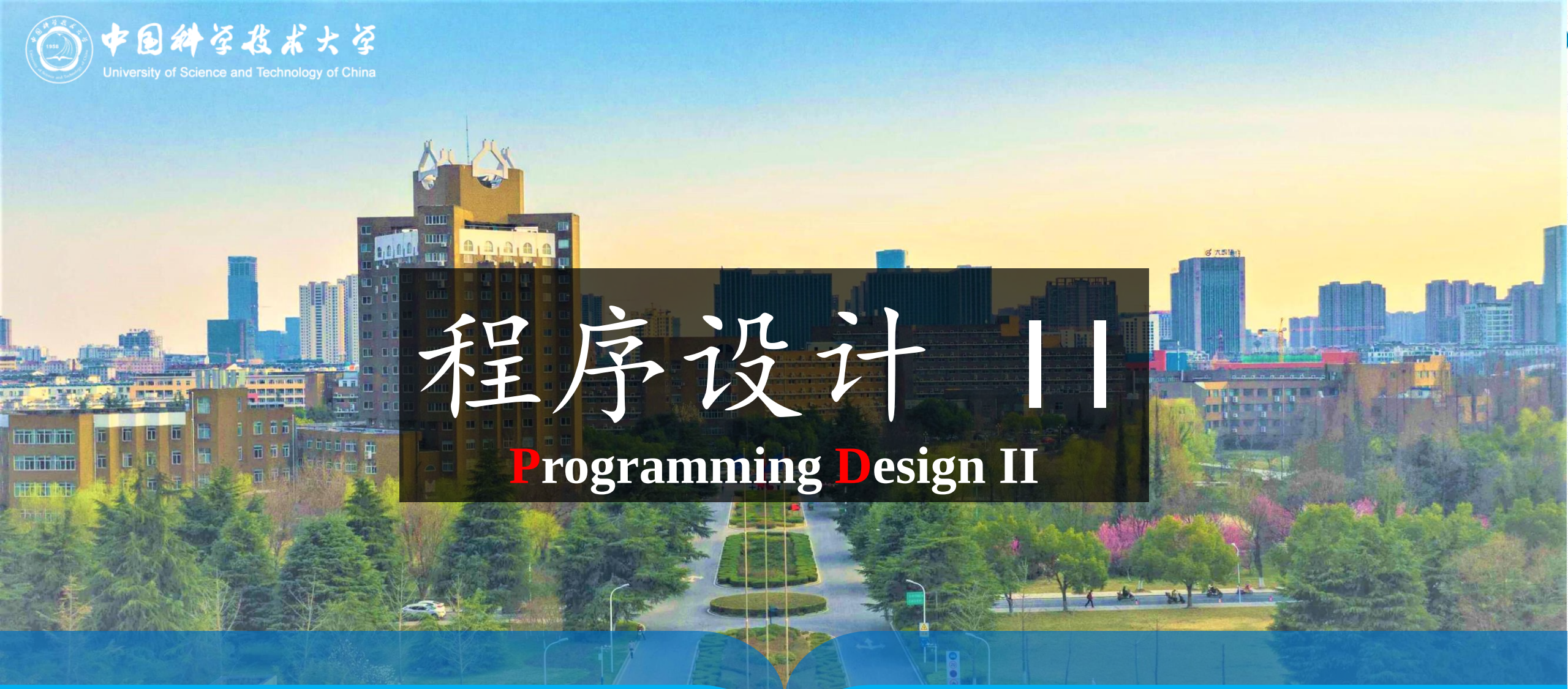

中国科学技术大学
University of Science and Technology of China
程序设计 II
Programming Design II

C++类的继承

主讲：吴锋

目录
CONTENTS

类的继承

- 类的继承是一种代码重用机制。
- 在现有的类（称为“基类”）的基础上，创建一个新类（称为“派生类”）。
- 派生类“共享”基类的所有数据成员和成员函数，也可以进一步添加自己的成员。

```
class 派生类类名：public/private/protected 基类类名 {
        // 派生类的新成员
};
```

类继承的例子

Base class	Derived classes
Student	GraduateStudent UndergraduateStudent
Shape	Circle Triangle Rectangle
Loan(贷款)	CarLoan(车贷) HomeImprovementLoan(房贷) MortgageLoan(抵押贷款)
Employee(雇员)	FacultyMember(教员) StaffMember(职员)
Account	CheckingAccount SavingsAccount

单继承

- 单继承是指派生类只继承一个基类，是最常用到的一种类继承方式（满足大部分的需求）。
- 派生类继承基类的所有数据成员和成员函数，下列函数和关系除外：
  - 基类的构造函数和析构函数；
  - 基类中用户定义的new和赋值运算符；
  - 基类的友元关系。

```
class Employee { … };
class Manager: public Employee { … };
```

多重继承

- 多重继承是指一个派生类同时继承多个基类，其作用和意义尚有争议（一般情况不需要）。
- 当多个基类中含同名成员时，存在二义性的问题！

```
class Manager { … };
class Temporary { … };

// 类 Consultant(顾问)
class Consultant: public Manager, public Temporary { … };
```

◎ 继承的访问控制

- 继承的访问修饰符: public / private / protected（缺省默认为private）。访问修饰符将影响到：
  - 派生类的内部对基类不同区成员的访问 (f2->Base);
  - 派生类的对象对基类不同区域成员的访问 (d1->Base);
  - 派生类的派生类对基类不同区域成员的访问 (f3->Base);

```
class Base{
     private:   int i1;
     protected: int j1;
     public:    int f1();
};
class Drv: public Base{
     private:   int i2;
     protected: int j2;
     public:    int f2();
};
```

```
class Drv2: public Drv {
     private: int i3;
     protected: int j3;
     public: int f3();
};

void main() {
    Drv  d1;
    Drv2 d2;
}
```

public 公有继承

- 基类成员对派生类的可见性：
  - 公有成员和保护成员可见，私有成员不可见。
- 基类成员对派生类对象的可见性：
  - 公有成员可见，保护成员和私有成员不可见。

```
class Base{ private:   int i1; protected: int j1; public: int f1(); };
class Drv: public Base{ public: int f2(); }; // i1 不可见，j1, f1() 可见
class Drv2: public Drv{ public: int f3(); }; // i1 不可见，j1, f1() 可见

void main() {
   Drv d1; // d1.i1, d1.j1 不可见，d1.f1() 可见
   Drv d2; // d2.i1, d2.j1 不可见，d2.f1() 可见
}
```

private 私有继承

- 基类成员对派生类的可见性：
  - 公有成员和保护成员可见，私有成员不可见。
  - 所有成员对派生类的派生类的成员不可见（即转变为派生类的私有成员）。
- 基类成员对派生类对象的可见性：
  - 所有成员都不可见。

```
class Base{ private:   int i1; protected: int j1; public: int f1(); };
class Drv: private Base{ public: int f2(); }; // i1 不可见，j1, f1() 可见
class Drv2: public Drv{ public: int f3(); };  // i1, j1, f1() 均不可见
void main() {
   Drv1 d1; // d1.i1, d1.j1, d1.f1() 均不可见
   Drv2 d2; // d2.i1, d2.j1, d2.f1() 均不可见
}
```

protected 保护继承

- 基类成员对派生类的可见性：
  - 公有成员和保护成员可见，私有成员不可见。
  - 公有成员和保护成员对派生类的派生类的成员可见（即转变为派生类的保护成员）。
- 基类成员对派生类对象的可见性
  - 所有成员都不可见

```
class Base{ private:   int i1; protected: int j1; public: int f1(); };
class Drv: protected Base{ public: int f2(); }; // i1 不可见，j1, f1() 可见
class Drv2: public Drv{ public: int f3(); }; // i1 不可见，j1, f1() 可见
void main() {
    Drv1 d1; // d1.i1, d1.j1, d1.f1() 均不可见
    Drv2 d2; // d2.i1, d2.j1, d2.f1() 均不可见
}
```

◎ 小结: 访问控制关系

基类性质	继承性质	派生类对基类可访问性	基类成员在派生类中性质	派生类对象对基类的可访问性	派生类的派生类对基类的可访问性
public	public	Yes	public	Yes	Yes
protected	public	Yes	protected	不可访问	Yes
private	public	不可访问	--	不可访问	不可访问
public	protected	Yes	protected	不可访问	Yes
protected	protected	Yes	protected	不可访问	Yes
private	protected	不可访问	--	不可访问	不可访问
public	private	Yes	private	不可访问	不可访问
protected	private	Yes	private	不可访问	不可访问
private	private	不可访问	--	不可访问	不可访问

private 继承公有化

```
class Pet {
public:
      char eat() const { return 'a'; }
      int speak() const { return 2; }
      float sleep() const { return 3.0; }
      float sleep(int) const { return 4.0; }
};
class Goldfish : Pet {     // （默认）private 私有继承
public:
       Pet::eat;           // 公有化
       Pet::sleep;         // 重载的成员均被公有化
};

void main() {
     Goldfish bob;
     bob.eat(); bob.sleep(); bob.sleep(1);
     //! bob.speak();      // 错误，私有成员不能访问
}
```

◎ 构造、析构的调用次序

- 创建一个派生类的对象，调用次序为：
  1. 调用基类的构造函数,初始化基类成员；
  2. 调用数据成员对象的构造函数 (若有多个，则按类中定义的先后次序调用)，初始化成员对象；
  3. 调用派生类自己的构造函数；初始化派生类成员。
- 析构一个派生类的对象，调用次序正好相反：
  1. 调用派生类自己的析构函数；
  2. 调用数据成员对象的析构函数 (若有多个，则按类中定义的先后次序逆序调用)；
  3. 调用基类的析构函数。

初始化基类成员

- 派生类在初始化时会隐式的调用基类的默认构造函数（必须存在），也可以在派生类的成员初始化表中显式的调用基类的构造函数。

```
class Employee {
    char *name; int age; float salary;
public:
    Employee(char *n, int a, float s): name(n), age(a), salary(s) {}
};

class Manager: public Employee {
    int level;
public:
    Manager(char *n, int a, float s, int l):Employee(n,a,s), level(l){}
};

Manager  m(“Maggie”, 20, 6000, 1);
```

继承与静态成员函数

- 静态成员函数可被继承到派生类中；
- 如果重新定义了一个静态成员，所有在基类中的其他重载函数会被隐藏；
- 与非静态成员函数的不同，静态成员函数不可以是虚函数。

成员函数重定义

- 在派生类中，可以重新定义基类的成员函数。重定义后，基类的同名函数在派生类被隐藏。
- 如果要在派生类中使用基类的成员函数，可以用基类的类名作为域名进行调用。
- 在重新定义基类的重载函数时（可以改变参数或返回值），则基类中的所有其他版本的重载函数被自动隐藏了，对派生类的对象不可见。

成员函数重定义

```
class Base { // 基类
public:
      int f() const {
            cout << "Base::f()\n";  return 0;
      }
      int f(string) const {  return 1;  }
};

class Derived1 : public Base { // 派生类1
public:
      // 成员函数重定义:
      int f() const {
            // int f(string)对Derived1不可见（被隐藏）
            // 调用的话，需要添加基类的域名
            Base::f(); Base::f("hello");
            cout << "Derived1::f()\n";  return 1;
      }
};
```

成员函数重定义

```
class Derived2 : public Base { // 派生类2
public:
          // 改变成员函数返回类型:
          void f() const {
              // int f()和int f(string)对Derived2都不可见
     cout << "Derived2::f()\n";
           }
};

class Derived3 : public Base {
public:
      // 改变成员函数参数列表:
      int f(int) const {
                // int f()和int f(string)对Derived3都不可见
                cout << "Derived3::f()\n";  return 3;
      }
};
```

成员函数重定义

```
void main() {
        string s("hello");
        Derived1 d1;
        x = d1.f();
        //!  d1.f(s);  // 错误，基类中参数为string的成员函数被隐藏

        Base &b = d1;
        b.f(s); // 正确，需要将类型转换为基类调用基类的成员函数

        Derived2 d2;
        //!  x = d2.f(); // 错误，基类中返回值为int的成员函数被隐藏
        //!  d2.f(s); // 错误，基类中参数为string的成员函数被隐藏

        Derived3 d3;
        //!  x = d3.f(); // 错误，基类中无参数的成员函数被隐藏
        x = d3.f(1);
}
```

◎ 派生类和基类的关系

- 继承最重要的思想实际上是说：派生类同时也是一个基类。也就是说：你可以发送给基类的任何消息，同样可以发送给派生类！
- 编译器通过允许被称作“向上类型转换(Upcast-ing)”的机制来继承的这一特征。

```
enum note { middleC, Csharp, Cflat };
class Instrument {
public:
  void play(note) const {}
};
class Wind : public Instrument { };
// 对Instrument和从Instrument派生出来的任何类都是有效的。
void tune(Instrument& i) { i.play(middleC); }
// 在函数调用时，实际上将Wind的引用转换成Instrument的引用。
void main() { Wind flute; tune(flute);  }
```

◎ 向上类型转换

- 向上类型转换不仅仅发生在虚实结合时，例如：

```
Wind   w;
Instrument  * ip = &w ;   // Upcasting
Instrument  & ir = w ;    // Upcasting
```

- 向上类型转换总是安全的：
  - 从派生类接口转换到基类接口可能出现的唯一的事情是失去成员，而不是获得他们。
- 编译器允许向上类型转换（如函数调用时）；既不需要显式的转换，也不需要其他特别的标记。

非显式类型转换

- 派生类的对象可以当成基类对象来处理
  - 派生类中包含基类中所有的成员，派生类可以赋值给基类
  - 派生类中可能包含基类中没有的成员
- 基类对象不能当成派生类的对象来处理
  - 会造成多出来的那些派生类成员未定义
  - 基类不能赋值给派生类，但可以通过重载赋值运算符来允许赋值
- 通过基类指针引用派生类对象
  - 可能会出现语法错误
  - 代码中只能引用基类的成员，否则会有语法错误
- 通过派生类指针引用基类对象
  - 隐式转换会造成语法错误（编译器不允许）
  - 否则，派生类指针必须先转换成基类指针（显式转换）

向上类型转换的问题

- 向上类型转换会丢失对象的类型信息。

```
Wind  w;
Instrument *iptr = &w;   // Upcasting
iptr->play(middleC);
```

  - 由于iptr的静态类型Instrument*；在编译时该函数调用将被绑定到 Instrument::play()；而不是Wind::play()！

向下类型转换

- 当基类指针(或引用)转换到派生类指针(或引用)，称作“向下类型转换”(Downcasting)。
- 向下类型转换不安全，因此，必须显式地进行(显式类型转换)，并判断转换是否成功。

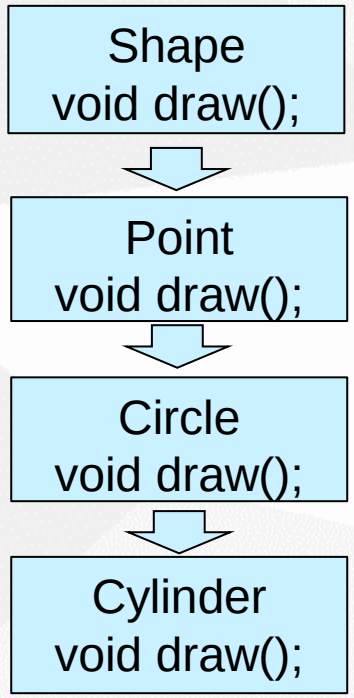
```
Instrument ins;
Wind *iptr = dynamic_cast<Wind*> (&in);
if (iptr) iptr->play(middleC);
```

  - 向下类型转换不成功时，iptr == 0

多态性

- 多态性：允许把不同的派生类用一个统一的基类接口（抽象接口）来处理。即一段代码可以同样地工作在所有这些具有同一个基类的不同派生类上。
- 也就是说,让一个对象做什么及如何做 (How)之前不再关心这个对象是什么 (What)。

```
Circle  c1(10,3,2);
Point p1(2,4);
Cylinder cy(…);

Shape *shp[ ] = {&c1, &p1, &cy};
for (int i = 0; i < 3; i++) {
    // 运行时决定调用哪一个draw()
    shp[i]->draw() ;
}
```



◎ 虚函数

- 虚函数：C++实现多态性机制的方式，是基类和派生类之间的一组被关键字 virtual 修饰的同名函数。编译器将对这些函数实行晚绑定。
- 晚绑定：发生在运行阶段的绑定称为“晚绑定”，又称为“动态绑定”。

```
class Shape {
    virtual void draw() = 0; // 纯虚函数
};
class Point: public Shape {
    virtual void  draw() { …… }
};
class Circle: public Point {
    virtual void  draw() { …… }
};
```

虚函数说明

- 含有纯虚函数的基类称为抽象基类，不能被实例化（即创建对象）。

```
Shape shape1; // 错误，抽象基类不能创建对象
```

- 如果想使用虚函数进行动态绑定，必须用指向基类的指针或引用来访问它。否则，编译器在处理时，会进行静态绑定。

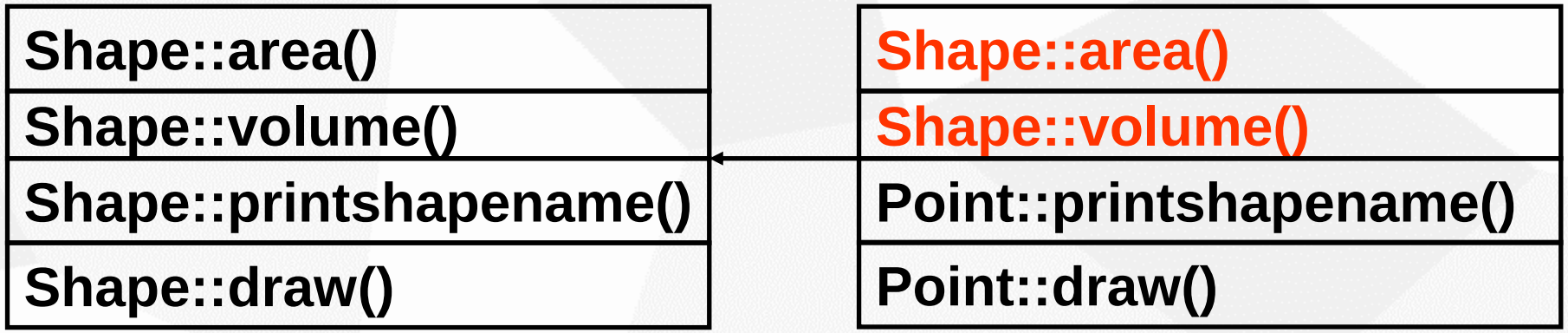
```
Circle  circle1( 3.5, 22, 8 );
Shape  *psh = & circle1;
psh -> draw();          // 动态绑定
Shape  &rs = circle1;
rs.draw() ;             // 动态绑定

Shape  sh = *psh ;
sh.draw() ;             // 静态绑定，调用 Shape::draw();
```

虚函数说明（续）

- 如果不希望编译器动态选择虚函数,而是要调用指定的某个基类的虚函数，可以用作用域分辨符显式指明，例如：

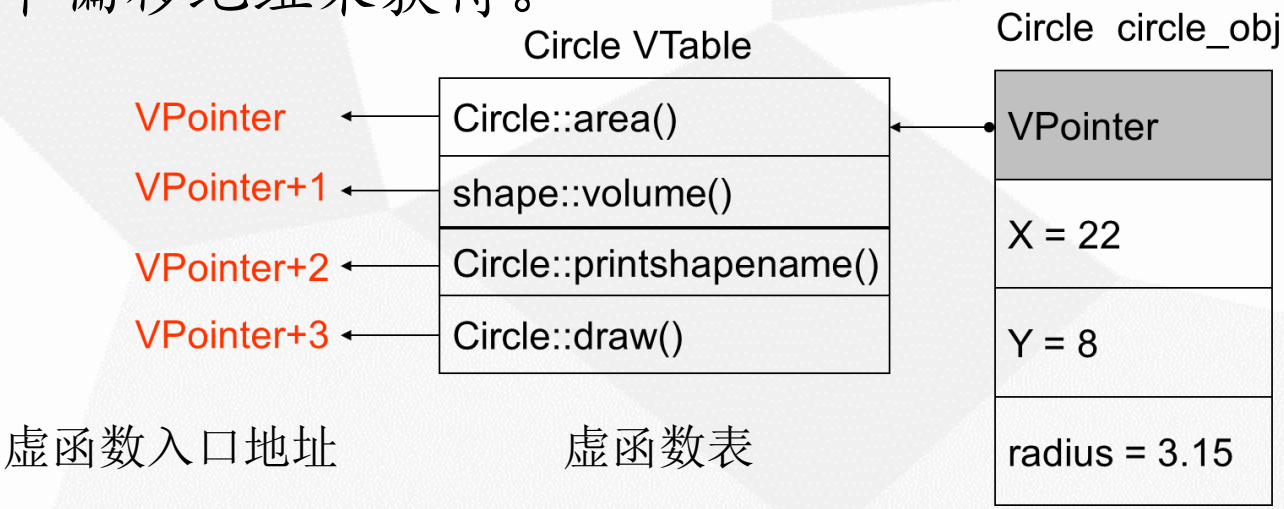
```
Circle  circle1( 3.5, 22, 8 );
Shape  *psh = & circle1;
psh->Shape::draw();                    // 静态绑定
```

- 如果派生类没有重新定义基类中的虚函数，则基类中虚函数自动成为派生类的虚函数。



虚函数表

- 编译器通过虚函数表 (VTable: Virtual Function Table) 实现虚函数的晚绑定（动态绑定）。
- 编译时，对虚函数调用不进行函数入口地址替换（静态绑定），只是为每个含有虚函数的类建立虚函数表。
- 实例化带虚函数的类的对象时，编译器在对象中附加一个虚指针 (VPointer) 指针，该指针指向该类的虚函数表。
- 不论类中有多少个虚函数，但仅给类的对象附加一个虚指针。类中的虚函数入口地址通过VPointer加上一个偏移地址来获得。



虚函数的调用流程

1. 将&circle传入baseclassptr;
2. 访问circle对象;
3. 访问Circle VTable;
4. 访问Circle::draw指针 (VPointer + 3)
5. 执行Circle::draw函数

```
Circle circle;
Shape *baseclassptr;
baseclassptr = &circle;      // 指向一个Circle对象
baseclassptr->draw() ;
```

抽象基类

- 如果只希望基类作为其派生类的一个接口，而不希望实际地创建一个基类的对象，可以把基类定义为抽象基类。
- 要把基类定义成一个抽象类，只需在基类中加入至少一个纯虚函数（Pure Virtual Function）。
- 纯虚函数告诉编译器在VTable中保留一个位置，但在这个特定位置中不放地址。抽象类的VTable是不完全的，这也是不允许实例化抽象类的原因。
- 当继承一个抽象类时，必须实现所有的纯虚函数，否则派生出的类也将是一个抽象类。

```
class Shape { // Shape 是一个抽象基类，不允许实例化
public:
      virtual double area() const { return 0.0; }
      virtual double volume() const { return 0.0; }
      virtual void draw() = 0; // 纯虚函数
};
```

虚函数与构造函数

- 创建一个包含虚函数的对象时，必须初始化它的VPointer以指向相应的VTable，这一工作由编译器秘密地插入到构造函数中的代码完成。
- 虚机制在构造函数中不起作用。也就是说，对于在构造函数中调用一个虚函数的情况，被调用的只是这个函数的本地版本。因为在生成对象的过程中，还无法知道对象的类型，无法进行动态绑定。
- 基于上述原因，构造函数不能声明为虚函数。

◎ 虚析构函数

```
class X {
private:
      char *p;
public:
      X(int sc) {
            p=new char[sz];
       }
      ~X() { delete[] p; }
};
```

```
class Y: public X {
private:
      char *pp;
public:
      Y(int sz1,int sz2):X(sz1) {
            pp=new char [sz2];
       }
       ~Y(){ delete [ ] pp; }
};
```

```
X *px = new Y(10,12);
// 由于Y类型的对象被转化成了X类型的指针，delete时调用
// 的析构函数是 ~X()，因此只删除了 p 指针所指向的空间，
// 而没有删除 pp 指针指向的空间（因为~Y()没有被执行）
delete px ;
```

◎ 虚析构函数（续）

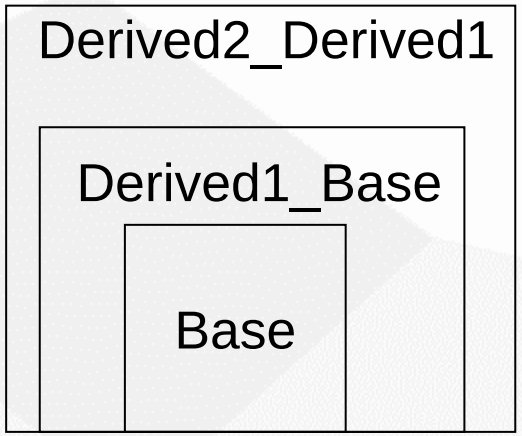
```
class X {
private:
      char *p;
public:
      X(int sc) {
            p=new char[sz];
      }
      virutual ~X()
       { delete[] p; }
};
```

```
class Y: public X {
private:
      char *pp;
public:
      Y(int sz1,int sz2):X(sz1) {
            pp=new char [sz2];
       }
      ~Y(){ delete [ ] pp; }
      // 自动成为虚析构函数
};
```

```
X *px = new Y(10,12);
// 对虚析构函数进行的是动态绑定，因此能够按照类型
// Y的方式撤销对象：即先执行 ~Y()，再执行 ~X()
// 在存在类的继承关系时，一般都将析构函数声明为虚函数
delete px ;
```

析构函数中调用虚函数

- 在一个普通的成员函数中调用一个虚函数，会使用晚绑定机制来调用这个函数。
- 但在析构函数中调用一个虚函数，则该虚函数的“本地”版本被调用：虚机制被忽略！
- 由于析构函数是从“外层”被调用起，也就是说，外层对象总是先于内层对象被撤销。因此，基类中调用的虚函数，很有可能操作在已被撤销的对象上（函数不能执行）。



析构函数中调用虚函数

```
class Base {
public:
    virtual ~Base() { f(); } // 执行本地的 Base::f()
    virtual void f() {}
};
class Derived : public Base {
public:
     ~Derived() { f(); } // 执行本地的 Derived::f()
     void f() {}
};
int main() {
  Base* bp = new Derived; delete bp;
  // 执行次序: ~Derived() -> Derived::f()
  //          -> ~Base() -> Base::f()
}
```

对虚函数的重新定义

- 派生类中重新定义一个基类中的虚函数会隐藏所有该函数的其他基类重载版本。
- 若重新定义的函数是虚函数，则编译器不允许改变它的返回值（若不是虚函数，这是可以的）。
- 这一限制保证了我们能够多态地通过基类调用函数，总是返回相同类型的值。

◎ 小结

- 继承反映的是类和类之间的层次关系，没有层次关系时不要滥用继承。通常，类之间的组合更符合模块化的思想。
- 继承的访问控制可以修改对基类成员的访问，能够进一步提高类的封装性。
- 在使用派生类时，要注意非显式的类型转换，搞清楚调用的到底是基类还是派生类的函数。
- 虚函数采用的是晚绑定，将是什么和怎么做相分离，是实现多态性的一种方式，但有计算开销。有继承关系时，虚析构函数通常是正确的选择。