C++语言程序设计

第七章　继承与派生

本章主要内容

- 类的继承
- 类成员的访问控制
- 单继承与多继承
- 派生类的构造、析构函数
- 类成员的标识与访问
- 深度探索

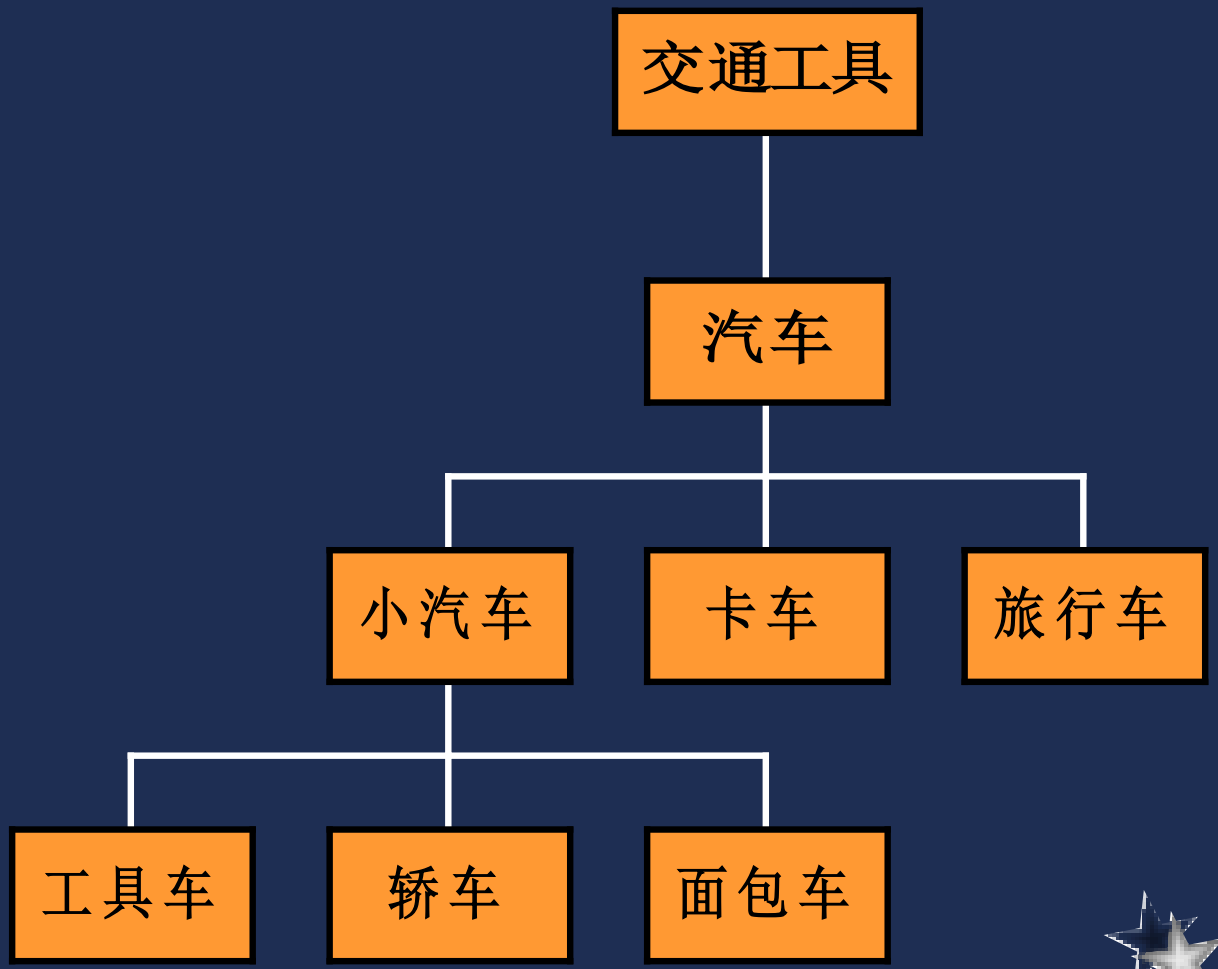
类的继承与派生

类的继承与派生

- 保持已有类的特性而构造新类的过程称为继承。
- 在已有类的基础上新增自己的特性而产生新类的过程称为派生。
- 被继承的已有类称为基类（或父类）。
- 派生出的新类称为派生类（或子类）。

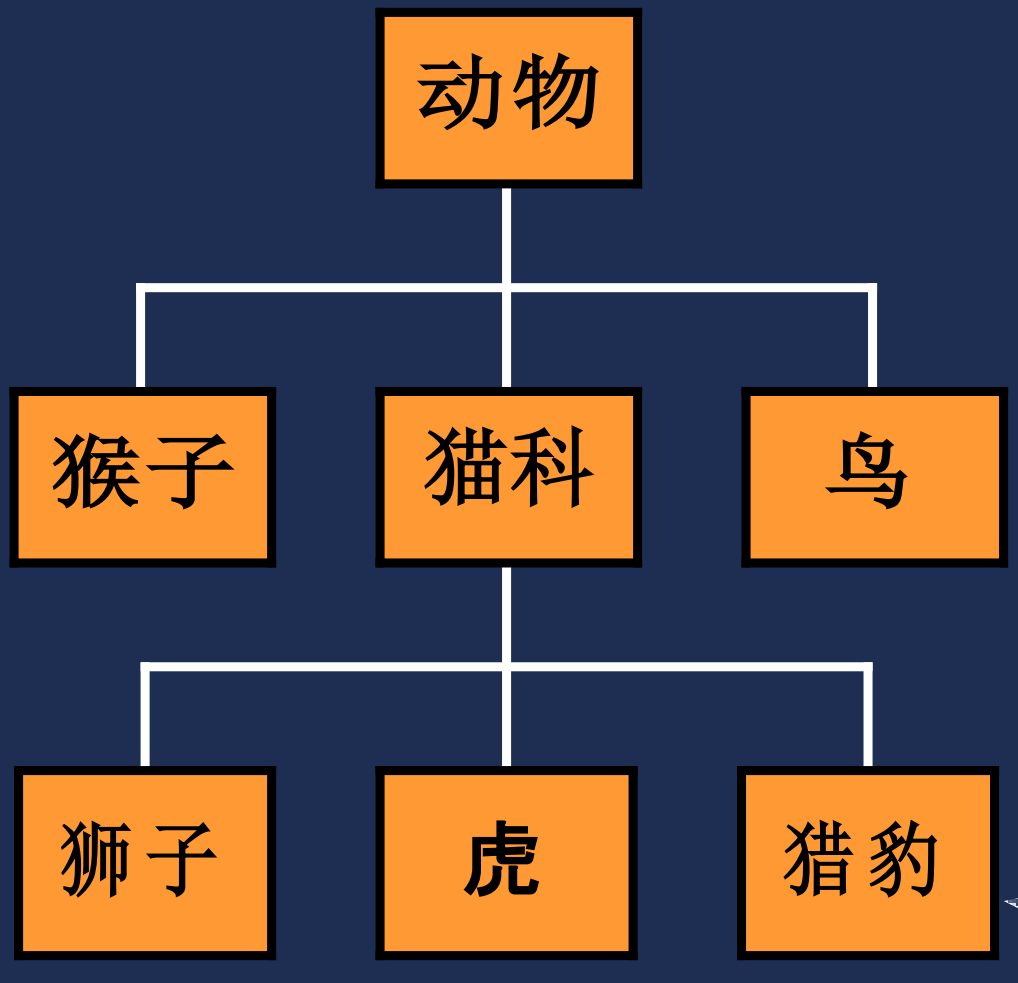
继承与派生问题举例

类的继承与派生



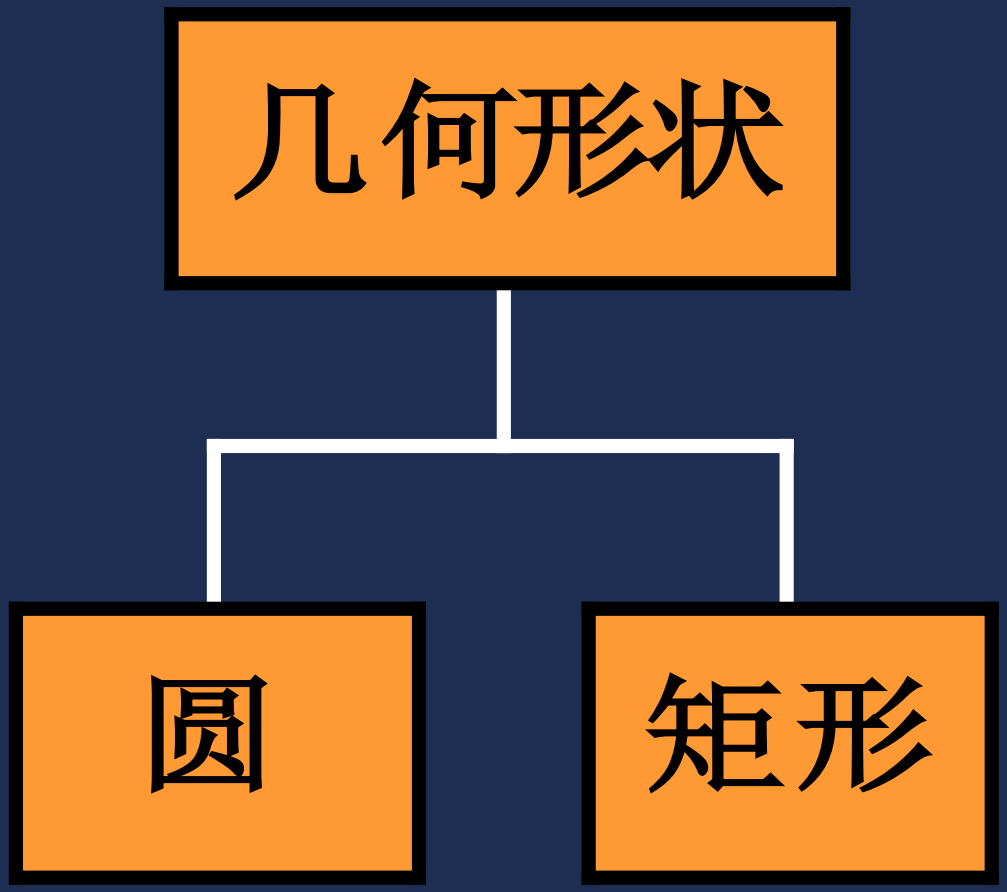
继承与派生问题举例

类的继承与派生



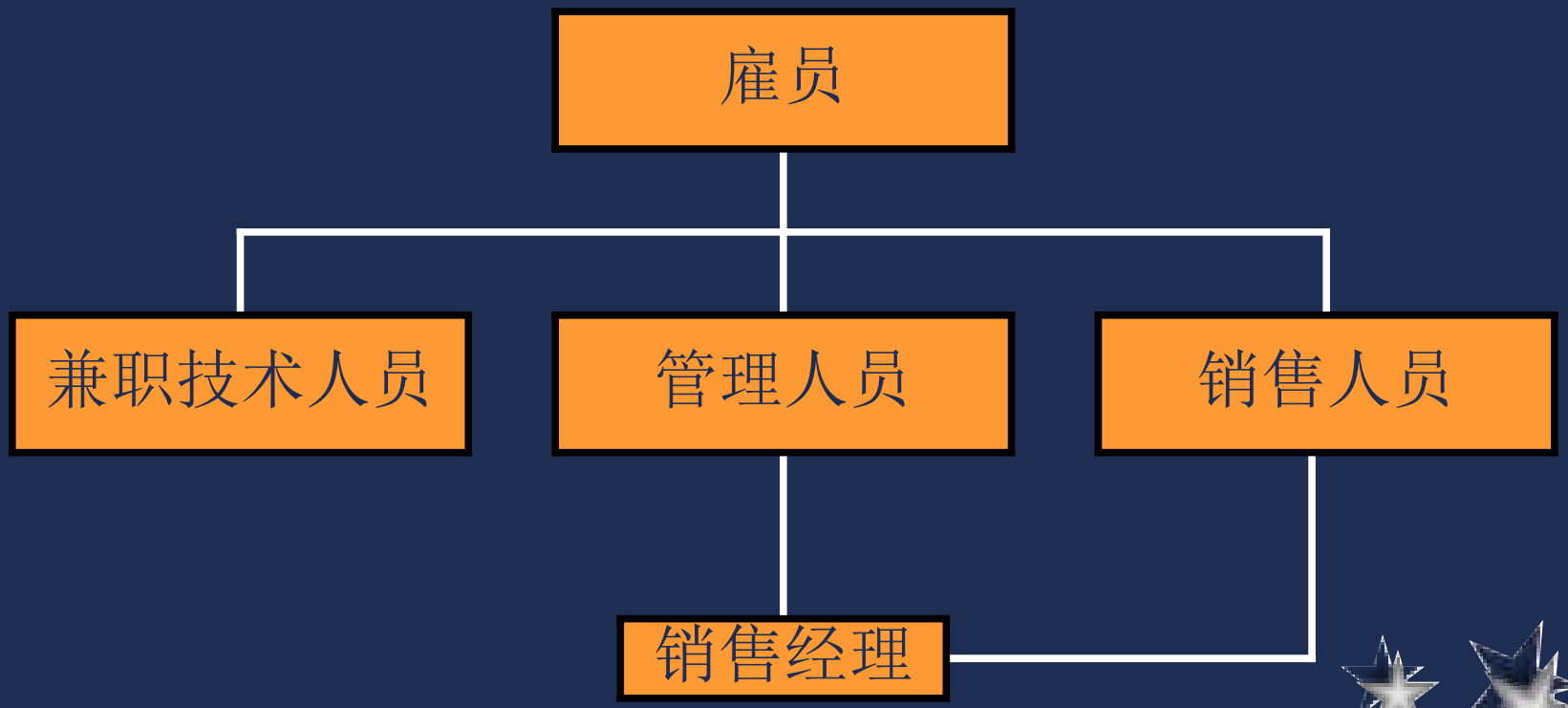
继承与派生问题举例

类的继承与派生



继承与派生问题举例

类的继承与派生



类的层次结构

- 由上到下，是一个具体化、特殊化的过程
- 由下到上，是一个抽象化的过程

继承与派生的目的

- 继承的目的：实现代码重用。
- 派生的目的：当新的问题出现，原有程序无法解决（或不能完全解决）时，需要对原有程序进行改造，实现代码扩充。

派生新类的过程

类的继承与派生

- 吸收基类成员
  - 基类的全部数据成员和除了构造、析构函数外的全部函数成员
- 改造基类成员
  - 基类成员的访问控制
  - 对基类数据或函数成员的隐藏
- 添加新的成员
  - 一般成员的添加
  - 构造函数和析构函数

派生类的声明


```
class 派生类名: 继承方式  基类名
{
        派生类成员声明;
};
```

继承方式

类成员的访问控制

- 不同继承方式的影响主要体现在：
  - 派生类**成员**对基类成员的访问权限
  - 通过派生类**对象**对基类成员的访问权限
- 三种继承方式
  - 公有继承
  - 私有继承，默认
  - 保护继承

公有继承(public)

类成员的访问控制

- 基类的public和protected成员的访问属性在派生类中保持不变，但基类的private成员不可直接访问。
- 派生类中的成员函数可以直接访问基类中的public和protected成员，但不能直接访问基类的private成员。
- 通过派生类的对象只能访问基类的public成员。

例7-1 公有继承举例

类成员的访问控制

```
class Point {   //基类Point类的定义
public:    //公有函数成员
   void initPoint(float x = 0, float y
   = 0) { this->x = x; this->y = y;}
   void move(float offX, float offY)
   { x += offX; y += offY; }
   float getX() const { return x; }
   float getY() const { return y; }
private:   //私有数据成员
   float x, y;
};
```

```
class Rectangle: public Point {     //派生类定义部分
public:     //新增公有函数成员
   void initRectangle(float x, float y, float w, float h) {
      initPoint(x, y); //调用基类公有成员函数
      this->w = w;
      this->h = h;
   }
   float getH() const { return h; }
   float getW() const { return w; }
private:    //新增私有数据成员
   float w, h;
};
```

```
#include <iostream>
#include <cmath>
using namespace std;
int main() {
  Rectangle rect;     //定义Rectangle类的对象
  //设置矩形的数据
  rect.initRectangle(2, 3, 20, 10);
  rect.move(3,2);     //移动矩形位置
  cout << "The data of rect(x,y,w,h): " <<
  endl;
  //输出矩形的特征参数
  cout << rect.getX() <<", "
       << rect.getY() << ", "
       << rect.getW() << ", "
       << rect.getH() << endl;
  return 0;
}
```

私有继承(private)

类成员的访问控制

- 基类的public和protected成员都以private身份出现在派生类中，但基类的private成员不可直接访问。
- 派生类中的成员函数可以直接访问基类中的public和protected成员，但不能直接访问基类的private成员。
- 通过派生类的对象不能直接访问基类中的任何成员。

例7-2 私有继承举例

类成员的访问控制

```
class Rectangle: private Point {    //派生类定义部分
public://新增公有函数成员
   void initRectangle(float x, float y, float w,
   float h) {
      initPoint(x, y); //调用基类公有成员函数
      this->w = w;
      this->h = h;
   }
   void move(float offX, float offY) {
      Point::move(offX, offY);
   }
   float getX() const { return Point::getX(); }
   float getY() const { return Point::getY(); }
   float getH() const { return h; }
   float getW() const { return w; }
private:       //新增私有数据成员
   float w, h;
};
```

```
#include <iostream>
#include <cmath>
using namespace std;

int main() {
   Rectangle rect; //定义Rectangle类的对象
   rect.initRectangle(2, 3, 20, 10); //设置矩形的数据
   rect.move(3,2); //移动矩形位置
   cout << "The data of rect(x,y,w,h): " << endl;
   cout << rect.getX() <<", " //输出矩形的特征参数
        << rect.getY() << ", "
        << rect.getW() << ", "
        << rect.getH() << endl;
   return 0;
}
```

保护继承(protected)

类成员的访问控制

- 基类的public和protected成员都以protected身份出现在派生类中，但基类的private成员不可直接访问。
- 派生类中的成员函数可以直接访问基类中的public和protected成员，但不能直接访问基类的private成员。
- 通过派生类的对象不能直接访问基类中的任何成员

protected 成员的特点与作用

类成员的访问控制

- 对建立其所在类对象的模块来说，它与 private 成员的性质相同。
- 对于其派生类来说，它与 public 成员的性质相同。
- 既实现了数据隐藏，又方便继承，实现代码重用。

例 protected 成员举例

类成员的访问控制

```
class A {
protected:
   int x;
};
int main() {
   A a;
   a.x = 5;      //错误
}
```

```
class A {
protected:
  int x;
};
class B: public A{
public:
  void function();
};
void B:function() {
  x = 5;   //正确
}
```

类型兼容规则

类型兼容

- 一个公有派生类的对象在使用上可以被当作基类的对象，反之则禁止。具体表现在：
  - 派生类的对象可以隐含转换为基类对象。
  - 派生类的对象可以初始化基类的引用。
  - 派生类的指针可以隐含转换为基类的指针。
- 通过基类对象名、指针只能使用从基类继承的成员

例7-3 类型兼容规则举例

类型兼容

```
//7_3.cpp
#include <iostream>
using namespace std;

class Base1 { //基类Base1定义
public:
   void display() const {
      cout << "Base1::display()" << endl;
   }
};
```

```
class Base2: public Base1 { //公有派生类Base2定义
public:
   void display() const {
     cout << "Base2::display()" << endl;
   }
};

class Derived: public Base2 { //公有派生类Derived定义
public:
   void display() const {
     cout << "Derived::display()" << endl;
   }
};

void fun(Base1 *ptr) { //参数为指向基类对象的指针
   ptr->display(); //"对象指针->成员名"
}
```

```
int main() {      //主函数
  Base1 base1;   //声明Base1类对象
  Base2 base2;   //声明Base2类对象
  Derived derived;    //声明Derived类对象

   //用Base1对象的指针调用fun函数
  fun(&base1);
  //用Base2对象的指针调用fun函数
  fun(&base2);
  //用Derived对象的指针调用fun函数
  fun(&derived);

  return 0;
}
```

运行结果：

Base1::display()

Base1 ::display()

Base1 ::display()

基类与派生类的对应关系

单继承与多继承

- 单继承
  - 派生类只从一个基类派生。
- 多继承
  - 派生类从多个基类派生。
- 多重派生
  - 由一个基类派生出多个不同的派生类。
- 多层派生
  - 派生类又作为基类，继续派生新的类。

多继承时派生类的声明


```
class 派生类名: 继承方式1  基类名1,
   继承方式2  基类名2, …
{
        派生类成员声明;
};
```

注意：每一个“继承方式”，只用于限制对紧随其后之基类的继承。

多继承举例

单继承与多继承

```
class A {
public:
   void setA(int);
   void showA() const;
private:
   int a;
};
class B {
public:
   void setB(int);
   void showB() const;
private:
   int b;
};
class C : public A,
   private B {
public:
   void setC(int, int,
   int);
   void showC() const;
private const:
   int c;
};
```

```
void  A::setA(int x) {
   a=x;
}
void B::setB(int x) {
   b=x;
}
void C::setC(int x, int y,
   int z) {
   //派生类成员直接访问基类的
   //公有成员
   setA(x);
   setB(y);
   c = z;
}
//其他函数实现略
```

```
int main() {
   C obj;
   obj.setA(5);
   obj.showA();
   obj.setC(6,7,9);
   obj.showC();
// obj.setB(6);  错误
// obj.showB(); 错误
   return 0;
}
```

继承时的构造函数

- 基类的构造函数不被继承，派生类中需要声明自己的构造函数。
- 定义构造函数时，只需要对本类中新增成员进行初始化，对继承来的基类成员的初始化，自动调用基类构造函数完成。
- 派生类的构造函数需要给基类的构造函数传递参数

单一继承时的构造函数

派生类名::派生类名(参数表) : 基类名(基类初始化参数表), 成员对象名1(成员对象1初始化参数表),...,成员对象名m(成员对象m初始化参数表)

{

其他初始化操作;

};

单一继承时的构造函数举例

```
#include<iostream>
using namecpace std;
class B {
public:
  B();
  B(int i);
  ~B();
  void print() const;
private:
  int b;
};
```

```
B::B() {
  b=0;
  cout << "B's default constructor called." <<
  endl;
}
B::B(int i) {
  b=i;
  cout << "B's constructor called." << endl;
}
B::~B() {
  cout << "B's destructor called." << endl;
}
void B::print() const {
  cout << b << endl;
}
```

```
class C: public B {
public:
  C();
  C(int i, int j);
  ~C();
  void print() const;
private:
  int c;
};
```

```
C::C() {
   c = 0;
   cout << "C's default constructor called." << endl;
}
C::C(int i,int j): B(i) {
   c = j;
   cout << "C's constructor called." << endl;
}
C::~C() {
   cout << "C's destructor called." << endl;
}
void C::print() const {
   B::print();
   cout << c << endl;
}
int main() {
   C obj(5, 6);
   obj.print();
   return 0;
}
```

多继承时的构造函数

派生类名::派生类名(参数表)：基类名1(基类1初始化参数表), 基类名2(基类2初始化参数表), ...基类名n(基类n初始化参数表),成员对象名1(成员对象1初始化参数表),...,成员对象名m(成员对象m初始化参数表)

{

其他初始化操作;

};

派生类与基类的构造函数

- 当基类中声明有缺省构造函数或未声明构造函数时，派生类构造函数可以不向基类构造函数传递参数，也可以不声明。构造派生类的对象时，基类的缺省构造函数将被调用。
- 当需要执行基类中带形参的构造函数来初始化基类数据时，派生类构造函数应在初始化列表中为基类构造函数提供参数。

构造函数的执行顺序

1. 调用基类构造函数，调用顺序按照它们被继承时声明的顺序（从左向右）。
2. 对派生类新增的成员对象进行初始化，初始化顺序按照它们在类中声明的顺序。
3. 执行派生类的构造函数体中的内容。

拷贝构造函数

- 若建立派生类对象时没有编写拷贝构造函数，编译器会生成一个隐含的拷贝构造函数，该函数先调用基类的拷贝构造函数，再为派生类新增的成员对象执行拷贝。
- 若编写派生类的拷贝构造函数，一般需要为基类相应的拷贝构造函数传递参数。例如:

```
C::C(C &c1): B(c1) {…}
```

例7-4 派生类构造函数举例

```
//7_4.cpp
#include <iostream>
using namespace std;
class Base1 {          //基类Base1，构造函数有参数
public:
   Base1(int i) { cout << "Constructing Base1 " <<
   i << endl; }
};
class Base2 {          //基类Base2，构造函数有参数
public:
   Base2(int j) { cout << "Constructing Base2 " <<
   j << endl; }
};
class Base3 {          //基类Base3，构造函数无参数
public:
   Base3() { cout << "Constructing Base3 *" <<
   endl; }
};
```

```
class Derived: public Base2, public Base1, public Base3
   {
//派生新类Derived，注意基类名的顺序
public:       //派生类的公有成员
   Derived(int a, int b, int c, int d): Base1(a),
   member2(d), member1(c), Base2(b)
   { }
   //注意基类名的个数与顺序，//注意成员对象名的个数与顺
   序
private:      //派生类的私有成员对象
   Base1 member1;
   Base2 member2;
   Base3 member3;
};

int main() {
   Derived obj(1, 2, 3, 4);
   return 0;
}
```

运行结果：

```
constructing Base2 2
constructing Base1 1
constructing Base3 *
constructing Base1 3
constructing Base2 4
constructing Base3 *
```

继承时的析构函数

- 析构函数也不被继承，派生类自行声明
- 声明方法与一般（无继承关系时）类的析构函数相同。
- 不需要显式地调用基类的析构函数，系统会自动隐式调用。
- 析构函数的调用次序与构造函数相反。

例7-5 派生类析构函数举例

```
//7_5.cpp
#include <iostream>
using namespace std;

class Base1 {  //基类Base1，构造函数有参数
public:
   Base1(int i) { cout << "Constructing Base1 " << i << endl; }
   ~Base1() { cout << "Destructing Base1" << endl; }
};

class Base2 {  //基类Base2，构造函数有参数
public:
   Base2(int j) { cout << "Constructing Base2 " << j << endl; }
   ~Base2() { cout << "Destructing Base2" << endl; }
};

class Base3 {  //基类Base3，构造函数无参数
public:
   Base3() { cout << "Constructing Base3 *" << endl; }
   ~Base3() { cout << "Destructing Base3" << endl; }
};
```

```
class Derived: public Base2, public Base1, public
   Base3 {
//派生新类Derived，注意基类名的顺序
public:      //派生类的公有成员
   Derived(int a, int b, int c, int d): Base1(a),
   member2(d), member1(c), Base2(b) { }
   //注意基类名的个数与顺序，注意成员对象名的个数与顺序
private:     //派生类的私有成员对象
   Base1 member1;
   Base2 member2;
   Base3 member3;
};

int main() {
   Derived obj(1, 2, 3, 4);
   return 0;
}
```

例7-5 运行结果

```
Constructing Base2 2
Constructing Base1 1
Constructing Base3 *
Constructing Base1 3
Constructing Base2 4
Constructing Base3 *
Destructing Base3
Destructing Base2
Destructing Base1
Destructing Base3
Destructing Base1
Destructing Base2
```

派生类成员的访问

- 派生类成员的访问属性
  - 不可访问的成员
  - 私有成员
  - 保护成员
  - 公有成员
- 两个问题
  - 唯一标识问题
  - 可见性问题

同名隐藏规则

当派生类与基类中有相同成员时：

- 若未强行指明，则通过派生类对象使用的是派生类中的同名成员。
- 如要通过派生类对象访问基类中被隐藏的同名成员，应使用作用域分辨符和基类名来限定。

例7-6 多继承同名隐藏举例(1)

```
//7_6.cpp
#include <iostream>
using namespace std;
class Base1 {   //定义基类Base1
public:
   int var;
   void fun() { cout << "Member of Base1" << endl; }
};
class Base2 {   //定义基类Base2
public:
   int var;
   void fun() { cout << "Member of Base2" << endl; }
};
class Derived: public Base1, public Base2 { //定义派生类Derived
public:
   int var;     //同名数据成员
   void fun() { cout << "Member of Derived" << endl; } //同名
   函数成员
};
```

```
int main() {
   Derived d;
   Derived *p = &d;

   d.var = 1;       //对象名.成员名标识
   d.fun();  //访问Derived类成员

   d.Base1::var = 2;      //作用域分辨符标识
   d.Base1::fun();//访问Base1基类成员

   p->Base2::var = 3;     //作用域分辨符标识
   p->Base2::fun();       //访问Base2基类成员

   return 0;
}
```

二义性问题

- 在多继承时，基类与派生类之间，或基类之间出现同名成员时，将出现访问时的二义性（不确定性）——采用虚函数（参见第8章）或同名隐藏规则来解决。
- 当派生类从多个基类派生，而这些基类又从同一个基类派生，则在访问此共同基类中的成员时，将产生二义性——采用虚基类来解决。

二义性问题举例（一）

```
class A {
public:
   void  f();
};
class B {
public:
   void f();
   void g()
};
```

```
class C: public A, piblic B {
public:
    void g();
    void h();
};
```

如果定义：C c1;

则 c1.f() 具有二义性

而 c1.g() 无二义性（同名隐藏）

二义性的解决方法

- 解决方法一：用基类名来限定
  c1.A::f()　或　c1.B::f()
- 解决方法二：同名隐藏
  在C 中声明一个同名成员函数f()，f()
  再根据需要调用 A::f()　或　B::f()

二义性问题举例（二）

```
class B {
public:
   int b;
}

class B1: public B {
private:
   int b1;
};

class B2: public B {
private:
   int b2;
};
```

```
class C : public B1,public
   B2 {
public:
   int f();
private:
   int d;
}
```

有二义性：

C c;

c.b

c.B::b

无二义性：

c.B1::b

c.B2::b

虚基类

- 虚基类的引入
  - 用于有共同基类的场合
- 声明
  - 以virtual修饰说明基类
    例：class B1:virtual public B
- 作用
  - 主要用来解决多继承时可能发生的对同一基类继承多次而产生的二义性问题.
  - 为最远派生类提供唯一的基类成员，而不重复产生多次拷贝
- 注意：
  - 在第一级继承时就要将共同基类设计为虚基类。

虚基类举例

虚基类

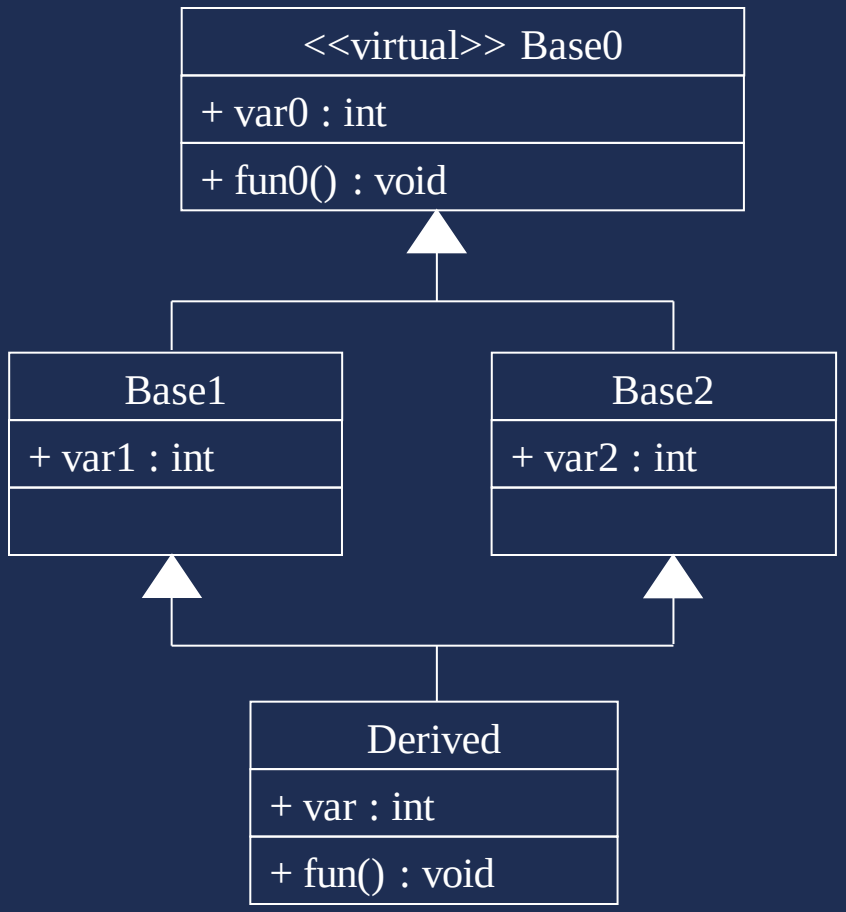
```
class B { public: int b; };
class B1: virtual public B { public: int b1; };
class B2: virtual public B { public: int b2; };
class C: public B1, public B2 { public: float
  d; };
```

下面的访问是正确的：

```
C cobj;
cobj.b;
```

例7-8虚基类举例

虚基类



Derived
Base0::var0:int Base1::var1:int Base2::var2:int var :int
Base0::fun0():void fun():void

```cpp
//7_8.cpp
#include <iostream>
using namespace std;

class Base0 {     //定义基类Base0
public:
   int var0;
   void fun0() { cout << "Member of Base0" << endl; }
};

class Base1: virtual public Base0 { //定义派生类Base1
public:     //新增外部接口
   int var1;
};

class Base2: virtual public Base0 { //定义派生类Base2
public:     //新增外部接口
   int var2;
};
```

```
class Derived: public Base1, public Base2 {
    //定义派生类Derived
public:   //新增外部接口
  int var;
  void fun() {
    cout << "Member of Derived" << endl;
  }
};

int main() {    //程序主函数
  Derived d;    //定义Derived类对象d
  d.var0 = 2;   //直接访问虚基类的数据成员
  d.fun0();     //直接访问虚基类的函数成员
  return 0;
}
```

虚基类及其派生类构造函数

虚基类

- 建立对象时所指定的类称为最远派生类。
- 虚基类的成员是由最远派生类的构造函数通过调用虚基类的构造函数进行初始化的。
- 在整个继承结构中，直接或间接继承虚基类的所有派生类，都必须在构造函数的成员初始化表中给出对虚基类的构造函数的调用。如果未列出，则表示调用该虚基类的默认构造函数。
- 在建立对象时，只有最远派生类的构造函数调用虚基类的构造函数，该派生类的其他基类对虚基类构造函数的调用被忽略。

有虚基类时的构造函数举例

虚基类

```
#include <iostream>
using namespace std;

class Base0 {  //定义基类Base0
public:
   Base0(int var) : var0(var) { }
   int var0;
   void fun0() { cout << "Member of Base0" << endl; }
};

class Base1: virtual public Base0 {   //定义派生类Base1
public://新增外部接口
   Base1(int var) : Base0(var) { }
   int var1;
};

class Base2: virtual public Base0 {   //定义派生类Base2
public://新增外部接口
   Base2(int var) : Base0(var) { }
   int var2;
};
```

```
class Derived: public Base1, public Base2 {
  //定义派生类Derived
public:    //新增外部接口
  Derived(int var) : Base0(var), Base1(var),
  Base2(var) { }
  int var;
  void fun() { cout << "Member of Derived" <<
  endl; }
};

int main() {     //程序主函数
  Derived d(1); //定义Derived类对象d
  d.var0 = 2;    //直接访问虚基类的数据成员
  d.fun0();      //直接访问虚基类的函数成员
  return 0;
}
```

组合与继承

深度探索

- 组合与继承：通过已有类来构造新类的两种基本方式
- 组合：B类中存在一个A类型的内嵌对象
  - 有一个（has-a）关系：表明每个B类型对象“有一个” A类型对象
  - A类型对象与B类型对象是部分与整体关系
  - B类型的接口不会直接作为A类型的接口

“has-a”举例

深度探索

```
class Engine {      //发动机类
public:
    void work();    //发动机运转
    ……
};
class Wheel {       //轮子类
public:
    void roll();    //轮子转动
    ……
};
class Automobile { //汽车类
public:
    void move();    //汽车移动
private:
    Engine engine; //汽车引擎
    Wheel wheels[4];//4个车轮
    ……
};
```

- 意义
  - 一辆汽车有一个发动机
  - 一辆汽车有四个轮子
- 接口
  - 作为整体的汽车不再具备发动机的运转功能，和轮子的转动功能，但通过将这些功能的整合，具有了自己的功能——移动

公有继承的意义

深度探索

- 公有继承：A类是B类的公有基类
  - 是一个（is-a）关系：表明每个B类型对象“是一个” A类型对象
  - A类型对象与B类型对象是一般与特殊关系
    - 回顾类的兼容性原则：在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
  - B类型对象包括A类型的全部接口

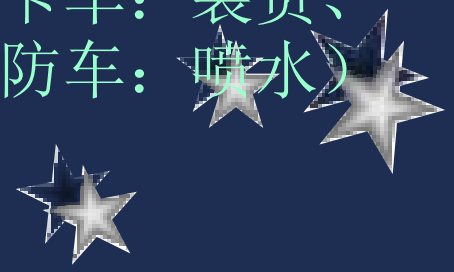
“is-a”举例

深度探索

```
class Truck: public Automobile{
//卡车
public:
    void load(…); //装货
    void dump(…); //卸货
private:
    ……
};

class Pumper: public Automobile {
//消防车
public:
    void water();  //喷水
private:
    ……
};
```

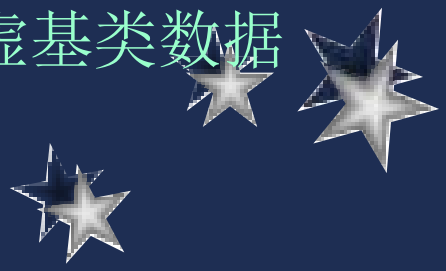
- 意义
  - 卡车是汽车
  - 消防车是汽车
- 接口
  - 卡车和消防车具有汽车的通用功能（移动）
  - 它们还各自具有自己的功能（卡车：装货、卸货；消防车：喷水）



派生类对象的内存布局

深度探索

- 派生类对象的内存布局
  - 因编译器而异
  - 内存布局应使类型兼容规则便于实现
    - 一个基类指针，无论其指向基类对象，还是派生类对象，通过它来访问一个基类中定义的数据成员，都可以用相同的步骤
- 不同情况下的内存布局
  - 单继承：基类数据在前，派生类新增数据在后
  - 多继承：各基类数据按顺序在前，派生类新增数据在后
  - 虚继承：需要增加指针，间接访问虚基类数据

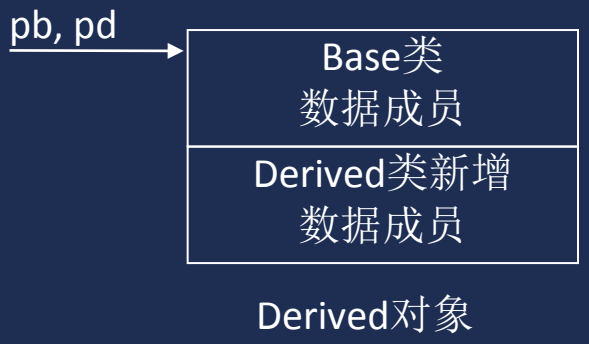


单继承情形

深度探索

```
class Base { … };
class Derived: public
   Base { … };

Derived *pd = new
   Derived();
Base *pb = pd;
```



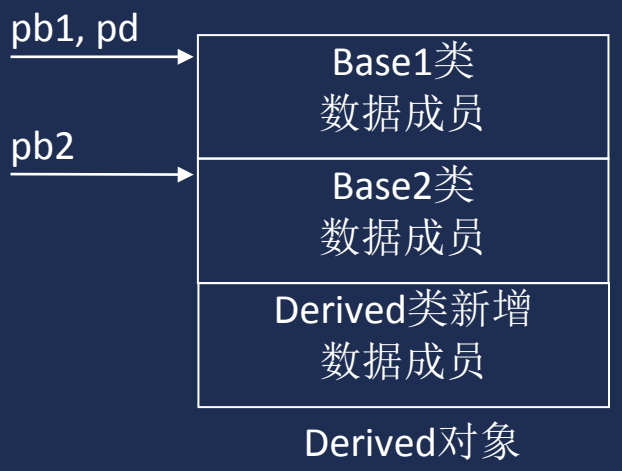
Derived类型指针pd转换为Base类型指针时，地址不需要改变

多继承情形

深度探索

```
class Base1 { … };
class Base2 { … };
class Derived: public
   Base1, public Base2
   { … };

Derived *pd = new
   Derived();
Base1 *pb1 = pd;
Base2 *pb2 = pd;
```



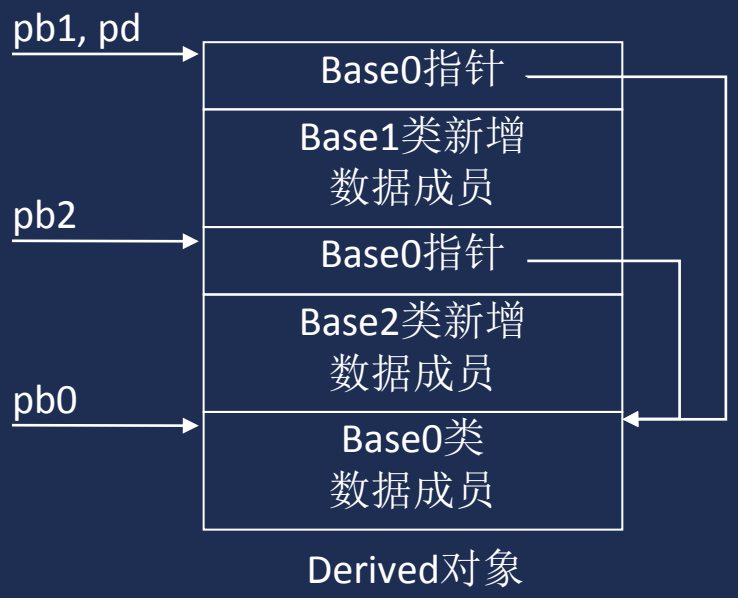
Derived类型指针pd转换为Base2类型指针时，原地址需要增加一个偏移量

虚拟继承情形

深度探索

```
class Base0 { … };
class Base1: virtual public
   Base0 { … };
class Base2: virtual public
   Base0 { … };
class Derived: public Base1,
   public Base2 { … };

Derived *pd = new Derived();
Base1 *pb1 = pd;
Base2 *pb2 = pd;
Base0 *pb0 = pb1;
```



通过指针间接访问虚基类的数据成员

基类向派生类的转换

深度探索

- 基类向派生类的转换
  - 基类指针可以转换为派生类指针
  - 基类引用可以转换为派生类引用
  - 需要用static_cast显式转换
- 例：

```
Base *pb = new Derived();
Derived *pd = static_cast<Derived *>(pb);

Derived d;
Base &rb = d;
Derived &rd = static_cast<Derived &>(rb);
```

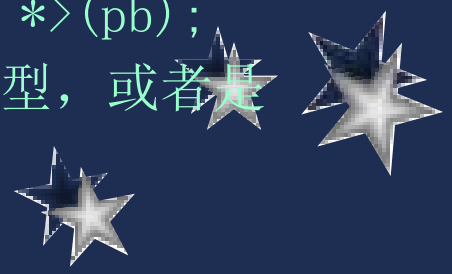
类型转换时的注意事项(1)

深度探索

- 基类对象一般无法被显式转换为派生类对象
  - 对象到对象的转换，需要调用构造函数创建新的对象
  - 若派生类有构造函数接收基类对象（或基类引用）的参数，才可行

```
Base b;
Derived d = static_cast<Derived>(b);
```

- 执行基类向派生类的转换时，一定要确保被转换的指针和引用所指向或引用的对象符合转换的目的类型：
  - 对于Derived *pd = static_cast<Derived *>(pb);
  - 一定要保证pb所指向的对象具有Derived类型，或者是Derived类型的派生类。



类型转换时的注意事项(2)

深度探索

- 如果A类型是B类型的虚基类，A类型指针无法通过static_cast显式转换为B类型的指针
  - 可以结合虚继承情况下的对象内存布局，思考为什么不允许这种转换
- void指针参加的转换，可能导致不可预期的后果：
  - 例：Derived类公共继承自Base1类和Base2类

```
Derived *pd = new Derived();
void *pv = pd;          //将Derived指针转换为void指针
Base2 *pb = static_cast<Base2 *>(pv);
```

  - 转换后pb与pd有相同的地址，而正常的转换下应有一个偏移量
  - 结论：有void指针参与的转换，兼容性规则不适用
- 更安全更灵活的基类向派生类转换方式——dynamic_cast，将在下一章介绍

小结与复习建议

- 主要内容
  - 类的继承、类成员的访问控制、单继承与多继承、派生类的构造和析构函数、类成员的标识与访问
- 达到的目标
  - 理解类的继承关系，学会使用继承关系实现代码的重用。