



第5章 多态

ffh



概述

- ◆ 多态性是不同的对象调用相同名称的函数，并可导致完全不同的行为的现象。 “同一接口，多种方法”。
- ◆ C++中，将多态性分为两类：编译时的多态性和运行时的多态性。

- 编译时的多态性

编译时的多态性是通过函数的重载或运算符的重载来实现的。

函数的重载是根据函数调用时，给出不同类型的实参或不同的实参个数，在程序执行前就能确定调用哪一个函数。

对于运算符的重载，是根据不同的运算对象在编译时就可确定执行哪一种运算。



概述

– 运行时的多态性

运行时的多态性是指在程序执行前，根据函数名和参数无法确定应该调用哪一个函数，必须在程序执行过程中，根据具体执行情况来动态地确定。又叫运行期绑定（**run-time binding**）

这种多态性是通过类的继承关系和虚函数来实现的，主要用于实现一些通用程序的设计。



5.1 运行期绑定与编译期绑定

◆ 入口地址（entry point）

构成C++程序的各个函数分别在计算机的内存中拥有了一段存储空间，一个函数在内存中起始的地址（starting address）称为这个函数的入口地址（entry point）。

例如，每个C++程序都有一个名为main的顶层函数，main函数在内存中获得的存储空间的起始地址就是main函数的入口地址。

◆ 多态就是将函数名称动态地绑定到函数入口地址的运行期绑定机制。

- 多态性实现：通过联编(binding，也称聚束)。
- 联编：将一个函数调用链接上相应的函数体代码的过程。





5.1 运行期绑定与编译期绑定

◆ 例5-1

```
#include <iostream>
using namespace std;
void sayHi( );
int main( )
{   sayHi( );
    return 0;
}
void sayHi( )
{   cout<<"Hello,cruel world!"<<endl;
}
```





5.1 运行期绑定与编译期绑定

- ◆ 编译器将所有对sayHi的调用绑定到实现sayHi的代码处，例如上例中main函数中对sayHi的调用被绑定到上面那条cout语句处。
- ◆ 由于sayHi函数被调用时，到底应该执行哪一段代码是由编译器在编译阶段就决定了的，因此我们将这种对sayHi函数的绑定方式称为**编译期绑定**
- ◆ 与编译期绑定不同的是，运行期绑定是直到程序运行时（而不是在编译时刻），才将函数名称绑定到其入口地址。
- ◆ 如果对一个函数的绑定发生在运行时刻而非编译时刻，我们就称该函数是多态的。





5.1 运行期绑定与编译期绑定

- ◆ 在纯面向对象语言（如Smalltalk）中，所有的函数都是多态的。而像C++这样的混合语言，函数既可以是多态的，也可以是非多态的，这要由绑定的时机是编译时刻还是运行时刻来决定。





5.1.1 C++多态的前提条件

◆ 虚函数

在基类中用关键字 **virtual** 修饰的成员函数称为虚成员函数，虚函数的定义格式为：

```
virtual <类名> <函数名> (参数)
    {函数体}
```

◆ C++中，只有满足某些特定条件的成员函数才可能是多态的。

- 必须存在一个继承体系结构。
- 继承体系结构中的一些类必须具有同名的虚成员函数。
- 至少有一个基类类型的指针或基类类型的引用。这个指针或引用可用来对虚成员函数进行调用。





5.1.1 C++多态的前提条件

◆ 例5-2

阐明多态及其三个前提条件的例子：(代码见下页)

- 存在一个继承体系结构，**TradesPerson**是基类，**Tinker**和**Tailor**是**TradesPerson**的派生类。
- 继承体系结构中有一个叫**sayHi**的虚成员函数，它在上述三个类中各自被定义了一次，因此共有三个不同的定义(但函数名相同)。
- **p**是个基类类型的指针。本例中**p**的数据类型是**TradesPerson***。指针**p**用来实施对虚成员函数**sayHi**的调用。





5.1.1 C++多态的前提条件

```
#include <iostream> // (例5-2)
using namespace std;
class TradesPerson
{ public:
    virtual void sayHi()
    {      cout<<"Just hi."<<endl;  }
};
class Tinker : public TradesPerson
{ public:
    virtual void sayHi()
    {      cout<<"Hi, I tinker."<<endl;      }
};
```





5.1.1 C++多态的前提条件

```
class Tailor : public TradesPerson
{ public:
    virtual void sayHi()
    {      cout<<"Hi, I tailor."<<endl;      }
};
int main()
{ TradesPerson* p;
  int which;
  do {
    cout<<"1 == TradesPerson, 2 == Tinker, 3 == Tailor";
    cin>>which;
  } while (which < 1 || which > 3);
```





5.1.1 C++多态的前提条件

switch (which)

```
{    case 1 : p = new TradesPerson; break;
      case 2 : p = new Tinker; break;
      case 3 : p = new Tailor; break;
}
```

p->sayHi(); //动态绑定

delete p;

return 0;

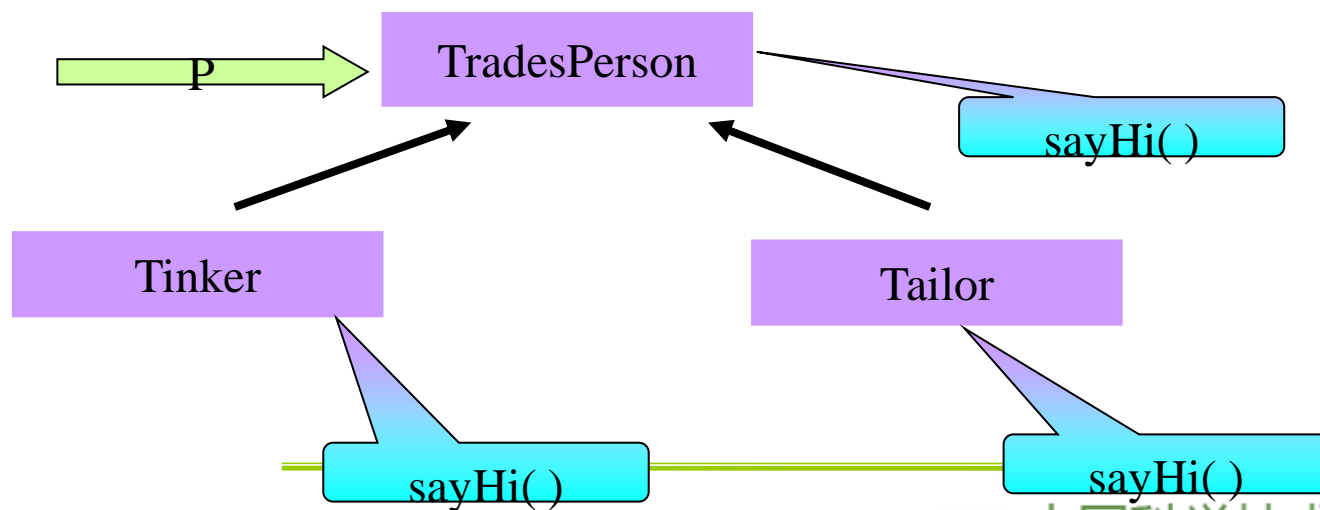
}





5.1.1 C++多态的前提条件

- ◆ 上例中：指针p可以指向：
 - TradesPerson对象， 或
 - Tinker对象 或
 - Tailor对象
- ◆ 不需要进行强制类型转换，因为基类类型的指针可以指向任何基类对象或派生类对象。





5.1.1 C++多态的前提条件

- ◆ 由于sayHi是虚函数，系统在运行时期才对调用动作进行绑定。
 - 如果p实际指向一个TradesPerson对象，系统将p->sayHi()这个调用动作绑定到TradesPerson::sayHi
 - 如果p实际指向Tinker对象，系统就将p->sayHi()这个调用动作绑定到Tinker::sayHi
 - 如果p实际指向Tailor对象，系统就将p->sayHi()这个调用动作绑定到Tailor::sayHi





5.1.1 C++多态的前提条件

- ◆ 例5-3 基类型的指针数组。用随机数决定生成哪个对象。

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
class TradesPerson
{ public:
    virtual void sayHi() {cout<<"Just hi."<<endl;}
};
class Tinker : public TradesPerson
{ public:
    virtual void sayHi() {cout<<"Hi, I tinker."<<endl;}
};
```





5.1.1 C++多态的前提条件

```
class Tailor : public TradesPerson
{ public:
    virtual void sayHi() {cout<<"Hi, I tailor."<<endl;}
};
```

```
int main() {
    srand( time(0) );
    TradesPerson* ptrs[10];
    unsigned which , i ;

    for (i=0; i<10; i++) {
        which = 1 + rand() % 3;
```





5.1.1 C++多态的前提条件

```
switch (which) {  
    case 1 : ptrs[i] = new TradesPerson; break;  
    case 2 : ptrs[i] = new Tinker; break;  
    case 3 : ptrs[i] = new Tailor; break;  
}  
}  
  
for (i=0; i<10; i++) {  
    ptrs[i] -> sayHi();  
    delete ptrs[ i ];  
}  
return 0;  
}
```





5.1.1 C++多态的前提条件

◆ `int rand(void);`

依据`srand (seed)`中指定的`seed`，返回一个随机整数。

一般计算机的随机数都是伪随机数，以一个数（种子）作为初始条件，然后用某种算法不停迭代产生随机数。

◆ `void srand(unsigned seed);`

用来初始化`rand()`的种子。

如果希望`rand()` 在每次程序运行时产生值不一样，必须给`srand(seed)`中的`seed`一个变值，这个变值必须在每次程序运行时都不一样，如常常使用系统时间来初始化，即：

```
srand(time(0));
```





5.1.1 C++多态的前提条件

- ◆ 当声明了基类的一个成员函数为虚函数后，即使该成员函数没有在派生类中被显式地声明为虚函数，但它在所有派生类中也将自动成为虚函数。

```
class TradesPerson
{ public:
    virtual void sayHi()      {      cout<<"Just hi."<<endl; }
};

class Tinker : public TradesPerson
{ public:
    void sayHi()      {      cout<<"Hi, I tinker."<<endl;      }
};
```





5.1.1 C++多态的前提条件

- ◆ 如果在派生类中的**sayHi**成员函数的声明没有使用关键字**virtual**，该派生类的用户为了确定它是否为虚函数，不得不检查**sayHi**在基类中的声明，将函数在所有派生类中声明为虚函数，就可以避免这种不便。
- ◆ 如果虚函数在类声明之外定义，关键字**virtual**仅在函数声明时需要，不需在函数定义中使用**virtual**关键字。

```
class C {    //例5-4
    public:
        virtual void m( );
        //...
};
void C::m( ) { //... }
```





5.1.1 C++多态的前提条件

- ◆ C++仅允许将成员函数定义为虚函数，顶层函数不能为虚函数。所以下列声明是错误的。

例5-5

```
virtual void f( ); //error
int main( )
{
}
```





补充举例

◆ 例：（5.1.1补充举例.txt）

定义基类High，其数据成员为高H，定义成员函数Show()为虚函数。

然后再由High派生出长方体类Cuboid与圆柱体类Cylinder。并在两个派生类中定义成员函数Show()为虚函数。

在主函数中，用基类High定义指针变量p，然后用指针p动态调用基类与派生类中虚函数Show()，显示长方体与圆柱体的体积。





举例

```
#include <iostream>
using namespace std;
class High
{ protected:
    float H;
public:
    High(float h)
    { H=h;}
    virtual void Show()           //在基类中定义虚函数Show()
    { cout<<"High="<<H<<endl;}
};
```





举例

```
class Cuboid:public High
{ private:
    float Length,Width;
public:
    Cuboid(float l=0,float w=0,float h=0):High(h)
    { Length=l; Width=w;}
    virtual void Show() //在长方体派生类中定义虚函数Show()
    { cout<<"Length="<<Length<<"\t";
      cout<<"Width="<<Width<<"\t";
      cout<<"High="<<H<<"\n";
      cout<<"Cuboid Volume="<<Length*Width*H<<endl;
    }
};
```





举例

```
class Cylinder:public High
{ private:
    float R;
public:
    Cylinder(float r=0,float h=0):High(h)
    {R=r;}
    virtual void Show() //在圆柱体派生类中定义虚函数Show()
    { cout<<"Radius="<<R<<"\t";
      cout<<"High="<<H<<"\n";
      cout<<"Cylinder Volume="<<R*R*3.1415*H<<endl;
    }
};
```





举例

```
void main(void)
{ High h(10),*p;
  Cuboid cu(10,10,10);
  Cylinder cy(10,10);
  h.Show();           // High=10
  cu.Show();
  cy.Show();
  p=&h;  p->Show();
  p=&cu; p->Show();
  p=&cy; p->Show();
}
```





举例

执行程序后输出：

High=10

Length=10 Width=10

High=10

Cubiod Volume=1000

Radius=10 High=10

Cylinder Volume=3141.5

High=10

Length=10 Width=10

High=10

Cubiod Volume=1000

Radius=10 High=10

Cylinder Volume=3141.5





练习

- ◆ 在C++中,要实现动态联编,必须使用(D)调用虚函数。
 - A.类名
 - B.派生类指针
 - C.对象名
 - D.基类指针
- ◆ 当一个类的某个函数被说明为virtual时, 该函数在该类的所有派生类中(A)。
 - A. 都是虚函数
 - B. 只有被重新说明时才是虚函数
 - C. 只有被重新说明为virtual时才是虚函数
 - D. 都不是虚函数
- ◆ 类B是类A的公有派生类, 类A和类B中都定义了虚函数func(), p是一个指向类A对象的指针, 则p->A::func()将(A)。
 - A. 调用类A中的函数func()
静态绑定
 - B. 调用类B中的函数func()
 - C. 既调用类A中函数, 也调用类B中的函数



练习

写出下列程序运行结果

```
class A { public:  virtual void func( )
                {cout<<"func in class A"<<endl;} };
class B{ public:  virtual void func( )
                {cout<<"func in class B"<<endl;} };
class C:public A,public B {
    public: void func( ) {cout<<"func in class C"<<endl;} };

int main( )
{
    C c;    A& pa=c; B& pb=c; C& pc=c;
    pa.func( ); pb.func( ); pc.func( );
}
```

输出：
func in class C
func in class C
func in class C





[练习] 分析程序，回答问题

```
class A // ( ppt28问题.txt )
{ public:
    virtual void act1( ) { cout << "A::act1( ) called." << endl; }
    void act2( ) { act1( ); }
};
class B : public A
{ public:
    void act1( ) { cout << "B::act1( ) called." << endl; }
};
void main ( )
{ B b;
    b.act2( );
}
```

输出：B::act1() called.





回答下列问题:

1. 该程序执行后的输出结果是什么？为什么？

`B::act1()` called.

因为B是A的派生类，`act1()`是类A中的虚函数，类B中的`act1()`自然是虚函数。

在`main()`函数中，`b.act2()`；调用类B中的`act2()`函数，B是子类，实际上调用`A::act2()`函数（`B::act2()`与`A::act2()`有相同的函数入口地址），在`A::act2()`函数的实现中调用`act1()`，由于有两个`act1()`函数，并且是虚函数，于是便产生动态联编，根据运行时的情况选择了`B::act1()`。所以输出结果是：`B::act1()` called.





回答下列问题:

2. 如果将A::act2()的实现改为:

```
void A::act2( )  
{  this->act1( );  
}
```

输出结果如何?

输出结果与前面相同。

因为加了**this**的限定，与没加是一样的，**this**是指向操作该成员函数的对象的指针。





回答下列问题:

3. 如将主函数**b.act2()** 改为:

b.A::act2();

输出结果是什么? 为什么?

B::act1() called.

原因同1: 调用**b.A::act2()**函数, 在**A::act2()**函数的实现中调用**act1()**, 由于**act1()**是虚函数, 于是便产生动态联编, 根据运行时的情况(当前是通过**b**对象调用, 即指向操作该成员函数的对象的指针**this**是**b**)选择了**B::act1()**。所以输出结果是: **B::act1() called.**





回答下列问题:

4. 如将主函数**b.act2()** 改为:

b.B::act2();

输出结果是什么? 为什么?

B::act1() called.

原因同上: **b.B::act2()** 与 **b.A::act2()** 或 **b.act2()** 等同。

调用**b.B::act2()**函数, 调用类B中的**act2()**函数, B是派生类实际上调用**A::act2()**函数, 在**A::act2()**函数的实现中调用**act1()**, 由于**act1()**是虚函数, 于是产生动态联编, 根据运行时的情况(当前是通过**b**对象调用, 即指向操作该成员函数的对象的指针**this**是**b**)选择了**B::act1()**。所以输出结果是: **B::act1() called.**





回答下列问题:

5. 如果将A::act1() 改为非虚函数，输出结果如何？

输出结果如下：

A::act1() called.

在main()函数中，b.act2(); 调用类B中的act2()函数，B是派生类实际上调用A::act2()函数，在A::act2()函数的实现中调用act1()，act1()为非虚函数，对act1()函数的调用进行的是静态联编，所以输出结果是：A::act1() called.





回答下列问题:

6. 如果将A::act1() 改为非虚函数，主函数b.act2() 改为:

b.B::act2();

输出结果是什么?

输出结果如下:

A::act1() called.

原因同上，在main()函数中，b.B::act2(); 调用类B中的act2()函数，B是派生类实际上调用A::act2()函数，在A::act2()函数的实现中调用act1()，act1()为非虚函数，对act1()函数的调用进行的是静态联编，所以输出结果是：A::act1() called.



回答下列问题:

7. 如果将A::act2()的实现改为:

```
void A::act2( )  
{    A::act1( );  
}
```

输出结果如何?

输出结果如下:

A::act1() called.

由于加了成员名的限定,在对act1()函数的调用进行的是静态联编,结果A::act1()函数被调用,因此产生上述结果。





5.1.2 虚成员函数继承

- ◆ 和普通成员函数一样，在派生类中虚成员函数也可以从基类继承。

例5-6

```
class TradesPerson{
    public:
        virtual void sayHi( ) {cout<<"Just hi."<<endl;}    };
class Tinker: public TradesPerson {
    //Remove Tinker::sayHi
};
int main( ){
    Tinker t1;
    t1.sayHi( ); // inherited sayHi
    //...
}
```

输出为：
Just hi.





5.1.3 运行期绑定和虚成员函数表

- ◆ C++使用vtable（虚成员函数表）来实现虚成员函数的运行期绑定。
- ◆ 虚成员函数表存在的用途是支持运行时查询，使得系统可以将某一函数名绑定到虚成员函数表中的特定入口地址。
- ◆ 虚成员函数表的实现是与系统无关的，程序员在使用虚成员函数时不需了解虚成员函数表的运行机制，甚至不需知道它的存在。





5.1.3 运行期绑定和虚成员函数表

```
class B{
public:
    virtual void m1( ){ }
    virtual void m2( ){ }
};
class D:public B{
public:virtual void m1( ){ }
};
```

虚成员函数	入口地址示例	虚成员函数	入口地址示例
B::m1	0x7723	D::m1	0x99a7
B::m2	0x23b4	D::m2	0x23b4





运行期绑定和虚成员函数表

```
int main() {  
    B  b1;    // base class object  
    D  d1;    // derived class object  
    B* p;     // pointer to base class  
    //...     // p is set to b1's or d1's address  
    p->m1();  //*** vtable lookup for run-time binding  
    //...  
}
```

p指向哪个对象，则执行相应的函数体。

- 使用动态绑定的程序会影响效率

因为虚成员函数表需要额外的存储空间，而且对虚成员函数表进行查询也需要额外的时间。





5.1.4 构造函数与析构函数

- ◆ 构造函数不能是虚成员函数，但析构函数可以是虚成员函数。

例5-8

```
class C{  
    public:  
        virtual C( );    //error  
        virtual C(int );    //error  
        virtual ~C( );    //OK  
        virtual void m( );  
}
```



为什么构造函数不可以是虚函数

- ◆ 原因在于虚拟调用是一种能够在给定信息不完全的情况下工作的机制。

特别地，虚拟允许调用某个函数，对于这个函数，仅仅知道它的接口，而不知道具体的对象类型。

但是要建立一个对象，必须拥有完全的信息。需要知道要建立的对象的具体类型。因此，对构造函数的调用不可能是虚拟的。





5.1.5 虚析构造函数

- ◆ 通过下例说明虚析构造函数的必要性：

例5-9：（例5-9.txt）

基类A的构造函数动态分配5个字节，其析构函数负责释放这块内存。

派生类Z的构造函数动态分配5 000个字节，其析构函数负责释放这块内存。

```
#include <iostream>
```

```
using namespace std;
```

```
class A {
```

```
    public:
```

```
        A() { cout<<endl<<"A() firing"<<endl; p = new char[5]; };
```

```
        ~A() { cout<<"~A() firing"<<endl; delete [ ] p; }
```

```
    private:
```

```
        char* p;
```

```
};
```





5.1.5 虚析构函数

```
class Z : public A
{ public:
    Z() {
        cout<<"Z() firing"<<endl;
        q = new char[5000];
    }
    ~Z() {
        cout<<"~Z() firing"<<endl;
        delete [] q;
    }
private:
    char* q;
};
```





5.1.5 虚析构函数

```
void f()
{ A* ptr;
  ptr = new Z();
  delete ptr;
}
int main()
{ for (unsigned I = 0; i<3; i++) f();
  return 0;
}
```





5.1.5 虚析构造函数

- ◆ 因析构造函数不是虚成员函数，所以编译器实施静态绑定。
根据ptr的数据类型A*来决定调用哪一个析构造函数，因此，仅调用了~A()，而没有调用~Z()，这样在Z()中分配的5000字节就不会被释放。
这就导致每调用f一次，就会丢失5000字节的内存。
- ◆ 解决上述问题方法：
定义基类的析构造函数~A()为虚成员函数，这样，其派生类的析构造函数也为虚成员函数。即使不明确地声明~Z()为virtual，~Z()仍为虚成员函数。
当通过ptr来删除其所指的对象时，编译器进行的是运行期绑定。因为ptr指向一个Z类型的对象，所以~Z()被调用，随后自下向上调用~A()。代码如下：





5.1.5 虚析构函数

```
class A {  
public:  
    A() { cout<<endl<<"A() firing"<<endl;  
        *p = new char[5]; };  
    virtual ~A() { cout<<"~A() firing"<<endl;  
        delete [ ] p;  }  
private:  
    char* p;  
};
```





5.1.5 虚析构函数

- ◆ 通常来说，如果基类有一个指向动态分配内存的数据成员，并定义了负责释放这块内存的析构函数，就应该将这个析构函数声明为虚成员函数，这样做可以保证在以后添加该类的派生类时发挥多态性的作用。
- ◆ 在MFC这样的商业库中，为了防止发生内存遗漏问题，库中的析构函数通常都是虚成员函数。





虚析构函数举例

```
#include <iostream>
#include <string>
#include <conio>
class TPerson
{ private:
    char *name;
public:
    TPerson(const char *s)
    { name=new char[strlen(s)+1];
      strcpy(name,s);
    }
    virtual void print() { cout<<name<<endl; }
    virtual ~TPerson() { delete [ ] name; }
};
```





虚析构造函数举例

```
class TStudent:public TPerson
{ private:
    char *major;
public:
    TStudent(const char *n, const char *m):TPerson(n)
    {   major=new char[strlen(m)+1];
        strcpy(major,m);    }
    virtual void print(){TPerson::print(); cout<<major<<endl; }
    ~TStudent() { delete[ ] major; }
};


void main()
{   TPerson *p=new TStudent("Zhang", "Computer");
    p->print();
    delete p; //TStudent::~~TStudent被调用
    getch();
}
```

VTABLE

XXX1

Tperson::print()
Tperson::~~Tperson

TPerson



```
* name;  
TPerson(const char *s)  
V void print( )  
V ~TPerson( )
```

TStudent

```
* name;  
V void print( )
```

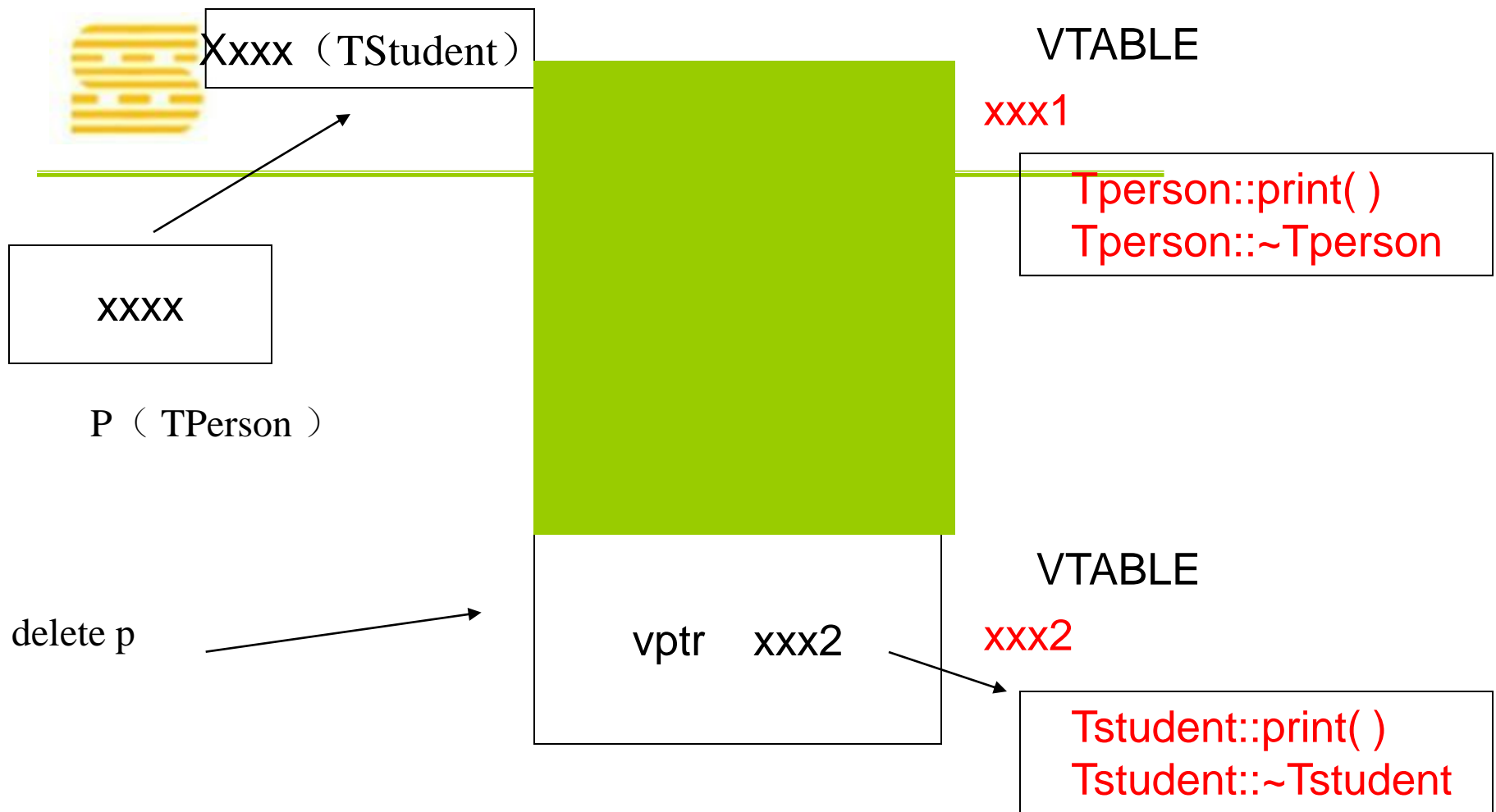
```
* major;  
TStudent(s, m)  
V void print( )  
V ~TStudent ( )
```

VTABLE

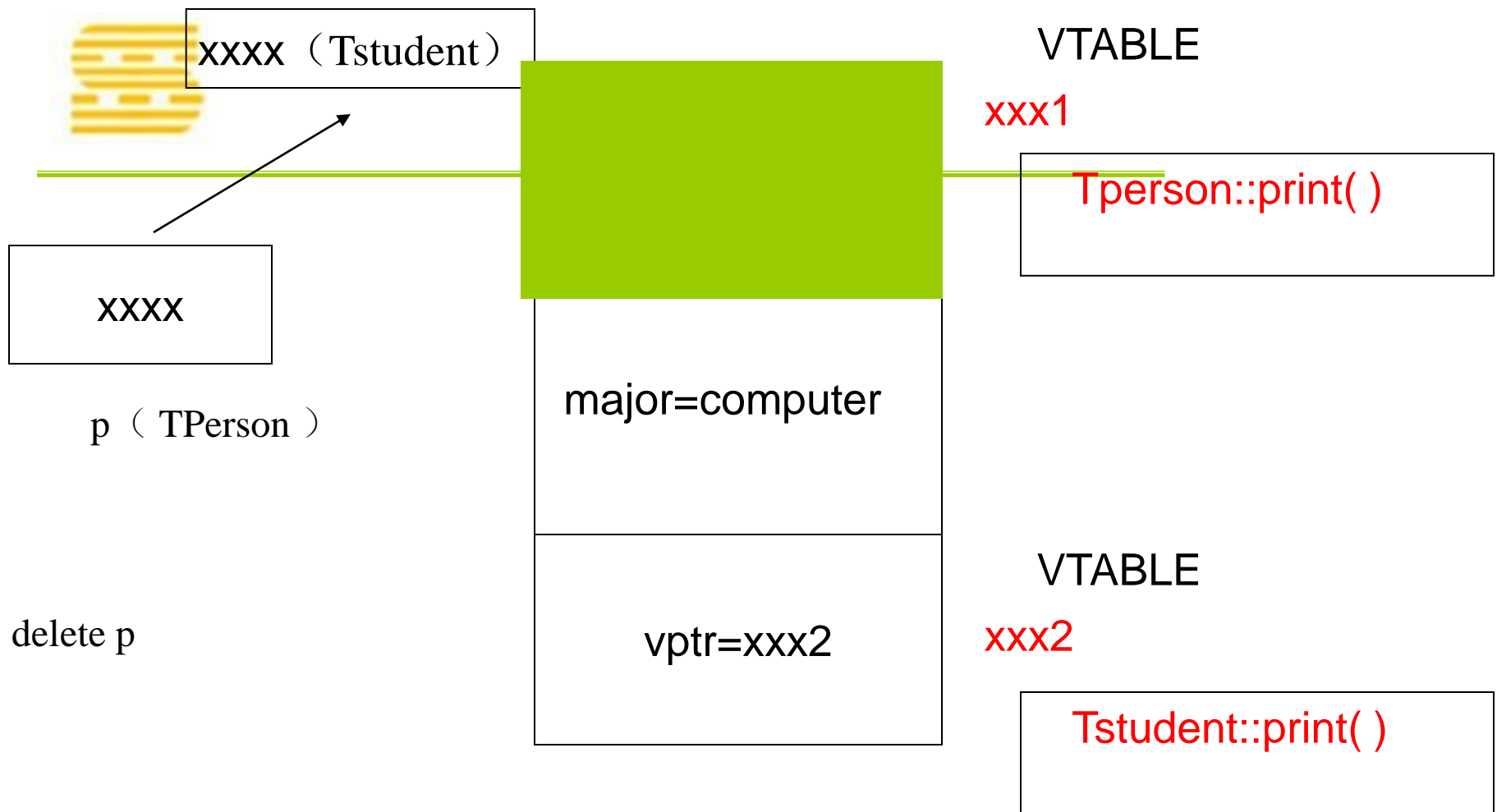
XXX2

Tstudent::print()
Tstudent::~~Tstudent





**释放p所指内存时，由于TStudent是虚析构，
先执行Tstudent析构，再执行Tperson析构，
不会造成内存泄漏**



若不是虚析构，释放p所指内存时，由于p是Tperson类型，执行Tperson析构，不会执行Tstudent析构，造成内存泄漏



5.1.6 对象成员函数和类成员函数

- ◆ 只有非静态成员函数（对象成员函数），才可以是虚成员函数。
- ◆ 静态成员函数（类成员函数）不可以是虚成员函数。

例5-11

```
class C{  
    public:  
        static virtual void f( );//Error  
        static void g( );  
        virtual void h( );  
};
```





练习

- ◆ 下列函数 中,不能说明为虚函数的是(c)。
 - A.私有成员函数
 - B.公有成员函数
 - C.构造函数
 - D.析构函数
- ◆ 下列关于虚函数的说明中, 正确的是 (B) 。
 - A. 从虚基类继承的函数都是虚函数
 - B. 虚函数不得是静态成员函数
 - C. 只能通过指针或引用调用虚函数
 - D. 抽象类中的成员函数都是虚函数





练习

◆ 写出下列程序运行结果

```
class A{ public: virtual ~A( ){ cout<<"A::~~A( ) called "<<endl; } };
class B:public A{ char *buf;
    public: B(int i) { buf=new char[i]; }
           virtual ~B( ){ delete []buf;
                           cout<<"B::~~B( ) called"<<endl; } };

void fun(A *a) { delete a; }

int main( )
{
    A *a=new B(10);
    fun(a);
}
```

输出 : B::~~B() called
A::~~A() called





5.2 示例程序：改进的影片跟踪管理

- ◆ 将影片跟踪管理的例子作如下修改：
 - 提供一个多态的成员函数input，可以从某个输入文件中读入有关Films、DirectorCuts和ForeignFilms的记录，输入文件中的每条记录都可映射到某个Film类层次中的对象，假设输入文件的格式是正确的。
 - 依据输入文件中的记录动态地创建Film类层次对象。
 - 将Film类层次中的input函数设计为虚函数，使其具有多态性。动态创建的对象通过input函数可从输入流中正确地读入数据。
 - 将Film类层次中的output函数设计为虚函数，使其具有多态性。动态创建的对象通过output函数可将信息正确地输出到标准输出流。

- ◆ 输入文件的格式如下：





输入文件的格式

Film

Mean Streets

Martin Scorsese

168

4

Film

The Best Years of Our Lives

William Wyler

172

4

ForeignFilm

Diva

Jean-Jacques Beineix

123

3

French





测试程序

```
#include "films.h" // class declarations, etc.
int main() {
    const unsigned n = 5;
    Film* films[ n ];
    // attempt to read input file and create objects
    if ( !Film::read_input( "films.dat", films, n ) ) {
        cerr << "Unable to read file films.dat: exiting." << endl;
        exit( EXIT_FAILURE );
    }
    // output to the standard output
    for ( unsigned i = 0; i < n; i++ )
        films[ i ]->output(); // polymorphic output
    return 0;
}
```



测试程序输出

Title: Mean Streets

Director: Martin Scorsese

Time: 168 mins

Quality: ****

Title: The Best Years of Our Lives

Director: William Wyler

Time: 172 mins

Quality: ****

Title: Diva

Director: Jean-Jacques Beineix

Time: 123 mins

Quality: ***

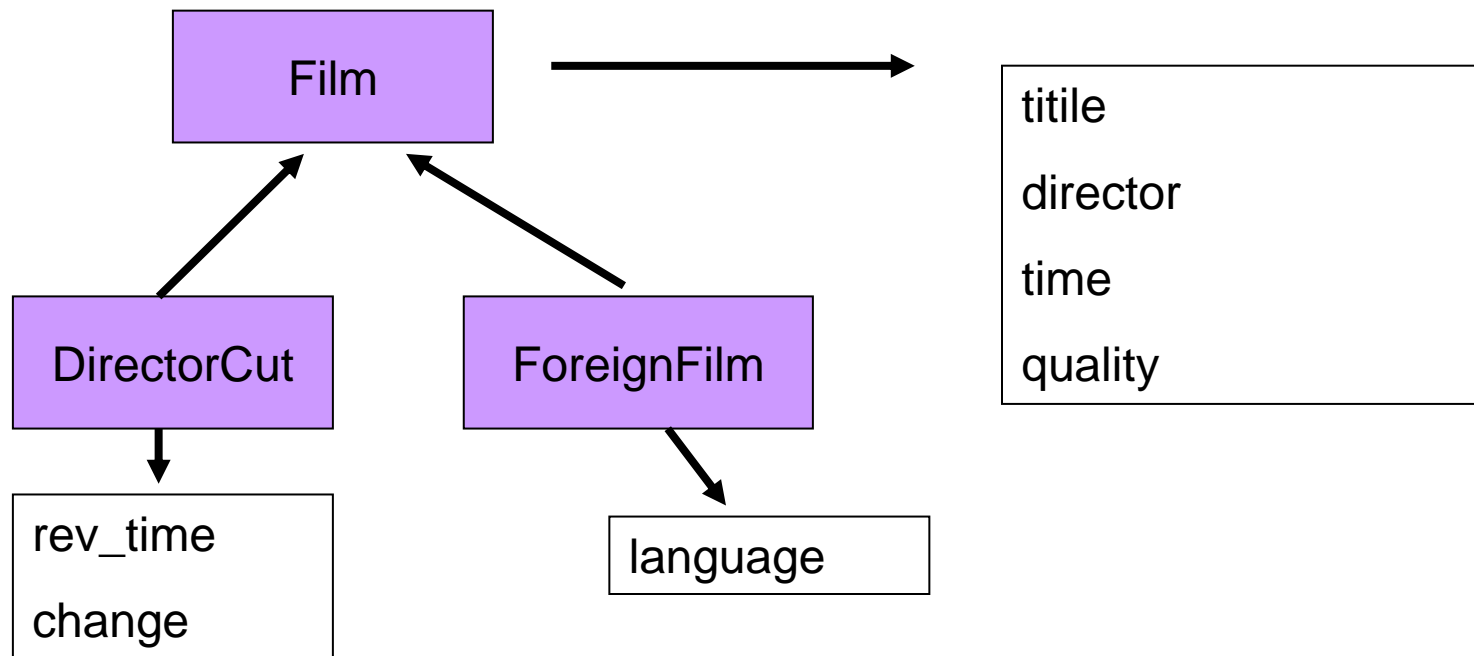
Language: French



示例程序：改进的影片跟踪管理

◆ 原程序回顾：

Film为基类、ForeignFilm和DirectorCut为派生类





```
class Film {
```

```
public:
```

```
void store_title( const string& t ) { title = t; }
```

```
void store_title( const char* t ) { title = t; }
```

```
void store_director( const string& d ) { director = d; }
```

```
void store_director( const char* d ) { director = d; }
```

```
void store_time( int t ) { time = t; }
```

```
void store_quality( int q ) { quality = q; }
```

```
void output() const;
```

```
private:
```

```
string title;
```


```
string director;
```

```
int time; // in minutes
```

```
int quality; // 0 (bad) to 4 (tops)
```

```
};
```





```
void Film::output() const {  
    cout << "Title: " << title << '\n';  
    cout << "Director: " << director << '\n';  


---

    cout << "Time: " << time << " mins" << '\n';  
    cout << "Quality: ";  
    for ( int i = 0; i < quality; i++ )    cout << '*';  
    cout << '\n';  
}
```




```
class DirectorCut : public Film {
```

```
public:
```

```
void store_rev_time( int t) { rev_time = t; }
```

```
void store_changes( const string& s) { changes = s; }
```

```
void store_changes( const char* s) { changes = s; }
```

```
void output() const;
```

```
private:
```

```
int rev_time;
```

```
string changes;
```

```
};
```

```
void DirectorCut::output() const {
```

```
    Film::output();
```

```
    cout << "Revised time: " << rev_time << " mins\n";
```

```
    cout << "Changes: " << changes << '\n';
```

```
}
```



```
class ForeignFilm : public Film {
```

```
public:
```

```
void store_language( const string& l ) { language = l; }
```

```
void store_language( const char* l ) { language = l; }
```

```
void output() const;
```

```
private:
```

```
string language;
```

```
};
```

```
void ForeignFilm::output() const {
```

```
Film::output();
```

```
cout << "Language: " << language << '\n';
```

```
}
```





示例程序：改进的影片跟踪管理

◆ 对FILM的改进

- 加了默认构造函数，使得在创建对象时确保所有的数据成员都被初始化。
- 增加了虚成员函数input
- 增加了虚成员函数output
- 增加了静态成员函数read-input



#include <iostream>

#include <fstream>

#include <string>

#include <cctype>

using namespace std;

class Film {

public:

Film() {

store_title(); //默认参数 空字符串

store_director();

store_time(); //默认参数 0

store_quality();

}

void store_title(const string& t) { title = t; }

void store_title(const char* t = "") { title = t; }





示例程序：改进的影片跟踪管理

```
void store_director( const string& d ) { director = d; }
void store_director( const char* d = "" ) { director = d; }
void store_time( int t = 0 ) { time = t; }
void store_quality( int q = 0 ) { quality = q; }
virtual void output();
virtual void input( ifstream& );
static bool read_input( const char* fname, Film*[ ], int );
private:
    string title;
    string director;
    int time; // in minutes
    int quality; // 0 (bad) to 4 (tops)
};
```



// Reads title, director, time, and quality.

```
void Film::input( ifstream& fin ) {  
    string inbuff;  
    getline( fin, inbuff ); store_title( inbuff );  
    getline( fin, inbuff ); store_director( inbuff );  
    getline( fin, inbuff ); store_time( atoi( inbuff.c_str() ) );  
    getline( fin, inbuff ); store_quality( atoi( inbuff.c_str() ) );  
}
```

// Writes title, director, time, and quality.

```
void Film::output() {  
    cout << "Title: " << title << endl;  
    cout << "Director: " << director << endl;  
    cout << "Time: " << time << " mins" << endl;  
    cout << "Quality: ";  
    for ( int i = 0; i < quality; i++ ) cout << '*';  
    cout << endl;  
}
```



示例程序：改进的影片跟踪管理

◆ 对DirectorCut的改进

- 加了默认构造函数，使得在创建对象时确保所有的数据成员都被初始化。
- 增加了虚成员函数input
- 增加了虚成员函数output



class DirectorCut : public Film {

public:

DirectorCut() {

store_rev_time();

store_changes();

}

void store_rev_time(int t = 0) { rev_time = t; }

void store_changes(const string& c) { changes = c; }

void store_changes(const char* c = "") { changes = c; }

virtual void output();

virtual void input(ifstream&);

private:

int rev_time;

string changes;

};





// Reads revised time and changes.

```
void DirectorCut::input( ifstream& fin ) {  
    Film::input( fin );  
    string inbuff;  
    getline( fin, inbuff );  store_rev_time( atoi( inbuff.c_str() ) );  
    getline( fin, inbuff );  store_changes( inbuff );  
}
```

// Writes revised time and changes.

```
void DirectorCut::output() {  
    Film::output();  
    cout << "Revised time: " << rev_time << endl;  
    cout << "Changes: " << changes << endl;  
}
```





示例程序：改进的影片跟踪管理

◆ 对ForeignFilm的改进

- 加了默认构造函数，使得在创建对象时确保所有的数据成员都被初始化。
- 增加了虚成员函数input
- 增加了虚成员函数output



```
class ForeignFilm : public Film {
```

```
public:
```

```
    ForeignFilm() { store_language(); }
```

```
    void store_language( const string& l ) { language = l; }
```

```
    void store_language( const char* l = "" ) { language = l; }
```

```
    virtual void output();
```

```
    virtual void input( ifstream& );
```

```
private:
```

```
    string language;
```

```
};
```

```
// Reads language.
```

```
void ForeignFilm::input( ifstream& fin ) {
```

```
    Film::input( fin );
```

```
    string inbuff;
```

```
    getline( fin, inbuff ); store_language( inbuff );
```

```
}
```





示例程序：改进的影片跟踪管理

```
// Writes language.  
void ForeignFilm::output() {  
    Film::output();  
    cout << "Language: " << language << endl;  
}
```





示例程序：改进的影片跟踪管理

◆ 基类中的静态成员函数read-input

功能：从输入文件中读入记录，记录按Film、ForeignFilm或DirectorCut来分类。读入像Film这样的记录组标志之后，动态创建相应的对象,并将指针储存在数组中，然后调用这个对象的input成员函数读入余下的记录，每个类的input成员函数设计为读入与这种类型相关的记录。

```
Film  
The Best Years of Our Lives  
William Wyler  
172  
4  
ForeignFilm  
Diva  
Jean-Jacques Beineix  
123  
3  
French
```

// class method: Film::read_input

// Reads data from an input file, dynamically creating the

// appropriate Film object for each record group. For instance,

// a ForeignFilm object is dynamically created if the data

// represent a foreign film rather than a regular film or a

// director's cut. Pointers to dynamically created objects are

// stored in the array films of size n. Returns true to signal success

// and false to signal failure.

bool Film::read_input(const char* file, Film* films[], int n) {


string inbuff;

ifstream fin(file);

if (!fin) // opened successfully?

return false; // if not, return false

int next = 0;



```
/* Read until end-of-file. Records fall into groups. 1st record in
each group is a string that represents a Film type:"Film",
"ForeignFilm", "DirectorCut", etc. After reading type record,
dynamically create an object of the type (e.g., a ForeignFilm
object), place it in the array films, and invoke its input method.*/
while ( getline( fin, inbuff ) && next < n ) {
    if ( inbuff == "Film" ) films[ next ] = new Film(); // regular film
    else if ( inbuff == "ForeignFilm" )
        films[ next ] = new ForeignFilm(); // foreign film
    else if ( inbuff == "DirectorCut" )
        films[ next ] = new DirectorCut(); // director's cut
    else /*** error condition: unrecognized film type
        continue;
    films[ next++ ]->input( fin ); // polymorphic method
}
fin.close();
return true;
}
```



输入文件的格式如下(回顾)

Film

Mean Streets

Martin Scorsese

168

4

Film

The Best Years of Our Lives

William Wyler

172

4

ForeignFilm

Diva

Jean-Jacques Beineix

123

3

French





测试程序(回顾)

```
#include "films.h" // class declarations, etc.
int main() {
    const unsigned n = 5;
    Film* films[ n ];
    // attempt to read input file and create objects
    if ( !Film::read_input( "films.dat", films, n ) ) {
        cerr << "Unable to read file films.dat: exiting." << endl;
        exit( EXIT_FAILURE );
    }
    // output to the standard output
    for ( unsigned i = 0; i < n; i++ )
        films[ i ]->output(); // polymorphic output
    return 0;
}
```





测试程序输出(回顾)

Title: Mean Streets

Director: Martin Scorsese

Time: 168 mins

Quality: ****

Title: The Best Years of Our Lives

Director: William Wyler

Time: 172 mins

Quality: ****

Title: Diva

Director: Jean-Jacques Beineix

Time: 123 mins

Quality: ***

Language: French



示例程序：改进的影片跟踪管理

- ◆ 本例中实现多态的三个条件
 - 存在一个继承体系结构：Film，ForeignFilm和DirectorCut
 - 继承体系结构中的一些类必须具有同名的virtual成员函数：input和output
 - 至少有一个基类类型的指针或基类类型的引用：变量film是基类Film的指针数组



5.3 重载、覆盖和遮蔽

- ◆ 5.3.1 重载（overloading）
- ◆ 在同一个命名空间中，两个函数同名但参数个数或类型不同，即函数名相同，函数签名不同。例如：

```
class Base {  
    public:  
    virtual void service() {}  
    virtual void service(int x) {}  
};
```

- ◆ 这两个函数既可以是虚函数也可以是一般函数
- ◆ 编译期绑定

重载函数和虚函数的区别在于前者是编译期绑定的，后者是运行期绑定。



重载

◆ 例5-12

```
Class C{
public:
    C() {...}
    C(int x) {...}    };
void f(double d) {...}
void f(char c) {...}
int main()
{   C c1;
    C c2(26);
    f(3.14);
    f('z');
}
```





5.3.2 覆盖（overriding）

◆ 覆盖：函数名相同，函数签名也相同，且运行期绑定

基类B中有一个虚成员函数m，派生类D也有一个具有相同函数签名的成员函数m，称D::m 覆盖 B::m。

如果成员函数不是虚函数，那么任何调用均为编译期绑定（遮蔽）

。





5.3.2 覆盖（overriding）

◆ 例：覆盖

```
class B{
public:
    virtual void m(){cout<<"B::m()"<<endl;}
};
class D:public B{
public:
    void m(){cout<<"D::m()"<<endl;}
};
```

覆盖

运行期绑定

```
int main(){
    B* p; D d;
    p=new D;
    p->m();
    d.B::m();
    d.m();
    return 0;
}
```

编译期绑定





遮蔽（隐藏，hiding）

- ◆ 遮蔽：函数名相同，函数签名可相同可不相同，且编译期绑定

基类**B**中有一个非虚成员函数**m**，派生类**D**也有一个具有相同函数名的成员函数**m**，称**D::m** 遮蔽 **B::m**。

- ◆ 例：函数签名相同的遮蔽





基类B中有一个非虚成员函数m，派生类D也有一个具有相同函数名的成员函数m，称D::m 遮蔽 B::m。

```
class B{
```

```
public:
```

```
void m(){cout<<"B::m()"<<endl;}
```

```
};
```

```
class D:public B{
```

```
public:
```

```
void m(){cout<<"D::m()"<<endl;}
```

```
};
```

```
int main(){
```

```
B* p; D d;
```

```
p=new D;
```

```
p->m();
```

```
d.B::m();
```

```
d.m();
```

```
return 0;}
```

遮蔽

编译期绑定





遮蔽（隐藏，hiding）

例：函数签名不相同的遮蔽

```
class B{
public:
    void m(int x){cout<<"B::m()"<<endl;}
};
```

遮蔽

```
class D:public B{
public:
    void m(){cout<<"D::m()"<<endl;}
};
```

```
int main(){
    B* p; D d;
    p=new D;
    p->m();
    p->m(3);
    d.m(3);
    d.B::m(3);
    d.m();
    return 0;
}
```

编译期绑定





例5-15

```
class B {
public:
    void m( int x ) { cout << x << endl; }
};

class D : public B {
public:
    void m() { cout << "Hi" << endl; }
};

int main() {
    D d1;
    d1.m();          // OK: D::m expects no arguments
    d1.m( 26 );      //***** ERROR: D::m expects no arguments
    return 0;
}
```





遮蔽（隐藏，hiding）

- ◆ 函数和非虚函数一样都可能产生名字遮蔽，实际情况是一旦派生类的虚函数不能覆盖基类的虚函数，就会产生虚函数的遮蔽。
- ◆ 例5-16：虚函数被遮蔽的例子
此例中，二个同名函数m毫不相关。
要产生多态效果，必须为二个类定义具有相同函数签名的虚函数。





遮蔽（隐藏，hiding）

```
class B {
public:
    virtual void m( int x ) { cout << x << endl; }
};
class D : public B {
public:
    virtual void m() { cout << "Hi" << endl; }
};
int main() {
    D d1;
    d1.m();
    d1.m( 26 ); //***** ERROR: D's m takes no arguments
    return 0;
}
```





遮蔽（隐藏，hiding）

例：虚函数被遮蔽

```
class B{  
public:
```

```
    virtual void m(int x){cout<<"B::m()"<<endl;}  
};
```

遮蔽

```
class D:public B{  
public:
```

```
    virtual void m(){cout<<"D::m()"<<endl;}  
};
```

```
int main(){  
    B* p; D d;  
    p=new D;  
    p->m();  
    p->m(3);  
    d.m(3);  
    d.B::m(3);  
    d.m();  
    return 0;  
}
```

编译期绑定





5.3.4 名字共享

- ◆ 需要共享函数名的几种情况：
 - 重载函数名的顶层函数
 - 重载构造函数
 - 非构造函数但是同一个类中名字相同的成员函数
 - 继承层次中的同名函数(特别是虚函数，多态)。





小结

- ◆ 多态分两种：
 - 静态多态：静态绑定，编译器决定调用哪个函数
 - 动态多态：动态绑定，运行时通过查虚函数表决定
- ◆ 多态与overload\override\hiding的关系
 - **重载**：静态多态，发生于同一命名空间；
 - **覆盖和遮蔽**：发生在具有继承关系的不同类的成员函数之间
 - 一般函数的hiding：静态多态
 - 虚函数hiding
 - **override**：动态多态（同时满足3个条件）
 - 其他：静态多态





小结

- ◆ 静态多态和动态多态的目的都是希望让程序员可以用高级、抽象的方法使用不同的代码。

例如: $x+y$, 让编译器自己根据 x 与 y 的类型来决定该如何完成 $x+y$ 的动作;

再如: `Ptr->service()`, 根据虚函数表在程序执行时自动选择正确的函数调用。

这样程序设计的弹性大, 扩充能力强, 代码重用性强。





练习

- ◆ 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)。
 - A.相同 B.不同 C.相容 D.部分相同
- ◆ 下列函数 中,不能说明为虚函数的是(c)。
 - A.私有成员函数 B.公有成员函数
 - C.构造函数 D.析构函数



5.4 抽象基类

◆ 为什么用抽象基类

- 有些应用，派生类必须覆盖父类的虚函数；
- 需指定派生类的公共接口；
 公共接口：一个成员函数的集合，任何支持该接口的类必须定义该集合中的所有函数。
- 无法定义基类中虚函数的具体操作，虚函数的具体操作完全取决于其不同的派生类。

这时，可将基类中的虚函数定义为纯虚函数，此时的类就是抽象基类。





5.4 抽象基类

◆ 5.4.1 抽象基类和纯虚成员函数

如果一个类中至少有一个纯虚函数，那么这个类被成为抽象类（**abstract class**）。

◆ 纯虚函数

纯虚函数：在虚成员函数声明的结尾加上=0。

virtual 函数类型 函数名称（参数表）=0;

如：

```
class B    //抽象基类B声明
{ public:  //外部接口
    virtual void display( )=0;  //纯虚函数成员
};
```





5.4.1 抽象基类和纯虚成员函数

- ◆ 由纯虚函数的定义格式可以看出如下几点：
 - 1) 由于纯虚函数无函数体，所以在派生类中没有重新定义纯虚函数之前，是不能调用这种函数的。
 - 2) 将函数名赋值为0的含义是，将指向函数体的指针值赋初值0。
- ◆ 抽象基类不能被实例化，而且必须至少有一个纯虚成员函数。

例5-18

```
class ABC {  
    public:  
        virtual void open()=0;  
};  
ABC obj; //error;
```





5.4.1 抽象基类和纯虚成员函数

注：

- 1) 虽不能声明抽象类的实例，也不能用抽象类作为参数类型，函数返回类型或显示转换类型，但可以声明抽象类的指针或引用，当用这种基类指针指向其派生类的对象时，必须在派生类中重载纯虚函数，否则会产生程序的运行错误。
- 2) 从抽象类派生出的新类，必须重新定义其父类的每一个纯虚函数，或者把这些函数继续声明为纯虚函数。

例5-9

```
class ABC{ public: virtual void open( )=0;};
class X:public ABC{
    public :virtual void open( ) { /* 函数体...*/ } };
class Y:public ABC{ };
ABC a1; //Error, abc is abstract
X x1;    //ok, X overrides open() and is not abstract
Y y1; //Error,Y is abstract: open() not defined
Y *p;
```





5.4 抽象基类

◆ 5.4.2 定义纯虚成员函数的限制

只有虚函数才可以成为纯虚成员函数，非虚函数或顶层函数都不能声明为纯虚成员函数。

```
void f( )=0; //error  
class C{  
public:  
    void open( )=0; //error  
};
```

◆ 5.4.3 抽象基类的使用

抽象基类定义公共接口，实现共享。

抽象基类通常只有**public** 成员函数。





5.4 抽象基类

例：求平面图形的面积 (5.4求平面图形的面积.txt)

```
#include <iostream>
using namespace std;
class Shape
{ protected:
    int a,b;
public:
    void set(int i,int j=0) { a=i; b=j;}
    virtual void print_area(void)=0; //纯虚函数
};
```





5.4 抽象基类

```
class triangle:public Shape
{   public:
    virtual void print_area(void)
    {   cout<<"\n triangle with height ";
        cout<<a<<" and base "<<b;
        cout<<" has an area of ";
        cout<<a*b/2;
    }
};
```





5.4 抽象基类

```
class circle:public Shape
{ public:
    virtual void print_area(void)
    { cout<<"\n Circle with radius "<<a;
      cout<<" has an area of ";
      cout<<3.14*a*a;    }    };

class rectangle:public Shape
{ public:
    virtual void print_area(void)
    { cout<<"\n rectangle with legnth "<<a;
      cout<<" and width "<<b<<" has an area of ";
      cout<<a*b<<"\n";    }

};
```





5.4 抽象基类

```
int main(void)
{ Shape *p; triangle t; circle c; rectangle r;
  p=&t; p->set(10,16); p->print_area();
  p=&c; p->set(100); p->print_area();
  p=&r; p->set(10,160); p->print_area();
  return 0; }
```

运行结果：

triangle with height 10 and base 16 has an area of 80

Circle with radius 100 has an area of 31400

rectangle with length 10 and width 160 has an area of 1600

若在主函数中增加说明：**Shape s;**

则因为抽象类**Shape**不能产生对象，编译时将给出错误信息。





shape

```
# a,b  
+ set(...)  
+ print_area()=0
```

triangle

```
# a,b  
+ set(...)  
+ print_area()=0  
  
+ print_area()
```

circle

```
# a,b  
+ set(...)  
+ print_area()=0  
  
+ print_area()
```

rectangle

```
# a,b  
+ set(...)  
+ print_area()=0  
  
+ print_area()
```





练习

- ◆ (C) 是一个在基类中说明的虚函数，它在该基类中没有定义，但要求任何派生类都必须定义自己的版本。
 - A. 虚析构造函数
 - B. 虚构造函数
 - C. 纯虚函数
 - D. 静态成员函数
- ◆ 以下基类中的成员函数，哪个表示纯虚函数 (C) 。
 - A. `virtual void vf(int);`
 - B. `void vf(int)=0;`
 - C. `virtual void vf()=0;`
 - D. `virtual void vf(int){ }`
- ◆ 下列描述中， (D) 是抽象类的特性。
 - A. 可以说明虚函数
 - B. 可以进行构造函数重载
 - C. 可以定义友元函数
 - D. 不能定义其对象



5.5 运行期类型识别

- ◆ C++支持运行期类型识别（Run-Time Type Identification, RTTI），运行期类型识别提供如下功能：
 - 在运行期对类型转换操作进行检查。（dynamic_cast）
 - 在运行期确定对象的类型(typeid)。
 - 扩展 C++ 提供的RTTI。





5.5.1 dynamic_cast操作符

- ◆ 在C++中，编译期合法的类型转换操作可能会在运行期引发错误，当转型操作涉及到对象指针或引用时，更易发生错误。
- ◆ 使用dynamic_cast操作符可用来在运行期对可疑的转型操作进行测试。





5.5.1 dynamic_cast操作符

- ◆ 例5-23：一个基类指针不经过明确的转型操作，就能指向基类对象或派生类对象；反过来，一个派生类指针指向基类对象是一种不明智的做法。

```
class B { };  
class D:public B { };  
int main( ) {  
    D* p;  
    p = new B; //error  
    p = static_cast<D*> (new B);  
}
```

使用static_cast进行转型操作是合法的，但可能会造成难以跟踪的运行期错误。





5.5.1 dynamic_cast操作符

例5-24 class B (例5-24.txt)

```
{ public:
    virtual void f() { cout<<"f() "<<endl; }
};
class D:public B{
    public:
        virtual void m(){cout<<"m() "<<endl;}
};
int main(){
    D* p = static_cast<D*> (new B);
    p->m();
    return 0;
}
```

编译通过，运行出错。

P实际指向一个B的对象，而B没有成员函数m,这种转型不安全.

(m非虚函数可正常运行)

若m()是非虚函数，则正常运行，输出为m()





5.5.1 dynamic_cast操作符

C++提供的dynamic_cast操作符可以在运行期检测某个转型动作是否安全。dynamic_cast和static_cast有同样的语法，不过dynamic_cast仅对多态类型有效。

例5-25

```
class C { //C类中无虚函数      };  
class T {      };  
int main( )  
{ T* p=dynamic_cast<T*>(new C); //error, 仅对有虚函数类型有效  
  return 0; }
```

注意：dynamic_cast操作正确的前提是转型的源类型必须有虚函数，
但与转型的目标类型是否有虚函数无关。

在<>中指定的dynamic_cast的目的类型必须是一个指针或引用。





5.5.1 dynamic_cast操作符

- ◆ 例5-27： 看一个正确的用法（例5-27.txt）

```
class B{
    public:
        virtual void f() {cout<<"f()"<<endl;}    };
class D:public B{
    public:
        void m() {cout<<"m()"<<endl;}    };
int main(){
    D *p=dynamic_cast<D*> (new B);
    if (p)  p->m();
    else
        cout<<"Error\n";
    return 0; }
```

能够判断转型是否安全，如果安全，则返回**B**对象的地址，否则返回NULL。本例返回NULL。





5.5.1 dynamic_cast操作符

◆ 例5-28: 用dynamic_cast判断运行期参数类型

```
#include <iostream>
#include <string>
#include <typeinfo>
using namespace std;
class Book
{ public: Book(string t) {title = t;}
    virtual void printTitle() const
        { cout << "Title: " << title << endl; }
    private: Book();
        string title;
};
```





5.5.1 dynamic_cast操作符

```
class Textbook : public Book
{ public: Textbook(string t,int l) : Book(t),level(l) {}
    void printTitle() const {cout<<"Textbook ";
        Book::printTitle(); }
        void printLevel() const
            {cout<<"Book level: "<<level<< endl; }
private: Textbook();
        int level;
};
```





5.5.1 dynamic_cast操作符

```
class PulpFiction:public Book
{ public:PulpFiction(string t):Book(t) {}
    void printTitle()
        const {cout<<"PulpFiction " ; Book::printTitle();}
    private:PulpFiction();
};

void printBookInfo(Book* bookPtr)
{ bookPtr->printTitle();
    Textbook* ptr=dynamic_cast<Textbook*>(bookPtr);
    if(ptr) ptr->printLevel();
}
```





5.5.1 dynamic_cast操作符

```
int main()
{ Book * ptr;
  int level, ans;
  string title;
  cout<<"Book's titles (no white space) ";
  cin >> title;
  do { cout << "1 == Textbook, 2 == PulpFiction " << endl;
        cin >> ans; } while( ans < 1 || ans > 2);
  if (1 == ans) { cout<<"Level "; cin >> level;
                ptr = new Textbook(title,level);}
  else ptr = new PulpFiction(title);
  printBookInfo(ptr);
  return 0;
}
```





5.5.1 dynamic_cast操作符

- ◆ 上例改进：通过多态来直接输出Textbook的printLevel()

```
class Textbook : public Book {  
public: Textbook(string t,int l) : Book(t),level(l) {}  
    void printTitle() const {  
        cout << "Textbook " ;   Book::printTitle();  
        printLevel();  
    }  
    void printLevel() const { cout << "Book level: " << level << endl; }  
private:  
    Textbook();  
    int level;  
};
```

这样，printBookInfo可简化为：

```
void printBookInfo(Book * bookPtr) {   bookPtr -> printTitle(); }
```





5.5.2 dynamic_cast的规则

在单继承的类层次中，假定基类B具有多态性，而类D是直接或间接从类B派生而来的。通过继承，类D也因此具有多态性，在这种情况下：

- 从派生类D*到基类B*的dynamic_cast可以进行，这称为向上转型（upcast）
- 从基类B*到派生类D*的dynamic_cast不能进行，这称为向下转型（downcast）

假定类A和类Z都有多态性，但它们之间不存在继承关系，这种情况下：

从A*到Z*和从Z*到A*的dynamic_cast不能进行

通常来说，向上转型可以成功，而向下转型不能成功。除了void *之外，无关类型之间的dynamic_cast也不会成功。





5.5.3 dynamic_cast与static_cast小结

- ◆ static_cast可施加于任何类型，不管该类型是否具有多态性
- ◆ dynamic_cast只能施加于具有多态性的类型，转型的目的类型必须是指针或引用。
- ◆ 使用方式：
 dynamic_cast<T*>ptr;
 dynamic_cast<T&>p;
- ◆ dynamic_cast能实施运行期类型检查。





5.5.4 typeid操作符

- ◆ 操作符**typeid**用来确定某个表达式的类型，要使用这个操作符，必须包含头文件**typeinfo**。

- ◆ 用法

`typeid(typename)` 或 `typeid(expression)`

例如:

```
float x ;
```

```
typeid(x)==typeid(float)           //true
```

```
typeid(x)==typeid(double)         //false
```

- ◆ 操作符**typeid**返回一个**type_info**类对象的引用，**type_info**是一个系统类，用来描述类型，这个操作符可施加于类型名（包括类名）或**C++**表达式。





5.5.4 typeid操作符

◆ 例5-30

定义如下指针：

```
Book * bookptr=new Textbook("test",1);
```

使用typeid操作符测试类层次

表达式	值
<code>typeid(bookptr)==typeid(Book*)</code>	true
<code>typeid(*bookptr)==typeid(Book)</code>	false
<code>typeid(bookptr)==typeid(Textbook*)</code>	false
<code>typeid(*bookptr)==typeid(textbook)</code>	true





5.5.4 typeid操作符

```
#include<iostream>
#include<complex>
#include<typeinfo>
using namespace std;
int main()
{ int a=3;
  complex<double> c(1,2); //复数类, STL
  cout<<typeid(c).name()<<" "<<c<<endl;
  cout<<typeid(a).name()<<" "<<a<<endl;
}
```

结果:

class std::complex<double> (1,2)

int 3





5.5.4 typeid操作符

```
#include<iostream>
using namespace std;
class Person{
public:
    void setAge(unsigned n){age=n; }
    unsigned getAge()const {return age;}
private:
    unsigned age;  };
int main()
{ Person p;
  cout<<typeid(p).name()<<endl;
  cout<<typeid(&p).name()<<endl;
}
```

结果:
class Person
class Person *





5.5.4 typeid操作符

◆ vs中

```
#include <iostream>
using namespace std;
class Base {};
class Derived : public Base {};
int main()
{   Derived d;
    Base& b = d;
    cout << typeid(b).name() << endl; }
```

输出: class Base

原因是Base类没定义虚函数，所以对编译器来说，Base类和Derived类之间的转换没意义，因为你不能通过基类指针或引用调用到派生类的函数。所以动态类型检测不会将Base& b = d; 中的b当成Derived类处理了





5.5.4 typeid操作符

◆ 稍加改动

```
#include <iostream>
using namespace std;
class Base { public: virtual ~Base(){} };
class Derived : public Base {};
int main()
{   Derived d;
    Base& b = d;
    cout << typeid(b).name() << endl; }
```

输出： class Derived





练习

```
class A { public: A( ) { }  
          virtual void f( ){cout << "A::f( ) called.\n";} };  
class B: public A { public: B( ) { f( ); }  
                   void g( ) { f( ); }    };  
class C: public B { public: C( ) { }  
                   virtual void f( ){cout << "C::f( ) called.\n";}    };  
void main ( ) { C c; c.f( ); }
```

◆ Ans:

A::f() called. //C c; --> B() --> B::f() --> A::f() 基类虚函数

C::f() called. //C.f() --> C::f()

构造函数调用的虚函数是**自己的类中实现的虚函数**。

如果自己类中没有实现这个虚函数，则**调用基类中的虚函数**而不是任何派生类中实现的虚函数。





练习

```
#include <iostream>
using namespace std;
class A
{ public:
    virtual void print(int x, int y) {cout <<"A:"<< x << ", " << y << endl;}
};
class B:public A
{ public:    // 派生与基类的虚函数类型不同
            出现函数隐藏
    virtual void print(int x, float y) {cout <<"B:"<< x << ", " << y <<endl;}
};
class C:public A
{ public:
    virtual void print(int x, int y) {cout <<"C:"<< x << ", " << y <<endl;}
};
```





练习

```
void show(A &a)
{ a.print(3, 8);
  a.print(6, 5.9);
}
```

```
void main( )
```

```
{  A a;      B b;      C c;
```

```
    show(b); 函数隐藏，没有实现多态，调用a的print方法
```

```
    show(c);
```

```
    a.print(1,2.6);
```

```
    b.print(3,4);      b.print(1,2.5);
```

```
    c.print(1,2);      c.print(3,4.5);
```

```
}
```

•结果:

A:3,8

A:6,5

C:3,8

C:6,5

A:1,2

B:3,4

B:1,2.5

C:1,2

C:3,4





练习

```
#include <iostream>
using namespace std;
class A
{ public:
    virtual void f( ) {cout << "A::f( ) called.\n";}
};
class B:public A
{ public:
    virtual void f( ) {cout << "B::f( ) called.\n";}
};
```





练习

```
int main( )  
{ B b;  
  A &x = b;  
  void (A::*pf)() = &A::f;  
  (x.*pf)();  
}
```

◆ Ans:
 B::f() called.





练习

```
#include <iostream>
using namespace std;
class point
{ public:
    point(int i = 0, int j = 0) {x0 = i; y0 = j;}
    virtual void set( ) = 0;
    virtual void draw( ) = 0;
protected:
    int x0, y0;
};
```



```
class line : public point
```

```
{ public:
```

```
    line(int i=0, int j=0, int m=0, int n=0) : point(i, j)
```

```
    {      x1 = m; y1 = n;      }
```

```
    void set( ) {cout << "line::set( ) called.\n";}
```

```
    void draw( ) {cout << "line::draw( ) called.\n";}
```

```
protected:
```

```
    int x1, y1;
```

```
};
```

```
class ellipse : public point
```

```
{ public:
```

```
    ellipse(int i=0, int j=0, int p=0, int q=0) : point(i,j)
```

```
    {      x2 = p; y2 = q;      }
```

```
    void set( ) {cout << "ellipse::set( ) called.\n";}
```

```
    void draw( ) {cout << "ellipse::draw( ) called.\n";}
```

```
protected:
```

```
    int x2, y2;
```

```
};
```



```
void drawobj(point *p)
```

```
{ p->draw( ); }
```

```
void setobj(point *p)
```

```
{ p->set( ); }
```

```
void main( )
```

```
{ line *lineobj = new line;
```

```
  ellipse *elliobj = new ellipse;
```

```
  drawobj(lineobj);
```

```
  drawobj(elliobj);
```

```
  cout << endl;
```

```
  setobj(lineobj);
```

```
  setobj(elliobj);
```

```
  cout << "\nRedraw the object ...\n";
```

```
  drawobj(lineobj);
```

```
  drawobj(elliobj);
```

```
}
```

ANS:

line::draw() called.

ellipse::draw() called.

line::set() called.

ellipse::set() called.

Redraw the object...

line::draw() called.

ellipse::draw() called.





举例：双向链表

- ◆ 建立一个双向链表，要完成插入一个结点、删除一个结点、查找某一个结点操作，并输出链表上各结点值。设结点只有一个整数。

分析：因链表的插入、删除、查找等操作都是相同类型的，所以可将实现链表操作部分设计成通用的程序。

一个结点的数据结构用两个类来表示。

类**IntObj**的数据成员描述结点信息，成员函数完成两个结点比较，输出结点数据等。

类**Node**的数据成员中，包括要构成双向链表时，指向后一个结点的后向指针**Next**，指向前一个结点的前向指针**Prev**，指向描述结点数据的指针**Info**。

另外定义一个类**List**，把它作为类**Node**的友元，它的成员数据包括指向链表的首指针**Head**，指向链尾的指针**Tail**，成员函数实现链表的各种操作，如插入一个结点，删除一个结点等。由于类**List**是类**Node**的友元，因此它的成员函数可以访问**Node**的所有成员。





举例：双向链表

```
#include <iostream>
using namespace std;
class Object //定义一个抽象类，用于派生描述结点信息的类
{ public:
    Object(){}           //缺省的构造函数
    virtual int IsEqual(Object &)=0; //实现两个结点数据比较的纯虚函数
    virtual void Show()=0; //输出一个结点上数据的纯虚函数
    virtual ~Object() {}   //析构函数定义为虚函数
};
```





举例：双向链表

```
class IntObj: public Object //由抽象类派生出描述结点数据的类
{ public:
    IntObj(int x=0)    { SetData(x); }
    void SetData(int x)    { data=x;}
    int IsEqual(Object &);
    void Show() { cout<<"Data="<<data<<" ";} //重新定义虚函数
private:
    int data;
};

int IntObj::IsEqual(Object &obj) //重写比较两个结点是否相等
{ IntObj & temp = static_cast<IntObj &>(obj); // (IntObj &) obj;
  return (data==temp.data);
}
```





举例：双向链表

```
class Node          //定义结点类
{ public:
    Node() //定义缺省的构造函数
    { Info=0;Pre=0;Next=0;}
    Node (Node &node) //完成拷贝功能的构造函数
    { Info=node.Info;   Pre=node.Pre;   Next=node.Next; }
    void FillInfo(Object *obj) //使Info指向数据域
    { Info=obj; }
    friend class List; //定义List为Node的友元类
private:
    Object *Info;      //指向描述结点的数据域
    Node *Pre, *Next; //用于构成链表前、后指针
};
```





举例：双向链表

```
class List          //实现双向链表操作的类
{ public:
    List() { Head=Tail=0;}    //置空链表
    ~List() { DelList();}     //释放链表所占的存储空间
    void AddNode(Node *);    //在链表尾增加一个结点的成员函数
    Node* DelNode(Node *);   //删除链表中指定结点的成员函数
    Node* LookUp(Object &);  //在链表中查找指定结点的成员函数
    void ShowList ();        //输出整条链表上的数据的成员函数
    void DelList();           //删除整条链表的成员函数
private:
    Node *Head, *Tail; //定义链表首和链表尾指针
};
```





举例：双向链表

```
void List::AddNode(Node * node)
{ if (Head==0)    //链表为空表时
  { Head=Tail=node; //链表首、尾指针指向结点
    node->Next=node->Pre=0;    //该结点前、后指针置为空
  }
  else            // 链表非空
  { Tail->Next=node;    //将结点加入到链表尾
    node->Pre=Tail;
    Tail=node;
    node->Next=0;
  }
}
```





举例：双向链表

```
Node * List::DelNode(Node * node) //返回删除的指定结点
{ if (node==Head)    //删除链表首结点
    if (node==Tail)    //链表只有一个结点
        Head=Tail=0;
    else
        { Head=node->Next; // 删除链表首结点
          Head->Pre=0;    }
    else
        { node->Pre->Next=node->Next;    //删除非首结点
          if (node!=Tail) node->Next->Pre=node->Pre;
          else Tail=node->Pre;    }
    node->Pre=node->Next=0;
    return(node);
}
```





举例：双向链表

```
Node * List::LookUp(Object &obj) //从链表中查找一个结点
{ Node *pn=Head;
  while (pn)
  { if (pn->Info->IsEqual(obj)) return pn;
    pn=pn->Next; }
  return 0;
}

void List::ShowList()           //输出链表上各结点的数据值
{ Node *p=Head;
  while (p)
  { p->Info->Show();
    p=p->Next; }
}
```





举例：双向链表

```
void List::DelList()  //删除整条链表
{
    Node *p, *q;
    p=Head;
    while(p)
    {
        delete p->Info;
        q=p;
        p=p->Next;
        delete q;
    }
}
```





举例：双向链表

```
void main(void)
{
    IntObj *p;
    Node *pn , *pt, node;
    List list;
    for (int i=0;i<5;i++) //建立包括五个结点的双向链表
    {
        p=new IntObj(i+100); //动态建立一个IntOb类的对象
        pn=new Node;          //建立一个新结点
        pn->FillInfo(p);       //填写结点的数据域
        list.AddNode(pn);      //将新结点加到链表尾
    }
    list.ShowList();          //输出链表上各结点
    cout<<endl;
}
```





举例：双向链表

```
IntObj da;  
da.SetData(102); //给要查找的结点置数据值  
pn=list.LookUp(da); //从链表上查找指定的结点  
if (pn) pt=list.DelNode(pn); //若找到，则从链表上删除该结点  
list.ShowList(); //输出已删除结点后的链表  
cout<<endl;  
  
if (pn) list.AddNode(pt); //将结点加入链表尾  
list.ShowList(); //输出已增加一个结点的链表  
cout<<endl;  
}
```





举例：双向链表

执行程序后输出

Data=100 Data=101 Data=102 Data=103 Data=104

Data=100 Data=101 Data=103 Data=104

Data=100 Data=101 Data=103 Data=104 Data=102





举例：双向链表

- ◆ 该例子只提供了双向链表的基本操作：连续五次将新建的结点加到链表尾，显示整个链表上各结点的数据值，从链表上查找一个指定的结点，再删除，并显示删除这个结点后的链表结点数据，再将删除的这个结点加到链表尾并显示新的链表。
- ◆ 例中的类IntObj是由抽象类Object派生而来的，可以根据实际数据结构的需要来定义从基类中继承来的虚函数IsEqual()和Show()的具体实现。在上例中，链表上的结点只有一个整数，所以只要判断两个结点上整数是否相同。由抽象类Object派生出来的不同的派生类均可重新定义这两个纯虚函数，这样就可以实现对不同类的对象使用相同的接口实现不同的操作。

