C++程序设计精要教程

华中科技大学

第6章 继承与构造

◆6.1 单继承类

- 继承是C++类型演化的重要机制，在保留原有类的属性和行为的基础上，派生出的新类可以有某种程度的变异。
- 通过继承，新类自动具有了原有类的属性和行为，因而只需定义原有类型没有的新的数据成员和函数成员。实现了软件重用，使得类之间具备了层次性。
- 派生类与基类：接受成员的新类称为派生类，提供成员的原有类型称为基类。
- C++既支持单继承又支持多继承。单继承只能获取一个基类的属性和行为。多继承可获取多个基类的属性和行为。
- 单继承是只有一个基类的继承方式。

第6章 继承与构造

◆6.1 单继承类

- **单继承的定义格式：**
  - class <派生类名>:<继承方式><基类名>
    {
    <派生类新定义成员>
    <派生类重定义基类同名的数据和函数成员>
    <派生类声明修改基类成员访问权限>
    };
  - <继承方式>指明派生类采用什么继承方式从基类获得成员，分为三种：private表示私有继承基类；protected表示保护继承基类；public表示公有继承基类。
- 注意区别继承方式（派生控制）和访问权限。派生控制和类的成员访问控制符的区别：派生控制作用于基类成员，类的成员访问控制符作用于当前类自定义的成员。

第6章 继承与构造

【例6.1、6.2】分别定义定位坐标LOCATION类和其派生的点POINT类。

```
#include <graphics.h>
class LOCATION{                  //定义定位坐标类
   int x, y;
public:
   int getx( ); int gety( );     //gety( )获得当前坐标y
   void moveto(int x,int y); //定义移动坐标函数成员
   LOCATION(int x,int y);
   ~LOCATION( );
};
void LOCATION::moveto(int x,int y){
   LOCATION::x=x;
   LOCATION::y=y;
}
int LOCATION::getx( ){ return x; }
int LOCATION::gety( ){ return y; }
```

第6章 继承与构造

```
LOCATION::LOCATION(int x,int y){
   LOCATION::x=x;   LOCATION::y=y;
}
LOCATION::~LOCATION( ){ }

class POINT:public LOCATION{
//定义点类，从LOCATION类继承，继承方式为public
   int visible;                              //新增可见属性
public:
   int isvisible( ){ return visible; }       //新增函数成员
   void show( ),hide( );
   void moveto(int x,int y);                 //重新定义与基类同名函数
   POINT(int x,int y):LOCATION(x,y) { visible=0; } //在构造派生类对象前先构造基类对象
~POINT( ){ hide( ); }
};
```

第6章 继承与构造

```
void POINT::show( ){
   visible=1;
   putpixel(getx( ),gety( ));
}
void POINT::hide( ){
   visible=0;
   putpixel(getx( ),gety( ));
}
void POINT::moveto(int x,int y){
   int v=isvisible( );
   if(v) hide( );
   LOCATION::moveto(x,y);    //不能去掉LOCATION::，会自递归
   if(v) show( );
}
void main(void){
   POINT p(3,6);
   p.LOCATION::moveto(7,8);          //调用基类moveto函数
   p.moveto(9,18);                   //调用派生类moveto函数
}
```

带类名访问基类的moveto函数，如果不带类名会导致无休止的递归调用。

问题：调用基类moveto函数会导致什么问题？

第6章 继承与构造

◆6.1 单继承类

- 多继承的派生类有多于一个的基类，派生类将是所有基类行为的组合。
- 派生类与基类：接受成员的新类称为派生类，如例中的Point类；提供成员的类称为基类，如例中的Location类。
- 基类是对派生类的抽象，提取了派生类的公共特征；而派生类是基类的具体化，通过增加属性或行为变为更有用的类型。
- 派生类可以看作基类定义的延续，先定义一个抽象程度较高的基类，该基类中有些操作并未实现（称为抽象方法，相应的基类为抽象类）；然后定义更为具体的派生类，实现抽象基类中未实现的操作。

第6章 继承与构造

◆6.1 单继承类

- C++通过多种控制派生的方法获得新的派生类，可在定义派生类时：
  - 添加新的数据成员和函数成员；
  - 改变继承来的基类成员的访问权限；
  - 重新定义同名的数据和函数成员（特别是实例函数成员，称为Override）。

第6章 继承与构造

◆6.1 单继承类

- **关于class、struct、union说明：**
  - 用class声明的类的继承方式缺省为private，因此，声明class POINT: private LOCATION等价于声明class POINT: LOCATION。
  - 派生类也可以用struct声明，不同之处在于：用struct声明的继承方式和访问权限缺省为public。
  - 用union声明的类既不能作派生类的基类，也不能作任何基类的派生类。
- 当基类成员被继承到派生类时，该成员在派生类中的访问权限由继承方式决定。必须慎重的选择继承方式，它是面向对象程序设计的一个非常重要的环节

第6章 继承与构造

◆6.2 继承方式

- 派生类可以有三种继承方式：公有继承public、保护继承protected、私有继承private。基类私有成员对派生类函数是不可见的。
  - ①公有继承：基类的公有成员和保护成员派生到派生类时，都保持原有的权限；
  - ②保护继承：基类的公有成员和保护成员派生后都成为派生类的保护成员；
  - ③私有继承：基类的公有成员和保护成员派生后都作为派生类的私有成员。

第6章 继承与构造

◆6.2 继承方式

- 基类成员继承到派生类时，其访问权限的变化同继承方式有关。
  - 假定private < protected < public。如果基类成员的访问权限高于派生控制，则派生后基类成员在派生类中的访问权限和派生控制一样；否则，基类成员的访问权限保持不变(派生控制符的“筛孔效应”)。

成员 \ 派生控制	private	protected	public
private	不可见	不可见	不可见
protected	private	protected	protected
public	private	protected	public

- 继承来的基类私有成员不能被派生类函数成员访问。

第6章 继承与构造

◆6.2 继承方式

- 基类的私有成员同样也被继承到派生类中，构成派生类的一部分（sizeof会计算基类私有实例数据成员），但对派生类函数成员不可见，不能被派生类函数成员访问。
  - 若派生类函数成员要访问基类的私有成员，则必须将其声明为基类的成员友元。
- 在派生类外部，对其成员访问的权限：
  - 对于新定义成员，按定义时的访问权限访问；
  - 对于继承来的基类成员，取决于这些成员在派生类中的访问权限，与其在基类中定义的访问权限无关。

第6章 继承与构造

◆6.2 继承方式

基类LOCATION的成员

private成员：

int x,y;

public成员：

int getx();

int gety();

void moveto( );

LOCATION( );

~LOCATION( );

派生类POINT新增成员

private成员：

int visible;

public成员：

int isvisible( );

void show( );

void hide( );

void moveto( );

POINT( );

~POINT( );

继承方式为public时POINT可访问的成员

private成员：

int visible;

public成员：

int isvisible( );

void show( );

void hide( );

void moveto( );

POINT( );

~POINT( );

int getx( );

int gety( );

void LOCATION::moveto( );

LOCATION( );

~LOCATION( );

第6章 继承与构造

◆6.2 继承方式

- 若类POINT的继承方式为public，基类函数getx、gety派生后的访问权限仍为public，对类POINT来说这是合理的，因为，对类POINT来说则类需要这样的函数成员；
- 同上，若类POINT的继承方式为public，基类函数成员moveto派生后的访问权限为public，对类POINT来说则是不合理的，因为类POINT自己定义了public函数成员moveto。在第6页中，主函数还能调用基类函数LOCATION::moveto。

第6章 继承与构造

class POINT: private LOCATION{….};

基类LOCATION的成员

private成员：

int x,y;

public成员：

int getx( );

int gety( );

void moveto( );

LOCATION( );

~LOCATION( );

派生类POINT新增成员

private成员：

int visible;

public成员：

int isvisible( );

void show( );

void hide( );

void moveto( );

POINT( );

~POINT( );

继承方式为private时POINT可访问的成员

private成员：

int visible;

int getx( );

int gety( );

void LOCATION::moveto( );

LOCATION( );

~LOCATION( );

public成员：

int isvisible( );

void show( );

void hide( );

void moveto( );

POINT( );

~POINT( );

第6章 继承与构造

◆6.2 继承方式

- 继承方式为private时，基类公有成员在派生类中的访问权限变为private。不合理时可以使用“基类名::成员”或“using 基类名::成员”修改某些成员的访问权限，派生类不能再定义同名的成员。

```
class POINT:private LOCATION{ //private可省略
        int visible;
public:
        LOCATION::getx;//修改权限成public，或者using LOCATION::getx;
        LOCATION::gety;//修改权限成public
        int isvisible( ){ return visible; }
        void show( ), hide( );
        void moveto(int x,int y);
        POINT(int x,int y):LOCATION(x,y){ visible=0; }
        ~POINT( ){ hide( ); }
};
```

- 需要指出的是，选用private作继承方式通常不是最好的选择。如果派生类POINT选用private作继承方式，却又未修改LOCATION::getx的访问权限，则getx在POINT类中的访问权限将变为private，从而使其它非派生类成员的函数无法访问private的POINT::getx。

第6章 继承与构造

◆6.2 继承方式

基类LOCATION的成员

private成员：

int x,y;

public成员：

int getx( );

int gety( );

void moveto( );

LOCATION( );

~LOCATION( );

派生类POINT新增成员

private成员：

int visible;

public成员：

int isvisible( );

void show( );

void hide( );

void moveto( );

POINT( );

~POINT( );

修改private派生的基类成员访问权限时

private成员：

int visible;

void LOCATION::moveto( );

LOCATION( );

~LOCATION( );

public成员：

int isvisible( );

void show( );

void hide( );

void moveto( );

POINT( );

~POINT( );

int getx( );

int gety( );

继承方式

- C++恢复访问权限是将派生类继承的基类成员的访问权限复原成和该成员在基类定义时的访问权限一样。派生类不仅可以恢复基类成员的访问权限，还可以改变访问权限（但不可改变基类私有成员访问权限）。

```
class A{
   int a;
protected:
   int b,c;
   int bb;
public:
   int d,e;
   int cc;
  ~A() {};
};
```

```
class B:protected A{
    int a;
    A::cc; // using A::cc;降低
protected:
   int b,f;
public:
   int e,g;
   using A::d; //恢复
   A::bb；//提升
}b;
```

在派生类任何访问权限下用using声明可以改变从基类继承的成员的访问权限

第6章 继承与构造

◆6.2 继承方式

- 基类成员经过继承方式被继承到派生类后，要注意访问权限的变化。
  - 按面向对象的作用域，和基类同名的派生类成员被优先访问。
  - 派生类中改写基类同名函数时，要注意区分这些同名函数，否则可能造成自递归调用。
- 标识符的作用范围可分为从小到大四种级别：①作用于函数成员内；②作用于类或者派生类内；③作用于基类内；④作用于虚基类内。
  - 标识符的作用范围越小，被访问到的优先级越高。如果希望访问作用范围更大的标识符，则可以用类名和作用域运算符进行限定。

第6章 继承与构造

【例6.3】以链表LIST类为基类定义集合类SET。

```
class LIST{
    struct NODE{                    //定义节点类
        int val;       NODE *next;
        NODE(int v, NODE *p){ val=v; next=p; }
        ~NODE( ){if(next) {delete next; next=0;}}
    }*head;                         //定义数据成员
public:
    int insert(int),  contains(int);
    LIST( ){ head=0; }          //0表示空指针
    ~LIST( ){ if(head){ delete head; head=0; //0表示空指针} }
};
int LIST::contains(int v){ //搜索链表，查询是否存在该节点
    NODE *h=head;
    while((h!=0)&&(h->val!=v))  h=h->next;
    return h!=0;                    //0表示空指针
}
```

第6章 继承与构造

```
int LIST::insert(int v){                    //在链表中插入新增节点
    head=new NODE(v,head);     //head指向新的节点，head->next=0
    return 1;
}
class SET:protected LIST{//采用保护继承方式
    int used;                               //集合元素的个数
public:
    LIST::contains;                         //修改contains函数访问权限
    int insert(int);                        //改写insert函数(Override)
    SET( ){ };                              //等价于SET( ):LIST( ){ };
};
int SET::insert(int v){                     //LIST::insert中的LIST不能省略：否则自递归
    if(!contains(v)&&LIST::insert(v))  return  ++used;
    return 0;
}
void main(void) { SET s;      s.insert(3);  s.contains(3); }
```

第6章 继承与构造

- 派生类不能访问基类私有成员，除非将派生类的声明为基类的友元类，或者将要访问基类私有成员的派生类函数成员声明为基类的友元。

```
class B;                    //前向声明类B
class A{
   int a, b;
public:
   A(int x){a=x;}
   friend B;                //声明B为A的友元类，B类成员可以访问A任何成员
};
class B:A{                  //缺省为private继承，等价于class B: private A{
   int b;
public:
   B(int x):A(x){ b=x; A::b=x; a+=3; } //可访问私有成员A::a,A::b
};
void main(void){ B x(7); }
```

第6章 继承与构造

◆6.4 构造与析构

- 单继承派生类的构造顺序比较容易确定：
  - 调用虚基类的构造函数；
  - 调用基类的构造函数；
  - 按照派生类中数据成员的声明顺序，依次调用数据成员的构造函数或初始化数据成员；
  - 最后执行派生类的构造函数构造派生类。
- 析构是构造的逆序。
- 以下情况派生类必须定义自己的构造函数：
  - 虚基类或基类只定义了带参数的构造函数；
  - 派生类自身定义了引用成员或只读成员，且这些成员没有类内就地初始化；
  - 派生类定义了需要使用带参数构造函数初始化的其它类对象成员，且这些成员没有类内就地初始化。

第6章 继承与构造

```
#include <iostream.h>
class A{
   int a;
public:
   A(int x):a(x){cout<<a;}//非const成员a也可在构造函数体内再次对a赋值
   ~A( ){cout<<a;}
};
class B:A{    //私有继承，等价于class B: private A{
   int b,c;
   const int d;//B中定义有只读成员且没有就地初始化，故必须定义构造函数初始化
   A x,y; //x、y的构造必须带参数，且没有就地初始化，故必须定义构造函数初始化
public:
   B(int v):b(v),y(b+2),x(b+1),d(b),A(v){//注意构造次序与成员初始化列表的出现顺序无关
      c=v;      cout<<b<<c<<d;      cout<<“C”; //c=v不是初始化，是重新赋值
   }
   ~B( ){cout<<“D”;}          //派生类数据成员实际构造顺序为b,c, d,x,y
};
void main(void){ B z(1); }          //输出结果：123111CD321
```

第6章 继承与构造

◆6.4 构造与析构

- 如果虚基类和基类的构造函数是无参的，则构造派生类对象时，派生类构造函数可以不用显式调用基类/虚基类的构造函数，编译程序会自动调用虚基类或基类的无参构造函数。
- 如果被引用的对象是用new生成的，则引用变量r必须用delete &r析构对象，否则被引用的对象将因无法完全释放空间（为对象申请的空间）而产生内存泄漏。
- 若被p（指针）指向的对象是用new生成的，则指针变量p必须用delete p析构对象，不能使用不调用析构函数的free(p)，否则将产生内存泄漏。

第6章 继承与构造

【例6.6】被引用的对象的析构。

```
#include <iostream.h>
class A{
   int  i;    int *s;
public:
   A(int x){
      s=new int[i=x];
      cout<<"(C): "<<i<<"\n";
   }
   ~A( ) {
      delete s;
      cout<<"(D): "<<i<<"\n";
   }
};
void sub1(void) {
   A &p=*new A(1);
}//内存泄露
void sub2(void){
   A *q=new A(2);
} //内存泄露
void sub3(void){
   A &p=*new A(3);
   delete &p;
}
void sub4(void) {
   A *q=new A(4);
   delete q;
}
void main(void){
   sub1( );    sub2( );
   sub3( );    sub4( );
}
```

输出：

```
(C): 1
(C): 2
(C): 3
(D): 3
(C): 4
(D): 4
```

第6章 继承与构造

◆6.5 父类和子类

- 如果派生类的继承方式为public，则这样的派生类称为基类的子类，而相应的基类则称为派生类的父类。
- C++允许父类指针直接指向子类对象，也允许父类引用直接引用子类对象。无须通过强制类型转换保持类型相容。编译时按父类说明的成员权限访问成员（编译程序只能根据类型定义静态检查语义，因此编译时把父类指针指向的对象都当作父类对象）。
- 通过父类指针调用虚函数时晚期绑定，根据对象的实际类型绑定到合适的成员函数。
- 父类指针实际指向的对象的类型不同，虚函数绑定的函数的行为就不同，从而产生多态。

第6章 继承与构造

◆6.5 父类和子类

- 编译程序只能根据类型定义静态地检查语义。由于父类指针可以直接指向子类对象，而到底是指向父类对象还是子类对象只能在运行时确定。
- 编译时，只能把父类指针指向的对象都当作父类对象。因此编译时：
  - 父类指针访问对象的数据成员或函数成员时，不能超越父类为相应对象成员规定的访问权限；
  - 也不能通过父类指针访问子类新增的成员，因为这些成员在父类中不存在，编译程序无法识别。

第6章 继承与构造

【例6.7】定义点类，并通过点类派生出圆类。

```
#include <iostream.h>
class POINT{
    int x,y;
public:
    int getx( ){ return x; }
    int gety( ){ return y; }
    void show( ){ cout<< “Show a point\n” ; }
    POINT(int x,int y){ POINT::x=x; POINT::y=y; }
};
class CIRCLE:public POINT{          //公有继承
    int r;                          //私有成员
public:
    int getr( ){ return r; }
    void show( ){ cout<< “Show a circle\n” ; }
    CIRCLE(int x,int y,int r):POINT(x,y){ CIRCLE::r=r; }
};
```

第6章 继承与构造

```
void main(void){
    CIRCLE c(3,7,8);
    POINT *p=&c;//父类对象指针p可以直接指向子类对象，不用类型转换
    cout<< “The circle with radius ” <<c.getr();


    cout << p->getr(); //错误，因为getr()函数不是父类的函数成员编译程序无法通过检查
    cout<< “ is at (” <<p->getx()<< “,” <<p->gety()<< “)\n” ;
    p->show();
    //p虽然指向子类对象，但调用的是父类的show函数（show函数不是虚函数，没有多态性）
}
```

输出结果：

```
The circle with radius 8 is at (3,7)
Show a point
```

父类和子类

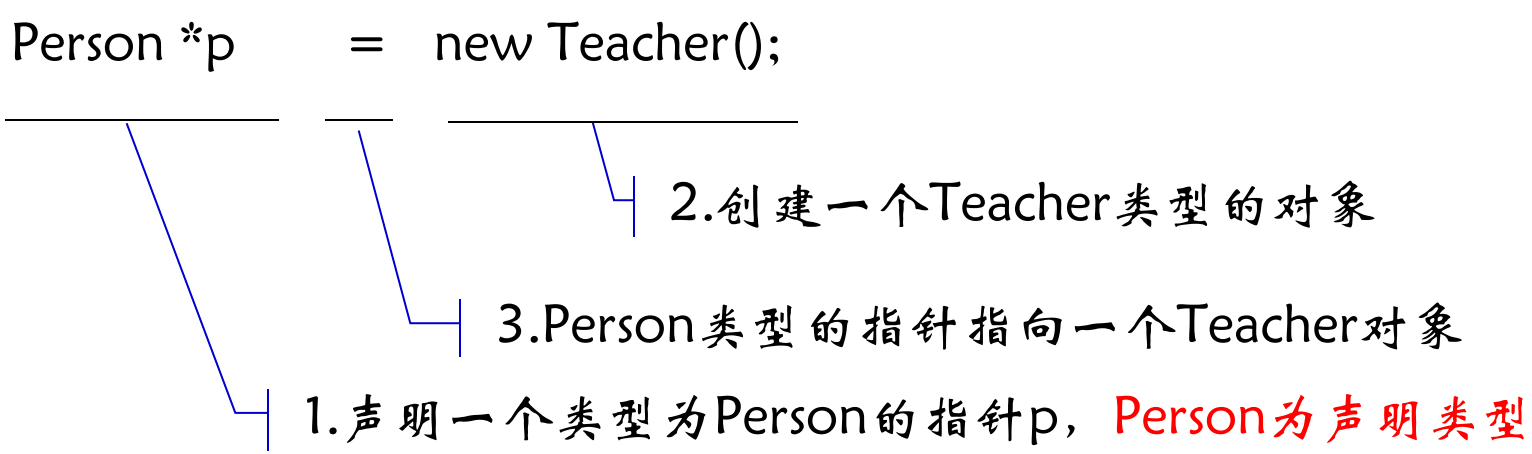
- C++允许父类指针指向子类对象而不用强制转换

```
class Person { public: void f() { cout << “P\n” ;}};
class Teacher: public Person()
{ public: void f(){ cout << “T\n” ;}};
Person *p = new Teacher();//左边父类，右边子类
p->f(); //输出的是P，为什么不输出T?
```

- 为什么父类指针可以直接指向子类对象?
  - 因为子类对象有多个类型，子类本身及父类、父类的父类…。（Teacher是Person）
  - 编译器认为子类指针赋给父类指针类型是匹配的

父类和子类

- 为什么p->f()调用的是Person的f()?
- 首先要理解类型检查（type checking）发生在编译（Compile time）时
- 然后要理解Person *p = new Teacher()的真正涵义



- 但第2步、第三步发生在运行时（Run time）
- 因为编译时程序还没运行，编译器无法知道p会指向什么类型对象，编译器在编译时只能根据变量p的声明类型（Person *）来类型检查
- 因此编译器在编译到p->f()语句时，认为调用的是Person的f()，绑定到Person的f()

父类和子类

- 子类指针不能指向父类对象（子类＝父类）必须强制转换

  Person *p = new Teacher();

  Teacher *t = (Teacher *)p; // Teacher *t = p会出错

或者 Teahcer *t = (Teacher *)new Person(); //必须强制类型转换

- 首
  （

```
if(! strcmp(typeid(*p).name(),"Teacher"))
    Teacher *t = (Teacher *)p;
```

- 因
  编译器
  Teacher类型的指针，类型是不匹配的
- 强制转换Teacher *t = (Teacher *)p的意思是告诉编译器，请不要再做类型检查，风险我自己承担。
- 强制类型转换的风险是：运行时如果p指向的对象不是Teacher的实例时程序会出错（Run Time Error）
- 因此在赋值前必须利用RTTI（运行时类型标识）来检查p是不是指向Teacher的实例

第6章 继承与构造

- 若基类和派生类没有构成父子关系，则：
  - 普通函数里，定义的基类指针不能直接指向派生类对象，而必须通过强制类型转换才能指向派生类对象。
  - 普通函数里定义的基类引用也不能直接引用派生类对象，而必须通过强制类型转换才能引用派生类对象。

【例6.7】引用父类对象的引用变量引用子类对象。

```
#include <iostream.h>
class A{
   int a;
public:
   int getv( ) { return a; }
   A( ) { a=0; }
   A(int x) { a=x; }
   ~A( ){ cout<<"~A\n"; }
};
```

```
class B: A{//非父子：private
   int b;
public:
   int getv( ) {
      return b+A::getv( );
   }
   B( ) { b=0; }//等于B( ):A( )
   B(int x):A(x) { b=x; }
   ~B( ){ cout<<"~B\n"; }
};
```

第6章 继承与构造

```
class C: public A{ //父子关系
    int c;              //私有成员c
public:
    int getv( ) { return c+A::getv( ); }
    C( ) { c=0; }//等价于C( ):A( ) { c=0; }
    C(int x):A(x) { c=x; }
    ~C( ){ cout<<"~C\n"; }
};
void main(void){
    A &p=*new C(3);      //直接引用C类对象：A和C父子
    A &q=*(A *)new B(5);          //强制转换引用B类对象：A和B非父子
    cout<< “p.getv( )=” <<p.getv( )<< “\n” ; //执行的是哪个getv?
    cout<<"q.getv( )="<<q.getv( )<<"\n"; //执行的是哪个getv?
    delete &p;                  //析构C(3)的父类A而非子类C（为什么？析构函数不是虚函数）
    delete &q;                  //析构B(5)的父类A 而非子类B
}
```

输出：

```
p.getv( )=3
q.getv( )=5
~A
~A
```

第6章 继承与构造

- 在派生类函数成员内部，定义的基类指针可以直接指向该派生类对象，即对派生类函数成员而言，基类被等同地当作父类。
- 如果函数声明为派生类的友元，则该友元定义的基类指针也可以直接指向该基类的派生类对象，也不必通过强制类型转换。

【例6.9】定义机车类VEHICLE，并派生出汽车类CAR。

```
class VEHICLE{
   int speed,weight,wheels;
public:
   VEHICLE(int spd,int wgt,int whl);
};
```

第6章 继承与构造

```
VEHICLE::VEHICLE(int spd, int wgt, int whl){
   speed=spd;   weight=wgt;   wheels=whl;
}
class CAR: private VEHICLE{       //非父子关系：private
   int seats;
public:
   VEHICLE *who( );
   CAR(int sd, int wt, int st);
   friend void main( );
};
CAR::CAR(int sd, int wt, int st):VEHICLE(sd, wt, 4) {  seats=st; }
VEHICLE *CAR::who( ){
   VEHICLE *p=this;  //派生类内的基类指针直接指向派生类对象
   VEHICLE &q=*this;//派生类内的基类引用直接引用派生类对象
   return  p;
}
//在派生类的友元main中，基类和派生类构成父子关系
void main(void) { CAR c(1,2,3); VEHICLE *p=&c;}
```

```
class A { };
class B : protected A {
public:
   //以下代码点可以同时访问基类和派生类的公有接口（函数）
   void f() {  A *p = this;  }
   static void g() { A *p = new B(); }
   friend void h() { A *p = new B(); }
};

void test() {
//判断基类指针是否可以直接指向派生类，就要看当前代码点是否可以同时访问基类和派生类的公有接口（函数）。对于非父子关系，当前代码点不能访问基类的公有接口，因此基类指针不能直接指向派生类而必须强制转换
   A *p1 = (A *)(new B); //对于非父子关系，可以用C style casting
   delete p1;
}
```

```
class A {
//          virtual void f() {};
};

class C:public A{
public:
            C( ) = default;
};

void test() {
            A *p3 = new C;

            //错误.运行时dynamic_cast的操作数p3必须是多态类型。
            //因为A没有虚函数，也就没有虚函数表，因此A不是多态类型.
            //若在A里加上虚函数，这个语句就可以
            C *p4 = dynamic_cast<C *>(p3); // 父类向子类转换，必须具有多态性

}
```

dynamic_cast使用注意事项