目录

[零、本笔记的书写习惯 1](#_Toc30658658)

[一、基本概念 1](#_Toc30658659)

[二、信息隐蔽的具体实现方法 2](#_Toc30658660)

[三、类和对象的相关知识点 2](#_Toc30658661)

[四、运算符重载 5](#_Toc30658662)

[五、数据转换 6](#_Toc30658663)

[六、继承与派生 6](#_Toc30658664)

[七、多态性（polymorphism）与虚函数 8](#_Toc30658665)

[八、标准输入输出流 10](#_Toc30658666)

[九、文件操作与文件流 13](#_Toc30658667)

[十、字符串流（内存流） 15](#_Toc30658668)

[十一、异常处理 15](#_Toc30658669)

[十二、命名空间 17](#_Toc30658670)

# 零、本笔记的书写习惯

1.橙色代表不确定的个人推论

2.红色代表可能在新标准中更改的内容

3.绿色代表批注与评论

4.蓝色代表关键词keywords，指令中其他首字母大写的是具体编辑内容，如

const DataType DataName

5.绛色代表代码

# 一、基本概念

DJGPP：是GCC在DOS/Windows平台上的实现，DJ是移植者姓名缩写，GPP即GCC++的意思

GCC：最早是GNU-C-Compiler的缩写，现在是GNU-Composed-Compiler（GNU编译器集群）的缩写；GNU是一个没有实际意义的自缩略语，来源于Not UNIX，即是开源的编译器，最著名的代表项目是Linux

RHIDE：是DJGPP的集成软件开发环境（IDE）；RH是作者名称缩写；即提供使用界面用于输入编辑，调试运行的程序

抽象（abstraction）：类（class）是对象的抽象，对象则是类的特例。

对象（object）：一个对象往往由一组属性（attribute）和一组行为（behavior）或方法（method）构成。在C++中，每个对象都是由数据和函数（即操作代码）这两部分组成的。

多态性（polymorphism）：由继承而产生的相关的不同的类，其对象对同一消息会做出不同的响应。多态性是面向对象程序设计的一个重要特征，能增加程序的灵活性。

方法（Method）：类的成员函数在面向对象程序理论中的称谓。

封装性（encapsulation）：面向对象程序设计方法的重要特点。一是将有关的数据和操作代码封装在一个对象中，形成一个基本单位，各个对象之间相互独立，互不干扰。二是将对象中某些部分对外隐蔽，即隐蔽其内部细节，只留下少量接口，以便与外界联系，接受外界的消息。

公用接口（public interface）：即类的公用成员函数，或者称对外接口。【C#笔记中对于接口有新的理解】

宏：将一个指定的标识符来代表一个字符串，这个标识符就是宏名

继承与重用（reusability）：在原有的类的基础上，增加一些属性和行为，建立一个新的类。

面向对象的程序设计：对象=算法+数据结构；程序=（对象+对象+……）+消息，或程序=对象s+消息。

面向对象的软件工程：1）面向对象分析（object oriented analysis，OOA）；2）面向对象设计（object oriented design，OOD）；3）面向对象编程（object oriented programming，OOP）；4）面向对象测试（object oriented test，OOT）；5.面向对象维护（object oriented soft maintenance，OOSM）。

面向过程的结构化程序设计：程序=算法+数据结构

私有实现（private implementation）：类中被操作的数据是私有的，实现的细节对用户是隐蔽的。

信息隐蔽：类的公用接口与私有实现的分离。

# 二、信息隐蔽的具体实现方法

将类的声明放在头文件中，用户可以通过包含头文件来引用这个类以及知晓类的成员函数的原型。将类的成员函数的定义放在另一个源文件中（不是包含主函数的文件），单独编译之后形成目标文件。每次使用时不需要再次编译这个文件，而且这个文件也可以连同头文件提供给其他用户而不会被擅自更改和挪用。

# 三、类和对象的相关知识点

1. 用构造函数（constructor）初始化对象p263

一般的成员函数定义法或参数初始化表

•参数初始化表的扩展形式用于派生类的构造，这里的数据成员可以看成是基类，用法 相似，可参见派生类的构造函数部分

Class::Class(DataType1 DataName1, …): DataMember1(DataName1), …{}

a.生成对象的时刻自动执行构造函数且对该对象只执行一次，而非由用户决定或者调用

b.构造函数没有返回值，没有函数类型，并非是void类

c.构造函数的重载和有默认参数的构造函数，二者一般不同时使用。这个不绝对，关键只是不要让编译器不知道调用哪个函数

c1.默认default构造函数是指不需要参数来进行调用的构造函数，可能是没有形参，或全为默认参数。

c2.若用户没有重载或定义构造函数，那么编译器往往自动生成一个无参数的默认构造函数，其会调用基类的默认构造函数以及所有非静态成员的默认构造函数。实际上，在多数情境下（特例见后文）其功能和用户定义的空构造函数（空函数体和空初始化列表）是一样的。如果用户重载了构造函数，那么编译器一般不会再自动定义默认构造函数。

c3.以下是一些C++11中关于默认构造函数的一些扩展用法，这些用法虽非特殊功能，但都是配合编译器让代码纠错功能更加强大。

class\_name() =delete; //删除默认构造函数，若调用则报错

class\_name() =default; //启用编译器定义的默认构造函数，即便已有用户定义的版本

class\_name()::class\_name() =default; //在类体外启用编译器定义的默认构造函数

c4.上面的类体外默认构造函数虽然是指定编译器生成的，但是在某些情境下仍被视为用户定义的，如在值初始化情境下value initialization。在该情境的某个特殊情况下，编译器生成的构造函数和用户定义的会使得初始化结果不一样，不过这并不常见。详情需要参考C++的各种initialization的定义和区别，然而这种过度的复杂情况只在用户不自行维护对象初始化并依靠编译器自身实现时才会发生。

d.给构造函数提供参数，对于对象数组和普通对象而言，指令不同

Class Object(X,Y,Z…) //声明普通对象

Class Object[…]={Class(X,Y,Z…),Class(…),Class(…)…} //声明对象数组

d1.按照Class Object()方式调用默认（无参数或全部默认参数）的构造函数时，一般不加空括号()，这是因为会产生本语句用以声明函数的歧义性（消歧义方式因编译器而异）。当然，使用Class Object = Class()或使用新标准的括号Class Object{}都可以。

d2.定义对象数组的例子中，直接调用构造函数原型而不加对象名，可以建立一个无名对象；此例中用作每个元素的赋值，也可以作为它用，如计算和数据转换

e.构造函数有很多类型和功用，比如一般构造函数，默认构造函数，复制构造函数（形如“类名 对象2（对象1）”，编译器自动定义其最简单的形式，也可自行定义）和转换构造函数（见下“数据转换”部分），但是这些函数的原型都是类似的，机理也是类似的，只是用处和语义不同。当一个类存在单参数构造函数时，可使用explicit关键词修饰为显式构造函数，即指定其不可以用作（隐式）转换构造函数。然而对于无参数的构造函数，其本身就无法成为转换构造函数或复制构造函数。

如explicit ClassName(…) {}

2. 析构函数（destructor）结束对象生命周期

a.析构函数没有返回值，没有函数类型，没有参数，即也不能被重载

b.构造与析构函数都应放在public下

c.对同一类存储类别的对象，先构造的后析构，后构造的先析构

3.对象指针

a.指向公共数据成员的指针，其用法与定义和普通数据指针类似

b.指向公共函数成员的指针，定义方法不同，即除了参数特征须相符外还必须限定类别

例：void (Class::\*p)( )

c.给指针赋值时也需注意，内存中对于成员函数只按类存放一次，而非按照对象存放

例：p=&Class::Function 而不是 ~~p=&Object.function~~

4.this指针

•this的值是当前被调用的成员函数所在对象的起始地址

•一般可以省略，即由编译器隐式调用，因为同一个类的所有成员函数使用同一个代码 段，通过this指针来储存当前正在被操作的对象，用以区分数据成员

5.常对象

const Class Object() 或 Class const Object()

a.声明为常对象的对象，其非const型成员函数不能被调用

b.声明为常对象的对象，其数据成员全部成为常数据成员，成员函数的类型不受影响

c.常对象的数据成员仍然可以用const型成员函数修改其值，只需在类体定义时加入：

mutable DataType DataName

c1.显然如果在类体内直接定义一个数据成员为const，则会与mutable指令冲突

c2.mutable只能修饰类的非静态数据成员

6.常数据/函数成员

静态常数据成员声明 const static DataType DataName

a.在类体定义中定义常数据成员或静态常数据成员，和定义常变量方法一致

const DataType DataName

b.只能通过构造函数的参数初始化表对常数据成员进行初始化，其实非静态成员变量的 初始化全部都必须在构造函数中进行（c++98中规定，c++11中使用初始化列表对于所 有非静态成员和静态常量成员都可以在类体中就地初始化）

b1.常对象的数据成员都是常数据成员，因此可以定义常对象的类一定含有使用参数初始化表的构造函数

c.常成员函数只能引用本类中的数据成员，而不能修改它们

c1.对于用mutable修饰的数据成员，const成员函数可以修改，即mutable可以突破const函数的限制

d.常成员函数的声明与定义都必须要有const关键字，调用时不必加

DataType FunctionName() const 或 DataType Class::FunctionName() const

e.常成员函数不能调用另一个非const成员函数

7.指向对象的常指针

Class \* const PointerName = …

常指针的值不能修改，即一次赋值之后，始终指向同一个对象，但对象的值可以改变

8.指向常对象的指针变量

用法类似于指向常变量的指针变量，这种指针变量多用于函数的形参，一来可以与常/变两种实参配对，二来可以保护形参指针所指的对象不被修改。

9.静态数据成员

声明static DataType DataName

a.静态数据成员不属于任何一个对象，在类的定义时便分配空间，可以引用

b.非常量的静态数据成员的初始化只能在类体外（头文件以外，目标文件内）进行，c++98和c++11版本的规定在此一致，若未进行初始化，则自动赋初值0

DataType Class::DataName=…

b1.静态数据成员就地初始化（类体内初始化）的条件是：必须是常量const，且必须是整型或者枚举型（c++98的规定，c++11中解除限制）

b2.即便不需要初始化，也必须在类体外再次声明非常量的静态数据成员，否则编译器会报错指出没有声明该变量。因此，静态数据成员在类体外用法类似全局变量。

c.静态数据成员可以用通过类名或者对象名引用

Class::DataName 或 Object.DataName

10.静态成员函数

声明static DataType FunctionName()

a.静态成员函数没有this指针，一般只用来访问本类中静态成员；若要坚持访问非静态成员，则必须确认静态成员函数定义时，所要引用的成员其所在对象有效。

b.定义静态成员函数时不加static

c.引用静态成员函数可以用类名或对象名（任意）

11.友元函数和友元成员函数

声明friend DataType FunctionName() 和 friend DataType Class::FunctionName

a.友元函数在类体中声明，类体外定义，定义时不再需要加friend

b.友元函数定义中可以引用类中的私有成员，但是引用时必须加上对象名，因为友元函数不属于该类。这对于其他类中的友元成员函数也一样，因为不属于该类，所以引用该类成员必须加上对象名。

c.一个函数（普通或成员函数）可以被多个类声明为友元。

12.友元类

声明friend ClassName

a.友元类中所有的函数都是目标类的友元函数，但此关系非双向，需要另行声明

b.友元类关系不能传递，仍需要另行声明

c.友元的使用会破坏信息隐蔽，需要在数据共享和隐蔽之间寻找平衡点

13.类模板（模板功能，亦可以用于函数）

声明template <class CustomTypeName1, class CustomTypeName2>

调用（定义对象）ClassName <DataType> ObjectName(…)

例：Student <int> stud(4,7)

调用（类体外定义成员函数）CustomTypeName ClassName <CustomTypeName>::Function()

例：numtype Compare <numtype> ::max()

a.模板的类型参数可以有多个，则定义对象时：

例：someclass <int,double> obj; 分别对应template<>语句中每个自定义类型

b.ClassName<CustomTypeName…>应理解为一个带参数的类

# 四、运算符重载

DataType operator X (形参表列)

例（对“+”重载，适用于复数）：Complex operator + (Complex &c1, Complex &c2)

(Complex为自定义类，代表复数类型)

1.简介与注意事项

a.运算符的重载实质上是函数的重载（“operator +”实际上相当于函数名）

b.直接调用构造函数而不加对象名（建立无名对象，见构造函数部分）的方法经常用于类的整体计算中

c.C++不许定义新的运算符，只能对已有的运算符进行重载

d.不能重载的运算符有5个：“.”成员访问运算符；“.\*”成员指针访问运算符；“::”域运算符；“sizeof”长度运算符；“?:”条件运算符

e.重载不能改变运算符对象的个数（单目，双目），运算符的优先级别，结合性（自右至左结合或自左至右）

f.重载运算符函数不能有默认参数，否则可以用户改变输入参数的个数，与e条目矛盾

g.重载运算符必须和自定义类型的对象一起使用，即其参数必须至少有一个为类对象或其引用；以此防止用户改变运算符原有的标准定义

g1.用于类对象的运算符一般都需要重载，除了“=”赋值运算符和“&”取地址运算符；但是对于赋值运算符，如果类的数据成员中包含指向动态分配内存的指针成员时，仍需要自己重载赋值运算符

h.运算符重载函数可以是成员函数，友元函数或普通函数，作成员函数时可以省略一个参数，同时也限制了该参数的形式必须为同类对象

h1.作为成员函数的运算符重载函数，编译器自动认为运算符前的操作对象即是函数所在对象，例如：cout<<自动解读为cout.operator<<

i.C++规定，有的运算符（赋值运算符、下标运算符、函数调用运算符）必须定义为类成员函数，另外的一些（流插入运算符和流提取运算符、类型转换运算符）则不能定义为类成员函数

i1.一般将单目运算符重载为成员函数，将双目运算符重载为友元函数

2.重载双目运算符

3.重载单目运算符

特别注意：自增（自减）运算符重载函数中，增加一个int型形参，就是后置自增（自减）运算符函数；换句话说，如果设置为成员函数，则前置自增（自减）运算符没有参数，后置的有一个int型参数。不过int型参数没有任何实际意义，不必带形参变量。

4.重载流插入和流提取运算符

声明和定义ostream& operator << (ostream &, 自定义类 &)

声明和定义istream& operator >> (istream &, 自定义类 &)

a.只能将流重载函数作为友元函数或普通函数，不能将其定义为成员函数，因为第一个参数不是class类型

b.ostream&和istream&都是类型名（引用类型），cin和cout分别是istream类和ostream类的对象

c.对于这类函数的定义，最后一定要返回istream&或ostream&类对象，即return …，这样做是便于连续输入输出

# 五、数据转换

1.转换构造函数（conversion constructor function）

数据转换成类ClassName (Data) 或

类转换成类ClassName (ClassName & X)

a.将数据转换为类，需要用到转换构造函数，一般是只有一个参数的构造函数重载，用法与一般构造函数一样；定义时应遵循一般常识来定义转换方法

b.数据转换既可以看成一个单独的指令，又可以视为调用构造函数的指令

c.类转换成类时，一般需要用一个类的成员函数（构造函数）来调用另一个类的数据，此时需要注意数据的公用私用的问题

2.类型转换函数（type conversion function）或类型转换运算符重载函数

operator DataType(){…}

作用是将一个类的对象转换成另一类型的数据

a.此函数没有参数，不能指定函数类型，返回值的类型是由DataType决定的（int/double/float…）；且只能作为成员函数（因此不需要指定参数），不能作为友元或普通函数

b.此相当于对数据转换运算符的重载，如对int()，double()的重载

c.类型转换函数和转换构造函数在计算中是可以被隐式调用的，即不用用户下命令，系统自动执行；这和标准数据类型之间的计算中存在的隐式转换类似

c1.隐式数据转换一般只在调用函数时，实参形参类型不匹配的情况下被系统执行。除此之外，系统不会执行隐式转换，否则会存在无意篡改用户程序的情况

d.类型转换函数，转换构造函数以及运算符重载函数的同时使用一般会带来矛盾

# 六、继承与派生

1.基本概念和思想

a.基类（base class）或父类（father class）

b.派生类（derived class）或子类（son class）

c.单继承（single inheritance）：一个派生类只从一个基类派生

d.多重继承（multiple inheritance）：一个派生类有两个或多个基类

e.继承与派生过程中，基类不被修改也不能被修改的原因：一是很多基类已经被各种程序使用，如果更改基类则会造成不兼容和失效；二是因为信息屏蔽导致用户得不到基类的源代码，无法进行修改；三是很多基类本就是设计成可供作为基类的，在面向对象程设的框架下继承派生体系已经成熟，擅自更改基类反而低效

2.派生类的声明

单继承声明：class 派生类名 [final] : [继承方式] 基类名 {派生类新增成员}

a.继承方式有public，private，protected；继承方式可以省略，则默认为private；公用继承中，基类的公用和保护成员在派生类中保持原有属性；私有继承中，二者成为派生类中私有成员；受保护继承中，二者成为派生类中受保护成员；所有继承方式中，基类的私有成员仍为基类私有，即派生类与类外都不得访问，以免破坏基类封装性

b.继承的成员不包括构造函数和析构函数；除此之外其他成员必须全部继承，没有选择

c.基类的成员函数只能访问基类的成员，不能访问派生类的成员（这是显然的）

d.私有继承方式的限制很多，一般不常使用，原则上适用于不希望将基类所有成员继续继承下去的情况；最常使用的方式是公用继承方式，这样多级继承后，所有数据的访问类型不变；公用派生类完整继承基类的功能，是基类真正的子类型或子类

e.在定义一个类型时，要考虑是否有可能会将其派生；则如果有些数据希望派生类可以使用，而类外不能引用，应将其定义为protected，而不是private

f. final关键字是可选的，用以指定当前派生类是最后一级派生，即不可再派生新类。

3.多级派生

直接派生类和间接派生类：一次派生关系的为直接派生，两次及以上派生关系的为间接

4.派生类构造函数

派生类构造函数名（总参数表列）:基类构造函数名（参数表列），子对象名（参数表列）

{只对派生类新增成员初始化的语句}

//子对象（subobject）即派生类中的成员，亦是对象

a.基类构造函数的部分是函数的调用而不是定义，所以“参数表列”里不需加数据类型；这些实参可以是常量、全局变量和派生类构造函数总参数表中的参数；如果选用或部分选用常量来给直接给基类构造函数的实参赋值，则不需要在总参列表中体现该参数

b.在类外定义，而在类中声明派生类构造函数时，不需要加基类构造函数及其参数表和子对象名及其参数表

c.派生类构造函数调用基类构造函数的方法实际上类似于参数初始化表方法，同样可以将派生类新增成员写成初始化表形式，跟在基类构造函数之后，则函数体为{}（空）

d.建立对象时，派生类构造函数先调用基类构造函数再初始化子对象，最后执行派生类构造函数本身；释放对象时反向，先执行派生类析构函数，再处理子对象，最后执行基类析构函数

e.编译系统是根据相同的参数名，而不是参数顺序来决定传递关系的。所以原则上，各个参数以及函数的书写顺序任意，但是习惯上应该对应

f.派生类构造函数的任务包括向基类构造函数和子对象构造函数传递参数以及初始化增成员；若基类或子对象的构造函数没有参数或没有定义，即派生类构造函数没有必要执行传递参数的任务，则可以在上面的指令中省略基类或子对象的构造函数部分

5.多层派生的构造函数

a.不要列出每一层派生类的构造函数，只需要写出上一层派生类（即其直接基类）的构造函数作为“基类构造函数”即可。形式同第4条给出的指令样式类似

b.调用构造函数的顺序：最原始的基类成员先开始初始化，最新增加的成员最后初始化

6.多重继承及二义性

声明：class 派生类名: [继承方式1] 基类名1，[继承方式2]基类名2…

{派生类新增成员}

a.多重继承的构造函数与单继承（第4条）类似，只需要逐个列出各个基类的构造函数及其参数表列，各基类的排列顺序任意

b.构造函数的执行顺序仍然为先基类最后派生类的顺序；多个基类的构造顺序按照派生类声明时各个基类出现的顺序为准

c.覆盖override：在派生类新增成员中，如果声明一个与基类成员同名的成员，则覆盖原成员；覆盖指的是不加作用域限定的情况下，默认引用的是派生类成员；基类的同名成员没有消失，仍然可以通过基类名加作用域运算符来引用

c1.对于成员函数同名同参才会覆盖，否则只是重载overload

d.若多个基类中存在同名成员，则必须加作用域限定才能正常访问每个成员，各个基类之间相互不覆盖，如“Object.BaseClass::Member”

e.一般来说，能用单一继承方式解决的问题不要用多重继承。有些面向对象的程设语言（Java，Samlltalk）并不支持多重继承

7.虚基类

class 派生类名:virtual 继承方式 基类名

a.在多重多层继承中，如果派生类的直接基类们有共同的直接基类（也就是派生类的间接基类），那么派生类则会继承多个重复的成员；通过在直接基类中声明间接基类为虚基类，可以在派生类中只继承其成员一次

b.虚基类是在以其为基础建立派生类时声明的，或者说并非是该类的固有属性，而是在继承关系中赋予的有作用范围（只在派生类和其继续派生的类中）的新属性

c.为最终派生类定义构造函数时，不仅要负责对其直接基类进行初始化，还要负责对虚基类初始化，即将虚基类视为其直接基类之一写入初始化指令；原因是最终派生类只继承一次虚基类成员，如果只调用直接基类构造函数，无法进行这种“单一”初始化

8.基类与派生类的转换和赋值关系

a.公用派生类又称子类，是完整继承基类功能并且可以和基类在部分场合替换的派生类

b.子类对象可以向基类对象赋值，其新增成员被舍弃；基类对象不能向子类对象赋值，同基类不同子类之间也不能赋值

c.子类对象可以向基类对象的引用赋值或初始化；赋值之后，引用变量和子类中基类部分共享同一存储单元，成为其“别名”

d.子类对象的地址可以赋给指向基类对象的指针变量，则指针变量指向子类中基类部分

9.继承与组合

a.在一个类中以另一个类的对象作为数据成员，称为类的组合（composition）

b.继承关系中，子类有时可以代替基类（见第8条），但组合关系中，成员类与组合类（或复合类，即包含类成员的类）之间没有这种关系

# 七、多态性（polymorphism）与虚函数

1.基本概念与思想

a.如果一种语言只支持类而不支持多态性，只能说是基于对象的，而不是面向对象的，如Ada，VB

b.C++程序设计中，多态性指：具有不同功能的函数可以用同一个函数名，则可以用一个函数名调用不同内容的函数

c.面向对象方法中这样表述多态性：向不同的对象发送同一个消息（调用函数），不同的对象在接受时会产生不同的行为（即方法）

d.静态多态性：在程序编译时系统就能决定调用的是哪个函数，例如函数重载和运算符重载（实则均为函数重载）；又称编译时的多态性

d1.静态关联（static binding）：又称早期关联（early binding）对应静态多态性，在编译时运行前即可确定调用的函数；包括用对象名调用虚函数也是此类，因为可以通过语句语法直接确定指向

e.动态多态性：程序运行时才动态确定操作所针对的对象，又称运行时多态性，靠虚函数实现

e1.动态关联（dynamic binding）：又称滞后关联（late binding），运行时才确定关联关系；用指针调用虚函数属于此类，因为编译时无法通过语法明确指向，实际上函数调用语句完全相同，只是指针的值不同，只有在运行到这一步时系统才能计算得出

f.关联、联编、编联、束定、绑定（binding）：确定调用的具体对象的过程

2.虚函数

virtual DataType FunctionName() //在声明函数时，前加关键词即可

作用：原本在基类和派生类中可以定义完全相同原型的函数，视为在派生类中重写/覆写override该函数。然而在某些时候，有关基类的一些代码需要在派生类上复用，但因为不得不更改源代码中的对象类型而造成不便。另一方面，指向基类的指针如果用来指向派生类，则一般只能访问派生类中的基类内容（除非使用显式类型转换）。因此，为了能够使用基类指针来调用派生类中的重写函数而实现调用语句一致化（即多态性和代码重用），虚函数诞生。派生类重写的虚函数可以被基类指针调用，当然，非虚函数还是遵循一般规律不能被基类指针访问。

a.在类外定义虚函数时，不必再加virtual

a1.虚函数只能用于成员函数声明，或者说用于类的继承功能中

b.当一个成员函数被声明为虚函数后，其派生类中的同名函数（各方面全相同）自动成为虚函数。派生类中再次声明该虚函数时，理论上可加可不加virtual，但是习惯上加

b1.同一类族中，某成员函数被声明为虚函数后，其他原型相同的函数不可再定义成非虚函数

b2.若派生类中没有对基类的虚函数重新定义，则派生类简单继承其直接基类的虚函数

c.虚函数是和指向基类对象的指针变量结合使用的：声明为虚函数的成员函数族（基类和每个派生类中新定义的同名函数），可以用指向基类对象的同一指针变量来调用，只要调用前将指针指向想要调用的函数的所在对象即可【一般情况下，基类对象指针即便指向派生类对象，也不能调用派生类成员，如果不使用虚函数，尽管可以用派生类对象为基类指针赋值，但调用的一定是基类的同名函数。这实际是使基类指针指向派生类中基类部分】

d. override关键字（C++11）用于派生类中对虚函数的覆盖，写于覆盖函数原型的尾部（若有const，在其之后）。这个关键字的作用仅在于告知编译器对虚函数的使用进行检查，减少程序出错的可能性。

d1.虚函数和函数重载的区别在于，虚函数的首部或者原型都是相同的，函数重载只是名称相同，参数个数与类型至少有一个不同

e. final关键字（C++11）用以指定某个派生类中的虚函数是最后一级覆盖，该派生类的子类不可再覆盖该虚函数。声明方式如下，注意final和override关键字顺序任意。

[virtual] ReturnType FunctionName(Parameter List) [=0] [final/override]

3.虚函数的功能和使用环境

a.看成员函数是否在继承后需要修改其功能，如果希望修改，尤其是多层继承中，希望该函数一直更新功能的情况，一般定义成虚函数

b.某些情况下（根据程序其他部分需要决定的），如果需要用基类的指针或引用来访问类族的成员函数，可以考虑声明为虚函数

4.虚析构函数

a.继承关系中，最好将基类的析构函数声明为虚函数；这样所有派生类的析构函数自动成为虚函数（这种情况下，派生类析构函数的名称显然不需要也无法与基类的相同）

b.构造函数不能声明为虚函数，因为构造函数执行时，类对象还没有完成建立过程，自然谈不上函数与类的绑定

5.纯虚函数与抽象类

纯虚函数声明virtual DataType FunctionName (…) =0;

a.纯虚函数是在声明虚函数时被“初始化”为0的函数，它没有函数体，“=0”也不代表函数返回值为0，它不能被调用，直到在派生类中重新定义有了函数体后

b.纯虚函数的作用就是在基类中为派生类保留一个函数名而已，但可完善多态性的结构

c.抽象类（abstract class）或称抽象基类（abstract base class）是一般专门用来生成派生类的基类；包含纯虚函数的类都是抽象类，因为纯虚函数不能被调用，抽象类也无法用来建立对象或称无法实例化

c1.一般能够用来建立对象的类，称为具体类（concrete class）

c2.可以定义指向抽象类数据的指针变量，用来实现多态性操作

# 八、标准输入输出流

1.基本概念

a.C++的输入与输出包括3方面：标准I/O，文件I/O，串I/O

b.标准I/O：对系统指定的标准设备的输入和输出，例如键盘与显示器

c.文件I/O：以外存磁盘/光盘文件为对象进行输入和输出

d.串I/O：对内存中指定的空间进行的输入和输出，或称字符串输入输出

e.C++的I/O操作特点（和C比较）：类型安全（type safe），C语言输入输出标准宽容度较大，编译系统不会对输入输出数据类型严格匹配，致使一些笔误也能编译通过，不过会导致意想不到的结果；因此会产生很多严重隐患，并且难以发现问题可扩展的，C++的输入输出可以用于用户自定义类型的数据，比如类、结构体和数组方便，C++输出不同基本类型数据时会自动调用对应重载函数，用户不必干预，C中必须分别指定相应的输出格式符

f.缓冲区：内存中为每一个数据流开辟一个内存缓冲区，当缓冲区满或收到特定信号时，缓冲区的数据统一流动；流是与内存缓冲区对应的，或说缓冲区中的数据就是流

g.C++中，输入输出流被定义为类，I/O类库中的类称为流类（stream class），用流类定义的对象称为流对象

h.cin和cout是流对象；在C语言中，用于输入输出的printf和scanf是函数

i. I/O类库中的常用流类

|  |  |  |
| --- | --- | --- |
| 类名 | 作用 | 头文件 |
| ios | 抽象基类 | iostream |
| istream | 通用输入流和其他输入流的基类 | iostream |
| ostream | 通用输出流和其他输出流的基类 |
| iostream | 通用输入输出流和其他输入输出流的基类 |
| ifstream | 输入文件流类 | fstream |
| ofstream | 输出文件流类 |
| fstream | 输入输出文件流类 |
| istrstream | 输入字符串流类 | strstream |
| ostrstream | 输出字符串流类 |
| strstream | 输入输出字符串流类 |

注：

(1) iostream类是多重继承的产物，ifstream是istream派生产物，以此类推常用头文件还有stdiostream（用于混合使用C和C++的I/O）和iomanip（用于格式化I/O）。

(2)fstream头文件包含iostream头文件，如遇二者皆用之情况，可省略后者

2.iostream头文件

a.iostream中定义的类有ios，istream，ostream，iostream，istream\_withassign，ostream\_withassign，iostream\_withassign；其中\_withassign是不带后缀的类的派生类

b.iostream中定义的对象

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 对象 | 含义 | 设备 | 对应的类 | C中相应标准文件 |
| cin | 标准输入流 | 键盘 | istream\_withassign | stdin |
| cout | 标准输出流 | 屏幕 | ostream\_withassign | stdout |
| cerr | 标准错误流 | 屏幕 | ostream\_withassign | stderr |
| clog | 标准错误流 | 屏幕 | ostream\_withassign | stderr |

用键盘输入时用cin，向显示器输出用cout；标准输入设备是键盘，标准输出设备是显示器

3.标准输出流

a. cout（即console output），意为在控制台（终端显示器）的输出，也可以被重定向输出到磁盘文件

b. cerr（即console error），被指定为与显示器关联，向标准错误设备输出有关出错信息

c. clog（即console log），作用与cerr相同，但是cerr不经过缓冲区，直接向显示器上输出有关信息，而clog的输出方式和cout类似

4.格式输出

a.使用控制符控制输出格式

|  |  |
| --- | --- |
| 控制符 | 作用 |
| dec | 设置整数的基数为10 |
| hex | 设置整数的基数为16 |
| oct | 设置整数的基数为8 |
| setbase(n) | 设置整数的基数为n（n只能为8,10,16三者之一） |
| setfill(c) | 设置填充字符c，c可以是字符常量或字符变量 |
| setprecision(n) | 设置实数的精度为n位 |
| setw(n) | 设置字段宽度为n位 |
| setiosflags() | 设置输出格式状态，括号中指定格式标志（见下面b1表） |
| resetioflags() | 终止已设置的输出格式状态，在括号中应指定内容 |

注：setprecision(n)，在以一般十进制小数形式输出时n代表有效数字。在以fixed（固定小数位数）形式和scientific（指数）形式输出时n为小数位数

b.用流对象的成员函数（cout成员）控制输出格式

|  |  |  |
| --- | --- | --- |
| 流成员函数 | 相应功能的控制符 | 作用 |
| precision(n) | setprecision(n) | 设置实数的精度为n位 |
| width(n) | setw(n) | 设置字段宽度为n位 |
| fill(c) | setfill(c) | 设置填充字符c |
| setf() | setiosflags() | 设置输出格式状态，括号中应给出格式状态 |
| unsetf() | resetioflags() | 终止已设置的输出格式状态，在括号中指定内容 |

b1.设置格式状态的格式标志

|  |  |
| --- | --- |
| 格式标志 | 作用 |
| ios::left | 输出数据左对齐 |
| ios::right | 输出数据右对齐 |
| ios::internal | 数值的符号位在域宽内左对齐，数值右对齐，中间由填充字符填充 |
| ios::dec | 设置整数的基数为10 |
| ios::oct | 设置整数的基数为8 |
| ios::hex | 设置整数的基数为16 |
| ios::showbase | 强制输出整数的基数（八进制数以0开头，十六进制以0x开头） |
| ios::showpoint | 强制输出浮点数的小点和尾数0 |
| ios::uppercase | 在以科学计数法输出E和以十六进制输出字母时以大写表示 |
| ios::showpos | 输出正数时给出“+”号 |
| ios::scientific | 设置浮点数以科学计数法（即指数形式）显示 |
| ios::fixed | 设置浮点数以固定的小数位数显示 |
| ios::unitbuf | 每次输出之后刷新所有的流 |
| ios::stdio | 每次输出之后清除stdout，stderr |
| ios::skipws | 忽略前导的空格 |

注：格式标志在类ios中被定义为枚举值，因此引用时要在前面加上“ios::”

c.注意事项

•成员函数width（n）和控制符setw（n）只对其后的第一个输出项有效，所以如果有需要，每次都输出一项都要调用一次函数或控制符

•b1表中，每个分组之内的格式标志之间可能存在排斥，只能选取一种，如果想更改设置为同一组的另一个状态，则必须先调用终止原状态的函数或控制符；在系统默认使用一些状态的情况下（如默认使用dec），虽然用户没有设置，仍然需要先终止再重设

•用setf函数设置格式状态时，可以包含两个或多个格式标志，用“|”组合这些标志

•对于格式控制，两种方法作用相同。只是使用控制符时，必须添加iomanip头文件，而使用成员函数则只需iostream即可；但是控制符的使用更加方便，可以在一个cout语句中连续使用多个控制符，而函数的使用和一般的类成员函数一样，一次只能调用一个函数，只是对于setf函数有连续声明格式标志的设定

5.输出流成员函数put

cout.put().put()…

a.put函数时cout的成员，只用来输出一个字符，但是可以连续调用

b.put函数的参数可以是字符（如‘a’）或者字符的ASCII代码或整形表达式（如65+32）

6.标准输入流

a.只有在输入完数据再按回车键之后，所有输入的数据才被送入键盘缓冲区，形成流

b.当遇到无效字符（与所需赋值的数据之类型不匹配）或者遇到文件结束符时，cin就处于出错状态，所有提取操作将终止；当出错时，cin的值为false，即0，若正常状态下，cin值为非零，即true

b1.当cin或其他流对象正常读取到数据流的结尾处时，即本次读取的下一次读取将会遇 到文件结束符时，在本次读取结束后下一次读取之前，流对象取值仍然为非零，即正常状态；当下一次读取遇到结束符时，才会变为false

c.文件结束符，在IBM PC及其兼容机中，为Ctrl+Z，在UNIX和Macintosh系统中，则为Ctrl+D

7.输入流成员函数get，getline

·cin.get()

a.不带参数，用来从指定的输入流中提取一个字符（包括空白字符），函数返回值即为读入的字符，如果遇到输入流中的文件结束符，则返回值返回文件结束标志EOF（End of File），一般EOF是一个负数，如-1

·cin.get(ch)

b.有一个参数，从输入流中读取一个字符，赋给字符变量ch，如果读取成功则返回非零值，反之返回0值

·cin.get(字符数组，字符个数n，终止字符)

·cin.get(字符指针，字符个数n，终止字符)

c.作用是从输入流中读取n-1个字符，赋给指定的字符数组或者字符指针指向的数组，如果在读取n-1个字符之前遇到指定的终止字符，则提前结束读取。同样，成功读取 返回非0，失败返回0

c1.第n个字符是留给字符串结束标志‘\0’的，若终止字符省写，则默认为‘\n’，即回车；终止字符不被读取，即读取结束至终止字符之前

·cin.getline(字符数组(或字符指针)，字符个数n，终止标志字符)

d.用法与3参数get函数类似，不过遇到终止字符结束后，输入流字符指针移到终止字符之后，即跳过终止字符指向下一个字符；而get函数遇到终止字符后指针不动，下次读取仍从该终止字符开始

e.（用法区别）用cin<<读取数据时以空白字符为终止标志，即空格，tab和回车；而用getline或get函数可以连续读取一系列字符，可以包括空格；但是cin<<可以读取标准类型的各类数据以及用户自定义数据，而get函数只用于输入字符型数据

8.输入流其他成员函数

·cin.eof()

a.可以表示文件是否结束，不同于EOF标志符，这是个函数；若到达文件末尾，即遇到文件结束符，eof返回值非零，反之为0

·cin.peek()

b.作用是观测下一个字符，函数返回值是指针指向的当前字符，观测后指针仍停留当前位置，若字符是文件结束符，函数值是-1（不同C++系统所用EOF值可能不同）

·cin.putback(ch)

c.作用是将字符ch返回到输入流，插入到当前指针位置之前的位置

·cin.ignore(n，终止字符)

d.作用是跳过输入流中的n个字符，或在遇到指定的终止字符时提前结束（此时跳过包含终止字符在内的若干字符）

d1.n的默认值为1，终止字符默认为EOF，则该函数可以省略参数

e.所有流成员函数，可以用istream或ostream类其他任意对象调用均可

# 九、文件操作与文件流

1.基本概念

a.文件（file）：一般指存储在外部介质上的数据的集合

b.常用文件：

·程序文件（progam file），如.cpp，.obj，.exe等

·数据文件（data file），一般外部存储输入输出数据的文件

c.数据文件按照数据组织形式划分：

·ASCII文件（又称text，即文本文件或字符文件），每个字节存放一个ASCII码，代表一个字符

·二进制文件（又称内部格式文件或字节文件），把内存中的数据按其在内存中的存储形式原样输出到磁盘上存放

2.文件流类与文件流对象

建立输出文件流对象：ofstream 流对象名称

打开磁盘文件（建立输入输出关系）：

·调用成员函数法 -- 文件流对象.open(外部文件名，输入输出方式)

·调用构造函数法 -- ostream 流对象名称(外部文件名，输入输出方式)

a.外部文件名可以是全路径，若缺省路径只留文件名，则默认为当前目录下文件

b.文件输入输出方式设置值（ios枚举常量）

|  |  |
| --- | --- |
| 方式 | 作用 |
| ios::in | 以输入方式打开文件 |
| ios::out | 以输出方式打开文件（默认方式），若已有该文件，则原内容被清除 |
| ios::app | 以输出方式打开文件，写入的数据添加在文件末尾 |
| ios::ate | 打开一个已有的文件，文件指针指向文件末尾 |
| ios::trune | 打开一个文件，若其已存在，则删除其数据，若不存在，则建立新文件；若已指定ios::out方式，而未指定app/ate/in，则默认此方式 |
| ios::binary | 以二进制方式打开一个文件，若不设置，则默认为ASCII方式 |
| ios::in|ios::out | 以输入和输出方式打开文件，文件可读可写 |
| ios::out|ios::binary | 以二进制方式打开一个输出文件 |
| ios::in|ios::binary | 以二进制方式打开一个输入文件 |
| ios::in|ios::out|ios::binary | 打开一个二进制文件，可读可写 |

c.每个打开的文件都有一个文件指针，其初始位置由输入输出方式指定（见b表），每读入一个字节，指针后移一个字节；指针移到最后，遇到文件结束符EOF

c1. EOF占一个字节，其值为-1；遇到结束符时，流对象成员函数eof()的返回值是非零值，一般为-1

d. 如果打开操作失败，open函数返回值为0，如果是调用了构造函数，则流对象值为0；同时流对象成员函数fail()返回值为1

关闭磁盘文件：流对象.close() //即调用一个成员函数即可

e.标准输入输出流中介绍的成员函数都可以用于外部文件的输入输出，只是对象名更改（针对ASCII文件或字符文件）

3.二进制文件的操作

a.二进制文件可以同时既能输入又能输出，这和一般的ASCII文件不同

成员函数read和write原型

istream& read(char\*buffer, int len)

ostream& write(const char\*buffer, int len)

字符指针buffer指向内存中一段存储空间用于输入和输出，len是读写的字节数

b.有时为了满足“字符指针”的要求，需要将其他指针转换为字符指针，char\*()

c.二进制文件的输入输出不必加空格与回车，因为不依靠这些来作为数据间隔，而是通过字节数；字节数的测定函数sizeof(…)

d.将数据从文件读入内存时，存放在buffer指定的内存空间中，但是该内存空间的格式必须要与读入的数据格式匹配，格式即是由该内存空间所存储变量的类型决定的

4.与文件指针有关的流成员函数

a.对于二进制文件，允许对其文件指针进行控制，按照用户意图移动到所需位置

b.文件流与文件指针有关的成员函数

|  |  |
| --- | --- |
| 成员函数 | 作用 |
| gcount() | 返回最后一次输入所读入的字节数 |
| tellg() | 返回输入文件指针的当前位置 |
| seekg(文件中的位置) | 将输入文件中指针移到指定的位置 |
| seekg(位移量，参照位置) | 以参照位置为基础移动若干字节（参照位置的用法见p451） |
| tellp() | 返回输出文件指针当前的位置 |
| seekp(文件中的位置) | 将输出文件中指针移到指定的位置 |
| seekp(位移量，参照位置) | 以参照位置为基础移动若干字节 |

注：“文件中的位置”和“位移量”已被指定为long型整数，以字节为单位。“参照位置”是下面三者之一（枚举常量）ios::beg（文件开头，这是默认值）；ios::cur（指针当前位置）；ios::end（文件末尾）

# 十、字符串流（内存流）

a.字符串流和标准输入输出流类似，所有数据经过缓冲区都必须经历二进制形式和ASCII码形式之间的转换。虽然字符串流的存储格式是字符数组（字符串），但是可以存放各种类型的数据（因为有二进制和ASCII的转换存在）

b.字符串流对象关联的不是文件，因此不需要打开和关闭文件（外部设备也被视为文件）

b1.因为不需要打开和关闭文件，则可以同时定义两个字符串流，针对同一字符数组输入输出，因为有各自的指针，互不干扰

c.字符串流没有文件结束标志，用户要指定一个特殊字符作为结束符，写入全部数据后要写入此字符，一般是“流对象<<ends”；ends是I/O操作符，即一个‘\0’

1.建立输出字符串流对象

构造函数原型 ostrstream::ostrstream(char \*buffer, int n, int mode=ios::out)

a. buffer是指向字符数组首元素的指针（也就是字符数组的名称），n为指定的流缓冲区的大小（一般选与字符数组的大小相同，可用sizeof(…)直接代替n，也可以不同，单位是“个字符”）

2.建立输入字符串流对象

构造函数原型 istrstream::istrstream(char \*buffer)

或 istrstream::istrstream(char \*buffer, int n)

a. n仍为指定缓冲区大小，若小于字符数组大小，则只读取前n个字符

3.建立输入输出字符串流对象

strstream::strstream(char \*buffer, int n, int mode)

# 十一、异常处理

1.基本思想

a.一般使出错的检查和处理不由同一函数来完成，把处理异常的任务上移到某一层去处理；底层函数只负责解决实际任务，主函数等顶层函数中设置异常处理

2.异常处理方法

throw运算符的使用：throw 表达式

try-catch结构：

try {被检查的语句}

catch(异常信息类型[变量名]) {进行异常处理的语句}

a. try-catch结构是一个整体，try和catch块之间不能有其他语句，但可以没有catch块；try和catch块中即使只有一个语句，花括号也不能省略；一个try-catch结构只能有一个try，却可以有多个catch，只是异常信息类型不同而已

b.如果try块内没有发生异常，则catch子句不起作用，直接跳过catch块；如果try块中发生异常，即throw运算符被调用，则立即跳出到与throw的表达式数据类型对应的catch块，执行完catch块内容后，不返回出错点也不终止程序，而是接着继续执行catch块之后的内容

c. 一般catch接受的只是异常信息类型，即throw运算符后表达式的数据类型，并不检查其值，因此不同数值相同类型的throw语句调用的是相同的catch块；catch语句的变量名用来接收从throw表达式传递的值，可以省略，数据类型可以是自定义数据类型或“类”

c1. catch子句可以不指定异常信息的类型，而用删节号，即catch(…)，则表示它可以捕捉任何类型的异常信息；一般把这种catch块放在所有其他同try-catch结构的catch块的最后，相当于“其他类型”，否则其他的catch块无法起作用

d. throw语句和try-catch结构不是对应关系，如果一个throw语句发出一个异常信息，它会按照由近及远由底层至上层的顺序寻找对应的catch块，可能会遍历多层多个try-catch结构，直到找到对应的catch块，否则系统调用一个系统函数terminate终止程序运行

e. 有时throw语句可以不包括表达式，作用是将当前在处理的异常信息再次抛出，交给更上一层catch块处理；这种throw语句一般用在某一层catch块中，尤其是catch(…)块，表示“处理不了当前异常信息，再次抛出原异常信息”

3.函数的异常情况指定

DataType FunctionName(…) throw(DataType1, DataType2, …)

a. 这是一种函数声明方式，一般情况下如果不加throw()部分，则函数可以抛出任何类型的异常信息；指定后，该函数可以抛出指定的几种类型的异常信息

a1. 如果throw()部分无参数，则该函数不能抛出异常信息；即使函数执行过程中出现了throw语句，也不执行，程序会非正常终止

b. 异常指定必须同时出现在函数声明和函数定义的首行中

4.C++标准异常类型

C++ 提供了一系列标准的异常，定义在 <exception> 中，我们可以在程序中使用这些标准的异常。它们是以父子类层次结构组织起来的（共3级）。下表中的颜色越深代表越接近基类层，白色背景的类是最靠近其上方的灰色类的子类。

what() 是异常类提供的一个公共方法，它已被所有子异常类重载。这将返回异常产生的原因。

|  |  |
| --- | --- |
| 异常 | 描述 |
| std::exception | 该异常是所有标准 C++ 异常的父类。 |
| std::bad\_alloc | 该异常可以通过 new 抛出。 |
| std::bad\_cast | 该异常可以通过 dynamic\_cast 抛出。 |
| std::bad\_exception | 这在处理 C++ 程序中无法预期的异常时非常有用。 |
| std::bad\_typeid | 该异常可以通过 typeid 抛出。 |
| std::logic\_error | 理论上可以通过读取代码来检测到的异常。 |
| std::domain\_error | 当使用了一个无效的数学域时，会抛出该异常。 |
| std::invalid\_argument | 当使用了无效的参数时，会抛出该异常。 |
| std::length\_error | 当创建了太长的 std::string 时，会抛出该异常。 |
| std::out\_of\_range | 该异常可以通过方法抛出，例如 std::vector 和 std::bitset<>::operator[]()。 |
| std::runtime\_error | 理论上不可以通过读取代码来检测到的异常。 |
| std::overflow\_error | 当发生数学上溢时，会抛出该异常。 |
| std::range\_error | 当尝试存储超出范围的值时，会抛出该异常。 |
| std::underflow\_error | 当发生数学下溢时，会抛出该异常。 |

# 十二、命名空间

1.基本思想与概念

a. 命名空间就是一个有名字的空间域，这个域内存放一些全局实体的名称，不同的命名空间中可以有同名的实体，彼此不冲突

b.一般所用的全局变量可以理解为全局命名空间，独立于所有有名字的命名空间之外，实际是由系统隐式声明的

2.定义命名空间

namespace NamespaceName {声明各种数据的语句}

a.命名空间中包含的名称种类：变量、常量、函数（可以是定义或声明）、结构体、类、模板、嵌套命名空间

b.命名空间的声明与使用方法和类差不多，命名空间中的成员也需要由“::”限定，唯一的不同是，类声明结束时有分号，而命名空间后没有分号

c.使用命名空间时注意，只有命名空间的成员前才可以加空间名和限定符号，比如用命名空间中的类定义对象时，类名前应加空间限定，而对象前不应再加；因为该对象不是命名空间中的成员，自然也不在其作用域内

3.使用命名空间的方法

namespace ShortName=OldName

a.为较长名称的命名空间起一个别名来简化

using 命名空间::成员

b.在该语句的作用域内，可以直接使用声明的成员名称而不必加空间限定；一次只能声明一个成员

b1.显然使用这个语句时不能同时指定同名的不同空间成员

using namespace 命名空间

c.作用与上一个语句类似，同时声明一个命名空间中所有成员为全局成员

4.无名命名空间

声明时只有namepsace关键字没有名称

a.作用相当于C语言中的静态声明，无名空间中的成员只能在该文件使用，相当于静态全局变量或函数

5.标准命名空间std

a.标准C++库中的所有标识符都是在std命名空间中定义的，或说标准头文件中的所有成员都是在std中定义的

b.标准命名空间std中的实体太多，一般专业编程不会用using namespace std将所有成员声明成全局成员，而是只将经常使用的一些用using一个一个声明，或置于一个头文件中，便于经常调用

6.C语言函数库

a.C语言函数库都是由带.h后缀的头文件声明的，但是C语言没有命名空间，因此使用时不必考虑命名空间；相当于默认对整个头文件使用using namespace命令

b.C++的头文件不带.h后缀，相似功能的C++头文件一般是在C的头文件名称前加“c”；但是需要注意cstring头文件对应string.h头文件，提供C语言中对字符串处理的有关函数的声明，而string头文件是C++增加的对字符串处理的新功能