# C++程序设计

主讲：王老师



尚德机构

学习是一种信仰

# 教材介绍

C++程序设计

（2008年版）

编著：刘振安

机械工业出版社

# 考试题型

单选题　　1分×20题 = 20分

填空题　　1分×20题 = 20分

程序改错题　4分×5题 = 20分

完成程序题　4分×5题 = 20分

程序分析题　5分×2题 = 10分

程序设计题 10分×1题 = 10分

# 第四章　类和对象

# 本章主要内容

- 类及其实例化
- 构造函数
- 析构函数
- 调用复制构造函数的综合实例
- 成员函数重载及默认参数

# §4.1 类及其实例化

对象就是一类物体的实例，将一组对象的共同特征抽象出来，就形成了类的概念，如从直角坐标系的点抽象出点的概念，这就是point类，point类只是表示类名是point，它的点位置有两个坐标属性，只有当产生一个具体的坐标点A，A的两个属性具有了具体的值，才能代表实际的点，即实例化了。

# 类的定义

## 一、类的定义

类是对一组性质相同的对象的程序描述，也属于一种用户自己构造的数据类型，也要遵循C＋＋的规定，如先声明后使用、是具有唯一标识符的实体、在类中声明的任何成员不能使用extern、auto和register关键字修饰、类中声明的变量属于该类等。

与其他数据类型不同的是，类除了数据，还可以有对数据进行操作的函数，分别称为类的数据成员和成员函数，而且不能在类的声明中对数据成员使用表达式进行初始化。

# 声明类

1、声明类

C++中声明类的一般形式为：

```
class 类名{
private:
    私有数据和函数
public:
    公有数据和函数
protected:
    保护数据和函数
};
```

# 声明类

类的声明以关键字class开始，其后跟类名，类所声明的内容用花括号括起来，称为类体，右花括号后的分号作为类的声明语句的结束标志。

类中定义的数据和函数称为这个类的成员，无论是数据成员还是成员函数，均具有一个访问权限，分别是私有、公有和保护，通过前加关键字private、public和protected来定义，如无关键字，则所有成员默认声明为private权限。

# 2、定义成员函数

成员函数在声明后还必须在程序中通过定义来实现对数据成员的操作。

定义成员函数的一般形式为：

```
返回类型 类名::成员函数名（形参列表）
{
    成员函数的函数体      //内部实现
}
```

其中“::”是作用域运算符，类名是指成员函数所属类的名字，用于表示其后的成员函数属于这个特定的类。其中返回类型是指这个成员函数返回值的类型。

# 2、定义成员函数

如：void Point :: Setxy(int a，int b)

{x = a；y = b；}

语句表示定义属于Point类的成员函数Setxy，该成员函数带有两个整型形参，函数没有返回值。

可以使用关键字inline将成员函数定义为内联函数，

如：inline int Point :: Getx（） {return x；}

如果在声明类的同时，在类体内给出成员函数的定义，也默认为内联函数，例如将声明Getx的语句“int Getx（）；”

改为“int Getx（）{return x；}”，则Getx为内联函数。

# 3、数据成员的赋值和初始化

赋值和初始化是两个不同的概念：

赋值是在有了对象A之后，对象A调用成员函数Setxy实现的，如A.Setxy(25，56);，不能在类体内或类体外面给数据成员赋值，例如用语句“int x=25,y=56;”给类的数据成员赋值，那么不论是在类体内还是类体外，都是错误的，类的数据成员的赋值，除通过例如语句” B=A;”实现，就只能通过调用类的赋值成员函数或构造函数才能实现。

初始化是指产生对象时就使对象的数据成员具有指定值，它是通过使用与类同名的构造函数实现的，如在类体中声明构造函数Point（int，int）后，在主函数中利用构造函数产生对象B并初始化：Point B（15，25）。

# 3、数据成员的赋值和初始化

- 不能在类体内或类体外面给数据成员赋值，当然在类体外面就更不允许了。
- 数据成员的具体值是用来描述对象的属性的。只有产生了一个具体的对象。这些数据值才有意义。
- 如果在产生对象时就使对象的数据成员具有指定值，则称为对象的初始化。
- 赋值和初始化时两个不同的概念：
  - ✓ 赋初值是在有了对象之后，对象调用自己的成员函数实现赋值操作；
  - ✓ 初始化是使用同名的构造函数实现的。

# 二、使用类的对象

例如对Point类，使用Point在程序中声明变量，具有Point类的类型的变量称为Point的对象，只有产生类的对象后，才能使用类的数据成员和成员函数。

类Point不仅可以声明对象，还可以定义对象的引用和对象的指针，其中对象和引用都使用运算符“.”访问对象的数据成员和成员函数，指针则使用“->”运算符，语法如下：

```
Point A，B；     //定义Point类型的对象A和B
Point *p = &A；   //定义指向对象A的Point类型的指针p
Point &R = B；    //定义R为Point类型对象B的引用
A.Setxy（25，88）；  //为对象A赋初值
R.Display（）；  //显示对象B的数据成员之值
p->Display（）；  //显示指针p所指对象的数据成员之值
```

例：使用内联函数定义Point类及使用Point类指针和引用

```
#include <iostream>
using namespace std;
class Point {                    //使用内联函数定义类point
private:
        int x,y;                 //私有数据成员
public:
        void Setxy(int a,int b)          {x=a;y=b;}
        void Move(int a,int b)           {x=x+a;y=y+b;}
        void Display( )  {cout<<x<<","<<y<<endl;}
        int Getx( ) {return x;}
        int Gety( ) {return y;}
};      //类定义以分号结束
void print(Point *a){a->Display( );}
void print(Point&a){a.Display( );}
void main( ){
        Point A,B,*p;  //声明对象和指针
        Point &RA=A;  //声明对象RA为对象A的引用
        A.Setxy(25,55);          //使用成员函数为对象A赋值
        B=A;        //例如通过int x = 25，y = 55；对类的私有数据赋值是错误的
        p=&B;
        p->Setxy(112,115);  //使用指针调用函数setxy重设B的值
        print (p);          //传递指针显示对象B的属性
        p->Display( );       //使用指针调用display函数显示对象B的属性
        RA.Move(-80,23);
        print(A);          //使用对象和对象指针的效果一样
        print(RA);}
```

```
112,115
112,115
-55,78
-55,78
```

# 二、使用类的对象

如果在上例中的print函数或主函数使用如下语句，将产生错误：

cout<<A.x<<“,”<<A.y<<endl;　　这是因为x和y都是对象A的私有数据成员，不能在类的外面直接使用它们，而只能通过类中的公有成员函数来使用它们。

通过上例，可归纳出有关类的一些规律如下：

①类的成员函数可以直接使用自己类的私有成员（数据成员和成员函数）；

②类外面的函数不能直接访问类的私有成员，而只能通过类的对象使用该类的公有成员函数；

③对象A和B的成员函数的代码一样，两个对象的区别是属性的取值。

# 二、使用类的对象

1、有关类的一些规律如下：类的成员函数可以直接使用自己类的私有成员（数据成员和成员函数）；类外面的函数不能直接访问类的私有成员，而只能通过类的对象使用该类的公有成员函数；对象A和B的成员函数的代码一样，两个对象的区别是属性的取值。

2、定义类对象指针的语法：

类名* 对象指针名；对象指针名=对象的地址；

也可以直接进行初始化。即：类名 * 对象指针名=对象的地址；

类对象的指针可以通过"->"运算符访问对象的成员，即：

对象指针名->对象成员名

# 三、数据封装

面向对象的程序设计是通过为数据和代码建立分块的内存区域，以便提供对程序进行模块化的一种程序设计方法，这些模块可以被用作样板，在需要时再建立副本。由此，对象是计算机内存中的一块区域，通过将内存分块，每个模块在功能上保持相对独立。

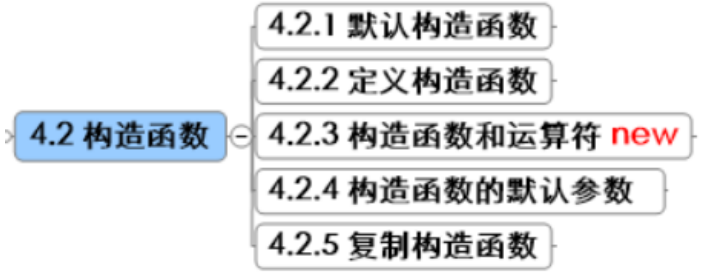
面向对象是消息处理机制，对象之间只能通过成员函数调用实现相互通信。当对象的一个函数被调用时，对象执行其内部的代码来响应这个调用，使对象呈现出一定的行为，对象被视为能作出动作的实体，对象使用这些动作完成相互之间的作用。

# 三、数据封装

C++通过**类**实现数据封装，即通过**指定各成员的访问权限**来实现。一般情况下将数据说明为私有的，以便隐藏数据；而将部分成员函数说明为公有的，用于提供外界和这个类的对象相互作用的接口（界面），从而使其他函数也可以访问和处理该类的对象。

公用的成员函数是外界所能观察到（访问到）的对象界面，它们所表达的功能构成对象的功能，使同一个对象的功能能够在不同的软件系统中保持不变，这样当数据结构发生变化时，只需要修改少量的类的成员函数的实现代码，就可以保证对象的功能不变。只要对象的功能保持不变，则公有的成员函数所形成的接口就不会发生变化，这样，对象内部实现所作的修改就不会影响使用该对象的软件系统，这就是数据封装的益处。

# §4.2 构造函数




构造函数是用来实现对象初始化的特殊成员函数。

建立一个对象时，对象的状态（数据成员的取值）是不确定的。为了使对象的状态确定，必须对其进行正确的初始化。C++有称为构造函数的特殊成员函数，它可自动进行对象的初始化。

# 一、默认构造函数



- 当没有为一个类定义任何构造函数的情况下，C++编译器会自动建立一个不带参数的构造函数，以Point类为例，默认构造函数的形式为：

  Point::Point（）{ } 即它的函数体是空的。

- 一旦程序定义了自己的构造函数，系统就不再提供默认构造函数，这时如果没有再定义一个无参数的构造函数，但又声明了一个没有初始化的对象（如Point A；），则因系统已不再提供默认构造函数而造成编译错误。
- 同理，如果程序中定义了有参数的构造函数，又存在需要先建立对象数组后进行赋值操作的情况，则必须为它定义一个无参数的构造函数。

# 二、定义构造函数



## 1、构造函数的声明与定义

为了提高安全性和效率，构造函数的名字必须和类名相同，并在定义构造函数时不能指定返回类型，即不要返回值，即使是void类型也不可以，另外类可有多个构造函数。

①构造函数在类体里的声明形式：

类名（形参1，形参2，...形参n）；　//可没有形参

# 二、定义构造函数

②构造函数的定义形式：

假设数据成员为x1、x2、...xn，则有以下两种形式：

类名::类名（形参1，形参2,...形参n）:x1（形参1),x2(形参2),...xn（形参n）{ }

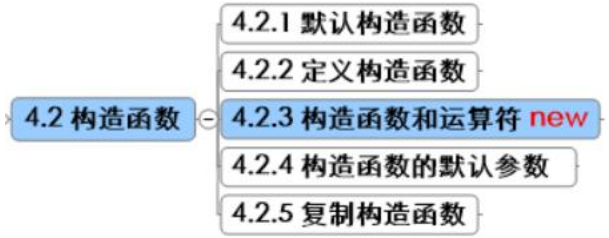
类名::类名（形参1，形参2，...形参n）

```
{
    x1 = 形参1;
    x2 = 形参2;
    ......
    xn=形参n;
}
```

构造函数的参数在排列时无顺序要求，只要保证相互对应即可，可以使用默认参数或重载。在程序中说明一个对象时，程序自动调用构造函数来初始化该对象。

# 2、构造函数的使用方法：自动调用

- 程序员不能在程序中显式地调用构造函数，构造函数是自动调用的。例如构造Point类的对象a，不能写成“Point a.Point（x，y）；”，只能写成“Point a（x，y）”。编译系统会自动调用构造函数Point（x，y）产生对象a并使用x和y将其正确的初始化。
- 在产生对象a之后，也不能使用a.Point（x，y）改变对象的属性值，而是要设计专门的成员函数如Setxy函数改变对象的属性值。
- 可以设计多个构造函数，编译系统根据对象产生的方法自动调用相应的构造函数，构造函数将在产生对象的同时初始化对象。




# 三、构造函数和运算符new

运算符new用于建立生存期可控的动态对象，new返回这个对象的指针，语法以Point类为例，见下面例子。

new和构造函数一同起作用，当使用new建立一个动态对象时，new首先分配足以保存Point类的一个对象所需要的内存，然后自动调用构造函数来初始化这块内存，再返回这个动态对象的地址。

使用运算符new建立的动态对象只能用运算符delete删除，所以使用完毕后，一定要记得使用delete释放空间。

运算符new用于建立生存期可控的动态对象，new返回这个对象的指针,使用运算符new建立的动态对象只能用运算符delete删除，以便释放所占空间。

例：构造函数的定义、使用过程及运算符new的使用方法

```
#include <iostream>
using namespace std;
class Point {
private:
        int x,y;
public:
        Point( );  //声明不带参数的构造函数
        Point(int,int);  //声明带两个参数的构造函数
};
Point::Point( ):x(0),y(0)  {cout<<"Initializing default"<<endl;}
//定义不带参数的构造函数
Point::Point(int a,int b):x(a),y(b)  //定义带两个参数的构造函数
{cout<<"Initializing"<<a<<","<<b<<endl;}

void main( ){
        Point A;  //使用不带参数的构造函数产生对象A
        Point B(15,25);  //使用带参数的构造函数产生对象B
        Point C[2];  //使用不带参数的构造函数产生对象数组C
        Point D[2]={Point(5,7),Point(8,12)};  //使用带参数的构造函数产生对象数组D
        Point *ptr1=new Point;  //使用不带参数的构造函数产生动态对象ptr1
        Point *ptr2=new Point(5,7);  //使用带参数的构造函数产生动态对象ptr2
        delete ptr1;  //删除动态对象ptr1，释放内存空间
        delete ptr2;  //删除动态对象ptr1，释放内存空间
}
```

```
Initializing default
Initializing15,25
Initializing default
Initializing default
Initializing5,7
Initializing8,12
Initializing default
Initializing5,7
```

# 四、构造函数的默认参数

如果程序定义了有参数构造函数，又想使用无参数形式的构造函数，解决的方法是将相应的构造函数全部使用默认参数设计。

例：设计构造函数的默认参数

```
#include <iostream>
using namespace std;
class Point {
private:
        int x,y;
public:
        Point(int=0,int=0);  //声明两个参数均为默认参数
};
Point::Point(int a,int b):x(a),y(b)  //定义构造函数
{cout<<"Initializing"<<a<<","<<b<<endl;}

void main( ){
        Point A;          //构造函数产生对象A
        Point B(15,25);   //构造函数产生对象B
        Point C[2];       //构造函数产生对象数组C
}
```

```
Initializing0,0
Initializing15,25
Initializing0,0
Initializing0,0
```

# 五、复制构造函数



复制构造函数的作用，就是通过拷贝方式使用一个类已有的对象来建立该类的一个新对象，所以又直译为拷贝构造函数。通常情况下，由编译器建立一个默认复制构造函数，其原形为：

类名::类名（const 类名&），或不加const修饰，但不论使用哪种形式的原型声明，为了提高程序的效率和安全性，都必须使用对象的引用作为形参。

如构造函数一样，复制构造函数也可自定义，如果自定义了，则编译器只调用自定义的复制构造函数，而不再调用默认的复制构造函数。

以Point类为例，复制构造函数的原型声明为：Point（Point&）；

它的定义如下：

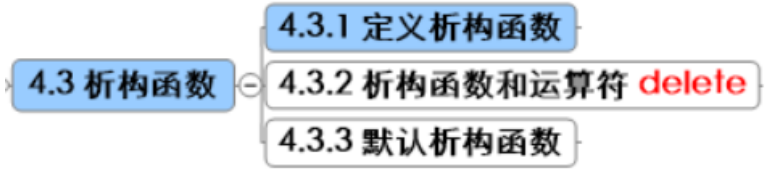
```
Point::Point（Point& t） {x = t.x；y = t.y；}
```

# §4.3 析构函数

析构函数和构造函数、复制构造函数都是构造型成员函数的基本成员，它的作用是在对象消失时，释放由构造函数分配的内存。

## 一、定义析构函数

因为调用析构函数也是由编译器来完成的，所以编译器必须总能知道应调用哪个函数最容易、也最符合逻辑的方式是指定这个函数的名称与类名一样。为了与构造函数区分，在析构函数前加一个“~”号，并且在定义析构函数时也不能指定返回类型，即使是void类型也不可以，同时析构函数也不能指定参数，但可以显式的说明参数为void，如A::~A（void），从函数重载角度分析，一个类只能定义一个析构函数，且不能指明参数，以便编译系统自动调用。




# §4.3 析构函数

①析构函数在类体里的声明形式：

～类名（）；

②析构函数的定义形式：

类名::～类名（）{ }

析构函数在对象的生存期结束时被自动调用，然后对象占用的内存被回收。

全局对象和静态对象的析构函数在程序运行结束之前调用。

类的对象数组的每个元素调用一次析构函数；

全局对象数组的析构函数在程序结束之前被调用。

# 二、析构函数和运算符delete

运算符delete与析构函数一起工作，当使用运算符delete删除一个动态对象时，它首先为这个对象调用析构函数，然后再释放这个动态对象占用的内存，这和使用运算符new建立动态对象的过程刚好相反。

例：使用Point类建立和释放一个动态对象数组

```
#include <iostream>
using namespace std;
class Point {
private:
        int x,y;
public:
        Point(int=0,int=0);  //声明两个参数均为默认参数
        ~Point( );  //声明析构函数
};

Point::Point(int a,int b):x(a),y(b)  //定义构造函数
{cout<<"Initializing"<<a<<","<<b<<endl;}
Point::~Point( )
{cout<<"Destructor is active"<<endl;} //定义析构函数

void main( ){
        Point *ptr=new Point[2];
        delete [ ]ptr;
}
```

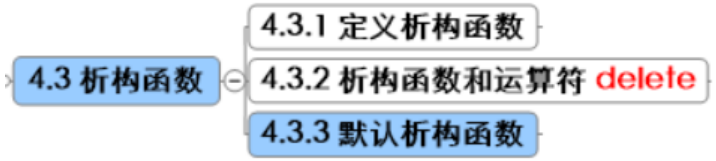
```
Initializing0,0
Initializing0,0
Destructor is active
Destructor is active
```

# 程序举例说明

表达式new Point[2]首先分配2个Point类的对象所需的内存，然后分别为这2个对象调用一次构造函数。

当使用delete释放动态对象数组时，用语句delete [ ]ptr；使运算符delete知道ptr指向的是动态对象数组，delete将为动态对象数组的每个对象调用一次析构函数，然后释放ptr所指向的内存。

当程序先后创建几个对象时，系统按照后建先析构的原则析构对象，当使用delete调用析构函数时，则按delete的顺序析构。



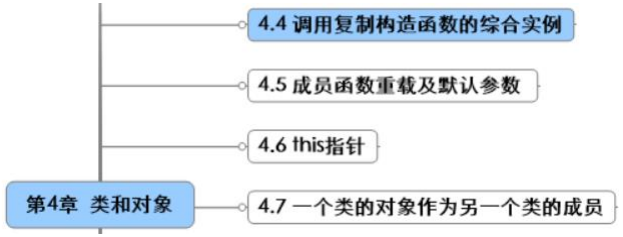

# 三、默认析构函数

如果在定义类时没有定义析构函数，则编译器将自动为类产生一个函数体为空的默认析构函数。

以Point类为例，默认析构函数形式如下：

Point::~Point（）{ }

# §4.4 调用复制构造函数的综合实例

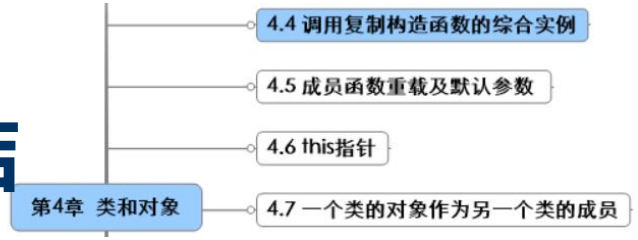



## 一、程序代码

```
#include <iostream>
using namespace std;
class Point {
private:
        int X,Y;
public:
        Point(int a=0,int b=0){X=a;Y=b;cout<<"初始化中"<<endl;}
    //定义有默认参数的构造函数（且为内联公有成员函数）
Point(const Point &p);      //声明复制构造函数
int GetX( ){return X;}
int GetY( ){return Y;}
void Show( ){cout<<"X="<<X<<",Y="<<Y<<endl;}
~Point( ){cout<<"删除…"<<X<<","<<Y<<endl;}
};
Point::Point(const Point &p){X=p.X;Y=p.Y;cout<<"拷贝初始化中
"<<endl;}   //定义必须使用对象的引用做形参的复制构造函数
void display(Point p){p.Show( );}    //点类对象做函数形参
void disp(Point &p){p.Show( );}      //点类对象的引用做函数形参
Point fun( ){Point A(101,202);return A;}  //函数返回值为点类对象
```

```
void main( ){
   Point A(42,35);     //定义点类对象A并赋值
   Point B(A);         //定义点类对象B，调用复制构造函数用A初始化B
   Point C(58,94);     //定义点类对象C并赋值
   cout<<"called display(B)"<<endl;
   display(B);
   cout<<"下一个…"<<endl;
   cout<<"called disp(B)"<<endl;
   disp(B);
   cout<<"call C = fun( )"<<endl;
   C=fun( );
   cout<<"called disp(C)"<<endl;
   disp(C);
   cout<<"out…"<<endl;
}
```

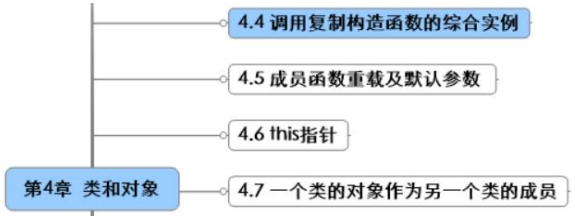
```
初始化中
拷贝初始化中
初始化中
called display(B)
拷贝初始化中
X=42,Y=35
删除…42,35
下一个…
called disp(B)
X=42,Y=35
call C = fun( )
初始化中
拷贝初始化中
删除…101,202
删除…101,202
called disp(C)
X=101,Y=202
out…
删除…101,202
删除…42,35
删除…42,35
```




# 三、调用复制构造函数的三种情况的小结

1、当用一个类的对象去初始化另一个对象时，需要调用复制构造函数。如例中：

```
Point A（14,25）；   //调用构造函数初始化对象A
Point B（A）；       //用对象A初始化对象B，调用复制构造函数
```




# 三、调用复制构造函数的三种情况的小结

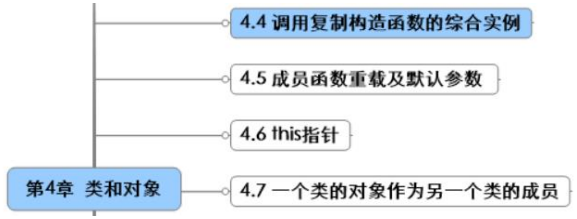
2、如果函数的形参是类的对象，调用函数时，进行形参与实参的结合时，需要调用复制构造函数。如例中：

Display(B)； //对象B作为display（）的实参，调用复制构造函数

因为函数display（Point p）是以对象作为形参，采用的是传对象的传值方式，所以需要产生一个副本，这就是临时对象，调用复制构造函数产生副本。在退出函数时，系统再自动调用析构函数析构临时对象。

而对于函数**disp(Point&)**而言，是以对象的引用作为形参，采用的是传引用的方式，所以不产生副本，即程序不需调用复制构造函数产生临时对象，而是直接进入disp( )的函数体执行语句。因没有临时对象产生，所以当退出disp( )函数时，也不需要调用析构函数来析构临时对象。

由此可见，函数参数使用对象的引用不会产生副本，从而不需调用复制构造函数和析构函数，方便安全，这也是推荐使用对象的引用而不使用对象的原因。



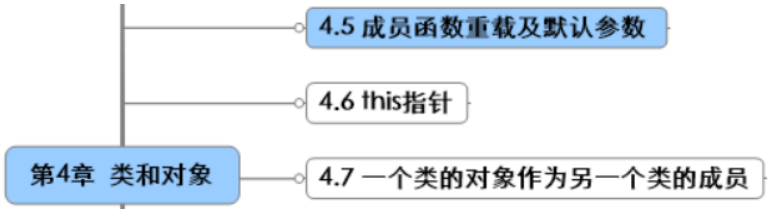

# 三、调用复制构造函数的三种情况的小结

3、如果函数的返回值是对象，当函数调用完成返回时，需要调用复制构造函数，产生临时对象，并在执行完返回值赋值语句后，析构临时对象和对象。如例中：

C = fun（）；　//fun的返回值赋给对象C，调用复制构造函数

程序执行过程为：调用函数fun（）{Point A(101,202); return A;}，在其内调用构造函数Point（int，int）创建对象A，然后执行return A；目的是使C = A，但这个赋值过程需要调用复制构造函数创建一个临时对象，用这个临时对象完成C = A，然后再析构这个临时对象，析构临时对象后，结束fun( )的函数调用，退出fun（）的函数体，此时再调用析构函数析构对象A。

4、上述实例的输出结果证实了调用构造函数和析构函数是按相反的顺序执行的，如果调用的函数程序里也产生了对象，则在函数程序里也遵循这一规律。




# §4.5 成员函数重载及默认参数

成员函数可重载或使用默认参数，下面例子就演示使用**私有成员函数封装函数**，并使用成员函数重载和默认参数。

例：构造一个求4个正整数中最大者的类Max，并在主程序中验证其功能。

小结：

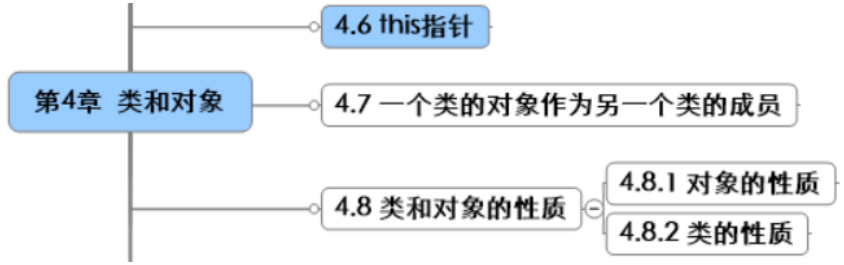
①程序演示了可在声明类的同时也声明类的对象，这里是声明对象数组A，作用与在主程序里使用语句“Max A[3]；”相同。当然，为了提高可读性，一般不在声明类时声明对象。

②Max类中重载了成员函数Maxi，其中一个原型为Maxi（ini，ini），用来求两数中的大者；另一个原型为Maxi（），它调用两次Maxi（ini，ini），然后再用这两次的结果作为Maxi（ini，ini）的参数，求出四个数中的最大值。

```
#include <iostream>
using namespace std;
class Max{  //声明类
private:  //封装数据成员和成员函数
        int a,b,c,d;  //数据成员
        int Maxi(int,int);  //声明Maxi只允许类内部的成员函数调用
public:
        void Set(int,int,int,int);  //公有成员函数Set的原型声明
        int Maxi( );  //求最大值
}A[3];  //声明类的对象数组，作用与主程序中语句“Max A[3]；”相同
      //类中成员函数的实现
int Max::Maxi(int x,int y) {return(x>y)?x:y;}  //求两个数的最大值
void Max::Set(int x1,int x2,int x3=0,int x4=0) {a=x1;b=x2;c=x3;d=x4;} //使用两个默认参数
int Max::Maxi( ){int x=Maxi(a,b);int y=Maxi(c,d);return Maxi(x,y);}

void main(){
        A[0].Set(12,45,76,89);  //为数组对象A[0]置初值
        A[1].Set(12,45,76);  //为数组对象A[1]置初值
        A[2].Set(12,45);  //为数组对象A[2]置初值
        for(int i=0;i<3;i++)
                cout<<A[i].Maxi( )<<" "; //输出对象求值结果
}
```

89 76 45



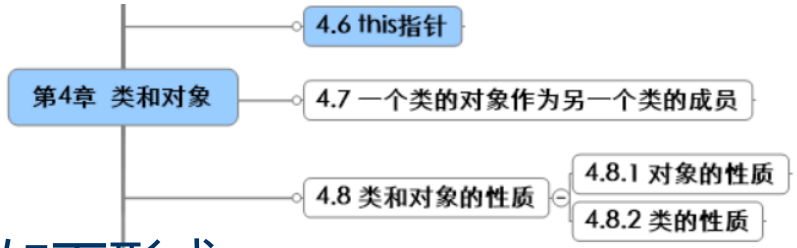

# §4.6 this指针

## 一、this指针的概念和作用

C++规定，当一个成员函数被调用时，系统将自动向它传递一个隐含的参数，该参数是**一个指向调用该函数的对象的指针，名为this指针**，从而使成员函数知道该对哪个对象进行操作。

使用this指针，保证了每个对象可以拥有自己的数据成员，但处理这些数据成员的代码却可以被所有的对象共享，从而提高了程序的安全性和效率。

this指针是C++实现封装的一种机制，它将对象和该对象调用的成员函数连接在一起，从而在外部看来，每个对象都拥有自己的成员函数。




# 二、this指针的实际形式

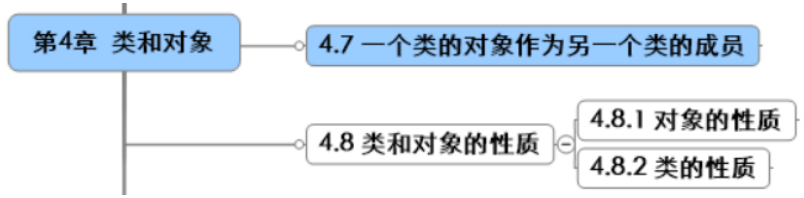
当执行A.Setxy(25,55)时，成员函数Setxy(int a,int b)实际上是如下形式：

```
void Point::Setxy（int a,int b,（Point * ）this)
{    this->x = a;
     this->y = b;
}          //此时成员函数的this指针指向对象A。
```

但是，一般情况下，即使在定义Setxy函数时使用指针，也不要给出隐含参数（Point *）this，写成下面的形式即可：

```
void Point::Setxy（int a，int b）
{    this->x = a;
     this->y = b;}
```

除非特殊需要，一般情况下都省略掉符号“this->”，而让系统进行默认设置。




# §4.7 一个类的对象作为另一个类的成员的实例

因为类本身就是一种新的数据类型，所以一个类的对象可以作为另一个类的成员。

例如有A、B两个类，可以通过在B类里定义A的对象作为B的数据成员，或者定义一个返回类型为A的函数作为B的成员函数。

例：使用Point类的对象及函数作为Rectangle类的数据成员及成员函数

```
#include <iostream>
using namespace std;
class Point{                        //定义点类Point
int x,y;                    //没有说明的，默认性质是private
public:
void Set(int a,int b){x=a;y=b;}  //定义内联的公有成员函数
int Getx( ){return x;}          //定义内联的公有成员函数
int Gety( ){return y;}          //定义内联的公有成员函数
};
class Rectangle{                            //定义矩形类Rectangle
Point Loc;
int H,W;                          //定义矩形类的高H和宽W
public:
void Set(int x,int y,int h,int w);
        Point * GetLoc( );     //声明返回Point类指针的成员函数GetLoc
        int GetHeight( ){return H;}        //定义内联的公有成员函数
        int GetWidth( ){return W;}         //定义内联的公有成员函数
};
void Rectangle::Set(int x,int y,int h,int w){Loc.Set(x,y); H=h;W=w;}
Point * Rectangle::GetLoc( ){return &Loc;}
```

```
void main( ){
Rectangle rect;  //定义Rectangle类的对象rect
rect.Set(10,2,25,20);
cout<<rect.GetHeight( )<<","<<rect.GetWidth(
)<<",";      //输出"25,20,"
Point * p=rect.GetLoc( );
cout<<p->Getx( )<<","<<p->Gety( )<<endl;
            //输出"10,2"
}
```

25,20,10,2

# §4.8 类和对象的性质

## 一、对象的性质

1、同一类的对象之间可以相互赋值，如语句：

Point A,B;　　A.Setxy（25,55）；　　B = A;

2、可以使用对象数组，如语句：Point A[3];　　定义数组A可以存储3个Point类的对象。

3、也可以使用指向对象的指针，使用取地址运算符&将一个对象的地址置于该指针中，如语句：　Point * p = &A;　　p->Display（）；

# 一、对象的性质

4、对象可以用作函数参数。

如果参数传递采用**传对象值**的方式，会在传值过程中产生副本，即临时对象，所以在被调用函数中对形参所作的改变**不影响调用函数中作为实参的对象**。

如果采取传对象的**引用（传地址）**的方式，当形参对象被修改时，相应的**实参对象也将被修改**。

如果采用**传对象地址值**的方式，则使用对象指针作为函数参数，**效果与传对象的引用一样**。

C++推荐使用对象的引用作为参数传递。为了避免被调用函数修改原来对象的数据成员，可以使用const修饰符。

# 一、对象的性质

C++推荐使用**对象的引用**作为参数传递，因为这样不会产生副本（即临时对象），在程序执行时不用调用构造函数，可直接进入函数体执行语句，在函数结束后，也不用调用析构函数析构临时对象。

如想避免被调用函数修改原来对象的数据成员，可使用**const**修饰符。

# 一、对象的性质

5、对象作为函数参数时，可以使用对象、对象引用和对象指针三种方式。以Point函数为例，它们的参数形式分别为：

```
void print（Point a）{a.Display；} //对象作为函数参数
void print（Point&a）{a.Display；} //对象引用作为函数参数
void print（Point * p）{p->Display；} //对象指针作为参数
```

它们的原型声明分别为：print（Point），print（Point&），print（Point *）。

对于对象A，print（&A）调用的是原型为print（Point *）的函数形式。

另外，函数重载不能同时采用如上3种同名函数，因为函数使用的参数为对象或对象引用时，编译系统无法区别这两个函数，重载只能选择其中的一种。

# 一、对象的性质

6、一个对象可以作为另一个类的成员。如上节例中，定义Point类的对象Loc为Rectangle类的数据成员，定义GetLoc( )为Rectangle类的返回Point类型指针的成员函数，当Rectangle类创建rect对象后，rect对象就将Point类的对象Loc作为自己的一个数据成员（此例中Loc充当的是顶点）。

# 二、类的性质

1、使用类的权限

①类本身的成员函数可以使用类的所有成员（私有和公有成员）。

②类的对象只能访问公有成员函数。

例如输出x只能使用A.Getx（），不能使用A.x。

③其他函数不能使用类的私有成员，也不能使用公有成员函数，它们只能通过定义类的对象为自己的数据成员，然后通过类的对象使用类的公有成员函数。

④虽然一个类可以包含另外一个类的对象，但这个类也只能通过被包含的类的对象使用那个类的成员函数，通过成员函数使用数据成员。

如Loc.Set（x，y）。

# 不完全的类的声明

2、不完全的类的声明

类不是内存中的物理实体，只有当使用类产生对象时，才进行内存分配，这种对象建立的过程称为实例化。

类必须在其成员使用之前先进行声明。然而，有时也可以在类没有完全定义之前就引用该类，此时将类作为一个整体来使用，如声明全局变量指针：

```
class MembersOnly;    //不完全的类声明
MembersOnly * club;    //定义全局变量类指针
Void main（）{... 函数体}    //主函数
class MembersOnly{...函数体};    //完全定义该类
```

不完全声明的类不能实例化，否则会编译出错；不完全声明仅用于类和结构，企图存取没有完全声明的类成员，也会引起编译错误。

# 空类

3、空类

尽管类的目的是封装代码和数据，它也可以不包括任何声明，如：class Empty{ };这种空类没有任何行为，但可以产生空类对象。

在开发大的项目时，需要在一些类还没有完全定义或实现时进行先期测试，定义空类可保证代码能正确被编译，从而允许先测试其中的一部分。此时常给空类增加一个无参数构造函数，如class Empty{public：Empty（）{ } } ，以消除强类型检查的编译器产生的警告。

# 类作用域

4、类作用域

①声明类时所使用的一对花括号形成类作用域，在类作用域中声明的标识符只在类中可见，如：class example { int num; } ;

int i=num; //定义整型数i，并用num赋值；错误，因为num在类外并没有定义，所以在此不可见

int num; //重新定义整型数num；正确，因为此num是重新定义的一个整型数，与类中说明的数据成员num具有不同的作用域

# 类作用域

②如果某成员函数的实现是在类定义之外给出的，则类作用域也包含该成员函数的作用域，因此，当在该成员函数内使用一个标识符时，编译器会先在类定义域中寻找，如下例：

```
class Myclass {
    int number;    //定义属于类Myclass的整型数number
public:
    void set(int);
};
int number;    //这个number不属于类Myclass
void Myclass::set（int i）{
number = i;    //使用的是类Myclass中的标识符number
}
```

# 类作用域

③类中的一个成员名可以通过使用类名和作用域运算符来显式的指定，称为成员名限定，例如：

```
void Myclass::set（int i）{
Myclass :: number = i;    //显示指定访问Myclass类中的标识符number
}
```

④在程序中，对象的生存期由对象说明来决定；类中各数据成员的生存期由对象的生存期决定，随对象存在和消失。

⑤使用struct关键字设计类与class相反，struct的默认控制权限是public，一般不用struct设计类，而是用struct设计只包含数据的结构，然后用这种结构作为类的数据成员。

# §4.9 面向对象的标记图



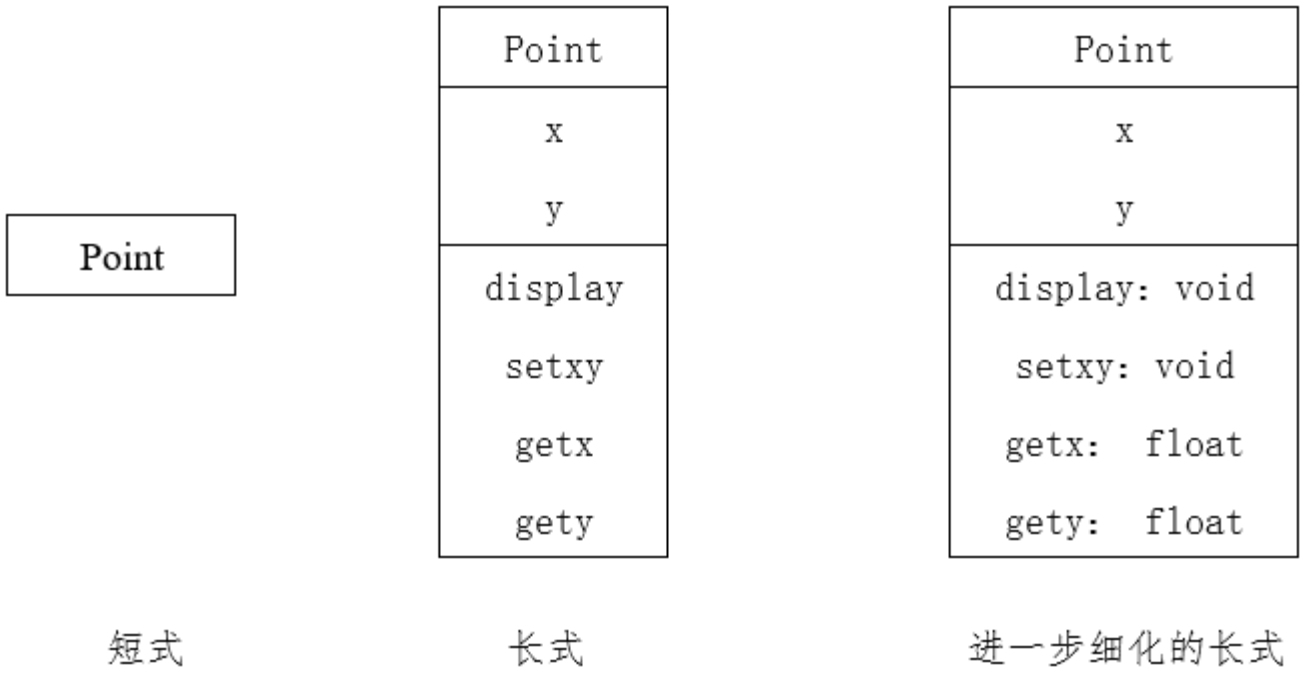
- 面向对象的早期，主要完善的是面向对象实现（即OOP阶段）阶段，后来转向面向对象分析（OOA）和面向对象设计（OOD），OOA和OOD更注重于使问题的描述更接近于真实的自然，它们采用一致的概念、原则和表示法，二者之间不存在鸿沟，不需要从分析文档到设计文档的转换。
- 为了设计、开发和相互交流的需要，人们采用图像形式将面向对象分析、设计和实现阶段对问题的描述直观的表示出来，称为标记图。实际使用的标记图种类很多，直到1992年OMG（面向对象管理组）制定的面向对象分析和设计的国际标准UML（统一建模语言）问世后，标记图的方法和技术才真正统一。
- UML是一种可视化建模语言，主要用于面向对象分析和建模。

# 一、类和对象的UML标记图

类和对象的标记符号类似，它们都只能表示静态特征。

## 1、类的标记图

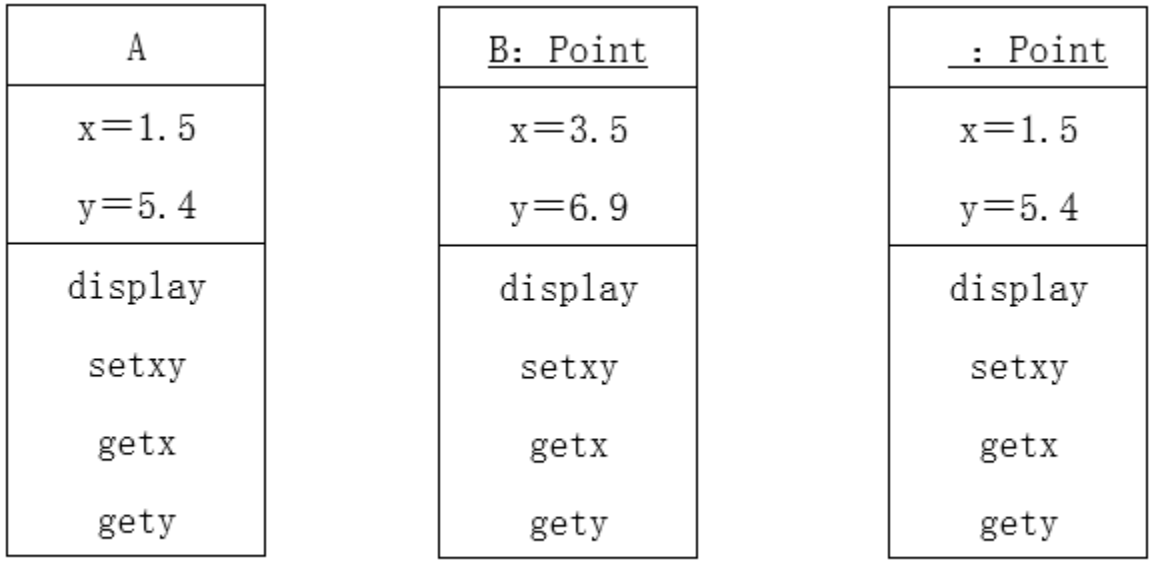
在UML语言中，类使用短式和长式两种方式表示。短式仅用1个含有类名的长方框表示，长式使用3个方框表示，最上面是类名，中间框填入属性（数据成员），最下面填入操作（成员函数），属性和操作可以根据需要进行细化。




短式　　长式　　进一步细化的长式

# 2、对象的标记图

对象的表示有3种方式，最简单的是只填写对象名称；完整的方式是在对象名的右边用冒号连接类名，并用下划线将它们标注出来；另外一种就是在还没有决定这个对象的名称时，可以不给出对象名，但不能省去冒号和类名。



# 二、对象的结构与连接



对象的结构是指对象之间的分类（继承）关系和组成（聚合）关系，统称关联关系。

对象的连接是指实例连接和消息连接，其中，对象之间的静态关系是通过对象属性之间的连接反映的，称为实例连接；对象行为之间的动态关系是通过对象行为（消息）之间的依赖关系表现的，称为消息连接。

# 二、对象的结构与连接

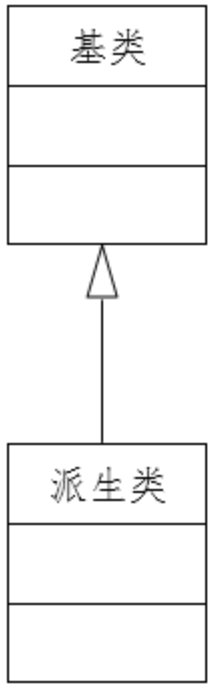


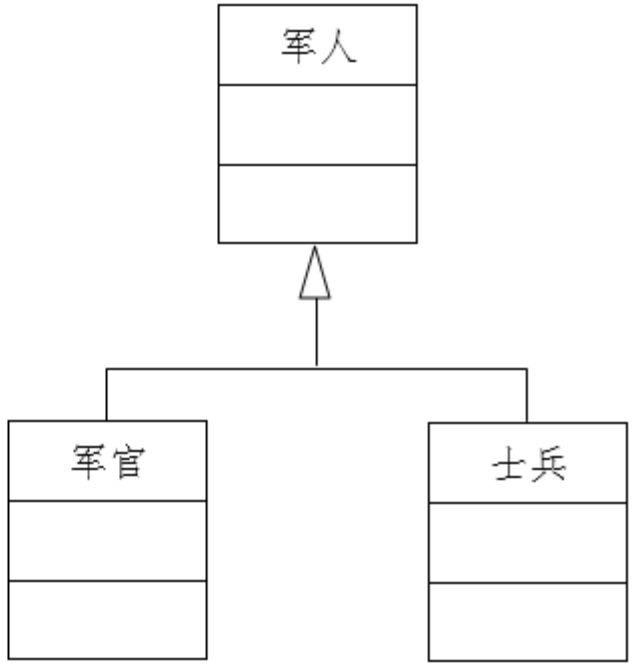
对象的结构是指对象之间的分类（继承）关系和组成（聚合）关系，统称关联关系。

对象的连接是指实例连接和消息连接，其中，对象之间的静态关系是通过对象属性之间的连接反映的，称为实例连接；对象行为之间的动态关系是通过对象行为（消息）之间的依赖关系表现的，称为消息连接。

# 一、分类关系及其表示

C + + 中的分类结构是指继承（基类/派生类）结构，UML使用一个空三角形表示继承关系，三角形指向基类。



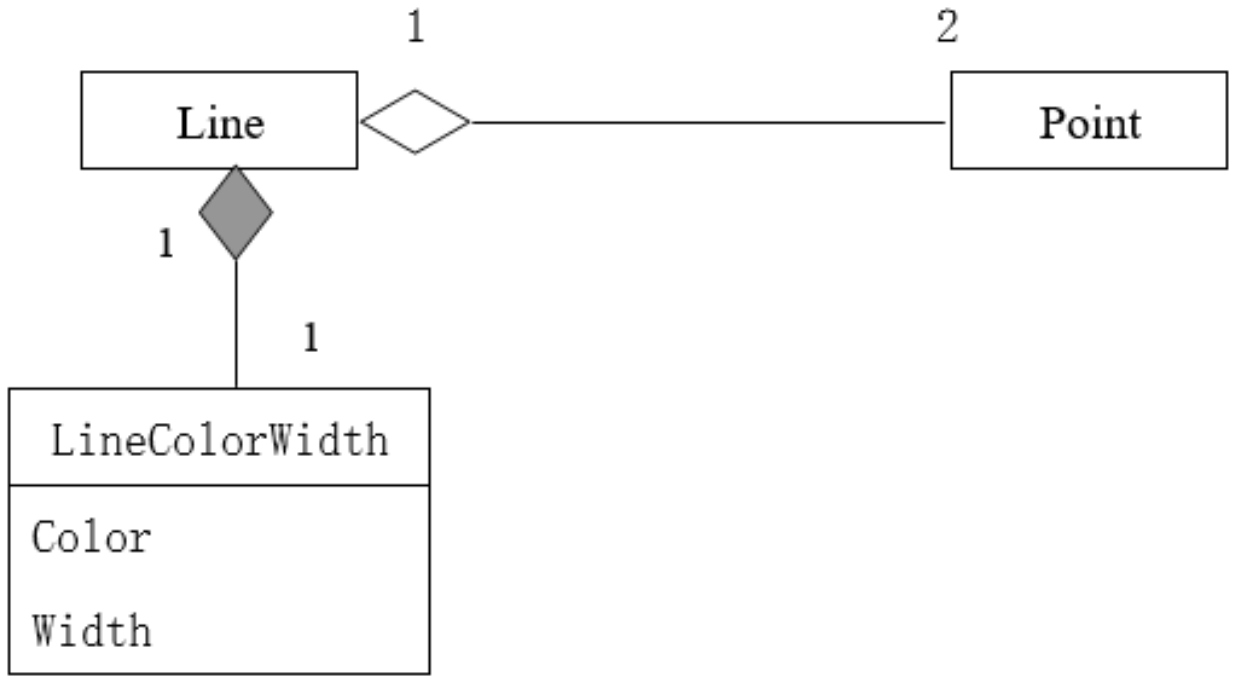
# 2、组合关系及其表示

组合关系说明的是整体与部分的关系，C++中最简单的是包含关系，如线段由两个点组成。

C++中的聚合有两种实现方式，一种是独立的定义，可以属于多个整体对象，并有不同的生存期，例如一个法律顾问可以属于几个单位，这种所属关系是可以动态变化的，称为聚集，用空心菱形表示，如下图中Line对象由两个Point对象组成。另一种方式是用一个类的对象作为一种广义的数据类型来定义整体对象的一个属性，构成一个嵌套对象，这种情况下，这个类的对象只能隶属于唯一的整体对象并与它同生同灭，称这种情况为组合，使用实心菱形表示，如下图中Line的颜色和宽度由另外一个嵌套对象LineColorWidth提供，它们生命周期一样。

# 2、组合关系及其表示

图中还给出关联时的数量关系。

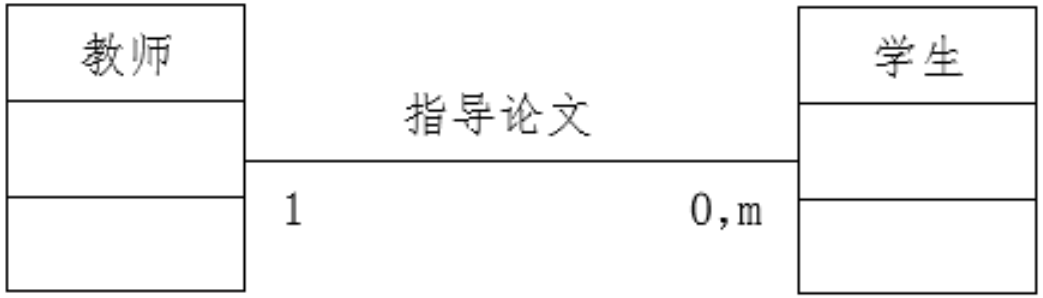


# 3、实例连接及其表示

实例连接反映对象之间的静态关系，例如车和驾驶员的关系，实例连接有一对一、一对多和多对多3种连接方式，用一条实线表示连接关系。

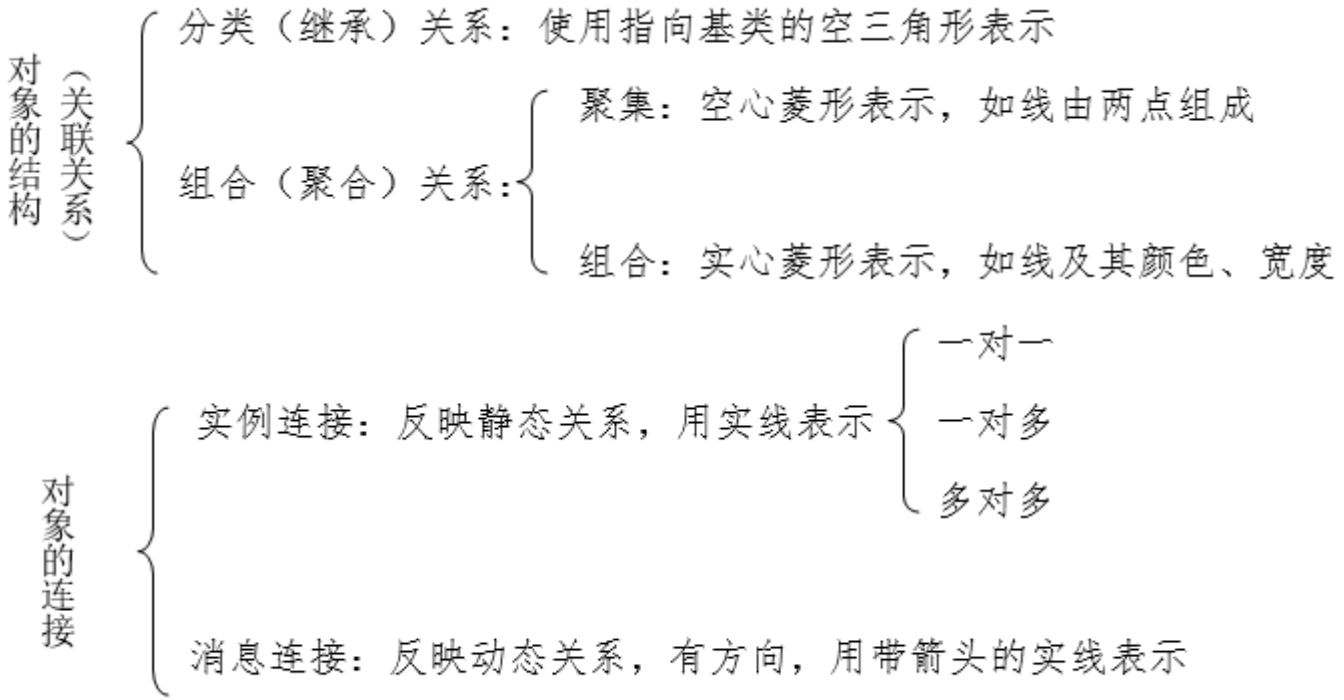
简单的实例连接是对象实例之间的一种二元关系，如教师类的对象实例和毕业班学生类的对象实例之间有一种指导毕业论文的关系，如下图：

这种连接的意义是表明一个教师为某些学生指导毕业论文。



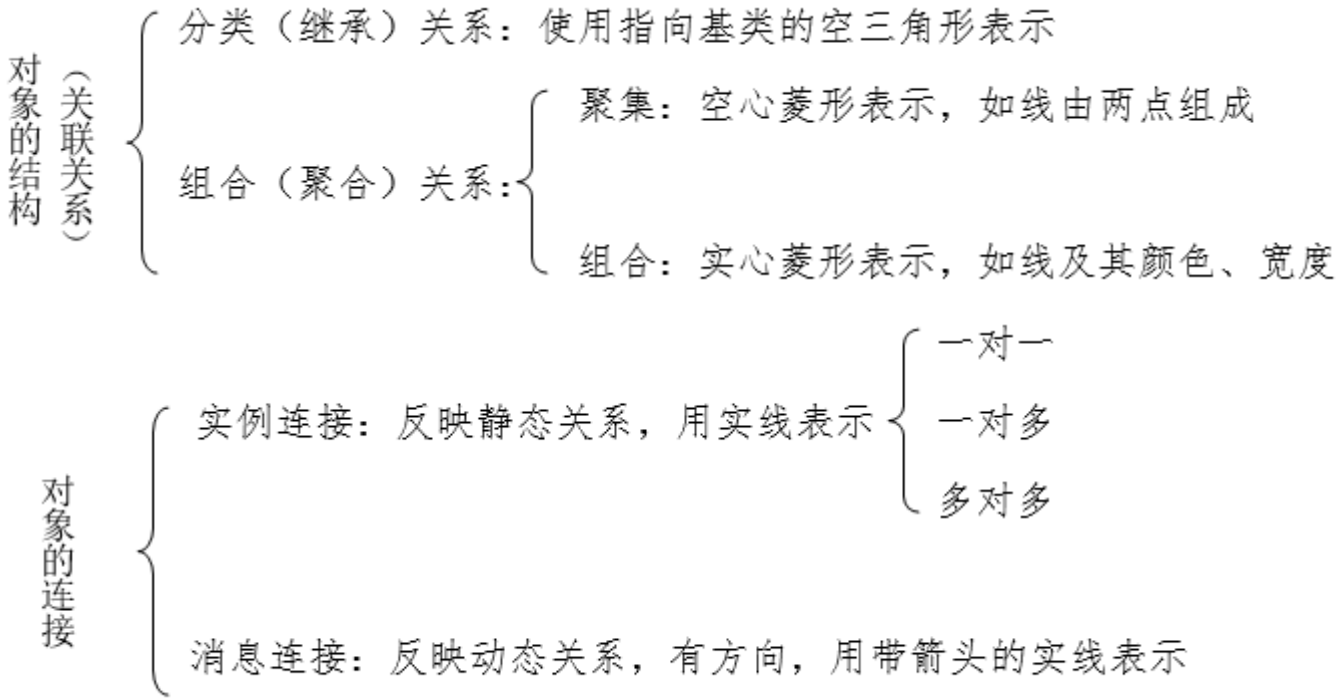
# 4、消息连接及其表示

- 消息连接描述对象之间的动态关系，即若一个对象在执行自己的操作时，需要通过消息请求另一个对象为它完成某种服务，则说第1个对象与第2个对象之间存在着消息连接。
- 消息连接是有方向的，使用一条带箭头的实线表示，从消息的发送者指向消息的接收者。



# 4、消息连接及其表示

- 消息连接描述对象之间的动态关系，即若一个对象在执行自己的操作时，需要通过消息请求另一个对象为它完成某种服务，则说第1个对象与第2个对象之间存在着消息连接。
- 消息连接是有方向的，使用一条带箭头的实线表示，从消息的发送者指向消息的接收者。



# 三、使用实例

假设已经定义了类Point，分别使用聚合和继承的方法组成Line类，不涉及操作，不用给出成员函数。

1、Line类包含Point类的对象

假设Point的对象为p1和p2，则它们构成Line类的两个数据成员，Line类使用这两对值作为两个坐标点，构成一条线段。显然，它们具有各自的生存周期，应使用空心菱形连接，关联关系是一对二。另外，Line类的数据成员p1和p2需要具有确定的属性值，这由Point类完成，为此Line类必须向Point类发消息，请求Point类为自己构造两个点对象，消息连接的箭头指向Point类。如图a：

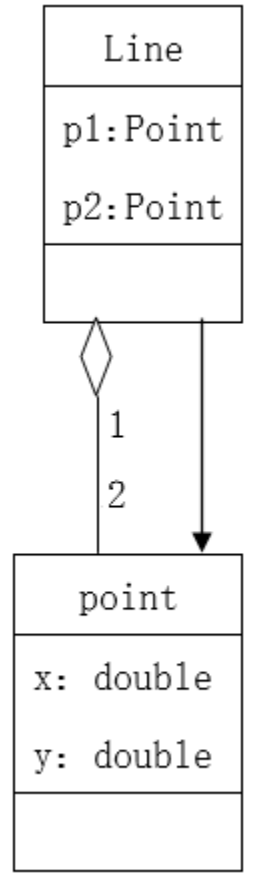



图 a

# 2、Line类继承Point类

## 2、Line类继承Point类

如用继承的思路构造Line类，则是Line类继承Point类的坐标点作为线段的一个坐标点，自己再定义一对属性作为另一个端点，这样就可构成一条线段。

因为基类是Point，所以三角形指向Point类，一般情况下，继承关系的消息传递规律清楚，为了保持图的清晰，可以不画它们之间的消息连接，另外，不管两者之间有多少消息，也均画出一条，如上图b。

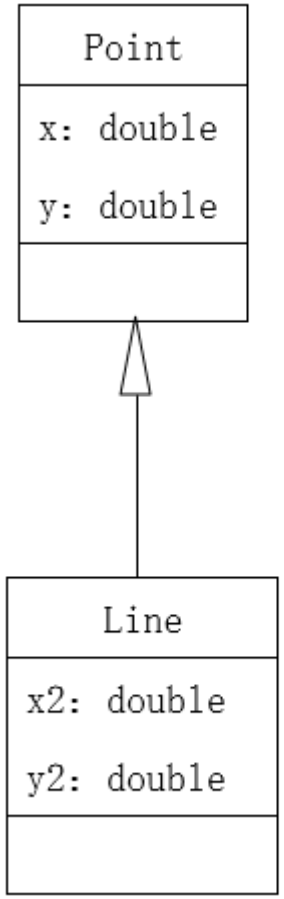



图 b

# 四、对象、类和消息

对象的属性是指描述对象的数据成员，对象属性的集合又称为对象的状态。

对象的行为是定义在对象属性上的一组操作的集合，操作（成员函数）是响应消息而完成的算法，表示对象内部实现的细节，对象的操作集合体现了对象的行为能力。

对象的属性和行为是对象定义的组成要素，分别代表了对象的静态和动态特征，无论对象是简单的或是复杂的，对象一般具有以下特征：

①有一个状态，由与其相关联的属性值集合所表征；

②有唯一标识名，可以区别于其他对象；

③有一组操作方法，每个操作决定对象的一种行为；

④对象的状态只能被自己的行为所改变；

⑤对象的操作包括自身操作（施加于自身）和施加于其他对象的操作；

⑥对象之间以消息传递的方式进行通信；

⑦一个对象的成员仍可以是一个对象。

其中，前三条是对象的基本特征，后四条属于特征的进一步定义。

# 四、对象、类和消息



消息是向对象发出的服务请求，它是面向对象系统中实现对象间的通信和请求任务的操作。消息传递是系统构成的基本元素，是程序运行的基本处理活动。一个对象所能接受的消息及其所带的参数，构成该对象的外部接口。对象接收它能识别的消息，并按照自己的方式来解释和执行。一个对象可以同时向多个对象发送消息，也可以接收多个对象发来的消息。消息值反映发送者的请求，由于消息的识别和解释取决于接收者，因而同样的消息在不同对象中可解释成不同的行为。

对象传送的消息一般由3部分组成：接收对象名、调用操作名和必要的参数。每个消息在类描述中用一个相应的方法给出，即使用成员函数定义操作。向对象发送一个消息，就是引用一个方法的过程，即实施对象的各种操作就是访问一个或多个在类中定义的方法。

消息协议是一个对象对外提供服务的规定格式说明，外界对象能够并且只能向该对象发送协议中所提供的消息，请求该对象服务。具体实现是将消息分为私有和公有消息，前者只供内部使用，后者是对外的接口，协议则是一个对象所能接受的所有公有消息的集合。

# §4.10 面向对象编程的文件规范

- 一般将类的说明放在头文件中，实现则放在.cpp文件中，主程序单独使用一个文件，这就是多文件编程规范。另外，非常简单的成员函数可以在声明中定义（即默认内联函数形式），而在.cpp文件中，要将头文件包含进去。
- 对规模较大的类，一般应该为每个类设立一个头文件和一个实现文件。但函数模板和类模板比较特殊，如果使用头文件说明，则要同时完成它们的定义。

# 一、编译指令

C++的源程序可包含各种编译指令，以指示编译器对源代码进行编译之前先对其进行预处理。所以的编译指令都以#开始，每条指令单独占用一行，同一行不能有其他编译指令和C++语句（注释例外）。

编译指令不是C++的一部分，但扩展了C++编程环境的使用范围，从而改善程序的组织和管理。

# 1、嵌入指令

嵌入指令#include指示编译器将一个源文件嵌入到该指令所在的位置。尖括号或双引号中的文件名可含有路径信息。如：

#include <\user\prog.h>

注意：由于编译指令不是C++的一部分，因此在这里表示反斜杠时只需使用一个反斜杠。如果在C++程序中表示上述文件名，则必须使用双反斜杠，如：

char fname [ ]= “\\user\\prog.h” ;

# 2、宏定义

宏定义以＃define开头，＃define指令定义一个标识符及串，在源程序中每次遇到该标识符时，编译器将自动用后面的串代替它。该标识符称为宏名，替换过程称为宏替换。

宏定义的一般形式为：＃define 宏名 替换正文

其中宏名必须是一个有效的C++标识符，替换正文可为任意字符组成的字符序列。宏名和替换正文之间至少有一个空格，一般将宏名写成大写字体。

注意：宏定义由新行结束，而不以分号结束，分号视为替换正文的一部分。当替换正文要书写在多行上时，除最后一行之外，每行的行尾要加上一个反斜线，表示宏定义继续到下一行，如：

```
＃define MAX(a，b) ((a)>(b)? \
                    (a):(b))
```

因宏定义有许多不安全因素，对需要使用无参数宏的场合，应该尽量使用const代替宏定义。如：语句“#define ABC 2150”和语句“const int ABC=2150;”作用一样，但后者比前者要安全。

宏定义不再使用时，可用＃undef删除。

# 3、条件编译指令

条件编译指令是指＃if、＃else、＃elif和＃endif，它们构成了类似C++的if选择结构，其中＃endif表示一条指令结束。

编译指令＃if用于控制编译器对源程序的某部分有选择的进行编译，该部分从＃if开始，到＃endif结束，如果＃if后面的常量表达值为真，则编译这部分，否则就不编译，此时，这部分代码相当于被删除。

编译指令＃else是在＃if测试失败的情况下建立另外一种选择，可以在＃else分支中使用编译指令＃error输出出错信息，形式为：＃error 出错信息　　其中出错信息是一个字符序列，当遇到＃error指令时，编译器显示其后面的出错信息，并中止对程序的编译。

编译指令可嵌套，规则和编译器对其处理的方式与if语句嵌套情况类似。

# 4、defined操作符

关键字defined不是指令，而是一个预处理操作符，用于判定一个标识符是否已经被 # defined 定义，如果标识符 identifier 已被 # defined 定义，则 defined（idertifier）为真，否则为假。

条件编译指令 # ifdef和 # ifndef用于测试其后的标识符是否被 # define定义，如果已被定义，则 # ifdef为真，# ifndef测试为假；如果没有被定义，则 # ifdef测试为假，# ifndef测试为真。

如：#if!defined (HEAD_H) 和 #ifndef HEAD_H 两语句效果一样。

# 二、在头文件中使用条件编译

在多文件设计中，由于文件包含指令可以嵌套使用，可能会出现不同文件包含了同一个头文件，这样会引起变量及类的重复定义。为了避免重复编译类的同一个头文件，可对这类头文件使用条件编译。例如Point.h可能会被嵌套包含，则使用如下形式：

```
#if! defined（POINT_H)
#define POINT_H
class Point{
..... //类体
};
#endif
```

# 本章总结


第4章 类和对象
4.1 类及其实例化
4.1.1 定义类
4.1.2 使用类的对象
4.1.3 数据封装
4.2 构造函数
4.2.1 默认构造函数
4.2.2 定义构造函数
4.2.3 构造函数和运算符 new
4.2.4 构造函数的默认参数
4.2.5 复制构造函数
4.3 析构函数
4.3.1 定义析构函数
4.3.2 析构函数和运算符 delete
4.3.3 默认析构函数
4.4 调用复制构造函数的综合实例
4.5 成员函数重载及默认参数
4.6 this指针
4.7 一个类的对象作为另一个类的成员
4.8 类和对象的性质
4.8.1 对象的性质
4.8.2 类的性质
4.9 面向对象的标记图
4.9.1 类和对象的UML标记图
4.9.2 对象的结构与连接
4.9.3 使用实例
4.9.4 对象、类和消息
4.10 面向对象编程的文件规范
4.10.1 编译指令
4.10.2 在头文件中使用条件编译

```
#include <iostream>
using namespace std;
class base{
    _____;
public:
    base(int x,int y){a=x;b=y;}
    _____(base &p){
        cout<<p.a<<","<<p.b<<endl;}
};
void main(){
    base b(78,87);
    b.show(b);
}
```

```
#include <iostream>
using namespace std;

class example
{
public:
        example(int n)
        {
                i=n;
                cout<<"Constructing ";
        }

        ~example( )
        {
                cout<<"Destructing ";
        }
        int get_i()
        {
                return i;
        }
private:
        int i;
};

int sqr_it(example o)
{
        return o.get_i()*o.get_i();
}

void main()
{
        example x(10);
        cout<<x.get_i()<<endl;
        cout<<sqr_it(x)<<endl;
}
```

```
#include <iostream>
using namespace std;
class example{
private:
        int i;
public:
        example(int n)
        {
                i=n;
                cout<<"Constructing ";
        }
        example(example &a)
        {
                i=a.i;
                cout<<"Copy Constructing "<<endl;
        }
        ~example( )
        {
                cout<<"Destructing ";
        }
        int get_i()
        {
                return i;
        }
};
int sqr_it(example o)
{
        return o.get_i()*o.get_i();
}

void main()
{
        example x(10);
        cout<<x.get_i()<<endl;
        cout<<sqr_it(x)<<endl;
}
```

```
Constructing 10
Copy Constructing
Destructing 100
Destructing
```

# 祝大家顺利通过考试！