

第1章

类和对象



对象和类

对象

- ❖ 在现实世界中，对象就是某个具体的事物或现象，如，一名学生是一个对象、一辆汽车是一个对象、一堂课是一个对象，等等。
- ❖ 每个对象都有其对应的特征和行为，如：
汽车对象有其外部颜色、发动机编号、车长、车宽等特性；
汽车具有能够前进、后退等能力的行为。
- ❖ 在计算机世界中，是用程序技巧来模仿现实世界中的对象，就是用数据与程序代码来模仿出对象的各种特征和行为。
- ❖ 对象就是属性和方法结合在一起所构成的不可分割的独立实体，即“数据加程序代码”。

类

- 具有相同属性和行为的一组对象，就称为类（Class），是对具有共同特征的事物进行统一描述。
- 例如：zhang和li两位同学都是“学生”，都具有学号、姓名、性别等属性，都具有可以告诉外人姓名、生日等的功能，所以，可将所有同学抽象成一个“学生类”。

类名
属性
操作

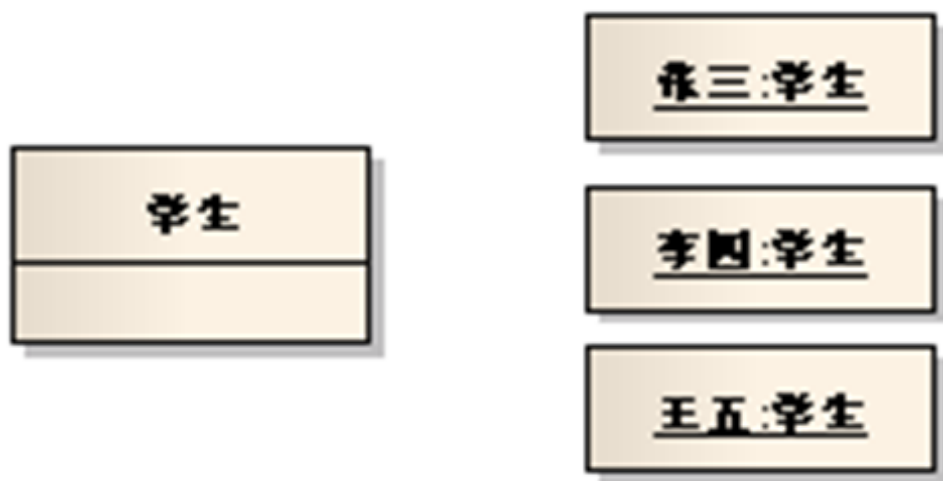
Student
No: String name: String sex: String birthdata: Date class: String
getNo():String setno(No:String) getname():String setname(name:String) getsex():String setsex(sex:String) getbirthdate():Date setbirthdate(birthdate:Date) getclass():String setclass(class:String)



类与对象的区别与联系

- 类是一个抽象的概念，相当于一个模具；而对象是一个类中的某个存在的、客观的具体实体，是类的一个实例（Instance），相当于是用模具制造出来的产品。例如：

所有的学生都具有共同的特征和行为，可以将它们抽象成一个Student类(学生类)；而一个具体的学生“张三”、“李四”、“王五”等都是Student类的一个个实体，是Student这个类的对象。



面向对象程序设计的实质

❖ 面向对象的程序设计围绕类的定义和类的使用展开的

面向对象程序设计中最突出的特征

■ 封装性

- 封装的单位是对象，对象属于某一个类。封装前需要做好数据抽象和功能抽象的工作，明确一个类中有哪些数据成员和成员函数，哪些成员需要隐藏信息，哪些成员对外公开，提供哪些对外接口

■ 继承性

- 继承是面向对象程序设计提高代码重用性的重要措施。继承使一个类（称为基类或父类）的数据成员和成员函数能被另一个类（称为派生类或子类）重用。

■ 多态性

- 多态性是指一种行为对应多种不同的实现方法。例如打排球，打乒乓球。多态性的意义在于用同一个接口实现不同的操作

内容安排

❖ 1.1 类

❖ 1.2 对象

❖ 1.3 构造函数与析构函数

❖ 1.4 对象成员、对象数组与堆对象

❖ 1.5 静态成员

❖ 1.6 友元函数和友元类

❖ 1.7 常对象与常成员

1.1 类

- ❖ 1.1.1 类的定义
- ❖ 1.1.2 类成员的访问控制
- ❖ 1.1.3 成员函数的实现

1.1.1 类的定义

- ❖ 从编程的角度：类是一种用户自定义的数据类型，称为类类型。
- ❖ 内容：类中有数据（即数据成员），及对数据进行操作的函数（即成员函数或函数成员）。
- ❖ 代码书写：包括类说明、类实现两部分。
 - 说明部分提供了对该类所有数据成员和成员函数的描述；
 - 实现部分则提供了所有成员函数的实现代码。

关键字 有效的C++标识符，不能是关键字，通常首字母大写

class 类名

{

public:

数据成员或成员函数声明

protected:

数据成员或成员函数声明

private:

数据成员或成员函数声明

} 类说明

}; ————— 必须有一个分号，表示类定义的结束

<各成员函数的实现代码> ————— 类实现

1.1.2 类成员的访问控制


- ❖ **public**、**protected**、**private**是类成员的访问控制关键字（访问权限修饰符或访问控制修饰符），可以以任何顺序出现。
- ❖ 它们决定类成员的访问属性（也叫访问权限）
 - **private**（私有属性）
 - 在类外不能被访问，只有类中的成员函数才能访问**private**部分的数据成员和成员函数。
 - **protected**（保护属性）
 - 只有类的成员函数及其子类（派生类）可以访问。
 - **public**（公有属性）
 - 可以被程序中的任何函数访问。**public**部分的成员多为成员函数，用来提供一个与外界接口，外界只有通过这个接口才可以实现对**private**部分成员的访问。
- ❖ 当未指明时，默认**private**，不建议使用。

❖ 例1-0: 下例中定义了一个描述点的类。

```
class Point
{
    public:
        void setxy(int a,int b); //设置坐标
        void displayxy(); //显示坐标
    private:
        int X,Y; //坐标
};
```


```
Void Point::setxy(int a,int b)
{
    X=a;
    Y=b;
}
```

```
void Point::setxy(int a,int b)
{
    int X;
    int Y;
    X=a;
    Y=b;
}
```



```
void Point::displayxy()
{
    cout<<X<<" "<<Y<<endl;
}
```

```
void Point::displayxy()
{
    int X,Y;
    cout<<X<<" "<<Y<<endl;
}
```



1.1.3 成员函数的实现

- ❖ 成员函数的实现，可放在类体内；也可在类体内给出函数声明，类体外给出实现。
- ❖ 放在类体内定义的函数被默认为内联函数，而放在类体外定义的函数是一般函数。

```
class Point
{
public:
    void setxy(int x,int y)
    {
        X=x;
        Y=y;
    }
    void displayxy();
private:
    int X,Y;
};
```

定义在类体中给出

```
class Point
{
public:
    void setxy(int x,int y);
    void displayxy();
private:
    int X,Y;
};
void Point::setxy(int x,int y)
{
    X=x;
    Y=y;
}
```

函数返回值的类型 类名::成员函数名 (参数说明)
{ 函数体 }

定义在类体外给出

1.1.3 成员函数的实现

❖ C++源代码组织：提倡分成两个文件：

一部分是类的说明（.h文件），另一部分是类的实现（.cpp文件）。

❖ 类的说明：仅包括类的所有数据成员以及成员函数的函数原型，放在头文件（例如point.h）中，便于共享使用。

❖ 类的实现，即成员函数实现放在与头文件同名的源文件（例如point.cpp）中，便于修改。

❖ 有利于为一个类的同一界面提供不同的内部实现。

1.1.3 成员函数的实现

//point.h

class Point

{

public:

void setxy(int,int); //设置坐标

void displayxy(); //显示坐标

private:

int X,Y; **//坐标**

};

//point.cpp

#include "point.h"

#include <iostream>

using namespace std;

void Point::setxy(int x,int y)

{

X=x;

Y=y;

}

void Point::displayxy()

{

cout<<"("<<X<<","<<Y<<)"<<endl;

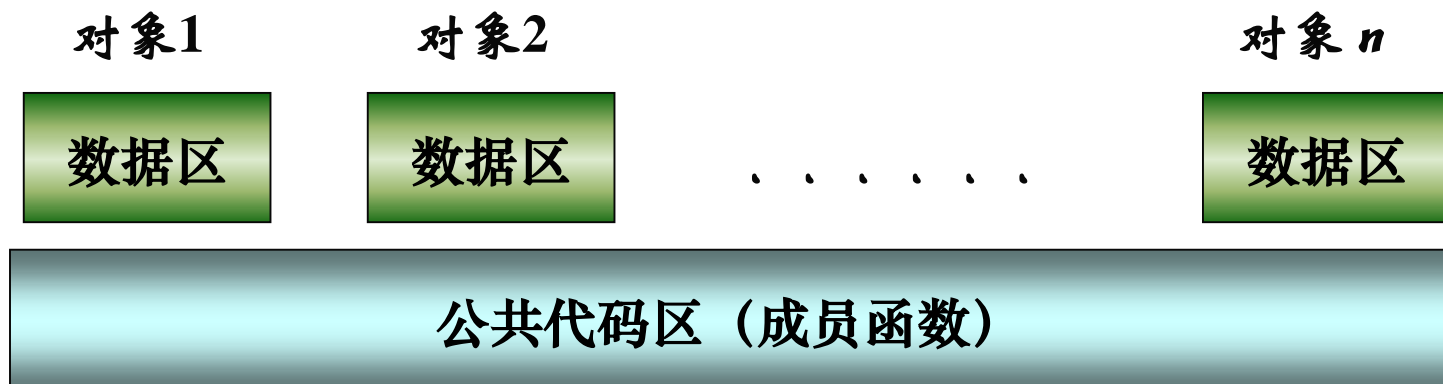
}

1.2 对象

- ❖ 1.2.1 对象的声明（定义）
- ❖ 1.2.2 对象的指针
- ❖ 1.2.3 对象成员的访问
- ❖ 1.2.4 `this` 指针
- ❖ 1.2.5 对象的作用域与生存期

1.2.1 对象的声明（定义）

- ❖ 面向对象思想：类是抽象的概念（模子），对象是具体的概念（实体），对象是类的某一特定实体（也称实例）。
- ❖ 程序设计角度：对象是类类型的变量。
- ❖ 对象的声明格式：
 - 类名 对象名（参数表）；



每个对象占用各自的存储单元，每个对象都各自具有该类的一套数据成员（静态成员除外），而成员函数是所有对象共有的。

1.2.1 对象的声明（定义）

//main.cpp

#include "point.h"

int main()

{

Point p1, p2;

p1.setxy(1,1);

p1.displayxy();

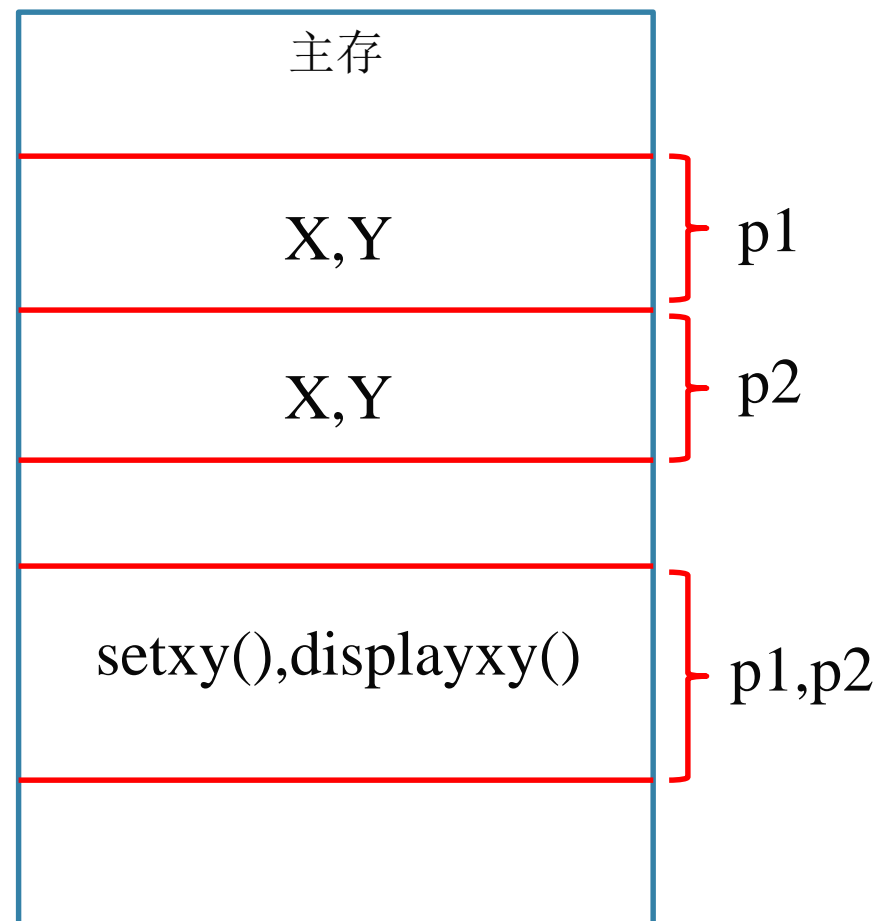
p2.setxy(10,20);

p2.displayxy();

return 0;

}

Point p1,p2; //定义对象p1和p2



1.2.1 对象的声明（定义）

```
//main.cpp
```

```
#include "point.h"
```

```
int main()
```

```
{
```

```
    Point p1, p2;
```

```
    p1.setxy(1,1);
```

```
    p1.displayxy();
```

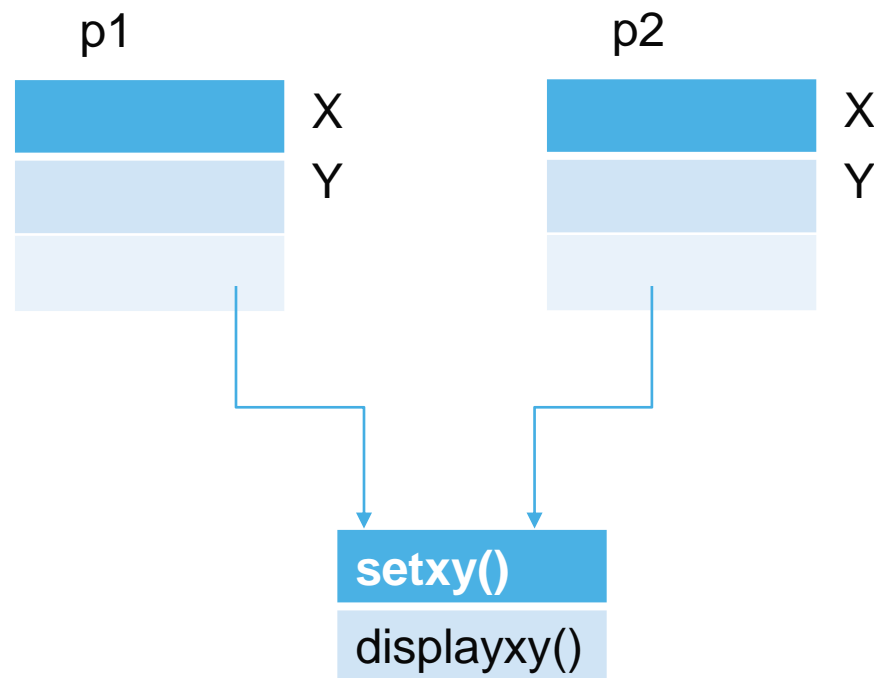
```
    p2.setxy(10,20);
```

```
    p2.displayxy();
```

```
    return 0;
```

```
}
```

Point p1,p2; //定义对象p1和p2



1.2.2 对象的指针

指针变量的声明格式:

类名 *指针变量名列表;

Point p1; //定义**Point**类的对象**p1**

Point *p2; //定义指向**Point**类的指针

p2=&p1; //使指针变量**p2**指向对象**p1**

1.2.3 对象成员的访问

- 格式1：对象名.成员名
- 格式2：指针变量名->成员名
- 格式3：(*指针变量名).成员名

❖ 【例1-1】对象成员的访问

1.2.3 对象成员的访问

Student.cpp文件

例1-1 Student.h文件

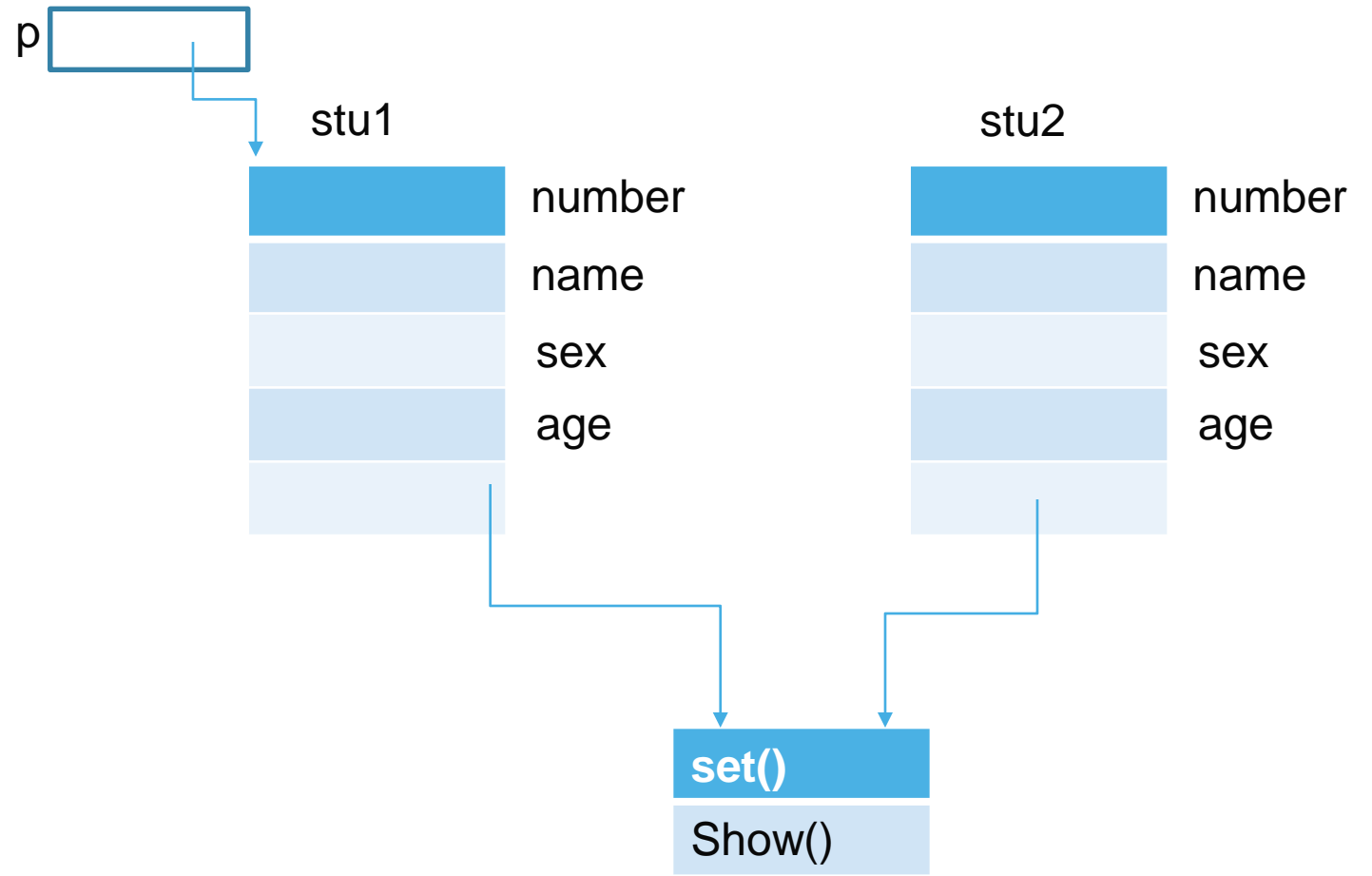
```
#include <iostream>
using namespace std;
class Student
{
private:
    int number;
    char name[20];
    char sex;
    int age;

public:
    void set(int a, string b, char c, int d);
    void show();
};
```

```
#include "Student.h"
#include <iostream>
#include <string>
using namespace std;
void Student::set(int a, char b, char c, int d)
{
    number = a;
    strcpy_s(name, b);
    sex = c;
    age = d;
}
void Student::show()
{
    cout << number << ", " << name << ", " << sex << ", " <<
    age << endl;
}
```

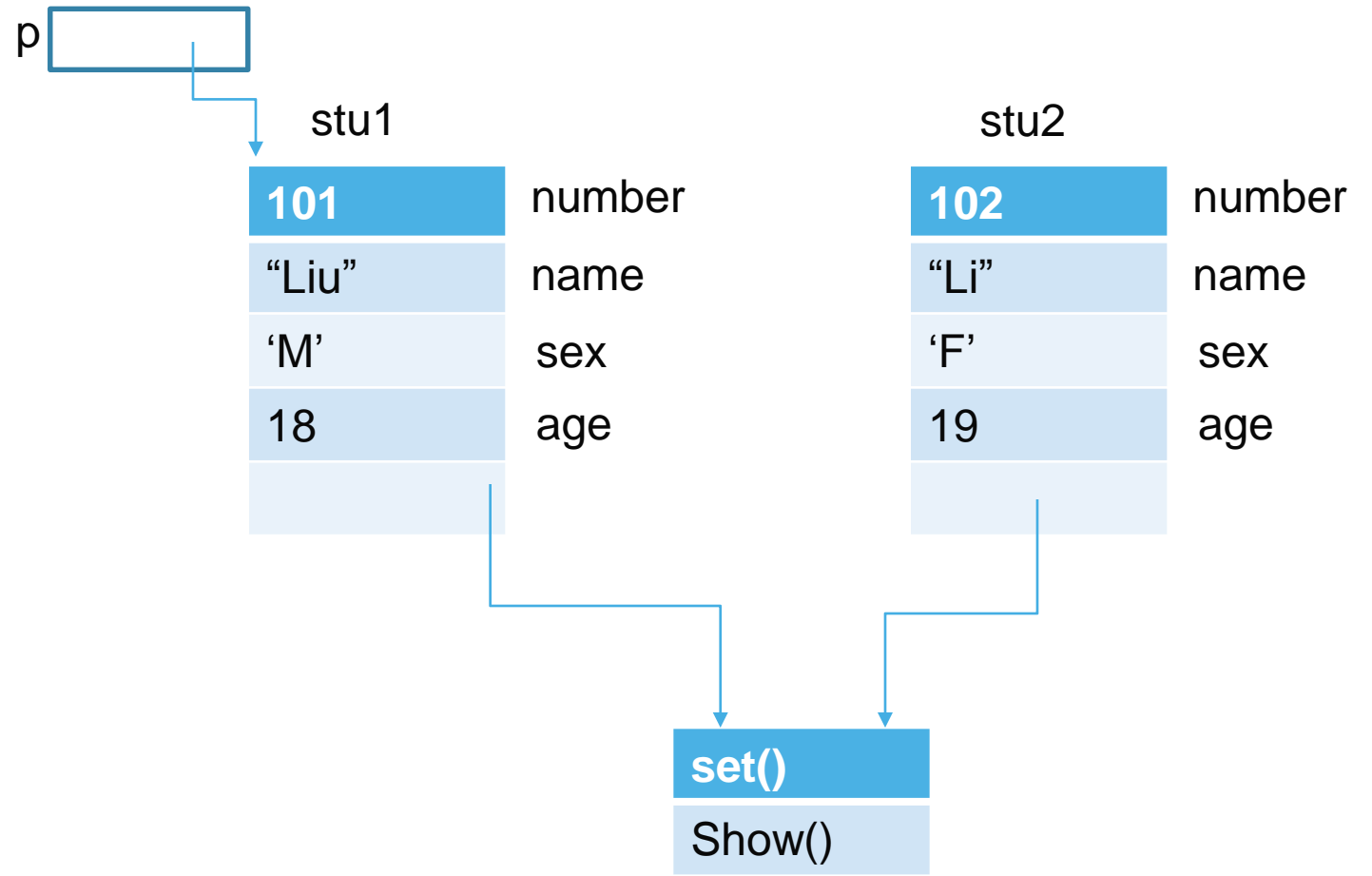
1.2.3 对象成员的访问

```
int main()
{
    Student stu1, stu2;
    Student *p;
    p = &stu1;
    stu1.set(101, "Liu", 'M', 18);
    (*p).show();
    stu2.set(102, "Li", 'F', 19);
    p = &stu2;
    p->show();
    stu1.show();
    stu2.show();
    return 0;
}
```



1.2.3 对象成员的访问

```
int main()
{
    Student stu1, stu2;
    Student *p;
    p = &stu1;
    stu1.set(101, "Liu", 'M', 18);
    (*p).show();
    stu2.set(102, "Li", 'F', 19);
    p = &stu2;
    p->show();
    stu1.show();
    stu2.show();
    return 0;
}
```



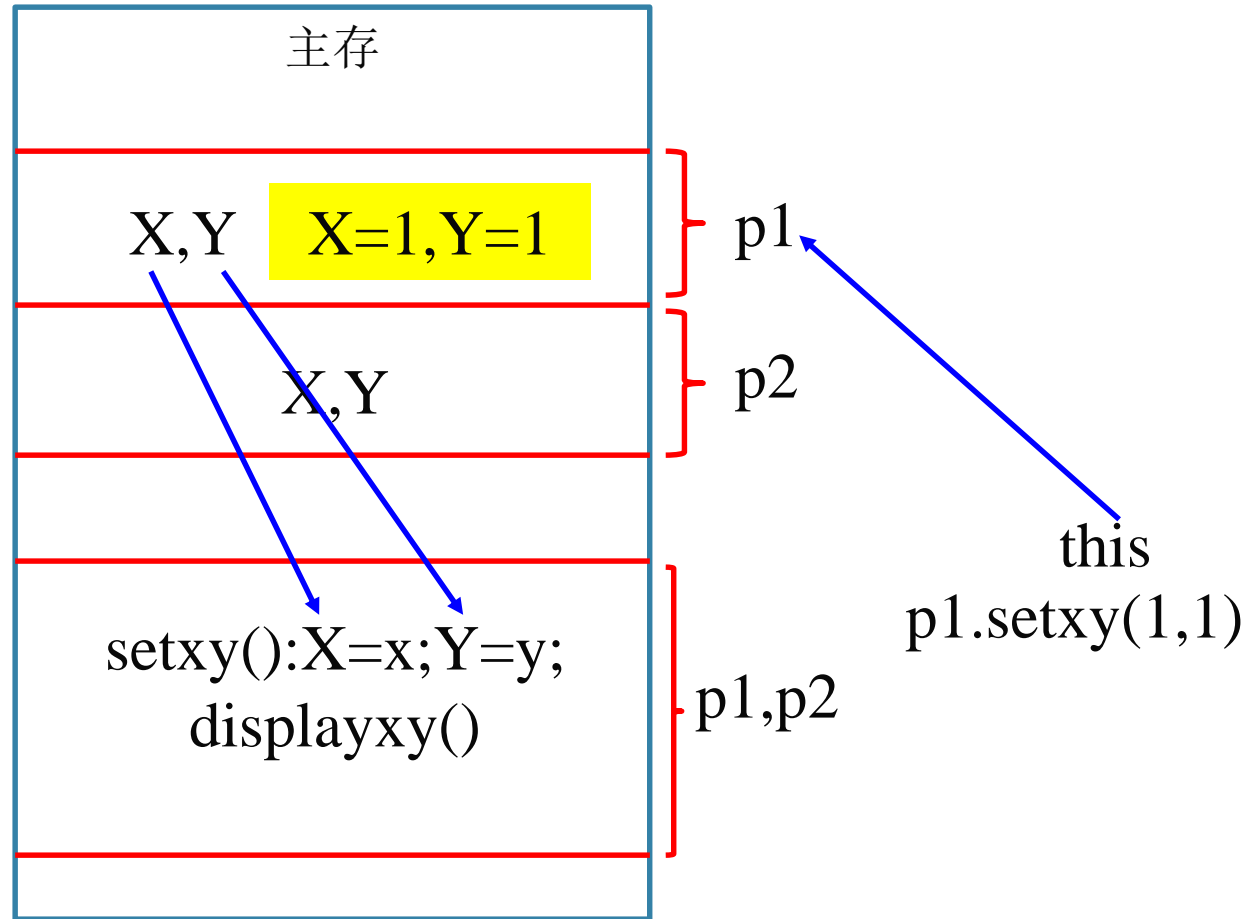
1.2.4 this 指针

- ❖ 在类的成员函数的形参表中都隐含一个指针变量**this**
- ❖ **this**为指向当前类的指针类型
- ❖ 当调用类的成员函数时，**this**指针变量被自动初始化为发出函数调用的对象的地址。

```
class Point
{
    public:
        void setxy(int a,int b)
        { X=a; y=b; }
    private:
        int X,Y;           //坐标
};
.....
Point p1;
p1.setxy(1,1);
```

```
struct Point
{
    int X,Y;               //坐标
};
void setxy(struct Point*this,int a, int b)
{
    this->X=a; this->Y=b;
}
...
struct Point p1;
setxy(&p1,1,1);
```

1.2.4 this 指针



1.2.4 this 指针

```
class Student
{
private:
    int number;
    string name;
    char sex;
    int age;
public:
    void set(int a,string b,char c,int d)
    {
        number = a;
        name = b;
        sex = c;
        age = d;
    }
    void show();
};
```

```
class Student
{
private:
    int number;
    string name;
    char sex;
    int age;
public:
    void set(int a,string b,char c,int d)
    {
        this->number = a;
        this->name = b;
        this->sex = c;
        this->age = d;
    }
    void show();
};
```

1.2.5 对象的作用域与生存期

```
#include "Student.h"
Student s1;
static Student s2;
void show()
{
    Student s3;
    static Student s4;
    s1.set(101, "Chen", 'M', 21);
    s2.set(102, "Wang", 'F', 20);
    s3.set(103, "Liu", 'M', 18);
    s4.set(104, "Sun", 'F', 22);
    s1.show();
    s2.show();
    s3.show();
    s4.show();
}
```

局部对象（不包括局部静态对象）的作用域是定义它的函数体，其生存期是从函数调用开始到函数调用结束。

构造局部对象的次序（即分配存储单元的次序）按照它们在函数体中声明的次序进行。

局部对象所占的存储空间被分配在程序的动态区（栈）中。

静态对象的作用域是定义它的函数体或程序文件；其生存期是整个程序的运行时间。

构造静态对象的次序按它们在程序中出现的先后次序，且在整个程序运行开始时（即在主函数运行前）只构造一次。

静态对象所占的存储空间被分配在程序的静态区（全局区）中。

全局对象的作用域是整个程序，其生存期是整个程序的运行时间。

它也是在程序运行时（即在主函数运行前）只构造一次。

全局对象所占的存储空间被分配在程序的静态区（全局区）中。

1.3 构造函数与析构函数

- ❖ 初始化工作：声明对象时，需要构造对象并给它的数据成员赋初值。
- ❖ 清理工作：对象使用结束时，需要进行一些清理工作，如释放所分配内存等。
- ❖ 对象的初始化和清理工作分别由两个特殊的成员函数来完成，即**构造函数**和**析构函数**。

1.3 构造函数与析构函数

- ❖ 1.3.1 构造函数
- ❖ 1.3.2 默认构造函数
- ❖ 1.3.3 构造函数的重载
- ❖ 1.3.4 具有默认参数值的构造函数（有缺省参数的构造函数）
- ❖ 1.3.5 初始化列表
- ❖ 1.3.6 拷贝构造函数（复制构造函数）
- ❖ 1.3.7 析构函数
- ❖ 1.3.8 浅拷贝与深拷贝

1.3.1 构造函数

❖ 构造函数（**Constructor**）是与类名同名的特殊的成员函数；负责对象的创建和初始化工作，其格式如下：

■ 类内：

类名（形参说明）

{ 函数体 }

■ 类外：

类名:: 类名（形参说明）

{ 函数体 }

❖ 【例1-2】修改【例1-1】中的**Student**类，为其定义构造函数，完成对象的初始化工作。

```

#include <iostream>
#include <string>
using namespace std;
class Student
{
private:
    int number;
    string name;
    char sex;
    int age;

public:
    Student(int a, string b, char c, int d);
    {
        void show();
        number = a;
        name = b;
        sex = c;
        age = d;
    }
    Student::Student(int a, string b, char c, int d)
    {
        number = a;
        name = b;
        sex = c;
        age = d;
    }
};

```

```

void Student::show()
{
    cout<<number<<"\t"<<name<<"\t"<<sex<<"\t"
<<age<<endl;
}

int main()
{
    Student stu(101,"zhao", 'F', 18);
    stu.show();
    return 0;
}

```

```

101      zhao      F      18
请按任意键继续. . .

```


1.3.1 构造函数

❖ 构造函数的特点:

- 构造函数是`public`属性，因为它是在创建对象的时候被自动调用（在类的外部）。
- 函数名与类名相同
- 无返回值，不能指定任何类型（包括`void`）
- 可以重载
- 不能被显式调用

```
Point p;  
p.Point();
```



1.3.2 默认构造函数

- ❖ 每个类必须有一个构造函数，没有构造函数就不能创建对象。
- ❖ 在类中，若没有显式定义构造函数，则C++编译器在编译时会提供一个默认构造函数。格式：

类名 ()

{ }

默认构造函数就是函数体为空的无参构造函数

1.3.2 默认构造函数

- ❖ 由于默认构造函数的函数体为空，所以如果创局部对象，则该对象的数据成员的初始值是不确定的。
- ❖ 只要类中定义了一个构造函数，则C++编译系统就不再提供默认构造函数。
即：如果已经为类定义了一个带参数的构造函数，但还想要无参构造函数，则必须自己定义。
- ❖ 【默认构造函数示例】

1.3.3 构造函数的重载

- ❖ 如果一个类中定义了两个以上的同名成员函数，但参数的个数和类型有所不同，称为类的成员函数的重载。
- ❖ 以下情况不是成员函数的重载：
 - 一个类的成员函数和另一个类的成员函数同名
 - 一个类的成员函数与一个类外的函数同名因为类名也是成员名的一部分。

```
class Student
{
...
public:
    void set(int a,string b,char c,int d);
    void set(int a, char c);
};
```

```
class Student
{
...
public:
    void set(int t d);
};
void set(int a)
class Teacher
{
...
public:
    void set(int )
};
```



❖ 在构造对象时，系统根据实参的类型及个数匹配调用合适的构造函数。

❖ 【例1-3】修改【例1-2】中的**Student**类，为其定义两个构造函数，完成对象的初始化工作。

```
❖ #include <iostream>
❖ #include <string>
❖ using namespace std;
❖ class Student
❖ {
❖ private:
❖     int number;
❖     string name;
❖     char sex;
❖     int age;
❖ public:
❖     Student();
❖     Student(int a, string b, char c, int d);
❖     void show();
❖ };
❖ Student::Student()
❖ {
❖     number = 0;
❖     name = "No name";
❖     sex = '!';
❖     age = 0;
❖ }
❖ }
```

构造函数的重载

```
❖ Student::Student(int a, string b, char c, int d)
❖ {
❖     number = a;
❖     name = b;
❖     sex = c;
❖     age = d;
❖ }
❖ void Student::show()
❖ {
❖     cout<<number<<"\t"<<name<<"\t"<<sex<<"\t"
❖     <<age<<endl;
❖ }
❖ int main()
❖ {
❖     Student stu1;
❖     Student stu2(101,"Zhao", 'F', 18);
❖     stu1.show();
❖     stu2.show();
❖     return 0;
❖ }
```

```
0      No name !      0
101    Zhao    F      18
请按任意键继续. . .
```

1.3.4 具有默认参数值的构造函数

❖ 当构造函数的形参具有默认值时，称为具有默认参数值的构造函数（即具有缺省参数的构造函数）。

❖ 注意：

- （1）只能出现在类定义的接口部分，而不能出现在类定义的实现部分。
- （2）所有具有默认值的参数必须处在参数表的最右边。
- （3）在使用具有缺省参数的构造函数时，要防止二义性。

【例1-4】修改【例1-2】中的Student类，定义有默认参数的构造函数，完成对象的初始化。

```

❖ #include <iostream>
❖ #include <string>
❖ using namespace std;
❖ class Student
❖ {
❖ private:
❖     int number;
❖     string name;
❖     char sex;
❖     int age;
❖ public: Student(); //不能有该函数，出现二义性
❖     Student(int a=0, string b="No name", char c='!', int d=0);
❖     void show();
❖ };
❖ Student::Student(int a, string b, char c, int d)
❖ {
❖     number = a;
❖     name = b;
❖     sex = c;
❖     age = d;
❖ }

```

有默认值

默认值从右向左依次

不能有默认值

❖ **void Student::show()**

❖ {

❖ **cout<<number<<"\t"<<name<<"\t"<<sex<<"\t"<<age<<endl;**

❖ }

❖ **int main()**

❖ {

❖ **Student stu1;**

❖ **Student stu2(101,"zhao", 'F', 18);**

❖ **stu1.show();**

❖ **stu2.show();**

❖ **return 0;**

❖ }

```
0      No name !      0
101    Zhao    F      18
请按任意键继续. . .
```

❖ #include <iostream> //缺省构造函数二义性示例

❖ #include <string>

❖ using namespace std;

❖ class Student

❖ {

❖ private:

❖ int number;

❖ string name;

❖ char sex;

❖ int age;

❖ public:

❖ Student();

❖ Student(int a=0, string b="No name", char c='!', int d=0);

❖ void show();

❖ };

❖ Student::Student()

❖ {

❖ number = 0;

❖ name = "No name";

❖ sex = '!';

❖ age = 0;

❖ }

❖ **Student::Student(int a, string b, char c, int d)**

❖ {

❖ **number = a;**

❖ **name = b;**

❖ **sex = c;**

❖ **age = d;**

❖ }

❖ **void Student::show()**

❖ {

❖ **cout<<number<<'\t'<<name<<'\t'<<sex<<'\t'<<age<<endl;**

❖ }

❖ **int main()**

❖ {

❖ **Student stu1;**

❖ **Student stu2(101,"zhao", 'F', 18);**

❖ **stu1.show();**

❖ **stu2.show();**

❖ **return 0;**

❖ }

调用哪个构造函数?

Student() ?

Student(int a=0, string b="No name", char c='!', int d=0)?

1.3.5 初始化列表

❖ 构造函数中对数据成员进行初始化有两种方式：

- 在函数体中用赋值语句
- 在参数表后用初始化列表

```
Student::Student(int a, string b, char c, int d): number(a), name(b), sex(c), age(d)
{
}
```

❖ 特点：

- 初始化列表与形参列表间用 “: ” 分割，各个初始化成员间用 “, ” 分割
- 效率高
- 是初始化对象某些特殊数据成员的唯一方法，例如对象成员

1.3.6 拷贝构造函数（复制构造函数）

❖ 拷贝构造函数是一种构造函数，声明如下：

类名 (const 类名 &参数名) ;

❖ 说明：

(1) 其名称与类名相同，且它只有一个参数。该参数是对该类对象的引用。

(2) 其功能是用一个对象去构造另外一个对象。

(3) 为防止所引用的对象被修改，通常把引用参数声明为const参数

❖ 【例1-5】 Person类是一个人员信息类。用普通构造函数生成对象per1，用拷贝构造函数生成对象per2。

- ❖ #include <iostream>
- ❖ #include <string>
- ❖ using namespace std;
- ❖ class Person
- ❖ {
- ❖ private:
- ❖ int num;
- ❖ string name;
- ❖ public:
- ❖ Person(int n, string str)
- ❖ {
- ❖ num = n;
- ❖ name = str;
- ❖ }
- ❖ **Person(const Person &x)**
- ❖ {
- ❖ num = x.num;
- ❖ name = x.name;
- ❖ }
- ❖ void show();
- ❖ };

```

void Person::show()
{
    cout<<"num="<<num<<endl;
    cout<<"name="<<name<<endl;
}
int main()
{
    Person per1(1001, "Tom");
    per1.show();
    Person per2(per1);
    per2.show();
    return 0;
}

```

```

num=1001
name=Tom
num=1001
name=Tom
请按任意键继续. . .

```

1.3.6 拷贝构造函数（复制构造函数）

❖ 拷贝构造函数具有一般构造函数的特性，其特点如下：

- ◆ 名字与类名相同，不能指定返回类型
- ◆ 只有一个形参，该参数是该类的对象的引用
- ◆ 不能被显式调用。

拷贝构造函数的三种调用

❖ (1) 由一个对象初始化同类的另一个对象

- `Person per1(1001,"Sheldon");`
- `Person per2(per1);`

创建对象per2时，系统自动调用拷贝构造函数，利用per1进行初始化

❖ (2) 对象作为函数实参传递给函数形参时

- `void Fun(Person p) {p.show();}`
- `Person per1(1001,"Sheldon");`
- `Fun(per1);`

调用Fun函数进行参数传递时，调用拷贝构造函数初始化形参对象p。函数调用结束，释放形参对象p

❖ (3) 函数返回值为对象时

- `Person Fun() {`
- `Person p(1001,"Sheldon");`
- `return p;`
- `}`
- `Person per1;`
- `per1=Fun();`

执行该语句时，系统调用拷贝构造函数用对象p去创建并初始化一个无名对象，函数返回时将无名对象的值赋给per1，然后无名对象释放

默认拷贝构造函数

- ❖ 如果类中没有定义拷贝构造函数，则系统自动生成一个**默认拷贝构造函数**。
- ❖ 该**函数的功能**是将已知对象的所有数据成员的**值**拷贝给对应的对象的所有数据成员（即**浅拷贝**）。

1.3.7 析构函数

- ❖ 析构函数也是一种特殊的成员函数，作用是在对象消失时，执行清理任务。
- ❖ 其函数名称是在类名的前面加上“~”，声明如下：

~类名 () ;

- ❖ 说明：

(1) 若在类中未定义析构函数，则系统会自动创建一个默认的析构函数，形式为：

~类名 ()
{ }

(2) 析构函数是**public**属性，在对象生命期结束时，系统自动调用。若在类的对象中**分配有动态内存**，就必须为该提供析构函数，完成内存回收。

(3) 析构函数**没有返回值和参数**

(4) 一个类**只能有一个析构函数，不能重载**。

(5) 对象被析构的顺序与其建立时的顺序正好相反。

❖ 【例1-6】分析程序的执行结果。

```
❖ #include <iostream>
❖ #include <cstring>
❖ using namespace std;
❖ class StringDef
❖ {
❖ private:
❖     char *text;
❖ public:
❖     StringDef(char * ch)
❖     {
❖         text = new char[strlen(ch)+1];
❖         strcpy_s(text, strlen(ch)+1, ch);
❖         cout<<"调用构造函数！"<<endl;}
❖     ~StringDef ()
❖     {
❖         delete[] text;
❖         cout<<"调用析构函数！"<<endl;
❖     }
❖     void show()
❖     {
❖         cout<<"text="<<text<<endl;    }
❖     };
```

```
int main()
{
    StringDef str ("Hello world!");
    str.show();
    getchar();
    return 0;
}
```

构造对象str时，系统自动调用构造函数，为text申请内存空间，并为text赋值为“Hello world! ”，输出一行：“调用构造函数！”

对象str生命期结束，系统自动调用析构函数，释放内存空间，输出一行：“调用析构函数！”

```
调用构造函数！
text=Hello world!
调用析构函数！
请按任意键继续. . .
```

1.3.8 浅拷贝与深拷贝

- ❖ **浅拷贝**：由默认的拷贝构造函数将已有对象的数据成员一一赋值给同类的新对象的各数据成员来构造新对象的方法就是浅拷贝，即**成员级拷贝**。
- ❖ **缺点**：
 - 若有选择化地进行复制，这时默认的拷贝构造函数就不能胜任。
 - 浅拷贝会带来数据安全方面的隐患（数据成员有指针类型时）。
- ❖ **深拷贝**对于指针型数据成员，拷贝的是其所指向的区域内容而不是指针值。
- ❖ 要完成深拷贝，用户必须自定义拷贝构造函数，并在其中先要申请存储空间，然后再将相应的内容拷贝过来。
- ❖ **【例1-7】** **Person**是一个人员信息类。用普通构造函数生成**per1**，用默认拷贝构造函数生成**per2**。（即浅拷贝）

```
❖ #include <iostream>
❖ #include <string>
❖ using namespace std;
❖ class Person
❖ {
❖ private:
❖     int num;
❖     char *name;
❖ public:
❖     Person(int n, char *str)
❖     {
❖         num = n;
❖         name = new char[strlen(str)+1];
❖         strcpy_s(name, strlen(str)+1, str);
❖     }
❖     ~Person()
❖     {
❖         cout<<"析构对象"<<name<<endl;
❖         delete[] name;
❖     }
❖     void show()
❖     {
❖         cout<<"num="<<num<<"\nname="<<name<<endl;
❖     }
❖ };
```

```
❖ int main()
❖ {
❖     Person per1(1001,"Tom");
❖     per1.show();
❖     Person per2(per1);
❖     per2.show();
❖     return 0;
❖ }
```

```
num=1001
name=Tom
num=1001
name=Tom
析构对象Tom
析构对象葺葺葺葺 1柳
```

Microsoft Visual C++ Runtime Library



Debug Assertion Failed!

Program: ...的教学\C++程序设计
 \ConsoleApplication3\Debug\ConsoleApplication3.exe
 File: minkernel\crts\ucrt\src\appcrt\heap\debug_heap.cpp
 Line: 904

Expression: _CrtIsValidHeapPointer(block)

For information on how your program can cause an assertion failure, see the Visual C++ documentation on asserts.

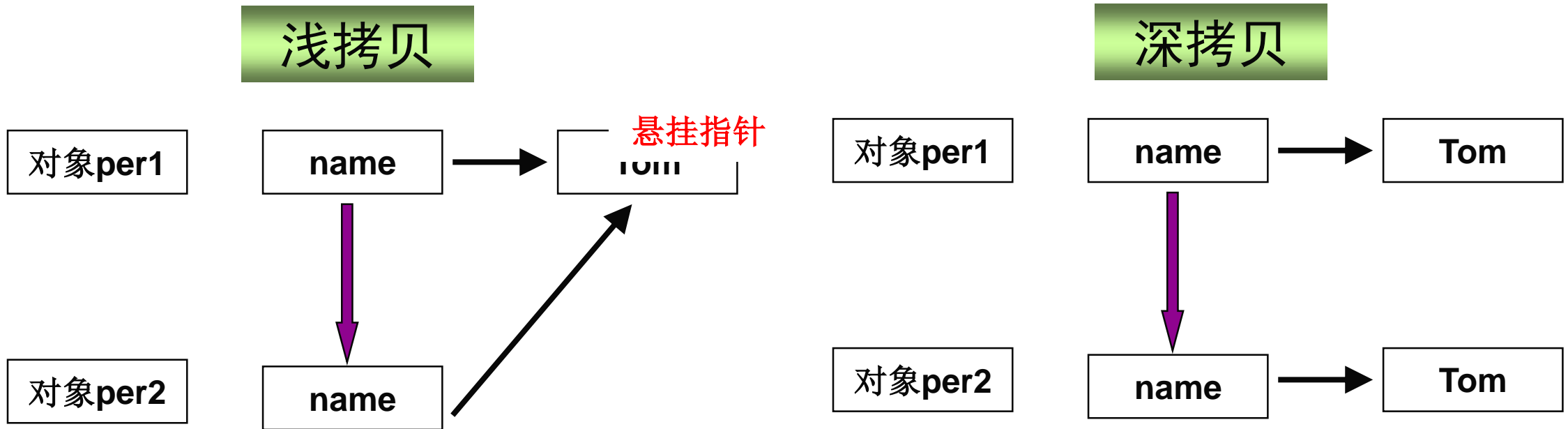
(Press Retry to debug the application)

中止(A)

重试(R)

忽略(I)

1.3.8 浅拷贝与深拷贝



❖ 练习：修改【例1-7】，添加拷贝构造函数，改为深拷贝。

```
❖ #include <iostream>
❖ #include <string>
❖ using namespace std;
❖ class Person
❖ {
❖ private:
❖     int num;
❖     char *name;
❖ public:
❖     Person(int n, char *str)
❖     {
❖         num = n;
❖         name = new char[strlen(str)+1];
❖         strcpy_s(name, strlen(str)+1, str);
❖     }
❖     Person(const Person &x)
❖     {
❖         num = x.num;
❖         name = new char[strlen(x.name)+1];
❖         strcpy_s(name, strlen(x.name)+1, x.name);
❖     }
```

```

❖ ~Person()
❖ {
❖     cout<<"析构对象"<<name<<endl;
❖     delete[] name;
❖ }
❖ void show()
❖ {
❖     cout<<"num="<<num<<"\nname="<<name<<endl;
❖ }
❖ };

```

创建对象per1时，系统自动调用普通构造函数

```

❖ int main()
❖ {
❖     Person per1(1001,"Sheldon");
❖     per1.show();
❖     Person per2(per1);
❖     per2.show();
❖     return 0;
❖ }

```

创建对象per2时，系统自动调用拷贝构造函数

per1和per2生存期结束，系统自动调用析构函数，先析构per2再析构per1

```

num=1001
name=Sheldon
num=1001
name=Sheldon
析构对象Sheldon
析构对象Sheldon
请按任意键继续. . .

```


1.4 对象成员、对象数组与堆对象

❖ 1.4.1 对象成员（也叫子对象）

❖ 1.4.2 对象数组

❖ 1.4.3 堆对象

1.4.1 对象成员

❖ 某个类的对象是另外一个类X的数据成员，称为X的对象成员，又称X的子对象。

```
class X
```

```
{
```

```
    类名1 成员1;
```

```
    类名2 成员2;
```

```
    .....
```

```
    类名n 成员n;
```

```
    .....
```

```
};
```

```
class CPU
```

```
{
```

```
    .....
```

```
};
```

```
class Memory
```

```
{
```

```
    .....
```

```
};
```

```
class Computer
```

```
{
```

```
    .....
```

```
    private:
```

```
        CPU cpu;
```

```
        Memroy mem;
```

```
};
```

对象做为类的数据成员，称之为对象成员（也叫子对象）

对象成员的初始化

- ❖ X的构造函数要负责初始化自己的普通成员，又要负责调用对象成员所属类的构造函数对对象成员初始化。
- ❖ 调用对象成员所属类的构造函数的方法是：利用**X类的构造函数初始化列表**，格式：
X::X（参数表0）：对象成员1（参数表1），对象成员2（参数表2），...，对象成员n（参数表n）
{ }
- ❖ 说明：
 - 同一个类中的不同对象成员，系统**按照它们在类中的定义顺序调用相应的构造函数**，而不是按照初始化表中的顺序。
 - 当建立X类的对象时，**先调用对象成员的构造函数，初始化对象成员，然后才执行X类的构造函数的函数体**，初始化X类中的其他成员。
- ❖ 【例1-8】Time Date Schedule。

```
❖ #include <iostream>
❖ using namespace std;
❖ class Date
❖ {
❖ private:
❖     int year;
❖     int month;
❖     int day;
❖ public:
❖     Date(int y, int m, int d)
❖     {
❖         cout<<"调用Date的构造函数"<<endl;
❖         year = y;
❖         month = m;
❖         day = d;
❖     }
❖     void show()
❖     {
❖         cout<<year<<"年"<<month<<"月"<<day<<"日";
❖     }
❖ };
```

```
❖ class Time
❖ {
❖ private:
❖     int hour;
❖     int minute;
❖     int second;
❖ public:
❖     Time(int h, int m, int s)
❖     {
❖         cout<<"调用Time的构造函数"<<endl;
❖         hour = h;
❖         minute = m;
❖         second = s;
❖     }
❖     void show()
❖     {
❖         cout<<hour<<"点"<<minute<<"分"<<second<<"秒"<<endl;
❖     }
❖ };
```

```
class Schedule
```

```
{
```

```
private:
```

```
int number;
```

```
Time time;
```

```
Date date;
```

```
//Time类对象
```

```
//Date类对象
```

```
Time(int h, int m, int s);
```

```
Date(int y, int m, int d);
```

```
public:
```

```
Schedule(int num, int y, int mo, int d, int h, int mi, int s):date(y, mo, d), time(h, mi, s)
```

```
{
```

```
    cout<<"调用Schedule的构造函数"<<endl;
```

```
    number = num;
```

```
}
```

```
void show()
```

```
{
```

```
    cout<<"序号"<<number<<":";
```

```
    date.show();
```

```
    time.show();
```

```
}
```

```
};
```

❖ **int main()**

❖ **{**

❖ **Schedule s1(1, 2010, 10, 1, 22, 22, 22**

❖ **s1.show();**

❖ **Schedule s2(2, 2010, 11, 1, 20, 20, 20);**

❖ **s2.show();**

❖ **return 0;**

❖ **}**

s1的number=1, time(22,22,22),
date(2010, 10, 1)

s1的number=2, time(20,20,20),
date(2010, 11, 1)

调用Time的构造函数
调用Date的构造函数
调用Schedule的构造函数
序号1:2010年10月1日22点22分22秒
调用Time的构造函数
调用Date的构造函数
调用Schedule的构造函数
序号2:2010年11月1日20点20分20秒
请按任意键继续. . .

1.4.2 对象数组

❖ 数组中的每个元素都是类的一个对象。

类名 数组名[数组大小];

❖ 例如: **Student s[3];**

❖ 说明: 数组中每个元素都是对象, 都要调用默认构造函数进行构造。

❖ 【例1-9】定义学生类**Student**, 完成若干个学生信息的输入和显示。


```
❖ #include <iostream>
❖ #include <string>
❖ using namespace std;
❖ class Student
❖ {
❖ private:
❖     string name;
❖     int age;
❖ public:
❖     void assignment(string n, int a);
❖     void show();
❖ };
❖ void Student::assignment(string n, int a)
❖ {
❖     name = n;
❖     age = a;
❖ }
❖ void Student::show()
❖ {
❖     cout<<"\t"<<name<<"\t"<<age<<endl;
❖ }
```

❖ `int main()`

❖ `{ const int N=100;`

对象数组，数组中的每个元素都是一个对象

❖ `int n, i;`

❖ `Student stu[N];`

❖ `cout<<"输入学生的人数（不大于100）:";`

❖ `cin>>n;`

❖ `string name;`

❖ `int age;`

❖ `for(i=0;i<n;i++)`

❖ `{ cout<<"输入第"<<i+1<<"个学生姓名和年龄: ";`

❖ `cin>>name>>age;`

❖ `stu[i].assignment(name, age);`

❖ `}`

❖ `cout<<"\n"; cout<<"编号\t姓名\t年龄"<<endl;`

❖ `for(i=0;i<n;i++)`

❖ `{ cout<<i+1;`

❖ `stu[i].show(); }`

❖ `return 0;`

❖ `}`

1.4.3 堆对象

- ❖ 为对象分配存储空间主要有静态分配和动态分配两种方式。
 - **静态分配方式**：在声明对象时分配存储空间，对象生命期结束时收回所存储空间。对象的创建和销毁是由程序本身决定的。
 - 例如声明：Point p1,p2; 时，即创建对象p1, p2。
 - **动态分配方式**：使用运算符new调用构造函数创建对象，此时在堆中为其分配内存空间；对象使用完毕时，要使用运算符delete来释放它所占用的内存空间，在这种分配方式下，对象的创建和销毁是由程序员决定的。
- ❖ 堆对象是在程序运行时根据需要随时可以被创建或删除的对象，只有堆对象采用动态分配方式。

1.4.3 堆对象

❖ 使用运算符**new**调用构造函数可以动态地创建对象，即堆对象。

❖ 语法为：

❖ **new** 类名(初始值列表)

❖ 例如：

- **Person *sa;**
- **sa=new Person(1001,"Tom");**

1.4.3 堆对象

❖ 使用运算符`delete`删除堆对象，专门用来释放由运算符`new`所创建的对象。

❖ 其使用语法为：

❖ `delete` 指针名

❖ 例如：

❖ `delete sa;`

1.4.3 堆对象

❖ 使用运算符 `new` [] 创建对象数组

❖ 其使用语法为：

`new 类名 [n]`

❖ 其中，n给出数组的大小，后面不能再跟构造函数参数，所以，从堆上分配对象数组，只能调用默认的构造函数，不能调用其他任何构造函数。例如：

❖ `Student *ptr;`

❖ `ptr=new Student[15];`



❖ 使用运算符**delete []**删除对象数组

❖ 其格式如下：

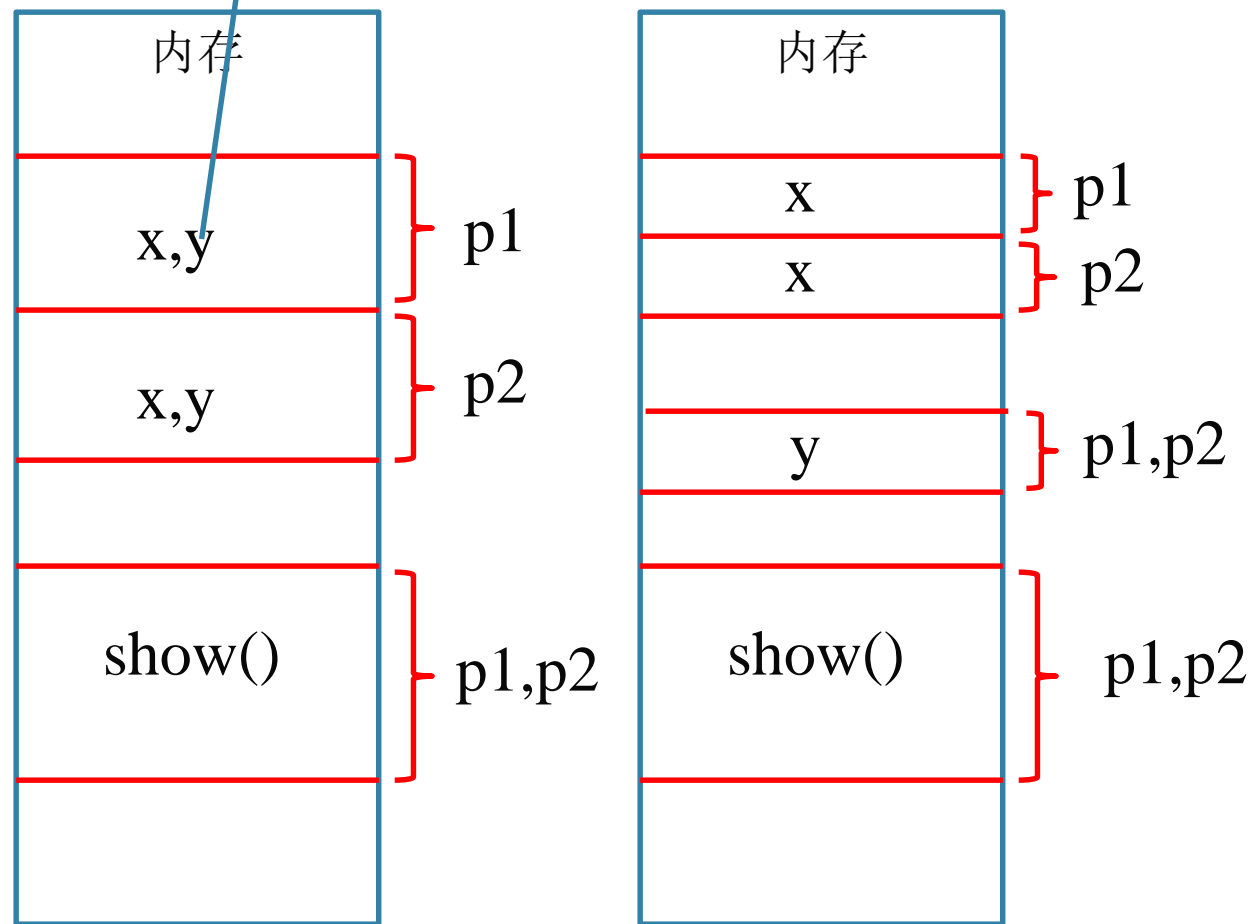
❖ **delete[]** 指针名

❖ 其中，指针名必须是指向**new[]**所创建的对象数组，且必须是**new[]**所返回的值。

❖ **【例1-10】 new和delete， new[]和delete[]的使用（VS中运行）**

y怎么被p1、p2共享?

1.5 静态成员



- ❖ 同一类的各个对象之间如何共享数据?
- ❖ 在类中设置一种“特殊”的成员——静态成员
- ❖ 静态成员是指声明为**static**的类成员，静态成员只有一个，为所有对象共享。
- ❖ 静态成员的分类
 - 静态数据成员
 - 静态成员函数

1.5.1 静态数据成员

class 类名

{

static 类型说明符 成员名;

.....

};

❖ 【例1-11】分析程序的执行结果。

```
❖ #include <iostream>
❖ using namespace std;
❖ class Sample
❖ {
❖ private:
❖     int x;
❖ public:
❖     static int y;
❖     Sample(int a)
❖     {
❖         → x=a;
❖         → x++;
❖         → y++;
❖     }
❖     void show()
❖     {
❖         cout<<"x="<<x<<"",y="<<y<<endl;
❖     }
❖ };
❖ int Sample::y=0;
```

所有对象共享一个y

int main()

{

Sample s1(5);

s1.show();

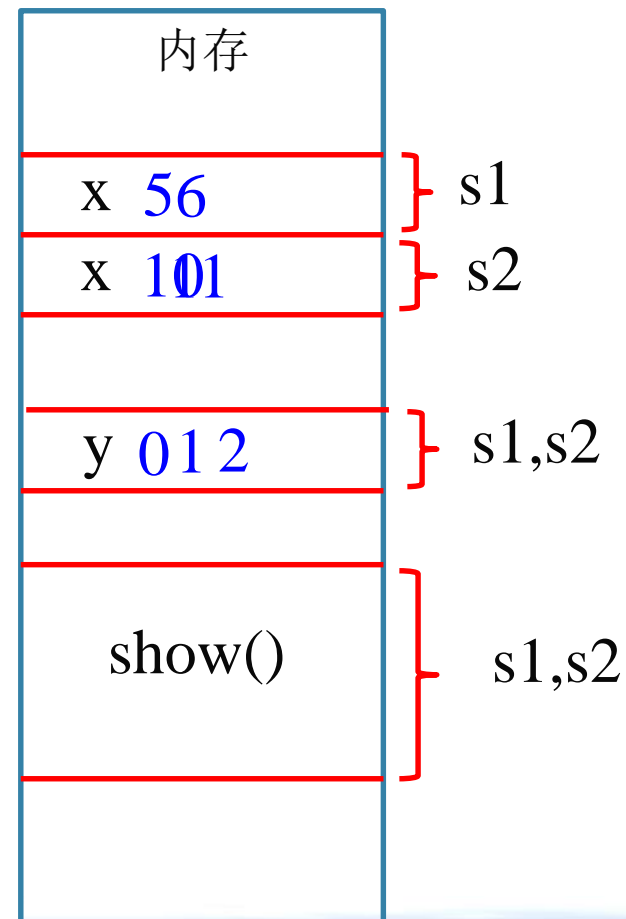
Sample s2(10);

s2.show();

return 0;

}

静态成员初始化



- (1) 静态数据成员不属于任何一个对象，而是属于类，被所有对象共享。
- (2) 一个对象的存储空间不包含静态数据成员。
- (3) 静态数据成员的分配是在程序一开始运行时（主函数运行前）。
- (4) 类的静态数据成员必须进行初始化，且初始化语句应当写在程序的全局区域中，并且必须指明其数据类型与所属类的类名，其格式如下：

数据类型标识符 类名::变量名=值;

- (5) 对于在类的**public**部分说明的静态数据成员，在类外，可以不使用成员函数而直接访问，格式是：

类名::静态数据成员名

1.5.2 静态成员函数

class 类名

{


static 类型 函数名 (形参)

{ 函数体}

.....

};

类的静态成员函数不与任何对象联系，它可以直接访问类的静态数据成员，但不能直接访问类的普通数据成员（没有this指针），因为普通数据成员只有对象存在时才有意义。



❖ **【例1-12】** 阅读程序，理解在类的内部和外部访问类的普通数据成员和静态数据成员的方法以及区别，理解调用类的普通成员函数和类的静态成员函数的方法以及区别。

❖ #include <iostream>	❖ void Test::show1()
❖ using namespace std;	❖ {
❖ class Test	❖ cout<<"m="<<m<<"\tq="<<q<<endl;
❖ {	❖ }
❖ private:	❖ void Test::show2()
❖ int n;	❖ {
❖ static int m;	❖ cout<<"m="<<m<<"\tq="<<q<<endl;
❖ public:	❖ cout<<"n="<<n<<"\tp="<<p<<endl;
❖ int p;	❖ }
❖ static int q;	❖ int main()
❖ Test(int a);	❖ {
❖ static void show1();	❖ Test t(3);
❖ void show2();	❖ cout<<"p="<<t.p<<"\tq="<< t.q <<"\tq="<< Test::q <<endl;
❖ };	❖ Test::show1();
❖ int Test::m=20;	❖ t.show1();
❖ int Test::q=30;	❖ t.show2();
❖ Test::Test(int a)	❖ return 0;
❖ {	❖ }
❖ n=a;	
❖ p=a;	
❖ }	

总结

❖ 在类的外部使用**public**静态数据成员或成员函数时，采用如下格式：

“**类名::静态成员名**” 或 “**对象名.静态成员名**”

❖ 在类的外部调用**public**普通数据成员和成员函数时，只能使用 “**对象名.普通成员名**”

❖ 类的静态成员函数不与任何对象联系，它可以直接访问类的静态数据成员，但不能直接访问类的普通数据成员，因为普通数据成员只有对象存在时才有意义（没有this指针）。

1.6 友元函数和友元类

- ❖ C++提供了一种允许外部类和外部函数访问类的私有成员和保护成员的辅助方法，即将它们声明为一个给定类的友元，使其具有类成员函数的访问权限。
- ❖ 在类中，可以利用关键字friend将外部类或外部函数声明为它的友元。
- ❖ 如果友元是一般函数或类的成员函数，称为友元函数；
- ❖ 如果友元是一个类，则称为友元类。友元类的所有成员函数都是友元函数。
- ❖ 但友元函数本身不是这个类的成员。

1.6.1 友元函数

- ❖ 友元函数：在类定义中用**friend**修饰声明的**非类成员函数**。
- ❖ 友元函数可以是一个普通的函数，也可以是其他类的成员函数；
- ❖ 不是本类的成员函数，但是在它的函数体中可以**通过对象名访问类的私有和保护成员**。
- ❖ 友元函数的声明：

```
class 类名
{
    .....//类中的其他成员
    friend 类型 函数名(形参表);
    .....//类中的其他成员
};
```

❖ 说明:

- (1) 友元的声明语句可以放在类的私有部分也可以放在类的公有部分，没有区别。
- (2) 友元函数能访问类的所有成员，一个函数可以是多个类的友元函数，在各个类中分别声明即可。
- (3) 友元函数的定义，既可在类的内部也可在类的外部；友元函数若要对类的成员进行访问，则必须在参数表中显式地指明要访问的对象。
- (4) 一个类的友元函数与该类的成员函数一样享有对该类一切成员的访问权。
- (5) 友元函数的调用方式、原理与普通函数相同。
- (6) **C++**不允许将构造函数、析构函数和虚函数声明为友元函数。



❖ 【例1-13】 设计一个点类**Point**，用成员函数、友元函数、普通函数分别求两点之间的距离。

```

❖ #include <iostream>
❖ #include <cmath>
❖ using namespace std;
❖ class Point
❖ {
❖ private:
❖     double x;
❖     double y;
❖ public:
❖     Point(int a, int b)
❖     {
❖         x=a;          y=b;    }
❖     static double distance1(Point &a, Point &b);
❖     friend double distance2(Point &a, Point &b);
❖     double getx()    //提取x的值
❖     {
❖         return x;      }
❖     double gety()    //提取y的值
❖     {
❖         return y;      }
❖ };

```

成员函数

友元函数

```

❖ double Point::distance1(Point &a, Point &b)
❖ {    return sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y));    }
❖ double distance2(Point &a, Point &b)
❖ {    return sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y));    }
❖ double distance3(Point &a, Point &b)
❖ {
❖     return sqrt((a.getx()-b.getx())*(a.getx()-b.getx())+(a.gety()-
❖     b.gety())*(a.gety()-b.gety()));
❖ }
❖ int main()
❖ {
❖     Point p1(3,5);
❖     Point p2(5,3);
❖     cout<<"用成员函数求两点之间的距离: "<<Point::distance1(p1,p2)<<endl;
❖     cout<<"用友元函数求两点之间的距离: "<<distance2(p1,p2)<<endl;
❖     cout<<"用普通函数求两点之间的距离: "<<distance3(p1,p2)<<endl;
❖     return 0;
❖ }

```

普通函数

1.6.2 友元类

❖ 类A中的某些成员函数要访问类B中的数据成员时：

```
class 类名B
{
    ..... // 类中的其他成员
    friend class 类名A;
    ..... // 类中的其他成员
};
```

❖ 一个类的友元类的所有成员函数都可视为该类的友元函数，都能访问类的私有和保护成员。

❖ 【例1-14】 分析程序的执行结果。

```
#include <iostream>
#include <cmath>
using namespace std;
class student
{
public:
    friend class teacher;//teacher是student的友元类
    student() {};
```

private:

```
    int number, score;//学号,成绩
};

class teacher
{
public:
    teacher(int i, int j);
    void display();

private:
    student a;
};
```

```
teacher::teacher(int i, int j)
{
    a.number = i;    a.score = j;
}

void teacher::display()
{
    cout << "No=" << a.number << " ";
    cout << "score=" << a.score << endl;
}

int main()
{
    student s;
    teacher t1(1001, 89), t2(1002, 78);
    cout << "第一个学生的信息 ";
    t1.display();
    cout << "第二个学生的信息 ";
    t2.display();
    return 0;
}
```

1.7 常对象与常成员

共享

&&

防止改变

❖ 常对象和常成员需用关键字**const**

❖ 常成员：

- 常数据成员
- 常成员函数

1.7.1 常对象

❖ 常对象是指对象常量，声明如下：

`const` 类名 对象名；

或 类名 `const` 对象名；

❖ 常对象的特点：

- (1) 常对象声明时必须进行初始化，且其值以后不能被修改。
- (2) 常对象只能调用常成员函数，不能调用类中的其他普通成员函数。

```
#include <iostream>
using namespace std;
class Test
{
private:
    int a,b;
public:
    Test(int x,int y):a(x),b(y)
    {
    }
    void show()
    {
        cout<<"a="<<a<<" b="<<b<<endl;
    }
};
int main()
{
    const Test obj(1,2);
    obj.show(); //error, 常对象只能调用常成员函数
    return 0;
}
```

1.7.2 常数据成员

❖ 常数据成员的声明如下：

const 类型名 数据成员名；

❖ 常数据成员的初始化要通过**构造函数的初始化列表完成**

类名::类名(形参表):常数据成员名1(值1), 常数据成员名2(值2), ...

```
{ //构造函数的函数体  
}
```

❖ 初始化后，其值不可更改。

❖ 【例1-15】分析程序的执行结果。

```

❖ #include <iostream>
❖ using namespace std;
❖ class Test
❖ {
❖ private:
❖     int a;
❖     const int b;
❖ public:
❖     Test(int x, int y):b(y)
❖     {
❖         a = x;
❖     }
❖     void show()
❖     { cout<<"a="<<a<<" b="<<b<<endl; }
❖ };
❖ int main()
❖ {
❖     Test obj(1,2);
❖     obj.show( );
❖     return 0;
❖ }

```

1.7.3 常成员函数

❖ 常成员函数的声明如下：

类型 函数名(形参表) const;

❖ 常成员函数的特点：

- (1) 只读函数，不能修改数据成员的值，也不能调用该类中非常成员函数。
- (2) **const**关键字是函数类型的一部分，定义部分中也要带**const**关键字。
- (3) **const**关键字可以区分重载。
- (4) 能被常对象和一般对象调用。

❖ 【例1-16】常成员函数和常对象的使用。

```
#include <iostream>
using namespace std;
class TestConst
{
private:
    int a;

public:
    TestConst(int x)
    {
        a = x;
    }
    void set(int x);
    void show();
    void show() const;
};
void TestConst::set(int x)
{
    a = x;
}
```

```
void TestConst::show()
{
    cout<<"a="<<a<<endl;
}
void TestConst::show() const
{
    cout<<"a="<<a<<endl;
}
int main()
{
    TestConst t1(100);
    const TestConst t2(200);
    t1.show();
    t2.show();
    t1.set(300);
    // t2.set(400); //error 常对象只能调用常成员函数
    t1.show();
    t2.show();
    return 0;
}
```

总结

- ❖ 一般对象和常对象都能调用常成员函数，但常对象只能调用常成员函数
- ❖ 常成员函数可以访问普通数据成员，但是不能修改它们的值
- ❖ 常成员函数不能调用非常成员函数，但普通成员函数可以调用常成员函数
- ❖ 类中不改变数据成员值的函数可以设计为常成员函数