
NANKAI UNIVERSITY
1919

第八章 类的继承与多态性

主讲：刘晓光 张海威
张 莹 殷爱茹
沈 玮 宋春瑶
李雨森

- 类的继承与派生
- 派生类
- 虚基类与虚拟继承
- 多态性与虚函数
- 综合示例

类的继承与派生

派生类

虚基类与虚拟继承

多态性与虚函数

综合示例

类的继承与派生

C++程序用不同的类定义来表示一组数据以及对这些数据的操作与处理，而类之间往往具有某种关系，“继承与派生”就是类间的一种常用关系。

- 例如，交通工具→汽车→轿车→红旗轿车

  具有层次关系！

  - 汽车　是一种特殊的　交通工具
  - 轿车　是一种特殊的　汽车
  - 红旗轿车　是一种特殊的　轿车

类的继承与派生

继承(inheritance)与派生

- 该机制是面向对象程序设计使代码可以复用的最重要的手段
- 它允许程序员在保持原有类特性的基础上进行扩展，增加功能
- 继承产生新的类，称为派生类
  - 派生类可以访问基类的保护成员（在基类中以关键字protected标识的成员）
- 继承呈现了面向对象程序设计的层次结构，体现了由简单到复杂的认识过程

类的继承与派生

层次概念是计算机的重要概念。通过继承（inheritance）的机制可对类（class）分层，提供类型/子类型的关系。

C++通过类派生（class derivation）的机制来支持继承。被继承的类称为基类（base class）或超类（superclass），新的类为派生类（derived class）或子类（subclass）。基类和派生类的集合称作类继承层次结构（hierarchy）

类的继承与派生

如果基类和派生类共享相同的公有接口，则派生类被称作基类的子类型（subtype）

派生反映了事物之间的联系，事物的共性与个性之间的关系

派生与独立设计若干相关的类，前者工作量少，重复的部分可以从基类继承来，不需要单独编程。

类的继承与派生

【例如】公司四种雇员档案的管理：

- employee（雇员）
  - 姓名、年龄、工资；
- manager（经理）
  - 姓名、年龄、工资、行政级别；
- engineer（工程师）
  - 姓名、年龄、工资、专业、学位；
- director（高级主管）
  - 姓名、年龄、工资、行政级别、职务。

【例8.1】假设公司雇员分为：雇员(employee)、经理(manager)、工程师(engineer)、高级主管(director)。且假定只关心这几类雇员各自的如下一些数据：

- employee(雇员)类：姓名、年龄、工资；
- manager(经理)类：姓名、年龄、工资、行政级别；
- engineer(工程师)类姓名、年龄、工资、专业、学位；
- director(高级主管)类：姓名、年龄、工资、行政级别、职务。

```
#include <iostream>
#include <string>
using namespace std;
class employee{//employee类将作为其它几个类的基类
      short age;
      float salary;
  protected:
      char * name;
  public:
      employee(short ag,float sa,char * na){
            age=ag;
            salary=sa;
            name=new char[strlen(na)+1];
            strcpy_s(name, strlen(na)+1 ,na);
      };
```

```
    void print (){
         cout<<"    "<<name<<": ";
         cout<<age<<" : ";
         cout<<salary<<endl;
    }
    ~employee()
    {
         delete[]name;
    }
};
```

```
class manager:public employee {//派生类
  int level;
public:
  manager(short ag, float sa, char* na, int
  lev):employee (ag,sa,na) {//对基类初始化负责
      level=lev;
  }
  void print() {
      employee::print();
      //调用基类print显示“共性”数据
      cout <<"     level:"<<level<<endl;
  }
};
```

```
/*注意：允许派生类中的print与基类的print重名，按
  如下规定进行处理：对子类而言，不加类名限定时默认
  为是处理子类成员，而要访问父类重名成员时，则要通
  过类名限定*/
class engineer:public employee {
    char speciality,adegree;
    public:
    ...
};
enum ptitle {PS,GM,VPS,VGM};
class director:public manager {
     ptitle post;
   public:
     ...
};
```

```
void main() {//主函数
  employee emp1(23,610.5,"zhang"),
  emp2(27,824.75,"zhao");
  manager man1(32,812.45,"li",11),
  man2(34,1200.5,"cui",7);
  engineer eng(26,1420.10,"meng",'E','M');
  director dir(38,1800.2,"zhou",2,GM);
  emp1.print();
  emp2.print();
  man1.print();
  man2.employee::print();   //调用基类的print
  eng.print();
  dir.print();
}
```

程序执行后的显示结果如下：

```
zhang: 23 : 610.5
zhao: 27 : 824.75
li: 32 : 812.45
level:11
cui: 34 : 1200.5
meng: 26 : 1420.1
speciality:E
academic degree:M
zhou: 38 : 1800.2
level:2
post:1
```

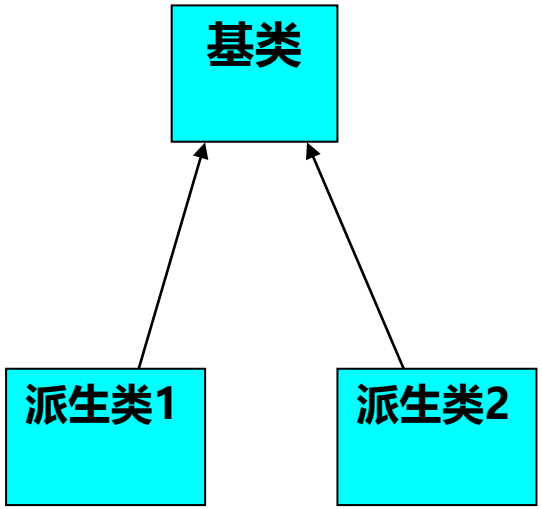
final关键字

在类定义时，使用关键字final限定，则该类不允许任何类继承

```
class Box final
{
protected:
   double length {1.0};
   double width {1.0};
   double height {1.0};
};
```

单继承

派生类只有一个直接基类的情况称为单继承（single-inheritance）。



一个基类可以直接派生出多个派生类

多级继承

在派生过程中，派生出来的新类同样可以作为基类再继续派生出更新的类，依此类推形成一个层次结构。直接参与派生出某类称为直接基类，而基类的基类，以及更深层的基类称为间接基类。

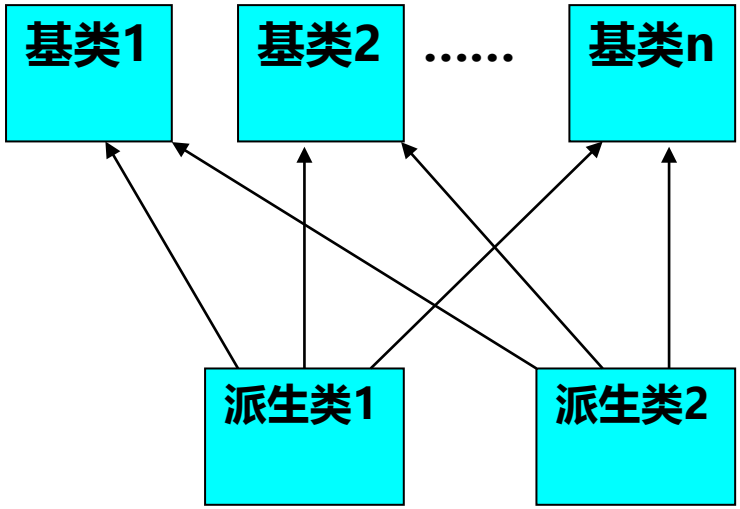
类族

- 同时一个基类可以直接派生出多个派生类。这样形成了一个相互关联的类族。如MFC就是这样的族类，它由一个CObject类派生出200个MFC类中的绝大多数。

多重继承

如果一个派生类可以同时有多个基类，称为多重继承（multiple-inheritance），这时的派生类同时得到了多个已有类的特征。

派生类可以由多个基类共同派生出来，称为多重继承。



派生编程的步骤

编制派生类时可分四步

吸收基类的成员

不论是数据成员，还是函数成员（非私有成员），除构造函数与析构函数外全盘接收

改造基类成员

声明一个和某基类成员同名的新成员,派生类中的新成员就屏蔽了基类同名成员称为同名覆盖（override）

发展新成员

派生类新成员必须与基类成员不同名，它的加入保证派生类在功能上有所发展。

重写构造函数与析构函数

派生编程的步骤

第二步中，新成员如是成员函数，参数表也必须一样，否则是重载。

第三步中，独有的新成员才是继承与派生的核心特征。

第四步是重写构造函数与析构函数，不管基类的构造函数和析构函数是否可用一律重写可免出错

类的继承与派生

派生类

虚基类与虚拟继承

多态性与虚函数

综合示例

派生类的定义

```
class <派生类类型名> : <基类表> {
  private:
      <各私有成员说明>;
  public:
      <各公有成员说明>;
  protected:
      <各保护成员说明>;
  <以关键字friend开头的友元说明>;
};
```

- <基类表>的一般格式为:

<派生方式> <基类名1>, ... , <派生方式> <基类名n>

  - <派生方式>又可为private、 public或protected

派生类的访问权限

派生方式(基类的被继承方式)	在基类中的存取权限	在派生类中的存取权限
public	public	public
public	potected	protected
public	private	(inaccessible)
potected	public	protected
potected	potected	protected
potected	private	(inaccessible)
private	public	private
private	potected	private
private	private	(inaccessible)

派生类的访问权限

public派生方式

- 使基类的公有成员和保护成员在派生类中仍然是公有成员和保护成员，而基类的私有成员不可在派生类中被存取。

protected派生方式

- 使基类的公有成员和保护成员在派生类中都变为保护成员，而基类的私有成员不可在派生类中被存取。

private派生方式

- 使基类的公有成员和保护成员在派生类中都变为私有成员，而基类的私有成员不可在派生类中被存取。

派生类的访问权限

派生类的成员可根据访问权限分为四类

- 不可访问的成员
  - 基类的private私有成员被继承过来后，这些成员在派生类中是不可访问的。
- 私有成员
  - 包括在派生类中新增加的private私有成员以及从基类私有继承过来的某些成员。这些成员在派生类中是可以访问的。

派生类的访问权限

派生类的成员可根据访问权限分为四类

- 保护成员
  - 包括在派生类中新增加的protected保护成员以及从基类继承过来的某些成员。这些成员在派生类中是可以访问的。
- 公有成员
  - 包括在派生类中新增加的public公有成员以及从基类公有继承过来的基类的public成员。这些成员不仅在派生类中可以访问，而且在建立派生类对象的模块中，也可以通过对象来访问它们

【例8.2】读程序，分析运行结果

```
class baseCla {
        int privData;
  protected:
        int protData;
  public:
        int publData;
};
class publDrvCla : public baseCla {
public:
   void usebaseClaData() {
        publData=11;          //OK!
        protData=12;          //OK!
        privData=13;          //ERROR!
   }
};
```

```
class claD21 : public publDrvCla {
public:
   void usebaseClaData() {
        publData=111;       //OK!
        protData=121;       //OK!
        privData=131;       //ERROR!
   }
};
class protDrvCla : protected baseCla {
public:
   void usebaseClaData() {
        publData=21;        //OK!
        protData=22;        //OK!
        privData=23;        //ERROR!
   }
};
```

```
class claD22 : public protDrvCla {
public:
   void usebaseClaData() {
        publData=211;       //OK!
        protData=221;       //OK!
        privData=231;       //ERROR!
   }
};
class claD221 : public claD22 {
public:
   void usebaseClaData() {
        publData=211;       //OK!
        protData=221;       //OK!
        privData=231;       //ERROR!
   }
};
```

```
class privDrvCla : private baseCla {
public:
   void usebaseClaData() {
        publData=31;        //OK!
        protData=32;        //OK!
        privData=33;        //ERROR!
   }
};
class claD23 : public privDrvCla {
public:
   void usebaseClaData() {
        publData=311;       //ERROR!
        protData=321;       //ERROR!
        privData=331;       //ERROR!
   }
};
```

```
void main() {
   baseCla ob0;
   ob0.publData=1;          //OK!
   ob0.protData=2;          //ERROR!
   ob0.privData=3;          //ERROR!
   claD21 d21;
   claD22 d22;
   claD23 d23;
   d21.publData=4;          //OK!
   d21.protData=5;          //ERROR!
   d21.privData=6;          //ERROR!
   d22.publData=7;          //ERROR!
   d22.protData=8;          //ERROR!
   d22.privData=9;          //ERROR!
   d23.publData=7;          //ERROR!
   d23.protData=8;          //ERROR!
   d23.privData=9;          //ERROR!
}
```

派生类对象的说明

派生类对象的说明与所有类对象的说明方式相同

<派生类名> <对象1>,<对象2>,…,<对象n>

类对象进行说明的同时，需要进行初始化

- 初始化列表
- 派生类的构造函数

派生类的构造函数

派生类的构造函数的一般格式如下：

<派生类名>(<参数总表>):<初始化符表>
{
    <构造函数体>
}

- <初始化符表> 按如下格式构成：
  - <基类名1>(<基类参数表1>)， ... ，<基类名n>(<基类参数表n>)，<对象成员名1>(<对象成员参数表1>)， ... ，<对象成员名m>(<对象成员参数表m>)
    - 若无对象成员时，则不出现此后半部分；基类名与对象成员名的次序无关紧要，各自出现的顺序可以任意

派生类的构造函数

派生类构造函数执行的一般次序如下：

- 调用各基类的构造函数，调用顺序为派生继承时的基类声明顺序。
- 若派生类含有对象成员的话，调用各对象成员的构造函数，调用顺序按照派生类中对象成员的声明顺序。
- 执行派生类构造函数的函数体。

派生类的构造函数

派生类构造函数与基类构造函数的联系

- 在派生类构造函数中，只要基类不是使用无参的构造函数或默认构造函数都要显式给出基类名和参数表。
- 如果基类没有定义构造函数，则派生类也可以不定义，全部采用系统给定的默认构造函数。
- 如果基类定义了带有形参表的构造函数时，派生类就应当定义构造函数。

【例8.3】分析程序的运行结果

```
class Box
{
protected:
  double length {1.0};
  double width {1.0};
  double height {1.0};

public:
  // Constructors
  Box(double lv, double wv, double hv)
    : length {lv}, width {wv}, height {hv}
  {
    cout << "Box(double, double, double) called.\n";
  }
```

```
  explicit Box(double side) : Box {side, side, side}
  {
    cout << "Box(double) called.\n";
  }

  // No-arg constructor
  Box() { cout << "Box() called.\n"; }

  // Function to calculate the volume
  double volume() const
  {
    return length*width*height;
  }
};
```

```
class Carton : public Box
{
private:
  string material {"Cardboard"};
public:
  Carton(double lv, double wv, double hv, string mat)
  : Box{lv, wv, hv}, material{mat} {
  cout << "Carton(double,double,double,string) called.\n";
  }
  explicit Carton(string mat) : material{mat}  {
    cout << "Carton(string) called.\n";
  }
  Carton(double side, string mat):Box{side}, material{mat}
  {
    cout << "Carton(double,string) called.\n";
  }
  Carton()  {    cout << "Carton() called.\n";  }
};
```

```
#include <iostream>
#include "Carton.h"  // For the Carton class

int main()
{
  // Create four Carton objects
  Carton carton1;
  cout << endl;
  Carton carton2 {"Thin cardboard"};
  cout << endl;
  Carton carton3 {4.0, 5.0, 6.0, "Plastic"};
  cout << endl;
  Carton carton4 {2.0, "paper"};
  cout << endl;
}
```

程序的运行结果为：

```
Box() called.
Carton() called.

Box() called.
Carton(string) called.

Box(double, double, double) called.
Carton(double,double,double,string) called.

Box(double, double, double) called.
Box(double) called.
Carton(double,string) called.
```

派生类构造函数的进一步讨论

如果把【例8.3】中Carton类的构造函数

```
Carton(double lv, double wv, double hv, string mat)
```

改为：

```
Carton(double lv, double wv, double hv, string mat)
:length{lv}, width{wv}, height{hv}, material{mat}
{
    cout << "Carton(double,double,double,string) called.\n";
}
```

Error:"length"不是类"Carton"的非静态成员或基类

```
Carton(double lv, double wv, double hv, string
mat):material{ mat }
{
    length = lv;
    width = wv;
    height = hv;
    cout << "Carton(double,double,double,string)
called.\n";
}
```

派生类的拷贝构造函数

可以为派生类编写拷贝构造函数，其格式与普通类的拷贝构造函数相同，适用场景也相同

派生类的拷贝构造函数，可以在成员初始化符表位置调用基类的拷贝构造函数，“拷贝”派生类中的基类部分；如果不显式地调用基类的拷贝构造函数，将自动调用基类的无参构造函数（如果有定义）或默认构造函数（如果有效）为派生类创建基类部分

【例8.4】为【例8.3】中的基类Box和派生类Carton分别添加拷贝构造函数

```
// Copy constructor of Box
 Box(const Box& box)
:length{box.length},width{box.width},height{box.height}
  {
    std::cout << "Box copy constructor" << std::endl;
  }

// Copy constructor of Carton
 Carton(const Carton& carton)
 : material {carton.material}
 {
   std::cout << "Carton copy constructor" << std::endl;
 }
```

```
int main()
{
    //Declare and initialize a Carton object
    Carton carton {20.0, 30.0, 40.0, "Glassine
board"};
    cout<<endl;
    Carton cartonCopy {carton};
    //Use copy constructor
    cout<<endl;
    cout<<"Volume of carton is
"<<carton.volume()<<endl;
    cout<<"Volume of cartonCopy is "<<
    cartonCopy.volume() <<endl;
    return 0;
}
```

程序运行结果

```
Box(double, double, double) called.
Carton(double,double,double,string) called.

Box() called.
Carton copy constructor

Volume of carton is 24000
Volume of cartonCopy is 1
```

程序运行结果

```
Box(double, double, double) called.
Carton(double,double,double,string) called.

Box() called.
Carton copy constructor

Volume of carton is 24000
Volume of cartonCopy is 1
```

在调用派生类Carton的时候，先调用了基类Box的无参构造函数

两个派生类对象carton和cartonCopy的volume()函数返回值不同

两个Carton对象的volume()值不同的原因就是cartonCopy中的基类部分，是由基类Box的无参构造函数创建的

```
// No-arg constructor
   Box() { cout << "Box() called.\n"; }
```

这个构造函数创建基类对象时，使用成员变量的默认值，因此cartonCopy的volume()值为1

```
class Box
{
protected:
   double length {1.0};
   double width {1.0};
   double height {1.0};
```

派生类对象的“深”拷贝

定义派生类拷贝构造函数时，显式地调用基类的拷贝构造函数，将派生类的基类部分“深拷贝”给相应的派生类对象。

```
// Copy constructor of Carton
 Carton(const Carton& carton)
 : Box(carton), material {carton.material}
 {
   std::cout << "Carton copy constructor" << std::endl;
 }
```

程序运行结果

```
Box(double, double, double) called.
Carton(double,double,double,string) called.

Box copy constructor
Carton copy constructor

Volume of carton is 24000
Volume of cartonCopy is 24000
```

派生类构造函数的“继承”

派生类中，可以使用**using**关键字，显式地“继承”基类的构造函数（无参构造函数除外），实质上是将基类构造函数当做派生类的构造函数使用，初始化派生类对象的基类部分

```
class Carton : public Box
{
  using Box::Box; // Inherit Box class constructors
private:
  std::string material {"Cardboard"};
public:
  Carton(double lv, double wv, double hv, std::string mat)
      : Box {lv, wv, hv}, material {mat}
 {std::cout <<"Carton(double,double,double,string) called.\n";}
};
```

```
int main()
{
  Carton cart; //error!
// Does not compile: default constructor is not inherited!
  Carton cube{4.0};
  // Calls inherited constructor Box(double)
  cout<< cube.volume()<<endl<<endl;
  Carton carton {1.0, 2.0, 3.0};
  // Calls inherited constructor Box(double,double,double)
  cout<< carton.volume()<<endl<<endl;
  Carton candyCarton (50.0, 30.0, 20.0, "Thin cardboard");
  // Calls Carton class constructor
  cout<<candyCarton.volume()<<endl;
}
```

程序运行结果

```
Box(double, double, double) called.
Box(double) called.
64

Box(double, double, double) called.
6

Box(double, double, double) called.
Carton(double,double,double,string)
called.
30000
```

派生类的析构函数

析构函数的功能是做善后工作。

- 只要在函数体内把派生类新增的一般成员处理好就可以了，而对新增的成员对象和基类的善后工作，系统会自己调用对象成员和基类的析构函数来完成。

析构函数各部分执行次序与构造函数相反

- 首先对派生类新增一般成员析构，然后对新增对象成员析构，最后对基类成员析构

【例8.5】读程序，分析运行结果

```
#include <iostream>
using namespace std;
class CB{
  int b;
public:
  CB(int n){    b=n;
        cout<<"CB::b="<<b<<endl; };
  ~CB(){cout<<"CBobj is destructing"<<endl;};
};
class CC{
    int c;
public:
  CC(int n1,int n2){ c=n1;
            cout<<"CC::c="<<c<<endl; };
  ~CC(){cout<<"CCobj is destructing"<<endl;};
};
```

```
class CD:public CB,public CC{
    int d;
public:
    CD(int n1,int n2,int n3,int n4)
       :CC(n3,n4),CB(n2){   //先CB,后CC
      d=n1;      cout<<"CD::d="<<d<<endl;
    }
    ~CD(){
     cout<<"CDobj is destructing"<<endl;
    }
};
void main(void){
      CD CDobj(2,4,6,8);
}
```

运行结果为：

```
CB::b=4
CC::c=6
CD::d=2
CDobj is destructing
CCobj is destructing
CBobj is destrcting
```

【思考】将派生类CD改写为如下形式后，请给出输出结果

```
class CD:public CB,public CC {
    int d;
    CC obcc;
    CB obcb;
public:
    CD(int n1,int n2,int n3,int n4)
  :CC(n3,n4), CB(n2), obcb(100+n2),
  obcc(100+n3,100+n4) {
      d=n1;      cout<<"CD::d="<<d<<endl;
    };
    ~CD(){cout<<"CDobj is destructing"<<endl;};
};  //先基类CB、CC，再对象成员CC、CB，最后派生类CD
```

输出结果：

```
CB::b=4
CC::c=6
CC::c=106
CB::b=104
CD::d=2
CDobj is destructing
CBobj is destructing
CCobj is destructing
CCobj is destructing
CBobj is destructing
```

友元的继承

基类的友元不继承

- 如果基类有友元类或友元函数，则其派生类不因继承关系也有此友元类或友元函数。

如果基类是某类的友元类，则这种友元关系是被继承的。即，被派生类继承过来的成员，如果原来是某类的友元，那么它作为派生类的成员仍然是某类的友元。总之：

- 基类的友元不一定是派生类的友元；
- 基类的成员是某类的友元，则其作为派生类继承的成员仍是某类的友元。

静态成员的继承

如果基类中被派生类继承的成员是静态成员，则其静态属性也随静态成员被继承。

如果基类的静态成员是公有的或是保护的，则它们被其派生类继承为派生类的静态成员。即：

- 这些成员通常用“<类名>::<成员名>”方式引用或调用。
- 这些成员无论有多少个对象被创建，都只有一个拷贝。它为基类和派生类的所有对象所共享。

赋值兼容性问题

派生类对象间的赋值操作依据下面的原则：

- 如果派生类有自己的赋值运算符的重载定义，即按重载后的运算符含义处理。
- 派生类未定义自己的赋值操作，而基类定义了赋值操作，则系统自动定义派生类赋值操作（按位拷贝），其中基类成员的赋值按基类的赋值操作进行。
- 二者都未定义专门的赋值操作，系统自动定义缺省赋值操作（按位进行拷贝）。

赋值兼容性问题

基类对象和派生类对象之间允许有下述的赋值关系（允许将派生类对象“当作”基类对象来使用）：

- 基类对象 = 派生类对象；
  - 只赋“共性成员”部分，反方向的赋值“派生类对象 = 基类对象”不被允许
- 指向基类对象的指针 = 派生类对象的地址；
  - 下述赋值不允许：指向派生类类型的指针 = 基类对象的地址。注：访问非基类成员部分时，要经过指针类型的强制转换
- 基类的引用 = 派生类对象；
  - 下述赋值不允许：派生类的引用 = 基类对象。
  - 通过引用只可以访问基类成员部分

【例8.6】读程序，分析运行结果

```
#include <iostream>
using namespace std;
class base{ //基类base
    int a;
public:
    base (int sa) {a=sa;}
    int geta(){return a;} };
class derived:public base { //派生类derived
    int b;
public:
    derived(int sa, int sb):base(sa) {b=sb;}
    int getb(){return b;}
};
```

```
void main () {
    base bs1(123);              // base 类对象bs1
    cout<<"bs1.geta()="<<bs1.geta()<<endl;
    derived der(246,468);// derived 类对象der
    bs1=der;  //OK! “基类对象 = 派生类对象;”
    cout<<"bs1.geta()="<<bs1.geta()<<endl;
    //der=bs1; //ERROR! “派生类对象 = 基类对象;”
```

```
    base *pb = &der;
    //“指向基类型的指针 = 派生类对象的地址;”
    cout<<"pb->geta()="<<pb->geta()<<endl;
    //访问基类成员部分
    //cout<<pb->getb()<<endl;
    //ERROR! 直接访问非基类成员部分
    cout<<"((derived *)pb)-
    >getb()="<<((derived *)pb)->getb()<<endl;
    //访问非基类成员部分时，要经过指针类型的强制转换
    //derived *pd = &bs1;
    //ERROR! “指向派生类类型的指针=基类对象的地址;”
}
```

程序执行后的显示结果如下：

```
bs1.geta()=123
bs1.geta()=246
pb->geta()=246
((derived *)pb)->getb()=468
```

练习8.1

设计日期类和时间类，并以此两类为基类派生日期时间类

- 要求：
  - 日期类包括年、月、日等成员
  - 时间类包括时、分、秒等成员
  - 日期时间类能够实现日期时间的求差、比较大小的功能
    - 使用运算符重载实现该功能，以时间日期类的对象描述日期时间之差
    - 考虑闰年的情况
    - 不考虑公元前

类的继承与派生

派生类

虚基类与虚拟继承

多态性与虚函数

综合示例

二义性问题

单继承时父类与子类间重名成员的处理

- 单继承时父类与子类间成员重名时，按如下规定进行处理：对子类而言，不加类名限定时默认为是处理子类成员，而要访问父类重名成员时，则要通过类名限定。

【例8.7】读程序，分析运行结果

```
#include <iostream>
using namespace std;
class CB {
public:
   int a;     CB(int x){a=x;}
   void showa(){
       cout<<"Class CB -- a="<<a<<endl;
   }
};
```

```
class CD:public CB {
public:
  int a;                    //与基类a同名
  CD(int x, int y):CB(x){a=y;}
  void showa(){             //与基类showa同名
      cout<<"Class CD -- a="<<a<<endl;
  }
  void print2a() {
      cout<<"a="<<a<<endl;                  //子类a
      cout<<"CB::a="<<CB::a<<endl;          //父类a
  }
};
```

```
void main() {
  CB CBobj(12);
  CBobj.showa();
  CD CDobj(48, 999);
  CDobj.showa();              //子类的showa
  CDobj.CB::showa();          //父类的showa
  cout<<"CDobj.a="<<CDobj.a<<endl;
  cout<<"CDobj.CB::a="<<CDobj.CB::a<<endl;
}
```

程序执行后的显示结果如下：

```
Class CB -- a=12
Class CD -- a=999
Class CB -- a=48
CDobj.a=999
CDobj.CB::a=48
```

二义性问题

多重继承情况下二基类间重名成员的处理

- 多重继承情况下二基类间成员重名时，按如下方式进行处理：对子类而言，不加类名限定时默认为是处理子类成员，而要访问父类重名成员时，则要通过类名限定。

【例8.8】读程序，分析运行结果

```
#include <iostream>
using namespace std;
class CB1 {
public:
   int a; CB1(int x){a=x;}
   void showa(){
       cout<<"Class CB1 ==> a="<<a<<endl;}
};
```

```
class CB2 {
 public:
  int a;
  CB2(int x){a=x;}
  void showa(){
      cout<<"Class CB2 ==> a="<<a<<endl;
  }
};
class CD:public CB1,  public CB2 {
public:
  int a;              //与二基类数据成员a同名
  CD(int  x,  int  y,  int  z):  CB1(x),  CB2(y)
  {a=z;}
```

```
    void showa(){  //与二基类成员函数showa同名
        cout<<"Class CD ==> a="<<a<<endl;
    }
    void print3a() {
        //显示出派生类的a及其二父类的重名成员a
        cout<<"a="<<a<<endl;
        cout<<"CB1::a="<<CB1::a<<endl;
        cout<<"CB2::a="<<CB2::a<<endl;
    }
};
```

```
void main() {
  CB1 CB1obj(11);
  CB1obj.showa();
  CD CDobj(101, 202,909);
  CDobj.showa();                    //子类showa
  CDobj.CB1::showa();               //父类showa
  cout<<"CDobj.a="<<CDobj.a<<endl;
  cout<<"CDobj.CB2::a="<<CDobj.CB2::a<<endl;
}
```

程序执行后的显示结果如下：

```
Class CB1 ==> a=11
Class CD ==> a=909
Class CB1 ==> a=101
CDobj.a=909
CDobj.CB2::a=202
```

二义性问题

多级混合继承(非虚拟继承)包含两个基类实例情况的处理

- 多级混合继承情况下，若类D从两条不同“路径”同时对类A进行了一般性继承（非虚拟继承）的话，则类D的对象中会同时包含着两个类A的实例。此时，对类D而言，要通过类名限定来指定访问两个类A实例中的哪一个。
- 【例8.9】读程序，分析运行结果

- 本例的类间继承关系示例如下:

```
class A
class B : public A
class C : public A
class D : public B, public C
```

- 存储结构示意:
  - (((A) B) ((A) C) D)
- 上述多级混合继承关系应用例举:
  - 例1.类A--人员类； 类B--学生类； 类C--助教类； 类D--学生助教类。
  - 例2.类A--人员类； 类B--学生类； 类C--工人类； 类D--工人学生类。
  - 例3.类A--家具类； 类B--沙发类； 类C--床类； 类D--沙发床类。

```
#include <iostream>
using namespace std;
class A {
public:
  int a;
  A(int x){a=x;}
  void showall(){cout<<"a="<<a<<endl;}
};
class B:public A {
public:
  int b;
  B(int x):A(x-1){b=x;}
};
```

```
class C:public A {
public:
   int c;
   C(int x):A(x-1){c=x;}
};
```

```
class D:public B,public C {
public:
   int d;
   D(int x, int y, int z):B(x+1),C(y+2){d=z;}
   void showall() {
       cout<<"C::a="<<C::a<<endl;
       //在类D定义范围内，要通过类名限定来指定
       //访问两个类A实例中的哪一个
       cout<<"B::a="<<B::a<<endl;
       cout<<"b,c,d="<<b<<", "<<c<<", "<<d<<endl;
       //b、c、d不重名，具有唯一性
   }
};
```

```
void main() {
  D Dobj(101, 202, 909);
  Dobj.showall();
  cout<<"--------------------"<<endl;
  cout<<"Dobj.C::a="<<Dobj.C::a<<endl;
     //访问类D的从C继承而来的a
  cout<<"Dobj.B::a="<<Dobj.B::a<<endl;
}
```

程序执行后的显示结果如下：

```
C::a=203
B::a=101
b,c,d=102, 204, 909
--------------------
Dobj.C::a=203
Dobj.B::a=101
```

虚基类与虚拟继承

派生类中可能包含了多个基类的实例。

- 多级混合继承情况下，若类D从两条不同“路径”同时对类A进行了虚拟继承的话，则类D的对象中只包含着类A的一个实例，这种继承也称为共享继承。被虚拟继承的基类A被称为虚基类（注意，虚基类的说明是在定义派生类时靠增加关键字virtual来指出的）。
- 说明格式：

```
class <派生类名> : virtual <派生方式> <基类名>
{ <派生类体> };
```

虚基类

【例8.9】采用虚拟继承后，

- 类间继承关系如下所示：

```
class A
class B : virtual public A
class C : virtual public A
class D : public B, public C
```

- 存储结构示意：

```
( ( (A) B C ) D )
```

- 系统进行“干预”，在派生类中只生成公共基类A的一个拷贝，从而可用于解决二义性问题

【例8.10】读程序，分析运行结果

```
#include <iostream>
using namespace std;
class A {
public:
   int a;
   void showa(){cout<<"a="<<a<<endl;}
};
class B: virtual public A//对类A进行了虚拟继承
{
public:
   int b;
};
```

```
class C: virtual public A//对类A进行了虚拟继承
{
public:
   int c;
};
class D : public B, public C
//派生类D的二基类B、C具有共同的基类A，但采用了虚
//继承，从而使类D的对象中只包含着类A的1个实例
{
public:
   int d;
};
```

```
void main() {
   D Dobj;          //说明D类对象
   Dobj.a=11;
   //若非虚拟继承时会出错!
   // -- 因为“D::a”具有二义性
   Dobj.b=22;
   Dobj.showa();
   //若非虚拟继承时会出错!
   // -- 因为“D::showa”具有二义性
   cout<<"Dobj.b="<<Dobj.b<<endl;
}
```

程序执行后的显示结果如下：

```
a=11
Dobj.b=22
```

类的继承与派生

派生类

虚基类与虚拟继承

多态性与虚函数

综合示例

多态性与虚函数

函数重载

函数重载（overloading）指的是，允许多个不同函数使用同一个函数名，但要求这些同名函数具有不同的参数表。

- 参数表中的参数个数不同;
- 参数表中对应的参数类型不同;
- 参数表中不同类型参数的次序不同。
- 【例如】

```
int abs(int n){
  return (n<0?-n:n);
}
```

```
float abs(float n){
      if (f<0) f=-f;
      return f;
}
```

静态联编

系统对函数重载这种多态性的分辨与处理，是在编译阶段完成的 -- 静态联编(static binding)

函数超载

函数超载(overriding)

- 仅在基类与其派生类的范围内实现；
- 允许多个不同函数使用完全相同的函数名、函数参数表以及函数返回类型；

动态联编(dynamic binding)

动态联编与虚函数以及程序中使用指向基类的指针密切相关。

C++规定，基类指针可以指向其派生类的对象（也即，可将派生类对象的地址赋给其基类指针变量），但反过来不可以。这一点正是函数超载及虚函数用法的基础。

虚函数

虚函数(virtual function)

- 在定义某一基类(或其派生类)时，若将其中的某一函数成员的属性说明为virtual，则称该函数为虚函数(virtual function)。
- 虚函数的使用与函数超载密切相关。若基类中某函数被说明为虚函数，则意味着其派生类中也要用到与该函数同名、同参数表、同返回类型、但函数(实现)体不同的这同一个所谓的超载函数。
- 在基类中定义虚函数，其派生类的同原型函数默认为虚函数，可省略virtual关键字。只在派生类中定义虚函数没有意义

【例8.11】虚函数示例

```
class graphelem {
protected:
   int color;
public:
   graphelem(int col) {
       color=col;
   }
   virtual void draw(){ ... };
   /*虚函数draw，每一个类都要“draw”出属于它的类对象图形*/
};
```

```
class line:public graphelem {
  public:
  virtual void draw(){ ... };
  //虚函数draw，负责画出“line”
  ...
};
class circle:public graphelem{
  public:
  virtual void draw(){ ... };
  //虚函数draw，负责画出“circle”
  ...
};
```

```
class triangle:public graphelem{
  public:
  virtual void draw(){ ... };
  //虚函数draw，负责画出“triangle”
  ...
};
```

【例8.12】建立【例8.11】定义的类line、类circle以及类triangle的类对象，而后调用它们各自的draw函数“画出”它们

方法1：直接通过类对象(由类对象可以唯一确定要调用哪一个类的draw函数)

```
line ln1;
circle cir1;
triangle tri1;
ln1.draw();
cir1.draw();
tri1.draw();
```

【例8.12】方法2：使用指向基类的指针（动态联编，要靠执行程序时其基类指针的“动态”取值来确定调用哪一个类的draw函数）

```
graphelem *pObj;
line ln1;
circle cir1;
triangle tri1;
pObj=&lin1;     pObj->draw();
pObj=&cir1;     pObj->draw();
pObj=&tri1;     pObj->draw();
```

【例8.13】假设inte_algo为基类，其中说明了一个虚函数integrate（用来计算定积分），并在其三个派生类中，也说明了该虚函数integrate

- 可使用函数integrateFunc来实现调用不同虚函数integrate的目的

```
void integrateFunc(inte_algo * p){
    //基类指针p可指向任一派生类的对象
    p->integrate();
    //调用的将是不同派生类的integrate函数
}
```

【例8.13】主调函数处使用：

integrateFunc(<某派生类的类对象地址>);

- 根据不同的派生类对象，调用各自派生类的成员函数
- 在编译阶段，系统无法确定究竟要调用哪一个派生类的integrate。此种情况下，将采用动态联编方式来处理：在运行阶段，通过p指针的当前值，去动态地确定对象所属类，而后找到对应虚函数。

设置

以下程序的执行结果是：
[填空1]
[填空2]
[填空3]

```
#include <iostream>
using namespace std;
class base
{
public:
        virtual void
who(){cout<<"base
class"<<endl;}
};
class derive1:public base
{
public:
        void
who(){cout<<"derive1
class"<<endl;}
};
class derive2:public base
{
public:
        void who(){cout<<"derive2
class"<<endl;}
};
int main()
{
        base obj1, *p;
        derive1 obj2;
        derive2 obj3;
        p=&obj1;
        p->who();
        p=&obj2;
        p->who();
        p=&obj3;
        p->who();
        return 0;
}
```

作答

纯虚函数

如果不准备在基类的虚函数中做任何事情，则可使用如下的格式将该虚函数说明成纯虚函数：

```
virtual <函数原型> = 0;
```

纯虚函数不能被直接调用，它只为其派生类的各虚函数规定了一个一致的“原型规格”，该虚函数的实现将在它的派生类中给出。

抽象基类

含有纯虚函数的基类称为抽象基类。

- 不可使用抽象基类来说明并创建它自己的对象，只有在创建其派生类对象时，才有抽象基类自身的实例伴随而生。
- 抽象基类是其各派生类之共同点的一个抽象综合，通过它，再“加上”各派生类的特有成员以及对基类中那一纯虚函数的具体实现，方可构成一个具体的实用类型。
- 如果一个抽象基类的派生类中没有定义基类中的那一纯虚函数、而只是继承了基类之纯虚函数的话，则这个派生类还是一个抽象基类（其中仍包含着继承而来的那一个纯虚函数）。

练习8.2

设计圆类，并以圆类为基类，派生圆柱类、圆锥类和圆球类（分别求出其面积和体积）

- 要求：
  - 自行确定各类具有的数据成员、函数成员，如果需要对象成员，再自行设计相关类；
  - 在设计程序过程中，尽量多地涉及类继承与多态性的重要概念，如虚函数、纯虚函数、抽象基类等等。


类的继承与派生


派生类


虚基类与虚拟继承


多态性与虚函数


综合示例

【例8.14】计算函数的定积分

- 采用下列方法来计算同一函数的定积分
  - 矩形法
  - 梯形法
  - simpson法
- 此三种方法均将区间[a,b]分为n等份，而后以不同方式求出各小段对应的小面积s[i]，并将它们相加到一起来作为近似结果

```
#include <iostream>
using namespace std;
float function(float x){   //欲积分的函数
  return 4.0/(1+x*x);
}
class inte_algo {
protected:
  float a,b;//a,b为积分区间的左右边界
  int n;//n表示把[a,b]划分成多少个小区段积分
  float h,sum; //h表示步长，sum表示积分结果值
public:
  inte_algo (float left,float right,int
  steps){        ...   }
  virtual void integrate(void);//虚函数
};
```

```
class rectangle:public inte_algo {
public:
   rectangle(float left,float right,int steps)
   :inte_algo (left,right,steps){}
   virtual void integrate(void);
   //派生类中说明同一个虚函数integrate
};
class ladder:public inte_algo {...};
class simpson:public inte_algo {...};
void inte_algo::integrate(){
   ...
}
```

```
void rectangle::integrate(){
  /*派生类rectangle之虚函数integrate的类外定义，采
  用矩形法来计算函数的定积分。 计算公式为:
sum=(f(a)+f(a+h)+f(a+2h)+...+f(a+(n-1)h))h */
  float al=a;
  /*al为调用f函数时的实参值，依次取值 a，a+h，a+2h
  ，... ，a+(n-1)h。*/
  for(int i=0; i<n; i++){//共在n个点处计算函数值
      sum+=function(al);
      al+=h;          //al每次增加一个步长h
  };
  sum*=h;
  cout<<sum<<endl;      //显示积分结果sum
}
```

```
void ladder::integrate(){        //梯形法
   ...
}

void simpson::integrate(){       //simpson法
   ...
}

void integrateFunc(inte_algo * p) {
  p->integrate();
}
```

```
void main(){
  rectangle rec(0.0, 1.0, 10);
  ladder lad(0.0, 1.0, 10);
  simpson sim(0.0, 1.0, 10);
  inte_algo *p;
  cout<<"input a num (1--rectangle method,
  2--ladder method, 3--simpson method)";
  int ii;
  cin>>ii;
  switch (ii) {
  case 1:                    //矩形法
     cout<<"rectangle method:  suum==>";
     integrateFunc(&rec);
     break;
```

```
    case 2:                    //梯形法
        cout<<"ladder method:   suum==>";
        integrateFunc(&lad);
        break;
    case 3:                    //simpson法
        cout<<"simpson method:   suum==>";
        integrateFunc(&sim);
        break;
    }
}
```

程序执行后的显示结果如下：

```
input a num (1--rectangle method, 2--ladder
  method, 3--simpson method)3
simpson method:   suum==>3.14159
```

也可将上述的integrateFunc函数以及main用如下样式的一个main函数来替代。

```
void main(){
  rectangle rec(0.0, 1.0, 10);
  ...
  cin>>ii;
  switch (ii) {
  case 1: //矩形法
     cout<<"rectangle method:  suum==>";
     p=&rec;   break;
  case 2: ...     //梯形法
  case 3: ...    //simpson法
  }
  p->integrate();
}
```

若只为实现上述功能，完全可以不用指针，而直接通过类对象来进行调用(此时将不再通过动态联编)。如，也可使用如下形式的main

```
void main(){
  rectangle rec(0.0, 1.0, 10);
  ...
  cin>>ii;
  switch (ii) {
  case 1:  //矩形法
     cout<<"rectangle method:  suum==>";
     rec.integrate();     break;
  case 2:  //梯形法 ;case 3:    //simpson法
  }
}
```

【例8.15】利用图元类画图

- 本程序自定义pixel、graphelem、line、rectangle、triangle、circle、square、figure等8个类（类型）并对它们进行使用
- 设立并处理以下图元:
  - 直线(line类)，矩形(rectangle类)，三角形(triangle类)，圆(circle类)，正方形(square类)。
- 将每一种图元设计成一个类，在每一个类的定义中，除含有其构造函数外，还包含一个可将本类的图元画出来的公有函数draw

由于每一个图元都要用到颜色(color)数据成员，所以设立一个基类graphelem，它含有protected型数据成员color，以及一个虚函数draw。
由于不准备在基类graphelem的虚函数draw中做任何事情，所以在其原型后加上“=0”字样而构成纯虚函数。从而使graphelem成为抽象基类。
程序中至少要设立具有以下关系的六个类：

- 抽象基类graphelem；
- 由graphelem直接派生出的四个类：line，rectangle，triangle，circle；
- 由rectangle派生出一个类：square。

由于“画”以上图元时，都要用到“点”的概念与位置，所以设立的第七个类为：pixel
为了通过虚函数进行动态联编处理，需说明并使用指向基类graphelem的指针(注，该程序中说明了10个这种指针，放在一个称为pg的数组中，既是说，pg[0]，pg[1]，... ，pg[9]均为这种指向基类graphelem的指针)。而后通过这些指针的动态取值(使它们指向不同的派生类)，进而利用函数调用“pg[i]->draw( )”(i=0， 1，... ，9)来“画”出组成一个图形的不同图元来。

本程序中的第八个类figure 中的paint成员函数正是用来完成上述“画”图元功能的。

虽然pixel，figure这两个类与其它六个类没有继承和派生的关系，但却有成员关系。类pixel的对象要作为graphelem及其派生类的成员和构造函数成员的参数。而类figure则以graphelem类的对象指针数组作为其数据成员

```
#include <iostream>
using namespace std;
class pixel {//类pixel，表示屏幕像素点
  int x,y;
public:
  pixel(){  x=0; y=0; }//构造函数一，无参
  pixel(int a, int b){ x=a; y=b;}
  //构造函数二，参数a、b表示点的位置
  pixel(const pixel& p){ x=p.x;  y=p.y; }
  //构造函数三，参数p为某个已存在对象
  int getx(){return x;}   //获取对象的x值
  int gety(){return y;}   //获取对象的y值
};
```

```
enum colort{//枚举类型，用于定义颜色常量（名字）
  black,blue,green,cyan,red,magenta,brown,
  lightgray,darkgray,lightblue,lightgreen,
  lightcyan,lightred,lightmagenta,yellow,
  white,blink
};
class graphelem{//基类graphelem（为抽象基类）
protected:
  colort color;      //颜色color
public:
  graphelem(colort col){ color=col; }
  virtual void draw( )=0; //纯虚函数draw
};
```

```
class line:public graphelem{   //派生类line
  pixel start, end; //成员为pixel类对象
public:
  line(pixel sta, pixel en, colort col)
  :graphelem(col),start(sta),end(en) { };
  virtual void draw( );          //虚函数draw
};
class rectangle:public graphelem{
  //派生类rectangle
  pixel ulcorner,lrcorner;
public:
  rectangle(pixel ul,pixel el,colort col)
  :graphelem(col),ulcorner(ul),lrcorner(el){}
  ;
  virtual void draw( );          //虚函数draw
};
```

```
class square:public rectangle{//派生类square
public:
  square(pixel ul,int lh,colort col)
  :rectangle(ul,pixel(ul.getx()+lh,ul.gety()
  +lh),col){};
  virtual void draw( );              //虚函数draw
};
```

```
class figure{//通过它的paint函数可画出组成一个图形的各图元
   graphelem *pg[10]; //pg数组含有10个指向基类的指针
public:
   figure(   graphelem *pg1=0, graphelem *pg2=0,
             graphelem *pg3=0, graphelem *pg4=0,
             graphelem *pg5=0, graphelem *pg6=0,
             graphelem *pg7=0, graphelem *pg8=0,
             graphelem *pg9=0, graphelem *pg10=0   ) {
      //具有参数默认值的构造函数，是抽象基类的指针
      pg[0]=pg1;  pg[1]=pg2;  pg[2]=pg3;  pg[3]=pg4;
      pg[4]=pg5;  pg[5]=pg6;  pg[6]=pg7;  pg[7]=pg8;
      pg[8]=pg9;  pg[9]=pg10;}
   void paint( ){                    //画出图形的各图元
      for(int i=0;i<10;i++)
        if(pg[i]!=0)
            pg[i]->draw( ); }
};
```

```
void main( ){
   //说明9个类对象（图元），它们是构成一个图形的九个“部件”
   square sq(pixel(40,40),120,black);     //正方
   circle ce1(pixel(100,100),50,green);     //圆
   circle ce2(pixel(100,100),2,blue);
   triangle
   tr1(pixel(100,62),pixel(98,97),pixel(102,97),blue);   //三角
triangle
   tr2(pixel(98,103),pixel(102,103),pixel(100,130),blue);
   rectangle re1(pixel(98,54),pixel(102,62),red )//长方
   rectangle re2(pixel(98,138),pixel(102,146),red);
   rectangle re3(pixel(54,98),pixel(62,102),red);
   rectangle re4(pixel(138,98),pixel(146,102),red);
   figure fig(&sq,&ce1,&ce2,&re1,&re2,&re3,&re4,&tr1,&tr2);
   //图形，用派生类对象地址初始化抽象基类指针
   fig.paint( );          //调用paint函数，“画”出fig对象的9个图元
}
```

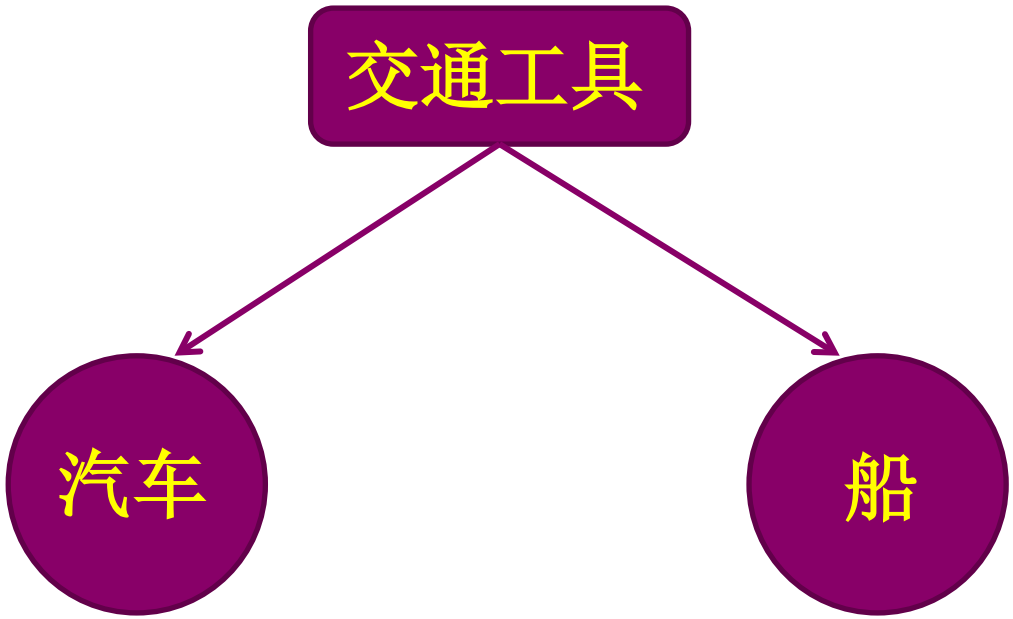
【例8.15】说明

- figure类的构造函数，含有10个参数，且该10个参数均为可缺省参数（被赋了缺省值的参数，调用时，相应实参可缺省）。说明figure类对象时，其实参可为10个（或少于10个）指向graphelem不同派生类对象的具体指针（注意，C++允许基类指针指向其派生类对象；而此处的各派生类对象正是准备“画”出的那些不同图元）。注意如下语法：函数定义处，若有可缺省参数的话，必须放于参数表的“最右边”，且要连续出现。

【例8.15】说明

- paint成员函数中，由于pg[i]为各派生类对象的地址（由构造函数的各实参带来），从而使随后的i循环可“画”出所设计图形的各图元（图元数不多于10）。
- main函数中，说明了一个由九个图元构成的figure类对象fig，这九个图元由调用构造函数时带去的那九个实参所确定，各实参均为指向graphelem不同派生类对象的具体指针（也即，对象地址）
- 这个程序尚不完整，缺少五个派生类中虚函数draw的具体实现代码。随C++编译版本的不同，draw函数的具体编制方式可能不同，而这方面的内容并不属于C++语言的基本规则范围之内

【例8.16】虚函数、动态联编、纯虚函数与抽象基类综合示例



- 交通工具类

```
class Vehicle  //交通工具，基类
{
public:
     Vehicle(int w) {
          weight = w;
     }
     virtual void ShowMe(){
        cout<<"我是交通工具！重量为"<<weight<<"吨"<<endl;
    }
protected:
          int weight;
};
```

- 汽车类

```
class Car: public Vehicle//汽车，派生类
{
    public:
        Car(int w,int a):Vehicle(w)   {
            aird = a;
        }
        virtual void ShowMe()   {//virtual可省略
            cout<<"我是汽车！排气量为
  "<<aird<<"CC"<<endl;
        }
    protected:
        int aird;
};
```

- 船类

```
class Boat: public Vehicle//船，派生类
{
    public:
        Boat(int w,float t):Vehicle(w) {
            tonnage = t;
        }
        virtual void ShowMe()   {//virtual可省略
            cout<<"我是船！排水量为"<<tonnage<<"顿
  "<<endl;
        }
    protected:
        float tonnage;
};
```

```
#include <iostream>
using namespace std;
int main(){
  Vehicle *pv = new Vehicle(10);
  pv ->ShowMe();
  Car c(15,200);
  Boat b(20,1.25f);
  pv = &c;
  pv->ShowMe();//基类指针，实现动态联编
  Vehicle v (10);//创建一个基类对象
  v = b;          //将派生类对象赋值给基类对象
  v.ShowMe();   //基类对象访问虚函数，未实现动态联编
  pv = &b;
  pv -> ShowMe();//基类指针，实现动态联编
  return 0;
}
```

- 将ShowMe()函数改为纯虚函数

```
class Vehicle  //交通工具，基类
{
    public:
        Vehicle(int w) {
            weight = w;
        }
        virtual void ShowMe() = 0;
    //纯虚函数，Vehicle类成为抽象基类
    //不能够创建Vehicle类的对象——体现“抽象”
     protected:
        int weight;
};
```

```
#include <iostream>
using namespace std;
int main(){
  Vehicle *pv = new Vehicle(10)//ERROR
  pv ->ShowMe();//ERROR纯虚函数不能直接调用
  Car c(15,200);
  Boat b(20,1.25f);
  Vehicle * pv = &c;//用派生类对象进行初始化
  pv->ShowMe();//基类指针，实现动态联编
  pv = &b;
  pv -> ShowMe();//基类指针，实现动态联编
  return 0;
}
```

第八章 结束