



上海理工大学

UNIVERSITY OF SHANGHAI FOR SCIENCE AND TECHNOLOGY

第六章 虚函数与多态性

任杰

健康科学与工程学院



主要内容:

- (1) 多态的概念及C++中多态的两种形式: 编译时多态和运行时多态
- (2) 类指针的关系
- (3) 虚函数与运行时多态
- (4) 纯虚函数与抽象类的应用

引入：多态性概述

多态性

- C++中多态性（Polymorphism）是指一个名字多种语义或者界面相同、多种实现；
实现 “一个接口，多种方法”
- 是面向对象程序设计的关键技术之一。
- 利用类的多态性，用户只需要发送一般形式的消息，而接收消息的对象会根据接收到的消息做出相应的动作。
- 多态性机制增加了面向对象软件系统的灵活性，提高了软件的可重用性和可扩充性。

多态的实现方法:

(1) 编译时的多态性 (静态多态)

静态联编 (绑定) 支持

通过函数的重载和运算符的重载、模板来实现的。

(2) 运行时的多态性(动态多态)

动态联编 (绑定) 支持

在程序执行过程中动态地确定，通过类继承关系和虚函数实现。

联编是指一个程序模块、代码之间互相关联的过程。

6.1 静态联编和动态联编

- 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**：在程序编译阶段就能实现的多态性称为**静态多态性**，也称**编译时的多态性**，可以通过**函数重载**和**运算符重载**实现。
- **动态联编**是在程序执行阶段实现的多态性称为**动态多态性**，也称**运行时的多态性**，可以通过**继承、虚函数、基类的指针或引用**等技术来实现。

1、编译时多态（静态联编）

函数的重载、运算符的重载在编译器编译代码时，根据参数列表的匹配关系，就可以确定下来具体调用的函数，也就是在程序执行前就已经确定下来函数调用关系。所以，前面所用的函数重载机制和运算符的重载都是编译时的多态。

普通成员函数重载可表达为以下形式：

(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

(2) 使用“ :: ”加以区分

例如： A :: Show () ;
有别于 B :: Show () ;

(3) 根据类对象加以区分

例如： Aobj . Show () 调用 A :: Show ()
Bobj . Show () 调用 B :: Show ()

静态联编—重载

注意：无论是非类成员和普通函数重载还是类成员函数的重载，都要求形式参数在个数、类型、顺序的一个或多个方面有所区别，否则就不是重载

静态联编—覆盖

上一章讲的类的继承关系中，派生类中出现了与基类中同名的成员函数，若参数列表不同则是函数重载，若函数原型信息完全一样，则称为函数覆盖（Override）。即覆盖是指在派生类中存在和基类函数原型完全一样的同名函数，而这一函数又不是虚函数，这种现象称为“同名覆盖”，而不是重载。

静态联编—覆盖

成员函数覆盖形式：例6-1

```
class Student
{
public:
    void Calculation()
    {
        cout<<"Call Student:: Calculation()\n";
    }
};
class GraduateStudent: public Student
{
public:
    void Calculation()
    {
        cout<<"Call GraduateStudent:: Calculation()"<<endl;
    }
};
int main()
{
    Student s;
    GraduateStudent gs;
    s.Calculation();           //调用Student:: Calculation()
    gs.Calculation();          //调用GraduateStudent::Calculation()
    gs.Student:: Calculation(); //调用Student:: Calculation()
}
```

➤ 派生类中定义与基类同名的成员函数，若特征相同，则是覆盖，若特征不同（参数列表）不同，则是重载。

重载、覆盖的区别

重载函数 (Overload):

- 同一域内的函数，函数名相同、参数表不同。
- 用参数表来区分，不能仅用函数的返回类型区分函数
- 使用时，可用参数表的不同来区分，匹配哪一个函数

覆盖函数(Override):

- 在派生类中，出现了与基类同名的成员（数据成员、函数成员），隐藏了基类的成员(数据、函数)
- 原型完全一样(隐含的**this**指针不同)，但域不同，可用域作用符来区分。
- 使用域作用符 `::` 来访问基类的成员

2.运行的多态（动态联编问题）

程序中的函数调用与函数体代码之间的关联需要推迟到运行阶段才能确定，就是动态联编。

在继承的类体系中，基类定义虚函数，派生类可各自定义虚函数的不同实现版本，当程序中用基类指针或引用调用该虚函数时将引发动态联编，系统会在运行阶段根据基类指针具体指向的对象调用该对象所属类的虚函数版本，从而实现运行时的多态。

所以，动态多态实现需要应用基类指针调用虚函数，这一部分是本章的重点。

问题的引入：对于例6-1中的student类和GraduateStudent类，计算学分的方法是不同的。当基类中定义的CalcTuition函数在派生类中重新定义，并用基类指针进行调用时会发生什么？

//例6.2

```
void Fn(Student &x)  //x是基类Student类型的引用，可以引用学生、研究生，
{
    x.Calculation();  //应根据x的引用对象不同，调用不同类的函数
}
```

```
int main()
{
```

```
    Student s;
    GraduateStudent gs;
```

```
    Fn(s);  //调用学生的学费计算函数
```

```
    Fn(gs);  //gs--->s 类型转换，因为是公有派生。
```

```
    //想调用研究生的学费计算，但现在不能实现，因为不是虚函数
```

```
}
```

➤ 给基类的引用参数传递派生类对象，也只能调用派生类继承的基类成员，而不能调用派生类重写的成员，需要应用虚函数机制。

问题：对于例6-1中的student类和GraduateStudent类，计算学分的方法是不同的。当基类中定义的CalaTuition函数在派生类中重新定义，并用基类指针进行调用时会发生什么？

答：虽然派生类重新定义了适合自己的学费计算函数，当实参时派生类对象gs的地址时，基类指针px指向了派生类对象，但因为系统在编译阶段根据基类指针类型确定了调用基类的CalaTuition()函数，实现静态多态，不能根据传递的对象不同调用不同函数。

解决方案：应用**虚函数**机制来确定运行时对象的类型并调用合适的成员函数，即将基类Calatuition函数定义为虚函数，实现动态联编，就可以根据运行时的指针具体指向的对象类型调用相应的计算学费函数。

6.2 类指针的关系

派生类对象也“是”基类对象，但两者不同。

派生类对象可以当做基类对象，这是因为派生类包含基类的所有成员。

但是基类对象无法被当做成派生类对象，因为派生类可能具有只有派生类才有的成员。

所以，将派生类指针指向基类对象的时候要进行显示的强制转换，否则会使基类对象中的派生类成员成为未定义的。

总结：基类指针和派生类指针指向基类对象和派生类对象的4种方法：

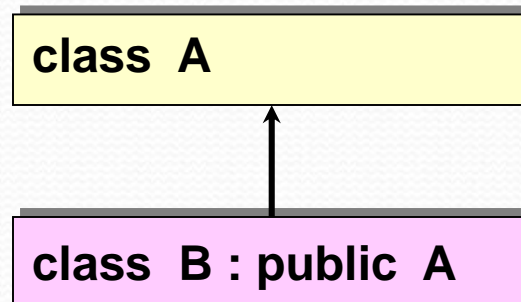
1. 基类指针指向基类对象，简单。只需要通过基类指针简单地调用基类的功能。
2. 派生类指针指向派生类对象，简单。只需要通过派生类指针简单地调用派生类功能。
3. 将基类指针指向派生类对象是安全的，因为派生类对象“是”它的基类的对象。
4. 将派生类指针指向基类对象，会产生编译错误。“是”关系只适用于从派生类到它的直接（或间接）基类，反过来不行。

基类对象并不包含派生类才有的成员，这些成员只能通过派生类指针调用

基类指针和派生类指针的使用

例如:

```
A *p;           // 指向类A 的对象的指针
A A_obj;        // 类A 的对象
B B_obj;        // 类B 的对象
p = &A_obj;     // p 指向类A 的对象
p = &B_obj;     // p 指向类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` 类自定义的成员 (除非用了显式类型转换)


```

class Name
{
    char *name;
public:
    Name(char *nm){ name=nm; }
    void Show_name(){ cout<<name<<endl; }
};

```

```

class Telephone:public Name

```

```

{
    char *telephone;
public:
    Telephone(char *nm,char *tel):Name(nm){telephone=tel;}
    void Show_telephone(){ cout<<telephone<<endl;}
};

```

```

int main()
{
    Name *p1,obj1("Wang");
    Telephone *p2,obj2("Zhang","5681553");
    p1=&obj1;
    p1->Show_name();

```

```

    p1=&obj2;
    p1->Show_name();    //p1->show_telephone: error
    p2=&obj2;
    p2->Show_name();
    p2->Show_telephone();
}

```

class Name

class Telephone : public Name

基类指针

基类指针
指向基类对象

基类指针
调用基类成员函数

```

class Name
{
    char *name;
public:
    Name(char *nm){ name=nm; }
    void Show_name(){ cout<<name<<endl; }
};
class Telephone:public Name
{
    char *telephone;
public:
    Telephone(char *nm,char *tel):Name(nm){telephone=tel;}
    void Show_telephone(){ cout<<telephone<<endl;}
};
int main()
{
    Name *p1,obj1("Wang");
    Telephone *p2,obj2("Zhang","5681553");
    p1=&obj1;
    p1->Show_name();
    p1=&obj2;
    p1->Show_name(); //p1->show_telephone: error
    p2=&obj2;
    p2->Show_name();
    p2->Show_telephone();
}

```

基类指针

派生类指针

基类指针指向派生类对象

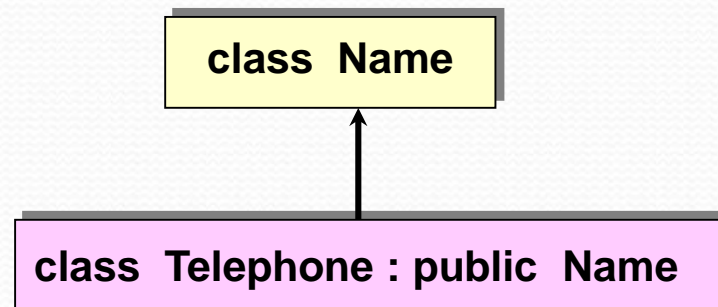
基类指针调用从基类继承的成员函数

派生类指针指向派生类对象

派生类指针调用派生类成员函数

➤ 基类指针只有经过强制类型转换之后，
才能引用派生类对象

```
class Name
{
    char *name;
public:
    Name(char *nm){ name=nm; }
    void Show_name(){ cout<<name<<endl; }
};
class Telephone:public Name
{
    char *telephone;
public:
    Telephone(char *nm,char *tel):Name(nm){telephone=tel;}
    void Show_telephone(){ cout<<telephone<<endl;}
};
int main()
{
    Name *p1,obj1("Wang");
    Telephone *p2,obj2("Zhang","5681553");
    p1=&obj1;
    p1->Show_name();
    p1=&obj2;
    p1->Show_name(); //p1->show_telephone: error
    ((Telephone *)p1)->show_telephon();
    p2=&obj2;
    p2->Show_name();
    p2->Show_telephone();
}
```



对基类指针强类型转换
调用派生类的成员函数

6.3 虚函数

前面的例子中，基类与派生类都定义了计算学费函数`CalaTuition()`，在用基类指针调用该函数时，应具体根据指针所指向的对象调用不同的函数，这就需要在基类中将该函数定义为虚函数，实现动态联编。

虚函数的定义：

虚函数是指一个在基类中被声明为`virtual`并在一个或多个派生类中被重定义的函数。基类中的虚函数需用`virtual`声明，派生类重定义时可以省略`virtual`。

虚函数的特别之处在于当用一个指向派生类对象的基类指针访问虚函数时，C++可以根据指向的对象类型确定在运行时调用哪个函数，所以，当指向不同的对象时，将执行该虚函数在不同派生类中的实现版本。


```

Class Base{
public:
    Base(int x,int y ){a=x;b=y;}
    virtual void show()
    { cout<<"调用基类Base的show()函数\n";
      cout<<a<<" "<<b<<endl;
    }
private:
    int a,b;
};

```

运行时的多态性（动态联编）：
在程序执行过程中动态地确定，
通过类继承关系和虚函数实现

```

Class Derived: public Base{
public:
    Derived(int x,int y,int z ):Base(x,y){c=z;}
    virtual void show()
    { cout<<"调用派生类的show()函数\n";
      cout<<c<<" "<<c<<endl;
    }
private:
    int c;
};

```

```

int main()
{ Base mb(50,50),*mp;
  Derived mc(10,20,30);
  mp=&mb; mp->show(); // 基类
  mp=&mc; mp->show(); // 派生类
  return 0;
}

```

1. 虚函数的定义

1、虚函数的定义

定义：

在基类中声明为virtual并在一个或多个派生类中被重新定义的成员函数

语法：

```
virtual 函数返回类型 函数名（参数表）  
{ 函数体 }
```

作用：

允许在派生类中重新定义与基类同名的函数，并且可以通过基类指针或引用来访问基类和派生类中的同名函数。

【例】 将print（）函数声明为虚函数以实现动态多态性，两个类的其余代码保持不变

```
#include <iostream>
using namespace std;
class Base    //定义基类Base
{public:
    Base(int x)    //基类的构造函数
    { a=x; }
    virtual void Print() //定义函数Print()为虚函数
    {cout<<"Base::a="<<a<<endl; }
    int a;    //基类的公有数据成员a
};
```

```
class Derive:public Base    //公有派生类
{public:    int a;
    //派生类中数据成员a与基类数据成员的同名
    Derive(int x,int y):Base(x) //派生类构造函数
    { a=y;    Base::a*=2; }
    virtual void Print( ) //此处的virtual可以省略
    {Base::Print();//访问基类函数
        cout<<"Derived::a="<<a<<endl;
        //访问本类中的同名成员
    }
};
```

//定义一个函数 f 以基类的引用作形式参数

```
void f(Base &bb)
```

```
{    bb.Print( ); } //通过引用调用Print( )函数
```

```
void main()
```

```
{ Base b(100),*ps; //定义基类指针ps实现动态多态性
```

```
  Derive d(200,300);
```

```
  ps=&b;           //将指针 ps 指向基类对象
```

```
  ps->Print( );    //调用的是基类的同名虚函数
```

```
  ps=&d;           //将指针 ps 指向 派生类对象
```

```
  ps->Print( );    //调用的是派生类的同名虚函数
```

```
  f(b);           //调用基类的Print( )函数
```

```
  f(d);           //调用派生类的Print( )函数
```

```
}
```

程序的运行结果为:

Base::a=100 //此行调用基类的print函数

Base::a=400 //调用派生类的print函数

Derived::a=300

Base::a=100

Base::a=400 //调用派生类的print函数

Derived::a=300

虚函数实现运行时多态性的设计方法

- 设计一个类层次，并在类层次中定义虚函数
- 在外部程序中定义基类的指针（引用）
- 程序中既可以把基类对象的地址赋给基类的指针，也可以把派生类对象的地址赋给基类的指针
- 当外部程序发送的消息是虚函数时，若基类指针指向基类对象时，系统就调用基类中的成员函数；若基类指针指向派生类对象时，系统就调用派生类中的成员函数。

虚函数的访问特性：

（1）正常访问特性：虚函数是类的成员函数，所以遵循类成员的访问规则。当基类型的对象直接调用虚函数时，调用的是基类中的虚函数版本，而用派生类对象直接调用虚函数时，调用的派生类中实现的虚函数版本。也就是说，当通过“对象名.”调用虚函数时，则在编译阶段根据调用对象的类型确定的虚函数版本，属于静态多态。如例6-7中6-6部分。

（2）特殊访问特性：通过基类指针、基类引用调用虚函数时，是在程序运行阶段根据具体指向或引用的对象，调用该对象所属类的虚函数实现版本，实现动态联编。如例6-7

2、虚函数的注意事项

- 1、用函数实现运行多态性的关键之处是必须用基类的指针（或引用）调用虚函数。
- 2、在派生类中重定义的基类虚函数可以省略virtual关键字，但必须与基类的虚函数原型相同，即其成员函数名、参数个数、参数类型以及返回值类型要求和基类的虚函数完全一样，否则其虚特性将丢失而成为普通的重载函数。
- 3、虚函数必须是类的成员函数，不能是全局函数、友元函数、静态成员函数。
- 4、一旦一个函数被说明为virtual，将保持虚特性，不管其经过多少层派生。
- 5、不能定义虚构造函数，因为建立一个派生类对象时，必须从类层次的根开始，沿着继承路径逐个调用基类的构造函数，不能选择性的调用构造函数，定义虚构造函数出现语法错误。
- 6、可以定义虚析构函数，用于指引 delete 运算符正确析构动态对象

虚析构造函数应用举例

未用虚析构造函数时

```
class B
{ public:
    ~B() { cout << "B::~~B() is
        alled.\n" ; }
};
```

```
class D : public B
{ public:
    ~D(){ cout << "D::~~D() is
        called.\n" ; }
};
```

```
int main()
```

```
{ B *Ap = new D ;
```

```
  D *Bp = new D ;
```

```
  cout << "delete first object:\n" ;
```

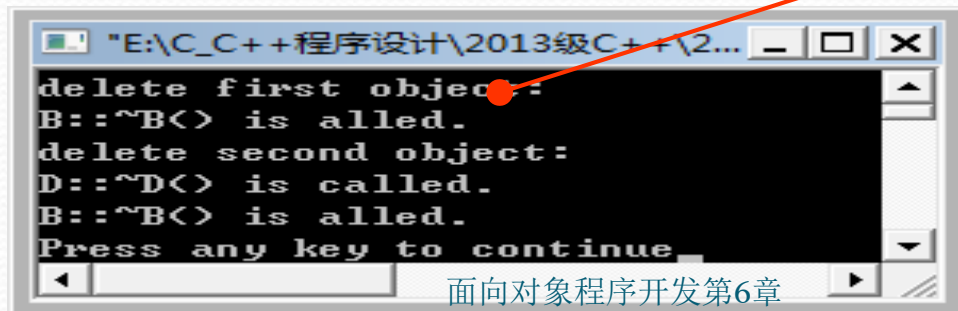
```
  delete Ap; //区分静态与动态联编
```

```
  cout << "delete second object:\n" ;
```

```
  delete Bp ;
```

```
}
```

用基类指针建立派
生类的动态对象



```
"E:\C_C++程序设计\2013级C++\2... _ _ X
delete first object:
B::~~B() is alled.
delete second object:
D::~~D() is called.
B::~~B() is alled.
Press any key to continue
```

析构由基类指针建立的派生类对象

只调用基类析构函数（不调用派生类析构）

虚析构造函数应用举例

使用虚析构造函数时

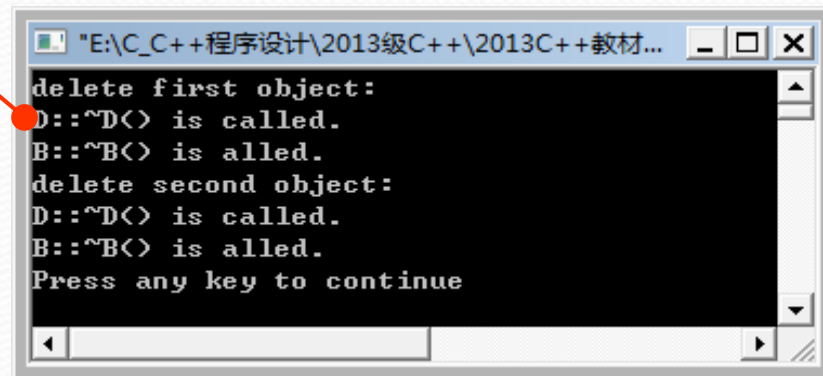
```
class B
{ public:
    virtual ~B() { cout << "B::~~B() is alled.\n" ; }
    alled.\n" ; }
};
```

```
class D : public B
{ public:
    ~D(){ cout << "D::~~D() is
called.\n" ; }
};
```

正确调用派生类析构
函数释放所有资源

基类定义析构造函数为虚函数，
则该基类的所有派生类的析构造函数自
动成为虚析构造函数。

```
int main()
{ B *Ap = new D ;
  D *Bp = new D ;
  cout << "delete first object:\n" ;
  delete Ap;
  cout << "delete second object:\n" ;
  delete Bp ;
}
```



```
"E:\C_++程序设计\2013级C++\2013C++教材..."
delete first object:
D::~~D() is called.
B::~~B() is called.
delete second object:
D::~~D() is called.
B::~~B() is called.
Press any key to continue
```

虚析构造函数应用举例

```
class B
{ public:
virtual ~B() { cout << "B::~~B() is alled.\n" ; }
    alled.\n" ; }
};
```

```
class D : public B
{ public:
    ~D(){ cout << "D::~~D() is
called.\n" ; }
```

```
int main()
{ B *Ap = new D ;
  D *Bp = new D ;
  cout << "delete first object:\n" ;
  delete Ap;
  cout << "delete second object:\n" ;
  delete Bp ;
}
```

- 定义了基类虚析构造函数，基类指针指向的 派生类动态对象也可以正确地用delete析构
- 设计类层次结构时，提供一个虚析构造函数，能够使派生类对象在不同状态下正确调用析构造函数

- 析构函数被声明为虚函数的理由：
 - 如果基类的析构函数声明为虚函数，则该类所有派生类的析构函数也自动成为虚函数而无需显式声明
 - 在这种情况下，由于实施多态性时是通过将基类的指针指向派生类的对象，如果删除该指针，就会调用该指针指向的派生类的析构函数，而派生类的析构函数又自动调用基类的析构函数，这样整个派生类的对象被完全释放

- 在虚函数定义和多态性的应用中，要注意以下几点
 - 一旦在基类中指定某成员函数为虚函数，那么，派生类（甚至是派生类的派生类，...）中对其重载定义的成员函数（原型与在基类中完全一样）均为虚函数
 - 若在函数原型声明中用**virtual**声明为虚函数，则在该函数定义时不能再加关键字**virtual**
 - 只有类的**非静态成员函数**才可以是虚函数。因为，通过虚函数表现多态性是一个类的派生关系，一般的普通函数不具备这种派生关系
 - 派生类必须以**公有方式**继承基类，这是**赋值兼容**原则的使用前提。派生类只有从基类公有继承，才允许基类的指针指向派生类对象，以及基类的引用是派生类对象的别名

6.4 纯虚函数和抽象类

1、纯虚函数

- (1) 是指在基类中声明但没有定义的虚函数
- (2) 纯虚函数的具体实现依赖于不同的派生类。
- (3) 定义纯虚函数的一般格式为：

`virtual 返回类型 函数名（参数表）=0; //该函数不需要定义具体实现的函数体。`

纯虚函数是一个在基类中说明的虚函数，它在基类中没有定义，而要求派生类根据需要定义自己的实现版本。通过纯虚函数，基类为各派生类提供一个公共接口。

注意：从基类继承来的纯虚函数，在派生类中仍然是虚函数，若派生类没有实现，则仍然是纯虚函数

- 纯虚函数的特点及用法如下：

- 纯虚函数是一种没有函数体的特殊虚函数，在原型声明时，将“=0”写在虚函数原型最后来表示。
- 纯虚函数不能被调用，因为它只有函数名，而无具体实现代码，无法实现具体的功能。该函数只有在派生类中被具体定义后才可调用
- 纯虚函数的作用在于基类给派生类提供一个标准的函数原型，统一接口，为实现动态多态性打下基础，派生类将根据需要给出纯虚函数的具体实现代码

2、抽象类

(1) 定义

如果一个类中至少有一个纯虚函数，那么这个类就是抽象类

纯虚函数的引入是为了方便使用多态特性，当在基类中需要定义虚函数，而又不需要具体实现时就需要定义纯虚函数。因为纯虚函数没有实现部分，不能产生对象，只能作为基类。

2、抽象类

(2) 使用抽象类的要求

抽象类不能实例化

抽象类只作为基类被继承

可以定义指向抽象类的指针或引用

如果抽象类的派生类中仍没有定义纯虚函数，则派生类也为一个抽象类。

(3) 纯虚函数为各派生类提供一个公共界面

体现 “一个接口，多种方法”：即将有关的类组织在一个继承层次结构中，由抽象类只描述这组子类共同的操作接口，而完整的实现留给子类。

- 对于抽象类，注意几点：

- 抽象类不能生成对象，因为该类中的纯虚函数无实现代码
- 抽象类可以定义其指针或引用，用来实现动态多态性。但是，不能用抽象类作为参数类型、函数返回值类型或显式转换的类型
- 抽象类的基类不能是普通类（即不是抽象类的类），抽象类只能作基类，是下面诸多的派生类的集中归宿。通常抽象类要有它的派生类，如果派生类中还有纯虚函数，则该派生类仍为抽象类。但是最终总会有具体类，来给纯虚函数一个具体实现，这样才有意义
- 抽象类除了必须至少有一个纯虚函数以外，还可以定义普通成员函数或虚函数

//figure.h

class figure

{ protected : double x,y;

public: void set_dim(double i, double j=0) { x = i ; y = j ; }

virtual void show_area() = 0 ; //纯虚函数

};

class triangle : public figure

{ public :

void show_area()

{ cout<<"Triangle with high "<<x<<" and base "<<y<<" has an area of "
<<x*0.5*y<<"\n"; }

};

class square : public figure

{ void show_area()

{ cout<<"Square with dimension "<<x<<"*"<<y<<" has an area of "
<<x*y<<"\n"; }

};

class circle : public figure

{ void show_area()

{ cout<<"Circle with radius "<<x;
cout<<" has an area of "<<3.14*x*x<<"\n";
}

}

};

例1：抽象类和纯虚函数应用


```
//figure.h
```

```
class figure
```

```
{ protected : double x,y;
```

```
public: void set_dim(double i, double j=0) { x =  
        virtual void show_area() = 0 ;
```

```
};
```

```
class triangle : public figure
```

```
{ public :
```

```
    void show_area()
```

```
    { cout<<"Triangle with high "<<x<<" and  
      "<<x*0.5*y<<"\n"; }
```

```
};
```

```
class square : public figure
```

```
{ p void show_area()
```

```
    { cout<<"Square with dimension "<<x<<"  
      <<x*y<<"\n"; }
```

```
};
```

```
class circle : public figure
```

```
{ { void show_area()
```

```
    { cout<<"Cir
```

```
    cout<<" ha
```

```
    }
```

```
};
```

```
};
```

例1: 抽象类和纯虚函数应用

```
#include<iostream>
```

```
using namespace std ;
```

```
#include"figure.h"
```

```
int main()
```

```
{ triangle t ;      //派生类对象
```

```
  square s ;   circle c;
```

```
  t.set_dim(10.0,5.0) ;
```

```
  t.show_area();
```

```
  s.set_dim(10.0,5.0) ;
```

```
  s.show_area() ;
```

```
  c.set_dim(9.0) ;
```

```
  c.show_area() ;
```

```
}
```

```
C:\WINDOWS\system32\cmd.exe
```

```
Triangle with high 10 and base 5 has an area of 25
```

```
Square with dimension 10×5 has an area of 50
```

```
Circle with radius 9 has an area of 254.34
```

```
请按任意键继续. . .
```

```
//figure.h
```

```
class figure
```

```
{ protected : double x,y;  
  public: void set_dim(double i, double  
          virtual void show_area()  
};
```

```
class triangle : public figure
```

```
{ public :  
  void show_area()  
  { cout<<"Triangle with high "<<  
    "<<x*0.5*y<<"\n"; }  
};
```

```
class square : public figure
```

```
{ p void show_area()  
  { cout<<"Square with dimension  
    <<x*y<<"\n"; }  
};
```

```
class circle : public figure
```

```
{ void show_area  
  { cout<<"Cir  
    cout<<" ha  
  }  
}  
};
```

```
#include<iostream>
```

```
using namespace std ;
```

```
#include"figure.h"
```

```
int main()
```

```
{ figure *p; // 声明抽象类指针
```

```
  triangle t; square s; circle c;
```

```
  p=&t;
```

```
  p->set_dim(10.0,5.0); //  
  triangle::set_dim()
```

```
  p->show_area();
```

```
  p=&s;
```

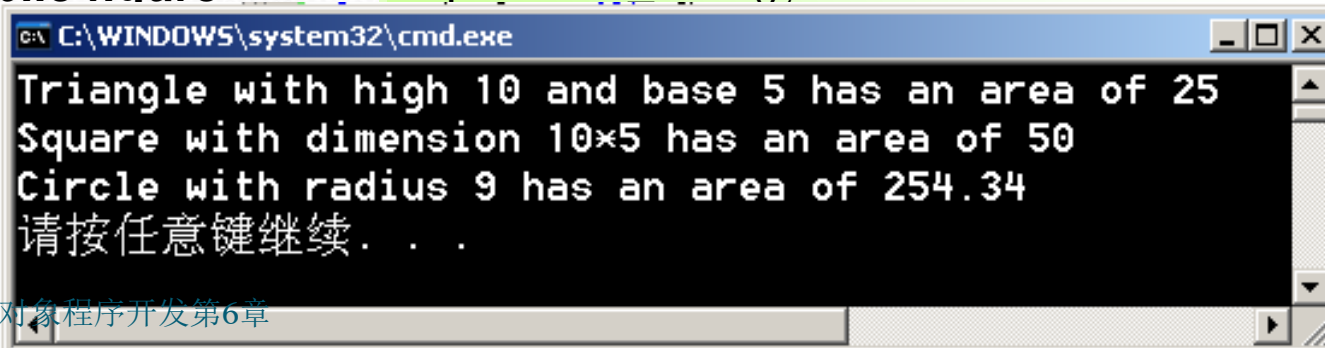
```
  p->set_dim(10.0,5.0); //  
  square::set_dim()
```

```
  p->show_area();
```

```
  p=&c;
```

```
  p->set_dim(9.0); //  
  circle::set_dim()
```

```
  p->show_area();
```



```
C:\WINDOWS\system32\cmd.exe  
Triangle with high 10 and base 5 has an area of 25  
Square with dimension 10x5 has an area of 50  
Circle with radius 9 has an area of 254.34  
请按任意键继续. . .
```


例1: 使用抽象类引用

```
#include<iostream>
using namespace std ;
class Number
{ public :   Number (int i) { val = i ; }
           virtual void Show() = 0 ;
  protected: int val ;
};
class Hex_type : public Number
{ public:   Hex_type(int i) : Number(i) { }
           void Show() { cout << "Hexadecimal: " << val << endl ; }
};
class Dec_type : public Number
{ public:   Dec_type(int i) : Number(i) { }
           void Show() { cout << "Decimal: " << val << endl ; }
};
class Oct_type : public Number
{ public:   Oct_type(int i) : Number(i) { }
           void Show() { cout << "Octal: " << oct << val << endl ; }
};
```

```
void fun( Number & n )    // 抽象类的引用参数
{ n.Show() ; }
int main()
{ Dec_type n1(50);
  fun(n1);
  Hex_type n2(50);
  fun(n2);
  Oct_type n3(50);
  fun(n3);
}
```

大家先理解一下
main函数中是如何调用的，结果会是什么样的

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

```
class Number
```

```
{ public :   Number (int i) { val =
virtual void Show()
```

```
protected: int val ;
```

```
};
```

```
class public Number
```

```
{ public:   Hex_type(int i) : Number
void Show() { cout <<
```

```
endl ; }
```

```
};
```

```
class Dec_type : public Number
```

```
{ public:   Dec_type(int i) : Number
void Show() { cout << "
```

```
};
```

```
class Oct_type : public Number
```

```
{ public:   Oct_type(int i) : Number(i) { }
```

```
void Show() { cout << "Octal: " << oct << val << endl ; }
```

```
};
```

```
void fun( Number & n )
```

```
{ n.Show() ; }
```

```
int main()
```

```
{ Dec_type n1(50);
```

```
  fun(n1);
```

```
  Hex_type n2(50);
```

```
  fun(n2);
```

```
  Oct_type n3(50);
```

```
  fun(n3);
```

```
}
```

// 抽象类的引用参数

// Dec_type::Show()

// Hex_type::Show()

// Oct_type::Show()

抽象类引用

例2：使用抽象类引用

```
#include<iostream>
using namespace std ;
class Number
{ public:   Number (int i) { val =
           virtual void Show()
protected: int val ;
};
class Hex_type : public Number
{ public:   Hex_type(int i) : Number(i) { }
           void Show() { cout << "Hex: " << val << endl ; }
};
class Dec_type : public Number
{ public:   Dec_type(int i) : Number(i) { }
           void Show() { cout << "Decimal: " << val << endl ; }
};
class Oct_type : public Number
{ public:   Oct_type(int i) : Number(i) { }
           void Show() { cout << "Octal: " << val << endl ; }
};

void fun( Number & n )           // 抽象类的引用参数
{ n.Show() ; }

int main()
{ Dec_type n1(50);
  fun(n1);                       // Dec_type::Show()
  Hex_type n2(50);
  fun(n2);                       // 派生类对象 Show()
  Oct_type n3(50);
  fun(n3);                       // Oct_type::Show()
}
```

函数调用

派生类对象

例2：使用抽象类引用

```
#include<iostream>
using namespace std ;
class Number
{ public :   Number (int i) { val =
             virtual void Show()
protected: int val ;
};
class Hex_type : public Number
{ public:   Hex_type(int i) : Number(i) { }
           void Show() { cout << "Hex: " << val << endl ; }
};
class Dec_type : public Number
{ public:   Dec_type(int i) : Number(i) { }
           void Show() { cout << "Decimal: " << val << endl ; }
};
class Oct_type : public Number
{ public:   Oct_type(int i) : Number(i) { }
           void Show() { cout << "Octal: " << oct << val << endl ; }
};
```

```
void fun( Number & n )    // 抽象类的引用参数
{ n.Show() ; }
int main()
{ Dec_type n1(50);
  fun(n1);                // Dec_type::Show()
  Hex_type n2(50);
  fun(n2);                // Hex_type::Show()
  Oct_type n3(50);
  fun(n3);                // Oct_type::Show()
}
```


例2：使用抽象类引用

```
#include<iostream>
using namespace std ;
class Number
{ public :   Number (int i) { val =
             virtual void Show()
protected: int val ;
};
class Hex_type : public Number
{ public:   Hex_type(int i) : Number(i) { }
           void Show() { cout << "Hex: " << val << endl ; }
};
class Dec_type : public Number
{ public:   Dec_type(int i) : Number(i) { }
           void Show() { cout << "Decimal: " << val << endl ; }
};
class Oct_type : public Number
{ public:   Oct_type(int i) : Number(i) { }
           void Show() { cout << "Octal: " << oct << val << endl ; }
};
```

```
void fun( Number & n )    // 抽象类的引用参数
{ n.Show() ; }
int main()
{ Dec_type n1(50);
  fun(n1);                // Dec_type::Show()
  Hex_type n2(50);
  fun(n2);                // Hex_type::Show()
  Oct_type n3(50);
  fun(n3);               // Oct_type::Show()
}
```

- 多态性使得发送同一消息产生不同的响应成为可能，方便了编程。本章主要有以下内容：
 - 多态性的两种形式：静态多态性和动态多态性。
 - 继承、虚函数、基类的指针和引用共同实现了动态多态性，虚函数在基类和公有派生类中要求函数原型完全一致，否则将变成同名覆盖现象而不具有动态多态性。
 - 只给出了函数声明而没有给出具体实现的虚成员函数称为纯虚函数;至少包含了一个纯虚函数的类称为抽象类。

小结

- 虚函数和多态性使软件设计易于扩充。
- 派生类重载接口相同的虚函数特性不变。
- 如果代码关联在编译时确定，称为静态联编。
- 基类指针可以指向派生类对象；以及基类中拥有虚函数，是支持多态性的前提。代码在运行时的晚期匹配称为动态联编。
- 如果一个基类中包含虚函数，通常把它的析构函数声明为虚析构函数。
- 纯虚函数由派生类定义实现版本。
- 具有纯虚函数的类称为抽象类。抽象类只能作为基类，不能建立实例化对象。抽象类指针使得派生的具体类对象具有多态操作能力。

窗口	对象	对象类别	对象属性及其取值		
			objectName 对象名称	windowTitle 标题	Text 文本
矩形面积计算	矩形面积计算对话框	Dialog（对话框）	RectangleDialog	矩形面积计算	无
	矩形长	Label	默认值	无	矩形长：
	矩形长（输入）	Line Edit	lengthEdit	无	无
	矩形宽	Label	默认值	无	矩形宽：
	矩形宽（输入）	Line Edit	widthEdit	无	无
	矩形面积	Label	默认值	无	圆柱体体积
	矩形面积显示	Line Edit	areaEdit	无	无
	计算按钮	Push Button	calculateBotton	无	计算
	退出按钮	Push Button	exitBotton	无	退出

窗口	对象	对象类别	对象属性及其取值		
			objectName 对象名称	windowTitle 标题	Text 文本
正方形面积计算	正方形面积计算对话框	Dialog（对话框）	SquareDialog	矩形面积计算	无
	正方形边长	Label	默认值	无	正方形边长：
	正方形边长（输入）	Line Edit	lengthEdit	无	无
	正方形面积	Label	默认值	无	圆柱体体积
	正方形面积显示	Line Edit	areaEdit	无	无
	计算按钮	Push Button	calculateBotton	无	计算
	退出按钮	Push Button	exitBotton	无	退出

窗口	对象	对象类别	对象属性及取值		
			objectName 对象名称	windowTitle 标题	Text 文本
梯形面积计算	梯形面积计算对话框	Dialog（对话框）	TrapezoidDialog	梯形面积计算	无
	梯形上底长	Label	默认值	无	梯形上底长：
	梯形上底长（输入）	Line Edit	topEdit	无	无
	梯形下底长	Label	默认值	无	梯形下底长：
	梯形下底长（输入）	Line Edit	bottomEdit	无	无
	梯形高	Label	默认值	无	梯形高：
	梯形高（输入）	Line Edit	heightEdit	无	
	正方形面积	Label	默认值	无	圆柱体体积
	正方形面积显示	Line Edit	areaEdit	无	无
	计算按钮	Push Button	calculateBotton	无	计算
	退出按钮	Push Button	exitBotton	无	退出



上海理工大学

UNIVERSITY OF SHANGHAI FOR SCIENCE AND TECHNOLOGY

UNIVERSITY OF SHANGHAI FOR SCIENCE AND TECHNOLOGY

谢谢!



应用案例

- 定义圆类Circle，其私有数据成员半径（radius）、公有成员函数返回面积 GetArea()；以圆类为基类派生定义圆球类Globe和圆柱体类Cylinder，计算圆球和圆柱体的表面积。
- 继续增加定义派生类圆锥体Cone，计算其表面积和体积。

练习

编程计算圆、矩形等形状的面积：

把各种形状的共性抽取出来，构建一个基类，然后从基类中派生出各种具体的形状。

基类中定义的面积函数可以返回o值。

```
class Shape
{
    virtual double Area ( ) ;
};
```

自定义Circle类、Rectangle类，计算面积输出。

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

若修改函数
如何?

```
};
```

```
class Hex_type : public Number
```

```
{ public: Hex_type(int i) : Number(i) { }
```

```
void Show() { cout << "Hex: " << val << endl ; }
```

```
};
```

```
class Dec_type : public Number
```

```
{ public: Dec_type(int i) : Number(i) { }
```

```
void Show() { cout << "Decimal: " << val << endl ; }
```

```
};
```

```
class Oct_type : public Number
```

```
{ public: Oct_type(int i) : Number(i) { }
```

```
void Show() { cout << "Octal: " << oct << val << endl ; }
```

```
};
```

```
void fun( Number *n )
```

```
{ n->Show() ; }
```

```
int main()
```

```
{ Dec_type n1(50);
```

```
fun(&n1);
```

```
Hex_type n2(50);
```

```
fun(&n2);
```

```
Oct_type n3(50);
```

```
fun(&n3);
```

```
}
```

// Dec_type::Show()

// Hex_type::Show()

// Oct_type::Show()

```
C:\WINDOWS\system32\c...
Decimal: 50
Hexadecimal: 32
Octal: 62
请按任意键继续...
```