C++语言程序设计：MOOC版

清华大学出版社（ISBN 978-7-302-42104-7）

第8章 面向对象程序设计之二

中国農業大學 阚道宏

第8章 面向对象程序设计之二

- 面向对象程序设计能提高程序开发效率
  - 分类管理程序代码，即类与对象编程
  - 重用类代码
    - 使用类定义对象
    - 使用已有类来定义新类
      - 组合
      - 继承

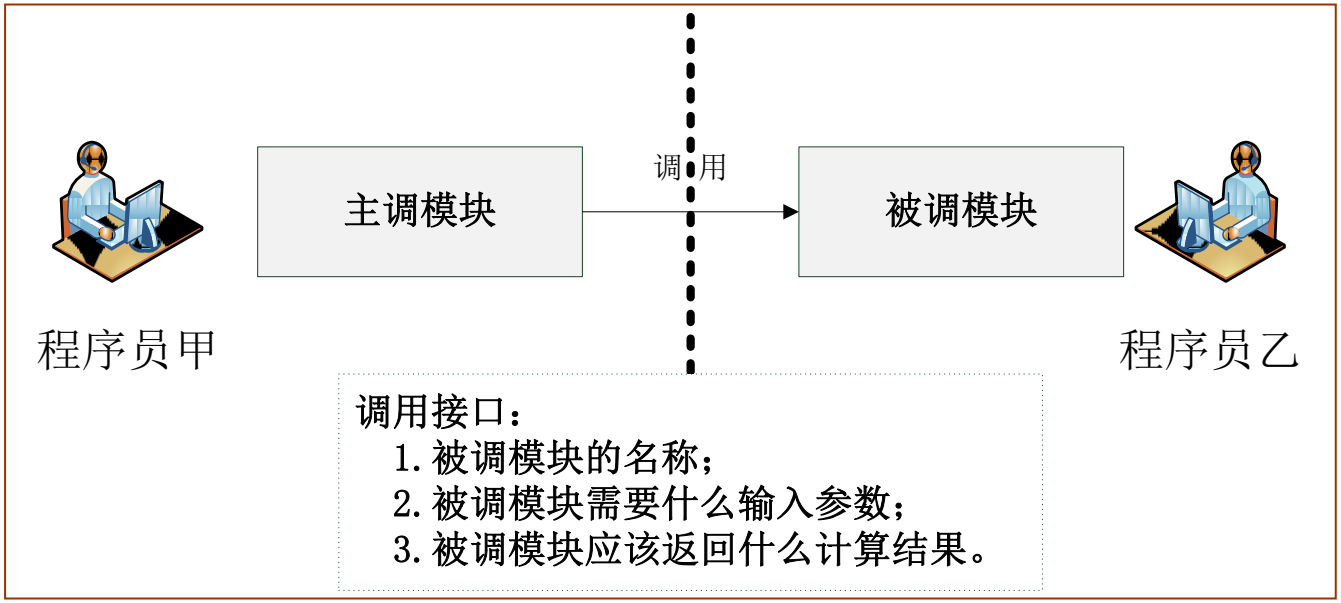
- 重用代码的程序员角色
  - 提供代码的程序员
  - 使用代码的程序员

第8章 面向对象程序设计之二

- 本章内容
  - 8.1 代码重用
  - 8.2 类的组合
  - 8.3 类的继承与派生
  - 8.4 多态性
  - 8.5 关于多继承的讨论

8.1 代码重用

- 程序=数据+算法
- 结构化程序设计中的代码重用



8.1 代码重用

- 结构化程序设计重用函数代码
  - 提供函数代码的程序员乙

```
double CArea(double r) { return (3.14*r*r); }
double CLen(double r) { return (3.14*2*r); }
```

  - 重用函数代码的程序员甲

```
int main( )
{
   double r;
   cin >> r;
   cout << CArea( r ) << endl;
   cout << CLen( r ) << endl;
   ......
}
```

  - 结构化程序设计重用的是算法代码，没有重用数据代码

8.1 代码重用

- 面向对象程序设计中的代码重用

定义类

接口

用类定义对象

程序员甲

为完成特定的程序功能，定义类时：
1. 需定义哪些变量，即数据成员？
2. 需定义什么函数，即函数成员？
3. 为让乙使用类，需将哪些成员开放给乙？
4. 为了让乙初始化对象，需定义什么构造函数？
5. 销毁对象如需清理内存，则定义析构函数来完成。

类接口：
1. 公有数据成员
2. 公有函数成员

程序员乙

使用类定义对象时，需了解：
1. 定义对象时，类有什么初始化方法？
2. 对象有什么公有数据成员？
3. 对象有什么公有函数成员？
4. 访问对象的公有数据成员，读写数据。
5. 调用对象的公有函数成员，处理数据。

8.1 代码重用

- 面向对象程序设计重用类代码
  - 用类定义对象

```
class Circle  // 圆形类：程序员乙
{
public:
   double r;  // 半径：数据成员
   double CArea( );  // 求面积：函数成员
   double CLen( );  // 求周长：函数成员
};

int main( )  // 程序员甲
{
   Circle  obj1;  // 定义圆形类的对象obj1
   cin >> obj1.r;  // 从键盘输入对象obj1的半径
   cout << obj1.CArea( ) << endl;  // 对象obj1的面积
   cout << obj1.CLen( ) << endl;  // 对象obj1的周长
   ......
}
```

8.1 代码重用

- 面向对象程序设计中的代码重用
  - 重用类代码
    - 用类定义对象
    - 通过组合定义新的类（称为组合类）
    - 通过继承定义新的类（称为派生类）
  - 类的5大要素
    - 数据成员
    - 函数成员
    - 访问权限
    - 构造函数
    - 析构函数

例8-1 圆形类Circle的定义代码

类头文件： Circle.h

```
class Circle  // 圆形类：声明部分，即声明成员
{
private:
    double r; // 半径：私有数据成员

public:
    void Input( ); // 输入半径：公有函数成员
    double CRadius( ); // 读取半径：公有函数成员
    double CArea( ); // 求面积：公有函数成员
    double CLen( ); // 求周长：公有函数成员

    Circle( ); // 无参构造函数：公有权限
    Circle( double x ); // 有参构造函数：公有权限
    Circle( Circle &x ); // 拷贝构造函数：公有权限
};
```

类源程序文件： Circle.cpp

```
#include <iostream>
using namespace std;
#include “Circle.h”  // 声明类Circle

// 圆形类：实现部分，具体的函数代码
void Circle :: Input( ) // 输入半径
{
    cin >> r;
    while ( r < 0 ) // 数据合法性检查
      cin >> r; // 如r<0，则重新输入
}
double Circle :: CRadius( ) // 读取半径
{   return  r;   }
double Circle :: CArea( ) // 求面积
{   return (3.14*r*r);   }
double Circle :: CLen( ) // 求周长
{   return (3.14*2*r);   }
Circle :: Circle( ) // 无参构造函数
{   r = 0;   }
Circle :: Circle( double x ) // 有参构造函数
{
    if ( x < 0 ) r = 0; // 如r<0，则置0
    else  r = x;
}
Circle :: Circle( Circle &x ) // 拷贝构造函数
{   r = x.r;   }
```

8.1 代码重用

- 使用类Circle的典型流程

```
Circle  obj;
obj.Input( );
cout << obj.CRadius( ) << endl;
cout << obj.CArea( ) << endl;
cout << obj.CLen( ) << endl;
```

- 对象初始化

```
Circle  obj1;
Circle  obj2( 5 );
Circle  obj3( obj2 );
```

- 析构函数：~Circle( ) { }

- 问题：如何基于Circle类来定义更复杂的类？

```
class Circle  // 圆形类：声明部分，即声明成员
{
private:
    double r; // 半径：私有数据成员

public:
    void Input( ); // 输入半径：公有函数成员
    double CRadius( ); // 读取半径：公有函数成员
    double CArea( ); // 求面积：公有函数成员
    double CLen( ); // 求周长：公有函数成员

    Circle( ); // 无参构造函数：公有权限
    Circle( double x ); // 有参构造函数：公有权限
    Circle( Circle &x ); // 拷贝构造函数：公有权限
};
```

8.2 类的组合

- 组合的编程原理是：程序员在定义新类的时候，使用已有的类来定义数据成员。这些数据成员是类类型的对象，被称为类的对象成员。C++语言将数据成员中包含对象成员的类称为组合类
- 按照数据类型的不同，组合类中数据成员可分为2种，即类类型的对象成员和基本数据类型的非对象成员

- 使用组合类定义对象，即组合类对象，其成员中也将包含对象成员和非对象成员

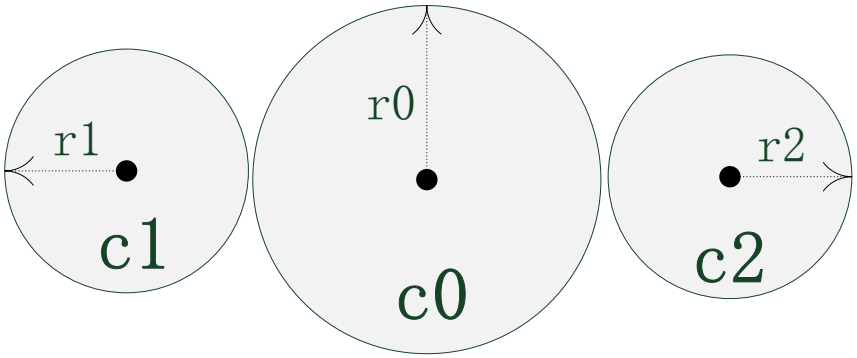
- 访问组合类对象中的非对象成员

  组合类对象名 . 非对象成员名
- 组合类对象中的对象成员还包含自己的下级成员，也就是说组合类对象包含多级成员。可以访问组合类对象中对象成员的下级成员，这是一种多级访问。多级访问的语法形式是：

  组合类对象名 . 对象成员名 . 对象成员的下级成员名
- 多级访问将受到多级权限的控制

8.2 类的组合

- 定义TriCircle类



  - 可以从零开始编写
  - 也可以基于已有的Circle类来编写组合类。类TriCircle可以认为是由3个Circle类对象组合而成的

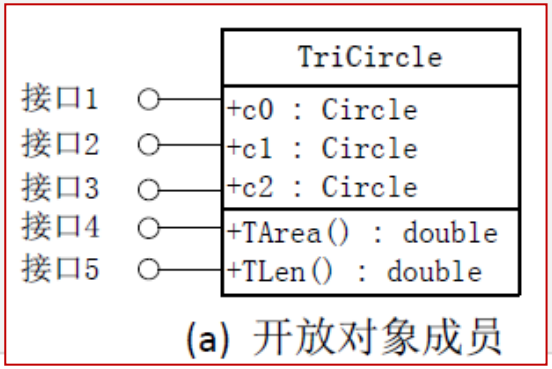
8.2 类的组合

例8-2 组合类TriCircle的定义代码

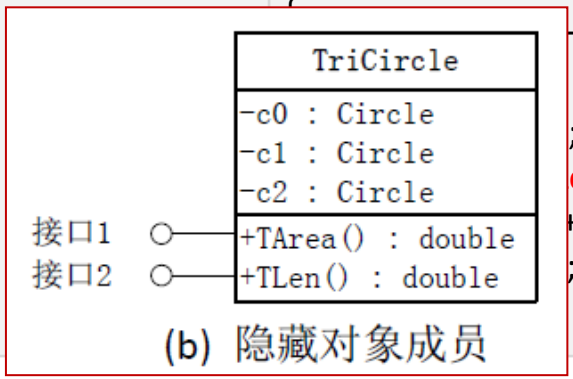
类头文件： TriCircle.h

```
1  #include “Circle.h” // 声明类Circle
2
3  class TriCircle  // 声明部分，即声明成员
4  {
5  public:
6     Circle c0, c1, c2; // 3个公有Circle类对象成员
7     double TArea( ); // 求总面积：公有函数成员
8     double TLen( ); // 求总周长：公有函数成员
9  };
10
11
12
13
14
15
16
17
```

private



(a) 开放对象成员



(b) 隐藏对象成员

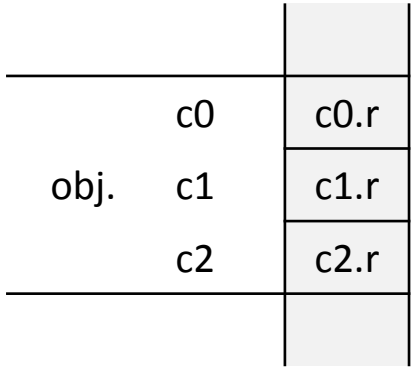
类源程序文件： TriCircle.cpp

```
#include “TriCircle.h” // 声明类TriCircle

// TriCircle类：实现部分，具体的函数代码
double TriCircle :: TArea( ) // 求总面积
{
   double totalArea;
   totalArea = c0.CArea( ) + c1.CArea( ) + c2.CArea( );
   return totalArea;
}

double TriCircle :: TLen( ) // 求总周长
{
   double totalLen;
   totalLen = c0.CLen( ) + c1.CLen( ) + c2.CLen( );
   return totalLen;
}
```

8.2 类的组合

- 组合类对象的定义与访问

  TriCircle obj;



obj.c0、obj.c1、obj.c2、obj.TArea( )、obj.TLen( )

– 多级访问的语法形式

组合类对象名 . 对象成员名 . 对象成员的下级成员名

8.2 类的组合

```
#include <iostream>
using namespace std;
#include “TriCircle.h”  // 类TriCircle的声明头文件

int main( )
{
   TriCircle  obj;  // 定义1个组合类TriCircle的对象obj

   // 调用组合类对象obj中对象成员c0的下级函数成员Input，输入c0的半径
   obj.c0.Input( );
   // 再调用c0的下级函数成员CArea和CLen，计算并显示c0的面积和周长
   cout << obj.c0.CArea( ) << “, ” << obj.c0.CLen( ) << endl;

   // 类似的，可计算出obj中对象成员c1、c2的面积和周长
   obj.c1.Input( );    cout << obj.c1.CArea( ) << “, ” << obj.c1.CLen( ) << endl;
   obj.c2.Input( );    cout << obj.c2.CArea( ) << “, ” << obj.c2.CLen( ) << endl;

   // 调用组合类对象obj中的非对象成员TArea和TLen，计算并显示总面积和总周长
   cout << obj.TArea( ) << “, ” << obj.TLen( ) << endl;
   return 0;
}
```

8.2 类的组合

TriCircle obj; // 定义1个组合类TriCircle的对象obj

TriCircle *p = &obj; // 定义1个组合类TriCircle的对象指针p，让其指向obj

// 通过对象指针p间接访问组合类对象obj中对象成员的下级函数成员
p->c0.Input( ); cout << p->c0.CArea( ) << “, ” << p->c0.CLen( ) << endl;
p->c1.Input( ); cout << p->c1.CArea( ) << “, ” << p->c1.CLen( ) << endl;
p->c2.Input( ); cout << p->c2.CArea( ) << “, ” << p->c2.CLen( ) << endl;

// 通过对象指针p间接访问组合类对象obj中的非对象成员TArea和TLen
cout << p->TArea( ) << “, ” << p->TLen( ) << endl;

8.2 类的组合

- 如何设计组合类中对象成员的访问权限
  - 组合类编程中有2种程序员角色，分别是定义组合类的程序员甲和使用组合类的程序员乙
  - 甲在使用对象成员组装组合类时，可根据功能要求决定将哪些对象成员开放给乙，哪些隐藏起来。开放就是将对象成员设定为公有权限，隐藏就是设定为保护权限或私有权限，这就是对象成员的二次封装
  - 乙使用组合类定义对象。在所定义出的组合类对象中，开放的对象成员可以访问，隐藏的则不可以访问。组合类对象中的对象成员还包含下级成员，这些下级成员也都有各自的访问权限，公有的才可以访问，否则不可以访问
  - 多级访问将受到多级权限的控制。访问组合类对象中对象成员的下级成员，只有对象成员和下级成员都是公有权限才可以访问，否则就不能访问

8.2 类的组合

- 组合类对象的构造与析构
  - 对象的构造与析构

    计算机执行定义对象语句时将创建对象，为其分配内存空间，并自动调用对象所属类的构造函数来初始化对象，这个过程就是对象的构造。当对象生存期结束时，计算机将销毁对象。销毁时自动调用对象所属类的析构函数来清理内存，然后释放其所占用的内存空间，这个过程就是对象的析构

  - 组合类对象的构造与析构
    - 对象成员（类类型的数据成员）
    - 非对象成员（基本数据类型的数据成员）

8.2 类的组合

- 组合类对象的构造与析构
  - 组合类的构造函数

组合类构造函数名(形参列表) : 对象成员名1(形参1), 对象成员名2(形参2), ...
{
  ..... // 在函数体中初始化其它非对象成员
}

  - 组合类对象中各数据成员的初始化顺序是：先调用对象成员所属类的构造函数，初始化对象成员；再执行组合类构造函数的函数体，初始化其它非对象成员

  - 如果组合类中有多个对象成员，那么这些对象成员的初始化顺序由其在组合类中的声明顺序决定，先声明者先初始化

8.2 类的组合

- 组合类TriCircle可添加如下3个重载构造函数
  - **有参构造函数**

```
TriCircle :: TriCircle( double p0, double p1, double p2 ) : c0(p0), c1(p1), c2(p2)
{ }
```

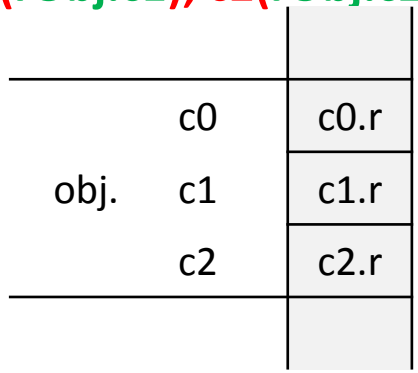
  - **无参构造函数。**

```
TriCircle :: TriCircle( ) { }
```

  - **拷贝构造函数**

```
TriCircle :: TriCircle( TriCircle &rObj ) : c0(rObj.c0), c1(rObj.c1), c2(rObj.c2) { }
```

```
TriCircle  obj( 5, 2, 3 );
TriCircle  obj1;
TriCircle  obj2( obj );
```



8.2 类的组合

- 组合类的析构函数
  - 当对象生存期结束时，计算机销毁对象，释放其内存空间，这个过程就是对象的析构。销毁对象时计算机会自动调用其所属类的析构函数
  - 组合类对象中数据成员的析构顺序是：先执行组合类析构函数的函数体，清理非对象成员；再调用对象成员所属类的析构函数，清理对象成员
  - 简单地说，对象的析构顺序与构造顺序相反，即先析构非对象成员，再析构对象成员

8.2 类的组合

- 类的聚合

例8-3 聚合类pTriCircle的定义代码

类头文件： pTriCircle.h

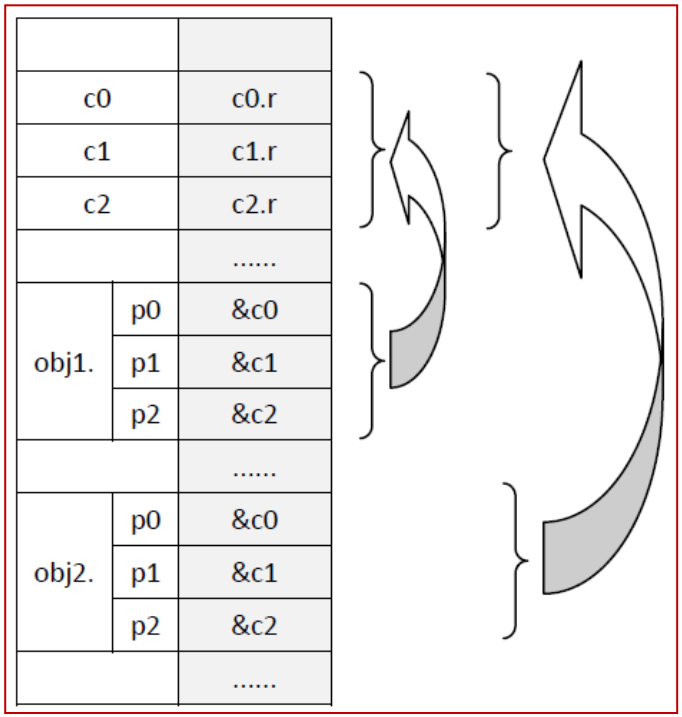
```
1  #include “Circle.h”  // 声明类Circle
2
3  class pTriCircle  // 声明部分，即声明成员
4  {
5  public:
6      Circle *p0, *p1, *p2;  // 公有Circle类的对象指针
7      double TArea( );  // 求总面积：公有函数成员
8      double TLen( );  // 求总周长：公有函数成员
9  };
10
11
12
13
14
15
16
17
```

类源程序文件： pTriCircle.cpp

```
#include “pTriCircle.h”  // 声明类pTriCircle

// pTriCircle类：实现部分，具体的函数代码
double pTriCircle :: TArea( )  // 求总面积
{
    double totalArea;
    totalArea = p0->CArea( ) +
          p1->CArea( ) + p2->CArea( );
    return  totalArea;
}
double pTriCircle :: TLen( )  // 求总周长
{
    double totalLen;
    totalLen = p0->CLen ( ) +
          p1->CLen( ) + p2->CLen( );
    return  totalLen;
}
```

8.2 类的组合

```
#include <iostream>
using namespace std;
#include “pTriCircle.h”  // 类pTriCircle的声明头文件

int main( )
{
   Circle c0, c1, c2;  // 先定义3个类Circle的对象c0、c1、c2
   c0.Input( );   c1.Input( );   c2.Input( );  // 输入3个圆的半径

   pTriCircle  obj1;  // 定义1个聚合类pTriCircle的对象obj1
   // 将obj1中3个对象指针分别指向已经创建的Circle类对象c0、c1、c2
   obj1.p0 = &c0;   obj1.p1 = &c1;   obj1.p2 = &c2;
   // 调用obj1中的函数成员TArea和TLen，计算并显示总面积和总周长
   cout << obj1.TArea( ) << “, ” << obj1.TLen( ) << endl;

   pTriCircle  obj2;  // 定义1个聚合类pTriCircle的对象obj2
   // 将obj2中3个对象指针也分别指向Circle类对象c0、c1、c2
   obj2.p0 = &c0;   obj2.p1 = &c1;   obj2.p2 = &c2;
   // 调用obj2中的函数成员TArea和TLen，计算并显示总面积和总周长
   cout << obj2.TArea( ) << “, ” << obj2.TLen( ) << endl;
   return 0;
}
```



8.2 类的组合

- 类的组合与聚合
  - 数据成员中包含对象成员的类称为组合类
  - 数据成员中包含对象指针的类称为聚合类，聚合类是一种特殊形式的组合类

- 聚合类与组合类的区别
  - 聚合类的对象成员是独立创建的，聚合类对象只包含指向对象成员的指针
  - 聚合类对象可以共用对象成员

8.2 类的组合

- 组合类总结
  - **代码重用**。组合类是一种有效的代码重用形式。程序员在设计新类的时候应先去了解有哪些可以重用的类。这些类可以是自己以前编写的，可以是集成开发环境IDE提供的，也可以是从市场上购买的。根据功能选择自己需要的类，然后用组合的方法定义新类
  - **自底向上**。类可以多级组合。用零件类定义组合类，组合类可继续作为零件类去定义更大的组合类，这就是类的多级组合。多级组合是一种“自底向上”的程序设计方法。类越往上组合，其功能就越有针对性，应用面也就越窄。多级组合过程中，每一级组合类都会根据自己的功能需要设定对象成员的访问权限。有多少级组合，就有多少层封装

8.3 类的继承与派生

- 设计新类时可继承已有的类，这个已有的类被称为基类或父类

- 基类是为解决以前的老问题而设计的，在面对新问题时其功能可能会显得不够完善。程序员需要在继承的基础上对基类进行派生，例如添加新功能，或者对从基类继承来的功能进行某些修改。派生的目的是为了解决新问题

- 通过继承与派生所得到的新类被称为派生类或子类

8.3 类的继承与派生

- 继承与派生的编程原理是：程序员在定义新类的时候，首先继承基类的数据成员和函数成员；在此基础上进行派生，为派生类添加新成员，或对从基类继承来的成员进行重新定义或修改其访问权限。在继承与派生的过程中，继承实现了基类代码的重用，派生则实现了基类代码的进化

- 派生类中的成员可分为2种：一是从基类继承来的成员，称为派生类中的基类成员，二是定义时新增的成员，称为派生类中的新增成员

8.3 类的继承与派生

C++语法：**定义派生类**

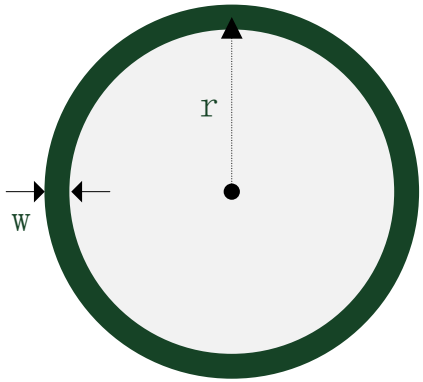
```
class 派生类名 : 继承方式1 基类1, 继承方式2 基类2, ..... // 派生类声明部分
{
public :
    新增公有成员
protected :
   新增保护成员
private :
   新增私有成员
};
各新增函数成员的完整定义 // 派生类实现部分
```

语法说明：

- 定义派生类时，在派生类名的后面添加**继承列表**，在声明部分的大括号里声明**新增成员**，在实现部分编写各新增函数成员的完整定义代码；
- **继承列表**指定派生类从哪些基类继承。派生类可以只从一个基类继承（**单继承**），也可以从多个基类继承（**多继承**）。每个基类以“**继承方式 基类名**”的形式声明，多个基类之间用“,”隔开；
- 派生类将继承基类中**除构造函数、析构函数之外**的所有数据成员和函数成员。基类的构造函数和析构函数不能被继承，派生类需重新编写自己的构造函数、析构函数。
- 继承后，派生类会对其基类成员按照**继承方式**进行**再次封装**。继承方式有3种：public（公有继承）、protected（保护继承）和private（私有继承）；
- public（**公有继承**）：派生类对其基类成员不做任何封装，它们在派生类中的访问权限与原来在基类中的权限相同；
- private（**私有继承**）：派生类对其基类成员做全封装，它们在派生类中的访问权限统统被改为private（私有权限），不管它们原来在基类中的权限是什么。使用私有继承，实际上是派生类要将其基类成员全部隐藏起来；
- protected（**保护继承**）：派生类对其基类成员做半封装。基类中的public成员被继承到派生类后，其访问权限被降格成protected（保护权限）。基类中的protected、private成员被继承到派生类后，其访问权限保持不变；
- 在**类声明部分**的大括号中声明新增的数据成员、函数成员，并指定各新增成员的访问权限。在**类实现部分**编写各新增函数成员的完整定义代码。

8.3 类的继承与派生

- 定义BorderCircle类



  - 可以从零开始编写
  - 也可以基于已有的Circle类来编写派生类
    - BorderCircle类可以继承Circle类中的半径r、求面积和周长的函数CArea、Clen
    - 然后再新增边框宽度w、求内圆面积和边框面积的函数InnerArea、BorderArea
    - Circle类中的输入函数Input只能输入半径，为此BorderCircle类重新定义了1个Input函数，这相当于是修改了原Input函数

8.3 类的继承与派生

同名覆盖：派生类中定义与基类成员重名的新增成员，新增成员将覆盖基类成员。通过成员名访问时，所访问到的将是新增成员，这就是新增成员对基类成员的同名覆盖。同名覆盖后，被覆盖的基类成员仍然存在，只是被隐藏了。可以访问被覆盖的基类成员，其访问形式是：基类名 :: 基类成员名

的定义代码

类源程序文件：BorderCircle.cpp

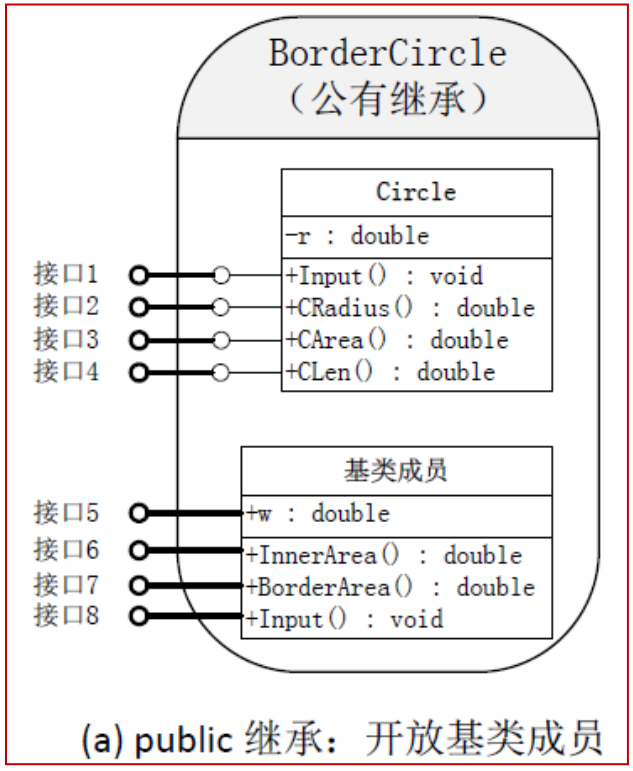
```
#include <iostream>
using namespace std;
#include "BorderCircle.h"

// BorderCircle类：实现部分
```

```
6  {
7  public:
8      double w; // 新增数据成员：边框宽度
9      // 以下为新增的函数成员
10     double InnerArea( ); // 求内圆面积
11     double BorderArea( ); // 求边框面积
12     void Input( ); // 输入半径和边框宽度
13 };
14
15
16
17
18
```

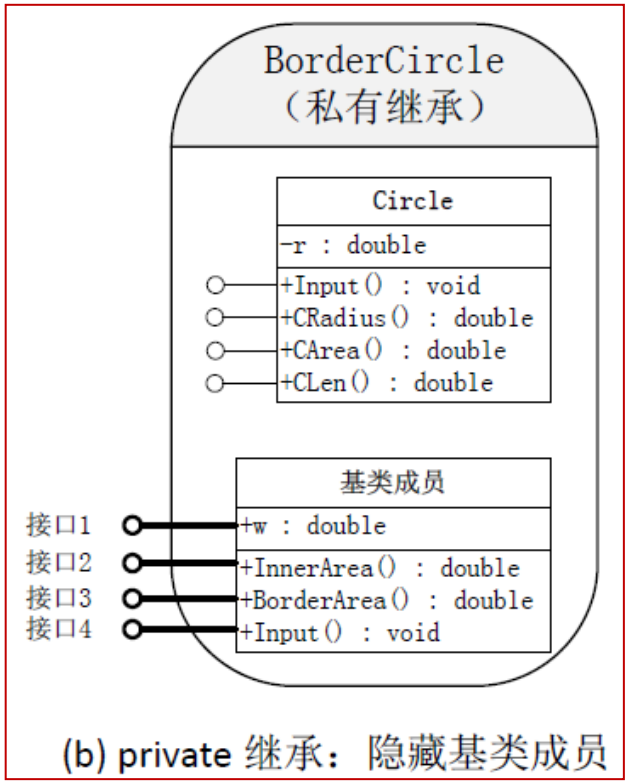
```
class Circle  // 圆形类：声明部分，即声明成员
{
private:
    double r; // 半径：私有数据成员

public:
    void Input( ); // 输入半径：公有函数成员
    double CRadius( ); // 读取半径：公有函数成员
    double CArea( ); // 求面积：公有函数成员
    double CLen( ); // 求周长：公有函数成员

    Circle( ); // 无参构造函数：公有权限
    Circle( double x ); // 有参构造函数：公有权限
    Circle( Circle &x ); // 拷贝函数：公有权限
};
```

8.3 类的继承与派生

- 派生类对基类成员的二次封装



(a) public 继承：开放基类成员

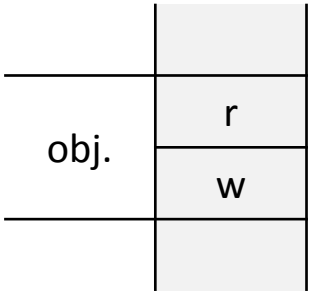


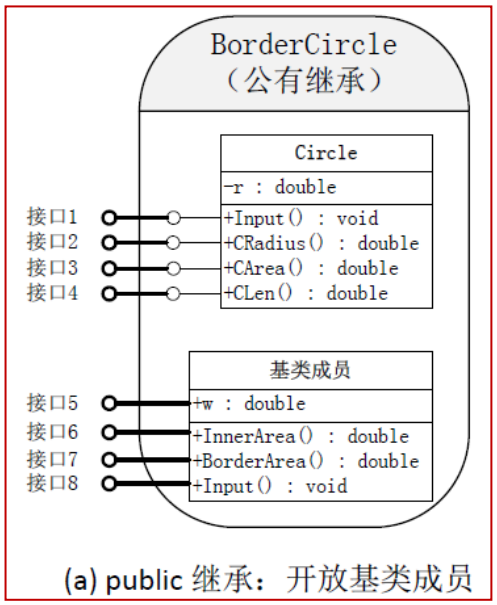
(b) private 继承：隐藏基类成员

8.3 类的继承与派生

- 派生类对象的定义与访问

  BorderCircle obj;





– 访问公有的基类成员

obj.Circle::Input( )、obj.CRadius( )、obj.CArea( )、obj.CLen( )

– 访问公有的新增成员

obj.w、obj.InnerArea( )、obj.BorderArea( )、obj.Input( )

8.3 类的继承与派生

```
#include <iostream>
using namespace std;
#include “BorderCircle.h”  // 类BorderCircle的声明头文件

int main( )
{
   BorderCircle  obj;  // 定义1个派生类BorderCircle的对象obj

   // 调用派生类对象obj中的新增函数成员Input，输入半径和边框宽度
   obj.Input( );  // 调用的是新增成员Input，重名的基类成员Input被覆盖了

   // 调用obj中的基类函数成员CArea和CLen，计算并显示圆的面积和周长
   cout << obj.CArea( ) << “, ” << obj.CLen( ) << endl;

   // 调用obj中的新增成员InnerArea和BorderArea，计算并显示内圆和边框的面积
   cout << obj.InnerArea( ) << “, ” << obj.BorderArea( ) << endl;
   return 0;
}
```

8.3 类的继承与派生

- 如何设计派生类的继承方式
  - 派生类编程中有2种程序员角色，分别是定义派生类的程序员甲和使用派生类的程序员乙。甲在设计派生类时需根据派生类的功能要求来决定：是将基类成员继续开放给乙，还是将它们隐藏起来。公有继承就是继续开放基类成员，私有继承或保护继承就是隐藏基类成员，这就是派生类对基类成员的二次封装
  - 乙使用派生类定义对象，是否可以访问其中的基类成员，这取决于该基类成员的访问权限。派生类对象中基类成员的访问权限由如下2个方面的因素决定：
    - 因素1：派生类的继承方式
    - 因素2：基类成员原来在基类中的访问权限
  - 派生类也可以任意多级。用基类定义派生类，派生类可以继续作为基类去定义更下级的派生类，这就是多级派生。多级派生过程中，每一级派生类都会根据自己的功能需要设定继承方式，这相当于对所继承的基类成员进行再次封装

8.3 类的继承与派生

- 保护权限与保护继承

例8-5 关于保护权限的C++演示程序

程序员甲：定义**类A**（类头文件A.h）

```
1  class A  // 定义类A
2  {
3  public:
4      A( int p1 = 0, int p2 = 0, int p3 = 0 )  // 构造函数
5      {  x = p1;  y = p2;  z = p3;  }
6      int  x;  // 公有权限
7  private:  int  y;  // 私有权限
8  protected:  int  z;  // 保护权限
9  };
10 // 类A没有函数成员，所以不需要类实现部分
```

类的保护权限是向其派生类定向开放的一种权限

程序员乙：使用类A定义**对象**（1.cpp）

```
1  #include <iostream>
2  using namespace std;
3  #include "A.h"  // 声明类A
4
5  int main( )
6  {
7      A  obj( 10, 20, 30 );
8      cout << obj.x << endl;  // 正确：public
9      cout << obj.y << endl;  // 错误：private
10     cout << obj.z << endl;  // 错误：protected
11     return 0;
12 }
13
14
```

程序员丙：使用类A定义**派生类B**（B.h）

```
#include <iostream>
using namespace std;
#include "A.h"  // 声明基类A

class B : public A
{
public:
    void funB( )  // 新增成员访问基类成员
    {
        cout << x << endl;  // 正确：public
        cout << y << endl;  // 错误：private
        cout << z << endl;  // 正确：protected
    }
};
```

例8-6 关于保护继承的C++演示程序

中的类A定义**派生类B**（类头文件B.h）

基类A

```
class B : protected  A  // 保护继承
{
public:
    void funB( )  // 新增成员访问基类成员
    {
        cout << x << endl;  // 正确：public
        cout << y << endl;  // 错误：private
        cout << z << endl;  // 正确：protected
    }
};
```

```
class B
{
protected:  int  x;  // 保护权限
private:  int  y;  // 私有权限
protected:  int  z;  // 保护权限
};
```

```
class A  // 定义类A
{
public
    A(
    {
    in
private
protected:  int  z;  // 保护权限
};
```

派生类的保护继承是向其下级派生类定向开放的一种半封装

程序员乙：使用类B定义**对象**（1.cpp）

```
#include <iostream>
using namespace std;
#include “B.h”  // 声明类B

int main( )
{
    B  obj( 10, 20, 30 );
    cout << obj.x << endl;  // 错误：protected
    cout << obj.y << endl;  // 错误：private
    cout << obj.z << endl;  // 错误：protected
    return 0;
}
```

程序员丙：使用类B定义**派生类C**（C.h）

```
#include <iostream>
using namespace std;
#include “B.h”  // 声明基类B

class C : protected  B
{
public:
    void funC( )  // 新增成员访问基类成员
    {
        cout << x << endl;  // 正确：protected
        cout << y << endl;  // 错误：private
        cout << z << endl;  // 正确：protected
    }
};
```

```
class B
{
protected: int x; // 保护权限
private: int y; // 私有权限
protected: int z; // 保护权限
};
```

例8-6 关于保护继承的C++演示程序

中的类A定义**派生类B**（类头文件B.h）

基类A

```
5  class B : protected A // 保护继承
6  {
7  public:
8      void funB( ) // 新增成员访问基类成员
9      {
10         cout << x << endl; // 正确：public
12         cout << y << endl; // 错误：private
13         cout << z << endl; // 正确：protected
14     }
15 };
```

派生类的保护继承是向其下级派生类定向开放的一种半封装

程序员乙：使用类B定义**对象**（1.cpp）

```
1  #include <iostream>
2  using namespace std;
3  #include "B.h" // 声明类B
4
5  int main( )
6  {
7      B obj( 10, 20, 30 );
8      cout << obj.x << endl; // 错误：protected
9      cout << obj.y << endl; // 错误：private
10     cout << obj.z << endl; // 错误：protected
11     return 0;
12 }
13
14
```

程序员丙：使用类B定义**派生类C**（C.h）

```
#include <iostream>
using namespace std;
#include "B.h" // 声明基类B

class C : protected B
{
public:
    void funC( ) // 新增成员访问基类成员
    {
        cout << x << endl; // 正确：protected
        cout << y << endl; // 错误：private
        cout << z << endl; // 正确：protected
    }
};
```

8.3 类的继承与派生

- 派生类对象的构造与析构
  - 对象的构造与析构

    计算机执行定义对象语句时将创建对象，为其分配内存空间，并自动调用对象所属类的构造函数来初始化对象，这个过程就是对象的构造。当对象生存期结束时，计算机将销毁对象。销毁时自动调用对象所属类的析构函数来清理内存，然后释放其所占用的内存空间，这个过程就是对象的析构

  - 派生类对象的构造与析构
    - 基类成员
    - 新增成员

8.3 类的继承与派生

- 派生类对象的构造与析构
  - 派生类的构造函数

派生类构造函数名(形参列表) : 基类名1(形参1), 基类名2(形参2), ...
{
  ..... // 在函数体中初始化新增成员
}

  - 派生类对象中各数据成员的初始化顺序是：先调用基类构造函数，初始化基类成员；再执行派生类构造函数的函数体，初始化新增成员

  - 如果派生类继承了多个基类，那么各基类成员的初始化顺序由其在派生类继承列表中的声明顺序决定，声明在前的基类成员先初始化

8.3 类的继承与派生

- 派生类BorderCircle可添加如下3个重载构造函数
  - 有参构造函数

```
BorderCircle :: BorderCircle( double p1, double p2 ) : Circle(p1)
{   w = p2;   }
```

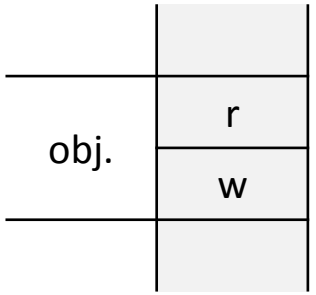
  - 无参构造函数

```
BorderCircle :: BorderCircle( ) {   w = 0;   }
```

  - 拷贝构造函数

```
BorderCircle :: BorderCircle( BorderCircle &rObj ) : Circle( rObj )
{   w = rObj.w;   }
```

```
BorderCircle  obj( 5, 2 );
BorderCircle  obj1;
BorderCircle  obj2( obj );
```



8.3 类的继承与派生

- 派生类的析构函数
  - 派生类对象中数据成员的析构顺序是：先执行派生类析构函数的函数体，清理新增成员；再调用基类析构函数，清理基类成员。简单地说，对象的析构顺序与构造顺序相反，即先析构新增成员，再析构基类成员

- 组合派生类的构造与析构
  - 从基类继承来的成员（基类成员），二是新增的对象成员，三是新增的非对象成员
  - 组合派生类的构造函数需依次初始化基类成员、新增对象成员、新增非对象成员。其中，初始化基类成员和新增对象成员需通过初始化列表，初始化新增的非对象成员则是在函数体中直接赋值
  - 组合派生类对象中各成员的析构顺序与其构造顺序相反，即先析构新增的非对象成员，再析构新增对象成员，最后才析构基类成员

例8-7 一个组合派生类的C++示意代码

类A1

```
1  class  A1
2  {
3  public:
4      int  a1;
5      A1( int x = 0 )  // 构造函数
6      {   a1 = x;   }
7  };
```

类A2

```
class  A2
{
public:
    int  a2;
    A2( int x = 0 )  // 构造函数
    {   a2 = x;   }
};
```

类B1

```
1  class  B1
2  {
3  public:
4      int  b1;
5      B1( int x = 0 )  // 构造函数
6      {   b1 = x;   }
7  };
```

类B2

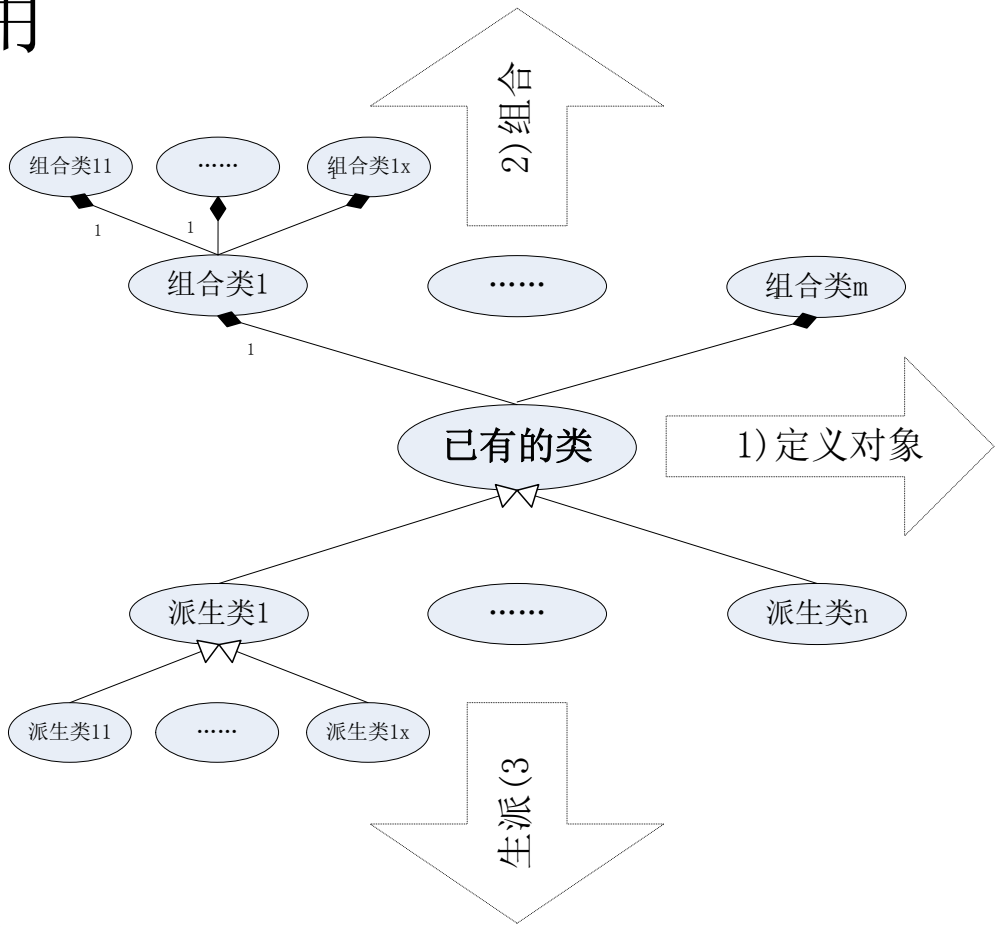
```
class  B2
{
public:
    int  b2;
    B2( int x = 0 )  // 构造函数
    {   b2 = x;   }
};
```

组合派生类C

```
class  C : public A1, public A2  // 继承基类A1、A2
{
public:
    B1  bObj1;  // 类B1的对象成员bObj1
    B2  bObj2;  // 类B2的对象成员bObj2
    int  c;
    // 组合派生类的构造函数：初始化基类成员、新增对象成员、新增非对象成员
    C( int p1=0, int p2=0, int p3=0, int p4=0, int p5=0 ) : A1(p1), A2(p2), bObj1(p3), bObj2(p4)
    {   c = p5;   }
};
```

8.3 类的继承与派生

- 继承与派生的应用
  - 重用类代码



8.3 类的继承与派生

- 继承与派生的应用
  - 凝练类代码

例8-8 本科生类和研究生类的C++示意代码

Undergraduate：本科生类

```
1  class Undergraduate  // 声明部分
2  {
3  public:
4      char Name[9], ID[11]; // 姓名、学号
5      int Age; // 年龄
6      double Score; // 课堂成绩
7      double PracticeScore; // 毕业设计成绩
8
9      void Input( ); // 输入学生信息
10     void ShowInfo( ); // 显示学生信息
11     double TotalScore( ); // 计算总成绩
12 };
13
```

Graduate：研究生类

```
class Graduate  // 声明部分
{
public:
    char Name[9], ID[11]; // 姓名、学号
    int Age; // 年龄
    double Score; // 课堂成绩
    double PaperScore; // 毕业论文成绩
    int Thesis; // 发表论文数量

    void Input( ); // 输入学生信息
    void ShowInfo( ); // 显示学生信息
    double TotalScore( ); // 计算总成绩
};
```

8.3 类的继承与派生

例8-9 通过抽象基类改写例8-8中类代码后的C++示意代码

Student：抽象出的基类（学生类，类头文件Student.h）

```
1  class Student  // 基类的声明部分
2  {
3  private:
4      char Name[9], ID[11]; // 姓名、学号
5      int Age; // 年龄
6      double Score; // 课堂成绩
7  public:
8      void Input( ); // 输入学生基本信息
9      void ShowInfo( ); // 显示学生基本信息
10 };
```

Undergraduate：本科生派生类

```
1  #include “Student.h”  // 声明基类Student
2  class Undergraduate : public Student
3  {
4  public:
5      double PracticeScore; // 毕业设计成绩
6
7      double TotalScore( ); // 计算总成绩
8      void Input( ); // 输入：同名覆盖
9      void ShowInfo( ); // 显示：同名覆盖
10 };
11
```

Graduate：研究生派生类

```
#include “Student.h”  // 声明基类Student
class Graduate : public Student
{
public:
    double PaperScore; // 毕业论文成绩
    int Thesis; // 发表论文数量

    double TotalScore( ); // 计算总成绩
    void Input( ); // 输入：同名覆盖
    void ShowInfo( ); // 显示：同名覆盖
};
```

8.4 多态性

- 多态性也是一个生物学概念，指的是生物会在不同层面上体现出形态的多样性
- 源程序中相同的程序元素可能会具有不同的语法解释，C++语言称这些程序元素具有多态性
  - 关键字多态、重载函数多态、运算符多态、对象多态、参数多态
- 对具有多态性的程序元素作出最终明确的语法解释，这称为多态的实现
- 不同多态形式具有不同的实现时间点，编译时实现的多态称为编译时多态，执行时实现的多态称为执行时多态

8.4 多态性

- 运算符的多态与重载

```
2 + 3  // 整数加法
2.5 + 3.5  // 浮点数加法

class Complex  // 复数类
{
private:
   double  real, image;  // 复数的实部和虚部
public:
   Complex(double x=0, double y=0) { real = x; image = y; } // 构造函数
   Complex(Complex &c) { real = c.real; image = c.image; } // 拷贝构造函数
   void Show( ) { cout << real << "+" << image << "i" << endl; } // 显示复数
};

Complex  c1(1, 3), c2(2, 4), c3;
c3 = c1 + c2;
```

8.4 多态性

- 重新定义C++语言已有运算符的运算规则，使同一运算符作用于不同类型数据时执行不同的运算，这就是运算符重载
  - 程序员可以为类重载运算符，实现类运算
  - 重载运算符使用函数的形式来重新定义运算符的运算规则

函数类型 operator 运算符(形式参数)
{　函数体　}

  - 将运算符函数定义为类的函数成员
  - 定义为类外的一个友元函数

8.4 多态性

例8-10 为复数类重载双目运算符“+”的C++示意代码

代码1：重载为复数类的**函数成员**

```
1  class Complex
2  {
3  private:
4       double  real, image;
5  public:
6       Complex(double x=0, double y=0)
7      {  real = x;  image = y;  }
8       Complex(Complex &c)
9       {  real = c.real;  image = c.image;  }
10      void Show( )
11      {
12       cout<<real<<“+”<<image<<“i”<<endl;
13      }
14      Complex  operator +( Complex c )
15      {
16         Complex result ;
17         result.real = real + c.real ;
18         result.image = image + c.image ;
19         return result ;
20      }
21 } ;
22
23
```

代码2：重载为复数类的**友元函数**

```
class Complex
{
private:
     double  real, image;
public:
     Complex(double x=0, double y=0)
    {  real = x;  image = y;  }
     Complex(Complex &c)
     {  real = c.real;  image = c.image;  }
     void Show( )
     {
      cout<<real<<“+”<<image<<“i”<<endl;
     }
   friend Complex operator +(Complex c1, Complex c2) ;
} ;

Complex  operator +( Complex c1, Complex c2 )
{
     Complex result ;
     result.real = c1.real + c2.real ;
     result.image = c1.image + c2.image ;
     return result ;
}
```

8.4 多态性

```
Complex  c1(1, 3), c2(2, 4), c3;
c3 = c1 + c2;  // 将对象c1+c2的和赋值给c3
c3.Show( );  // 显示复数c3，显示结果：3+7i
```

- 计算机执行“c1+c2”的加法运算相当于执行1次函数调用，其调用形式如下：
  - 若运算符“+”被重载为复数类的函数成员，则调用形式为“c1.+(c2)”。其中，c1是对象名，“.”是成员运算符，“+”是函数成员名，c2是实参。
  - 若运算符“+”被重载为复数类的友元函数，则调用形式为“+(c1, c2)”。其中，“+”是友元函数名，c1和c2是实参

8.4 多态性

例8-11 为复数类重载单目运算符“++”的C++示意代码

```
1  class Complex
2  {
3  private:
4      double  real, image;
5  public:
6      Complex(double x=0, double y=0)
7      {  real = x;  image = y;  }
8      Complex(Complex &c)
9      {  real = c.real;  image = c.image;  }
10     void Show( )
11     {  cout<<real<<“+”<<image<<“i”<<endl;  }
12     Complex &  operator ++( )  // 实现前置“++”运算符的函数成员
13     {
14         real ++;   image++;  // 运算规则：实部与虚部各加1
15         return  *this ;  // 返回前置“++”表达式的结果：加1之后对象的引用
16     }
17     Complex  operator ++( int )  // 实现后置“++”运算符的函数成员
18     {
19        Complex temp( *this );
20        real ++;   image++;  // 运算规则：实部与虚部各加1
21        return  temp ;  // 返回后置“++”表达式的结果：加1之前的对象
22     }
23 };
```

8.4 多态性

- 前置自增运算

```
Complex  c1(1, 3), c2;
c2 = ++c1;  // 前置自增运算
c1.Show( );  // 显示复数c1，显示结果：2+4i，实部与虚部各加1
c2.Show( );  // 显示复数c2，显示结果：2+4i，即c1自增之后的结果
```

- 后置自增运算

```
Complex  c1(1, 3), c2;
c2 = c1++;  // 后置自增运算
c1.Show( );  // 显示复数c1，显示结果：2+4i，实部与虚部各加1
c2.Show( );  // 显示复数c2，显示结果：1+3i，即c1自增之前的结果
```

8.4 多态性

- 关系运算符“==”

```
bool  Complex :: operator ==( Complex c )
{
    return ( real == c.real && image == c.image ) ;
}

Complex  c1(1, 3), c2(2, 4);
if (c1 == c2)  // 比较c1、c2是否相等
   // ..... 代码省略
```

8.4 多态性

- 赋值运算符“=”

```
Complex &  Complex :: operator =( Complex &c )
{
   real = c.real ;   image = c.image ; // 一一对应地拷贝数据成员
   return  *this ; // 返回赋值后对象的引用
}

Complex  c1(1, 3), c2;
c1.Show( ); // 显示复数c1，显示结果：1+3i
c2 = c1; // 将对象c1赋值给c2，即一一对应地拷贝数据成员
c2.Show( ); // 显示复数c2，显示结果与c1相同：1+3i
```

8.4 多态性

- 运算符重载的语法细则：
  - 除了下面的5个运算符，C++语言中的其它运算符都可以重载。5个不能重载的运算符是：
    条件运算符“?:”、sizeof运算符、成员运算符“.”、指针运算符“*”和作用域运算符“::”
  - 重载后，运算符的优先级和结合性不会改变
  - 重载后，运算符的操作数个数不能改变，同时至少要有一个操作数是自定义数据类型
  - 重载后，运算符的含义应与原运算符相似，否则会给使用类的程序员造成困惑

8.4 多态性

- 对象的替换与多态
  - 派生类是一种基类，具有基类的所有功能
  - 面向对象程序设计利用派生类和基类之间的这种特殊关系，常常将派生类对象当作基类对象来使用，或者用基类来代表派生类，其目的是为提高程序代码的可重用性

8.4 多态性

- 程序代码的可重用性

```
void fun( int x ) { ..... } // 处理int型数据的函数fun

fun( 5 ); // 正确：调用函数时，实参的数据类型应与形参一致
fun( 5.8 ); // 错误：5.8是double类型，与形参的类型不一致
```

  - 结论：C++语言对数据类型一致性的要求比较严格，属于强类型检查的计算机语言。C语言、Java语言也属于强类型检查的语言。因为数据类型不一致，不能重用函数fun的代码来处理double型数据

```
void fun( double x ) { ..... } // 处理double型数据的重载函数
```

8.4 多态性

- 程序代码的可重用性

```
class A { ..... }; // 定义一个类A
void aFun( A x ) { ..... } // 处理A类数据的函数aFun，x是一个A类对象

A aObj; // 定义一个A类对象aObj
aFun( aObj ); // 正确：调用函数时，实参的数据类型与形参一致

class B { ..... }; // 定义一个类B
B bObj; // 定义一个B类对象bObj
aFun( bObj ); // 错误：bObj的类型与函数aFun中形参的类型不一致
```

– 结论：不能重用函数aFun的代码来处理B类的对象数据

```
class B : public A { ..... }; // 类B公有继承类A，是类A的派生类
```

8.4 多态性

- 对象的替换与多态
  - 在面向对象程序设计中，重用处理基类对象的程序代码来处理派生类对象，这是非常普遍的需求
  - 如果派生类对象能够与基类对象一起共用程序代码，它将极大地提高程序开发和维护的效率
  - 面向对象程序设计方法利用派生类和基类之间存在的特殊关系，提出了对象的替换与多态
    - Liskov替换准则：将派生类对象当作基类对象来使用
    - 对象多态性：用基类来代表派生类

– 提高程序代码的可重用性

8.4 多态性

- 钟表类Clock

例8-12 钟表类Clock的C++示意代码（类头文件名Clock.h）

```
1  class Clock  // 钟表类Clock
2  {
3  private:
4      int  hour, minute, second;  // 时分秒：北京时间
5  public:
6      void Set( int h, int m, int s )  // 设置时间
7      {
8         hour = h;  minute = m;  second = s;
9      }
10     void Show( )  // 显示时间
11     {
12        cout << hour << ":" << minute << ":" << second;
13     }
14 };
```

```
Clock  obj;  // 定义1个类Clock的钟表对象obj
obj.Set( 8, 30, 0 );  // 将钟表对象obj的时间设为8点30分（北京时间）
obj.Show( );  // 显示钟表对象obj的时间，显示结果：8:30:0
```

例8-13 钟表类Clock的3个派生类示意代码

手表类：类头文件名Watch.h

```
#include “Clock.h” // 声明基类Clock

class Watch : public Clock // 公有继承
{
public:
    int band; // 表带类型
    void Show( ) // 显示时间：重写
    {
        cout << “Watch ”;
        Clock :: Show( );
    }
    // ..... 省略其余代码
};
```

挂钟类：类头文件名WallClock.h

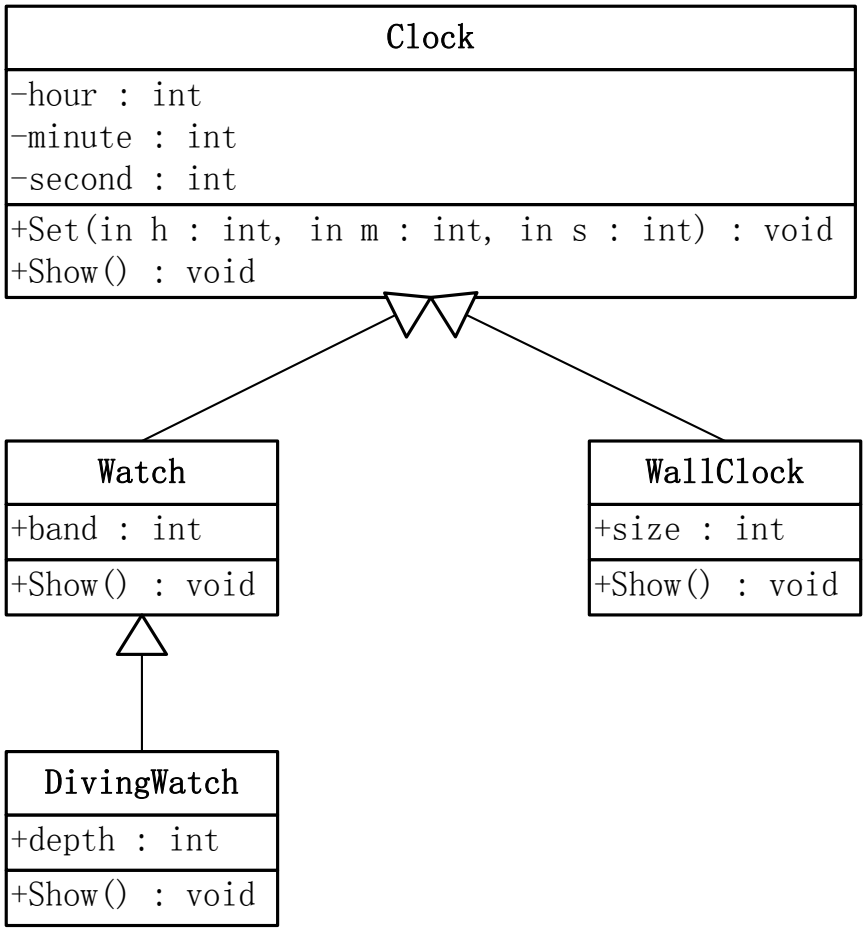
```
#include “Clock.h” // 声明基类Clock

class WallClock : public Clock // 公有继承
{
public:
    int size; // 表盘尺寸
    void Show( ) // 显示时间：重写
    {
        cout << “WallClock ”;
        Clock :: Show( );
    }
    // ..... 省略其余代码
};
```

潜水表类：类头文件名DivingWatch.h

```
#include “Watch.h” // 声明基类Watch

class DivingWatch : public Watch // 公有继承
{
public:
    int depth; // 最大下潜深度
    void Show( ) // 显示时间：重写
    {
        cout << “DivingWatch ”;
        Clock :: Show( );
    }
    // ..... 省略其余代码
};
```

8.4 多态性

- 基类及其下面的各级派生类共同组成了一个具有继承关系和共同特性的类的家族，我们称之为类族



8.4 多态性

- Liskov替换准则

```
void SetGMT( Clock &rObj,  int hGMT,  int mGMT,  int sGMT )
{
   rObj.Set( hGMT+8, mGMT, sGMT );  // 将小时数加8，即晚8个小时
}
```

讨论1：使用函数SetGMT给派生类对象设置格林尼治时间，重用函数SetGMT的代码，可以吗？

讨论2：函数SetGMT的形参rObj是基类Clock的引用，可以引用派生类Watch的对象吗？

```
Clock  obj;  // 定义1个基类Clock
SetGMT( obj, 8, 30, 0 );  // 将基类
obj.Show( );  // 显示基类对象ob
```

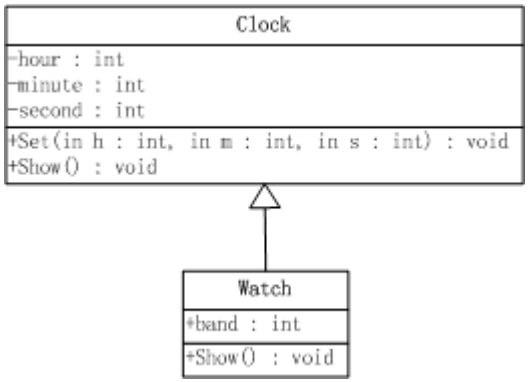
问题：

```
Watch  obj1;  // 定义1个派生类Watch的对象obj1
SetGMT( obj1, 8, 30, 0 );  // 将派生类对象obj1的时间设为8点30分（GMT时间）
obj.Show( );
```

8.4 多态性

- Liskov替换准则
  - 为了让基类对象及其派生类对象之间可以重用代码，C++语言制定了如下的类型兼容语法规则：
    - 派生类的对象可以赋值给基类对象；
    - 派生类的对象可以初始化基类引用；
    - 派生类对象的地址可以赋值给基类的对象指针，或者说基类的对象指针可以指向派生类对象。
  - 应用类型兼容语法规则有1个前提条件和1个使用限制：
    - 前提条件：派生类必须公有继承基类。
    - 使用限制：通过基类对象、引用或对象指针访问派生类对象，只能访问其基类成员

8.4 多态性



- Liskov替换准则

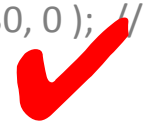
```
Watch  obj1; // 定义1个派生类Watch的对象obj1
obj1.Set( 8, 30, 0 ); // 将手表对象obj1的时间设为8点30分（北京时间）
obj1.Show( ); // 显示手表对象obj1的时间，显示结果：Watch 8:30:0
obj1.band = 1; // 设置手表对象obj1的表带类型，假设1代表皮革

// 演示1：将派生类对象obj1赋值给基类对象obj
Clock  obj;
obj = obj1; // 派生类的对象可以赋值给基类对象
obj.Show( ); // 访问赋值后基类对象obj的成员，显示其时间。显示结果：8:30:0
cout << obj.band; // 错误：赋值后基类对象obj不包含派生类对象obj1中的新增成员

// 演示2：通过基类引用rObj访问派生类对象obj1
Clock  &rObj = obj1; // 定义基类引用rObj，引用派生类对象obj1
rObj.Show( ); // 访问所引用派生类对象obj1的成员，显示其时间。显示结果：8:30:0
cout << rObj.band; // 错误：通过引用访问派生类对象obj1不能访问其新增成员

// 演示3：通过基类对象指针pObj访问派生类对象obj1
Clock  *pObj = &obj1; // 定义基类指针对象pObj，指向派生类对象obj1
pObj->Show( ); // 访问所指向派生类对象obj1的成员，显示其时间。显示结果：8:30:0
cout << pObj->band; // 错误：通过对象指针访问派生类对象obj1不能访问其新增成员
```

8.4 多态性

- Liskov替换准则
  - 应用Liskov替换准则，可以将派生类对象当作基类对象来处理，即用基类对象替换派生类对象。将派生类对象当作基类对象处理的好处是，使某些处理基类对象的代码可以被派生类对象重用

```
void SetGMT( Clock &rObj,  int hGMT,  int mGMT,  int sGMT )
{
   rObj.Set( hGMT+8, mGMT, sGMT );  // 将小时数加8，即晚8个小时
}
```

```
Clock  obj;  // 定义1个基类Clock的对象obj
SetGMT( obj, 8, 30, 0 );  // 将基类对象obj的时间设为8点30分（GMT时间）
obj.Show( );  // 显示基类对象obj的时间，显示结果：16:30:0

Watch  obj1;  // 定义1个派生类Watch的对象obj1
SetGMT( obj1, 8, 30, 0 );  // 将派生类对象obj1的时间设为8点30分（GMT时间）
obj.Show( );
```



8.4 多态性

- 对象多态性

显示**基类Clock**对象的时间
```
Clock  obj;   SetGMT( obj, 8, 30, 0 );
obj.Show( ); // 显示基类对象obj的时间，显示结果：16:30:0

void ShowBeijing( Clock &rObj )
{
  rObj.Show( );   cout << “（北京时间）”;
}

ShowBeijing( obj ); // 显示：16:30:0（北京时间）
```

rObj.Show( );

```
ShowBeijing( obj1 ); // 显示：16:30:0（北京时间）
```

讨论：如何重用函数ShowBeijing的代码，并能区分基类和派生类对象，再分别调用其对应的显示时间函数Show？

显示**派生类Watch**对象的时间
```
void ShowBeijing_Watch( Watch &rObj )
{
  rObj.Show( );
   cout << “（北京时间）”;
}
Watch  obj1;   SetGMT( obj1, 8, 30, 0 );
ShowBeijing_Watch( obj1 ); // 显示：Watch 16:30:0（北京时间）
```

8.4 多态性

- 什么是对象多态性？
  - 米老鼠、唐老鸭分别是老鼠类和鸭子类的对象
    - 向米老鼠下达指令“Go”，米老鼠将迈开4条腿迅速移动
    - 向唐老鸭下达指令“Go”，唐老鸭将迈开2条腿蹒跚而行
    - 不同对象在执行相同指令“Go”的时候会表现出不同的形态，这就是对象的多态性
  - 使用类A、类B分别定义对象aObj和bObj，假设它们都有1个名为Fun的函数成员，但其算法和功能都各不相同。将调用对象函数成员Fun的操作做如下类比：
    - 调用对象的函数成员Fun：aObj.Fun( ); bObj.Fun( ); 类似于向对象aObj、bObj分别下达了相同的指令“Fun”。
    - 执行函数Fun：类似于对象aObj、bObj各自执行指令“Fun”。
    - 完成不同的功能：类似于对象aObj、bObj表现出不同的形态

8.4 多态性

- 什么是对象多态性？
  - 面向对象程序设计借用拟人化的说法，将调用对象的某个函数成员称为向对象发送一条消息
  - 将执行函数成员完成某种程序功能称为对象响应该消息所表现出的行为
  - 不同对象接收相同的消息，但会表现出不同的行为，这就称为对象的多态性，或称对象具有多态性
  - 从程序角度，对象多态性就是：调用不同对象的同名函数成员，但所执行的函数不同，完成的程序功能不同。导致对象多态性的同名函数成员有3种不同形式：
    - 不同类之间的同名函数成员。类成员具有类作用域，不同类之间的函数成员可以重名，互不干扰。
    - 类中的重载函数。类中的函数成员之间可以重名，只要它们的形参个数不同，或类型不同。重载函数成员导致的多态在本质上属于重载函数多态。
    - 派生类中的同名覆盖。派生类中新增的函数成员可以与从基类继承来的函数成员重名，但它们不是重载函数。

8.4 多态性

- 应用Liskov替换准则，将派生类对象当作基类对象来处理，即用基类对象替换派生类对象。其目的是让某些处理基类对象的代码可以被派生类对象重用。这里“**某些**”代码的含义是：这些代码在通过基类的引用或对象指针访问派生类对象时，只需访问其基类成员

```
void SetGMT( Clock &rObj,  int hGMT,  int mGMT,  int sGMT )
{
   rObj.Set( hGMT+8, mGMT, sGMT );
   // 不管是基类或派生类对象，都调用基类成员Set
}
```

8.4 多态性

- 还有“**另外一些**”代码：这些代码在通过基类的引用或对象指针访问同一类族的对象时，需根据实际引用或指向的对象类型，自动调用该类同名函数成员中的新增成员（而不是基类成员）

```
void ShowBeijing( Clock &rObj )
{
   rObj.Show( ); // 希望能区分基类和派生类对象，自动调用对应的成员
   cout << “（北京时间）”;
}
```

- 应用对象多态性：相当于是用基类来代表派生类。通过基类引用或对象指针调用派生类对象的函数成员，应能够根据实际引用或指向的对象类型，自动调用该类同名函数成员中的新增成员

- C++语言使用虚函数的语法形式来实现类族中对象的多态性

8.4 多态性

- 实现对象多态性
  - 首先，在定义基类时使用“virtual”关键字将函数成员声明成虚函数
  - 然后通过公有继承定义派生类，并重写虚函数成员，也就是新增1个与虚函数同名的函数成员
  - 使用基类引用或对象指针
    - 调用基类对象的函数成员：自动调用基类成员
    - 调用派生类对象的函数成员
      - 普通函数成员：自动调用基类成员
      - 虚函数成员：自动调用派生类成员

8.4 多态性

- 虚函数的声明与调用

```
class A  // 基类A
{
public:
   virtual  void fun1( );  // 函数成员fun1被声明为虚函数
   void fun2( );  // 函数成员fun2被声明为非虚函数
};
void A::fun1 ( ) { cout << “Base class A: virtual fun1( ) called.” << endl; }
void A::fun2 ( ) { cout << “Base class A: non-virtual fun2( ) called.” << endl; }

class B : public A  // 派生类B
{
public:
   virtual  void fun1( );  // 重写基类的虚函数成员fun1
   void fun2( );  // 重写基类的非虚函数成员fun2
};
void B::fun1 ( ) { cout << “Derived class B: virtual fun1( ) called.” << endl; }
void B::fun2 ( ) { cout << “Derived class B: non-virtual fun2( ) called.” << endl; }
```

8.4 多态性

- 声明虚函数的语法细则
  - 只能在类声明部分声明虚函数。在类实现部分定义函数成员时不能使用“virtual”关键字
  - 基类中声明的虚函数成员被继承到派生类后，自动成为派生类的虚函数成员
  - 派生类可以重写基类虚函数成员。如果重写后的函数原型与基类虚函数成员完全一致，则该函数自动成为派生类的虚函数成员，无论声明时加不加“virtual”关键字
  - 类函数成员中的静态函数、构造函数不能是虚函数。析构函数可以是虚函数

```
A  aObj;  // 定义1个基类对象aObj
B  bObj;  // 定义1个派生类对象bObj

// 演示1：通过对象名分别调用虚函数成员fun1和非虚函数成员fun2，对比调用结果
```

```
// 演示2：通过基类引用分别调用虚函数成员和非虚函数成员，对比调用结果
A  &raObj = aObj;  // 定义1个基类引用raObj，引用基类对象aObj
raObj.fun1( );  // 调用结果：调用了基类对象aObj的虚函数成员fun1
raObj.fun2( );  // 调用结果：调用了基类对象aObj的非虚函数成员fun2
A  &rbObj = bObj;  // 定义1个基类引用rbObj，引用派生类对象bObj
rbObj.fun1( );  // 调用结果：调用了派生类对象bObj的新增函数成员fun1
rbObj.fun2( );  // 调用结果：调用了派生类对象bObj的基类函数成员fun2
// 结论2：通过基类引用访问派生类对象的虚函数成员将访问其新增成员（多态）
//  通过基类引用访问派生类对象的非虚函数成员将访问其基类成员
```

```
// 演示3：通过基类对象指针分别调用虚函数成员和非虚函数成员，对比调用结果
A  *paObj = &aObj;  // 定义1个基类对象指针paObj，指向基类对象aObj
paObj->fun1( );  // 调用结果：调用了基类对象aObj的虚函数成员fun1
paObj->fun2( );  // 调用结果：调用了基类对象aObj的非虚函数成员fun2
A  *pbObj = &bObj;  // 定义1个基类对象指针pbObj，指向派生类对象bObj
pbObj->fun1( );  // 调用结果：调用了派生类对象bObj的新增函数成员fun1
pbObj->fun2( );  // 调用结果：调用了派生类对象bObj的基类函数成员fun2
// 结论3：通过基类对象指针访问派生类对象的虚函数成员将访问其新增成员（多态）
//  通过基类对象指针访问派生类对象的非虚函数成员将访问其基类成员
```

8.4 多态性

```
class Clock  // 钟表类Clock
{
private:
    int  hour, minute, second;  // 时分秒：北京时间
public:
    void Set( int h, int m, int s )  // 设置时间
    {   hour = h;  minute = m;  second = s;   }
    virtual void Show( )  // 显示时间
    {
       cout << hour << ":" << minute << ":" << second;
    }
};
```

对象之间的多态性要满

员

，并重写虚函数成员（属于

- 只有满足这3个条件
  才会分别调用各自
  性

```
void ShowBeijing( Clock &rObj )  // 显示北京时间
{
   rObj.Show( );
   cout << "（北京时间）";
}

Clock  obj;   SetGMT( obj, 8, 30, 0 );
ShowBeijing( obj );  // 显示：16:30:0（北京时间）

Watch  obj1;   SetGMT( obj1, 8, 30, 0 );
ShowBeijing( obj1 );  // 显示：Watch 16:30:0（北京时间）
```

8.4 多态性

- 程序员该如何应用对象的替换与多态来提高程序代码的可重用性。为了让某个类族共用程序代码：
  - 定义基类时首先需确定将哪些函数成员声明成虚函数，将需要自动调用派生类新增的同名函数成员定义成虚函数
  - 定义派生类时公有继承基类，并重写那些从基类继承来的虚函数成员
  - 编写类族共用的程序代码时，需定义基类引用或对象指针来访问该类族的对象（不管是基类对象，还是派生类对象），然后通过基类引用或对象指针来调用派生类对象的函数成员，调用虚函数将自动调用派生类中重写的虚函数，否则将调用从基类继承来的函数成员

8.4 多态性

- 抽象类

```
class Circle  // 圆形类：声明部分
{
public:
   double r;  // 半径：数据成员
   double CArea( );  // 求面积：函数成员
   double CLen( );  // 求周长：函数成员
};
```

```
class Rectangle  // 长方形类：声明部分
{
public:
   double a, b;  // 长宽：数据成员
   double RArea( );  // 求面积：函数成员
   double RLen( );  // 求周长：函数成员
};
```

类定义中，“只声明，未定义”的函数成员被称为纯虚函数

```
class Shape  // 形状类：声明部分
{
public:
   virtual  double Area( ) = 0 ;  // 求面积：函数成员

   virtual  double Len( ) = 0 ;  // 求周长：函数成员
};
```

8.4 多态性

- 含有纯虚函数成员的类就是**抽象类**
  - **抽象类不能实例化**

    不能使用抽象类定义对象（即不能实例化），因为抽象类中含有未定义的纯虚函数，其类型定义还不完整。但可以定义抽象类的引用、对象指针，所定义的引用、对象指针可以引用或指向其派生类的实例化对象。

  - **抽象类可以作为基类定义派生类**
    - 抽象类可以作为基类定义派生类。派生类继承抽象类中除构造函数、析构函数之外的所有成员，包括纯虚函数成员
    - 纯虚函数成员只声明了函数原型，没有定义函数体代码。因此派生类继承纯虚函数成员时，只是继承其函数原型，即函数接口。派生类需要为纯虚函数成员编写函数体代码，称为**实现**纯虚函数成员
    - 派生类如果实现了所有的纯虚函数成员，那么它就变成了一个普通的类，可以实例化

8.4 多态性

	派生类Circle	派生类Rectangle
1	class Circle : public Shape // 圆形类	class Rectangle : public Shape // 长方形类
2	{	{
3	public:	public:
4	double r; // 半径：新增数据成员	double a, b; // 长宽：数据成员
5	Circle(double x=0) // 构造函数	Rectangle(double x=0, double y=0) // 构造函数
6	{ r = x; }	{ a = x; b = y; }
7	double Area( ) // 实现纯虚函数Area	double Area( ) // 实现纯虚函数Area
8	{ return (3.14*r*r); }	{ return (a*b); }
9	double Len( ) // 实现纯虚函数Len	double Len( ) // 实现纯虚函数Len
10	{ return (3.14*2*r); }	{ return (2*(a+b)); }
11	};	};

8.4 多态性

- 抽象类的应用
  - 统一类族接口

    通常，派生类继承基类是为了重用基类的代码。如果基类是抽象类，其中的纯虚函数成员并没有定义函数体代码，只声明了函数原型。基类声明纯虚函数成员的目的不是为了重用其代码，而是为了统一类族对外的接口。在基类中声明纯虚函数成员，各派生类按照各自的功能要求实现这些纯虚函数，这样类族中所有的派生类都具有相同的接口

```
Circle  cObj( 10 );  // 定义1个圆形类对象cObj
Rectangle rObj( 5, 10 );  // 定义1个长方形类对象rObj

cout << cObj.Area( ) << "," << cObj.Len( );  // 显示圆形对象cObj的面积和周长
cout << rObj.Area( ) << "," << rObj.Len( );  // 显示长方形对象rObj的面积和周长
```

8.4 多态性

- 抽象类的应用
  - 重用代码

    抽象类中定义的纯虚函数具有虚函数的特性，调用时具有多态性。在基类中声明纯虚函数成员的另一个目的是利用虚函数调用时的多态性，让类族中的所有派生类对象可以重用相同的代码

```
void ShapeInfo( Shape *pObj )  // 显示面积和周长等形状信息
{   cout << pObj->Area( ) << "," << pObj->Len( ) << endl;   }

int main( )
{
   Circle  cObj( 10 );  // 定义1个圆形类对象cObj
   Rectangle rObj( 5, 10 );  // 定义1个长方形类对象rObj
   ShapeInfo( &cObj );  // 重用函数ShapeInfo，显示圆形对象cObj的信息
   ShapeInfo( &rObj );  // 重用函数ShapeInfo，显示长方形形对象rObj的信息
   return 0;
}
```

8.5关于多继承的讨论

- 派生类可以从多个基类继承，这就是**多继承**。多继承派生类存在比较复杂的成员重名问题，其具体表现形式有3种：
  - **新增成员与基类成员重名**。在新增成员与基类成员重名的情况下访问派生类对象，所访问到的是新增成员，还是基类成员？这要由访问形式来决定：
    - 如果通过派生类的对象名、引用或对象指针访问派生类对象，则访问到的是新增成员，此时新增成员覆盖同名的基类成员（**同名覆盖**）。
    - 如果通过基类的对象名、引用或对象指针访问派生类对象，则访问到的是基类成员。此时派生类对象被当作基类对象使用（**Liskov替换准则**）。
    - 如果基类定义虚函数成员，派生类公有继承基类并重写虚函数（属于新增成员），那么通过基类的引用或对象指针访问派生类对象所访问到的将是新增的虚函数成员。这就是调用对象中虚函数成员时所呈现出的多态性（**对象多态性**）。
  - **多个基类之间的成员重名**。如果多个基类之间有重名的成员，同时继承这些基类会造成派生类中基类成员之间的重名。
  - **同一基类被重复继承**。多级派生时，从同一基类派生出多个派生类，这多个派生类再被多继承到同一个下级派生类。该下级派生类将包含多份基类成员的拷贝，也就是同一基类被重复继承

8.5关于多继承的讨论

例8-14 一个双继承派生类的C++示意代码

基类A1

```
1  class  A1
2  {
3  public:
4     int  a1;
5     int  a;
6     void  fun( )
7     {   cout << a1 << “,” << a << endl;   }
8  };
```

基类A2

```
class  A2
{
public:
   int  a2;
   int  a;
   void  fun( )
   {   cout << a2 << “,” << a << endl;   }
};
```

双继承派生类B

```
1  class  B : public A1, public A2  // 同时继承基类A1、A2
```

继承A1：a1、a、fun　　继承A2：a2、a、fun

```
B  bObj;  // 定义派生类对象bObj
cin >> bObj.a1 >> bObj.a2;  // 访问不重名的基类成员，直接使用成员名
cin >> bObj.A1::a >> bObj.A2::a;  // 访问重名的基类成员，需在成员名前加“基类名::”
bObj.A1::fun( );  // 调用从类A1继承来的基类函数成员fun，显示a1和A1::a的值
bObj.A2::fun( );  // 调用从类A2继承来的基类函数成员fun，显示a2和A2::a的值
```

8.5关于多继承的讨论

例8-15 一个重复继承的C++示意代码

基类A

```
1  class  A
2  {
3  public:
4      int  a;
5      void  fun( )   {   cout << a << endl;   }
6  };
```

虚基类

```
1  class  A1 : virtual public A  // 继承虚基类A
2  {
3  public:
4     // ..... 不新增任何新成员
5     // 派生类A1继承了1份基类A的成员
6  };
```

```
class  A2 : virtual public A  // 继承虚基类A
{
public:
   // ..... 不新增任何新成员
   // 派生类A2也继承了1份基类A的成员
};
```

```
class  B : public A1, public A2  // 同时继承类A1、A2
{
public:
   // ..... 不新增任何新成员
   // 派生类B同时继承类A1、A2，继承后将包含2份完全相同的基类A的成员
};
```

8.5关于多继承的讨论

- 类的继承与派生主要有2个作用：
  - **重用类代码**。派生类继承基类，就是重用基类的代码，使用其功能，从而提高程序的开发效率
  - **统一类族接口**。抽象类中包含纯虚函数成员。纯虚函数只声明函数原型，没有定义代码，没有实现任何功能。如果基类是抽象类，在基类中声明纯虚函数成员，其派生类按照各自的功能要求实现这些纯虚函数。这样以该基类为根的类族中的所有派生类都将具有相同的对外接口，更便于类族的使用

- Java和C#语言
  - 只允许单继承。派生类只能继承一个基类，即只能重用一个基类的代码
  - 但为了统一类族接口，它们引入了一个新的概念：**接口**

本章要点

- 需学会使用组合和继承的方法来定义新类，这样可以提高类代码的开发效率
- 应理解类在组合或继承时可以进行二次封装
- 应从提高对象处理算法代码可重用性的角度，这样可以更容易地理解对象多态性
- 只需要了解多继承的基本原理即可