

第3章 类与对象

- 本章主要教学内容

- 抽象与封装、类的结构、定义、访问权限
- 构造函数、析构函数、静态成员和类对象、**this**指针
- 对象拷贝与对象移动

- 本章教学重点

- 学会设计类

- 1. 类（**class**）是实现数据封装和信息隐藏的工具，掌握从现实问题中抽象与封装出反映客观事物的类，并利用**C++**的**class(struct)**将其设计为可用于程序的自定义类型，掌握数据抽象的实现方法，学会设计类是本章学习的重点，也是继承和多态的基础。

- 2. 构造函数和析构函数

- 教学难点

- 对象创建、复制、赋值操作过程中的函数调用（构造函数、复制构造函数、赋值函数，以及它们的移动函数版本）及相关成员函数的设计。
- 对象的移动构造和移动赋值，**this**指针及对象自引用成员函数调用。
- 类设计的思想和方法

3.1 类的抽象与封装

1. ADT的概念

- ADT (Abstract Data Type) 是指由用户定义，用以描述应用问题的数据模型，它常由基本数据类型（如int, char, double）组合而成，并包括一组服务（即实现特定功能的函数），常称之为ADT的接口。

2. 面向对象程序设计的主要任务

- 是对求解问题域中的各类事物进行数据抽象，然后把它封装成对应的ADT——类。



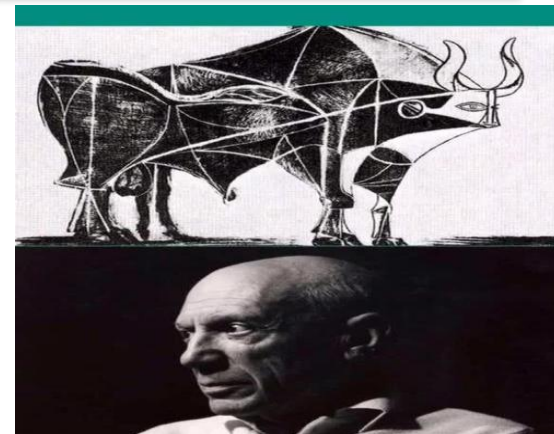
抽象

```
class Student{  
    char sid[8];  
    char name[8];  
    int age;  
    char school[2];  
    setSid(.....) ;  
    setAge(.....) ;  
    .....  
}
```

3.1.1 抽象

1. 抽象的概念

- 抽象是人们认识客观事物的一种常用方法，指在描述客观事物时，**有意去掉被考察对象的次要部分和具体细节，只抽取出与当前问题相关的重要性特征进行考察**，形成可以代表对应事物的概念。
- 抽象关注事物的**外部视图**，设计事物提供给其它事物使用的功能接口，并不涉及这些功能的内部实现。



2. 抽象的类型

- 过程抽象
- 数据抽象

抽象只设计出手电筒的外形，色彩，使用开关。不关心内部实现



3. 抽象的结果

- 形成对事物的ADT描述，**设计出了能够代表客观事物特征和行为的ADT的接口**。其它对象通过此接口能够了解到此ADT是什么类型的对象，具有什么特征和功能，以及使用这些功能（接口函数）的方法。
- 但是，**抽象并未实现ADT的内部细节**（包括特征数据的定义和接口函数的实现）

3.1.1 抽象

4. 过程抽象

- 是面向过程程序设计采用的以“**功能为中心**”的抽象方法，它**将整个系统的功能划分为若干部分，每部分则由若干过程（函数）完成**。
- 该方法强调过程的功能设计，只须准确地描述每个过程所要完成的功能，而忽略实现功能的详细细节（即**只设计出函数应该提供的功能，不涉及具体的编码实现**）。
- 若要实现两数的加、减、乘、除运算，采用过程抽象方法会得到类似于下面的抽象结果：
 - add(a,b); //功能：完成 $a+b$
 - sub(a,b); //功能：完成 $a-b$
 - mul(a,b); //功能：完成 $a\times b$
 - div(a,b); //功能：完成 $a\div b$

3.1.1 抽象

- 过程抽象的结果

- 过程抽象的结果给出了函数原形，包括函数名称，形参表和能够提供的功能，但并未编写实现函数功能的程序代码。
 - 至于这些函数如何编码实现，并不是过程抽象关心的事情（函数的内部实现可以采用多种形式，但并不会影响函数的使用）。
- 过程抽象实际上提供了信息隐藏和重用性，函数使用者只需要知道函数的名称，功能和参数形式，无需了解其实现细节，就可以进行函数调用，应用它的功能。

3.1.1 抽象

- 5. 数据抽象

- 数据抽象的概念

- 面向对象程序设计方法采用以“**数据为中心**”的抽象方法

- 数据抽象的方法

- 有意忽略事物与当前问题域无关的、不重要的部分和具体细节，**抽取同类事物与当前所研究问题相关联的、共有的基本特征和行为**，形成关于该事物的抽象数据类型

- 数据抽象的结果

- 形成了描述客观事物的ADT接口。
 - ADT中用数据表示事物的基本特征，称为**数据成员**；用函数表示其行为，称为**成员函数**。

3.1.1 抽象

【例3-1】某社区要对小区内的宠物狗实行信息化管理，设计出表示宠物狗的抽象数据类型。

(1) 问题分析

- 现实生活的各种宠物狗差别很大：有的高大，有的矮小；有的嘴长，有的嘴短；有的毛红，有的毛白；有的跑得快，有的跑得慢；有的叫声大，有的叫声小……
- 要详细地把各种狗的全部特征和行为描述出来非常困难。但是，这里的问题域是小区对宠物狗的管理，因此，不需要把狗的所有特征和行为都描述出来。
- 比如，狗喜欢什么饮食，食量大小，睡眠习惯，狗尾长短等特征和行为则与本问题域没有太大关系，可以不予考虑。反之，狗的名字和主人是谁等特征对本问题域而言，却是一个不可忽略的问题。

3.1.1 抽象

(2) 数据抽象

- 忽略掉与本问题域无关的特征和行为：宠物狗的叫声大小，狗的听力好坏，饮食习惯.....。
- 忽略不重要的、次要的宠物狗特征和行为：狗出生地在哪里，狗父狗母是谁，有无狗兄狗弟，狗毛的长短.....。
- 对于感兴趣的、与本问题研究有关的宠物狗共性特征进行抽取和描述。如毛色，尺寸、主人.....
 - 在抽象狗毛颜色时，忽略掉狗毛的长短、粗细、各种色彩等，只关注狗毛是有颜色的，用color表示；
 - 在抽象狗的高低时，忽略藏獒比吉娃娃要高大许多，只关注狗是有高度的，用high表示；
 -

3.1.1 抽象

- 宠物狗的初次抽象

抽象类型	Dog
重要特征	owner, name, color, high, len, breed
重要行为	run()

(3) 以数据为中心的抽象思想

- 信息隐藏**：数据通常被视为对象的“内部机密”，不允许直接访问，也就是说，Dog的name, color等特征数据会被隐藏起来，在程序中不能够直接操作它们。
- 接口公开**：为被隐匿的数据设计访问函数，只有通过访问函数才能操作数据。
- 由此可见，以数据为中心，并非只有数据，还包括对数据的操作。

3.1.1 抽象

- (3) 以数据为中心的抽象思想

- 为数据设计访问函数的一般方法

针对抽象出的每个特征数据X，设计出getX/setX两个读写该数据的函数，形式如下：

T x;

T getX() { return x; }

void setX(T y) { x=y; }

- 其中X是特征数据的名称，T代表x的数据类型，getX函数用于读取X的值，setX用于设置X的值。

3.1.1 抽象

宠物狗的抽象过程

现实中的各种宠物狗



具有不同的身体特征和行为



抽象过程

忽略狗嘴、狗耳、狗腿等与本问题域无关的特征.....

忽略狗毛粗细、长短，听力等不重要特征

忽略掉各种不同狗的叫声大小

睡眠习惯、摇头.....

抽取狗名、主人、颜色等与本问题相关的重要特征和行为

抽象结果

Dog	
name	breed
owner	
color	
high	
len	
run()	setLen()
setName()	getLen()
getName()	
setOwner()	
getOwner()	
setColor()	
getColor()	
setHigh()	
getHigh()	
setBreed()	
getBreed()	

3.1.1 抽象

- 宠物狗的抽象结果

类型	class Dog	
重要特征（数据成员）	string name	name 为宠物狗名，字符串类型数据
	string owner	owner 为狗的主人名字，字符串类型数据
	string color	color 为狗的颜色，字符串类型表示，如“黑色”
	double high	high 为狗的身高，用双精度数表示
	double len	len 为狗的长短，用双精度数表示
	string breed	breed 为狗的品种，用字符串表示，如“贵宾犬”
接口（成员函数）	void run();	按照狗的品种，输出狗跑的大致状态，速度等；
	void setName(string); / string getName();	设置/获取狗的名字
	void setOwner(string); / string getOwner();	设置/获取狗的主人名字
	void setHigh(double); / double getHigh();	设置/获取狗的身高数据
	void setLen(double); / double getLen();	设置/获取狗的长短数据
	void setBreed(string); / string getBreed();	设置/获取狗的品种
	void setColor(string); / string getColor();	设置/获取狗的颜色

3.1.2 封装

1. 为什么需要封装

- 抽象将对象可以被观察到的行为，设计成对应抽象数据类型的一组接口访问函数，用户可以了解该抽象类型的全部功能和调用方法。
- 但抽象只是设计出了**ADT**的接口函数，并没有实现函数功能。导致了接口与实现的分离，**封装是对接口的实现**。

2. 封装的概念

- 封装是**一种**将抽象形成的数据类型包装成可用于程序设计的**软件包装方法**。
- 它将**数据和基于数据的操作捆绑成一个整体，并且编码实现抽象所设计的接口功能**。
- **实现信息隐藏技术**
 - 将私密部分封装隐匿（通常包括对数据成员、接口的实现细节，以及抽象类型的结构）不让用户知道和访问。
 - 提供**对外接口**，用户能够通过接口访问该抽象类型的功能。

3.1.2 封装

3. 抽象与封装的关系

- 封装与抽象是一对互补的概念，抽象关注的是对象的外部视图，封装关注的则是对象的内部实现。
- 封装用来完成数据抽象设计的目标：**用户只能通过接口访问抽象数据类型的功能**，他只须向接口函数传递正确的参数，就能够使用该接口的功能，不必知道这些功能的实现细节，以及内部数据的状态。

类型	Class Dog
重要特征	<code>string name</code> <code>string owner</code> <code>string color</code>
接口	<code>void run();</code> <code>void setName(string); / string getName();</code> <code>void setOwner(string); / string getOwner();</code>

3.1.2 封装

• 4. 封装的实现技术

- 面向对象程序设计语言通过类（**class**）来实现封装，也可以说封装好之后的抽象数据类型称为类。
- **类具有封装能力**，能够将抽象类型的数据和操作函数包装成一个整体，并将数据的内部结构和接口的实现细节隐藏起来，只向外界提供接口。除了能够通过接口访问类的功能之外，外部对象对类的内部构造和实现细节是一无所知的。
- **class的基本结构如下：**

```
class 类名{
```

```
public:
```

```
    公有成员;
```

```
//接口
```

```
private:
```

```
    私有成员;
```

```
//信息隐藏
```

```
};
```


3.1.2 封装

【例3-2】用class对宠物狗的抽象结果进行封装，完成宠物狗抽象数据类型Dog的最后设计。

(1) 问题分析

- 对例3-1已经完成了对宠物狗的抽象结果进行封装：
 - 需要隐藏的数据成员
 - 包括name, owner, high, len, breed, 只需要用C++的数据类型定义这些数据成员，并把它们放置在class的private区域。
 - 为了便于字符串的输入输出，用C++的string类型定义name, owner, breed; 对于high和len可以用int定义（以厘米为单位），也可以用double类型（以米为单位）。
 - 接口的实现
 - 接口函数是围绕数据成员设置的，因此其参数类型与其对应数据成员类型相同。可以对每个数据成员设计一个读/写函数。
 - 例如：setHigh/getHigh用于设置/读取high，而high是double，因此只需要向setHigh函数传递double形的参数，并用它设置high就完成了对狗高的修改。getHigh返回double类型的值。

(2) 宠物狗封装好的抽象数据类型**Dog**如下。

```
class Dog {  
public:  
    void run() {cout << "I am " << name << ",my speed is " << rand()<< endl; }  
    void setName(string Dname) { name = Dname; };  
    void setOwner(string Dname) { owner = Dname; }  
    void setHigh(double Dhigh) { high = Dhigh; }  
    void setLen(double Dlen) { len = Dlen; }  
    string getName() { return name; }  
    string getOwner() { return owner; }  
    double getHigh() { return high; }  
    .....  
private:  
    string name;  
    string owner;  
    string color;  
    double high;  
    double len;  
    string breed;  
};
```

接口

信息
隐匿

经过封装之后的**Dog**就成了在程序设计中可用的用户自定义数据类型，就像用**int**、**char**等内置数据类型定义变量一样使用！

3.2 结构

3.2.1 C++对结构的扩展

- C++扩展了C语言结构的功能，不仅**可以包含数据**，而且**可以包含函数**，同时还引入了private、public和protected三个访问权限限定符，其目的是**实现数据封装和信息隐藏**。

1. C++结构的定义形式

struct 类名{

[public:] //实现**接口功能**，允许被访问的函数和
 成员； //数据成员放在此区域

private: //实现**信息隐藏**，只能被结构内部
 成员； //访问的成员放在此区域

protected: //保护成员，与**继承有关**
 成员；

};

3.2.1 C++对struct的扩展

2. 对于结构的几点说明

① 成员类型

- **数据成员**：结构中的数据称为数据成员。在一些面向对象语言（如**JAVA**等）称为属性或字段。
- **成员函数**：结构中的函数称为函数成员。在**C++**中函数成员常称为成员函数，另一些面向对象语言（如**JAVA**等）则称为方法。

② 设置访问权限的原因：隐藏

- 将数据和操作数据的函数包装在一起的主要目的是实现数据封装和信息隐藏，信息隐藏就是使结构中的数据和~~对数据~~对数据进行操作的细节对外不可见。

3.2.1 C++对struct的扩展

③ 成员访问控制权限

– 访问权限控制标识符

- **public**
 - 可以从结构（类）的外部（使用者）访问
 - 实现抽象定义类的接口
 - **private**
 - 仅供结构（类）的内部（自身成员函数）访问
 - 实现信息隐藏
 - **protected**
 - 供结构（类）的内部及后代访问
- public、protected和private用于设置成员访问权限，这些访问控制符可以按任意次序出现任意多次。也可以将多个public区域合并为一个区域。
- 在默认情况下（即没有指定访问控制权限），成员的访问控制权限为**public**。

3.2.1 C++对struct的扩展

【例3-3】用struct对圆进行抽象，构造出计算圆周长和面积的抽象数据类型。

(1) 问题分析

- 圆是一种常见的几何图形，具有圆心和半径，但本问题只需要计算圆的周长和面积，与圆心没有太大关系，可以将其忽略。只需要考虑圆的半径（`r`），周长（`perimeter`），面积（`area`）。

(2) 数据抽象

- 忽略圆心之后，半径就是唯一的数据成员了，用`r`表示，按照抽象的原则，将它设置为私有数据成员，以实现信息隐藏。并`setR/getR`接口函数用于设置/读取`r`的信息，`perimeter`函数用于计算圆的周长，`area`函数计算圆的面积。

3.2.1 C++对struct的扩展

(3) UML类图

- UML (Unified Modeling Language) 即**统一建模语言**，它定义了用例图、类图、对象图、状态图、活动图、序列图、协作图、构件图、部署图等**9**个标准图形，用于从不同的侧面对系统进行描述，能够表达软件设计中的动态和静态信息，便于系统的分析和构造，是**面向对象软件的标准化建模语言**。
- **类图**是UML中最重要也是最常见的一种图形，用于描述类的成员组成和关系。**类图**用一个矩形表示，其中包括类名、数据成员和成员函数三部分。且在类图中，
 - “+”表示**public**访问特性，
 - “-”表示**private**访问特性，
 - “#”表示**protected**（保护）访问特性，表示方法是在类成员的前面写上与其访问特性相对应的符号。

3.2.1 C++对struct的扩展

(4) 类图的结构及Circle的抽象结果: Circle类图

类名
数据成员
成员函数

Circle
-r:double
+setR(double radio):void +getR():double +perimete():double +area():double

3.2.1 C++对struct的扩展

(5) 封装后的Circle类及应用测试

//Eg3-3.cpp

#include<iostream>

#include<string>

using namespace std;

struct Circle {

public: //下面是公有成员函数，可被外部访问

void setR(double radio) { r = radio; }

double getR() { return r; }

double perimeter() { return 2 * 3.14*r; }

double area() { return 3.14*r*r; }

3.2.1 C++对struct的扩展

`private:` //下面的是私有成员，只能被内部访问

`double r;`

`};`

`void main() {`

`Circle c;` //可以像int定义变量一样，用Circle定义变量

`//c.r = 4;` //错误,r为private，不能被Circle外部的函数访问

 //下面的语句访问了对象c的public成员函数

`c.setR(4);`

`cout << "r=" << c.getR()`

`<< "\tperimeter=" << c.perimeter()`

`<< "\area=" << c.area() << endl;`

`}`

3.2.2 类

1. Class的功能及应用

- class具有信息隐藏能力，能够完成接口与实现的分离，用于把数据抽象的结果封装成可以用于程序设计的抽象数据类型，是面向对象程序设计语言中通用的数据封装工具。
- Java, Python, Rube, php.....都用class来定义类。

2. Class的结构

- 在C++中，class具有与struct完全相同的功能，用法一致。

```
class class_name{
```

```
    [private:]
```

```
//可以省略
```

```
    成员;
```

```
    public:
```

```
    成员;
```

```
    protected:
```

```
    成员;
```

```
};
```

```
//分号必不可少
```

3.2.2 类

3. 对class的几点说明

① 类成员

- class声明中的数据和函数通称成员，其中数据称为数据成员，函数则常被称作成员函数。

② 访问权限控制

- class声明中的访问限定符private、public、protected出现次数没有限制，也没有先后次序之分。
- 方便用户了解类的可访问接口，通常将public成员放在前面，private成员的声明放在类的后面。

③ 类域

- class或struct后一对大括号“{.....};”所包围的区域是一种独立的作用域，称为类域。
- 同一类域里的成员不受访问public、private和protected访问权限的限制，相互之间可以直接访问。

3.2.2 类

④ **class**和**struct**的区别

- **struct**也是一种类，与**class**具有**相同的功能**，用法也相同。
- 两者唯一的区别是，在没有指定成员的访问权限时，**struct**中的成员具有**public**权限，而**class**中的成员则具有**private**权限。
- 在实际应用中，为兼容C程序的编程特征，常用**struct**设计只有数据成员的结构，用**class**设计既用数据成员，也有成员函数的类。

3.2.2 类

【例3-4】 设计复数类Complex，提供复数的修改、输入和显示功能。

(1) 问题分析

复数由实部和虚部组成，能够进行加、减、乘、除等数学运算，但**本问题并未要求实现这些功能，因此忽略这些运算。**

(2) 数据抽象

- 出于信息隐藏目的，将复数**实部、虚部**设置为**double**类型的私有成员
- **设置输入、修改、显示它们的接口函数**inputData, setReal, setImage, display。
- 数据抽象结果如图所示。

Complex
- image: double - real: double
+ display(): void + inputData(): void + setImage(double): void + setReal(double): void

3.2.2 类

4. 封装后的**complex**类及其应用

```
#include<iostream>
```

```
using namespace std;
```

```
class Complex{
```

```
public:
```

```
    void display() { cout << real << "+" << image << "i" << endl; }
```

```
    void inputData() {
```

```
        cout << "input real: ";                cin >> real;
```

```
        cout << endl << "input image: ";        cin >> image;
```

```
    }
```

```
    void setImage(double i) { image = i; }
```

```
    void setReal(double r) { real = r; }
```

```
private:
```

```
    double image;
```

```
    double real;
```

```
};
```

3.2.2 类

C1

```
void main() {  
    Complex c1;  
    //c1.image = 9.2;
```

```
    c1.inputData();  
    c1.display();  
    c1.setImage(9.2);  
    c1.setReal(5.3);  
    c1.display();  
}
```



访问接口

setImage()	image
Display()	
setReal()	real
inputData()	

// 错误

公有接口

信息隐匿

成员函数的访问方法

在类外，通过对象
成员访问符“.”调用
public成员

3.3 数据成员

1. 数据成员的类型

- 数据成员可以是任何数据类型，如整型、浮点型、字符型、数组、指针、引用等，也可以是另外一个类的对象或指向对象的指针。
- 数据成员可以是指向自身类的指针或引用，不能是自身类的对象。
- 可以是const常量，不能是constexpr常量。
- 可以用decltype推断定义，但不能用auto推断定义。
- 此外，数据成员不能指定为寄存器（register）和外部（extern）存储类型。

• 分析类B数据成员错误原因

```
class A{ /* ..... */;
class B{
private:
    int a;
    A obja1, *obja2;           //正确
    B *objb, &objr;           //正确
    B b1;                     //错误
    auto b=a+1;               //错误
    decltype (7.9) a;         //正确
    extern int c;              //错误
    const int x;               //正确
    constexpr int y;          //错误
public:
    // .....
};
```


3.3 数据成员

2. 数据成员的类内初始值

11C++

- 但C++11标准规定，可以为数据成员提供一个类内初始值，用于创建类对象时初始化数据成员。

```
class A {  
private:  
    int a = 0,y = { 0 };           //C++11之前错误， C++11之后正确  
    int b[3] = {1,2,3 };           //C++11之前错误， C++11之后正确  
    const int c = a;                //C++11之前错误， C++11之后正确  
    //.....  
};
```

3. 数据成员初始化说明

- 类的声明（或定义）只是增加了一种自定义数据类型，此时类内部的数据成员并没有获取到相应的内存空间。
- **只有在用类定义对象时，才会为数据成员分配空间**，在这个时间点上才会用相应的初始值初始化数据成员。

小结

如何设计类？

设计方法

抽象与
封装

数据
抽象

过程
抽象

设计工具

```
class/struct{  
private:  
    data  
public:  
    fun()  
}
```

数据成员定义

```
class/  
任何类型  
自身类引用、指针  
const,decltype  
不能:  
    auto  
    自身类对象  
    Register  
    Extern  
public:  
    fun()  
}
```

即将学习

成员函数定义

初始
化

类内初始值

构造函数

复制

赋值

移动赋值

定
义

类内定义

类外定义

3.4 成员函数

3.4.1 成员函数定义方式和内联函数

1. 类内定义成员函数

- 在声明类时，直接在类的内部就给出成员函数的定义，以这种方式定义的成员函数若符合内联函数的条件就会被处理为内联函数。

```
class Date{  
    int day,month,year;  
public:  
    void init(int d,int m,int y) { day=d;month=m; year=y;} //inline  
    int getDay() {return day;} //inline函数  
    .....  
};
```

3.4.1 成员函数定义方式和内联函数

2、类外定义成员函数

- 在声明类时，若只声明了成员函数的原型，就需要在类的外部定义该成员函数，方法是：

returntype class_name::f_Name(T1 p1, T2 p2,...);

- 例如：类**Data**定义

```
class Date{
    int day,month,year;
public:
    void init(int ,int ,int );           //省略了形式参数
    int getDay();
    inline int getMonth()
};

int Date::getDay() {return day;}         //非inline函数
int Date::getMonth(){return month;}    //inline函数
inline void Date::init(int d,int m,int y) { //inline函数
    day=d; month=m;      year=y;
}
```

3.4.1 成员函数定义方式和内联函数

• 类外成员函数定义的几点说明

- ① 若采用类外方式定义成员函数，则类声明时成员**函数原型中的形参名可以省略**，只声明各个形参的类型；
- ② 在类外定义成员函数时，**成员函数的返回类型、函数名称、参数表必须与成员函数原型的声明完全相同**，而且必须指出每个形参的名字；
- ③ 在类外定义成员函数时，必须在成员函数名前面加上类名，并且在类名与成员函数之间用“**::**”间隔。

```
class D{
    int d,m,y;
public:
    int init(int ,int ,int );
        //省略了形式参数
    int getDay();
    inline int getMonth()
};
int D::init(int i ,int j ,int k)
{
    d=i
    m=j
    y=k;
    return 0;
}
```

3.4.2 常量成员函数

1. 常量成员函数的功能与定义

- 在C++中，常量成员函数用于禁止成员函数修改数据成员的值。
- 定义方式

```
class X{  
    .....  
    T f(...) const;  
    .....  
};
```

f被声明为常量成员函数，在其函数体内不能有任何修改类X数据成员的语句

3.4.3 常量成员函数

2. 常量成员函数应用案例

```
class Employee{
    char *name;
    double salary;
public:
    void init(const char *Name,const double y);
    double getSalary() const;           //不能通过它修改name和salary
    char *getName()const;              //不能通过它修改name和salary
    void addSalary(double x) const;     //不能通过它修改name和salary
};

double Employee::getSalary() const    //正确
{ return salary; }

void Employee::addSalary(double x) const
{ salary+=x; }                       //错误，常量成员函数不能修改数据成员

char *Employee::getName()
{ return name; }                     //错误，缺少const，与类中声明的原型不符
```

3.4.2 常量成员函数

3. 常量函数使用注意事项

- ① 只有类的成员函数才能定义为常量函数（本质上限定**this**指针为底层**const**），普通函数不能定义为常量函数。下面的函数定义是错误的：

```
int f(int x) const{           //错误，普通函数不能指定为const
    int b=x*x;
    return b;
}
```

- ② 常量参数与常量成员函数是有区别的，常量参数限制函数对参数的修改，但与数据成员是否被修改无关。

```
void Employee::init(const char *Name, const double y)
{ //参数是常量，但不是常量成员函数
    name=new char[strlen(Name)+1];
    strcpy(name,Name);
    salary=y;
}
```

```
class Employee{
    char *name;
    double salary;
public:
    .....
};
```


3.4.3 成员函数重载和默认参数

- 成员函数重载和默认参数规则
 - 重载成员函数**必须具有不同的参数表**
 - 在指定参数缺省值时，如果某个参数指定的缺省值，就要求它**右边的全部参数都必须指定默认值**。

```
class Date {  
    int day, month, year;  
public:  
    void init(int d, int m=8, int y=2016)  
        { day = d; month = m; year = y; }  
    void init(int d, int m)  
        { day = d; month = m; year = 2016; }  
    void init(int d)  
        { day = d; month = 8; year = d; }  
};
```

本程序有什么
问题？

**Date d(1)将
产生冲突**



3.5 对象

1. 对象的概念

- 类描述了同类事物共有的属性和行为，类是抽象的、概念性的范畴，**本质是一种数据类型。用类定义的变量就是对象**，对象是实际存在的个体，需要在内存空间中独立存在。
- 在用**类定义对象时，才会为数据成员分配内存空间**，对象同C程序中的变量一样，可以是全局对象、局部对象、静态对象，遵守同样的作用域和生存期规则。
- 广义地讲，在面向对象程序设计中**用任何数据类型定义的变量都可以称为对象**。

2. 对象的定义

- 定义对象的方法与定义一个普通变量相同，形式如下：
类名 对象1,对象2.....
类名 对象1(参数表)， 对象2（参数表）.....

3.5 对象



【例3-5】设计时钟类，要求能够完成时间的设置和显示，并创建时钟类的对象，演示对象的概念和用法。

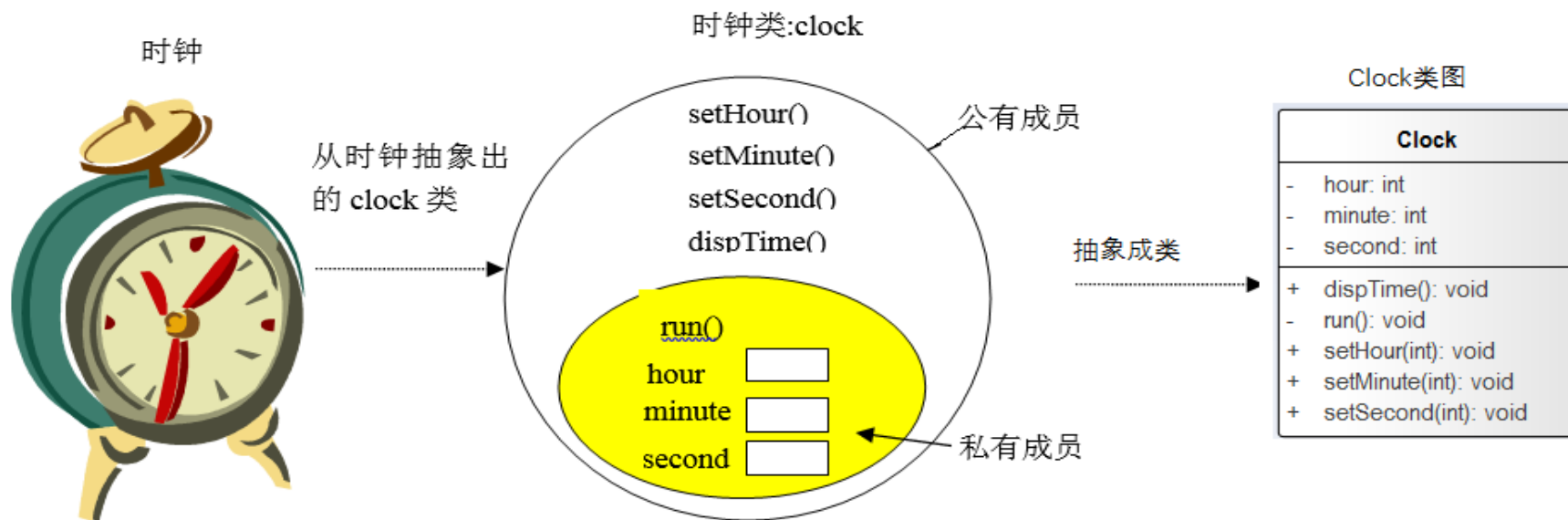
(1) 问题分析

- **接口设计**：任何时钟都是一个独立存在的有形实体，在钟面设置有时针、分针、秒针，人们通过这些指针查看时间，如果时间不准确，还可以通过这些指针调整时间。这些是时钟提供给人们使用的接口，可以设置对应的**public**函数来实现这一功能。
- **信息隐藏**：时、分、秒的保存和变化，以及指针运行是电流驱动还是机械驱动呢？由时钟内部管理，人们无须了解和干预，可以设置**private**成员来实现对它们的隐藏。

3.5 对象

(2) 数据抽象

- 用类Clock来抽象与封装时钟类，用私有成员hour、minute、second表示时、分、秒，函数run仿真时钟内部运行机制；
- 把时钟提供给人们的操作（如设置时、分、秒）分别用setTime、setMinute和setSecond公有成员函数来模仿，通过它们调整时间，用dispTime模仿时间的显示。



3.5 对象

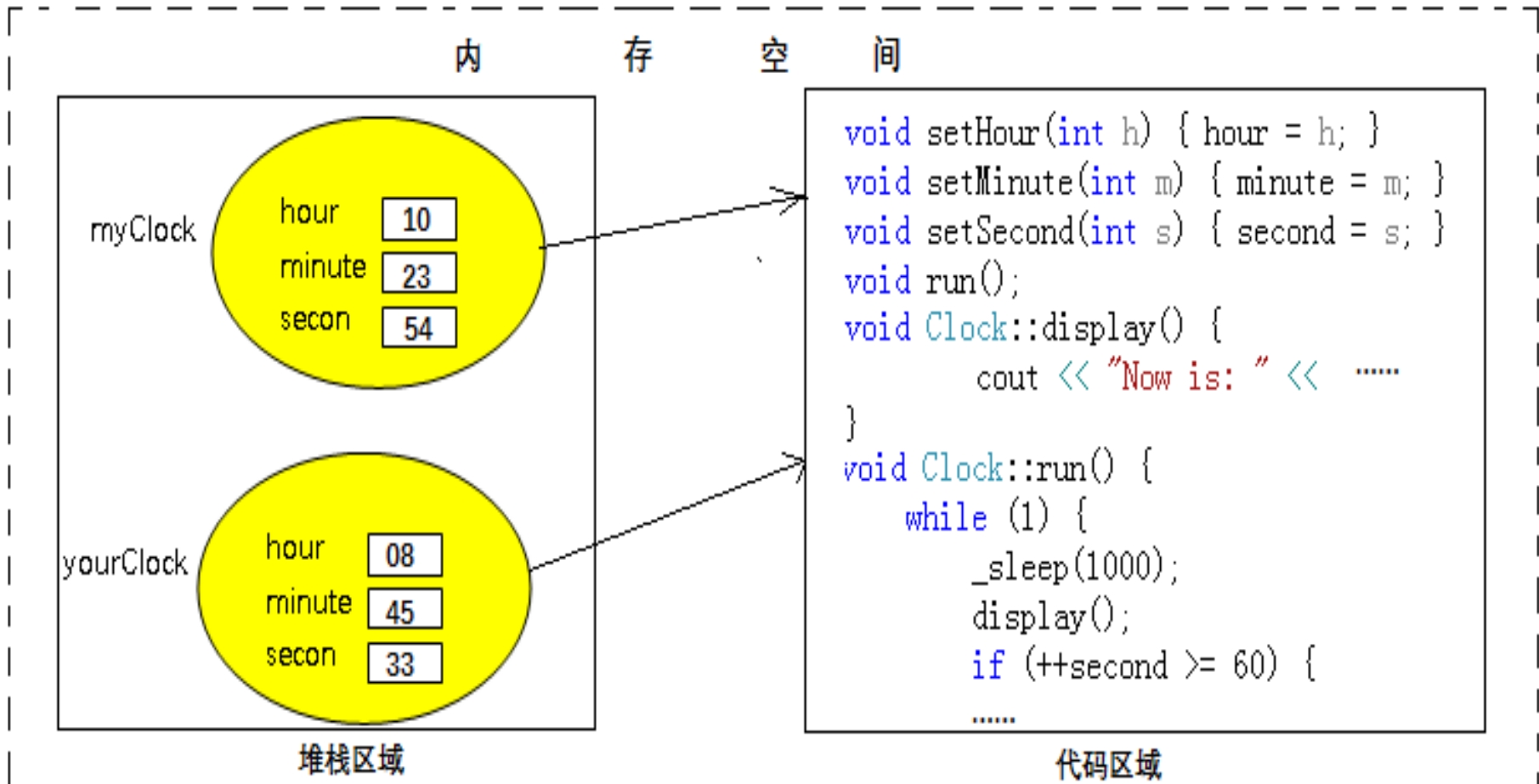
(3) Clock对象定义及内存结构

Clock myClock, yourClock;

- 这条语句定义了两个对象，每个对象都有8个成员，其中有hour、minute、second三个数据成员，run、dispTime、setHour、setMinute和setSecond五个成员函数。
- 每个对象的数据成员具有独立的内存空间，而成员函数则只有一份内存副本，由同类型所有对象共用。

3.5 对象

–myClock, yourClock对象内存结构示意图



3.5 对象

2. 对象的引用

(1) 通过对象访问成员函数

对象引用是指调用对象的接口函数获取类的功能，方法是用成员访问限定符“.”作为对象名和对象成员之间的间隔符。形式如下：

对象名.数据成员名；

对象名.成员函数名(实参表)；

— 例如，访问myClock的成员：

```
myClock.setHour(12);
```

```
myClock.dispTime();
```

3.5 对象

(2) 通过指针访问成员函数

- 如果定义了对象指针，在通过指针访问对象的公有成员时，要用“->”作为指针对象和对象成员之间的间隔符。

```
Clock *pClock;
```

```
pClock=new Clock;
```

```
pClock->setHour(10);
```

```
pClock->dispTime();
```

(3) 访问对象的注意事项

- 在类外只能访问对象的公有成员，不能访问对象的私有和受保护成员。

3.5 对象

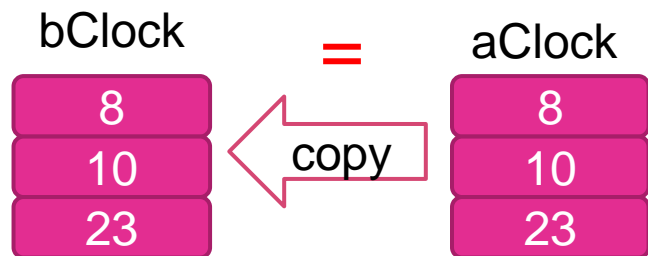
3、对象赋值

- 同一个类的不同**对象之间**，以及同一个类的对象**指针之间**可以相互赋值。

对象名1=对象名2;

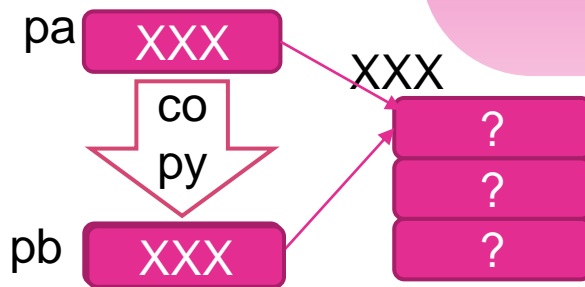
```
Clock *pa,*pb,aClock,bClock;
```

```
bClock=aClock;
```



```
pa=new Clock;
```

```
pb=pa;
```



对象赋值的注意事项:

- 1、两个对象必须类型相同
- 2、进行数据成员的值拷贝，赋值之后，两不相干
- 3、若对象有指针数据成员，赋值可能产生问题

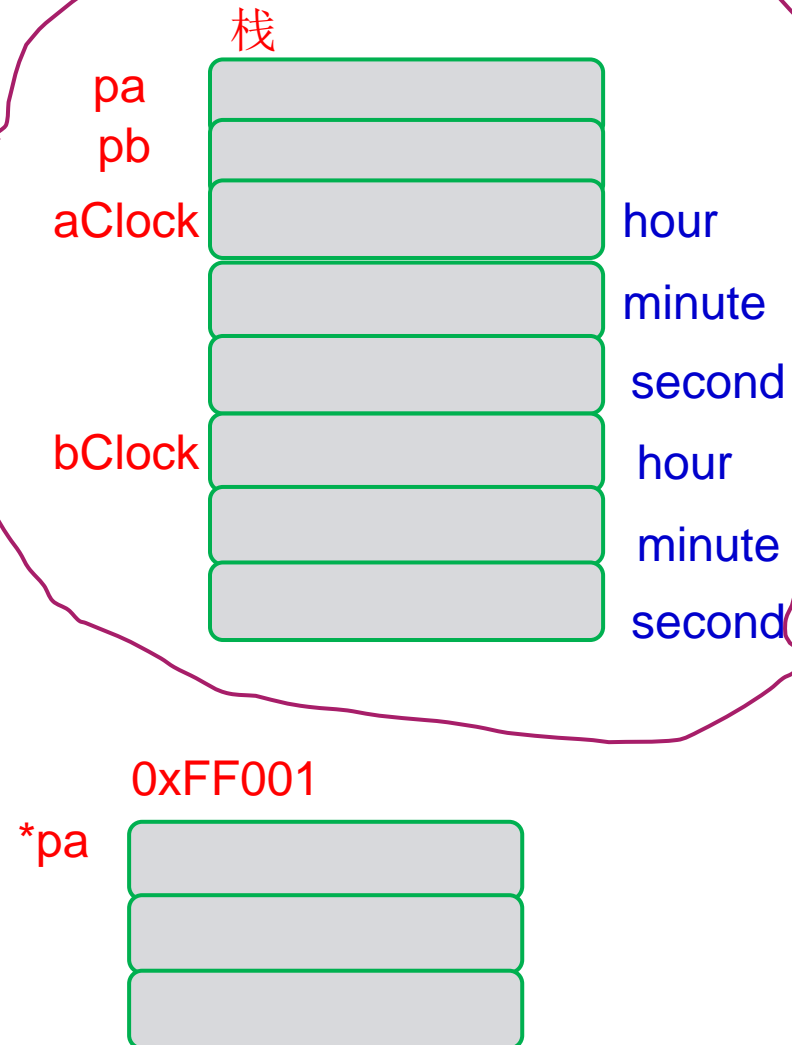
3.5 对象

【例3-5】 **Clock**类及其对象的完整例程。

```
#include<iostream>
#include<string>
using namespace std;
class Clock{
public:
    void setHour(int h) { hour=h; }
    void setMinute(int m) { minute=m; }
    void setSecond(int s) { second=s; }
    void dispTime(){
        cout<<"Now is: "<<hour<<":"<<minute<<":"<<second<<endl;
    }
private:
    int hour,minute,second;
};
```

3.5 对象

```
void main(){  
    Clock *pa,*pb,aClock,bClock;  
    aClock.setHour(16);  
    aClock.setMinute(12);  
    aClock.setSecond(27);  
    bClock=aClock;  
    pa=new Clock;  
    pa->setHour(10);  
    pa->setMinute(23);  
    pa->setSecond(34);  
    pb=pa;  
    pa->dispTime();  
    pb->dispTime();  
    aClock.dispTime();  
    bClock.dispTime();  
}
```



3.6 构造函数设计

1. 构造函数和析构函数与类的关系

- **构造函数与析构函数**是设计类时必须考虑和设计的两个极其特殊的函数，它们可以由系统自动执行，也可以明确定义，在程序中不可显示地调用它们。理解这两个函数对学好面向对象程序设计技术是大有帮助的。
- 构造函数的主要作用是用于建立对象时对对象的数据成员进行初始化；而析构函数主要用于对象生命期结束时回收对象。

2. 设计类时应当考虑的重要函数

- 从程序设计的角度出发，在**设计类时还应当考虑对象定义时数据成员的初始化，对象之间的复制、赋值、移动和销毁等问题**，而这些是通过构造函数、复制构造函数、赋值运算符函数、移动复制构造函数、移动赋值运算符和析构函数来实现的。

3.6.1 编译器默认添加的成员函数

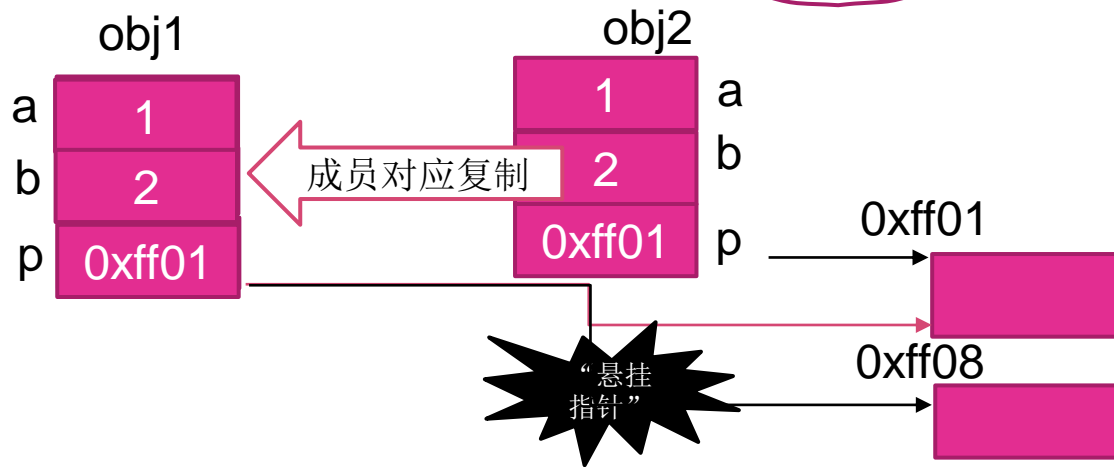
C++编译器会在需要的时候为符合条件的类自动在类中添加以下6个成员函数，

- ①默认构造函数；
- ②复制构造函数；
- ③赋值运算符函数；
- ④移动复制构造函数； c++11
- ⑤移动赋值运算符函数 c++11
- ⑥析构函数。

默认构造、赋值函数不能解决这3种对象复制，会产生“悬挂指针”

```
class A{  
    int a,b,c;  
    char *p  
    .....  
}  
A obj1, obj2  
A obj3(obj1);  
obj1=obj2;  
A f(){ return A(..);}  
Obj2=f();
```

类没有定义任何构造函数和析构函数，就符合编译器自动为类默认生成上6个函数的条件！



编译器默认添加的成员函数

```
class A {  
    type1 x1;  
    type2 x2;  
    ...  
    typen xn;  
public:  
    A():x1(),x2(),.....,xn(){ } //L1, 默认构造函数  
  
    A(const A& other) :x1(other.x1), //L2, 复制构造函数  
        ....., xn(other.xn) { }  
  
    A(const A&& other) :x1(move(other.x1)), //L3, 移动复制构造函数  
        ....., xn(move(other.xn)) { }  
  
    A& operator=(const A& other) { //L4, 赋值运算符函数  
        x1 = other.x1;           ....., xn = other.xn;  
        return *this;  
    }  
  
    A& operator=(const A&& other) { //L5, 移动赋值运算符函数  
        x1 = move(other.x1);  
        ....., xn = move(other.xn);  
        return *this;  
    }  
  
    ~A() { //L6, 析构函数  
        xn.~typen();.....  
        x1.~type1();  
    }  
};
```

编译器自动为类生成的成员函数在没有指针时是可信可用的。当有指针成员且在析构函数中delete指针会产生“悬挂指针”，引起运行的错误。两种解决办法可参考：

“5法则”，是指复制构造函数、移动构造函数、赋值运算符函数、移动赋值运算符函数和析构函数5个操作要么全部显式设计，要么一个也不设计。

“0法则”，是指用C++ 11中的智能指针unique_ptr或shared_ptr来代替传统指针。

对象构造、析构和复制的禁止

```
class A {  
    //.....  
public:  
    A() = delete;  
    A(const A& other) = delete;  
    A(const A&& other) = delete;  
    A& operator=(const A& other) = delete;  
    A& operator=(const A&& other) = delete;  
    ~A() = delete;  
};  
  
A obj1;                //错误，默认构造被禁止  
A obj2(obj1);          //错误，复制构造被禁止  
obj1=obj2;             //错误，赋值运算被禁止
```

如果不允许对象间的复制、赋值或移动操作，可以用`delete`禁止相关成员函数

3.6.1 构造函数和类内初始值

1、构造函数和类内初始值的概念

- **构造函数**（**constructor**）
 - 与类同名的特殊成员函数，主要用来初始化对象的数据成员。
- **类内初始值**
 - 指在类声明时，为数据成员指定的初始值。 **C++11**
 - 以前的标准禁止在类声明数据成员时指定初值。
- **在类中的定义形式如下**

```
class X{  
    .....  
    X(...);           //构造函数  
    T m=a;            //类内初始值    11C++  
    .....  
}
```


3.6.2 构造函数和类内初始值

2、构造函数的特点

- ① 构造函数与类同名，并且没有返回类型；
- ② 构造函数可以被重载；
- ③ 构造函数由系统自动调用，不允许在程序中显式调用；
- ④ 构造函数不能被声明为const函数。

```
class A {  
public:  
    A() {}  
    A(int, int) {}  
};
```

3. 类内初始值和构造函数的执行时机

在用类定义对象时，将按以下次序**依次执行**：

- ① 编译器建立对象，**为数据成员分配内存空间**；
- ② 若指定了数据成员的类内初始值，则用**类内初始值初始化数据成员**；
- ③ 根据定义对象时提供的参数匹配正确的构造函数，**执行构造函数**。

3.6.2 构造函数和类内初始值

4. 构造函数的调用

– 只能在定义对象时，由系统自动调用！

– 调用形式：

类名 对象名（参数表）；

- 系统将根据参数表调用某个构造函数
- 若无参数表将调用缺省构造函数。

– 不允许程序员在程序中显示调用构造函数的名称，任何时候都不允许！

```
class A {  
public:  
    A() {}  
    A(int, int) {}  
};  
A a1;  
A a2(3,4);  
A a[10];  
  
a2.A(5,5); //error
```

3.6.2 构造函数和类内初始值

【例3-6】 一个桌子类的构造函数和类内初始值。

```
#include <iostream>
using namespace std;
class Desk {
public:
    Desk(int, int); //构造函数声明
    void outData() {
        cout << "Wight= " << weight << "\tHeight=" << high << endl;
        cout << "Lenght=" << length << "\tWidth=" << width << endl;
    }
private:
    int width,length, weight=2, high=3;
};
Desk::Desk(int l, int w) { //构造函数定义
    length = l; width = w;
    cout << "call constructor !" << endl;
}
void main() {
    Desk d(3, 5);
    d.outData();
}
```

构用**Desk**定义对象**d**时，会自动调用构造函数。对于语句，

Desk d(3,5);

编译器可能将其扩展成下面的语句组：

- ① **Desk d;**
- ② 执行类内初始化;
- ③ **d.Desk::Desk(3,5);**

3.6.2 构造函数和类内初始值

5、使用构造函数应注意的问题

- ① 构造函数不能有返回类型，即使**void**也不行。
- ② 构造函数由系统自动调用，不能在程序中显式调用构造函数。
- ③ 构造函数的调用时机是定义对象之后的第一时间，即构造函数是对象的第一个被调用函数。
- ④ 定义对象数组或用**new**创建动态对象时，也要调用构造函数。但定义对象数组时，必须有不需要参数的构造函数
- ⑤ 构造函数通常应定义为公有成员，因为在程序中定义对象时，要涉及构造函数的调用，尽管是由编译系统进行的隐式调用，但也是在类外进行的成员函数访问。

```
class A {  
public:  
    A() {}  
    A(int, int) {}  
};  
A a1;  
A a2(3,4);  
A a[10];  
  
a2.A(5,5); //error
```

构造函数错误分析案例

```
class Desk{
    Desk(){          weight=high=width=length=0;}    //无参构造函数为private
public:
    void Desk::Desk(int ww,int l,int w,int h) { //错误，不能有返回类型
        weight=ww; high=l;
        width=w; length=h;
    }
private:
    int weight,length,width,high;
};

void main(){
    Desk d(2,3,3,5);           //构造函数在定义对象时调用
    d.Desk(1,2,3,4);           //错误，构造函数不能被显式调用
    Desk a[10];                //错误,调用无参构造函数，但它是private
    Desk *pd;
    Desk d;                    //错误，调用Desk::Desk(), 但它是private
    pd=new Desk(1,1,1,1);      //调用构造函数Desk::Desk(int,int,int,int)
}
```

3.6.3 默认构造函数

- 默认构造函数的概念
 - 创建类对象时**没有显式提供初始化值**时调用的构造函数，称为默认构造函数。
- 默认构造函数的类型
 1. **不带参数的构造函数**
 2. 为**所有的形参都提供了默认值**的构造函数。
- 对象定义规则和应用默认构造函数的典型情况
 - **C++规则：在定义对象时，必须调用构造函数**
 - 定义无参对象；
 - 定义数组
 - 在派生类中可由系统自动调用基类或子对象的默认构造函数实施相应对象的初始化

3.6.3 默认构造函数

1、系统添加的默认构造函数

C++规定，每个类必须有构造函数，如果一个类没有定义任何构造函数，在需要时编译器将会自动为它生成一个默认构造函数。

```
class X {  
    X():x1(), ..., xn() {}//系统默认构造函数类似于此  
    .....  
}
```

– 在用默认构造函数创建对象时：

- ① 如果创建的是全局对象或静态对象，则对象所有数据成员初始化为0；
- ② 如果创建的是局部对象，即不进行对象数据成员的初始化。

3.6.3 默认构造函数

【例】 point类的默认构造函数。

//Eg-point.cpp

#include <iostream>

using namespace std;

class point{

private:

int x,y;

public:

void setpoint(int a,int b) { x=a;

int getx() { return x; }

int gety() { return y; }

};

point p1;

void main(){

static point p2;

point p3;

cout<<"p1: "<<p1.getx()<<","<<p1.gety()<<endl;

cout<<"p2: "<<p2.getx()<<","<<p2.gety()<<endl;

cout<<"p3: "<<p3.getx()<<","<<p3.gety()<<endl;

}

Point类没有定义任何构造函数，但在定义对象时必须调用构造函数！

编译器会为point合成无参数构造函数，并在定义p1,p2,p3时调用此构造函数！

//定义全局对象

//定义静态局部对象

//定义局部对象

3.6.3 默认构造函数

- 使用默认构造函数的注意事项

- ① 在类没有定义任何构造函数时，系统才会为类合成默认构造函数。一旦定义了任何形式的构造函数，系统就不再产生默认构造函数。
 - 在类有需要参数构造函数时，若需要创建无参对象，必须显式定义无参构造函数。但在C++11中，也可以用下面的方式要求编译器创建合成的默认构造函数。

```
class X {  
    X()=default;    // 11C++  
    X(.....){}    //需要参数的构造函数  
    .....  
}
```

【例】未定义无参构造函数引发的错误。

```
#include <iostream>  
using namespace std;  
class point{  
private:  
    int x,y;  
public:  
    point(int a,int b) { x=a; y=b; }  
    // .....  
};  
point p1;    //错误  
void main(){  
    static point p2;    //错误  
    point p3,*p4,a[10];    //错误  
    p4=new point;    //错误  
}
```

3.6.3 默认构造函数

② 默认构造函数引发的错误

- 在在某些情况下合成的默认构造函数会执行错误操作。比如，类具有数组或指针成员时，默认构造函数执行对象初始化时，很有可能产生“**指针悬挂**”问题。

```
Class B{  
    .....  
    B(int,int);  
}
```

③ 编译器不生成默认构造函数的情况

- 在在某些情况下，编译器不会为类创建默认构造函数。
 - 类A的一个数据成员是用类B创建的，但类B有其它构造函数，却没有默认构造函数，在这种情况下类A必须定义构造函数，并负责为对象成员提供构造函数初值。
 - 为类定义了某个构造函数
 - 有常量、引用成员，且未用类内初始值初始化
- 上面三种情况，需要程序员显式定义构造函数

```
Class A{  
    B b(1,2);  
    .....  
}
```

A必须定义构造函数并通过它为对象成员b提供构造初值。

3.6.3 默认构造函数

【例3-7】 设计表示平面坐标位置的点类，可以修改和获取点的x、y坐标值，设置构造函数对点的数据成员进行初始化，并且能够用数组保存一系列的点。

• 问题分析与数据抽象

- 将点抽象成Point类，将它的坐标值x、y设置为私有数据成员，并设置setPoint接口修改x、y的坐标值，设置getx，gety接口获取坐标点的x，y值，设置构造函数Point（int xx,int yy）初始化点的坐标值。
- 由于要定义数组，而且已定义了有参数的构造函数，编译器就不会再创建合成的默认构造函数了，必须显式定义默认构造函数将坐标点初始化为0。

Point	
-	x: int
-	y: int
+	getx(): int
+	gety(): int
+	Point(int, int)
+	Point()
+	setPoint(int, int): void

3.6.3 默认构造函数

[illegible]

3.6.3 默认构造函数

```
void main() {  
    static Point p2;           //L5 调用构造函数Point()  
    Point p3;                  //L6 调用构造函数Point()  
    Point a[10];               //L7 调用构造函数Point()  
    Point *p4;                 //L8 不调用任何构造函数  
    p4 = new Point;            //L9 调用构造函数Point()  
    p4->setPoint(8, 9);  
    cout << "p0: " << p0.getx() << "," << p0.gety() << endl;  
    cout << "p1: " << p1.getx() << "," << p1.gety() << endl;    //L10  
    cout << "p2: " << p2.getx() << "," << p2.gety() << endl;  
    cout << "p3: " << p3.getx() << "," << p3.gety() << endl;  
    cout << "p4: " << p4->getx() << "," << p4->gety() << endl;  
    cout << "a[0]: " << a[0].getx() << "," << a[0].gety() << endl;  
}
```

3.6.3 默认构造函数

2. 缺省参数构造函数

- 在数据成员的取值比较固定时，可以通过为构造函数参数提供缺省参数初始化它们。

【例3-8】在多数情况下，新建点坐标都是（0,0），修改例3-7设计的Point类，设置构造函数缺省参数值为坐标（0,0）。

```
#include <iostream>
using namespace std;
class Point {
private:
    int x, y;
public:
    Point(int a=0, int b=0) { setPoint(a, b); }    //L1
    //Point() { x = 0; y = 0; }    //定义无参对象时会与缺省参数构造函数冲突
    void setPoint(int a, int b) { x = a; y = b; }
    .....
};
```

3.6.3 默认构造函数

Point p1(1,1);	//L2 调用point(int ,int)构造函数
void main (){	
static Point p2;	//L3 调用point(a,b), a、b 默认为0
Point p3,a[10];	//L4 调用point(a,b), a、b 默认为0
Point *p4;	
P4=new Point;	//L5 调用point(a,b), a、b 默认为0
.....	
}	

3.6.3 默认构造函数

缺省参数的构造函数与无参构造函数的冲突问题

```
//CH.cpp
class X{
public:
    X(){};
    X(int i=0){x=i;};
private:
    int x;
};
main(){
    X one(12);
    X two;
}
```

X two调用:

X:X()

还是

X:X(int i=0)

?

编译器不能区别，将产生二义性冲突



3.6.3 重载构造函数

- 构造函数重载

- 与普通函数的重载一样，重载的构造函数必须具有不同的函数原型。形式如下：

```
class A{  
    A(int ){...}  
    A(int,int){....}  
    A(double ){...}  
}  
A a(1),b(2,3),c(3.4)
```

- **【例3-9】** 设计一个日期类，能够接受年、月、日3个参数，或者月份和日期2个参数，或者日期1个参数，或没有参数建立对象，若未提供年、月、日，设置为2008年8月8日

3.6.4 重载构造函数

- 问题分析与数据抽象

- 日期类的年、月、日可以用`year`、`month`、`day`三个数据成员表示，出于信息隐藏目的将它们设置为`private`成员。
- 围绕3个数据成员，可以设置`setDay`、`getDay`等读写数据的公有接口，同时设置`dispDate`接口函数显示对象的年、月、日信息。
- 用4个具有不同参数的构造函数来满足题目要求4种方式建立对象的要求。

Tdate	
-	day: int
-	month: int
-	year: int
+	dispDate(): void
+	getDay(): int
+	getMonth(): int
+	getYear(): int
+	setDay(int): void
+	setMonth(int): void
+	setYear(int): void
+	Tdate()
+	Tdate(int, int, int)
+	Tdate(int, int)
+	Tdate(int): void

3.6.4 重载构造函数

```
//Eg3-9.cpp
#include <iostream>
using namespace std;
class Tdate {
public:
    Tdate();
    Tdate(int d);
    Tdate(int m, int d);
    Tdate(int m, int d, int y);
    //..... //省略掉了设置和读取数据成员值的接口函数
    void display(){ cout << month << "/" << day
                    << "/" << year << endl; }
private:
    int year=2008,month=8, day=8;
};
```

// 11C₊₊

3.6.4 重载构造函数

```
Tdate::Tdate() {display();}  
Tdate::Tdate(int d) {  
    day = d;  
    display();  
}  
Tdate::Tdate(int m, int d) {  
    month = m; day = d;  
    display();  
}  
Tdate::Tdate(int m, int d, int y) {  
    month = m; day = d; year = y;  
    display();  
}  
void main() {  
    Tdate oneday;  
    Tdate aday();  
    Tdate bday1(10);  
    Tdate bday2 = 10;  
    Tdate cday(2, 12);  
    Tdate dday(1, 2, 1998);  
}
```

L2语句不会调用构造函数定义对象，它声明了一个返回Tdate类型的函数。

Tdate bday2 = 10;

等价于：

Tdate bday2(10);

//L1

//L2, 可以吗?

//L3

//L4

//L5

//L6

3.6.4 重载构造函数

- 缺省参数与重载构造函数的合理利用
 - 将上面的几个构造函数结合为一个：
 - **class Tdate{**
 public:
 Tdate(int m=4,int d=15,int y=1995)
 {
 month=m; day=d; year=y;
 cout <<month <<"/" <<day <<"/" <<year <<endl;
 }
 //其他公共成员
 protected:
 int month;
 int day;
 int year;
 };

3.6.5 构造函数与初始化列表

1. 初始化列表的概念

- 在构造函数形参表和函数体之间为成员赋初值的一种方式，似于下面的形式

构造函数名(参数表): 成员1(初始值),成员2(初始值),...{
.....
}

- 介于参数表后面的“:”与函数体{...}之间的内容就是成员初始化列表。其含义是将括号中的初始值参数的值赋给该括号前面的成员。

3.6.5 构造函数与初始化列表

【例3-10】用初始化列表初始化Tdate的month和day成员。

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

class Tdate {
public:
    Tdate(int year, int m, int d);
    //.....//其他公共成员
protected:
    int month = 12, day, year;
};

Tdate::Tdate(int y, int m, int d) : month(m), day(d) {
    year = y;
    cout << year << "/" << month << "/" << day << endl;
}

void main() {
    Tdate bday2(2003, 10, 1);
}
```

```
#include <iostream>
using namespace std;
class Tdate {
public:
    Tdate(int m, int d, int y);
    //.....//其他公共成员
protected:
    int day=12,month=day, year;
};
Tdate::Tdate(int m, int d, int y) :day(d) {
    year = y;
    cout << month << "/" << day << "/" << year << endl;
}
void main() {
    Tdate bday2(10, 1, 2003);
}
```

输出什么?

1/1/2003

12/1/2003



3.6.5 构造函数与初始化列表

2、使用构造函数初始化列表的注意

① 构造函数初始化列表的执行次序

- 初始化列表中成员初始化次序与它们在类中的声明次序相同，与初始列表中的次序无关。对**Tdate**类而言，下面3个构造函数完全相同。

```
Tdate::Tdate(int m,int d,int y)
    :month(m),day(d),year(y){}
```

```
Tdate::Tdate(int m,int d,int y)
    :year(y),month(m),day(d){}
```

```
Tdate::Tdate(int m,int d,int y)
    :day(d),year(y),month(m){}
```

- 尽管三个构造函数初始化列表中的**month**、**day**和**year**的次序不同，但它们都是按照**month→day→year**的次序初始化的，这个次序是其在**Tdate**中的声明次序。

3.6.5 构造函数与初始化列表

② 构造函数初始化列表的执行时间。

- 如果数据成员有类内初始值，则执行次序为：

类内初始值→构造函数初始化列表→构造函数体

③ 必须采用初始化列表（或类内初始值）进行初始化的成员

- 常量成员，引用成员，类对象成员，派生类构造函数对基类构造函数的调用。
- 类内初始化值是C++11标准才有的，在VC6.0中不能用

3.6.5 构造函数与初始化列表

【例3-11】常量和引用成员必须通过类内初始化列表进行初始化。

程序初

```
#include <iostream>
using namespace std;
class A {
    int x, y, j;
    const int i=4;           // 11C++
    int &k;
public:
    A(int a, int b, int c) : j(b), k(c), x(y) {
        y = a;
        cout << "x=" << x << "\t" << "y=" << y << endl;
        cout << "i=" << i << "\t" << "j=" << j << "\t" << "k=" << k << endl;
    };
void main() {
    int m = 6;
    A x(4, 5, m);
}
```

• 本程序的运行结果如下:

• x=?

• i=4

j=5

y=4

k=6

3.7 析构函数

1、析构函数的概念

- 析构函数（**destructor**）是与类同名的另一个特殊成员函数，作用与构造函数相反，用于在对象生存期结束时，完成对象的清理工作。

2、定义语法

```
class X
{
    ~X () {.....};
}
```

3、析构函数特点

- 函数名为~加类名
- 无参数
- 无返回值
- 不能重载：每个类仅有一个析构函数

3.7 析构函数

4、析构函数调用时机

- 对象生命期结束时自动调用
- 自动/局部对象：定义的语句块结束处
- 全局对象：程序结束时
- 静态对象：程序结束时

□ 使用

- 一般情况下，缺省的析构函数可以信任
- 善后处理，一般是释放动态分配的内存

3.7 析构函数

【例3-13】 析构函数和构造函数的应用。

```
#include <iostream>
using namespace std;
class A{
private:
    int i;
public:
    A(int x){    i=x;
        cout<<"constructor: "<<i<<endl;
    }
    ~A(){    cout<<"destructor : "<<i<<endl; }
};
void main(){
    A a1(1);
    A a2(2);
    A a3(3);
}
```

本程序的运行结果如下：

```
constructor: 1
constructor: 2
constructor: 3
destructor : 3
destructor : 2
destructor : 1
```

3.7 析构函数

5、使用析构说明

- ① 每个类都应该有一个析构函数。如果没有显式定义析构函数，**C++**编译器将产生一个最小化的默认析构函数，称为**合成的析构函数**，类似于：

X::~~X(){ }

- ② 若有多个对象同时结束生存期，**C++**将按照与调用构造函数相反的次序调用析构函数。
- ③ 构造函数和析构函数都可以是**inline**函数。
- ④ 构造函数和析构函数通常都需要在类外被调用，常设置为**public**访问属性。
- ⑤ 合成的析构函数通常都能够满足对象析构的要求。但在某些情况下，必须编写析构函数才能够完成对象销毁前的资源清理工作。常见情况是用它来释放由构造函数分配的自由存储空间。

RAII (Resource Acquisition is Initialization) 设计理论

即“资源获取即初始化”。实质利用对象生命周期来控制程序资源（如内存、文件句柄、网络连接等等）的技术，要求在构造函数中申请分配资源，在析构函数中释放资源。

【例3-14】 用析构函数释放构造函数分配的自由存储空间。

```
#include <iostream>
using namespace std;
class B{
private:
    int *a;  char *pc;
public:
    inline B(int x){
        a=new int[x];    pc=new char;
    }
    inline ~B(){
        delete []a;    delete pc;
    }
};
void main(){
    B x(10);
}
```

像**B**这样的类，在构造函数中进行了动态内存空间的分配，系统合成的默认析构函数就不能回收此空间，**必须编写析构函数**，回收动态内存空间，否则会产生内存泄漏

3.8 赋值运算符函数、拷贝构造函数和移动函数设计

1、为什么要设计这几个特殊成员函数

- 在面向对象程序设计过程中，对象的赋值、拷贝和移动极其普遍，这些操作是通过赋值运算符函数、拷贝构造函数或移动函数完成的。

2、默认赋值运算符函数、拷贝构造函数和移动函数

- 由于赋值、拷贝操作的普遍性，致使每个类都应该具有这些成员函数，如果在设计类时没有显式地定义它们，编译器就会自动为该类添加默认的赋值运算符函数、拷贝构造函数和移动构造函数，定义各函数的默认操作。
- 系统生成的默认函数大多数情况能够正确完成对象的赋值、拷贝和移动操作。但在某些情况下，默认函数的默认操作会出问题。
- 比较典型的情况是当类具有指针类型数据成员的时候，依赖默认赋值运算符函数进行对象赋值，或默认拷贝构造函数进行对象复制都会产生“指针悬挂”问题。这时，就必须显式定义类的赋值运算符函数和拷贝构造函数了。

3.8 赋值运算符/复制构造函数

1. 赋值运算符和复制构造函数的调用时机

- 用于实现同类对象间的赋值复制或复制构造。
- 当把类的一个对象赋值给另外一个对象时，就会调用赋值运算符成员函数来完成对象间的赋值。
- 用一个已定义对象初始化新建对象时，调用复制构造函数
- 类似于下面的形式

```
class A{.....};
```

```
A a, b;
```

```
a=b;           //调用赋值运算符函数
```

```
A C1(a),c2(a),c3{a},c4={a} //调用复制构造函数
```

- “=” 即赋值运算符，它是所有类都拥有的一个成员函数，称为赋值运算符成员函数，功能是把“=”右边对象的数据成员复制给左边对象。

3.8赋值运算符/复制构造函数

2. 默认赋值运算符/复制构造函数

- 如果类没有显式定义赋值运算符/复制构造函数，编译器会自动为该合成一个默认的赋值运算符/复制构造函数成员函数，以按位复制（bit-by-bit）的方式实现对象非静态数据成员的复制。

```
class A {  
    char *ptr;  
    int n;  
public:  
}  
A a, c;  
A b(a);  
c=a;
```

3. 对象赋值操作的执行过程

- ① 查找该类是否提供了显式的赋值运算符/复制构造函数，如果有且是可访问的（即public成员），就用此赋值运算符/复制构造进行对象赋值；如果提供了但不是可访问的（即private或protected成员），就产生编译错误。
- ② 如果该类没有显式定义赋值运算符/复制构造函数，就为该生成一个合成赋值运算符/复制构造函数，执行默认的复制操作。

3.8赋值运算符/复制构造函数

4. 需要定义赋值运算符/复制构造的情况

- 在通常情况下，默认赋值运算符/复制构造函数足以解决对象之间的赋值运算符/复制构造问题。但是，当类包含有指针数据成员时，合成赋值运算符/复制构造函数常会引发“指针悬挂”问题。

【例3-16】 有字符串类String，具有指针数据成员ptr用于存放字符串内容，n存放字符串编号。该类没有重载赋值运算符函数，编译器默认添加的赋值运算符/复制构造成员函数会引发指针悬挂问题。

3.8赋值运算符/复制构造函数

```
//Eg3-16.cpp
#include <iostream>
#include <string>
using namespace std;
class String{
    char *ptr;
    int n;
public:
    String(char * s,int a){
        ptr=new char[strlen(s)+1];
        strcpy(ptr,s);
        n=a;
    }
    ~String(){delete ptr;}
    void print(){cout<<ptr<<endl;}
};
```

3.8赋值运算符/复制构造函数

```
void main(){
    String p1((char*)"Hello",8);           //L1
    {   String p2((char*)"chong qing",10);   //L2
        p2=p1;                               //L3
        cout<<"p2:";                         //L4
        p2.print();                          //L5
    }                                         //L6 p2生存期结束
    cout<<"p1:";                             //L7
    p1.print();                             //L8,错误
}                                           //L9
```

运行结果:

p2:Hello

p1:葺葺葺葺葺葺葺葺葺葺葺葺???

3.8赋值运算符/复制构造函数

• 错误分析

- 当执行“p2=p1”时，由于String类没有提供赋值运算符函数，C++将为它生成合成赋值运算符函数，并调用它进行对象赋值。
- 编译器合成的赋值运算符函数类似于下面的形式：

```
String& String::operator=(const String &s){
```

```
    ptr=s.ptr;
```

```
    n=s.n;
```

```
    return *this;
```

```
}
```

1. L2执行后，p1,p2的ptr指针指向了同一内存单元。P2对象析构时会调用它的析构函数delete此内存区域。因此输出p1.ptr所指内容时是无效字符。
2. 当p1析构函数调用时，会再次delete同一内存区域，产生错误。称为“指针悬挂”

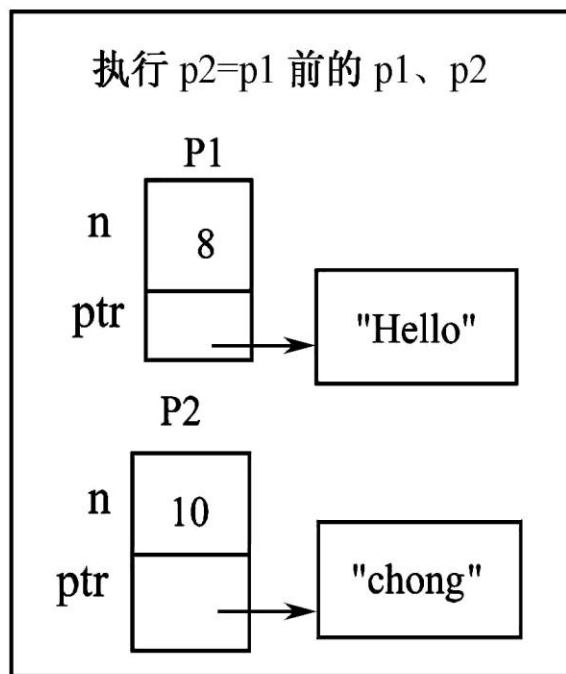
“p2=p1”操作就相当于执行以下两条语句：

```
p2.n=p1.n; //L1
```

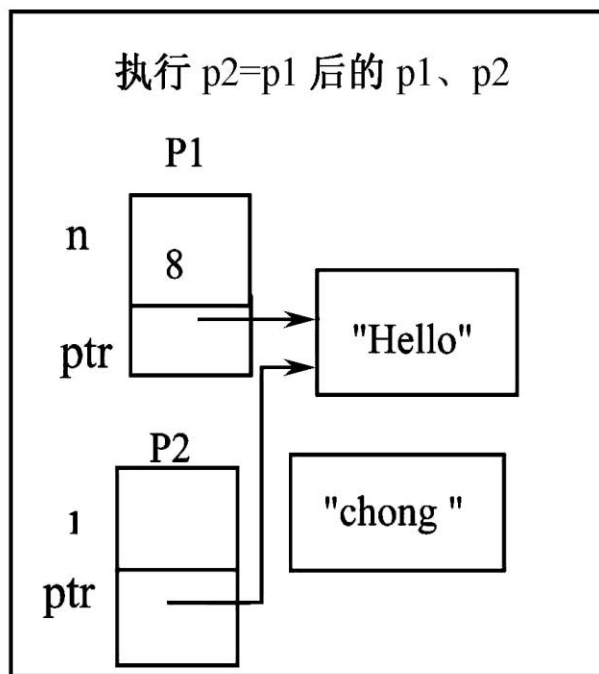
```
p2.ptr=p1.ptr; //L2
```

3.8赋值运算符/复制构造函数

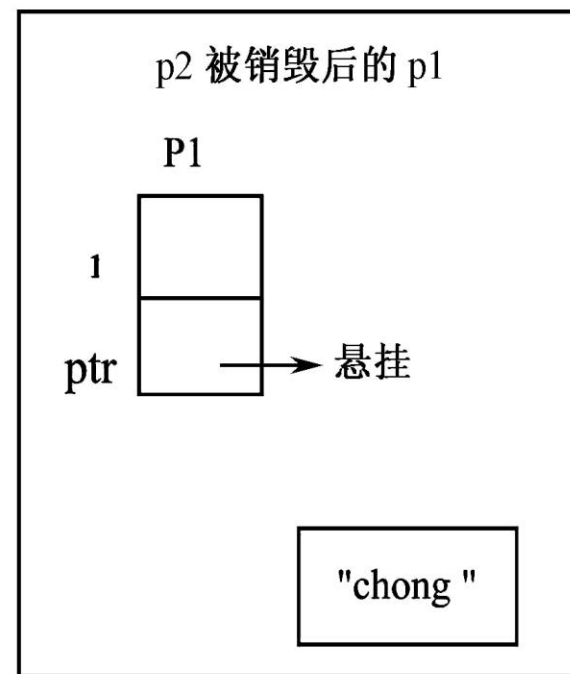
- 指针悬挂



(a)



(b)



(c)

3.8赋值运算符/复制构造函数

③ 赋值运算符/复制构造的重载方法

赋值运算符是一个二元运算符，常返回本类对象的引用，其定义形式：

```
class X{
```

```
.....
```

```
    X& operator=(const X &source, .....);
```

```
    X(const X &source, .....);    //复制构造函数
```

```
};
```

- 两个可以有多个参数。若有多个参数，则要求除第一个参数外的其余参数都要有默认值。第一个参数必须是自身类类型的引用，且通常是const类型。

3.8 赋值运算符函数

【例3-17】 定义类**String**的赋值运算符成员函数，解决赋值操作引起的指针悬挂问题。

```
class String{  
    String& operator=(const String& s); //重载赋值运算符函数  
    .....  
};  
.....  
String& String::operator=(const String& s) {  
    if(this==&s) return *this;  
    delete []ptr;  
    ptr=new char[strlen(s.ptr)+1];  
    strcpy(ptr,s.ptr);  
    return *this;  
}  
void main(){  
    .....  
}
```

现在，在执行对象的赋值操作时，将调用此重载函数完成。不会有指针悬挂问题了！

3.8赋值运算符/复制构造

2. 至少以下几种情况会调用复制构造函数。

```
class X{;
```

```
X obj1;
```

```
X obj2 = obj1;    //情况1: 调用复制构造函数
```

```
X obj3(obj1);     //情况2: 调用复制构造函数
```

```
f(X o);           //情况3: 对象作函数参数, 调用复制构造函数
```

```
X f(...) {...return t;} //情况4, 调用(移动)复制构造函数
```

```
X a[4]={obj1,obj2}    //情况5: a[0],a[1]调用复制构造函数,  
//      a[2],a[3]调用默认构造函数
```

3.8.2 复制构造函数

3. 对象定义与构造函数调用关系

— 对象定义常见形式有以下几种

① 类名 对象名;

1. 调用无参构造函数

② 类名 对象名 (实参表);

③ 类名 对象名=类名 (实参表);

④ 类名 对象名= (实参);

2, 3, 4调用参数匹配的构造函数

⑤ 类名 对象名 (已定义对象);

⑥ 类名 对象名=已定义对象;

5, 6调用复制构造函数

3.8.2 复制构造函数

4. 小结

- ① 复制构造函数与一般构造函数相同，与类同名，没有返回类型，可以重载。
- ② 复制构造函数的参数常常是**const**类型的本类对象的引用。
- ③ 在多数情况下，默认复制构造函数能够完成对象的复制创建工作，**但当类具有指针类型的数据成员时，默认复制构造函数就可能产生指针悬挂问题**，需要提供显式的拷贝构造函数。
- ④ 对复制构造函数的调用常在类的外部进行，应该将它指定为类的公有成员。

3.9 静态成员

1. 普通成员与静态成员区别

— 普通数据成员

- 每个对象拥有独立的数据成员拷贝
- 不能在确定对象之外存在

— 静态数据成员 (**static data member**)

- 被类的所有成员所共享
- 与类关联，而不与特定的对象关联
- 即便类没有任何对象时，就已经存在
- 生命期与程序相同

3.9 静态成员

2. 静态数据成员和静态成员函数的声明

```
class X{  
    类型 普通数据成员;  
    static 类型 静态数据成员名;  
    static 类型 静态成员函数名 (.....);  
    .....  
};
```

3. 静态成员的定义

(1) 静态数据成员的两种定义形式:

类型 类名::静态成员名;
类型 类名::静态成员名=初始值;

(2) 静态成员函数的定义

- 静态成员函数的定义，除了在类声明中的成员函数前面加上**static**关键字之外，其定义与普通函数没有区别。

(3) 定义静态成员的注意事项

- ① 在类外定义静态数据成员时，不能加上**static**限定词;
- ② 在定义静态数据成员时可以指定它的初始值（第2种定义形式），若定义时没有指定初值，系统默认其初值为0。

3.9 静态成员

3. 静态成员访问

(1) 通过类名访问(非静态成员不具有这种访问方式)

类名::静态数据成员名;

类名::静态成员函数名(参数表);

(2) 通过对象访问

对象名.静态成员名;

对象名.静态成员函数名(参数表)

```
class X{
public:
    int n;
    static int m;
    static int getm()
        {return m;}
};
int X::m=0;
void main(){
    X a;
    cout<<X::getm();
    cout<<a.getm();
}
```

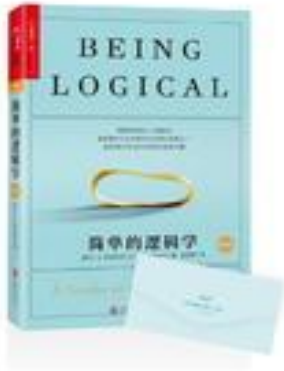

3.9 静态成员

(3) 静态成员访问的注意事项

- ① 第一种通过类名访问成员的方式是非静态数据成员访问不允许的。
- ② 同普通成员函数一样，静态成员函数也可以在类内或类外定义，还可以定义成内联函数；
- ③ 静态成员函数只能访问静态成员（包括静态的数据成员和成员函数），不能访问非静态成员。
- ④ 在类外定义静态成员函数时，不能加上**static**限定词。
- ⑤ 静态成员函数可以在定义类的任何对象之前被调用，非静态成员只有在定义对象后，通过对象才能访问。

3.9 静态成员

- **【例3-21】** 设计一个书类，能够保存书名、定价，所有书的本数和总价。



- **设计思路:**

为了实现这一要求，可以将书名、定价设计为普通数据成员，将书的本数和总价设计为静态数据成员

3.9 静态成员

- 问题分析与数据抽象

- 用Book表示书类，每本书都有书名和定价，可以抽象出数据成员**bkName**和**price**来表示它们。
- 书的本数和总价则不是每本书都有的数据，整个书类用一个变量统计就可以了，用静态成员**number**、**totalPrice**表示书的本数和总价正好符合要求。
- 为了访问数据成员，以数据成员为中心，分别为每个成员设置修改成员值的接口函数**setxx**和读取成员值的**getxx**函数，以及显示书本信息和统计结果的函数**display**。
- 每定义一本新书就增加本书和总价，每析构一本书就减少本书和总价，可以通过**构造函数**和**析构函数**实现这一要求。
- 修改书价也会引起总价的变化。

3.9 静态成员

- Book类抽象的结果
 - 其中下划线成员表示静态成员

Book	
-	bkName: string
-	<u>number: int</u>
-	price: double
-	<u>totalPrice: double</u>
<hr/>	
+	Book()
+	Book(string, double)
+	display(): void
+	getName(): string
+	getPrice(): double
+	setName(string): void
+	setPrice(): void

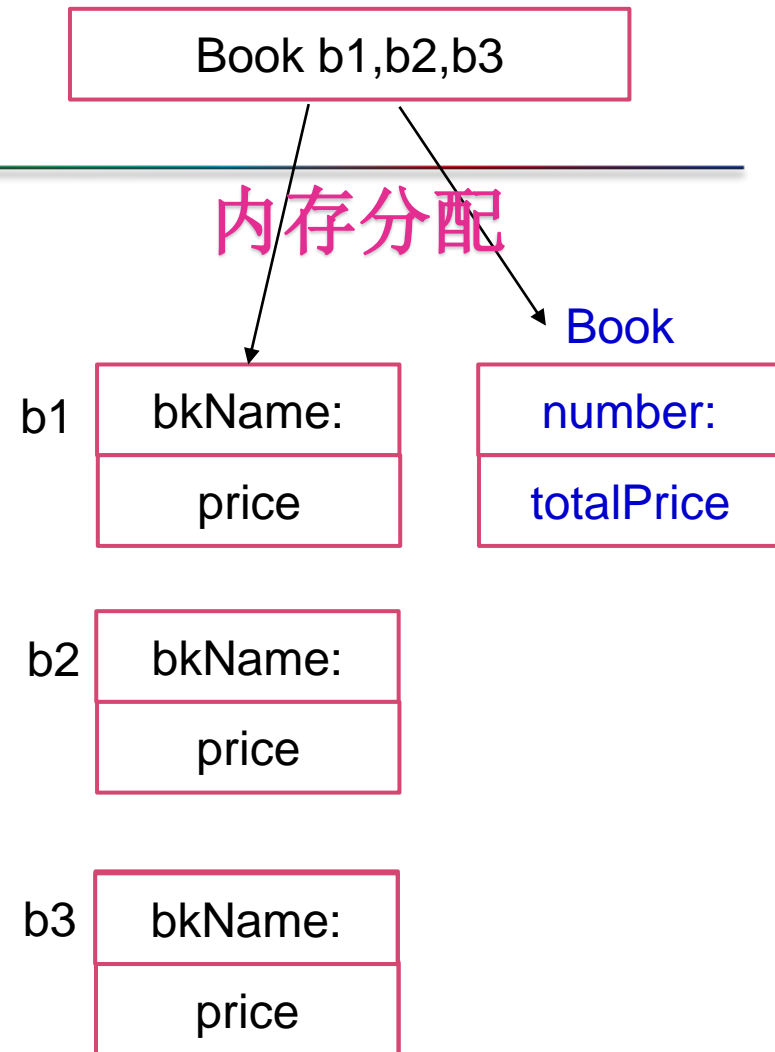
```

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

class Book {
private:
    string bkName;
    double price;
    static int number;
    static double totalPrice;

public:
    Book() { bkName = ""; price = 0; number++; };
    Book(string , double);
    ~Book();
    void setName(string bname) { bkName = bname; }
    void setPrice(double bprice) {
        totalPrice -= price;
        price = bprice;
        totalPrice += price;
    }
    double getPrice() { return price; }
    string getName() { return bkName; }
    static int getNumber() { return number; }
    static double getTotalPrice() { return totalPrice; }
    void display();
};

```



3.9 静态成员

```
Book::Book(string name,double Price) { //构造函数,
```

```
    bkName=name;
```

```
    price=Price;
```

```
    number++;
```

```
    totalPrice+=price;
```

```
}
```

```
Book::~~Book(){
```

```
    number--;
```

```
//析构一本书就减少书的本数
```

```
    totalPrice-=price;
```

```
//析构一本书就减少书的总价
```

```
}
```

//此函数仅是一个验证，表示非静态成员函数可以访问静态的数据和函数成员

3.9 静态成员

```
void Book::display(){
    cout<<"book name :"<<bkName<<" "<<"price :"  
        <<price<<endl;  
    cout<<"number:"    <<number<<" "<<"totalPrice:"  
        "<<totalPrice<<endl;  
    cout<<"call static function "<<getNumber()<<endl;  
}
```

int Book::number=0; //定义并初始化静态数据成员

double Book::totalPrice=0;

3.9 静态成员

```
void main(){
    Book b1("C++ 程序设计",32.5);
    Book b2("数据库系统原理",23);
    cout<<b1.getName()<<"\t"<<b1.getPrice()<<endl;        //L1
    cout<<b2.getName()<<"\t"<<b2.getPrice()<<endl;        //L2
    cout<<"总共: " <<b1.getNumber() <<"\t本书"            //L3
        <<"\t总价:  " <<b1.getTotalPrice() <<"\t元"<<endl;
    {
        Book b3("数据库系统原理",23);
        cout<<"总共: " <<b1.getNumber()<<"\t本书"          //L4
            <<"\t总价:  " <<b1.getTotalPrice()<<"\t元"<<endl;
    }
    cout<<"总共: " <<Book::getNumber() <<"\t本书"          //L5
        <<"\t总价:  " <<Book::getTotalPrice()<<"\t元"<<endl;
    b2.display();
}
```


3.10 this 指针

1、关于this指针

- **this**是类成员函数中用于标识调用对象自引用的**隐式指针参数**，代表调用成员函数的对象自身的地址（即成员函数所属对象的首地址:哪个对象在访问成员函数，**this**就是该对象所在内存区域的首地址），并且不允许修改，所以被指定为**const**指针，形式如下：

```
class X{.....
```

```
    int f (.....)
```

```
};
```

```
X a;
```

```
a.f(...);
```

编译器处理后

```
int f(X *const this, .....)
```

实际调用

```
f (& a, .....);
```

2、访问this指针

```
X::f (.....)
```

```
{
```

```
    this->member
```

```
}
```

3.10 this 指针

3、this指针的实现

【例3-23】 一个point类。

```
class point{
private:
    int x,y;
public:
    point(int a=0,int b=10)
        {x=a; y=b; }
    int getx( )
        { return x; }
    int gety( )
        { return y; }
    void move(int a,int b)
        { x=a;y=b; }
};

main()
{
    point p1,p2;
    p1.move(10,20)
    p2.move(3,4);
}
```

1、编译器改变类成员的定义，用额外的this指针重新定义每个类成员函数

```
inline point(point *this,int a,int b){
    this->x=a; this->y=b;
}

inline getx(point *this){return this->x;}
inline gety(point *this){return this->y;}
inline void move(point *this ,int a,int b)
    {this->x=a;this->y=b;}
```

2、编译器改变每个类成员函数的调用，加上一个额外的实参，即被调用对象的地址

```
inline void move(&p1 ,10,20)
```

```
inline void move(&p2 ,3,4)
```

3.10 this 指针

4、this指针的两种常见应用

- 使用**this**指针返回调用对象

```
class X
{
    X & f () { ..... return *this; };
    X & g () { ..... return *this; };
};
```

编译后的函数

```
X& f(X *this){..... return *this;}
X& g(X *this){..... return *this;}
X a;
a.f();
.....
```

函数调用

```
f(a){.....return a; }
```

- 使用**this**指针
区分二义性

```
class X
```

```
{
    int i;
    f (int i)
    {
        this->i = i;
    }
}
```

3.10 this 指针

5、关于this

- ① 尽管this是一个隐式指针，但在类的成员函数中可以显式地使用它。
- ② 在类X的非const成员函数里，this的类型就是X*。然而this并不是一个常规变量，不能给它赋值，但可以通过它修改数据成员的值。在类的const成员函数里，this被设置成const X*类型，不能通过它修改对象的数据成员值。
- ③ 静态成员函数没有this指针，因此在静态成员函数中不能访问对象的非静态数据成员

3.10 this 指针

6、this返回对象地址或自引用的成员函数

- 在类成员函数中，可以通过this指针返回对象的地址或引用，这也是**this的常用方式**。引用是一个地址，允许函数返回引用就意味着函数调用可以被再次赋值，即允许函数调用出现在赋值语句的左边。

【例3-24】 有日期类，设计修改其年、月、日的成员函数，测试通过this指针返回对象的指针和引用的各种情况。

在本例程的代码中，注意分析以下调用可行的原因：

d1.setYear(2007).setMonth(03).setDay(30)

3.10 this 指针

```
//Eg3-24.cpp
#include <iostream>
using namespace std;
class Tdate{
private:
    int yy,mm,dd;
public:
    Tdate(int y=2006,int m=01,int d=01);
    Tdate &setYear(int year);
    Tdate &setMonth(int month);
    Tdate *setDay(int day);
    Tdate setDate(int y,int m,int d);
    void display();
};
```

3.10 this 指针

```
Tdate::Tdate(int y,int m,int d){ yy=y;mm=m;dd=d;}
Tdate& Tdate::setYear(int year){ yy=year; return *this;}
Tdate& Tdate::setMonth(int month){ mm=month; return *this;}
Tdate* Tdate::setDay(int day){ dd=day; return this;}
Tdate Tdate::setDate(int y,int m,int d){
    yy=y; mm=m; dd=d;
    return *this;
}
void Tdate::display(){
    cout<<"addres is: "<<this<<"\t"
        <<yy<<":"<<mm<<":"<<dd<<endl;
}
```

```
void main(){
    Tdate d1,d2; //L1
    cout<<"d1 "; d1.display(); //L2
    cout<<"d2 "; d2.display(); //L3
    d1.setYear(2007).setMonth(03).setDay(30); //L4
    cout<<"d1 "; d1.display(); //L5
    d1.setDate(2000,01,10).setDay(30); //L6
    cout<<"d1 "; d1.display(); //L7
    Tdate *p; //L8
    p=d1.setDay(21); //L9
    cout<<" p "; //L10
    p->display(); //L11
    Tdate d3=d2.setYear(2006).setMonth(4); //L12
    cout<<"d3 "; d3.display(); //L13
    d1.setYear(2007).setMonth(03)=d3; //L14
    cout<<"d1 "; d1.display(); //L15
}
```


3.11 对象应用

3.11.1 成员访问操作符

操作符	.	->	*	.*	->*
作用	成员选择	指针成员选择	解引用	成员解引用	指针解引用成员选择
案例	x.m	p->m	*x	x.*m	p->*m

【例3-25】设计具有姓名、编号、年龄的简单类**Person**，能够输出和修改**Person**的编号和年龄。

```
class Person {  
public:  
    char* name = nullptr;  
    int id, int age;  
    void outData() {  
        cout<<"id : "<<id<<"\tname : "<<name<<"\tage : "<<age<<endl;  
    }  
    int modifyId(int Id) {  
        id = Id;    return age;  
    }  
    int modifyAge(int Age) {  
        age = Age;    return age;  
    }  
};
```

3.11.1 成员访问操作符

```
int main() {  
    // 1: 圆点 “.” 成员选择符的应用方法  
    Person p1;  
    p1.name = new char(10);  
    p1.id = 10001;  
    p1.age = 10;  
    strcpy(p1.name, "Tom");  
    p1.outData();  
    cout << "id: " << p1.id << "\tname: "  
        << p1.name << "\tage: " << p1.age  
        << endl;  
  
    // 2: 指针 “->” 成员选择操作符的应用  
    Person* p2;  
    p2 = new Person();  
    p2->age = 20;  
    p2->id = 1002;  
    p2->name = new char(10);  
    strcpy(p2->name, "Jack");  
  
    // 3: 解引用 “*” 操作符的应用  
    (*p2).age = 21;  
    cout<<"id: "<<(*p2).id<<"\tname: "  
        <<(*p2).name<<"\tage: "  
        <<(*p2).age<<endl;  
}
```

```
// 4:成员解引用 “.*” 的应用  
int(Person:: * P_int) = &Person::age;  
char* (Person:: * Pname) = &Person::name;  
int(Person:: * pf1)(int) = &Person::modifyAge;  
void(Person:: * pf2)() = &Person::outData;  
  
p1.*P_int = 23;  
P_int = &Person::id;  
p1.*P_int= 10004;  
(p1.*pf2)();  
(p1.*pf1)(30);  
p1.outData();  
  
// 5:指针成员解引用 “->*” 的应用  
p2->*P_int = 40;  
P_int = &Person::id;  
p2->*P_int = 10005;  
(p2->*pf2)();  
(p2->*pf1)(32);  
p2->outData();  
return 0;  
}
```

运行结果

id: 10001	name: Tom	age: 10
id: 10001	name: Tom	age: 10
id: 10002	name: Jack	age: 21
id: 10004	name: Tom	age: 23
id: 10004	name: Tom	age: 30
id: 10005	name: Jack	age: 21
id: 10005	name: Jack	age: 32

3.11 对象应用

3.11.2 对象数组和对象指针

- 类实际是一种自定义数据类型，可以用它来定义各种不同的变量（即对象）。对象数组就是用类定义的数组，它的每个元素都是对象。
- 也可以定义对象的指针，用指针指向类对象。对象指针与结构指针的访问方法相同，即用：

—>

(*指针).

两种操作符访问其所指对象的成员。

3.11.2 对象数组和对象指针

【例3-26】 对象数组和对象指针的应用。

```
#include <iostream>
using namespace std;
class point {
private:
    int x=0, y=0;                                //L1
public:
    point() { x = 1; y = 1; }                     //L2
    point(int a=0 , int b=0 ) { x = a; y = b; }    //L3
    int getx() { return x; }
    int gety() { return y; }
};
```

3.11.2 对象数组和对象指针

```
void main() {  
    //point p;                                产生二义性  
    point p1(3, 3);                          //定义单个对象  
    point p[3]={ {2,2},{3,3}, {4,4} };      //L4, 列表是3次调用构造函数的参数  
    point p2[3];                             //L5  
    point* pt;                               //L6  
    for (int i = 0; i<2; i++) {  
        cout<<"p["<<i<<"].x="<<p[i].getx()<< "\t";  
        cout<<"p["<<i<<"].y="<<p[i].gety()<< endl;  
    }  
    pt = &p1;                                //指向单个对象的指针  
    cout<<"Point pt->x:"<<pt->getx()<<endl;  //指针对象访问方法1  
    pt = p2;                                //指向对象数组的指针  
    cout<<"Point Array pt->x : "<<pt->getx()<< endl;  
    pt++;                                    //指向对象数组下一元素  
    cout<<"Point Array pt->x : "<<pt->getx()<< endl;  
    cout<<"Point (*pt).x : "<<(*pt).getx()<< endl; //指针对象访问方法2  
}
```

3.11.1 对象数组和对象指针

- 定义对象数组的几点说明

- ① 定义对象时通常要调用默认构造函数。没有定义任何构造函数的类可以定义对象数组，因为C++会为这种类自动合成一个默认构造函数。
- ② 如果一个类同时具有无参构造函数和全部参数都有缺省值的构造函数，但在定义无参对象或数组时，将产生二义性，“point p;”就属于这种情况。
- ③ 如果一个类只有需要参数的构造函数（不包括全部参数都有默认值的情况），只能采用列表方式定义数组，在列表中为数组对象提供构造函数初值，如L4语句所示。

3.11.3 向函数传递对象

1. 类类型参数的概念

- 类类型可以作为函数的参数类型，通过它向函数传递对象。在函数中访问参数对象时，**只能访问参数对象的public成员**。

2. 类类型参数的传递方式

– 值传递（对象的一个拷贝）

以按位复制的方式，将实参对象的每个数据成员的值按位拷贝到形参对象的各数据成员中。**参数传递完成后，形参与实参就没有关系了**，所以按值传递对象的方式不能修改实参对象的值。

– 地址（指针）传递

– 引用传递

3.11.3 向函数传递对象

【例3-27】按传值、传引用、传指针的方式向函数传递参数对象。

```
#include <iostream>
using namespace std;
class A{
    int val;
public:
    A(int i){ val=i;}
    int getval(){ return val; }
    void setval(int i){ val=i; }
};
void display(A ob){ cout<<ob.getval()<<endl; }
void change1(A ob){ ob.setval(50); }
void change2(A & ob){ ob.setval(50); }
void change3(A * ob){ ob->setval(100); }
```


3.11.3 向函数传递对象

```
void main(){
    A a(10);
    cout<<"Value of a before calling change  ----";
    display(a);
    change1(a);
    cout<<"Value of a after calling change1()----";
    display(a);
    change2(a);
    cout<<"Value of a after calling change2()----";
    display(a);
    change3(&a);
    cout<<"Value of a after calling change3()----";
    display(a);
}
```

分析输出结果

3.11.3 向函数传递对象

3.对参数对象的访问

- ① 普通函数（非类成员）接收参数对象后，在函数体内必须按照访问权限访问对象成员，即只能访问对象的公有成员。
- ② 类成员函数可以访问本类参数对象的私有、保护、公有成员。

3.11.4 类对象成员

1、类对象成员的基本知识

- 类的数据成员一般都是基本数据类型，但也可以是结构、联合、枚举之类的自定义数据类型，还可以是其他类的对象。
- 如果用其他类的对象作为类的成员，则称之为对象成员。
- 类对象作成员的形式如下：

class X{

 类名1 成员名1;

 类名2 成员名2;

 类名n 成员名n;

};

3.11.4 类对象成员

2、对象成员初始化

- 拥有对象成员类必须对对象成员进行初始化。
- 对象成员的初始化方式包括类内初始化或构造函数初始化列表
- 当对象成员所在类有类内初始值或默认构造函数（包括系统自动生成的默认构造函数）时，可以省略对象成员的初始化代码，编译器会自动调用它们。
- 若对象成员没有类内初始值，也没有默认构造函数，就必须在构造函数初始化列表中显式初始化对象成员，否则会产生错误。

3.11.4 类对象成员

3、在构造函数初始化列表中初始化对象成员

包含对象成员类的构造函数的定义形式：

```
X::X(参数表0.....): 成员名1 (参数表1) ,.....成员名  
    n(参数表n)  
{  
    构造函数体  
}
```

- 参数表*i*(*i*为1到*n*)给出了初始化对象的成员所需要的数据，它们一般来自参数表0

3.11.4 类对象成员

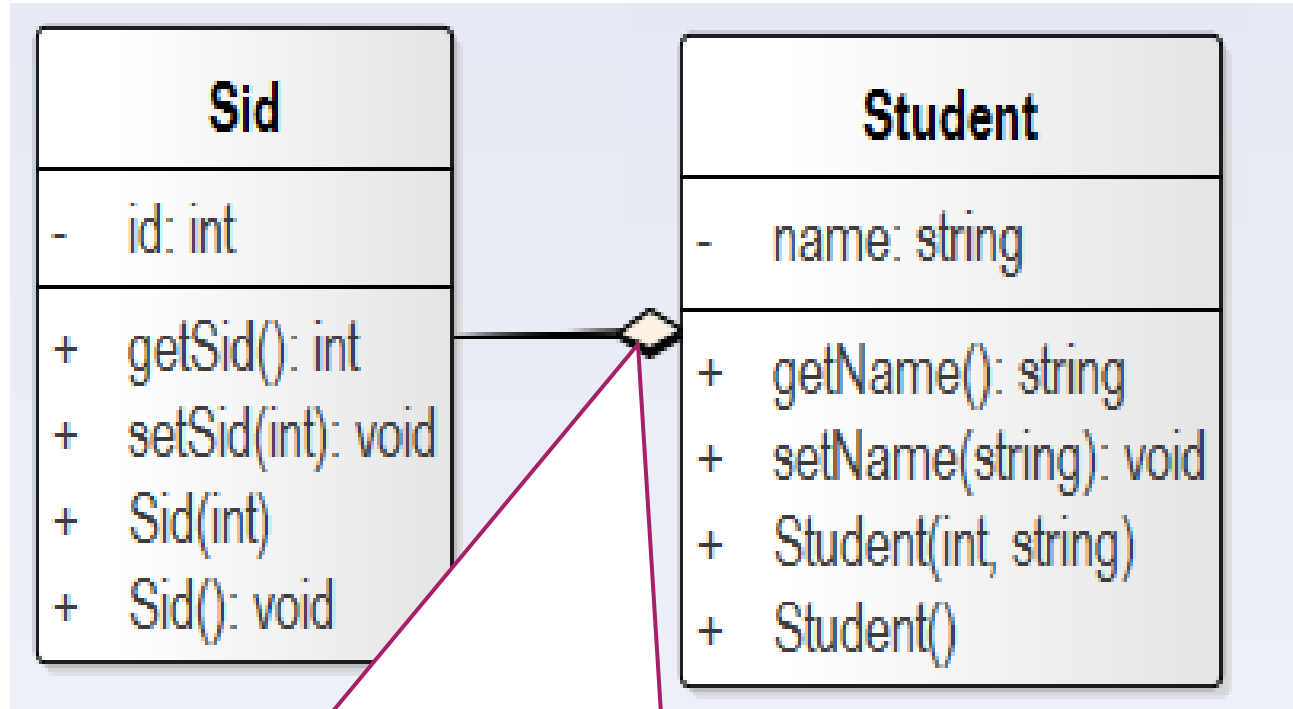
【例3-28】 设计ID类能够完成学生学号的管理，设计学生类Student，完成学生学号和姓名的管理。

问题分析与抽象：

- 本例主要用来探讨对象成员的初始化和应用问题。
- ID类只用于管理学号的输入和修改，用数据成员id表示学号，setSid和getSid修改和返回学号；
- 学生类Student有学号和姓名，用数据成员sid和name表示，其中sid已由ID类实现了，可以通过类成员引用其功能，因此Student只需实现name管理的问题，但要考虑对象成员sid的初始化问题，**必须在Student构造函数初始化列表中对sid进行初始化**

3.11.4 类对象成员

问题域类的抽象结果



Sid和**Student**的UML图。图中的棱形连接了**Sid**和**Student**，表示两类之间具**聚合关系**，棱形所在方是整体，包含有另一方的一个或多个对象

3.11.4 类对象成员

```
//Eg3-26.cpp
```

```
#include <iostream>
```

```
#include<string>
```

```
using namespace std;
```

```
class Sid {
```

```
public:
```

```
    ~Sid() { cout << "Sid des..." << id << endl; };
```

```
    Sid(int sid) :id(sid) { cout << "Sid cons..." << id << endl; }
```

```
    int getSid() { return id; }
```

```
    void setSid(int sid) { id = sid; }
```

```
private:
```

```
    int id;
```

```
};
```

对象成员初始化列表

3.11.4 类对象成员

```
class Student {
```

```
public:
```

```
    Sid m_sid;
```

```
    //L1
```

```
    //Sid m_sid = 9818;
```

```
    //L2 类内初始值
```

```
    11C++
```

```
    //Sid m_sid = Sid(9818);
```

```
    //L3 类内初始值
```

```
    11C++
```

```
    ~Student() {
```

```
        cout << "Stu des.." << name
```

```
            << "\t" << m_sid.getSid() << endl;
```

```
    }
```

```
    Student(string sname,int stuid): m_sid(stuid),name(sname) {
```

```
        cout << "Stu con.." << name << "\t"
```

```
            <<m_sid.getSid() << endl;
```

```
    }
```

对象成员也可以采用L2或L3的初始化方式

3.11.4 类对象成员

```
    string getName() { return name; }
    void setName(string sname) { name = sname; }
private:
    string name;
};
void main() {
    Student s("Randy", 9818);
    s.setName("tom"); //L5
    cout <<s.getName()<<"\t"
         <<s.m_sid.getSid()<<endl; //L6
}
```

3.11.4 类对象成员

4、对象成员访问

类对象成员同样遵守**public**、**private**、**protected**访问权限的约束限定。

— 例如，如果把**sid**设置成**student**类的私有成员：

```
class Student{  
private:  
    Sid sid;  
    .....  
}
```

```
Student s1("Tom",1811);
```

```
s1. sid.getSid(); //错误，sid是Student类的私有成员  
//它的所有成员函数都是private
```

3.11.4 类对象成员

5、对象成员构造次序

C++对象成员的构造次序与它们在类中的声明次序相同，与它们在构造函数初始化列表中的次序无关。

【例3-29】 对象成员的构造次序

```
#include <iostream>
using namespace std;
class A {
    int a;
public:
    A(int i = 1) :a(i)
    { cout << "constructing A:" << a << endl; }
};
```

```
class B {  
    int b;  
  
public:  
    B(int i) :b(i)  
        { cout << "constructing B:" << b << endl; }  
};
```

```
class C {  
    A a1, a2;  
    B b1, b2;  
  
public:  
    C(int i2, int i3, int i4) :b1(i3), b2(i4), a2(i2) {}  
};  
  
void main() {  
    C x(2, 3, 4);  
}
```

运行结果如下，分析各行输出来源？

constructing A:1
constructing A:2
constructing B:3
constructing B:4

3.12 类的作用域和对象的生存期

1、类的作用域

- 类构成了一种特殊的作用域，称为类域。类域是指类定义时的一对花括号所括起来的范围，形式如下：

```
class X{           //类域开始
    .....
};                //类域结束
```

- 类域范围内的成员之间（不用像函数参数那样需要传递）可以互相访问，不受成员访问控制权限的限定
- 类外的函数则只能访问类的公有成员。

3.12 类的作用域和对象的生存期

例：简单的类域示例

```
class X{
```

//X的类域开始了

```
    int a,b;
```

```
    float c;
```

```
public:
```

```
    int f1(int i) {
```

```
        int a,y;
```

```
        a=i;
```

```
        X::a=9;
```

```
        return a*a;
```

```
    }
```

//X最外层{}所框定的范围就是X的类域

//同一类域中的函数和数据可以相互访问

```
    void f2(int j) {
```

```
        //y=1;
```

```
        b=f1(j);
```

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

```
    }
```

//错误，y未定义，y只在f1内有效

```
};
```

//X的类域结束了，在后面就只能访问X的公有成员了

```
X n, *p;
```

```
n.f1(2);
```

//正确，在类域外访问类的公有成员

```
n.a=2;
```

//错误，在类域外不能访问类的私有成员

```
p->f1(3);
```

3.12 类的作用域和对象的生存期

2、对象的生存期

(1) 对象的生存期概念

- 是指对象从它被创建开始到被销毁之间的时间。

(2) 生存期类型

- **静态生存期**是指对象具有与程序运行期相同的生存期，这类对象一旦被建立后，它将一直存在，直到程序运行结束时才被销毁。
- **动态生存期**是指局部对象的生存期，局部对象具有块作用域，它的生存期是从它的定义位置开始，遇到离它最近的“}”就结束了。
- 全局对象和静态对象具有静态生存期。

3.12 类的作用域和对象的生存期

3. 各类对象的生存期

- ① 生存期与对象的构造次序和销毁次序密切相关。
- ② 局部对象和静态对象的构造次序与它们在块中的声明次序相同，即在块中先声明的就先构造，块即对象定义所在的一对{}所框定的代码区域。
- ③ 所有的全局对象在main之前构造，在main结束之后销毁。
- ④ 对象数据成员（包括对象成员）的构造次序与其在类中的声明次序相同，而与它们在构造函数的初始化列表中的次序无关。
- ⑤ 在对象生存期结束时，具有相同生存期的对象将按与构造的相反次序销毁。
- ⑥ 非静态对象的生存期与其作用域是一致的，而静态对象的生存期则长于其作用域，程序结束时静态对象的生存期才结束。

3.12 类的作用域和对象的生存期

【例3-30】 对象的生存期分析。

//Eg3-30.cpp

#include <iostream>

using namespace std;

class X{

public:

X(int ii = 1){ i=ii; cout << "X (" << ii << ") created" << endl;}

~X(){ cout << "X (" << i << ") destroyed" << endl; }

private:

int i ;

};

3.12 类的作用域和对象的生存期

```
class Z{
public:
    Z():x3(3),x2(2){ cout << "Z created" << endl; }
    ~Z(){ cout << "Z destroyed" << endl; };
private:
    X x1, x2, x3;
};
```

X a(200);

//a的生命期开始了

```
void main (void){
```

```
    Z z;           //z的生命期开始了，且其成员对象x1\x2\x3的生命期也开始了，且先于它
    {
```

```
        X c(100);
```

//c的生命期开始了

```
        static X b(50);
```

//b的生命期开始了

```
    }
```

//c的生命期结束了

```
}
```

//z、x3、x2、x1、b的生命期依次结束

//main()函数结束后，a的生命期才结束

3.13 友元

1、友元的概念

- 类的封装性使得使该类外部的函数只能访问其public成员。但类可以授予指定函数特权，让它可以访问该类的所有成员——包括public、protected、private类型的成员。这个获得了特权的函数就是友元。

2、友元函数的声明与定义

```
class X{
```

```
.....
```

```
friend T f(...);    //声明f为X类的友元，f的形参通常是X类的对象
```

```
};
```

```
.....
```

```
T f(...) { ..... }
```

```
//友元不是类成员函数，定义时不能用“X::f”限定函数名
```

3.13 友元

【例3-31】 Point是处理屏幕坐标点的类，为它设计计算两点之间距离的友元函数。

```
#include <iostream>
#include <cmath>
using namespace std;
class point{
private:
    int x,y;
    friend int dist1(point p1,point p2); //声明dist1为point类的友元
public:
    point(int a=10,int b=10){ x=a; y=b; }
    int getx( ){ return x ;}
    int gety( ){ return y; }
};
```

3.13 友元

```
int dist1(point p1,point p2){  
    double x=(p2.x-p1.x);  
    double y=(p2.y-p1.y);  
    return sqrt(x*x+y*y);  
}  
int dist2(point p1,point p2){  
    double x=p2.getx()-p1.getx();  
    double y=p2.gety()-p1.gety();  
    return sqrt(x*x+y*y);  
}  
void main(){  
    point p1(2,5),p2(4,20);  
    cout<<dist1(p1,p2)<<endl;  
    cout<<dist2(p1,p2)<<endl;  
}
```

//友元可以直接访问对象的私有成员

//dist2是普通函数

//普通函数只能访问对象的公有成员

3.13 友元

3、友元类

- 一个类可以是另一个类的友元，友元类的所有成员函数都是另一个类的友元函数，能够直接访问另一个类的所有成员（包括**public**、**private**和**protected**）。

【例】 通过友元类的成员函数直接访问对象的私有成员。

//Eg.cpp

```
#include <iostream>
```

```
using namespace std;
```

```
class A{
```

```
private:
```

```
    int x,y;
```

```
public:
```

```
    A(int i,int j){x=i;y=j;}
```

```
    int getX(){return x;}
```

```
    int getY(){return y;}
```

```
    friend class B;
```

```
};
```

//声明类B是类A的友元类

3.13 友元

```
class B{
private:
    int z;
public:
    int add(A a){ return a.x+a.y+z; }
    int mul(A a){ return a.x*a.y*z; }
    B(int i=0){ z=i; }
};

void main(){
    A a(2,3);
    B b(4);
    cout<<b.add(a)<<endl;
    cout<<b.mul(a)<<endl;
}
```

//A类对象作参数

//A类对象作参数

//输出9

//输出24

3.13 友元

4、友元成员函数

- 可以指定类的某个成员函数是另一个类的友元，也就是**友元成员函数**。友元成员函数可以直接访问另一个类的私有成员或保护成员，但该类不是友元的成员函数就只能通过公有成员函数访问其他类的私有和保护成员。
- 友元成员函数的定义步骤有讲究

【例】有两个类**A**和**B**，将类**A**的成员函数**sum** 定义成类**B**的友元成员函数，使它能够计算两个类数据成员的总和。

3.13 友元

```
//CH3-.cpp
```

```
#include <iostream>
```

```
using namespace std;
```

```
class B;
```

//L1: 前向引用声明

```
class A{
```

```
private:
```

```
    int x,y;
```

```
public:
```

```
    A(int i,int j){x=i;y=j;}
```

```
    int sum(B b);    //在此只能声明sum(), 其定义须在class B定义之后
```

```
};
```

3.13 友元

```
class B{
private:
    int z;
public:
    B(int i=0){z=i;}
    friend int A::sum(B b);
};
int A::sum(B b){return x+y+b.z;}
void main(){
    A a(2,3);
    B b(4);
    cout<<a.sum(b)<<endl;
}
```

//sum()的定义只能在B定义之后

//输出9

3.13 友元

5. 友元使用注意事项

- ① 在类域中的函数原型前加上关键字**friend**，就将该函数指定为该类的友元了。关键字**friend**用于声明友元，它只能出现在类的声明中。
- ② 友元函数**并非类的成员函数**，所以它不受**public**、**protected**、**private**的限定，无论将它放在**public**区，或者**protected**区，还是**private**区，都是完全相同的。
- ③ 友元**不具逆向性和传递性**。即，若**A**是**B**的友元，并不表示**B**是**A**的友元（除非特别声明）；若**A**是**B**的友元，**B**是**C**的友元，也不能代表**A**是**C**的友元（除非特别声明）。
- ④ 友元使编程更简洁，程序运行效率也更高，但它可以直接访问类的私有成员，破坏了类的封装性和信息隐藏。不建议多用友元。

3.14 编程实作：接口与实现的分离

1、类的常见组织方式

- 类的接口

- 即指类的声明，常保存为与类同名的.h头文件

- 实现

- 是指类的成员函数的定义。放在一个与类同名的源程序中（即扩展名为.cpp的文件）。

【例3-32】 建立一个整数堆栈类**stack**，栈的默认大小为10元素，能够完成数据的入栈和出栈处理。将类的声明（即接口）存放在单独的头文件**Stack.h**中

3.14 编程实作：接口与实现的分离

(1) 问题分析

- 堆栈是计算机领域中广泛应用的一种数据存取技术，是一种按顺序存取的数据结构，类似于生活中按层次存放衣服的箱子，后放入的衣服压在上次入箱的衣服务上面，称为**入栈（push）**；取出衣服时每次都只能取最上层的衣服，称为**出栈（pop）**。

(2) 数据抽象

- 可以用数组、链表之类的数据存取技术实现堆栈，通过限定只能在数组或链表的一端进行数据读写就能够实现。
- 本例将堆栈抽象成**Stack**类，用数组**data**保存堆栈的数据，为了实现只在数组一端进行读写数据的操作，设置**top**指针指示栈顶元素，每次只能够读出它指身的元素，每读出一个数据，**top**就向下移动一个元素位置；同样，每次保存数据时，只能保存在**top**指向的位置，每存入一个数据，**top**就向上移动一个位置，再设置**maxSize**表示数组的最大下标，表示堆栈容量。

3.14 编程实作：接口与实现的分离

Stack抽象结果

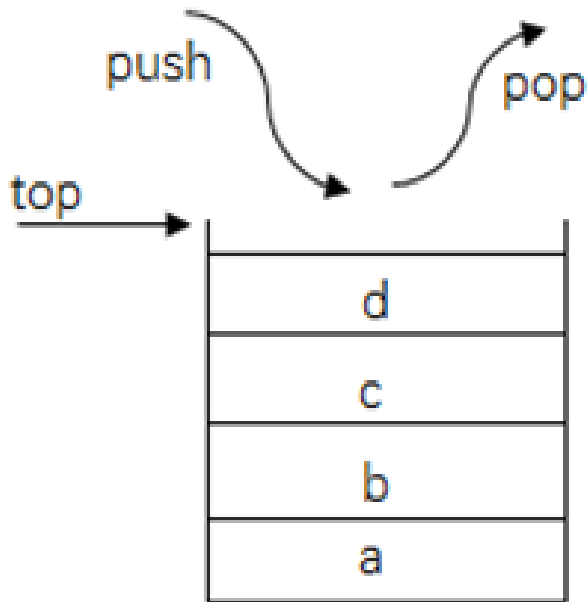


图 3-14 堆栈示意图

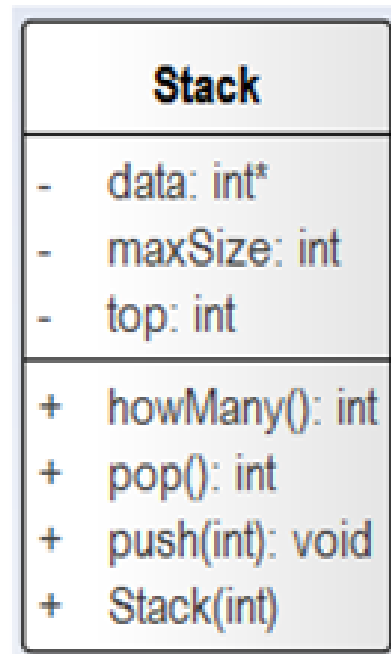


图 3-15 堆栈类图

3.14 编程实作：接口与实现的分离

(3) stack类的接口

//堆栈stack的头文件：Stack.h

```
#ifndef Stack_h
```

```
#define Stack_h
```

```
class Stack{
```

```
private:
```

```
    int *data;
```

```
    int count;
```

```
    int size;
```

```
public:
```

```
    Stack(int stacksize=10);
```

```
    ~Stack();
```

```
    void Push(int x);
```

```
    int Pop();
```

```
    int howMany();
```

```
};
```

```
#endif
```

//存放栈数据

//存放栈顶指针

//栈的容量

//构造函数建立具有10元素的默认栈

//元素入栈

//元素出栈

//判定栈中有多个元素

//堆栈stack的源文件: stack.cpp

#include "stack.h"

#include <iostream>

using namespace std;

Stack::Stack(int stacksize){

if(stacksize>0){

size=stacksize;

data=new int[stacksize];

for(int i=0;i<size;i++) data[i]=0;

}

else { data=0; size=0; }

count=0;

}

Stack::~~Stack(){ delete []data;}

void Stack::Push(int x){

if(count<size){ data[count]=x; count++; }

else{

cout<<"堆栈已满, 不能再压入数据: "<<x<<endl;

}}

int Stack::Pop(){

if(count<=0){

cout<<"堆栈已空! "<<endl;

exit(1);

}

count--;

return data[count];

}

int Stack::howMany(){

return count;

}

//包含头文件

//push和pop都用到了cout, 所以包含此头文件

Stack类的实现

//堆栈操作失败, 退出程序!

3.14 编程实作：接口与实现的分离

，把**stack.h**和**stack.cpp**复制到**stackmain.cpp**所在的目录中。

//应用栈类的主程序： **stackmain.cpp**

```
#include "stack.h"
#include <iostream>
using namespace std;
void main(){
    Stack s1;
    s1.Push(1);
    s1.Push(12);
    s1.Push(32);
    int x1=s1.Pop();
    int x2=s1.Pop();
    int x3=s1.Pop();
    cout<<x1<<"\t"<<x2<<"\t"<<x3<<endl;
}
```

应用**Stack**实现堆
栈操作