|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 函数类型 | 名称 | 参数 | 定义方式 | 使用方式 | 提示 | 复制/删除 |
| 构造函数 | 一般构造函数 | 有 | 显式 | 类名 对象名（初始化式）; | 1、使用初始化列表可以指定用哪个参数初始化哪一个成员  2、可以重载或拥有默认参数（默认参数只能在类体内指定）  3、若该类为另一个类的成员，声明时使用的构造函数用大括号 | / |
| 默认构造函数 | 无 | 显式 | 类名 对象名; | 是自己指定的构造函数 | / |
| 合成默认构造函数 | 无 | 隐式 | 类名 对象名; | 1、是编译器自动生成的构造函数  2、若无显式指定默认构造函数，则编译器自动生成 | / |
| 复制构造函数 | 有 | 显式 | 1、类名 对象名（对象）;  2、类名 对象名1=对象名2 | 1、从无到有地创建对象  2、即使未显式指定默认构造函数，编译器仍然自动生成  3、只能使用同类对象来初始化另一个对象 | 可以使用深复制（需进行动态内存申请） |
| 合成复制构造函数 | 无 | 隐式 | 浅复制 |
| 析构函数 | 一般析构函数 | 无 | 显式 | 一般情况下在超出作用域时自动调用 | 1、若无显式指定析构函数，则编译器自动生成  2、一个类只能有一个析构函数  3、先构造的对象后析构（用进出栈理解）  4、一般情况下，析构函数是用来释放构造对象时获得的资源（如申请到的动态内存） | 删除指针成员指向的对象 |
| 合成析构函数 | 隐式 | 不删除指针成员指向的对象 |

对象指针：

class Data {

public:

typedef unsigned int index;

char get() const;

char get(index st, index eb) const;

string content;

index cursor, top, bottom;

};

1、指向对象：

Data d, \* p = &d;

2、指向类的数据成员

string Data::\* ps = &Data::content;

3、指向类的成员函数（只指向函数名，重载时也可使用指针间接访问）

char (Data:: \* pmf)()const = &Data::get;

若将字符串赋值给数据成员content，有四种方式访问：

|  |  |  |
| --- | --- | --- |
|  | 直接访问对象 | 间接访问对象 |
| 直接访问数据成员 | d.content = "字符串"; | p->content = "字符串"; |
| 间接访问数据成员 | d.\*ps = "字符串"; | p->\*ps = "字符串"; |

若调用成员函数get()并返回给字符变量c，亦有四种方式访问：

|  |  |  |
| --- | --- | --- |
|  | 直接访问对象 | 间接访问对象 |
| 直接访问成员函数 | char c = d.get(); | char c = p->get(); |
| 间接访问成员函数 | char c = (d.\*pmf)(); | char c = (p->\*pmf)(); |

4、this指针

①改写成员函数

源代码：void Point::set(int a, int b) { x = a, y = b; }

编译器改写：void Point::set(Point\* const this, int a, int b) { this->x = a, this->y = b; }

②改写函数调用

源代码：one.set(10, 10);

编译器改写：Point::set(&one, 10, 10);

③使用this指针的时候

（1）在类的非静态成员函数中返回类对象本身的时候，直接使用return \*this；

（2）当参数与数据成员名相同时，如this->n = n （不能写成n=n）。

④this指针有const限定，形式：类名\* const this，指向是不允许改变的

类的作用域：

1、静态成员、类中定义的类型成员需要直接通过类作用域运算符“::”来访问（嵌套类也是如此）

2、在外围类的public部分定义的嵌套类定义了可在任何地方使用的类型；在外围类的protected部分定义的嵌套类定义了只能由外围类、友元或派生类访问的类型；在外围类的private部分定义的嵌套类定义了只能被外围类或其友元访问的类型。

嵌套类可以直接引用外围类的静态成员、类型名和枚举成员，当然，引用外围类作用域之外的类型名或静态成员，需要作用域运算符（::）。

|  |  |  |  |
| --- | --- | --- | --- |
|  | 常对象 | 常数据成员 | 常成员函数 |
| 定义形式 | 1、类名 const 对象名  2、const 类名 对象名 | const 数据成员类型 数据成员名 | 返回类型 函数名（形参列表） const  {  函数体  }  （类体外定义也可） |
| 提示 | 1、数据成员均为常变量  2、所有数据成员需要初值，只能通过构造函数赋值  3、在常对象中指定个别变量可变，可以在类声明中用关键字mutable | 1、任何函数都不允许修改常数据成员的值  2、常数据成员只能通过构造函数初始化列表进行初始化 | 1、常成员函数对所有数据成员可以访问，无法修改  2、常对象只能调用常成员函数  3、由2可知，常成员函数不能调用另一个非常成员函数 |

常对象、常数据成员、常成员函数：

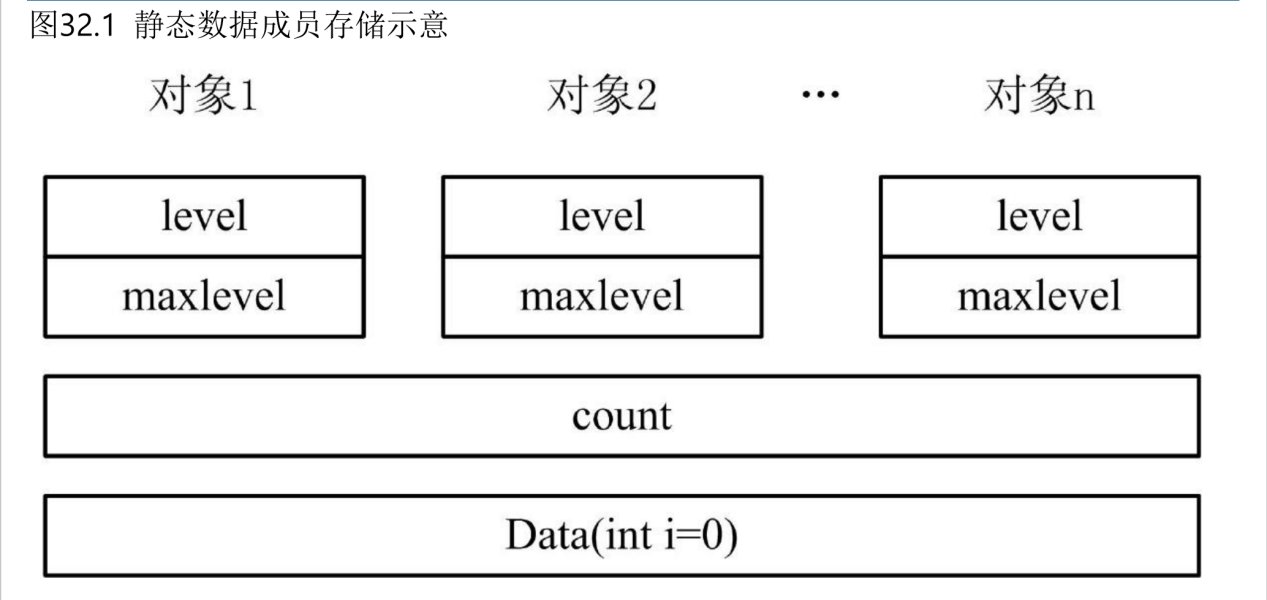
const限定访问关系表：

指向对象的常指针、指向常对象的指针、对象的常引用：

|  |  |  |  |
| --- | --- | --- | --- |
|  | 指向对象的常指针 | 指向常对象的指针 | 对象的常引用 |
| 定义形式 | 类名\* const 指针变量名=对象地址 | const 类名\* 指针变量名=对象地址 | const 类名& 引用变量名 |
| 提示 | 1、指针始终保持其初值，程序中不能修改其指向  2、常指针所指向的对象却不  一定是const的 | 1、指针指向可以改变（可以在定义时不赋初值），但不能通过它改变所指向的对象的值  2、常对象只能被该类指针指向  3、该类指针却可指向任意对象，但被指向的对象不能通过指针改变自身 | 1、与指向常对象的指针一样，常用作函数参数  2、在1的情况下，可以保证数据安全，同时节省时间和空间的开销 |

静态数据成员：

存储方式：

说明：

1、只要在类中定义了静态数据成员，即使不定义任何对象，也为静态数据成员分配空间，它可以在尚未建立对象时就被引用（因为其独立于非静态数据成员存储）。

2、静态数据成员必须在类外部定义一次，且仅有一次，同时进行初始化，初始化方式为：数据成员类型 类名::静态数据成员名=初始化式（不能通过构造函数初始化）。

静态成员函数：

声明方式：

static 返回类型 函数名（形参列表）

{

函数体

}

说明：

1、如果要在类外调用公有的静态成员函数，可以用类作用域运算符（::）和通过对象名调用静态成员函数。

2、非静态成员函数有this指针，而静态成员函数没有this指针。因此，静态成员函数不能访问本类中的非静态成员。静态成员函数就是专门为了访问静态数据成员的。

3、静态成员函数不能被声明为const。

4、类的非静态成员函数可以调用用静态成员函数，但反之不能。

友元函数：

定义方式（类内声明）：

1、为本类的友元函数：

friend 返回类型 函数名（形参列表）;

2、为其他类的友元函数：

friend 返回类型 其他类名::函数名（形参列表）;

友元类：

定义方式（以类A、B为例）：

class B; //提前声明B，以便后续声明友元（不能少！）

class A {

…

friend B;

};

class B { //友元类

…（可访问A类的所有成员）

};

说明：

1、友元的关系是单向的而不是双向的，且不能传递和继承。

2、友元的使用有助于数据共享，能提高程序的效率，但也不要滥用，要在数据共享与信息隐蔽之间选择一个恰当的平衡点。

派生类：

定义方式：

class 派生类名:类派生列表 {

成员列表

};

类派生列表：

指定了一个或多个基类，形式为：访问标号 基类名

访问标号表示继承方式，有public、protected、private三种

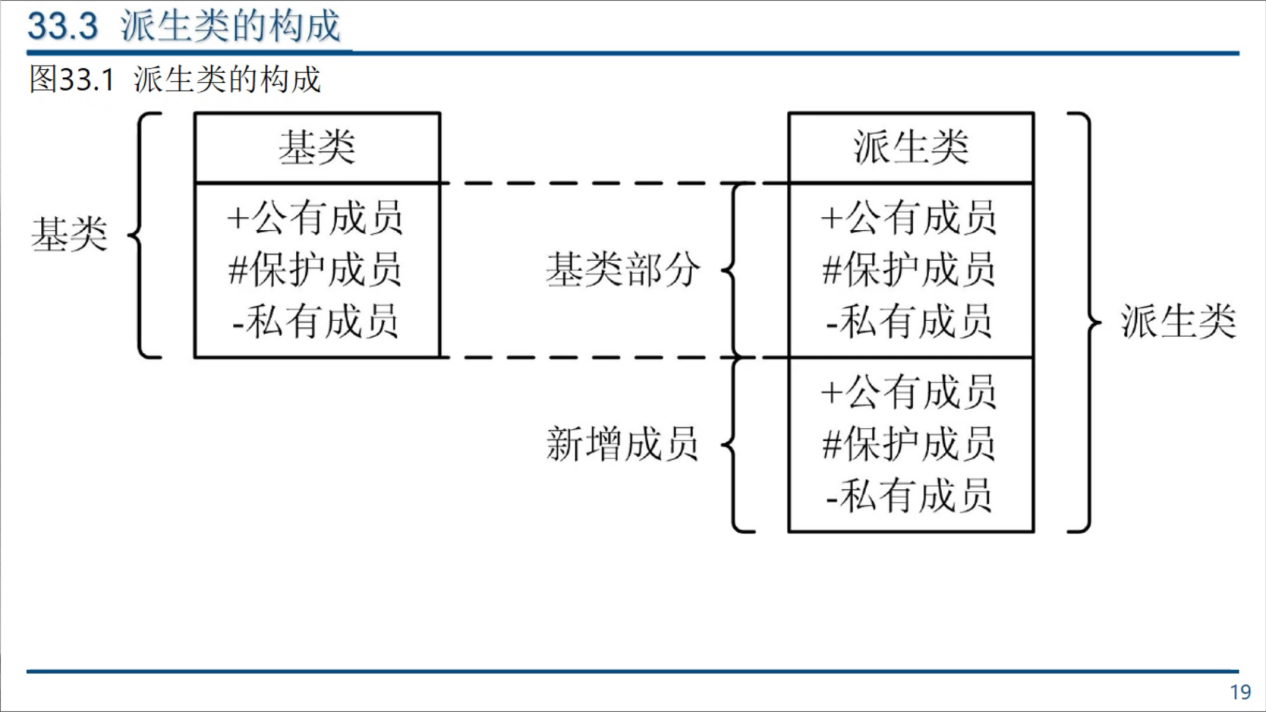
说明：

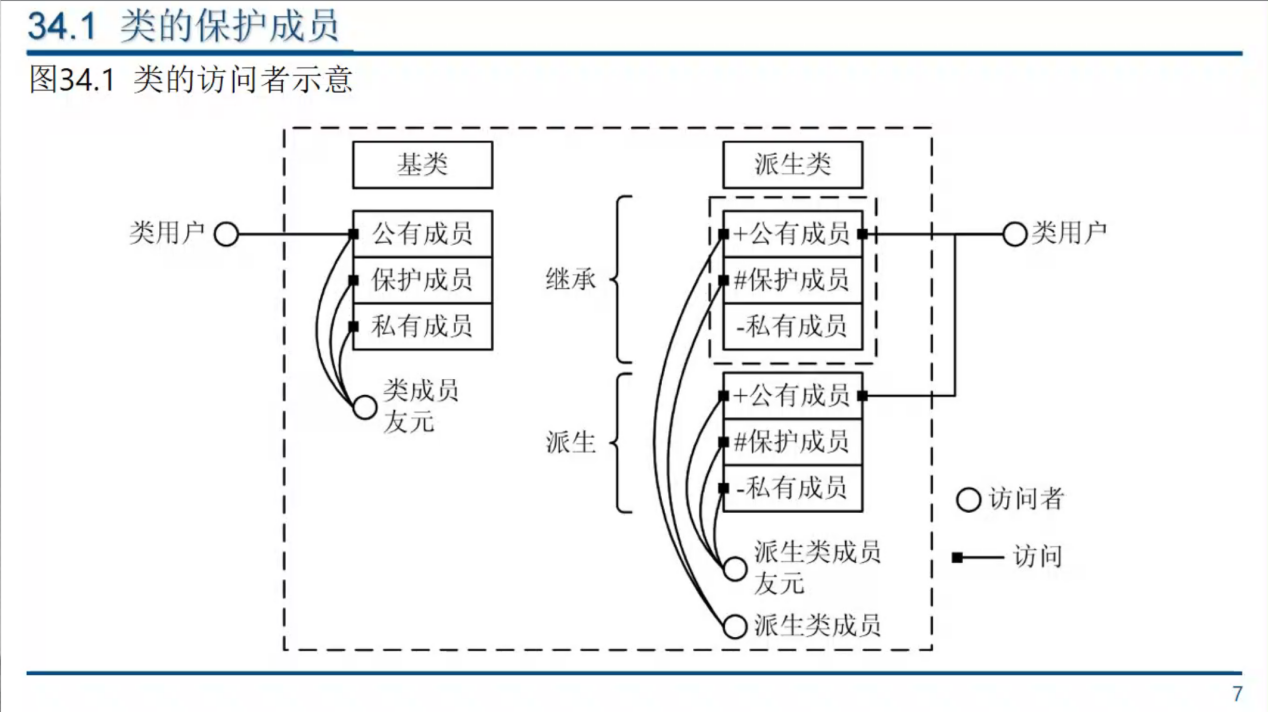
1、用作基类的类必须是已定义的类，这一规则说明不可能从类自身派生出一个类。

2、友元关系不能继承。

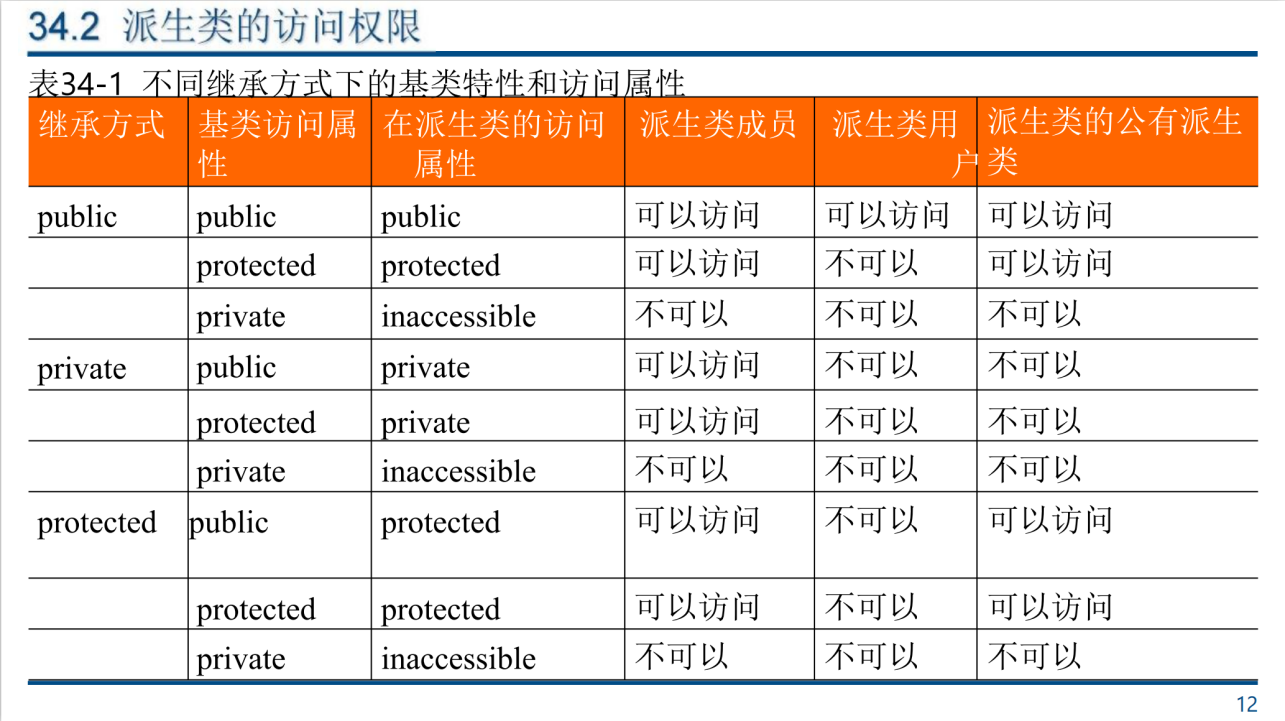
3、无论从基类派生出多少个派生类，每个静态成员只有一个实例。

4、可以在派生类中声明一个与基类成员同名的成员，则派生类中的新成员会覆盖基类的同名成员，就实现了修改基类成员功能的效果。

构成：

访问者示意：

不同继承方式对于基类成员在派生类中访问属性的变化：

多级派生的情况下，保护继承和私有继承会进一步地将基类的访问权限隐蔽成不可访问的。而基类成员的访问权限本在定义的时候就已经被人为设计好了，所以一般地，保护继承与私有继承在实际编程中是极少使用的，它们只在技术理论上有意义。

赋值兼容规则：

定义：需要基类对象的任何地方，都可以使用公有派生类的对象来替代。

替代情况：

1、派生类的对象可以赋值给基类对象；

2、派生类的对象可以初始化基类的引用；

3、派生类对象的地址可以赋给指向基类的指针。

例：（Base为基类，Derive为派生类）

Base b, \* pb;

Derive d;

b = d; //派生类对象赋值给基类，复制基类继承部分

Base& rb = d; //基类引用到派生类对象

pb = &d; //基类指针指向派生类对象

派生类构造函数：

派生类名(形式参数列表) : 基类名(基类构造函数实参列表),派生类初始化列表 {

派生类初始化函数体

}

基类名(基类构造函数实参列表)即调用基类的构造函数

派生类初始化列表可以初始化新增数据成员（尽量在此初始化）

多重继承：

二义性问题：多重继承时，多个基类可能出现同名的成员。在派生类中如果使用一个表达式的含义能解释为可以访问多个基类的成员，则这种对基类成员的访问就是不确定的，称这种访问具有二义性。C++不允许这种访问存在。

解决方法：访问同名成员时使用类作用域限定符。

形式：

对象名.基类名::成员名

对象指针名->基类名::成员名

虚基类：

含义：在继承间接共同基类时只保留一份成员。

定义形式：（在派生类定义中的继承方式处声明）

class 派生类名: virtual 访问标号 虚基类名,... {

成员列表

};

注意：为了保证虚基类在派生类中只继承一次，应当在该基类的所有直接派生类中声明为虚基类。否则仍然会出现对基类的多次继承。

初始化：

1、如果在虚基类中定义了带参数的构造函数，而且没有定义默认构造函数，则在其所有派生类(包括直接派生和间接派生)中，都要通过构造函数的初始化表对虚基类进行初始化。

2、在最后的派生类中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。

例：

class A { public: A(int) {} }; //定义基类A, A有带参数的构造函数

class B : virtual public A { public: B(int a):A(a) {} };

//对基类A初始化

class C : virtual public A { public: C(int a):A(a) {} };

//对基类A初始化

class D : public B,public C { public: D(int a):A(a),B(a),C(a) {} };

多态性：

含义：同样的消息被不同类型的对象接收时导致不同的行为。

1、重载多态

函数重载、运算符重载

2、强制多态

即类型转换

3、类型参数化多态

将类作为函数或类的参数、函数模板和类模板

4、包含多态

虚函数

静态联编：在编译阶段就将函数实现和函数调用绑定起来称为静态联编（绑定）（static binding）。它对函数的选择是基于指向对象的指针（或者引用）的类型。

动态联编：在程序运行的时候才进行函数实现和函数调用的绑定称为动态联编（dynamic binding）。它对函数的选择是基于指针指向对象的（或者引用）的类型。

虚函数：

定义形式；

virtual 函数类型 函数名()

{

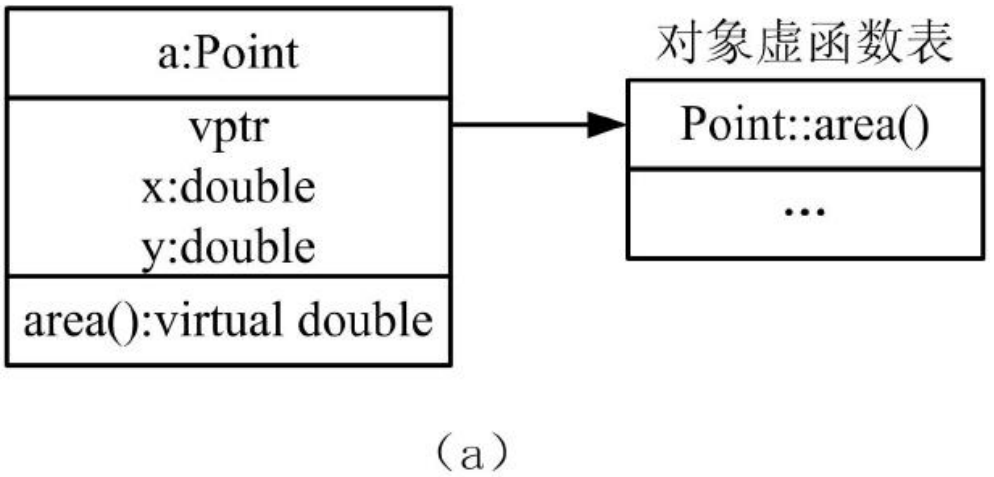
函数体

}

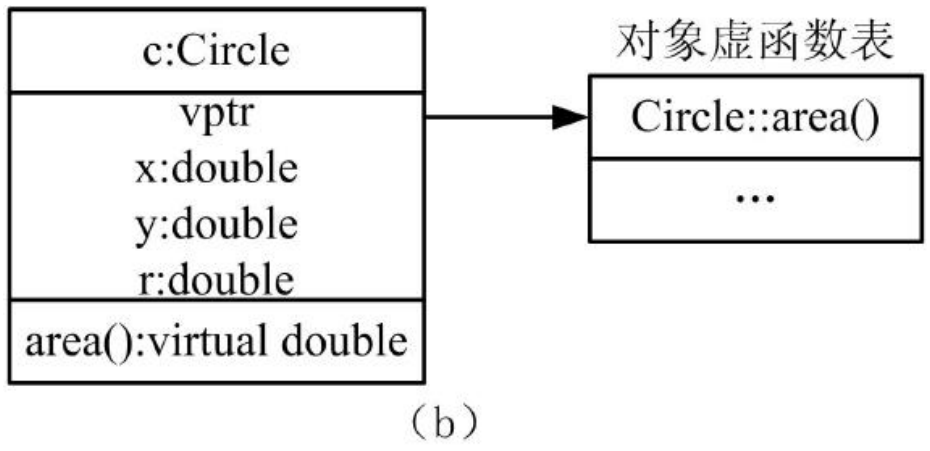
原理：

当编译器编译含有虚函数的类时，将为它建立一个虚函数表VTABLE（virtual table），它相当于一个指针数组，存放每个虚函数的入口地址。编译器为该类增加一个额外的数据成员，这个数据成员是一个指向虚函数表的指针，通常称为vptr。

Point类只有一个虚函数area，所以虚函数表里只有一项。如图（a）是Point对象UML示意，vptr指向虚函数表成员Point::area()。



如果派生类Circle没有重写这个虚函数area，则派生类的虚函数表里的元素所指向的地址就是基类Point的虚函数area的地址。如果派生类Circle重写这个虚函数area,这时编译器将派生类虚函数表里的vptr指向Circle::area()，即指向派生类area虚函数的地址。如图（b）



当调用虚函数时，先通过vptr找到虚函数表，然后再找出虚函数的真正地址，再调用它。

派生类能继承基类的虚函数表，而且只要是和基类同名（参数也相同）的成员函数，无论是否使用virtual声明，它们都自动成为虚函数。如果派生类没有改写继承基类的虚函数，则函数指针调用基类的虚函数。如果派生类改写了基类的虚函数，编译器将重新为派生类的虚函数建立地址，函数指针会调用改写以后的虚函数。