第3章 关联与连接

客观世界中不仅存在大量的事物，还存在事物之间的相互作用，事物及其相互作用才创造了丰富多彩的大千世界。

第2章主要从编程实现角度介绍了类的封装和对象的职责，现在又回到客观世界中，继续讨论客观事物之间的关系。


视频讲解

3.1 关联与连接的概念

关联和连接是面向对象思想中的两个基本概念。连接(Link)是对客观事物之间的关系的抽象，是客观事物之间的关系在计算机世界中的反映；关联(Association)是对连接的抽象，是一种关系在计算机世界中的反映。

语文中，使用陈述语句描述客观事物及其相互关系，陈述语句主要采用“主谓宾”格式，其中，主语和宾语使用名词，表示客观事物；谓语为动词，表示客观事物之间的关系。

例如，张三喜欢足球，李四喜欢篮球，王五喜欢篮球，陈述了一个人与一种体育运动之间的对应关系。人与体育运动之间的对应关系有很多种，这里只陈述了其中一种称为“喜欢”的对应关系，即人“喜欢”体育运动。

数学中，使用元素之间的对应关系来描述客观事物及其相互关系。例如，张三、李四、王五是集合“人”中的元素，足球、篮球是集合“体育运动”中的元素，“喜欢”是集合“人”和“体育运动”中元素之间的对应关系，这种对应关系如图3.1所示。

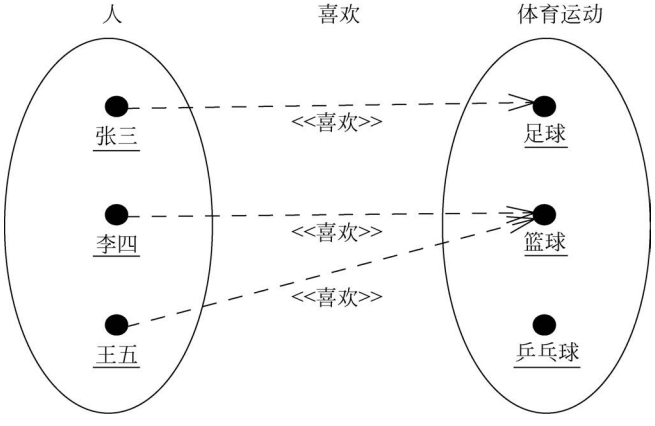

图3.1 人“喜欢”体育运动的语义

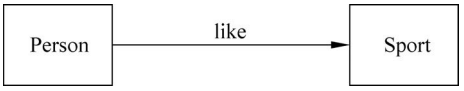

图 3.2 “人喜欢体育运动”的类图

如图 3.1 所示,“人喜欢体育运动”中的“喜欢”表示了一种对应关系,可使用类图表示这种对应关系。表示“人喜欢体育运动”的类图,如图 3.2 所示。

如图 3.2 所示的类图,表达的意思是“Person like Sport”,翻译为中文是“人喜欢体育运动”,其数学上的语义如图 3.1 所示,表示集合“人”和“体育运动”之间的一种对应关系。

面向对象思想中,将类与类之间的一种对应关系称为一个关联,其中的一个对应称为该关联的一个连接。

如图 3.2 中,like 是类 Person 和 Sport 之间的一个关联。关联 like 中包含 Person 对象与 Sport 对象之间的多个对应,每个对应称为关联 like 的一个连接。如图 3.1 中,(张三,足球)、(李四,篮球)和(王五,篮球)是关联 like 中的三个连接,分别表示张三喜欢足球、李四喜欢篮球、王五喜欢篮球。

3.2 关联的实现

对应关系有一对一、多对一和多对多 3 种类型,对应关系的类型对计算机中的实现有重要影响,因此,数学中专门讨论了对应关系的 3 种类型,并为计算机中的实现提供了理论基础。

对应关系的 3 种类型中,可将一对一关系视为特殊的多对一关系,因此,只需讨论多对一和多对多两种关联的实现方法。

3.2.1 使用指针实现多对一关联

一般需要结合具体的应用场景分析对应关系的类型。例如,如果需要回答“每个人最喜欢的体育运动是什么?”,可将人与体育运动的对应关系类型视为“多对一”关系,即一个人喜欢一种体育运动,一种体育运动可被多个人喜欢,其类图如图 3.3 所示。

图 3.3 最喜欢的体育运动

图 3.3 中,将“一个人喜欢一种体育运动”中的“一种”标注在 Sport 端,并表示为数字“1”。将“一种体育运动可被多个人喜欢”中的“多个”标注在 Person 端,并表示为符号“*”。

关联 like 从 Person 端关联到 Sport 端,具有方向性,将表示方向的箭头称为导航(Navigation),将关联的“端”称为角色(Role),将表示对应数目的“1”和“*”称为重数(Multiplicity)。

关联的导航、角色和重数对编程实现有很大影响,首先需要明确标注这些信息,然后为关联中的每个类增加属性和成员函数。关联 like 及其中的类,如图 3.4 所示。

类 Person 有属性 sport,属性 sport 不是“人”的属性,而是用于存储关联 like 中的一个连接,其数据类型为指向 Sport 的指针,指向最喜欢的运动。

实际上,类 Person 的对象中使用成员指针 sport 指向一个类 Sport 的对象,表示多对一关联 like 中的一个连接。使用成员指针表示多对一关联中的连接,示例代码如例 3.1 所示。

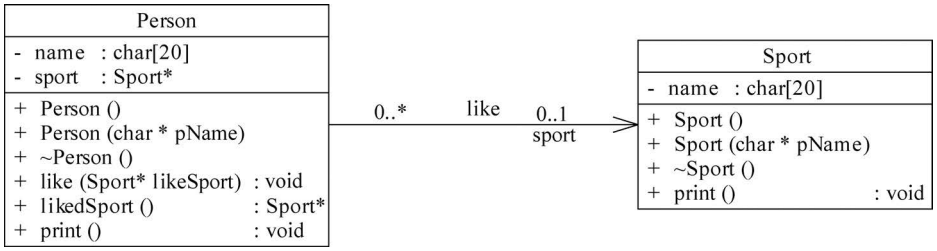

图 3.4　关联 like 及其中的类

【例 3.1】 使用成员指针表示多对一关联中的连接。

```
// Sport.h
class Sport
{
public:
    Sport();
    Sport(char * pName);
    void print();
    ~Sport();

private:
    char name[20];
};
```

```
// Person.h
#include "Sport.h"
class Person
{
public:
    Person();
    Person(char * pName);
    ~Person();
    void like(Sport * likeSport);
    Sport * likedSport();
    void print();

private:
    char name[20];
    Sport * sport;    //指向最喜欢的运动
};
```

```
//Sport.cpp
#include <iostream>
#include <string.h>
#include "Sport.h"
using namespace std;

Sport::Sport(){
}
Sport::Sport(char * pName){
    strncpy(name, pName, sizeof(name));
```

```
    name[sizeof(name) - 1] = '\0';
}
void Sport::print(){
    cout << name << endl;
}
Sport::~Sport(){
}
```

类 Person 中,声明了 void like(Sport * likeSport)和 Sport * likedSport()两个成员函数,其中,like()成员函数的功能为将最喜欢的体育运动 likeSport 的地址存储到属性 sport。likedSport()成员函数返回最喜欢的体育运动 likeSport 的地址。这两个成员函数主要维护了关联 like 中的连接,示例代码如例 3.2 所示。

【例 3.2】 维护多对一关联中的连接。

```
//Person.cpp
#include <iostream>
#include <string.h>
#include "Sport.h"
#include "Person.h"
using namespace std;

Person::Person(){
}
Person::Person(char * pName){
    strncpy(name, pName, sizeof(name));
    name[sizeof(name) - 1] = '\0';
}
Person::~Person(){
}
void Person::like(Sport * likeSport){
    sport = likeSport;
}
Sport * Person::likedSport(){
    return sport;
}
void Person::print(){
    cout << name;
}
```

```
//app.cpp
#include <iostream>
#include "Sport.h"
#include "Person.h"
using namespace std;

void main(){
    Sport s1("足球");
    Sport s2("篮球");
    Sport s3("乒乓球");

    Person p1("张三");
```

```
    p1.like(&s1);
    p1.print();
    cout << "最喜欢";
    p1.likedSport() -> print();

    Person p2("李四");
    p2.like(&s2);
    p2.print();
    cout << "最喜欢";
    p2.likedSport() -> print();

    Person p3("王五");
    p3.like(&s2);
    p3.print();
    cout << "最喜欢";
    p3.likedSport() -> print();
}
```

例 3.2 中，表达式语句 p1.like(&s1)调用类 Person 的 like()成员函数，设置了张三最喜欢的足球，建立了类 Person 的对象 p1 与类 Sport 的对象 s1 之间的一个连接。

表达式 p1.likedSport()-> print()包含 4 个运算，其计算顺序如图 3.5 所示。

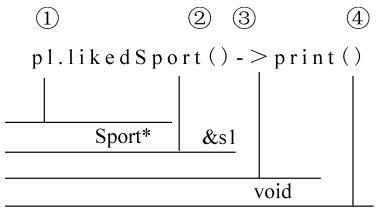

图 3.5　表达式 p1.likedSport()-> print()的计算顺序

如图 3.5 所示，表达式 p1.likedSport()-> print()的语义为：①计算 p1.likedSport()中的点运算，选择类 Person 的 likedSport()成员函数；②调用 Person::likedSport()成员函数，返回一个指向 Sport 对象 s1 的指针 &s；③计算选择运算(->)，选择类 Sport 的 print()成员函数；④调用 Sport::print()成员函数，输出张三最喜欢的“足球”。

例 3.2 中，先创建了类 Sport 的 3 个对象，然后创建类 Person 的对象 p1("张三")，并执行函数调用 p1.like(&s1)设置喜欢的体育运动项目 s1，最后输出。程序运行过程如图 3.6 所示。

main()函数中，创建了类 Person 的 3 个对象，类 Sport 的 3 个对象，建立了关联 like 中的 3 个连接。创建的对象及其连接，如图 3.7 所示。

如图 3.7 所示，对象 p2 和 p3 连接到同一个对象 s2，即“篮球”项目，符合实际情况。

例 3.2 程序的输出结果如下。

```
张三最喜欢足球
李四最喜欢篮球
王五最喜欢篮球
```

使用指针实现多对一关系，能够准确表示多对一关联的语义，是编程中最常用的方法之一。

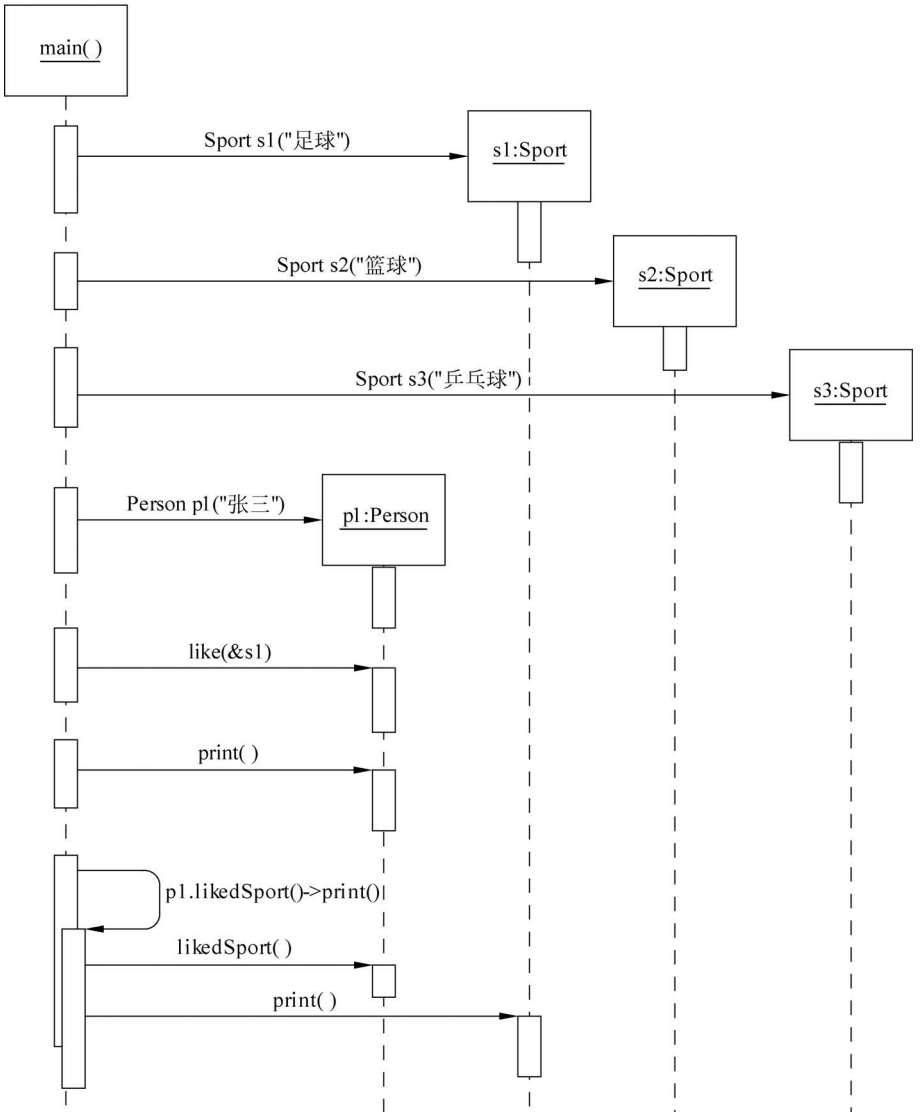

图 3.6　例 3.1 程序运行过程

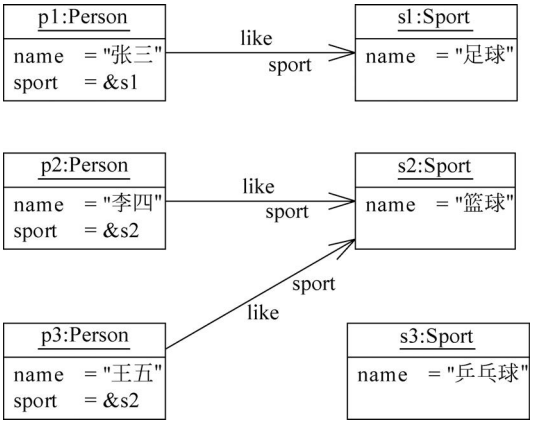

图 3.7　例 3.1 中创建的对象及其连接

3.2.2　使用指针数组实现多对多关联

如果想知道一个人喜欢的所有体育运动项目，则人和体育运动之间的关联就是多对多类型，可以使用指针数组来存储这种关系。为了简单明了，假设一个人最多喜欢 3 项体育运动，多对多关联如图 3.8 所示。

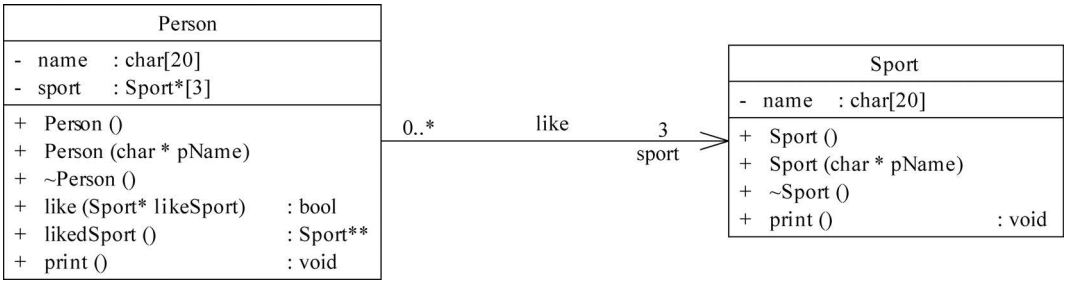

图 3.8　多对多关联

如图 3.8 所示的类图中，Sport 端的重数为 3，表示“一个人最多喜欢 3 项体育运动”，并使用一个指向 Sport 对象的指针数组 sport[3]表示关联 like 中的连接。使用指针数组表示多对多关联中的连接，示例代码如例 3.3 所示。

【例 3.3】　使用指针数组表示多对多关联中的连接。

```
// Person.h
#include "Sport.h"
class Person
{
public:
    Person();
    Person(char * pName);
    ~Person();
    bool like(Sport * likeSport);
    Sport ** likedSport();
    void print();

private:
    char name[20];
    Sport * sport[3];   //指向喜欢的 3 项运动
};
```

使用指针数组 sport 表示多对多关联 like 中的连接，需要根据这种表示方法调用与此相关的成员函数，以正确维护关联 like 中的连接，示例代码如例 3.4 所示。

【例 3.4】　维护多对多关联中的连接。

```
//Person.cpp
#include <iostream>
#include <string.h>
#include "Sport.h"
#include "Person.h"
using namespace std;
```

```
Person::Person(){
    for (int i = 0; i < 3; i++)
        sport[i] = NULL;          //设置为空,表示没有指向对象
}
Person::Person(char * pName){
    strncpy(name, pName, sizeof(name));
    name[sizeof(name) - 1] = '\0';
    for (int i = 0; i < 3; i++)
        sport[i] = NULL;
}
Person::~Person(){
}
bool Person::like(Sport * likeSport){
    int i = 0;
    while (i < 3&&sport[i])
        i++;
    if (i < 3){
        sport[i] = likeSport;
        return true;
    }
    else{
        return false;             //超过 3 项返回错误
    }
}
Sport ** Person::likedSport(){
    return sport;
}
void Person::print(){
    cout << name;
    cout << "喜欢";
    for (int i = 0; i < 3; i++)
        sport[i] -> print();
}
```

主要调用了 like()和 likedSport()成员函数。likedSport()成员函数的原型调整为 Sport ** likedSport(),返回一个指针数组。like()成员函数的原型调整为 bool like(Sport * likeSport),每次调用传递一项体育运动项目,可多次调用,如果超过 3 项就返回 false,表示存储失败。

```
//app.cpp
//#include "Sport.h"
//#include "Person.h"
using namespace std;
void main(){
    Sport s1("足球");
    Sport s2("篮球");
    Sport s3("乒乓球");
    Sport s4("跳高");

    Person p1("张三");
    p1.like(&s1);
```

```
    p1.like(&s2);
    p1.like(&s3);
    p1.like(&s4);     //超过3项不存储
    p1.print();

    Person p2("李四");
    p2.like(&s4);
    p2.like(&s3);
    p2.like(&s2);
    p2.like(&s1);     //超过3项不存储
    p2.print();
}
```

例3.4中，创建了2个Person对象，4个Sport对象。Person对象和Sport对象之间是多对多对应关系，其多对多对应关系如图3.9所示。

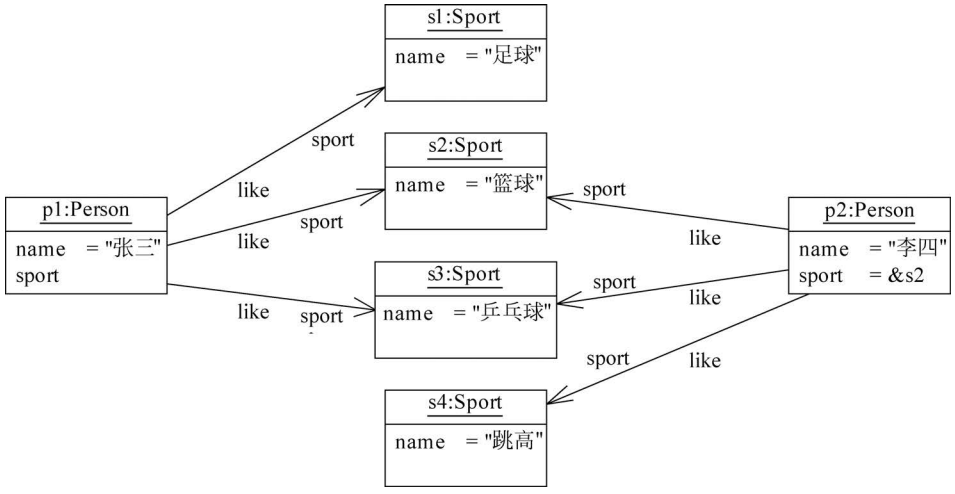

图3.9　多对多对应关系

例3.4程序的输出结果如下。

```
张三喜欢足球
篮球
乒乓球
李四喜欢跳高
乒乓球
篮球
```

上面介绍了使用指针或指针数组表示关联的两种方法，这两种表示方法都能够准确地表示关联的语义。

但很多面向对象程序设计语言不支持或不建议使用指针，怎么办？

解决的办法是将指针改为引用，即使用对象的引用来实现关联。从本质上讲，引用是对指针的封装，引用的内部实现仍然使用指针，但引用更加安全。

可将上述示例程序中的指针修改为引用，使用引用或引用数组来实现关联。建议读者尝试一下，一定会发现很多问题，从而对学习编程有更深刻的体会。

3.3 组合与聚合关联


视频讲解

关联分为一般关联(Association)、聚合(Aggregation)和组合(Composition)3 种类型。一般关联,主要描述客观事物之间的相互关系,前面已经进行了介绍。

在介绍聚合和组合两个概念前,先讨论观察事物内部构成的思维方式。按照“一个客观事物由更小的客观事物构成的”观点,可将一个客观事物视为一个“整体”,将构成这个事物的更小事物视为“整体”的一个“部分”。

例如,一个人由头、躯体、肢体组成。具体地说,一个人是一个整体,包含一个头、一个躯体和四肢等部分,而每个部分也可以再细分。例如,头可分为眉、眼、耳、鼻、口等五官,包含两条眉、两只眼、两只耳、一只鼻、一张口。人体的构成,如图 3.10 所示。

例如,汽车由很多系统组成,包含很多零部分,主要有发动机、车轮等,可用类图描述汽车与发动机、轮子之间的关系,其类图如图 3.11 所示。

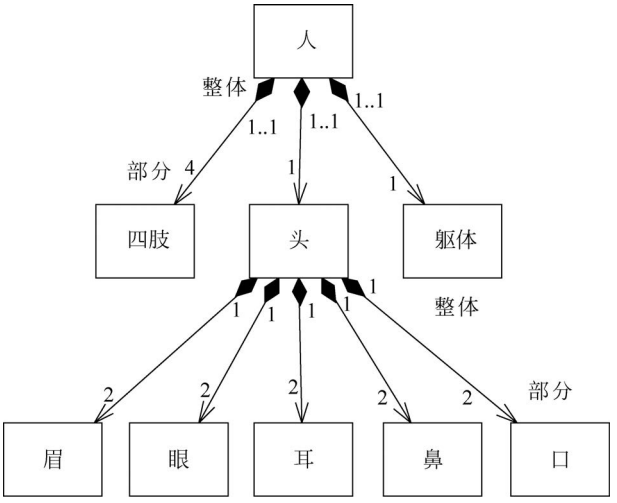

图 3.10 人体的构成

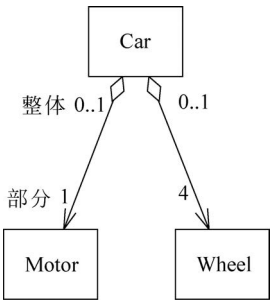

图 3.11 汽车的类图

如图 3.11 所示,汽车包含发动机和车轮,一辆汽车有一台发动机,有 4 个车轮。一台发动机可安装到一辆汽车,一个车轮可安装到一辆汽车,但安装到一辆汽车后就不能同时安装到另外一辆汽车。

每个事物都有自己的生命周期,不仅要观察每个事物由哪些“部分”构成,还要观察每个“部分”的生命周期与事物的生命周期是否同步。

当一个人出生时,这个人就有了头、躯体、肢体,当一个去世时,这个人的头、躯体、肢体同时死亡,因此,人这个“整体”与头、躯体、肢体等“部分”具有相同的生命周期。

可以将一辆汽车的发动机和车轮从该汽车上拆卸下来,再安装到另外一辆汽车上,显然,汽车这个“整体”与其“部分”具有不同的生命周期。

在面向对象程序设计中,组合关联和聚合关联都是描述“整体”与“部分”之间的构成关系,并将具有相同生命周期的构成关系称为组合关联,将具有不同生命周期的构成关系称为聚合关联。

图 3.10 中，使用组合关联来描述人的内部结构；图 3.11 中，使用聚合关联来描述汽车的内部结构。

3.3.1　使用对象实现组合关联

当一个学生入学时，学校为每个学生分配一个唯一的学号，学生与学号之间具有组合关联的特点，可使用组合关联描述学生与学号之间的关系。学生和学号之间的组合关联，如图 3.12 所示。

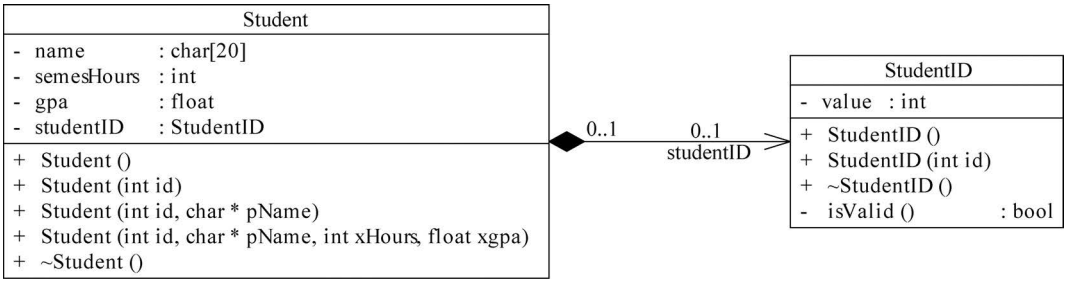

图 3.12　学生和学号之间的组合关联

一般使用对象表示多对一组合关联中的连接。例如，类 Student 中，声明了属性 studentID，其数据类型为类 StudentID，用于存储类 StudentID 的一个对象，表示多对一组合关联 studentID 中的一个连接，示例代码如例 3.5 所示。

【例 3.5】 使用对象表示多对一组合关联的连接。

```
//StudentID.h
class StudentID
{
public:
    StudentID();
    StudentID(int id);
    ~StudentID();
    void print();

private:
    bool isValid(void);
    int value;
};
```

```
//StudentID.cpp
#include <iostream>
#include <string.h>
#include "StudentID.h"
using namespace std;

StudentID::StudentID(){
    cout << "\t" << "调用构造函数 StudentID()" << endl;
}
StudentID::StudentID(int id){
    cout << "\t"<<"调用构造函数 StudentID(" << id << ")" << endl;
```

```
    if (isValid())
        value = id;
}
StudentID::~StudentID(){
    cout << "析构 StudentID:" << value << endl;
}
void StudentID::print(){
    cout << value << endl;
}
bool StudentID::isValid(){
    //可增加判断学号是否符合编码规则的代码
    return true;
}
```

```
//Student.h
#include "StudentID.h"

class Student
{
public:
    Student();
    Student(int id);
    Student(int id, char * pName);
    Student(int id, char * pName, int xHours, float xgpa);
    ~Student();

private:
    char name[20];
    int semesHours;
    float gpa;
    StudentID studentID;   //表示多对一组合关联中的连接
};
```

为了同步“整体”和“部分”之间的生命周期,在构造函数和析构函数的基础上增加了一种表达方式,专门用于维护多对一组合关联的连接,示例代码如例 3.6 所示。

【例 3.6】 维护多对一组合关联的连接。

```
// Student.cpp
#include <iostream>
#include <string.h>
#include "StudentID.h"
#include "Student.h"
using namespace std;

Student::Student(){
    cout << "调用构造函数 Student()" << endl << endl;
}

Student::Student(int id) :studentID(id)  //增加了新的语法
{
    cout << "调用构造函数 Student(" << id << ")" << endl << endl;
```

```
}

Student::Student(int id, char * pName) :studentID(id)
{
    cout << "调用构造函数 Student(" << id << "," << pName
        << ")" << endl << endl;
    strncpy(name, pName, sizeof(name));
    name[sizeof(name) - 1] = '\0';
}

Student::Student(int id, char * pName, int xHours, float xgpa) :studentID(id)
{
    cout << "调用构造函数 Student(" << id << "," << pName
        << "," << xHours << "," << xgpa
        << ")" << endl << endl;
    strncpy(name, pName, sizeof(name));
    name[sizeof(name) - 1] = '\0';
    semesHours = xHours;
    gpa = xgpa;
}

Student::~Student()
{
    cout << "析构 Student:";
    studentID.print();
}
```

Student 的构造函数中，使用冒号(:)语法同步 Student 对象与成员对象 studentID 的生命周期。例如，语句 Student::Student(int id):studentID(id)，冒号的前面是构造函数的原型 Student::Student(int id)，而后面的 studentID(id)规定了创建成员对象 studentID 的方法，即创建 Student 对象的过程中，按照 StudentID studentID(id)的语义创建成员对象 studentID。

```
//app.cpp
#include "StudentID.h"
#include "Student.h"

void main()
{
    Student s1(210101,"Randy");
    Student s2(210102, "Randy");
    Student s3(210103, "Jenny",10,3.5);
}
```

main()函数中使用 3 条语句创建了类 Student 的 3 个对象。其中，语句 Student s1(210101,"Randy")创建学生 Student 的对象 s1，对象 s1 的创建过程如图 3.13 所示。

如图 3.13 所示，语句 Student s1(210101,"Randy")创建学生 Student 的对象 s1 时，先按照声明顺序依次给成员变量 name[20]、semesHours、gpa 和 studentID 分配内存，然后再执行函数调用 Student(210101,"Randy")初始化对象 s1。

执行函数调用 Student(210101,"Randy")过程中，先将实参 210101 和"Randy"分别

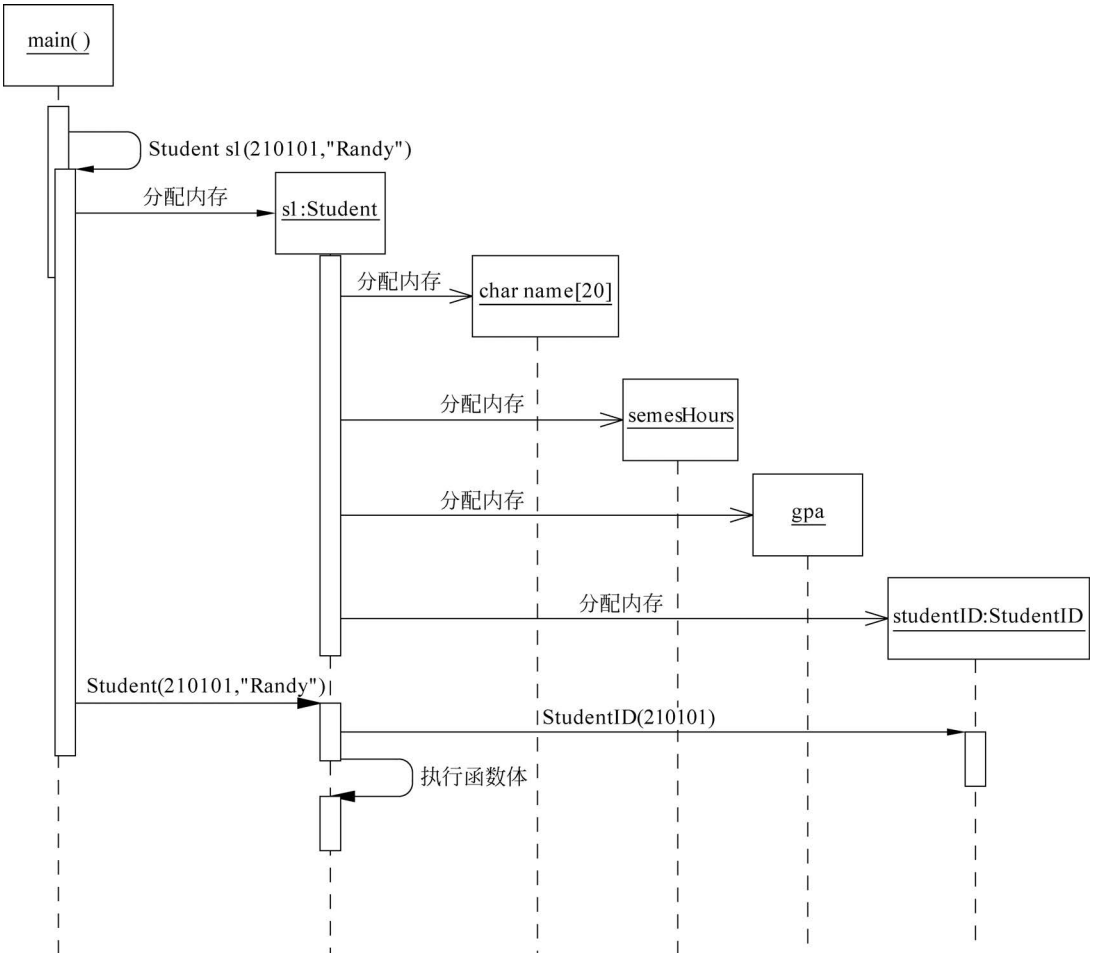

图 3.13　对象 s1 的创建过程

传递给形参 id 和 pName，然后执行函数调用 studentID. StudentID(id)初始化成员对象 studentID，最后执行类 Student 的构造函数的函数体。对象 s1 的物理结构如图 3.14 所示。

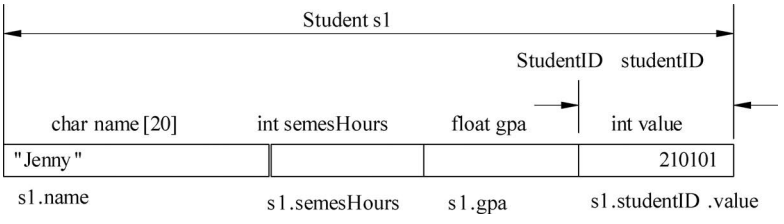

图 3.14　对象 s1 的物理结构

如图 3.14 所示，对象 s1 的内存中包含对象 s1. studentID 的内存，对象 s1. studentID 的内存中包含 s1. studentID. value 的内存，其中的值为学号 210101。

创建和删除对象 s1 过程中，都将对象视为一个整体来管理，并没有区分是成员对象还是成员变量。这样，程序员就不需要关心对象的内部结构，不需要考虑哪些是成员变量，哪些是成员对象，都可以当成变量来处理，使用非常方便。

例 3.6 中，总共创建了类 Student 的 3 个对象和 StudentID 的 3 个对象，每个 Student 对象包含一个 StudentID 对象，表示组合关联 studentID 中的一个连接。组合关联 studentID 中的连接如图 3.15 所示。

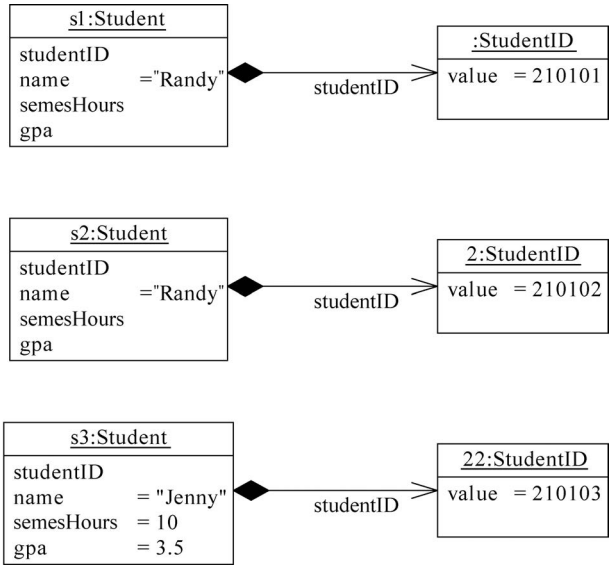

图 3.15　组合关联 studentID 中的连接

例 3.6 程序的输出结果如下。

```
        调用构造函数 StudentID(210101)
调用构造函数 Student(210101,Randy)

        调用构造函数 StudentID(210102)
调用构造函数 Student(210102,Randy)

        调用构造函数 StudentID(210103)
调用构造函数 Student(210103,Jenny,10,3.5)

析构 Student:210103
析构 StudentID:210103
析构 Student:210102
析构 StudentID:210102
析构 Student:210101
析构 StudentID:210101
```

从逻辑上讲，类 Student 的对象中的成员对象 studentID，也是一个对象，创建时也要调用类 StudentID 的构造函数，在删除时也要调用其析构函数。删除对象的步骤是，先调用类 StudentID 的析构函数，再调用类 Student 的析构函数，最后回收类 Student 的对象的所有内存。

例 3.6 中，创建了类 Student 的对象及其类 StudentID 的对象，在删除类 Student 的对象时，也同时删除了类 StudentID 的对象，同步了类 Student 的对象及其类 StudentID 对象的生命周期，实现了多对一组合关联的语义。

3.3.2 使用指针实现组合关联

除了使用对象实现组合关联外,还可使用指向对象的指针实现组合关联,但需要编写代码同步对象的生命周期。

例如,可使用指针实现类 Person 和 Tdate 之间的组合关联,但需要在构造函数和析构函数中编写代码以同步对象的生命周期。使用指针存储组合关联中的连接,如图 3.16 所示。

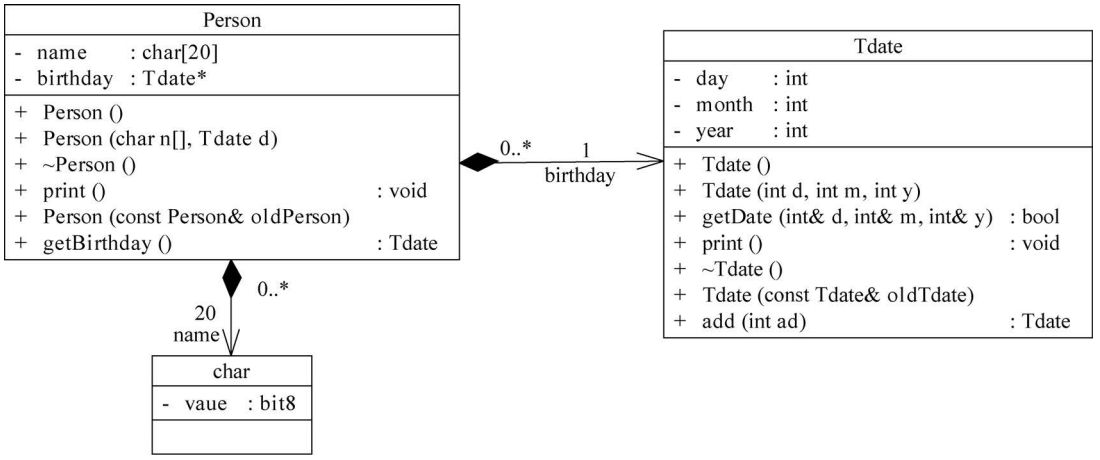

图 3.16 使用指针存储组合关联中的连接

如图 3.16 所示的类 Person 中,声明了一个属性 birthday,用于指向组合关联 birthday 中连接的 Tdate 对象。Person 对象的内存中只存储了指针 birthday,而没有存储其指向的 Tdate 对象。

因连接的 Tdate 对象不在 Person 对象的内存,创建 Person 对象时系统不会自动创建连接的 Tdate 对象,删除 Person 对象时系统也不会自动删除连接的 Tdate 对象,因此,需要将创建 Tdate 的职责赋予 Person 的构造函数和拷贝构造函数,删除 Tdate 的职责赋予 Person 的析构函数,以实现组合关联的语义。使用指针实现组合关联的示例代码如例 3.7 所示。

【例 3.7】 使用指针实现组合关联。

```
#include <iostream>
#include "Tdate.h"
#include <iostream>
using namespace std;
class Person
{
public:
    Person(){};
    Person(char n[],Tdate d) {
        strncpy(name, n, sizeof(name));
        name[sizeof(name) - 1] = '\0';
        birthday = new Tdate(d);                    //创建一个新的 Tdate 对象
    };
```

```
    ~Person(){
        delete birthday;            //删除连接的 Tdate 对象
    };
    void print(){
        cout << "Person:"<< name <<",";
        birthday -> print();
    };
    Tdate getBirthday() const{
        return Tdate( * birthday);  //返回一个新对象而没有返回指针 birthday,以保证安全
    }
    Person(const Person& oldPerson){
        memcpy(this, &oldPerson, sizeof(Person)); // 将 oldPerson 的内存中的数据复制到内存
        birthday = new Tdate(oldPerson.getBirthday());   //创建一个新的 Tdate 对象
    };
private:
    Tdate * birthday;               //使用指针表示多对一组合关联
    char name[20];
};
```

拷贝构造函数 Person(const Person& oldPerson)中,语句 memcpy(this, &oldPerson, sizeof(Person))将 oldPerson 的内存中的数据复制到当前对象的内存中,其中只复制了指向连接对象的指针,需要为当前 Person 对象重新创建一个 Tdate 对象。语句 birthday = new Tdate(oldPerson.getBirthday())就实现了一功能。

```
Person fn(Person p){
    return p;
}
Person * fnPtr(Person * p){
    return p;
}
void main(){
    Tdate d1(1, 2, 2000);
    Tdate d2(1, 2, 2021);

    cout << " **** Person 对象 **** " << endl;
    Person p1("张三",d1);
    Person p2("李四", d1);

    cout << " **** 传递 * Person 对象 **** " << endl;
    fn(p1).print();

    cout << " **** 传递 * Person 对象指针 **** " << endl;
    fnPtr(&p2) -> print();

    cout << " **** main 语句结束 **** " << endl;
}
```

例 3.7 程序的输出结果如下。

```
构造:00CFF710 -> 2/1/2000
构造:00CFF6FC -> 2/1/2021
```

```
**** Person 对象 ****
拷贝构造:00CFF58C -> 2/1/2000
拷贝构造:01059D30 -> 2/1/2000
析构:2/1/2000
拷贝构造:00CFF58C -> 2/1/2000
拷贝构造:0105BA58 -> 2/1/2000
析构:2/1/2000
**** 传递 * Person 对象 ****
拷贝构造:0105BAA0 -> 2/1/2000
拷贝构造:0105BAE8 -> 2/1/2000
析构:2/1/2000
Person:张三,2/1/2000
析构:2/1/2000
**** 传递 * Person 指针 ****
Person:李四,2/1/2000
**** main 语句结束 ****
```

例 3.7 中创建了 2 个 Person 对象,4 个 Tdate 对象。2 个 Person 对象分别连接到其中的 2 个 Tdate 对象,创建的对象及其连接如图 3.17 所示。

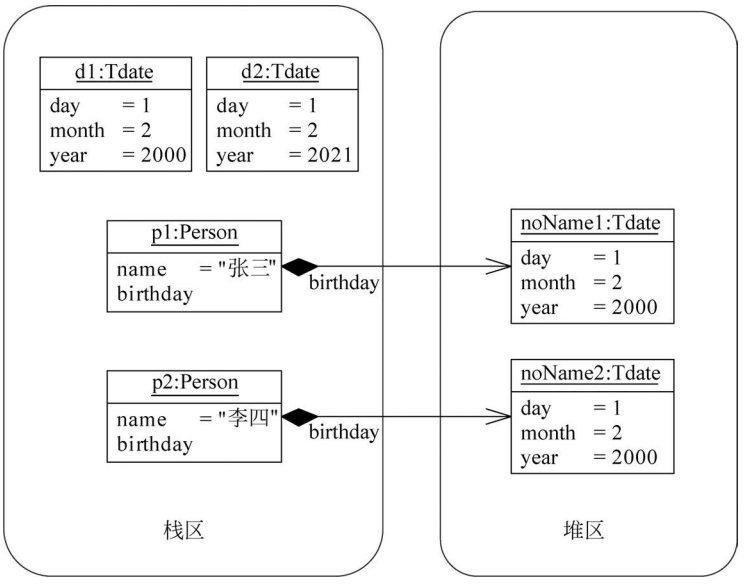

图 3.17 例 3.7 创建的对象及其连接

创建 2 个 Person 对象时,同时在堆中创建了 2 个无名 Tdate 对象,删除这两个 Person 对象时,也同时删除了堆中的 2 个无名 Tdate 对象。Person 对象的创建和删除过程如图 3.18 所示。

fn(p1).print()和 fnPtr(&p2)->print()两条语句,演示了采用传值和传地址两种方式传递 Person 对象的过程。函数调用 fn(p1)中,调用了拷贝构造函数传递和返回 Person 对象。其中,创建和删除了连接的 Tdate 对象,而函数调用 fnPtr(&p2)中,只传递和返回了 Person 对象的地址,效率明显高于函数调用 fn(p1)。

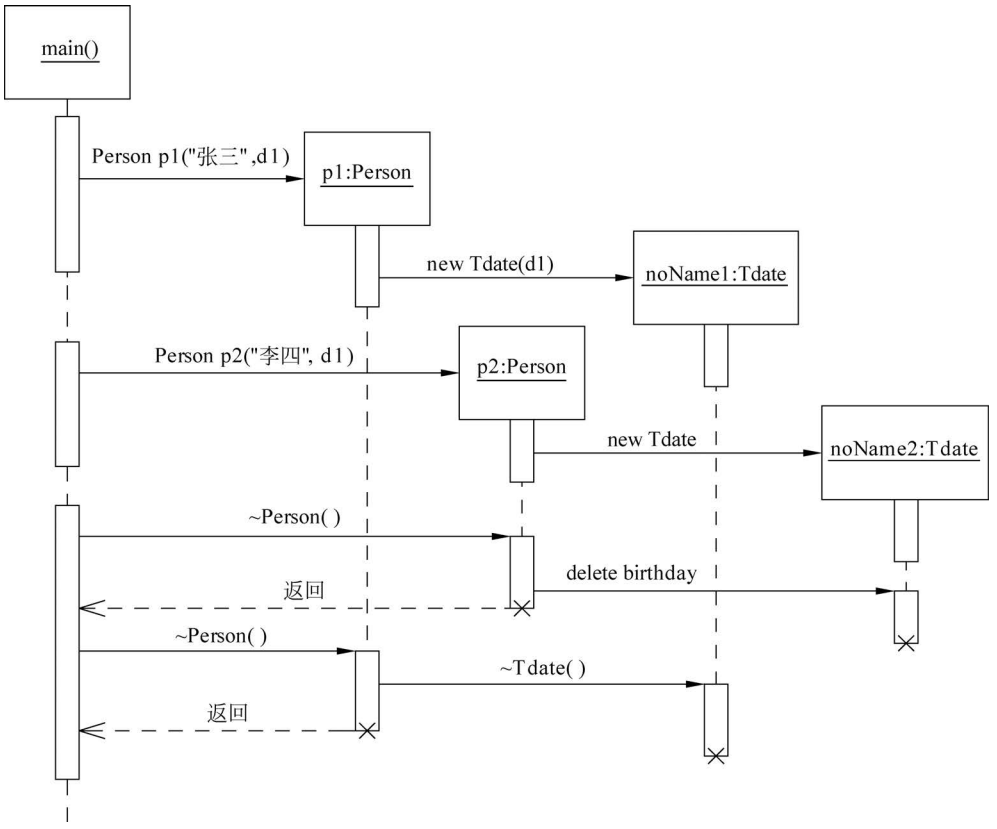

图 3.18　Person 对象的创建和删除过程

类 Person 中，类 Tdate 主要起到数据类型的作用，这就要求类 Tdate 的对象具有更好的计算特性。

作为数据类型使用的类，主要是为了计算，一般不会关注其具体的业务功能。

在组合关联中，由于“整体”和“部分”的生命周期相同，在创建“整体”时需要同时创建“部分”，在删除“整体”时需要同时删除“部分”，创建和删除“部分”的职责自然而然地赋予了构造函数和析构函数。

3.3.3　使用代码实现聚合关联

聚合关联中，“整体”和“部分”都有自己的生命周期，由于“整体”和“部分”的生命周期不同，一般不会使用成员对象表示聚合关联中的连接，而使用成员指针表示聚合关联中的连接。

例如，图 3.11 中，汽车与发动机、车轮之间的关系都是聚合关联，可使用指针表示汽车与发动机之间的连接，使用指针数组表示汽车与车轮之间的多个连接。汽车及其部件如图 3.19 所示。

如图 3.19 所示，类 Motor 和 Wheel 中，都设计了属性 serialNumber，该属性用于存储产品序列号，以保证计算机中每一个对象都与实际的汽车部件一一对应。类 Motor 和 Wheel 的代码如例 3.8 所示。

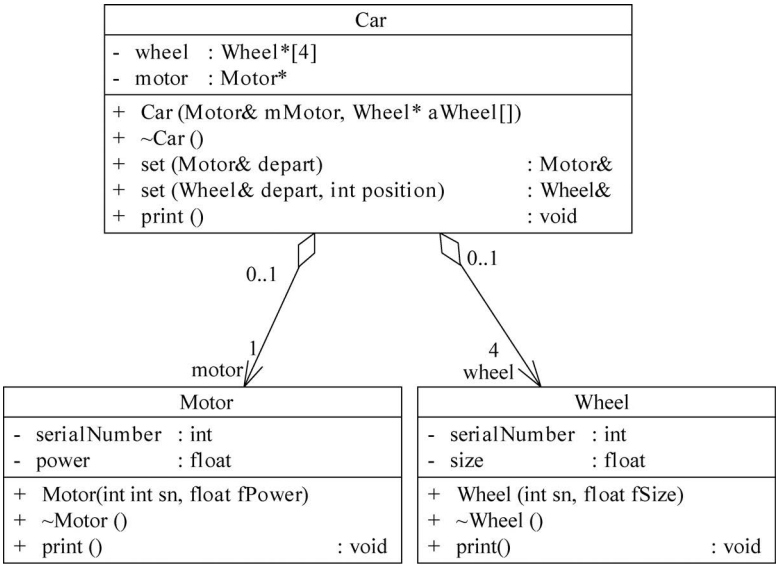

图 3.19 汽车及其部件

【例 3.8】 类 Motor 和 Wheel。

```
// Motor.h
class Motor
{
public:
    Motor(int sn,float fPower);
    ~Motor();
    void print();

private:
    int serialNumber;              //产品序列号
    float power;                   //发动机的排量
};
```

```
// Motor.cpp
#include <iostream>
#include "Motor.h"
using namespace std;

Motor::Motor(int sn,float fPower){
    cout << "调用构造函数 Motor(" << sn << "," << fPower << ")" << endl ;
    serialNumber = sn;
    power = fPower;
}
Motor::~Motor(){
    cout << "析构 Motor:" << serialNumber << endl;
}
void Motor::print(){
    cout << serialNumber << "," << power ;
}
```

```
// Wheel.h
class Wheel
{
public:
    Wheel(int sn,float fSize);
    ~Wheel();
    void print();
private:
    int serialNumber;              //产品序列号
    float size;                    //车轮大小
};

//Wheel.cpp
#include "Wheel.h"
#include <iostream>
using namespace std;

Wheel::Wheel(int sn, float fSize){
    cout << "调用构造函数 Wheel(" << sn <<","<< fSize << ")" << endl ;
    serialNumber = sn;
    size = fSize;
}
Wheel::~Wheel(){
    cout << "析构 Wheel:" << serialNumber << endl;
}
void Wheel::print(){
    cout << serialNumber << "," << size ;
}
```

类 Car 中设计了 wheel 和 motor 两个属性，这两个属性分别表示两个聚合关联 wheel 和 motor，其中，属性 wheel 是一个指针数组，共有 4 个指针，分别对应一辆汽车的 4 个车轮；属性 motor 是指向类 Motor 的一个对象，对应汽车的发动机。

类 Car 中还设计了 5 个成员函数，其中，Car(Motor& mMotor, Wheel * aWheel[])构造函数有两个参数，第一个参数是 Motor 的引用，用于传递代表发动机的对象；第二个参数是一个指针数组，用于传递代表车轮的 4 个对象，其功能相当于将一台发动机和 4 个车轮组装成一辆汽车，以模拟汽车的组装工序。

Motor&set(Motor&depart)成员函数的功能为替换汽车的发动机，将通过参数 depart 传递的发动机安装(set)到汽车上，并以引用方式返回拆卸下来的发动机。Wheel&set(Wheel&depart, int position) 成员函数的功能为更换一个车轮，将通过参数 depart 传递的一个车轮安装(set)在汽车的位置(position)上，并以引用方式返回拆卸下来的车轮。这两个成员函数模拟了更换汽车部件的实际场景。类 Car 的代码如例 3.9 所示。

【例 3.9】 类 Car。

```
// Car.h
#include"Wheel.h"
#include "Motor.h"

class Car
```

```
{
public:
    Car(Motor& mMotor, Wheel ** aWheel);
    ~Car();
    Motor& set(Motor& depart);
    Wheel& set(Wheel& depart, int position);
    void print();

private:
    Wheel * wheel[4];   //指向 4 个车轮
    Motor * motor;      //指向发动机
};
```

其中,Wheel * wheel[4]声明了一个指针数组,用于指向代表 4 个车轮的对象。Motor * motor 声明了一个指针,用于指向代表发动机的对象。

```
// Car.cpp
#include "Motor.h"
#include "Wheel.h"
#include "Car.h"
#include <iostream>
using namespace std;

Car::Car(Motor& mMotor, Wheel * aWheel[]) :motor(&mMotor)
{
    cout << "调用构造函数 Car(";
    mMotor.print();
    cout << ")" << endl;

    for (int i = 0; i < 4; i++){
        wheel[i] = aWheel[i];
    }
}
Car::~Car(){
    cout << "析构 Car:" << endl;
    for (int i = 0; i < 4; i++){        //删除 4 个车轮对象
        delete wheel[i];
    }
    delete motor;                       //删除发动机对象
}
Motor& Car::set(Motor& depart){
    Motor& rt = *motor;
    motor = &depart;
    return rt;
}
Wheel& Car::set(Wheel& depart, int position){
    Wheel& rt = *wheel[position];
    wheel[position] = &depart;
    return rt;
}
void Car::print(){
    cout << "汽车\t 发动机:";
```

```
    motor->print();
    cout << "\t\t 车轮:";
    for (int i = 0; i < 4; i++){
        wheel[i]->print();
        cout << "|";
    }
    cout << endl;
}
```

在制造汽车的过程中,最后将已制造出的各种部件组装成一辆汽车,因此,类 Car 的构造函数中没有创建 Motor 和 Wheel 的对象,只存储了表示连接的指针。但当一辆汽车报废时,需要报废汽车的所有部件,因此,析构函数~Car()中,删除了代表发动机和 4 个车轮的对象。

参考汽车的装配场景,编程一个主程序,示例代码如例 3.10 所示。

【例 3.10】 主程序。

```
//CarApp.cpp
#include "Motor.h"
#include "Wheel.h"
#include "Car.h"
#include <iostream>
using namespace std;

void main(){
    cout << "********** 创建发动机 **********" << endl;
    Motor& m1 = *(new Motor(101,1.6));          //为堆中的对象取了一个名称,以便后面使用
    Motor& m2 = *(new Motor(102,2.0));
    Motor& m3 = *(new Motor(103,1.6));
    cout << "********** 创建车轮 **********" << endl;
    const int len = 10;
    Wheel* wBase[len];
    for (int i = 0; i < len; i++){
        wBase[i] = new Wheel(201+i,10);
    }
    cout << "********** 创建汽车 **********" << endl;
    Car& c1 = *(new Car(m1, wBase));
    c1.print();
    Car& c2 = *(new Car(m2, wBase+4));
    c2.print();
    cout << "********** 更换并报废部件 **********" << endl;
    Wheel&w1 = c1.set(*wBase[8], 1);            //换第一个车轮
    delete &w1;                                 //报废车轮,删除它

    Motor&t = c1.set(m3);                       //换发动机
    delete &t;                                  //报废发动机,删除它
    cout << "********** 报废汽车 **********" << endl;
    delete &c1;
    delete &c2;
    cout << "********** 删除库存中没有使用的车轮 **********" << endl;
    delete wBase[9];
}
```

CarApp.cpp 中,负责创建类 Car、Motor 和 Wheel 的对象,也负责删除没有安装到汽车上的 Motor 和 Wheel 对象。类 Car、Motor 和 Wheel 的对象都存储在堆区中,需要通过代

码来创建和删除这些对象。

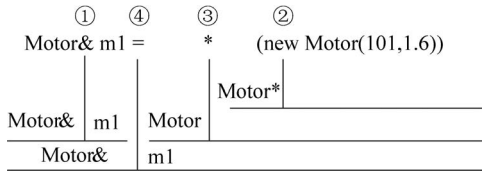

图 3.20 表达式 Motor& m1= *(new Motor(101,1.6))的计算顺序

表达式语句 Motor&m1 = *(new Motor(101,1.6))的作用是在堆区中创建 Motor 的一个无名对象,并给它取一个别名,其计算顺序如图 3.20 所示。

如图 3.20 所示,表达式 Motor& m1 = *(new Motor(101,1.6))中包含 4 个步骤:①定义 Motor 对象的一个引用 m1,从本质上讲,可将变量或对象的定义视为一种运算,事实上很多面向对象程序设计语言也是这样做的;②在堆中创建 Motor 的一个无名对象,并返回这个对象的指针;③取指针的对象,即堆中创建的无名对象;④将引用 m1 初始化为无名对象,即使用 m1 命名无名对象。

main()函数中,先在堆中创建了类 Motor 的 3 个对象 m1、m2 和 m3,然后创建了类 Wheel 的 10 个对象,并用指针数组存储 10 个对象的指针,最后创建了类 Car 的两个对象 c1 和 c2。汽车和部件的对象及其连接,如图 3.21 所示。

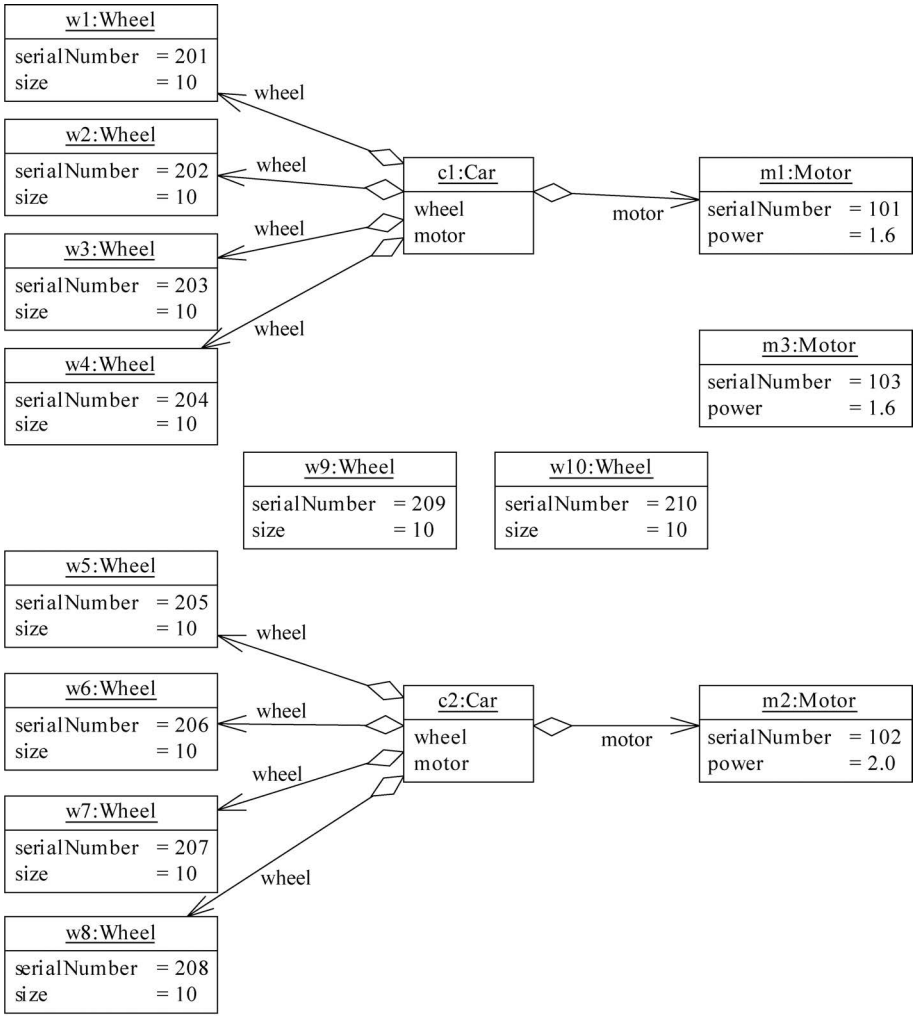

图 3.21 汽车和部件的对象及其连接

创建了类 Car 的两个对象 c1 和 c2 后，使用表达式 Wheel&w1 = c1.set(*wBase[8],1)更换汽车 c1 第 1 个位置上的车轮，其中，c1.set(*wBase[8],1)是一个函数调用，调用了类 Car 的 Wheel& Car::set(Wheel& depart, int position)成员函数，其通过引用方式返回一个无名的 Wheel 对象。为了访问返回的对象，为返回的对象定义了一个引用 w1。该表达式的计算顺序如图 3.22 所示。

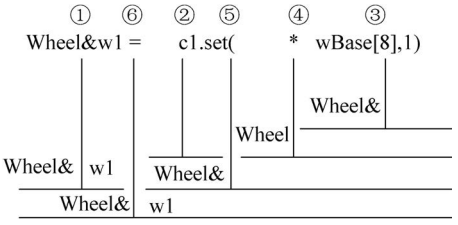

图 3.22　表达式 Wheel&w1=c1.set(*wBase[8],1)的计算顺序

如图 3.22 所示，表达式 Wheel&w1=c1.set(*wBase[8],1)中包含 6 个步骤，其中，第①步定义了 Wheel 对象的一个引用 w1；第⑤步，函数调用 c1.set(*wBase[8],1)返回了一个对象的引用，返回的引用没有名称；第⑥步，为返回的对象定义了一个引用 w1。

后面的语句 delete &w1 中，使用定义的引用 w1 删除函数调用 c1.set(*wBase[8],1)返回的无名对象。

例 3.10 程序的输出结果如下。

```
********** 创建发动机 **********
调用构造函数 Motor(101,1.6)
调用构造函数 Motor(102,2)
调用构造函数 Motor(103,1.6)
********** 创建车轮 **********
调用构造函数 Wheel(201,10)
调用构造函数 Wheel(202,10)
调用构造函数 Wheel(203,10)
调用构造函数 Wheel(204,10)
调用构造函数 Wheel(205,10)
调用构造函数 Wheel(206,10)
调用构造函数 Wheel(207,10)
调用构造函数 Wheel(208,10)
调用构造函数 Wheel(209,10)
调用构造函数 Wheel(210,10)
********** 创建汽车 **********
调用构造函数 Car(101,1.6)
汽车 发动机:101,1.6          车轮:201,10|202,10|203,10|204,10|
调用构造函数 Car(102,2)
汽车 发动机:102,2            车轮:205,10|206,10|207,10|208,10|
********** 更换并报废部件 **********
析构 Wheel:202
析构 Motor:101
********** 报废汽车 **********
析构 Wheel:201
析构 Wheel:209
析构 Wheel:203
析构 Wheel:204
析构 Motor:103
析构 Car:
析构 Wheel:205
```

```
析构 Wheel:206
析构 Wheel:207
析构 Wheel:208
析构 Motor:102
析构 Car:
********** 删除库存中没有使用的车轮 **********
析构 Wheel:210
```

可对照图 3.21 并结合例 3.10 程序的输出结果,分析对象的删除过程。

在“更换并报废部件”过程中,删除序列号为 202 的 Wheel 对象和序列号为 101 的 Motor 对象。在“报废汽车”过程中,Car 的构造函数中删除了连接到的 4 个 Wheel 对象和 1 个 Motor 对象,此时,总共删除了 9 个 Wheel 对象、3 个 Motor 对象和 2 个 Car 对象,还剩下序列号为 210 的 1 个 Wheel 对象没有删除。最后增加了一条语句删除它。这样就删除了程序中创建的所有对象。

示例中,CarApp.cpp 的代码最复杂,其次是 Car.cpp 中的代码,其他源文件中的代码都比较简单,需要仔细阅读 CarApp.cpp 和 Car.cpp 中的代码。

总之,实现聚合关系的一般方法是使用指针来表示聚合关系中的连接,通过代码来管理各个对象的生命周期。方法很简单,但需要程序员单独管理各个对象,工作量大,非常烦琐,特别是在需要管理的对象成百上千时,工作量和工作难度会超过人工能力的范围。

为了解决这个问题,常常会编写一些程序来专门管理内存中的对象,以减轻程序的负担。例如,在上述示例中,增加两个类 MotorBase 和 WheelBase 专门管理对象。类 MotorBase 负责创建和删除类 Motor 的所有对象,承担管理 Motor 对象的职责。类 MotorBase 负责创建和删除类 Wheel 的所有对象,承担管理 Wheel 对象的职责。类 Car 只负责使用这些对象,承担使用这些对象的职责,这三个类各司其职、分工合作,共同完成所需完成的任务。专门管理对象的类 MotorBase 和 WheelBase 如图 3.23 所示。

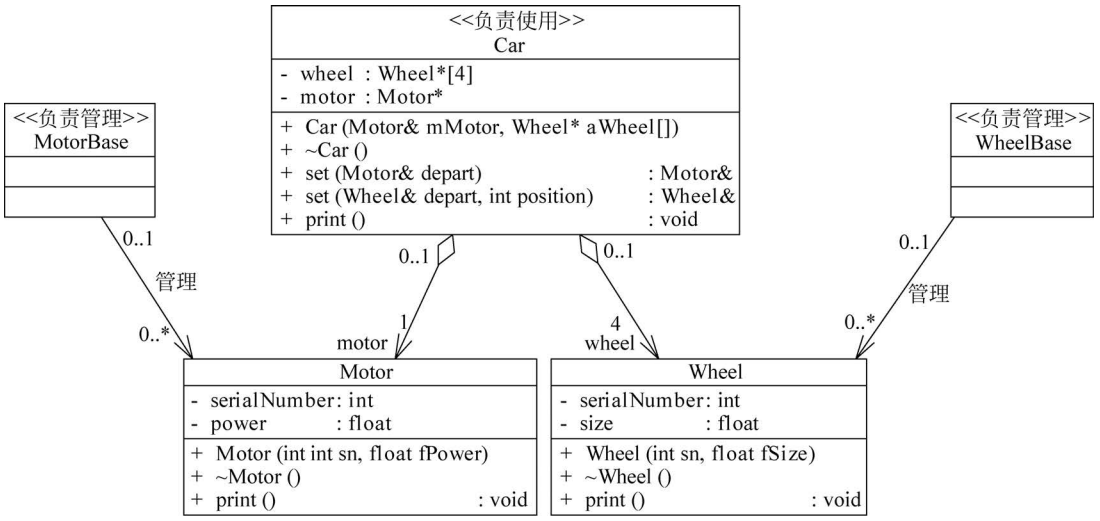

图 3.23 专门管理对象的类 MotorBase 和 WheelBase

面向对象程序设计中,管理程序中的对象是很重要的基础工作,在学习面向对象程序设计过程中需要掌握相关的技术,理解其原理。只有这样,才能理解系统提供的自动管理对象

功能，例如，理解 Java 语言中提供的自动清理对象功能。

3.4　深入理解类及其对象

类与对象是面向对象程序设计的基础，封装类并给其对象赋予相应的职责是面向对象程序设计的主要工作。下面从编程实现视角分析类中隐含的(组合)关联及其连接，讨论对象的内部结构，探讨类及其对象的本质。

例如，表示学生的类 Student，具有自己的属性 name 和 gpa，存在与学号之间的组合关联 studentID，其类图如图 3.24 所示。

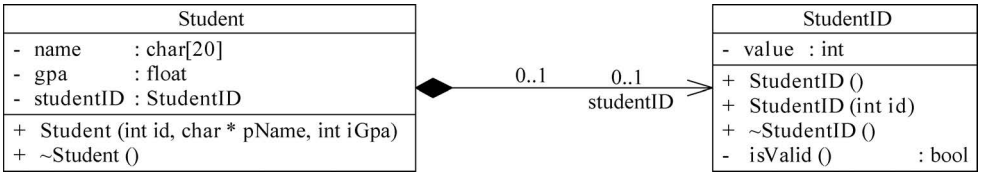

图 3.24　简化后的学生类

使用语句 Student s1(210101,"Randy",3.5)创建类 Student 的对象 s1，对象 s1 用于表示客观世界中的一个学生 Randy，可按照映射来理解对象 s1 与其属性值之间的对应关系。对象 s1 中的映射如图 3.25 所示。

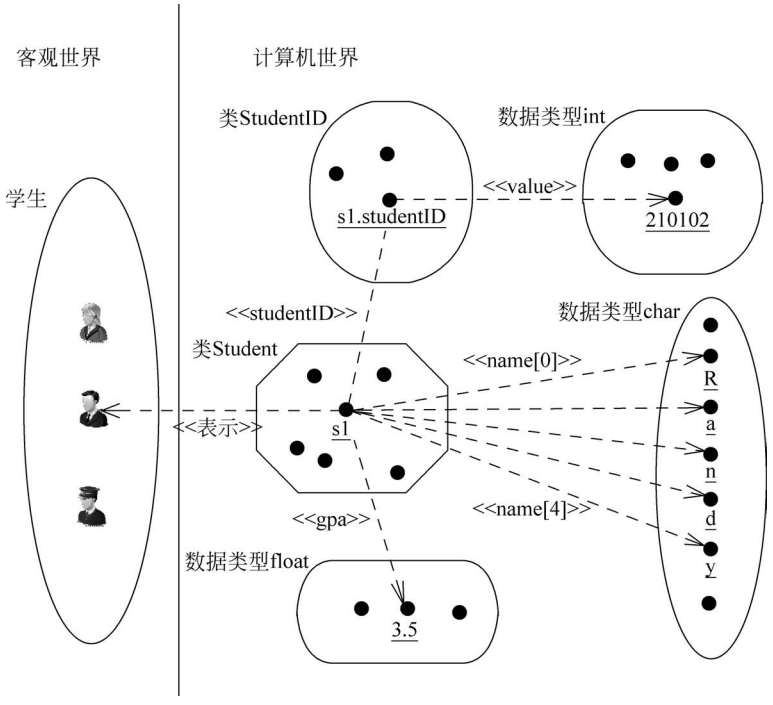

图 3.25　Student 对象 s1(210101,"Randy",3.5)中的映射

如图 3.25 所示，客观世界中的学生可视为一个集合，每一个学生都视为学生集合中的一个元素。同样，计算机世界中的类 Student 和类 StudentID 也可视为一个集合，它们的对

象就是集合中的元素，数据类型 char、int 和 float 也可视为一个集合，每一个值就是数据类型中的一个元素。

使用对象 s1 表示客观世界中的一个学生 Randy，可理解为，集合 Student 中的一个元素 s1 映射到学生集合中的元素 Randy。此外，将集合 Student 中元素 s1 映射到集合 float 中的元素 3.5，并用元素 3.5 表示学生 Randy 的绩点；将元素 s1 映射到集合 StudentID 中的元素 s1.studentID，再映射到集合 int 中的一个元素，并用元素 210101 表示学生 Randy 的学号。

元素 s1 映射到集合 char 中的多个元素，相对复杂一些，其中包含 5 个映射，使用一个数组 name 来存储这 5 个映射，并通过数组的下标映射到集合 char 中的 5 个元素。对象 s1 中的映射，如图 3.26 所示。

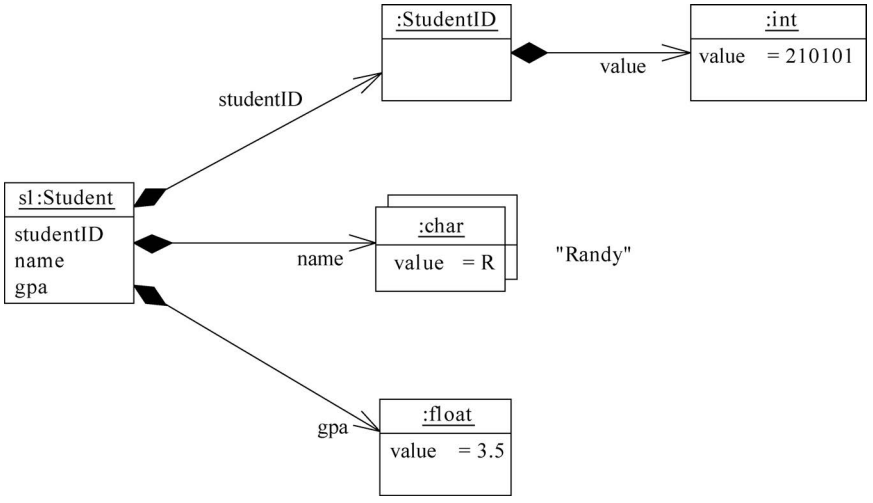

图 3.26　对象 s1 中的映射

图 3.26 中，使用组合连接描述了对象 s1 到其属性值的映射，箭头描述了映射的方向。语句 Student s2(210103,"Jenny",3.0)创建了对象 s2，对象 s2 中的映射，如图 3.27 所示。

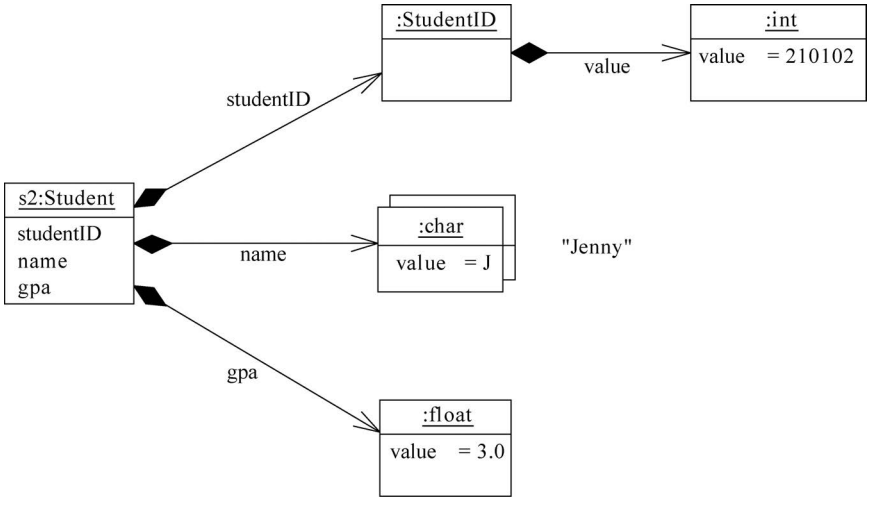

图 3.27　对象 s2 中的映射

比较分析如图 3.26 和图 3.27 所示的对象图，可将其中的对象抽象为类，将其中的连接抽象为组合关联，绘制出一张类图，这张类图描述了类 Student 中包含的映射关系。类 Student 中包含的映射关系如图 3.28 所示。

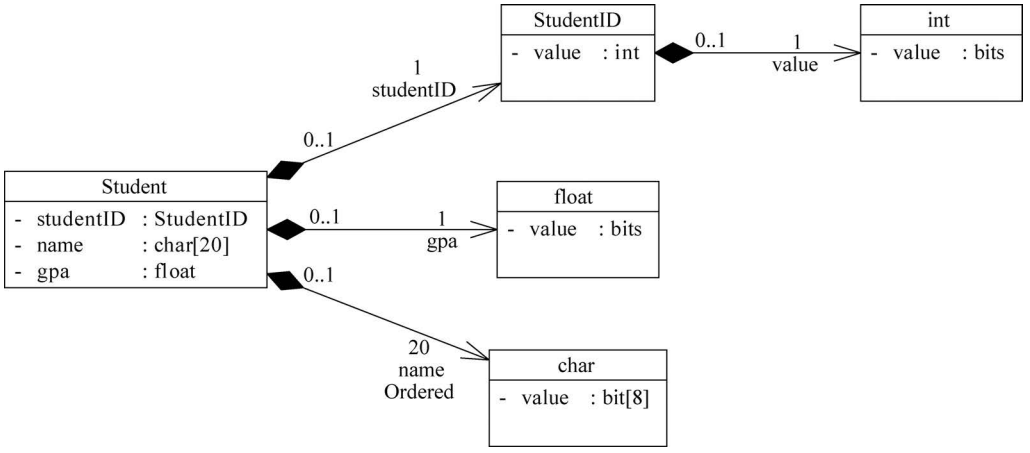

图 3.28　类 Student 中包含的映射关系

分析对比如图 3.24 和图 3.28 所示的两张类图会发现，类的每个属性都描述了一种映射关系，其对象中的每个属性值都描述了一种映射关系中的一个映射。

> 类的一个属性描述了一种映射关系，对象的一个属性值描述了其中的一个映射。

经过前面的讨论，要理解如图 3.24 所示的类 Student，需要将其细化为如图 3.28 所示的类图，并按照细化后的类图联想到如图 3.25 所示的实际场景，才能真正理解类 Student 的语义。

同样，也应该按照这个思路理解对象图。例如，在理解描述对象 s1 的对象图时，需要按照如图 3.26 所示的对象图来理解，并再联想到如图 3.25 所示的实际场景，才能真正理解对象 s1 的语义。描述对象 s1 的对象图，如图 3.29 所示。

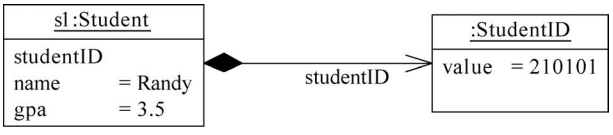

图 3.29　对象 s1 的对象图

> 计算机世界中的对象怎样表示客观世界中的事物？成员函数调用怎样表示客观事物的行为？这是两个基本问题，建议读者带着这两个基本问题重读本章前面的内容，相信会有自己的答案。

3.5　字符串

字符串是使用最多的数据，没有之一。字符串最终都是用字符数组来存储，下面先讨论字符数组的语义，再介绍封装字符串类的方法。

3.5.1 数组中的概念及其关系

一个数组 Array 包含多个元素 element，每个元素 element 都有相同的数据类型 Type，数组 Array 与数据类型 Type 之间属于组合关联，元素 element 就是这个组合关联的名称。数组 Array 还有一个数组名 arrayName，数组名 arrayName 是一个指针 ptr(地址)，指向数组 Array。

上述描述中，使用了数组 Array、数据类型 Type 和指针 ptr 3 个概念，并用元素 element 和数组名 arrayName 描述了这 3 个概念之间的关系。数组中的概念及其关系，如图 3.30 所示。

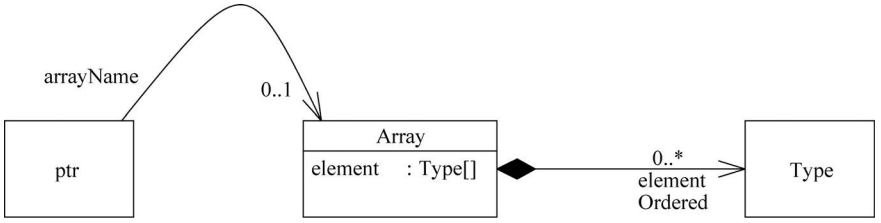

图 3.30 数组中的概念及其关系

希望按照如图 3.30 所示的关系，可通过指针 ptr 访问到数组，通过数组可访问到数组中元素，通过数组元素最终访问到元素值，即从一个指针 ptr 最终映射到一个元素值。

但有一个问题，数组与元素之间是一对多关系，一个数组不能映射到其中的一个元素。为了解决这个问题，规定了数组中的元素是有序的，并使用序号来确定访问哪个元素。这就是数组下标运算的由来。

3.5.2 字符数组的语义

3.3.2 节示例中介绍了类 Person，下面仍然以它为例继续讨论字符数组的语义。

如图 3.16 所示，类 Person 的属性 name 是一个字符数组，字符数组 name[20]作为一个整体存储在其对象的内存中。按照如图 3.30 所示的概念及其关系，可使用组合关联表示类 Person 到字符数组、字符数组到数据类型 char 之间的映射关系，细化属性 name 的语义。属性 name 隐含的组合关联如图 3.31 所示。

如图 3.31 所示，使用组合关联 element 将 20 个字符组合连接成字符数组 charArray，再使用组合关联 name 将字符数组 charArray 组合连接到 Person 对象，20 个字符最终成为 Person 对象的一部分。

在实际应用中，姓名不可能刚好都是 20 个字符。为了解决这个问题，提出了两种解决办法。第一种解决办法是，允许比规定的 20 个字符少，并增加一个字符串的结束标志'\0'，结束标志前面的元素才有对应字符，这样会浪费后面的内存。如图 3.16 所示类 Person 中，就是选择了这种解决办法。

第二种解决办法是，使用动态数组存储姓名中的字符。一个姓名中有多少个字符就分配多少个字节。这个办法能够充分利用内存，但也存在一个问题，编译器自动管理的内存必须是固定长度的，不能是动态变化的，因此，需要程序员编写代码来动态管理数组的内存。

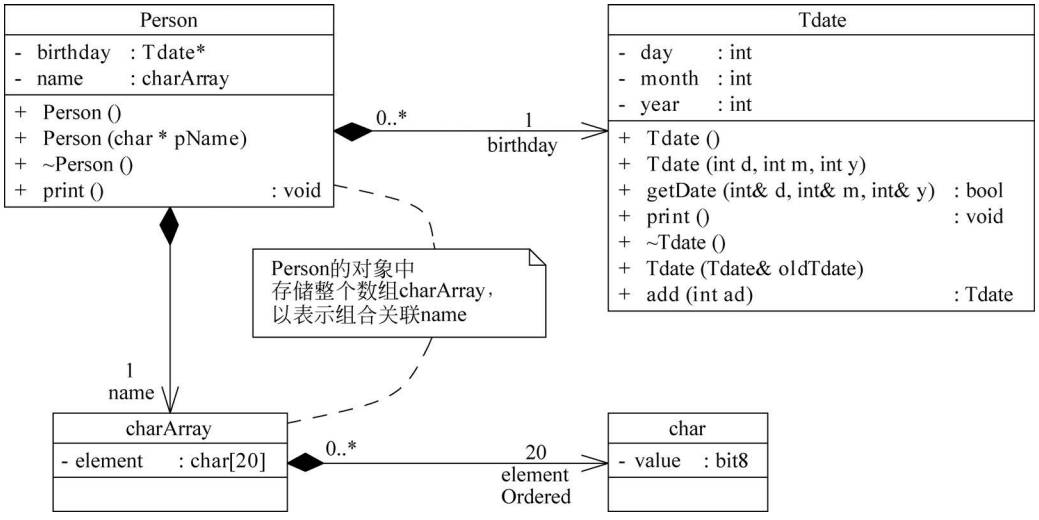

图 3.31　类 Person 的属性 name 隐含的组合关联

动态字符数组的指针是固定长度的，可以嵌套到 Person 对象的内存，并让系统自动管理。存储字符数组指针的类 Person 如图 3.32 所示。

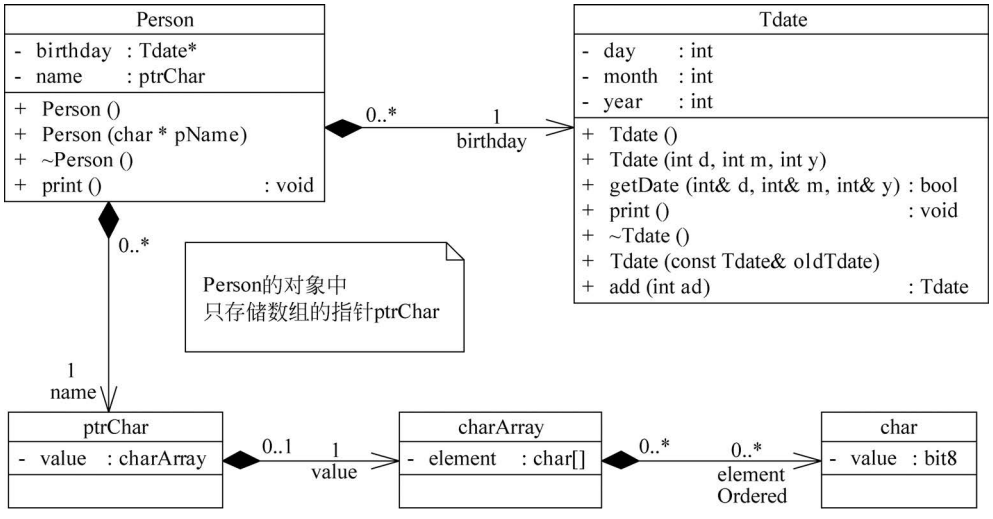

图 3.32　存储字符数组指针的类 Person

如图 3.32 所示的类图，清晰地表示出一个人到一个字符之间的映射关系。例如，姓名为 Randy 的 Person 对象 p 中，对象 p 通过属性 name 映射到指针 address，指针 address 映射到一个字符数组 A，字符数组 A 映射到其中的一个元素 A[i]，元素 A[i]再映射到“Randy”中的一个字符。总共需要经过 4 个映射，才能访问姓名中的一个字符。Person 对象 p 到字符的映射过程如图 3.33 所示。

在实际应用中，如果将属性 name 直接存储指向字符数组的指针，一般会省略图 3.32 中的类 ptrChar 和 charArray 及其映射关系，但从省略的类图中也要读出这层语义。

属性 name 中存储指向字符数组的指针时，系统不会自动管理指向的字符数组，而需要通过编程来同步 Person 对象和字符数组 charArray 的生命周期，实现组合关联的语义。具体编程方法可参考 3.3 节。

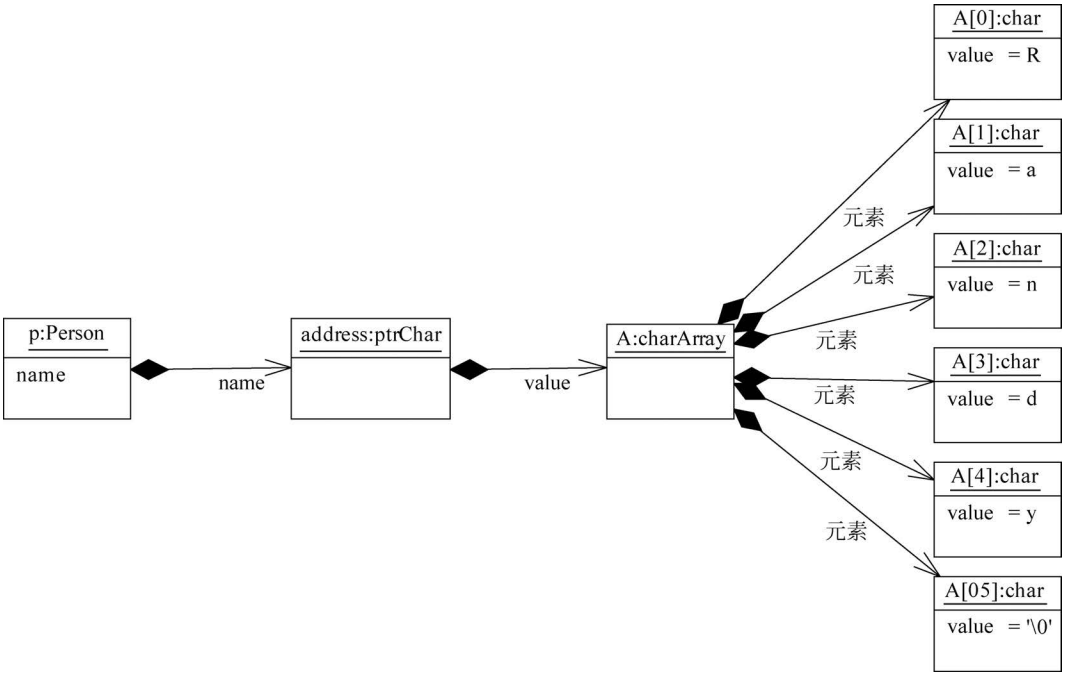

图 3.33 Person 对象 p 到字符的映射过程

3.5.3 自定义字符串类 myString

为了方便使用,可将字符数组封装为一个类 myString,并赋予类 myString 存储和管理字符串的职责。

可将如图 3.32 所示的类 ptrChar 组合关联到类 myString,用于存储字符数组的地址,并增加表示字符数组大小的属性 len。类 myString 及其组合关联如图 3.34 所示。

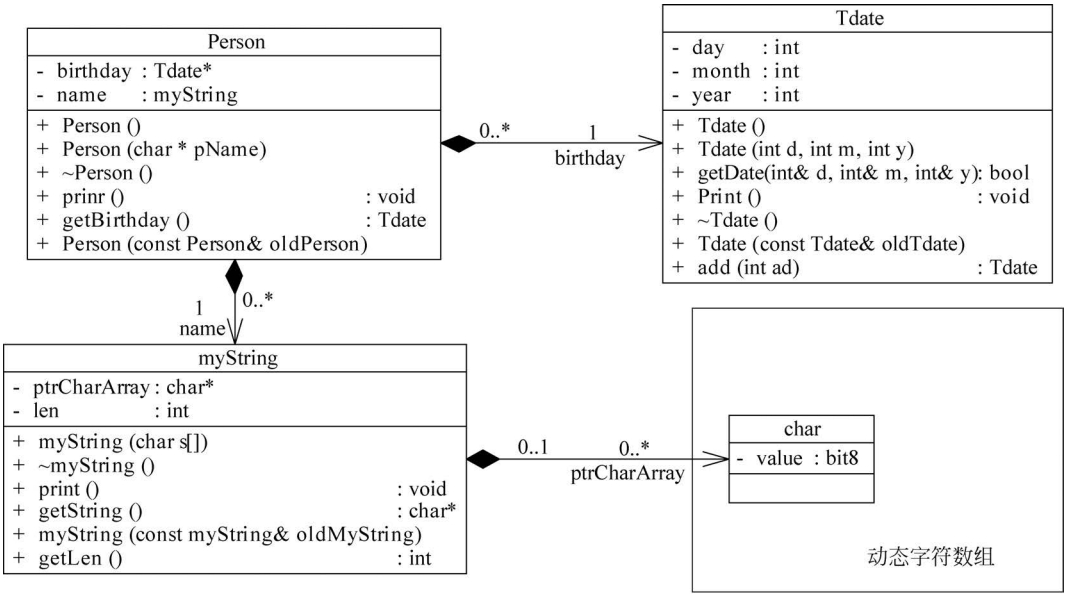

图 3.34 类 myString 及其组合关联

如图 3.34 所示，类 myString 中封装了两个属性，属性 ptrCharArray 用于存储动态字符数组的地址，属性 len 用于存储字符串的长度。

类 myString 中封装了 6 个成员函数，其中，myString(char s[])构造函数负责创建并初始化字符数组，～myString()析构函数负责删除字符数组，通过构造函数和析构函数同步类 myString 与字符数组的生命周期，实现组合关联 ptrCharArray 的语义。类 myString 的代码如例 3.11 所示。

【例 3.11】 字符串类 myString。

```
// myString .h
class myString
{
public:
    myString(char s[]);
    ～myString();
    void print()const;
    int getLen()const;
    char * getString()const;
    myString(const myString& oldMyString);
private:
    char * ptrCharArray;
    int len;
};
```

```
// myString.cpp
#include "myString.h"
#include <iostream>
using namespace std;

myString::myString(char s[]){
    len = strlen(s);
    ptrCharArray = new char[len + 1];
    strncpy(ptrCharArray, s, len + 1);
    ptrCharArray[len] = '\0';
}
myString::～myString(){
    delete ptrCharArray;
}
void myString::print()const{
    cout << ptrCharArray << endl;
}
int myString::getLen()const {
    return len;
}
const char * myString::getString()const{
    return ptrCharArray;
}
myString::myString(const myString& oldMyString){
    len = oldMyString.getLen();
    ptrCharArray = new char[len + 1];
    strncpy(ptrCharArray, oldMyString.getString(), len + 1);
}
```

类 myString 的代码中,getLen()成员函数返回字符串的长度,getString()成员函数返回字符串的指针,但这个指针指向的是 const char*,不允许通过这个指针修改字符数组中的字符,以保证安全。

类 Person 中,可以像使用基本数据类型一样使用类 myString,像变量一样使用类 myString 的对象。类 Person 中使用类 myString 的主要代码如例 3.12 所示。

【例 3.12】 类 Person 中使用类 myString。

```
// Person.cpp
#include "myString.h"
#include <iostream>
using namespace std;
class Person
{
public:
    Person(){}
    Person(myString n) :name(n.getString()){}
    void print(){
        cout << "Person:";
        name.print();
    }
    Person(const Person& oldPerson) :name(oldPerson.name) {}
private:
    myString name;   //表示到 myString 的组合关联
};
```

类 Person 的代码中,myString name 表示到 myString 的组合关联,代码 name(n.getString())维护这个组合关联。代码简洁,充分体现了类 myString 的优越性。

```
Person fn(Person p){
    return p;
}
myString fn(myString p){
    return p;
}

void main(){
    cout << "**** 创建 myString 对象 **** " << endl;
    myString s1("张三");
    s1.print();
    myString s2(s1);
    s2.print();

    cout << endl << "**** 传递 * myString 对象 **** " << endl;
    fn(s1).print();

    cout << endl << "**** 创建 Person 对象 **** " << endl;
    Person p1("李四");
    p1.print();
    Person p2(s1);
    p2.print();
```

```
    cout << endl << " **** 传递 * Person 对象 **** " << endl;
    fn(p1).print();

    cout << endl << " *** 堆中创建 * Person 对象 **** " << endl;
    Person * p3 = new Person("王五");
    p3 -> print();

    cout << " **** main 语句结束 **** " << endl;
}
```

例 3.12 程序的输出结果如下。

```
 **** 创建 myString 对象 ****
张三
张三

 **** 传递 * myString 对象 ****
张三

 **** 创建 Person 对象 ****
Person:李四
Person:张三

 **** 传递 * Person 对象 ****
Person:李四

 *** 堆中创建 * Person 对象 ****
Person:王五
 **** main 语句结束 ****
```

类 myString 封装了字符数组，编写主程序和类 Person 代码的程序员，不需要了解类 myString 的内部实现，只需将类 myString 当成数据类型来使用，这样明显降低了编程难度，提高了编程效率。

3.6　应用举例：链表

链表是常用的数据结构，常常用于动态管理创建的对象。例如，使用链表管理学生，其类图如图 3.35 所示。

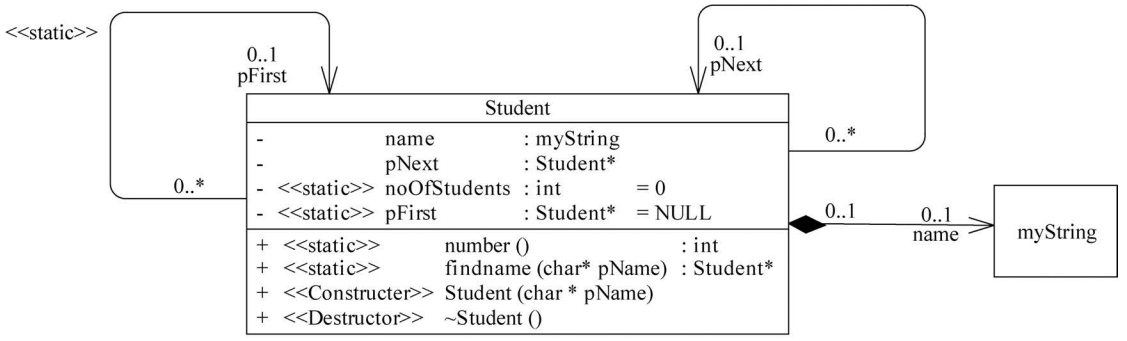

图 3.35　使用链表管理学生

在如图 3.35 所示的类图中,总共有 3 个关联。关联 pFirst 和 pNext 是自关联,即类 Student 关联到它自己,其连接是一个 Student 对象到另一个 Student 对象。关联 pFirst 是静态的,使用一个静态指针 static Student * pFirst 来表示,类 Student 的所有对象共享指针变量 pFirst,用于指向链表中第一个 Student 对象。使用一个指针 Student * pNext 表示关联 pNext,类 Student 的每个对象都有一个指针变量 pNext,用于指向链表中的下一个 Student 对象。

关联 name 是一个组合关联,其中的类 myString 是在 3.5 节中封装的,这里直接使用,用于存储学生的姓名。使用链表管理学生,示例代码如例 3.13 所示。

【例 3.13】 使用链表管理学生。

```
#include "myString.h"
#include "Student.h"

#include <iostream>
using namespace std;

class Student
{
public:
    static int number(void);
    static Student * findname(char * pName);
    Student(char * pName);
    ~Student();
private:
    myString name;                  //存储组合关联 name 中的连接
    Student * pNext;                //指向链表中的下一个 Student 对象
    static Student * pFirst;        //指向链表中的第一个 Student 对象
    static int noOfStudents;        //存储学生人数
};
```

```
int Student::noOfStudents = 0;
Student * Student::pFirst = NULL;

int Student::number(void){
    return noOfStudents;
}
Student * Student::findname(char * pName){
    for (Student * pS = pFirst; pS; pS = pS->pNext)
    if (strcmp(pS->name.getString(), pName) == 0)
        return pS;
    return NULL;
}
Student::Student(char * pName) :name(pName)
{
    cout << "插入:" << this->name.getString() << endl;
    pNext = pFirst;                 //每新建一个结点(对象),就将其挂在链首
    pFirst = this;
}
Student::~Student(){
    cout << "删除:" << this->name.getString() << endl;
```

```
    if (pFirst == this){          //如果要删除链首结点,则只要链首指针指向下一个
        pFirst = pNext;
        return;
    }
    for (Student * pS = pFirst; pS; pS = pS->pNext){
        if (pS->pNext == this){ //找到时,pS 指向当前结点的结点
            pS->pNext = pNext; //pNext 即 this->pNext
            return;
        }
    }
}
```

```
void main(){
    Student s1("Randy");
    new Student("Jenny");
    Student s2("Kinsey");
    cout << "查找 Jenny:" ;
    Student * pS1 = Student::findname("Jenny");
    if (pS1)
        cout << "ok." << endl;
    else
        cout << "no find." << endl;

    delete pS1;

    cout << "查找 Jenny:" ;
    Student * pS2 = Student::findname("Jenny");
    if (pS2)
        cout << "ok." << endl;
    else
        cout << "no find." << endl;
}
```

main()函数中,总共创建了 3 个 Student 对象,构成一个链表,其中,语句 new Student ("Jenny")在堆中创建了一个无名对象。3 个 Student 对象构成的链表如图 3.36 所示。

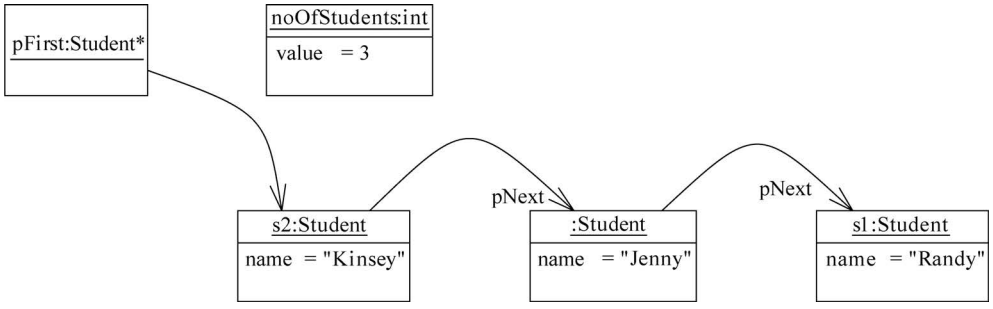

图 3.36　3 个 Student 对象构成的链表

语句 Student * pS2=Student::findname("Jenny")在链表中按照姓名查找,Student::findname("Jenny")返回链表中的无名对象。语句 delete pS1 删除找到的无名对象,删除对象后的链表如图 3.37 所示。

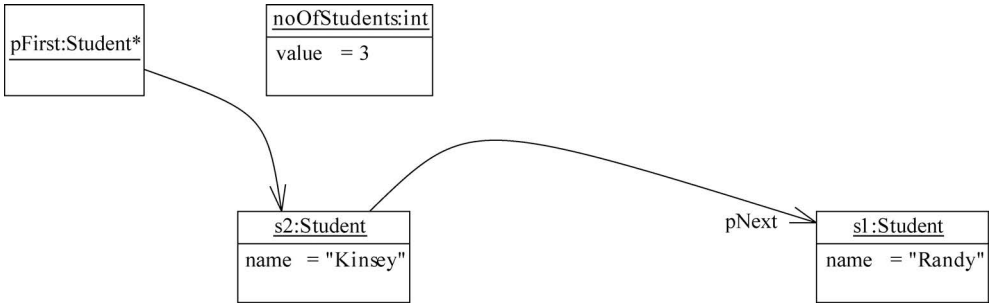

图 3.37 删除无名对象后的链表

例 3.13 程序的输出结果如下。

```
插入:Randy
插入:Jenny
插入:Kinsey
查找 Jenny:ok.
删除:Jenny
查找 Jenny:no find.
删除:Kinsey
删除:Randy
```

例 3.13 中的类 Student 中,没有定义拷贝构造函数,默认拷贝构造函数不会将创建的 Student 对象插入到链表,因此,如果将 Student 对象作为函数的参数或返回值,函数调用过程中的中间对象没有插入到链表中。

读者可以编写代码,测试函数调用过程中能否正确传递 Student 对象,评价 Student 对象参与计算的能力。实际上,这是一个比较复杂的问题,需要平衡多方面的因素,值得深入分析。

小结

本章从客观事物之间的关系出发,主要学习了关联及连接的概念以及描述事物之间关系的方法,重点学习了使用组合关联描述客观事物的内部结构,以及一般关联、组合关联和聚合关联的编程实现技术和方法,最后学习了字符串和链表两个应用案例。

学习了关联及连接的概念,举例说明了使用指针和指针数组实现多对一、多对多关联的编程方法和技术。希望读者能够理解关联及连接的概念,掌握使用关联的方法和编程技术。

学习了组合关联和聚合关联的概念,举例说明了使用组合关联描述事物内部构成的方法,以及组合关联和聚合关联的表示方式和编程技术。希望读者能够理解组合关联和聚合关联的概念,掌握其使用方法和编程技术。

最后学习了字符串和链表两个应用案例,希望读者能够从关联角度理解字符串和链表等复杂数据结构,了解封装复杂数据结构的思路,掌握编程实现的方法。

练习

1. 饭厅中拟放置 1 张饭桌和 8 个凳子,并抽象出了饭厅、饭桌和凳子 3 个类及其关系,

如图 3.38 所示。请使用集合和映射描述其中的类及其关系。

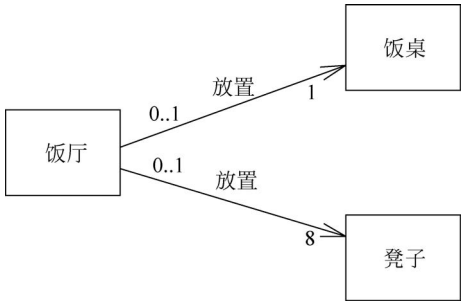

图 3.38　饭厅、饭桌和凳子及其关系

2. 结合实际情况完善图 3.38 中的类图，增加描述饭厅、饭桌和凳子特性的属性，给每个类赋予适当的职责并增加相应的成员函数，最后编程实现。

3. 3.2 节中使用指针表示关联，为了提高代码的安全性，请使用引用表示关联，并改写其中的例程代码。

4. 按照如图 3.39 所示的类图编写程序。要求使用动态数组存储一个人的姓名，类 Person 的对象负责管理其中存储的姓名，并编程实现。

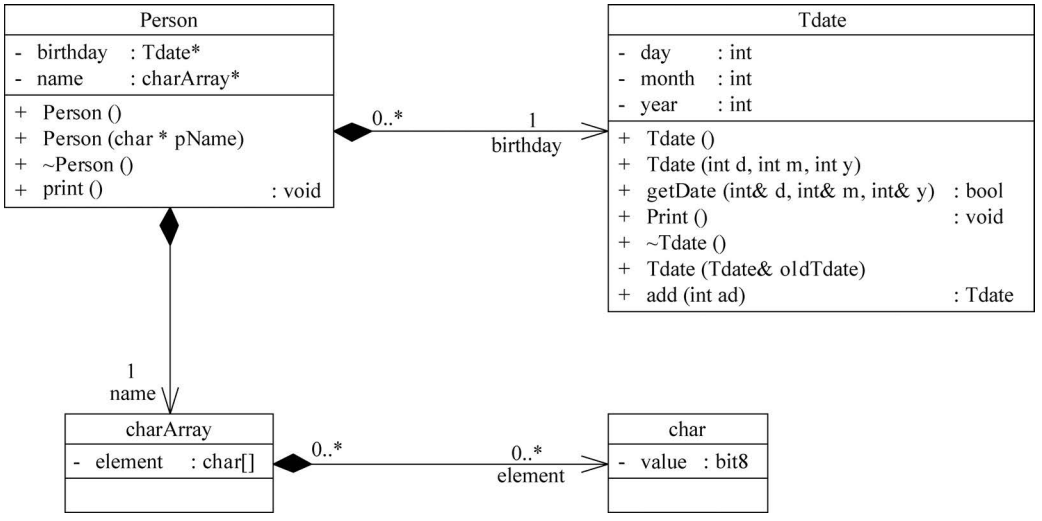

图 3.39　使用动态数组存储姓名

其中，char 为基本数据类型，charArray 为动态字符数组，代码中可以不声明类 charArray。

5. 例 3.12 中创建了类 Person 的对象 p1 和 p2，请使用对象图描述对象 p1 和 p2 的结构及其数据，并使用映射表示对象 p1 与其数据之间的关系。

6. 例 3.12 中使用表达式 fn(p1).print()检测类 Person 对象参与计算的能力，请使用时序图描述执行表达式 fn(p1).print()的过程。

7. 例 3.13 中使用链表管理类 Student 的对象，请使用时序图描述程序的执行过程。

8. 每学期选修课程时需要先认真阅读本专业的培养方案，分析需要学习的课程以及相关规定，然后再根据自己的具体情况从开设的课程中选择适当的课程，以保证毕业时能够修

完规定的课程,取得要求的学分。选课的一般步骤为:①根据所学的专业找到培养方案,根据自己的情况从本专业的培养方案中明确所执行的培养方案,确定需要学习的所有课程;②根据学习课程的进度从下学期开设课程中选择修读的课程。选课中涉及的专业、培养方案、课程等主要因素及其关系如图 3.40 所示。

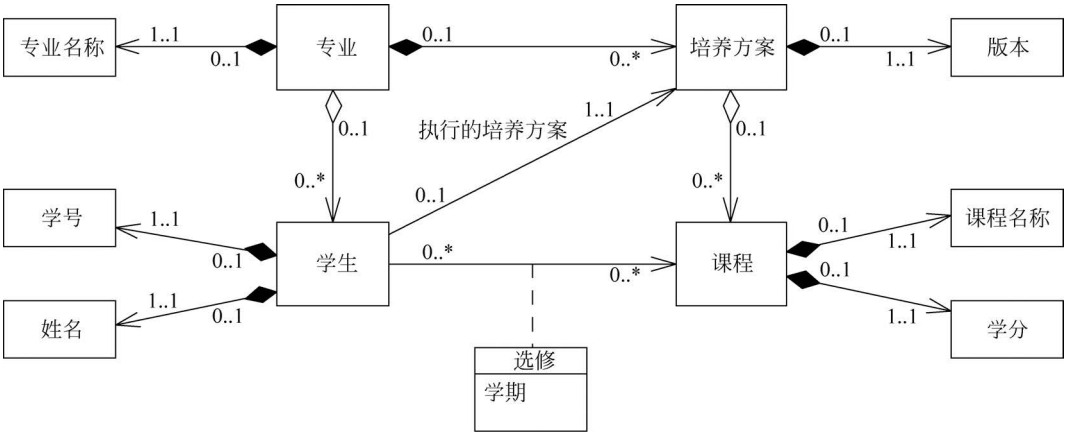

图 3.40　选课中涉及的主要因素及其关系

(1) 按照集合和映射等数学思维分析所在学校的本专业培养方案,修改完善如图 3.40 所示的类图,然后选择关联的表示方式以及属性的数据类型,最终设计出包含类、关联以及属性的类图,以描述选课场景。

(2) 分析所在学校的选课系统,使用时序图分解其中的核心功能,确定每个类的主要职责,最终设计出成员函数。

(3) 按照前面的设计编写程序,并调试通过。

为了便于理解,在设计时可采用中文命名类及属性和方法,在编写代码前再转换为英文。