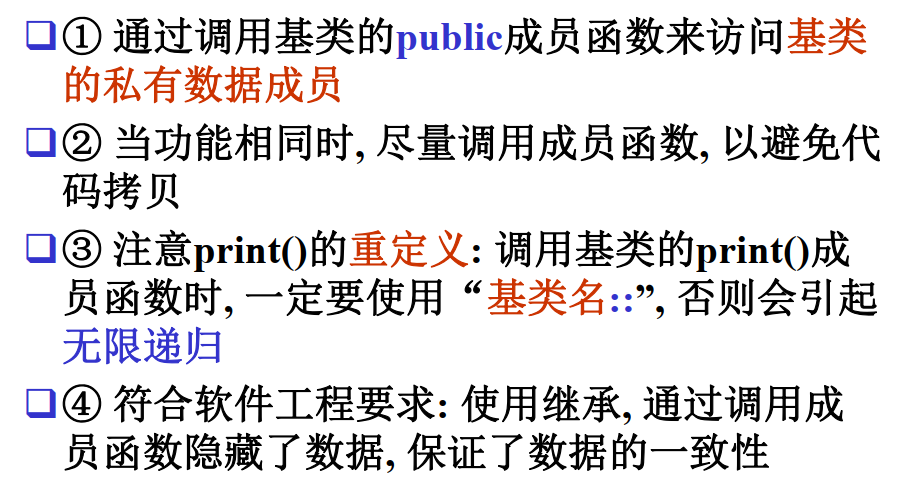
函数+ 派生类的成员和友元函数访问



派生类中基类的构造：



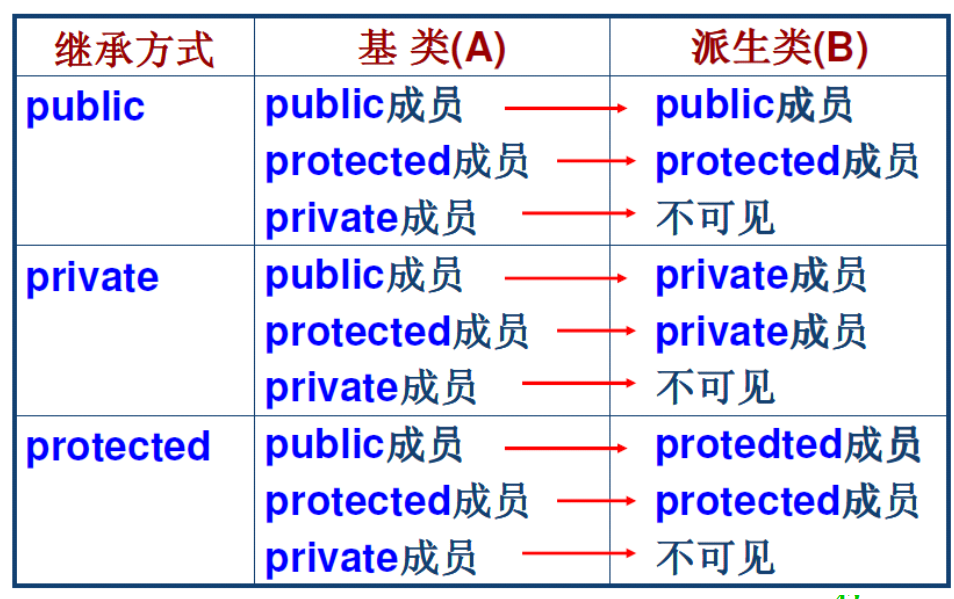


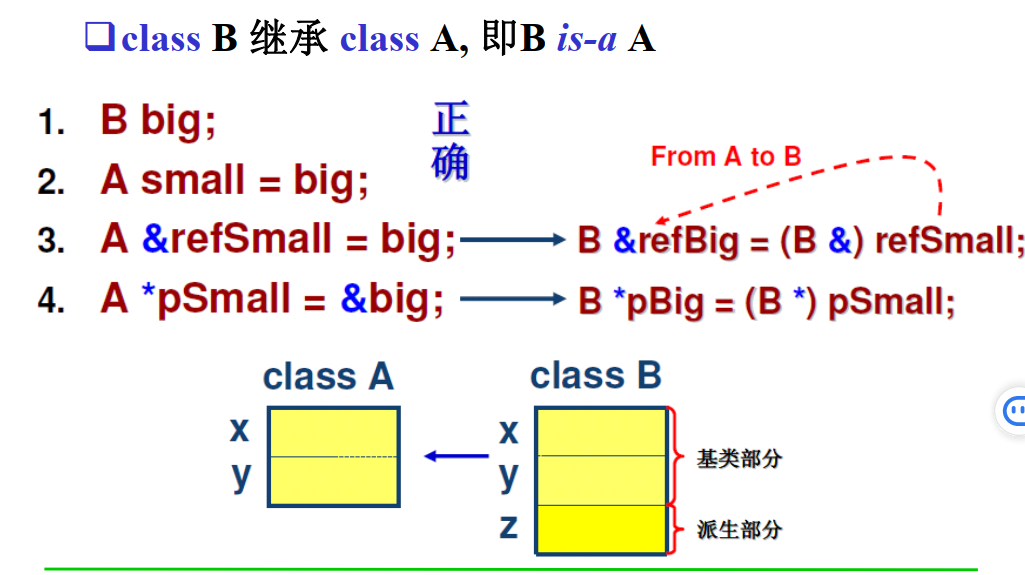


派生类也可以调用基类函数，但最好加一个作用域“A：：“，或者基类加一个virtual函数 这样能对应











派生类中覆盖(Overridden)的虚函数和基类中

的虚函数必须函数签名和返回值均相同



class A！！！！关于多态和虚函数

{public:

void testfuc() {func();}

void func() {cout << "A::fuc\t";vfunc(); }

virtual void vfunc() {cout << "A::vfunc" << endl;}};

class B :public A {

public:void func()

{cout << "B::func nothing called" << endl;}

virtual void vfunc()

{cout << "B::vfunc" << endl;}};

int main()

{ A a;

a.func();//A::fuc A::vfunc

B b;

b.func();//B::func nothing called

b.testfuc();//A::fuc B::vfunc

A\* p = &b;

p->vfunc();//B::vfunc

p->func();//A::fuc B::vfunc

p->testfuc();//A::fuc B::vfunc}

class A {

public:

A(){ aC++; }//ac=10

virtual ~A() {aC--;}static int aC;

};int A ::aC = 10;

class B :public A

{

public:

B() { aC++; }//ac=20

~B() { aC--; }static int aC;

};int B ::aC = 20;

A a;

B b;

void f(A a) { cout << a.aC << endl; }

int main() {

//B在构造的时候也会调用A的构造函数，a和b同时++

A al;//A13 B21

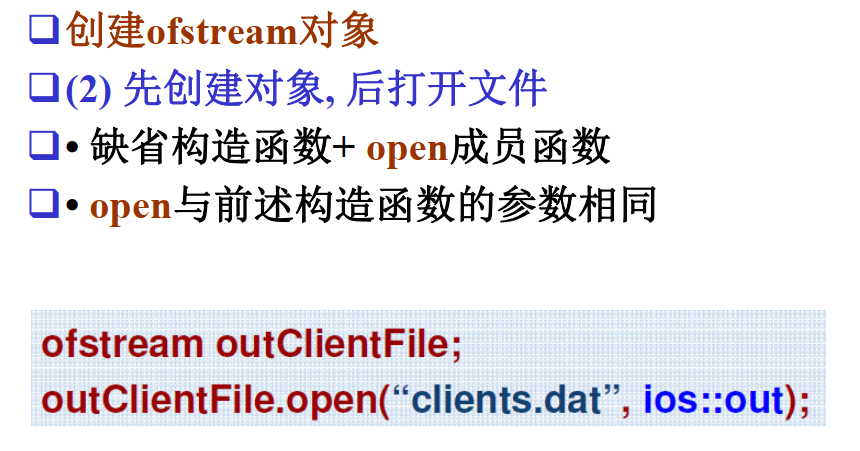
A\* p = new B();//A14 B22 创建指针不算构造函数 这里只构造了一个B

delete p;//A13 B21

f(b);//13 A在参数列表被构造 下一行又被析构 但是这里的f输出a的值（a的拷贝构造）

//A12 B21}

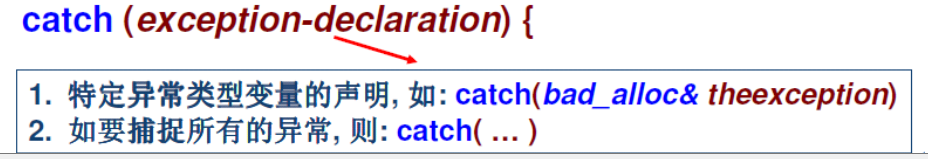


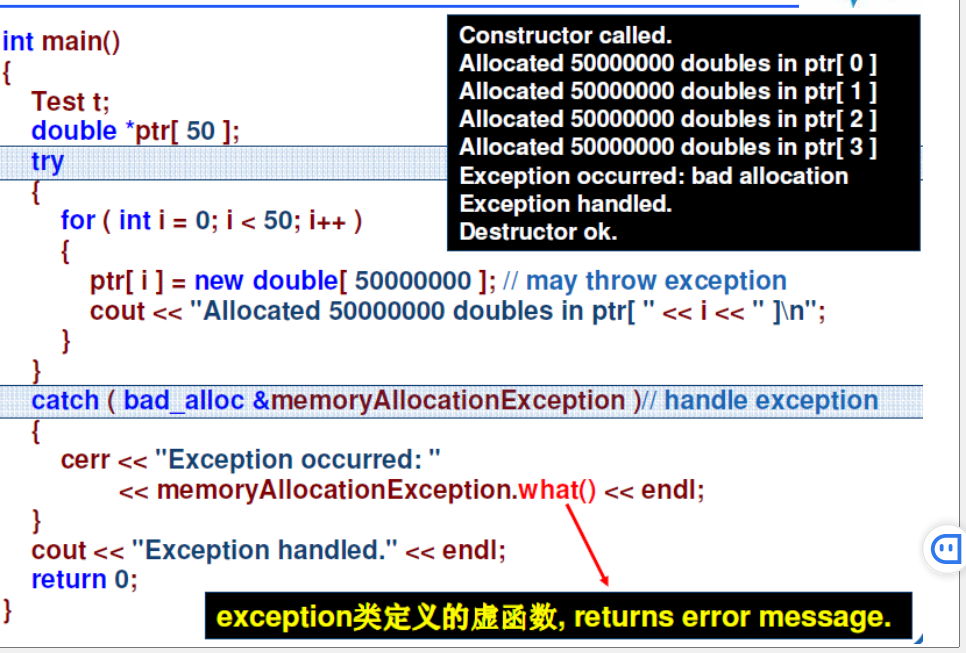


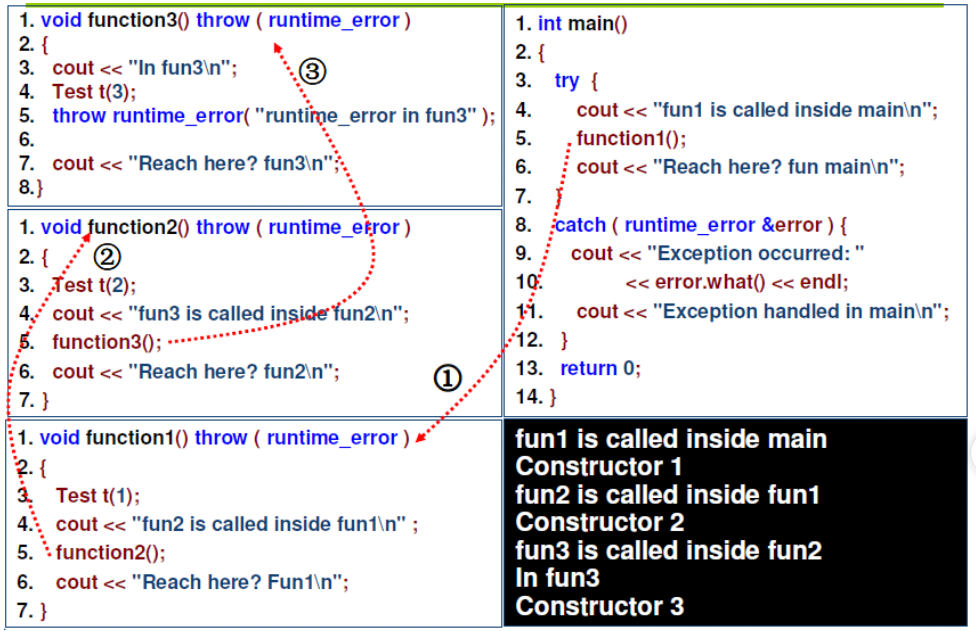
文件关闭记得调用“.close()“函数

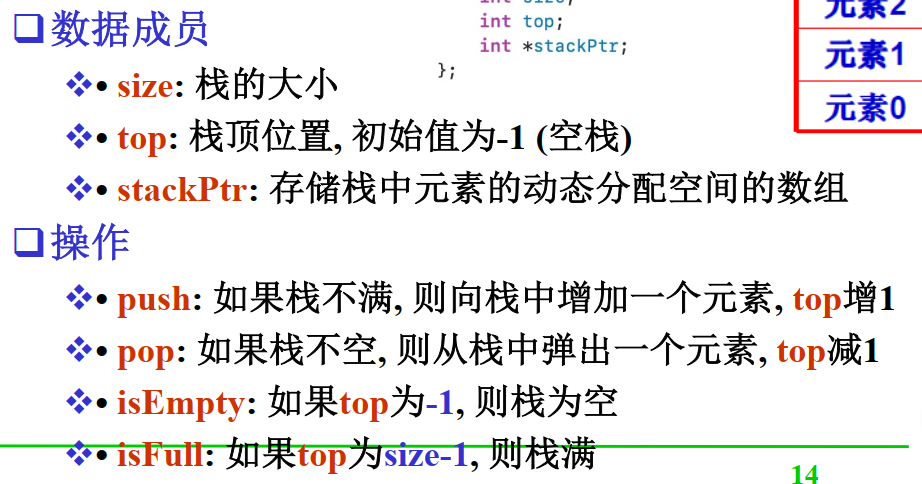




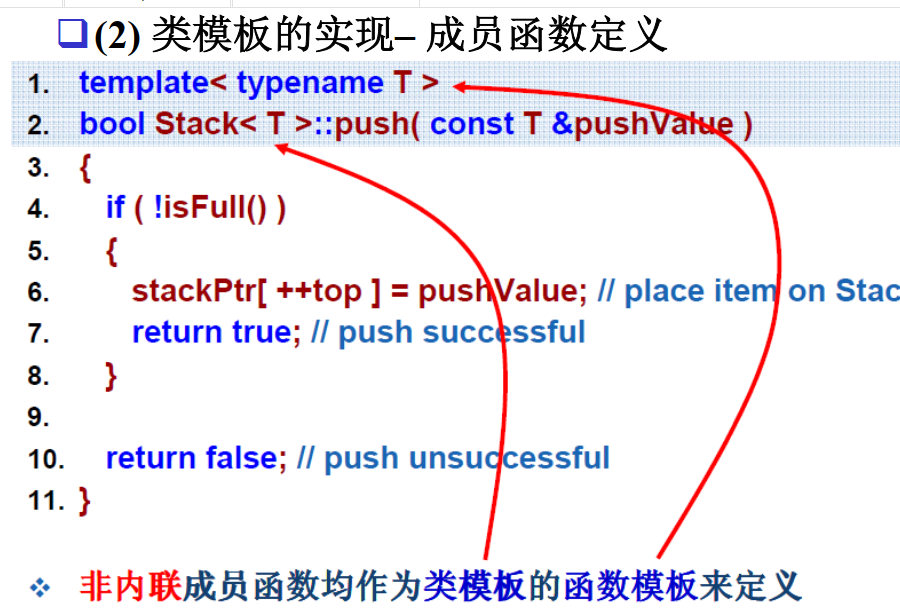








！！类模板中的T可用于成员函数(形参、局部变量和返回值)、数据成员



顺序文件输入输出

#include<iostream>

using namespace std;

#include<fstream>

#include<string>

#include<vector>

#include<iomanip>

int main()

{

ofstream fs("datasize.dat");

if (!fs.is\_open())

{cout << "Fail to open!";return -1; }

else

{cout.flags (ios::left);fs << left << setw(25) <<"char" << sizeof(char)<<endl;fs << setiosflags(ios::left) << setw(25) << "unsigned char" << sizeof(unsigned char) << endl;}

fs.close();

ifstream ifs("datasize.dat");

if (!ifs.is\_open())

{ cout << "Fail to open!";

return -1;}

/\*char buf[1024] = { 0 };

while (ifs >> buf)

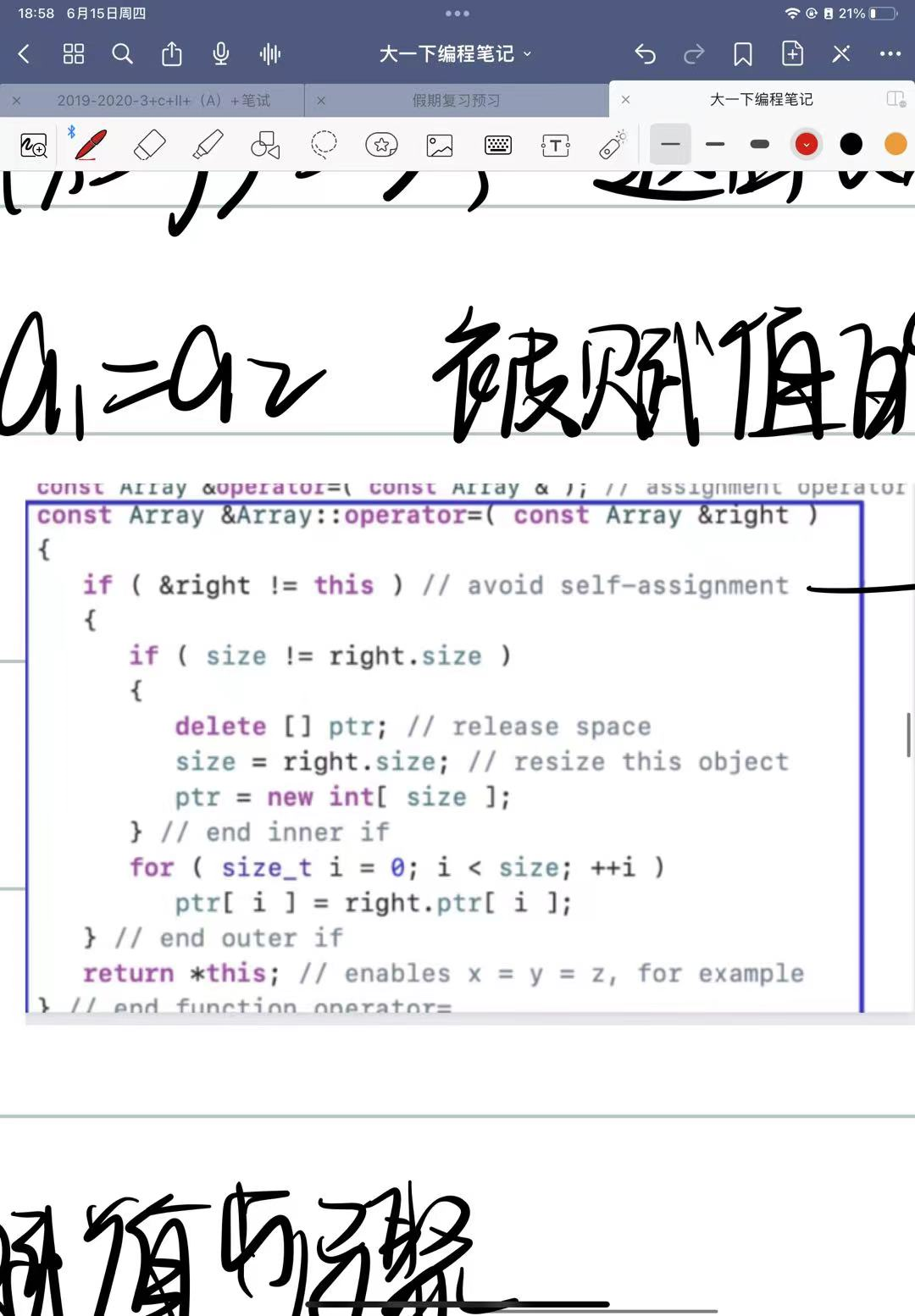
{cout << buf << endl;}\*/

string line;

// 读取文件中每一行并输出

while (getline(ifs, line)) {cout << line << endl;}

ifs.close();

return 0；}  
赋值运算符首先要判断是否给自己赋值！！

步骤：1.拿到size，2.new int[size]，3.循环赋值！

返回类型：字符串引用，加上const防止（a=b）=c

·除非基类有缺省构造函数，否则继承类的构造函数后要加上：date（a，b）

Class B: public A, public C

运算符重载：

Person operator+(const Person& p)

{ Person temp;

temp.m\_A = this->m\_A + p.m\_A;

return temp;}

//全局函数实现左移重载

//ostream对象只能有一个

ostream& operator<<(ostream& out, Person& p) {

out << "a:" << p.m\_A << " b:" << p.m\_B;

return out;

}

！！！自增运算符重载

//前置++

MyInteger& operator++() {//先++ ：m\_Num++;

//再返回： return \*this;}

//后置++

MyInteger operator++(int) {//先返回

MyInteger temp = \*this; //记录当前本身的值，然后让本身的值加1，但是返回的是以前的值，达到先返回后++；m\_Num++;return temp;}

//重载赋值运算符

Person& operator=(Person &p)

{

if (m\_Age != NULL)

{

delete m\_Age;

m\_Age = NULL;

}

//编译器提供的代码是浅拷贝

//m\_Age = p.m\_Age;

//提供深拷贝 解决浅拷贝的问题

m\_Age = new int(\*p.m\_Age);

//返回自身

return \*this;}

//判断相等运算符重载（类内）

bool operator==(Person & p)

{if (this->m\_Name == p.m\_Name && this->m\_Age == p.m\_Age)

{return true;}

return false；}

//函数调用运算符重载：

·void operator()(string text)

{cout << text << endl;}//Myprint类内

MyPrint myFunc;

myFunc("hello world");

·int operator()(int v1, int v2)

{return v1 + v2;}

cout << MyAdd()(100, 100) << endl;

异常抛出：（先析构再cout）

#include <iostream>

#include <stdexcept>

using namespace std;

class Obj

{public:

Obj(int i = 0) :id(i)

{cout << "Con obj " << id << endl;}

~Obj()

{cout << "Des obj " << id << endl;}

int id;};

void function3() throw (runtime\_error)

{Obj b31(31);

throw runtime\_error("Rerror in function3");

Obj b32(32);}

void function2() throw (runtime\_error)

{Obj b21(21);

function3();

Obj b22(22);}

void function1() throw (runtime\_error)

{try{function2();

Obj b11(11);}

catch (runtime\_error)

{cout << "Nothing happened." << endl;throw;}

Obj b12(12);}

int main()

{try

{Obj b41(41);

function1();

cout << "End try" << endl;}

catch (runtime\_error& error)

{cout << error.what() << endl;

Obj b42(42);}

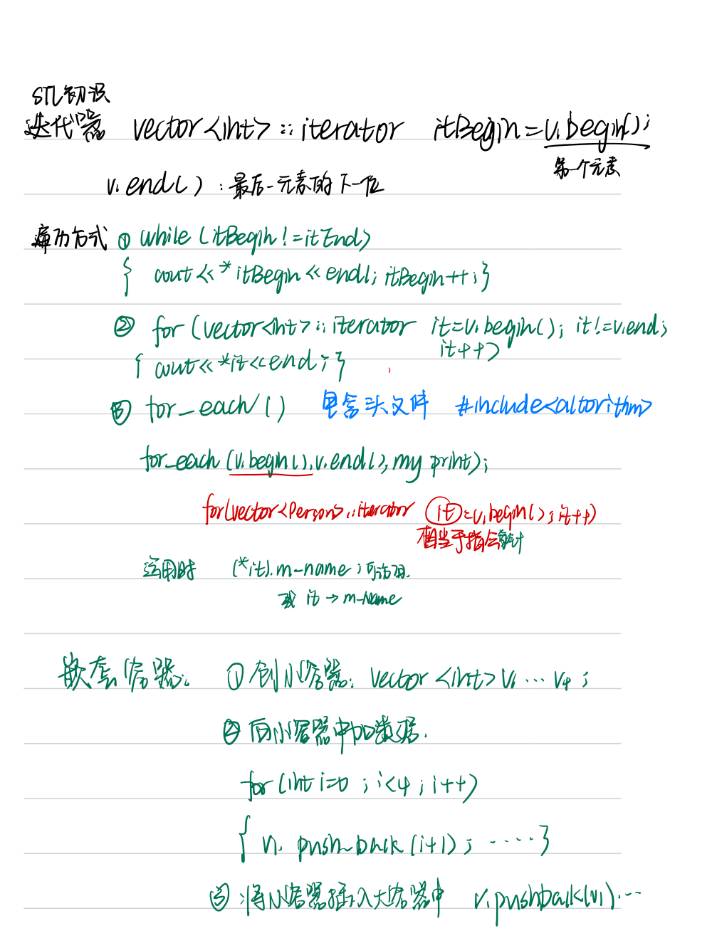
cout << "exit main function" << endl;

return 0;}

访问循环队列：

i=(i+1)%q.size

类模板里的运算符重载直接写在友元声明里！



\* 访问子类同名成员 直接访问即可

\* 访问父类同名成员 需要加作用域

当子类与父类拥有同名的成员函数，子类会隐藏父类中同名成员函数，加作用域可以访问到父类中同名函数

template<typename T>

// Person p("孙悟空", 1000); // 错误 类模板使用时候，不可以用自动类型推导

Person <string ,int>p("孙悟空", 1000); //必须使用显示指定类型的方式，使用类模板；类模板在模板参数列表中可以有默认参数：Person <string> p("猪八戒", 999);

#pragma warning(disable:4996)

template<class T>//数组类封装

class MyArray

{

public:

//构造函数

MyArray(int capacity)

{ this->m\_Capacity = capacity;

this->m\_Size = 0;

pAddress = new T[this->m\_Capacity];

}

//拷贝构造

MyArray(const MyArray & arr)

{ this->m\_Capacity = arr.m\_Capacity;

this->m\_Size = arr.m\_Size;

this->pAddress = new T[this->m\_Capacity];

for (int i = 0; i < this->m\_Size; i++)

{//如果T为对象，而且还包含指针，必须需要重载 = 操作符，因为这个等号不是 构造 而是赋值， // 普通类型可以直接= 但是指针类型需要深拷贝

this->pAddress[i] = arr.pAddress[i];

} }

//重载= 操作符 防止浅拷贝问题

MyArray& operator=(const MyArray& myarray) { if (this->pAddress != NULL) {

delete[] this->pAddress;

this->m\_Capacity = 0;

this->m\_Size = 0;}

this->m\_Capacity = myarray.m\_Capacity;

this->m\_Size = myarray.m\_Size;

this->pAddress = new T[this->m\_Capacity];

for (int i = 0; i < this->m\_Size; i++) {

this->pAddress[i] = myarray[i];

}return \*this;

} //重载[] 操作符 arr[0]

T& operator [](int index)

{return this->pAddress[index]; //不考虑越界，用户自己去处理

//尾插法

void Push\_back(const T & val)

{if (this->m\_Capacity == this->m\_Size)

{return;}

this->pAddress[this->m\_Size] = val;

this->m\_Size++;

} //尾删法

void Pop\_back()

{ if (this->m\_Size == 0)

{return;}

this->m\_Size--;}

//获取数组容量

int getCapacity()

{return this->m\_Capacity;

}

//获取数组大小

int getSize()

{return this->m\_Size;

}

//析构

~MyArray()

{if (this->pAddress != NULL)

{delete[] this->pAddress;

this->pAddress = NULL;

this->m\_Capacity = 0;

this->m\_Size = 0;

}}

//迭代器

vector<int>::iterator pBegin = v.begin();

vector<int>::iterator pEnd = v.end();

//第一种遍历方式：

while (pBegin != pEnd) {

cout << \*pBegin << endl;

pBegin++;

}

//队列

构造函数：

queue<T>que; //queue采用模板类实现默认构造形式

queue(const queue &que); /拷贝构造函数

赋值操作：

queue& operator=(const queue &que); //重载等号操作

数据存取：

push(elem);//往队尾添加元素

pop();//从队头移除第一个元素

back();返回最后一个元素

front();//返回第一个元素

大小操作：

empty();//判断堆栈是否为空

size();//返回栈的大小

写文件步骤如下：

1. 包含头文件

\#include <fstream\>

2. 创建流对象

ofstream ofs;

3. 打开文件

ofs.open("文件路径",打开方式);

4. 写数据

ofs << "写入的数据";

5. 关闭文件

ofs.close();

void test01()

{

ofstream ofs;

ofs.open("test.txt", ios::out);

ofs << "姓名：张三" << endl;

ofs << "性别：男" << endl;

ofs << "年龄：18" << endl;

ofs.close();

}

读文件步骤如下：

1. 包含头文件

\#include <fstream\>

2. 创建流对象

ifstream ifs;

3. 打开文件并判断文件是否打开成功

ifs.open("文件路径",打开方式);

4. 读数据

四种方式读取

5. 关闭文件

ifs.close();

#include <fstream>

#include <string>

void test01()

{

ifstream ifs;

ifs.open("test.txt", ios::in);

if (!ifs.is\_open())

{

cout << "文件打开失败" << endl;

return;

}

//第一种方式

//char buf[1024] = { 0 };

//while (ifs >> buf)

//{

// cout << buf << endl;

//}

//或者

//string buf;

//while (getline(ifs, buf))

//{

// cout << buf << endl;

//}

char c;

while ((c = ifs.get()) != EOF)

{

cout << c;

}

ifs.close();

}