C++语言程序设计

第四章 类与对象

中国科大 黄章进

本章主要内容

- 面向对象的思想
- OOP的基本特点
- 类概念和声明
- 对象
- 构造函数
- 析构函数
- 内联成员函数
- 拷贝构造函数
- 类的组合
- 结构体与联合体
- 深度探索

面向对象的思想

回顾：面向过程的设计方法

- 重点：
  - 如何实现的细节和过程，将数据与函数分开。
- 形式：
  - 主模块+若干个子模块（main()+子函数）。
- 特点：
  - 自顶向下，逐步求精——功能分解。
- 缺点：
  - 效率低，程序的可重用性差。

面向对象的思想

面向对象的方法

- 目的：
  - 实现软件设计的产业化。
- 观点：
  - 自然界是由实体（对象）所组成。
- 程序设计方法：
  - 使用面向对象的观点来描述模仿并处理现实问题。
- 要求：
  - 高度概括、分类、和抽象。

OOP的基本特点

抽象

抽象是对具体对象（问题）进行概括，抽出这一类对象的公共性质并加以描述的过程。

- 先注意问题的本质及描述，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）。
- 行为抽象：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：通过类的声明。

抽象实例——钟表

- 数据抽象：

  int hour,int minute,int second

- 行为抽象：

  setTime(),showTime()

抽象实例——钟表类


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

抽象实例——人

- 数据抽象：

  string name, string gender, int age, int id

- 行为抽象：

  生物属性角度：

  getCloth(), eat(), walk(),…

  社会属性角度：

  work(), promote() ,…

OOP的基本特点

封装

将抽象出的数据成员、代码成员相结合，将它们视为一个整体。

- 目的是曾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{}

封装

- 实例：

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

外部接口

特定的访问权限

边界

继承与派生

OOP的基本特点

是C++中支持层次分类的一种机制，允许程序员在保持原有类特性的基础上，进行更具体、更详细的说明。

实现：声明派生类——见第7章

多态性

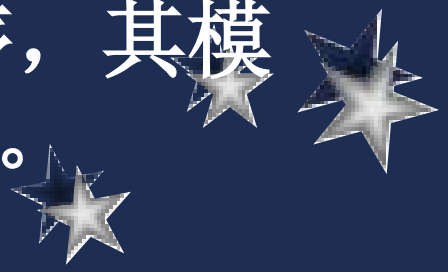
OOP的基本特点

- 多态：一段程序能处理多种类型对象的能力
  - 强制多态：数据类型转换（隐式或显式）
  - 重载多态：函数重载，运算符重载（第8章）
  - 包含多态：虚函数（第8章）
  - 类型参数化多态：函数模板，类模板（第9章）
- 目的：达到行为标识统一，减少程序中标识符的个数。

C++中的类

类和对象

- 类是具有相同属性和行为的一组对象的集合，它为属于该类的全部对象提供了统一的抽象描述，其内部包括属性和行为两个主要部分。
- 利用类可以实现数据的封装、隐藏、继承与派生。
- 利用类易于编写大型复杂程序，其模块化程度比C中采用函数更高。



类的声明形式

类是一种用户自定义类型，声明形式：

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

公有类型成员

类和对象

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。

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。

保护类型

类和对象

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，第七章讲。

类的成员

类和对象

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

成员函数

成员数据

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

成员数据

- 与一般的变量声明相同，但需要将它放在类的声明体中。

成员函数

- 在类中说明原型，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形参值的函数

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。

内联成员函数举例(一)

类和对象

```
class Point {
 public:
    void init(int initX, int initY) {
      x = initX;
      y = initY;
    }
    int getX() { return x; }
    int getY() { return y; }
 private:
    int x, y;
};
```

内联成员函数举例(二)

类和对象

```
class Point {
 public:
    void init(int initX, int initY);
    int getX();
    int getY();
 private:
    int x, y;
};
```

```
inline void Point::
            init(int initX,int initY) {
    x = initX;
    y = initY;
}

inline int Point::getX() {
    return x;
}

inline int Point::GetY() {
    return y;
}
```

对象

- 类的对象是该类的某一特定实体，即类类型的变量。
- 声明形式：

  类名　对象名;

- 例：

  Clock　myClock;

类中成员的访问方式

- 类中成员互访
  - 直接使用成员名
- 类外访问
  - 使用“对象名.成员名”方式访问 public 属性的成员
    - 对象名.数据成员名
    - 对象名.函数成员名(参数表)

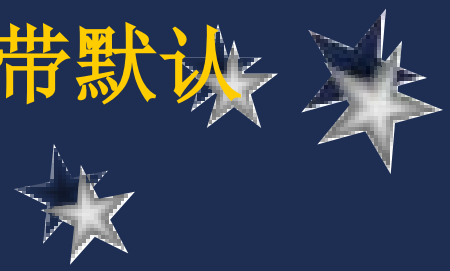
例4-1类的应用举例

类和对象

```
#include<iostream>
using namespace std;
class Clock {
        ......//类的声明略
};
//......类的实现略
int main() {
   Clock myClock;
   myClock.setTime(8, 30, 30);
   myClock.showTime();
   return 0;
}
```

构造函数

- 构造函数的作用是在对象被创建时使用特定的值构造对象，或者说将对象初始化为一个特定的状态。
- 在对象创建时由系统自动调用。
- 如果程序中未声明，则系统自动产生出一个隐含的参数列表为空的构造函数
- 允许为内联函数、重载函数、带默认形参值的函数



构造函数举例


```
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;
};
```

构造函数的实现：

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

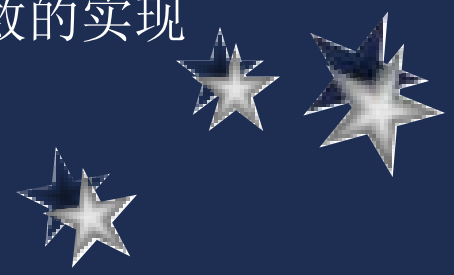
建立对象时构造函数的作用：

```
int main() {
   Clock c(0,0,0); //隐含调用构造函数，将初始值作为实参。
   c.showTime();
   return 0;
}
```

拷贝构造函数

拷贝构造函数是一种特殊的构造函数，其形参为本类对象的引用。

```
class 类名 {
public :
    类名（形参）; //构造函数
    类名（类名 &对象名）; //拷贝构造函数
        ...
};
类名::类（类名 &对象名）//拷贝构造函数的实现
{    函数体    }
```



拷贝构造函数(例4-2)


```
class Point {
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;
};
```

```
Point::Point (Point& p) {
  x = p.x;
  y = p.y;
  cout << "Calling the copy constructor "
     << endl;
}
```

拷贝构造函数(例4-2)

- 当用类的一个对象去初始化该类的另一个对象时系统自动调用拷贝构造函数实现拷贝赋值。

```
int main() {
    Point a(1,2);
    Point b = a; //拷贝构造函数被调用
    cout << b.getX() << endl;
}
```

拷贝构造函数(例4-2)

- 若函数的形参为类对象，调用函数时，实参赋值给形参，系统自动调用拷贝构造函数。例如：

```
void fun1(Point p) {
  cout << p.getX() << endl;
}
int main() {
  Point a(1, 2);
  fun1(a); //调用拷贝构造函数
  return 0;
}
```

值传递才调用拷贝构造函数，引用传递不调用

拷贝构造函数(例4-2)

- 当函数的返回值是类对象时，系统自动调用拷贝构造函数。例如：

```
Point fun2() {
  Point a(1, 2);
  return a; //调用拷贝构造函数
}
int main() {
  Point b;
  b = fun2();
  return 0;
}
```

隐含的拷贝构造函数

如果程序员没有为类声明拷贝构造函数，则编译器自己生成一个隐含的拷贝构造函数。

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。

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隐含的析构函数。

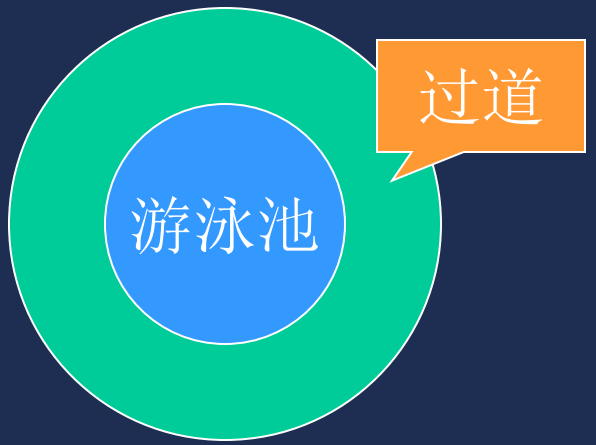
构造函数和析构函数举例


```
#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
```

类的应用举例(例4-3)

一圆形游泳池如图所示，现在需在其周围建一圆形过道，并在其四周围上栅栏。栅栏价格为35元/米，过道造价为20元/平方米。过道宽度为3米，游泳池半径由键盘输入。要求编程计算并输出过道和栅栏的造价。



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

const float PI = 3.141593;                    //给出p的值
const float FENCE_PRICE = 35;  //栅栏的单价
const float CONCRETE_PRICE = 20;//过道水泥单价

class Circle { //声明定义类Circle 及其数据和方法

public:          //外部接口
  Circle(float r);          //构造函数
  float circumference();  //计算圆的周长
  float area();             //计算圆的面积

private:  //私有数据成员
  float radius;             //圆半径
};
```

```
//类的实现

//构造函数初始化数据成员radius
Circle::Circle(float r) {
  radius = r;
}

//计算圆的周长
float Circle::circumference() {
    return 2 * PI * radius;
}

//计算圆的面积
float Circle::area() {
    return PI * radius * radius;
}
```

```
int main () {
  float radius;
  cout << "Enter the radius of the pool: ";
  // 提示用户输入半径
  cin >> radius;

  Circle pool(radius);          //游泳池边界
  Circle poolRim(radius + 3);  //栅栏

    //计算栅栏造价并输出
    float fenceCost = poolRim.circumference() *
  FENCE_PRICE;
    cout << "Fencing Cost is $" << fenceCost <<
  endl;
```

```
    //计算过道造价并输出
    float concreteCost = (poolRim.area() - pool.area())
    * CONCRETE_PRICE;
    cout << "Concrete Cost is $" << concreteCost << endl;

    return 0;
}
```

运行结果

```
Enter the radius of the pool: 10
Fencing Cost is $2858.85
Concrete Cost is $4335.4
```

组合类的概念

类的组合

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

举例

类的组合

```
class Point {
private:
   float x, y; //点的坐标
public:
   Point(float h, float v); //构造函数
   float getX(); //取X坐标
   float getY(); //取Y坐标
   void draw(); //在(x,y)处画点
};
//...函数的实现略
```

```
class Line {
private:
  Point p1, p2; //线段的两个端点
public:
  Line(Point a, Point b); //构造函数
  void draw(void); //画出线段
};
//...函数的实现略
```

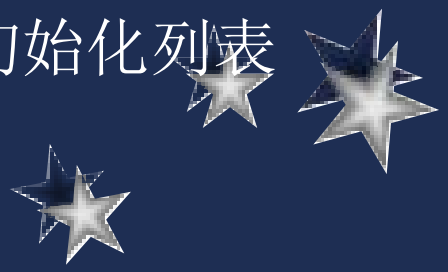
组合类的构造函数设计

类的组合

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

```
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)，……
{   本类初始化   }
```

初始化列表



组合类的构造函数调用

类的组合

- 构造函数调用顺序：先调用内嵌对象的构造函数（按内嵌时的声明顺序，先声明者先构造）。然后调用本类的构造函数。（析构函数的调用顺序相反）
- 初始化列表中未出现的内嵌对象，用默认构造函数（即无形参的）初始化
- 编译器自动生成的隐含默认构造函数中，内嵌对象全部用默认构造函数初始化

组合类的构造函数调用

类的组合

- 编译器自动生成的隐含的复制构造函数，自动调用内嵌对象的复制构造函数，为各个内嵌对象初始化
- 为组合类编写复制构造函数，需要为内嵌成员对象的复制构造函数传递参数

类的组合举例（二）


```
class Part { //部件类
public:
   Part();
   Part(int i);
   ~Part();
   void Print();
private:
   int val;
};
```

```
class Whole {
public:
  Whole();
  Whole(int i, int j, int k);
  ~Whole();
  void Print();
private:
  Part one;
  Part two;
  int date;
};
```

```
Whole::Whole() {
  date=0;
}
Whole::Whole(int i, int j, int k)
          : two(i), one(j), date(k)
{}

//...其他函数的实现略
```

例4-4：线段类

```
#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;
};

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

```
//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;
};

//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);
}

//组合类的拷贝构造函数
Line::Line (Line &l): p1(l.p1), p2(l.p2) {
   cout << "Calling the copy constructor of Line" << endl;
   len = l.len;
}
```

```
//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);    //利用拷贝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;
}
```

前向引用声明

类的组合

- 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。

前向引用声明举例


```
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。需要注意的是，尽管使用了前向引用声明，但是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。请看下面的程序段：

```
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


```
class Fred;      //前向引用声明

class Barney {
public:
  ……
  void method() {
    x.yabbaDabbaDo();   //错误：Fred类的对象在定义之前被使用
  }
 private:
  Fred &x;//正确，经过前向引用声明，可以声明Fred类的对象引用
};

class Fred {
public:
  void yabbaDabbaDo();
private:
  Barney &y;
};
```

前向引用声明注意事项

- 应该记住：当你使用前向引用声明时，你只能使用被声明的符号，而不能涉及类的任何细节。

UML简介

- UML(统一建模语言)语言是一种可视化的的面向对象建模语言。
  - 由OMG（对象管理组织）于1997年开始推行
- http://www.uml.org/
- UML建模工具：
  - ArgoUML

http://www.argouml-users.net/index.php

UML简介

- 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中有4种类型的事物

- 结构事物（Structural things）
- 动作事物（Behavioral things）
- 分组事物（Grouping things）
- 注释事物（Annotational things）

UML中的关系

- 依赖（Dependencies）
- 关联（Association）
- 泛化（Generalization）
- 实现（Realization）

UML图形标识

UML中的9种图

- 类图（Class diagram）
- 对象图（Object diagram）
- 用例图（Use case diagram）
- 顺序图（Sequence diagram）
- 协作图（Collaboration diagram）
- 状态图（Statechart diagram）
- 活动图（Activity diagram）
- 组件图（Component diagram）
- 实施图（Deployment diagram）

UML图形标识

类图

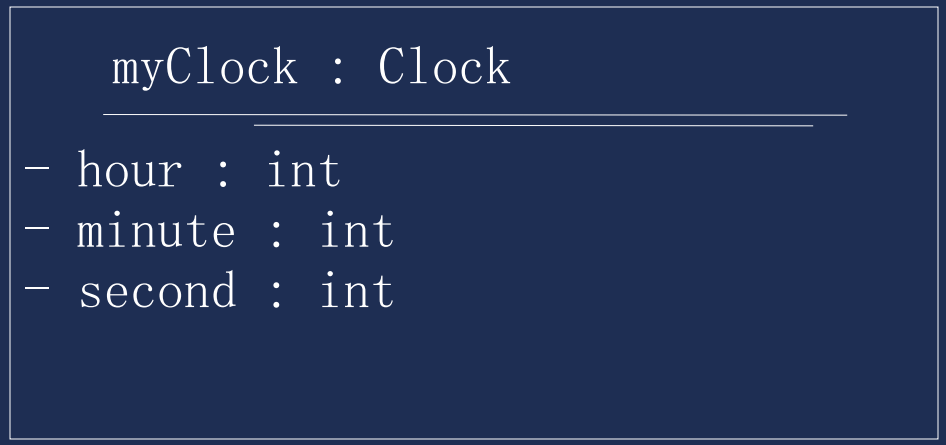
- 举例：Clock类的完整表示

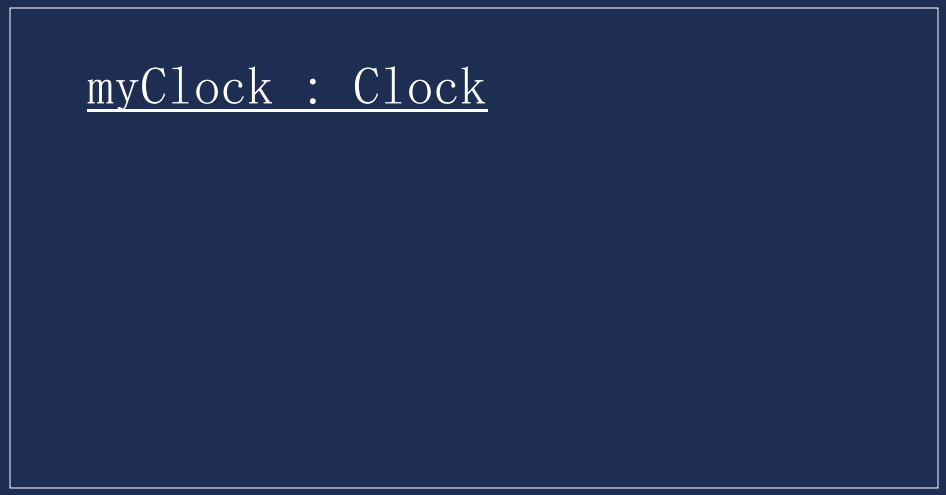
Clock
- hour : int - minute : int - second : int
+ showTime() : void + setTime(newH:int=0,newM:int=0,newS:int=0):void

- Clock类的简洁表示

对象图

UML图形标识





类与对象关系的图形标识

UML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”

类与对象关系的图形标识

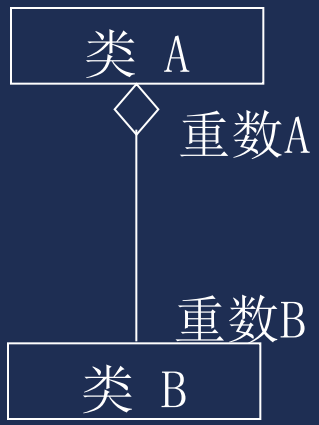
- 作用关系——关联

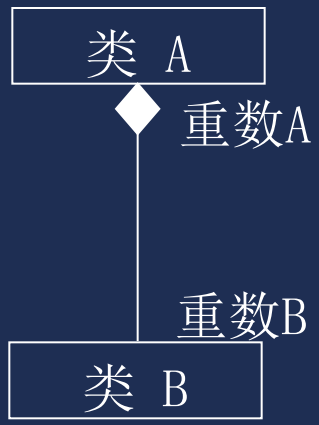
图中的“重数A”决定了类B的每个对象与类A的多少个对象发生作用，同样“重数B”决定了类A的每个对象与类B的多少个对象发生作用。

类与对象关系的图形标识

UML图形标识

- 包含关系——聚集和组合



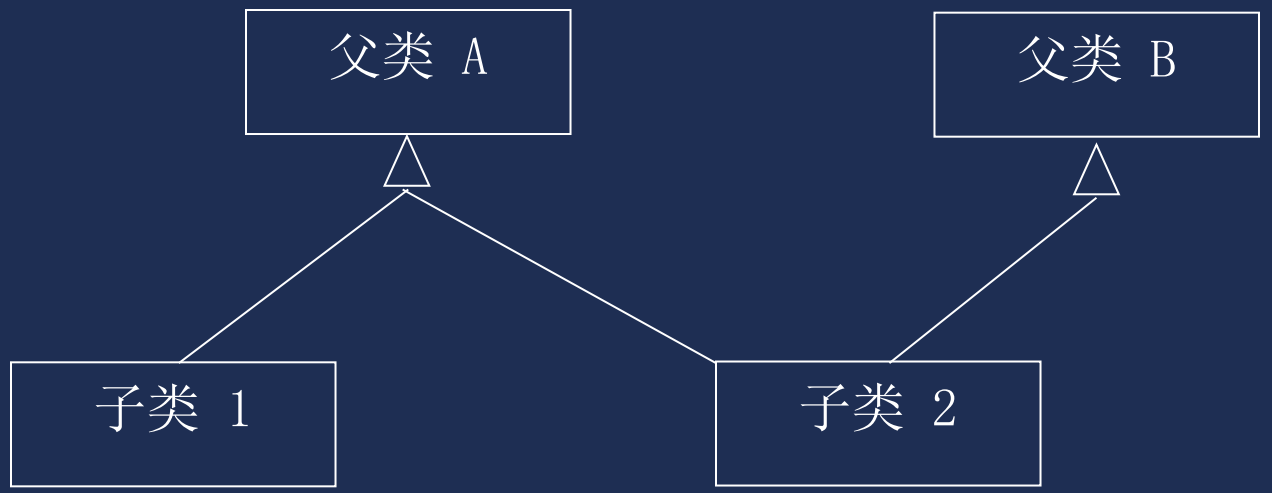


聚集表示类之间的关系是整体与部分的关系，“包含”、“组成”、“分为……部分”等都是聚集关系。

类与对象关系的图形标识

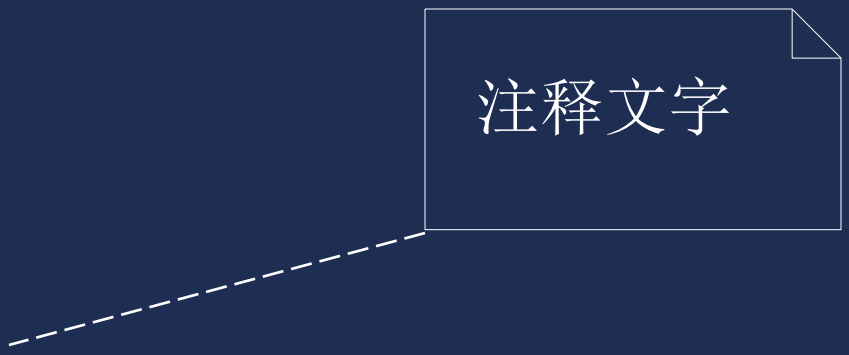
UML图形标识

- 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。



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

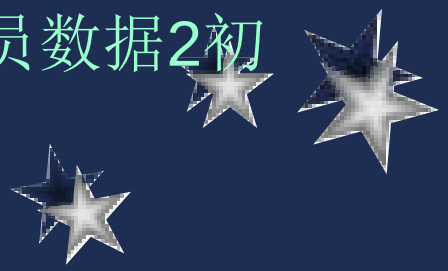
结构体的定义和初始化

- 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
};
```

- 一些结构体变量的初始化可以用以下形式

```
类型名 变量名 = { 成员数据1初值, 成员数据2初值, …… };
```



结构体举例(例4-7)


```
#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
};
```

结构体举例(例4-7)


```
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

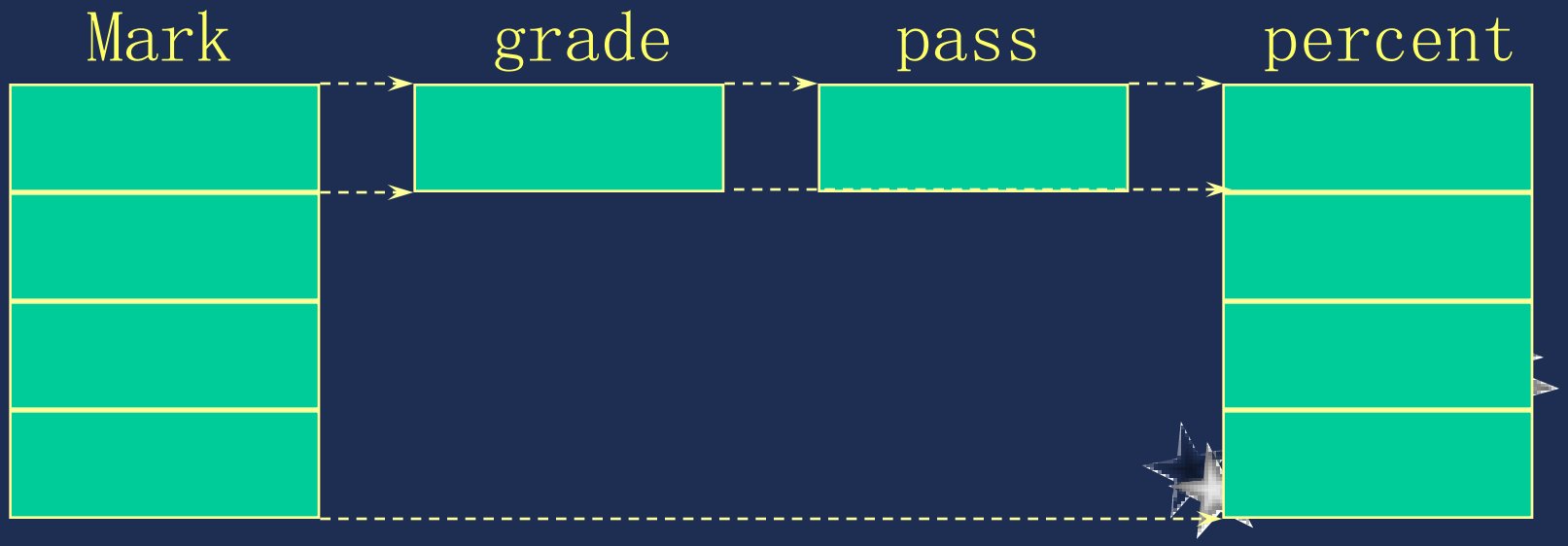
- 声明形式

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

- 特点：
  - 成员共用相同的内存单元
  - 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;
```

联合体举例(例4-8)


```
#include <string>
#include <iostream>
using namespace std;
class ExamInfo {
private:
   string name;          //课程名称
   enum { GRADE, PASS, PERCENTAGE } mode;//采用何种计分
   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
   };
```

联合体举例(例4-8)

结构体与联合体

```
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();
}
```

联合体举例(例4-8)


```
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;
}
```

联合体举例(例4-8)


```
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
```

位域

深度探索

- 位域的声明形式
  - 数据类型说明符　成员名：位数;
- 位域的作用
  - 通过“打包”，使类的不同成员共享相同的字节，从而节省存储空间。
- 注意事项
  - 具体的打包方式，因编译器而异;
  - 只有bool、char、int和枚举类型的成员，允许定义为位域;
  - 节省空间，但可能增加时间开销。

例4-10

深度探索

- 设计一个结构体存储学生的成绩信息，需要包括学号、年级和成绩三项内容，学号的范围是0到99,999,999，年级分为freshman、sophomore、junior、senior四种，成绩包括A、B、C、D四个等级。

例4-10

深度探索

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

enum Level { FRESHMAN, SOPHOMORE, JUNIOR, SENIOR };
enum Grade { A, B, C, D };
class Student {
public:
    Student(unsigned number, Level level, Grade grade)
          : number(number), level(level), grade(grade) { }
    void show();
private:
    unsigned number : 27;
    Level level : 2;
    Grade grade : 2;
};
```

例4-10

深度探索

```
void Student::show() {
    cout << "Number:    " << number << endl;
    cout << "Level:     ";
    switch (level) {
           case FRESHMAN:  cout << "freshman"; break;
           case SOPHOMORE: cout << "sophomore"; break;
           case JUNIOR:    cout << "junior"; break;
           case SENIOR:    cout << "senior"; break;
    }
    cout << endl;
    cout << "Grade:     ";
    switch (grade) {
           case A: cout << "A"; break;
           case B: cout << "B"; break;
           case C: cout << "C"; break;
           case D: cout << "D"; break;
    }
    cout << endl;
}
```

例4-10

深度探索

```
int main() {
    Student s(12345678, SOPHOMORE, B);
    cout << "Size of Student: " << sizeof(Student) <<
    endl;
    s.show();
    return 0;
}
```

运行结果：（运行结果第一行会因编译环境的不同而有所差异）

```
Size of Student: 4
Number:    12345678
Level:     sophomore
Grade:     B
```

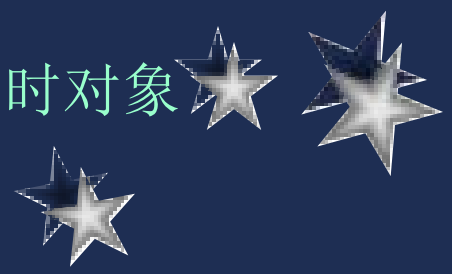
临时对象与类型转换

深度探索

（例4-2中Point构造函数：Point(int xx = 0, int yy = 0)）

- 临时对象的显式创建
  - 可以直接调用类的构造函数显式创建临时对象

  例：Line x(Point(1, 1), Point(4, 5));
  - 临时对象到表达式执行完毕后即销毁
- 单参数构造函数可以设立类型转换
  - Point(1)表示创建一个临时对象，同时也表示显式类型转换
  - 与Point(1)等价的形式：
    - (Point) 1
    - static_cast<Point>(1)
  - 无论形式为何，执行转换时都会创建临时对象



隐含转换

深度探索

- 由构造函数确立的类型转换，可以隐含执行
  - 例：Line x(1, 4);
  - 效果：构造以(1,0)和(4,0)两坐标为端点的线段，这里Point的构造函数被隐含调用
- 避免隐含转换的发生
  - 在构造函数中使用explicit关键字，explicit要写在类定义中的构造函数前

例4-11

深度探索

```
#include <iostream>
using namespace std;
class Complex { //复数类
private:
  float real, imag;//复数的实部和虚部
public:
  //构造函数，可以当作隐式类型转换使用
  Complex(float real = 0, float imag = 0) : real(real), imag(imag) { }
  Complex add(Complex c) {   //复数加法，生成临时对象并返回
    return Complex(real + c.real, imag + c.imag);
  }
  Complex sub(Complex c) {   //复数减法，生成临时对象并返回
    return Complex(real - c.real, imag - c.imag);
  }
  Complex mul(Complex c) {   //复数乘法，生成临时对象并返回
    return Complex(real * c.real - imag * c.imag, real * c.imag + imag * c.real);
  }
```

例4-11

深度探索

```
  void show() {  //显示复数
    if (imag >= 0)
      cout << real << " + " << imag << 'i' << endl;
    else
      cout << real << " - " << -imag << 'i' << endl;
  }
};
int main() {
    Complex z(1, 2);                    //z = 1 + 2i
    z.add(Complex(3, 4)).show();        //Complex(3, 4)是临时对象
    static_cast<Complex>(5).sub(z).show(); //输出5 - z，使用了显示类型转换
    z.mul(-3.0f).show(); //输出z * (-3)，使用了隐含类型转换
    return 0;
}
```

运行结果：

```
4 + 6i
4 - 2i
-3 - 6i
```

成员函数的调用

深度探索

- 成员函数调用的实现机制
  - 问题的关键：如何传递调用的目的对象
  - 解决办法：把目的对象的引用当作参数传递

```
void Clock::setTime(int
newH, int newM, int newS) {
  hour = newH;
  minute = newM;
  second = newS;
}
```

```
void Clock_setTime(_Clock &_this,
int newH, int newM, int newS) {
  _this.hour = newH;
  _this.minute = newM;
  _this.second = newS;
}
```

```
myClock.setTime(8, 30, 30);
```

```
Clock_setTime(myClock, 8, 30, 30);
```

构造函数的调用

深度探索

```
Line::Line(Point xp1, Point xp2)
    : p1(xp1), p2(xp2) {
  double x = p1.getX() - p2.getX();
  double y = p1.getY() - p2.getY();
  len = sqrt(x * x + y * y);
}
```

```
void Line_Line(_Line &_this, _Point
xp1, _Point xp2) {
  Point_Point(_this.p1, xp1);
  Point_Point(_this.p2, xp2);
  double x = Point_getX(_this.p1) -
Point_getX(_this.p2);
  double y = Point_getY(_this.p1) -
Point_getY(_this.p2);
  _this.len = sqrt(x * x + y * y);
}
```

```
Line line(myp1, myp2);
```

```
_Line line;
Line_Line(line, myp1,
myp2);
```

先分配空间，再调用构造函数

对象作为参数的传递方式

深度探索

- 对象参数的传递方式
  - 通过运行栈来传递
  - 主调函数调用拷贝构造函数，在运行栈的传参区域上创建对象
  - 被调函数可以读取传参区域上的对象
- 有时对拷贝构造函数的调用可以省去

  例： z.add(Complex(3, 4))
  - 直接调用构造函数Complex(float, float)，在运行栈的传参区域上建立对象

对象作为返回值的传递方式

深度探索

- 传递方式
  - 在主调函数中创建临时对象
  - 主调函数把该对象地址（引用）传递给被调函数
  - 被调函数返回时，在该地址上执行拷贝构造

```
Point fun2() {
  Point a(1, 2);
  return a;
}
```

```
void fun2(_Point &result) {
  _Point a;
  Point_Point(a, 1, 2);
  Point_Point(result, a);
}
```

```
b = fun2();
```

```
_Point temp;
fun2(temp);
b = temp;
```

对象作为返回值的传递方式

深度探索

- 有时返回时可以不调用拷贝构造函数

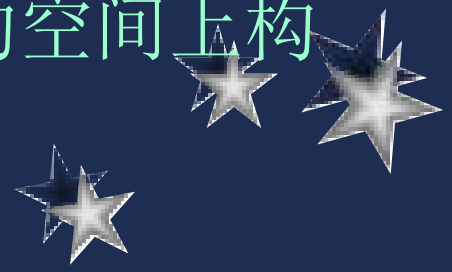
  例：return Point(1, 2);

  – 直接调用构造函数Point(int, int)，生成返回的对象

- 有时主调函数中可以不建立临时对象

  例：Point p = fun2();

  – 先为p申请空间，调用fun2()前传递p的地址，这样在返回时可直接在p的空间上构造返回对象



小结与复习建议

- 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、拷贝构造函数、类的组合
- 达到的目标
  - 学会将一段功能相对独立的程序写成一个函数，为下一章学习类和对象打好必要的基础。