广州大学

GU CN Computer Networks Institute

面向对象程序设计

Object-Oriented Programming

主讲人：王国军 教授

计算机科学与网络工程学院

csgjwang@gzhu.edu.cn

http://trust.gzhu.edu.cn/

办公室：行政西楼前座532室

根据《C++程序设计基础》（第6版）和《Visual C++面向对象与可视化程序设计》（第4版）制作，仅供广州大学计算机、软件工程、网络工程、网络空间安全、人工智能2023级讲课使用。

第9章 虚函数与多态性

9.1 静态联编

9.2 类指针的关系

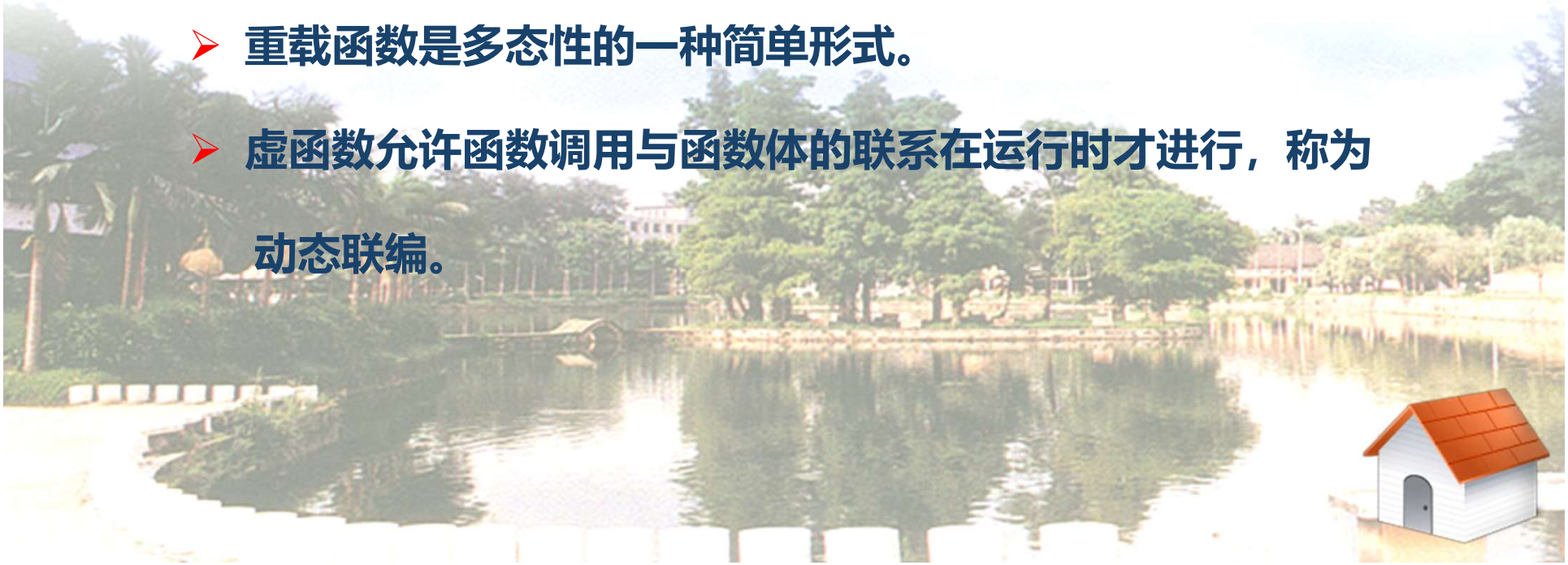
9.3 虚函数和动态联编

9.4 纯虚函数和抽象类

9.5 虚函数和多态性的应用

第9章 虚函数与多态性

- 多态性（Polymorphism）是指一个名字，多种语义；或接口相同，多种实现。
- 重载函数是多态性的一种简单形式。
- 虚函数允许函数调用与函数体的联系在运行时才进行，称为动态联编。



第9章 虚函数与多态性

9.1 静态联编

- 联编是指一个程序模块、代码之间互相关联的过程。
- 静态联编，是程序的匹配、连接在编译阶段实现，也称为早期匹配。

  重载函数使用静态联编。
- 动态联编是指程序联编推迟到运行时进行，所以又称为晚期联编。

  switch 语句、if 语句、循环语句是动态联编的例子。

9.1 静态联编

普通成员函数重载可表达为两种形式：

1. 在一个类说明中重载

例如： **void Show (int , char) ;**

void Show (char * , float) ;

9.1 静态联编

普通成员函数重载可表达为两种形式：

1. 在一个类说明中重载
2. 基类的成员函数在派生类中重载。有 3 种编译区分方法：

（1）根据参数的特征加以区分

例如： void Show (int , char); 与

void Show (char * , float); 不是同一函数，编译能够区分

9.1 静态联编

普通成员函数重载可表达为两种形式：

1. 在一个类说明中重载
2. 基类的成员函数在派生类重载。有 3 种编译区分方法：

（1）根据参数的特征加以区分

（2）使用“ :: ”加以区分

例如： A :: Show ( );

有别于 B :: Show ( );

9.1 静态联编

普通成员函数重载可表达为两种形式：

1. 在一个类说明中重载
2. 基类的成员函数在派生类重载。有 3 种编译区分方法：
   （1）根据参数的特征加以区分
   （2）使用“ :: ”加以区分
   （3）根据类对象加以区分

根据this指针类型区分

例如： Aobj . Show ( ) 调用 A :: Show ( )

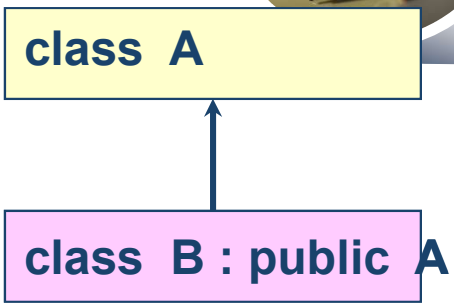
Bobj . Show ( ) 调用 B :: Show ( )

9.2 类指针的关系

基类指针和派生类指针与基类对象和派生类对象4种可能匹配：

- 直接用基类指针引用基类对象；
- 直接用派生类指针引用派生类对象；
- 用基类指针引用一个派生类对象；
- 用派生类指针引用一个基类对象。

9.2.1 基类指针引用派生类对象



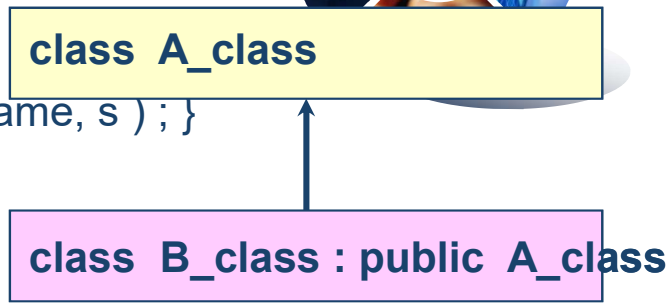
例如：

```
A   * p ;          // 指向类型 A 的对象的指针
A   A_obj ;        // 类型 A 的对象
B   B_obj ;        // 类型 B 的对象
p = & A_obj ;      // p 指向类型 A 的对象
p = & B_obj ;      // p 指向类型 B 的对象，它是 A 的派生类
```

- 利用 p，可以通过 B_obj 访问所有从 A 类继承的元素，

  但不能用 p访问 B 类自己定义的元素 （除非用了显式类型转换）

例9-1 使用基类指针引用派生类对象




```
#include<iostream>
#include<cstring>
using namespace std ;
class  A_class
{   char name[20] ;
    public :    void  put_name( const char * s ) { strcpy_s( name, s ) ; }
                void  show_name() { cout << name << "\n" ; }
};
class  B_class  : public  A_class
{     char phone_num[ 20 ] ;
    public :    void  put_phone( const char * num )  { strcpy_s ( phone_num , num ) ; }
                void  show_phone()  { cout << phone_num << "\n" ; }
};
int main()
{  A_class  * A_p ;      A_class  A_obj ;
   B_class   B_obj ;
   A_p = & A_obj ;
   A_p -> put_name( "Wang xiao hua" ) ;   A_p -> show_name() ;
   A_p = & B_obj ;
   A_p -> put_name( "Chen ming" ) ;     A_p -> show_name() ;
   B_obj.put_phone ( "5555_12345678" );
   ( ( B_class * ) A_p ) -> show_phone() ;
}
```

例9-1 使用基类指针引用派生类对象

```
#include<iostream>
#include<cstring>
using namespace std ;
class  A_class
{      char name[20] ;
   public :    void  put_name( const char * s ) { strcpy_s( name, s ) ; }
               void  show_name() { cout << name << "\n" ; }
};
class  B_class  : public  A_class
{      char phone_num[ 20 ] ;
   public :    void  put_phone(const cha
num ) ; }
               void  show_phone()  { cout << phone_num << "\n" ; }
};
int main()
{ A_class  * A_p ;      A_class  A_obj ;
   B_class   B_obj ;
   A_p = & A_obj ;
   A_p -> put_name( "Wang xiao hua" ) ;   A_p -> show_name() ;
   A_p = & B_obj ;
   A_p -> put_name( "Chen ming" ) ;     A_p -> show_name() ;
   B_obj.put_phone ( "5555_12345678" );
   ( ( B_class * ) A_p ) -> show_phone() ;
}
```

基类指针

y_s (phone_num ,

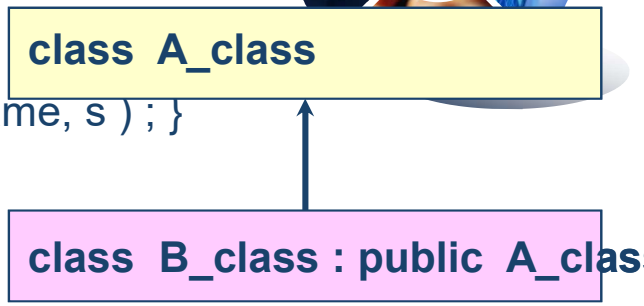
class A_class

class B_class : public A_class

例9-1 使用基类指针引用派生类对象

```
#include<iostream>
#include<cstring>
using namespace std ;
class  A_class
{      char name[20] ;
    public :    void  put_name(const char * s ) { strcpy_s( name, s ) ; }
                void  show_name() { cout << name << "\n" ; }
};
class  B_class  : public  A_class
{      char phone_num[ 20 ] ;
    public :    void  put_phone(const char * num
                void  show_phone()  { cout << ph
};
int main()
{ A_class  * A_p ;      A_class  A_obj ;
   B_class   B_obj ;
   A_p = & A_obj ;
   A_p -> put_name( "Wang xiao hua" ) ;   A_p -> show_name() ;
   A_p = & B_obj ;
   A_p -> put_name( "Chen ming" ) ;     A_p -> show_name() ;
   B_obj.put_phone ( "5555_12345678" );
   ( ( B_class * ) A_p ) -> show_phone() ;
}
```

m , num) ; }



基类指针

指向基类对象

例9-1 使用基类指针引用派生类对象

```
#include<iostream>
#include<cstring>
using namespace std ;
class  A_class
{      char name[20] ;
   public :    void  put_name(const char * s ) { strcpy_s( name, s ) ; }
               void  show_name() { cout << name << "\n" ; }
};
class  B_class  : public  A_class
{      char phone_num[ 20 ] ;
   public :    void  put_phone(const char * num )  { strcpy_s ( phone_num , num ) ; }
               void  show_phone()  { cout << phone_num << "\n" ; }
};
int main()
{ A_class  * A_p ;      A_class  A_obj ;
  B_class   B_obj ;
  A_p = & A_obj ;
  A_p -> put_name( "Wang xiao hua" ) ;   A_p -> show_name() ;
  A_p = & B_obj ;
  A_p -> put_name( "Chen ming" ) ;     A_p -> show_name() ;
  B_obj.put_phone ( "5555_12345678" );
  ( ( B_class * ) A_p ) -> show_phone() ;
}
```

class A_class

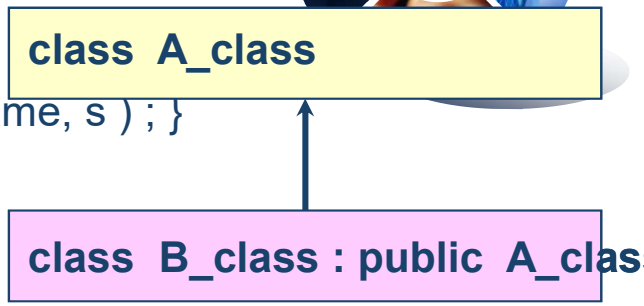
class B_class : public A_class

基类指针

调用基类成员函数

例9-1 使用基类指针引用派生类对象

```
#include<iostream>
#include<cstring>
using namespace std ;
class  A_class
{      char name[20] ;
    public :    void  put_name(const char * s ) { strcpy_s( name, s ) ; }
                void  show_name() { cout << name << "\n" ; }
};
class  B_class  : public  A_class
{      char phone_num[ 20 ] ;
    public :    void  put_phone(const char * num )  { strcpy_s ( phone_num , num ) ; }
                void  show_phone()  { cout << phone_num << "\n" ; }
};
int main()
{ A_class  * A_p ;      A_class  A_obj ;
   B_class   B_obj ;
   A_p = & A_obj ;
   A_p -> put_name( "Wang xiao hua" ) ;   A_p -> show_name() ;
   A_p = & B_obj ;
   A_p -> put_name( "Chen ming" ) ;     A_p -> show_name() ;
   B_obj.put_phone ( "5555_12345678" );
   ( ( B_class * ) A_p ) -> show_phone() ;
}
```

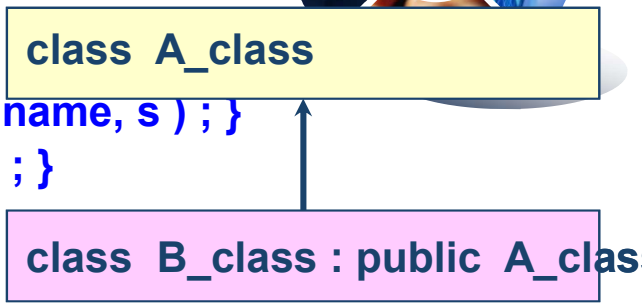


基类指针

指向派生类对象

例9-1 使用基类指针引用派生类对象

```
#include<iostream>
#include<cstring>
using namespace std ;
class  A_class
{      char name[20] ;
   public :    void  put_name(const char * s ) { strcpy_s( name, s ) ; }
               void  show_name() { cout << name << "\n" ; }
};
class  B_class  : public  A_class
{      char phone_num[ 20 ] ;
   public :    void  put_phone(const char * num )  { strcpy_s ( phone_num , num ) ; }
               void  show_phone()  { cout << phone_num << "\n" ; }
};
int main()
{ A_class  * A_p ;      A_class  A_obj ;
  B_class   B_obj ;
  A_p = & A_obj ;
  A_p -> put_name( "Wang xiao hua" ) ;   A_p -> show_name() ;
  A_p = & B_obj ;
  A_p -> put_name( "Chen ming" ) ;     A_p -> show_name() ;
  B_obj.put_phone ( "5555_12345678" );
  ( ( B_class * ) A_p ) -> show_phone() ;
}
```



调用

从基类继承的成员函数

例9-1 使用基类指针引用派生类对象

```
#include<iostream>
#include<cstring>
using namespace std ;
class  A_class
{      char name[20] ;
    public :    void  put_name(const char * s ) { strcpy_s( name, s ) ; }
                void  show_name() { cout << name << "\n" ; }
};
class  B_class  : public  A_class
{      char phone_num[ 20 ] ;
    public :    void  put_phone(const char * num )  { strcpy_s ( phone_num , num ) ; }
                void  show_phone()  { cout << phone_num << "\n" ; }
};
int main()
{ A_class  * A_p ;      A_class  A_obj ;
   B_class   B_obj ;
   A_p = & A_obj ;
   A_p -> put_name( "War                                    ow_name() ;
   A_p = & B_obj ;
   A_p -> put_name( "Chen ming" ) ;     A_p -> show_name() ;
   B_obj.put_phone ( "5555_12345678" );
   ( ( B_class * ) A_p ) -> show_phone() ;
}
```

class A_class

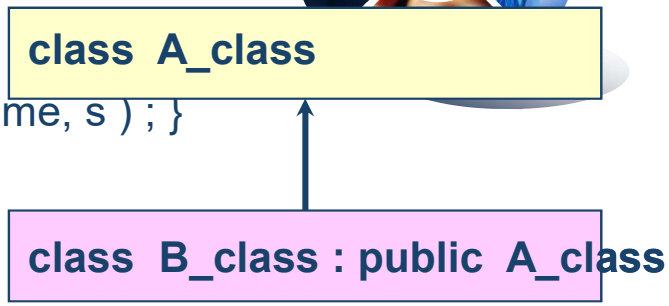
class B_class : public A_class

用派生类对象

调用派生类的成员函数

例9-1 使用基类指针引用派生类对象

```
#include<iostream>
#include<cstring>
using namespace std ;
class  A_class
{      char name[20] ;
    public :    void  put_name(const char * s ) { strcpy_s( name, s ) ; }
                void  show_name() { cout << name << "\n" ; }
};
class  B_class  : public  A_class
{      char phone_num[ 20 ] ;
    public :    void  put_phone(const char * num )  { strcpy_s ( phone_num , num ) ; }
                void  show_phone()  { cout << phone_num << "\n" ; }
};
int main()
{ A_class  * A_p ;      A_class  A_obj
   B_class   B_obj ;
   A_p = & A_obj ;
   A_p -> put_name( "Wang xiao hua" ) ;    A_p -> show_name() ;
   A_p = & B_obj ;
   A_p -> put_name( "Chen ming" ) ;     A_p -> show_name() ;
   B_obj.put_phone ( "5555_12345678" );
   ( ( B_class * ) A_p ) -> show_phone() ;
}
```

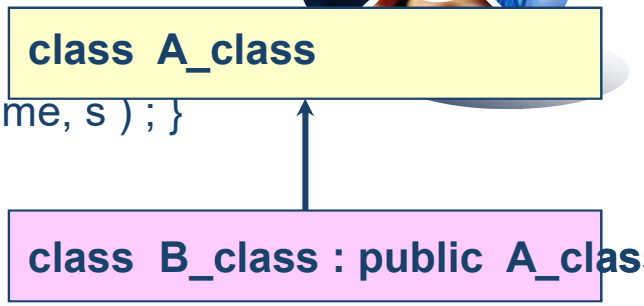


对基类指针强制类型转换

调用派生类的成员函数

例9-1 使用基类指针引用派生类对象

```
#include<iostream>
#include<cstring>
using namespace std ;
class  A_class
{      char name[20] ;
    public :    void  put_name(const char * s ) { strcpy_s( name, s ) ; }
                void  show_name() { cout << name << "\n" ; }
};
class  B_class  : public  A_class
{      char phone_num[ 20 ] ;
    public :    void  put_phone(const char * num )  { strcpy_s ( phone_num , num ) ; }
                void  show_phone()  { cout << phone_num << "\n" ; }
};
int main()
{ A_class  * A_p ;      A_class  A_obj ;
   B_class   B_obj ;
   A_p = & A_obj ;
   A_p -> put_name( "Wang xiao hua" ) ;   A_p -> show_name() ;
   A_p = & B_obj ;
   A_p -> put_name( "Chen ming" ) ;     A_p -> sho
   B_obj.put_phone ( "5555_12345678" );
   ( ( B_class * ) A_p ) -> show_phone() ;
}
```



9.2.2 派生类指针引用基类对象

- 派生类指针只有经过强制类型转换之后，才能引用基类对象的同名成员函数？

例9-2 日期时间程序。在派生类中调用基类同名成员函数

```
#include<iostream>
using namespace std ;
class Date
{ public:
      Date( int y, int m, int d )   { SetDate( y, m, d ); }
      void SetDate( int y, int m, int d ) { year = y ; month = m ; day = d ; }
      void Print() { cout << year << '/' << month << '/' << day << "; " ; }
  protected :    int year , month , day ;
} ;
class DateTime : public Date
{ public :
      DateTime( int y, int m, int d, int h, int mi, int s ) : Date( y, m, d ) { SetTime( h, mi,
s ); }
      void SetTime( int h, int mi, int s )  { hours = h;  minutes = mi;  seconds = s; }
      void Print() { cout << hours << ':' << minutes << ':' << seconds << '\n' ;  }
  private:    int hours , minutes , seconds ;
};
int main()
   { DateTime dt( 2009, 1, 1, 12, 30, 0 ) ;
     DateTime * pdt = &dt;
     ((Date) dt).Print();              //对象类型转换，调用基类成员函数
     dt.Print();
     ((Date *)pdt)->Print();           //对象指针类型转换，调用基类成员函数
     pdt->Print() ;
   }
```

例9-2 日期时间程序。在派生类中调用基类同名成员函数

```
#include<iostream>
using namespace std ;
class Date
{ public:
      Date( int y, int m, int d )   { SetDate( y, m, d ); }
      void SetDate( int y, int m, int d ) { year = y ; month = m ; day = d ; }
      void Print() { cout << year << '/' << month << '/' << day << "; " ; }
  protected :    int year , month , day ;
} ;
class DateTime : public Date
{ public :
      DateTime( int y, int m, int d,                    d ) { SetTime( h, mi,
s ); }
      void SetTime( int h, int mi, int s )                  = mi;  seconds = s; }
      void Print() { cout << hours << ':' << minutes << ':' << seconds << '\n' ;  }
  private:    int hours , minutes , se      nds ;
};
int main()
   { DateTime dt( 2009, 1, 1, 12, 30, 0 ) ;
     DateTime * pdt = &dt;
     dt.Date :: Print();          //对象类型转换，调用基类成员函数
     dt.Print();
     ((Date *)pdt)->Print();          //对象指针类型转换，调用基类成员函数
     pdt->Print() ;
   }
```

等价吗？

为什么？

例9-2 日期时间程序。在派生类中调用基类同名成员函数

```
#include<iostream>
using namespace std ;
class Date
{ public:
      Date( int y, int m, int d )   { SetDate( y, m, d ); }
      void SetDate( int y, int m, int d ) { year = y ; month = m ; day = d ; }
      void Print() { cout << year << '/' << month << '/' << day << "; " ; }
  protected :    int year , month , day ;
} ;
class DateTime : public Date
{ public :
      DateTime( int y, int m, int d, int h, int mi, int s ) : Date( y, m, d ) { SetTime( h, mi,
s ); }
      void SetTime( int h, int mi, int s ) { hours = h; minutes = mi; seconds = s; }
      void Print() { cout << hours << ':' << minutes << ':' << seconds << '\n' ;  }
  private:    int hours , minutes , seconds ;
};
int main()
   { DateTime dt( 2009, 1, 1, 12, 30, 0 ) ;
     DateTime * pdt = &dt;
     ((Date) dt).Print();          //对象类型转换，调用基类成员函数
     dt.Print();
     pdt->Date :: Print();         //对象指针类型转换，调用基类成员函数
     pdt->Print() ;
   }
```

等价吗？

为什么？

例9-2 日期时间程序。在派生类中调用基类同名成员函数

```
#include<iostream>
using namespace std ;
class Date
{ public:
      Date( int y, int m, int d )   { SetDate( y, m, d ); }
      void SetDate( int y, int m, int d ) { year = y ; month = m ; day = d ; }
      void Print() { cout << year << '/' << month << '/' << day << "; " ; }
  protected :    int year , month , day ;
} ;
class DateTime : public Date
{ public :
      DateTime( int y, int m, int d, int h, int mi, int s ) : Date( y, m, d ) { SetTime( h, mi,
s ); }
      void SetTime( int h, int mi, int s )  { hours = h;  minutes = mi;  seconds = s; }
      void Print()
        { ( ( Date * ) this ) -> Print();
            cout << hours << ':' << minutes << ':' << seconds << '\n' ;
        }
  private:    int hours , minutes , seconds ;
};
int main()
   { DateTime dt( 2009, 1, 1, 12, 30, 0 ) ;
     dt.Print() ;
   }
```

例9-2 日期时间程序。在派生类中调用基类同名成员函数

```
#include<iostream>
using namespace std ;
class Date
{ public:
      Date( int y, int m, int d )   { SetDate( y, m, d ); }
      void SetDate( int y, int m, int d ) { year = y ; month = m ; day = d ; }
      void Print() { cout << year << '/' << month << '/' << day << "; " ; }
  protected :    int year , month , day ;
} ;
class DateTime : public Date
{ public :
      DateTime( int y, int m, int d, int h, int mi, ir                    e( h, mi,
s ); }
      void SetTime( int h, int mi, int s )  { hours = h;  minutes = mi;  seconds = s; }
      void Print()
        { ( ( Date * ) this ) -> Print();
          cout << hours << ':' << minutes << ':' << seconds << '\n' ;
        }
  private:    int hours , minutes , seconds ;
};
int main()
   { DateTime dt( 2009, 1, 1, 12, 30, 0 ) ;
     dt.Print() ;
   }
```

对 **this** 指针作类型转换
调用基类同名成员函数

例9-2 日期时间程序。在派生类中调用基类同名成员函数

```
#include<iostream>
using namespace std ;
class Date
{ public:
      Date( int y, int m, int d )   { SetDate( y, m, d ); }
      void SetDate( int y, int m, int d ) { year = y ; month = m ; day = d ; }
      void Print() { cout << year << '/' << month << '/' << day << "; " ; }
  protected :    int year , month , day ;
} ;
class DateTime : public Date
{ public :
      DateTime( int y, int m, int d, int h, int mi, ir                    e( h, mi,
s ); }
      void SetTime( int h, int mi, int s )  { hours = h;  minutes = mi;  seconds = s; }
      void Print()
        { ( ( Date * ) this ) -> Print();
          cout << hours << ':' << minutes << ':' << seconds << '\n' ;
        }
  private:    int hours , minutes , seconds ;
};
int main()
   { DateTime dt( 2009, 1, 1, 12, 30, 0 ) ;
     dt.Print() ;
   }
```

对 **this** 指针作类型转换

调用基类同名成员函数

例9-2 日期时间程序。在派生类中调用基类同名成员函数

```
#include<iostream>
using namespace std ;
class Date
{ public:
      Date( int y, int m, int d )   { SetDate( y, m, d ); }
      void SetDate( int y, int m, int d ) { year = y ; month = m ; day = d ; }
      void Print() { cout << year << '/' << month << '/' << day << "; " ; }
  protected :    int year , month , day ;
} ;
class DateTime : public Date
{ public :
      DateTime( int y, int m, int d, int h, int mi, ir                    e( h, mi,
s ); }
      void SetTime( int h, int mi, int s )  { hours = h;  minutes = mi;  seconds = s; }
      void Print()
        { ( ( Date ) ( *this ) ) . Print();
          cout << hours << ':' << minutes << ':' << seconds << '\n' ;
        }
  private:    int hours , minutes , seconds ;
};
int main()
   { DateTime dt( 2009, 1, 1, 12, 30, 0 ) ;
     dt.Print() ;
   }
```

对 *this 对象作类型转换

调用基类同名成员函数

例9-2 日期时间程序。在派生类中调用基类同名成员函数

```
#include<iostream>
using namespace std ;
class Date
{ public:
      Date( int y, int m, int d )   { SetDate( y
      void SetDate( int y, int m, int d ) { ye                    }
      void Print() { cout << year << '/                          }
  protected :    int year , month , day ;
} ;
class DateTime : public Date
{ public :
      DateTime( int y, int m, int d, int h, int mi, int s ) : Date( y, m, d ) { SetTime( h, mi,
s ); }
      void SetTime( int h, int mi, int s )  { hours = h;  minutes = mi;  seconds = s; }
      void Print()
        {  Date :: Print();                    ;
           cout << hours << ':' << minutes << ':' << seconds << '\n' ;
        }
  private:    int hours , minutes , seconds ;
};
int main()
   { DateTime dt( 2009, 1, 1, 12, 30, 0 ) ;
     dt.Print() ;
   }
```

等价吗？

为什么？

MOOC视频与简评

清华郑莉、李超老师-1-导学（7分钟）：

简评：

1、多态性（Polymorphism）：操作接口具有表现多种不同形态的能力

2、静态联编（静态绑定，编译时绑定，早绑定）：函数重载、运算符重载

3、动态联编（动态绑定，运行时绑定，晚绑定）：使用“virtual”解决上一章那个“通用的display函数”的问题

MOOC视频与简评

回顾：清华郑莉老师-第七章 继承与派生-

5-基类与派生类类型转换（7分钟）：

类型转换

- 公有派生类对象可以被当作基类的对象使用，反之则不可。
  - 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。

```
#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

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

    return 0;
}
```

MOOC视频与简评

```
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

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

    return 0;
}
```

运行结果：

```
Base1::display()
Base1::display()
Base1::display()
```

MOOC视频与简评

清华郑莉老师-6-虚函数（11分钟）：

简评：

1、通过virtual关键字告诉编译器推迟到运行时决定调用（执行）哪个函数体。

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

class Base1 {
public:
  virtual void display() const;  //虚函数
};
void Base1::display() const {
  cout << "Base1::display()" << endl;
}
```

MOOC视频与简评

```
class Base2::public Base1 {
public:
    virtual void display() const;
};
void Base2::display() const {
   cout << "Base2::display()" << endl;
}
class Derived: public Base2 {
public:
    virtual void display() const;
};
void Derived::display() const {
   cout << "Derived::display()" << endl;
}
```

MOOC视频与简评

```
void fun(Base1 *ptr) {
    ptr->display();
}

int main() {
    Base1 base1;
    Base2 base2;
    Derived derived;
    fun(&base1);
    fun(&base2);
    fun(&derived);
    return 0;
}
```

运行结果：
Base1::display()
Base2::display()
Derived::display()

MOOC视频与简评

virtual 关键字

- 派生类可以不显式地用virtual声明虚函数，这时系统就会用以下规则来判断派生类的一个函数成员是不是虚函数：
  - 该函数是否与基类的虚函数有相同的名称、参数个数及对应参数类型；
  - 该函数是否与基类的虚函数有相同的返回值或者满足类型兼容规则的指针、引用型的返回值；
- 如果从名称、参数及返回值三个方面检查之后，派生类的函数满足上述条件，就会自动确定为虚函数。这时，派生类的虚函数便覆盖了基类的虚函数。
- 派生类中的虚函数还会隐藏基类中同名函数的所有其它重载形式。
- 一般习惯于在派生类的函数中也使用virtual关键字，以增加程序的可读性。

MOOC视频与简评

回顾：清华郑莉老师-第七章 继承与派生-

清华郑莉老师-9-派生类的析构函数（5分钟）：

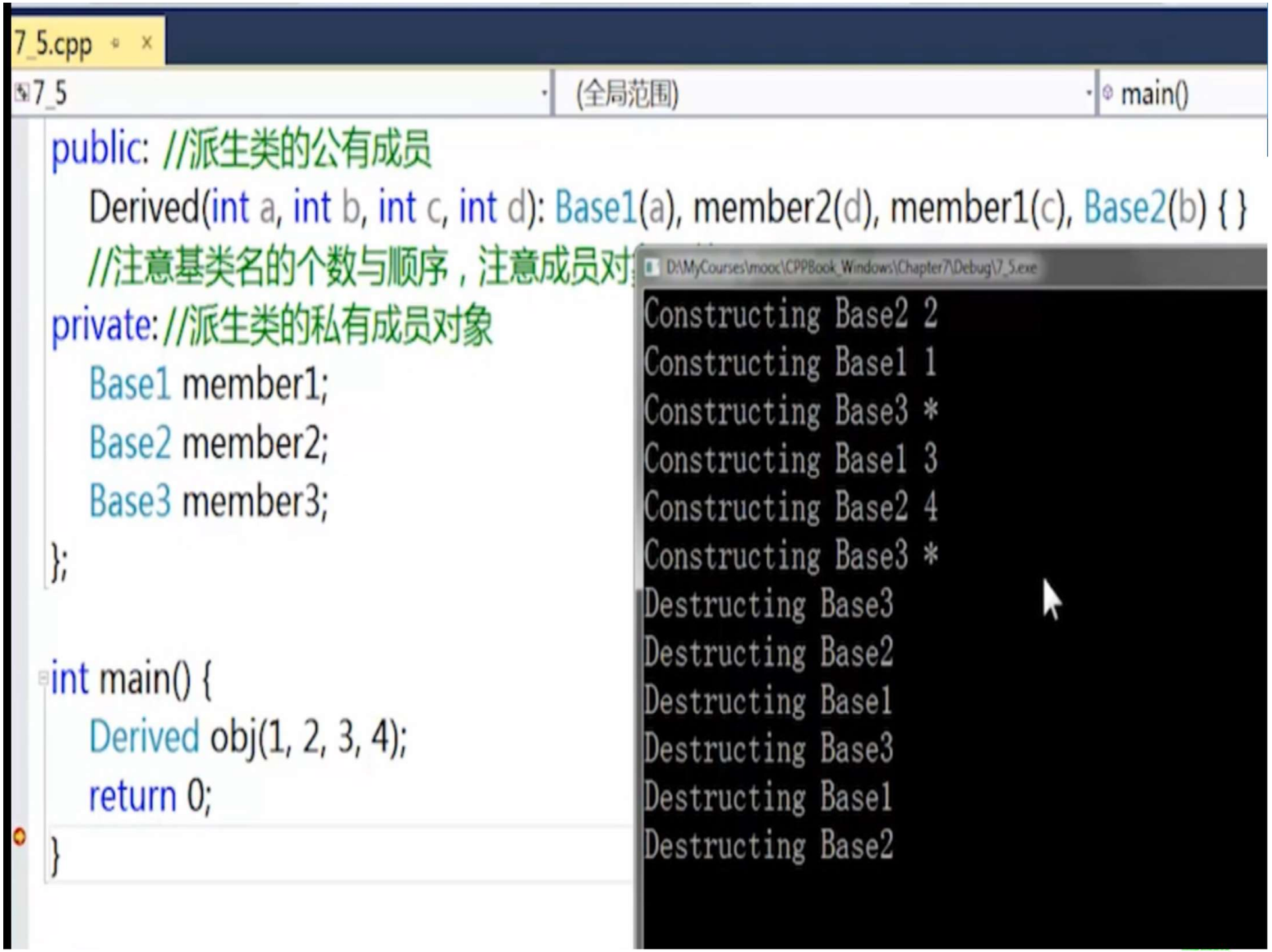
- 析构函数不被继承，派生类如果需要，要自行声明析构函数。
- 声明方法与无继承关系时类的析构函数相同。
- 不需要显式地调用基类的析构函数，系统会自动隐式调用。
- 先执行派生类析构函数的函数体，再调用基类的析构函数。

MOOC视频与简评

```cpp
//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; }
};
```

MOOC视频与简评

```
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;
```


7_5.cpp
7_5
(全局范围)
main()
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
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;
}
D:\MyCourses\mooc\CPPBook_Windows\Chapter7\Debug\7_5.exe
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

MOOC视频与简评

清华郑莉老师-7-虚析构函数（7分钟）：

简评：

如果你打算允许其他人通过基类指针调用对象的析构函数（通过delete这样做是正常的），就需要让基类的析构函数成为虚函数，否则执行delete的结果是不确定的。

MOOC视频与简评

```
#include <iostream>
using namespace std;
class Base {
public:
   ~Base(); //不是虚函数
};
Base::~Base() {
   cout<< "Base destructor" << endl;
 }
class Derived: public Base{
public:
   Derived();
   ~Derived(); //不是虚函数
private:
   int *p;
};
```

```
Derived::Derived() {
        p = new int(0);
}
Derived::~Derived() {
  cout <<  "Derived destructor" << endl;
        delete p;
}

void fun(Base* b) {
        delete b; //静态绑定，只会调用~Base()
}

int main() {
        Base *b = new Derived();
        fun(b);
        return 0;
}
```

运行结果：

Base destructor

MOOC视频与简评

```
#include <iostream>
using namespace std;
class Base {
public:
   virtual ~Base();
};
Base::~Base() {
   cout<< "Base destructor" << endl;
 }
class Derived: public Base{
public:
   Derived();
   virtual ~Derived();
private:
   int *p;
};
```

```
Derived::Derived() {
        p = new int(0);
}
Derived::~Derived() {
  cout <<  "Derived destructor"<<endl;
        delete p;
}

void fun(Base* b) {
        delete b;
}

int main() {
        Base *b = new Derived();
        fun(b);
        return 0;
}
```

运行结果：

```
Derived destructor
Base destructor
```

清華大學

MOOC视频与简评

清华郑莉老师-8-虚表和动态绑定（6分钟）：

虚表

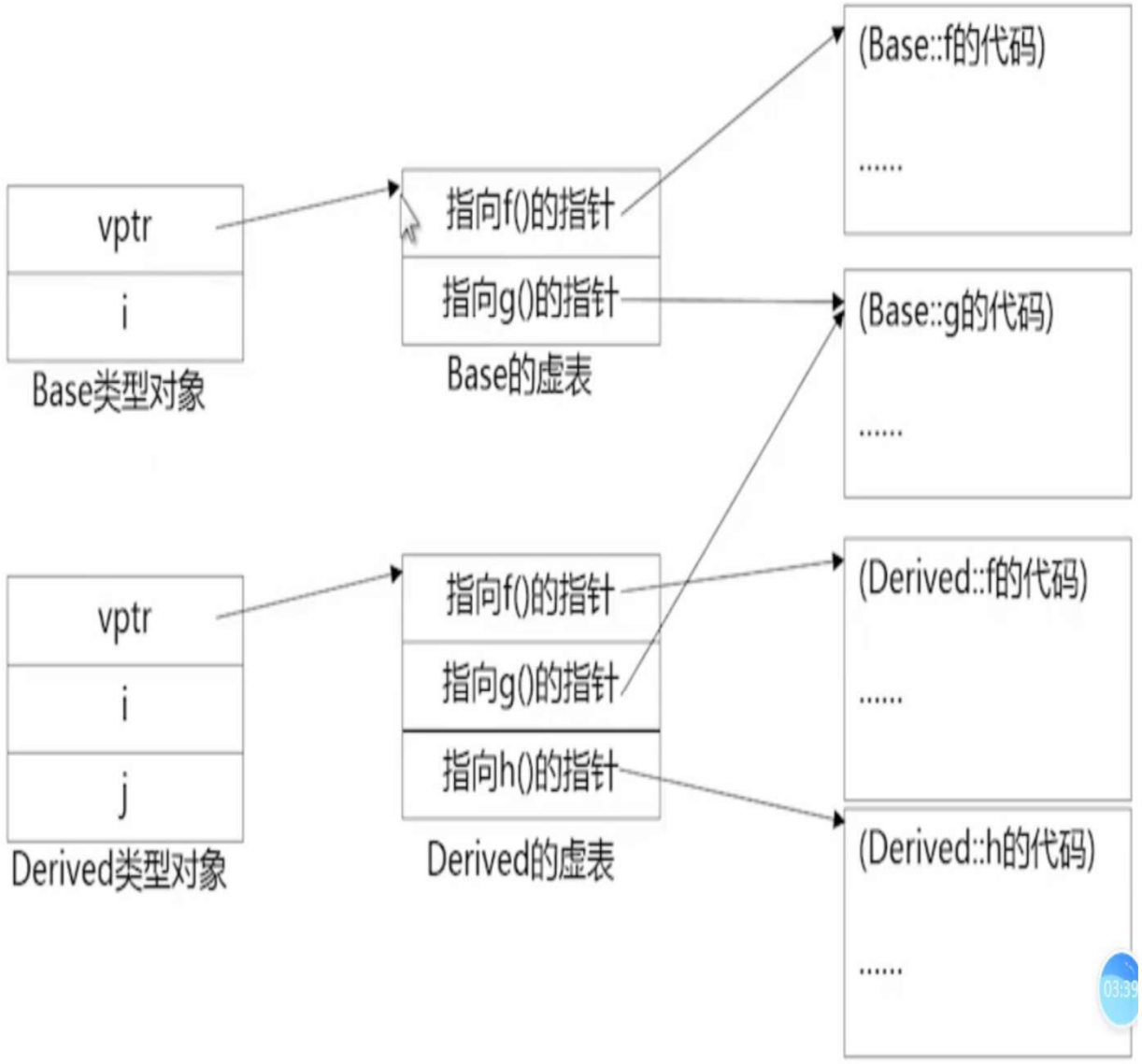
- 每个多态类有一个虚表（virtual table）
- 虚表中有当前类的各个虚函数的入口地址
- 每个对象有一个指向当前类的虚表的指针（虚指针vptr）

动态绑定的实现

- 构造函数中为对象的虚指针赋值
- 通过多态类型的指针或引用调用成员函数时，通过虚指针找到虚表，进而找到所调用的虚函数的入口地址
- 通过该入口地址调用虚函数

MOOC视频与简评

```
class Base {
public:
    virtual void f();
    virtual void g();
private:
    int i;
};

class Derived: public Base {
public:
    virtual void f(); //覆盖Base::f
    virtual void h(); //新增的虚函数
private:
    int j;
};
```



MOOC视频与简评

清华郑莉老师-9-抽象类（8分钟）：

简评：

纯虚函数

- 纯虚函数是一个在基类中声明的虚函数，它在该基类中没有定义具体的操作内容，要求各派生类根据实际需要定义自己的版本，纯虚函数的声明格式为：

```
virtual 函数类型 函数名(参数表) = 0;
```

MOOC视频与简评

清华郑莉老师-9-抽象类（8分钟）：

抽象类作用

- 将有关的数据和行为组织在一个继承层次结构中，保证派生类具有要求的行为。
- 对于暂时无法实现的函数，可以声明为纯虚函数，留给派生类去实现。

注意

- 抽象类只能作为基类来使用。
- 不能定义抽象类的对象。

MOOC视频与简评

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

class Base1 {
public:
   virtual void display() const = 0; //纯虚函数
};

class Base2: public Base1 {
public:
   virtual void display() const; //覆盖基类的虚函数
};
void Base2::display() const {
   cout << "Base2::display()" << endl;
}
```

给定义成纯虚函数

MOOC视频与简评

```
class Derived: public Base2 {
public:
    virtual void display() const; //覆盖基类的虚函数
};
void Derived::display() const {
   cout << "Derived::display()" << endl;
}
void fun(Base1 *ptr) {
   ptr->display();
}
int main() {
   Base2 base2;
   Derived derived;
   fun(&base2);
   fun(&derived);
   return 0;
}
```

运行结果：

Base2::display()

Derived::display()

也实现了这个display函数

MOOC视频与简评

清华郑莉老师-10-override与final（9分钟）：

override

- 多态行为的基础：基类声明虚函数，派生类声明一个函数覆盖该虚函数；
- 覆盖要求：函数签名（signature）完全一致。
- 函数签名包括：函数名 参数列表 const

MOOC视频与简评

清华郑莉老师-10-override与final（9分钟）：

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

class Base {
public:
  virtual void f1(int) const;
  virtual ~Base() {
};
};

void Base::f1(int) const {
  cout << "Base f1" << endl;
  return;
}
```

```
class Derived: public Base {
public:
  void f1(int);
  ~Derived() {
};
};

void Derived::f1(int) {
  cout << "derived f1" << endl;
}
```

```
int main() {
  Base *b;
  b = new Base;
  b->f1(1);
  b = new Derived;
  b->f1(1);
  return 0;
}
```

运行结果
Base f1
Base f1

MOOC视频与简评

- C++11 引入显式函数覆盖，在编译期而非运行期捕获此类错误。
- 在虚函数显式重载中运用，编译器会检查基类是否存在一虚拟函数，与派生类中带有声明override的虚拟函数，有相同的函数签名（signature）；若不存在，则会回报错误。

结果没写对

自动连播

MOOC视频与简评

```
struct Base1 final { };

struct Derived1 : Base1 { }; // 编译错误：Base1为final，不允许被继承

struct Base2 {
    virtual void f() final;
};

struct Derived2 : Base2 {
    void f(); // 编译错误：Base2::f 为final，不允许被覆盖
};
```

MOOC视频与简评

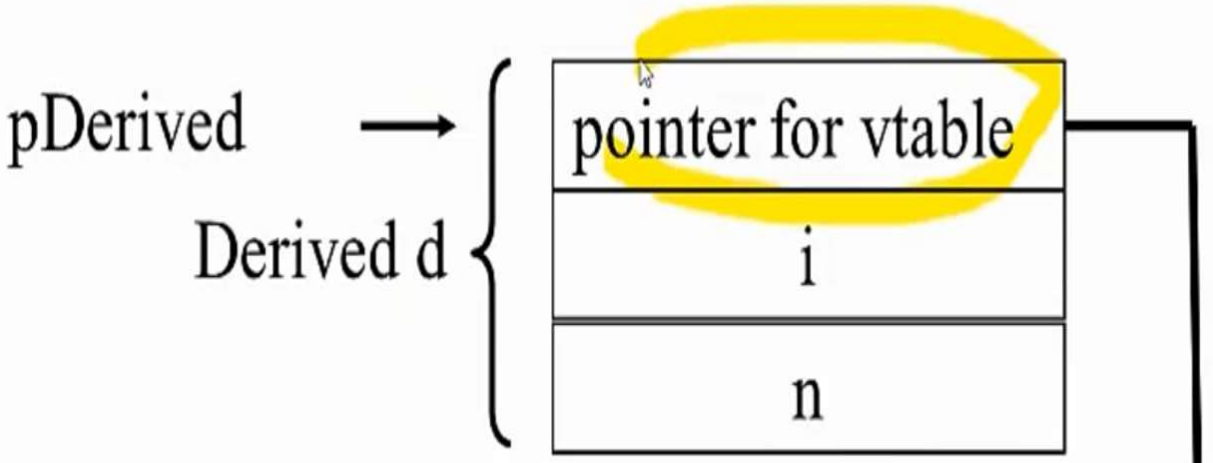
北大郭炜老师-第六周多态-4-多态的实现原理（15分钟）：

简评：

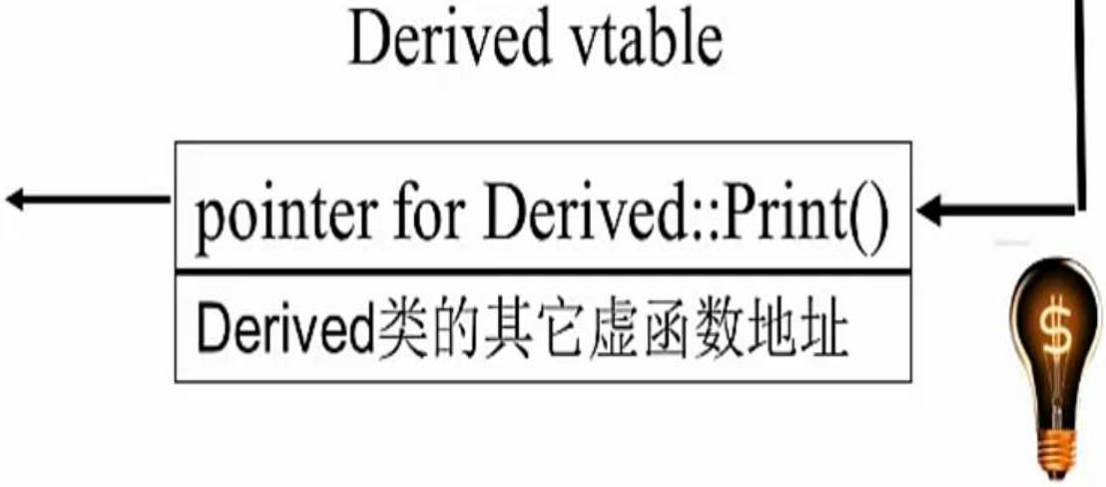
```
class Base {
        public:
        int i;
        virtual void Print() { cout << "Base:Print" ; }
};
class Derived : public Base{
        public:
        int n;
        virtual void Print() {  cout <<"Drived:Print" << endl; }
};
int main() {
        Derived d;
        cout << sizeof( Base) << ","<< sizeof( Derived ) ;
        return 0;
}
```

程序运行输出结果： 8, 12

多态实现的关键 --- 虚函数表




```
pBase = pDerived;
pBase->Print();
```



➢多态的函数调用语句被编译成一系列根据基类指针所指向的（或基类引用所引用的）对象中存放的虚函数表的地址，在虚函数表中查找虚函数地址，并调用虚函数的指令。

MOOC视频与简评

中国大学MOOC

```
#include <iostream>
using namespace std;
class A   {
        public: virtual void Func() { cout << "A::Func" << endl; }
};
class B:public A  {
        public: virtual void Func() { cout << "B::Func" << endl; }
};
int main()  {
        A a;
        A * pa = new B();
        pa->Func();
        //64位程序指针为8字节
        long long * p1 = (long long * ) & a;
        long long * p2 = (long long * ) pa;
        * p2 = * p1;
        pa->Func();
        return 0;
}
```

MOOC视频与简评

中国大学MOOC

```
#include <iostream>
using namespace std;
class A   {
        public: virtual void Func() { cout << "A::Func" << endl; }
};
class B:public A  {
        public: virtual void Func() { cout << "B::Func" << endl; }
};
int main()  {
        A a;
        A * pa = new B();
        pa->Func();
        //64位程序指针为8字节
        long long * p1 = (long long * ) & a;
        long long * p2 = (long long * ) pa;
        * p2 = * p1;
        pa->Func();
        return 0;
}
```

MOOC视频与简评

清华郑莉老师-11-本章小结（8分钟）：

作　　业

习题

第9章 测验题

单选题 1分

设置

（9.2）静态联编又叫作（　）。

A 早期联编

B 晚期联编

C 延迟联编

D 以上三者都行

提交

单选题 1分

设置

（9.2）基类的指针与派生类指针，可以分别指向基类对象或派生类对象而形成4种情形。在这4种情形中，需要进行强制类型转换的是（　）。

A 基类指针指向基类对象

B 基类指针指向派生类对象

C 派生类指针指向基类对象

D 派生类指针指向派生类对象

提交

单选题 1分

设置

（9.2）当基类指针指向派生类对象时，（ ）。

A 只能调用基类自己定义的成员函数

B 发生语法错误

C 可以调用派生类的全部成员函数

D 以上说法全部错误

提交

GU CN Computer Networks Institute

单选题 1分

设置

（9.2）当基类指针指向派生类对象时，利用基类指针调用派生类中与基类同名但被派生类重写后的成员函数时，调用的是（　）。

A 基类的成员函数

B 派生类的成员函数

C 不确定

D 先调用基类的，再调用派生类的

提交

单选题 1分

设置

（9.2）当派生类指针指向基类对象时（　）。

A 可以直接调用基类的成员函数

B 可以调用派生类对象的成员函数

C 此处必须强制将派生类指针转换成基类指针才能调用基类的成员函数

D 以上说法都不对

提交

GU CN Computer Networks Institute

单选题 1分

设置

（9.3）在C++中，要实现动态联编，必须使用（　）调用虚函数。

A 基类指针

B 对象名

C 派生类指针

D 类名

提交

单选题 1分

设置

(9.3) 下列函数中，不能说明为虚函数的是（ ）。

A 析构函数

B 构造函数

C 公有成员函数

D 私有成员函数

提交

单选题 1分

设置

（9.3）在派生类中，重载一个虚函数时，要求函数名、参数的个数、参数的类型、参数的顺序和函数的返回值(　)。

A 部分相同

B 相容

C 不同

D 相同

提交

单选题 1分

设置

（9.3）下面关于构造函数和析构函数的描述，错误的是（　）。

A 析构函数中调用虚函数采用静态联编

B 对虚析构函数的调用可以采用动态联编

C 当基类的析构函数是虚函数时，其派生类的析构函数也一定是虚函数

D 构造函数可以声明为虚函数

提交

单选题 1分

设置

（9.3）在C++中，根据（　）识别类层次中不同类定义的虚函数版本。

A 参数个数

B 参数类型

C 函数名

D this指针类型

提交

单选题 1分

设置

（9.3）虚析构函数的作用是（　）。

A 虚基类必须定义虚析构函数

B 类对象作用域结束时释放资源

C delete动态对象时释放资源

D 无意义

提交

单选题 1分

设置

（9.4）在下面函数原型中，（　）声明了fun为纯虚函数。

A　void fun()=0;

B　virtual void fun()=0;

C　virtual void fun();

D　virtual void fun(){ };

提交

单选题 1分

设置

（9.4）若一个类中含有纯虚函数，则该类称为（　）。

A 基类

B 纯基类

C 抽象类

D 派生类

提交

单选题 1分

设置

（9.4）假设Aclass为抽象类，下列正确的说明语句是（　）。

A　Aclass fun(int) ;

B　Aclass * p ;

C　int fun(Aclass) ;

D　Aclass Obj ;

提交

单选题 1分

设置

（9.4）下面描述中，正确的是（　）。

A 虚函数是没有实现的函数

B 纯虚函数是返回值等于0的函数

C 抽象类是只有纯虚函数的类

D 抽象类指针可以指向不同的派生类

提交

单选题 1分

设置

（9.4）异质链表是（　）。

A 用数组组织类对象

B 用链表组织类对象

C 用抽象类指针指向派生类对象

D 用抽象类指针构造派生类对象链表

提交