C++复习笔记

Hushrush.2021.05.17

1. 2021复习提纲

见文件《2021复习提纲》

1. 考试内容复习

**以四次作业和一次实验为出题基本素材，可能就是作业题的稍加修改**

* 1. 函数重载

两个及以上函数（外部或成员函数）共用一个函数名叫做函数重载。重载函数的参数必须不同，编译器根据实参类型自动确定调用哪个函数。

* 1. 内联函数

内联函数时冠以关键字“inline”的函数。与宏定义#define类似[[1]](#footnote-1)，出现调用该函数的地方编译器就用函数体的代码替换调用表达式，这样能以空间换时间，加快代码执行，减少调用开销。外部函数和成员函数都可以选择是否内联，但一般类内部定义的函数默认为内联函数。

例2.2.1：

|  |
| --- |
| class A  {  int test;  public:  A();//构造函数，定义在外面，不在函数内部  void show();  void increat(int incr\_num=2)//这叫内部定义  { //注意这里的=2指没有实参的话参数默认为2  test += incr\_num;  }  };  A::A()//这里才是构造函数的定义，这个构造函数不是内联函数  {  test = 1000;  }  inline void A::show()//通过inline定义的内联函数  {  cout << test << endl;  } |

一般内联函数的函数体很短。

* 1. const修饰符

常量：如’A’,10等字面常量；

符号常量：如#define PI 3.14（宏定义）；

常变量：用const定义。

注意：只有const修饰的常变量才有存储空间，另外两个都是编译时替换。

特点：

（1）常量、符号常量、常变量等运行时均不可改变；

（2）使用const修饰符时必须定义时初始化，如const int a;是错误的；

（3）**常量指针是指指针自己是常量，指向常量的指针自己不一定是常量；如**

**char \*const name=”chen”;//name是常变量，name指向的位置不能改变，但name指向的内容可以改变。**

**const char \*name=”chen” ;//name指向的空间内容不能改变，但是name可以指向别的地方。**

（4）const定义整型常量时可以不用int；

const int A = 10;

const A = 10;

（5）const比#define安全，因为#define不会检查类型（窄化转换）；

（6）const可以修饰参数、返回和函数本身。

* 1. new和delete

1. new和delete都是运算符，可以重载；malloc和free是库函数；
2. new可以自动计算所需空间，不用sizeof()；
3. new不需要指针类型转换；
4. new可以初始化，为数组动态分配空间

int \*p=new int(9);//&p=9

int \*p=new int[9];//p是数组首地址，数组长度是9

会自动调用构造、析构函数。

1. 释放数组时要加中括号

delete [] p;

所指空间被释放后，指针应该赋值为NULL；否则是野指针，非常危险。

* 1. 作用域运算符::

1. 使用类的静态成员[[2]](#footnote-2)；
2. 解除被隐藏的全局函数、变量

int num = 1;

int main()

{

int num = 2;

cout<<num<<endl;//2

cout<<::num<<endl;//1

}

* 1. 引用

是对象，变量的别名。

int i=5; //声明一个整型变量

int &j=i; //声明一个引用j指向整型变量I

j=4;

注意：

1. 定义引用时就要赋值；

int i = 0;

int &j = i;

int & k;//false

k=i;//false

1. 引用不能被引用，不能再次赋值；
2. 数组，指针不能引用；
3. 不要返回函数局部变量的引用，因为出函数后局部变量被销毁。
4. 主要用于定义函数的参数和返回值。定义返回值有技巧。

值传递：

* 1. 类和对象

1. 只有构造函数和析构函数不用写返回类型；
2. 注意class的定义格式，大括号后面有分号；
3. 成员函数的定义可以在类的声明中给出，也可以在类外部定义。大型程序一般定义在头文件。声明中给出则属于内联函数，外部定义必须加inline才是内联函数。

要注意定义的格式：

|  |
| --- |
| class complex  {      double real;      double imag;  public:      complex(){}//一般定义一个空的构造函数，用于数组定义等      complex(double r,double i)//类内部定义      {          real=r;          imag=i;      }      double realcomplex();  };  double complex::realcomplex()//注意定义格式，**先返回类型再类名**  {      return real;  } |

构造、析构函数和拷贝构造函数有默认，自己定义后则不再默认。所以定义一个空的构造函数。

1. private，public，protected没有顺序
2. 类中成员可以是对象，但不能是自身类的对象，否则构造时会循环。（可以是本类对象的指针或引用）数据成员为其他类对象时，其他类需要先定义或声明；
3. 类体中数据成员不能初始化；
4. 数据成员不能用auto、register、extern，但可以是static（静态数据成员）。成员函数也不能用extern
5. 构造函数注意事项：
6. 构造函数的名字与类名相同
7. 构造函数是公有的
8. 可以有参数但不能有返回类型
9. **不能显式调用（派生类传参时除外）**

**例如：complex A (1.1,1.2);//√**

**complex A;**

**A(1.1,1.2)//×**

1. **可以重载，但要注意二义性问题，特别是带默认参数的时候，因为这类函数接受所有较少的实参**
2. 析构函数注意事项：
3. 名字为~类名
4. 没有参数没有返回值不能重载
5. 可以显式调用但是不要显式调用，会造成重复释放内存
6. 若批量创建对象，可以采用带默认参数的构造函数，但要注意这不是默认的构造函数

|  |
| --- |
| class complex  {      double real;double imag;  public:      complex(double r,double i)      {real=r;imag = i;}  };  int main()  {      complex A(1.1,1.2);      complex B(1.1);//先给前面的参数。real=1.1,imag=0      complex C();//ppt上带括号，本地发现不应该带括号      return 0;  } |

如果这例中有一个带一个double形参的函数就错了，有二义性。

1. **拷贝构造函数注意事项**
2. **只有一个参数，即对某个对象的引用**

|  |
| --- |
| class complex  {      double real;double imag;  public:      complex(double r,double i)      {real=r;imag = i;}      complex(const complex &c);//注意const和complex，被引用对象不能修改，不要因为有了修饰符就忘记了数据类型  };  complex::complex(const complex &c)  {      ……  } |

Const是为了提醒用户，也可以不用

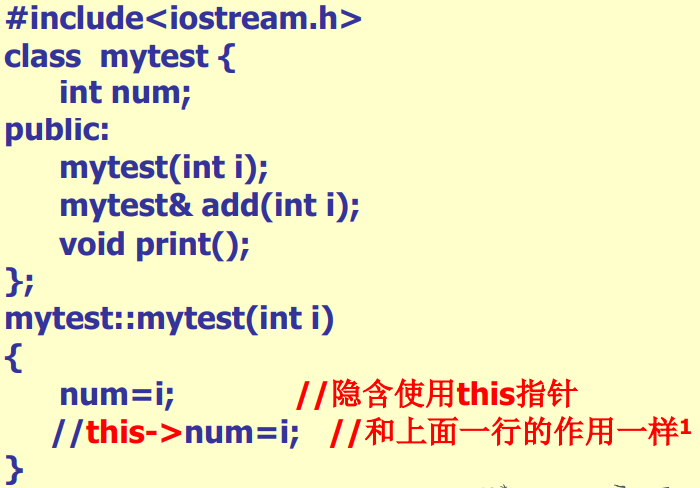
&不能省略，因为如果省略是值传递，值传递又要拷贝构造，会进入死循环

1. 默认拷贝构造函数是浅拷贝
2. 同类对象可以互访私有成员，不同类对象不行
3. 构造函数默认规则

不写构造函数则默认两个构造函数，写一般构造函数则默认拷贝构造函数，但写拷贝构造函数就不会默认一般构造函数。

1. 注意static局部对象（不是static静态成员）首次创建调用构造函数，main函数终止或exit后析构。
2. 对象指针也可以用new动态创建，但调用的是默认构造函数或带默认值的构造函数。
3. this指针只能用，不能定义或修改。

|  |
| --- |
| 当一个对象调用成员函数的时候，编译程序先将对象的地址赋给this指针，然后调用成员函数，该成员函数隐含地通过this 指针存取数据成员，有些情况下要显式使用this指针 |



1. 对象作为参数时注意引用传递不用构造一个新对象，但是值传递时要构造一个新对象会使用**拷贝构造函数**
2. **静态成员注意事项**

静态数据成员同类对象均可见，所以一般定义成私有。静态成员有一块单独的存储区域，所有对象共享这块区域。

注意静态数据成员的初始化方式：**在类体外初始化**

静态数据成员在定义对象之前就分配了空间，也要在**定义对象之前初始化**

|  |
| --- |
| #include <iostream.h>  class Student {  static int count; //声明静态数据成员count，统计学  生总数  int StudentNo;  public:  Student() //构造函数  { count++; StudentNo=count; }  void print() //输出对象的值  {  cout <<“Student”<<StudentNo<<“ ”;  cout<<“cout=”<<count<<endl;  }  };  **int Student::count=0;//初始化静态成员** |

**注意作用域运算符：：和被作用的紧挨，int在前面！！！**

**这里解释static：在函数体内或外定义的static都放在全局变量去而不是函数的堆栈区，在程序结束之前不收回。**

**在类外重新声明的时候不写static**

1. **静态成员函数**

**意义在于操作私有的静态数据成员，因此不能操作非静态成员，因为不知道哪一个的。如果非要操作静态成员，可以传一个参数说明是哪一个成员。**

|  |
| --- |
| #include <iostream.h>  class Student{ //定义了Student类  public:  int aaa;  static int number( Student &ss) //静态成员函数  { cout<<ss.aaa<<endl;  return noOfStudents;}  protected:  char name[40];  static int noOfStudents;  };  int Student::noOfStudents=1; //初始化静态变量  void main()  {  Student s,ss;  cout<<s.number(ss)<<endl; //使用对象引用静态成员函数  //cout<<Student::number()<<endl;  } |

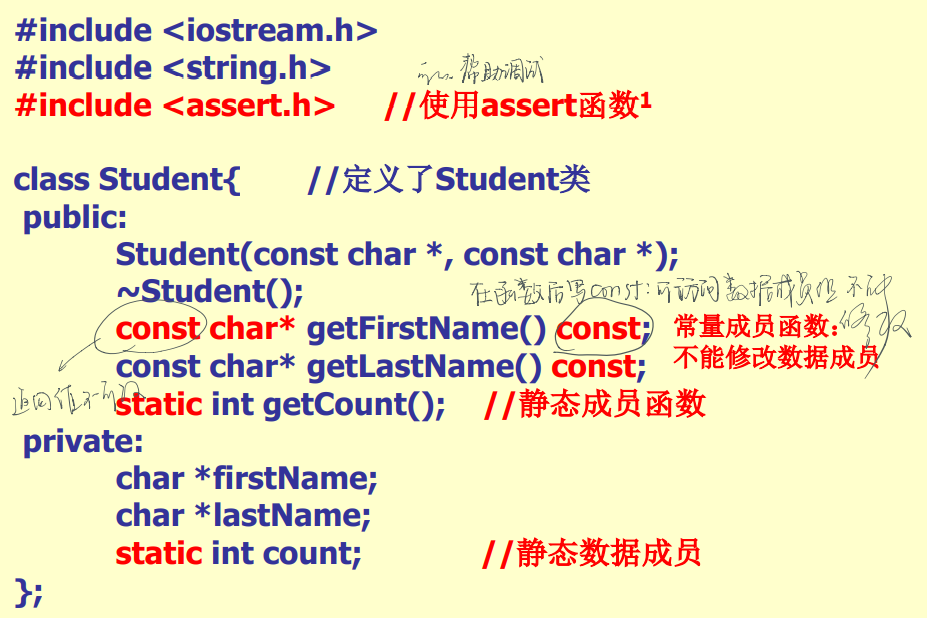
**静态成员函数在定义对象之前也可以用。**

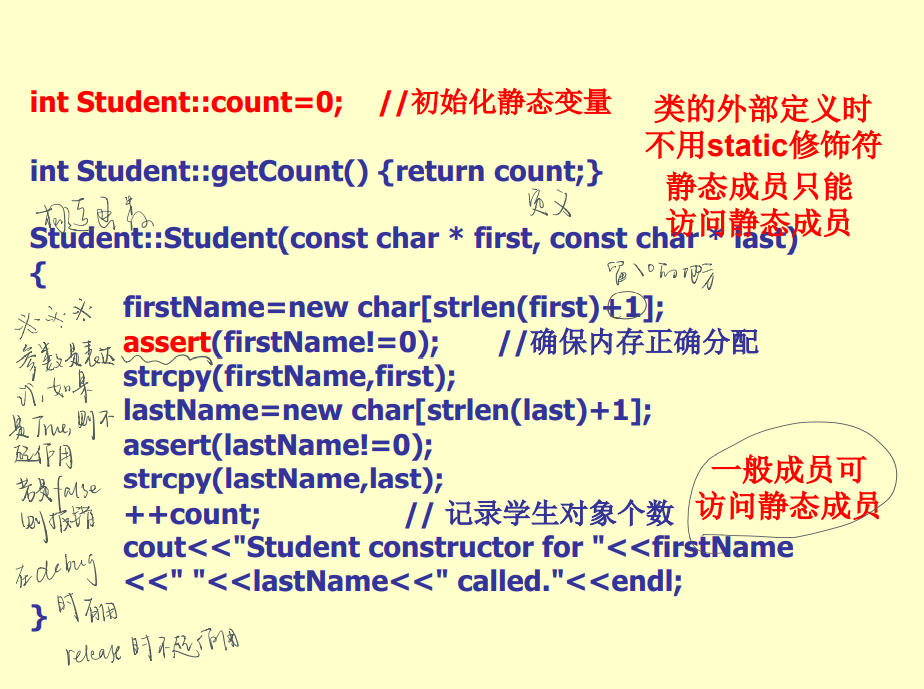
**用类名：：静态成员函数和某一个对象名.静态成员函数是一样的。**

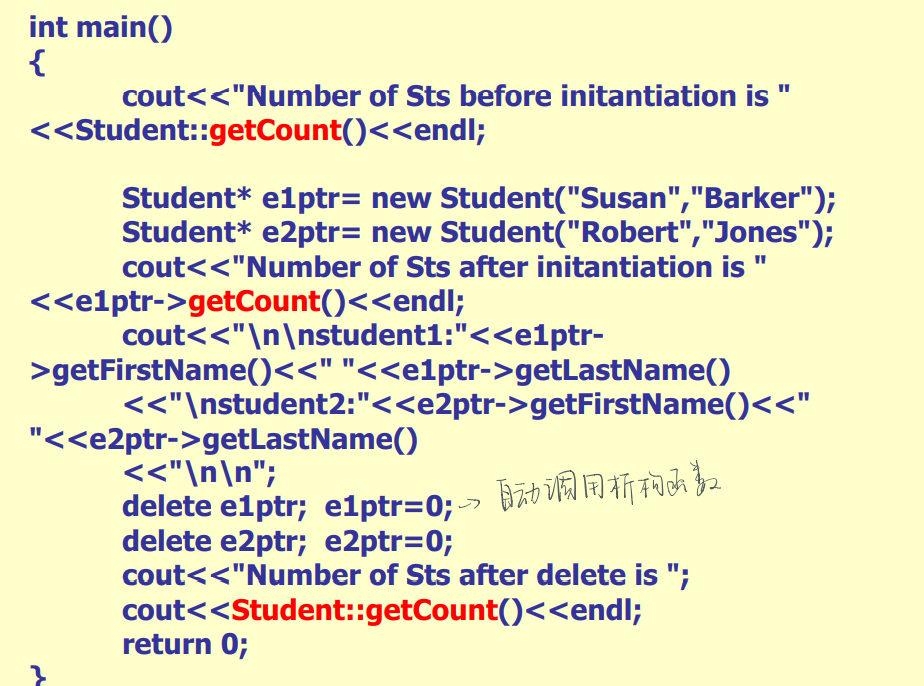
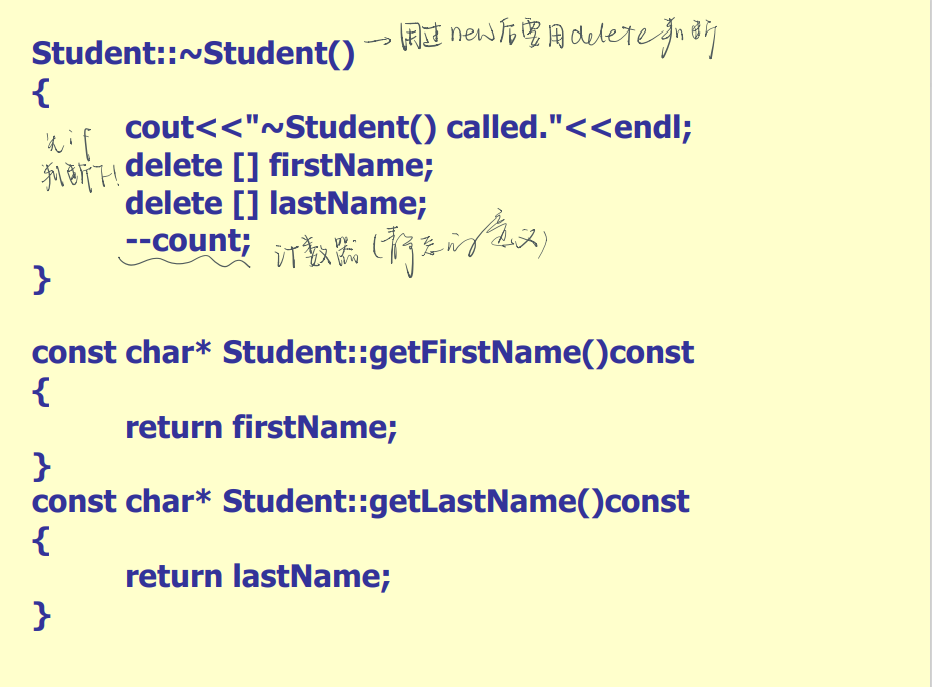
**静态成员函数没有this指针。**

**静态成员函数可以定义为内联的或类外定义，类外定义时不要写static**

**静态数据成员也有public、private、protected。**

****

****

****

1. 返回有const不能修改

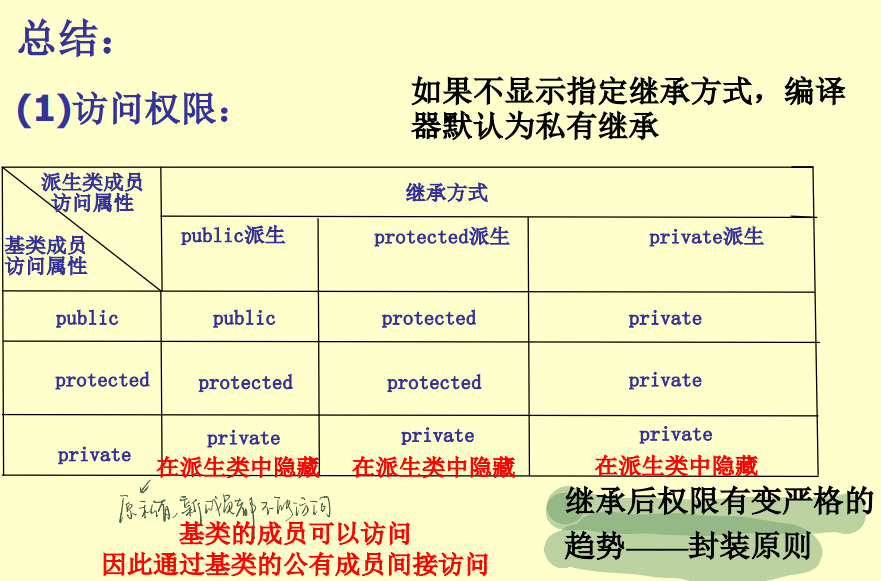
|  |
| --- |
| #include <iostream.h>  #include <string.h>  class Student{ //定义了Student类  public:  Student(const char \*);  ~Student() {delete [] firstName;}  const char\* getFirstName() {return firstName;}  private:  char \*firstName;  };  Student::Student(const char \* first)  {  firstName=new char[strlen(first)+1];  strcpy(firstName,first);}  void main()  {  char\* q;  Student s("abc");  q=s.getFirstName();q[1]='w'; //错误，不能修改返回的string  } |

1. 友元注意事项
2. 某个类的友元函数不是该类的函数，但是可以访问该类所有对象的成员。
3. 友元函数必须在类体内声明，friend在返回类型前面。友元函数也可以在类体内声明。
4. 友元函数前面不要：：因为不是成员函数。
5. 一个类的成员函数可以作为另一个类的友元函数。注意顺序：某个类A的成员函数是另一个类B的友元函数，则先声明B，再定义类A，最后定义类B。也即含friend的最后写。
6. 某个类可以是另一个类的友元
7. 若有对象成员，则先构造对象成员再构造当前对象
   1. 继承和派生类
8. 友元、构造函数、析构函数不能继承。
9. 重写和重新定义都是派生类中首部完全一样的函数，但是基类中该函数有virtual就是重写。重写是运行时多态。
10. 访问权限问题

对于外部，公有派生时基类的公有成员（函数）可以访问、私有成员（函数）不能访问。私有派生时基类的公私有成员（函数）均不能访问。

对于派生类成员，无论何种派生方式，基类的所有公有成员都可以被派生类的成员访问，基类所有的私有成员都不能被派生类成员访问，但可以通过基类的公有成员函数访问。

保护成员：可以被派生类成员访问但不能被外部访问



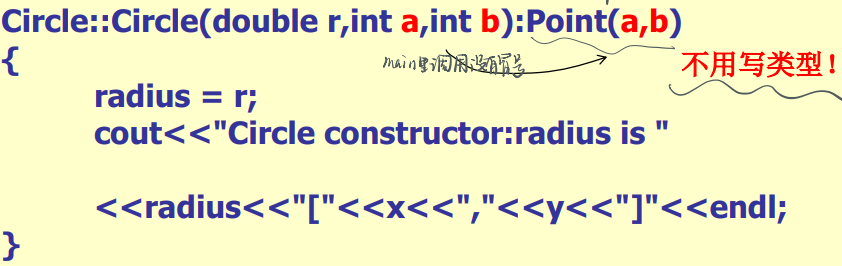
1. 派生类的构造函数和析构函数

定义派生类对象时需要调用基类构造函数初始化基类成员，有派生类的构造函数初始化派生类成员。

执行顺序：创建派生类对象时先调用基类构造函数再调用派生类构造函数；撤销派生类对象时先调用派生类析构函数再调用基类析构函数。

注意执行构造函数不是为了创建对象，而是为了初始化对象。

传参问题：基类构造函数没有参数时派生类可以不传参，甚至可以不定义构造函数。基类构造函数有参数时必须显式调用基类构造函数，用参数表。



1. 多重继承

一般情况下不要重新定义继承来的非虚函数而采用重写继承来的虚函数。重新定义后继承来的原函数被隐藏，不过可以用基类：：函数名调用。但是这样不符合面向对象的思想，因为基类最好看作一个黑盒。

多重继承的方式是

|  |
| --- |
| class 类名:继承方式1 基类1，继承方式2 基类2，……  {}； |

构造时

|  |
| --- |
| Z(int sa,int sb,int sc):X(sa),Y(sb) //Z的构造函数  { c=sc; } //并调用基类构造函数 |

多重继承具有模糊性（菱形继承）

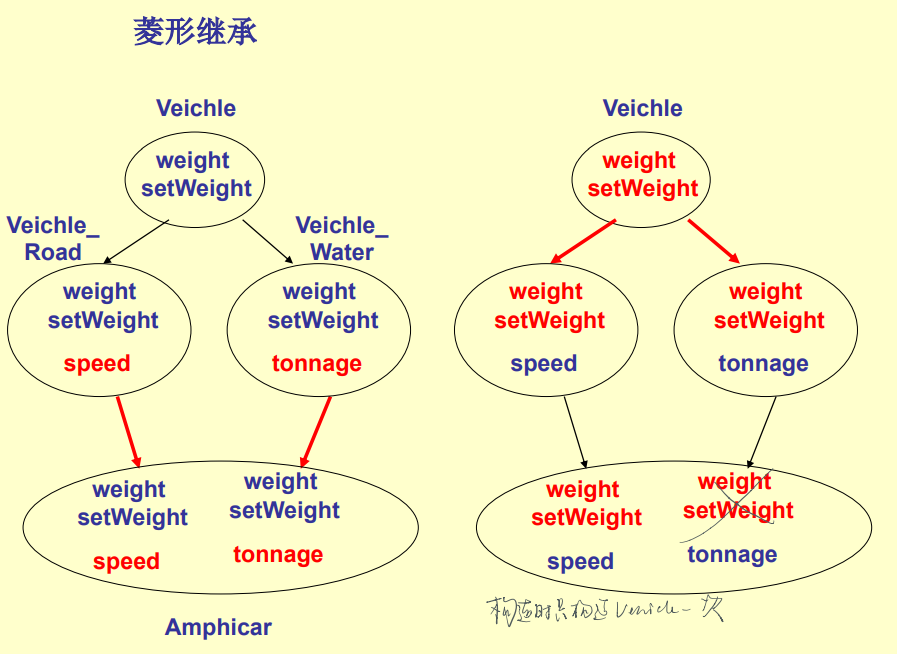
|  |
| --- |
| class Amphicar : public Veichle\_Road, public Veichle\_Water  {  public:  //多重继承，调用基类构造函数  Amphicar(double w, int s, float t)  : Veichle\_Road(w,s), Veichle\_Water(w,t)  //正确, 分别初始化两个w，系统能分清楚两个w  {  cout<< “ Amphicar constructor”<<endl;  }  void display( ) //显示Amphicar成员  {  cout<<“ weight: ”<< weight<<“,”<<endl; //错误,因为多继承带来的歧义性  cout<<" speed:"<<speed<<endl;  cout<<" tonnage:" << tonnage<<endl;  }  } |

可以用作用域运算符饮鸩止渴[[3]](#footnote-3)，但最好还是用**虚拟继承。**

1. **虚拟继承**

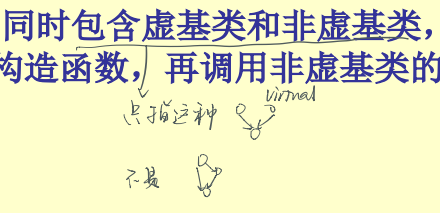
**虚拟继承使公共基类的成员只产生一份拷贝，该公共基类为虚基类[[4]](#footnote-4)。**

**虚拟继承是在派生方式处注明的。虚基类只构造一次。**

****

**注意：**

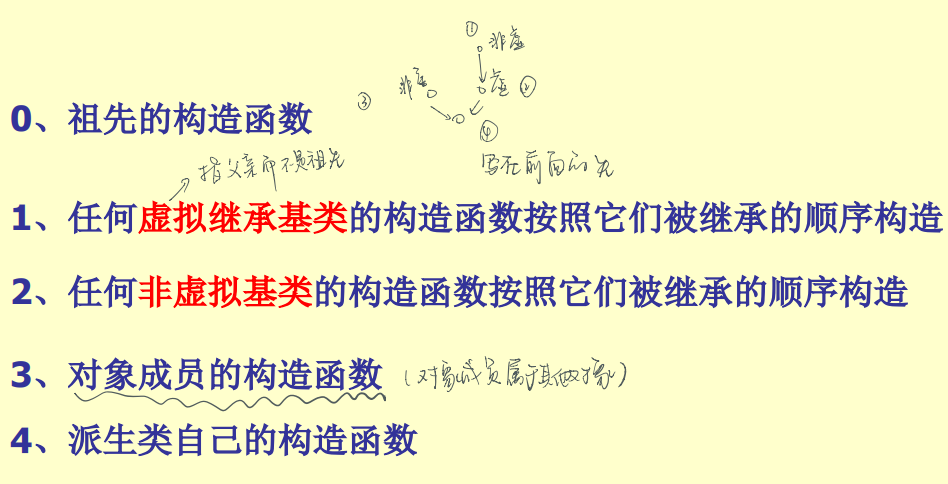
**如果虚基类的构造函数带参数，那么所有直接或间接派生的类中必须用初始化列表的方式调用虚基类的构造函数。但只有建立对象的派生类的构造函数真的调用虚基类的构造函数，其他对构造函数的调用被忽略（但还是要写！）从而保证虚基类子对象值初始化一次。**

**如果同一个继承层次（如图）中同时包含虚基类和非虚基类，则先调用虚基类构造函数后调用非虚基类的构造函数。**

**如果虚基类是由非虚基类派生而来，则同样先调用祖先构造函数再调用这个虚基类的构造函数。**

**能不用多重继承就不用多重继承（二义性）。**

**构造顺序总结：**

****

* 1. 多态性Polymorphism

1. 编译时多态和运行时多态

**重载Overload**是同样的接口完成不同的操作，即不同函数使用同一个函数名

**重写Override**是不同类的对象对同一消息做出不同相应

**联编Binding（或绑定）**是一个程序自身彼此关联的过程

**编译时多态由函数重载和运算符重载实现、运行时多态由继承和虚函数实现。**

1. 运算符重载

[friend]类型[类名::]operator(参数)

|  |
| --- |
| #include <iostream>  using namespace std;  class Integer //大写  {  public:      Integer(int i=0)      { val=i; }      Integer operator+(const Integer& a) //重载运算符+      {      return Integer(val+a.val);      }      void print( )const      {          cout<<"value="<<val<<endl;      }  private:      int val;  };  void main( )  {      Integer a(5),b(3),c;      cout<<"object a ";      a.print();      cout<<"object b ";      b.print();      c=a+b;      cout<<"object c ";      c.print();  } |

运算符的运算顺序、优先级、目数都不改变，有一些符号不能重载，如

. \* :: ?: # ## sizeof

运算符重载包括成员运算符重载（重载函数定义为成员函数）和友元运算符重载（重载函数定义为友元函数）。注意定义的顺序

|  |
| --- |
| 声明的格式:  class X ｛  …  返回 operator@（参数表）;  }; 可以写成内联形式  定义的格式:  返回 X ::operator@（参数表）  {  函数体  }  @是要重载的运算符 |

成员运算符重载函数属于一个类；可以用this指针；双目运算符则参数表中有一个参数，单目运算符则不要参数（隐含this）。

友元运算符函数不属于任何类对象；没有this指针；双目运算符则有两个参数，单目运算符则有一个参数。

1. 值返回和引用返回

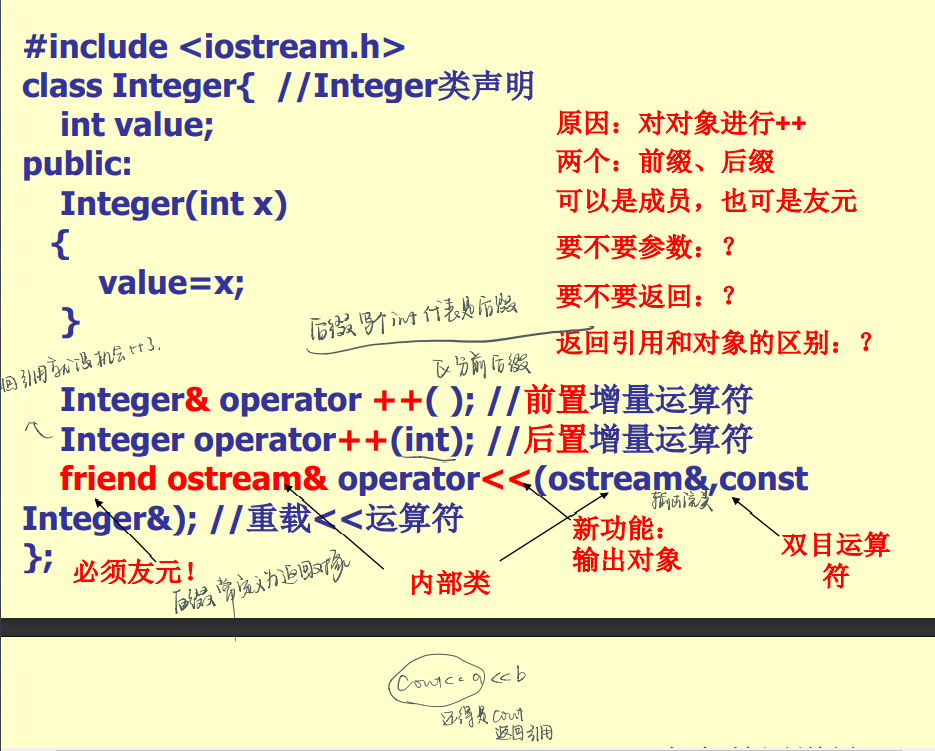
采用值返回的方式则先将return的内容复制给rvalue，再将rvalue赋值给左值；采用引用返回则返回对象的引用，没有复制的过程。

注意：值返回的函数都不能作左值，即不能修改。因此连用的运算符一般必须采用引用返回。引用返回可以返回参数（若参数也是引用型）；借助this指针返回（友元则不行，因为没有this指针）；返回static对象（在静态区）；new一个对象返回（动态开辟的在堆区）。

当运算符函数是一个成员函数时，最左边的操作数 (或者只有左边的操作数)必须是运算符类的一个对象（或者是该类的一个引用）。如果左边的操作数是一个不同类的对象或者是一个内部类的对象，该运算符函数必须作为一个友元函数实现。

注意后缀++不能再++；注意重载()、[]、->等赋值运算符必须采用成员函数实现。流运算符<<只能用友元函数实现，因为<<的对象是不同类型的。

前缀++和后缀++的区别：



1. 转换运算符重载

对于用户自定义类型，可以通过（1）构造函数；（2）类型转换函数进行类型转换。类型转换函数就是一个构造函数。

|  |
| --- |
|  |

类型转换运算符可重载

|  |
| --- |
|  |

类型转换函数声明的一般格式为: operator 类型名( );

(1) 类型转换运算符将对象转换成类型名规定的类型。转换形式就像强制类型转换一样。

(2)如果没有重载类型转换运算符，直接使用强制转换是不行的。因为强制转换运算符只能对基本数据类型进行操作，对自定义类型的操作没有定义

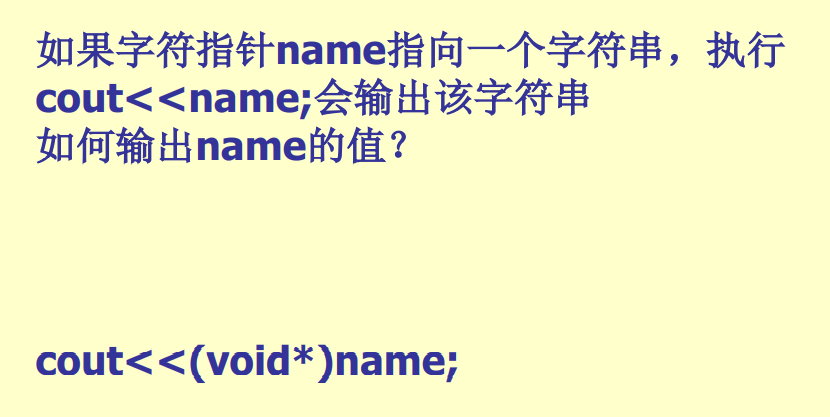
(3) 没有返回类型，因为类型名就代表了它的返回类型，故返回类型显得多余。

(4)类型转换运算符重载函数没有参数，因为单目

(5)类型转换运算符重载函数只能定义为类的成员函数，不能定义为类的友元函数

(6)类型转换运算符重载函数中必须把目的类型的数据作为函数的返回值

(7)一个类可以定义多个类型的类型转换函数，C++ 编译器根据操作数的类型自动选择一个合适的类型转换函数与之匹配，在可能出现歧义的情况下，必须显式地使用相应类型的类型转换函数进行转换



只有字符指针有这个问题

1. 虚函数

对象指针的问题：基类指针可以指向派生类的对象，而派生类指针不能指向基类对象。但是基类指针指向派生类对象只能看到积累部分。私有继承时也不能指向派生类对象。

虚函数就是在基类中被关键字virtual说明，在派生类中重新定义的函数 在派生类中重新定义时，其函数原型，包括返回类型、函数名、参数个数与参数类型的顺序， 都必须与基类中的原型完全相同

纯虚函数在基类中没有定义，仅作为公共接口。含有纯虚函数的被称为抽象类。抽象类不能实例化。

抽象类不能用作参数类型、函数返回类型或进行显式类型转换 的类型。一般声明为抽象类的指针或引用，该指针指向它的派生类，从而实现多态性。设计抽象类的目的是为了多态地使用它的成员函数。

如果在派生类中，并没有对其抽象基类中的纯虚函数进行定义，则该虚函数在派生类中仍为纯虚函数，而该派生类也是一个抽象类

引用，包括值传递与引用传递、值返回与引用返回

附：一些问题

1. 0的区别



1. 强制类型转换

C++引入新形式的强制类型转换

C语言中： int i=10; float x= (float)i;

C++中： int i=10; float x= float(i); //类型名像函数名一样使用

两种方法在C++都能够使用，但推荐使用后面的一种形式（float是类，构造函数）

1. 内联函数消除了#define的不安全性（#define不会检查）。 [↑](#footnote-ref-1)
2. 一般对象的成员应该用对象名.成员，但是静态成员全类对象公用，可以直接用类名::静态成员。 [↑](#footnote-ref-2)
3. 如果还要用祖先的代码，为什么要继承父类？ [↑](#footnote-ref-3)
4. 虚基类和虚函数毫无关系。 [↑](#footnote-ref-4)