面向对象分析与设计

第三章 关于类和对象的进一步讨论

3.1 构造函数

3.1.1 对象的初始化

- 类的数据成员是不能在**声明**类时初始化

```
class Time
{
    hour=0;//错误
    minute=0;
    sec=0;
};
```

- 因为类不是一个实体，是一种抽象类型，不占存储空间，所以不能容纳数据
- 如果一个类中所有的成员都是公用的，则可以在定义对象时对数据成员进行初始化。

```
class Time
{
    public://声明为公用成员
        hour;
        minute;
        sec;
};
Time t1={14,56,30};//将t1初始化为14:56:30
```

3.1.2 构造函数的作用

- C++提供了**构造函数**(constructor)来处理对象的初始化。
- 构造函数是一种**特殊的成员函数**，不需要用户来调用它，而是在建立对象时**自动执行。**
- 构造函数的名字必须与**类名同名**，而不能由用户任意命名。它**不具有任何类型，不返回任何值。**

```
//例3.1
//在例2.3基础上定义构造成员函数。

#include <iostream>
using namespace std;
class Time
{
    public:
        Time( )//定义构造成员函数，函数名与类名相同
        {
            hour=0;//利用构造函数对对象中的数据成员赋初值
            minute=0;
            sec=0;
        }
         void set_time( );//函数声明
```

```
        void show_time( );//函数声明
    private:
        int hour;//私有数据成员
        int minute;
        int sec;
};
void Time::set_time( )//定义成员函数，向数据成员赋值
{
    cin>>hour;
    cin>>minute;
    cin>>sec;
}
void Time::show_time( )//定义成员函数，输出数据成员的值
{
    cout<<hour<<":"<<minute<<":"<<sec<<endl;
}
int main( )
{
    Time t1;//建立对象t1，同时调用构造函数t1.Time( )
    t1.set_time( );//对t1的数据成员赋值
    t1.show_time( );//显示t1的数据成员的值
    Time t2;//建立对象t2，同时调用构造函数t2.Time( )
    t2.show_time( );//显示t2的数据成员的
    return 0;
}
```

在类外定义构造函数:

```
Time::Time( )
{
    hour=0;
    minute=0;
    sec=0;
}
```

- 在类对象进入其作用域时调用构造函数。
- 构造函数没有返回值，因此也不需要在定义构造函数时声明类型。
- 构造函数不需用户调用，也不能被用户调用。
- 在构造函数的函数体中不仅可以对数据成员赋初值，而且可以包含其他语句。
- 如果用户自己没有定义构造函数，则C++系统会自动生成一个构造函数，该构造函数的函数体是空的，也没有参数，不执行初始化操作。

3.1.3 带参数的构造函数

- 带参数的构造函数，在调用不同对象的构造函数时，从外面将不同的数据传递给构造函数，以实现不同的初始化。
- 构造函数首部的一般格式为

  构造函数名(类型 1 形参1，类型2 形参2，...)

  实参是在定义对象时给出的。

  定义对象的一般格式为

  类名 对象名(实参1，实参2，...);

```
//例3.2
//有两个长方柱，其长、宽、高分别为：  (1)12,20,25;(2)10,14,20。求它们的体积。
```

```
#include <iostream>
using namespace std;
class Box
{
    public:
        Box(int,int,int);//声明带参数的构造函数
        int volume( );//声明计算体积的函数
    private:
        int height;
        int width;
        int length;
};
Box::Box(int h,int w,int len)//在类外定义带参数的构造函数
{
    height=h;
    width=w;
    length=len;
}
int Box::volume( )//定义计算体积的函数
{
     return(height*width*length);
}

int main( )
{
      Box box1(12,25,30);//建立对象box1
      cout<<"The volume of box1 is "<<box1.volume( )<<endl;
      Box box2(15,30,21);//建立对象box2
      cout<<"The volume of box2 is "<<box2.volume( )<<endl;
      return 0;
}
```

3.1.4 用参数初始化表对数据成员初始化

- 不在函数体内对数据成员初始化，而是在函数首部实现。
- 一般格式为
  Box::Box(int h,int w,int len): height(h),width(w), length(len){ }

3.1.5 构造函数的重载

- 在一个类里可以定义多个构造函数，所有的构造函数都同名，只是参数的个数或参数的类型不同。

```
//例3.3
//在例3.2的基础上，定义两个构造函数，其中一个无参数，一个有参数。

#include <iostream>
using namespace std;
class Box
{
    public:
        Box( );//声明一个无参的构造函数
        Box(int h,int w,int len):height(h),width(w),length(len){ }//声明一个有参的构造函数
        int volume( );
    private:
```

```
        int height;
        int width;
        int length;
};
Box::Box( )//定义一个无参的构造函数
{
    height=10;
    width=10;
    length=10;
}
int Box::volume( )
{return(height*width*length);}

int main( )
{
    Box box1;//建立对象box1,不指定实参
    cout<<"The volume of box1 is "<<box1.volume( )<<endl;
    Box box2(15,30,25);//建立对象box2,指定3个实参
    cout<<"The volume of box2 is "<<box2.volume( )<<endl;
    return 0;
}

在本程序中定义了两个重载的构造函数，其实还可以定义其他重载构造函数，其原型声明可以为
Box::Box(int h);//有1个参数的构造函数
Box::Box(int h,int w);//有两个参数的构造函数
在建立对象时分别给定1个参数和2个参数。
```

- 调用构造函数时不必给出实参的构造函数，称为默认构造函数(default constructor)。无参的构造函数属于默认构造函数。一个类只能有一个默认构造函数。
- 如果在建立对象时选用的是无参构造函数，应注意正确书写定义对象的语句。Box box1;Box box1();
- 尽管在一个类中可以包含多个构造函数，但是对于每一个对象来说，建立对象时只执行其中一个构造函数，并非每个构造函数都被执行。

3.1.6 带默认参数的构造函数

```
//例3.4

#include <iostream>
using namespace std;
class Box
{
    public:
        Box(int h=10,int w=10,int len=10);//指定默认参数
        int volume( );//声明计算体积的函数
   private:
        int height;
        int width;
        int length;
};
Box::Box(int h,int w,int len)//定义时不用再指定默认值
{
    height=h;
    width=w;
    length=len;
}
```

```
int Box::volume( )
{
    return(height*width*length);
}
int main( )
{
    Box box1;//没有给实参
    cout<<"The volume of box1 is "<<box1.volume( )<<endl;
    Box box2(15);//只给定一个实参
    cout<<"The volume of box2 is "<<box2.volume( )<<endl;
    Box box3(15,30);//只给定2个实参
    cout<<"The volume of box3 is "<<box3.volume( )<<endl;
    Box box4(15,30,20);//给定3个实参
    cout<<"The volume of box4 is "<<box4.volume( )<<endl;
    return 0;
}
```

- 要在**声明**构造函数时指定默认值，而不能只在定义构造函数时指定默认值。
- 在声明构造函数时，形参名可以省略。

```
Box(int=10,int=10,int=10);
```

- 如果构造函数的全部参数都指定了默认值，则在定义对象时可以给一个或几个实参，也可以不给出。
- 一个类只能有一个默认构造函数。

```
Box();
Box(int=10,int=10,int=10);
Box box1;?
```

- 重载时注意歧义出现

  定义对象如下：

```
Box();
Box(int,int=10,int=10);
Box(int,int);
```

  定义对象如下：

```
Box box1;
Box box2(15);
Box box3(15,30);
```

3.2 析构函数

- 析构函数(destructor)也是一个特殊的成员函数，它的作用与构造函数相反，它的名字是类名的前面加一个“~”符号。
- 当对象的生命期结束时，会自动执行析构函数。
  - ①如果在一个函数中定义了一个对象(它是自动局部对象)，当这个函数被调用结束时，对象应该释放，在对象释放前自动执行析构函数。
  - ②在main函数结束或调用exit函数结束程序时，调用static局部对象的析构函数。

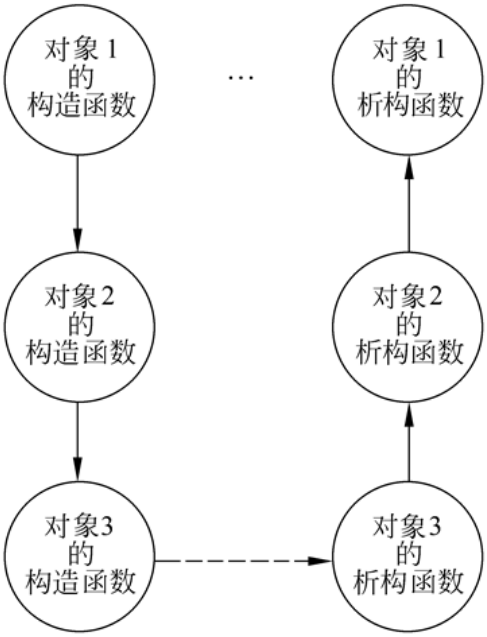
    - ③如果定义了一个全局对象，则在程序的流程离开其作用域时(如main函数结束或调用exit函数) 时，调用该全局对象的析构函数。
    - ④如果用new运算符动态地建立了一个对象，当用delete运算符释放该对象时，先调用该对象的析构函数。
- 析构函数不是删除对象，而是在撤销对象占用的内存之前完成一些清理工作，使这部分内存可以被程序分配给新对象使用。
- 析构函数不返回任何值，没有函数类型，也没有函数参数。因此它不能被重载。**一个类可以有多个构造函数，但只能有一个析构函数。**
- 析构函数的作用并不仅限于释放资源方面，它还可以完成类的设计者所指定的任何操作。
- 一般情况下，应当在声明类的同时定义析构函数。如果用户没有定义析构函数，C++编译系统会自动生成一个析构函数，但什么操作都不进行。

```
//例3.5
//包含构造函数和析构函数的C++程序。

#include<string>
#include<iostream>
using namespace std;
class Student//声明Student类
{
    public:
        Student(int n,string nam,char s )//定义构造函
        {
            num=n;
            name=nam;
            sex=s;
            cout<<”Constructor called.”<<endl;//输出有关信息
        }
        ~Student( )//定义析构函数
        {
            cout<<”Destructor called.”<<endl;//输出有关信息
        }
        void display( )//定义成员函数
        {
            cout<<”num: ”<<num<<endl;
            cout<<”name: ”<<name<<endl;
            cout<<”sex: ”<<sex<<endl<<endl;
        }
    private:
        int num;
        string name;
        char sex;
};

int main( )
{
    Student stud1(10010,”Wang_li”,’f’);//建立对象stud1
    stud1.display( );//输出学生1的数据
    Student stud2(10011,”Zhang_fun”,’m’);//定义对象stud2
    stud2.display( );//输出学生2的数据
    return 0;
}

程序运行结果如下：
Constructor called.                    (执行stud1的构造函数)
```

```
num: 10010                          (执行stud1的display函数)
name:Wang_li
sex: f

Constructor called.                 (执行stud2的构造函数)
num: 10011                          (执行stud2的display函数)
name:Zhang_fun
sex:m
Destructor called.                  (执行stud2的析构函数)
Destructor called.                  (执行stud1的析构函数)
```

3.3调用构造函数和析构函数的顺序

- 调用析构函数的次序正好与调用构造函数的次序相反： 最先被调用的构造函数，其对应的(同一对象中的)析构函数最后被调用，而最后被调用的构造函数，其对应的析构函数最先被调用。
- **先构造后析构，后构造先析构。**
- 但有时会受生命期和作用域的影响



3.4 对象数组

- 数组可以由对象组成(对象数组的每一个元素都是同类的对象)。

```
Student stud[50];
```

- 在建立数组时，同样要调用构造函数。如果有50个元素，需要调用50次构造函数。在需要时可以在定义数组时提供实参以实现初始化。
- 如果构造函数只有一个参数，在定义数组时可以直接在等号后面的花括号内提供实参。

```
Student::Student(int);
Student stud[3]={60,70,78};
```

- 如果构造函数有多个参数，则不能用在定义数组时直接提供所有实参的方法。很容易造成实参与形参的对应关系不清晰，出现歧义性。

```
//类Student的构造函数有多个参数，且为默认参数：
    Student:: Student(int=1001,int=18,int=60);

//定义对象数组为：
    Student stud[3]={1005,60,70};     //有歧义
    Student stud[3]={60,70,78,45};    //不合法
```

- 如果构造函数有多个参数，在定义对象数组时应当在花括号中分别写出构造函数并指定实参。

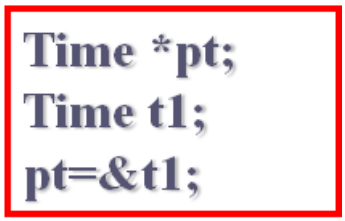
```
//如果构造函数有3个参数，分别代表学号、年龄、成绩。则可以这样定义对象数组：
Student Stud[3]=
{//定义对象数组
    Student(1001,18,87),  //调用第1个元素的构造函数
    Student(1002,19,76),  //调用第2个元素的构造函数
    Student(1003,18,72)   //调用第3个元素的构造函数
};
```

```
//例3.6
//对象数组的使用方法。

#include <iostream>
using namespace std;
class Box
{
    public:
    //声明有默认参数的构造函数
        Box(int h=10,int w=12,int len=15):height(h),width(w),length(len){}
        int volume( );
    private:
        int height;
        int width;
        int length;
};
int Box::volume( )
{
    return(height*width*length);
}
int main( )
{
    Box a[3]=
    {//定义对象数组
        Box(10,12,15),//调用构造函数Box，提供第1个元素的实参
        Box(15,18,20),//调用构造函数Box，提供第2个元素的实参
        Box(16,20,26)//调用构造函数Box，提供第3个元素的实参
    };
    cout<<"volume of a[0] is "<<a[0].volume( )<<endl;
    cout<<"volume of a[1] is "<<a[1].volume( )<<endl;
    cout<<"volume of a[2] is "<<a[2].volume( )<<endl;
}
```

3.5 对象指针

3.5.1 指向对象的指针

- 在建立对象时，会为每一个对象分配存储空间，空间的起始地址就是对象的指针。




```
class Time
{
    public:
        int hour;
        int minute;
        int sec;
        void get_time( );
};
void Time::get_time( )//在类外定义set_time函数
{
    cout<<hour<<“:”<<minute<<“:”<<sec;
}
```

- 定义指向类对象的指针变量的一般形式：
  类名 *对象指针名

```
*pt
(*pt).hour =等价= pt->hour
(*pt).get_time( ) =等价= pt->get_time( )
```

3.5.2 指向对象成员的指针

1. 指向对象数据成员的指针

- 定义形式：
  数据类型名 *指针变量名;

```
int *p1;
p1 = &t1.hour;
cout<<*p1<<endl;
```

2. 指向对象成员函数的指针

- 普通定义形式：
  数据类型名 (*指针变量名)(参数列表);

```
void (*p)( );
p=fun;
(*p)( );
p=t1.get_time;//错误
```

- 指向成员函数的定义形式：
  数据类型名 (类名::*指针变量名)(参数列表);

```
void (Time :: *p2)( );
```

指针变量名=&类名::成员函数名;

```
p2=[&]Time::get_time;
(*p)( );
```

```
//例3.7
//对象指针的使用方法。

#include <iostream>
using namespace std;
class Time
{
    public:
        Time(int,int,int);
        int hour;
        int minute;
        int sec;
        void get_time();
};

Time::Time(int h,int m,int s)
{
    hour=h;
    minute=m;
    sec=s;
}
void Time::get_time()
{
    cout<<hour<<":"<<minute<<":"<<sec<<endl;
}

int main()
{
    Time t1(10,13,56);
    int *p1=&t1.hour;
    cout<<*p1<<endl;
    t1.get_time();
    Time *p2=&t1;
    p2->get_time();
    void (Time::*p3)();
    p3=&Time::get_time;
    (t1.*p3)();
    return 0;
}
```

3.5.3 this指针

- 同一类的n个对象有n组相同大小的空间存放n个对象中的数据成员，但是不同的对象却调用相同的函数代码段。
- 每一个成员函数中都包含一个特殊的指针，这个指针的名字是固定的，称为this。
- this是指向本来对象的指针，其值为当前被调用的成员函数所在的对象的起始地址。

```
//例如，调用a.volume ()实际上是执行
(this->height)*(this->width)*(this->length)
//相当于
(a.height)*(a.width)*(a.length)
```

- this指针是隐式使用的，是作为函数参数传递给成员函数的

```
//如：
int Box::volume( )
{
    return(height*width*length);
}
//处理为
int Box::volume( Box *this)
{
    return(this->height*this->width*this->length);
}
//调用方式为：a.volume(&a);
```

- 有需要的时候可以显示的使用this
- 可用`*this`表示被调用成员函数所在的对象，即`*this`就是this所指向的对象，

```
//如：
return((*this).height*(*this).width)*(*this).length))
```

- 调用对象a的成员函数f，实际上就是在调用成员函数时使用this指针指向对象a，从而访问对象a的成员。

3.6 共用对象的保护

- 既要使数据在一定范围内共享，又不愿它被随意修改，从技术上可以把数据指定为**只读型**的。
- C++提供const手段，将数据、对象、成员函数指定为常量，从而实现了只读要求，达到保护数据的目的。

3.6.1 常对象

- 定义格式：

  const 类名 对象名(实参表);

  或：

  类名 const 对象名(实参表);

  把对象定义为常对象，对象中的数据成员就是常变量，在定义时必须带实参作为数据成员的初值，在程序中**不允许修改**常对象的数据成员值。
- 如果一个常对象的成员函数未被定义为常成员函数（除构造函数和析构函数外），则对象不能调用这样的成员函数。

```
//如：
const Time t1( 10,15,36);
t1.get_time();//错误，不能调用
```

为了访问常对象中的数据成员，要定义常成员函数：

```
void  get_time()  const
```

但仍然不能修改常对象中数据成员的值。

- 如果在常对象中要修改某个数据成员，C++提供了指定可变的数据成员方法。
  格式： **mutable 类型 数据成员;**
  在定义数据成员时加mutable后，将数据成员声明为可变的数据成员，就可以用声明为const的成员函数修改它的值。

3.6.2 常对象成员

1. 常数据成员

- **格式： const 类型 数据成员名**

  将类中的数据成员定义为具有只读的性质。
  注意只能通过带参数初始表的构造函数对常数据成员进行初始化。

```
//例：
const   int  hour;
Time :: Time( int h )
{hour = h;}//错误
//应该写成：
Time :: Time( int h ) : hour (h) {}
```

- 在类中声明了某个常数据成员后，该类中每个对象的这个数据成员的值都是只读的，而每个对象的这个数据成员的值可以不同，由定义对象时给出。

2. 常成员函数

- 定义格式：
  类型 函数名 (形参表) const
  const 是函数类型的一部分，在声明函数原型和定义函数时都要用const关键字。
- 常成员函数不能修改对象的数据成员，也不能调用该类中没有由关键字const修饰的成员函数，从而保证了在常成员函数中不会修改数据成员的值。
- 如果一个对象被说明为常对象，则通过该对象只能调用它的常成员函数。
- 一般成员函数可以访问或修改本类中的非 const数据成员。而常成员函数只能读本类中的数据成员，而不能写它们。

数据成员	非 const 成员函数	const 成员函数
非 const 的数据成员	可以引用,也可以改变值	可以引用,但不可以改变值
const 数据成员	可以引用,但不可以改变值	可以引用,但不可以改变值
const 对象的数据成员	不允许引用和改变值	可以引用,但不可以改变值

- 如果类中有部分数据成员的值要求为只读，可以将它们声明为const，这样成员函数只能读这些数据成员的值，但不能修改它们的值。
- 如果所有数据成员的值为只读，可将对象声明为const，在类中必须声明const 成员函数，常对象只能通过常成员函数读数据成员。

- 常对象不能调用非const成员函数。
  提醒：如果常对象的成员函数未加const，编译系统将其当作非const成员函数；常成员函数不能调用非const成员函数。

3.6.3 指向对象的常指针

- 如果在定义指向对象的指针时，使用了关键字 const ， 它就是一个常指针，必须在定义时对其初始化。并且在程序运行中不能再修改指针的值。
  类名 * const 指针变量名 = 对象地址

```
//例：
Time t1(10,12,15), t2;
Time * const p1 = & t1;
//在此后，程序中不能修改p1。
//例：
Time * const p1 = & t2;//错误语句
```

- 指向对象的常指针，在程序运行中始终指的是同一个对象。即指针变量的值始终不变，但它所指对象的数据成员值可以修改。
- 当需要将一个指针变量固定地与一个对象相联系时，就可将指针变量指定为const。
- 往往用常指针作为函数的形参，目的是不允许在函数中修改指针变量的值，让它始终指向原来的对象。

3.6.4 指向常对象的指针变量

- 首先了解指向常变量的指针变量
  const char *ptr;

  注意const的位置在最左侧，它与类型名char紧连，表示指针变量ptr指向的char变量是常变量，不能通过ptr来改变其值的。
- 定义指向常变量的指针变量的一般形式为
  const 类型名 *指针变量名;
- 如果一个变量已被声明为常变量，只能用指向常变量的指针变量指向它，而不能用一般的(指向非const型变量的)指针变量去指向它。
- 指向常变量的指针变量除了可以指向常变量外，还可以指向未被声明为const的变量。此时不能通过此指针变量改变该变量的值。如果希望在任何情况下都不能改变c1的值，则应把它定义为const型。
- 如果函数的形参是指向非const型变量的指针，实参只能用指向非const变量的指针，而不能用指向const变量的指针，这样，在执行函数的过程中可以改变形参指针变量所指向的变量(也就是实参指针所指向的变量)的值。
- 如果函数的形参是指向const型变量的指针，在执行函数过程中显然不能改变指针变量所指向的变量的值，因此允许实参是指向const变量的指针，或指向非const变量的指针。指向常对象的指针变量的概念和使用是与此类似的，只要将“变量”换成“对象”即可。
- 如果一个对象已被声明为常对象，只能用指向常对象的指针变量指向它，而不能用一般的(指向非const型对象的)指针变量去指向它。
- 如果定义了一个指向常对象的指针变量，并使它指向一个非const的对象，则其指向的对象是不能通过指针来改变的。如果希望在任何情况下t1的值都不能改变，则应把它定义为const型。
- 指向常对象的指针最常用于函数的形参，目的是在保护形参指针所指向的对象，使它在函数执行过程中不被修改。

- 当希望在调用函数时对象的值不被修改，就应当把形参定义为指向常对象的指针变量，同时用对象的地址作实参(对象可以是const或非const型)。如果要求该对象不仅在调用函数过程中不被改变，而且要求它在程序执行过程中都不改变，则应把它定义为const型。
- 如果定义了一个指向常对象的指针变量，是不能通过它改变所指向的对象的值的，但是指针变量本身的值是可以改变的。

3.6.5 对象的常引用

- 一个变量的引用就是变量的别名。实质上，变量名和引用名都指向同一段内存单元。如果形参为变量的引用名，实参为变量名，则在调用函数进行虚实结合时，并不是为形参另外开辟一个存储空间(常称为建立实参的一个拷贝)，而是把实参变量的地址传给形参(引用名)，这样引用名也指向实参变量。

```
//例3.8
//对象的常引用。

#include <iostream>
using namespace std;
class Time
{
    public:
        Time(int,int,int);
        int hour;
        int minute;
        int sec;
};

Time::Time(int h,int m,int s)//定义构造函数
{
    hour=h;
    minute=m;
    sec=s;
}
void fun(Time &t)//形参t是Time类对象的引用
{
    t.hour=18;
}
int main( )
{
    Time t1(10,13,56);// t1是Time类对象
    fun(t1);//实参是Time类对象，可以通过引用来修改实参t1的值
    cout<<t1.hour<<endl;//输出t1.hour的值为18
    return 0;
}
```

如果不希望在函数中修改实参t1的值，可以把引用变量t声明为const(常引用)，函数原型为

则在函数中不能改变t的值，也就是不能改变其对应的实参t1的值。

```
void fun(const Time &t);
```

则在函数中不能改变t的值，也就是不能改变其对应的实参t1的值。

- 在C++面向对象程序设计中，经常用常指针和常引用作函数参数。这样既能保证数据安全，使数据不能被随意修改，在调用函数时又不必建立实参的拷贝。用常指针和常引用作函数参数，可以提高程序运行效率。

3.6.6 const数据类型的小结

形　式	含　义
Time const t1; 或 const Time t1;	t1 是常对象,其值在任何情况下都不能改变
void Time::fun( )const	fun 是 Time 类中的常成员函数,可以引用,但不能修改本类中的数据成员
Time * const p;	p 是指向 Time 对象的常指针,p 的值(即 p 的指向)不能改变
const Time *p;	p 是指向 Time 类常对象的指针,其指向的类对象的值不能通过指针来改变
Time &t1 =t;	t1 是 Time 类对象 t 的引用,t 和 t1 指向同一段内存空间

3.7 对象的动态建立和释放

- 可以用new运算符动态建立对象，用delete运算符撤销对象。
- 如果已经定义了一个Box类，可以用下面的方法动态地建立一个对象：编译系统开辟了一段内存空间，并在此内存空间中存放一个Box类对象，同时调用该类的构造函数，以使该对象初始化(如果已对构造函数赋予此功能的话)。但是此时用户还无法访问这个对象，因为这个对象既没有对象名，用户也不知道它的地址。这种对象称为无名对象，它确实是存在的，但它没有名字。
- 用new运算符动态地分配内存后，将返回一个指向新对象的指针的值，即所分配的内存空间的起始地址。用户可以获得这个地址，并通过这个地址来访问这个对象。需要定义一个指向本类的对象的指针变量来存放该地址。

```
Box *pt;//定义一个指向Box类对象的指针变量pt
pt=new Box;//在pt中存放了新建对象的起始地址
//在程序中就可以通过pt访问这个新建的对象。如
cout<<pt->height;//输出该对象的height成员
cout<<pt->volume( );//调用该对象的volume函数，计算并输出体积
```

- C++还允许在执行new时，对新建立的对象进行初始化。

```
//如
Box *pt=new Box(12,15,18);
```

- 调用对象既可以通过对象名，也可以通过指针。用new建立的动态对象一般是不用对象名的，是通过指针访问的，它主要应用于动态的数据结构，如链表。访问链表中的结点，并不需要通过对象名，而是在上一个结点中存放下一个结点的地址，从而由上一个结点找到下一个结点，构成链接的关系。
- 在执行new运算时，如果内存量不足，无法开辟所需的内存空间，目前大多数C++编译系统都使new返回一个0指针值。只要检测返回值是否为0，就可判断分配内存是否成功。
- 在不再需要使用由new建立的对象时，可以用delete运算符予以释放。

```
//如
delete pt;//释放pt指向的内存空间
//这就撤销了pt指向的对象。此后程序不能再使用该对象。
```

- 如果用一个指针变量pt先后指向不同的动态对象，应注意指针变量的当前指向，以免删错了对象。
- 在执行delete运算符时，在释放内存空间之前，自动调用析构函数，完成有关善后清理工作。

3.8 对象的赋值和复制

3.8.1 对象的赋值

- 对象之间的赋值也是通过赋值运算符“=”进行。一般形式为
  对象名1 = 对象名2;
  对象名1和对象名2必须属于同一个类。

```
//例如
Student stud1,stud2;//定义两个同类的对象
//
stud2=stud1;//将stud1赋给stud2
```

```
//例3.9
//对象的赋值。

#include <iostream>
using namespace std;
class Box
{
    public:
        Box(int=10,int=10, int=10);//声明有默认参数的构造函数
        int volume( );
    private:
        int height;
        int width;
        int length;
};
Box::Box(int h,int w,int len)
{
    height=h;
    width=w;
    length=len;
}
int main( )
{
    Box box1(15,30,25),box2;//定义两个对象box1和box2
    cout<<"The volume of box1 is "<<box1.volume( )<<endl;
    box2=box1;//将box1的值赋给box2
    cout<<"The volume of box2 is "<<box2.volume( )<<endl;
    return 0;
}

运行结果如下:
The volume of box1 is 11250
The volume of box2 is 11250
(1) 对象的赋值只对其中的数据成员赋值，而不对成员函数赋值。
(2) 类的数据成员中不能包括动态分配的数据，否则在赋值时可能出现严重后果。
```

3.8.2 对象的复制

- 对象的复制机制就是将对象在某一瞬时的状态保留下来。即用一个已有的对象快速地复制出多个完全相同的对象。如
  Box box2(box1);
  其作用是用已有的对象box1去克隆出一个新对象box2。
- 其一般形式为
  类名 对象2(对象1);
  用对象1复制出对象2。
- C++还有另一种复制形式，用赋值号代替括号，如
  Box box2=box1;　　//用box1初始化box2
  其一般形式为
  类名 对象名1 = 对象名2;
- 可以在一个语句中进行多个对象的复制。如
  Box box2=box1,box3=box2;
  按box1来复制box2和box3。

```
int main( )
 {
    Box box1(15,30,25);//定义box1
    cout<<box1.volume( )<<endl;
    Box box2=box1,box3=box2; //按Box1复制box2,box3
    cout<<box2.volume( )<<endl;
    cout<<box3.volume( )<<endl;
 }
```

- 赋值和赋值的区别：
  对象的赋值是对一个已经存在的对象赋值，因此必须先定义被赋值的对象，才能进行赋值；
  对象的复制则是从无到有地建立一个新对象，并使它与一个已有的对象完全相同(包括对象的结构和成员的值)。
- 普通构造函数和复制构造函数的区别：
  - **形式上**
    类名(形参列表)；//普通构造函数Box(int h,int w);
    类名(类名&对象名)；//复制构造函数Box(Box &b);
  - **建立对象时，实参类型不同**
    Box box1(12,15,16);//实参是整数,调用普通构造函数
    Box box2(box1);//实参是对象名,调用复制构造函数
  - **在什么情况下调用**
    普通构造函数在建立对象时调用；
    复制构造函数用已有对象复制一个新对象时被调用：
    - 在程序中需要建立一个新对象，并用另一个同类的对象初始化时
    - 当函数的参数是类的对象时；

```
void fun(Box b){}
int main()
{
    Box box1(12,15,18);
    fun(box1);//调用函数时将复制一个新对象b
    return 0;
}
```

    - 函数的返回值是类的对象；

```
Box f()
{
    Box box1(12,15,18);
    return box1;
}
int main()
{
    Box box2;
    box2=f();//调用f函数，返回Box类的临时对象
}
```

3.9 静态成员

可以用静态的数据成员实现多个对象之间的数据共享

3.9.1 静态数据成员

- 静态数据成员是一种特殊的数据成员。它以关键字static开头。

```
//例如
class Box
{
    public:
        int volume( );
    private:
        static int height;//把height为静态的数据成员
        int width;
        int length;
};
```

- 静态数据成员不属于某一个对象，在为对象所分配的空间中不包括静态数据成员所占的空间。只要在类中定义了静态数据成员，即使不定义对象，也为静态数据成员分配空间，它可以被引用。
- 静态数据成员是在程序编译时被分配空间的，到程序结束时才释放空间。
- 静态数据成员可以初始化，但只能在**类体外**进行初始化。

```
//如
int Box::height=10;
```

- 初始化的一般形式为
  数据类型 类名::静态数据成员名=初值;
  不必在初始化语句中加static。
  如果未对静态数据成员赋初值，则编译系统会自动赋予初值0。
- 静态数据成员既可以通过对象名引用，也可以通过类名来引用。

```
//例3.10
//引用静态数据成员。

#include <iostream>
using namespace std;
class Box
{
    public:
```

```
        Box(int,int);
        int volume( );
        static int height;//把height定义为公用的静态的数据成员
        int width;
        int length;
};
Box::Box(int w,int len)//通过构造函数对width和length赋初值
{
    width=w;
    length=len;
}
int Box::volume( )
{
    return(height*width*length);
}
int Box::height=10;//对静态数据成员height初始化
int main( )
{
     Box a(15,20),b(20,30);
     cout<<a.height<<endl;//通过对象名a引用静态数据成员
     cout<<b.height<<endl;//通过对象名b引用静态数据成员
     cout<<Box::height<<endl;//通过类名引用静态数据成员
     cout<<a.volume( )<<endl;//调用volume函数
}
```

3.9.2 静态成员函数

- 在类中声明函数的前面加static就成了静态成员函数。

```
//如
static int volume( );
```

静态成员函数是**类的一部分**，而不是对象的一部分。如果要在类外调用公用的静态成员函数，要用类名和域运算符“::”。

```
//如
Box::volume( );
```

实际上也允许通过对象名调用静态成员函数

```
//如
a.volume( );
```

- 静态成员函数的作用不是为了对象之间的沟通，而是为了能处理静态数据成员。
- 静态成员函数与非静态成员函数的根本区别是：非静态成员函数有this指针，静态成员函数没有this指针。
- 静态成员函数可以直接引用本类中的静态数据成员

```
cout<<height<<endl;//若height已声明为static,合法
cout<<width<<endl;//若width是非静态数据成员,不合法
```

- 静态成员函数不能直接访问本类中的非静态成员。除非加对象名和成员运算符“.”。

```
//如
cout<<a.width<<endl;
```

```
//例3.11
//静态成员函数的应用。

#include <iostream>
using namespace std;
class Student//定义Student类
{
    public:
        Student(int n,int a,float s):num(n),age(a),score(s){ }
        void total( );
        static float average( );//声明静态成员函数
    private:
        int num;
        int age;
        float score;
        static float sum;//静态数据成员
        static int count;//静态数据成员
};
void Student::total( )//定义非静态成员函数
{
    sum+=score;//累加总分
    count++;//累计已统计的人数
}
float Student::average( )//定义静态成员函数
{
    return(sum/count);
}

float Student::sum=0;//对静态数据成员初始化
int Student::count=0;//对静态数据成员初始化

int main( )
{
    Student stud[3]=
    {//定义对象数组并初始化
        Student(1001,18,70),
        Student(1002,19,78),
        Student(1005,20,98)
    };
    int n;
    cout<<"please input the number of students:";
    cin>>n;//输入需要求前面多少名学生的平均成绩
    for(int i=0;i<n;i++)//调用3次total函数
        stud[i].total( );
    cout<<"the average score of "<<n<<" students is"<<Student::average( )
<<endl;
    return 0;
}
```

3.10 友元

友元(friend)，可以访问与其有好友关系的类中的**私有成员**。友元包括友元函数和友元类。

3.10.1 友元函数

- 如果在**本类以外**的其他地方定义了一个函数(这个函数可以是不属于任何类的非成员函数，也可以是其他类的成员函数)，在类体中用friend对其进行声明，此函数就称为本类的友元函数。友元函数可以访问这个类中的私有成员。

1. 将普通函数声明为友元函数

```
//例3.12
//友元函数的简单例子。

#include <iostream>
using namespace std;

class Time
{
    public:
        Time(int,int,int);
        friend void display(Time &);//声明display函数为Time类的友元函数
    private://以下数据是私有数据成员
       int hour;
       int minute;
       int sec;
};
Time::Time(int h,int m,int s)//构造函数
{
    hour=h;
    minute=m;
    sec=s;
}
void display(Time& t)//友元函数，形参t是Time类对象的引用
{
    cout<<t.hour<<":"<<t.minute<<":"<<t.sec<<endl;
}
int main( )
{
    Time t1(10,13,56);
    display(t1);//调用display函数，实参t1是Time类对象
    return 0;
}
```

2. 友元成员函数

friend函数可以是另一个类中的成员函数。

```
//例3.13
//友元成员函数的简单应用。

#include <iostream>
using namespace std;
class Date;//对Date类的提前引用声明
class Time//定义Time类
{
    public:
        Time(int,int,int);
        void display(Date &);//display是成员函数，形参是Date类对象的引用
    private:
```

```
        int hour;
        int minute;
        int sec;
};
class Date//声明Date类
{
    public:
        Date(int,int,int);
        friend void Time::display(Date &);//声明Time中的display函数为友元成员函数
    private:
       int month;
       int day;
       int year;
};
Time::Time(int h,int m,int s)//类Time的构造函数
{
    hour=h;
    minute=m;
    sec=s;
}
void Time::display(Date &d)//输出年、月、日和时、分、秒
{
    cout<<d.month<<"/"<<d.day<<"/"<<d.year<<endl;
    cout<<hour<<":"<<minute<<":"<<sec<<endl;
}
Date::Date(int m,int d,int y)//类Date的构造函数
{
    month=m;
    day=d;
    year=y;
}
int main( )
{
    Time t1(10,13,56);//定义Time类对象t1
    Date d1(12,25,2004);//定义Date类对象d1
    t1.display(d1);//调用t1的display，实参是Date类对象d1
    return 0;
}
```

3.10.2 友元类

- 不仅可以将一个函数声明为一个类的“朋友”，而且可以将一个类(例如B类)声明为另一个类(例如A类)的“朋友”。这时B类就是A类的友元类。友元类B中的所有函数都是A类的友元函数，可以访问A类中的所有成员。
- 在A类的定义体中用以下语句声明B类为其友元类：

  `friend B;`

  声明友元类的一般形式为

  friend 类名;
- 友元的使用说明
  - 友元的关系是**单向**的而不是双向的。
  - 友元的关系不能传递。
- 实际中，一般并不把整个类声明为友元类，而只将确实有需要的成员函数声明为友元函数，这样更安全一些。

- 友元可以访问其他类中的私有成员，不能不说这是对封装原则的一个小的破坏。但是它能有助于数据共享，能提高程序的效率。

3.11 类模板

- 有多个类，功能相同，仅仅是数据类型不同：

```
class Compare_int
{
    public:
        Compare(int a,int b)
            {x=a;y=b;}
        int max( )
            {return(x>y)?x:y;}
        int min( )
            {return(x<y)?x:y;}
     private:
        int x,y;
};
class Compare_float
{
    public:
        Compare(float a,float b)
            {x=a;y=b;}
        float max( )
            {return(x>y)?x:y;}
        float min( )
            {return(x<y)?x:y;}
    private:
        float x,y;
};
template<class numtype>                    //声明一个模
class Compare                              //类模板名为Compare
{
    public:
        Compare(numtype a,numtype b)
            {x=a;y=b;}
        numtype max( )
            {return (x>y)?x:y;}
        numtype min( )
            {return (x<y)?x:y;}
    private:
        numtype x,y;
};
```

- 可以声明一个通用的类模板，它可以有一个或多个虚拟的类型参数。
  template <class 类型参数名>
- 类模板包含数据类型，称为参数化的类。
- 类模板是类的抽象，类是类模板的实例。
- 声明了类模板后如何使用

```
Compare cmpl(4,7);
Compare cmpl <int> (4,7);
```

```
//例9.14
//声明一个类模板，利用它分别实现两个整数、浮点数和字符的比较，求出大数和小数。

#include <iostream>
using namespace std;
template<class numtype>//定义类模板
class Compare
{
    public:
        Compare(numtype a,numtype b)
            {x=a;y=b;}
        numtype max( )
            {return (x>y)?x:y;}
        numtype min( )
            {return (x<y)?x:y;}
    private:
        numtype x,y;
};
int main( )
{
    Compare <int> cmp1(3,7);
    cout<<cmp1.max( )<<" is the Maximum of two integer numbers."<<endl;
    cout<<cmp1.min( )<<" is the Minimum of two integer numbers.
"<<endl<<endl;
    Compare <float> cmp2(45.78,93.6);
    cout<<cmp2.max( )<<" is the Maximum of two float numbers."<<endl;
    cout<<cmp2.min( )<<" is the Minimum of two float numbers."<<endl<<endl;
    Compare <char> cmp3('a','A'); cout<<cmp3.max( )<<" is the Maximum of two
characters."<<endl;
    cout<<cmp3.min( )<<" is the Minimum of two characters."<<endl;
    return 0;
}
```

- 如果改为在类模板外定义，不能用一般定义类成员函数的形式：

```
numtype Compare::max( ) {/* ... */}
//不能这样定义类模板中的成员函数
```

  而应当写成类模板的形式：

```
template<class numtype>
    numtype Compare<numtype>::max( )
        { return (x>y)?x:y;}
```

- 归纳以上的介绍，可以这样声明和使用类模板：
  - 先写出一个实际的类。由于其语义明确，含义清楚，一般不会出错。
  - 将此类中准备改变的类型名(如int要改变为float或char)改用一个自己指定的虚拟类型名(如上例中的numtype)。
  - 在类声明前面加入一行，格式为

```
template<class 虚拟类型参数>
```

```
//如
template<class numtype>//注意本行末尾无分号
class Compare
{/* ... */};//类体
```

- 用类模板定义对象时用以下形式：
  类模板名<实际类型名> 对象名;
  类模板名<实际类型名> 对象名(实参表列);

```
//如
Compare<int> cmp;
Compare<int> cmp(3,7);
```

- 如果在类模板外定义成员函数，应写成类模板形式：
  template<class 虚拟类型参数>
  函数类型 类模板名<虚拟类型参数>::成员函数名(函数形参表列) {...}

- 说明
  - 类模板的类型参数可以有一个或多个，每个类型前面都必须加class，

```
//如
template<class T1,class T2>
class someclass
{/* ... */};
```

在定义对象时分别代入实际的类型名，如

```
someclass<int,double> obj;
```

  - 和使用类一样，使用类模板时要注意其作用域，只能在其有效作用域内用它定义对象。
  - 模板可以有层次，一个类模板可以作为基类，派生出派生模板类。

第四章 运算符重载

4.1 什么是运算符重载

- 所谓重载，就是重新赋予新的含义。函数重载就是对一个已有的函数赋予新的含义，使之实现新功能。

```
//例4.1
//通过函数来实现复数相加。

#include <iostream>
using namespace std;
class Complex//定义Complex类
{
    public:
        Complex( ){real=0;imag=0;}//定义构造函数
        Complex(double r,double i){real=r;imag=i;}
        Complex complex_add(Complex &c2);
        void display( );//声明输出函数
    private:
```

```
        double real;//实部
        double imag;//虚部
};
Complex Complex::complex_add(Complex &c2)
{
    Complex c;
    c.real=real+c2.real;
    c.imag=imag+c2.imag;
    return c;
}
void Complex::display( )//定义输出函数
{cout<<"("<<real<<","<<imag<<"i)"<<endl;}
int main( )
{
    Complex c1(3,4),c2(5,-10),c3;//定义3个复数对象
    c3=c1.complex_add(c2);//调用复数相加函数
    cout<<"c1="; c1.display( );//输出c1的值
    cout<<"c2="; c2.display( );//输出c2的值
    cout<<"c1+c2="; c3.display( );//输出c3的值
    return 0;
}
```

4.2 运算符重载的方法

- 运算符重载的方法是定义一个重载运算符的函数，在需要执行被重载的运算符时，系统就自动调用该函数，以实现相应的运算。运算符重载是通过定义函数实现的。运算符重载实质上是函数的重载。
- 重载运算符的函数一般格式如下：
  函数类型 operator 运算符名称 (形参表列)
  { 对运算符的重载处理 }

```
//例如，想将"+"用于Complex类(复数)的加法运算，函数的原型可以是这样的：
Complex operator+ (Complex& c1,Complex& c2);
```

```
//例4.2
//改写例4.1，重载运算符"+"，使之能用于两个复数相加。

#include <iostream>
using namespace std;
class Complex
{
    public:
        Complex( ){real=0;imag=0;}
        Complex(double r,double i){real=r;imag=i;}
        Complex operator+(Complex &c2);
        void display( );
    private:
        double real;
        double imag;
};
Complex Complex::operator+(Complex &c2)
{
    Complex c;
    c.real=real+c2.real;
    c.imag=imag+c2.imag;
```

```
    return c;
}

void Complex::display( )
{cout<<"("<<real<<","<<imag<<"i)"<<endl;}

int main( )
{
    Complex c1(3,4),c2(5,-10),c3;
    c3=c1+c2;//运算符+用于复数运算
    cout<<"c1=";c1.display( );
    cout<<"c2=";c2.display( );
    cout<<"c1+c2=";c3.display( );
    return 0;
}
```

- 比较例4.1和例4.2，只有两处不同：
    - 在例4.2中以`operator+`函数取代了例4.1中的`complex_add`函数，而且只是函数名不同，函数体和函数返回值的类型都是相同的。
    - 在main函数中，以“`c3=c1+c2;`”取代了例4.1中的“`c3=c1.complex_add(c2);`”。在将运算符+重载为类的成员函数后，C++编译系统将程序中的表达式c1+c2解释为
      `c1.operator+(c2)  //其中c1和c2是Complex类的对象`
      即以c2为实参调用c1的运算符重载函数`operator+(Complex &c2)`，进行求值，得到两个复数之和。

4.3 重载运算符的规则

- C++不允许用户自己定义新的运算符，只能对已有的C++运算符进行重载。
- C++允许重载的运算符
  C++中绝大部分的运算符允许重载。
  不能重载的运算符只有5个：

运算符	类型
.	成员访问运算符
.*	成员指针访问运算符
::	域运算符
sizeof	长度运算符
?:	条件运算符

- 重载不能改变运算符运算对象(即操作数)的个数。
- 重载不能改变运算符的优先级别。
- 重载不能改变运算符的结合性。
- 重载运算符的函数不能有默认的参数，否则就改变了运算符参数的个数，与前面矛盾。
- 重载的运算符必须和用户定义的自定义类型的对象一起使用，其参数至少应有一个是类对象(或类对象的引用)。也就是说，参数不能全部是C++的标准类型，以防止用户修改用于标准类型数据的运算符的性质。

```
int operator + (int a, int b)
{return(a-b);}
```

- 用于类对象的运算符一般必须重载，但有两个例外，运算符“=”和“&”不必用户重载。
    - 赋值运算符(=)可以用于每一个类对象，可以利用它在同类对象之间相互赋值。
    - 地址运算符&也不必重载，它能返回类对象在内存中的起始地址。
- 应当使重载运算符的功能类似于该运算符作用于标准类型数据时所实现的功能。
- 运算符重载函数可以是类的成员函数(如例4.2)，也可以是类的友元函数，还可以是既非类的成员函数也不是友元函数的普通函数。

4.4 运算符重载函数作为类成员函数和友元函数

重载函数operator+访问了两个对象中的成员，一个是this指针指向的对象中的成员，一个是形参对象中的成员。如`this->real+c2.real`，`this->real`就是`c1.real`。
在将运算符函数重载为成员函数后，如果出现含该运算符的表达式，如c1+c2,编译系统把它解释为`c1.operator+(c2)`
即通过对象c1调用运算符重载函数，并以表达式中第二个参数(运算符右侧的类对象c2)作为函数实参。
运算符重载函数的返回值是Complex类型，返回值是复数c1和c2之和`(Complex(c1.real + c2.real,c1.imag+c2.imag))`。

运算符重载函数除了可以作为类的成员函数外，还可以是非成员函数。

```
//例4.3
//将运算符“+”重载为适用于复数加法，重载函数不作为成员函数，而放在类外，作为Complex类的友元函数。

#include <iostream>
using namespace std;
class Complex
{
    public:
        Complex( )
            {real=0;imag=0;}
        Complex(double r,double i)
            {real=r;imag=i;}
    friend Complex operator + (Complex&c1,Complex &c2);//重载函数作为友元函数
    void display( );
    private:
        double real;
        double imag;
};

Complex operator + (Complex &c1,Complex &c2)//定义作为友元函数的重载函数
    {return Complex(c1.real+c2.real, c1.imag+c2.imag);}
void Complex::display( )
    {cout<<”(”<<real<<”,”<<imag<<”i)”<<endl;}
int main( )
{
    Complex c1(3,4),c2(5,-10),c3;
    c3=c1+c2;
    cout<<”c1=”; c1.display( );
    cout<<”c2=”; c2.display( );
    cout<<”c1+c2 =”; c3.display( );
}
```

在将运算符“+”重载为非成员函数后，C++编译系统将程序中的表达式c1+c2解释为`operator+(c1,c2)`
即执行c1+c2相当于调用以下函数：

```
Complex operator + (Complex &c1,Complex &c2)
{return Complex(c1.real+c2.real, c1.imag+c2.imag);}
```

求出两个复数之和。运行结果同例4.2。

- 运算符重载函数可以是类的成员函数，也可以是类的友元函数，还可以是既非类的成员函数也不是友元函数的普通函数。
- 首先，只有在极少的情况下才使用既不是类的成员函数也不是友元函数的普通函数，原因是普通函数不能直接访问类的私有成员。
- 如果将运算符重载函数作为成员函数，它可以通过this指针自由地访问本类的数据成员，因此可以少写一个函数的参数。**但必须要求运算表达式第一个参数(即运算符左侧的操作数)是一个类对象，而且与运算符函数的类型相同。**
- 如想将一个复数和一个整数相加，如`c1+i`，可以将运算符重载函数作为成员函数，如下面的形式：

  ```
  Complex Complex::operator+(int &i)//运算符重载函数作为Complex类的成员函数
      {return Complex(real+i,imag);}
  ```

  注意在表达式中重载的运算符“+”左侧应为Complex类的对象，如

  ```
  c3=c2+i;
  //不能写成↓
  c3=i+c2;//编译出错
  ```

- 如果函数需要访问类的私有成员，则必须声明为友元函数。可以在Complex类中声明：

  ```
  friend Complex operator+(int &i,Complex &c);//第一个参数可以不是类对象
  ```

  在类外定义友元函数：

  ```
  Complex operator+(int &i, Complex &c)
  //运算符重载函数不是成员函数
      {return Complex(i+c.real,c.imag);}
  ```

- 将双目运算符重载为友元函数时，在函数的形参表列中必须有两个参数，不能省略，形参的顺序任意，不要求第一个参数必须为类对象。但在使用运算符的表达式中，要求运算符左侧的操作数与函数第一个参数对应，运算符右侧的操作数与函数的第二个参数对应。如

  ```
  c3=i+c2;//正确，类型匹配
  c3=c2+i;//错误，类型不匹配
  ```

- 注意，数学上的交换律在此不适用。如果希望适用交换律，则应再重载一次运算符“+”。如

  ```
  Complex operator+(Complex &c, int &i)
  //此时第一个参数为类对象
      {return Complex(i+c.real,c.imag);}
  ```

编译系统会根据表达式的形式选择调用与之匹配的运算符重载函数。

- 可以将以上两个运算符重载函数都作为友元函数，也可以将一个运算符重载函数(运算符左侧为对象名的) 作为成员函数，另一个(运算符左侧不是对象名的)作为友元函数。**但不可能将两个都作为成员函数。**
- 单目运算符重载为成员函数，双目运算符重载为友元函数。

4.5 重载双目运算符

例4.4 定义一个字符串类String，用来存放不定长的字符串，重载运算符“==”,“<”和“>”，用于两个字符串的等于、小于和大于的比较运算。

编写C++程序的指导思想是： 先搭框架，逐步扩充，由简到繁，最后完善。边编程，边调试，边扩充。

例4.4

(1)先建立一个string类

(2)重载运算符“>”

(3)扩展到3个运算符重载

(4)最后完善输出

4.6 重载单目运算符

单目运算符只有一个操作数，如!a，-b，&c，*p，还有最常用的++i和--i等。重载单目运算符的方法与重载双目运算符的方法是类似的。但由于单目运算符只有一个操作数，因此运算符重载函数只有一个参数，如果运算符重载函数作为成员函数，则还可省略此参数。

例4.5

例4.6

4.7 重载流插入运算符和流提取运算符

- C++的流插入运算符“<<”和流提取运算符“>>”是C++在类库中提供的，所有C++编译系统都在类库中提供输入流类istream和输出流类ostream。cin和cout分别是istream类和ostream类的对象。
- 在类库提供的头文件中已经对“<<”和“>>”进行了重载，使之作为流插入运算符和流提取运算符，能用来输出和输入C++标准类型的数据。
- 用户自己定义的类型的数据，是不能直接用“<<”和“>>”来输出和输入的。如果想用它们输出和输入自己声明的类型的数据，必须对它们重载。
- 对“<<”和“>>”重载的函数形式如下：

  istream & operator >> (istream &,自定义类 &);

  ostream & operator << (ostream &,自定义类 &);
- 重载运算符“>>”的函数的第一个参数和函数的类型都必须是istream&类型，第二个参数是要进行输入操作的类。重载“<<”的函数的第一个参数和函数的类型都必须是ostream&类型，第二个参数是要进行输出操作的类。因此，只能将重载“>>”和“<<”的函数作为友元函数或普通的函数，而不能将它们定义为成员函数。

4.7.1 重载流插入运算符<<

- 例4.7
- 程序中重载了运算符“<<”，运算符重载函数中的形参output是ostream类对象的引用，形参名output是用户任意起的。分析main函数最后第二行：

  cout<<c3;

  运算符“<<”的左面是cout,前面已提到cout是ostream类对象。“<<”的右面是c3，它是Complex类对象。

- 由于已将运算符“<<”的重载函数声明为Complex类的友元函数，编译系统把“cout<<c3”解释为
  operator<<(cout,c3)
  即以cout和c3作为实参，调用下面的operator<<函数：
  ostream& operator<<(ostream& output,Complex& c)
  { output<<"("<<c.real<<"+"<<c.imag<<"i)";
  return output;}
  调用函数时，形参output成为cout的引用，形参c成为c3的引用。因此调用函数的过程相当于执行：
  cout<<"("<<c3.real<<"+"<<c3.imag<<"i)";
  return cout;
- 在已知cout<<c3的返回值是cout的当前值。如果有以下输出：
  cout<<c3<<c2;
  先处理cout<<c3，即
  (cout<<c3)<<c2;
  而执行(cout<<c3)得到的结果就是具有新内容的流对象cout，因此，(cout<<c3)<<c2相当于cout(新值)<<c2。运算符“<<”左侧是ostream类对象cout，右侧是Complex类对象c2,则再次调用运算符“<<”重载函数，接着向输出流插入c2的数据。
- 区分什么情况下的“<<”是标准类型数据的流插入符，什么情况下的“<<”是重载的流插入符。
  如：
  cout<<c3<<5<<endl;
  有下划线的是调用重载的流插入符，后面两个“<<”不是重载的流插入符，因为它的右侧不是Complex类对象而是标准类型的数据，是用预定义的流插入符处理的。
- 说明： 在本程序中，在Complex类中定义了运算符“<<”重载函数为友元函数，因此只有在输出Complex类对象时才能使用重载的运算符，对其他类型的对象是无效的。如
  cout<<time1; //time1是Time类对象，不能使用用于Complex类的重载运算符

4.7.2 重载流插入运算符>>

- 重载流提取运算符的目的是希望将“>>”用于输入自定义类型的对象的信息。
- 例4.8

4.8 不同类型数据间的转换

4.8.1 标准类型数据间的转换

- 在C++中，某些不同类型数据之间可以自动转换，

  例如

```
int i = 6;
i = 7.5 + i;
```

  编译系统对 7.5是作为double型数处理的，在求解表达式时，先将6转换成double型，然后与7.5相加，得到和为13.5，在向整型变量i赋值时，将13.5转换为整数13，然后赋给i。这种转换是由C++编译系统自动完成的，用户不需干预。这种转换称为隐式类型转换。
- C++还提供显式类型转换，程序人员在程序中指定将一种指定的数据转换成另一指定的类型，其形式为
  类型名(数据)
  如

```
int(89.5)
```

其作用是将89.5转换为整型数89。

对于用户自己声明的类型，编译系统并不知道怎样进行转换。解决这个问题的关键是让编译系统知道怎样去进行这些转换，需要定义专门的函数来处理。

4.8.2 用转换构造函数进行类型转换

- 几种构造函数：
    - 默认构造函数。以Complex类为例，函数原型的形式为

```
Complex( );//没有参数
```

    - 用于初始化的构造函数。函数原型的形式为

```
Complex(double r,double i);//形参表列中一般有两个以上参数
```

    - 用于复制对象的复制构造函数。函数原型的形式为

```
Complex (Complex &c);//形参是本类对象的引用
```

- 转换构造函数： 只有一个形参，如

```
 Complex(double r) {real=r;imag=0;}
```

其作用是将double型的参数r转换成Complex类的对象，将r作为复数的实部，虚部为0。用户可以根据需要定义转换构造函数，在函数体中告诉编译系统怎样去进行转换。

- 在类体中，可以有转换构造函数，也可以没有转换构造函数，视需要而定。以上几种构造函数可以同时出现在同一个类中，它们是构造函数的重载。编译系统会根据建立对象时给出的实参的个数与类型选择形参与之匹配的构造函数。
- 使用转换构造函数将一个指定的数据转换为类对象的方法如下：
  （1） 先声明一个类。
  （2） 在这个类中定义一个只有一个参数的构造函数，参数的类型是需要转换的类型，在函数体中指定转换的方法。
  （3） 在该类的作用域内可以用以下形式进行类型转换：
  类名(指定类型的数据)
  就可以将指定类型的数据转换为此类的对象。
- 也可以将另一个类的对象转换成转换构造函数所在的类对象。

```
Teacher(Student &s)
{
    num=s.num;
    strcpy(name,s.name);
    sex=s.sex;
}
```

4.8.3 用类型转换函数进行类型转换

- 用转换构造函数可以将一个指定类型的数据转换为类的对象。但是不能反过来将一个类的对象转换为一个其他类型的数据(例如将一个Complex类对象转换成double类型数据)。
- C++提供类型转换函数(type conversion function)来解决这个问题。类型转换函数的作用是将一个类的对象转换成另一类型的数据。

- 如果已声明了一个Complex类，可以在Complex类中这样定义类型转换函数：

```
operator double( )
    {return real;}
```

类型转换函数的一般形式为：
operator 类型名( )
{实现转换的语句}

- 在函数名前面**不能指定函数类型**，**函数没有参数。**其返回值的类型是由函数名中指定的类型名来确定的。类型转换函数只能作为**成员函数**，因为转换的主体是本类的对象。不能作为友元函数或普通函数。
- 从函数形式可以看到，它与运算符重载函数相似，都是用关键字operator开头，只是被重载的是类型名。double类型经过重载后，除了原有的含义外，还获得新的含义(将一个Complex类对象转换为double类型数据，并指定了转换方法)。
- 程序中的Complex类对具有双重身份，既是Complex类对象，又可作为double类型数据。Complex类对象只有在需要时才进行转换，要根据表达式的上下文来决定。

```
d1=d2+c1;
c2=c1+d2;//相当于c2=c1+Complex(d2);
```

- 转换构造函数和类型转换运算符有一个共同的功能： 当需要的时候，编译系统会自动调用这些函数，建立一个无名的临时对象(或临时变量)。
- 例4.9

分析：
(1) 如果在Complex类中没有定义类型转换函数operator double，程序编译将出错。
(2) 如果在main函数中加一个语句：
c3=c2;
由于赋值号两侧都是同一类的数据，是可以合法进行赋值的，没有必要把c2转换为double型数据。

(3) 如果在Complex类中声明了重载运算符“+”函数作为友元函数：
Complex operator+ (Complex c1,Complex c2) //定义运算符“+”重载函数
{return Complex(c1.real+c2.real, c1.imag+c2.imag);}

若在main函数中有语句
c3=c1+c2;
将c1和c2按Complex类对象处理，相加后赋值给同类对象c3。
如果改为
d=c1+c2; //d为double型变量
将c1与c2两个类对象相加，得到一个临时的Complex类对象，由于它不能赋值给double型变量，而又有对double的重载函数，于是调用此函数，把临时类对象转换为double数据，然后赋给d。

- 例4.10

分析
(1) 如果没有定义转换构造函数，则此程序编译出错。
(2) 在类Complex中定义了转换构造函数，并具体规定了怎样构成一个复数。由于已重载了算符“+”，在处理表达式c1+2.5时，编译系统把它解释为
operator+(c1,2.5)
上面的函数调用相当于
operator+(c1,Complex(2.5)) //运行结果为(5.5+4i)

(3) 如果把“c3=c1+2.5;”改为c3=2.5+c1; 程序可以通过编译和正常运行。过程与前相同。
在已定义了相应的转换构造函数情况下，将运算符“+”函数重载为友元函数，在进行两个复数相加时，可以用交换律。

如果运算符函数重载为成员函数，它的第一个参数必须是本类的对象。当第一个操作数不是类对象时，不能将运算符函数重载为成员函数。如果将运算符“+”函数重载为类的成员函数，交换律不适用。
由于这个原因，一般情况下将双目运算符函数重载为友元函数。单目运算符则多重载为成员函数。

(4) 如果一定要将运算符函数重载为成员函数，而第一个操作数又不是类对象时，只有一个办法能够解决，再重载一个运算符“+”函数，其第一个参数为double型。当然此函数只能是友元函数，函数原型为

```
friend operator+(double,Complex &);
```

显然这样做不太方便，还是将双目运算符函数重载为友元函数方便些。

(5) 在上面程序的基础上增加类型转换函数：

```
operator double( ){return real;}
```

此时Complex类的公用部分为

```
public:
  Complex( ){real=0;imag=0;}
  Complex(double r){real=r;imag=0;}  //转换构造函数
  Complex(double r,double i){real=r;imag=i;}
  operator double( ){return real;}     //类型转换函数
  friend Complex operator+ (Complex c1,Complex c2);
  void display( );
```

第五章 继承与派生

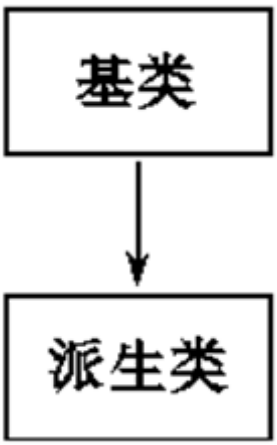
5.1 继承与派生的概念

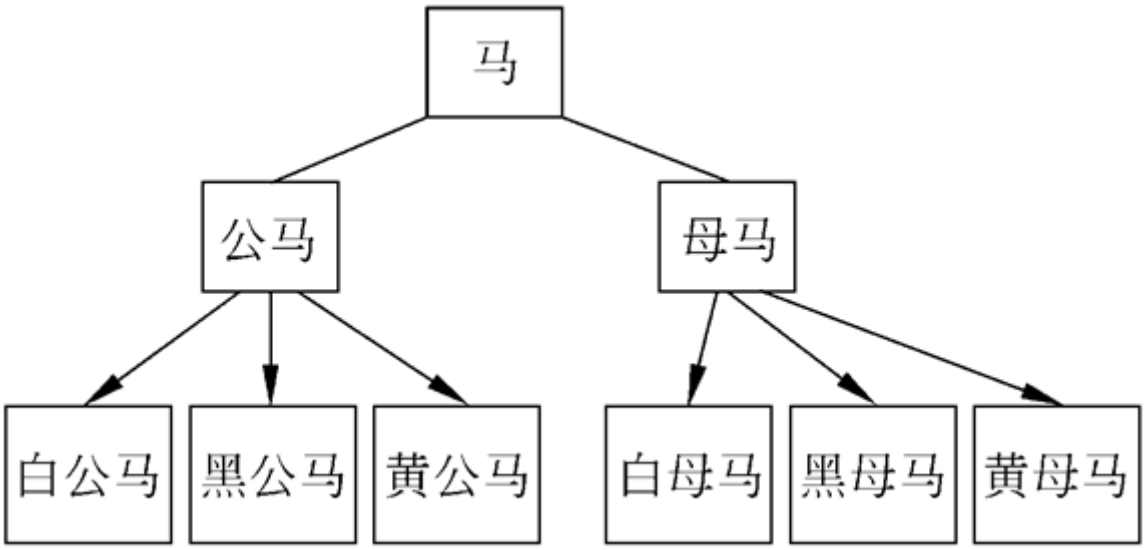
- C++中，软件可重用性是通过继承来实现的。

```
//例如：
class Student//声明Student类
{
    public:
        void display( )//定义成员函数
        {
            cout<<”num: ”<<num<<endl;
            cout<<”name: ”<<name<<endl;
            cout<<”sex: ”<<sex<<endl<<endl;
        }
    private:
        int num;
        string name;
        char sex;
};
class Student1//声明Student1类
{
    public:
        void display( )//定义成员函数
        {
            cout<<”num: ”<<num<<endl;
            cout<<”name: ”<<name<<endl;
            cout<<”sex: ”<<sex<<endl<<endl;
```

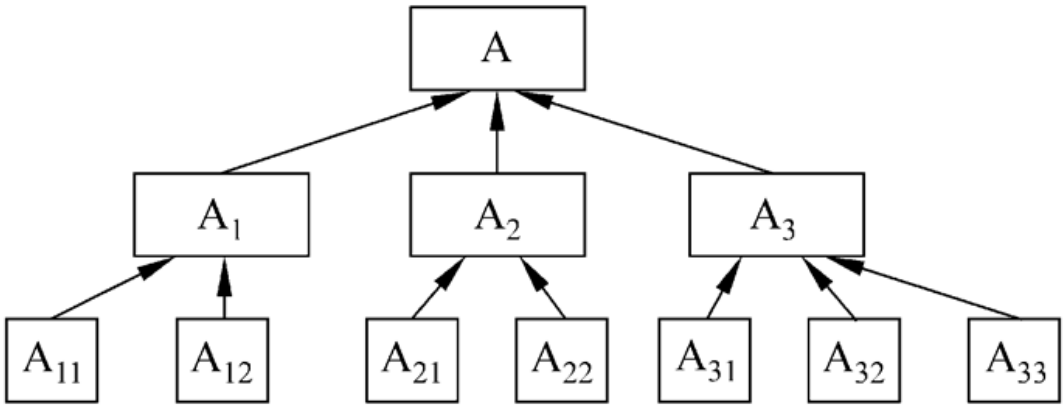
```
            cout<<"age: "<<sex<<endl<<endl;
            cout<<"address: "<<sex<<endl<<endl;
        }
    private:
        int num;
        string name;
        char sex;
        int age;
        char addr[20];
};
```

- 所谓"继承"就是在一个已存在的类的基础上建立一个新的类。
- 已存在的类称为"基类(base class)"或"父类(father class)"。新建立的类称为"派生类(derived class)"或"子类(son class)"。

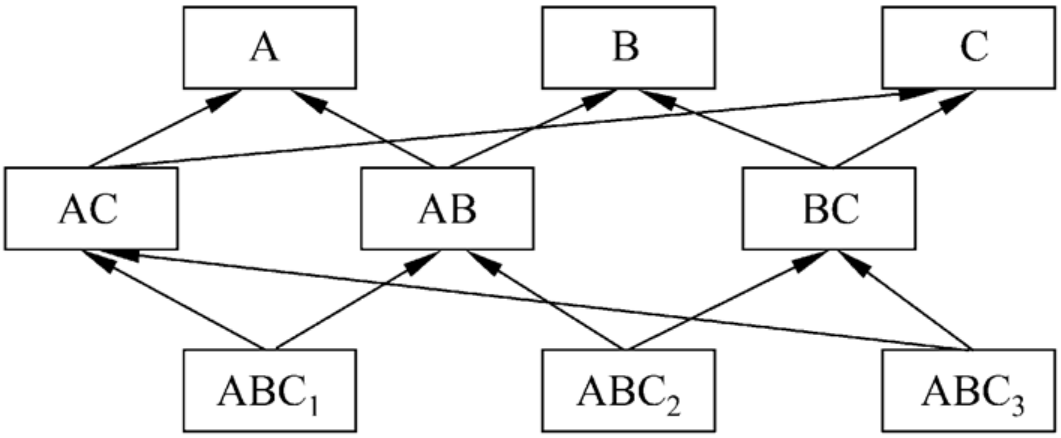




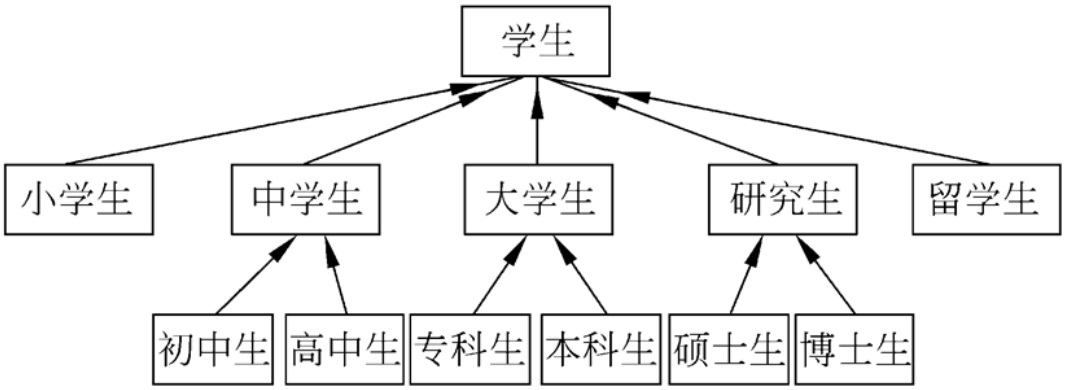
- 从已有的类(父类)产生一个新的子类，称为类的派生。派生类继承了基类的所有数据成员和成员函数，并可以对成员作必要的增加或调整。
- 一个基类可以派生出多个派生类，每一个派生类又可以作为基类再派生出新的派生类，因此基类和派生类是相对而言的。
- 一个派生类只从一个基类派生，这称为单继承(single inheritance)，这种继承关系所形成的层次是一个树形结构



- 一个派生类不仅可以从一个基类派生，也可以从多个基类派生。一个派生类有两个或多个基类的称为多重继承(multiple inheritance)



- 基类和派生类的关系为: 派生类是基类的具体化，而基类则是派生类的抽象。



5.2 派生类的声明方式

假设已经声明了一个基类Student，在此基础上通过单继承建立一个派生类Student1:

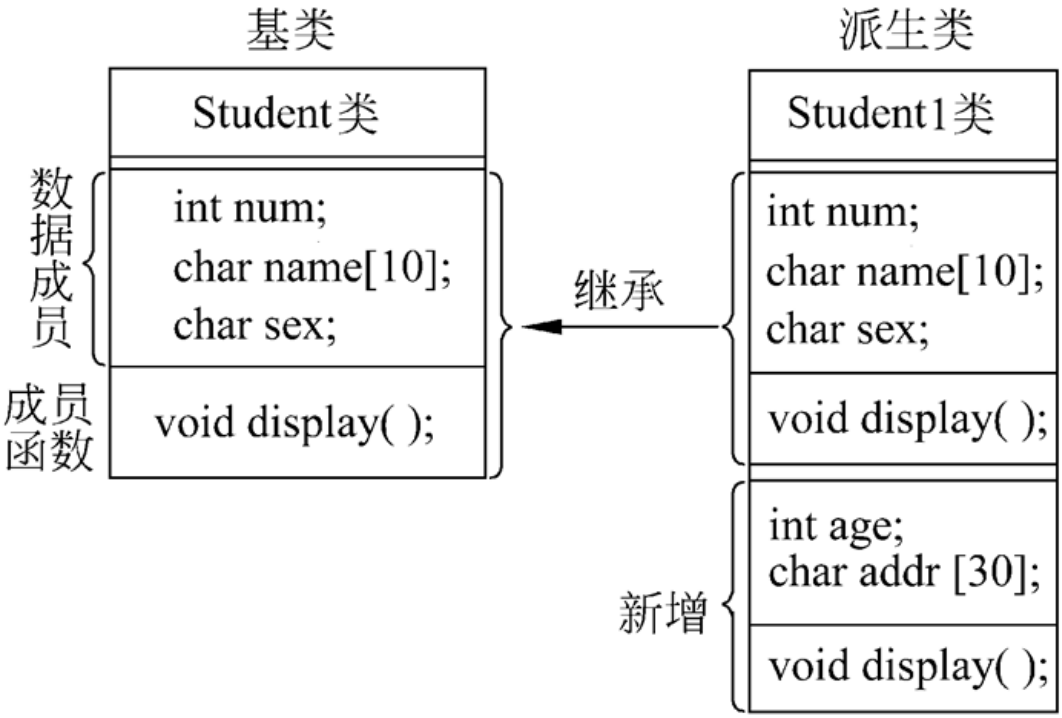
```
class Student1: public Student//声明基类是Student
{
    public:
        void display_1( )//新增加的成员函数
        {
            cout<<"age: "<<age<<endl;
            cout<<"address: "<<addr<<endl;
        }
    private:
        int age;                                    //新增加的数据成员
        string addr;                                //新增加的数据成员
};
```

- 基类名前面有public的称为“公用继承(public inheritance)”。
- 声明派生类的一般形式为

  class 派生类名: ［继承方式］ 基类名
  {
  派生类新增加的成员
  };
- 继承方式包括: public(公用的),private(私有的)和protected(受保护的)，此项是可选的，如果不写此项，则默认为private(私有的)。

5.3 派生类的构成

- 派生类中的成员包括**从基类继承过来的成员**和**自己增加的成员**两大部分。

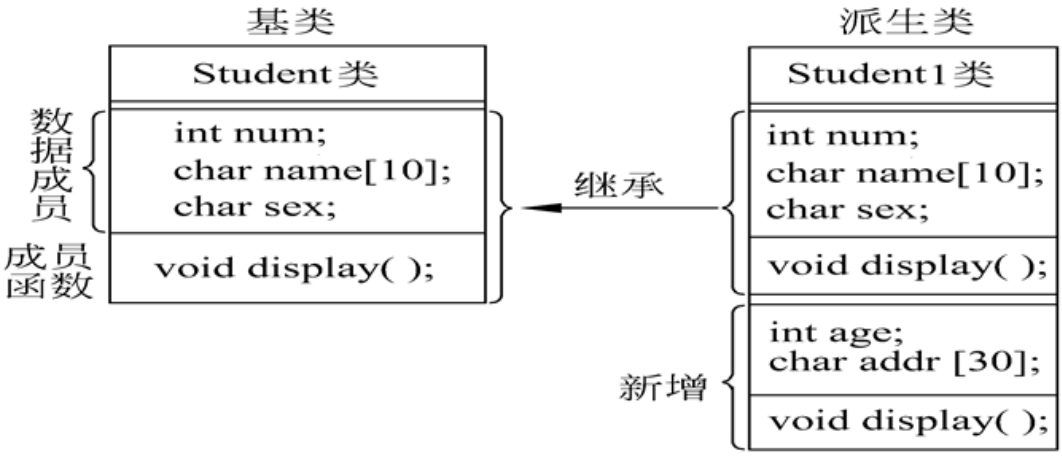


- 构造一个派生类包括以下3部分工作:
  - **从基类接收成员。**派生类把基类全部的成员(不包括构造函数和析构函数)接收过来，也就是说是没有选择的接受。
    注意：慎重选择基类，避免冗余数据。
  - **调整从基类接收的成员。**接收基类成员是不能选择的，但可以对这些成员作某些调整。比如改变基类成员在派生类中的访问属性。
    注意：派生类和基类的同名函数。
  - **在声明派生类时增加的成员。**这部分内容体现了派生类对基类功能的扩展。
    例如：display()和display1()
  - 在声明派生类时，一般还应当自己定义派生类的构造函数和析构函数，因为**构造函数和析构函数是不能从基类继承的。**

5.4 派生类成员的访问属性

- 对基类成员和派生类自己增加的成员是按不同的原则处理的。
  - 基类的成员函数访问基类成员。(√)
  - 派生类的成员函数访问派生类自己增加的成员。(√)
  - 基类的成员函数访问派生类的成员。(×)
  - 派生类的成员函数访问基类的成员。(?)
  - 在派生类外访问派生类的成员。(可访问公有成员)

- 在派生类外访问基类的成员。(？)



- 不同的继承方式决定了基类成员在派生类中的访问属性。
    - **公用继承**(public inheritance)：基类的公用成员和保护成员在派生类中保持原有访问属性，其私有成员仍为基类私有。
    - **私有继承**(private inheritance)：基类的公用成员和保护成员在派生类中成了私有成员。其私有成员仍为基类私有。
    - **受保护的继承**(protected inheritance)：基类的公用成员和保护成员在派生类中成了保护成员，其私有成员仍为基类私有。保护成员的意思是: 不能被外界引用，但可以被派生类的成员引用。

5.4.1 公用继承

- 在定义一个派生类时将基类的继承方式指定为public的，称为公用继承，用公用继承方式建立的派生类称为公用派生类(public derived class)，其基类称为公用基类(public base class)。
- 公用基类在派生类中的访问属性

公用基类的成员	在公用派生类中的访问属性
私有成员	**不可访问**
公用成员	**公用**
保护成员	**保护**

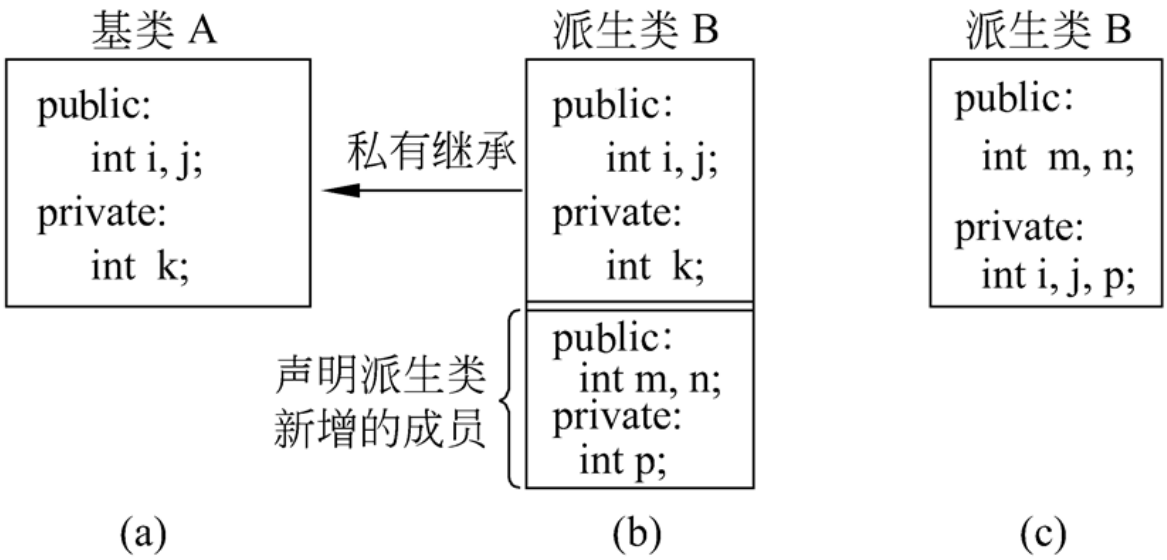
```
//例5.1
//访问公有基类的成员。
Class Student//声明基类
{
    public://基类公用成员
        void get_value( )
        {
            cin>>num>>name>>sex;
        }
        void display( )
        {
            cout<<" num: "<<num<<endl;
            cout<<" name: "<<name<<endl;
            cout<<" sex: "<<sex<<endl;
        }
    private://基类私有成员
        int num;
        string name;
```

```
        char sex;
};
class Student1: public Student//以public方式声明Student1
{
    public:
        void display_1( )
        {
            cout<<" num: "<<num<<endl;  //引用基类私有成员错误
            cout<<" name: "<<name<<endl; //错误
            cout<<" sex: "<<sex<<endl; //错误
            cout<<" age: "<<age<<endl;  //引用派生类的私有成员正确
            cout<<" address: "<<addr<<endl;//正确
        }
  private:
      int age;
      string addr;
 };
```

5.4.2 私有继承

- 在声明一个派生类时将基类的继承方式指定为private的，称为私有继承，用私有继承方式建立的派生类称为私有派生类(private derived class)，其基类称为私有基类(private base class)。
- 私有基类在派生类中的访问属性

私有基类中的成员	在私有派生类中的访问属性
私有成员	不可访问
公用成员	私有
保护成员	私有




```
//例5.2
//将例5.1中的公用继承方式改为用私有继承方式(基类Student不改)。
class Student1: private Student//用私有继承方式声明Student1
{  public:
       void display_1( )                          //输出两个数据成员的值
       {cout<<"age: "<<age<<endl;
        cout<<"address: "<<addr<<endl;}
```

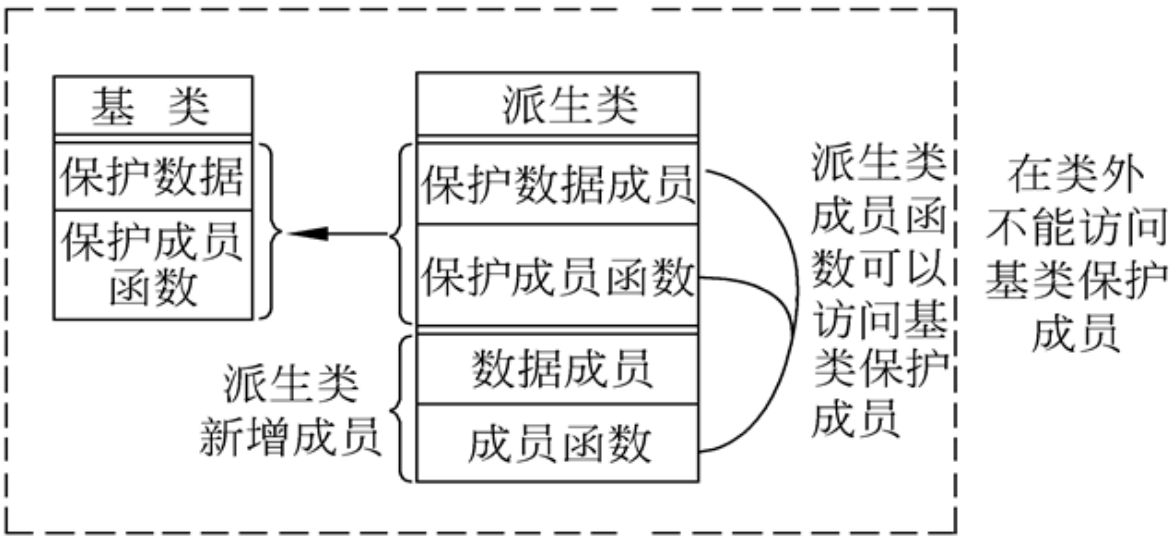
```
   private:
       int age;
       string addr;
};
int main( )
{
    Student1  stud1;//定义一个Student1类的对象stud1
    stud1.display();//错误，私有基类的公用成员函数在派生类中是私有函数
    stud1.display_1( );//正确。display_1函数是Student1类的公用函数
    stud1.age=18;//错误。外界不能引用派生类的私有成员
    return 0;
}
```

- 不能在类外通过派生类对象(如stud1)引用从私有基类继承过来的任何成员(如stud1.display()或stud1.num)。
- 派生类的成员函数不能访问私有基类的私有成员，但可以访问私有基类的公用成员(如stud1.display_1函数可以调用基类的公用成员函数display，但不能引用基类的私有成员num)。
- 可以通过派生类的成员函数调用私有基类的公用成员函数

```
void display_1( )//输出5个数据成员的值
{
    display();  //调用基类的公用成员函数，输出3个数据成员
    cout<<"age: "<<age<<endl; //输出派生类的私有数据成员
    cout<<"address: "<<addr<<endl;
}
int main( )
{
    Student1 stud1;
    stud1.display_1( );
    return 0;
}
```

5.4.3 保护成员和保护继承

- 由protected声明的成员称为"保护成员"。从类的用户角度来看，保护成员等价于私有成员。但有一点与私有成员不同，**保护成员可以被派生类的成员函数引用。**



- 如果基类声明了私有成员，那么任何派生类都是不能访问它们的，若希望在派生类中能访问它们，应当把它们声明为保护成员。
- 在定义一个派生类时将基类的继承方式指定为protected的，称为保护继承，用保护继承方式建立的派生类称为保护派生类(protected derived class)，其基类称为受保护的基类(protected base class)，简称保护基类。
- 保护继承的特点是: 保护基类的公用成员和保护成员在派生类中都成了保护成员，其私有成员仍为基类私有。也就是把基类原有的公用成员也保护起来，不让类外任意访问。
- 保护基类在派生类中的访问属性

保护基类中的成员	在保护派生类中的访问属性
私有成员	不可访问
公用成员	保护
保护成员	保护

- 私有继承和保护继承的比较
    - 在**直接派生类**中，以上两种继承方式的作用是相同的: 在类外不能访问任何成员，而在派生类中可以通过成员函数访问基类中的公用成员和保护成员。

```
class A
{
    public:
        int a,b;
    private:
        int c,d;
    protected:
        int g,h;
}
```

```
class B: private A
{
    public:
        int m,n;
    private:
        int i,j;
}

//派生类B:
public:
    int m,n;
private:
    int a,b,i,j,g,h;
```

```
class C: protected A
{
    public:
        int m,n;
    private:
        int i,j;
}

//派生类C:
```

```
public:
    int m,n;
private:
    int i,j;
protected:
    int a,b,g,h;
```

- 在**继续派生**中，如果以公用继承方式派生出一个新派生类，原来私有基类中的成员在新派生类中都成为不可访问的成员，无论在派生类内或外都不能访问，而原来保护基类中的公用成员和保护成员在新派生类中为保护成员，可以被新派生类的成员函数访问。

```
class D: public B
{
    public:
        int x,y;
    private:
        int k,l;
}

//派生类D:
public:
    int m,n,x,y;
private:
    int k,l;
```

```
class E: public B
{
    public:
        int x,y;
    private:
        int k,l;
}

//派生类E:
public:
    int m,n,x,y;
private:
    int k,l;
protected:
    int a,b,g,h;
```

```
//例5.3
//在派生类中引用保护成员。

#include <iostream>
#include <string>
using namespace std;
class Student//声明基类
{
    public://基类公用成员
        void display( );
    protected://基类保护成员
        int num;
        string name;
```

```
        char sex;
};
void Student::display( )//定义基类成员函数
{
    cout<<"num: "<<num<<endl;
    cout<<"name: "<<name<<endl;
    cout<<"sex: "<<sex<<endl;
}
class Student1: protected Student//用protected方式声明Student1
{
    public:
        void display1( );//派生类公用成员函数
    private:
        int age;//派生类私有数据成员
        string addr;//派生类私有数据成员
};
void Student1::display1( )//定义派生类公用成员函数
{
    cout<<"num: "<<num<<endl;//引用基类的保护成员，合法
    cout<<"name: "<<name<<endl;//合法
    cout<<"sex: "<<sex<<endl;//合法
    cout<<"age: "<<age<<endl;//合法
    cout<<"address: "<<addr<<endl;//合法
}
int main( )
{
    Student1 stud1;//stud1是派生类Student1类的对象
    stud1.display1( );//合法，display1是派生类中的公用成员函数
    stud1.num=10023;//错误，外界不能访问保护成员
    return 0;
 }
```

- 基类成员在派生类中的访问属性

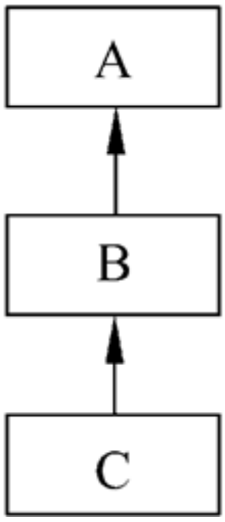
基类中的成员	在公用派生类中的访问属性	在私有派生类中的访问属性	在保护派生类中的访问属性
私有成员	不可访问	不可访问	不可访问
公用成员	公用	私有	保护
保护成员	保护	私有	保护

- 派生类中的成员的访问属性

派生类中的成员	在派生类中	在派生类外部	在下层公用派生类中
派生类中访问属性为公用的成员	可以	可以	可以
派生类中访问属性为保护的成员	可以	不可以	可以
派生类中访问属性为私有的成员	可以	不可以	不可以
在派生类中不可访问的成员	不可以	不可以	不可以

5.4.4 多级派生时的访问属性

- 类A为基类，类B是类A的派生类，类C是类B的派生类，则类C也是类A的派生类。类B称为类A的直接派生类，类C称为类A的间接派生类。类A是类B的直接基类，是类C的间接基类。




```
//例5.4
//多级派生的访问属性。
如果声明了以下的类:
class A//基类
{
    public:
        int i;
    protected:
        void f2( );
        int j;
    private:
        int k;
};
class B: public A
{
    public:
        void f3( );
     protected:
        void f4( );
     private:
         int m;
};
class C: protected B
```

```
{
    public:
        void f5( );
    private:
        int n;
};
```

类A是类B的公用基类，类B是类C的保护基类。各成员在不同类中的访问属性如下:

	i	f2	j	k	f3	f4	m	f5	n
基类A	公用	保护	保护	私有					
公用派生B	公用	保护	保护	不可访问	公用	保护	私有		
保护派生C	保护	保护	保护	不可访问	保护	保护	不可访问	公用	私有

5.5 派生类的构造函数和析构函数

在执行派生类的构造函数时，使派生类的数据成员和基类的数据成员同时都被初始化。解决这个问题的思路是: 在执行派生类的构造函数时，调用基类的构造函数。

5.5.1 简单的派生类的构造函数

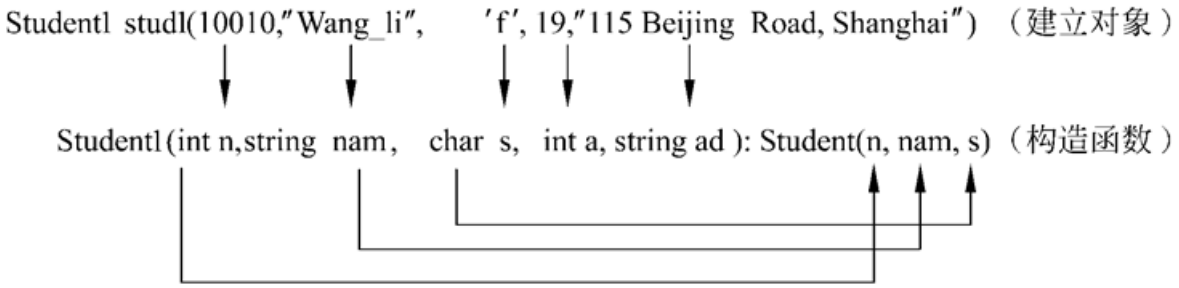
- 任何派生类都包含基类的成员，简单的派生类只有一个基类，而且只有一级派生(只有直接派生类，没有间接派生类)，在派生类的数据成员中不包含基类的对象(即子对象)。

```
//例5.5
//简单的派生类的构造函数。

#include <iostream>
#include<string>
using namespace std;
class Student//声明基类Student
{
    public:
        Student(int n,string name,char s)//基类构造函数
        {
            num=n;
            name=nam;
            sex=s;
        }
        ~Student( ){ }//基类析构函数
    protected://保护部分
        int num;
        string name;
        char sex ;
};
class Student1: public Student//声明派生类Student1
{
    public://派生类的公用部分
        Student1(int n,string name,char s,int
a,stringad):Student(n,nam,s)//派生类构造函数
```

```
        {
            age=a;//在函数体中只对派生类新增的数据成员初始化
            addr=ad;
        }
        void show( )
        {
            cout<<"num: "<<num<<endl;
            cout<<"name: "<<name<<endl;
            cout<<"sex: "<<sex<<endl;
            cout<<"age: "<<age<<endl;
            cout<<"address: "<<addr<<endl<<endl;
        }
        ~Student1( ){ }//派生类析构函数
    private://派生类的私有部分
        int age;
        string addr;
};
int main( )
{
    Student1 stud1(10010,"Wang-li",'f',19,"115 BeijingRoad,Shanghai");
    Student1 stud2(10011,"Zhang-fun",'m',21,"213Shanghai Road,Beijing");
    stud1.show( );//输出第一个学生的数据
    stud2.show( );//输出第二个学生的数据
    return 0;
}
```

- 派生类构造函数其一般形式为
  派生类构造函数名(总参数表列): 基类构造函数名(参数表列)
  { 派生类中新增数据成员初始化语句 }

```
//如:
Student1(int n, string nam, char s, int a, string ad):Student(n, nam,
s)
```



- 派生类构造函数的其它形式

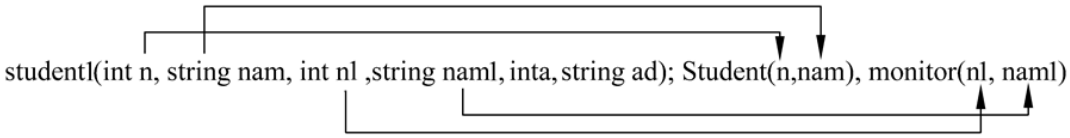
```
Student1(int n,string nam,char s,int a, string ad): Student(n,nam,s)

Student1(int n,string nam,char s,int a, string
ad):Student(n,nam,s),age(a),addr(ad){}
```

- 在建立一个对象时，执行构造函数的顺序是:
  - ①派生类构造函数先调用基类构造函数;

    - ②再执行派生类构造函数本身(即派生类构造函数的函数体)。
- 在派生类对象释放时，先执行派生类析构函数，再执行其基类析构函数。

5.5.2 有子对象的派生类的构造函数

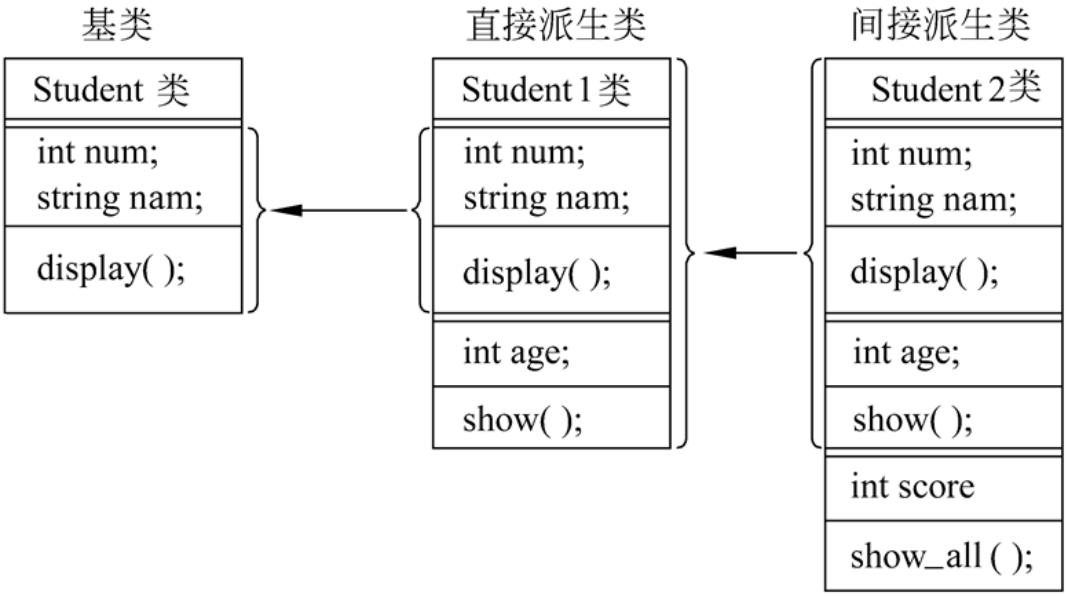
- 类的数据成员中还可以包含类对象，如可以在声明一个类时包含这样的数据成员:
  Student s1;// Student是已声明的类名，s1是Student类的对象
  这时，s1就是类对象中的内嵌对象，称为子对象(subobject)，即对象中的对象。
- 包含子对象的派生类的构造函数
- 例5.6
- 子对象的初始化是在建立派生类时通过调用派生类构造函数来实现的。
- 派生类的构造函数的任务包括三个部分：
    - 对基类数据成员初始化
    - 对子对象数据成员初始化
    - 对派生类数据成员初始化
- 派生类函数首部
  Student1(int n,string nam,int n1,string nam1,int a,string ad):
  Student(n,nam),monitor(n1,nam1)



- 定义派生类的构造函数的一般形式为：
  派生类构造函数名(总参数列表):基类构造函数名(参数列表),子对象名(参数列表){派生类中新增数据成员初始化语句}
- 执行派生类构造函数的顺序是：
    - 调用基类构造函数，对基类数据成员初始化；
    - 调用子对象构造函数，对子对象数据成员初始化；
    - 再执行派生类构造函数本身，对派生类数据成员初始化。
- 函数首部中，基类构造函数和子对象的次序可以是任意的
  Student1(int n,string nam,int n1,string nam1,int a,string ad): monitor(n1,nam1)
  ,Student(n,nam)

5.5.3 多层派生时的构造函数

- 例5.7
- 派生关系如图：



- 不需要列出每一层派生类的构造函数，只须写出其上一层的派生类的构造函数。
- 初始化的顺序：
    - 先初始化基类的数据成员；
    - 再初始化直接派生类的数据成员；
    - 最后再初始化间接派生类的数据成员。

5.5.4 派生类构造函数的特殊形式

- 当不需要对派生类新增的成员进行任何初始化操作时，派生类构造函数的函数体可以为空，即构造函数是空函数，如
  Student1(int n, strin nam,int n1, string nam1): Student(n,nam),monitor(n1,nam1) { }
  此派生类构造函数的作用只是为了将参数传递给基类构造函数和子对象，并在执行派生类构造函数时调用基类构造函数和子对象构造函数。
- 如果在基类中没有定义构造函数，或定义了**没有参数**的构造函数，那么在定义派生类构造函数时可不写基类构造函数。因为此时派生类构造函数没有向基类构造函数传递参数的任务。调用派生类构造函数时系统会自动首先调用基类的默认构造函数。

5.5.5 派生类的析构函数

- 在派生时，派生类是不能继承基类的析构函数的，也需要通过派生类的析构函数去调用基类的析构函数。在派生类中可以根据需要定义自己的析构函数，用来对派生类中所增加的成员进行清理工作。基类的清理工作仍然由基类的析构函数负责。在执行派生类的析构函数时，系统会自动调用基类的析构函数和子对象的析构函数，对基类和子对象进行清理。
- 调用的顺序与构造函数正好相反: 先执行派生类自己的析构函数，对派生类新增加的成员进行清理，然后调用子对象的析构函数，对子对象进行清理，最后调用基类的析构函数，对基类进行清理。

5.6 多重继承

一个派生类有两个或多个基类，派生类从两个或多个基类中继承所需的属性。这种行为称为多重继承(multiple inheritance)。

5.6.1 声明多重继承的方法

- 如果已声明了类A、类B和类C，可以声明多重继承的派生类D:
  class D:public A,private B,protected C
  {类D新增加的成员}
  D按不同的继承方式的规则继承A,B,C的属性，确定各基类的成员在派生类中的访问权限。

5.6.2 多重继承派生类的构造函数

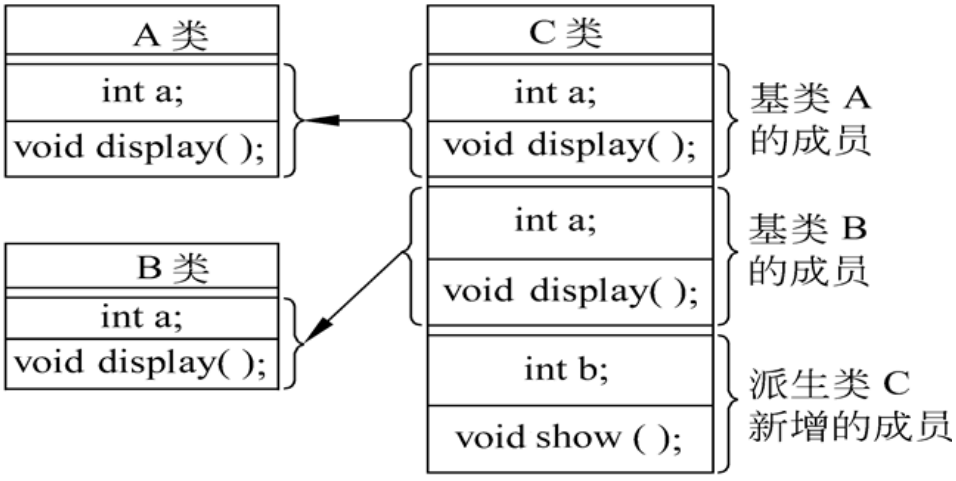
- 多重继承派生类的构造函数形式与单继承时的构造函数形式基本相同，只是在初始表中包含多个基类构造函数。如
  派生类构造函数名(总参数表列): 基类1构造函数(参数表列), 基类2构造函数(参数表列), 基类3构造函数 (参数表列)
  {派生类中新增数成员据成员初始化语句}
- 各基类的排列顺序任意
- 派生类构造函数的执行顺序为: 先调用基类的构造函数，再执行派生类构造函数。调用基类构造函数的顺序是按照声明派生类时基类出现的顺序。

```
//例5.8
//声明一个教师(Teacher)类和一个学生(Student)类，用多重继承的方式声明一个研究生(Graduate)派生类。

#include <iostream>
#include <string>
using namespace std;
class Teacher//声明类Teacher(教师)
{
    public://公用部分
        Teacher(string nam,int a, string t)//构造函数
        {
            name=nam;
            age=a;
            title=t;
        }
        void display( )//输出教师有关数据
        {
            cout<<"name:"<<name<<endl;
            cout<<"age"<<age<<endl;
            cout<<"title:"<<title<<endl;
        }
    protected://保护部分
        string name;
        int age;
        string title;
};
class Student//定义类Student(学生)
{
    public:
        Student(char nam[],char s,float sco)
        {
            strcpy(name1,nam);
            sex=s;
            score=sco;
        }//构造函数
        void display1( )//输出学生有关数据
        {
            cout<<"name:"<<name1<<endl;
            cout<<"sex:"<<sex<<endl;
            cout<<"score:"<<score<<endl;
        }
    protected://保护部分
        string name1;
        char sex;
```

```
        float score;//成绩
 };
class Graduate:public Teacher,public Student//声明多重继承的派生类Graduate
{
    public:
        Graduate(string nam,int a,char s, string t,float sco,float
w):Teacher(nam,a,t),Student(nam,s,sco),wage(w){ }
        void show( )//输出研究生的有关数据
        {
            cout<<"name:"<<name<<endl;
            cout<<"age:"<<age<<endl;
            cout<<"sex:"<<sex<<endl;
            cout<<"score:"<<score<<endl;
            cout<<"title:"<<title<<endl;
            cout<<"wages:"<<wage<<endl;
        }
    private:
        float wage;//工资
};
int main( )
{
    Graduate grad1("Wangli",24,'f',"assistant",89.5,1234.5);
    grad1.show( );
    return 0;
}
```

5.6.3 多重继承引起的二义性问题

- 如果类A和类B中都有成员函数display和数据成员a，类C是类A和类B的直接派生类。
    - 两个基类有同名成员




```
class A
{
    public:
        int a;
        void display( );
};
class B
{
    public:
        int a;
        void display( );
};
```

```
class C :public A,public B
{
    public:
        int b;
        void show();
};
```

如果在main函数中定义C类对象c1，并调用数据成员a和成员函数display:

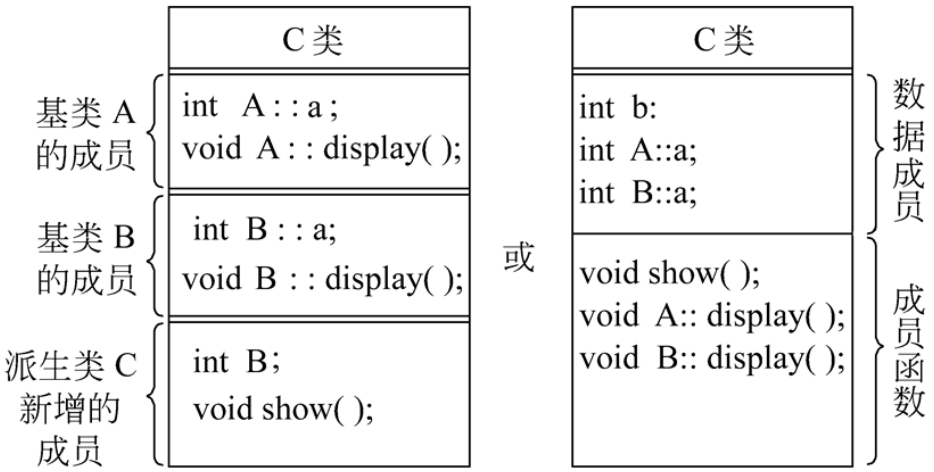
```
C c1;
c1.a=3;
c1.display();
```

可以用基类名来限定:

```
c1.A::a=3;//引用c1对象中的基类A的数据成员a
c1.A::display(); //调用c1对象中的基类A的成员函数display
```

如果是在派生类C中通过派生类成员函数show访问基类A的display和a，可以不必写对象名而直接写

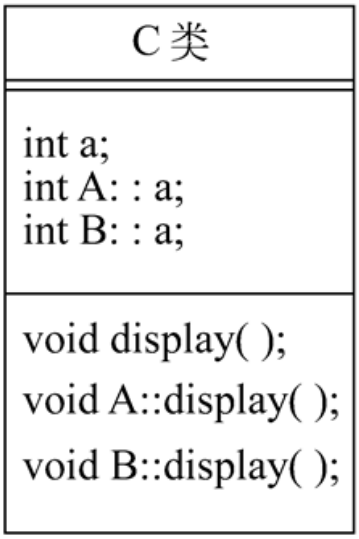
```
A::a=3;//指当前对象
A::display( );
```



- 两个基类和派生类三者有同名成员

  将上面的C类声明改为

```
class C :public A,public B
{
    int a;
    void display();
};
```



如果在main函数中定义C类对象c1，并调用数据成员a和成员函数display:

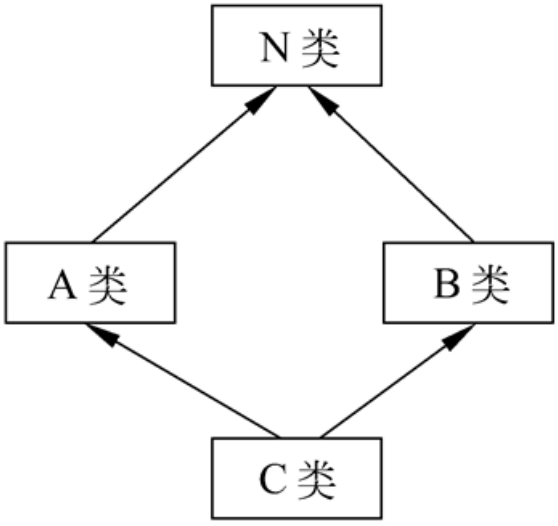
```
C c1;
c1.a=3;
c1.display( );
```

规则是: 基类的同名成员在派生类中被屏蔽，成为“不可见”的，或者说，派生类新增加的同名成员覆盖了基类中的同名成员。因此如果在定义派生类对象的模块中通过对象名访问同名的成员，则访问的是派生类的成员。不同的成员函数，只有在函数名和参数个数相同、类型相匹配的情况下才发生同名覆盖，如果只有函数名相同而参数不同，不会发生同名覆盖，而属于函数重载。

要在派生类外访问基类A中的成员，应指明作用域A，写成以下形式:

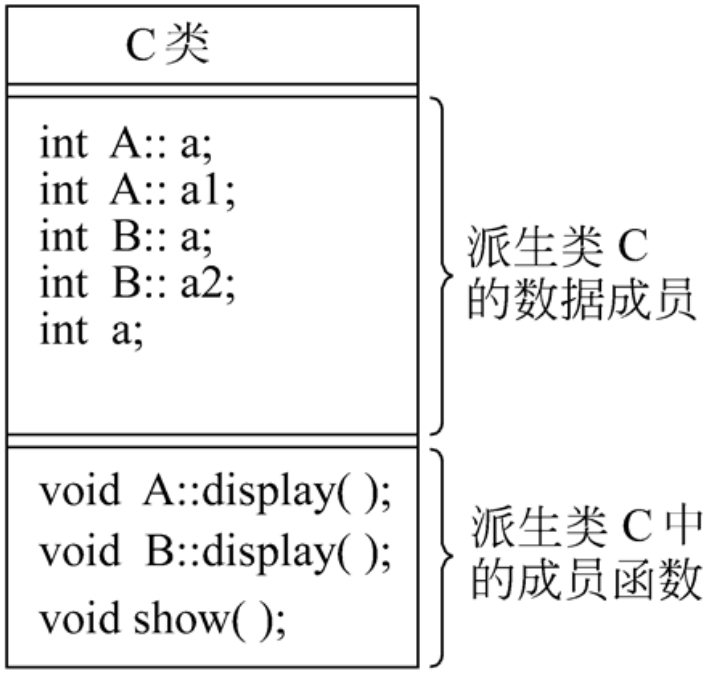
```
c1.A::a=3;//表示是派生类对象c1中的基类A中的数据成员a
c1.A::display();//表示是派生类对象c1中的基类A中的成员函数display
```

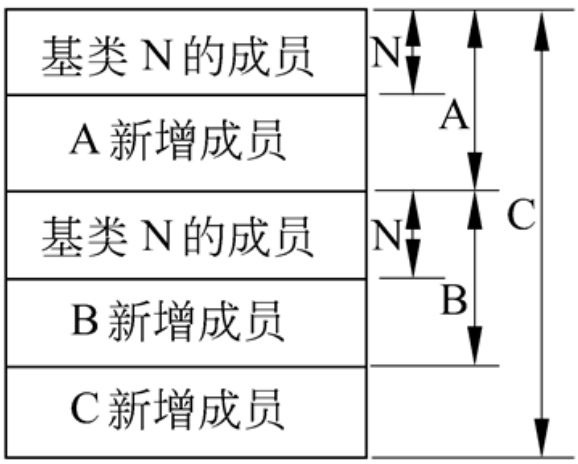
- 类A和类B是从同一个基类派生的




```
class N
{
    public:
        int a;
        void display(){cout<<”A::a=”<<a<<endl;}
};
class A:public N
```

```
{
    public:
        int a1;
};
class B:public N
{
    public:
        int a2;
};
class C :public A,public B
{
    public:
        int a3;
        void show( ){cout<<"a3="<<a3<<endl;}
};
int main( )
{
    C c1;//定义C类对象c1
    /* ... */
}
```





```
c1.a=3;c1.display( );
或
c1.N::a=3;
c1.N::display();
```

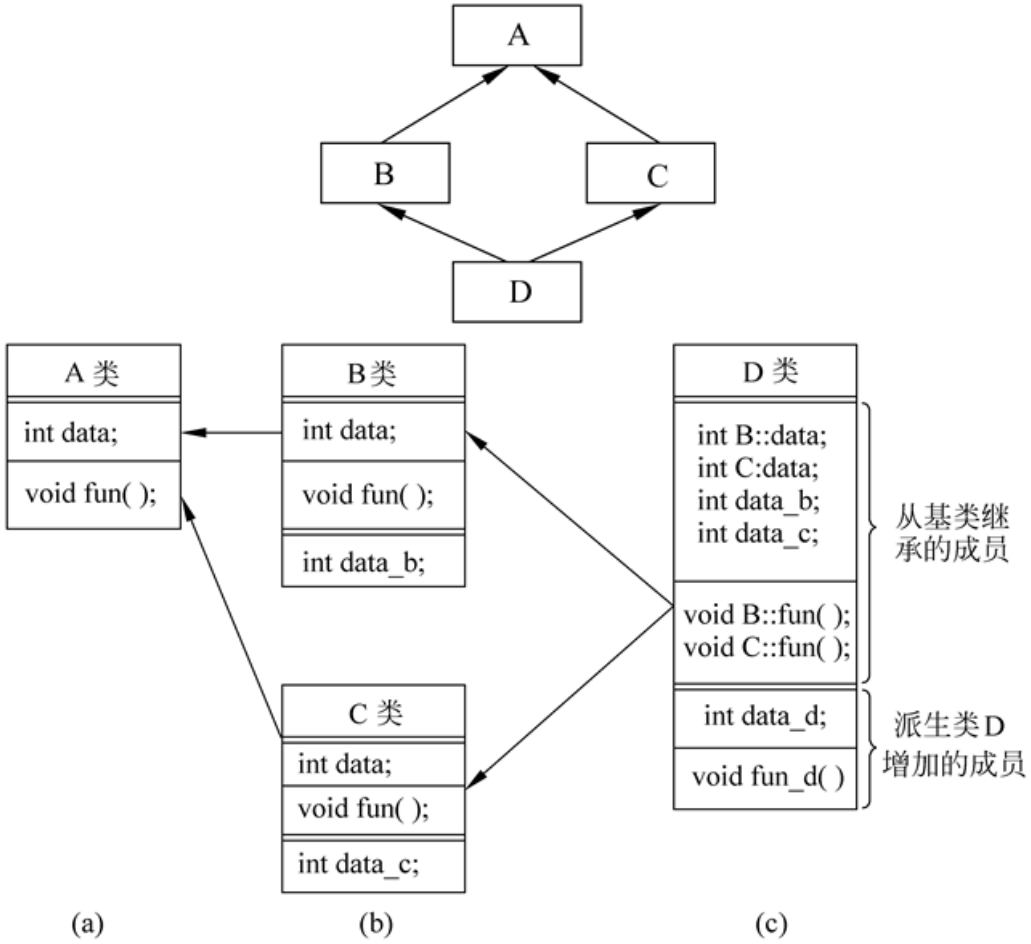
因为这样依然无法区别是类A中从基类N继承下来的成员，还是类B中从基类N继承下来的成员。应当通过类N的直接派生类名来指出要访问的是类N的哪一个派生类中的基类成员。如

```
c1.A::a=3; c1.A::display();//要访问的是类N的派生类A中的基
```

5.6.4 虚基类

1. 虚基类的作用

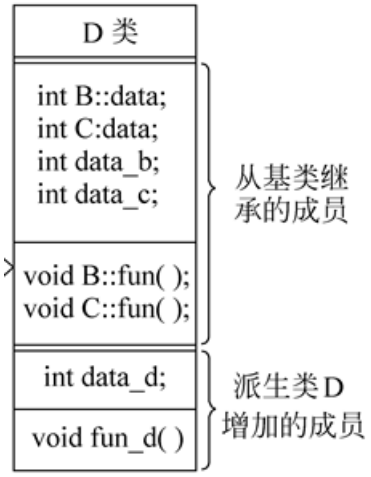
- 一个派生类有多个直接基类，而直接基类又有一个共同的基类，则最终在派生类中会保留该间接共同基类数据成员的多份同名成员。
- 在引用的时候容易引起二义性，必须使用作用域标。识符唯一标识。
- C++提供虚基类的方法，使得在继承间接共同基类时只保留一份成员。



- 现在，将类A声明为虚基类，方法如下:

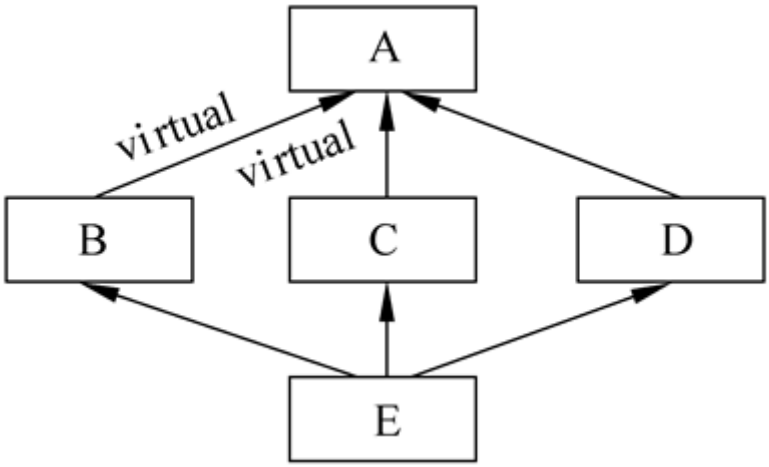
```
class A//声明基类A
{/* ... */};
class B :virtual public A//声明类B是类A的公用派生类，A是B的虚基类
{/* ... */};
class C :virtual public A//声明类C是类A的公用派生类，A是C的虚基类
{/* ... */};
```

- 注意: **虚基类并不是在声明基类时声明的，而是在声明派生类时，指定继承方式时声明的。**因为一个基类可以在生成一个派生类时作为虚基类，而在生成另一个派生类时不作为虚基类。
- 声明虚基类的一般形式：
  class 派生类名:virtual 继承方式 基类名
- 当基类通过多条派生路径被一个派生类继承时，该派生类只继承该基类一次。
- 在派生类B和C中作了上面的虚基类声明后，派生类D中的成员如图所示。



D类
int data; int data_b; int data_c ;
void fun( );
int data_d;
void fun_d( );

- 需要注意: 为了保证虚基类在派生类中只继承一次，应当在该基类的所有直接派生类中声明为虚基类。否则仍然会出现对基类的多次继承。



2. 虚基类的初始化

- 如果在虚基类中定义了带参数的构造函数，而且没有定义默认构造函数，则**在其所有派生类**(包括直接派生或间接派生的派生类)中，通过构造函数的初始化表对虚基类进行初始化。
-
```
1 //例如
2 class A//定义基类A
```

```
{
    A(int i){ }
    /* ... */
};
class B:virtual public A//A作为B的虚基类
{
    B(int n):A(n){ }
    /* ... */
};
class C:virtual public A//A作为C的虚基类
{
    C(int n):A(n){ }
    /* ... */
};
class D:public B,public C
{
    D(int n):A(n),B(n),C(n){ }
    /* ... */
};
```

- 规定: **在最后的派生类中不仅要负责对其直接基类进行初始化，还要负责对虚基类初始化。**

3. 虚基类的简单应用举例

- 例5.9

5.7 基类与派生类的转换

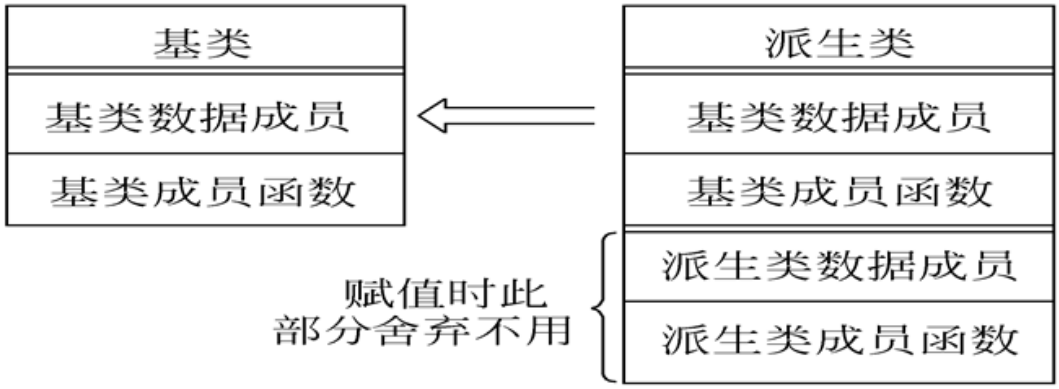
- 只有公用派生类才是基类真正的子类型，它完整地继承了基类的功能。
- 基类与派生类对象之间有赋值兼容关系，由于派生类中包含从基类继承的成员，因此可以**将派生类**的值赋给基类对象，在用到基类对象的时候可以用其子类对象代替。

5.7.1 派生类对象可以向基类对象赋值

- 基类与派生类之间也有赋值兼容关系。

```
A a1;//定义基类A对象a1
B b1;//定义类A的公用派生类B的对象b1
a1=b1;//用派生类B对象b1对基类对象a1赋值
```

- 在赋值时舍弃派生类自己的成员。
- 所谓赋值只是对数据成员赋值，对成员函数不存在赋值问题。



- 赋值后不能企图通过对象a1去访问派生类对象b1的成员，因为b1的成员与a1的成员是不同的。

```
a1.age=23;
b1.age=21;
```

- 不能用基类对象对其子类对象赋值，因为基类对象不包含派生类的成员。
- 同一基类的不同派生类对象之间也不能赋值，因为包含的成员不同。

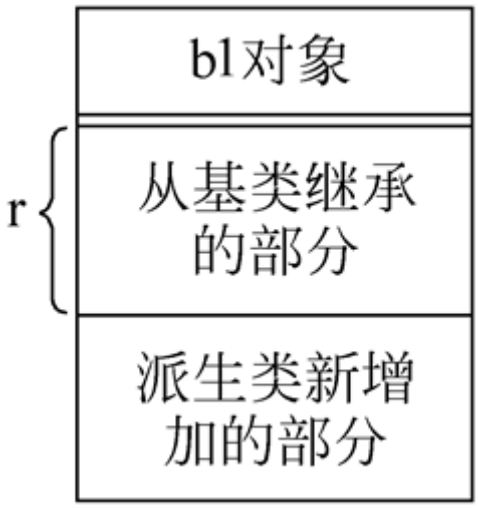
5.7.2 派生类对象可以替代基类对象向基类对象的引用进行赋值或初始化

- 如已定义了基类A对象a1，可以定义a1的引用变量:

```
A a1;//定义基类A对象a1
B b1;//定义公用派生类B对象b1
A& r=a1;
```

这时，引用变量r是a1的别名，r和a1共享同一段存储单元。也可以用子类对象初始化引用变量r。

```
A a1;//定义基类A对象a1
B b1;//定义公用派生类B对象b1
A& r=b1;
或者
A& r=a1;
r=b1;//用派生类B对象b1对a1的引用变量r赋值
```



5.7.3 如果函数的参数是基类对象或基类对象的引用，相应的实参可以用子类对象

- 如有一函数fun:

```
void fun(A& r)//形参是类A的对象的引用变量
{
    cout<<r.num<<endl;
}
//在调用fun函数时可以用派生类B的对象b1作实参:
fun(b1);
```

```
    输出类B的对象b1的基类数据成员num的值。在fun函数中只能输出派生类中基类成员的值。
```

5.7.4 派生类对象的地址可以赋给指向基类对象的指针变量，即指向基类对象的指针变量也可以指向派生类对象

- 例5.10

5.8 继承与组合

- 在一个类中以另一个类的对象作为数据成员的，称为类的组合。
- 例如，声明Professor(教授)类是Teacher(教师)类的派生类，另有一个类BirthDate(生日)，包含year,month,day等数据成员。可以将教授生日的信息加入到Professor类的声明中。如

```
class Teacher//教师类
{
    public:
        /* ... */
    private:
        int num;
        string name;
        char sex;
};
class BirthDate//生日类
{
    public:
        /* ... */
    private:
        int year;
        int month;
        int day;
};
class Professor:public Teacher//教授类
{
    public:
        /* ... */
    private://BirthDate类的对象作为数据成员
        BirthDate birthday;
};
```

- 类的组合和继承一样，是软件重用的重要方式。组合和继承都是有效地利用已有类的资源。但二者的概念和用法不同。
    - Professor类通过继承，从Teacher类得到了num,name,age,sex等数据成员，通过组合，从BirthDate类得到了year,month,day等数据成员。
    - 继承是纵向的，组合是横向的。
-

```
//如果有
void fun1(Teacher &);
void fun2(BirthDate &);
//在main函数中调用这两个函数:
fun1(prof1);
fun2(prof1.birthday);
fun2(prof1);
```

5.9 继承在软件开发中的重要意义

- 有许多基类是被程序的其他部分或其他程序使用的，这些程序要求保留原有的基类不受破坏。
- 用户往往得不到基类的源代码。
- 在类库中，一个基类可能已被指定与用户所需的多种组件建立了某种关系，因此在类库中的基类是不容许修改的。
- 实际上，许多基类并不是从已有的其他程序中选取来的，而是专门作为基类设计的。
- 在面向对象程序设计中，需要设计类的层次结构，从最初的抽象类出发，每一层派生类的建立都逐步地向着目标的具体实现前进。

第六章 多态性与虚函数

6.1 多态性的概念

- **多态性**是指具有不同功能的函数可以用同一个函数名，这样就可以用一个函数名调用不同内容的函数。
- 在面向对象方法中表述为: 向不同的对象发送同一个消息，不同的对象在接收时会产生不同的行为(即方法)。也就是说，每个对象可以用自己的方式去响应共同的消息。
- 多态性分为两类: 静态多态性和动态多态性。
    - **静态多态性**是在程序编译时系统就能决定调用的是哪个函数，因此静态多态性又称编译时的多态性。静态多态性是通过**函数的重载**实现的(运算符重载实质上也是函数重载)。
    - **动态多态性**是在程序运行过程中才动态地确定操作所针对的对象。它又称运行时的多态性。动态多态性是通过**虚函数**(virtual function)实现的。
- 当一个基类被继承为不同的派生类时，各派生类可以使用与基类成员相同的成员名，如果在运行时用同一个成员名调用类对象的成员，会调用哪个对象的成员?
- 通过继承而产生了相关的不同的派生类，与基类成员同名的成员在不同的派生类中有不同的含义。也可以说，多态性是"**一个接口，多种方法**"。

6.2 一个典型的例子

例6.1 先建立一个Point(点)类，包含数据成员x,y(坐标点)。以它为基类，派生出一个Circle(圆)类，增加数据成员r(半径)，再以Circle类为直接基类，派生出一个Cylinder(圆柱体)类，再增加数据成员h(高)。要求编写程序，重载运算符"<<"和">>"，使之能用于输出以上类对象。

对于一个比较大的程序，应当分成若干步骤进行：**先声明基类，再声明派生类，逐级进行，分步调试。**

6.3 虚函数

6.3.1 虚函数的作用

- 在类的继承层次结构中，在不同的层次中可以出现名字相同、参数个数和类型都相同而功能不同的函数。编译系统按照同名覆盖的原则决定调用的对象。
- 在例6.1程序中用`cy1.area( )`调用的是派生类Cylinder中的成员函数area。如果想调用cy1中的直接基类Circle的area函数，应当表示为: `cy1.Circle::area( )`。用这种方法来区分两个同名的函数。
- 能否用同一个调用形式，**既能调用派生类又能调用基类的同名函数。**
- 在程序中不是通过不同的对象名去调用不同派生层次中的同名函数，而是通过**指针**调用它们。例如，用同一个语句`pt->display( );`可以调用不同派生层次中的display函数，只需在调用前给指针变量pt赋以不同的值(使之指向不同的类对象)即可。
- 例6.2.1

- C++中的虚函数就是用来解决这个问题的。虚函数的作用是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6.2.2
- 在Student类中声明display函数时，在最左面加一个关键字`virtual`，即`virtual void display( );`
- 由虚函数实现的动态多态性就是: 同一类族中不同类的对象，对同一函数调用作出不同的响应。虚函数的使用方法是:
    - (1) 在**基类**用virtual声明成员函数为虚函数。这样就可以在派生类中重新定义此函数，为它赋予新的功能，并能方便地被调用。在类外定义虚函数时，不必再加virtual。
    - (2) 在派生类中重新定义此函数，要求函数名、函数类型、函数参数个数和类型全部与基类的虚函数相同，并根据派生类的需要重新定义函数体。
      C++规定，**当一个成员函数被声明为虚函数后，其派生类中的同名函数都自动成为虚函数。**因此在派生类重新声明该虚函数时，可以加virtual，也可以不加，但习惯上一般在每一层声明该函数时都加virtual，使程序更加清晰。
    - (3) 定义一个指向基类对象的指针变量，并使它指向同一类族中需要调用该函数的对象。
    - (4) 通过该指针变量调用此虚函数，此时调用的就是指针变量指向的对象的同名函数。
- 函数重载处理的是同一层次上的同名函数问题，而虚函数处理的是不同派生层次上的同名函数问题，前者是横向重载，后者可以理解为纵向重载。
- 但与重载不同的是: 同一类族的虚函数的首部是相同的，而函数重载时函数的首部是不同的(参数个数或类型不同)。

6.3.2 静态关联和动态关联

- 编译系统要根据已有的信息，对同名函数的调用作出判断。对于调用同一类族中的虚函数，应当在调用时用一定的方式告诉编译系统，你要调用的是哪个类对象中的函数。这样编译系统在对程序进行编译时，即能确定调用的是哪个类对象中的函数。
- 确定调用的具体对象的过程称为关联(binding)。在这里是指把一个函数名与一个类对象捆绑在一起，建立关联。一般地说，关联指把一个标识符和一个存储地址联系起来。
- 前面所提到的函数重载和通过对象名调用的虚函数，在编译时即可确定其调用的虚函数属于哪一个类，其过程称为静态关联(static binding)，由于是在运行前进行关联的，故又称为早期关联(early binding)。函数重载属静态关联。
- 在调用虚函数时并没有指定对象名，是**通过基类指针与虚函数的结合**来实现多态性的。先定义了一个指向基类的指针变量，并使它指向相应的类对象，然后通过这个基类指针去调用虚函数(例如`pt->display( )`)。
- 在运行阶段，基类指针变量先指向了某一个类对象，然后通过此指针变量调用该对象中的函数。此时调用哪一个对象的函数无疑是确定的。例如，先使pt指向grad1，再执行“pt->display( )”，当然是调用grad1中的display函数。由于是在运行阶段把虚函数和类对象“绑定”在一起的，因此，此过程称为动态关联(dynamic binding)。这种多态性是动态的多态性，即运行阶段的多态性。
- 在运行阶段，指针可以先后指向不同的类对象，从而调用同一类族中不同类的虚函数。由于动态关联是在编译以后的运行阶段进行的，因此也称为滞后关联(late binding)。

6.3.3 在什么情况下应当声明虚函数

- 只能用virtual声明**类的成员函数**，使它成为虚函数，而不能将类外的普通函数声明为虚函数。
- 因为虚函数的作用是允许在派生类中对基类的虚函数重新定义。显然，它只能用于类的继承层次结构中。

- 一个成员函数被声明为虚函数后，在同一类族中的类就不能再定义一个非virtual的但与该虚函数具有相同的参数(包括个数和类型)和函数返回值类型的同名函数。
- 把一个成员函数声明为虚函数主要考虑以下几点:
    - (1) 首先看成员函数所在的类是否会作为基类。然后看成员函数在类的继承后有无可能被更改功能，如果希望更改其功能的，一般应该将它声明为虚函数。
    - (2) 如果成员函数在类被继承后功能不需修改，或派生类用不到该函数，则不要把它声明为虚函数。不要仅仅考虑到要作为基类而把类中的所有成员函数都声明为虚函数。
    - (3) 应考虑对成员函数的调用是通过对象名还是通过基类指针或引用去访问，如果是通过基类指针或引用去访问的，则应当声明为虚函数。
    - (4) 有时，在定义虚函数时，并不定义其函数体，即函数体是空的。它的作用只是定义了一个虚函数名，具体功能留给派生类去添加。

6.3.4 虚析构函数

- 析构函数的作用是在对象撤销之前做必要的“清理现场”的工作。当派生类的对象从内存中撤销时一般先调用派生类的析构函数，然后再调用基类的析构函数。但是，如果用new运算符建立了临时对象，若基类中有析构函数，并且定义了一个指向该基类的指针变量。在程序用带指针参数的delete运算符撤销对象时，会发生一个情况: 系统会只执行基类的析构函数，而不执行派生类的析构函数。
- 例6.3
- 如果将基类的析构函数声明为虚函数时，由该基类所派生的所有派生类的析构函数也都自动成为虚函数，即使派生类的析构函数与基类的析构函数名字不相同。
- 把基类的析构函数声明为虚函数，将使所有派生类的析构函数自动成为虚函数。这样，如果程序中显式地用了delete运算符准备删除一个对象，而delete运算符的操作对象用了指向派生类对象的基类指针，则系统会调用相应类的析构函数。
- **构造函数不能声明为虚函数**。这是因为在执行构造函数时类对象还未完成建立过程，当然谈不上函数与类对象的绑定。

6.4 纯虚函数与抽象类

6.4.1 纯虚函数

- 在基类Point中加一个area函数，并声明为虚函数:

```
virtual float area( ) const {return 0;}
```

其返回值为0，表示“点”是没有面积的。其实，在基类中并不使用这个函数，其返回值也是没有意义的。为简化，可以不写出这种无意义的函数体，只给出函数的原型，并在后面加上“=0”，如

```
virtual float area( ) const =0;//纯虚函数
```

这就将area声明为一个纯虚函数(pure virtual function)。

- 纯虚函数是在声明虚函数时被“初始化”为0的函数。声明纯虚函数的一般形式是
  virtual 函数类型 函数名 (参数表列) =0;
- 注意:
    - ①纯虚函数没有函数体;
    - ②最后面的“=0”并不表示函数返回值为0，它只起形式上的作用，告诉编译系统“这是纯虚函数”;
    - ③这是一个声明语句，最后应有分号。

- 纯虚函数只有函数的名字而不具备函数的功能，**不能被调用**。在派生类中对此函数提供定义后，它才能具备函数的功能，可被调用。
- 纯虚函数的作用是在基类中为其派生类保留一个函数的名字，以便派生类根据需要对它进行定义。如果在基类中没有保留函数名字，则无法实现多态性。
- 如果在一个类中声明了纯虚函数，而在其派生类中没有对该函数定义，则该虚函数在派生类中仍然为纯虚函数。

6.4.2 抽象类

- 不用来定义对象而只作为一种基本类型用作继承的类，称为抽象类(abstract class)，由于它常用作基类，通常称为抽象基类(abstract base class)。
- 凡是包含纯虚函数的类都是抽象类。因为纯虚函数是不能被调用的，包含纯虚函数的类是无法建立对象的。抽象类的作用是作为一个类族的共同基类，或者说，为一个类族提供一个公共接口。
- 一个类层次结构中当然也可不包含任何抽象类，每一层次的类都是实际可用的，可以用来建立对象的。但是，许多好的面向对象的系统，其层次结构的顶部是一个抽象类，甚至顶部有好几层都是抽象类。
- 如果在抽象类所派生出的新类中对基类的所有纯虚函数进行了定义，那么这些函数就被赋予了功能，可以被调用。这个派生类就不是抽象类，而是可以用来定义对象的具体类(concrete class)。如果在派生类中没有对所有纯虚函数进行定义，则此派生类仍然是抽象类，不能用来定义对象。
- 虽然抽象类不能定义对象(或者说抽象类不能实例化)，但是可以定义指向抽象类数据的指针变量。当派生类成为具体类之后，就可以用这种指针指向派生类对象，然后通过该指针调用虚函数，实现多态性的操作。

6.4.3 应用实例

- 例6.4
- 一个基类如果包含一个或一个以上纯虚函数，就是抽象基类。抽象基类不**能也不必要**定义对象。
- 抽象基类与普通基类不同，它一般并不是现实存在的对象的抽象(例如圆形(Circle)就是千千万万个实际的圆的抽象)，它可以没有任何物理上的或其他实际意义方面的含义。
- 在类的层次结构中，顶层或最上面的几层可以是抽象基类。抽象基类体现了本类族中各类的共性，把各类中共有的成员函数集中在抽象基类中声明。
- 抽象基类是本类族的公共接口。或者说，从同一基类派生出的多个类有同一接口。
- 区别静态关联和动态关联。
- 如果在基类声明了虚函数，则在派生类中凡是与该函数有相同的函数名、函数类型、参数个数和类型的函数，均为虚函数(不论在派生类中是否用virtual声明)。
- 使用虚函数提高了程序的可扩充性。把类的声明与类的使用分离。这对于设计类库的软件开发商来说尤为重要。开发商设计了各种各样的类，但不向用户提供源代码，用户可以不知道类是怎样声明的，但是可以使用这些类来派生出自己的类。