C++语言程序设计

第四章　类与对象

信息工程学院　王红平
Email：cugwhp@qq.com

本章主要内容

- 面向对象的思想
- OOP的基本特点
- 类和对象
- 构造函数和析构函数
- 类的组合
- 结构体和联合体

面向对象的思想

回顾：面向过程的设计方法

- 重点：
  - 如何实现细节过程，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。

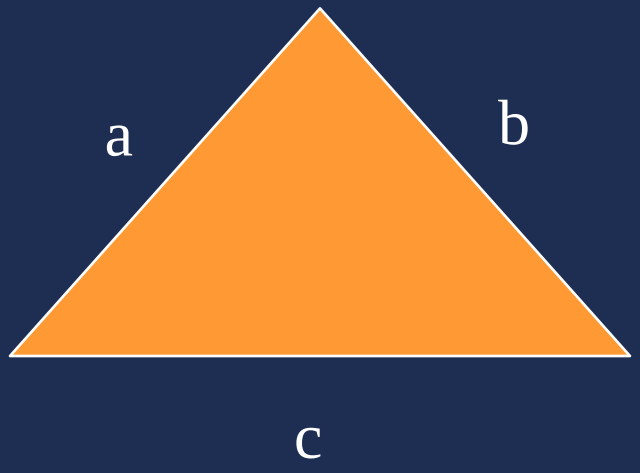
面向对象的基本概念
——对象

- 一般意义上的对象：
  - 是现实世界中一个实际存在的事物。
  - 可以是有形的（比如一辆汽车），也可以是无形的（比如一项计划）。
  - 是构成世界的一个独立单位，具有：
    - 静态特征：可以用某种数据来描述
    - 动态特征：对象所表现的行为或具有的功能

面向对象的基本概念
——对象

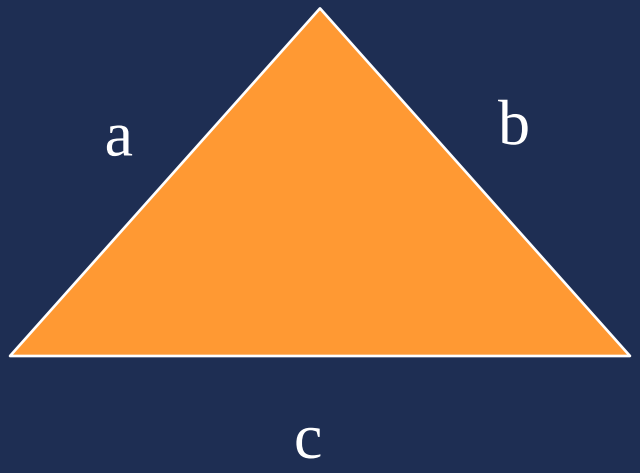
- 面向对象方法中的对象：
  - 是系统中用来描述客观事物的一个实体，它是用来构成系统的一个基本单位。对象由一组属性和一组行为构成。
  - 属性：用来描述对象静态特征的数据项。
  - 行为：用来描述对象动态特征的操作序列。

面向过程举例




```
double GetPerimeter(double a,double b,double c);
double GetArea(double a,double b,double c);
double  GetAngle(double a,double b,double c,int which);
……
```

面向对象举例



A

double a;double b;double c;
double GetPerimeter();
double GetArea();
double GetAngle(int which);
……

面向对象的方法

4.1面向对象程序设计的基本特点

- 抽象
- 封装
- 继承
- 多态

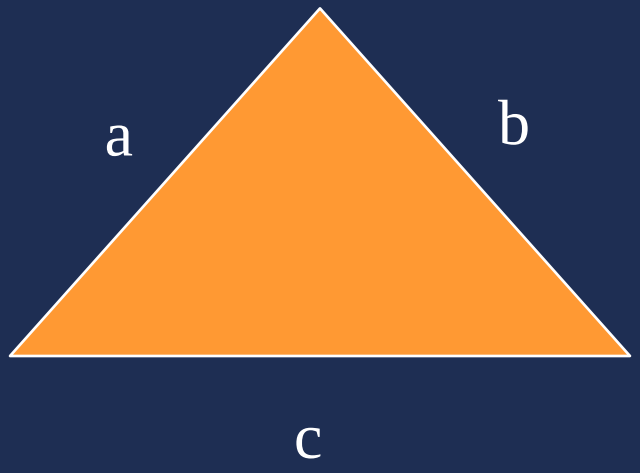
4.1.1抽象

OOP的基本特点

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）。
- 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：通过类的声明。

面向对象举例



A

double a;double b;double c;
double GetPerimeter();
double GetArea();
double GetAngle(int which);
……

抽象实例——钟表

- 数据抽象：

  int Hour，int Minute，int Second

- 代码抽象：

  SetTime()，ShowTime()

抽象实例——钟表类

OOP的基本特点

```
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：

  char name[20],char gender,int age,int id

- 代码抽象：

  生物属性角度：

  GetCloth(), Eat(), Step(),…

  社会属性角度：

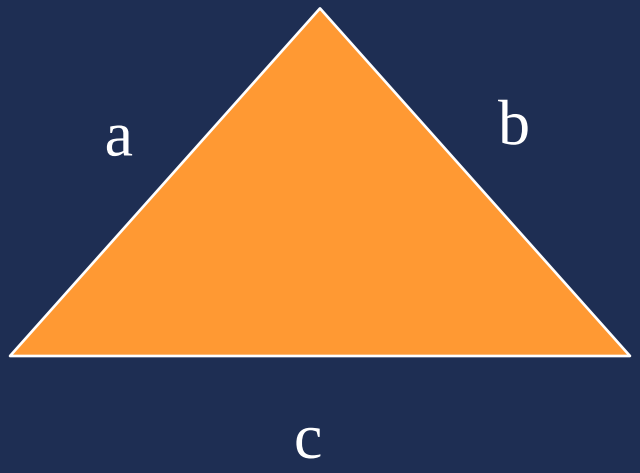
  Work(), Promote() ,…

4.1.2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。

面向对象举例



A

double a;double b;double c;
double GetPerimeter();
double GetArea();
double GetAngle(int which);
……

4.1.2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

4.1.3继承

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，进行更具体的说明。

实现：声明派生类——*第七章*

4.1.4多态性

- 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——*第八章*

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。

类的定义形式

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

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

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, Minute, 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

- 在类中说明原形，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)


```
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)


```
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

例4-1类的应用举例


```
#include<iostream>
using namespace std;
class Clock
{
        ......//类的声明略
}
//......类的实现略
void main(void)
{     Clock  myClock;
       myClock.SetTime(8,30,30);
       myClock.ShowTime();
}
```

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默认形式的构造函数
- 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：

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

复制构造函数

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。

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

例4-2 复制构造函数举例


```
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<<“复制构造函数被调用"<<endl;
}
```

例4-2复制构造函数举例

- 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复制构造函数实现拷贝赋值。

```
void main(void)
{   Point A(1,2);
    Point B(A); //复制构造函数被调用
    cout<<B.GetX()<<endl;
}
```

例4-2复制构造函数举例

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

```
void fun1(Point p)
{    cout<<p.GetX()<<endl;
}
void main()
{    Point A(1,2);
     fun1(A); //调用复制构造函数
}
```

复制构造函数(例4-2)

- 当函数的返回值是类对象时，系统自动调用复制构造函数。例如：

```
Point fun2()
{     Point A(1,2);
      return A; //调用复制构造函数
}
void main()
{
      Point B;
      B=fun2();
}
```

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。

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。

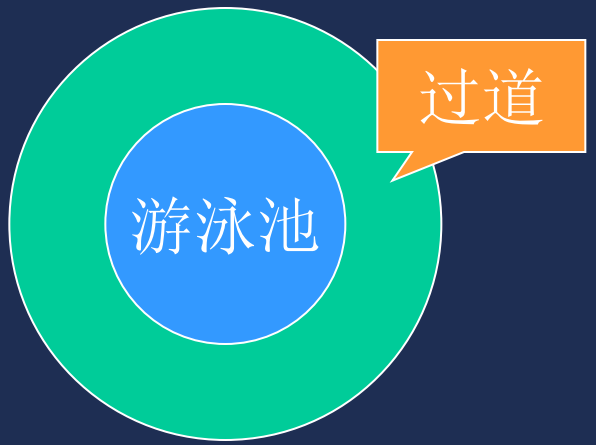
构造函数和析构函数举例


```
#include<iostream>
using namespace std;
class Point
{
   public:
      Point(int xx,int yy);
      ~Point();
      //...其它函数原形
   private:
      int X,int Y;
};
```

```
Point::Point(int xx,int yy)
{      X=xx;    Y=yy;
}
Point::~Point()
{
}
//...其它函数的实现略
```

类的应用举例(例4-3)

一圆型游泳池如图所示，现在需在其周围建一圆型过道，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 = 3.14159;
const float FencePrice = 35;
const float ConcretePrice = 20;

//声明类Circle 及其数据和方法
class Circle
{
  private:
    float   radius;

  public:
    Circle(float r);  //构造函数

    float Circumference() const; //圆周长
    float Area() const;  //圆面积
};
```

```
// 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() const
{
    return 2 * PI * radius;
}

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

```
void main ()
{
   float radius;
   float FenceCost, ConcreteCost;

   // 提示用户输入半径
   cout<<"Enter the radius of the pool: ";
   cin>>radius;

   // 声明 Circle 对象
   Circle Pool(radius);
   Circle PoolRim(radius + 3);
```

```
// 计算栅栏造价并输出
    FenceCost = PoolRim.Circumference() * FencePrice;
    cout << "Fencing Cost is ￥" << FenceCost << endl;

    // 计算过道造价并输出
    ConcreteCost = (PoolRim.Area() -
    Pool.Area())*ConcretePrice;
    cout << "Concrete Cost is ￥" << ConcreteCost << endl;
}
```

运行结果

```
Enter the radius of the pool: 10
Fencing Cost is ￥2858.85
Concrete Cost is ￥4335.39
```

组合的概念

类的组合

- 类中的成员数据是另一个类的对象。
- 可以在已有的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(void); //取X坐标
        float GetY(void); //取Y坐标
        void Draw(void); //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
```

类组合的构造函数设计

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。
- 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 }
```

类组合的构造函数调用

- 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）
- 若调用默认构造函数（即无形参的），则内嵌对象的初始化也将调用相应的默认构造函数。

类的组合举例（二）

类的组合

```
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)
{}

//...其它函数的实现略
```

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，但具体声明在其它地方。

前向引用声明举例

类的组合

```
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

类的组合

```
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类的对象指针
};

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。

结构体——结构的声明

- 结构的概念
  - 结构是由不同数据类型的数据组成的集合体。
- 声明结构类型

```
struct  结构名
{
    数据类型   成员名 1;
    数据类型   成员名 2;
    屔  :
    数据类型   成员名 n;
};
```

结构体——结构的声明

- 举例：

```
struct student   //学生信息结构体
{
    int num;  //学号
    char name[20];  //姓名
    char gender;  //性别
    int age;  //年龄
    float score;  //成绩
    char addr[30];  //住址
}
```

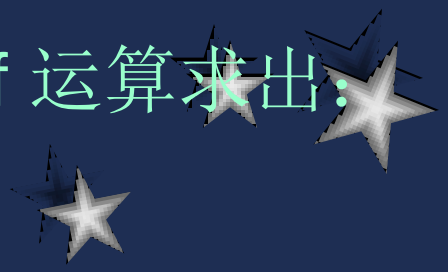
结构体——结构变量说明

- 变量说明形式

  结构名 结构变量名;

- 注意：
  - 结构变量的存储类型概念、它的寿命、可见性及使用范围与普通变量完全一致。
  - 结构变量说明在结构类型声明之后，二者也可同时进行。
  - 结构变量占内存大小可用 sizeof 运算求出：

    sizeof(运算量)



结构体
——结构变量的初始化和使用

- 初始化

  说明结构变量的同时可以直接设置初值。

- 使用

  结构体成员的引用形式：

  结构变量名.成员名

例2-12

结构体变量的初始化和使用

```
#include <iostream>
#include <iomanip>
using namespace std;
struct student   //学生信息结构体
{  int num;  //学号
    char name[20];  //姓名
    char gender;  //性别
    int age;  //年龄
}stu={97001,"Lin Lin",'F',19};
void main()
{  cout<<setw(7)<<stu.num<<setw(20)<<stu.name
        <<setw(3)<<stu.sex<<setw(3)<<stu.age;
}
```

运行结果:

```
 97001            Lin Lin   F 19
```

联合体

自定义数据类型

- 声明形式：

```
union 联合名
  {
     数据类型   成员名 1;
     数据类型   成员名 2;
     …… :
     数据类型   成员名 n;
  };
```

- 联合体类型变量说明的语法形式

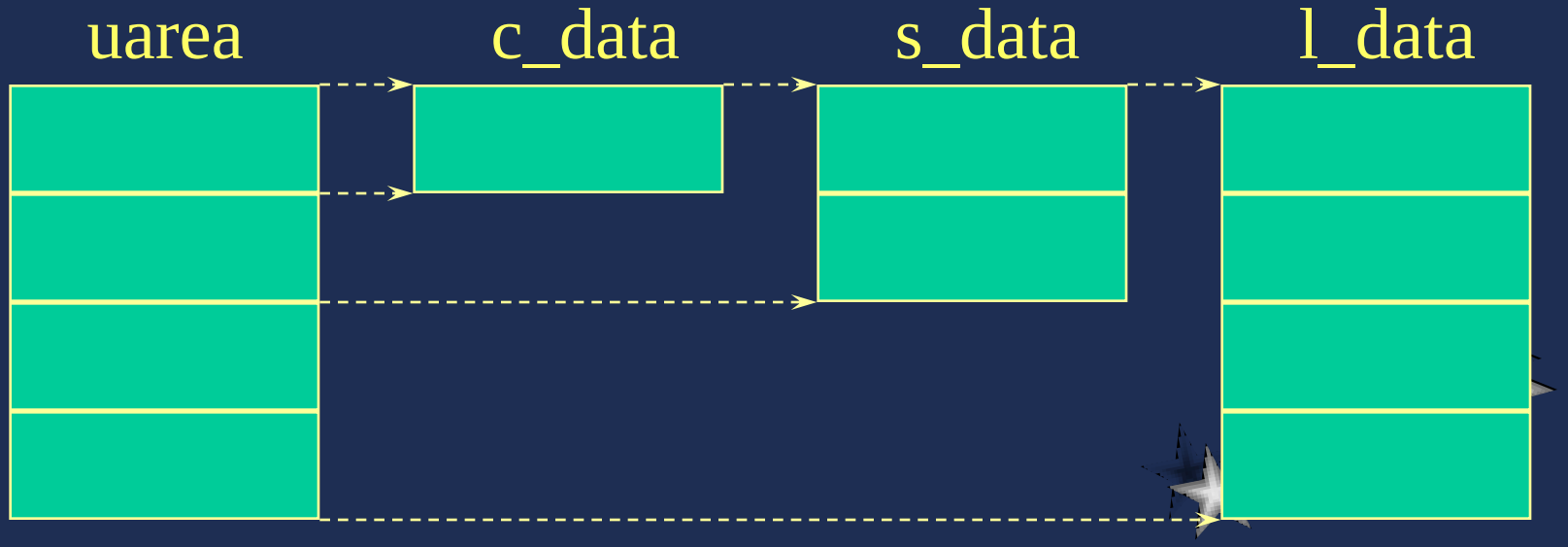
  联合名 联合变量名;

- 引用形式：

  联合名.成员名

联合体

例：
```
union uarea
{   char    c_data;
    short   s_data;
    long    l_data;
}
```



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;
```

小结

UML图形标识

UML简介

- UML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（realuzation）

UML图形标识

UML中的9种图

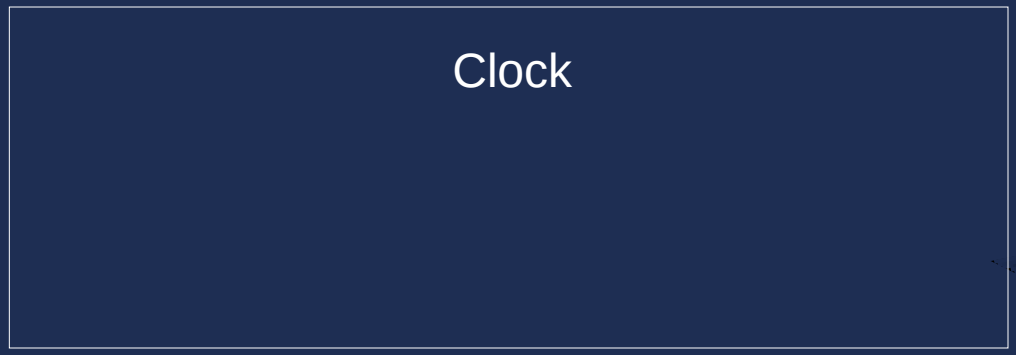
- 类图（class diagram）
- 对象图（class diagram）
- 用例图（Use case diagram）
- 顺序图（Sequence diagram）
- 协作图（Collaboration diagram）
- 状态图（Statechart diagram）
- 活动图（Activity diagram）
- 组件图（Compomnent diagram）
- 实施图（Deployment diagram）

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

myClock : Clock

- Hour : int
- Minute : int
- Second : int

myClock : Clock

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

UML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

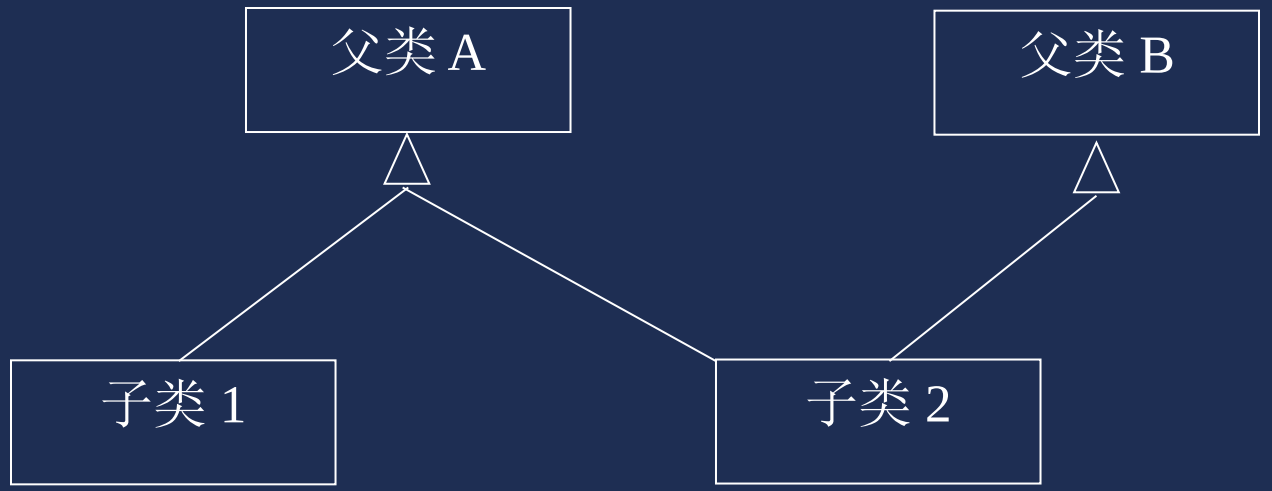
- 包含关系——聚集和组合

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

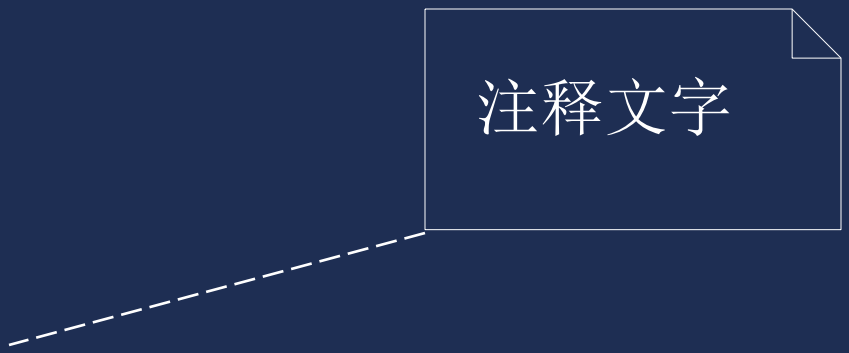
UML图形标识

- 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。



小结

面向对象的基本概念
——类

面向对象的方法

- 分类——人类通常的思维方法
- 分类所依据的原则——抽象
  - 忽略事物的非本质特征，只注意那些与当前目标有关的本质特征，从而找出事物的共性，把具有共同性质的事物划分为一类，得出一个抽象的概念。
  - 例如，石头、树木、汽车、房屋等都是人们在长期的生产和生活实践中抽象出的概念。

面向对象的基本概念
——类

- 面向对象方法中的"类"
  - 具有相同属性和服务的一组对象的集合
  - 为属于该类的全部对象提供了抽象的描述，包括属性和行为两个主要部分。
  - 类与对象的关系：
    犹如模具与铸件之间的关系，一个属于某类的对象称为该类的一个实例。

面向对象的基本概念
——封装

- 把对象的属性和服务结合成一个独立的系统单元。
- 尽可能隐蔽对象的内部细节。对外形成一个边界（或者说一道屏障），只保留有限的对外接口使之与外部发生联系。

面向对象的基本概念
——继承

- 继承对于软件复用有着重要意义，是面向对象技术能够提高软件开发效率的重要原因之一。
- 定义：特殊类的对象拥有其一般类的全部属性与服务，称作特殊类对一般类的继承。
- 例如：将轮船作为一个一般类，客轮便是一个特殊类。

面向对象的基本概念
——多态性

- 多态是指在一般类中定义的属性或行为，被特殊类继承之后，可以具有不同的数据类型或表现出不同的行为。这使得同一个属性或行为在一般类及其各个特殊类中具有不同的语义。
- 例如：

  数的加法->实数的加法
  ->复数的加法

面向对象的软件工程

- 面向对象的软件工程是面向对象方法在软件工程领域的全面应用。它包括:
  - 面向对象的分析（OOA）
  - 面向对象的设计（OOD）
  - 面向对象的编程（OOP）
  - 面向对象的测试（OOT）
  - 面向对象的软件维护（OOSM）