计算机程序设计

刘龙军 副教授

2025年10月21日

目录

类与对象

引子---软件怎样组织?

- 计算机里面编程:
  - 按钮对象:
  - 按钮的内容、大小，按钮的字体、图案等等
  - 针对按钮的各种操作，创建、单击、双击、拖动等
- 班级对象:
  - 班级的静态特征，所属的系和专业、班级的人数，所在的教室等。这种静态特征称为属性;
  - 班级的动态特征，如学习、开会、体育比赛等，这种动态特征称为行为。

引子

- 任何一个对象都应当具有这两个要素，一是属性(attribute)；二是行为(behavior)，即能根据外界给的信息进行相应的操作。**对象是由一组属性和一组行为构成的。**
- 面向对象的程序设计采用了以上**人们所熟悉的这种思路**。使用面向对象的程序设计方法设计一个复杂的软件系统时，首要的问题是确定该系统是由***哪些对象组成的***，并且***设计这些对象***。在C++中，每个对象都是由数据和函数(即操作代码)这两部分组成的。

引子

- 对一个对象进行封装处理，把它的一部分属性和功能对外界屏蔽，也就是说从外界是看不到的、甚至是不可知的。
- 使用对象的人完全可以不必知道对象内部的具体细节，只需了解其外部功能即可自如地操作对象。
- 把对象的内部实现和外部行为分隔开来

引子

- 传统的面向过程程序设计是围绕功能进行的，用一个函数实现一个功能。**所有的数据都是公用的**，一个函数可以使用任何一组数据，而一组数据又能被多个函数所使用。程序设计者必须考虑每一个细节，什么时候对什么数据进行操作 。
- 面向对象程序设计采取的是另外一种思路。它面对的是一个个对象。

  实际上，每一组数据都是有特定的用途的，是某种操作的对象。也就是说，**一组操作调用一组数据**。

对象

- 程序设计者的任务包括两个方面：一是设计所需的各种类和对象，即决定把哪些数据和操作封装在一起；二是考虑怎样向有关对象发送消息，以完成所需的任务。各个对象的操作完成了，整体任务也就完成了。
- 因此人们设想把相关的数据和操作放在一起，形成一个整体，与外界相对分隔。这就是面向对象的程序设计中的对象。

面向过程的程序组成：

- 在面向过程的结构化程序设计中，人们常使用这样的公式来表述程序

程序=算法＋数据结构

面向对象的程序组成：

- 对象 = 算法 + 数据结构
- 程序=（对象+对象+对象+……）+ 消息
- 消息的作用就是对对象的控制。
- 程序设计的关键是设计好每一个对象以及确定向这些对象发出的命令，使各对象完成相应的操作。

说明

- 每一个实体都是对象。有一些对象是具有相同的结构和特性的。
- 每个对象都属于一个特定的类型。
- 在C++中对象的类型称为类(class)。类代表了某一批对象的共性和特征。类是对象的抽象，而对象是类的具体实例(instance)。

类的定义

- 类是一种复杂的数据**类型**，它是将不同类型的数据和与这些数据相关的运算***封装***在一起的 集合体。
- 类将一些数据及与数据相关的**函数**封装在一起，使类中的数据得到很好的“保护”。在大型程序中**不会被随意修改**。

说明


```
class  Student
{ private :
    char Name[20];
    float Math;
    float Chiese;
  public :
    float  average;
    void SetName(char *name);
    void  SetMath(float math);
    void  SetChinese(float ch);
    float GetAverage(void);
};
```

说明

- 用关键字priviate限定的成员称为私有成员，对私有成员限定在该类的内部使用，即只允许该类中的成员函数使用私有的成员数据，
- 对于私有的成员函数，只能被该类内的成员函数调用；
- 类就相当于私有成员的作用域。

作用域

- 作用域是指程序中所说明的标识符在哪一个区间内有效，即在哪一个区间内可以使用或引用该标识符。
- 在C++中，作用域共分为五类：**块作用域、文件作用域、函数原型作用域、函数作用域和类的作用域。**

块作用域

- 我们把用花括号括起来的一部分程序称为一个块。在块内说明的标识符，只能在该块内引用，即其作用域在该块内，开始于标识符的说明处，结束于块的结尾处。
- 在一个函数内部定义的变量或在一个块中定义的变量称为局部变量

说明

- 在函数内或复合语句内部定义的变量，其作用域是从定义的位置起到函数体或复合语句的结束。形参也是局部变量。

```
float  f1( int  a)
{  int  b,c;
   .....
}
```

a,b,c有效

```
void main(void )
{  int  m, n;
   .....
}
```

m,n有效

```
float  f2( int  x,  int
y)
{  int  i, j;
   .....
}
```

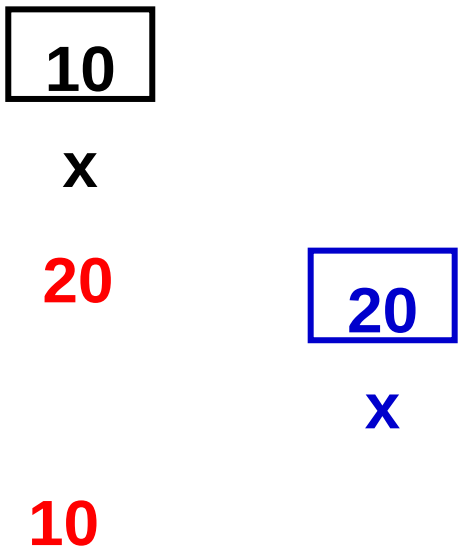
x,y,i,j 有效

说明

- 主函数main中定义的变量，也只在主函数中有效，同样属于局部变量
- 不同的函数可以使用相同名字的局部变量，它们在内存中分属不同的存储区间，互不干扰。

定义变量既是在内存中开辟区间

```
void main(void)
{   int  x=10;
    {  int  x=20;
       cout<<x<<endl;
    }
    cout<<x<<endl;
}
```



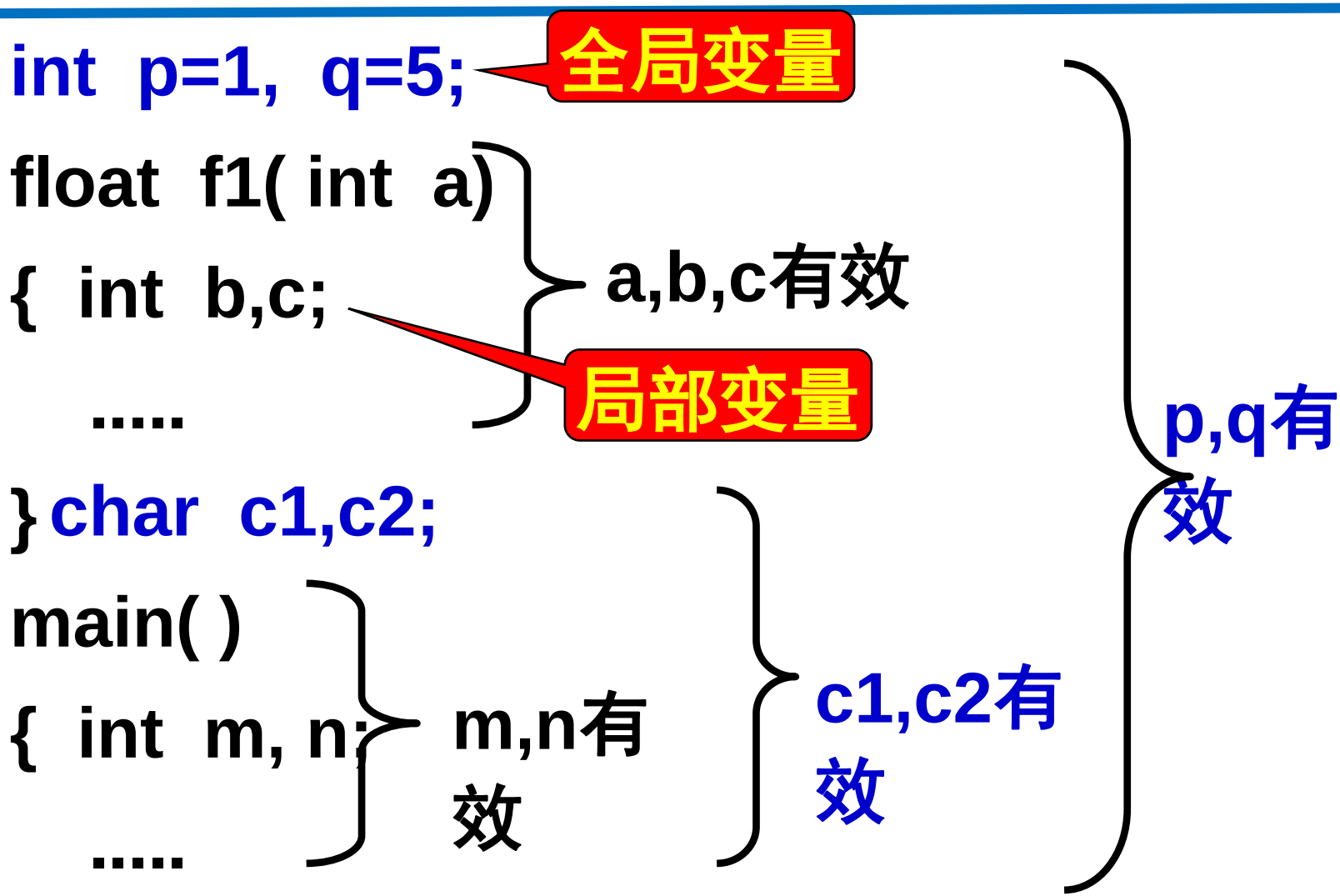
说明

- 具有块作用域的标识符在其作用域内，将屏蔽其作用块包含本块的同名标识符，即
- 变量名相同，局部更优先。

文件作用域

- 在函数外定义的变量称为全局变量。
- 全局变量的作用域称为文件作用域，即在整个文件中都是可以访问的
- 其缺省的作用范围是:从定义全局变量的位置开始到该源程序文件结束
- 当在块作用域内的变量与全局变量同名时，局部变量优先（近水楼台先得月）。

说明

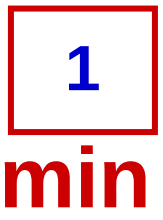


全局变量增加了函数间数据联系的渠道，在函数调用时可以得到多于一个的返回值。

说明（内存段空间）

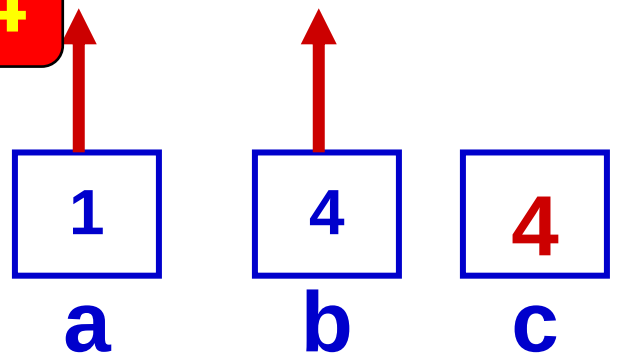
全局变量

```
int  min;
int  max (int x, int  y)
{   int  z;
     min=(x<y)?x : y;
     z=(x>y)? x : y ;
     return  z;
}
void main (void)
{   int   a,b,c;
     cin>>a>>b;
     c=max (a , b) ;
     cout<<“The max is”<<c<<endl;
     cout<<“ The min
is”<<min<<endl;  }
```



min 在main( )和max( )中均有效，在内存中有唯一的存储空间。

函数值为4

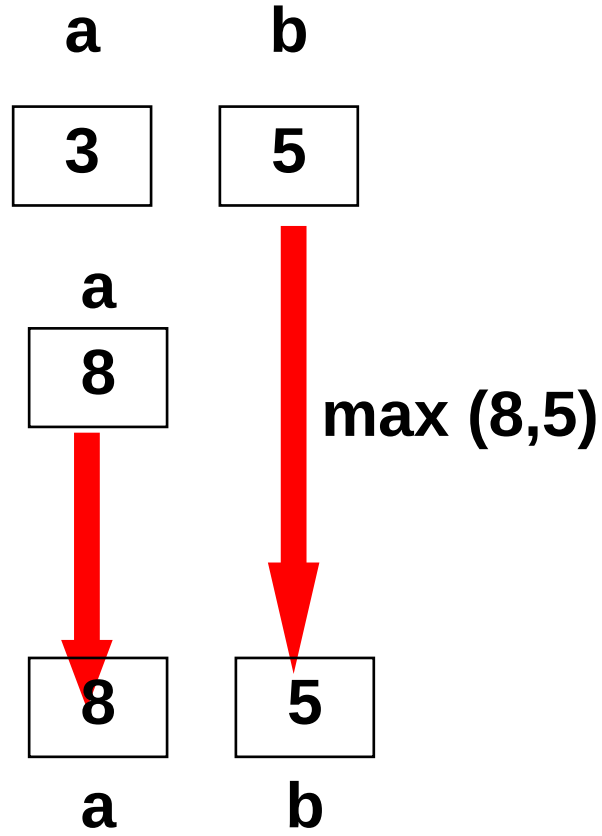



```
The max is  4
The min  is  1
```

说明

- 在同一个源文件中，外部变量与局部变量同名，**则在局部变量的作用范围内，外部变量不起作用**。

```
int  a=3, b=5;
int max(int  a, int  b)
{  int c;
    c=a>b? a:b;
    return  c;
}
void main(void)
{  int   a=8;
    cout<<max(a,b)<<endl;
}
```



输出：8

类内作用域

- 用关键字public限定的成员称为公有成员，公有成员的数据或函数不受类的限制，**可以在类内或类外自由使用**；对类而言是透明的。
- 用关键字protected所限定的成员称为保护成员，只允许在类内及该类的**派生类**中使用**保护的数据或函数**。即保护成员的作用域是**该类及该类的派生类**。

说明

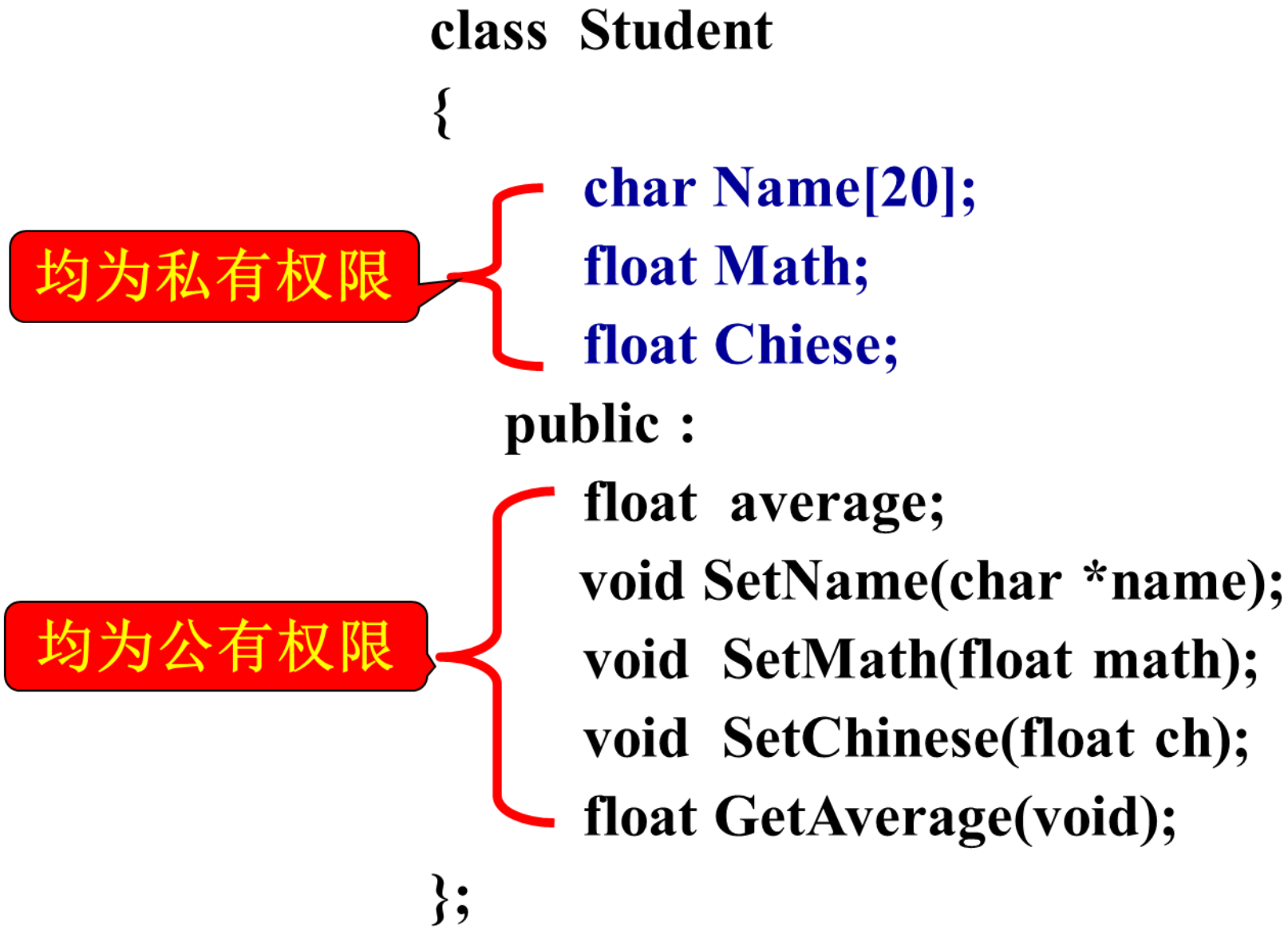
	私有成员	公有成员	保护成员
类内函数	可以调用	可以调用	可以调用
类外函数	不可调用	可以调用	不可调用

	私有函数	公有函数	保护函数
类内函数	可以调用	可以调用	可以调用
类外函数	不可调用	可以调用	不可调用

说明

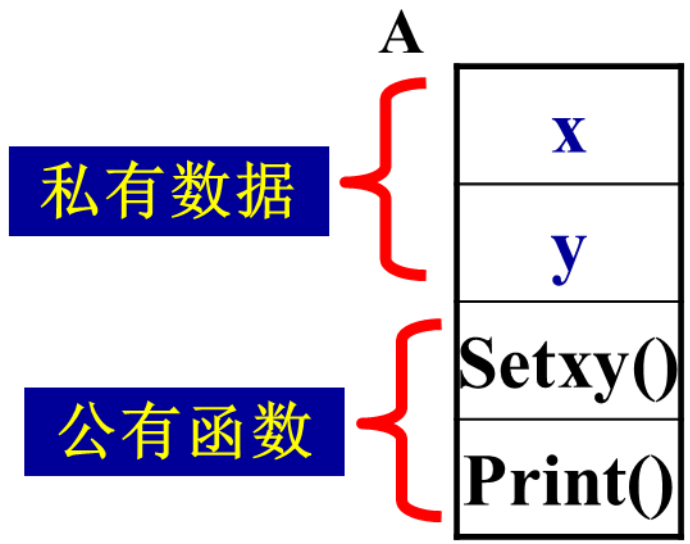
- 每一个限制词(private等)在类体中可使用多次。一旦使用了限制词，该限制词一直有效，直到下一个限制词开始为止。
- 如果未加说明，类中成员**默认的访问权限是private**，即私有的。

说明


class Student
{
char Name[20];
float Math;
float Chiese;
public :
float average;
void SetName(char *name);
void SetMath(float math);
void SetChinese(float ch);
float GetAverage(void);
};
均为私有权限
均为公有权限

说明

```
class  A
{     float  x, y;
  public:
      void Setxy(float a,float  b)
     {  x=a;   y=b;   }
      void  Print(void)
    {   cout<<x<<'\t'<<y<<endl; }
};
```



- 在类外不能直接使用 x 或 y，必须通过Setxy()给 x 或 y 赋值，通过Print()输出 x 或 y 。

说明

- 成员函数与成员数据的定义不分先后，可以先说明函数原型，再在类体外定义函数体。

```
class  A
{      float  x, y;
 public:
    void Setxy(float a,float  b)
   {  x=a;   y=b;   }
    void  Print(void)
   {   cout<<x<<'\t'<<y<<endl;  }
};
```

在类体内定义成员函数

说明

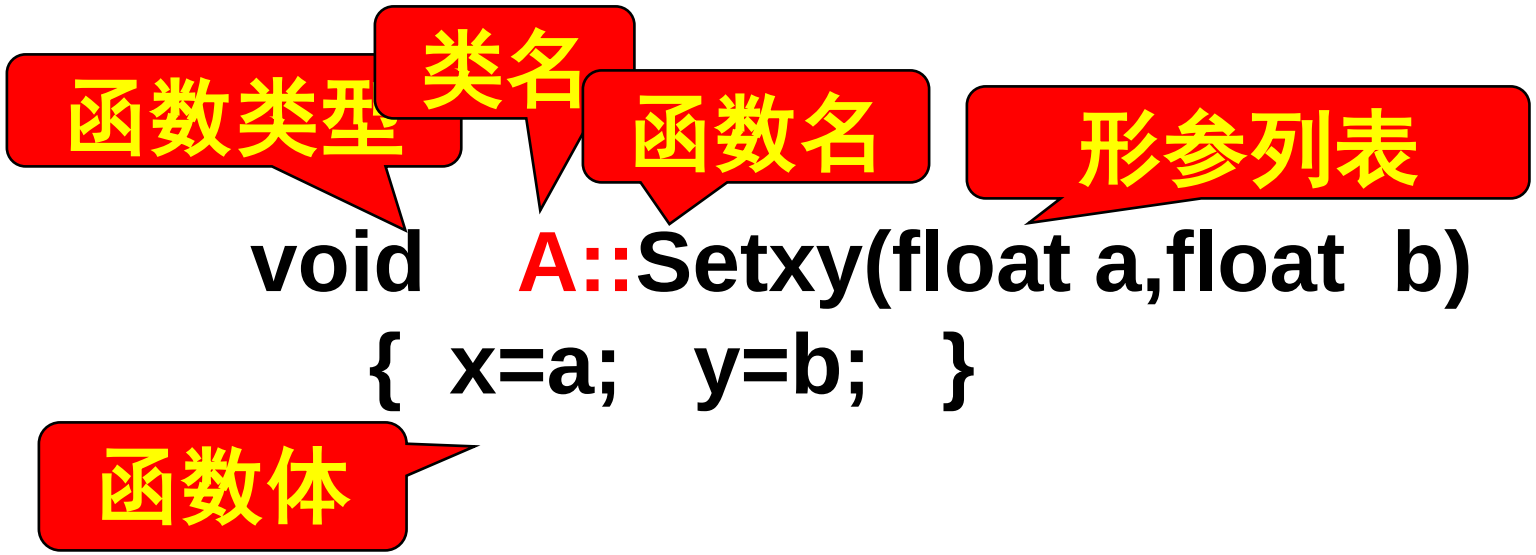
```
class  A
{      float  x, y;
 public:
     void Setxy(float a,float  b);
     void  Print(void);
};
void A::Setxy(float a,float  b)
    {  x=a;   y=b;   }
void  A::Print(void)
    {   cout<<x<<'\t'<<y<<endl;  }
```

在类体内说明成员函数原型

在类体外定义成员函数

在类体外定义成员函数的格式：

```
<type>   < class_name > :: < func_name > (<参数表>)
{
..... //函数体  （与一般函数的对比）
}
```



说明

- 在定义一个类时，要注意如下几点：
- 1、类具有封装性，并且类只是定义了一种结构
- 2、在定义类时，只是定义了一种导出的数据类型，并不为类分配存储空间，所以，在定义类中的数据成员时，不能对其初始化。如：

```
class  Test
{
    int  x=5,y=6;   //是不允许的
}
```

对象

- 在定义类时，只是定义了一种**数据类型**，即说明程序中可能会出现该类型的数据，并不为类分配存储空间。
- 只有在定义了属于类的变量后，系统才会为类的变量分配空间

类的变量我们称之为对象。

- 对象是**类的实例**，定义对象之前，一定要先说明该对象的类。

说明

- 不同对象占据内存中的不同区域，它们所保存的数据各不相同，但对成员数据进行操作的成员函数的程序代码均是一样的。
- 对象的定义格式：
- 《存储类型》类名　对象1，对象2《,.....》；

Student　st1，st2；

类名　对象名

- 在建立对象时，只为对象分配用于保存数据成员的内存空间，而成员函数的代码为该类的每一个对象所共享。

说明

- 对象的定义方法同结构体定义变量的方法一样，也分三种，**当类中有数据成员的访问权限为私有时，不允许对对象进行初始化。**

```
class  A {
        float  x,y;
public:
       void Setxy( float a, float b  ){  x=a;   y=b;   }
       void  Print(void) {  cout<<x<<'\t'<<y<<endl;  }
} a1,a2;
```

定义全局对象

```
void  main(void)
{   A   a3,a4;
}
```

定义局部对象

对象的使用

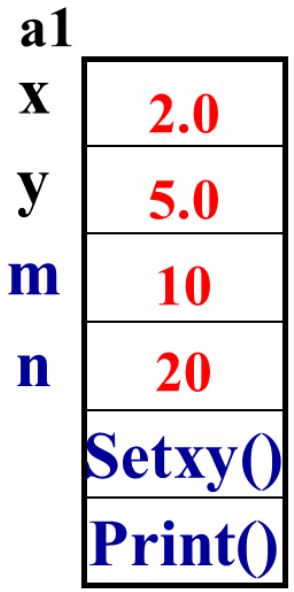
- 一个对象的成员就是该对象的类所定义的成员，有成员数据和成员函数，引用时同结构体变量类似，用“.”运算符。

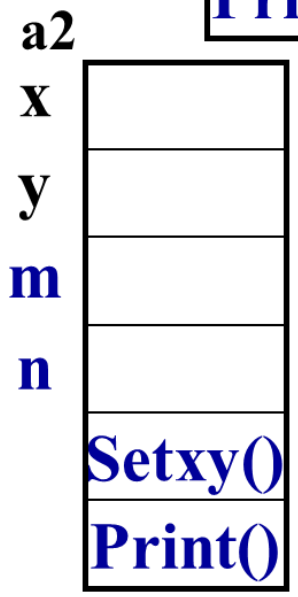
说明

```
class  A {
       float  x,y;
public:
       float   m,n;
       void Setxy( float a, float b  ){  x=a;   y=b;   }
       void  Print(void) {  cout<<x<<'\t'<<y<<endl;  }
};

void main(void)
{   A  a1,a2;   //定义对象
    a1.m=10;   a1.n=20; //为公有成员数据赋值
    a1.Setxy(2.0 , 5.0); //为私有成员数据赋值
    a1.Print();
}
```

输出： 2　　5





说明

- 用成员选择运算符“.”只能访问对象的公有成员，而不能访问对象的私有成员或保护成员。若要访问对象的私有的数据成员，只能通过对象的公有成员函数来获取。

说明

```
class  A {
        float  x,y;
public:
        float   m,n;
        void Setxy( float a, float b  ){  x=a;   y=b;   }
        void  Print(void) {  cout<<x<<'\t'<<y<<endl;  }
};
void main(void)
{    A  a1,a2;
     a1.m=10;   a1.n=20; //为公有成员数据赋值
     a1.x=2;  a1.y=5;
     a1.Setxy(2.0,5.0);
    a1.Print();
}
```

必须通过类内公有函数访问私有数据成员

非法，私有成员不能在类外访问

说明

- 同类型的对象之间可以整体赋值，这种赋值与对象的成员的访问权限无关

```
class  A {
       float  x,y;
public:
      float   m,n;
      void Setxy( float a, float b  ){  x=a;   y=b;   }
      void  Print(void) {  cout<<x<<'\t'<<y<<endl;  }
};
void main(void)
{   A  a1,a2;
     a1.m=10;   a1.n=20;        //为公有成员数据赋值
     a1.Setxy(2.0,5.0);
     a2=a1;
     a1.Print();   a2.Print();
}
```

同类型的对象之间可以整体赋值

相当于成员数据间相互赋值

说明

- 对象可以作函数的入口参数（实参、形参），也可以作函数的出口参数。这与一般变量作为函数的参数是完全相同的。
- 可以定义类类型的指针，类类型的引用，对象数组，指向类类型的指针数组和指向一维或多维对象数组的指针变量
- 一个类的对象，可作为另一个类的成员

类作用域、类类型的作用域和对象的作用域

- 类体的区域称为类作用域。类的成员函数与成员数据，其作用域都是属于类的作用域，仅在该类的范围内有效，**故不能在主函数中直接通过函数名和成员名来调用函数。**

说明

```
class  A {
        float  x,y;
public:
        float   m,n;
        void Setxy( float a, float b  ){  x=a;   y=b;   }
        void  Print(void) {  cout<<x<<'\t'<<y<<endl;  }
};
  void main(void)
  {    A  a1,a2;
        a1.m=20;   a1.n=10;
        a1.Setxy(2.0,   5.0);
       a1.Print();
  }
```

用对象名调用

```
void main(void)
{  A  a1,a2;
    m=20;  n=10;
    Setxy(2.0,   5.0);
    Print();
}
```

不能直接调用

说明

- **类类型的作用域**：在函数定义之外定义的类，**其类名的作用域为文件作用域**；而在函数体内定义的类，**其类名的作用域为块作用域** 。
- 对象的作用域与前面介绍的变量作用域完全相同，**全局对象、局部对象、局部静态对象**等。

说明

```
class  A {
      float  x,y;
public:
      float   m,n;
      void Setxy( float a, float b  ){  x=a;   y=b;   }
      void  Print(void) {  cout<<x<<'\t'<<y<<endl;  }
}a3,a4;
void main(void)
{   A  a1,a2;
    class B{
            int i,j;
    public :
         void Setij(int  m,  int n){   i=m;   j=n;  }
     };
    B   b1,b2;
    a1.Setxy(2.0,  5.0);    b1.Setij(1,2);
}
```

类A：文件作用域，在整个文件中有效

全局对象

局部对象

类B：块作用域，在函数内有效。

类的嵌套

- 在定义一个类时，在其类体中又包含了一个类的完整定义，称为类的嵌套。
- 类是**允许嵌套定义的**。

说明

```
class  A {
      class B{
            int i,j;
      public :
          void Setij(int  m,  int n){  i=m;  j=n;  }
       };
       float  x,y;
public:
      B  b1,b2;
      void Setxy( float a, float b  ){  x=a;   y=b;   }
      void  Print(void) {  cout<<x<<'\t'<<y<<endl;  }
};
```

类B包含在类A中，为嵌套定义

嵌套类的对象

在类A的定义中，并不为b1,b2分配空间，只有在定义类A的对象时，才为嵌套类的对象分配空间。嵌套类的作用域在类A的定义结束时结束。

类引用举例（三角形类：三角形的三边及与三边相关的运算）

```
class Triangle{
private:
        float  a,b,c;   //三边为私有成员数据
public:
        void  Setabc(float x, float y, float z);//置三边的值
        void  Getabc(float  &x,  float &y,  float &z);//取三边的值
        float  Perimeter(void);//计算三角形的周长
        float Area(void);//计算三角形的面积
        void  Print(void);//打印相关信息
};
```

说明

```
void Triangle::Setabc (float x,float y,float z)
    {a =x;  b=y;  c=z;   } //置三边的值
void  Triangle::Getabc (float &x,float &y,float &z) //取三边的值
          {x=a;   y=b;    z=c;}
float Triangle::Perimeter (void)
 {return (a+b+c)/2;} //计算三角形的周长
float Triangle::Area (void)   //计算三角形的面积
{float area，p;
  p= Perimeter();
 area=sqrt((p-a)*(p-b)*(p-c)*p);
 return area;
}
void Triangle::Print(void) //打印相关信息
{cout<<"Peri="<<Perimeter()<<'\t'<<"Area="<<Area()<<endl;}
```

说明

```
void main(void)
{
        Triangle  Tri1; //定义三角形类的一个实例（对象）
        Tri1.Setabc (4,5,6);    //为三边置初值
        float x,y,z;
        Tri1.Getabc (x,y,z);   //将三边的值为x,y,z赋值
        cout<<x<<'\t'<<y<<'\t'<<z<<endl;
        cout<<“s=”<<Tri1.Perimeter ()<<endl;//求三角形的周长
        cout<<“Area=”<<Tri1.Area ()<<endl;//求三角形的面积
        cout<<"Tri1:"<<endl;
        Tri1.Print ();//打印有关信息
}
```

类引用举例（学生类：学生的姓名成绩及相关的运算）


```
class Stu
{       char Name[20];          //学生姓名
        float Chinese;          //语文成绩
        float Math;             //数学成绩
public:
        float Average(void);    //计算平均成绩
        float Sum(void);        //计算总分
        void Show(void);        //打印信息
        void SetStudent(char*,float,float);//为对象置姓名、成绩
        void SetName(char *);           //为对象置姓名
        char *GetName(void);            //取得学生姓名
};
```

说明

```
float Stu::Average(void){  return (Chinese+Math)/2;}//平均成绩
float Stu::Sum(void){ return Chinese+Math;  }//总分
void Stu::Show(void) //打印信息
{   cout<<"Name:  "<<Name<<endl<<"Score:  "<<Chinese<<'\t'<<
  Math<<'\t'<<"average:  "<<Average()<<'\t'<<"Sum:   "<<Sum()<<endl;
 }
void Stu::SetStudent(char *name,float chinese,float math)
{       strcpy(Name,name);   //置姓名
        Chinese=chinese;      //置语文成绩
        Math=math;            //置数学成绩   }
void Stu::SetName(char *name) {strcpy(Name,name);};//置姓名
char * Stu::GetName(void){  return Name;}//返回姓名
```

说明

```
void main(void)
{       Stu  p1,p2;
        p1.SetStudent(“Li   qing”,98,96);//对象置初值
        p2.SetStudent("Wang Gang",90,88); //对象置初值
        p1.Show();//打印信息
        p2.Show();//打印信息
        p1.SetName (“Zhao jian”);//重新置p1对象的名字
        p1.Show ();
     cout<<“p1.Name:  ”<<p1.GetName ()<<endl;//打印对象的名字
    cout<<“p1.average:  ”<<p1.Average ()<<endl;//打印对象的成绩
}
```

成员函数的重载

- 类中的成员函数可以带有缺省的参数，也可以重载成员函数 。
- 重载时，函数的形参必须在类型或数目上不同。

说明

```
class  Test{
                int  x , y;
                int  m,  n;
   public:
      void Setxy(int  a, int b){x=a;    y=b;}
      void  Setxy(int a,int b,int c,int d){ x=a;y=b;m=c;n=d;}
      void Printxy(int x){cout<< “m="<<m<<'\t'<<“n="<<n<<endl;}
      void  Printxy(void) {cout<<"x="<<x<<'\t'<<"y="<<y<<endl;}
} ;
void  main(void)
{       Test    p1,p2;
        p1.Setxy(3, 5); p2.Setxy(10,20,30,40);//参数不同
        p1.Printxy();
        p2.Printxy();    p2.Printxy(2);//参数、类型不同
}
```

输出：x=3 y=5
x=10 y=20
m=30 n=40

说明

```
class Stu
{       char Name[20];
        float Chinese;
        float Math;
        float English;
        float Physical;
public:
        float Average(void);//语文、数学平均成绩
        float Average(int  n);//四门课的平均成绩
        float Sum(void);//语文、数学总分
        float  Sum(int  n);//四门课总分
        void Show(void);
        void SetStudent(char*,float,float);//置姓名、语文、数学初值
        void SetStudent(char *, float,float,float,float);//置姓名、成绩
        void SetName(char *);
        char *GetName(void);
};
```

说明

- 可以有缺省参数的成员函数，若形参不完全缺省，则必须从形参的右边开始缺省。

说明

缺省参数的成员函数

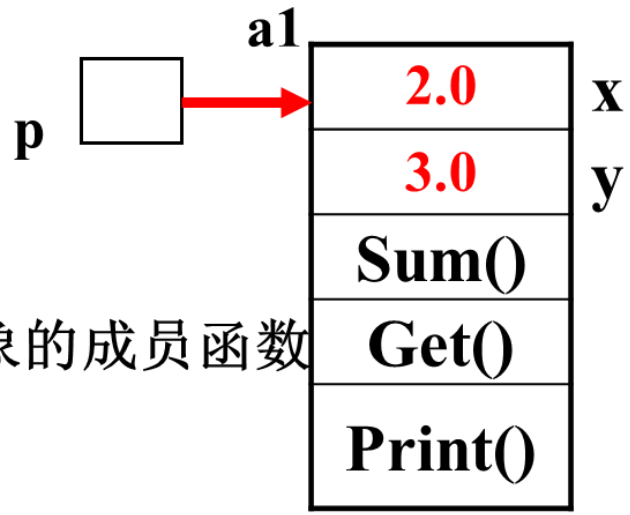
```
class A{
        float  x,y;
public:
        float Sum(void) {   return  x+y;   }
        void Set(float a,float b=10.0) {    x=a;        y=b;}
    void Print(void) {cout<<"x="<<x<<'\t'<<"y="<<y<<endl; }
};
void main(void)
{       A a1,a2;
        a1.Set (2.0,4.0);
        cout<<"a1:  ";          a1.Print ();
        cout<<"a1.sum="<<a1.Sum ()<<endl;
        a2.Set(20.0);
        cout<<"a2:  ";          a2.Print ();
        cout<<"a2.sum="<<a2.Sum ()<<endl;
}
```

不缺省参数，a1.x=2，a1.y=4

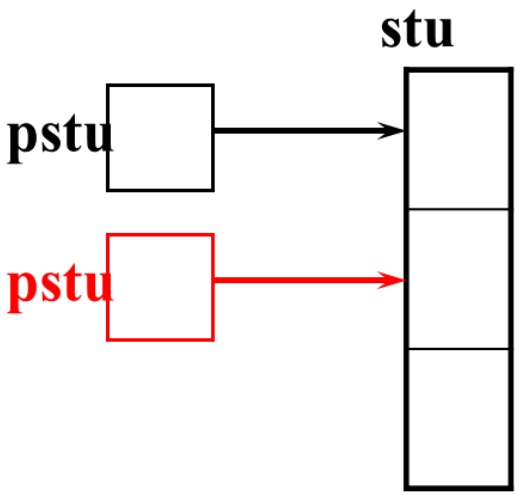
缺省参数，a2.x=20，a2.y=10

定义类的指针及如何用指针来引用对象

```
class A{
        float  x,y;
public:
        float Sum(void) {   return  x+y;   }
        void Set(float  a,float  b) {    x=a;     y=b;}
    void Print(void) {cout<<"x="<<x<<'\t'<<"y="<<y<<endl;   }
};
void main(void)
{       A a1,a2;
        A *p;           //定义类的指针
        p=&a1;          //给指针赋值
        p->Set(2.0,  3.0); //通过指针引用对象的成员函数
        p->Print();
        cout<<p->Sum()<<endl;
        a2.Set(10.0, 20.0);   a2.Print();
}
```



定义类的数组及数组中元素的引用




```
void main(void)
{
        Stu stu[3];     //定义类的数组
        Stu  *pstu;     //定义类的指针
        pstu=stu;       //为指针赋值
        int i;
        stu[0].SetStudent (“A”,90,90);//通过数组元素的引用赋值
        stu[1].SetStudent ("B",80,80);
        stu[2].SetStudent ("C",70,70);
        for(i=0;i<3;i++)
        {
                pstu->Show ();//指针变量指向数组元素
                pstu++;         //指针变量加一，指向下一元素
        }
}
```

返回引用类型的成员函数(可以返回私有数据成员的引用)


```
class A{
        float  x,y;
public:
        float &Getx(void ) {  return  x;  } //返回x的引用
        void Set(float a,float b) {   x=a;   y=b;  }
        void Print(void) {  cout<<x<<'\t'<<y<<endl;  }
};

void main(void)
{       A  a1,a2;
        a1.Set (3,5);
        cout<<"a1:  ";          a1.Print ();
        a1.Getx()=30;     //将a1对象中的x成员赋值
        cout<<"changed  a1: ";
        a1.Print ();
}
```

缺省

缺省参数的成员函数

```
class A{
        float  x,y;
public:
        float Sum(void) {   return  x+y;   }
        void Set(float a,float b=10.0) {    x=a;        y=b;}
    void Print(void) {cout<<"x="<<x<<'\t'<<"y="<<y<<endl; }
};
void main(void)
{       A a1,a2;
        a1.Set (2.0,4.0);
        cout<<"a1:  ";          a1.Print ();
        cout<<"a1.sum="<<a1.Sum ()<<endl;
        a2.Set(20.0);
        cout<<"a2:  ";          a2.Print ();
        cout<<"a2.sum="<<a2.Sum ()<<endl;
}
```

不缺省参数，a1.x=2，a1.y=4

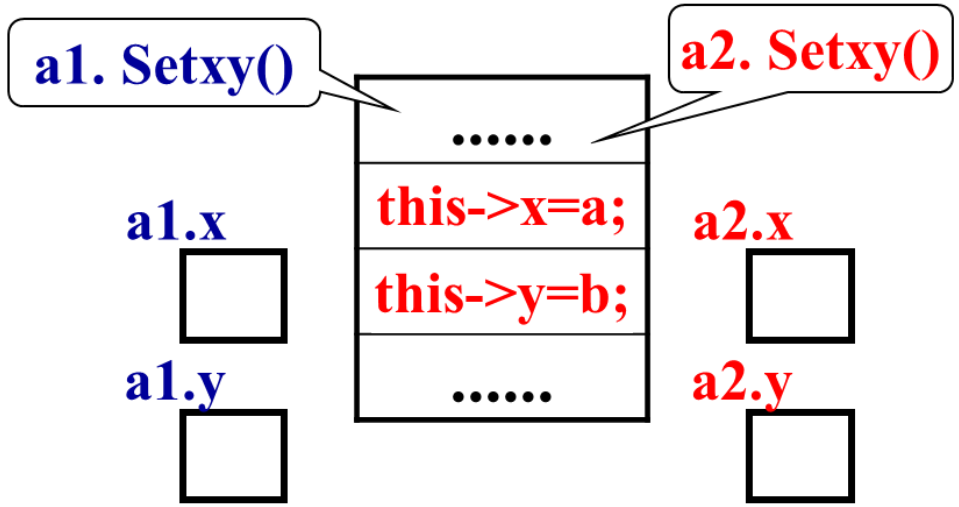
缺省参数，a2.x=20，a2.y=10

this 指针

- 不同对象占据内存中的不同区域，它们所保存的数据各不相同，但对成员数据进行操作的成员函数的程序代码均是一样的。

```
class A{
      int x,y;
   public:
    void Setxy(int a, int b)
   {  x=a;   y=b;}
};
A   a1,  a2;
```

```
a1.Setxy(1,2);
a2.Setxy(3,4);
```



系统自动将对象的指针带到成员函数中

说明

- 当对一个对象调用成员函数时，编译程序先将对象的地址赋给this指针，然后调用成员函数，每次成员函数存取数据成员时，也隐含使用this指针。

this指针具有如下形式的缺省说明：

类名

```
Stu   *const   this;
```

即 this 指针里的地址是一个常量

构造函数与析构函数

定义

- 构造函数和析构函数是在类体中说明的**两种特殊的成员函数**。
- 构造函数是在创建对象时，使用给定的值来**将对象初始化**。
- 析构函数的功能正好相反，是在系统释放对象前，**对对象做一些善后工作**。
- 构造函数**可以带参数、可以重载，同时没有返回值**。
- 构造函数是类的成员函数，系统约定构造函数名必须与类名相同。**构造函数提供了初始化对象的一种简单的方法**。

上节课回顾

- 类的封装与形式， VS 结构体对比。
- 对象，对象即变量。
- 用类封装数据，编程三步走
- 作业域、函数重载 函数参数值缺省

- 对象与数组、指针、函数参数、函数返回值、结构体等结合

说明

```
class A{
        float  x,y;
public:
      A(float a,float b){  x=a;   y=b;}  //构造函数，初始化对象
      float Sum(void) {   return  x+y;   }
      void Set(float  a,float  b) {    x=a;      y=b;}
      Print(void) {  cout<<"x="<<x<<'\t'<<"y="<<y<<endl;}
};
void main(void)
{       A  a1(2.0, 3.0);//定义时调用构造函数初始化
        A  a2(1.0,2.0);
        a2.Set(10.0, 20.0);   //利用成员函数重新为对象赋值
       a1.Print();
        a2.Print();
}
```

说明

- **构造函数的函数名必须与类名相同。构造函数的主要作用是完成初始化对象的数据成员以及其它的初始化工作**
- **在定义构造函数时，不能指定函数返回值的类型，也不能指定为void类型。**
- **一个类可以定义若干个构造函数（不同的初始化值）。当定义多个构造函数时，必须满足函数重载的原则**
- **构造函数可以指定参数的缺省值**

说明

- 若定义的类要说明该类的对象时，构造函数必须是**公有的成员函数**。如果定义的类仅用于派生其它类时，则可将构造函数定义为**保护的成员函数**。
- 由于构造函数属于类的成员函数，它对私有数据成员、保护的数据成员和公有的数据成员**均能**进行初始化。

说明

```
class  A{
        float   x,y;
public:
        A(float a, float b=10)     {  x=a;   y=b;  }
        A()    {    x=0;  y=0;}
        void  Print(void) {cout<<x<<'\t'<<y<<endl;  }
};
void main(void)
{   A  a1, a2(20.0), a3(3.0,  7.0);
     a1.Print();
     a2.Print();
     a3.Print();
}
```

带缺省参数的构造函数

不带参数的构造函数

每一个对象必须要有相应的构造函数

```
0       0
20      10
3       7
```

说明

每一个对象必须要有相应的构造函数

若没有显式定义构造函数，
系统默认缺省的构造函数。

```
class  A{
        float   x,y;
public:
        A()  { }
        void  Print(void) {cout<<x<<'\t'<<y<<endl;  }
};

A  a1,  a2;
```

对象开辟了空间，但没有初始化

隐含的缺省的构造函数

只允许这样定义对象

对局部对象，静态对象，全局对象的初始化

- 对于局部对象，每次定义对象时，都要调用构造函数。
- 对于静态对象，是在首次定义对象时，调用构造函数的，且由于对象一直存在，只调用一次构造函数。
- 对于全局对象，是在main函数执行之前调用构造函数的。

说明

```
class A
{   int x,y;
    public:
      A(int a)  {  x=a; cout<<"1\n";}
      A(int a, int b) {  x=a,y=b;cout<<"2\n";}
};
A   a1(3);
void  f(void)  {   A  b(2,3); }

void main(void)
{  A  a2(4,5);
   f();  f();
}
```

```
1

2

2

2
```

说明

```
class  A{
        float   x,y;
public:
  A(float a, float b){x=a;y=b;cout<<"初始化自动局部对象\n";}
  A(){  x=0;  y=0;  cout<<"初始化静态局部对象\n";}
  A(float  a){  x=a;  y=0;  cout<<"初始化全局对象\n"; }
 void  Print(void){  cout<<x<<'\t'<<y<<endl;  }
};
A  a0(100.0);//定义全局对象
void f(void)
{  cout<<" 进入f()函数\n";A  a2(1,2);
    static  A  a3;  //初始化局部静态对象
 }
void main(void)
{   cout<<"进入main函数\n";
    A  a1(3.0,  7.0);//定义局部自动对象
        f();    f();  }
```

初始化全局对象
进入main函数
初始化自动局部对象
进入f()函数
初始化自动局部对象
初始化局部静态变量
进入f()函数
初始化自动局部对象

缺省的构造函数

- 在定义类时，若没有定义类的构造函数，则编译器**自动**产生一个缺省的构造函数，其格式为：
- className::className() { }
- 缺省的构造函数并不对所产生对象的数据成员赋初值；**即新产生对象的数据成员的值是不确定的。**

说明

```
class A{
        float  x,y;
public:
        A(){    }//缺省的构造函数，编译器自动产生,可以不写
        float Sum(void) {   return  x+y;   }
        void Set(float a,float b) {    x=a;      y=b;}
    void Print(void) {cout<<"x="<<x<<'\t'<<"y="<<y<<endl;   }
};
void main(void)
{       A a1,a2;//产生对象时，自动调用缺省的构造函数，不赋值
        a1.Set (2.0,4.0);
        cout<<"a1:  ";
        a1.Print ();
        cout<<"a1.sum="<<a1.Sum ()<<endl;
        a2.Print();//打印随机值
}
```

说明

- 关于缺省的构造函数
- 1、在定义类时，只要显式定义了一个类的构造函数，则编译器就不产生缺省的构造函数
- 2、所有的对象在定义时，必须调用构造函数
- 不存在没有构造函数的对象！

说明

```
class A{
        float x,y;
public:
   A(float a,float b)    {      x=a; y=b; }
   void Print(void){   cout<<x<<'\t'<<y<<endl;  }
};
void main(void)
{       A  a1;
        A  a2(3.0,30.0);
}
```

显式定义了构造函数，不产生缺省的构造函数

error,定义时，没有构造函数可供调用

说明

- 在类中，若定义了没有参数的构造函数，或各参数均有缺省值的构造函数也称为缺省的构造函数，**缺省的构造函数只能有一个**

- 产生对象时，系统必定要调用构造函数。**所以任一对象的构造函数必须唯一**

说明

```
class A{
        float x,y;
public:
    A(float a=10,float b=20){       x=a; y=b; }
    A(){ }
   void Print(void){    cout<<x<<'\t'<<y<<endl;  }
};
void main(void)
{       A  a1;
        A  a2(3.0,30.0);
}
```

两个函数均为缺省的构造函数

两个构造函数均可供调用，构造函数不唯一

构造函数与new运算符

- 可以使用new运算符来**动态地**建立对象。建立时要**自动调用构造函数**，以便完成初始化对象的数据成员。最后返回这个动态对象的**起始地址。**
- 用new运算符产生的动态对象，在不再使用这种对象时，必须用delete运算符来释放对象所占用的存储空间。
- **用new建立类的对象时，可以使用参数初始化动态空间。**

说明

```
class  A{
        float   x,y;
public:
        A(float a, float b)     {       x=a;y=b;       }
        A()     {       x=0;  y=0;      }
        void  Print(void)       {  cout<<x<<'\t'<<y<<endl;  }
};
void main(void)
{   A   *pa1,*pa2;
    pa1=new  A(3.0, 5.0);//用new动态开辟对象空间，初始化
    pa2=new A;//用new动态开辟空间，调用构造函数初始化
    pa1->Print();
    pa2->Print();                                   3   5
    delete  pa1;  //用delete释放空间
    delete  pa2; //用delete释放空间                 0   0
}
```

析构函数

- 析构函数的作用与构造函数正好相反，是在对象的生命期结束时，释放系统为对象所分配的空间，即要撤消一个对象。

析构函数也是类的成员函数，定义析构函数的格式为：

```
ClassName::~ClassName( )
{
   .......
//        函数体;
 }
```

析构函数的特点

- 1、析构函数是**成员函数**，函数体可写在类体内，也可写在类体外。
- 2、析构函数是一个特殊的成员函数，函数名必须与类名相同，并在其**前面加上字符“~”**，以便和构造函数名相区别。
- 3、析构函数**不能带有任何参数，不能有返回值，不指定函数类型**。
- 4、一个类中，只能定义一个析构函数，**析构函数不允许重载**。
- 5、析构函数是在撤消对象时**由系统自动调用的**。在程序的执行过程中，当遇到某一对象的生存期结束时，系统自动调用析构函数，然后再收回为对象分配的存储空间。

例子

```
class A{
        float x,y;
public:
  A(float a,float b)
        { x=a; y=b; cout<<"调用非缺省的构造函数\n";}
  A()  { x=0; y=0; cout<<"调用缺省的构造函数\n" ;}
  ~A() {     cout<<"调用析构函数\n";}
   void Print(void) {   cout<<x<<'\t'<<y<<endl; }
};
void main(void)
{       A  a1;
        A  a2(3.0,30.0);
        cout<<"退出主函数\n";
}
```

调用缺省的构造函数

调用非缺省的构造函数

退出主函数

调用析构函数

调用析构函数

不同存储类型的对象调用构造函数及析构函数

- 1、对于全局定义的对象（在函数外定义的对象），在程序开始执行时，调用构造函数；到程序结束时，调用析构函数。
- 2、对于局部定义的对象（在函数内定义的对象），当程序执行到定义对象的地方时，调用构造函数；在退出对象的作用域时，调用析构函数。
- 3、用static定义的局部对象，在首次到达对象的定义时调用构造函数；到程序结束时，调用析构函数

说明

- 对于用new运算符动态生成的对象，在产生对象时调用构造函数，只有使用delete运算符来释放对象时，才调用析构函数。若不使用delete来撤消动态生成的对象，程序结束时，对象仍存在，并占用相应的存储空间，即系统不能自动地调用析构函数来撤消动态生成的对象。

说明

```
class  A{
        float   x,y;
public:
  A(float a, float b){x=a;y=b;cout<<"初始化自动局部对象\n";}
  A(){  x=0;  y=0;  cout<<"初始化静态局部对象\n";}
  A(float  a){  x=a;  y=0;  cout<<"初始化全局对象\n"; }
 ~A(){  cout<<“调用析构函数” <<endl; }
};
A  a0(100.0);//定义全局对象
void f(void)
{  cout<<" 进入f()函数\n";
   A  ab(10.0,  20.0);//定义局部自动对象
   static  A  a3;  //初始化局部静态对象
}
void main(void)
{   cout<<"进入main函数\n";
          f();    f();  }
```

```
初始化全局对象
进入main函数
进入f()函数
初始化自动局部对象
初始化静态局部对象
调用析构函数
进入f()函数
初始化自动局部对象
调用析构函数
调用析构函数
调用析构函数
```

动态构造及析构对象数组

- 用new运算符来动态生成对象数组时，自动调用构造函数，而用delete运算符来释放p1所指向的对象数组占用的存储空间时，在指针变量的前面必须加上[ ]， 才能将数组元素所占用的空间全部释放。否则，只释放第0个元素所占用的空间。

```
pa1=new  A[3];
.....
delete  [ ]pa1;
```

说明

```
class  A{
        float   x,y;
public:
   A(float a=0, float b=0){x=a;  y=b;cout<<"调用了构造函数\n";}
   void  Print(void){  cout<<x<<'\t'<<y<<endl;  }
   ~A() {  cout<<"调用了析构函数\n";  }
};
void main(void)
{   cout<<"进入main()函数\n";
    A   *pa1;
   pa1=new  A[3];//开辟数组空间
        cout<<"\n完成开辟数组空间\n\n";
   delete  [ ]pa1;  //必须用[]删除开辟的空间
   cout<<"退出main()函数\n";
}
```

进入main()函数
调用了构造函数
调用了构造函数
调用了构造函数
完成开辟数组空间
调用了析构函数
调用了析构函数
调用了析构函数
退出main()函数

缺省的析构函数

- 若在类的定义中没有显式地定义析构函数时，则编译器自动地产生一个缺省的析构函数，其格式为：
- ClassName::~ClassName() { };

- 任何对象都必须有构造函数和析构函数，但在撤消对象时，要释放对象的数据成员用new运算符分配的动态空间时，必须显式地定义析构函数。

缺省的析构函数

- 若在类的定义中没有显式地定义析构函数时，则编译器自动地产生一个缺省的析构函数，其格式为：
- ClassName::~ClassName() { };

- 任何对象都必须有构造函数和析构函数，但在撤消对象时，要释放对象的数据成员用new运算符分配的动态空间时，必须显式地定义析构函数。

完成拷贝功能的构造函数

- 可以在定义一个对象的时候**用另一个对象为其初始化**，即构造函数的参数**是另一个对象的引用**，这种构造函数常为完成拷贝功能的构造函数。

完成拷贝功能的构造函数的一般格式为：

```
ClassName::ClassName(ClassName  &<变量名>)
{       ……
//   函数体完成对应数据成员的赋值
}
```

例子

```
class  A{
        float   x,y;
public：
       A(float a=0, float b=0){x=a;  y=b;}
       A(A &a)
        { x=a.x;  y=a.y;}
};
 void main(void)
 {  A   a1(1.0,2.0);
     A   a2(a1);
 }
```

形参必须是同类型对象的引用

实参是同类型的对象

说明

```
class  A{
        float   x,y;
public:
  A(float a=0, float b=0){x=a;  y=b;cout<<"调用了构造函数\n";}
  A(A &a)
      { x=a.x;y=a.y;cout<<“调用了完成拷贝功能的构造函数\n”; }
  void  Print(void){  cout<<x<<'\t'<<y<<endl;  }
  ~A() {  cout<<"调用了析构函数\n";  }
};
void main(void)
{  A   a1(1.0,2.0);
   A   a2(a1);
   a1.Print();
   a2.Print();
}
```

用已有的对象中的数据为新创建的对象赋值

```
调用了构造函数
调用了完成拷贝功能的构造函数
1          2
1          2
调用了析构函数
调用了析构函数
```

说明

- 如果没有定义完成拷贝功能的构造函数，编译器自动生成一个隐含的完成拷贝功能的构造函数，依次完成类中对应数据成员的拷贝。

```
A::A(A &a)
{
    x=a.x;
    y=a.y;
}
```

隐含的构造函数

说明

```
class  A{
        float   x,y;
public:
  A(float a=0, float b=0){x=a;  y=b;cout<<"调用了构造函数\n";}
  void  Print(void){  cout<<x<<'\t'<<y<<endl;  }
  ~A() {  cout<<"调用了析构函数\n";  }
};
void main(void)
{  A   a1(1.0,2.0);
   A   a2(a1);
   A   a3=a1;//可以这样赋值
   a1.Print();
   a2.Print();
   a3.Print();
}
```

隐含了拷贝的构造函数

```
调用了构造函数
1          2
1          2
1          2
调用了析构函数
调用了析构函数
调用了析构函数
```

继承与派生

- 继承性是面向对象程序设计中最重要的机制。这种机制提供了**无限重复利用程序资源的一种途径**。通过C++语言中的继承机制，可以**扩充和完善旧的程序设计以适应新的需求**。这样不仅可以节省程序开发的时间和资源，并且为未来程序增添了新的资源。

例子

```
class Student
{   int num;
    char name[30];
    char sex;
 public:
   void display( )                    //对成员函数display的定义
    {cout<<"num: "<<num<<endl;
     cout<<"name: "<<name<<endl;
     cout<<"sex: "<<sex<<endl; }
};
```

```
class Studend1
{    int num;          //此行原来已有
     char name[20];     //此行原来已有
     char sex;         //此行原来已有
     int age;
     char addr［20］;
public:
   void display( ) ;        //此行原来已有
    {cout<<"num: "<<num<<endl;   //此行原来已有
     cout<<"name: "<<name<<endl;//此行原来已有
     cout<<"sex: "<<sex<<endl;     //此行原来已有
     cout<<"age: "<<age<<endl;
     cout<<"address: "<<addr<<endl;}
};
```

说明

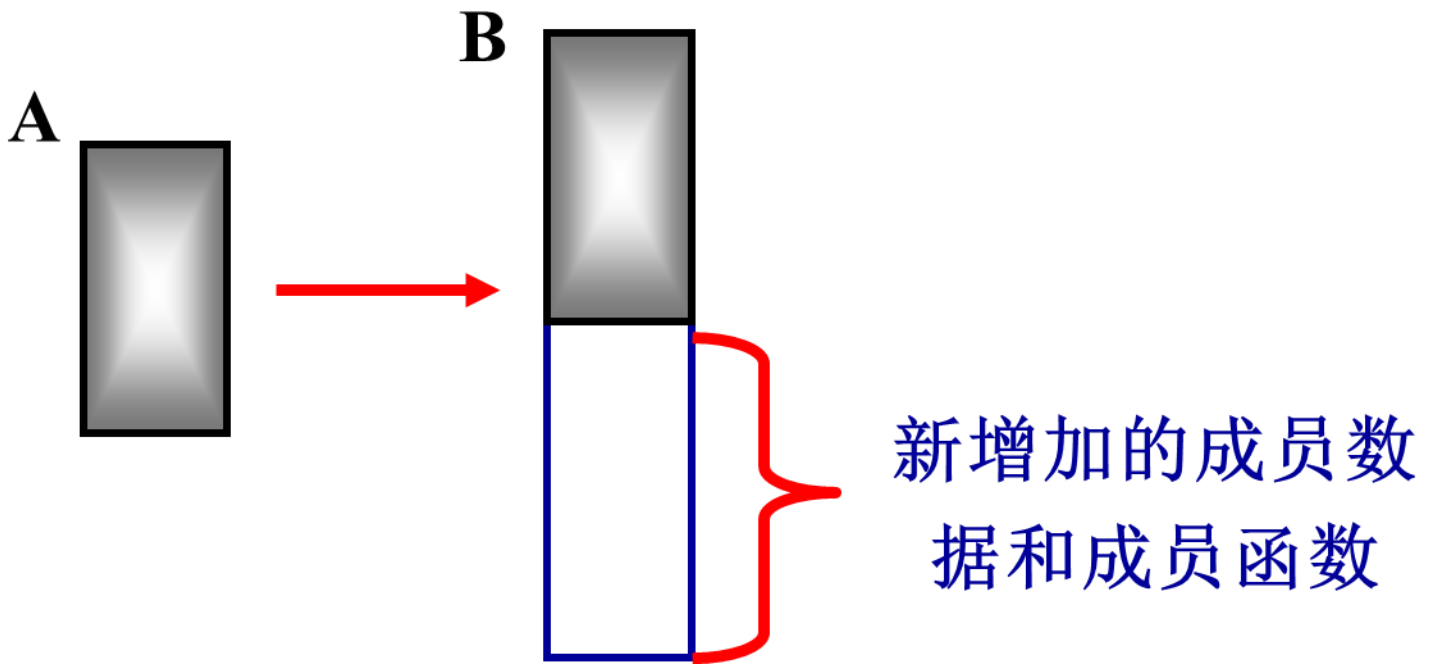
- 利用原来定义的类Student作为基础，再加上新的内容即可，以减少重复的工作量。 C++提供的继承机制就是为了解决这个问题。
- 在C++中所谓“继承”就是在一个已存在的类的基础上建立一个新的类。已存在的类称为“基类(base class)”或“父类(father class)”。新建立的类称为“派生类(derived class)”或“子类(son class)”

说明

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

说明

类A派生类B：类A为基类，类B为派生类。



说明

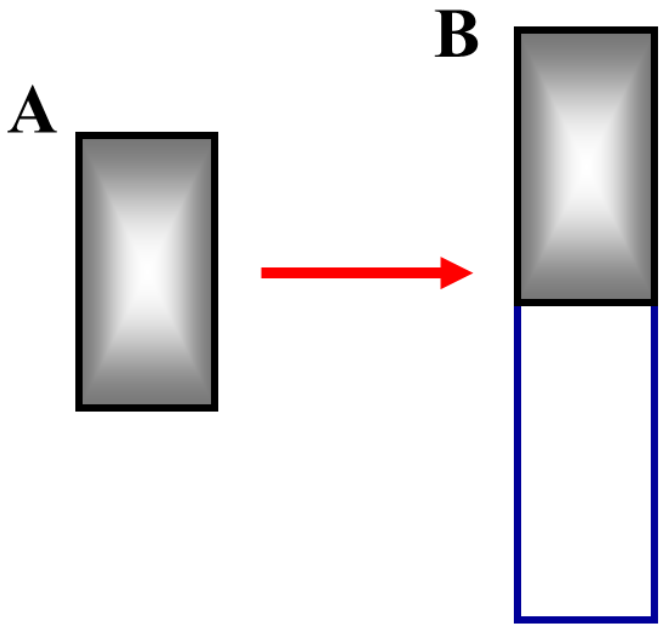
- 在C++语言中，一个派生类可以从**一个基类**派生，也可以从**多个基类**派生。从一个基类派生的继承称为单继承；从多个基类派生的继承称为多继承。
- 通过继承机制，**可以利用已有的数据类型来定义新的数据类型**。所定义的**新的数据类型不仅拥有新定义的成员，而且还同时拥有旧的成员**。我们称已存在的用来派生新类的类为基类，又称为父类。由已存在的类派生出的新类称为派生类，又称为子类。

说明

- 当从已有的类中派生出新的类时，可以对派生类做以下几种变化：
  - 1、 可以**继承**基类的成员数据或成员函数。
  - 2、可以**增加**新的成员变量。
  - 3、可以**增加**新的成员函数。
  - 4、可以**重新定义**已有的成员函数。
  - 5、可以**改变现有**的成员属性。
- 在C++中有二种继承：单一继承和多重继承。当一个派生类仅由一个基类派生时，称为单一继承；而当一个派生类由二个或更多个基类所派生时，称为多重继承。

说明

类A派生类B：类A为基类，类B为派生类。




```
class B：  public    A{...};
class B：  protected   A{...};
class B：  private   A{...};
class B：     A {...};
```

A为私有派生

但派生并不是简单的扩充，有可能改变基类的性质。

有三种派生方式：公有派生、保护派生、私有派生。

默认的是私有派生。

从一个基类派生一个类的一般格式

```
class  ClassName:<Access>BaseClassName
{
    private:
     ......;      //私有成员说明
    public:
     ......;      //公有成员说明
    protected:
     ......;      //保护成员说明
}
```

派生类名　继承方式　基类名

派生类中新增加的成员

public: 表示公有基类

private:表示私有基类(默认)

protected:表示保护基类

公有派生

公有派生，派生类中保持基类的成员特性

class ClassName: public BaseClassName

公有派生时，基类中所有成员在派生类中保持各个成员的访问权限。

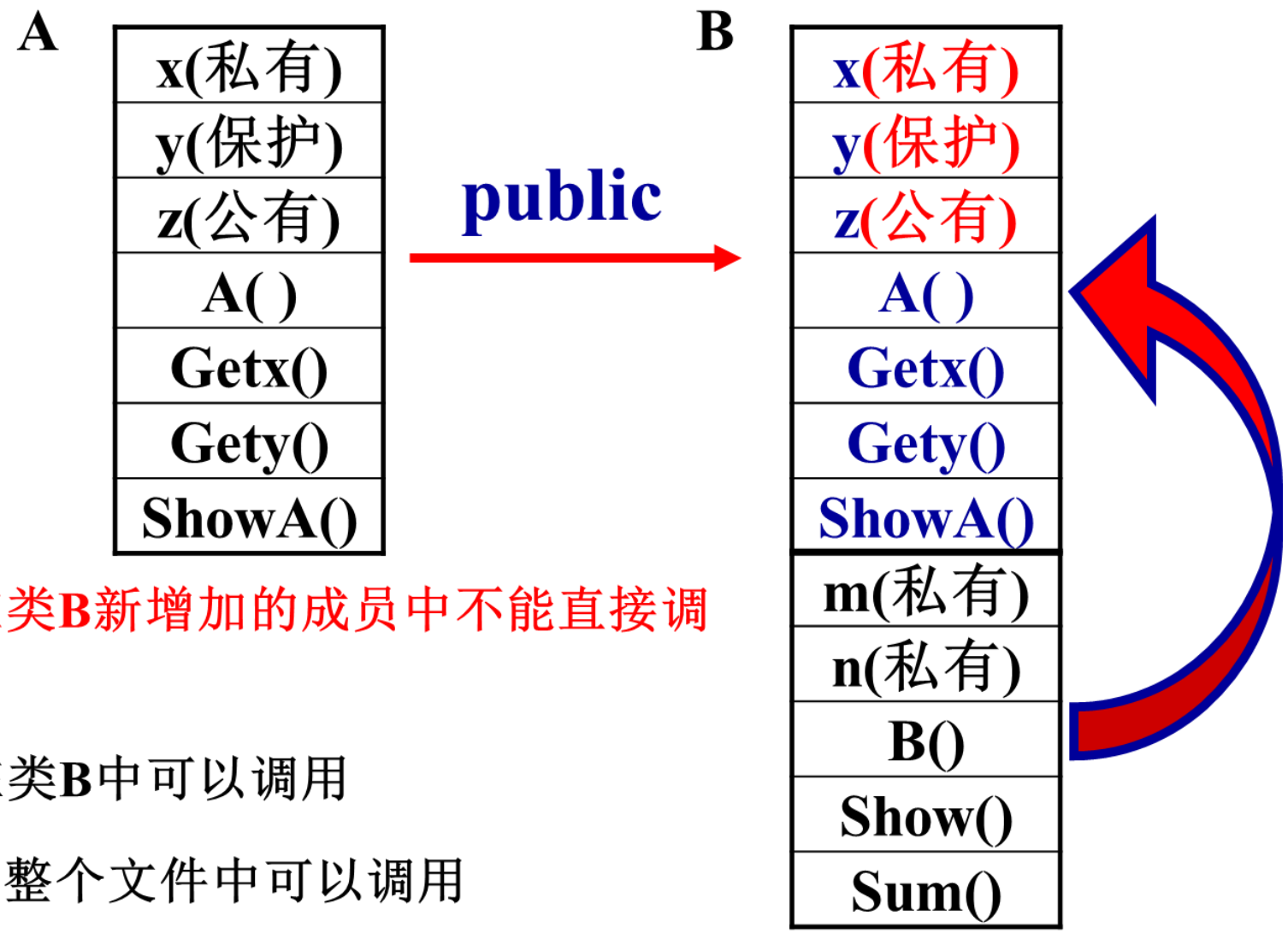
基类成员属性	派生类中	派生类外
公有	可以引用	可以引用
保护	可以引用	不可引用
私有	不可引用	不可引用

基类：**public:** 在派生类和类外可以使用

protected: 在派生类中使用

private: 不能在派生类中使用

说明



x在类B新增加的成员中不能直接调用

y在类B中可以调用

z在整个文件中可以调用

对类B的对象初始化即是对x,y,z,m,n等全部成员的初始化

说明

```
class A{   int x;
protected:  int y;
public:     int z;
  A(int a,int b,int c){x=a;y=b;z=c;}//基类初始化
 int Getx(){return x;}     //返回x
 int Gety(){return y;}     //返回y
 void ShowA(){cout<< "x="<<x<<'\t'<<"y="<<y<<'\t'<<"z="<<z<<'\n';}
};
class B:public A{
          int m,n;
public:  B(int a,int b,int c,int d,int e):A(a,b,c){m=d;n=e;}
   void Show(){cout<<“m="<<m<<'\t'<<“n="<<n<<'\n';
   cout<<"x="<<Getx()<<'\t'<<"y="<<y<<'\t'<<"z="<<z<<'\n'; }
  int Sum(){return ( Getx()+y+z+m+n);}
 };
void  main(void)
{   B b1(1,2,3,4,5);
    b1.ShowA();        b1.Show();
     cout<< "Sum="<<b1.Sum()<<'\n';cout<<"x="<<b1.Getx()<<'\t';
    cout << "y=" <<b1.Gety()<<'\t';          cout << "z="<<b1.z<<'\n';}
```

因为y是基类保护，所以在派生类中可以直接引用。而在类外不可直接引用。

公有派生

对基类初始化

因为z是基类公有，所以在派生类中和类外均可直接引用。

因为x是基类私有，所以在派生类和类外中不能直接引用

私有派生

私有派生，派生类中基类公有和保护成员成为私有

class ClassName: private BaseClassName

私有派生时，基类中公有成员和保护成员在派生类中均变为私有的，在派生类中仍可直接使用这些成员，基类中的私有成员，在派生类中不可直接使用。

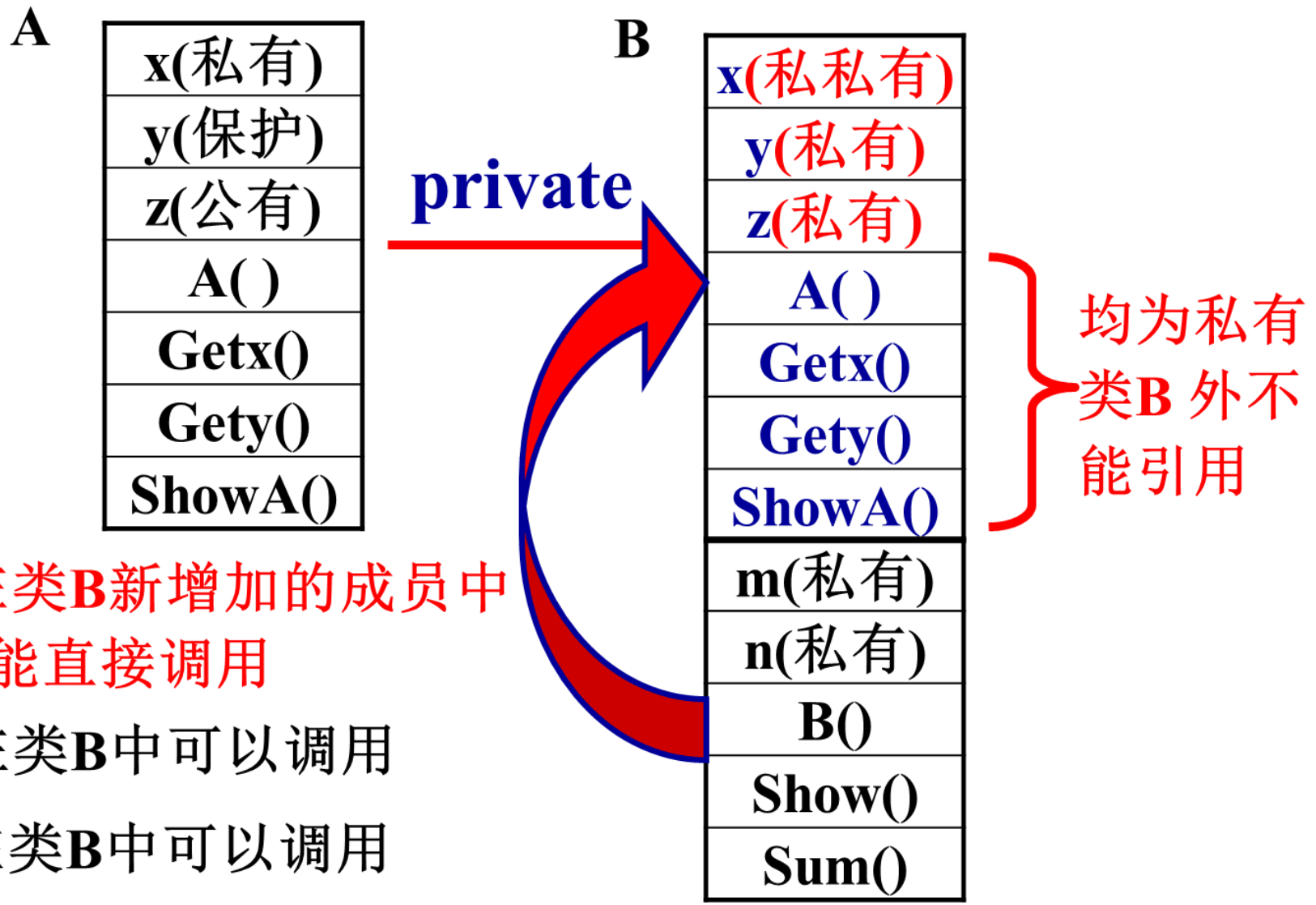
基类成员属性	派生类	派生类外
公有	可以引用	不可引用
保护	可以引用	不可引用
私有	不可引用	不可引用

基类：**public:** (变为私有)在派生类中使用，类外不可使用

protected: (变为私有）在派生类中使用，类外不可使用

private: 不能在派生类中和类外使用

说明



对类B的对象初始化即是对x,y,z,m,n等全部成员的初始化

说明

```
class A{   int x;
protected:  int y;
public:     int z;
  A(int a,int b,int c){x=a;y=b;z=c;}//基类初始化
 int  Getx(){return x;}     //返回x
 int  Gety(){return y;}     //返回y
 void ShowA(){cout<< "x="<<x<<'\t'<<"y="<<y<<'\t'<<"z="<<z<<'\n';}
};
class B:private A{
          int m,n;
public:  B(int a,int b,int c,int d,int e):A(a,b,c){m=d;n=e;}
   void Show(){cout<<“m="<<m<<'\t'<<“n="<<n<<'\n';
   cout<<"x="<<Getx()<<'\t'<<"y="<<y<<'\t'<<"z="<<z<<'\n'; }
  int Sum(){return ( Getx()+y+z+m+n);}
 };
void  main(void)
{   B b1(1,2,3,4,5);
    b1.ShowA();       b1.Show();
     cout<< "Sum="<<b1.Sum()<<'\n';cout<<"x="<<b1.Getx()<<'\t';
    cout << "y=" <<b1.Gety()<<'\t';        cout << "z="<<b1.z<<'\n';}
```

y是基类保护，所以在派生类中可以直接引用。而在类外不可直接引用。

私有派生

对基类初始化

z是基类公有，私有派生变为私有，所以在派生类中可直接引用，而在类外不可。

因为x是基类私有，所以在派生类和类外中不能直接引用

A a1(1,2,3); a1.ShowA();

这些函数都是基类公有，在类外不可使用。

保护派生

保护派生，派生类中基类公有和保护成员降级使用

class ClassName: protected BaseClassName

保护派生时，基类中公有成员和保护成员在派生类中均变为保护的和私有的，在派生类中仍可直接使用这些成员，基类中的私有成员，在派生类中不可直接使用。

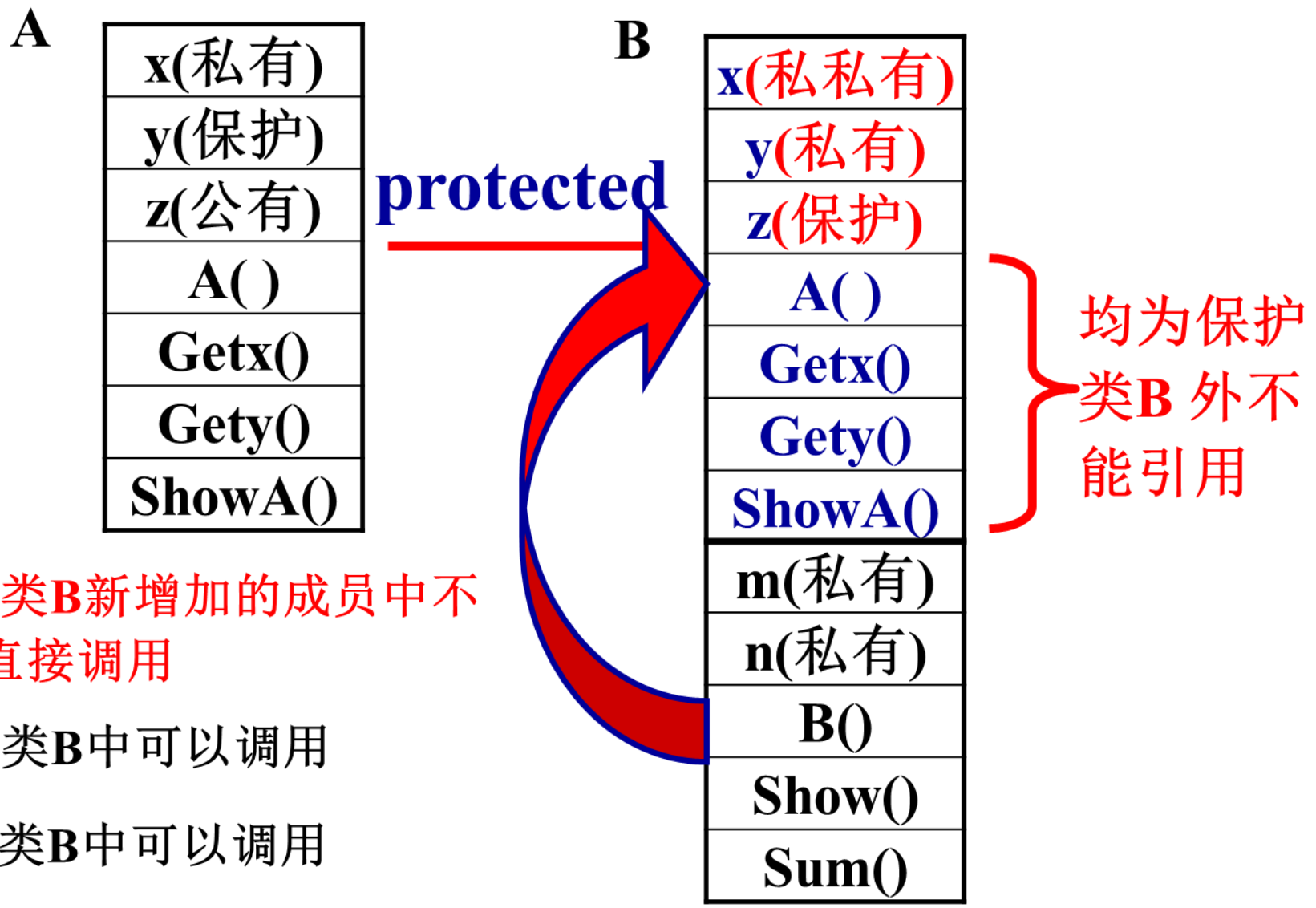
基类成员属性	派生类	派生类外
公有	可以引用	不可引用
保护	可以引用	不可引用
私有	不可引用	不可引用

基类：**public:** (变为保护)在派生类中使用，类外不可使用

protected: (变为私有）在派生类中使用，类外不可使用

private: 不能在派生类中和类外使用

说明



对类B的对象初始化即是对x,y,z,m,n等全部成员的初始化

说明

- protected 成员是一种具有血缘关系内外有别的成员。它对派生类的对象而言，是公开成员，可以访问，**对血缘外部而言，与私有成员一样被隐蔽。**

抽象类与保护的成员函数

- 当定义了一个类，这个类只能用作基类来派生出新的类，而不能用这种类来定义对象时，称这种类为抽象类。当对某些特殊的对象要进行很好地封装时，需要定义抽象类。

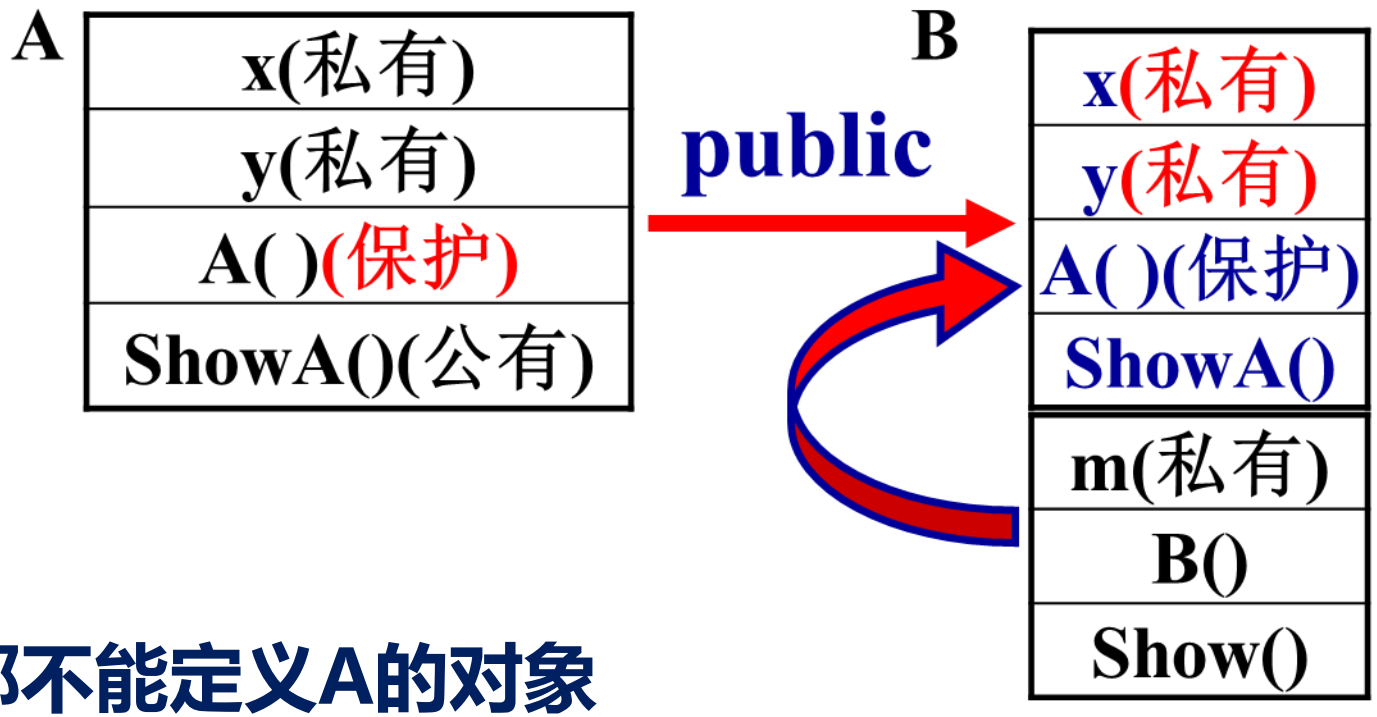
- 将类的构造函数或析构函数的访问权限定义为保护的时，这种类为抽象类

说明

- 当把类中的构造函数或析构函数说明为私有的时，所定义的类通常是没有任何实用意义的，一般情况下，不能用它来产生对象，也不能用它来产生派生类。

说明

- 在类B中不能定义A的对象



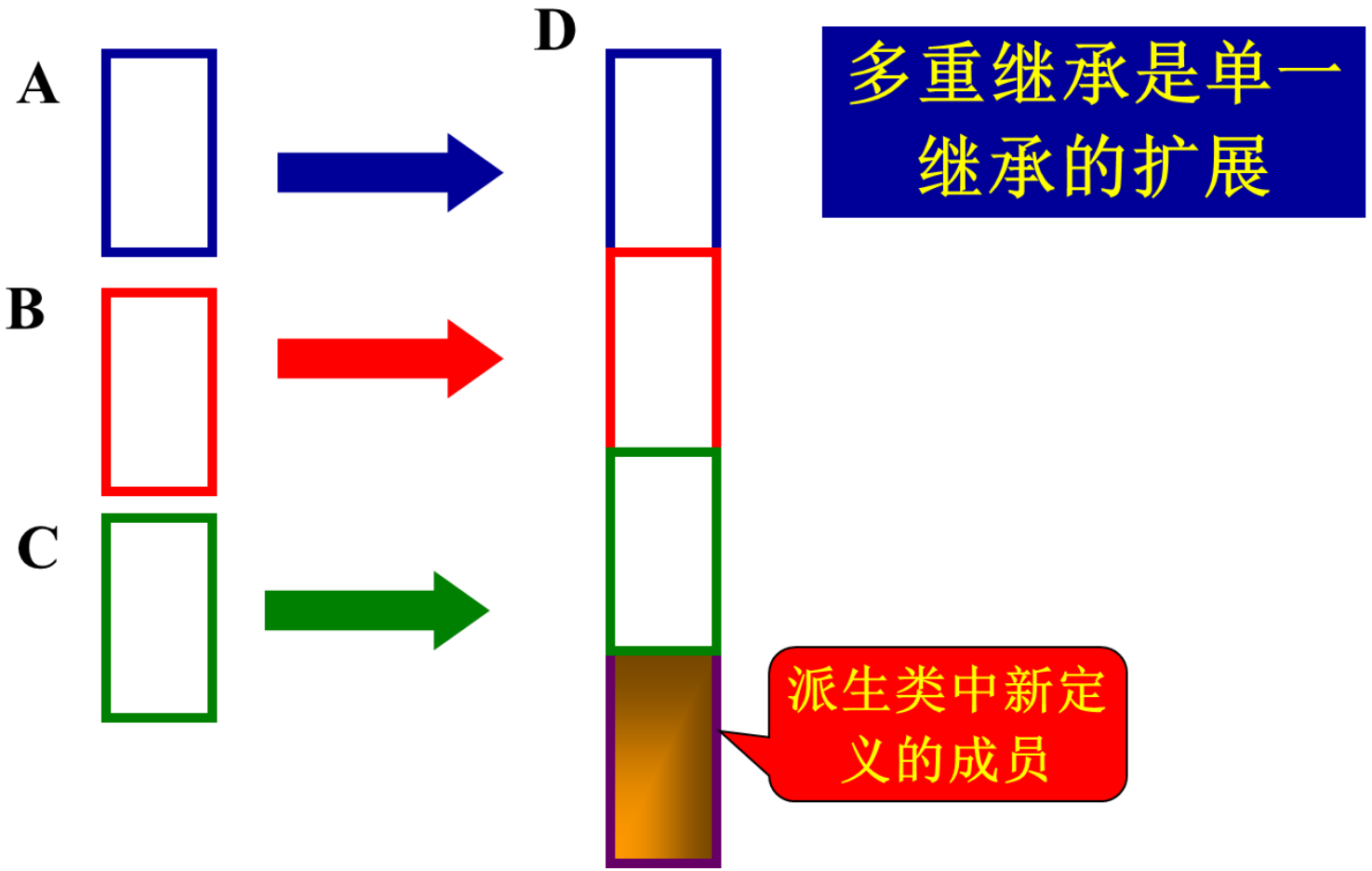
- 在任何时候都不能定义A的对象
- 但可以在初始化类B的对象时初始化原类A中的成员，因为A()在类B中是可以被调用的。

说明

```
class  A {   int  x, y;
protected:   A(int a,int b){x=a;y=b;}//基类初始化
public:
void ShowA(){cout<< "x="<<x<<'\t'<<"y="<<y<<'\n';}
};
class B: public  A{
          int m;
A  a1;  //在派生类中也不可以定义A的对象，实际上还是类外调用
public:                a1(4,5)
     B(int a,int b,int c):A(a,b)//可以在派生类中调用A的构造函数
      {m=c;}
   void Show(){     cout<<“m="<<m<<'\n';    ShowA();  }
 };
void  main(void)
{   B b1(1,2,3); //可以定义派生类对象
    b1.Show();
    A  aa;       //不可定义A的对象          }
```

多重继承

- 可以用多个基类来派生一个类。



说明

格式为：　　继承方式

```
class  类名:<Access>类名1,..., <Access>类名n
{
    private:    ..... ;    //私有成员说明;
    public:     ..... ;    //公有成员说明;
    protected:  ..... ;    //保护的成员说明;
};

class  D: public  A, protected B, private C
{   ....//派生类中新增加成员
};
```

说明

```
class A{ int x1,y1;
public:  A(int a,int b)      { x1=a; y1=b;     }
void ShowA(void){ cout<<"A.x="<<x1<<'\t'<<"A.y="<<y1<<endl; }
};
class  B{int x2,y2;
public:  B(int a,int b)      {x2=a; y2=b;      }
  void ShowB(void){ cout<<"B.x="<<x2<<'\t'<<"B.y="<<y2<<endl; }
};
class  C:public A,private B{
            int x,y;
public:  C(int a,int b,int c,int d,int e,int f):A(a,b),B(c,d)      {x=e; y=f;          }
    void ShowC(void){cout<<"C.x="<<x<<'\t'<<"C.y="<<y<<endl;
                     ShowA();ShowB();        }
};
void main(void)
{           C c(1,2,3,4,5,6);
            c.ShowC();
            c.ShowA ();
            c.ShowB ();
}
```

公有派生　私有派生

仍为公有　成为私有

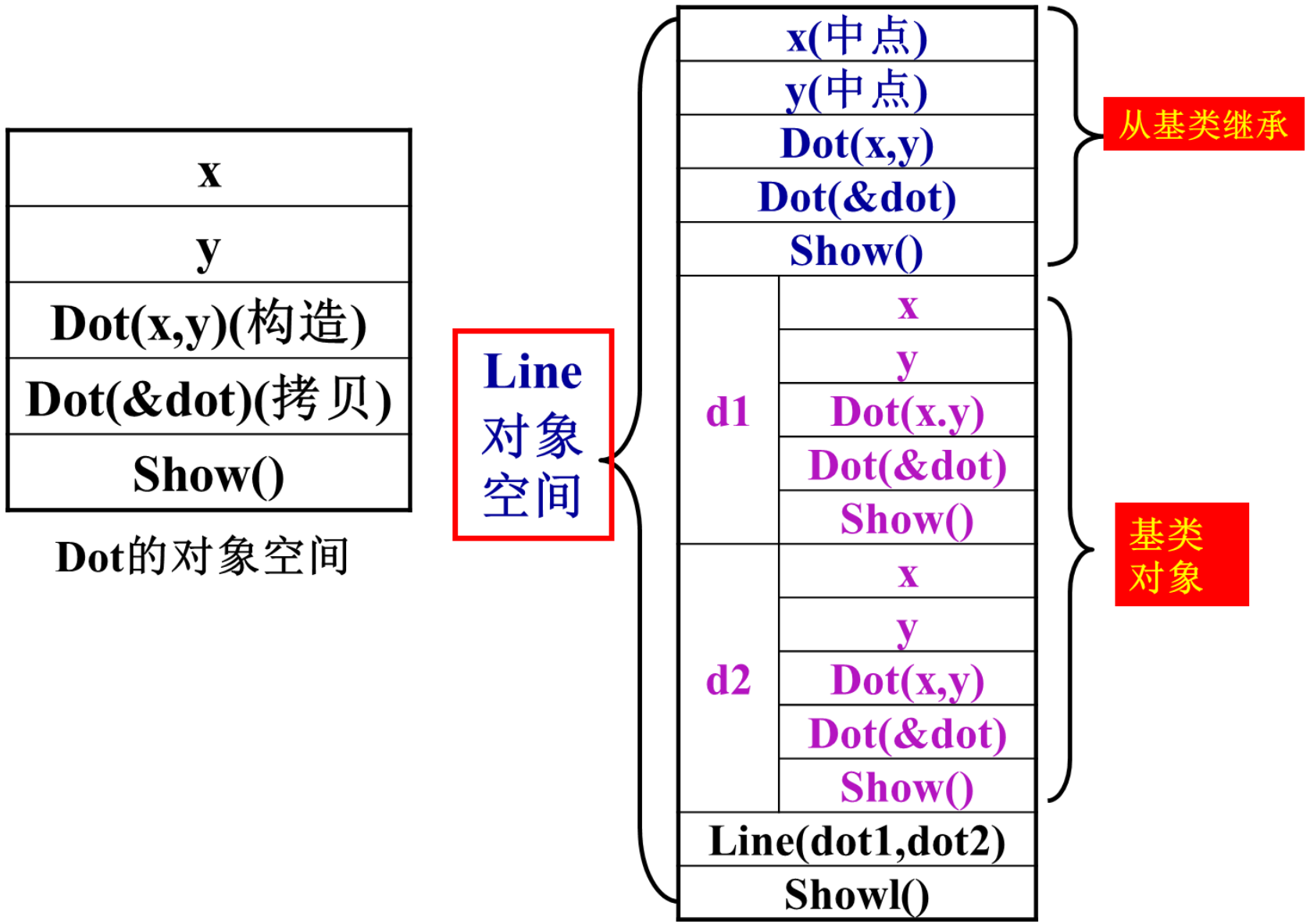
非法，私有类外不可调用

```
B  b1(10,20);
b1.ShowB();
```

例子---继承的实际应用体会

- 在平面上作两个点，连一直线，求直线的长度和直线中点的坐标。
- 基类为Dot，有两个公有数据成员，即平面上的坐标（x,y），同时有构造函数及打印函数。
- 派生类为Line，有两个基类Dot对象，分别存放两点的坐标，同时，从基类继承了一个Dot数据，存放直线中点的坐标。

说明


x
y
Dot(x,y)(构造)
Dot(&dot)(拷贝)
Show()
Dot的对象空间
Line
对象
空间
x(中点)
y(中点)
Dot(x,y)
Dot(&dot)
Show()
从基类继承
d1
x
y
Dot(x.y)
Dot(&dot)
Show()
d2
x
y
Dot(x,y)
Dot(&dot)
Show()
基类
对象
Line(dot1,dot2)
Showl()

说明

```
class Dot{
public:  float x,y;
         Dot(float a=0, float b=0){ x=a; y=b;}
         void Show(void){cout<<"x="<<x<<'\t'<<"y="<<y<<endl;}
};
class Line:public Dot{
         Dot d1,d2;
public:  Line(Dot dot1,Dot dot2):d1(dot1),d2(dot2)
         { x=(d1.x+d2.x)/2;    y=(d1.x+d2.y)/2;}
 void Showl(void){ cout<<"Dot1:  "; d1.Show(); cout<<"Dot2:  "; d2.Show();
 cout<<"Length="<<sqrt((d1.x-d2.x)*(d1.x-d2.x)+(d1.y-d2.y)*(d1.y-d2.y))<<endl;
 cout<<"Center:  "<<"x="<<x<<'\t'<<"y="<<y<<endl;      }
};
void main(void)
{   float a,b;
    cout<<"Input Dot1: \n";  cin>>a>>b;
    Dot dot1(a,b);//调用Dot的构造函数
   cout<<"Input Dot2: \n"; cin>>a>>b;
   Dot dot2(a,b);            Line line(dot1,dot2);   line.Showl();}
```

用坐标初始化Dot对象

打印坐标

在Line中新说明的成员

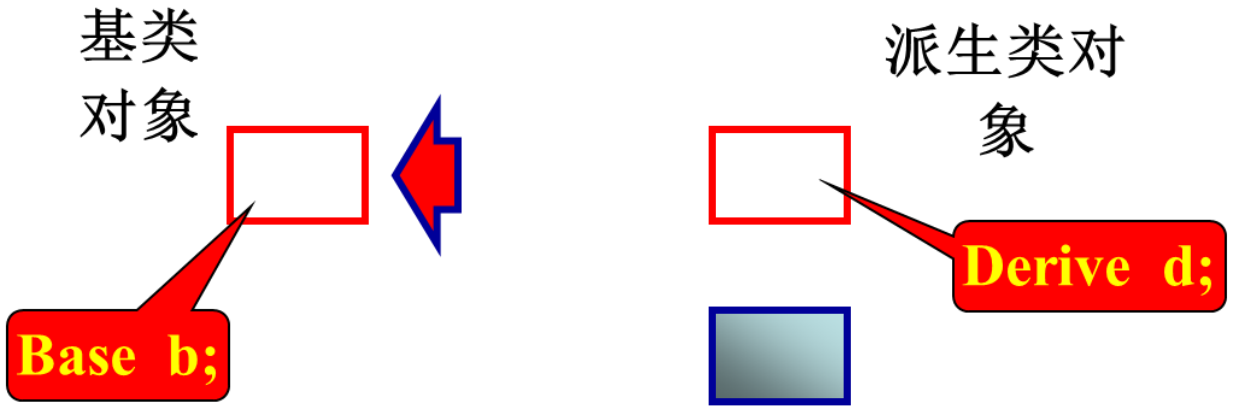
对成员初始化

x，y是继承基类的成员

说明

赋值兼容规则

相互之间能否赋值？

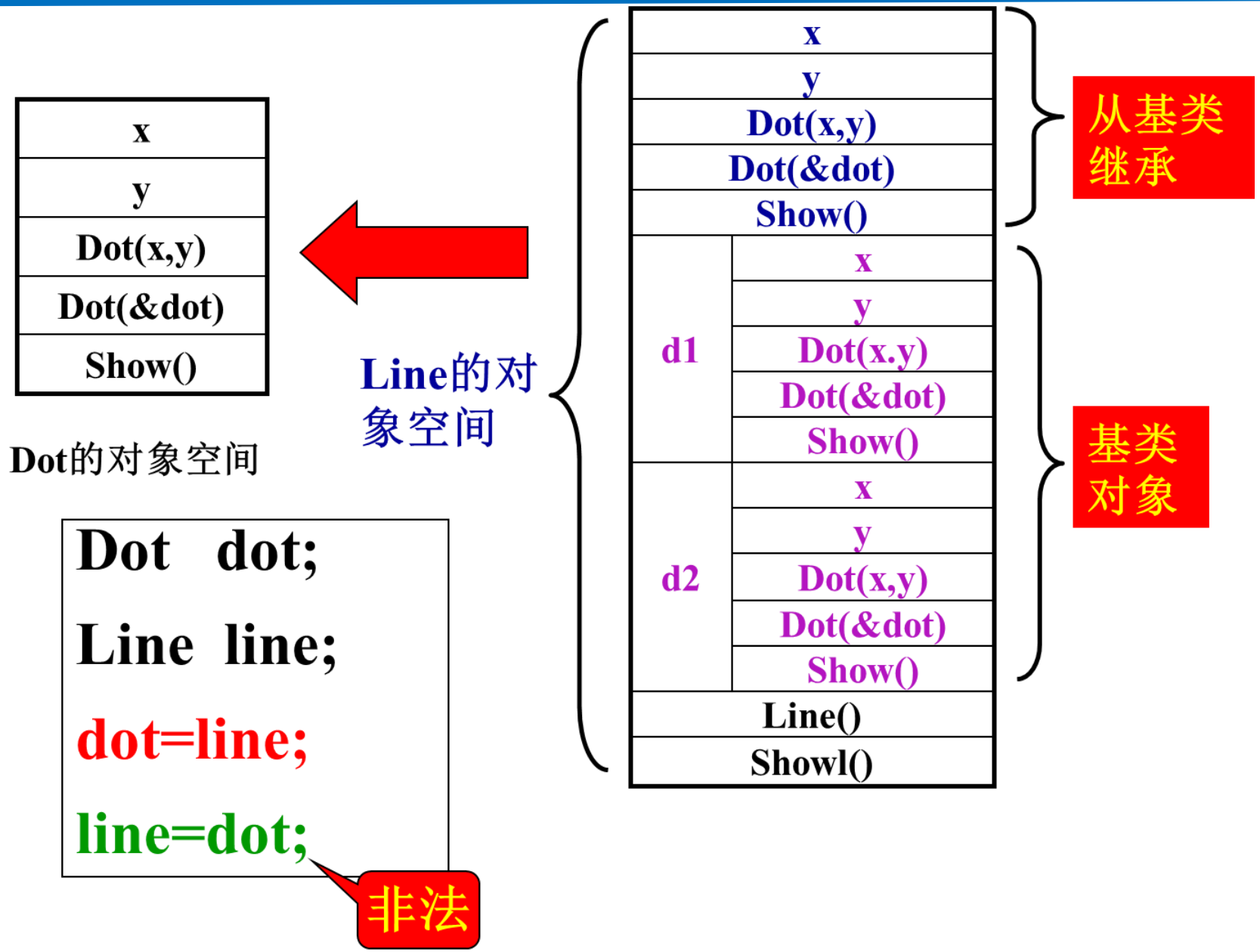


可以将派生类对象的值赋给基类对象。反之不行

```
b = d;
```

只是将从基类继承来的成员赋值。

说明


x
y
Dot(x,y)
Dot(&dot)
Show()
Dot的对象空间
Line的对象空间
x
y
Dot(x,y)
Dot(&dot)
Show()
d1
x
y
Dot(x.y)
Dot(&dot)
Show()
d2
x
y
Dot(x,y)
Dot(&dot)
Show()
Line()
Showl()
从基类继承
基类对象
Dot dot;
Line line;
dot=line;
line=dot;
非法

运算符重载与模版

重载

- **回顾**：所谓函数的重载是指完成不同功能的函数可以具有相同的函数名。 C++的编译器是根据函数的实参来确定应该调用哪一个函数的。

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

int  fun (int a)
{   return a*a;  }
```

```
void  main(void)
{  cout<<fun(3,5)<<endl;
   cout<<fun(5)<<endl;
}
```

例子

```
int   sum,a=3,b=2;
sum=a+b;
```

(int)=(int) + (int)

```
float   add, x=3.2, y=2.5;
add=x+y;
```

(float)=(float) + (float)

系统自动识别数据类型

```
char   str[4], c1[2]="a",   c2[2]="b";
str=c1+c2;
```

(char *)=(char *) + (char *)

编译系统中的运算符“＋”本身不能做这种运算，若使上式可以运算，必须重新定义“＋”运算符，这种重新定义的过程成为运算符的重载。

运算符重载

- 运算符重载就是赋予已有的运算符多重含义。C++通过重新定义运算符，使它能够用于特定类的对象执行特定的功能

```
class A
{  float x,y;
public:
    A(float a=0, float b=0){  x=a; y=b; }
}
void main(void)
{  A   a(2,3),  b(3,4),  c;
     c=a+b;
}
```

两对象不能使用＋，必须重新定义＋

运算符重载

- 运算符的重载从另一个方面体现了OOP技术的多态性，且同一运算符根据不同的运算对象可以完成不同的操作。
- 为了重载运算符，必须定义一个函数，并告诉编译器，遇到这个重载运算符就调用该函数，由这个函数来完成该运算符应该完成的操作。这种函数称为运算符重载函数，它通常是类的成员函数或者是友元函数。
- 运算符的操作数通常也应该是类的对象。

operator关键字

■ 重载为类的成员函数

格式如下：

<类名> operator<运算符>(<参数表>)

{ 函数体 }

A operator + (A &);　　//重载了类A的“+”运算符

其中：operator是定义运算符重载函数的关键字，它与其后的运算符一起构成函数名。

例子

没有重载运算符的例子

```
class A
{       int i;
public:A(int a=0)      { i=a;  }
       void Show(void){     cout<<"i="<<i<<endl;        }
       void AddA(A &a,     A &b) //利用函数进行类之间的运算
       {       i=a.i+b.i;      }
};
void main(void)
{       A a1(10),a2(20),a3;
       a1.Show ();
       a2.Show ();
//     a3=a1+a2;                    //不可直接运算
       a3.AddA(a1,a2);              //调用专门的功能函数
       a3.Show ();
}
```

利用函数完成了加法运算

用和作对象调用函数

例子

```
class A
{       int i;
public:A(int a=0){ i=a;       }
       void Show(void){     cout<<"i="<<i<<endl;       }
       void AddA(A &a,   A &b)    //利用函数进行类之间的运算
       {       i=a.i+b.i;       }
       A operator +(A &a)  //重载运算符+
       {       A   t;   t.i=i+a.i;       return t;       }
};
void main(void)
{       A a1(10),a2(20),a3;
       a1.Show ();
       a2.Show ();
       a3=a1+a2;     //重新解释了加法，可以直接进行类的运算
       a3.AddA(a1,a2);                  //调用专门的功能函数
       a3.Show ();
}
```

相当于a3=a1.operator+(a2)

重载运算符与一般函数的比较：

- 相同：1）均为类的成员函数；2）实现同一功能

返回值　函数名　形参列表

```
void  AddA(A &a,   A &b)
{        i=a.i+b.i;       }
```

函数调用：

```
a3.AddA(a1,a2);
```

由对象a3调用

返回值　函数名

```
A operator +(A &a)
{  A   t;
   t.i=i+a.i;
   return t;
}
```

形参

函数调用：

```
a3=a1+a2;
a3=a1.operator+(a2);
```

由对象a1调用

例子

返回值

函数名

```
A operator +(A &a)
{  A  t;
   t.i=i+a.i;
   return t;
}
```

形参

总结：

重新定义运算符，由左操作符调用右操作符。最后将函数返回值赋给运算结果的对象。

函数调用：

```
a3=a1+a2;
a3=a1.operator+(a2);
```

由对象a1调用

例子

```
class A
{       int i;
public:A(int a=0){ i=a;       }
       void Show(void){     cout<<"i="<<i<<endl;       }
       void AddA(A &a,  A &b)    //利用函数进行类之间的运算
       {       i=a.i+b.i;       }
       A operator +(A &a)  //重载运算符+
       {       A   t;   t.i=i+a.i;       return t;       }
};
void main(void)
{       A a1(10),a2(20),a3;
       a1.Show ();
       a2.Show ();
       a3=a1+a2;     //重新解释了加法，可以直接进行类的运算
       a3.AddA(a1,a2);                  //调用专门的功能函数
       a3.Show ();
}
```

相当于a3=a1.operator+(a2)

例子

- 当用成员函数实现运算符的重载时，运算符重载函数的参数只能有二种情况：**没有参数或带有一个参数**。对于只有一个操作数的运算符(如++)，在重载这种运算符时，**通常不能有参数**；而对于有二个操作数的运算符，**只能带有一个参数。这参数可以是对象，对象的引用，或其它类型的参数。在C++中不允许重载有三个操作数的运算符**

注意

- 在C++中,大多允许运算符重载
- 在C++中不允许重载的运算符
  - 1.作用域操作符: ::
  - 2.条件操作符: ?:
  - 3.点操作符: .
  - 4.指向成员操作的指针操作符: ->*,.*
  - 5.预处理符号: #
- 只能对C++中已定义了的运算符进行重载，而且，当重载一个运算符时，该运算符的优先级和结合律是不能改变的。

例子

```
class room{
        float Length;
        float Wide;
public:  room(float a=0.0,float b=0.0){  Length=a;   Wide=b; }
  void Show(void){cout<<"Length="<<Length<<'\t'<<"Wide="<<Wide<<endl;}
  void ShowArea(void){  cout<<"Area="<<Length*Wide<<endl;   }
  room operator+(room &);//重载运算符+，函数原型
};
room room::operator + (room &rr)  //重载运算符，函数定义
{  room rr;
   rr.Length =Length+r.Length;
   rr.Wide =Wide+r.Wide ;
   return rr;
}
void main(void)
{  room r1(3,2),r2(1,4), r3,r4;
   r1.Show ();    r2.Show ();
   r3=r1+r2;                r3.Show ();
   r4=r1+r2+r3;             r4.Show ();}
```

r4=r1+r2+r3;

(r1+r2); (r1+r2)+r3;

r4=r1+(r2+r3);

(r2+r3); r1+(r2+r3);

运算符的优先级和结合律是不能改变的

例子

```
class A
{       int i;
public:A(int a=0){ i=a;       }
       void Show(void){     cout<<"i="<<i<<endl;       }
       A operator +(A &a)  //重载运算符+
       {       A   t;   t.i=i+a.i;       return t;       }
       void operator+=(A &a)
       {       i=i+a.i;       }
};
void main(void)
{       A a1(10),a2(20),a3;
       a1.Show ();
       a2.Show ();
       a3=a1+a2;
       a1+=a2;
       a3.Show ();
}
```

由左操作符调用右操作符，没有返回值，故函数类型为void。

相当于a3=a1.operator+(a2)

相当于a1.operator+=(a2)

单目运算符的重载

- 只具有一个操作数的运算符为单目运算符，最常用的为＋＋及－－。

```
A   a;
++a;
a++;
```

```
A   a, b;
b=++a;
b=a++;
```

- 可以看出，虽然运算后对象a的值一致，但先自加或后自加的重载运算符函数的返回值不一致，必须在重载时予以区分。

例子

- ++为前置运算时，它的运算符重载函数的一般格式为：
  - <type> operator ++( )
  - { ……; }
- ++为后置运算时，它的运算符重载函数的一般格式为：
  - <type> operator ++(int)
  - { ……; }

```
A  a, b;
b=++a;
b=a++;
```

```
A operator ++(  ){ .... }
A operator ++(int){ .... }
```

例子（代码）

```
class A
{    float x, y;
public:
     A(float a=0, float b=0){  x=a;  y=b;  }
    A operator ++( ){A  t;  t.x=++ x; t.y=++y;  return t;}
   A operator ++(int) {  A  t;  t.x=x++;  t.y=y++;  return t;}
};
void main(void)
{  A   a(2,3), b;
    b=++a;
    b=a++;
}
```

例子（存储分析）

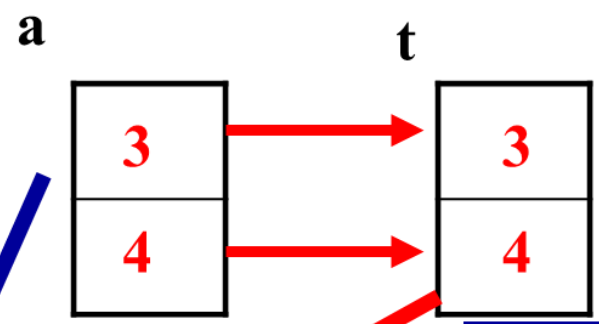
返回值　函数名

```
A operator ++( )
{ A  t;
   t.x=++ x;
   t.y=++y;
   return t;
}

  b=++a;

  b=a.operator++( );
```

```
A operator ++( )
{  ++ x;
   ++y;
   return *this;
}
```



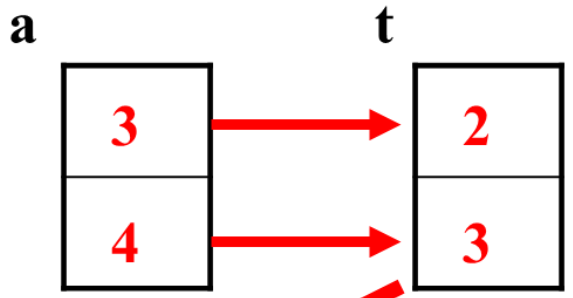
将对象本身作为函数值返回赋给b

t作为函数值返回赋给b

例子（存储分析）

返回值　函数名

```
A operator ++(int)
{  A  t;
   t.x=x++;
   t.y=y++;
   return t;
}
```




```
   b=a++;
```

b=a.operator++(3);

t作为函数值返回赋给b

例子 （代码）

```
class incount
{       int c1,c2;
public:
   incount(int a=0,int b=0){   c1=a; c2=b;  }
   void Show(void){cout<<"c1="<<c1<<'\t'<<"c2="<<c2<<endl;}
   incount operator++(){  c1++; c2++;       return *this;      }//前置++
   incount operator ++(int)//后置++
   {  incount c;    c.c1=c1;          c.c2=c2;          c1++;    c2++;    return c;
   }
};
void main(void)
{       incount ic1(10,20),ic2(100,200), ic3 , ic4;
        ic1.Show ();
        ic2.Show ();
        ic3=++ic1;//ic3=ic1.operator++()
        ic4=ic2++;//ic4=ic4.operator++(3)
        ic3.Show ();
        ic4.Show ();
}
```

因为结果要赋值，所以函数要有返回值

```
ic1.c1=11    ic3.c2=21
ic3.c1=11    ic3.c2=21
ic2.c1=20    ic2.c2=200
ic4.c1=100   ic4.c2=200
```

例子 （代码----无返回值呢）

```
class incount
{           int c1,c2;
public:
   incount(int a=0,int b=0){   c1=a; c2=b;  }
   void Show(void){cout<<"c1="<<c1<<'\t'<<"c2="<<c2<<endl;}
   incount operator++(){ c1++; c2++;       return *this;      }//前置++
   incount operator ++(int)//后置++
   { incount c;    c.c1=c1;          c.c2=c2;          c1++;    c2++;    return c;
   }
};
void main(void)
{           incount ic1(10,20),ic2(100,200), ic3 , ic4;
            ic1.Show ();
            ic2.Show ();
            ic3=++ic1;//ic3=ic1.operator++()
            ic4=ic2++;//ic4=ic4.operator++(3)
            ic3.Show ();
            ic4.Show ();
}
```

因为结果要赋值，所以函数要有返回值

```
ic1.c1=11    ic3.c2=21
ic3.c1=11    ic3.c2=21
ic2.c1=20    ic2.c2=200
ic4.c1=100   ic4.c2=200
```

说明

- 运算符重载为成员函数时，是由一个操作数调用另一个操作数。
- 即函数的实参只有一个或没有。

```
A  a ,b ,  c;
c=a+b;
```

实际上是c=a.operator+(b);

```
c=++a;
```

实际上是c=a.operator++( );

```
c+=a;
```

实际上是c.operator+=(a);

重载+=

运算符重载为友元函数

- 友元函数是在类外的普通函数，与一般函数的区别是可以调用类中的私有或保护数据。
- 将运算符的重载函数定义为友元函数，参与运算的对象全部成为函数参数。

```
A  a ,b ,  c;

c=a+b;
```

实际上是 c=operator+(a, b);

```
c=++a;
```

实际上是 c=operator++(a);

```
c+=a;
```

实际上是 operator+=(c, a);

例子

- 格式为：

```
friend <类型说明> operator<运算符>(<参数表>)
{.....}

c=a+b;   //  c=operator+( a, b)

friend   A   operator + (A &a, A &b)
{.....}
```

例子

```
class A
{       int i;
public:
        A(int a=0)      { i=a;  }
        void Show(void)     {       cout<<"i="<<i<<endl;        }
 friend A operator +(A &,A &);//友元函数，两个参数，为引用
};
A operator +(A &a , A &b)
        {A t;   t.i=a.i+b.i;     return t; }
void main(void)
{       A a1(10),a2(20),a3;
        a1.Show ();
        a2.Show ();
        a3=a1+a2;     //重新解释了加法，可以直接进行类的运算
        a3.Show ();
}
```

相当于a3=operator+(a1,a2)

例子

- ++为前置运算时，它的运算符重载函数的一般格式为：
- A operator ++(A &a)
- { ……; }
- ++为后置运算时，它的运算符重载函数的一般格式为：
- A operator ++(A &a, int)
- { ……; }

```
A   a, b;

b=++a;      A operator ++( A a ){ .... }

b=a++;      A operator ++(A a, int){ .... }
```

例子（代码）

```
class A
{       int i;
public:public:
        A(int a=0)      { i=a;  }
        void Show(void)     {       cout<<"i="<<i<<endl;        }
        friend A operator++(A &a){ a.i++;  retrurn a;}
        friend A operator++(A &a, int n)
        { A  t;         t.i=a.i;   a.i++;       return t;}
};
void main(void)
{       A a1(10),a2,a3;
        a2=++a1;
        a3=a1++;
        a2.Show();     a3.Show ();
}
```

相当于a2=operator++(a1)

相当于a3=operator++(a1,int)

例子

```
class incount
{       int c1,c2;
public:
   incount(int a=0,int b=0){   c1=a; c2=b;  }
   void Show(void){cout<<"c1="<<c1<<'\t'<<"c2="<<c2<<endl;}
   friend incount operator ++(incount &);//前置
   friend incount operator ++(incount &,int);//后置
};
incount operator++(incount &c)
{       c.c1++;  c.c2++;  return c;}
incount operator ++(incount &c,int)
{       incount cc;       cc=c;     c.c1++;  c.c2++;  return cc;}
void main(void)
{       incount ic1(10,20),ic2(100,200), ic3 , ic4;
        ic1.Show ();
        ic2.Show ();
        ic3=++ic1;//ic3=operator(ic1)
        ic4=ic2++;//ic4=operator(ic2，n)
        ic3.Show ();
        ic4.Show ();}
```

例子

```
class  ThreeD{
        float x,y,z;
public:  ThreeD(float  a=0,float b=0, float c=0){x=a;y=b;z=c;}
        friend ThreeD & operator ++(ThreeD &);//前置
        friend ThreeD   operator ++(ThreeD & ,int);//后置
        void Show(){cout << "x="<<x<<'\t'<<"y="<<y<<'\t'<<"z="<<z<<'\n';}
};
ThreeD & operator ++(ThreeD & t)
{       t.x++;   t.y++;   t.z++;   return t;          }
ThreeD   operator ++(ThreeD &t ,int )
{       ThreeD   temp=t;          t.x++;   t.y++;   t.z++;   return temp;}
void  main(void )
{       ThreeD m1(25,50, 100),m2(1,2,3),m3;
        m1.Show();
        m3=++m1;
        m1.Show();       m3.Show();
        m3=m2++;
        m2.Show();       m3.Show();
}
```

例子

- 对双目运算符，重载为成员函数时，仅一个参数，另一个被隐含；重载为友元函数时，有两个参数，没有隐含参数。
- 一般来说，单目运算符最好被重载为成员函数；对双目运算符最好被重载友元函数。

模板

- 函数模板
- 在一个int 型数组中，查找最大的数…
- 在一个double 型数组中，查找最大的数…
- 在一个float 型数组中，查找最大的数…
- 在一个Objext[] 型数组中，查找最大的数…

模板

- 模板是一个泛型编程的概念，即不考虑类型的一种编程方式，模板分为模板函数和模板类
- 人们需要编写很多个形式和功能都很相似的函数来实现某些功能只因为类型不同，就需要重载，很多个版本，写起来就比较麻烦，也比较啰嗦，此时，就可以使用模板函数来解决这种问题

模板函数

- 模板函数：建立一个通用的函数，**其返回值类型，形参的类型都不确定指定**
- 而是**采用虚拟类型来代替**，对于功能相同的函数，就无须重复写代码
- 模板函数和普通函数长相及使用方法非常类似，唯一的区别，**类型参数化。**

语法

- template <typename T> //T代表起的类型名， template 关键字

```
T findmax(T arr[], int len)
{
    T val = arr[0];
    …
}
```

template <class T> //typename 和 class 是声明虚拟类型的关键字

例子 （代码）

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

template <typename T>
T findmax(T arr[],int len)
{
        T val = arr[0];
        for(int i = 0;i<len;i++)
        {
                if(arr[i] > val)
                {
                        val = arr[i];
                }
        }
        return val;
}

int main()
{
        int arr[4] = {1,5,7,80};
        int result1 = findmax <int>(arr,4);
        //在使用函数名为findmax<int>时，表示把int类型带入模板
        printf("MAX = %d\n",result1);

        double abc[3] = {1.0,5.1,6.2};
        double result2 = findmax<double>(abc,3);
        printf("MAX = %.1f\n",result2);

        return 0;
}
```

模板函数调用

- （1） 第一种： 显式类型调用

  函数名 <实际类型1，实际类型2>（参数列表）；

  如： findmax<int>(1,2);

- (2) 第二种： 隐式类型调用

  函数名（参数）；

  如： findmax(1,2)

普通函数和函数模板的区别

- 函数只可以有一种数据类型相匹配。模板有多种类型
- 隐式调用优先使用普通函数，只有普通函数不匹配才使用函数模板
- 函数模板只有在调用时，才会构建函数，而普通函数是在编译时。
- 普通函数调用时候可以发生自动类型转换，而函数模板不行。

模板函数重载

- 和普通函数的重载相似、在同一个作用域，
- 函数名相同参数列表不同的多个函数参数不同：

  个数不同　类型不同　顺序不同

- （1）前提条件函数名必须相同
- （2）返回值不能决定调用哪个函数体
- （3）形参列表的不同才是决定是否重载的重要原因

模板函数优点

- 它是类型无关的，因此具有很高的可复用性。
- 它是平台无关的，可移植性
- 可用于基本数据类型

编译后的代码包含相同函数/类的多个副本

```
template <typename T>
T myMax(T x, T y)
{
    return (x > y)? x: y;
}

int main()
{
   cout << myMax<int>(3, 7) << endl;
   cout << myMax<char>('g', 'e') << endl;
   return 0;
}
```

Compiler internally generates and adds below code

```
int myMax(int x, int y)
{
    return (x > y)? x: y;
}
```

Compiler internally generates and adds below code.

```
char myMax(char x, char y)
{
    return (x > y)? x: y;
}
```

类模板

- 类模板，可以定义相同的操作，拥有不同数据类型的成员属性。
- 模板类的作用和模板函数的相同，都是为了减少代码量方便操作
- 通常使用template来声明。告诉编译器，碰到T不要报错，表示一种泛型.
- 一个类模板(类生成类)允许用户为类定义个一种模式，使得类中的某些数据成员、默认成员函数的参数，某些成员函数的返回值，能够取任意类型(包括系统预定义的和用户自定义的)。

定义

```
template<typename 类型参数>
class 类名
{
    类成员说明
};
```

例子（代码）

```
template <typename T>
class Complex{

private:
   T a;
   T b;

public:
   //构造函数
   Complex(T a, T b)
   {
      this->a = a;
      this->b = b;
   }

   //运算符重载
   Complex<T> operator+(Complex &c)
   {
      Complex<T> tmp(this->a+c.a, this->b+c.b);
      cout<<tmp.a<<endl;
      cout<<tmp.b<<endl;
      return tmp;
   }
};

int main()
{
   //对象的定义，必须声明模板类型，因为要分配内容
   Complex<int> a(10,20);
   Complex<int> b(20,30);
   Complex<int> c = a + b;

   return 0;
}
```

例子

- 类模板进过实例化之后产生了三个模板类，并产生了A、B、C三个对象。
- 注意：类的成员函数在类外实现时需要在函数定义之前进行模板声明，在成员函数名前写上“类名::”；

```
template <typename T>
class Arr
{
public:
    Arr(T x,T y)
    {
        a=x;
        b=y;
        cout << "Arr()" << endl;
    }
    T Area();
private:
    T a,b;
};

//类模板的成员方法类外实现
template<typename T>
void Arr<T>::Area()
{
     return x*y;
    cout << "void Arr<T>::Area()" << endl;
}

int main()
{
    Arr<int>A(1,2);
    Arr<float>B(2,3);
    Arr<double>C(3,4);
    return 0;
}
```

注意

- 类模板，实际上是建立一个通用类，内部的数据成员，成员函数的返回值类型和参数类型不具体说明，用类型参数来代表。
- 当使用类模板定义对象时，系统会根据实参的类型去替换模板中的类型参数，从而实例化出一个模板类，用以实现具体的功能。
- 类模板不编译
- 会在编译期根据使用方式，生成对应的类代码
- 类模板中没有使用到的成员方法不会在编译器生成对应的指令
- 类模板使用时必须加上模板类型参数，类模板无法自己推导类型参数

实际使用例子 （代码）

- 自己建立 类模板 类函数等 实现代码的通用性与可重用性！

本章结束！