第 4 章　类与对象

郑　莉　　清华大学

教材：C++语言程序设计（第5版）　郑莉　　清华大学出版社

目录

面向对象程序设计的基本特点

这一节我们首先来了解面向对象程序设计的基本特点

面向对象程序设计的基本特点是：抽象、封装、继承、多态

抽象实例——钟表

```
class  Clock {
  public:
    void setTime(int newH, int newM, int newS);
    void showTime();
  private:
    int hour, minute, second;
};
```

抽象

- 对同一类对象的共同属性和行为进行概括，形成类。
  - 先注意问题的本质，其次是实现过程或细节。
  - 数据抽象：描述某类对象的属性或状态（对象相互区别的物理量）。
  - 代码抽象：描述某类对象的共有的行为特征或具有的功能。
  - 抽象的实现：类。

类的封装举例

外部接口

```
class  Clock {
  public: void setTime(int newH, int newM, int newS);
          void showTime();
  private: int hour, minute, second;
};
```

边界

特定的访问权限

封装

- 将抽象出的数据、代码封装在一起，形成类。
  - 目的：增强安全性和简化编程，使用者不必了解具体的实现细节，而只需要通过外部接口，以特定的访问权限，来使用类的成员。
  - 实现封装：类声明中的{}

继承

- 在已有类的基础上，进行扩展形成新的类。
- 详见第7章

多态

- 多态：同一名称，不同的功能实现方式。
- 目的：达到行为标识统一，减少程序中标识符的个数。
- 实现：重载函数和虚函数——见第8章

类和对象的定义

- 对象是现实中的对象在程序中的模拟。
- 类是同一类对象的抽象，对象是类的某一特定实体。
- 定义类的对象，才可以通过对象使用类中定义的功能。

例4_1-1：钟表类

类的定义

```
#include<iostream>
using namespace std;
class Clock{
public:
   void setTime(int newH = 0, int newM = 0, int newS = 0);
   void showTime();
private:
   int hour, minute, second;
};
```

例4_1-1：钟表类

成员函数的实现

```
void Clock::setTime(int newH, int newM, int newS) {
   hour = newH;
   minute = newM;
   second = newS;
}
void Clock::showTime() {
   cout << hour << ":" << minute << ":" << second << endl;
}
```

例4_1-1：钟表类

对象的使用

```
int main() {
   Clock myClock;
   myClock.setTime(8, 30, 30);
   myClock.showTime();
   return 0;
}
```

运行结果：
8:30:30

类定义的语法形式

```
class 类名称
{
   public:
                    公有成员（外部接口）
   private:
                     私有成员
   protected:
                     保护型成员
}
```

类内初始值

- 可以为数据成员提供一个类内初始值
- 在创建对象时，类内初始值用于初始化数据成员
- 没有初始值的成员将被默认初始化。

类内初始值举例

```
class Clock {
public:
        void setTime(int newH, int newM, int newS);
        void showTime();
private:
        int hour = 0, minute = 0, second = 0;
};
```

类成员的访问控制

- 公有类型成员
- 私有类型成员
- 保护类型成员

类成员的访问控制

- 公有类型成员
  - 在关键字public后面声明，它们是类与外部的接口，任何外部函数都可以访问公有类型数据和函数。

类成员的访问控制

- 私有类型成员
  - 在关键字private后面声明，只允许本类中的函数访问，而类外部的任何函数都不能访问。
  - 如果紧跟在类名称的后面声明私有成员，则关键字private可以省略。

类成员的访问控制

- 保护类型成员
  - 与private类似，其差别表现在继承与派生时对派生类的影响不同，详见第七章。

对象定义的语法

类名　对象名;

例：Clock　myClock;

类中成员互相访问

直接使用成员名访问

类外访问

使用“对象名.成员名”方式访问 public 属性的成员

类的成员函数

- 在类中说明函数原型；
- 可以在类外给出函数体实现，并在函数名前使用类名加以限定；
- 也可以直接在类中给出函数体，形成内联成员函数；
- 允许声明重载函数和带默认参数值的函数。

内联成员函数

- 为了提高运行时的效率，对于较简单的函数可以声明为内联形式。
- 内联函数体中不要有复杂结构（如循环语句和switch语句）。
- 在类中声明内联成员函数的方式：
  - 将函数体放在类的声明中。
  - 使用inline关键字。

构造函数

<4.3.1～4.3.2>

这一节我们来学习类的构造函数。

定义对象时，如何进行初始化呢？对象的初始化方法需要在程序中规定好，为此，C++语法提供了一个特殊的机制——构造函数，下面我们就具体来学习如何在构造函数中规定对象的初始化方法

例4_1-2：构造函数

```
//类定义
class Clock {
public:
   Clock(int newH,int newM,int newS);//构造函数
   void setTime(int newH, int newM, int newS);
   void showTime();
private:
   int hour, minute, second;
};
```

例4_1-2： 构造函数

```
//构造函数的实现：
Clock::Clock(int newH,int newM,int newS): hour(newH),minute(newM),
   second(newS) {
   }
//其它函数实现同例4_1

int main() {
  Clock c(0,0,0); //自动调用构造函数
  c.showTime();
   return 0;
}
```

例4_1-2： 构造函数

```
class Clock {
public:
        Clock(int newH, int newM, int newS);  //构造函数
        Clock(); //默认构造函数
        void setTime(int newH, int newM, int newS);
        void showTime();
private:
        int hour, minute, second;
};
Clock::Clock(): hour(0),minute(0),second(0)  { }//默认构造函数
//其它函数实现同前
```

例4_1-2： 构造函数

```
int main() {
    Clock c1(0, 0, 0);          //调用有参数的构造函数
    Clock c2; //调用无参数的构造函数
    ……
}
```

构造函数的作用

- 在对象被创建时使用特定的值构造对象，将对象初始化为一个特定的初始状态。
- 例如：
  - 希望在构造一个Clock类对象时，将初试时间设为0:0:0，就可以通过构造函数来设置。

构造函数的形式

- 函数名与类名相同；
- 不能定义返回值类型，也不能有return语句；
- 可以有形式参数，也可以没有形式参数；
- 可以是内联函数；
- 可以重载；
- 可以带默认参数值。

构造函数的调用时机

- 在对象创建时被自动调用
- 例如：

```
Clock myClock（0,0,0）；
```

默认构造函数

- 调用时可以不需要实参的构造函数
  - 参数表为空的构造函数
  - 全部参数都有默认值的构造函数
- 下面两个都是默认构造函数，如在类中同时出现，将产生编译错误：

```
Clock();
Clock(int newH=0,int newM=0,int newS=0);
```

隐含生成的构造函数

- 如果程序中未定义构造函数，编译器将在需要时自动生成一个默认构造函数
  - 参数列表为空，不为数据成员设置初始值；
  - 如果类内定义了成员的初始值，则使用类内定义的初始值；
  - 如果没有定义类内的初始值，则以默认方式初始化；
  - 基本类型的数据默认初始化的值是不确定的。

“=default”

- 如果类中已定义构造函数，默认情况下编译器就不再隐含生成默认构造函数。
  如果此时依然希望编译器隐含生成默认构造函数，可以使用“=default”。
- 例如

```
class Clock {
public:
    Clock() =default; //指示编译器提供默认构造函数
    Clock(int newH, int newM, int newS);  //构造函数
private:
    int hour, minute, second;
};
```

委托构造函数

<4.3.3>

这一节我们学习委托构造函数

类中往往有多个构造函数，只是参数表和初始化列表不同，其初始化算法都是相同的，这时，为了避免代码重复，可以使用委托构造函数。

回顾

Clock类的两个构造函数：

```
Clock:: Clock(int newH, int newM, int newS) :
hour(newH),minute(newM),  second(newS)  {  //构造函数
}
Clock::Clock(): hour(0),minute(0),second(0) { }//默认构造函数
```

委托构造函数

- 委托构造函数使用类的其他构造函数执行初始化过程
- 例如：

```
Clock:: Clock(int newH, int newM, int newS):
hour(newH),minute(newM),  second(newS){
}
Clock:: Clock(): Clock(0, 0, 0) { }
```

复制构造函数

我们经常会需要用一个已经存在的对象，去初始化新的对象，这时就需要一种特殊的构造函数——复制构造函数；

默认的复制构造函数可以实现对应数据成员一一复制；

自定义的默认构造函数可以实现特殊的复制功能。

例4-2：Point类

```
class Point {   //Point 类的定义
public:
    Point(int xx=0, int yy=0) { x = xx; y = yy; }    //构造函数，内联
    Point(const Point& p); //复制构造函数
    void setX(int xx) {x=xx;}
    void setY(int yy) {y=yy;}
    int getX() const { return x; } //常函数（第5章）
    int getY() const { return y; } //常函数（第5章）
private:
    int x, y; //私有数据
};
```

例4-2：Point类

```
//复制构造函数的实现
Point::Point (const Point& p) {
  x = p.x;
  y = p.y;
  cout << "Calling the copy constructor " << endl;
}
```

例4-2：Point类

```
//形参为Point类对象
void fun1(Point p) {
    cout << p.getX() << endl;
}

//返回值为Point类对象
Point fun2() {
        Point a(1, 2);
        return a;
}
```

```
int main() {
   Point a(4, 5);
   Point b(a);   //用a初始化b。
   cout << b.getX() << endl;
   fun1(b);      //对象b作为fun1的实参
   b = fun2();  //函数的返回值是类对象
   cout << b.getX() << endl;
   return 0;
}
```

复制构造函数定义

复制构造函数是一种特殊的构造函数，其形参为本类的对象引用。作用是用一个已存在的对象去初始化同类型的新对象。

```
class 类名 {
public :
    类名（形参）；//构造函数
    类名（const 类名 &对象名）；//复制构造函数
    //      …
};
类名::类（ const 类名 &对象名）//复制构造函数的实现
{   函数体   }
```

隐含的复制构造函数

- 如果程序员没有为类声明复制构造函数，则编译器自己生成一个隐含的复制构造函数。
- 这个构造函数执行的功能是：用作为初始值的对象的每个数据成员的值，初始化将要建立的对象的对应数据成员。

“=delete”

- 如果不希望对象被复制构造
  - C++98做法：将复制构造函数声明为private，并且不提供函数的实现。
  - C++11做法：用“=delete”指示编译器不生成默认复制构造函数。
    - 例：

```
class Point {   //Point 类的定义
public:
  Point(int xx=0, int yy=0) { x = xx; y = yy; }    //构造函数，内联
  Point(const Point& p) =delete;  //指示编译器不生成默认复制构造函数
private:
  int x, y; //私有数据
};
```

复制构造函数被调用的三种情况

- 定义一个对象时，以本类另一个对象作为初始值，发生复制构造；
- 如果函数的形参是类的对象，调用函数时，将使用实参对象初始化形参对象，发生复制构造；
- 如果函数的返回值是类的对象，函数执行完成返回主调函数时，将使用return语句中的对象初始化一个临时无名对象，传递给主调函数，此时发生复制构造。
  - 这种情况如何避免不必要的复制？

左值与右值

- 左值是位于赋值运算左侧的对象变量，右值是位于赋值运算右侧的值，右值不依附于对象。
  - int i = 42; // i 是左值，42是右值
  - int foolbar();
  - int j = foolbar(); //foolbar()是右值
- 左值和右值之间的转换
  - int i = 5, j = 6; // i 和 j 是左值
  - int k = i + j; // 自动转化为右值表达式

右值引用

- 对持久存在变量的引用称为左值引用，用&表示（即第3章引用类型）
- 对短暂存在可被移动的右值的引用称之为右值引用，用&&表示
  - float n = 6;
  - float &lr_n = n; //左值引用
  - float &&rr_n = n; //错误，右值引用不能绑定到左值
  - float &&rr_n = n * n; //右值表达式绑定到右值引用
- 通过标准库<utility>中的move函数可将左值对象移动为右值
  - float n = 10;
  - float &&rr_n = std::move(n); //将n转化为右值
  - 使用move函数承诺除对n重新赋值或销毁外，不以rr_n以外方式使用

移动构造函数

基于右值引用，移动构造函数通过移动数据方式构造新对象，与复制构造函数类似，移动构造函数参数为该类对象的右值引用。示例如下

```
#include<utility>
class astring {
public :
   std::string s;
   astring (astring&& o) noexcept: s(std::move(o.s)) //显式移动所有成员
   { 函数体 }
}
```

- 移动构造函数不分配新内存，理论上不会报错，为配合异常捕获机制，需声明noexcept表明不会抛出异常（将于12章异常处理介绍）
- 被移动的对象不应再使用，需要销毁或重新赋值

补充4_1 移动构造举例

```
#include <utility>
#include<string>
using namespace std;
class MyStr {
    public:
        string s;
        MyStr() :s("") {};//无参构造函数
        MyStr(string _s) :s(move(_s)) {};//有参构造函数
        MyStr(MyStr &str)//复制构造函数
        { //函数体略 }
        MyStr(MyStr &&str) noexcept:s(move(str.s)){}//移动构造函数，声明不会抛出异常
};
MyStr getString(){
    MyStr tmpStr;
    return tmpStr; //tmpStr为“将死之值”视为右值，首选调用移动构造函数
}
int main(){
    getString();
}
```

析构函数

例：构造函数和析构函数

```
#include <iostream>
using namespace std;
class Point {
public:
  Point(int xx,int yy);
  ~Point();
  //…其他函数原型
private:
  int x, y;
};
```

```
Point::Point(int xx,int yy)
{
  x = xx;
  y = yy;
}
Point::~Point() {
}
//…其他函数的实现略
```

析构函数

- 完成对象被删除前的一些清理工作。
- 在对象的生存期结束的时刻系统自动调用它，然后再释放此对象所属的空间。
- 如果程序中未声明析构函数，编译器将自动产生一个默认的析构函数，其函数体为空。

类的组合

这一节我们学习类的组合。

在制造业，我们一直在用部件组装的生产模式，在程序中是否也可以用这种方法呢？

C++语言支持这种部件组装的思想，就是类组合的机制：在定义一个新类时，可以用已有类的对象作为成员。

例4_4：类的组合，线段（Line）类

```
#include <iostream>
#include <cmath>
using namespace std;
class Point {//Point类定义
public:
        Point(int xx = 0, int yy = 0) {
                x = xx;
                y = yy;
        }
        Point(Point &p);
        int getX() { return x; }
        int getY() { return y; }
private:
        int x, y;
};
```

例4_4：类的组合，线段（Line）类

```
Point::Point(Point &p) {  //复制构造函数的实现
        x = p.x;
        y = p.y;
        cout << "Calling the copy constructor of Point" << endl;
}
```

例4_4：类的组合，线段（Line）类

```
//类的组合
class Line {  //Line类的定义
public:       //外部接口
        Line(Point xp1, Point xp2);
        Line(Line &l);
        double getLen() { return len; }
private:      //私有数据成员
        Point p1, p2;//Point类的对象p1,p2
        double len;
};
```

例4_4：类的组合，线段（Line）类

```
//组合类的构造函数
Line::Line(Point xp1, Point xp2) : p1(xp1), p2(xp2) {
        cout << "Calling constructor of Line" << endl;
        double x = static_cast<double>(p1.getX() - p2.getX());
        double y = static_cast<double>(p1.getY() - p2.getY());
        len = sqrt(x * x + y * y);
}
Line::Line (Line &l): p1(l.p1), p2(l.p2) {//组合类的复制构造函数
        cout << "Calling the copy constructor of Line" << endl;
        len = l.len;
}
```

例4_4：类的组合，线段（Line）类

```
//主函数
int main() {
        Point myp1(1, 1), myp2(4, 5);     //建立Point类的对象
        Line line(myp1, myp2);    //建立Line类的对象
        Line line2(line);       //利用复制构造函数建立一个新对象
        cout << "The length of the line is: ";
        cout << line.getLen() << endl;
        cout << "The length of the line2 is: ";
        cout << line2.getLen() << endl;
        return 0;
}
```

运行结果如下：

```
Calling the copy constructor of Point
Calling the copy constructor of Point
Calling the copy constructor of Point
Calling the copy constructor of Point
Calling the copy constructor of Point
Calling constructor of Line
Calling the copy constructor of Point
Calling the copy constructor of Point
Calling the copy constructor of Point
Calling the copy constructor of Line
The length of the line is: 5
The length of the line2 is: 5
```

组合的概念

- 类中的成员是另一个类的对象。
- 可以在已有抽象的基础上实现更复杂的抽象。

类组合的构造函数设计

- 原则：不仅要负责对本类中的基本类型成员数据初始化，也要对对象成员初始化。
- 声明形式：

```
类名::类名(对象成员所需的形参，本类成员形参)
        :对象1(参数)，对象2(参数)，……
{
//函数体其他语句
}
```

构造组合类对象时的初始化次序

- 首先对构造函数初始化列表中列出的成员（包括基本类型成员和对象成员）进行初始化，初始化次序是成员在类体中定义的次序。
  - 成员对象构造函数调用顺序：按对象成员的声明顺序，先声明者先构造。
  - 初始化列表中未出现的成员对象，调用用默认构造函数（即无形参的）初始化
- 处理完初始化列表之后，再执行构造函数的函数体。

前向引用声明

这一节我们来学习前向引用声明

例:

```
class B;  //前向引用声明
class A {
public:
  void f(B b);
};
class B {
public:
  void g(A a);
};
```

前向引用声明

- 类应该先声明，后使用
- 如果需要在某个类的声明之前，引用该类，则应进行前向引用声明。
- 前向引用声明只为程序引入一个标识符，但具体声明在其他地方。

例:

```
class Fred; //前向引用声明
class Barney {
    Fred x; //错误: 类Fred的声明尚不完善
};
class Fred {
    Barney y;
};
```

前向引用声明注意事项

- 使用前向引用声明虽然可以解决一些问题，但它并不是万能的。
- 在提供一个完整的类声明之前，不能声明该类的对象，也不能在内联成员函数中使用该类的对象。
- 当使用前向引用声明时，只能使用被声明的符号，而不能涉及类的任何细节。

结构体

看MOOC自学

结构体

- 结构体是一种特殊形态的类
  - 与类的唯一区别：类的缺省访问权限是private，结构体的缺省访问权限是public
  - 结构体存在的主要原因：与C语言保持兼容
- 什么时候用结构体而不用类
  - 定义主要用来保存数据、而没有什么操作的类型
  - 人们习惯将结构体的数据成员设为公有，因此这时用结构体更方便

结构体的定义

- 结构体定义

```
struct 结构体名称 {
    公有成员
protected:
    保护型成员
private:
    私有成员
};
```

结构体的初始化

- 如果一个结构体的全部数据成员都是公共成员，并且没有用户定义的构造函数，没有基类和虚函数（基类和虚函数将在后面的章节中介绍），这个结构体的变量可以用下面的语法形式赋初值
  - 类型名 变量名 = { 成员数据1初值, 成员数据2初值, ... };

例4-7用结构体表示学生的基本信息

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

struct Student {        //学生信息结构体
      int num;          //学号
      string name;      //姓名，字符串对象，将在第6章详细介绍
      char sex;         //性别
      int age;          //年龄
};
```

例4-7（续）

```
int main() {
    Student stu = { 97001, "Lin Lin", 'F', 19 };
    cout << "Num:  " << stu.num << endl;
    cout << "Name: " << stu.name << endl;
    cout << "Sex:  " << stu.sex << endl;
    cout << "Age:  " << stu.age << endl;
    return 0;
}
```

运行结果：

```
Num:  97001
Name: Lin Lin
Sex:  F
Age:  19
```

联合体

看MOOC自学

联合体

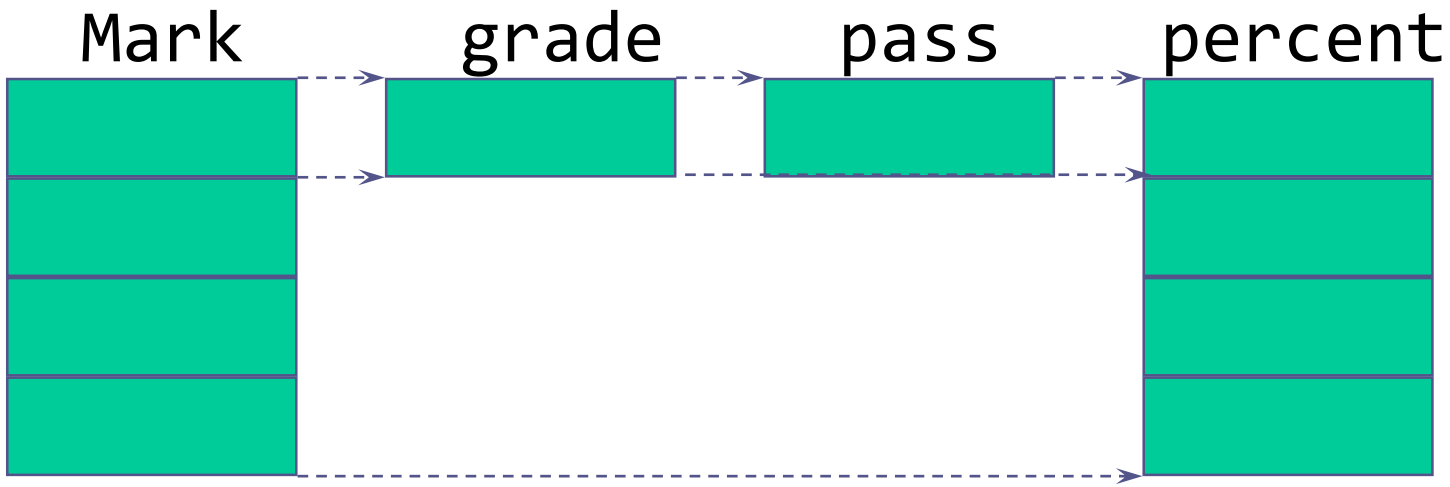
- 声明形式

```
union 联合体名称 {
    公有成员
protected:
    保护型成员
private:
    私有成员
};
```

- 特点：
  - 成员共用同一组内存单元
  - 任何两个成员不会同时有效

联合体的内存分配

```
union Mark {      //表示成绩的联合体
  char grade;     //等级制的成绩
  bool pass;      //只记是否通过课程的成绩
  int percent;    //百分制的成绩
};
```



无名联合

无名联合没有标记名，只是声明一个成员项的集合，这些成员项具有相同的内存地址，可以由成员项的名字直接访问。

例:

```
union {
  int i;
  float f;
}
```

在程序中可以这样使用:

```
i = 10;
f = 2.2;
```

例4-8使用联合体保存成绩信息，并且输出。

```
#include <string>
#include <iostream>
using namespace std;
class ExamInfo {
private:
     string name;     //课程名称
     enum { GRADE, PASS, PERCENTAGE } mode;//计分方式
     union {
          char grade;      //等级制的成绩
          bool pass; //只记是否通过课程的成绩
          int percent;     //百分制的成绩
     };
```

例4-8（续）

```
public:
    //三种构造函数，分别用等级、是否通过和百分初始化
    ExamInfo(string name, char grade)
        : name(name), mode(GRADE), grade(grade) { }
    ExamInfo(string name, bool pass)
        : name(name), mode(PASS), pass(pass) { }
    ExamInfo(string name, int percent)
        : name(name), mode(PERCENTAGE), percent(percent) { }
    void show();
}
```

例4-8（续）

```
void ExamInfo::show() {
     cout << name << ": ";
     switch (mode) {
       case GRADE: cout << grade;  break;
       case PASS: cout << (pass ? "PASS" : "FAIL"); break;
       case PERCENTAGE: cout << percent; break;
     }
     cout << endl;
}
```

例4-8（续）

```
int main() {
    ExamInfo course1("English", 'B');
    ExamInfo course2("Calculus", true);
    ExamInfo course3("C++ Programming", 85);
    course1.show();
    course2.show();
    course3.show();
    return 0;
}
```

运行结果:

```
English: B
Calculus: PASS
C++ Programming: 85
```

枚举类型

枚举类型

- 只要将需要的变量值一一列举出来，便构成了一个枚举类型。
- C++包含两种枚举类型：不限定作用域的枚举类型和限定作用域的枚举类。
- 不限定作用域枚举类型声明形式如下：
  - enum 枚举类型名 {变量值列表};
  - 例如：

```
enum Weekday
{SUN, MON, TUE, WED, THU, FRI, SAT};
```

- 关于限定作用域的enum类将在第4章和第5章详细介绍。

不限定作用域枚举类型说明

- 对枚举元素按常量处理，不能对它们赋值。例如，不能写：SUN = 0;
- 枚举元素具有默认值，它们依次为： 0,1,2,……。
- 也可以在声明时另行指定枚举元素的值，如：
  - enum Weekday{SUN=7,MON=1,TUE,WED, THU,FRI,SAT};
- 枚举值可以进行关系运算。
- 整数值不能直接赋给枚举变量，如需要将整数赋值给枚举变量，应进行强制类型转换。

例4-9

- 设某次体育比赛的结果有四种可能：胜（WIN）、负（LOSE）、平局（TIE）、比赛取消（CANCEL），编写程序顺序输出这四种情况。
  - 分析：由于比赛结果只有四种可能，所以可以声明一个枚举类型，声明一个枚举类型的变量来存放比赛结果。

例4-9

```
#include <iostream>
using namespace std;
enum GameResult {WIN, LOSE, TIE, CANCEL};
int main() {
    GameResult result;
    enum GameResult omit = CANCEL;
    for (int count = WIN; count <= CANCEL; count++) {
      result = GameResult(count);
      if (result == omit)
        cout << "The game was cancelled" << endl;
      else {
        cout << "The game was played ";
        if (result == WIN)      cout << "and we won!";
        if (result == LOSE)       cout << "and we lost.";
        cout << endl;
      }
    }
    return 0;
}
```

运行结果

```
The game was played and we won!
The game was played and we lost.
The game was played
The game was cancelled
```

枚举类——强类型枚举

- 定义语法形式

  enum class 枚举类型名: 底层类型 {枚举值列表};

- 例:

```
enum class Type { General, Light, Medium, Heavy};
enum class Type: char { General, Light, Medium, Heavy};
enum class Category { General=1, Pistol, MachineGun, Cannon};
```

枚举类的优势

- 强作用域，其作用域限制在枚举类中。
  - 例：使用Type的枚举值General：
    Type::General
- 转换限制，枚举类对象不可以与整型隐式地互相转换。
- 可以指定底层类型
  - 例：
    enum class Type: char { General, Light, Medium, Heavy};

枚举类举例

```
#include<iostream>
using namespace std;
enum class Side{ Right, Left };
enum class Thing{ Wrong, Right };  //不冲突
int main()
{
    Side s = Side::Right;
    Thing w = Thing::Wrong;
    cout << (s == w) << endl;  //编译错误，无法直接比较不同枚举类
    return 0;
}
```

UML简介

看MOOC自学

UML简介

- UML（Unified Modeling Language）语言是一种可视化的的面向对象建模语言。
- UML有三个基本的部分
  - 事物（Things）
    UML中重要的组成部分，在模型中属于最静态的部分，代表概念上的或物理上的元素
  - 关系（Relationships）
    关系把事物紧密联系在一起
  - 图（Diagrams）
    图是很多有相互相关的事物的组

UML类图

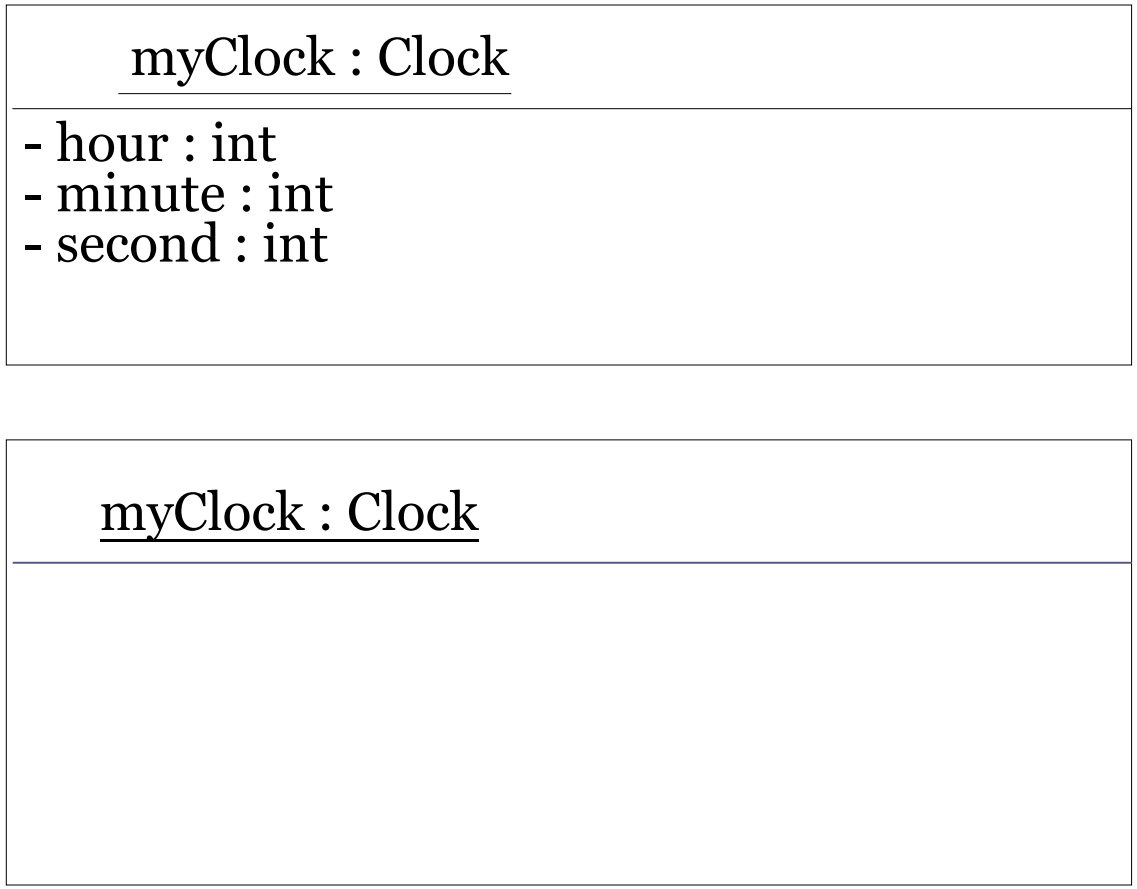
- 举例：Clock类的完整表示

Clock
- hour : int - minute : int - second : int
+ showTime() : void + setTime(newH:int=0,newM:int=0,newS:int=0):void

- Clock类的简洁表示

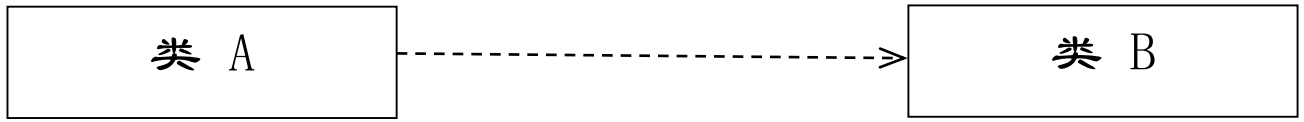
Clock

对象图



几种关系的图形标识

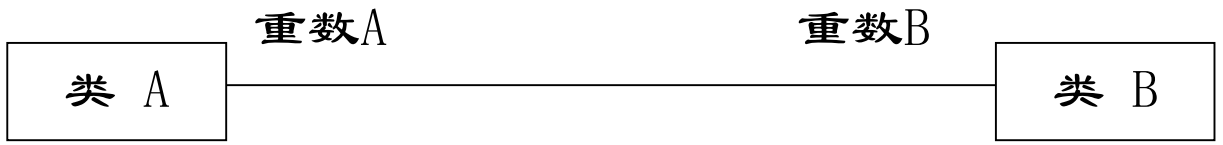
- 依赖关系



图中的“类A”是源，“类B”是目标，表示“类A”使用了“类B”，或称“类A”依赖“类B”

几种关系的图形标识

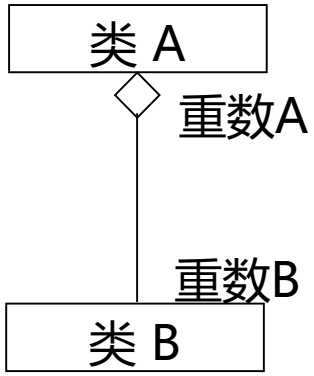
- 作用关系——关联



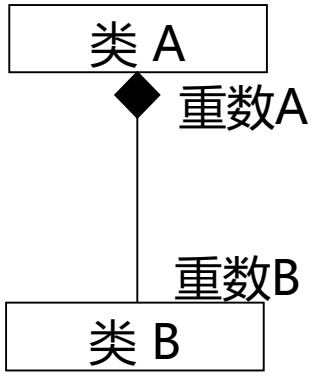
图中的“重数A”决定了类B的每个对象与类A的多少个对象发生作用，同样“重数B”决定了类A的每个对象与类B的多少个对象发生作用。

几种关系的图形标识

- 包含关系——聚集和组合



共享聚集



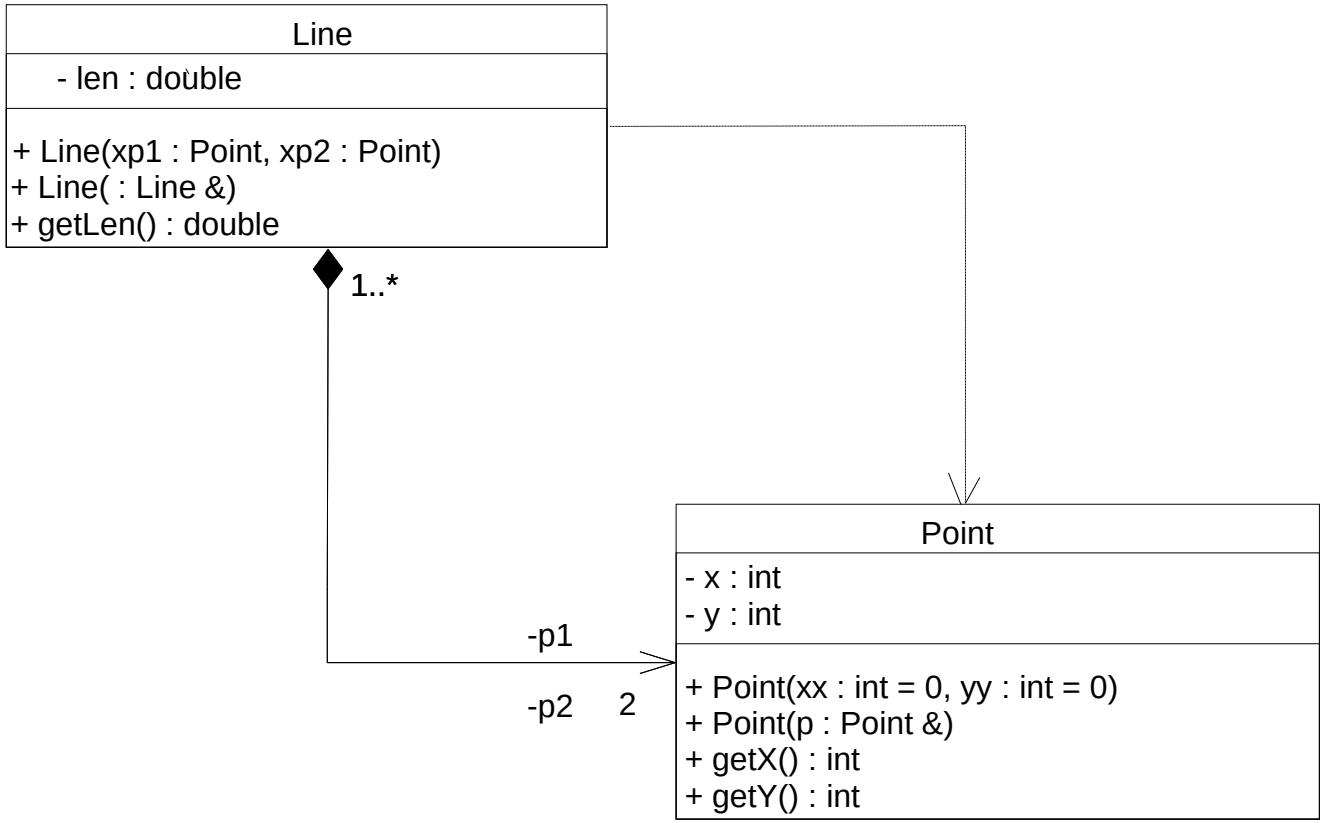
组成聚集（组合）

聚集表示类之间的关系是整体与部分的关系，“包含”、“组成”、“分为……部分”等都是聚集关系。

共享聚集：部分可以参加多个整体；

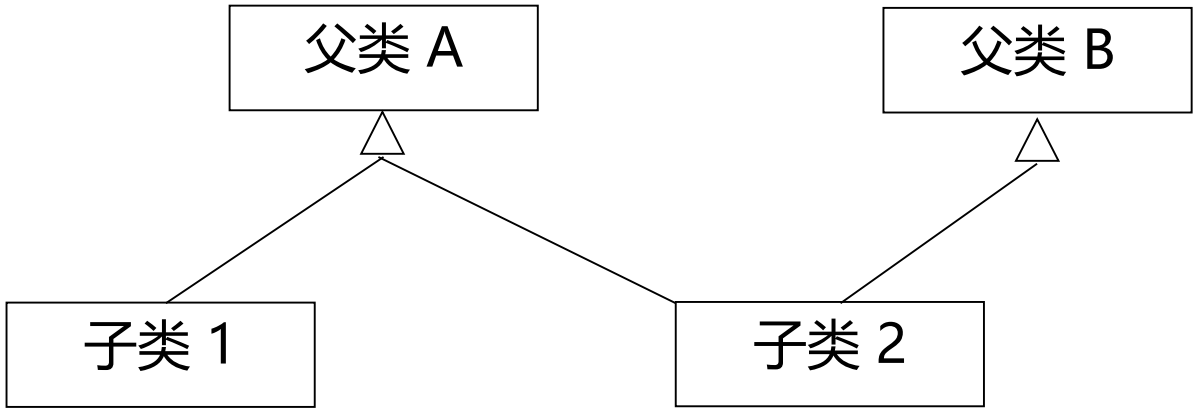
组成聚集(组合)：整体拥有各个部分，整体与部分共存，如果整体不存在了，那么部分也就不存在了。

例4-5 采用UML方法来描述例4-4中Line类和Point类的关系



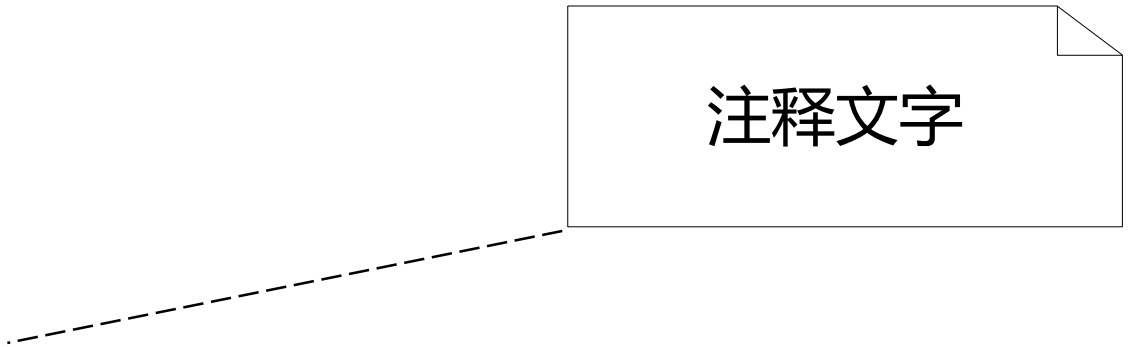
几种关系的图形标识

继承关系——泛化

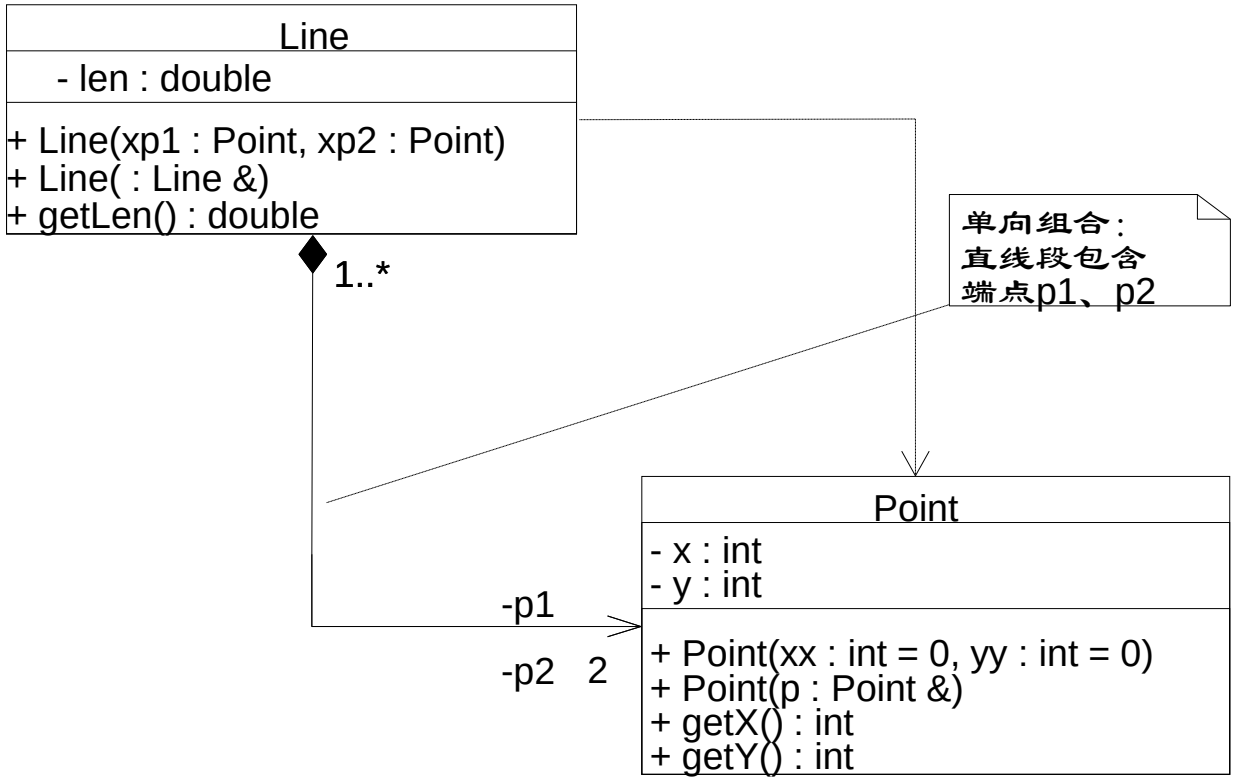


注释

- 在UML图形上，注释表示为带有褶角的矩形，然后用虚线连接到UML的其他元素上，它是一种用于在图中附加文字注释的机制。



例4-6 带有注释的Line类和Point类关系的描述



小结

- 主要内容
  - 面向对象的基本概念、类和对象的声明、构造函数、析构函数、内联成员函数、复制构造函数、类的组合
- 达到的目标
  - 掌握面向对象的基本概念；
  - 掌握类设计的思想、类和对象声明的语法；
  - 理解构造函数、复制构造函数和析构函数的作用和调用过程，掌握相关的语法；
  - 理解内联成员函数的作用，掌握相关语法；
  - 理解类的组合在面向对象设计中的意义，掌握类组合的语法。

小结

- 本周作业
  - 复习巩固语法，自己运行和观察本章所有例题——无需提交；
  - SPOC：视频后面的小题、第四章最后的编程题；
  - 学生用书上面的“实验四”，“实验二”结构体那一题。