第5章 设计与实现

在计算机发展的早期，计算机主要解决客观世界中的计算问题，并以计算为出发点产生了结构化程序设计思想。随着计算机的普及，计算机应用开始涉及生活和工作中的方方面面，需要解决客观世界中各种各样的问题，这就促使编程的出发点从计算转移到对客观事物的处理，从而产生了面向对象程序设计思想。

面向对象程序设计思想解决的核心问题有两个：一是怎样抽象和描述客观世界中的事物？二是怎样使用计算描述事物的变迁？

本章通过两个示例来说明解决上述两个核心问题的基本方法。

5.1 Josephus 游戏

Josephus 游戏是从小孩游戏中抽象出的一个问题，是学习面向对象设计和编程实现方法的经典示例。

Josephus 游戏：一群小孩围成一圈，任意假定一个数 m，从第一个小孩起，顺时针方向数数，每数到第 m 个小孩时，该小孩便离开。小孩不断离开，圈子不断缩小。最后剩下的一个小孩便是胜利者。


视频讲解

5.1.1 分析设计

面向对象程序设计中，一般从选择典型场景开始，先分析典型场景中的事物及其关系，抽象出对象及其连接，并给对象赋予相应的职责，然后再设计出类及其关联，最后选择适当的技术编码实现。

1. 发现对象及其连接

这一步的主要工作包括选择典型游戏场景、描述典型游戏场景和标识其中的事物及关系。

(1) 选择典型游戏场景。与实际应用相比，Josephus 游戏只有小孩数和间隔数两个变化因素，已经非常简单了，但仍然不可穷尽，因此，需要选择具有代表性的一场游戏作为典型场景，分析每个小孩在游戏中承担的职责和所做的事情，发现涉及的事物及其关系。

选择 Josephus 游戏的典型场景时，可选择 10 以内的最大素数 7 作为小孩数，除 1 外的最小素数 3 作为间隔数。这样选择，不仅能反映玩游戏的共同特性，具有很好的代表性，而且玩游戏的工作量也比较小。

(2) 描述典型游戏场景。选择典型场景后，重复玩这个游戏，描述 7 个小孩玩游戏的场

景及过程。7 个小孩手牵手围成一个圈，从第 1 个开始数，数到第 3 个时，第 3 个小孩离开，再继续数，数到 3 的小孩离开，直到只剩下一个小孩时，这个小孩就是获胜者。

描述 7 个小孩玩游戏的场景及过程时，应尽量直观简洁，便于理解。使用图来描述玩游戏中的事物及其关系，并在图上推演游戏的玩耍过程，是常用的方法。

按照游戏的玩耍过程，先在图中画 7 个小孩，然后用线条将小孩连接起来，构成一个圆圈，按照顺时针方向在线条上加箭头，描述游戏开始时的情况。

玩耍的是一场游戏，图中画一个矩形来表示；小孩围成一个圆圈，画一个圆圈来表示；游戏中涉及小孩数和间隔数，画两个矩形来表示；有一个获胜者，画一个小孩来表示，并用线条连接到表示一场游戏的矩形。圆圈是游戏中围成的，也用线条将圆圈连接到表示一场游戏的矩形。

(3) 标识游戏场景中的事物及其关系。描述典型游戏场景时，选择什么图标不是关键，只需形象直观、容易理解，但重要的是，准确标识游戏场景中的事物及关系。标识事物及其关系时，往往涉及语言知识和数学知识，一般要综合运用语言和数学中的表示方法来标识事物及其关系。

Josephus 游戏中，可用编号来标识小孩，用中文词组来标识游戏、圆圈等客观事物，用数学符号或英文单词来标识小孩数、间隔数等与数量有关的事物。事物的标识，即名称，力求简洁准确，方便理解。

标识游戏场景中的事物后，就可以在图上玩游戏了。玩耍游戏的过程中会发现，开始位置和数到的小孩位置频繁出现，说明其很重要，分别命名为开始位置和当前位置。开始位置和当前位置表示小孩在圆圈中的位置，用带箭头线条将位置与圆圈连接起来。

上述静态和动态分析过程中，先发现并标识游戏场景中的事物，然后表示出事物之间的关系。典型游戏场景中的事物及其关系如图 5.1 所示。

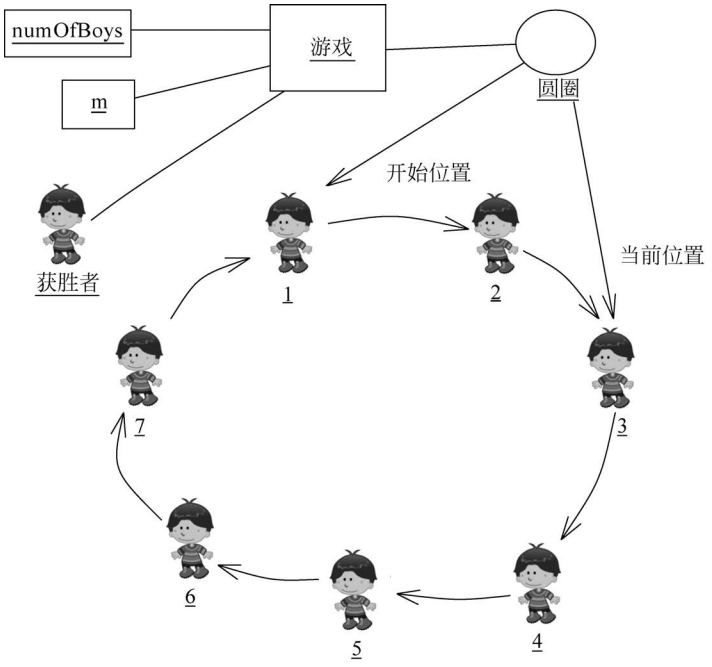

图 5.1　典型游戏场景中的事物及其关系

按照面向对象程序设计思想,将图 5.1 中的图标视为对象,将对象之间的连线视为连接,图 5.1 形象地表示了 Josephus 游戏中涉及的对象和连接。

前面主要针对所选择的游戏场景,找出了 Josephus 游戏中涉及的对象及其连接,这些对象和连接是后面工作的基础,非常重要。因此,需要评估玩游戏场景的代表性,如果代表性不好,则需要重新选择典型场景,并重复前面的工作。评估的基本方法是,选择自己觉得有代表性的几种场景,多玩几次游戏,就能有直观的判断。

2. 合并对象并划分职责

找出了对象及其连接后,再将一些对象合并到另外的对象,以减少需要管理的对象数量。

Josephus 游戏中,小孩数、间隔数和获胜者都从属于一场游戏,因此,将对象“小孩数”和“间隔数”组合到对象“游戏”,将“获胜者”聚合到对象“游戏”,并赋予对象“游戏”管理这 3 个对象的职责。

每场游戏都要构成一个圆圈,因而将对象“圆圈”聚合到“游戏”,赋予“游戏”管理“圆圈”的职责。

事物之间的其他联系,都视为一般关联的连接,按照连线的箭头方向划分管理职责。例如,每个小孩自己负责管理自己,至少要记住自己的下一个小孩,并承担加入圆圈和离开圆圈等职责。对象之间的组合连接如图 5.2 所示。

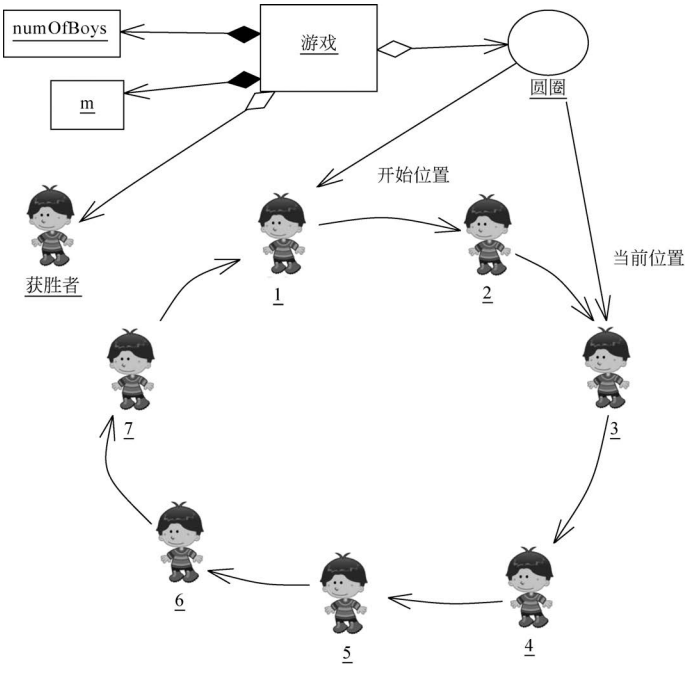

图 5.2 对象之间的组合连接

如图 5.2 所示,对象“游戏”拥有了对象“小孩数”“间隔数”和“获胜者”,到对象“圆圈”有一个聚合连接。对象“圆圈”到“小孩”有两个一般连接,多个“小孩”通过一般连接构成一个圆圈。

3. 抽象类及关联

前面将一些对象组合到另外的对象，减少了对象的数量，但仍然觉得太多，再对它们进行分类，并按照类别来管理游戏中的对象。

在如图 5.2 所示的对象中，主要存在“小孩”“游戏”和“圆圈”3 类别，可将这 3 个类别视为 3 个类“Boy”“Jose”和“Ring”，并在图中标出每个对象的类别。也对连接进行分类，抽象出 4 类连接，并将连接的类别标注在中间。标注类别的对象及连接如图 5.3 所示。

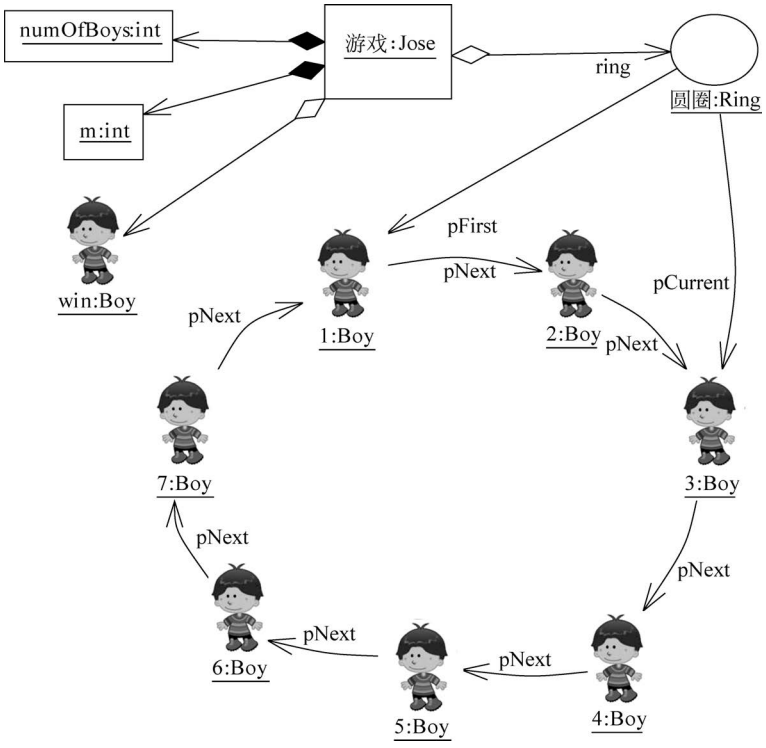

图 5.3　标注类别的对象及连接

实际上，图 5.3 中已经抽象出了所需的类及其关联，使用类图很容易描述出来。Josephus 游戏中的类及其关联如图 5.4 所示。

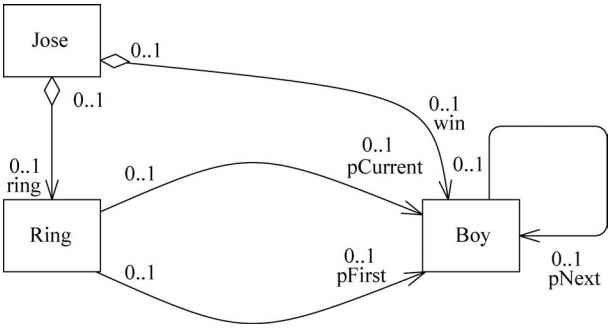

图 5.4　Josephus 游戏中的类及其关联

图 5.4 中标注了每个关联的重数，对比分析图 5.3 中每类连接的对应关系类型，就很容易理解关联的重数。

4. 发现类的属性

找出了类及其关联后,再寻找类的属性,一般分为 3 个步骤。每个类的属性如图 5.5 所示。

第一步,按照如图 5.3 所示的组合连接,将 numOfBoys 和 m 作为类 Jose 的属性。

第二步,将关联对端的名称作为属性名,用于表示类之间的关联。这时,只列出表示关联的属性,以后再考虑怎样实现。

第三步,针对每个类,增加与场景有关的属性。例如,可为小孩增加姓名、性别等表示基本信息的属性,但为了简单,只增加了小孩的编号 code。

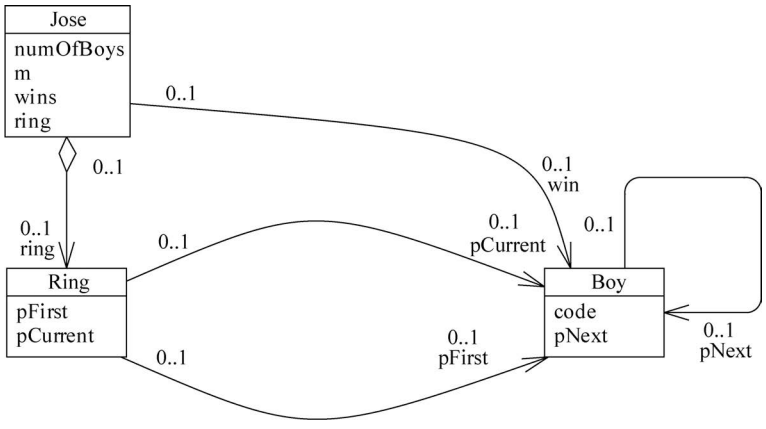

图 5.5 类中增加的属性

5. 发现类的方法

如果将一个对象拥有的成员变量或成员对象视为对象的“财富”,那么,这个对象就应该同时具有管理这些“财富”的义务和使用好这些“财富”的责任。类的方法就具体承担这些义务和责任。

为了找到类的方法,又需要回到玩游戏的场景,但这次玩游戏的目的与前面不同,需要你带着对象一起玩,并告诉每个对象需要做哪些事情,怎样协作,最终找到每类对象的共同行为。

对象的智商太低,带着他们一起玩游戏是一件困难的事,因此,可以分成两件事来完成,先告诉他们要做什么事情,然后再告诉他们怎样协作。

第一件事,告诉对象们,分为游戏准备、游戏开始和宣布获胜者 3 个阶段,你来宣布获胜者,其他两阶段的事他们自己做。

告诉对象们,Jose 类的对象全面负责游戏准备,Ring 类的对象协助 Jose 类的对象管理圆圈,Boy 类的对象自己围成圆圈。

游戏开始阶段,Jose 类的对象全面负责,Ring 类的对象协助完成“数数”,Boy 类的对象负责提供下一个小孩,并自己离开圆圈。Ring 类的对象任务比较复杂,将“往下数 n 个”从“数数”中独立出来,这样简单点。对象的职责与协作如图 5.6 所示。

做完第一件事后,再做第二件事,告诉你的对象们怎么协作。因对象智商太低,只能由你站在每个对象的角度,帮他们梳理出完成任务所需要的数据,并明确告诉每个对象,在做一件事时,给他们哪些数据,应返回哪些结果。

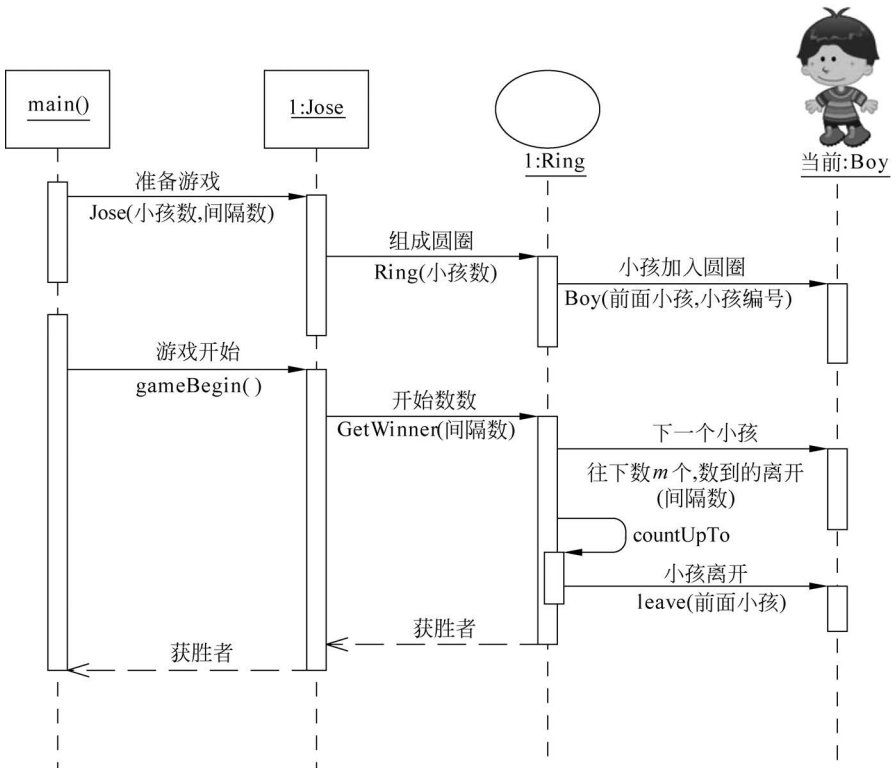

图 5.6　对象的职责与协作

实际上,从计算视角,将每件事视为一个函数,确定每个函数的输入和输出,并保证从输入能够得到输出。这部分使用的是结构化程序设计中的技术。

如图 5.6 所示的信息流中,上面的文字表示的是要做的事情,下面的部分是函数的原型。现在的函数原型还不完善,可后面再进行细化。

如图 5.6 所示,明确了各类对象的职责,以及协作的约定(函数原型),其中的职责都是该类对象的共同职责,也要遵守协作的约定,因此,可以将协作的约定作为类的成员函数原型,将职责作为成员函数的功能,这样就为每个类设计出成员函数。有些成员函数原型中的数据类型还不能确定,可先标注出来,后面再解决。类的属性和成员函数如图 5.7 所示。

图 5.7 描述拟开发程序的静态结构,图 5.6 描述了程序的动态交互过程,这两张图分别从静态和动态两个角度描述了拟开发的程序,但还不是真正的程序,因此,常常将这两张图描述的程序称为程序的模型,更一般地,称为软件模型。

5.1.2　编码实现


视频讲解

前面建立的软件模型,没有涉及具体的编程实现技术,甚至没有考虑实现的程序设计语言,也就是说,前面建立的软件模型与具体实现技术和实现平台无关,可以选用任何适当的技术来实现,使用任何计算语言来编码。

1. 选择实现技术

选择链表来存储“圆圈”,选用 C++编程。当然,也可以选择数组等其他方法来存储“圆圈”,也可以使用其他程序设计语言来编码。

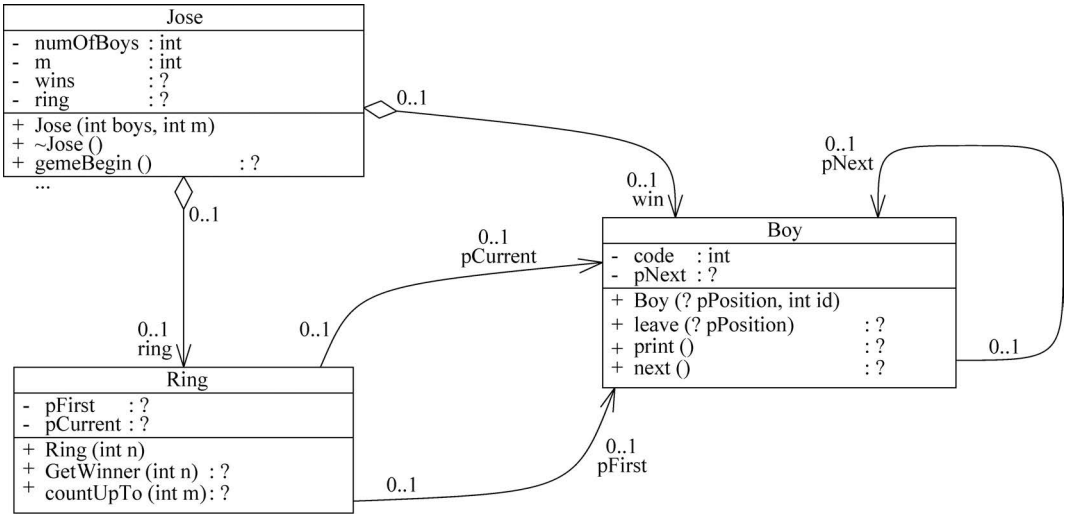

图 5.7 类的属性和成员函数

选择了实现技术和程序设计语言后，又要回到玩游戏的典型场景，在链表上继续玩游戏。典型场景中的对象及连接如图 5.8 所示。

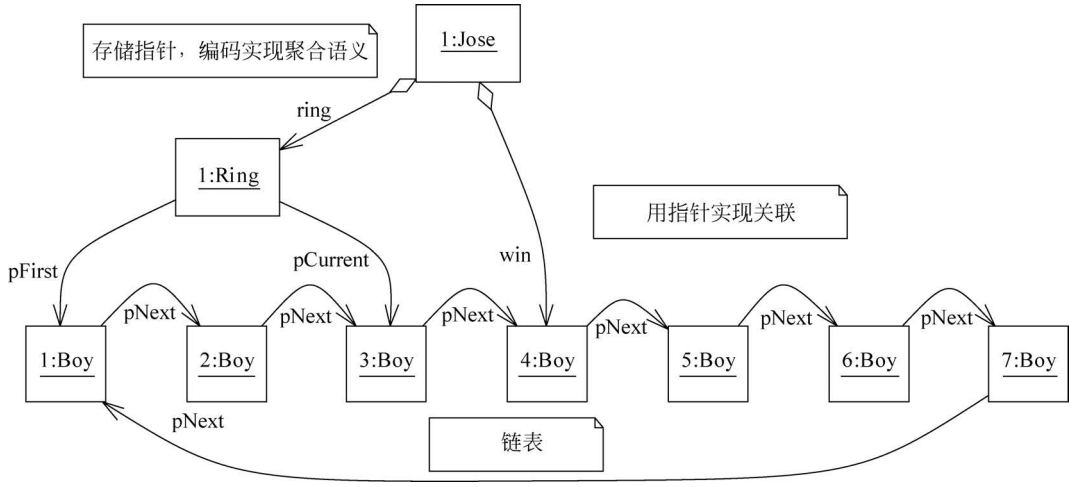

图 5.8 典型场景中的对象及连接

选择链表来存储“圆圈”后，就确定了关联 pNext、pFirst 和 pCurrent 的具体实现方法，也可以使用指针来表示两个聚合关联 ring 和 win，并通过编码来实现聚合关联的语义。

最后，按照 C++语言的要求，为每个类的属性选择数据类型，然后再给函数的参数和返回值指定相应的数据类型，后面就可以使用 C++语言编写代码。具有 C++语言和链表特性的类如图 5.9 所示。

如图 5.9 所示，选择指针来表示关联中的连接，将类中相应的属性设置为指针类型。另外，也需要评价各类对象参与计算的能力，并设置了相应的构造函数和析构函数。

建立程序的模型后，编码就比较简单了。编码的主要步骤是：①按照描述模型的类图声明各个类；②按照描述模型的时序图编写各成员函数的代码；③调试通过。

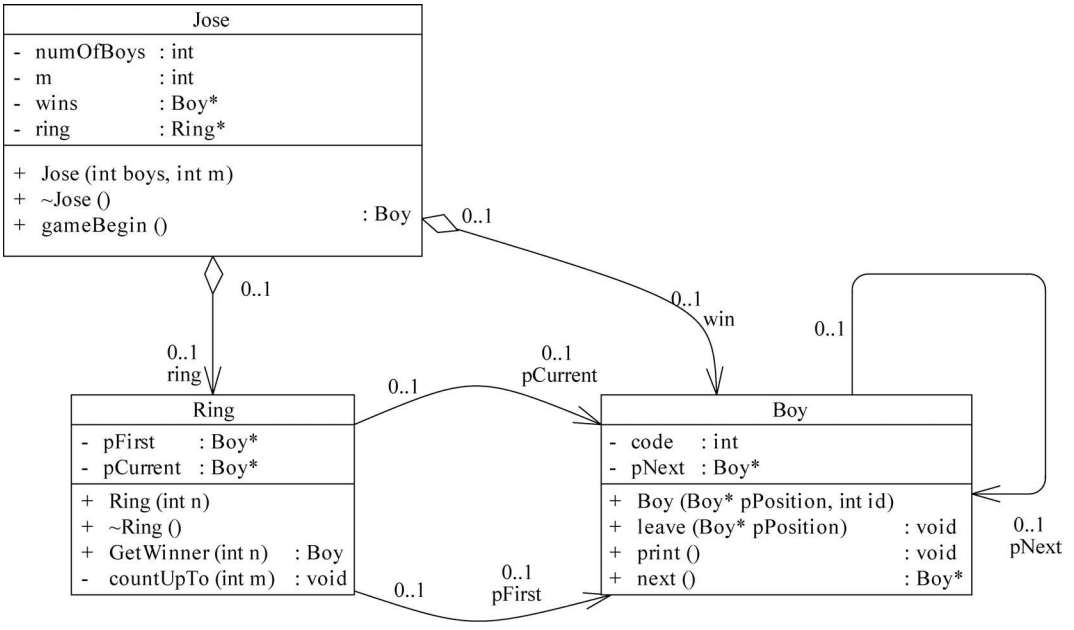

图 5.9　具有 C++语言和链表特性的类图

2. 编程实现类 Boy

按照类之间的依赖关系编写代码，能提高编程效率。如图 5.9 所示的箭头刚好表示了类之间的依赖关系，因此，先编写类 Boy 的代码。

按照如图 5.9 所示的类 Boy，编写类 Boy 的声明代码，按照如图 5.6 所示的交互过程，编写每个成员函数的实现代码，承担相应的职责，实现成员函数的功能。

如图 5.6 所示的交互过程是整个程序的交互过程，可将与类 Boy 相关的部分提取出来，并进一步细化。与类 Boy 相关的交互过程如图 5.10 所示。

编写代码时会发现，在如图 5.6 所示的交互过程中，没有明确哪个对象创建和删除 Boy 对象，因此，如图 5.10 所示的交互过程，将创建和删除 Boy 对象的职责赋予了 Ring 对象。需要注意的是，给 Ring 对象增加的职责，涉及与 Ring 对象的协作，必须通知 Ring 对象，即修改图 5.10 中的交互过程，以便按照新的交互过程编写类 Ring 的代码。

细化了与类 Boy 相关的交互过程后，就可以按照这个交互过程编写 next()、leave(Boy * pPosition)成员函数，以及 Boy(Boy * pPosition, int id)构造函数的代码。因删除对象时，不需要做特别的事，直接使用默认析构函数。类 Boy 的实现代码如例 5.1 所示。

【例 5.1】 类 Boy 的实现代码。

```
//Boy.h
class Boy
{
public:
    Boy(Boy * pPosition, int id);
    void leave(Boy * pPosition);
    void print();
    Boy * next();
```

```
protected:
    int code;
    Boy * pNext;
};
```

```
#include "Boy.h"
#include <iostream>
using namespace std;

Boy::Boy(Boy * pPosition, int id){
    code = id;
    if (!pPosition){
        this->pNext = this;                    //只有一个小孩时,自己指向自己
    }
    else{
        this->pNext = pPosition->pNext;        //插入到小孩 * pPosition 的后面
        pPosition->pNext = this;               //与上一条不能交换
    }
}
void Boy::leave(Boy * pPosition){
    pPosition->pNext = this->pNext;
    cout << "离开:" << code << endl;
}
void Boy::print(){
    cout << "Id:" << code;
}
Boy * Boy::next(){
    return pNext;
}
```

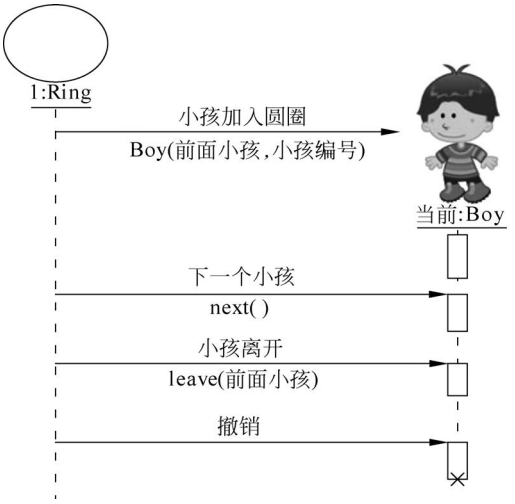

图 5.10　与类 Boy 相关的交互过程

Boy(Boy * pPosition,int id)构造函数的功能是加入圆圈(循环链表),参数 Boy * pPosition 表示插入的位置,插入到小孩 * pPosition 的后面。当增加第 4 个小孩时,

pPosition 指向第 3 个小孩，this 指向自己，应先将 pPosition-> pNext 中的地址（代表第 1 个小孩的对象的地址）赋值给自己的 pNext，然后再将自己的地址 this 赋值给第 3 个小孩的 pNext，顺序不能交换。插入第 4 个小孩前的情况如图 5.11 所示。

当圆圈中没有小孩时，插入的小孩是第一个小孩，插入后，该小孩的下一个小孩还是他自己，需要特别处理。约定 Ring 对象通过参数 pPosition 传递一个空指针 NULL，Boy 对象接收到一个空指针时，只需将自己的地址 this 赋值给自己的 pNext。插入第 1 个小孩后的情况如图 5.12 所示。

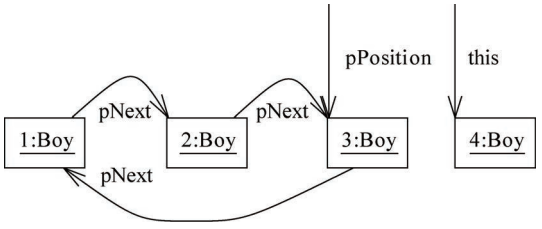

图 5.11　插入第 4 个小孩前的情况

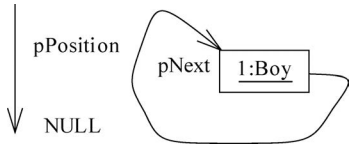

图 5.12　插入第 1 个小孩后的情况

leave(Boy * pPosition)成员函数的功能是指针 pPosition 指向的小孩从链表中离开，通过表达式 pPosition-> pNext = this-> pNext 来实现，比较简单，但也要对照图来理解。

3. 编程实现类 Ring

同样，按照如图 5.9 所示的类 Ring 编写类 Ring 的声明代码，按照如图 5.6 所示的交互过程编写类 Ring 成员函数的实现代码。可将与类 Boy 相关的部分提取出来，其中增加了创建和删除 Boy 对象的职责。与类 Ring 相关的交互过程如图 5.13 所示。

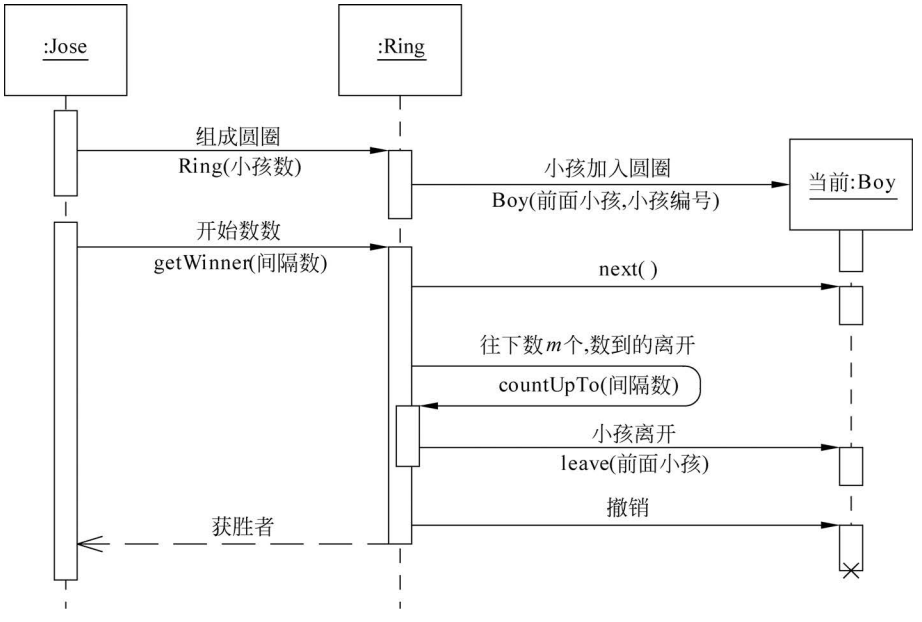

图 5.13　与类 Ring 相关的交互过程

按照如图 5.13 所示的 3 个成员函数及其职责，并对照如图 5.8 所示的对象及连接，逐个编写类 Ring 的成员函数代码。类 Ring 的实现代码如例 5.2 所示。

【例 5.2】 类 Ring 的实现代码。

```
//Ring.h
#include "Boy.h"
class Ring
{
public:
    Ring();
    Ring(int n);
    ~Ring();
    Boy getWinner(int m);
private:
    void countUpTo(int m);
    Boy * pFirst;
    Boy * pCurrent;
};
```

```
//Ring.cpp
#include "Ring.h"
#include "Boy.h"

Ring::Ring(int n){
    pFirst = pCurrent = new Boy(NULL, 1);
    Boy * pB = pFirst;
    for (int i = 2; i <= n; i++){
        pB = new Boy(pB, i);
    }
}
Boy Ring::getWinner(int m){
    //数小孩
    while (pCurrent != pCurrent->next()){
        countUpTo(m);           //往下数 m 个小孩,数到的小孩离开
    }
    //返回获胜者
    Boy win( * pCurrent);       //另外创建(复制)一个 Boy 对象
    delete pCurrent;
    return win;
}
void Ring::countUpTo(int m){
    //往下数 m 个小孩
    Boy * pLast;
    for (int i = m; i > 1; i-- ){
        pLast = pCurrent;
        pCurrent = pCurrent->next();
    }
    //数到的小孩离开
    pCurrent->leave(pLast);     //当前从圆圈中离开, pLast 指向前面的小孩
    delete pCurrent;            //删除当前小孩

    pCurrent = pLast->next(); //当前小孩是上一个小孩的下一个
   }
Ring::~Ring(){
}
```

Ring(int n)构造函数的功能是创建 n 个 Boy 对象的循环链表，其中，指针 pB 总是指向最后创建的 Boy 对象。循环中，每次执行语句 pB＝new Boy(pB,i)就创建一个 Boy 对象，并将指针 pB 作为插入位置传递给类 Boy 的构造函数，创建 Boy 对象后，也将新建 Boy 对象的地址赋值给指针 pB，从而保证了 Boy 对象在循环链表的顺序与创建的顺序相同。Boy 对象在循环链表中的顺序如图 5.8 所示。

getWinner(int m)和 countUpTo(int m)成员函数包含 Josephus 游戏的核心代码，总共 9 行。其中，指针 pCurrent 是 Ring 对象的成员变量，两个成员函数共同维护这个指针，保证其总是指向数到的小孩，这样，处理逻辑就变得非常简洁。

另外，删除 Boy 对象的职责，也在这 9 行代码中完成。可在如图 5.8 所示的链表上，按照每条语句的语义人工执行这 9 行代码，能够比较容易地理解 Josephus 游戏的流程，体会到指针 pCurrent 的重要性。

getWinner(int m)中，返回获胜者时，没有返回堆中的 Boy 对象，而是另外创建了一个临时对象 win，并将它返回。这是因为删除 Boy 对象是 Ring 对象的职责，不能委托给其他类的对象，否则会出现类的边界不清，如果出现 bug 都不知道修改哪个类的代码。

4. 编程实现类 Jose

按照如图 5.9 所示的类 Jose 编写类 Jose 的声明代码，按照如图 5.6 所示的交互过程编写类 Jose 成员函数的实现代码，示例代码如例 5.3 所示。

【例 5.3】 类 Jose 成员函数的实现代码。

```
// Jose.h
#include "Ring.h"
#include "Boy.h"

class Jose
{
public:
    Jose(int boys, int interval);
    ~Jose();
    Boy gameBegin();
private:
    int numOfBoys;
    int m;
    Ring* ring;
    Boy* win;
};
```

```
// Jose.cpp
#include "Ring.h"
#include "Boy.h"
#include "Jose.h"

Jose::Jose(int boys, int interval){
    numOfBoys = boys;
    m = interval;
    ring = new Ring(boys);
```

```
    win = NULL;
}
Jose::~Jose(){
    delete ring;
    delete win;
}
Boy Jose::gameBegin(){
    if (!win)
        win = new Boy(ring->getWinner(m));
    return *win;
}
```

Jose 负责创建和删除一个 Ring 对象,在构造函数和析构函数中完成。语句

win = new Boy(ring->getWinner(m))

包含选择成员、函数调用、new 和赋值 4 个运算,其中,函数调用 ring->getWinner(m)返回的是代表获胜者的 Boy 对象,运算 new 使用这个 Boy 对象在堆中构造一个新 Boy 对象,并将其地址赋值给指针变量 win。最终,指针变量 win 指向代表获胜者的对象,并且该对象的生命周期与 Jose 对象相同,因此,在 main()函数中允许多次调用 gameBegin()函数,但类 Jose 的一个对象只调用类 Ring 的 getWinner()成员函数一次。

5. 编写 main()函数

创建两个 Jose 对象,同时玩两场游戏,示例代码如例 5.4 所示。

【例 5.4】 玩两场游戏。

```
#include "Ring.h"
#include "Boy.h"
#include "Jose.h"

#include <iostream>
using namespace std;

void main(){
    Jose js1(7, 3);
    js1.gameBegin().print();

    int m, n;
    cout << endl << "请输出小孩数和间隔数....." << endl;
    cin >> n >> m;
    Jose js2(n, m);
    js2.gameBegin().print();

    cout << endl << "现在宣布" << endl
        << "第一场获胜者是:" << endl;
    js1.gameBegin().print();
    cout << endl << "第二场获胜者是:" << endl;
    js2.gameBegin().print();
}
```

例 5.4 程序的输出结果如下。

```
离开:3
离开:6
离开:2
离开:7
离开:5
离开:1
Id:4
请输出小孩数和间隔数.....
13
7
离开:7
离开:1
离开:9
离开:4
离开:13
离开:11
离开:10
离开:12
离开:3
离开:8
离开:2
离开:5
Id:6
现在宣布
第一场获胜者是:
Id:4
第二场获胜者是:
Id:6
```

5.1.3　程序维护

运行程序时,只宣布了获胜者的编号,非常不友好,至少应该宣布获胜者的姓名。为了增加程序的友好性,可按照“小孩是人”的常识在如图 5.9 所示的类图中增加一个类 Person,并增加类 Boy 到类 Person 继承关系,这样,就能重用类 Person 的代码,管理小孩姓名、年龄等基本信息。增加继承关系后的类图如图 5.14 所示。

图 5.14 中,增加类 Boy 到类 Person 继承关系,同时也将管理 Person 对象的职责赋予 Boy 对象,需要调整对象职责及其协作。

按照划分职责的一般原则,将输入小孩信息的职责赋予 main()函数,然后通过类 Jose 和 Ring 的对象将小孩信息传递给类 Boy 的对象,最后,Boy 对象负责管理小孩信息。增加小孩信息后的对象职责及其协作如图 5.15 所示。

按照如图 5.15 所示的对象职责及其交互,修改类 Jose、Ring 和 Boy 的构造函数原型,增加传递小孩信息的参数,修改后的构造函数原型如下。

```
Boy(Boy * pPosition,int id,Person * ps);
Ring(int n,Person * ps[]);
Jose(int boys,int interval,Person * ps[]);
```

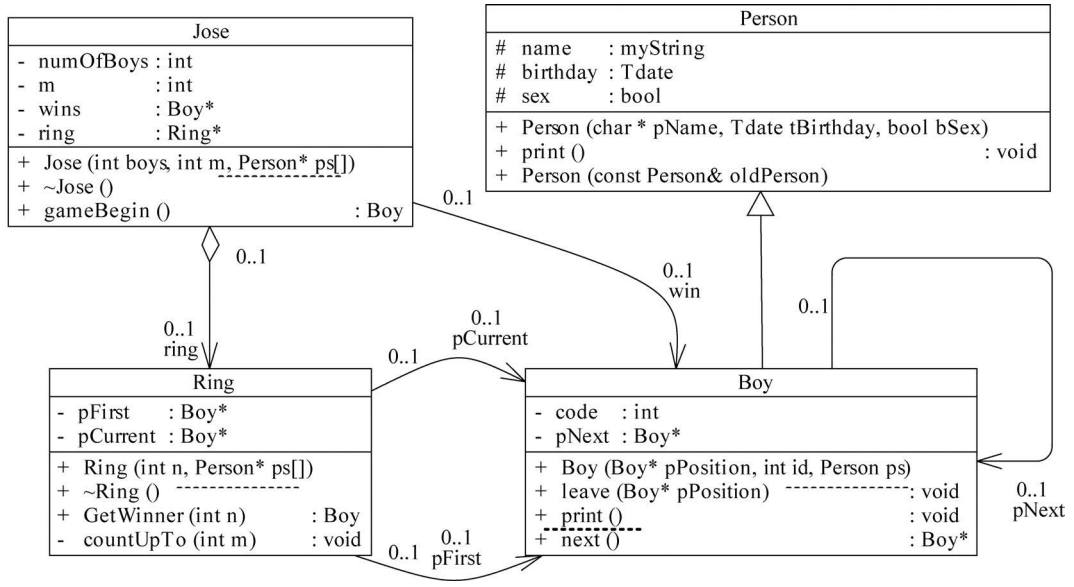

图 5.14 增加继承关系后的类图

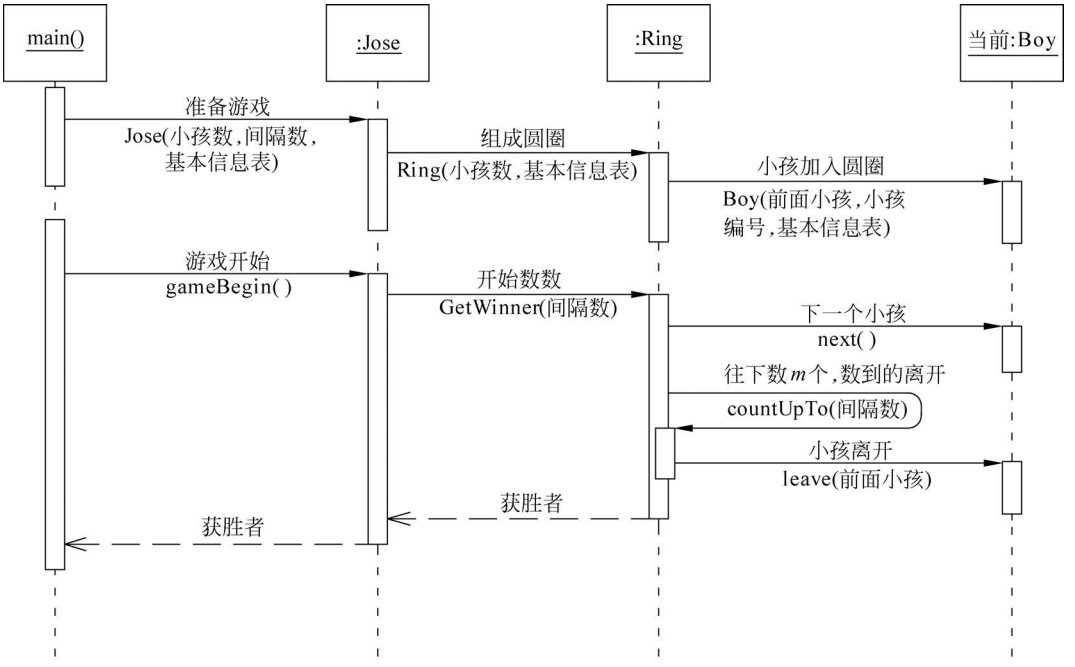

图 5.15 增加小孩信息后的对象职责及其协作

按照上述构造函数原型在 3 个类的头文件中修改其声明代码,然后,按照如图 5.14 所示的构造函数原型和如图 5.15 所示的对象职责及其交互,修改构造函数的实现代码。修改的实现代码如例 5.5 所示。

【例 5.5】 修改的实现代码。

```
Boy::Boy(Boy * pPosition, int id, Person ps) :Person(ps)
```

```
{
    code = id;
    if (!pPosition){
        this->pNext = this;        //只有一个小孩时,自己指向自己
    }
    else{
        this->pNext = pPosition->pNext;
        pPosition->pNext = this;
    }
}
Ring::Ring(int n, Person* ps[]){
    pFirst = pCurrent = new Boy(NULL, 1, *ps[1]);
    Boy* pB = pFirst;
    for (int i = 2; i <= n; i++){
        pB = new Boy(pB, i, *ps[i]);
    }
}
Jose::Jose(int boys, int interval, Person* ps[]){
    numOfBoys = boys;
    m = interval;
    ring = new Ring(boys, *ps);
    win = NULL;
}
```

上述代码中,通过类 Person 的对象或指针数组传递小孩信息。

除此之外,还需调整类 Boy 成员函数 print()的职责,其中,调用类 Person 的 print()成员函数输出小孩信息。

```
void Boy::print(){
    cout << "Id:" << code;
    Person::print();
}
```

根据前面的讨论,修改例 5.4 玩两场游戏中的类 Boy、Ring 和 Jose 的头文件和 cpp 文件,并根据需要引入头文件 Person.h 及 Tdate.h 和 myString.h,然后,在 main()函数中创建一个指向小孩信息的指针数组,最后,创建 Jose 的对象,同时将指向小孩信息的指针数组传递给其构造函数,就可以开始玩游戏了。

前面因需求变化而调整程序的设计、修改程序的代码,这类工作称为程序维护。实际应用中,程序维护的时间一般都很长,每次的需求变化比较小,修改的代码量也比较小,但往往修改比较频繁,因此,程序的易维护性很重要。

显然,前面设计的 Josephus 游戏程序,在增加小孩信息时,只修改了少量代码,其易维护性比较好。从中也发现,程序维护的工作量主要集中在调整设计和调试程序两个阶段,这也是维护程序的特点。

针对 Josephus 游戏,可能的需求变化有很多。例如,目前很多比赛都决出金牌、银牌和铜牌,这时就需要调整玩游戏的流程。又如,组织小学生玩 Josephus 游戏,希望公布哪个班级的同学取得了胜利,可将类 Boy 调整为从类 Student 继承,类 Student 的对象存储班级和学生的信息。读者可以自己试一试,按照变化后的需求修改程序。

5.2 矩阵计算

大数据智能化中的很多问题都要归结于矩阵计算,矩阵计算是大数据智能化的基础。


视频讲解

5.2.1 矩阵和向量的乘法

矩阵的一般形式为:

$$\boldsymbol{A}_{mn}=\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ a_{21} & \cdots & a_{2n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}$$

列向量的一般形式为:

$$\boldsymbol{V}_n=\begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix}$$

矩阵 $\boldsymbol{A}_{mn}$ 与向量 $\boldsymbol{V}_n$ 相乘,为一个列向量 $\boldsymbol{B}_m$,即

$$\boldsymbol{A}_{mn}\times\boldsymbol{V}_n=(b_i)_m$$

其中,

$$b_i=\sum_{j=1}^{n}a_{ij}\times v_j$$

可为矩阵 $\boldsymbol{A}_{mn}$ 和向量 $\boldsymbol{V}_n$ 声明两个类 Matrix 和 Vector,并定义 Multiply()函数实现矩阵与向量的乘法,其类图如图 5.16 所示。

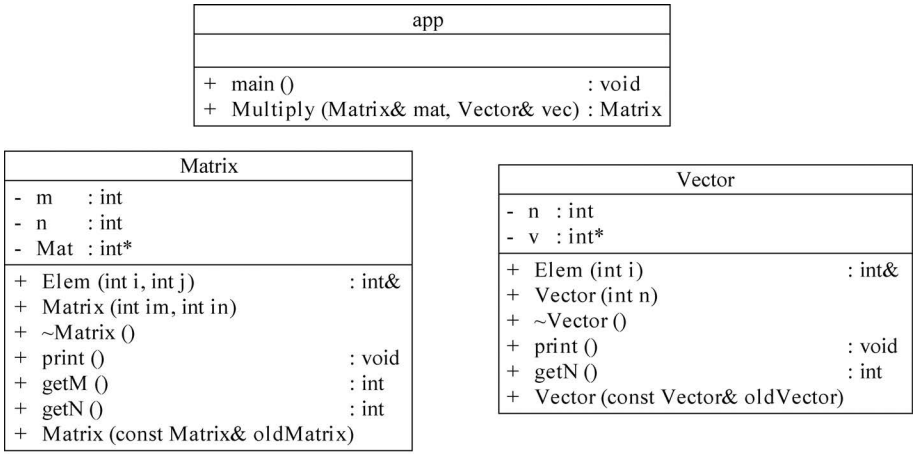

图 5.16 矩阵和向量相乘

实现矩阵与向量乘法的函数原型为:

Vector Multiply(Matrix& mat, Vector& vec)

类 Matrix 和 Vector 中,都使用动态数组来存储所包含的元素,并定义 Elem()成员函数来访问这些元素。Elem()成员函数返回元素的引用,以便在 Multiply()函数中读写元素

的值。矩阵与向量的乘法示例如例 5.6 所示。

【例 5.6】 矩阵与向量的乘法。

```
// Vector.h
class Vector
{
public:
    int& Elem(int i);
    Vector(int n);
    Vector(const Vector& oldVector);
    ~Vector();
    void print(void);
    int getN(void);
private:
    int n;
    int * v;
};
```

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

int& Vector::Elem(int i){
    if (i < n)
        return v[i];
    else
        cout << "超标越界!!";
}
Vector::Vector(int n){
    Vector::n = n;
    v = new int[n];
}
Vector::~Vector(){
    delete v;
}
void Vector::print(){
    cout << v[0];
    for (int i = 1; i < n; i++)
        cout << "\t" << v[i];
}
int Vector::getN(){
    return n;
}
Vector::Vector(const Vector& oldVector){
    n = oldVector.n;
    v = new int[n];
    for (int i = 0; i < n; i++){
        v[i] = oldVector.v[i];
    }
}
```

```
// Matrix.h
class Matrix
```

```
{
public:
    int& Elem(int i, int j);
    Matrix(int m, int n);
    Matrix(const Matrix& oldMatrix);
    ~Matrix();
    void print();
    int getM();
    int getN();
private:
    int m;
    int n;
    int * Mat;
};
```

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

int& Matrix::Elem(int i, int j){
    if (i < m&&j < n)
        return Mat[i * n + j];   //计算第 i 行第 j 列元素的位置
    else
        cout << "超标越界!!";
}
Matrix::Matrix(int m, int n){
    Matrix::m = m;
    Matrix::n = n;
    Mat = new int[m * n];
}
Matrix::~Matrix(){
    delete[](int * )Mat;
}
void Matrix::print(){
    for (int i = 0; i < m; i++){
        cout << Mat[i * n + 0];
        for (int j = 1; j < n; j++){
            cout << "\t" << Mat[i * n + j];
        }
        cout << endl;
    }
}
int Matrix::getM(){
    return m;
}
int Matrix::getN(){
    return n;
}
Matrix::Matrix(const Matrix& oldMatrix){
    m = oldMatrix.m;
    n = oldMatrix.n;
    Mat = new int[m * n];
    for (int i = 0; i < m * n;i++)
        Mat[i] = oldMatrix.Mat[i];
```

```
}
```

```
#include "Matrix.h"
#include "Vector.h"
#include <iostream>
using namespace std;

Vector Multiply(Matrix& mat, Vector& vec){
    //省略检查行列的代码
    Vector c(mat.getM());

    for (int i = 0; i < mat.getM(); i++){
        c.Elem(i) = 0;
        for (int j = 0; j < mat.getN(); j++){
            c.Elem(i) += mat.Elem(i, j) * vec.Elem(j);
        }
    }
    return c;
}
void main(){
    int m = 3, n = 4;

    cout << "矩阵:" << endl;
    Matrix a(m, n);
    for (int i = 0; i < m; i++){
        for (int j = 0; j < n; j++){
            a.Elem(i, j) = (i+1) * 10 + j+1;
        }
    }
    a.print();

    cout << endl << "向量:" << endl;
    Vector v(n);
    for (int i = 0; i < n; i++){
        v.Elem(i) = (i+1) * 2;
    }
    v.print();
    cout << endl << "矩阵×向量:" << endl;
    Multiply(a, v).print();
}
```

例 5.6 中，使用 Elem()成员函数取矩阵或向量中的元素，并在 Elem()成员函数中增加检查下标是否越界的逻辑，提高了代码的安全性。

5.2.2　使用友元提高运行速度

封装能够提高程序的安全性和可维护性，但也会降低程序的运行速度。矩阵运算往往涉及的数据量都很大，计算复杂度也比较高，希望减少封装中的部分限制，允许一些特定的类或函数可以直接访问私有的或保护的数据，以提高运行速度。

面向对象程序设计中，借鉴日常中的“朋友”概念提出了友元(friend)的概念，可以将一些函数或类指定为一个类的友元，并按照“朋友”值得相信的常识，允许这些函数或类直接访问其私有的或保护的成员。

可将例 5.6 中的 Multiply()函数设置为类 Vector 和 Matrix 的友元,Multiply()函数中就可直接访问类 Vector 和 Matrix 的数据成员,减少函数调用,从而提高运行速度。

声明友元的语法非常简单,只需要在类的声明中使用关键字 friend 声明其友元。在类 Vector 和 Matrix 的声明中,将 Multiply()普通函数声明为友元,示例代码如例 5.7 所示。

【例 5.7】 将普通函数声明为友元。

```
class Vector;
class Matrix
{
    friend Vector Multiply(Matrix& mat, Vector& vec);      //声明友元函数
    //以下代码省略
};
```

```
class Matrix;
class Vector
{
    friend Vector Multiply(Matrix&, Vector&);      //声明友元函数
    //以下代码省略
};
```

类 Vector 和 Matrix 中,将 Multiply()函数声明为友元。因 Vector Multiply(Matrix& mat,Vector& vec)中用到了两个类 Vector 和 Matrix,将函数 Multiply()声明为友元的语句前,必须先声明这两个类。例如,类 Vector 中声明友元函数的语句前面,使用语句 class Matrix 声明 Matrix 是一个类,在编译语句 friend Vector Multiply(Matrix& mat, Vector& vec)时,编译器就知道 Vector 和 Matrix 是类,能通过语法检查,否则就会报语法错误。

将 Multiply()函数声明为友元后,Multiply()函数中就可以直接访问类 Vector 和 Matrix 中存储的元素。

```
Vector Multiply(Matrix& mat, Vector& vec){
    //省略检查行列的代码
    Vector c(mat.m);
    for (int i = 0; i < mat.m; i++){
        c.v[i] = 0;
        for (int j = 0; j < mat.n; j++){
            c.v[i] += mat.Mat[i * mat.n + j] * vec.v[j];
        }
    }
    return c;
}
```

Multiply()函数中,直接访问类 Vector 和 Matrix 中的动态数组,运行速度明显提高。

例 5.7 中,使用一个全局函数实现矩阵和向量的乘法,除此之外,还可以使用成员函数实现矩阵和向量的乘法。例如,按照“A 乘以 B”的习惯,在类 Matrix 中增加 Multiply()成员函数,用于实现矩阵和向量的乘法,其类图如图 5.17 所示。

Multiply()成员函数中,只有一个类型为 Vector 的参数 vec,为了可以直接访问 vec 中的数组元素,将类 Matrix 的 Multiply()成员函数声明为类 Vector 友元。将成员函数声明为友元,示例代码如例 5.8 所示。

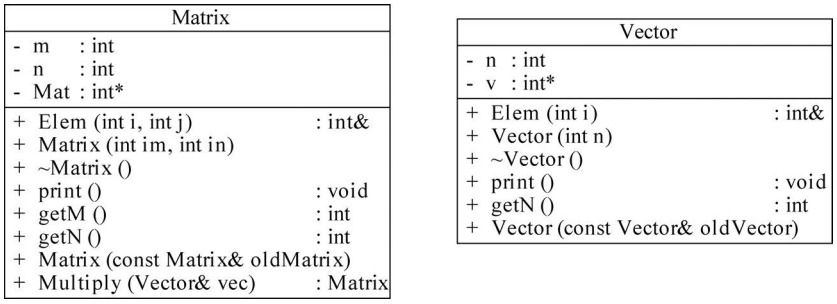

图 5.17　使用成员函数实现乘法运算

【例 5.8】 将成员函数声明为友元。

```
#include "Matrix.h"
class Vector
{
    friend Vector Matrix::Multiply(Vector& vec); //将类 Matrix 的成员函数声明为友元
    //friend class Matrix;                       //声明友元类

    //以下代码省略
};
```

如果将类 Matrix 声明为类 Vector 的友元，类 Matrix 的所有成员函数都可以直接访问类 Vector 的所有成员，存在很大的安全隐患，因此，只将其 Multiply()成员函数声明为类 Vector 的友元。

```
Vector Matrix::Multiply( Vector& vec){
    //省略检查行列的代码
    Vector c(m);
    for (int i = 0; i < m; i++){
        c.v[i] = 0;
        for (int j = 0; j < n; j++){
            c.v[i] += Mat[i * n + j] * vec.v[j];
        }
    }
    return c;
}
```

例 5.7 的 main()函数中，只需要将最后一条语句修改为：

a. Multiply(v). print();

其中，矩阵 a 是被乘数，向量 v 是乘数。

封装是面向对象程序设计的基本特征，而友元相当于在封装的边界上打了一个“洞”，开了一个“后门”，存在很大的安全隐患，因此，在使用友元前，一定要问自己三次，“能够不用友元吗?”“能够不用友元吗?”“能够不用友元吗?”。

5.3　异常处理

先考虑正常情况再考虑非正常情况，是人们的思维习惯。在实际应用开发中，首先

要区分“正常”情况和“非正常”情况,然后针对正常情况进行分析设计和编程实现,而先不考虑“非正常”情况。Josephus 游戏和矩阵计算两个程序就是按照上述思维习惯开发的。

例如,玩 Josephus 游戏,正常情况是几个或几十个小孩玩这个游戏,不正常的情况是几万个甚至几十万个小孩玩这个游戏。开发 Josephus 游戏程序时,从正常情况中选择 7 个小孩玩游戏的典型场景,并根据这个典型场景开发了 Josephus 游戏程序。按照“正常情况”选择测试用例,最终保证程序在正常情况下能够正常运行。在整个开发过程中,根本没有考虑几万个甚至几十万个小孩玩这个游戏的“非正常”情况。

又例如,进行矩阵计算时,自然会想到计算的数据量一般都很大,矩阵计算作为很多应用的基础,编写的代码往往供其他程序员使用,这些都属于正常情况,因此,开发程序中,自然就会考虑能否申请到内存、数组下标是否越界等问题,并增加相应代码,而不仅是按照矩阵计算的数学公式编写程序。

在计算机中,将非正常情况称为异常情况,简称异常,由异常情况导致的错误称为异常错误。

异常只是一个抽象概念,其本质作用是将正常情况和异常情况分开处理。在实际开发中应结合具体情况来判别异常情况,更要针对导致的错误特点以及处理错误的方便程度来划分异常。

例如,矩阵计算中,数据量很大,编写的代码供其他程序员使用,这些都属于正常情况。如果针对这些情况编写出处理错误的代码,并将这些代码嵌入到矩阵计算的代码中,显然会冲淡按照矩阵运算的数学公式编写代码这个重点,容易出现基本计算错误。因此,应将申请内存失败、数组下标越界等情况视为异常,先按照矩阵运算的数学公式编写代码,然后专门处理这些异常情况。

遵循先考虑正常情况再考虑异常情况的思维习惯,提出了包含抛出、捕获和处理三个步骤的异常处理机制,下面以矩阵和向量乘法为例,介绍使用异常处理机制编程的步骤和方法。

5.3.1 异常分类和错误定义

计算机系统或网络系统等运行环境中的原因可能导致程序运行异常,一般将这类异常称为系统异常,将系统异常导致的错误称为系统错误。例如,计算机的内存不足、打印机没有开机、文件不存在、网络不通,等等。

除了系统异常外,程序各模块在协作过程中也可能出现异常。如矩阵计算中,数组下标越界、矩阵与向量相乘时行列数量不符合要求,等等。

总之,程序在运行过程中,可能出现的异常一般会很多,不可能也没有必要处理所有的异常,而是先对可能出现的异常进行分类,然后按照这个分类分别进行处理。

矩阵计算中,可将可能出现的异常统称为异常 MyErr,然后将异常 MyErr 分为运行环境中出现的异常 SysErr 和程序内部逻辑中出现的异常 LogicErr,再将异常 SysErr 分为与文件相关的异常 FileErr 和与内存相关的异常 MemErr,将异常 LogicErr 分为矩阵中出现的异常 MatErr 和向量中出现的异常 VecErr,以及乘法中出现的异常 MulErr。异常及其分类如图 5.18 所示。

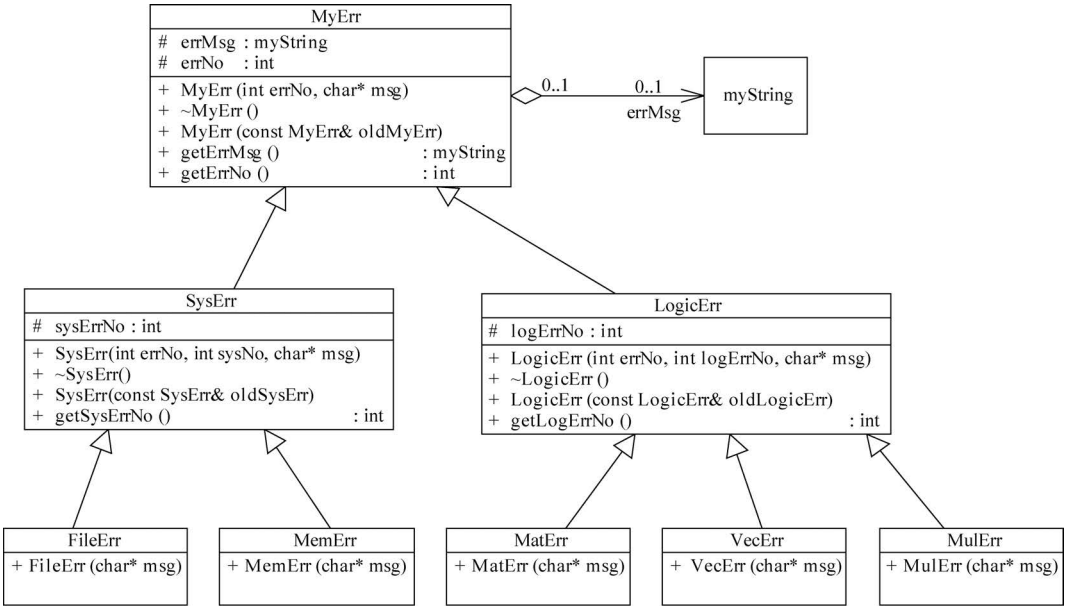

图 5.18　异常及其分类

识别异常的目的是定义错误并进行相应处理，一般按照异常的类别对异常错误进行分类，每类异常对应一类异常错误，因此，以 Err 为后缀来命名异常的类别，表示这个类别也是异常错误的类别。

错误的信息一般分为错误号和错误提示信息。错误提示信息一般只包含错误的简单描述，主要是给人阅读的，而错误号用于标识错误的类别，一般按照异常的层次关系进行编码，主要在计算机内部使用，或用于查阅错误的详细资料。

为了有效管理异常错误，在各个类中增加了相应的构造函数、拷贝构造函数和成员函数。描述异常及其错误的代码如例 5.9 所示。

【例 5.9】　描述异常及其错误。

```
// MyErr.h
class MyErr
{
public:
    MyErr(int errNo,const char * msg);
    MyErr(const MyErr& oldMyErr);
    myString getErrMsg();
    int getErrNo();
protected:
    myString errMsg;
    int errNo;
};
class SysErr : public MyErr
{
public:
    SysErr(int sysNo, const char * msg);
    SysErr(const SysErr& oldSysErr);
    int getSysErrNo();
```

```
protected:
    int sysErrNo;
};
class FileErr : public SysErr
{
public:
    FileErr(const char * msg);
};
class MemErr : public SysErr
{
public:
    MemErr(const char * msg);
};
class LogicErr : public MyErr
{
public:
    LogicErr(int logErrNo, const char * msg);
    LogicErr(const LogicErr& oldLogicErr);
    int getLogErrNo(void);
protected:
    int logErrNo;
};
class MatErr : public LogicErr
{
public:
    MatErr(const char * msg);
};
class VecErr : public LogicErr
{
public:
    VecErr(const char * msg);
};
class MulErr : public LogicErr
{
public:
    MulErr(const char * msg);
};
```

```
// MyErr.cpp
#include "myString.h"
#include "MyErr.h"

MyErr::MyErr(int errNo, const char * msg) :errMsg(msg)
{
    MyErr::errNo = errNo;
}
MyErr::MyErr(const MyErr& oldMyErr) : errMsg(oldMyErr.errMsg)
{
    errNo = oldMyErr.errNo;
}
myString MyErr::getErrMsg(){
    return errMsg;
}
```

```
int MyErr::getErrNo(){
    return errNo;
}

SysErr::SysErr(int sysNo, const char * msg) : MyErr(1, msg)
{
    sysErrNo = sysNo;
}
SysErr::SysErr(const SysErr& oldSysErr) :MyErr(oldSysErr.errNo, oldSysErr.errMsg.getString())
{
    sysErrNo = oldSysErr.sysErrNo;
}
int SysErr::getSysErrNo(){
    return sysErrNo;
}
FileErr::FileErr(const char * msg) :SysErr(1, msg)
{}
MemErr::MemErr(const char * msg) : SysErr(2, msg)
{}
LogicErr::LogicErr(int logErrNo, const char * msg) : MyErr(2, msg)
{
    LogicErr::logErrNo = logErrNo;
}
LogicErr::LogicErr(const LogicErr& oldLogicErr) : MyErr(oldLogicErr.errNo, oldLogicErr.
errMsg.getString())
{
    logErrNo = oldLogicErr.logErrNo;
}
int LogicErr::getLogErrNo(){
    return logErrNo;
}
MatErr::MatErr(const char * msg) :LogicErr(1, msg)
{}
VecErr::VecErr(const char * msg) : LogicErr(2, msg)
{}
MulErr::MulErr(const char * msg) : LogicErr(3, msg)
{}
```

上述代码中，将错误号分为两级，第一级为 errNo，第二级包含 sysErrNo 和 logErrNo。按照异常的分类进行编号，将其编号写进代码，在构造错误时只需要指定错误信息，不需要指定错误编号，以减少编程时的工作量，并保证编程的统一性。

5.3.2　识别异常和抛出错误

异常分类和定义错误后，应按照类及成员函数的职责，识别异常并抛出错误。如果是在自己职责范围内能够处理的错误，也应该进行处理；如果超出了自己的职责范围或处理不了，则抛出错误，留给上层代码处理。

例 5.6 中，类 Vector 和 Matrix 的职责是管理好向量和矩阵，按照其职责，类 Vector 应该识别向量中的异常，然后使用语句 throw 抛出异常错误。同样，类 Matrix 应该识别矩阵中的异常并抛出异常错误，不能超出自己的职责。

数组下标越界和申请内存失败这两个异常错误,两个类 Vector 和 Matrix 都不知道怎么处理,因此,只抛出了异常错误,让高层的 Multiply()和 main()函数来处理。示例代码如例 5.10 所示。

【例 5.10】 判断异常并抛出错误。

```
#include "MyErr.h"
#include "Vector.h"
#include <iostream>
using namespace std;

int& Vector::Elem(int i){
    if (i < n)
        return v[i];
    else
        throw VecErr("向量中数组下标越界!");
}
Vector::Vector(int n){
    Vector::n = n;
    v = new int[n];
    if (!v)
        throw MemErr("向量中申请内存失败!");
}
Vector::Vector(const Vector& oldVector){
    n = oldVector.n;
    v = new int[n];
    if (!v)
        throw MemErr("向量中申请内存失败!");
    //省略下面的代码
}
```

上述代码中,识别了数组下标越界和申请内存失败两种异常,并用 throw 语句抛出异常错误 VecErr("向量中数组下标越界!")、MemErr("向量中申请内存失败!")和 MemErr("向量中申请内存失败!")。

```
#include "MyErr.h"
#include "Matrix.h"
#include "Vector.h"
#include <iostream>
using namespace std;

int& Matrix::Elem(int i, int j){
    if (i < m * n)
        return Mat[i * n + j];
    else
        throw MatErr("矩阵中数组下标越界!");
}
Matrix::Matrix(int m, int n){
    Matrix::m = m;
    Matrix::n = n;
    Mat = new int[m * n];
```

```
    if (!Mat)
        throw MemErr("矩阵中申请内存失败!");
}
Matrix::Matrix(const Matrix& oldMatrix){
    m = oldMatrix.m;
    n = oldMatrix.n;
    Mat = new int[m * n];
    if (!Mat)
        throw MemErr("矩阵中申请内存失败!");
    for (int i = 0; i < m * n;i++)
        Mat[i] = oldMatrix.Mat[i];
}
```

5.3.3　捕获异常并处理错误

Multiply()函数中,使用 try …catch 语句来捕获被调用函数中抛出的异常错误,并针对接收到的异常错误进行相应处理。示例代码如例 5.11 所示。

【例 5.11】 捕获异常并进行相应处理。

```
#include "myString.h"
#include "MyErr.h"
#include "Matrix.h"
#include "Vector.h"
#include <iostream>
using namespace std;

Vector Multiply(Matrix& mat, Vector& vec){
    //检查行列的代码
    if (mat.getN() != vec.getN())
        throw MatErr("矩阵和向量的行列不符合相乘条件!");
    try{                    //下面是例 5.3 中的代码
        Vector c(mat.getM());
        for (int i = 0; i < mat.getM(); i++){
            c.Elem(i) = 0;
            for (int j = 0; j < mat.getN(); j++){
                c.Elem(i) += mat.Elem(i, j) * vec.Elem(j);
            }
        }
        return c;
    }
    catch (MemErr e){
        cerr << "错误号:" << e.getErrNo() << "\t 系统错误号:" << e.getSysErrNo()
            << "\t 错误信息:" << e.getErrMsg().getString();
        throw MemErr("申请内存失败!");
    }
    catch (VecErr e){
        cerr << "错误号:" << e.getErrNo() << "\t 逻辑错误号:" << e.getLogErrNo()
            << "\t 错误信息:" << e.getErrMsg().getString();
    }
}
```

Multiply()函数中,首先使用条件表达式 mat.getN()! =vec.getN()检查异常,如果条件为真,说明矩阵和向量的行列不正确,这是因调用者没有做好本职工作而出现的错误,应该由调用者来处理,因此,只抛出异常错误 MatErr("矩阵和向量的行列不符合相乘条件!")。

后面是 try…catch 语句,try 和 catch 之间包含一段代码,其作用是捕获执行这段代码过程中抛出的异常,也包括执行函数调用过程中抛出的异常。

后面有两个 catch 语句,其中,语句 catch(MemErr e)捕获 MemErr 类型的错误,e 是类 MemErr 的一个对象。紧跟的语句块处理这个错误,先在标准错误设备 cerr 上输出错误信息,再抛出一个异常错误 MemErr("申请内存失败!")。同样,语句 catch(VecErr e)捕获 MemErr 类型的错误,并进行相应处理。

上述代码,按照 Multiply()函数的职责,识别"行列不符合相乘条件"这种异常并抛出了错误,捕获了 MemErr 类型的异常,因超出了自己的职责范围而再次抛出了一个错误。

"数组下标越界"一定是 Multiply()函数中的错误引起的,改正 VecErr 和 MatErr 类型的错误是 Multiply()函数的职责,因此,捕获并处理 VecErr 类型的异常。但 Multiply()函数中忘记了捕获并处理 MatErr 类型的异常。

```
void main(){
    int m = 3, n = 4;
    try{                    //下面是例 5.3 中的代码
        cout << "矩阵:" << endl;
        Matrix a(m, n);
        for (int i = 0; i < m; i++){
            for (int j = 0; j < n; j++){
                a.Elem(i, j) = (i + 1) * 10 + j + 1;
            }
        }
        a.print();

        cout << endl << "向量:" << endl;
        Vector v(n);
        for (int i = 0; i < n; i++){
            v.Elem(i) = (i + 1) * 2;
        }
        v.print();
        cout << endl << "矩阵×向量:" << endl;
        Multiply (a,v).print();
    }
    catch (MatErr e){
        cerr << "错误号:" << e.getErrNo() << "\t 系统错误号:" << e.getLogErrNo()
            << "\t 错误信息:" << e.getErrMsg().getString();
    }
    catch (MemErr e){
        cerr << "错误号:" << e.getErrNo() << "\t 系统错误号:" << e.getSysErrNo()
            << "\t 错误信息:" << e.getErrMsg().getString();
    }
}
```

main()函数中，使用 try…catch 语句，捕获异常错误并做相应处理。上述代码中，标出了捕获异常错误的代码范围及可能抛出异常错误的语句。

main()函数也处理不了内存问题，只能向用户报告，用户再来处理这个错误，如扩展内存或减少计算的数据量等。

小结

本章主要学习了综合运用前面学习的知识和技术解决实际问题的步骤和方法，重点学习了分析设计、编码实现和程序维护阶段的主要步骤和基本方法，以及根据数学模型编程的技术和方法，最后学习了异常处理及其编程方法。

以 Josephus 游戏为例，学习了从应用场景中发现对象及其连接、抽象出类及其关联、建立静态模型和动态模型的步骤和方法，以及根据建立的模型选择实现技术的思路和编写代码的方法。希望读者能够理解分析设计、编码实现和程序维护阶段中的一般步骤及其主要任务，知道每个阶段涉及的主要方法和技术，能够理解简单的静态模型和动态模型，掌握根据模型编写代码的方法。

以矩阵乘法为例，学习了根据数学模型设计程序和编写代码的方法，以及使用友元提高运行速度的编程方法。希望读者能够从计算角度进一步理解设计和编码的方法与技术。

最后学习了异常处理的相关概念，举例说明了异常分类、错误定义的思路以及编程方法。希望读者理解异常的概念，了解异常处理机制，掌握使用异常处理的编程方法。

练习

1. Josephus 游戏示例中，选择了链表存储圆圈中的小孩。如果将链表改为对象数组，选择对象数组存储圆圈中的小孩，请重新描述在对象数组上玩 Josephus 游戏的场景，并使用类图设计出程序的静态模型，最后编程实现。

2. 请在 Josephus 游戏示例中增加异常处理功能。

3. Josephus 游戏中，要求决出金牌、银牌和铜牌，请重新设计程序的静态模型和动态模型，并编程实现。

4. 组织小学生玩 Josephus 游戏，希望公布哪个班级的哪个同学取得了胜利，请重新设计出程序的静态模型和动态模型，并编程实现。

5. 矩阵计算是大数据智能化的基础，5.2 节的示例中只介绍了矩阵和向量的乘法，请扩展示例程序的功能，编程实现矩阵的乘法、加法、减法和转置运算。

6. 根据下面的描述和要求设计并编写程序。

一个四口之家，大家都知道父亲会开车，母亲会唱歌。但是父亲还会修电视机，只有家里人知道。小孩既会开车又会唱歌甚至也会修电视机。母亲瞒着任何人在外面做小工以补贴家用。此外，男孩还会打乒乓球，女孩还会跳舞。

主程序中描述这四口之家一天的活动：先是父亲出去开车，然后母亲出去工作(唱歌)，母亲下班后去做两小时小工。小孩在俱乐部打球，在父亲回家后，开车玩，后又高兴地唱歌。晚上，两个小孩和父亲一起修电视机。

后来父亲的修电视机技术让大家知道了，人们经常上门要他修电视机。这时，程序要做什么样的变动？

7. 第3章习题8要求针对大学生的选课场景设计并实现一个选课程序，请按照5.1节中的步骤和方法重新进行设计并实现。