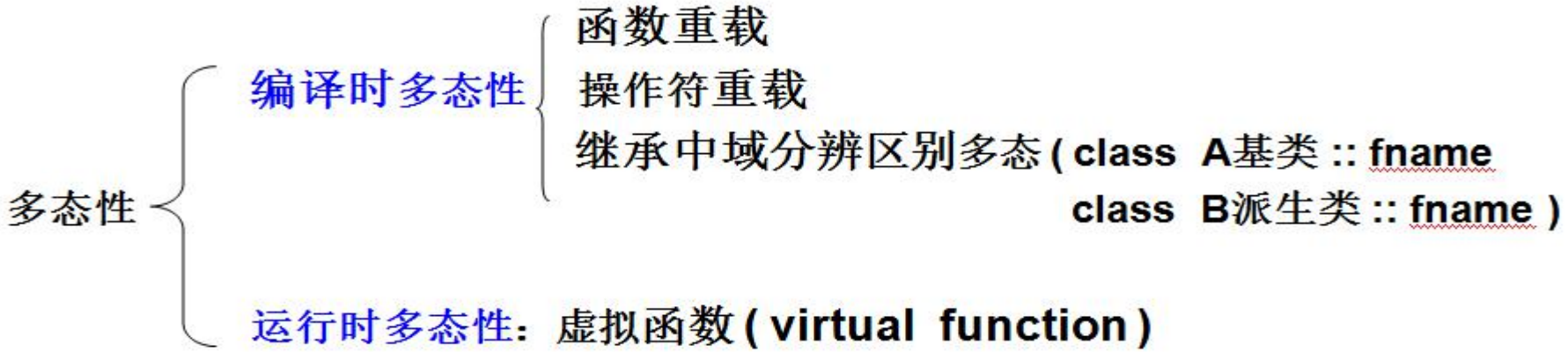
多态与抽象类

吴清锋

2021年春

掀起你的盖头来

- 多态是面向对象编程的一个强大功能，在大多数C++程序中都要用到；
- 多态性
  - 目的：指发出同样的消息被不同类型的对象接收时有可能导致完全不同的行为。
  - 是一现象：指的是同一符号或名字在不同情况下具有不同解释，即是指同一个函数的多种形态。
- C++支持两种多态性，编译时的多态性和运行时的多态性。



概述：运行时多态

运行时的多态性是指在程序执行前，无法根据函数名和参数来确定该调用哪一个函数，必须在程序执行过程中，根据执行的具体情况来动态地确定。它是通过类继承关系和虚函数来实现的。目的也是建立一种通用的程序。通用性是程序追求的主要目标之一。

回忆：公有继承下的赋值兼容规则

所谓赋值兼容规则指的是不同类型的对象间允许相互赋值的规定。面向对象程序设计语言中，在公有派生的情况下，允许将派生类的对象赋值给基类的对象，但反过来却不行，即不允许将基类的对象赋值给派生类的对象。这是因为一个派生类对象的存储空间总是大于它的基类对象的存储空间。若将基类对象赋值给派生类对象，这个派生类对象中将会出现一些未赋值的不确定成员。

多态是以继承为基础的，而“公有继承下的赋值兼容规则”则是多态的理论支撑！

允许将派生类的对象赋值给基类的对象，有以下三种具体作法：

1.直接将派生类对象赋值给基类对象，例如：

```
Base objB;
Derived objD;  //假设Derived已定义为Base的派生类
ObjB=objD;  //合法
ObjD=objB;  //非法
```

2.定义派生类对象的基类引用，例如：

```
Base &b=objD
```

3.用指向基类对象的指针指向它的派生类对象，例如：

```
Base *pb=&objD;
```

程序员视角：对于多态的期望

假设有：

- 基类及对象 **Base objB;**　　派生类及对象**Derived objD;**
- 若有**f()**

不是函数的重载

原先：

- f(基类) { … }
- f(派生类) { …}

现在期望：

- 一个 f(基类) { .. }
- 根据传递的实参，f表现出与传递信息对应的操作，程序员不需要单独再写

- 自动适应类型变化的性质，就是多态性！
  这是一个极为强大的机制，代码人员常不能事先确定要处理哪种类型的对象，即在设计期间或编译期间不能确定类型，只能在运行期间确定。使用多态性可以轻松地解决这个问题！

编程中出现的困惑1：fun()函数的诉求

```
class Base {
  public:
   void Print()  {       //大
          cout<<"Base::Print"<<endl;
   }
class Derived: public Base {
  public:
   void Print()  {     //小
          cout<<"Derived::Print"<<endl;
   }
 void fun(const Base &cb) {
          cb.Print();
 }
 int main() {
  Derived obj;
  fun(obj);
  return 0;
}
```

- 结果是：
  Base::Print

- 分析：
  obj虽然是派生类对象，但在赋值给基类对象操作之后，被同化了。

编程中出现的困惑2：基类指针成员函数调用

```
#include<iostream.h>
class Member{
public:
    void answer()
{ cout<<"I am a member.\n"; }
};
class  Teacher: public Member{
public:
   void answer()
{ cout<<"I am a teacher.\n"; }
};
class Student: public Member{
public:
   void answer()
{ cout<<"I am a student.\n"; }
};
```

```
void main() {
        Member aMember;
        Teacher  aTeacher;
        Student   aStudent;
        Member *Who; //基类指针
        Who=&aMember;
        Who->answer();
        Who=&aTeacher;
        Who->answer();
        Who=&aStudent;
        Who->answer();
```

程序输出结果：

I am a member.

I am a member.

I am a member.

没有出现程序设计者期待的随着赋值不同而展现不同的结果

这是由于函数answer()不是一个虚函数，所有对它的调用语句都是静态绑定的，即三个Who->answer()；调用执行的是同样的代码。

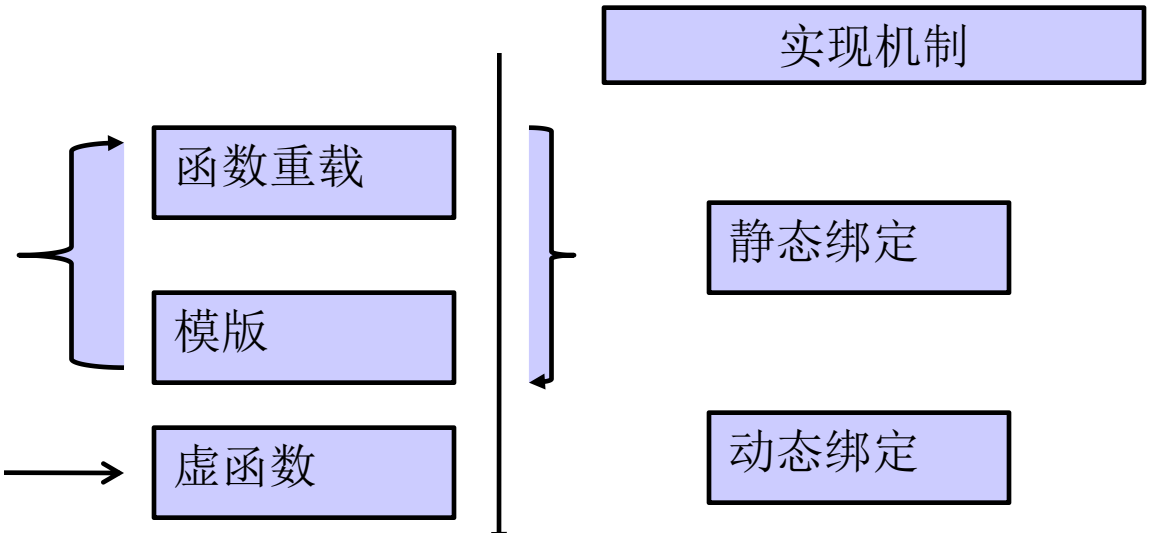
究竟什么是多态？

- 一个面向对象的系统常常要求一组具有基本语义的方法**能在同一个接口（如函数）**下为不同的对象服务（解释为不同不同意义），这就是多态性。
- **多态是一种能力，即：同样的消息被不同对象接收时，产生完全不同的行为。**

⇑

调用同名不同功能的函数

- 多态分类：
  - 编译时多态
  - 运行时多态



虚函数与多态类

在一个类中用关键字virtual 说明的成员函数称为虚函数。定义了虚函数的类称为多态类。虚函数原型语句格式：

virtual 类型 函数名（参数表）；

- 在基类中某个成员函数被声明为虚函数后，这个成员函数通常要在派生类中被重新定义。定义一个虚函数的目的是为了在程序运行时自动选择各派生类中的重定义版本，所以一个多态的基类一定要定义一个以上的派生类才有意义。
- 在派生类中重新定义虚函数时，其函数原型（包括返回值类型、函数名、参数个数、参数类型及顺序）必须与基类中的原型完全相同。否则编译时会出错或被当作函数重载。
- 一个指向基类的指针可以指向它的公有继承的派生类。定义虚函数的目的就是想统一用一个基类对象指针去访问不同派生类中虚函数的重定义代码。

多态类与虚函数访问

```
#include<iostream.h>
class Poultry {
public:
   virtual void can_fly ()  //虚函数定义
{  cout<<"Yes! I can."<<endl; }
};
class Cock : public Poultry{
public:
void can_fly ()  //虚函数重定义
{  cout<<"Yes! I can. But I can't fly high."<<end
};
class Duck : public Poultry{
public:
void can_fly ()  //虚函数重定义
{  cout<<"No! I can't. But I can swimming."<<e
};
```

```
void main()
{
        Poultry   anyPoultry, *ptr;
        Cock aCock;
        Duck aDuck;
        ptr =&anyPoultry;
        ptr->can_fly();
        ptr =&aCock;
        ptr->can_fly();
        ptr =&aDuck;
        ptr->can_fly();
        aCock.can_fly();
        aDuck.can_fly();
}
```

程序输出：

```
Yes! I can.
Yes! I can. But I can't fly high.
No! I can't. But I can swimming.
Yes! I can. But I can't fly high.
No! I can't. But I can swimming.
```

以上运行结果表明，只要把派生类对象的地址传递给基类指针，就可以直接使用基类指针调用派生类中虚函数的重定义版本，与通过派生类对象名调用该函数有相同的执行结果。虚函数的引入，使类似ptr->can_fly(); 这样的函数调用语句有了动态选择执行代码的智能。这一点是通过动态绑定技术实现的。

例子2：虚函数的作用。

```
#include<iostream.h>
class Base {
   private:
     int a,b;
   public:
      Base(int x,int y)        {  a=x; b=y; }
      virtual void show()      //定义虚函数show()
     { cout<<"Base----------\n"; cout<<a<<" "<<b<<endl;}
    };
class Derived : public Base    {
    private:
     int c;
    public:
     Derived(int x,int y,int z):Base(x,y){c=z; }
     void show()             //重新定义虚函数show()
     { cout<< "Derived---------\n"; cout<<c<<endl;}
 };
```

```
void main()
{
        Base mb(60,60),*pc;
        Derived mc(10,20,30);
        pc=&mb;
        pc->show();     //调用基类Base的show()版本
        pc=&mc;
        pc->show();     //调用派生类Derived的show()版本
 }
```

程序运行结果如下：

```
Base---------
60   60
Derived--------
30
```

代码的思考

- 多态性仅用于类层次结构（即多态需要使用派生类），从一个类中派生另一个类是多态性的基本条件。
- 虚函数的本质：不是重载声明而是覆盖。
- 多态调用方式：通过基类指针或引用，执行时会根据指针指向的对象的类，决定调用哪个函数。若使用：成员访问运算符.来实现，那是非多态调用

虚函数的特性

- 虚函数是动态绑定的基础，是非静态的成员函数。
- 在类的声明中，在函数原型之前写**virtual**。
- **virtual** 只用来说明类声明中的原型，不能用在函数实现时。
- 具有继承性，基类中声明了虚函数，派生类中无论是否说明，同原型函数都自动为虚函数。

虚函数与重载函数的关系

在一个派生类中重新定义基类的虚函数，但它不同于一般的函数重载。

◆ 普通的函数重载时，其函数的参数或参数类型必须有所不同，函数的返回类型也可以不同。

◆ 当覆盖一个虚函数时，也就是说在派生类中重新定义虚函数时，要求函数名、返回类型、参数个数、参数的类型和顺序与基类中的虚函数原型完全相同。

◆ 如果仅仅返回类型不同，其余均相同，系统会给出错误信息;

◆若仅仅函数名相同，而参数的个数、类型或顺序不同，系统将它作为普通的函数重载，这时将丢失虚函数的特性。

支撑技术：绑定及动态绑定其实现技术

所谓绑定（binding）是指编译程序将源程序中的函数调用语句与该函数的执行代码联系在一起的过程。即，绑定就是让函数调用语句找到它要执行的代码。
在非面向对象程序设计语言中，绑定过程都是静态的，也即，所有的绑定都是在编译期完成的。静态绑定的函数调用语句有唯一确定的语义，不可能具有多态性。

动态绑定是通过虚函数表实现的。

对于含有虚函数的多态类，编译器为每个对象生成一个虚表指针。即在每个对象的内存映像中增加了一个_vfptr指针，它指向一个虚函数表vtable。在基类的虚函数表中列出基类所有虚函数的入口地址，在派生类的虚函数表中列出派生类的所有虚函数的入口地址。

多态点类定义及测试

```
#include<iostream.h>
class Point{
  protected:
    int x, y;
  public:
    Point(int a, int b){x=a; y=b;}
    virtual void print()
      { cout<<"(“<<x<<”,”<<y<<”)”<<endl; }
    virtual void forward(int d);
};
class Point_3d: public Point{
    int z;
  public:
    Point_3d(int a, int b, int c):Point(a,b){z=c;}
    virtual void print()
{ cout<<"[“<<x<<”,”<<y<<”,” <<z<<”]”<<endl; }
    virtual void forward(int d);
};
```

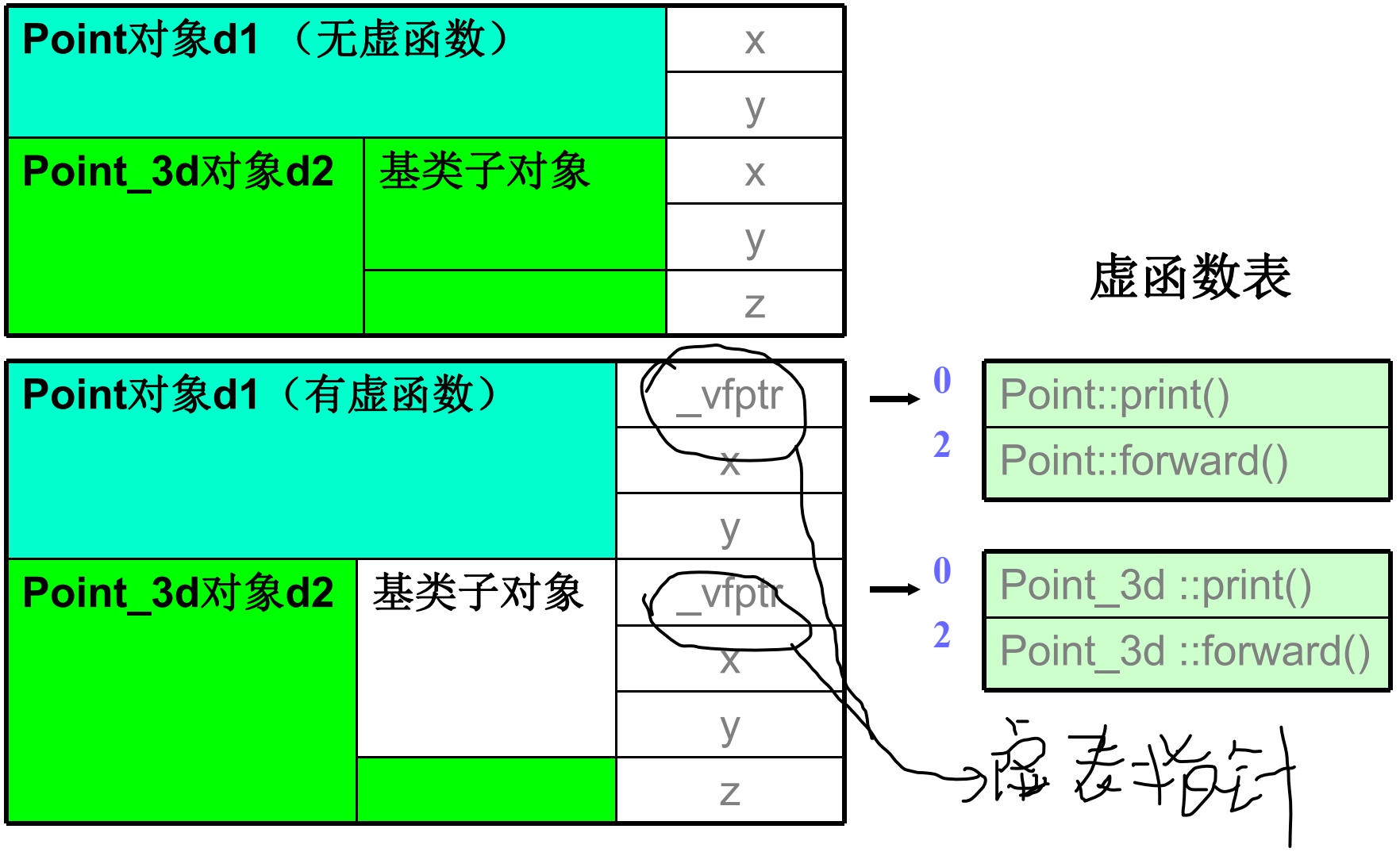
程序运行结果

```
(2,4)
[1,3,5]
(2,4)
[6,8,10]
```

```
void Point::forward(int d)
{ x=x+d; y=y+d;}
void Point_3d::forward(int d)
{ x=x+d; y=y+d; z=z+d;}

void main()
{
        Point d1(2,4);
        Point_3d d2(1,3,5);
        Point *ptr;
        ptr=&d1;
        ptr->print();
        ptr=&d2;
        ptr->print();
        ptr->forward(5);
        ptr=&d1;
        ptr->print();
        ptr=&d2;
        ptr->print();
}
```

有无虚函数时对象内存映象对比



虚函数动态绑定的调用步骤

首先为多态类的基类声明一个指针变量，然后让这个指针变量指向此多态类继承树中某一个类的对象。

由于基类指针指向对象内存映像的首地址，它直接访问的是该对象的虚表指针，下一步由具体对象的虚表指针就可以访问到该对象所在的类中为虚函数定义的那一份代码。

虚函数应用：“一个接口，多种实现”

虚函数同派生类的结合可使C++支持运行时的多态性，实现了在基类定义派生类所拥有的通用接口，而在派生类定义具体的实现方法，即常说的“同一接口，多种方法”，它帮助程序员处理越来越复杂的程序。

具体来说，类的公开成员函数可以看作类封装体向外界提供的接口，虚函数的引入使得基类和它的派生类有了一个共同的接口。这种“一个接口，多种实现”概念所体现的动态多分支选择机制为面向对象系统提供了控制更大复杂性的能力。

利用虚函数求平面图形的面积1

重点分析：类Circle 、Triangle、Rectangle具有相同的接口area()，但有各自不同的实现方法。

```
#include<iostream.h>
const double PI=3.1416;
class  Figure{
protected:
        double a, b;
public:
        Figure(double x, double y=0)   { a=x; b=y;}
        virtual void area(){
        cout<<"Can't define area for an abstraction figure"<<endl;
        }
};
class Circle: public Figure {
public:
        Circle (double x):Figure(x){}
        virtual void area(){
        cout<<"Circle with radius "<<a<< " has an area of  "<<PI*a*a<<endl;
        }
};
```

利用虚函数求平面图形的面积2

```
class Triangle: public Figure {
public:
        Triangle(double x, double y):Figure(x, y){}
        virtual void area(){
        cout<<"Triangle with base "<<a<<" and high "<<b<<" has an area of
"<<0.5*a*b<<endl;
        }
};
class Rectangle: public Figure {
public:
        Rectangle(double x, double y):Figure(x, y){}
        virtual void area(){
        cout<<"Rectangle with lenth "<<a<< " and width "<<b<<" has an area of
"<<a*b<<endl;
        }
};
```

利用虚函数求平面图形的面积3

```
void main()
{
        Figure *p;
        Circle obj_c(10);
        Triangle obj_t(10, 5);
        Rectangle obj_r(12, 6);
        p=&obj_c;
        p->area();
        p=&obj_t;
        p->area ();
        p=&obj_r;
        p->area ();
}
```

程序运行结果：

```
Circle with radius 10 has an area of  314.16
Triangle with base 10 and high 5 has an area of 25
Rectangle with lenth 12 and width 6 has an area of  72
```

纯虚函数与抽象类

1 纯虚函数的定义

纯虚函数是在基类中只有说明而没有实现定义的虚函数，它的任何派生类都必须定义自己的实现版本。

纯虚函数定义形式：

virtual 类型 函数名（参数表）=0；

例如：对于从Circle 、Triangle、Rectangle抽象出的公共基类Figure，求面积的运算是无实际意义的：

```
virtual void area() {
    cout<<"Can't define area ...."<<endl; }
```

去掉其内联函数定义（即花括号）部分后show_area()就变成了纯虚函数。

纯虚函数的实现

通过将虚函数声明为纯虚函数，类的设计者强迫它的所有派生类都必须定义自己的方法实现版本。如果某一派生类没有给予出自己的实现版本而又企图创建它的对象，则将发生编译错误。

2 抽象类的概念和定义

- 从概念上讲，抽象类是表示一组具有某些共性的具体类的公共特征的

根据抽象类的概念和语法，判断以下语句的正误：

Figure a; //错，不能创建抽象类的对象

Figure *ptr; //对，可以声明指向抽象类的指针

Figure function1(); //错，抽象类不能作为函数返回类型

void function2(Figure); //错，抽象类不能作为函数参数类型

Figure& function3(Figure &); //对，可以声明指向抽象类的引用

例如：Figure类，经过修改area()函数为纯虚函数而成为抽象类：

```
class Figure {
  public:
                virtual void area()=0;
  };
```

例 用抽象类实现的菜单程序1

```
#include<iostream.h>
class Menu{  //抽象类
public:
        virtual void action()=0;  //纯虚函数
};
class Item1: public Menu{
public:
        virtual void action(){
                cout<<"Do something for create a new file"<<endl<<endl;
        }
};
class Item2: public Menu{
public:
        virtual void action(){
                cout<<"open a old file to read or write"<<endl<<endl;
        }
};
```

例 用抽象类实现的菜单程序2

```
class Item3: public Menu{
public:
        virtual void action()
    {   cout<<"save file to disk"<<endl<<endl;; }
};
class Item4: public Menu{
public:
        virtual void action()
    {   cout<<"save to another file"<<endl<<endl;       }
};
class Item5: public Menu{
public:
        virtual void action(){
                cout<<"print file"<<endl<<endl; }
};
```

例 用抽象类实现的菜单程序3

```
void main()
{       int select;
        Menu *fptr[5];
        fptr[0]=new Item1;
        fptr[1]=new Item2;
        fptr[2]=new Item3;
        fptr[3]=new Item4;
        fptr[4]=new Item5;
        do{     cout<<"1 new file"<<endl;
                cout<<"2 open file"<<endl;
                cout<<"3 new file"<<endl;
                cout<<"4 save to another file"<<endl;
                cout<<"5 print file"<<endl;
                cout<<"0 exit"<<endl<<endl;
                cin>>select;
                if (select>=1 && select<=5)
                        fptr[select-1]->action();
        }while(select!=0);
}
```

●从语法上看，抽象类是一种特殊的多态类，具有动态的多态性。

●比起多态类来，抽象类更侧重于表达类的抽象层次。抽象类和它的实现类的关系虽然也是一种继承关系，但这种继承与之前讨论的继承有一种质的区别。非抽象类的继承着眼点在于代码重用，称为类继承，抽象类的继承着眼点在于为一组具有某些共性的具体类提供统一的访问接口，称为接口继承。接口继承的目的是为复杂对象提供构造基础。

虚函数应用举例(自己阅读)

2. 多态数据结构

堆栈、队列、链表等数据结构中的数据通常都是单一类型的。利用类的多态性，可以构造异质的数据结构，即数据单元由不同类的对象组成的结构。例如，一个可以压入不同（长度）对象的堆栈。多态数据结构是面向对象的数据库、多媒体数据库的数据存储基础。例10.5的程序中定义了基类Phone和它三个派生类BP_user、Fax_user、Mobile_user, 它们分别代表拥有不同通信设备的人员， 这几个类具有两个共同的接口：insert()和print()。基类和派生类各自的实现代码能正确地创建和打印自己的对象。能够使用相同的对象指针是这几个类的对象的“异”中之“同”，正是这种共性，使他们能够进入同一个数据结构——链表中。

例10.5 多态（异质）链表实现。1

```
#include<iostream.h>
#include<string.h>
class Phone{
   friend class List;
protected:
   char name[20];
   char cityNo[5];
   char phoneNo[10];
   static Phone *ptr;
   Phone *next;
public:
   Phone(char*,char*,char*);
   virtual void insert();
   virtual void print();
};
```

```
class BP_user: public Phone{
   char server[10];
   char call[10];
public:
   BP_user(char*,char*,char*,char*,char*);
   void print();    //重新定义
   void insert();  //重新定义
};
class Fax_user: public Phone{
   char fax[10];
public:
   Fax_user(char*,char*,char*,char*);
   void print();    //重新定义
   void insert();  //重新定义
};
class Mobile_user: public Phone{
    char mobileNo[12];
public:
    Mobile_user(char*,char*,char*,char*);
     void print();    //重新定义
     void insert();  //重新定义
};
```

例10.5 多态（异质）链表实现。2

```
class List{
        Phone *head;  //链表头指针
public:
        List(){head=0;}
        void insert_node(Phone *node);  //插入一个对象
        void remove(char *name);  //删除一个对象
                        void print_list();  //打印链表
};

//以下是Phone及其派生类的实现部分

Phone::Phone(char* name,char* cityNo,char* phoneNo){
            Strcpy(Phone::name, name);
            strcpy(Phone::cityNo, cityNo);
            strcpy(Phone::phoneNo, phoneNo);
            next=NULL;
}
```

例10.5 多态（异质）链表实现。3

```
BP_user::BP_user(char* name,char* cityNo,char* phoneNo,char* server,
char* call)
:Phone(name,cityNo,phoneNo){
        strcpy(  BP_user::server, server);
        strcpy( BP_user::call, call);
}
Fax_user::Fax_user(char* name,char* cityNo,char* phoneNo,char* fax)
:Phone(name,cityNo,phoneNo){
        strcpy(  Fax_user::fax, fax);
}
Mobile_user::Mobile_user(char* name,char* cityNo,char* phoneNo,char*
mobileNo)
:Phone(name,cityNo,phoneNo){
        strcpy(  Mobile_user::mobileNo, mobileNo);
}
```

例10.5 多态（异质）链表实现。4

```
void Phone::insert(){
        ptr=new Phone(name, cityNo, phoneNo);
}
void BP_user::insert(){
        ptr=new BP_user(name, cityNo, phoneNo, server, call);
}
void Fax_user::insert(){
        ptr=new Fax_user(name, cityNo, phoneNo, fax);
}
void Mobile_user::insert(){
        ptr=new Mobile_user(name, cityNo, phoneNo, mobileNo);
}
```

例10.5 多态（异质）链表实现。5

```
void Phone::print(){
        cout<<endl<<"Name: "<<name<<"  Phone: "<<cityNo<<"-
"<<phoneNo<<endl;
}
void BP_user::print(){
        Phone::print();
        cout<<"BP: "<<server<<"-"<<call<<endl;
}
void Fax_user::print(){
        Phone::print();
        cout<<"Fax: "<<fax<<endl;
}
void Mobile_user::print(){
        Phone::print();
        cout<<"Mobile phone number: "<<mobileNo<<endl;
}
```

例10.5 多态（异质）链表实现。6

```
//以下是类List的实现部分
void List::insert_node(Phone *node){  //向异质链表中插入一个对象
        char key[20];
        strcpy(key,node->name);//以待插入对象的姓名作为关键字
        Phone *current_node=head;
        Phone *last_node=NULL;
        while(current_node!=0 &&strcmp(current_node->name, key)<0)
        {
                last_node=current_node;
                current_node=current_node->next;
        }
        node->insert();//虚函数调用将待插入对象赋给ptr指向的单元
        node->ptr->next=current_node;//插入链中
        if(last_node==0)
                head=node->ptr;
        else last_node->next=node->ptr;
}
```

```
void List::remove(char *name){  //按姓名删除链表中一个对象
    Phone *current_node=head;
    Phone *last_node=NULL;
    while(current_node!=0 &&strcmp(current_node->name, name)!=0)
    {
            last_node=current_node;
            current_node=current_node->next;
    }
    if(current_node!=0 &&last_node==0)//若为链首元素
    {
             head=current_node->next;
             delete current_node;
    }
    else if (current_node!=0 && last_node!=0)//若为链非首元素
    {
            last_node->next=current_node->next;
            delete current_node;
    }
}
void List::print_list()
{
       Phone *current=head;
       while(current!=0){
       current->print();
       current=current->next;
    }
}
```

例10.5 多态（异质）链表实现。7

例10.5 多态（异质）链表实现。8

```
Phone *Phone::ptr=0;
void main()
{           List people;
            Phone p1("Chen Kun","0851","3871186");
            Phone p2("A Zhi Gu Li","0991","4847191");
            Mobile_user p3("Zhang Zhiming","0851","6855441","13037863048");
            BP_user p4("Du Yajun","028","7722974","96960","2833955");
            Fax_user p5("Liang Tiao","023","65105787","65106879");
            people.insert_node(&p1);
            people.insert_node(&p2);
            people.insert_node(&p3);
            people.insert_node(&p4);
            people.insert_node(&p5);
            cout<<endl<<"Now there are five recoreds in the people object of
class List:"<<endl;
            people.print_list();
            people.remove("A Zhi Gu Li");
            people.remove("Chen Kun");
            cout<<endl<<"After remove two recoreds from the people object of
class List:"<<endl;
            people.print_list();
}
```

例10.5 多态（异质）链表实现。 9

讨论：

链表类List为类Phone的友元类，可以不经函数调用访问它的私有成员，提高访问效率。也可访问Phone及其派生类的静态成员——Phone类对象指针ptr。

异质链表中各类对象共存，结点之间只能用指针链接。各类对象独立，指针不能互传。任何一个派生类类的指针作为链表指针都有是不恰当的，必须抽象出一个基类，用基类指针来指向下一个对象，基类指针充当了异质链表指针的角色。

往异质链表中插入哪个类的对象，可以通过参数来传递。插入函数的参数为Phone *node, 形参node为基类指针，它也可以指向派生类的对象。

链表中按各对象共同具有的数据成员name作为关键字顺序排列。

基类中定义的静态指针是各对象的公用指针，不依赖对象而生存，使友元类可以随时取得插入结点指针值。

链表中各元素为不同类的对象，应有不同的输出函数。输出顺着链进行，在每个结点处用current->print()调用，current为指向当前对象的指针。

破解困惑的思路（勉强）

```
#include<iostream.h>
class A{
public:
        void show(){ cout<<"A"; }
};
class B:public A {
public:
        void show(){ cout<<"B"; }
};
void fn(A & x) {x.show();}
main()
{       A a,*pc;
        B b;
        fn(a); fn(b);
}
```

类型域方案可以做到，即实现**fn**函数如下：

```
void fn(A& x){
  switch(x.type){
    case A::A:
      x.calcTuition();
    break;
    case A::B:
      B& rx =
    static_cast<B&>(x);
      rx.calcTuition();
    break;
  }
}
```

评价：不敢恭维这种方法，因为它导致类编程与应用编程互相依赖，因而破坏了只关注局部细节的抽象编程.

虚函数与消息调用

动态绑定的目的是为程序引入消息机制。程序用指针形式实现的虚函数调用就是消息调用。

引发消息调用的人机交互操作或某一对象的状态改变统称消息事件。

不同的消息由消息名区分。消息调用可以携带多个不同的参数，其中第一个（隐含）参数是消息指针，即定义该消息的基类指针，消息调用时该指针指向接收消息的对象（即以接收消息的对象的内存映像地址为实参）。不同的消息接收者对同一消息调用执行不同的代码，做出不同的响应。例如在WINDOWS系统中，鼠标单击事件引发一条消息调用，接收消息的可能是窗体，某个按钮或者某一菜单项，系统将根据鼠标击中的位置分别做出相应的反应。