封装与接口：

函数参数的缺省值必须在不缺省值之后

函数参数缺省值可能造成二义性，使得编译不通过

auto关键词可以自动推导变量类型，所以必须在定义时初始化，在编译期间确定类型，函数参数不能被声明为auto（auto i=1;）

decltype可以对变量或者表达式结果的类型进行推导，方便定义（int i=1;decltype(i) p=2;）

内存的申请和释放：

申请单个变量空间：int\* ptr=new int(10);

申请数组空间：int\* array=new int[10];

删除单个变量空间：delete ptr;

删除数组空间：delete[] array;

NULL==0

零指针nullptr

基于范围的for循环：int arr[3]={1,3,9};for(int e:arr){cout<<e<<endl;}

冒号前为用于迭代的变量，冒号后为迭代范围

内联函数：inline int max(int a,int b){…} 使用内联函数，编译器自动生成等价的表达式

避免对大段代码使用内联修饰符；避免对包含循环和复杂控制结构的函数使用内联定义；避免将内联函数的声明和定义分开；定义在类声明中的函数默认为内联函数；一般构造函数、析构函数都被定义为内联函数；内联修饰符更像建议；编译器可以拒绝不合理的内联请求；编译器会对一些没有内联修饰符的函数自行判断是否转化为内联函数

由类设计者决定如何初始化和清除；编译器决定何时初始化和清除

对象初始化是编译器在创建对象处自动生成调用构造函数的代码来完成的，构造函数是类的特殊成员函数

构造函数没有返回值，函数名与类名相同

类的构造函数可以重载，即可以使用不同的函数参数进行初始化

构造函数也可以使用初始化列表初始化成员数据，列表在函数体之前、函数参数列表之后，用冒号开头（Student(int id):ID2(id),ID1(ID2),Info(){}）

初始化列表的成员是按照声明的顺序初始化的，而不是按照出现在初始化列表中的顺序

在构造函数的初始化列表中，还可以调用其他构造函数，称为“委派构造函数”

可以进行就地初始化（就地初始化是一种简便的表达方式，实际操作仍然在对象构造中执行）

默认构造函数（不带任何参数的构造函数）

使用默认构造函数生成对象时，其定义格式为：

ClassName a;

ClassName b=ClassName();

在定义对象时，会先调用成员变量的构造，再执行自己的构造函数（L4P.14）

如果没有手动定义默认构造函数，编译器会隐式合成一个无操作的默认构造函数

如果已经定义了其他构造函数，那么编译器不会隐式合成默认构造函数

有时需要手动指定生成默认版本的构造函数，显式声明默认构造函数（A()=default;），这样，即使存在其他构造函数，编译器也会定义隐式默认构造函数

显式删除构造函数：（A(int i)=delete;）

对象数组的初始化：

1. 如果有默认构造函数：A a[50];
2. 如果构造函数只有一个参数：A a[30]={1,3,5};
3. 构造函数有多个参数：A a[3]={A(1,2),A(3,5),A(0,7)};

析构函数：

会先执行自己的析构函数再调用成员变量的析构函数

当没有自定义析构函数时，会自动生成隐式析构函数

局部对象：在程序执行到时初始化，在作用域结束后被析构

全局对象：在main函数调用前初始化，在同一个编译单元中按照定义顺序初始化（同一源文件），不同编译单元中，对象初始化顺序不确定，在main函数return后，对象被析构

全局对象的构造顺序不确定，故而不能有依赖关系，所以最好使用参数替代全局对象

引用：int v0; int & v1=v0;

引用必须在定义时初始化，且不能修改引用指向

函数参数可以是引用类型，表示函数的形参与实参是同一个变量，改变形参将改变实参，函数返回值可以是引用，但不能指向函数临时变量

类的运算符重载：

运算符重载的两种方式：①全局函数的运算符重载（A operator+(A a, A b){…}）②成员函数的运算符重载（class A{public: A operator+(A b){…}}）

可以重载的运算符：①双目运算符②关系运算符③逻辑运算符④单目运算符⑤自增自减运算符⑥位运算符⑦赋值运算符⑧空间申请与释放⑨其他：()，->，，，[]

int a = ++b; //先完成b+1操作，再赋值

int a = b++; //先完成赋值，再b+1操作

返回类型 operator \*\* （参数）

=,[],(),->算符只能通过成员函数来重载

当没有自定义operator=时，编译器会自动合成⼀个 默认版本的赋值操作，在类内定义operator=，编译器则不会自动合成，一般不用全局函数重载

流运算符重载：

istream& operator>> (istream& in, Test& dst );

ostream& operator<< (ostream& out, const Test& src );

不修改istream和ostream类的情况下，只能使 用全局函数重载

友元：

在类内被声明为友元的函数或类具有对出现友元声明的类的private和protected成员的访问权限，即为可以访问该类的一切成员

友元的声明只能在类内进行

Class A{

Friend void foo(A &a);

}

被友元声明的函数一定不是当前类的成员函数，即使其定义在类内

当前类的成员函数也不需要友元修饰

还可以声明别的类的成员函数为当前类的友元（friend void X::foo(Y);）

友元的声明与当前所在域为private和public无关

一个普通函数可以是多个类的友元函数

友元类：

可对class、struct、union进行友元声明，代表该类的所有成员函数均为友元函数

对基础类型的友元声明会被忽略

前置声明

注意事项：①类A中声明B为友元类，B可以访问A，但是A不能访问B②友元不传递③友元不继承④友元声明不能定义新的类

静态变量：

static修饰的变量，初次定义时需要初始化，且只能初始化一次，存储在静态存储区，生命周期持续到整个程序结束，静态全局变量是内部可链接的，作用域仅限其声明的文件，不能被其他文件所用，避免与其他文件中同名变量冲突（即使用extern修饰也不行）

静态函数：

static修饰，内部可链接，作用域为其声明文件，不能被其他文件所用

非静态全局变量和非静态全局函数是外部可链接的，可以被其他文件所用

静态数据成员：使用static修饰的数据成员，隶属于类，称为类的静态数据成员，也称为类变量，性质为：①该成员被类的所有对象共享②类的静态成员既可以通过对象访问，也可以通过类名访问，如ClassName::static\_var ③类的静态数据成员要在实现文件中赋初值 ④和全局变量一样，类的静态数据尘缘在程序开始前初始化

静态数据成员的多文件编译：

静态数据成员应该在.h文件中声明，在.cpp文件中定义，不能在.h文件中同时声明和定义，否则会因为.h的多次包含而被重复定义

静态成员函数：在返回值前添加static修饰的成员函数，称为类的静态成员函数，也可以通过对象来访问，也就是ClassName::static\_function

静态成员函数不能访问非静态成员：静态成员函数属于整个类，在类实例化之前就已经有了内存空间，但是非静态成员必须要在类实例化之后才会分配空间

常量：

1. const修饰变量时，必须就地初始化，该变量的值在其生命周期内不会变化
2. 修饰引用和指针时，不能通过该引用和指针修改相应变量的值，常用于函数参数以保证函数体中不会修改参数的值
3. 修饰函数返回值时（const int\* func(){…}），函数返回值的内容（或其指向的内容）不能被修改

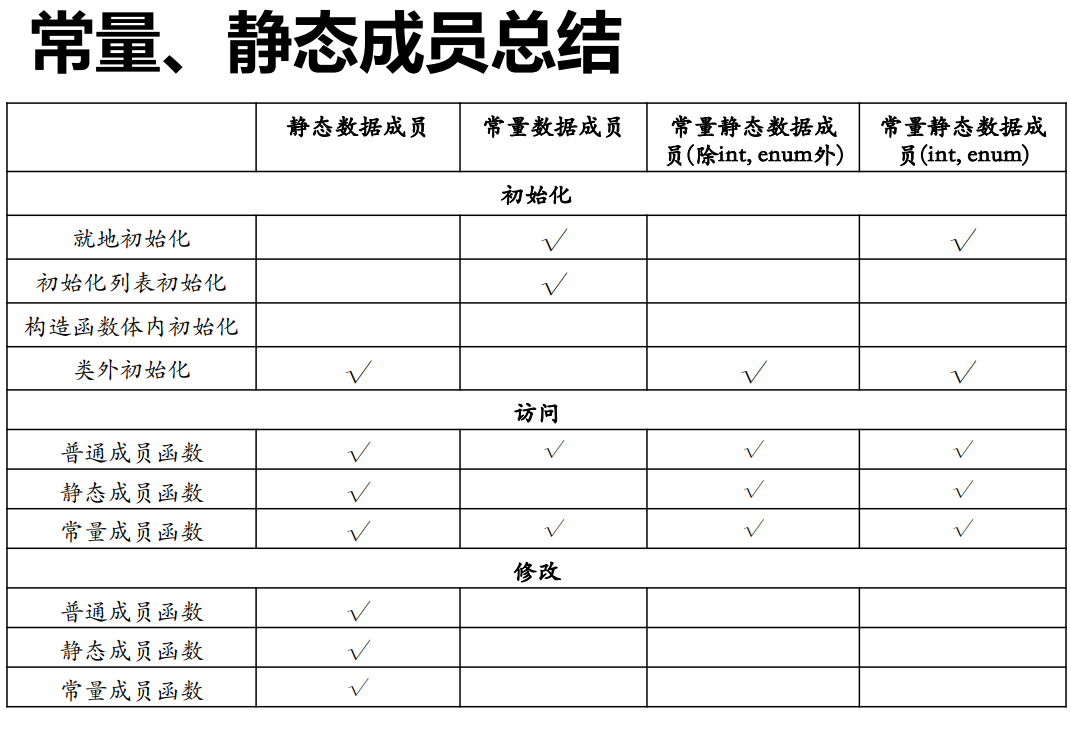
常量数据成员：使用const修饰的数据成员，称为类的常量数据成员，在对象整个生命周期内不可更改。常量数据成员可以在初始化列表中初始化，就地初始化，但是不允许在构造函数的函数体中通过赋值来设置

常量成员函数：成员函数被const修饰，其访问权限为，实现语句不能修改类的数据成员，即不能改变对象的状态（注意格式为 ReturnType Func(…)const{…}，要和另外的格式来区别 const ReturnType Func(…){…}，这是返回值修饰了const）

若对象被定义为常量 const ClassName a; 那么其只能调用const修饰的成员函数，也就是相当于对象其中的数据不能变

常量静态变量：既是常量又是静态的变量，需要在类外定义，和静态变量一样，但是int和enum类型可以就地初始化，满足访问权限的任意函数均可以访问，但不能修改

不存在常量静态函数：静态函数隶属于类，可以不实例化，而对于常量修饰函数则必须绑定在对象上，因为未知对象是否为常量对象，所以二者相互冲突



常量全局/局部对象的构造与析构时机和普通全 局/局部对象相同

静态全局对象的构造与析构时机和普通全局对象相同

局部静态对象：在程序执行到该静态局部对象时，其被初始化，离开作用域不析构，再次执行到该对象的代码时不再初始化，而是直接使用上一次的对象，在main函数结束后析构

类静态对象的构造与析构：类静态对象是指类A中的对象a作为类B的静态变量，那么a的构造和析构与全局对象类似，在main调用前初始化，在main完成后对象被析构，与B是否实例化无关

参数对象的析构与构造：①如果参数是对象的引用，则不存在初始化和析构②如果类含有指针成员，那么指向同一个地址的释放内存很可能导致错误

对象的new和delete： A \*pA =new A(一些参数);delete pA;

**引用与复制：**

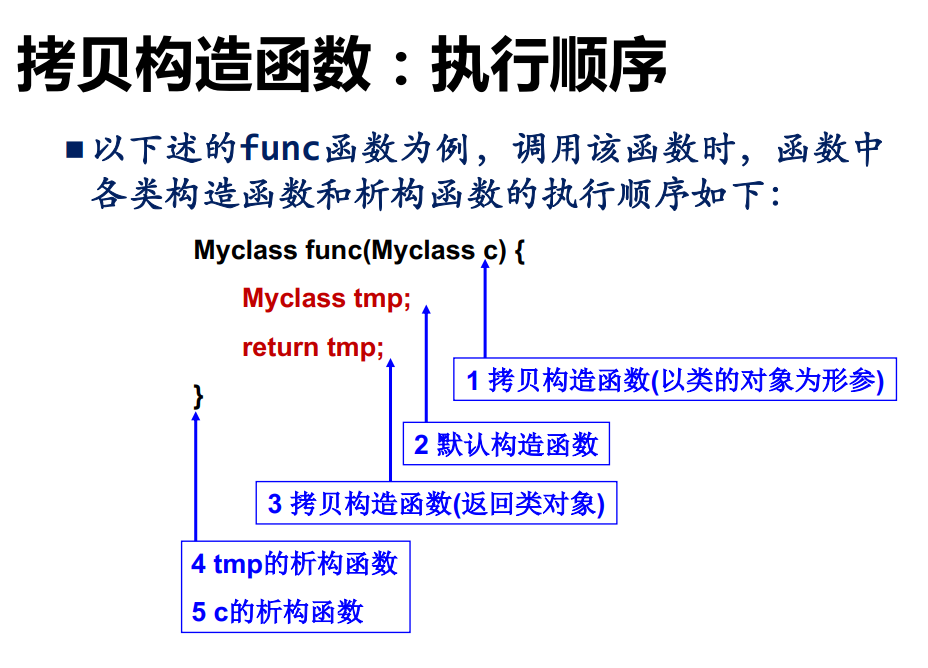
拷贝构造函数：特殊的构造函数，参数是同类对象的常量引用（Test a;Test b(a);Test c=a;）

函数调用时以类的对象为形参，而函数返回类对象

编译器会自动生成拷贝构造函数

类的新对象被定义后，会调用构造函数或拷贝构造函数，如果调用拷贝构造且没有显式定义，那么会自动合成，会调用所有数据成员的拷贝构造函数或者拷贝赋值运算符（默认拷贝方式为位拷贝，遇到指针型成员可能会出错，会导致多个指针类型的变量指向同一地址）

拷贝构造函数的执行顺序：



拷贝构造函数使得程序效率降低，所以①往往使用引用、常量引用传递参数或者返回对象②将拷贝构造函数声明为private②使用delete使得编译器不生成拷贝构造函数的隐式定义

func(Myclass a)改为func(const Myclass& a)

Myclass func(…)改为Myclass& func(…)

右值引用：

左值：可以取地址，有名字的值

右值：不能取地址，没有名字的值，常见为常值、函数返回值、表达式

右值无法取地址，但是可以利用右值引用（int &&e=a+b;），右值引用不能绑定左值

常量左值引用也可以绑定右值



（右值引用为左值）

对于函数，如果没有定义右值引用的版本，那么可能会出错

移动构造函数：

右值引用可以延续即将销毁的变量的周期，从而减少拷贝，使用右值引用作为参数的构造函数叫做移动构造函数

拷贝构造函数：ClassName(const ClassName& VariableName)

移动构造函数：ClassName(ClassName&& VariableName);

移动构造函数直接利用了原来临时对象中的堆内存，新对象无需开辟内存，临时对象无需释放内存，从而提高效率

移动构造函数加快了利用右值初始化的速度，为了对左值使用同样的移动构造，使用std::move函数（Test a; Test b = std::move(a);）

Move不对对象进行任何操作，仅仅是类型转化，即转化为右值

拷贝和移动构造函数的调用时机：

判定依据：引用的绑定规则

拷贝构造函数形参类型为常量左值引用，可以绑定常量左值、左值和右值

移动构造函数形参类型为右值引用，只可以绑定右值

引用绑定存在优先级，当传入实参为右值时优先匹配形参类型为右值引用的函数

拷贝构造函数的调用时机：

1. 使用一个类对象、引用、常量初始化另一个新对象
2. 以类的对象为函数参数，传入实参为类的对象、引用、常量引用
3. 函数返回类对象（没有显式定义移动构造函数，不进行返回值优化）

移动构造函数的常见调用时机：

1. 使用一个类对象的右值初始化另一个新对象
2. 以类的对象为函数形参，同时传入实参为类对象的右值
3. 函数返回类对象（类中显式定义移动构造函数，不进行返回值优化，如return Test();return tmp;）

拷贝赋值运算符：

已定义的对象之间的赋值可以使用对象的拷贝赋值运算符

（ClassName& operator= (const ClassName& right) {

if (this != &right)

{// 避免自⼰赋值给自⼰ // 将right对象中的内容拷贝到当前对象中...

}

return \*this; }）

移动赋值运算符：

和移动构造函数原理类似

Test& operator= (Test&& right) {

if (this == &right) cout << "same obj!\n";

else {

this->buf = right.buf; //直接赋值地址

right.buf = nullptr; cout << "operator=(Test&&) called.\n";

}

return \*this; }

拷贝和移动赋值运算符的调用时机和拷贝、移动赋值构造函数的调用时机类似，主要判断依据时引用的绑定规则（P.45）

类型转化：

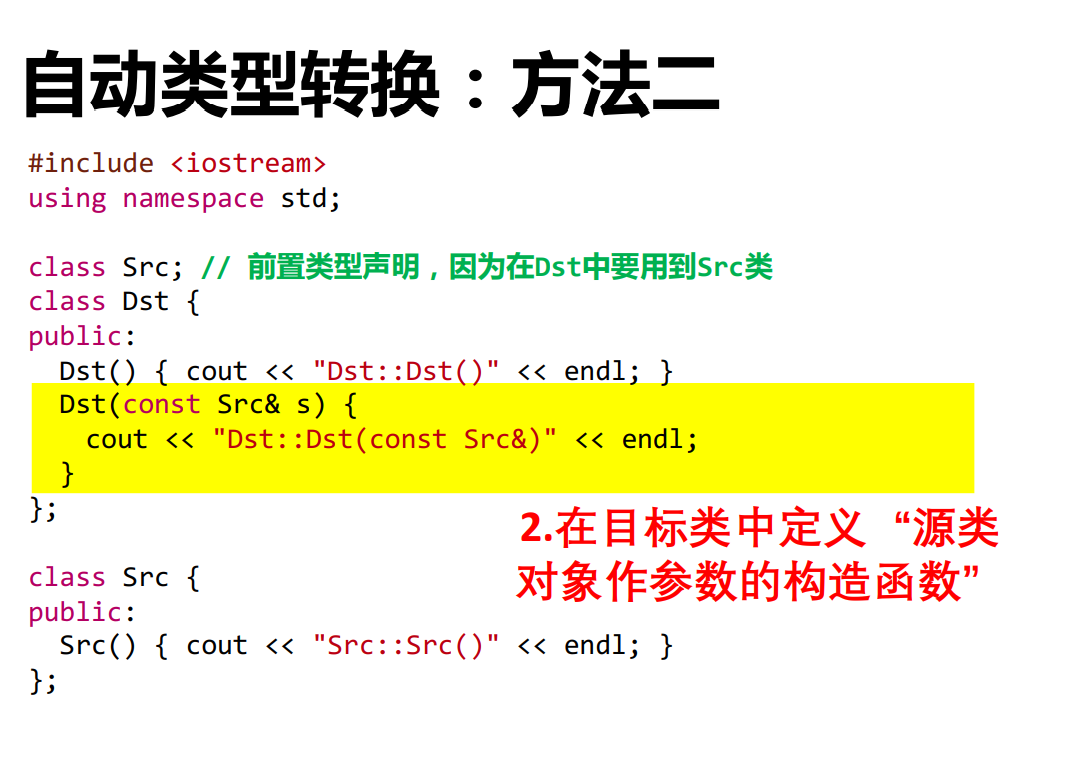
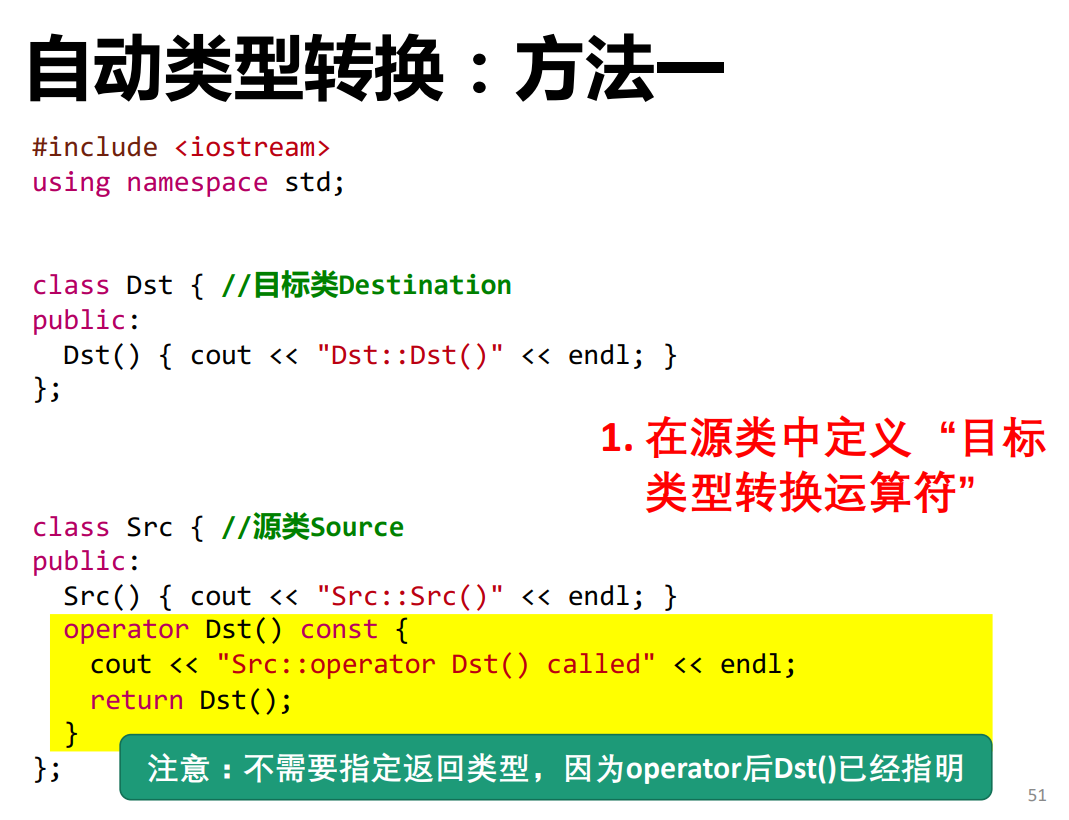
编译器发现表达式和函数调用所需类型不同时，会进行自动类型转化

自动类型转化可以通过定义特定的转换运算符和构造函数完成

有时还可以进行强制类型转化

方法：

1. 在源类中定义目标类型转化运算符
2. 在目标类中定义源类对象作为参数的构造函数



禁止自动类型转换：

使用explicit修饰类型转换运算符或者类型转换构造函数，则相应的类型转换必须显式进行

强制类型转换：

const\_cast，去除类型的const或者volatile属性

static\_cast，无条件转换，转换为静态类型

dynamic\_cast，动态类型转换，如派⽣类和基类 之间的多态类型转换。

reinterpret\_cast，仅仅重新解释类型，但没有 进⾏⼆进制的转换

对象组合：①已有类的对象作为新类的公有数据成员，可以直接访问②已有类的对象作为新类的私有数据对象，新类可以调整旧类的对外接口（相当于转换接口）

对象构造与析构函数的次序：先完成子对象的构造，再完成当前对象的构造，子对象的构造次序由类中的声明顺序决定，析构次序与构造次序相反

对象继承：

被继承的已有类为基类，或父类

继承得到的新类为派生类，或子类或扩展类

class Derived: [private] Base {…}

class Derived:public Base{…}

class Derived:protected Base{…}

不能被继承的：①构造函数：创建派生类对象，必须调用派生类的构造函数，派生类构造函数调用基类的构造函数，以创建基类部分，如果没有显式调用，则会自动调用，一般默认不继承，但是可以使用using继承②析构函数：析构时先调用派生类析构函数再调用基类析构函数③赋值运算符：编译器不会继承基类的赋值运算符（因为参数为基类），但会自动合成隐式的赋值运算符④友元函数：因为不是类成员

在派生类的构造中，如果没有显式调用基类的默认构造函数，那么会自动调用，如果需要显式调用，那么只能在派生类的构造函数的初始化成员列表中进行，既可以调用基类的默认构造函数，也可以调用带参数的构造函数

先执行基类的构造函数初始化继承来的数据，再执行派生类的构造函数

对象析构时先执行派生类的析构函数，再执行编译器自动调用的基类析构函数

using Base::Base;//这样来声明默认调用一系列基类构造函数来构造自身

如果基类中某个构造函数被声明为私有成员函数，那么不能在派生类中声明继承该构造函数

如果派生类使用了继承构造函数，那么编译器就不会为派生类再生成隐式定义的默认构造函数

Public继承的派生类，其基类中公有成员仍能在派生类中保持公有

Private继承的派生类，基类的公有部分也变成了私有

无论何种继承，基类中的私有成员，不允许在派生类成员函数中访问，更无法用对象访问！

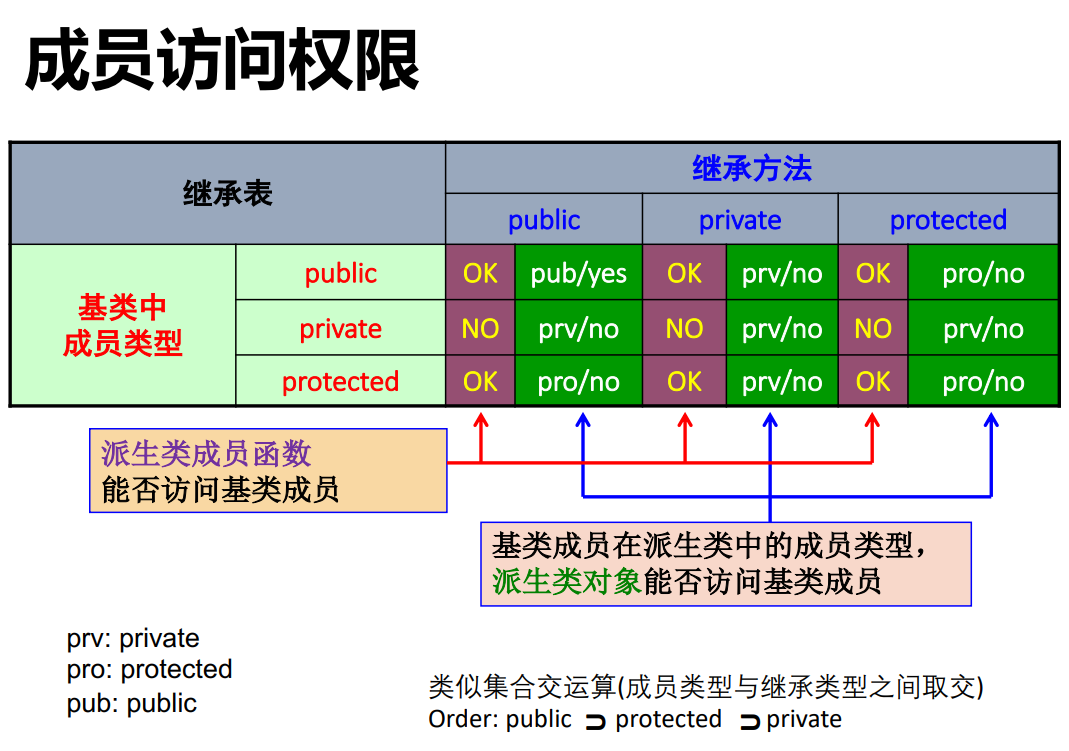
基类中的公有成员允许在派生类成员函数中访问，如果为public继承，则可以被对象访问；如果为private或者protected继承，则无法被派生类对象访问，此时若想让某个基类公有成员能被派生类对象访问，可在派生类public部分使用using关键词声明

对于基类中的保护成员，允许在派生类成员函数中访问，但不允许被外界函数访问

对于public继承，基类的成员保持原来状态

对于private继承，基类的所有成员称为私有成员

对于protected继承，基类的公有成员和保护成员称为保护成员，基类的私有成员仍然是私有的



重写隐藏与重载：

重载：提供同名函数的不同实现，属于静态多态。要求函数名必须相同，函数参数必须不同，作用域相同，比如在同一个类或者同名全局函数

重写隐藏：在派生类中重新定义基类函数，实现其特殊功能。其屏蔽了基类的所有其他同名函数，函数名必须相同，参数无要求

重写隐藏发生时，基类中其他任何同名重载函数都被屏蔽，不能提供给派生类对象，如果要恢复指定的基类成员函数，即去除屏蔽，则使用using Derived::func; 但是完全一致（也就是参数也一致）的函数还是优先调用派生类重写的

Using关键词还可用于①指示命名空间（using namespace std;）②将另一个命名空间的成员引入当前命名空间（using std::cout;）③定义类型别名（using a=int;）

多重继承：

class IOFile: public InputFile, public OutputFile{};

存在问题：①数据存储，如果继承的两个类继承了同一个类，那么可能带来数据冗余②二义性，如果基类中含有同名成员a

**虚函数：**

向上类型转换：

派生类对象、引用、指针转化为基类的对象、引用、指针，就是向上类型转换，只对public继承有效，对private和protected无效，向上类型转化可以由编译器自动完成，凡是可以接受基类的对象、引用、指针的地方都可以使用派生类对象、引用、指针

对象切片：

当派生类的对象（不是指针或者引用）被转换为基类对象时，其被切片为基类的子对象而出现数据丢失

指针（引用）的向上转换：

当派生类的指针（引用）被转换为基类指针（引用）时，不会创建新的对象，但是只会保留基类的接口

函数调用捆绑：

把函数体与函数调用相联系称为捆绑

当捆绑发生在程序运行之前（由编译器和连接器完成）时，称为早捆绑

当捆绑会根据对象的实际类型，发生在程序运行时，称为晚捆绑，或动态捆绑、运行时捆绑

虚函数：

若一个函数在基类中被声明为虚函数，即在开头加上virtual关键词（virtual Returntype Funcname(参数);），那么再在派生类中重新定义后，基类指针或引用调用该成员函数时会根据实际类型决定调用基类函数还是派生类重写函数

若某成员函数在基类中声明为虚函数，那么派生类中重写覆盖（同名，同参数）时，无论是否声明为虚函数，都默认为虚函数

这是晚绑定，但是只对指针和引用有效

虚函数表：

运行时通过虚函数表确定对象的实际类型

定义为每个包含虚函数表的类用于存储虚函数地址的表

每个包含虚函数的类中，编译器秘密地放一个虚函数指针（VPTR），指向该类的虚函数表（VTABLE）

当通过基类指针调用虚函数时，编译器静态地插⼊能取 得这个VPTR并在VTABLE表中查找函数地址的代码，这样 就能调用正确的函数并引起晚捆绑的发⽣。

编译期间建立虚函数表，记录每个类中已声明虚函数的入口地址

运行期间建立虚函数指针，在构造函数中发生，指向相应的虚函数表

虚函数与构造函数：

构造函数不能也不必是虚函数P.24

在构造函数中调用一个虚构函数，则调用的是本地版本，也就是虚函数机制不工作

虚函数与析构函数：

析构函数能是虚函数，且常常为虚函数，虚析构函数仍需要函数体

这样当删除基类对象指针时，编译器 将根据指针所指对象的实际类型，调用相应的析构函 数，否则可能造成内存泄漏

在析构函数中调用虚函数，也是调用的本地版本

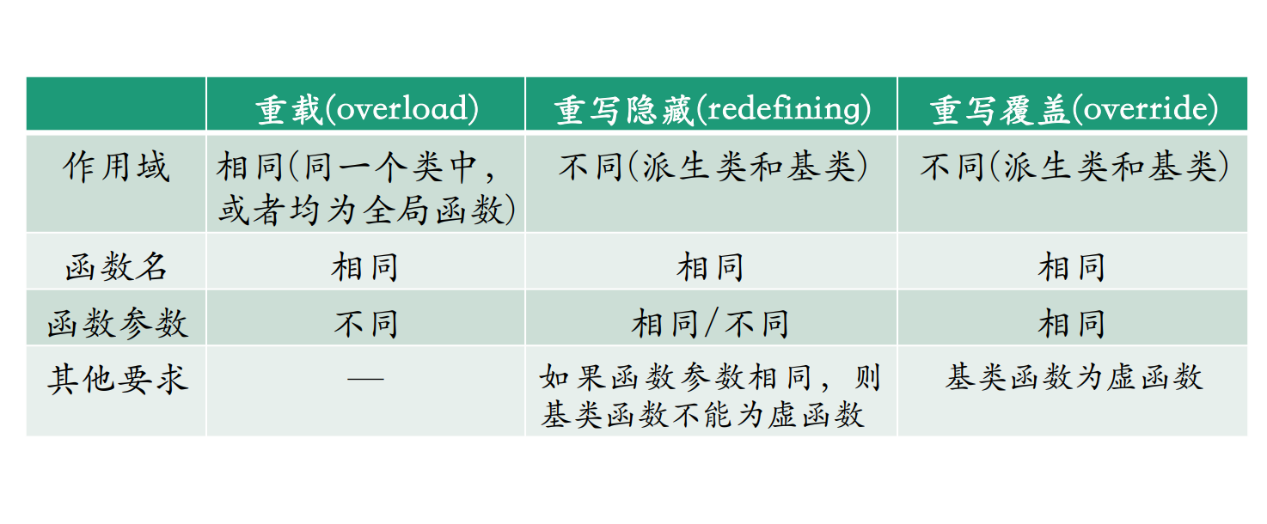
重载、重写覆盖、重写隐藏：

重载：函数名相同，参数必须不同，作用域相同

重写覆盖：派生类重新定义基类中的虚函数，函数名和函数参数必须相同，返回值一般应该相同，那么原来的虚函数指针会被新的虚函数指针覆盖

重写隐藏：派生类中重新定义基类中的函数，函数名相同，参数不同或者基类中的函数不是虚函数，重写隐藏中虚函数表不会发生覆盖

重写覆盖和重写隐藏都会覆盖基类中的同名函数



Override关键字：

override关键字明确地告诉编译器⼀个函数是对 基类中⼀个虚函数的重写覆盖，编译器将对重写 覆盖要满⾜的条件进⾏检查，正确的重写覆盖才 能通过编译。

void foo(int ) override {cout<<"Derived3::foo(int )"<<endl;};

final关键字：

为了不让使用者继承，使用final

在虚函数声明或定义中使用时，final确保函数为虚 且不可被派⽣类重写。可在继承关系链的“中途”进⾏ 设定，禁⽌后续派⽣类对指定虚函数重写。

void foo() final {};

在类定义中使用时，final指定此类不可被继承。

**多态与模板：**

纯虚函数：

虚函数可以进一步声明为纯虚函数，包含纯虚函数的类，被称为抽象类

virtual 返回类型 函数名(形式参数) = 0;

抽象类不允许定义对象，主要是提供接口，能够避免对象切片

抽象类至少含有一个纯虚函数

基类纯虚函数被派生类重写覆盖之前仍然时纯虚函数，因此继承一个抽象类，除了纯虚析构函数以外，必须实现所有纯虚函数，否则继承类也是抽象类

纯虚析构函数：

纯虚析构函数仍然需要函数体，目的是使基类称为抽象类，不能创建其对象

对于纯虚析构函数而言，即便派生类中不显式实现，编译 器也会自动合成默认析构函数。因此，即使派生类不显式 覆盖纯虚析构函数，只要派生类覆盖了其他纯虚函数，该 派生类就不是抽象类，可以定义派生类对象。

向下类型转化：基类指针、引用转换成派生类指针、引用

使用基类指针表示派生类时失去了特性，所以再次转化为派生类指针，但是需要保证转化的正确性，利用虚函数表进行动态类型检查

dynamic\_cast：

使用dynamic\_cast的对象必须有虚函数，因为它使用 了存储在虚函数表中的信息判断实际的类型。

obj\_p，obj\_r分别是T1类型的指针和引用

T2\* pObj = dynamic\_cast(obj\_p); //转换为T2指针，运行时失败返回nullptr

T2& refObj = dynamic\_cast<T2&>(obj\_r); //转换为T2引用，运行时失败抛出bad\_cast异常

（在向下转换中，T1必须是多态类型（声明或继承了至 少一个虚函数的类））

也可以使用static\_cast

static\_cast在编译时静态浏览类层次，只检查继承关系。

obj\_p，obj\_r分别是T1类型的指针和引用

T2\* pObj = static\_cast(obj\_p); //转换为T2指针

T2& refObj = static\_cast<T2&>(obj\_r); //转换为T2引用

多重继承中的虚函数：

最多继承一个非抽象类

可以继承多个抽象类（接口）

避免 多重继承的二义性

利用 一个对象可以实现多个接口

多态：按照基类的接口定义，调用指针或引用所指对象 的接口函数，函数执行过程因对象实际所属派生 类的不同而呈现不同的效果（表现），这个现象 被称为“多态”。

产生多态效果的条件：继承 && 虚函数 && (引 用 或 指针)

函数模板和类模板：

有些算法实现与类型无关，所以可以将函数的参 数类型也定义为一种特殊的“参数”，这样就得 到了“函数模板”。

template <typename T> ReturnType Func(Args)；

函数模板在调用时，编译器能自动推导出实际参 数的类型（这个过程叫做实例化）。

在使用时也可以手动指定调用类型：Func<T>(…);

类模板：

template <typename T> class A {};

在利用类定义对象时：A<int> a(1);

类模板的成员函数的类外定义：

template <typename T>

void A<T>::print(){};

类模板的模板参数：

1. 类型参数：使用typename或class标记
2. 非类型参数：整数，枚举，指针（指向对象或函数）， 引用（引用对象或引用函数）。

（非类型参数的例子：

template<typename T , unsigned int>

class array {

T elems[size];

...

};

这样可以定义对象：

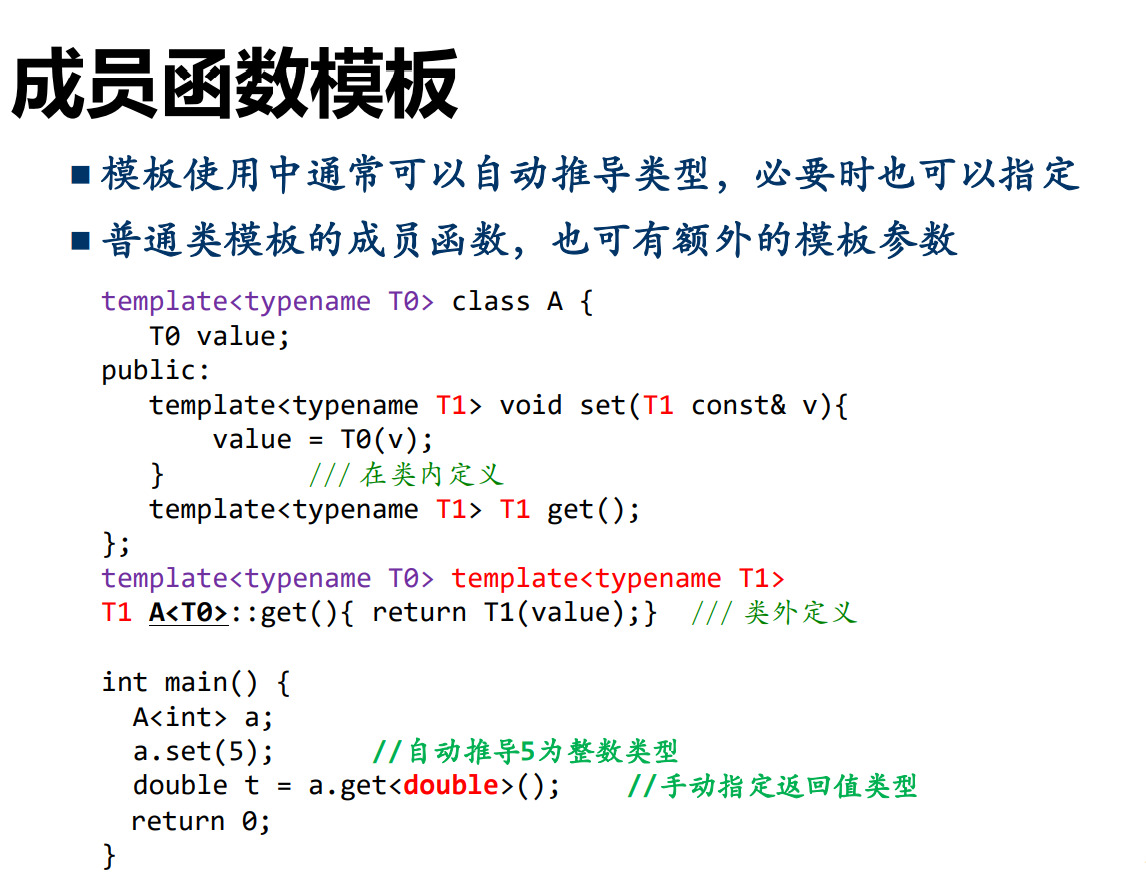
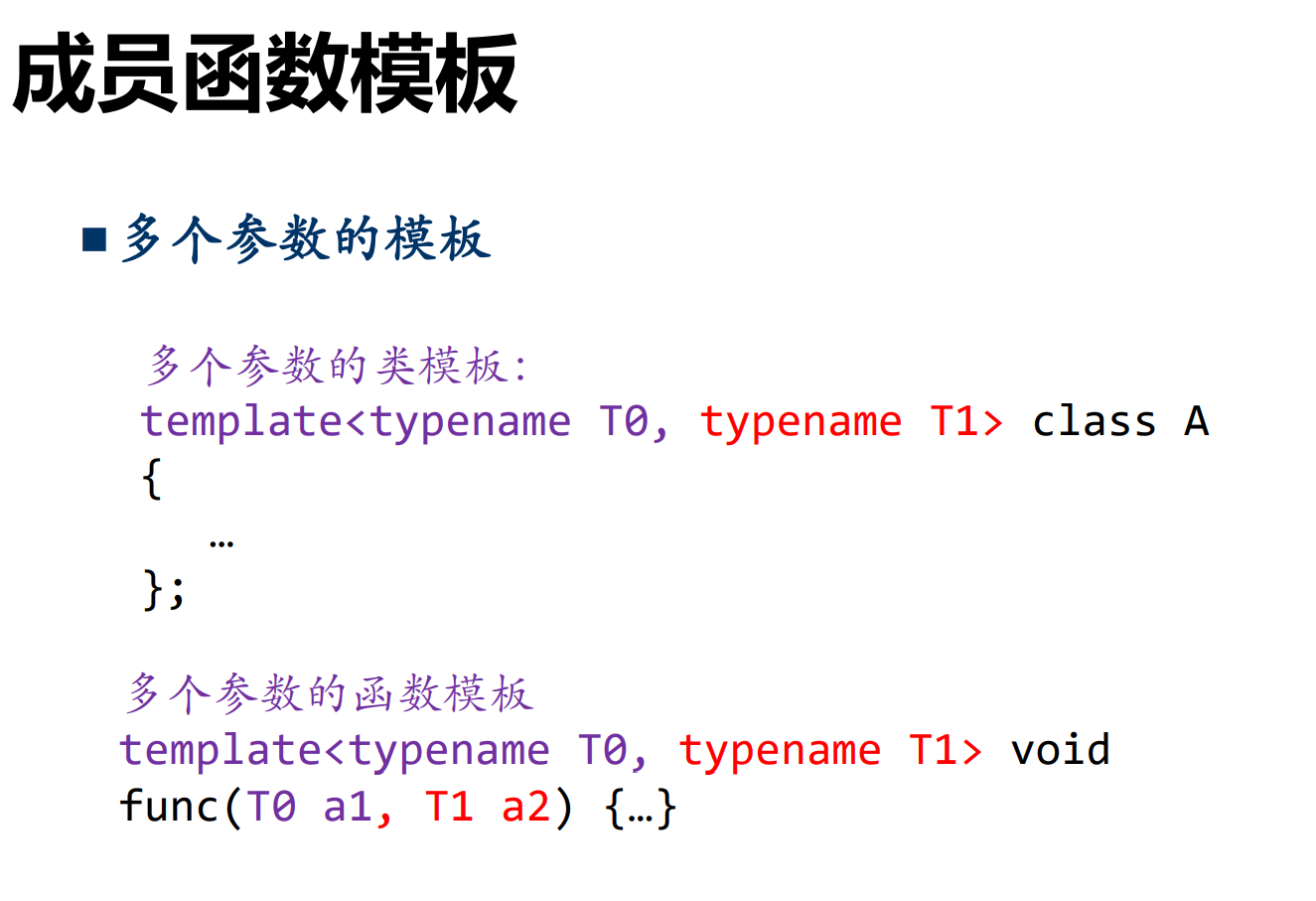
Array<char ,10> array0;

）

成员函数模板：

普通类的成员函数也可以定义为模板函数





对模板的处理是在编译期进行的，每当编译器发 现对模板的一种参数的使用，就生成对应参数的 一份代码。

这意味着所有模板参数必须在编译期确定，不可 以使用变量

int n = 5; myClass a; //错误

const int n = 5; myClass b; //正确

模板库必须在头文件中实现，不 可以分开编译

**模板与STL：**

命名空间：

定义命名空间：

namespace A {

int x,y;

}

使用时using namespace A;

STL：标准模板库

STL容器（数据结构）包括：简单容器、序列容器、关系容器

1. pair

由两个单独数据组成

std::pair<int , int> t;

t.first=4;

t.second=5;

创建也可以利用函数make\_pair

auto t = std::make\_pair(“abc”, 7.8);

支持小于、等于等比较运算符。

先比较first再比较second

1. tuple

pair的扩展，由若干成员组成的元组

创建：auto t = std::make\_tuple(“abc”, 7.8, 123, ‘3’);

tie函数—返回左值引用的元组：std::tie(x, y, z) = std::make\_tuple(“abc”, 7.8, 123); //等价于 x = "abc"; y = 7.8; z = 123

可以通过v0 = std::get<3>(tuple1);调取其中的第3个元素，但是其中不能为变量

1. vector

会自动扩展容量的数组，以循序的方式维护变量集合

允许直接以下标访问

创建：std::vector<int> x;

当前数组长度：x.size();

清空：x.clear();

在末尾添加：x.push\_back(1);

在末尾删除：x.pop.back();

使用迭代器在中间添加或删除：x.insert(x.begin()+1,5);

x.erase(x.begin()+1);

迭代器：

提供⼀种⽅法顺序访问⼀个聚合对象中各个元素, ⽽又不需暴露该对象的内部表示。

为遍历不同的聚合结构（需拥有相同的基类）提 供⼀个统⼀的接⼝

使用上类似指针。

vector::iterator iter;\\ 定义了⼀个名为iter的变量，它的数据类型是由 vector定义的iterator类型。

x.begin()，返回vector中第⼀个 元素的迭代器。

x.end()，返回vector中最后⼀个元 素之后的位置的迭代器

下⼀个元素：++iter

上⼀个元素：--iter

下n个元素：iter += n

上n个元素：iter -= n

访问元素值——解引用运算符 \*：\*iter = 5;

解引用运算符返回的是左值引用

元素位置差为迭代器相减

当迭代器不再指向本应指向的元素时，称此迭代 器失效：

1. 调用insert/erase后，所修改位置之后的所有迭代器 失效。（原先的内存空间存储的元素被改变）
2. 调用push\_back等修改vector⼤小的⽅法时，可能会 使所有迭代器失效

对于vector，如果size达到了capacity，则另申请⼀片 capacity\*2的空间，并整体迁移vector内容，整体迁移过程使所有迭代器失效

1. 链表容器list

定义：std::list<int> l;

插入前端：l.push\_front(1);

插入末端：l.push\_back(2);

查询：std::find(l.begin(), l.end(), 2); //返回迭代器

插入指定位置：l.insert(it, 4); //it为迭代器

不支持下标等随机访问

支持高速在任意位置插入、删除数据

访问主要依赖迭代器

插入和删除操作不会导致迭代器失效

1. 无序集合set

不重复元素构成的无序集合

内部按照大小顺序排列

无序是指不保持插入顺序

定义：std::set<int> s;

插入：s.insert(val);（不允许重复）

查询元素：s.insert(val);\\返回迭代器

删除：s.erase(s.find(val));

统计（检验是否存在）：s.count(val);\\结果是0或1

1. 关联数组map

每个元素由两个数据项组成，map将一个数据项映射到另一个数据项中

其值类型为pair<key,T>

Map中元素的key必须互不相同

可以通过下标访问（即使key不是整数）。下标访 问时如果元素不存在，则创建对应元素。

定义：std::map<std::string , int> s;

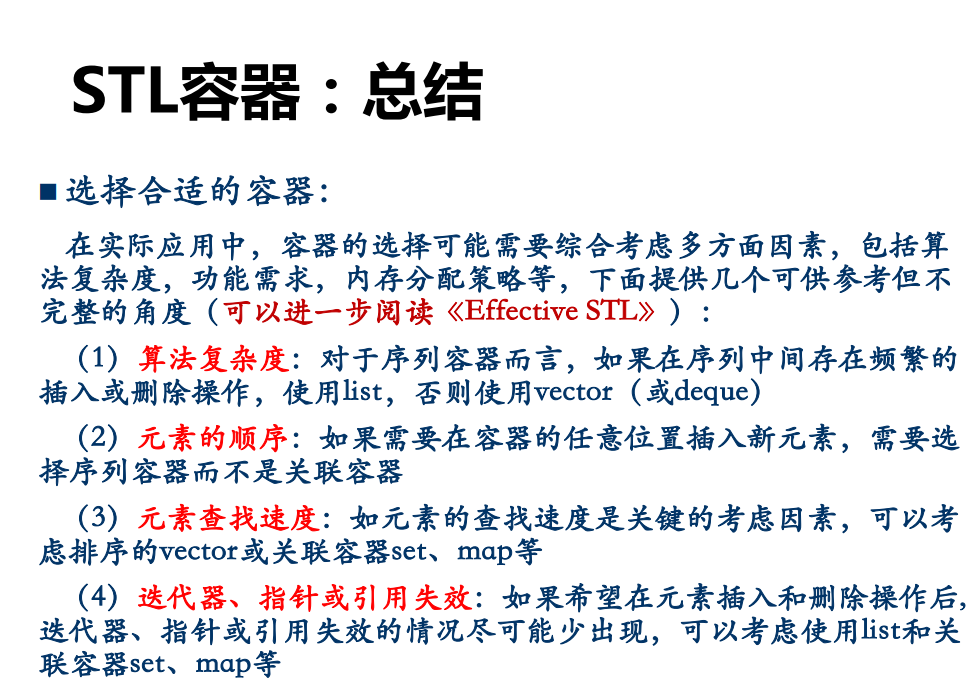
插入：s.insert(std::make\_pair(std::string("Tuesday"), 2));

查询：s.find(key);//返回迭代器（键为key）

统计（看是否存在）：s.count(key); // 返回0或1

删除：s.erase(s.find(key)); //导致被删元素的迭代器失效

map常用作稀疏数组或以字符串为下标的数组。



函数模板与类模板特化：P.63

L12：函数对象和智能指针

字符串是char的数组

STL提供了string类型

String几种构造方式：

1. string s0(“inishdah”);（等价于 string s0=“sahskdjcn”;）
2. string s1;
3. string s2(s0,8,3);(从8开始，截取长度为3)
4. string s3(“sacjhiaj”,5);(截取前5个字符)
5. string s4(10,’x’);(将’x’字符复制10遍)
6. string s5(s0.begin(),s0.begin()+7);(从开头开始截取长度为7的字符串)

可以将string转化为字符串：string str;const char\* p=str.c\_str();（返回值为常量字符指针const char\*）

String类型的变量可以使用下标访问如str[1]

查询长度：str.size()（也可以使用str.length()）

清空：str.clear()

查询是否为零：str.empty()

迭代访问：for(char c : str)

向尾部增加：str.push\_back(‘a’);或者str.append(s2);（也可以使用str+=’a’或者str+=s2）

三种输入方式：

1. cin>>firstname;（读取可见字符直到遇到空格）
2. getline(cin,fullname);（读取一整行）
3. getline(cin,fullname,’#’);（读到指定分隔符’#’位置，可以读入换行符）

拼接：string s2=s0+” ”+s1;

比较：直接按照字典序比较字符串大小（string a=”alice”;string b=”bob”;a==b;a<b;）前面判别式为假，后面为真

数值类型向字符串转化：to\_string (1);

字符串类型向数值转化：

stoi(“2001”);

stoi(“shcsda”,&sz);（sz为读入长度）

int c = stoi("40c3", nullptr, 16) //c=0x40c3 十六进制

int d = stoi("0x7f", nullptr, 0) //d=0x7f 自动检查进制

stod(“34.5”);

ostream& operator<<(ostream& out, const Test& src) {

out << src.id << endl;

return out;

}

Ostream为STL库中所有输出流的基类。重载了针对基础类型的输出流运算符(<<)，统一了输出接口，会将数据送到标准输出流（屏幕）

如何格式化输出 – #include

cout << fixed << 2018.0 << " " << 0.0001 << endl; //浮点数 -> 2018.000000 0.000100

cout << scientific << 2018.0 << " " << 0.0001 << endl; //科学计数法 -> 2.018000e+03 1.000000e-04

cout << defaultfloat; //还原默认输出格式

cout << setprecision(2) << 3.1415926 << endl; //输出精度设置为2 -> 3.2

cout << oct << 12 << " " << hex << 12 << endl; //八进制输出 -> 14 十六进制输出 ->c

cout << dec; //还原十进制

cout << setw(3) << setfill('\*') << 5 << endl; //设置对齐长度为3，对齐字符为\* -> \*\*5

endl是一个函数 • 等同于输出'\n'，再清空缓冲区 os.flush()。可以调用 endl(cout);

返回重载流运算符的引用是为了防止赋值

Ostream的赋值构造函数被delete了，禁止复制，只可以移动，所以只有cout一个全局变量

文件输入输出流：

ifstream是istream的子类，功能是从文件中读入数据

P.29

外设：IO、中断、时钟、通讯（UART，SPI，I2C）、ADC