

Delphi 程序设计 大学教程

刘艺 罗兵 周安栋 编著

 机械工业出版社
China Machine Press

目 录

第 1 章	绪论.....	7
1.1	程序与程序设计.....	7
1.1.1	程序与计算机.....	7
1.1.2	算法与数据结构.....	10
1.1.3	程序设计过程.....	13
1.2	程序设计语言.....	14
1.2.1	发展历史.....	15
1.2.2	语言的类型.....	15
1.2.3	高级语言的分类.....	16
1.3	Delphi 语言介绍.....	17
1.3.1	Delphi 是什么.....	17
1.3.2	Delphi 发展的历史.....	18
1.3.3	Delphi 程序的编写、编译和运行.....	19
1.4	本章小结.....	24
1.5	本章习题.....	25
第 2 章	程序设计基础.....	28
2.1	数据和数据处理.....	28
2.1.1	计算机的结构.....	28
2.1.2	数据的表示.....	29
2.1.3	数据的处理.....	31
2.2	数据类型.....	32
2.2.1	常量和变量.....	33
2.2.2	简单数据类型.....	35
2.2.3	复杂数据类型.....	40
2.2.4	类型关系*.....	45
2.3	程序.....	47
2.3.1	一个简单的 Delphi 程序.....	47
2.3.2	Delphi 语言要素.....	51
2.3.3	撰写规范的程序代码.....	54
2.4	本章小结.....	59
2.5	本章习题.....	61
第 3 章	运算与流程控制.....	68
3.1	表达式.....	68
3.2	运算符.....	68
3.2.1	赋值运算符.....	69
3.2.2	逻辑运算符.....	69
3.2.3	算术运算符.....	70
3.2.4	位运算符.....	71
3.2.5	增减运算符.....	71
3.3	运算符的优先级.....	72
3.4	流程控制.....	73

3.4.1	顺序结构.....	73
3.4.2	选择结构.....	73
3.4.3	循环结构.....	82
3.5	本章小结.....	90
3.6	本章习题.....	92
第 4 章	过程与函数.....	97
4.1	过程与函数的编写.....	97
4.1.1	过程.....	97
4.1.2	函数.....	98
4.1.3	指示字*.....	100
4.1.4	程序型类型*.....	102
4.2	参数.....	106
4.2.1	参数类型.....	107
4.2.2	无类型参数*.....	110
4.2.3	缺省参数.....	111
4.3	过程与函数的使用.....	112
4.3.1	调用过程和函数.....	112
4.3.2	过程与函数的重载.....	113
4.4	本章小结.....	115
4.5	本章习题.....	116
第 5 章	算法与数据结构.....	120
5.1	算法.....	120
5.1.1	算法的描述.....	121
5.1.2	常用算法.....	126
5.1.3	算法复杂性分析*.....	130
5.2	集合.....	134
5.2.1	关系运算.....	134
5.2.2	增删元素.....	134
5.2.3	交集运算.....	135
5.3	数组.....	135
5.3.1	静态数组.....	135
5.3.2	动态数组.....	137
5.3.3	排序.....	139
5.3.4	查找.....	142
5.3.5	数组参数.....	144
5.4	抽象数据类型*.....	146
5.4.1	数据类型的层次结构.....	146
5.4.2	链表.....	148
5.4.3	栈.....	153
5.4.4	队列.....	156
5.5	本章小结.....	157
5.6	本章习题.....	159
第 6 章	程序结构与结构化设计.....	163
6.1	Delphi 程序结构分析.....	163

6.1.1	Program——主程序	164
6.1.2	Unit——单元	165
6.1.3	单元的引用	169
6.1.4	标识符的作用范围	171
6.2	结构化程序设计基础	171
6.2.1	结构化设计的特征	172
6.2.2	构造结构化程序的规则	173
6.2.3	结构化程序设计方法	174
6.3	结构化设计应用举例	174
6.3.1	问题及分析	174
6.3.2	结构化设计	175
6.3.3	范例程序的实现	177
6.4	本章小结	191
6.5	本章习题	192
第 7 章	面向对象与对象模型	197
7.1	面向对象的概念	197
7.1.1	面向对象基本原理	197
7.1.2	建立面向对象的思维	199
7.1.3	UML 和对象建模	200
7.2	类	204
7.2.1	什么是类	204
7.2.2	类成员	204
7.2.3	类成员的可见性	205
7.3	方法和属性	206
7.3.1	什么是方法	206
7.3.2	方法的绑定	209
7.3.3	属性	212
7.4	本章小结	214
7.5	本章习题	215
第 8 章	面向对象程序设计	220
8.1	对象	220
8.1.1	理解对象	220
8.1.2	使用对象	221
8.1.3	对象之间的关系	230
8.2	继承	235
8.2.1	使用继承	235
8.2.2	继承与合成	244
8.3	多态	247
8.3.1	多态与动态绑定	247
8.3.2	方法的覆盖、隐藏和重载	250
8.4	接口	252
8.4.1	接口的概念	252
8.4.2	Delphi 对象接口	253
8.4.3	接口应用实例	259

8.5	本章小结.....	269
8.6	本章习题.....	270
第 9 章	开发过程与程序调试.....	280
9.1	软件开发过程概述.....	280
9.1.1	软件生命周期.....	280
9.1.2	软件开发过程.....	281
9.2	调试与测试.....	285
9.2.1	程序调试.....	285
9.2.2	软件质量与测试.....	289
9.3	异常与异常处理.....	292
9.3.1	异常与 Delphi 的异常类.....	292
9.3.2	异常保护与处理机制.....	295
9.3.3	利用异常处理编程.....	299
9.4	本章小结.....	301
9.5	本章习题.....	303
第 10 章	设计 Windows 应用程序.....	309
10.1	可视化程序设计.....	309
10.1.1	图形用户界面.....	309
10.1.2	可视化组件.....	312
10.2	Windows 窗体.....	314
10.2.1	应用程序和主窗体.....	314
10.2.2	添加其他窗体.....	316
10.2.3	动态创建窗体.....	317
10.3	菜单和工具栏.....	319
10.3.1	设计菜单.....	319
10.3.2	设计工具栏.....	322
10.3.3	设计动作.....	324
10.4	使用控件.....	326
10.4.1	控件的属性和布局.....	326
10.4.2	事件处理模型.....	331
10.4.3	示例程序: EditPad.....	334
10.5	本章小结.....	340
10.6	本章习题.....	342
第 11 章	设计数据库应用程序.....	345
11.1	数据库和数据库系统.....	345
11.1.1	数据库管理系统.....	345
11.1.2	数据库应用程序.....	347
11.1.3	数据库安全.....	347
11.2	Delphi 数据库体系结构.....	348
11.2.1	本地数据库和远程数据库.....	348
11.2.2	选择合适的体系结构.....	349
11.2.3	连接数据库服务器.....	350
11.2.4	Delphi 数据库组件介绍.....	351
11.3	基于 ADO 的数据库应用程序.....	352

11.3.1	ADO 概述	352
11.3.2	连接 ADO 数据库	353
11.3.3	ADO 数据集	356
11.3.4	设计用户界面	358
11.3.5	示例程序：图书管理系统	359
11.4	SQL 数据库编程	372
11.4.1	SQL 语言简介	372
11.4.2	使用 SQL 编程	374
11.4.3	示例程序：SQL 查询窗体	379
11.5	本章小结	385
11.6	本章习题	386
附录	391
附录 A	ASCII 码	391
附录 B	Unicode 码	394

前 言

欢迎进入 Delphi 的世界学习计算机程序设计课程。这将是一次美妙和激动人心的探索，可能将为你今后从事的充满挑战和令人兴奋的职业奠定软件编程的基础。因为众所周知计算机在我们的日常生活中扮演了一个重要的角色而且在未来也将一样。

计算机科学是一个充满了挑战和发展机遇的年轻学科，而计算机程序设计则是这门学科的重要基础。随着计算机在各行各业的广泛应用，很多非计算机专业的课程设置中也把计算机程序设计列为公共基础课之一。

既然是作为基础课的教材，那么本书所假定的读者可以既不具有程序设计经验，也没有面向对象技术的概念和 Windows 程序设计知识，甚至没有太多的计算机知识。即使是一个对计算机一无所知的人，也将能通过一天天学习本书而获取所有有关的基本知识，学习程序设计。如果读者是一位很有经验的程序员，已在其它程序设计语言中掌握了一定的开发技能，也能在本书中发现很多有用的信息。

本书与程序设计课程

计算机程序设计既是一门概念复杂，知识面广的理论课，也是一门面向实战、需要动手的实践课。几乎所有的初学编程者都梦想着有朝一日能在计算机上驰骋，让一行行程序在自己敲击键盘的手下源源不断地流出，真正成为驾驭计算机的主人。然而，学完程序设计课程后，实际开始编写程序时，却往往会觉得难以下手、无所适从。尽管自己刻苦学习，高分通过考试，但不能体会到所学知识给实际编程带来的便利和优势。

为什么会这样？一方面原因是我们的学生在学习时没有掌握程序设计的一般过程，没有深入了解通用程序设计语言的本质规律。另一方面是我们的教学体制僵化、教材陈旧，教学思想和内容跟不上时代的发展，与软件开发实际情况脱节。

计算机程序设计语言是一种实现对计算机操作和控制的人造语言，与人类的自然语言有一定差距。程序设计语言仅仅是程序设计的手段和途径而并不是程序设计全部。因此，掌握程序设计语言并不意味着就精通程序设计，就能写出优秀的程序。实际上，程序设计所涉及的领域、知识和技能要远远超出我们的想象。因此本教材对于程序设计课程在一些方面有着自己不同的理解：

程序设计首先是一个过程

程序设计过程通常分为问题建模、算法设计、编写代码和编译调试等 4 个阶段。不同阶段的任务是相对独立的，不能混为一谈。即使是一个比较简单的程序，我们也应该养成先分析，再下手，最后调试的习惯，严格遵循程序设计过程。因为在缺乏对问题深入、全面分析的情况下，就匆匆动手编写程序，将会增加失败的风险，带来后期修改、维护的麻烦。因此学习程序设计，不但不能回避程序设计过程，更要从软件开发过程和软件生命周期的高度来了解和掌握程序设计过程，从一开始就要养成遵从程序设计准则从事程序设计的良好习惯。有别于其他程序设计教材，本书强调程序设计过程和软件开发过程的重要性，为读者介绍了有关软件建模与测试的基本原理和技术。特别考虑到现代软件开发依赖于集体合作和项目管

理，是汇集了很多程序设计过程的更大的过程。因此，除了在书中增加有关软件过程实施和管理的介绍外，还把如何撰写规范的程序代码作为重要一节，使得读者在学习程序设计之初就了解程序设计的规范，注重编写程序的规范性、正确性和可靠性，对于培养将来参与大型软件开发所需要的分工合作团队成员十分重要。

程序设计还是一种解决问题的方法和能力

程序设计课程主要是学习用计算机解决问题的思考方法，培养编程应用能力，而不是仅仅学会某个程序设计语言的语法规则。很多学生能弄清楚循环、if-else 结构以及算术表达式，但很难把一个编程问题分解成结构良好的 Delphi 程序。这暴露了程序设计教学中偏重语法细节，忽略总体思想方法和整体过程实现的问题。

尽管程序设计理论的发展为解决问题提供了很多有效方法，但对于初学者而言学习的捷近应该是抓住最核心的思想方法：即结构化方法和面向对象方法。为实现这个目的，我们既把结构化算法分析和设计作为教材重点，也把面向对象分析和设计作为重点。对于前者，我们以顺序结构、选择结构和循环结构这三种基本结构为基础，讲解常用的结构化算法；对于后者，我们则围绕面向对象的抽象性、继承性、多态性和封装性这 4 个本质特点阐述面向对象程序设计的基本方法。通过强调基本概念、基本方法、基本应用，我们旨在为初学者奠定扎实的程序设计基础，树立良好的编程思想。通过大量的实例分析和范例程序设计过程演示，我们力图给初学者建立完整印象，培养其从整体上思考问题和解决问题的编程能力。

程序设计最终是对程序设计语言的应用

程序设计和程序设计语言存在着有趣的辩证关系。程序设计可以用不同的程序设计语言来实现，但是不同的程序设计语言又决定着能使用怎样的程序设计思想方法和技术技巧，制约着程序设计的实现能力和效率。本书使用 Delphi 作为学习程序设计的语言，并不是因为 Delphi 有强大的可视化编程功能，而是因为 Delphi 不但继承了 Pascal 语言完美的结构化风格，而且还具有面向对象语言的真正优势。更可喜的是 Delphi 还在继续发展，不断吸取现代编程语言的精华。这一切使得 Delphi 具备现代通用程序设计语言的主流特征，特别适合教学使用。因此学习 Delphi 语言，掌握 Delphi 程序设计方法是本课程的另一重要任务。

本书虽然以 Delphi 语言为背景介绍程序设计语言的相关知识，但是重点强调的是一些通用的思想方法，而放弃了 Delphi 的一些奇技淫巧。读者应该注意到，不同的程序设计语言其语法和风格可能迥异，但无论哪一种语言，都是以数据（类型）、操作（运算）、控制（逻辑流程）为基本内容。更进一步讲，学习一门程序设计语言，应该超越语言的具体表述格式，不拘泥于繁芜的语法现象，而是站在抽象的高度，掌握程序设计的基本概念，深入了解程序设计语言的本质规律。这样将会为深入学习其他程序设计语言带来便利。

本书的结构

这本书是为计算机程序设计课程编写的。该课程对于理工科类的大学相当于公共外语那样的公共基础课，它通过讲授一门具体的计算机语言，来帮助学生掌握程序设计的基础知识和基本应用。同时，对于未接触过计算机科学的学生，这本书还涉及和介绍了与程序设计相关的计算机科学知识。全书由程序设计基础、算法与结构化程序设计、面向对象程序设计、

Windows 程序设计 4 个部分组成本书的核心知识，并涉及计算机基础、数据和控制、程序设计理论、软件工程知识等四大知识领域，汇集 20 个相关知识点。虽然在本书中讨论的内容有一定的理论性，但这些理论都是用实际程序问题表达的，是为了帮助读者建立完整的程序设计知识体系结构。

本书的组织结构如下图所示，其中有些知识点是通过各章节的迭代，循序渐进，不断深入的：

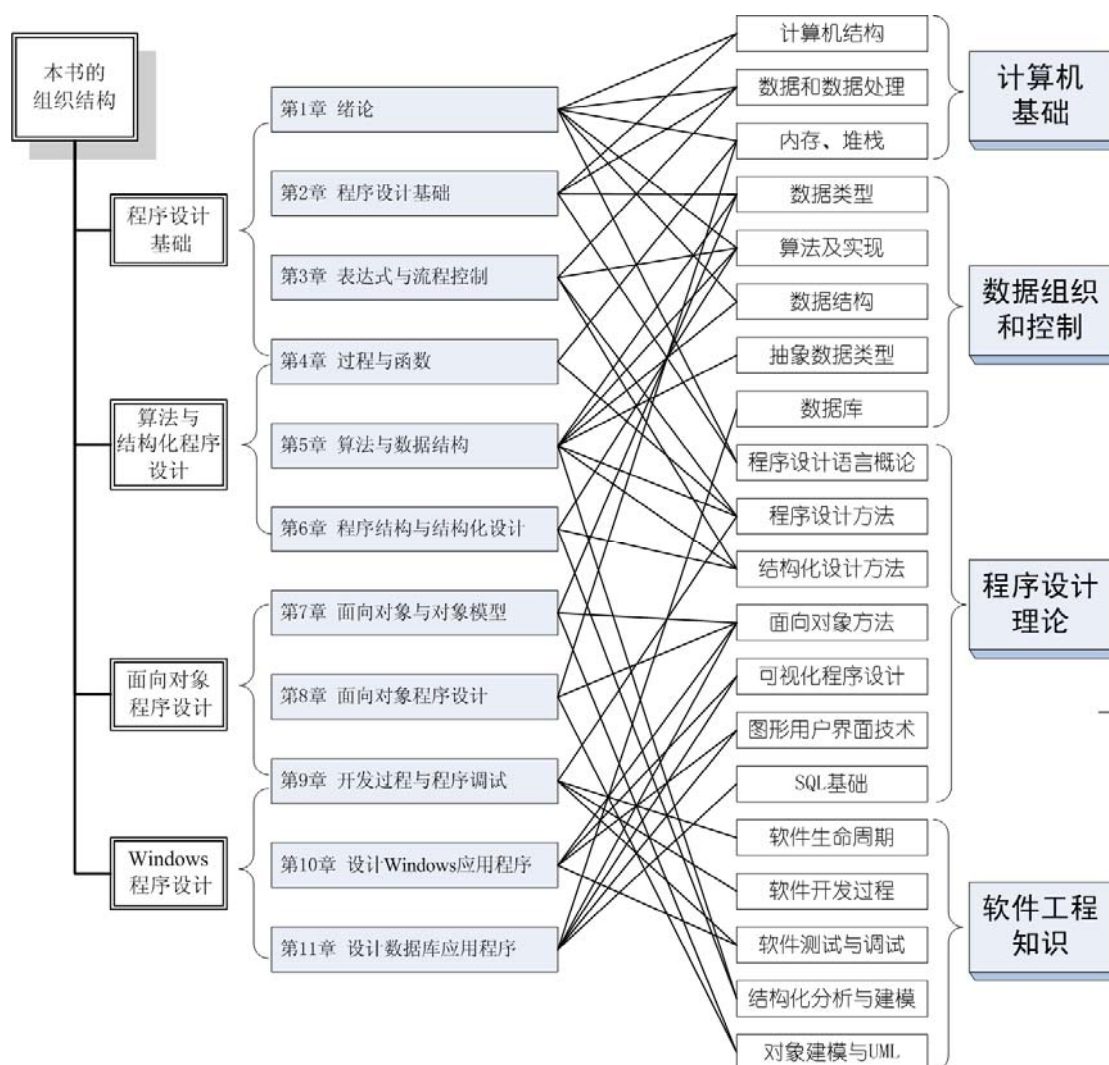


图 P.1 本书组织结构与知识点

本书分为 11 章，各章的主要内容如下：

- **第 1 章 绪论** 介绍程序设计的基本概念和初步认识 Delphi。重点帮助读者搞清楚什么是计算机程序、程序设计、程序设计语言等基本概念。
- **第 2 章 程序设计基础** 以数据和数据处理作为程序设计的基础，通过了解变量、常量和数据类型开始探索 Delphi 程序设计语言。通过建立第一个 Delphi 程序介绍 Delphi 语言要素，说明如何撰写规范的程序代码。
- **第 3 章 表达式与流程控制** 介绍 Delphi 运算符和表达式。讲解表达式语法、各种算术、逻辑和其它运算符。并通过介绍实现顺序、选择、循环结构的常用 Delphi 语句，帮助读者完整地理解如何控制程序的逻辑流程。
- **第 4 章 过程与函数** 讲解 Delphi 过程和函数的声明与实现，介绍参数类型与传递方式。最后讨论过程和函数的使用。

- **第5章 算法与数据结构** 介绍算法的概念及常用算法。并通过集合、数组、链表、栈、队列等数据结构，讨论算法的应用及算法的 Delphi 程序实现。
- **第6章 程序结构与结构化设计** 剖析 Delphi 程序的基本结构，介绍结构化程序设计方法。引入单元概念并且显示如何从较简单的程序块构造复杂的结构化程序。最后通过一个结构化设计的应用实例，演示结构化程序设计的完整实现过程。
- **第7章 面向对象与对象模型** 介绍面向对象的概念和对象建模的方法，讲解对象模型中的核心部分：类及类的成员。使读者学会如何用 Delphi 语言声明和实现一个类以及如何使用方法和属性。
- **第8章 面向对象程序设计** 围绕面向对象的抽象性、继承性、多态性和封装性这4个特点讲解面向对象程序设计的基本方法。重点讲授如何使用对象和对象接口，如何在程序设计中继承、合成和多态等面向对象技术。
- **第9章 开发过程与程序调试** 介绍软件的开发过程及过程的实施管理，从软件质量的高度讨论了程序的调试与测试，重点讲述了 Delphi 程序的调试方法和程序中的异常处理。
- **第10章 设计 Windows 应用程序** 讲解 Windows 应用程序的一般设计方法，包括如何创建窗体、设计界面、使用控件、事件编程等。读者可以学到 Delphi 在可视化快速开发 Windows 应用程序方面的强大功能。
- **第11章 设计数据库应用程序** 本章读者将学习如何用 Delphi 创建关系数据库应用程序。同时还介绍一些数据库理论基础。除了学会使用 Delphi 功能强大的数据库组件，本章还通过实例重点讲解和演示了 ADO 和 SQL 数据库编程。

尽管本书包含了以上章节内容，但实际的教学进度和授课内容可以灵活确定，因为这要取决于课堂教学的安排或读者实际技能及对所讨论问题的熟悉程度。教学时数建议安排在20~60课时之间。其中标记*的章节仅供有能力的学生选学。

本书的读者对象

- 对计算机一无所知的人，想通过程序设计了解和使用计算机，并通过学习而获取所有有关的基本知识。显然，Delphi 是一种比较容易学习和入门的语言。
- 程序设计入门学生，选用本书作为程序设计基础教材，希望掌握一门程序设计语言，为深入学习其它计算机课程奠定基础。显然 Delphi 无论是在结构化还是面向对象，或者是可视化 Windows 编程方面都能胜任教学的需要。从而为学习程序设计提供更全面的知识结构体系。
- 因工作或科研需要的非专业编程人员，希望迅速掌握一门程序设计语言以完成不太复杂的编程任务。
- 非计算机专业的编程爱好者，改行从事程序员工作，有一些实际的经验，但没有系统学习过相关专业知识。希望通过本书重温程序设计知识，补习相关概念和理论。
- 有一定经验的程序员，已在其它编程语言及其软件环境中掌握了一定的开发技能，但没有使用过 Delphi 语言或对 Delphi 语言一知半解。希望在本书中系统学习 Delphi 程序设计，发现 Delphi 与其所熟悉语言的不同点，并由此掌握 Delphi 语言。

本书的特色

本书的一些特色不仅使得本书与众不同，同时也特别有助于入门者去学习。

概念和知识面

贯穿本书，我们始终强调概念要比数学模型更重要，我们认为对概念的理解必然左右对模型的理解。同时，我们还特别注意开阅读者的知识面，使读者能够站在现代软件开发和软件工程这个比较开阔的层面上了解程序设计，而不是局限于繁琐的程序设计语言规则上。

代码编写能力

初学 Delphi 程序设计的人，大多数会被 Delphi 强大的可视化编程功能所吸引。的确快速应用开发(RAD)的思想正在改变程序设计的方法。而可视化程序设计实际上已经重造了开发者的工作平台。以前编写程序是通过基于字符的编辑器键入一条条语句的方法，现在我们可以交互式地在窗体上点击和拖放 (click-and-drop) 组件，并使用短小精悍的代码段连接它们。过去，即使是开发很小的 Windows 应用程序，也需要很多细心而且繁杂的基础工作，先写出非常长的源代码文件，然后才可编译和测试其结果。Delphi 改变了这种模式。它首先创建一个 Windows 应用程序的初始框架代码，即缺省的用户界面，而无须写任何程序。程序员的创造力被用于真正的程序设计任务，而不是浪费在窗口的几个控件上。

可是，千万不要被容易的、可视化的、基于组件的程序设计所迷惑，把程序设计变成了点击和拖放组件，然后在窗体上重新排列组件的工作。如果是这样学习程序设计，我们培养的是拖拉现成组件的“拖拉员”而不是创造代码的程序员。真正的程序设计需要很多知识、技能和创造力。为此，我们的教材不同于大多数 Delphi 程序设计教材以可视化程序设计为重点的编写模式。为避免过早地引入可视化组件的使用，分散学习程序设计语言基础知识的注意力，我们在本书第 10 章设计 Windows 应用程序之前尽量以控制台程序为示例讲解程序设计方法，重在学习程序设计语言的本质，培养代码的编写能力。

图文并茂

阅读本书后将会发现本书图文并茂。全书有 100 多幅精心设计的图片，这些图片可以帮助读者增进对文字的理解。

示例程序

本书尽可能地运用示例程序来表述概念和模型，同时尽量提供完整的范例程序和程序设计过程。本书所有示例程序和习题中的程序都已在 Windows 环境下 Delphi 7 中编译运行通过，建议读者在学习时选择 Delphi 7 的版本。

另外部分示例程序的源代码也可以在以下网址找到：<http://www.liu-yi.net>

小结和习题

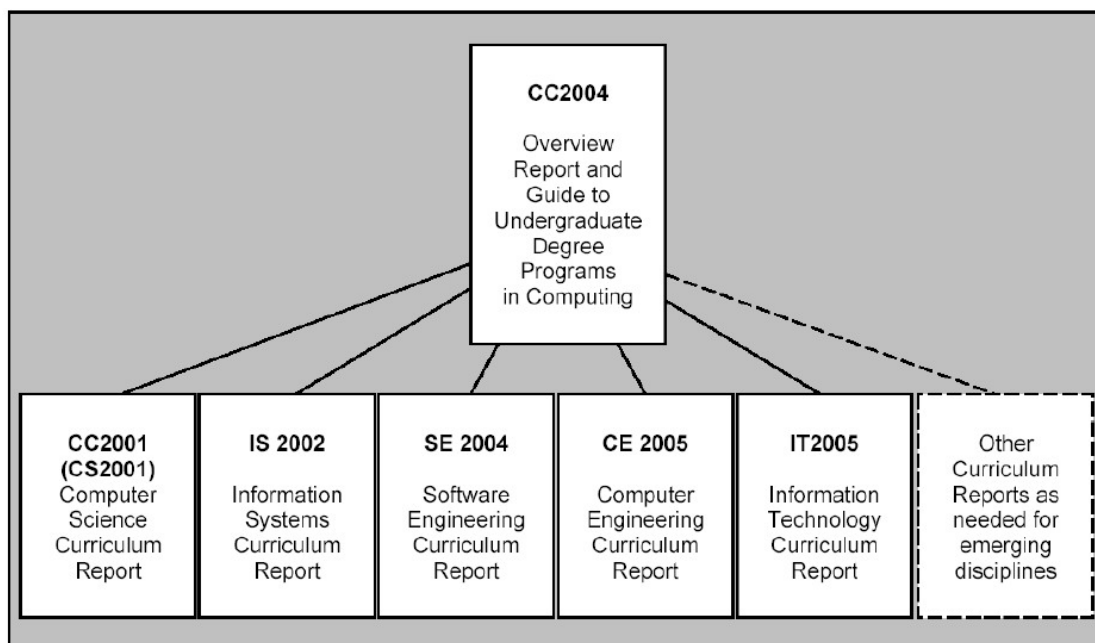
每一章的结尾都包括本章小结和本章习题。本章小结包括了对本章中所有关键内容和知识点的简明概括，是复习时的参考。本章习题包括了三部分内容：复习题，测试题和练习题。

- 复习题：测试本章中所有的要点和概念，帮助学生复习巩固重点内容。
- 测试题：通过多项选择题，客观地测试学生对所学知识的理解和掌握程度。
- 练习题：通过课后练习题，检查学生能否运用掌握的概念和知识独立思考，解决问题。

本教材配有专用的习题解答及课程设计教辅书籍。另外部分习题答案也可以在以下网址找到：<http://www.liu-yi.net>

CC2004 课程体系

从 1990 年开始，美国电气和电子工程师协会计算机社团（the Computer Society of the Institute for Electrical and Electronic Engineers，简称 IEEE-CS）和计算机学会（Association for Computing Machinery，简称 ACM）就着手开发新的本科生计算机课程体系。1991 年联合推出了 Computing Curricula1991（简称 CC1991），当时仅限于 Computer Science 和 Computer Engineering 两个专业的课程。1998 年秋季开始，IEEE-CS 和 ACM 联合投入新的力量更新该课程体系，并在 2001 年开发出 Computing Curricula2001（简称 CC2001），并将该计算机课程体系扩大到 Computer Science、Computer Engineering、Software Engineering、Information Systems 等多个专业。在 CC2001 的实施中，专家们发现，计算机课程所涉及的学科专业和教学范围正在不断扩大，而且在内容和教学方面的变化也日新月异。IEEE-CS 和 ACM 意识到 10 年一次的 Computing Curricula 修订已经难以满足要求，于是联合国际信息处理联合会（International Federation for Information Processing，简称 IFIP）、英国计算机协会（British Computer Society，简称 BCS）等更多的组织开发了 Computing Curricula2004（简称 CC2004），使之成为开放的、可扩充的、适合多专业的、整合了计算机教学相关原则体系观点的课程体系指南。其结构参见下图。



为了进一步反映当代计算机科学技术的发展水平，与国际主流计算机教育思想接轨。通过多年来对 IEEE-CS 和 ACM 的 Computing Curricula 课程体系的跟踪研究，我们在本教材的编写中，借鉴了 CC2004 课程体系的最新研究成果，同时吸取了国外同类教材的优秀经验，其目的是进一步推动教材和课程改革，培养有竞争力人才。

致谢

本书是在作者多年科研和教学基础上编写的,主要参考了作者已发表的文章和著作以及教学中积累的资料。书中还用到了其他中外文教材、资料,由于无法在此一一列举,现谨对这些教材和资料的作者表示衷心的感谢。

参与本教材编写工作的人员还有网通吉林市分公司的孙滔,太原师范学院计算机中心的刘星,海军工程大学的段立、李启元、杜军、吴苗、曹旭峰,南京航空航天大学无人机研究所的吴英,以及杨德刚、刘藩、吴永逸、洪蕾等。

一本书的出版离不开许多人的支持,尤其是这本书。为此感谢我们的家人和朋友。我们在忍受写作之苦的同时,牺牲了与他们共享天伦之乐的宝贵时光。

由于作者水平有限,本书中难免有疏漏和不妥之处,恳请各位专家、同仁和读者不吝赐教,并在此表示特别感谢!



<http://www.liu-yi.net>
2005年2月17日于南京

第5章 算法与数据结构

在第一章绪论中我们讲过,使用计算机解决问题时,除了需要使用计算机语言描述算法,还必将涉及数据结构。从这个意义上讲,程序是建立在数据结构基础上使用计算机语言描述的算法,因此简单地讲,程序也可以表示成:算法+数据结构。

学习计算机程序设计,目的在于利用计算机来解决遇到的实际问题。从这个方面来说,Delphi 只是一种设计程序的计算机语言工具。至于具体怎么用它来实现程序,解决实际问题,通常还需要相应的算法和数据结构的支持。算法是抽象的,它不仅限于 Delphi 语言,实际上算法理论独立于任何一种程序设计语言,而且算法本身博大精深、自成体系。算法也是具体的,算法的应用还必须结合不同的数据结构,并在具体的程序中得以实现。

所谓数据结构就是用来组织信息的一种结构体,它使访问和处理信息变得容易而有效。数组就是一种最常见的数据结构例子,它其中的所有数据都具有相同的类型,也就是说属于同一类。其每个元素是通过它在结构中的位置(索引或下标)来访问的。为了解决更复杂的问题,通常还需要开发特殊的数据结构来储存和维护信息,其中有些数据结构,如:链表、栈、队列等已经成为计算机科学中经典的数据结构。

本章先概括性地介绍算法的基础知识,并结合集合、数组、链表、栈、队列等数据结构,讨论算法的应用及一些典型算法的 Delphi 程序实现。

5.1 算法

算法的英文单词为“Algorithm”。这个单词一直到 1957 年之前,在《韦氏新世界词典》中还未出现,我们只能找到带有它的古代涵义的“Algorism(算术)”,指的是用阿拉伯数字进行算术运算的过程。一本早期的德文数学词典《数学大全辞典》,给出了另一个单词“Algorithmus”的如下定义:“在这个名称之下,组合了四种类型的算术计算的概念,即加法、乘法、减法、除法”。拉丁语中用到过一个短语“algorithmus infinitesimalis(无限小方法)”,在当时就用来表示数学家莱布尼兹所发明的以无限小量进行计算的微积分方法。1950 年左右,“algorithm”一词经常地同欧几里德算法“Euclid's algorithm”联系在一起(这个算法就是在欧几里德的《几何原本》中所阐述的求两个数的最大公约数的过程,也即辗转相除法)。

现在,基本可以明确算法的含义:算法是为了解决某一问题在有限步骤内、定义了具体操作序列的规则集合。

通俗点说,算法就是针对一类特定问题,使用计算机解题的过程。在这个过程中,无论是形成解题思路还是编写程序,都是在实施某种算法。前者是推理实现的算法,后者是操作实现的算法。

一个算法应该具有以下五个重要的特征:

- **确切性(No ambiguity)** 算法的每一步骤必须有确切的定义。而不应该有二义性,例如,在算法中不能出现诸如“赋值为 100 或 1000”。
- **输入(Input)** 有 0 个或多个输入,用于初始化运算对象。所谓 0 个输入是指无需输入条件,而算法本身定出了初始条件。
- **输出(Output)** 没有输出的算法是毫无意义的。一个算法应该有一个或多个输出,以反映对输入数据加工后的结果。
- **可行性(Feasibility)** 算法原则上能够精确地运行,而且对于算法中的每种运算,在原理上人们应该能用笔和纸做有限次运算后完成。

- **有穷性 (Finite)** 算法必须保证执行有限步之后结束。只具有前面四个特征的规则集合，称不上算法。例如，尽管操作系统能完成很多任务，但是它的计算过程并不终止，而是无穷无尽的执行、等待执行，所以操作系统不是算法。

5.1.1 算法的描述

1. 伪代码描述

伪代码 (Pseudo-code) 是一种算法描述语言。使用伪代码的目的是为了使被描述的算法可以容易地以任何一种编程语言 (如 Delphi、C、Java 等) 实现。因此，伪代码必须结构清晰，代码简单，可读性好，并且类似自然语言。

下面我们介绍一种常用的类似 Pascal 语言 (Delphi 语言的前生) 的伪代码。语法规则如下：

在伪代码中，每一条指令占一行 (else if 例外)，指令后不跟任何符号。书写上的“缩进”表示程序中的分支程序结构，这种缩进风格也适用于 if-then-else 语句。用缩进取代传统 Pascal 中的 begin 和 end 语句来表示程序的块结构可以大大提高代码的清晰性；同一模块的语句有相同的缩进量，次一级模块的语句相对与其父级模块的语句缩进；例如：

```
line 1
line 2
  sub line 1
  sub line 2
    sub sub line 1
    sub sub line 2
  sub line 3
line 3
```

而在 Pascal 中这种关系用 begin 和 end 的嵌套来表示：

```
line 1;
line 2;
begin
  sub line 1;
  sub line 2;
  begin
    sub sub line 1;
    sub sub line 2;
  end;
  sub line 3;
end;
line 3;
```

在伪代码中，变量名和保留字不区分大小写，这一点和 Pascal 相同，与 C 或 C++ 不同；

在伪代码中，变量不需声明，但变量局限于特定过程，不能不加显示的说明就使用全局变量；

赋值语句用符号 \leftarrow 表示， $x \leftarrow \text{exp}$ 表示将 exp 的值赋给 x ，其中 x 是一个变量， exp 是一个与 x 同类型的变量或表达式（该表达式的结果与 x 同类型）；多重赋值 $i \leftarrow j \leftarrow e$ 是将表达式 e 的值赋给变量 i 和 j ，这种表示与 $j \leftarrow e$ 和 $i \leftarrow e$ 等价。例如：

```
x ← y
x ← 20*(y+1)
x ← y ← 30
```

以上语句用 Pascal 分别表示为：

```
x := y;
x := 20*(y+1);
x := 30; y := 30;
```

选择语句用 if-then-else 来表示，并且这种 if-then-else 可以嵌套，与 Pascal 中的 if-then-else 没有什么区别。例如：

```
if (Condition1)
  then [ Block 1 ]
  else if (Condition2)
    then [ Block 2 ]
    else [ Block 3 ]
```

循环语句有三种：while 循环、repeat-until 循环和 for 循环，其语法均与 Pascal 类似，只是用缩进代替 begin - end；例如：

```
1. x ← 0
2. y ← 0
3. z ← 0
4. while x < N
  4.1 do x ← x + 1
  4.2   y ← x + y
  4.3   for t ← 0 to 10
    4.3.1 do z ← ( z + x * y ) / 100
    4.3.2   repeat
      4.3.2.1 y ← y + 1
      4.3.2.2 z ← z - y
    4.3.3. until z < 0
  4.4.   z ← x * y
5. y ← y / 2
```

上述语句用 Pascal 来描述是：

```

x := 0;
y := 0;
z := 0;
while x < N do
begin
  x := x + 1;
  y := x + y;
  for t := 0 to 10 do
  begin
    z := ( z + x * y ) / 100;
    repeat
      y := y + 1;
      z := z - y;
    until z < 0;
  end;
  z := x * y;
end;
y := y / 2;

```

数组元素的存取有数组名后跟“[索引]”表示。例如 A[j] 指示数组 A 的第 j 个元素。符号“…”用来指示数组中值的范围。例如：

A[1…j] 表示含元素 A[1], A[2], …, A[j] 的子数组；

复合数据用对象 (Object) 来表示，对象由属性 (attribute) 和域 (field) 构成。域的存取是由域名后接由方括号括住的对象名表示。

数组可被看作是一个对象，其属性有 length，表示其中元素的个数，则 length[A] 就表示数组 A 中的元素的个数。在表示数组元素和对象属性时都要用方括号，一般来说从上下文可以看出其含义。

用于表示一个数组或对象的变量被看作是指向表示数组或对象的数据的一个指针。对于某个对象 x 的所有域 f，赋值 $y \leftarrow x$ 就使 $f[y]=f[x]$ ，更进一步，若有 $f[x] \leftarrow 3$ ，则不仅有 $f[x]=3$ ，同时有 $f[y]=3$ ，换言之，在赋值 $y \leftarrow x$ 后，x 和 y 指向同一个对象。

有时，一个指针不指向任何对象，这时我们赋给他 nil。

伪代码的函数和过程语法也与 Pascal 类似。函数值利用 return 语句来返回，调用方法与 Pascal 类似；过程用 call 过程名语句来调用；例如：

1. $x \leftarrow t + 10$
2. $y \leftarrow \sin(x)$
3. call CalValue(x,y)

参数用按值传递方式传给一个过程：被调用过程接受参数的一份副本，若他对某个参数赋值，则这种变化对发出调用的过程是不可见的。当传递一个对象时，只是拷贝指向该对象的指针，而不拷贝其各个域。

2. 图形描述

经验告诉我们画图往往是一种分析和解决问题的好办法。因为图形直观、易懂，容易说明问题。所以，即使不是几何学的问题，如果我们能给出适当的几何图形表示，也会使问题变得容易处理。程序设计中，能够用来表示算法基本概念的图主要有：PAD 图、N/S 盒图、流程图。

(1) PAD 图

问题分析图 (Problem Analysis Diagram)，简称 PAD。PAD 的目的在于以图表现程序的逻辑结构，使程序易读、易记、易理解。用以提高程序的设计、构造、检查、维护等的生产效率。PAD 使用二维树型结构图描述程序的逻辑，它的控制构造主要是基于 Pascal 的。可以说，PAD 也是 Pascal 程序的二维展开图式。因此，PAD 也可看作是 PAscal Diagram 的缩写。它是一种结构化程序设计描述工具，适用于自顶向下、逐步求精的程序开发方法。PAD 图的控制结构、图形元素分别如图 5-1 和图 5-2 所示。

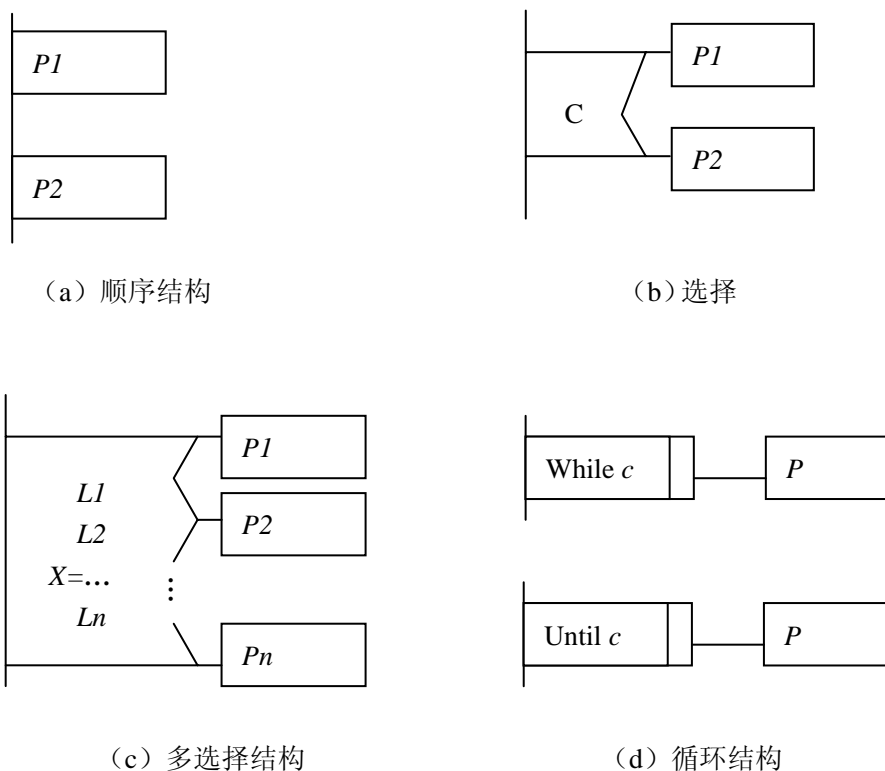


图 5-1 PAD 表示的控制结构

图形元素	功能	说明
	输入框	框内标明待输入变量名
	输出框	框内标明待输出变量名
	处理框	框内标明操作、处理或功能名

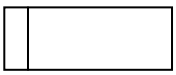
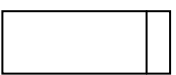

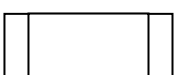
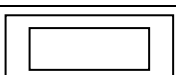
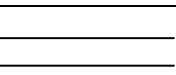

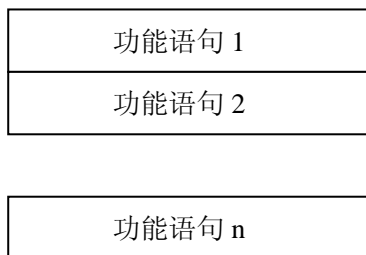
	重复框	先判断后执行循环体操作，框内为循环条件
	重复框	先执行循环体操作后判断，框内为循环条件
	选择框	框内为选择条件，可多路选择
	子算法（函数）定义框	框内为子算法名
	定义框	当图大于一张纸面时，将图的一部分定义为子图，框内为定义名
	定义	连接定义框、函数定义框及它们的具体定义
	语句标号	圈中书写标号名称

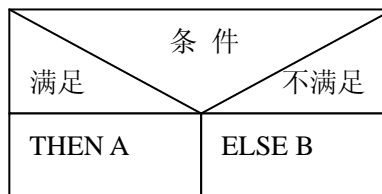
图 5-2 PAD 的图形符号

(2) N/S 盒图

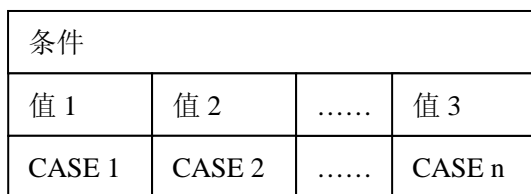
N/S 图是 I.Nassi 和 B.Shneiderman 提出的一种不需要有向线段，无需上下左右前后追踪程序流程控制的程序流程图，该图非常适合描述结构化程序或者算法的结构化实现，能够较好地反映算法和程序的层次结构，可读性好，具有自顶向下逐步求精的特征。N/S 盒图描述的控制结构如图 5-3 所示。



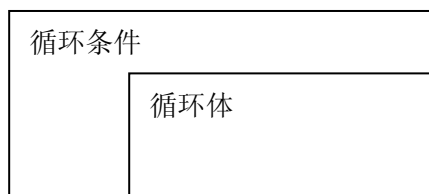
(a) 顺序结构



(b) 选择结构



(c) 多选择结构



(d) 当型循环结构

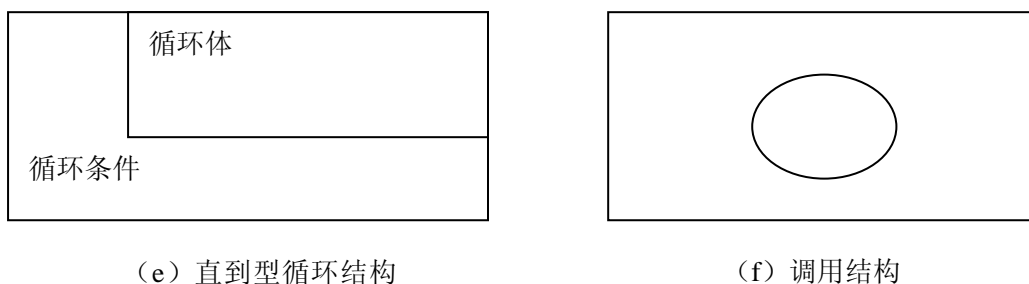


图 5-3 N/S 盒图表示的控制结构

(3) 流程图

流程图是最古老、最广泛使用的程序设计工具，也是展示程序逻辑流程的有效工具。流程图是最常用的算法图形表示法。它使用框图的形式掩盖了算法所有的细节方面，它只显示算法从开始到结束的整个流程。在程序设计环境下，它能用于设计一个完整的程序或者部分程序。程序流程图常用图形符号及控制结构图例如图 5-4 所示。

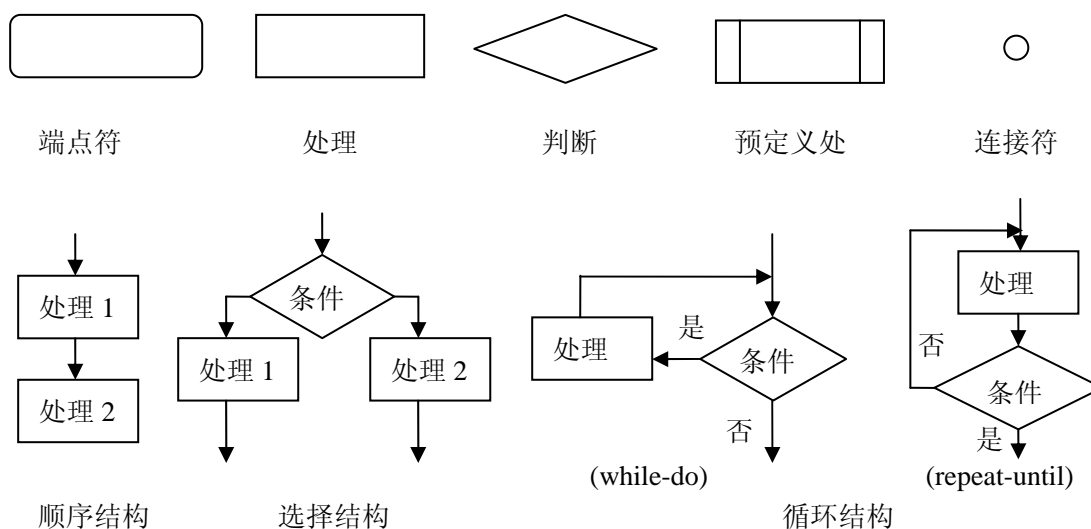


图 5-4 程序流程图常用图形符号及控制结构图例

5.1.2 常用算法

1. 基本算法

基本算法大都比较简单，是其他算法的基础。这类算法在程序中应用非常普遍，如：累加求和、累乘求积、求最大和最小值等。

这里我们来讨论在一组数据中求最大值的算法。它的思想是通过一个判断结构找到两个数中的较大值。如果把这个结构放在循环中，就可以得到一组数中的最大值。以一个从 1000 个数中找到最大值的算法为例，该算法的流程图如图 5-5 所示。

根据流程图我们还可以写出如示例程序 5-1 所示的伪代码。这里需要一个计数器 Count 用来计数，并用一个变量 Largest 保存当前比较出来的最大数。在初始化阶段给这个计数器

赋值为 0，每循环一次就对它加 1。当计数器等于 1000 时，退出循环。

之所以用伪代码来表述算法，是因为算法与具体计算机语言无关。也就是说，这个求 1000 个数中最大值的算法既可以用 Delphi 语言实现，也可以用 C++、Java 等语言实现。下面我们就看看用 Delphi 程序是如何实现的，程序代码如示例程序 5-2 所示。

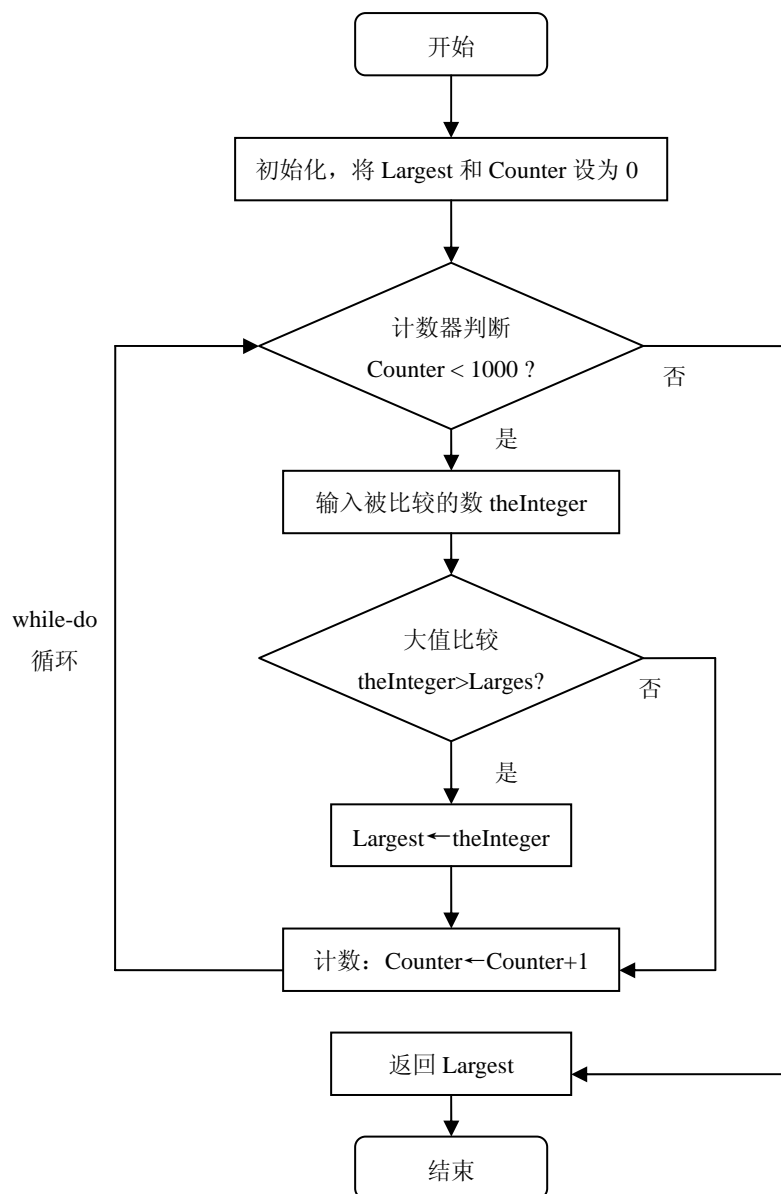


图 5-5 从 1000 个数中求最大值算法的流程图

示例程序 5-1 用伪代码写的在 1000 个数中求最大值的算法

```

FindLargest
Input: 1000 positive integers
1. Largest ← 0
2. Counter ← 0
3. while (Counter < 1000)
    3.1 Input theInteger
    3.2 if (theInteger > Largest)
        then
  
```

```

        3.2.1 Largest←theInteger
    end if
    3.3 Counter←Counter+1
end while
4. Return Largest
end

```

示例程序 5-2 在 1000 个数中求最大值算法的 Delphi 程序实现

```

program Max;
{$APPTYPE CONSOLE}
uses
    SysUtils;

var
    Largest,Counter,theInteger:integer;
begin
    { TODO -oUser -cConsole Main : Insert code here }
    //初始化
    Largest:=0;
    Counter:=0;
    while(Counter < 1000) do
    begin
        //输入被比较的数
        write('请输入第'+IntToStr(Counter+1)+'个被比较的数: ');
        readln(theInteger);
        //大值比较
        if (theInteger > Largest) then
            Largest:=theInteger;
        Inc(Counter); //计数
    end; // while
    writeln('求出最大数是: '+IntToStr(Largest));
    writeln('按回车键退出');
    readln;
end.

```

求解一组数中的最小值和上面求最大值的方法相似，只有两个小小的不同。首先，用一个判断结构求出两个数中的较小值。其次，在初始化时使用一个很大的而不能是太小的数。

2. 排序算法

排序算法根据数据的值对它们进行排列。排序是为了把不规则的信息进行整理，以提高查找信息的效率。如果没有排序，想象一下，在一个不经排序的电话本中查找某人的电话号码是多么麻烦的一件事。

常用的排序方法包括：选择排序、冒泡排序、插入排序等。这三种方法是程序设计中使用的快速排序的基础。

- **选择排序** 在选择排序中，数字列表被分为两个子列表——已排序和未排序的——它们通过假想的一堵墙分开。找到未排序子列表中最小的数字并把它和未排序子列表中第一个数字进行交换，经过每次选择和交换，两个子列表中假想的这堵墙向前移动一个元素，这样每次排序列表中将增加一个元素而未排序列表中将减少一个元素，这样就完成了一次分类扫描。一个含有 n 个元素的数字列表需要 $n-1$ 次扫描来完成数据的重新排列。
- **冒泡排序** 在冒泡排序方法中，数字列表数被分为两个子列：已排序的和未排序的。在未排序的子列表中，最小的元素通过冒泡的方法选出来并移到已排序的子列表中。当把最小的元素移到已排序列表后，将墙向前移动一个元素，使得已排序数的元素个数增加 1 个，而未排序数的元素个数减少 1 个。每次元素从未排序子列表中移到已排序子列表中，便完成一次分类扫描。一个含有 n 个元素的列表，冒泡排序需要 $n-1$ 次扫描，每次扫描比较 $n-1$ 个相邻元素来完成数据排序。
- **插入排序** 插入排序是最常用的排序技术之一，经常在扑克牌游戏中使用。游戏人员将每个拿到的牌插入到手中合适的位置，以便手中的牌以一定的顺序排列。在插入排序中，排序列表被分为两部分：已排序的和未排序的。在每次扫描过程中，未排序子列表中的第一个元素被取出，然后转换到已排序的子列表中，并且插入到合适的位置。可以看到，一个含有 n 个元素列表至少需要 $n-1$ 次排序。

其它的排序算法还有：快速排序、合并排序、希尔（Shell）排序、堆排序等等。也许读者会问为什么会有这么多的排序算法？原因就在于需要排序的数据的类型。一种方法对大多数已经排序好的数据很有效，而另一种方法对完全未排序的数据很有效。为了决定哪种方法更适合特定的程序，需要一种叫做算法复杂性的尺度来衡量。我们将在下一节的“算法复杂性分析”中讨论这个问题。

有关排序算法的 Delphi 程序实现则需要用到数据结构方面的知识。在本章的“5.3 数组”一节中，我们会举例讲解排序算法的程序实现。

3. 查找算法

查找是一种在列表（list）中确定目标所在位置的算法。在一个列表中，查找意味给定一个值，并在包含该值的列表中找到该值的第一个元素的位置（索引）。

对于列表有两种基本的查找方法：顺序查找和折半查找。顺序查找可以在任何列表中查找，折半查找则需要列表是有序的。

- **顺序查找** 顺序查找用于查找无序的列表。通常用这种方法来查找较小的列表或是不常用的列表。其它情况下，为了提高效率，最好的方法是首先将列表排序然后使用后面将要介绍的折半查找进行查找。顺序查找是从表头开始查找，当找到目标元素或确信查找目标不在列表中时，查找过程结束（因为已经查找到列表的末尾了）。
- **折半查找** 顺序查找是很慢的。如果一个列表里有一百万个元素，在最坏的情况下需要进行一百万次比较。如果这个列表是无序的，则顺序查找是唯一的方法。如果这个列表是有序的，那么就可以使用一个更有效率的方法称之为折半查找。折半查找是最常用的查找算法。折半查找是从一个列表的中间的元素来测试的，这将能够判别出目标在列表里的前半部还是后半部分。如果在前半部分，就不需要查找后半部分。如果在后半部分，就不需要查找前半部分。换句话说，可以通过判断排除一半的列表。重复这个过程直到找到目标或是目标不在这个列表里。

有关查找算法的 Delphi 程序实现则需要用到数据结构方面的知识。在本章的“5.3 数组”一节中，我们会举例讲解查找算法的程序实现。

4. 迭代和递归算法

迭代和递归是用于编写解决问题的算法的两种途径。一种使用迭代，另一种使用递归。

- **迭代** “迭”是屡次和反复的意思，“代”是替换的意思，合起来，“迭代”就是反复替换的意思，也就是使用一个中间变量保存中间结果，不断反复计算求解最终值。
- **递归** 递归是一个算法自我调用的过程，用递归调用的算法就是递归算法。递归调用会产生无法终止运算的可能，因此必须在适当的情况下终止递归调用。根据递归定义设计的递归算法中，非递归的初始定义就用做程序的终止条件。

考虑一个计算阶乘的简单例子，我们可以分别用迭代和递归这两种算法来实现。示例程序 5-3 是阶乘迭代算法的伪代码，而示例程序 5-4 则是阶乘递归算法的伪代码。

示例程序 5-3 阶乘迭代算法

```

Factorial
Input: A positive integer num
1. FactN ← 1
2. i ← 1
3. While (i < or = num)
    3.1 FactN ← FactN × i
    3.2 Increment i
end while
Return FactN
end

```

示例程序 5-4 阶乘递归算法

```

Factorial
Input: A positive integer num
1. if (num = 0)
    then
        1.1 Return 1
    else
        1.2 return num × Factorial(num-1)
    end if
end

```

5.1.3 算法复杂性分析*

算法的复杂性是指：在执行时，算法所需要计算机资源的量。需要的时间资源的量称作时间复杂性，需要的空间（即存储器）资源的量称作空间复杂性。这个量应该集中反映算法中所采用的方法的效率，而从运行该算法的实际计算机中抽象出来。换句话说，这个量应该是只依赖于算法要解的问题的规模、算法的输入和算法本身的函数。如果分别用 N 、 I 和 A 来表示算法要解问题的规模、算法的输入和算法本身，用 C 表示算法的复杂性，那么应该有：

$$C = F(N, I, A)$$

其中 $F(N, I, A)$ 是 N, I, A 的一个确定的三元函数。如果把时间复杂性和空间复杂性分开, 并分别用 T 和 S 来表示, 那么应该有:

$$T = T(N, I, A) \quad \text{和} \quad S = S(N, I, A)$$

通常, 我们让 A 隐含在复杂性函数名当中, 因而时间和空间的复杂性将分别简写为

$$T = T(N, I) \quad \text{和} \quad S = S(N, I)$$

由于时间复杂性和空间复杂性概念类同, 计算方法相似, 且空间复杂性分析相对地简单些, 所以下文将主要地讨论时间复杂性。

1. 时间复杂性

时间复杂性描述了算法在计算机上执行时, 所占用的计算机时间资源的情况。它是一种抽象的描述方式, 并不是指与算法实现效率有关的算法执行时间, 而是指理论上与问题规模、算法输入及算法本身相关的某些操作次数的总和, 通常记为 $T(n)$ 。问题规模逐渐增大后时间复杂度的极限形式称为渐进时间复杂性 (Asymptotic Time Complexity), 渐进时间复杂性确定了算法所能解决问题的规模, 通常用来分析随着问题规模的加大, 算法对时间需求的增长速度。

比较时间复杂性时经常使用这样的表达方式: 如果存在一个常数 $C > 0$, 一个算法能够在 $C \cdot n^2$ 的时间内处理完规模大小为 n 的输入, 则该算法的时间复杂性为 $O(n^2)$, 称为 n^2 级。从计算时间上可以把算法分成两类, 凡可用多项式来对其计算时间限界的算法, 称为多项式时间算法 (Polynomial Time Algorithm); 而计算时间用指数函数限界的算法称为指数时间算法 (Exponential Time Algorithm)。例如, 一个计算时间指数为 $O(1)$ 的算法, 它的基本运算执行的次数是固定的, 因此, 总的时间由一个常数 (即, 零次多项式) 来限界, 而一个时间为 $O(n^2)$ 的算法则由一个二次多项式来限界。以下七种计算时间的算法是最为常见的, 其关系为:

$$O(1) < O(\log_2 n) < O(n) < O(n \log_2 n) < O(n^2) < O(n^3) < O(2^n)$$

通过如图 5-6 所示的计算时间函数示意图, 能够更加直观的理解它们的相互关系。

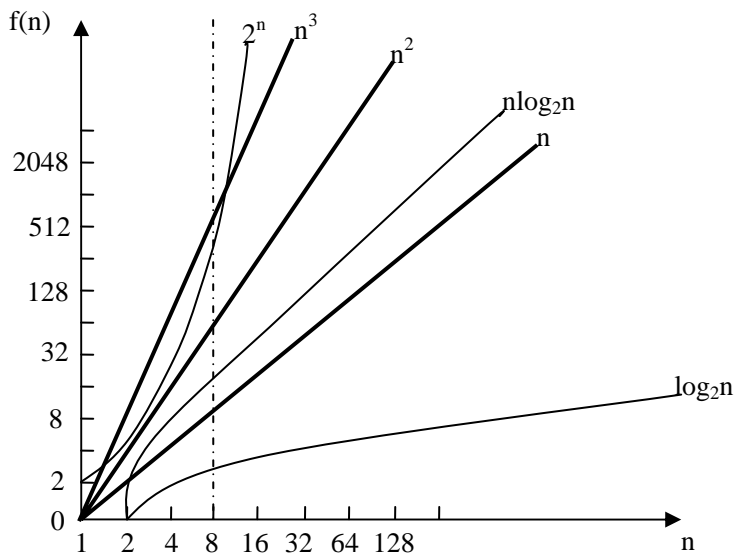


图 5-6 常用计算时间函数的曲线示意图

这些算法时间复杂性中， $O(2^n)$ 随着问题规模 n 的增长最快，如果能设计出解决同样问题的算法，而时间复杂性仅为 $O(n^2)$ 等，则就是大的飞跃，采用好算法与劣算法编制的程序，其运行效率是有显著差别的。

2. Fibonacci 问题

为了获得对时间复杂性的更为清晰的认识，我们将讨论一个很有趣的数列——Fibonacci 数列；同时，通过这个数列，引出两个非常重要的算法概念——递推与递归。

Fibonacci 数列是一个无穷数列，具体构成为：

0, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

它可以定义为：

$$\begin{cases} F(0) = F(1) = 1 \\ F(i) = F(i-1) + F(i-2) \quad i > 1 \end{cases}$$

即后面的每个 Fibonacci 值是前两个 Fibonacci 值之和。

该数列在自然界里经常发生，可以看作是一种形式特别的螺旋线。相邻的 Fibonacci 数的比率收敛于常数 1.618...，称为黄金率或黄金分割率。人们在生活和工作中，发现黄金分割率在美学上能获得赏心悦目的效果，所以建筑师常常将门窗、房间、建筑物的长宽比设计为黄金分割率；邮局的明信片长宽比也常被设计为黄金分割率。

现在的任务是如何求得 Fibonacci 数列中的第 n 个数。下面介绍两种求解方案：递推和递归。

(1) 递推与迭代

递推与迭代都是重要的数学方法，它们在计算机中的应用也非常广泛。递推关系是一种随着变量的递增产生连锁反映的关系。所谓递推法，就是通过迭代运算对问题进行求解的方法。

采用递推与迭代的方式实现 Fibonacci 数列的 Delphi 程序代码如下：

```
function FIB ( N : integer ) : integer;
```

```

begin
  F[1] := 0;           //递推边界 1
  F[2] := 1;         //递推边界 2
  For n := 3 to N do
    F[n] := F[n-1] + F[n-2]; //迭代
end;

```

从上面的例子中，我们可以得到递推算法的一般模式为：

- 1、确定初值 $v = v_0$;
- 2、当条件 $p(v)$ 成立，则重复执行 $v = f(v)$;
- 3、先求得初值 $v = v_0$ ，然后将每次计算的值 v 代入递推公式 $v = f(v)$ ，求得新的 v ，直到条件 $p(v)$ 不成立为止。

以递推的方式求解 Fibonacci 数列时，需要进行 $(N-3)$ 次迭代，其时间复杂度为 $O(n)$ 。

(2) 递归

递归在数学和计算机中经常遇到。利用递归方法可以用有限的语句来定义无限集合，但在递归定义中至少有一条是非递归的，即初始定义，否则就会产生逻辑性错误。

在计算机科学中，递归还经常应用于递归调用方面，即函数或者过程自己调用自己。实际上，递归是把一个不能或者不好直接求解的“大问题”转化成一个或者几个“小问题”来解决，在把这些“小问题”进一步分解成更小的“小问题”来解决，如此分解，直至每个“小问题”都可以直接解决（此时分解到递归出口）。但递归分解又不是随意地分解，递归分解要保证“大问题”与“小问题”相似，即求解过程与环境都相似。

递归按其调用方式可分为直接调用和间接调用。

采用递归的方式实现 Fibonacci 数列的 Delphi 程序代码如下：

```

function FIB ( N : integer ) : integer;
begin
  if N = 1 then FIB := 0           //递归边界 1
  else if N = 2 then FIB := 1     //递归边界 2
  else FIB := FIB(N - 1) + FIB(N - 2) //递归调用
end;

```

以递归方式求解 Fibonacci 数列时，是把它分解成求 $F(n-1)$ 与 $F(n-2)$ 两部分，而每一部分又再分为两部分，依此类推。在求解 $F(n-1)$ 时，又重复去求了一次 $F(n-2)$ ，……。也就是说，求 $F(n)$ 时，重复计算了 2 次 $F(n-2)$ ，重复计算了 4 次 $F(n-3)$ ，……。总的递归次数是 2 的 $n-1$ 次方，其时间复杂度为 $O(2^n)$ ，计算量是非常大的。

递归算法的效率往往很低，费时和费内存空间。但是递归调用也有其长处，它能使一个蕴含递归关系且结构复杂的程序简洁、精练，增加可读性。特别是难于找到从边界条件到解的全过程的情况下，采用递归算法编程比较合适。

在程序设计中，实际运行的程序并非都写成递归形式。有时先写一个递归程序，而后，为了使程序在反复执行时避免占用过多的机器时间，需要将递归程序转化为非递归的程序。实践证明，当可以将递归转化为递推方法来处理问题时，递推方法的效率比递归方法要高。

5.2 集合

集合类型是一群相同类型元素的组合，这些类型必须是有限的几种类型，如：整型、布尔型、字符型、枚举型和子界型。在检查一个值是否属于一个特定集合时，集合类型非常有用。集合类型的定义方法：

```
set of BaseType
```

例如：

```
type
```

```
TInt    = 0..100;           //BaseType 为序数类型（字界类型）
T1      = set of TInt;     //定义集合类型
TDate   = set of (Wed,Mon,Thu,Sun,Sat);
TChar   = set of ('a', 'b', 'c');
```

注意:虽然集合中的值没有固定的顺序,但同一个值在一个集合中最多出现一次(即集合中的元素应该是互不相同的)。

Delphi 中提供了几个用于集合的运算符,由于这些运算比较特殊,只是对集合类型使用,所以在这里提前介绍。集合可以利用这些运算符判断集合与集合之间的关系,对集合增删元素,或对集合进行求交集运算等。

5.2.1 关系运算

用 `in` 运算符来判断一个给定的元素是否在一个集合中,下面的代码判断在前面所定义的集合 `T1` 中是否有 `80`:

```
if (80 in T1) then DoSomething           //继续运行
```

下面的代码判断在 `TDate` 中是否有集合元素 `Mon`:

```
if not (Mon in TDate) then DoSomething   //继续运行
```

5.2.2 增删元素

用 `+`、`-` 运算符或 `Include` 和 `Exclude` 过程,可以对一个集合变量增删元素,例如:

```
var
  CharSet : TChar;
begin
  CharSet := ['a'];           //初始化集合变量 ;
  Include(T1, 101);          //在集合中增加 101 ;
  CharSet := CharSet + ['d']; //在集合中增加'd' ;
  Exclude(CharSet, 'a');     //在集合中删除'a' ;
```

```

    CharSet := CharSet-['a', 'b'];           //在集合中删除'a', ' b' ;
end;

```

技巧: 建议尽可能地用Include和Exclude来增删元素, 而少用+、-运算符。因为Include和Exclude仅需要一条机器指令。

5.2.3 交集运算

Delphi 中用*运算符来计算两个集合的交集, 表达式 Set1*Set2 的运算结果是产生出 Set1 和 Set2 这两个集合中都存在的元素, 下面的例子用来判断在一个给定的集合中是否有'a'、'b'、'c'这几个元素:

```

if ['a', 'b', 'c']*CharSet = ['a', 'b', 'c'] then
    DoSomething //继续程序

```

5.3 数组

数组用于表示相同类型的元素的有序集合, 这里所说的“相同类型”即数组的基类型, 可以是系统预定义类型, 也可以是用户自定义类型。数组中每个元素都有一个唯一的索引, 因此, 与集合不同, 同一数组中可以含有多个相同的值。根据数组的分配方式可将数组分为: 静态数组和动态数组。

5.3.1 静态数组

定义静态数组类型的语法为:

数组类型名称 = array [索引类型...] of 基类型

数组类型的定义中, 索引类型可以有多个, 但必须都是序数类型 (在实际应用中, 索引类型通常是整数子界)。而且它们的整个范围不能超过 2GB。从定义可以看出, 静态数组中元素的数量是有限的, 其大小是所有索引类型范围的乘积。根据索引类型的个数, 又可以将数组分为一维数组和多维数组。顾名思义, 一维数组就是指索引类型的个数只有一个的数组; 以同样的思路, 可以理解多维数组。

最简单的情况是一维数组, 它仅有一个单独的索引类型。例如:

```

type
    TMyArray = array[1..100] of Integer; //声明数组类型
var
    myArray : TMyArray; //定义数组变量

```

这里声明了一个叫做 TMyArray 的数组类型, 用于保存 100 个整型值。接着定义一个名为 myArray 的变量。对于该声明, myArray[8]表示数组 myArray 中第 8 个整数。如果创建一个

静态数组而没有向其元素赋值，那么仍然会给未使用的元素分配内存并且包含了随机数据。

上面的变量声明也可以简化为下面的形式：

```
var  
    myArray : array[1..100] of Integer;
```

多维数组可以看作其元素是数组的数组。例如：

```
type  
    TMyArray = array[1..10] of array[1..10] of Real;
```

它等价于：

```
type TMyArray = array[1..10, 1..10] of Real;
```

上面的两种声明，都代表一个存储 10×10 个实数值的数组。若接着声明一个类型为 TMyArray 的变量 myArray，那么可以就可以采用下面的方式访问数组中的元素：

```
myArray[2, 5]
```

或

```
myArray[2][5]
```

使用标准函数 **Low** 和 **High** 可分别返回数组中第一个索引类型其范围的最低和最高值。还可用标准函数 **Length** 返回数组中第一维的元素数量。

需要强调的是：数据类型跟变量是两个不同的概念，程序不能直接使用类型，而必须通过变量定义语句来定义一个该类型的变量使用，例如：

```
var  
    array1, array2 : TMyArray;
```

上例中定义了两个数组变量 array1 和 array2，它们的类型是 TMyArray，TMyArray 是前面已经定义过的数组类型。

Delphi 允许只用一个赋值语句，就能把 array2 中每一个元素的值相应赋值给 array1 中的元素，如下所示：

```
array1 := array2;
```

数组间的整体赋值只适用于同一个数组类型的数组，即使两个数组看上去完全一样，也不能整体赋值，例如：

```
var  
    array1, array2: array[1..10] of Integer;
```

```
var  
    array3: array[1..10] of Integer;
```

对于上述三个数组变量，`array1:=array2` 是合法的，但是 `array1:=array3` 就不合法了。因为这种定义方式，系统认为它们是两种不同类型，而产生类型相容的问题。但是，可以通过逐个元素的赋值，使得数组一样，这是因为它们的基类是赋值相容的。

如果这样声明：

```
type  
    TMyArray = array [1..10] of Integer;
```

然后：

```
var  
    array1,array2: TMyArray;  
  
var  
    array3: TMyArray;
```

则此时三个数组全部是一个类型（它们有相同的类型名称），所以可以全部的赋值相容，即可以整体赋值。

5.3.2 动态数组

动态数组可以使用不确定的数组长度，而在程序中动态地分配数组的存储空间，以便更灵活地使用数组的特性。定义一个动态数组的语法如下：

数组类型名称：array of 数据类型

这里可以看到，动态数组没有给定好的长度。取而代之的是，向动态数组赋值或把动态数组传递给 `SetLength` 过程时，动态数组的内存被重新分配。其语法如下：

```
SetLength(数组名, 数组元素个数)
```

例如，声明一个动态数组变量：

```
var  
    myArray : array of Real;
```

它是一个一维的实数动态数组。该声明并不为 `myArray` 变量分配内存。要为数组创建内存，则需要继续进行如下操作：

```
SetLength(myArray, 50);
```

它要求系统在内存中分配一个拥有 50 个实数类型元素的空间,数组的索引为从 0 到 49。动态数组总是以整数作为索引,而且索引总是以 0 开始。

动态数组变量是一个隐含的指针,要释放动态数组,可以把空指针 `nil` 赋给引用数组的变量,或者把变量作为参数传递给标准过程 `Finalize`。倘若对动态数组没有其它的引用,这两种方法都可以释放数组。长度为 0 的动态数组,其值为空指针 `nil`。

如果 `A` 和 `B` 是同一动态数组类型的变量,那么 `A:=B` 执行的操作是把 `A` 指向与 `B` 相同的数组(这里不需要在执行操作之前为 `A` 分配内存)。与静态数组不同,动态数组在数据写入之前不会被自动复制。例如,

```
var
  A, B: array of Integer;
begin
  SetLength(A, 1);
  A[0] := 1;
  B := A;
  B[0] := 2;
end;
```

执行这段代码后的结果是: `A[0]` 的值是 2。如果 `A` 和 `B` 是静态数组,那么此时 `A[0]` 仍为 1。

而且,在比较动态数组变量时,比较的是它们的引用,而不是数组本身的值。例如:

```
var
  A, B: array of Integer;
  equal: boolean;
begin
  SetLength(A, 1);
  SetLength(B, 1);
  A[0] := 2;
  B[0] := 2;
  equal := (A = B); // equal is false
  equal := (A[0] = B[0]); // equal is true
end;
```

执行这段代码时,表达式 `A = B` 返回 `False` (因为此时 `A` 和 `B` 引用的是两个不同的动态数组),而表达式 `A[0] = B[0]` 返回 `True`。

一旦一个动态数组被分配,就可以将其作为参数传递给标准函数 `Length`、`High`、`Low`。`Length` 返回数组中元素的个数,`High` 返回数组的最大索引,`Low` 返回 0。对于零长度的数组,`High` 返回 -1 (此时违背常规,即 `High < Low`)。

动态数组同样也可以是多维的,称之为多维动态数组。它的声明需要迭代使用 `array of ...` 结构。例如:

```
type
```

```
    TTwoMyArray = array of array of string;  
var  
    myArray: TTwoMyArray;
```

这里声明了一个二维动态数组。要初始化该数组，同样需要调用标准过程 `SetLength`，不同的是要使用两个整数参数。例如：

```
SetLength(myArray, 6, 8);
```

它分配了一个 6×8 的二维数组，相当于一个矩阵。`myArray[0,0]`表示数组中的一个元素。也可以创建非矩阵的动态数组。首先，调用 `SetLength`，将数组的第一个维数 `n` 作为参数。例如：

```
var  
    myArray: array of array of Integer;  
  
SetLength(myArray, 4);
```

这里为二维数组 `Ints` 分配了 4 行但没有任何列。然后，可以单独的对某一列分配行（各列可以指定不同的长度）。例如：

```
SetLength(myArray[1], 8);  
myArray [1, 3] := 100;
```

这里为 `myArray` 第 2 列指定了 8 个整数的长度。接着，执行下一条语句后，可以对第 2 行的第 4 个元素赋值。

5.3.3 排序

上面分别讲述了数组的有关概念，从上可以知道在数组中可以存放大量的数据，相当于一个小型的数据集。接下来将讲述数组的两个基本的操作：数组的排序和查找。这两个操作需要用到排序和查找算法。

由于数组中存有同一数据类型的数据，而有时为了更有效地使用数组，就需要对数组中的元素进行排序，使其按一定的顺序排列好，比如按关键字的值从小到大排序。前面我们讲过，排序算法有很多，如：冒泡排序、选择排序、插入排序、快速排序、合并排序、希尔 (Shell) 排序、堆排序等。在这里仅讨论一维数组的冒泡排序和快速排序这两个算法，以便重点掌握这两个排序算法的基本原理与实现方法。

1. 冒泡排序

冒泡法是最简单的排序方法。这种方法的基本思想是，将待排序的元素看作是竖着排列的“气泡”，较小的元素比较轻，从而要往上浮。在冒泡排序算法中我们要对这个“气泡”序列处理若干遍。所谓一遍处理，就是自底向上检查一遍这个序列，并时刻注意两个相邻的元素的顺序是否正确。如果发现两个相邻元素的顺序不对，即轻的元素在下面，就交换它们

的位置。显然，处理一遍之后，最轻的元素就浮到了最高位置；处理二遍之后，次轻的元素就浮到了次高位置。在作第二遍处理时，由于最高位置上的元素已是最轻元素，所以不必检查。一般地，第 i 遍处理时，不必检查第 i 高位置以上的元素，因为经过前面 $i-1$ 遍的处理，它们已正确地排好序。这样一直进行下去就可以实现对该组数据的排序。

说明：冒泡技术也称为下沉技术 (sinking sort technique)，因为大数在比较和交换的过程之后，沉到了数组的“底部”。

冒泡排序使用了双重循环，外层循环每次扫描过程中迭代一次；每次内层循环则将某一元素冒泡置顶部（左部）。我们把流程图和伪代码留给读者作为练习。这个算法的 Delphi 程序实现如示例程序 5-5 所示。

示例程序 5-5 冒泡排序算法的 Delphi 程序实现

```
var
    a:array[1..10] of integer=(2,5,3,8,7,6,9,10,49,25); //待排序的数组

// 冒泡排序过程
procedure BubbleSort(var a:atype);
var
    i, j, temp:integer;
begin
    for i:=low(a) to high(a)-1 do
        for j:=low(a) to high(a)-i do
            if a[j]<a[j+1] then
                begin
                    //交换两个数据
                    temp:=a[j];
                    a[j]:=a[j+1];
                    a[j+1]:=temp;
                end;
        end;
end.
```

上述算法将较小的元素看作较轻的气泡，每次最小的元素浮到表尾，实现了数据的由大到小的排序。其中 $\text{low}(a)$ 和 $\text{high}(a)$ 分别表示数组 a 的第一个元素和最后一个元素的位置。图 5-7 演示了冒泡排序第一轮比较和交换的过程。在该过程中，从前到后（图示为从左到右）依次扫描比较相邻的元素，通过位置交换，最终将 2 移到了最后位置上，显然这个数就是最小元素。

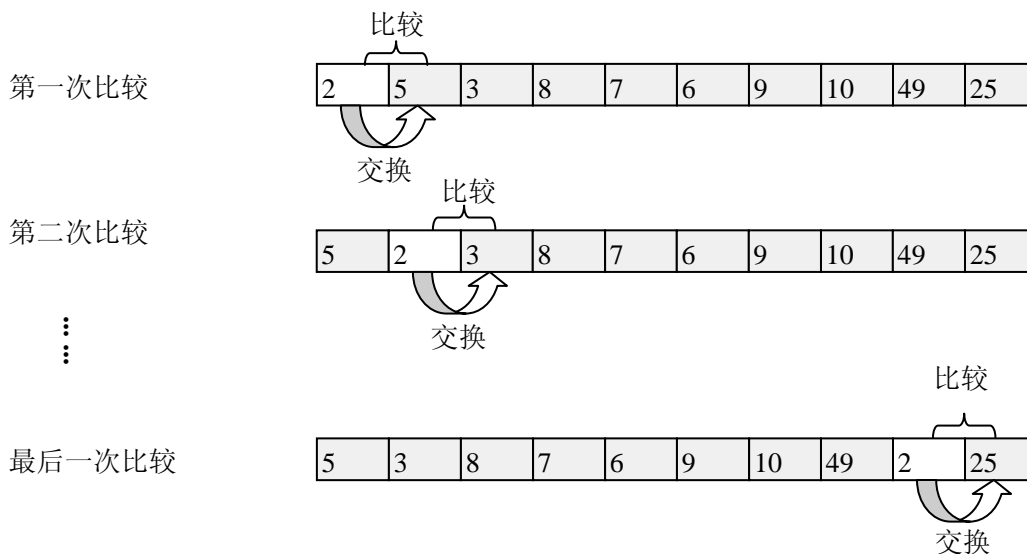


图 5-7 冒泡排序第一轮比较和交换的过程演示

程序运行后就可以实现对这 10 个数据的降序排列，从 $a[1]$ 到 $a[10]$ 其值依次为：49, 25, 10, 9, 8, 7, 6, 5, 3, 2。

2. 快速排序

由著名计算机科学家霍尔 (C.A.R.Hoare) 设计的快速排序是对冒泡排序的一种改进。它 (这次以升序排列为例) 的基本思想是：首先选取某个元素 u ，通过一趟排序将待排序的数组元素分割成独立的两部分，其中前一部分元素的均比 u 小，而后一部分元素均比 u 大，然后再分别对这两部分元素继续进行排序，被分成的两个部分以后不用再归并，最终达到整个数组序列有序。我们称这种重新整理叫做划分， u 称为划分元素。

假设待排序的是数组，例如 $A[s..t]$ ，我们可以首先选取第一个元素 $A[s]$ (这并非必需的，仅仅是为了方便) 作为划分元素，然后重新排列其余元素，将所有值比它小的元素都安置在它的位置之前，所有值比它大的元素都安置在它的位置之后。这样，以该划分元素所在的位置 i 作为分界线，将序列分割成两个子序列 $\{a[s], a[s+1], \dots, a[i-1]\}$ 和 $\{a[i+1], a[i+2], \dots, a[t]\}$ 。这个过程称作一趟快速排序 (或一次划分)。一趟快速排序的具体算法是：附设两个指针 i 和 j ，它们的初值分别为 s 和 t ，设支点元素为 $temp:=a[s]$ ，则首先从 j 所指位置向前搜索找到第一个关键字小于 $temp$ 的元素和 $a[i]$ 互换，然后从 i 所指位置起向后搜索，找到第一个值大于 $temp$ 的元素和 $a[j]$ 互换，重复这两个步骤，直到 $i=j$ 为止。

实际算法中，要考虑到交换的数据处在末端的情况。即当 $a[i]$ 的值在这组数中最大，此时 $a[i]$ 要放置到末端，而不需要和其它数据交换。为此，引入一个新的数据 $a[t+1]$ ，并假定 $a[i] < a[t+1]$ ，引入 $a[t+1]$ 的目的是为了在特殊的情况下，也能控制程序的顺利进行。

另外，在排序过程中对划分元素 $temp$ 的赋值是多余的，因为只有在一趟排序结束时， $i=j$ 的位置才是 $temp$ 的最后位置。由此可先将 $temp$ 暂存，排序过程中只作 $a[i]$ 或 $a[j]$ 的单向移动，直到一趟排序结束后再将 $temp$ 移至正确的位置上。

快速排序算法的 Delphi 实现在下一章的“结构化程序设计举例”一节中有介绍，详细代码如示例程序 6-2 所示。

5.3.4 查找

数组的查找就是从数组中找出需要的数据项，也叫检索。查找问题的一般提法是：设一数组有 n 个元素，每个元素由一个称为关键字的数据项和若干相关数据项的值组成，对给定值 K ，要查找出这 n 个数组元素中是否存在关键字等于 K 的某个元素。通常有两种结果，一种是能够检索到，即查找成功，此时查找的结果为给出整个元素的信息，或指出该元素所在的位置；另一种是找不到，即失败，此时查找的结果为不成功的信息。

查找的算法很多，有顺序查找、折半查找、散列值查找、转移表查找等。

1. 顺序查找

顺序查找方法很简单，就是将要查找的数据的关键字按一定的顺序挨个与数组中的数据进行比较，相等时就找到了所要的数据。示例程序 5-6 是最常见的实现代码。

示例程序 5-6 顺序查找

```
//在数组 a 中顺序查找值为 key 的元素。  
for i:=low(a) to high(a) do  
  if a[i]=key then  
    result:=i//若检索成功则返回该数组元素的位置（即索引）  
  else  
    result:=-1; //若检索失败则返回-1
```

这种方法使用起来很简单，但查找次数多，速度慢。比如说有一数组的数据项的值分别为：1, 2, 3, 4, 5, ..., 999, 1000 现要找出其中最大的元素 1000，如果是从前向后找，就要检索数组中的每一个元素，对它们进行分析比较，才能找到 1000。

2. 折半查找

若数据已按大小顺序排列好了，则采用折半查找（又称二分检索）方法可以有效地减少检索次数，大大提高检索效率。折半查找的方法是：设有一组已排序的数据序列，用序列的中间项与检索的关键字比较，若相等，则表示找到了要找的数据。若不相等，就进一步比较这两个数的大小，若中间项大于关键字，则下一次用序列的前半部的中间项与该关键字比较；否则，下一次用序列的后半部的中间项与该关键字比较。这样子每检索一次，就可以使检索区间缩小二分之一，故称为折半查找。如此一直进行下去，直至找到或确定数据序列中没有所要找的数据是为止。例如：已知一个已排好序的数组，其数据元素如下：

(2, 13, 19, 21, 37, 56, 64, 75, 80, 88, 98)

现要查找关键字为 21 的数据元素。首先将 21 与这 11 个数据中处于中间位置数据 56 进行比较，21 小，所以下一次就将查找范围变为 (2, 13, 19, 21, 37)；接下来再将 21 与该范围内处于中间位置的 19 进行比较，21 大，就将下一次的查找范围缩小为 (21, 37)；然后再对比这个范围内的数据 37，21 小，所以下一次查找的范围为 (21)；最后比较目标数和中间数，它们相等，所以找到了查找的目标，位置为 3。具体过程如图 5-8 所示。

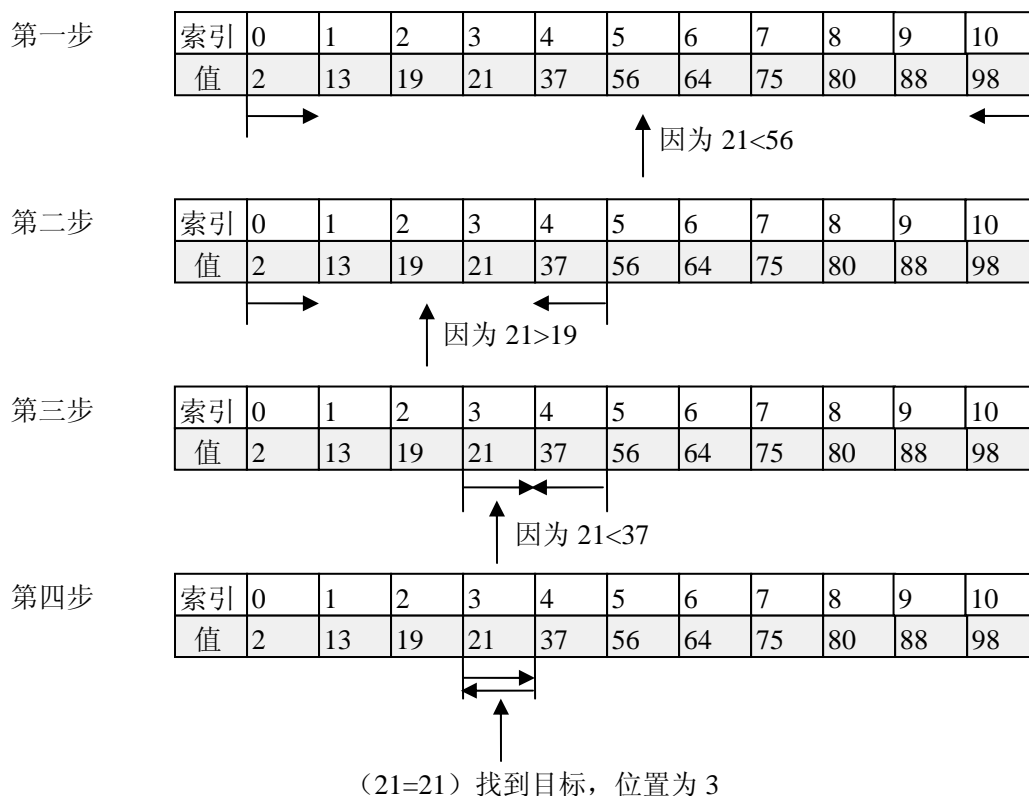


图 5-8 折半查找的过程演示

由此可知, 折半查找每查找一次就将查找范围缩小一半, 从而可以大大地减少了比较的次数, 提高了查找的效率。其 Delphi 算法实现代码如示例程序 5-7 所示。

示例程序 5-7 折半查找函数

```
function binsrch(a:ArrayType;k:integer):integer;
var
  l,h,mid:integer;
  found:boolean;
begin
  l:=low(a);
  h:=high(a);
  found:=false;
  while(l<=h) and (not found) do
  begin
    mid:=round((l+h)/2); //round 为取整函数
    if k>a[mid] then //关键字大于中间项
      l:=mid+1
    else
      if k=a[mid] then //关键字等于中间项
        found:=true
      else //关键字小于中间项
        h:=mid-1;
```

```
end;
if found then
    result:=mid //找到了则返回数组元素所在的位置
else
    result:=-1;//没有找到则返回-1
end;//while
end;
```

技巧：round函数是Delphi自带的函数。该函数总是返回实数最接近的整数，如果实数恰好是最近的两个整数的中间值，则返回这个整数其中的偶数。这种方法又称为“银行家取整法（Banker's Rounding）”。

注意：上述算法只适用于数组是从小到大排列的情况，如果给出的数组是从大到小排列的，则只需要将代码的判断条件（ $k > a[mid]$ ）改为（ $k < a[mid]$ ）即可。

5.3.5 数组参数

Delphi 语言实现了开放式数组。所谓开放式数组，是指使用数组作为形参传递给过程或函数时，其长度是不确定的。因此，在调用这个过程或函数时，可以传递不同长度的数组作为实参。对于开放式数组参数，又可分为如下两类：

- 开放式数组参数
- 变体开放式数组参数

1. 开放式数组参数

开放式数组参数允许不同大小的数组传递给相同的过程或函数。要定义含有开放式数组参数的例程，需要在参数声明中使用语法 `array of type`（优于 `array[X..Y] of type`）。开放式数组的定义和用法请参考下列示范程序：

```
//定义两个长度不同的数组变量
var
    X1:Array[1..10] of Real;
var
    X2:Array[1..20] of Real;

//过程的定义，其形参是开放式数组
procedure MyProce(X:Array of Real);
begin
    //过程体略
end;

//调用过程
begin
    MyProc(X1);
```

```
MyProc(X2);
end;
```

注意：实参的类型必须与形参数组的类型相同。

开放数组参数的语法类似于动态数组类型，但它们的用途完全不同。上面的例子创建的函数接受任何 **real** 类型元素组成的数组，包括（但不限于）动态实数数组。

在过程或函数体中，开放数组参数应遵从于下列规则：

- 总是零基准数组。即数组的下界总是 0，上界是实参中元素的个数减 1。标准函数 **Low** 和 **High** 分别返回 0 和 **Length** - 1。函数 **SizeOf** 返回传递给例程的实际数组的尺寸。如果实参仅仅是一个简单变量，就把它看作是只有一个元素的数组。
- 不允许对整个开放数组参数赋值。一般的数组可以作为整体赋值，但作为形参的开放式数组是不允许整体赋值的，而只能访问它的元素。
- 只能作为开放数组参数或无类型的变量参数被传递到其他过程或函数，而且，还不能传递到 **SetLength** 标准过程。
- 可以代替数组，传递一个开放数组参数基类型的变量，该变量将被视为长度是 1 数组。
- 当把一个数组作为开放数组的值参数传递时，编译器将在内存中开辟一块区域存放实参的副本，等过程或函数退出后再释放这块区域。

注意：开放式数组作为值参数时，传递值过大可能会出现堆栈溢出问题。而开放式数组作为常量参数或变量参数时，编译器传递的仅仅是一个引用，当然这也带来一个新的问题，如果在过程或函数中修改了参数的值，实际上就是修改了实参自身。

2. 变体开放式数组参数

与上述开放式数组参数略有不同的，变体开放式数组参数允许向单个过程或函数传递不同类型表达式的数组。要定义含有变体开放式数组参数的过程或函数，需要参数类型指定为 **array of const**。如：

```
procedure MyProc(Arr: array of const);
```

它声明了一个叫做 **MyProc** 的过程，该过程可以接受不同种类的数组参数。

实际上，**array of const** 结构等价于 **array of TVarRec**。**TVarRec** 的类型声明预定义在单元 **System** 中，用于表示一个记录，该记录中有一个可以保存多种值（整数、布尔、字符、实数、串、指针、类、类引用、接口、变体等）的变体部分。同时，**TVarRec** 中的 **VType** 字段表示数组中每个元素的类型。

下面的例子（摘自《Object Pascal Reference》）在函数中使用了变体开放式数组参数，该函数对接受的每个元素创建一个串表示法，最后连接成一个串。该函数中调用的串处理例程都定义在 **SysUtils** 单元中。

```
function MakeStr(const Args: array of const): string;
const
  BoolChars: array[Boolean] of Char = ('F', 'T');
var
  I: Integer;
begin
```

```

result := '';
for I := 0 to High(Args) do
  with Args[I] do
    case VType of
      vtInteger:      result := result + IntToStr(VInteger);
      vtBoolean:      result := result + BoolChars[VBoolean];
      vtChar:          result := result + VChar;
      vtExtended:     result := result + FloatToStr(VExtended^);
      vtString:        result := result + VString^;
      vtPChar:         result := result + VPChar;
      vtObject:        result := result + VObject.ClassName;
      vtClass:         result := result + VClass.ClassName;
      vtAnsiString:   result := result + string(VAnsiString);
      vtCurrency:      result := result + CurrToStr(VCurrency^);
      vtVariant:       result := result + string(VVariant^);
      vtInt64:         result := result + IntToStr(VInt64^);
    end;
  end;
end;

```

于是，在程序中就可以采用下面的方式来调用该函数：

```
MakeStr(['test', 100, ' ', True, 3.14159, TForm])
```

最后，调用函数将返回一个字符串：“test100 T3.14159TForm”。

注意：上面的代码中，部分类型使用了指针。这是因为一些类型是作为指针传递而不传递值；特别是长串，作为指针传递并且必需转换为string类型。

5.4 抽象数据类型*

抽象数据类型（Abstract Data Type，简称 ADT）就是与对该数据类型有意义的操作封装在一起的数据声明。将数据和操作封装起来并对用户隐藏。用户可通过操作接口对数据进行输入、存取、修改和删除等操作。用户使用抽象数据类型时不需要知道数据结构。这就是抽象数据类型的含义，其主要体现了一个抽象的概念。本节将重点讨论的抽象数据类型，包括：链表、栈、对列。

说明：有的教科书上将面向对象程序设计中的类也看作是一种抽象数据类型。本书将面向对象程序设计中的类、对象接口等类型的概念与抽象数据类型的概念分开讨论。

5.4.1 数据类型的层次结构

类型（type），指特定类别数据对象的集合（即通常出现在 Delphi 中作为 type 声明的部分）。对于任何一门语言而言，类型都是基础性的东西。通俗地讲，类型好比是用来解释存

储于某内存位置上的值的规则。

早期高级程序设计语言中类型概念的范围比较窄，只是指同一类数据对象的集合。这就是我们经常用到的基本数据类型。20 世纪 70 年代开始，数据类型的概念得到较大扩展，一种数据类型已不仅仅包括它所表征的全体数据对象，而且还包括在这些数据对象所能够进行的操作的集合。

对于基本数据类型，一般都内建于高级程序设计语言中，无需另外声明或定义。考虑到这些类型比较常用，对这些类型数据对象的操作比较简单和单一，所以 Delphi 提供了说明基本数据类型变量的语法以及在基本数据类型数据对象上的操作集合。

除了基本数据类型外，现代高级语言还提供了多种语法规则，使程序员能够定义自己的复杂数据类型，这些复杂数据类型是基本数据类型的组合或集合，如 Delphi 中的记录、数组等。

如果这些复杂的数据类型不但包括某类数据对象，同时还包括用户定义的、语言系统无法预知的操作集合。这种新的数据类型就已经不是传统意义上的类型，人们称它为抽象数据类型。抽象数据类型在说明形式和实现上采用封装的手段将数据对象集合与操作集合联系在一起。这些操作不单单是简单的加、减、乘、除等具体的数学运算，而是任何由程序员定义的以这些类型数据对象为参数的函数，这些函数都以某种方式使用了这些数据对象的属性（值），是一些抽象的操作。

在程序设计中，使用抽象数据类型与使用非抽象类型相比，具有明显的优点：

- 程序设计层次分明，将顶层设计与底层设计隔开，在进行顶层设计时，只需要考虑如何定义抽象类型，不必考虑它所用到的数据和运算分别如何表示和实现；在进行数据表示和运算实现等底层设计时，只要抽象数据类型本身定义清楚，不必考虑其应用场合。这样程序设计的复杂性降低了，条理性增强了，有助于迅速开发出程序的原型。
- 由于程序的算法设计与所使用的数据结构被分隔开，允许自由选择数据结构，便于优化算法和提高程序运行效率。
- 数据模型及其上的运算统一在抽象数据类型中，反映了二者之间的内在联系，便于程序设计时在空间和时间复杂度之间进行折衷。
- 由于顶层设计和底层设计被局部化，在设计中，如果出现差错，将是局部的，因而容易定位和纠正错误。在设计中常常要做的增、删、改也都是局部的，因而也都容易进行。因此可以肯定，用抽象数据类型表述的程序具有很好的可维护性。
- 程序风格呈现十分明显的模块化，程序结构清晰，层次分明，而且由于抽象数据类型提供了对数据和运算的封装，使得移植和重用更加方便。

抽象数据类型所具有的这些优点读者会在后面的具体介绍和应用剖析中进一步体会到。

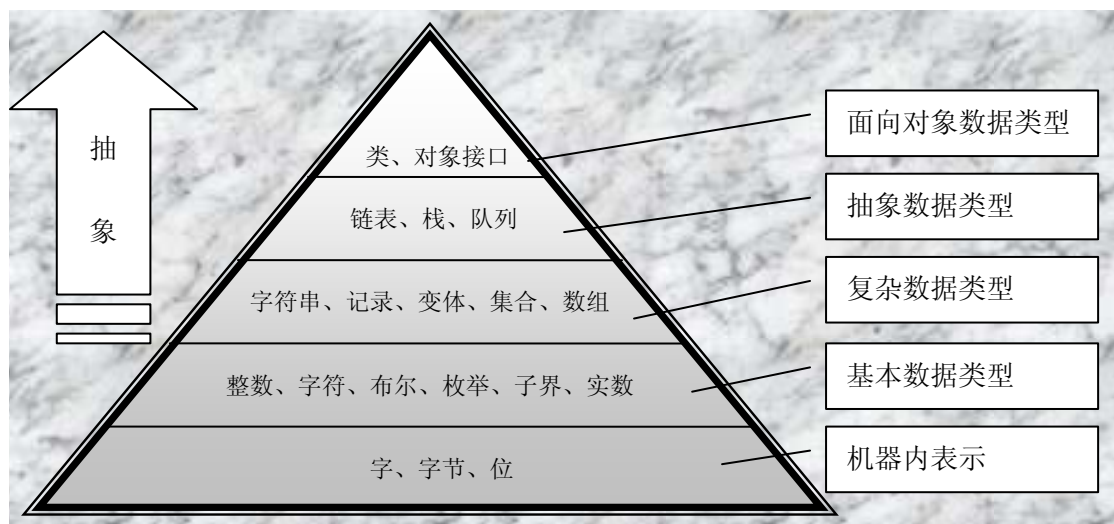


图 5-9 Delphi 数据类型的层次结构

抽象数据类型的发展还促进了面向对象程序设计的兴起，并产生了面向对象程序设计更抽象的类(class)、接口 (Interface) 等面向对象数据类型。这些类型通过继承，还可以派生出用户需要的新类型，这使得新类型的创建更加方便。

面向对象数据类型大大提高了数据类型的表现能力，特别适合于按照人类对客观事物认识的发展规律进行大型程序的开发和设计。在后续与面向对象程序设计相关的章节中我们再详细介绍。

图 5-9 显示了呈金字塔形状的 Delphi 数据类型的层次结构。数据类型通过从底层向顶层不断抽象，其结构更趋复杂，功能也不断增强。

5.4.2 链表

抽象数据类型的一个实例就是线性列表。线性列表是一种具有顺序结构的列表，在该列表中每个元素都有唯一的后继元素。线性列表可以通过链表来实现。

1. 几种链表类型

(1) 单链表

链表是一组元素的序列，在这个序列中每个元素总是与它前面的元素相链接（除第一个元素外），从而形成单链表。单链表关系的实现可以通过指针来描述。图 5-10 就是一个链表的示意图。

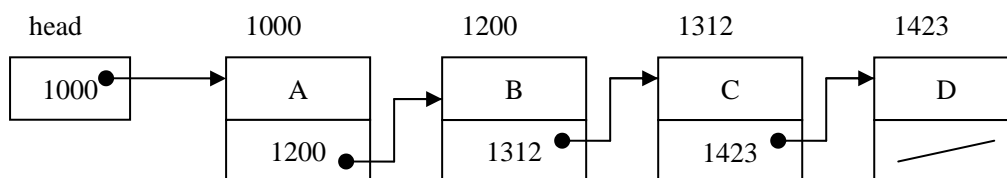


图 5-10 链表示意图

链表中的元素也称为结点，第一个结点称为表头，最后一个结点称为表尾。指向表头的

指针称为头指针，在这个指针变量中存放着表头的地址。结点用记录描述，至少包含两个域，一个域用来存放数据，其类型根据存放数据的类型而定，称为值域；另一个域用来存放下一个结点的地址，称为指针域。表尾不指向任何结点，设想有一空指针 NIL（空指针在链表图中常用“/”表示）。对单链表的操作有创建、插入、删除等。

(2) 循环链表

将单链表的形式稍加改动，让表中最后一个结点的指针指向单链表的表头结点，这样就形成了一个循环链表，如图 5-11 所示。

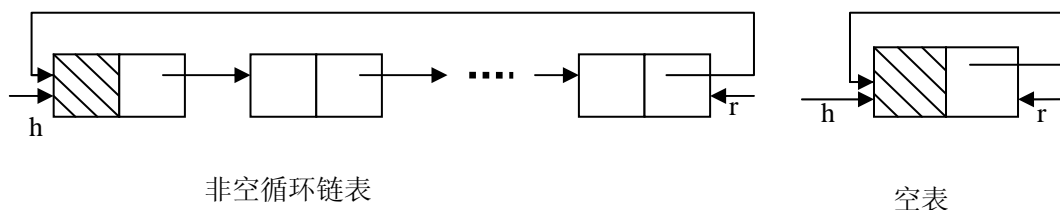


图 5-11 循环链表的示意图

使用循环链表的主要优点是，从表中任一结点均可找到表中其它的结点。

(3) 双向链表

在单链表中，从任何一个结点能通过指针域找到它的后继结点，但无法找到它的前趋结点，而双向链表则正是弥补了单链表的这个不足。在双向链表中的每一个结点除了数据字段外，还包含两个指针，一个指针指向该结点的后继结点，另一个指针指向它的前趋结点；双向链表有两个好处：一是可以从两个方向搜索某个结点；二是提高了链表的可靠性，因为无论利用向前这一链还是向后这一链，都可以遍历整个链表，如果有一根链失效，还可以利用另一根链修复整个链表。

2. 链表的应用

链表在编程中的应用主要在于管理大量的字符串、数值、对象等，就其操作的便利性而言往往可以取代动态数组（数组不可取代的优势在于维数的扩展）。在 Delphi 的 VCL 中，有一些现成的可以当作链表使用的抽象数据类型，如：TList、TStrings 和 TCollection。TList 类主要用于管理引用集合，因为它内部是管理一个指针数组；TStrings 的派生类主要用于管理字符串集合和对象集合，如 TStringList；而 TCollection 类主要用于管理由 TCollectionItem 派生类的集合。

TList 是 Delphi 中最基本的链表，它能实现链表常用的操作，包括：增加、删除、定位、查找、遍历等，这足够解决一切关于有序列表的问题。示例程序 5-8 是一个 TList 的使用演示程序，演示了链表增加、删除、查找、遍历等操作。

由于 TList 所存储的是指针，并且函数的参数也是声明为 Pointer 数据类型，因此 TList 用于管理其他数据类型时需要在操作上将其转换为指针。例如示例程序 5-8 中的 MyList 链表操作的就是记录类型的指针变量 ARecord。该程序运行结果如图 5-12 所示。

当删除链表的一个元素（节点）时，该元素后续的各元素索引值将发生变化（例如：被删除的元素索引为 6，其后续元素的索引将由原来的 7 改变为 6，各后续元素依次发生这种 $index:=index-1$ 的变化，元素总数减少），因此遍历链表过程中有删除操作发生时，应该采用逆序遍历，否则会出现超界错误。

链表对象 MyList 使用时需要用 TList.Create; 语句创建，使用完毕后需要用 MyList.free;

语句释放。try-finally-end 语句块保证了即使该语句块中出现异常，链表对象 MyList 也将被释放。

说明：对象的创建和释放将在后面的面向对象程序设计有关章节中详细介绍。

示例程序 5-8 TList 的使用演示程序

```
program ListExample;

{$APPTYPE CONSOLE}

uses
  SysUtils, Classes;

type
  PMyList = ^AList;
  AList = record
    age: Integer;
    name: string;
  end;

var
  MyList: TList;
  ARecord: PMyList;
  B,C: Byte;

//遍历显示链表的子过程
procedure iterator;
var
  age: Integer;
  name: string;
begin
  writeln('遍历显示链表: ');
  writeln('-----');
  for B := 0 to (MyList.Count - 1) do
  begin
    ARecord:= MyList.Items[B];
    age:=ARecord^.age;
    name:=ARecord^.name;
    writeln(name+':'+IntToStr(age)+'岁; '); {显示}
  end;
end;

begin
  MyList := TList.Create;
  try
```

```
New(ARecord);
ARecord^.age := 5;
ARecord^.name := '司马光';
MyList.Add(ARecord); //增加操作
New(ARecord);
ARecord^.age := 10;
ARecord^.name := '孔融';
MyList.Add(ARecord);
New(ARecord);
ARecord^.age := 15;
ARecord^.name := '周瑜';
MyList.Add(ARecord);
New(ARecord);
ARecord^.age := 3;
ARecord^.name := '曹冲';
MyList.Add(ARecord);
iterator; //遍历显示链表
writeln('【删除 10 岁以下的记录.....】');
C:=MyList.Count - 1;
for B := C downto 0 do
begin
    ARecord := MyList.Items[B];
    if ARecord^.age<10 then
        MyList.Delete(B); //删除操作
    end;
    iterator; //遍历显示链表
finally
    MyList.Free;
end;
readln;
end.
```

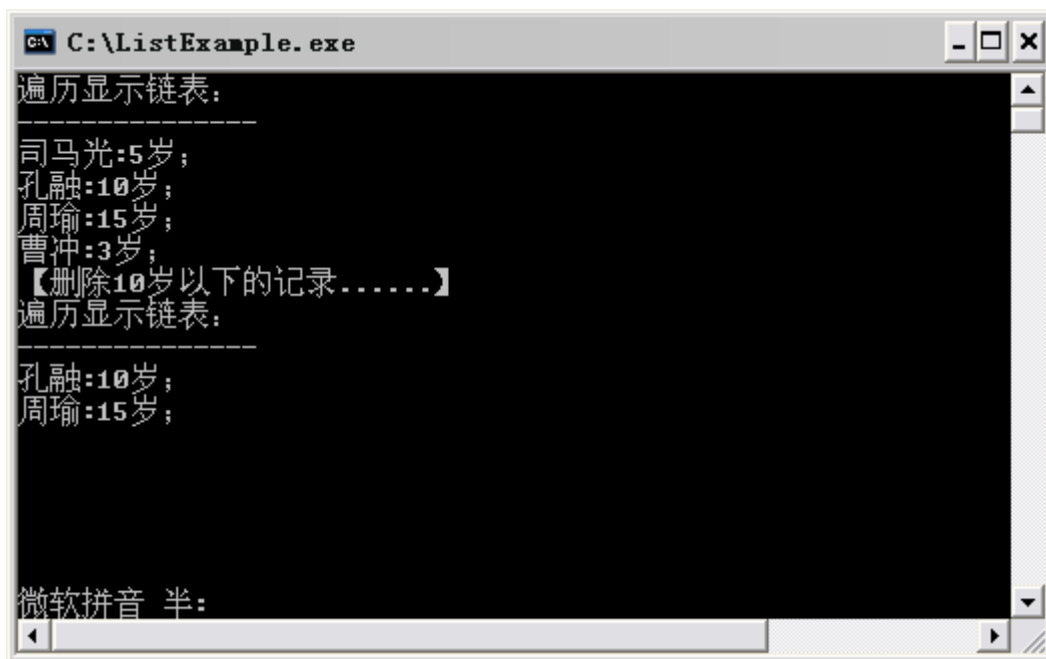


图 5-12 TList 演示程序的运行结果

在 Delphi 面向对象程序设计中，TStrings 类是众多列表型组件的基础。例如 TListBox 组件的 Items 属性、TMemo 组件的 Lines 属性都属于 TStrings 类。TStrings 是一个抽象类，它不能创建任何实例，但使用其派生类 TStringList 可创建字符串列表对象，还可把它赋值给任何 Delphi 组件对象的 TStrings 属性。TStringList 为存储字符串而分配内存空间。当用户需要位置来存储字符串列表时，建议使用这种类型来创建独立的变量。

TStringList 对象一个最常用的用途是读写字符串。示例程序 5-9 的代码介绍了怎样创建一个 String List，如何将 TStringList 存储的字符串列表保存在一个文本文件中：

示例程序 5-9 TStringList 的使用演示程序

```
var
    SL:TStringList;
begin
    SL:=TStringList.Create;
    try
        SL.Add('这是第一行');
        SL.Add('这是第二行');
        SL.Add('结束');
        SL.SaveToFile('Anyname.txt');
    finally
        SL.Free;
    end;
end;
```

示例程序 5-9 首先创建一个 TStringList 类型的字符串链表对象 SL，然后在 SL 链表中添加一些字符串，调用 SaveToFile 方法把这些字符串写到名为 Anyname.txt 的文本文件中。最后调用 Free 过程从内存中释放创建的列表对象。try-finally-end 语句块保证了即使该语句

块中出现异常，SL 也将被释放。

程序运行后，在当前目录下的 Anyname.txt 文件中有如下内容：

这是第一行

这是第二行

结束

5.4.3 栈

栈是限定仅在一端进行插入或删除操作的线性表。对于栈来说，允许进行插入或删除操作的一端称为栈顶 (top)，而另一端称为栈底 (bottom)。假设有一个栈 $S = (a_1, a_2, \dots, a_n)$ ， a_1 先进栈， a_n 最后进栈。出栈时只能在栈顶进行，所以 a_n 先出栈， a_1 最后出栈。因此又称栈为后进先出 (Last In First Out, 简称 LIFO) 线性表。

栈有两种存储结构即顺序存储结构和链式存储结构。栈的顺序存储结构是利用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素，同时设指针 top 指示栈顶元素的当前位置。空栈的栈顶指针值为零。栈的链式存储结构由其栈顶的指针唯一确定。

1. 栈的基本操作

栈有许多操作，基本操作有入栈和出栈。

- **入栈 (push)** 入栈是在栈顶添加新的元素。入栈后，新元素成为栈顶元素。入栈操作要注意栈的空间足够大，以防止栈内没有空间来容纳新元素，否则栈将处于溢出状态，不能添加元素。
- **出栈 (pop)** 出栈是将栈顶的元素移走并返回给用户。当最后一个元素被删除后，栈必须设为空状态。当栈为空的时候调用出栈操作，栈将处于下溢状态。

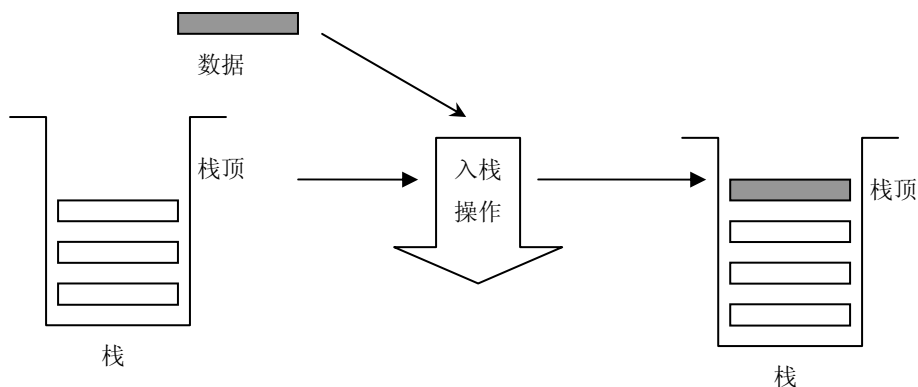


图 5-13 入栈操作

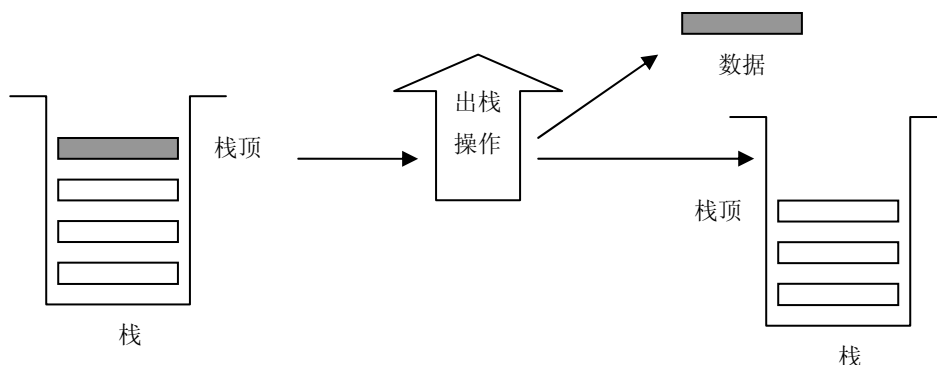


图 5-14 出栈操作

2. 栈的应用

栈的应用非常广泛，在这里只介绍一个倒转数据的简单应用。

倒转数据是对一组给定的数据，对其中的数据元素重新排放位置，使得意向性元素互换，中间所有元素也相应地进行交换。例如：数组 $A[4]$ 的元素值为 1, 2, 3, 4，现在要将它变成 4, 3, 2, 1。具体操作可以从 $A[0]$ 到 $A[3]$ 一个接一个的入栈，再一个一个地出栈，按 $A[0]$ 到 $A[3]$ 的顺序重新存放赋值，这样就实现了对数组元素的倒转。

在 Delphi 的 VCL 中，有现成的 `TStack` 抽象数据类型（即 `TStack` 类，类和面向对象的概念后面的章节将详细讲解）可以使用。该数据类型提供了与其封装在一起的入栈和出栈函数 `push` 和 `pop`（即 `TStack` 的 `push` 和 `pop` 方法）。为了使用该数据类型，还需要在程序中引用（uses）`Contnrs` 单元。

示例程序 5-10 是我们设计的一个简单的倒转数据应用程序。该程序中创建了一个 `TStack` 抽象数据类型的栈对象 `s`，并通过 `s.push` 和 `s.pop` 函数实现入栈和出栈。注意，这里入栈和出栈的是地址指针，即对数组元素值的一个引用。所以需要事先声明一个字符串类型的指针变量 `ps`：

```
var
  ps:^String;
```

将数组元素值的引用赋给指针变量，需要在其前面使用 `@`，即：

```
ps:=@a[i];
```

将指针变量所引用的值赋给数组元素，需要在其后使用 `^`，即：

```
b[i]:=ps^
```

调试好的程序运行结果如图 5-15 所示。

示例程序 5-10 一个简单的倒转数据应用程序

```
program stack;
```

```
{$APPTYPE CONSOLE}

uses
  SysUtils,
  Contnrs;

var
  s:TStack;
  a,b:array[0..3] of String;
  i:integer;
  ps:^String;
  tmp1,tmp2:string;

begin
  { 初始化 }
  i:=0;
  s:=TStack.Create; // 创建栈对象
  try
    while i<=high(a) do
    begin
      writeln('输入入栈的内容: ');
      readln(tmp1);
      a[i]:=tmp1;
      ps:=@a[i];
      s.push(ps); // 入栈
      inc(i);
    end;
    writeln('输入完毕! ');
    i:=0;
    while s.Count>0 do
    begin
      ps:=s.pop; // 出栈
      b[i]:=ps^ ;
      writeln('出栈: '+b[i]);
      inc(i);
    end;
  finally
    s.Free;
  end;

  //以下程序显示比较倒转数据后的效果
  i:=0;
  tmp1:='';
  tmp2:='';
```

```

while i<=high(a) do
begin
    tmp1:=tmp1+' '+a[i];
    tmp2:=tmp2+' '+b[i];
    inc(i);
end;
writeln(tmp1);
writeln(tmp2);
readln;

end.

```

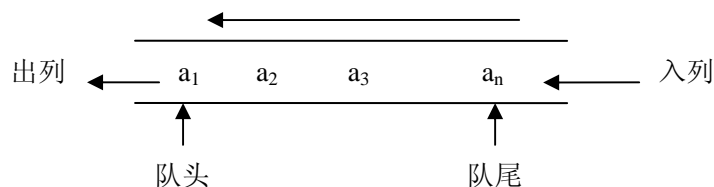


图 5-15 一个简单的倒转数据应用程序运行结果

回溯也是一种栈的重要应用。回溯，即回到（或恢复到）前面的数据，在计算机游戏、决策分析和专家系统等应用程序中经常见到。有关用栈实现回溯的 Delphi 应用实例，有兴趣的读者可以参见《Delphi 模式编程》（刘艺著，机械工业出版社 2004 年 9 月出版）的第 392 页，该示例讨论了一个使用备忘录模式的地理信息系统如何回溯到先前的标注状态。

5.4.4 队列

队列也是线性列表的一种特殊情况，其所有的插入均限定在表的一端进行，而所有的删除则限定在表的另一端进行。允许插入的一端称队尾，允许删除的一端称队头。队列的结构特点是先进队列的元素先出队列。因此，通常把队列叫做先进先出（First In First Out，简称 FIFO）线性表。



队列是最常用的抽象数据类型之一，事实上，在所有的操作系统以及网络中都有队列的身影，在其它技术领域更是数不胜数。例如处理用户请求、任务和指令。在计算机系统中，需要用队列来完成对作业或对系统设备如打印池的处理。

同样，在 Delphi 的 VCL 中，有现成的 TQueue 抽象数据类型（即 TQueue 类，类和面向对象的概念后面的章节将详细讲解）可以使用。该数据类型提供了与其封装在一起的入列和出列函数 push 和 pop（即 TQueue 的 push 和 pop 方法）。为了使用该数据类型，别忘了在程序中引用 Contnrs 单元。

5.5 本章小结

- 算法是为了解决某一问题在有限步骤内、定义了具体操作序列的规则集合。一个算法应该具有的五个重要特征是：确切性、输入、输出、可行性、有穷性。
- 算法不依赖于具体的编程语言，它可以用伪代码和图形这两种方式来描述。
- 伪代码是一种算法描述语言。使用伪代码的目的是为了使被描述的算法可以容易地以任何一种编程语言实现。因此，伪代码必须结构清晰，代码简单，可读性好，并且类似自然语言。
- 程序设计中，能够用来表示算法的图主要有：PAD 图、N/S 盒图、流程图。PAD 的目的在于以图表现程序的逻辑结构，使程序易读、易记、易理解。N/S 图是一种不需要有向线段，无需上下左右前后追踪程序流程控制的程序流程图，该图非常适合描述结构化程序或者算法的结构化实现，能够较好地反映算法和程序的层次结构，具有自顶向下逐步求精的特征。流程图则使用大图的形式掩盖了算法所有的细节方面，它只显示算法从开始到结束的整个流程。适用于设计一个完整的程序或者部分程序。
- 常用算法包括基本算法、排序算法、查找算法、迭代和递归算法等。
- 基本算法大都比较简单，是其他算法的基础。这类算法在程序中应用非常普遍，如：累加求和、累乘求积、求最大和最小值等。
- 排序算法根据数据的值对它们进行排列。排序是为了把不规则的信息进行整理，以提高查找效率。常用的排序方法包括：选择排序、冒泡排序、插入排序、快速排序、合并排序、希尔排序、堆排序等。
- 查找是一种在列表中确定目标所在位置的算法。基本的查找方法有顺序查找和折半查找。
- 迭代和递归是用于编写解决问题的算法的两种途径。迭代就是反复替换的意思，它通过使用一个中间变量保存中间结果，不断反复计算求解最终值。递归是一个算法自我调用的过程，用递归调用的算法就是递归算法。
- 算法的复杂性是指在执行时，算法所需要计算机资源的量。需要的时间资源的量称作时间复杂性，需要的空间（即存储器）资源的量称作空间复杂性。这个量集中反映了算法中所采用的方法的效率，它应该是只依赖于算法要解的问题的规模、算法的输入和算法本身的函数。

- 集合类型是一群相同类型元素的组合，这些类型必须是有限类型。集合类型的定义是 `set of BaseType`，集合的操作包括：关系运算、增删元素、交集运算。
- 数组用于表示相同类型的元素的有序集合，数组中每个元素都有一个唯一的索引。根据数组的分配方式可将数组分为：静态数组和动态数组。其中，动态数组可以使用不确定的数组长度，而在程序中动态地分配数组的存储空间。
- 使用标准函数 `Low` 和 `High` 可分别返回数组中第一个索引类型其范围的最低和最高值。还可用标准函数 `Length` 返回数组中第一维的元素数量。
- 冒泡法的基本思想是，将待排序的元素看作是竖着排列的“气泡”，较小的元素比较轻，从而要往上浮。一个含有 n 个元素的列表，冒泡排序需要 $n-1$ 次扫描来完成数据排序。
- 快速排序是对冒泡排序的一种改进。它的基本思想是，通过一轮排序将待排序的数组元素分割成独立的两部分，其中一部分元素的关键字均比另一部分元素的关键字小，则分别可对这两部分元素继续进行排序，以达到整个数组序列有序。
- 顺序查找就是将要查找的数据的关键字按一定的顺序挨个与列表中的数据进行比较，相等时就找到了所要的数据。
- 对有序的列表可以使用更有效率的折半查找。折半查找是从一个列表的中间的元素来测试的，这将能够判别出目标在列表里的前半部还是后半部分。如果在前半部分，就不需要查找后半部分。如果在后半部分，就不需要查找前半部分。换句话说，可以通过判断排除一半的列表。重复这个过程直到找到目标或是目标不在这个列表里。
- 开放式数组是指使用数组作为形参传递给过程或函数时，其长度是不确定的。因此，在调用这个过程或函数时，可以传递不同长度的数组作为实参。
- 抽象数据类型（ADT）就是与对该数据类型有意义的操作封装在一起的数据声明。它将数据和操作封装起来，用户可通过操作接口对数据进行操作。常用的抽象数据类型有链表、栈、队列等。
- 链表是一组互相链接的元素序列，分为：单链表、循环链表、双向链表等。若在这个序列中每个元素总是与它前面的元素相链接（除第一个元素外），则形成单链表。单链表关系的实现可以通过指针来描述。
- 在 Delphi 的 VCL 中，有一些现成的可以当作链表使用的抽象数据类型，如：`TList`、`TStrings` 和 `TCollection`。其中，`TList` 是 Delphi 中最基本的链表，它能实现链表常用的操作，包括：增加、删除、定位、查找、遍历等，这足够解决一切关于有序列表的问题。
- 栈是限定仅在一端进行插入或删除操作的线性表。栈又称为后进先出（LIFO）线性表。
- 栈的应用非常广泛，如：倒转数据、回溯等。在 Delphi 的 VCL 中，有现成的 `TStack` 抽象数据类型可以使用。该数据类型提供了与其封装在一起的入栈和出栈函数 `push` 和 `pop`。为了使用该数据类型，需要在程序中引用 `Contnrs` 单元。
- 队列是线性列表的一种特殊情况，其所有的插入均限定在表的一端进行，而所有的删除则限定在表的另一端进行。队列的结构特点是先进队列的元素先出队列。队列又称为先进先出（FIFO）线性表。
- 队列主要应用于排队处理用户请求、任务和指令。在计算机系统中，需要用队列来完成对作业或对系统设备如打印池的处理。同样，在 Delphi 的 VCL 中，有现成的 `TQueue` 抽象数据类型可以使用。该数据类型提供了与其封装在一起的入队和出队函数 `push` 和 `pop`。

5.6 本章习题

复习题

1. 什么是算法？算法有哪些特征？
2. 常用算法包括那些？
3. 什么是排序算法？排序算法有哪些？
4. 什么是查找算法？查找算法有哪些？
5. 什么是抽象数据类型？常见的抽象数据类型有哪些？
6. 什么是数组？Delphi 中数组可分为哪两类？他们是如何使用的？

测试题

7. 以下方式不可以用来表示算法的是____。
A、伪代码
B、流程图
C、Pascal 语言
D、PAD 图
8. 累加求和是____
A、基本算法
B、排序算法
C、查找算法
D、迭代和递归算法
9. ____是用于编写解决问题的算法的两种途径。
A、求最大值和最小值
B、函数和过程
C、查找和排序
D、迭代和递归
10. 集合的操作包括：____
A、关系运算、增删元素、交集运算
B、关系运算、逻辑元素、交集运算
C、关系运算、算术元素、交集运算
D、关系运算、赋值元素、交集运算
11. 使用标准函数____可返回一维数组的最小索引值。
A、Index
B、Low
C、High
D、First

12. 对一组数据 (84, 47, 25, 15, 21) 排序, 数据的排列次序在排序的过程中变化如为:

(1) 84 47 25 15 21

(2) 15 47 25 84 21

(3) 15 21 25 84 47

(4) 15 21 25 47 84

则采用的排序方法是_____。

- A、选择排序
- B、冒泡排序
- C、快速排序
- D、插入排序

13. Delphi 的 VCL 中, 有一些现成的可以当作链表使用的抽象数据类型, 但不包括: _____。

- A、TList
- B、TStrings
- C、TListBox
- D、TCollection

14. Delphi 的 VCL 中, 可以当作栈使用的抽象数据类型是_____。

- A、TStack
- B、TQueue
- C、TObjectList
- D、TControl

15. 以下说法不正确的是_____。

- A、开放式数组是指使用数组作为形参传递给过程或函数时, 其长度是不确定的。因此, 在调用这个过程或函数时, 无法传递不同长度的数组作为实参。
- B、TQueue 抽象数据类型提供了与其封装在一起的入列和出列函数 push 和 pop。
- C、链表常用的操作, 包括: 增加、删除、定位、查找、遍历等。
- D、对于没有排序的数列无法使用折半查找。

16. 以下数组类型声明不正确的是_____。

A、

type

```
TMyArray = array[1..10] of array[1..10] of Real;
```

B、

type

```
TMyArray = array[1..10] [1..10] of Real;
```

C、

type

```
TMyArray = array[1..10, 1..10] of Real;
```

D、

```

type
  TMyArray = array of array of Real;

```

练习题

17. 下列集合操作的结果各是什么？
- `[100,102,104,105,106]+[100,101,102,103,104]`
 - `[one,tow,three,four]-[five,four,six]`
 - `['a','b','c','D','E','F']*['A','B','c','d']`

18. 画出冒泡排序的流程图，并给出伪代码。

19. 将下列程序改写为非递归形式。

```

function exam(x:integer):integer;
begin
  if x=0 then exam:=1
  else
    exam:=x*exam(x-1)
end;

```

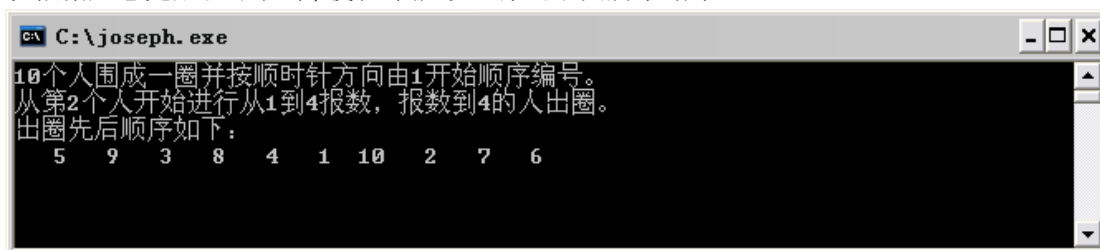
20. 若一本书厚 250 页，每页 20 行，每行 40 个字符。假设全书的内容已经存入了三维数组变量 `book` 中，其下标为 `page`、`line` 和 `column`。要求编程序在屏幕上输出任意指定的连续几页上的内容（即从某起始页 `startp` 显示至某结束页 `endp`）。

21. 设有 n 个人围成一圈并按顺时针方向由 $1\sim n$ 编号。从第 s 个人开始进行从 1 到 m 报数，报数到第 m 个人，此人出圈，再从他的下一个人重新开始 1 到 m 报数，如此进行下去，直到所有的人都出圈为止。

解决此问题的方法是：

- 将 $1\sim n$ 个人的序号存入一维数组 `p` 中；
- 按照出圈先后顺序重新排列数组 `p`，第 1 个报数出圈的人置于数组最后，而最后报数出圈的人置于数组最前；
- 按出圈先后顺序显示这些人的序号。

现根据题意完成以下程序，使程序能够运行出下图所示结果：



```

program joseph;

{$APPTYPE CONSOLE}

```

```

uses

```

```

SysUtils;

const
  n=10;
  s=2;
  m=4;
var
  p:array[1..n] of integer;
  i,j,k,m1,s1:integer;
  w:_____ ;

begin
  m1:=m;//1 到 m 报数
  s1:=s;//第 s 个人开始进行报数
  for i:=1 to n do
    _____ ;
  for i:=n downto 1 do
  begin
    s1:=_____ ;
    if s1=0 then _____ ;
    w:=p[s1];
    for j:=s1 to ____ do p[j]:=p[j+1];
    p[i]:=w;
  end;
  writeln(inttostr(n)+'个人围成一圈并按顺时针方向由 1 开始顺序编号。');
  writeln('从第'+inttostr(s)+'个人开始进行从 1 到'
    +inttostr(m)+'报数,报数到'+inttostr(m)+'的人出圈。');
  writeln('出圈先后顺序如下: ');
  for _____ do
    write(p[i]:4);
  readln;
end.

```

22. 给定一串整数数列, 求出所有递增和递减子序列的数目, 如数列 7、2、6、9、8、3、5、2、1, 则可以分为 (7、2), (2、6、9), (9、8、3), (3、5), (5、2、1) 五个子序列, 答案就是 5。现在要求允许用户输入 10 个整数, 然后进行处理, 最后输出答案, 请编程实现。

第6章 程序结构与结构化设计

通过结构可以了解一个实体的很多基本信息，Delphi 程序也是有结构的。通过研究 Delphi 程序的结构组成，可以帮助我们掌握 Delphi 应用程序及其组成模块的编写规则，理解源代码文件的组织方法。

Delphi 程序的结构组成还体现了结构化设计的风格以及代码封装和重用的原则。本章中，在了解了 Delphi 程序的结构之后，我们还要学习经典的结构化程序设计知识，进一步掌握结构化、模块化和自顶向下逐步求精的设计方法。最后，再通过一个具体的实例演示结构化程序设计的完整过程，并深入剖析 Delphi 应用程序的内部构造和代码实现，将前面所学的知识作一个回顾和总结。

6.1 Delphi 程序结构分析

一个 Delphi 应用程序可由多个源代码模块组成。这样可以把一个大型程序分成多个逻辑相关的模块，并用来创建在不同程序中使用的程序库。Delphi 应用程序通常由称为 Program 和 Unit 的两类不同的源代码模块组成。每个 Delphi 应用程序都有一个首先执行的 Program 主程序，Program 作为主程序块将激活其他执行各种任务的所需的二级程序块——Unit 单元。绝大多数 Delphi 应用程序都是按照这样部署的。虽然开发过程中的实际程序，都会更复杂一些，但其基本的程序结构与图 6-1 所示的相似。

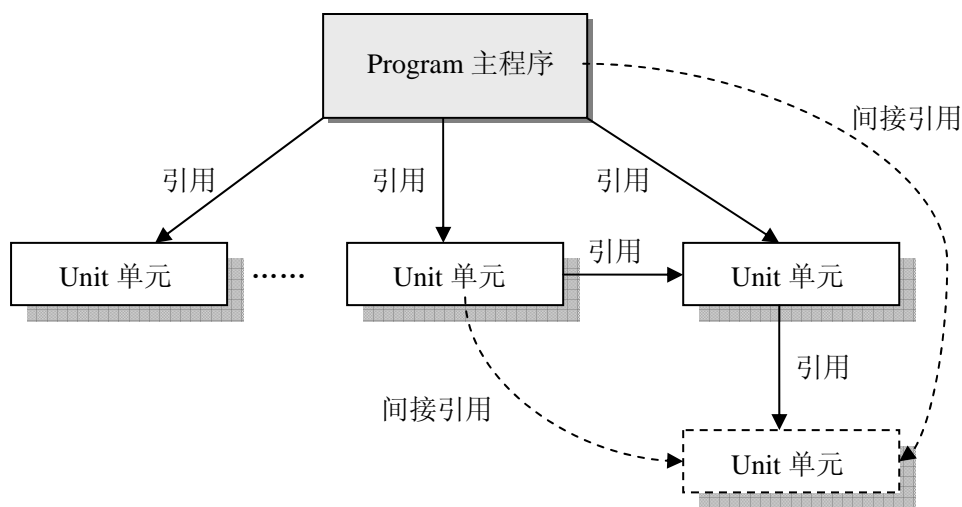


图 6-1 Delphi 程序的结构

一个 Program 主程序可以引用多个 Unit 单元，Unit 单元也可以引用别的 Unit 单元，甚至单元之间还可以相互引用。根据引用的传递关系，还可以产生间接引用的关系。所以规模稍微大一些的程序，可能不止两级或三级，层次可能会更多些。单元和层次正是 Delphi 程序易于模块化和结构化的基础。

6.1.1 Program——主程序

Program 主程序就是 Delphi 中的项目文件，它是一个特殊的源代码文件，类似于 C 语言中的 Main 程序，即为应用程序的主程序。一个程序可以由多个 Program 组成，也可以只由一个 Program 组成。Program 主程序的基本结构如图 6-2 所示。

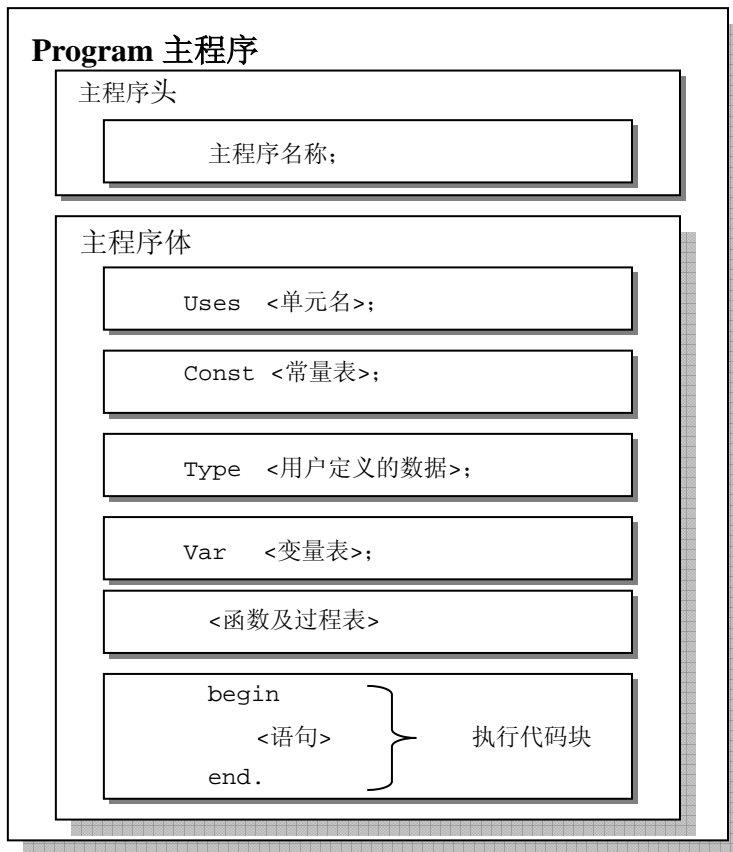


图 6-2 Program 主程序的基本结构

编写 Program 主程序代码时，我们要注意：

- Delphi 程序以关键字 `program` 后带该程序名称为开始。Delphi 是大小写无关语言，你可以用关键字 `Program` 代替 `program`，甚至可以用 `PROGRAM` 代替 `Program` 而不会出现任何编译错误。
- `Uses` 关键字列出了所有该程序需要引用的其他程序单元名，各单元之间用逗号“,”隔开，最后结束用分号“;”。这些单元对 Program 主程序提供了重要的支持和操作。
- `const` 关键字列出该程序中定义的常量名。常量是有固定值的名字。
- `type` 关键字列出在 Program 主程序中定义的用户自定义类型名。Delphi 允许用户定义自己的数据类型，这些类型表示某些特定的信息。
- `var` 关键字列出在 Program 主程序中定义的变量名。如前所述，变量存放可以被修改的信息。
- 在 Program 主程序中，也可以定义自己的过程或函数，它们在 Program 主程序的 `begin-end` 块中被调用。这些过程和函数是半独立的程序组件，可以存取在 Program 主程序中定义的常数、类型和变量，也能存取使用到的单元中的信息。
- `begin` 和 `End` 关键字定义首先执行的语句块。为了代码的有效性，在这个块中至少

应有一条语句。但有趣的是，即使 `begin-End` 块中没有一条语句，Delphi 编译器也不会报错。

注意：`uses`部分列出单元的顺序决定它们初始化的顺序，并影响编译器定位标识符的顺序。如果两个单元定义了一个相同名字的类型，编译器将总是使用先编译的那个单元的类型。

所有程序单元总是自动的引用 `System` 单元，而不必在 `Program` 主程序的 `uses` 部分显式指定。`System` 单元主要用于实现一些低级的运行期程序的支持，如文件输入输出 (I/O)、字符串操作、浮点运算、动态内存分配等。对于控制台应用程序以及其它一些简单的应用程序，程序中可以不含 `uses` 子句，此时只是使用了唯一的标准单元：`System`。但对于绝大多数情况下，程序 (program) 以及后面将介绍的库 (library) 等，都不可避免地要使用 `uses` 子句，并用 `to` 到 Borland 在 VCL (或 CLX) 中提供的大量例程或类。

在 Windows 程序设计中，由于 Delphi 会将 `Program` 主程序作为项目文件，所以通常情况下，建议不要手工编辑项目文件，Delphi 的 IDE 会自动维护该文件的源代码。如果确实需要在项目文件中添加一些特别的程序逻辑时，也应当尽量将语句封装在其他程序单元的过程、函数或对象中，并在项目中仅根据需要进行简单调用即可。如若不然，那么项目文件将显得臃肿，而且逻辑结构也欠清晰。

6.1.2 Unit——单元

Delphi 应用程序中的单元 (unit) 实际上就是一个程序模块，因此单元是程序模块化的基础。Delphi 在 Windows 程序设计中，每个窗体都对应一个单元。例如，在 Delphi 的集成环境中通过 `File | New | Form` 菜单命令，在项目中添加一个新窗体，实际上是增加了一个新单元 (也就是建立了该新窗体的类)。我们也可以在应用程序中添加不带窗体的单元，只要选择 `File | New | Unit` 菜单命令即可。有关 Windows 程序设计我们会在后续章节详细讲解。

1. 单元文件的基本框架

Delphi 使用单元来建立可重用的程序模块，每个单元都在其各自相应的单元 (.pas) 文件中保存代码。单元一般由类型 (type)、常量、变量以及例程 (函数和过程) 组成。程序员通过这些单元撰写功能单一的代码，因为他们比较容易维护和修改。最后再将不同的单元引用、汇集，构成大程序。

一个 Delphi 单元以单元头开始，接下来是 `interface` (接口部分)、`implementation` (实现部分)、`initialization` (初始化部分) 和 `finalization` (结束部分)。其中，初始化部分和结束部分是可选的。图 6-3 显示了单元文件的基本框架。

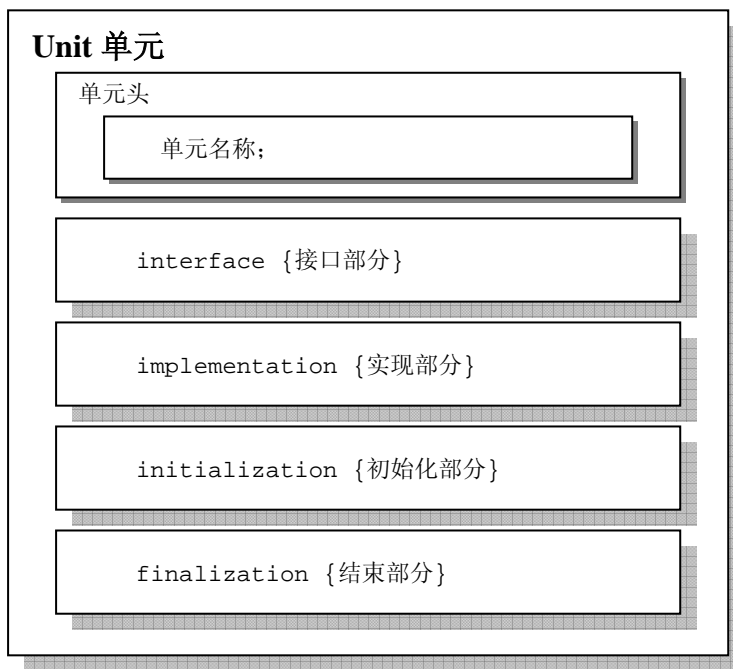


图 6-3 单元文件的基本框架

注意：单元名与文件名要保持相同，而且在一个项目中，单元名称必须唯一。

2. 接口部分

单元接口以保留字 `interface` 开始，直到实现部分结束。接口部分用于声明常量、类型、变量、过程、函数等，这些声明对于其它使用了该单元的单元、项目、库、包等是可用的，因此可以称为公共 (`public`) 实体。接口部分的结构如图 6-4 所示。

对照程序单元和 `Program` 主程序的结构，可以发现 `Unit` 和 `Program` 中都包含有 `uses`、`const`、`type`、`var` 子句。显然在 Delphi 的主程序和单元文件中，各子句使用了相同的关键字，因而可以推断它们的作用也基本相似。下面我们将详细介绍程序单元 `interface` 部分的各个子句。

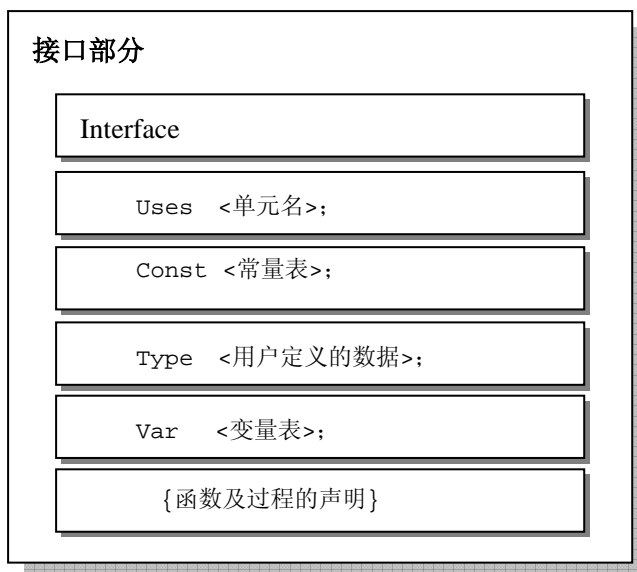


图 6-4 接口部分的结构

(1) uses 子句

接口部分包含的 uses 子句必需紧随保留字 interface 之后出现。uses 子句告诉编译程序：该单元使用了那些其它单元。如果引用了不止一个单元，各单元之间用逗号间隔，形成一个单元引用表。

Delphi 在发行时提供了许多预先定义的单元，在构造程序时可以直接而简便的使用。例如，从 Delphi 组件面板将一个按钮拖放进设计的窗体中，Delphi 就自动在当前 Unit 单元中加入要引用的 StdCtrls 单元（因为按钮控件 TButton 是在 StdCtrls 单元中定义的）。如果你使用的函数或控件不是来自于 Delphi 预先定义的单元，就必须自己手工加入需要引用的单元名称，否则编译时会出错。

(2) const 子句

const 子句列出具有某个固定值的标识符名字。这些值可以是整型，浮点型、字符串和其他预定义的，或用户定义的数据类型。例如：

```
const
    LANGUAGE          = 'Pascal';
    HourPerDay        = 24;
    SecondPerHour     = 60.0 * 60.0;
```

技巧：建议使用详细名称表示常数值，该方法将使你的程序易读且易改。不要给常数起不易理解的名字。

(3) type 子句

type 子句允许声明自定义的数据类型。这些新数据类型可以是记录，或者是类，或其他。例如：

```
type
    TDate_Rec = record           //日期记录
        Day, Month, Year : Integer;
    end;

    TDate_Class = class         //日期类
    private
        StartTime, FinishTime : TDate_Rec;
    public
        function calculateTime(start,finish: TDate_Rec):integer;
    end;
```

(4) var 子句

Var 子句声明单元输出的变量。也就是说，在单元接口部分所声明的变量都是全局变量，只要其他单元引用了该单元，就能操纵和使用该变量。由于该变量暴露在外，无法预料其他程序单元对该变量值所进行的改动，因此声明全局变量一定要慎重。除非必要，一般应尽量少在接口部分声明变量，而改在实现部分声明。

如前所述，变量存储的方式依赖于变量的数据类型。如果变量有相同的数据类型，可以将这些变量放在一个以逗号间隔的表中。例如：

```
var
    Second: Integer;
    Hour : Integer;
    Minute : Integer;
    aDate: TDate_Rec;           //记录类型
    Duration: TDate_Class;     //面向对象类型
```

(5) 过程或函数声明子句

接口部分中声明的过程或函数仅包括例程头。例如，程序 Project1 中的 Button1Click() 过程，在 interface 部分就只有如下内容：

```
procedure Button1Click(Sender: TObject);
```

过程或函数的定义块位于接口部分之后的实现部分中，因此尽管并没有使用 forward 指示字，接口节中的过程和函数声明也可以看作是向前声明。

3. 实现部分

在单元中，实现部分以保留字 **implementation** 开始，直到初始化部分开始（如果初始化部分存在）或直到单元结束。在接口部分中声明的过程和函数（或类中的方法），都在实现部分中定义。在实现部分中，可以对这些过程和函数以任意的顺序定义和调用。此外，对这些在接口部分中声明的过程和函数，在实现部分中定义时可以省略参数列表。但如果在定义时要包括参数列表，则必需与接口部分中的声明严格匹配。

除了定义过程和函数以外，实现部分也可以声明常量、类型（包括类）、变量等。需要说明的是：这些实体仅限于在当前单元的实现部分中使用，因此叫做私有（**private**）实体，也就是说，这些实体对于引用该单元的其它单元是不可见的。

实现部分也可以包括 **uses** 子句，但必需紧随保留字 **implementation** 之后出现。从上面的 Unit 单元的语法框架中，可以看出：在接口部分和实现部分，先后两次出现 **uses** 子句。不过，它们的作用是不相同的。如果在单元的接口部分中需要引用其它单元（例如从其它单元中继承类），则被引用的单元名必需列在接口部分的 **uses** 子句中；如果在实现部分中引用其他单元（例如使用其它单元中的常量、例程），则被使用的单元名既可以列在接口部分的 **uses** 子句中，也可以列在实现部分的 **uses** 子句中。需要注意的是，被使用的单元，其单元名在这两处 **uses** 子句中只能出现一次，否则编译器将不认可。

4. 初始化和结束部分

初始化部分是可选的，它以保留字 **initialization** 开始一直到结束部分开始（如果单元中有结束部分）或单元结束（如果单元中没有结束部分）。初始化部分含有用于执行的语句，当程序开始执行时，将根据初始化部分出现的顺序依次执行其语句。例如，如果需要对某些数据结构进行初始化，那么可以将初始化语句置于初始化部分。

在一个主程序或程序单元当中所引用到的各个程序单元中，它们的初始化部分执行的先后关系，是其在 **uses** 子句中出现的顺序决定的。

初始化部分中的语句在程序运行开始时执行一次，程序运行中将不再执行。运行中，其它语句也无法将执行点跳转到初始化部分中。

在程序单元的结构中，结束部分也是可有可无的。当且仅当单元中有初始化部分时才允许有结束部分。结束部分以保留字 `finalization` 开始，直到单元结束。当主程序终止时，结束部分中的语句被执行。在初始化部分中分配的资源，通常在结束部分中释放。

对于项目中的多个单元，结束部分执行的顺序与初始化部分执行的顺序相反。例如，如果在应用程序启动时按照单元 A、B、C 的顺序进行初始化，那么在应用程序结束时将按照单元 C、B、A 的顺序执行相应的结束部分。

一旦单元初始化部分的代码开始执行，则相应的结束部分将在应用程序结束时一定会被执行到。因此，结束部分必须有办法处理还没有完全初始化完毕的数据，因为万一运行时有错误发生，初始化部分代码可能得不到完全执行。

结束部分中的语句在程序结束时执行一次，程序运行中不执行。运行中，其它语句也无法将执行点跳转到结束部分中。

6.1.3 单元的引用

程序单元写在 `uses` 子句中的顺序决定了 `initialization` 部分的程序代码执行的顺序，也会影响编译程序判别标识符的方式。如果有两个程序单元声明了同名变量、常数、数据类型、过程或者函数，编译程序实际上用的是写在 `uses` 子句里比较后面的那个程序单元中声明的那一个。

写在 `uses` 子句中的程序单元，只需要包括那些应用程序或程序单元会直接用到的。也就是说，如果程序单元 A 会引用程序单元 B 声明的常数、数据类型、变量、过程或函数，那么 A 就必须清楚地把 B 写在 `uses` 子句中。如果 B 会引用程序单元 C 里的标识符，A 就间接依赖 C。在这种情况下，C 不需要出现在 A 的 `uses` 子句中，可是编译程序仍然必须要能够找到 B 和 C 才能处理 A。

以下的这个范例程序说明了上述的间接依赖关系：

```
program Prog;
uses Unit2;
const a=b;
...
unit Unit2;
interface
uses Unit1;
const b=c;
...
unit Unit1;
interface
const c=1;
...
```

在这个范例里，`Prog` 直接依赖 `Unit2`，而 `Unit2` 又直接依赖 `Unit1`，因此 `Prog` 是间接依赖 `Unit1`。因为 `Unit1` 并没有出现在 `Prog` 的 `uses` 子句当中，所以 `Prog` 不能用到 `Unit1` 所声明

的任何标识符。

要编译一个程序模块，编译程序必须要能够找到该模块所依赖的各个程序单元，不管是直接依赖还是间接依赖。除非这些程序单元的源程序有所改变，否则编译程序只需要这些程序单元的.dcu文件，而并不是它们的源程序（.pas文件）。

如果程序单元的 **interface** 部分做了修改，那么所有依赖它的程序单元都必须重新编译。但如果只是修改程序单元的 **implementation** 或其它部分，依赖它的程序单元并不需要重新编译。编译程序会自动追踪记录所有程序单元的依赖关系，而且只有在需要的时候才会重新编译它们。

当程序单元间彼此直接或间接地相互引用，也就说这些程序单元彼此互相依赖着对方。只要没有在两个程序单元 **interface** 部分的 **uses** 子句形成循环，这种相互依赖关系是被允许的。换句话说，如果从程序单元的 **interface** 部分出发，沿着有依赖关系的程序单元的 **interface** 部分，一定不可能再走回到原先的程序单元。如果要让有相互依赖关系的程序单元能够合乎语法规则，每一条循环引用（**Circular Reference**）的路径就至少必须通过一个程序单元的 **implementation** 部分的 **uses** 子句。

以最简单的两个互相依赖的程序单元为例，这两个程序单元不能把对方写在它们 **interface** 部分的 **uses** 子句中。也就是说，以下的范例程序在编译的时候会有错误：

```
unit Unit2;
interface
uses Unit1;
...

unit Unit1;
interface
uses Unit2;
...
```

然而，如果其中一个程序单元把对方写在了 **implementation** 部分的 **uses** 子句中，这两个程序单元就可以合法地互相引用对方的数据了。

```
unit Unit2;
interface
uses Unit1;
...

unit Unit1;
interface
...
implementation
uses Unit2;
...
```

如果可能，尽量把所要引用的程序单元都列在 **implementation** 部分的 **uses** 子句，这是一个减少发生循环引用的好方法。一个程序单元只有在一种情况下才一定要列在其它程序单元

的 `interface` 部分的 `uses` 子句，即其它程序单元在它们的 `interface` 部分已引用到该程序单元所定义的标识符的时候。

6.1.4 标识符的作用范围

标识符的作用范围就是它的生命期，因此标识符的作用范围决定了它的可访问性。标识符范围从它被声明开始，随包围声明的代码段结束而结束。比如，在过程中声明的一个常量、变量、类型或其他符号标识符随着过程的 `end` 语句而结束。标识符声明及其随后的语句代码构成了一个区块，如下所示：

```
标识符声明
begin
  语句代码
  ...
end;
```

变量、函数等标识符只能用在它所声明的范围内。其声明的位置决定了它的作用范围。下面讨论在不同声明位置中标识符的作用范围。

- **单元中** 单元由接口部分和实现部分组成。在接口部分声明的标识符对所有其他单元和引用该单元的程序都是可以访问的。此外，这些标识符，特别是常量、数据类型、类和对象等，对实现部分也是可访问的。相对而言，在实现部分声明的标识符访问受点限制，他们仅限于本单元内使用。他们只对本单元内其后的常量、数据类型、变量、类、对象以及例程（函数和过程）等是可以访问的。
- **主程序中** `Program` 主程序中可以声明常量、数据类型、变量、类、对象以及例程。例程可以访问主程序中声明过的常量、数据类型、变量、类、对象。
- **例程中** 例程中声明了的参数和标识符，其作用范围严格受限，只能在局部使用。该例程以外的其他程序都不能访问例程内部的参数和标识符。因此位于例程中的标识符也称为局域标识符。
- **嵌套例程中** 嵌套例程作为子程序可以访问上层父辈程序中声明的标识符，但嵌套例程中声明的标识符仅限于自己内部使用，外部程序无法访问。
- **类中** 类对于作用范围有自己的规则。后面 7.2.3 节“类成员的可见性”中我们详细介绍。

由此可见，声明在程序、函数或过程声明中的标识符，其有效范围限制在所声明的区块中。声明在单元接口部分中的标识符，其有效范围包括引用到该单元的程序单元或主程序。声明在函数或过程中的变量有效范围较小，称为局部变量；而那些作用范围较大的变量称为全局变量。

6.2 结构化程序设计基础

荷兰学者 Dijkstra 提出的“结构化程序设计”的思想和规则，能够使程序具有合理的结构，以保证和验证程序的正确性。该规则要求程序设计者不能随心所欲地编写程序，而要按照一定的结构形式来设计和编写程序。其主要目的是使程序具有良好的结构，使程序易于设计，易于理解，易于调试修改，以提高设计和维护程序工作的效率。

Delphi 的前生 Pascal 作为优秀的结构化程序设计语言，曾经是学习结构化设计的首选语言。不少经典教科书中用其编写的优雅的结构化程序往往是学习结构化编程的典范。

结构化程序设计不仅是学习编程的基础，还是软件开发进程中一个重要部分，即使在现代软件项目中，该方法仍然占有重要的地位。

6.2.1 结构化设计的特征

前面介绍过三种基本的控制结构：顺序结构、选择结构和循环结构。图 6-5 使用流程图的形式总结了这三种结构及其对应的实现语句。在图中用圆圈表示每种结构的单入口点和单出口点。

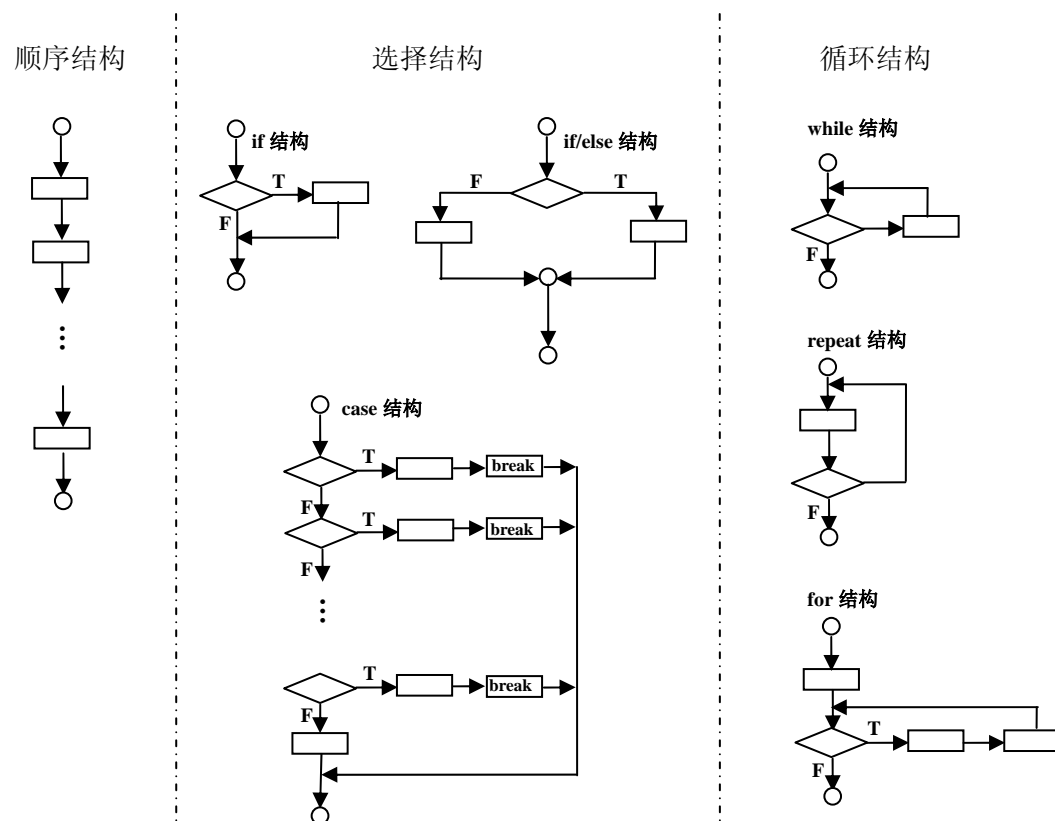


图 6-5 Delphi 的单入单出控制结构

从图 6-5 中可以看出，三种基本结构具有以下特征：

- 一个入口；
- 一个出口；
- 结构中每一部分都应当有被执行到的机会，也就是说，每一部分都应当有一条从入口到出口的路径通过它（至少通过一次）；
- 没有死循环。

结构化程序要求每一基本结构具有单入口和单出口的性质是十分重要的，这是为了便于保证和验证程序的正确性。设计程序时，一个结构接着一个结构地顺序写下来，整个程序结构如同一串珠子一样顺序清楚，层次分明。在需要修改程序时，可以将某一基本结构单独孤立出来进行修改，由于单入口单出口的性质，不致影响到其它的基本结构。

6.2.2 构造结构化程序的规则

正如 Bohm 和 Jacopini 所证明的，任何程序均可用由三种基本的控制结构实现。但是我们也必须明白任意的连接控制结构将导致非结构化程序。例如图 6-6 就是一个非结构化的流程图。

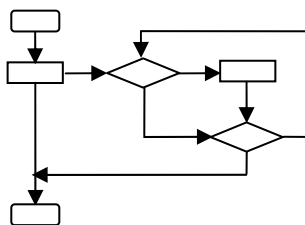


图 6-6 非结构化程序流程图

为了构造正确的结构化程序，编程人员总结出如下三条规则：

- 规则 1、任何矩形框可用两个顺序的矩形框代替；
- 规则 2、任何矩形框可用任何简单控制结构代替；
- 规则 3、规则 1 和规则 2 可根据需要按任何顺序使用任意次。

（假设矩形流程符号表示包括输入输出在内的任何动作）

为了理解这三个规则，可以参看图 6-7。从图中可以看出，规则 1 使用起来很简单，它可以产生许多顺序的矩形框的结构化流程图，类似于堆，因此可称规则 1 为堆规则。规则 2 称为嵌套规则，对简单的流程图反复的使用规则 2 可以获得一个具有清晰结构的嵌套控制结构的流程图。规则 3 实际上说明规则可重复使用，它可以用来生成规模更大、关系更复杂并且嵌套层次更多的结构。

遵循这三条规则，就不会产生非结构化的流程图。如果需要确定某个流程是否是结构化的，可以反过来应用上述规则，如果最终能够简化到最简单的流程图，就说明这个流程图是结构化的；反之，就是非结构化的。

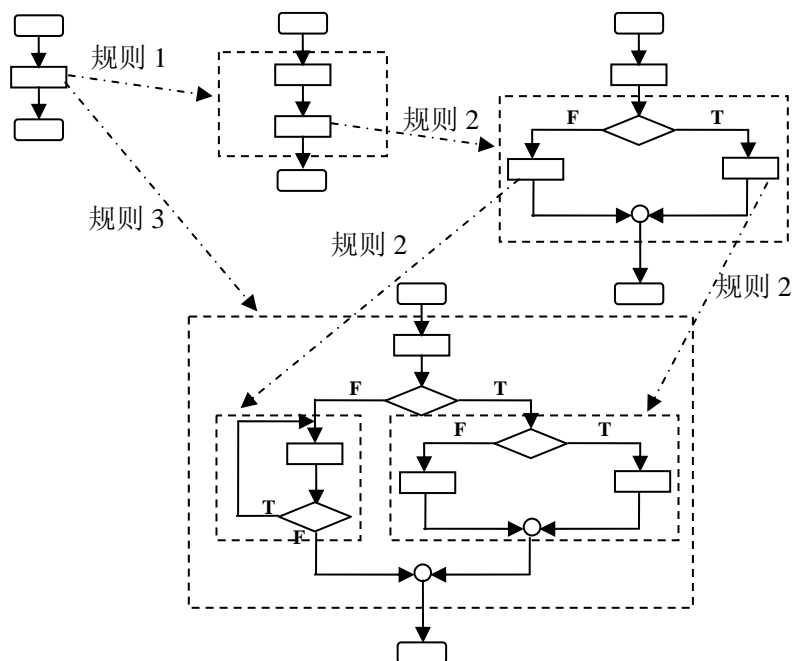


图 6-7 构造结构化程序规则的应用

6.2.3 结构化程序设计方法

面对一个问题，应怎样进行思考、着手解决呢？通常有两种不同的方法：

一种是自顶向下，逐步细化；一种是自下而上，逐步积累。以写文章为例，通常会先从全局考虑，明确文章的主要内容，然后按照主要内容的关系将整个文章分成几个部分，接着会再进一步考虑每个部分分成几节，每节分成几段，每段应包含什么内容。用这种方法逐步分解，最终形成一篇文章的整体框架。该方法的主旨是：“自顶向下，逐步细化”。

另外一种写文章的方式是：不拟提纲，首先做的是写好各种卡片形式的片断，想到那里就写到那里，最后再通过组织，将卡片“串”起来，形成一篇文章。类似的，这种方法的主旨是：“自下而上，逐步积累”。显然，使用第一种方法，作者更容易考虑周全、行文流畅，写出来的文章结构会更加清晰，层次更加分明。

结构化程序设计方法的主要思想与第一种写文章的方法类似，它采用的正是模块化、自顶向下逐步求精的设计原则。结构化程序设计强调程序设计风格和程序结构的规范化，提倡使用清晰的程序结构。怎样才能得到一个结构化的程序呢？如果我们面临一个比较复杂的问题，是难以一下子写出一个层次分明、结构清晰、算法正确的程序的。这时我们就可以使用结构化程序设计方法，化繁为简。用结构化程序设计方法也便于验证算法的正确性：在向下一层展开之前应仔细检查本层设计是否正确，只有保证本层是正确的前提下，才能向下细化；如果每一层设计都没有问题，则整个算法就是正确的；由于每一层向下细化时都不太复杂，因此容易保证整个算法的正确性，检查时也是由上而下逐层检查。

结构化程序设计方法提高了程序的清晰简单性，与非结构化方法编写的程序相比，程序更容易理解，因而更容易测试、调试、修改，也更容易保证程序和算法的正确性。

6.3 结构化设计应用举例

结构化程序设计方法在实际中使用的主要方式是自顶向下逐步求精的形式。为了说明该设计方法的主要思想和步骤，本节将给出一个范例，通过这个范例我们也可以对前面所学的知识作一个总结。

6.3.1 问题及分析

本范例求解的是数值排序和查找的问题。该问题的描述如下：

对随机产生的一组的整数（10 个）进行排序，要求能提供两种排序的选择：冒泡排序和快速排序，然后再在已排序好的组数中查找用户提供的数据，如果查得此数据在这组数中，则显示该数在排序好组数中的位置：“查得此数据位置为 X”（X 为实际的位置），如果没有发现，则提示：“无此数据”。要求使用折半查找算法，因此查找数据之前，先要判断数组是否已经排序，如果还没有进行排序，则采用冒泡排序法先进行排序，再检索。在用户没有确定退出程序之前，用户可以重复选择排序和查找操作。

经过认真阅读问题说明，我们可以得出以下几点：

- 用户退出程序之前，可以重复操作，说明主体上是一个循环控制结构。
- 主要的流程为：输入数据、选择排序、检索数据。

- 用户可以选择的操作：随机产生数据、数据排序或者数据查找。
- 排序时，提供用户两种选择：冒泡排序和快速排序。
- 使用查找算法之前，需要判断是否已经排好序。

6.3.2 结构化设计

通过对问题的认识以及分析，我们可以得到程序的基本结构是一个类似 repeat 循环的结构；然后我们可以对循环的主要操作按照结构化规则 1 进行细化，分为数据输入和数据处理两个模块；接着，根据题意及分析结果，数据处理又可以按照规则 2 使用选择结构代替，为用户提供两种操作的选择，当然，这两种操作都是有输出动作的。因此，这个程序流程的结构化过程如图 6-8 所示。

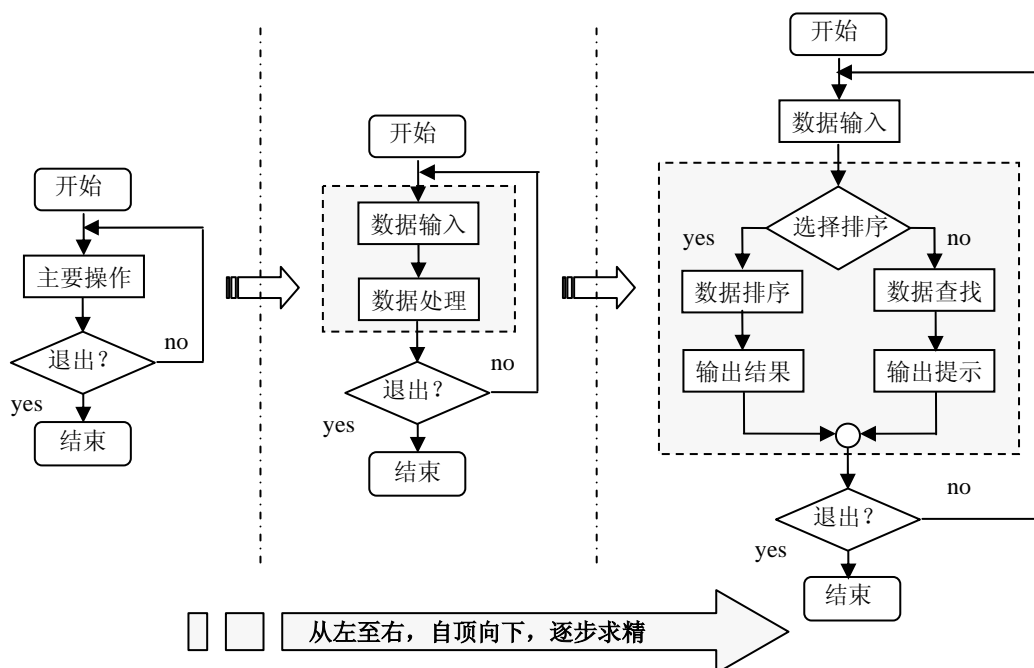


图 6-8 自顶向下的结构分析

根据问题的要求，程序设计需要继续往下层细化。在排序时，为用户提供排序方法的选择，可以是冒泡排序或者是快速排序中的一种，排序完毕后，将有序的数据输出。对于数据的检索，则首先需要判断待检索的数据是否已经排序，若没有排序，则先用冒泡排序方法进行排序，再进行数据的检索；若已经排序了，则可以直接进行检索操作。检索所使用的方法都是一样的：折半查找。这样，该程序的结构就可以更具体了，图 6-9 描述了这一分析层次上的程序结构流程图。

在实际的编程过程中，上面的结构化流程图应该继续细化下去，也就是到具体的算法这一级别。由于本书的前面已经比较详细的介绍过这几种基本的算法了，所以在这里就不再往下分析、求精了。

通过自顶向下逐步求精的设计方法分析，我们得到了一个基本的程序流程图。现在的工作是：需要将它转换为代码的形式。这应该不是太难，只需要按基本的流程图结构与控制语句之间的相互映射关系，就可以构造这个程序。

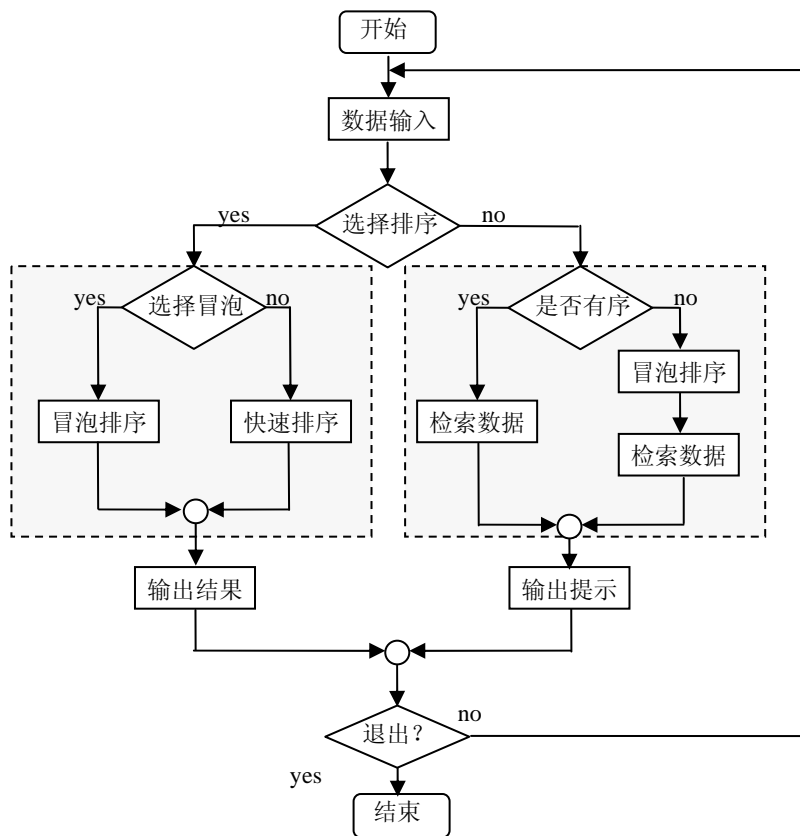


图 6-9 程序的结构化流程图

由于程序设计并没有细化到程序的语句级，为了描述方便我们有时还需要采用伪代码，为结构化流程图与代码之间构建一座桥梁。该程序的伪代码如示例程序 6-1 所示。

示例程序 6-1 程序的伪代码

```

repeat
begin
    产生 10 个无序的随机数据
    if 选择排序 then
    begin
        if 选择冒泡排序 then 冒泡排序
        else 快速排序
        输出排序结果
    end
    else
    begin
        if 数组已经排序 then 二分法检索数据
        else
        begin
            冒泡排序
            二分法检索数据
        end
    end
end
  
```

```
        输出检索结果的提示
    end
end
until 用户退出程序
```

这段伪代码基本上给出了该范例程序的整体框架。通过使用伪代码，我们还能够继续细化、求精，直到接近最终的实际代码。这样的结构化程序设计过程最终使我们能够顺利实现整个程序的编码和调试工作。

6.3.3 范例程序的实现

本章的重点是研究 Delphi 程序的结构并介绍 Delphi 结构化程序设计方法。通过范例程序，我们不但可以学习如何使用 Delphi 来进行一个实际程序的开发，实现我们前面所做的结构化设计，还可以剖析一个完整的 Delphi 应用程序的结构，并将我们所学的算法知识做一个回顾。

我们将范例程序命名为“排序和查找算法演示程序”。该应用程序分为用户界面和算法逻辑两层结构。不同的是，这次采用的用户界面是图形化界面，而且涉及到 Windows 编程（这部分内容在本书后面章节有详细讲解），但这并不妨碍我们理解程序，因为核心程序集中在算法逻辑部分中。相反，生动的图形界面会让我们更直观地理解排序和查找算法的效果，并为我们进一步学习 Delphi 带来兴趣。

1. 范例程序的结构组成

为了按照结构化的设计实现范例程序，首先我们画出程序模块的组成框图，如图 6-10 所示。然后，我们在 Delphi 中进行如下操作：

- 通过主菜单的 File | New | Application 菜单项创建一个 Application 项目（即 Windows 应用程序），并将该项目命名为 SortAndFind 保存。
- 该项目文件对应着我们的主程序。项目中还包含了一个缺省的主窗体 Form1。我们可以将该窗体用作用户操作的图形界面。我们将窗体文件（也是一个单元文件）命名为 uInterface 保存。
- 通过主菜单的 File | New | Unit 菜单项，我们可以为该项目创建一个程序单元作为新的程序模块。用这样的方法，我们创建 uSort 单元作为排序算法逻辑模块，创建 uBinSearch 单元作为查找算法逻辑模块。

最后，我们通过主菜单的 View | Project Manager 菜单项打开 Delphi 项目管理器，可以发现范例程序项目中的文件组织与我们设计的框图一致。

范例“排序和查找算法演示程序”的结构组成说明了该应用程序包含有 SortAndFind 主程序以及 uInterface、uSort 和 uBinSearch 3 个单元。这 3 个单元分别对应着 3 个程序模块：图形界面模块、排序算法模块和查找算法模块。下面我们分别来了解这些部分是如何用 Delphi 代码实现的。

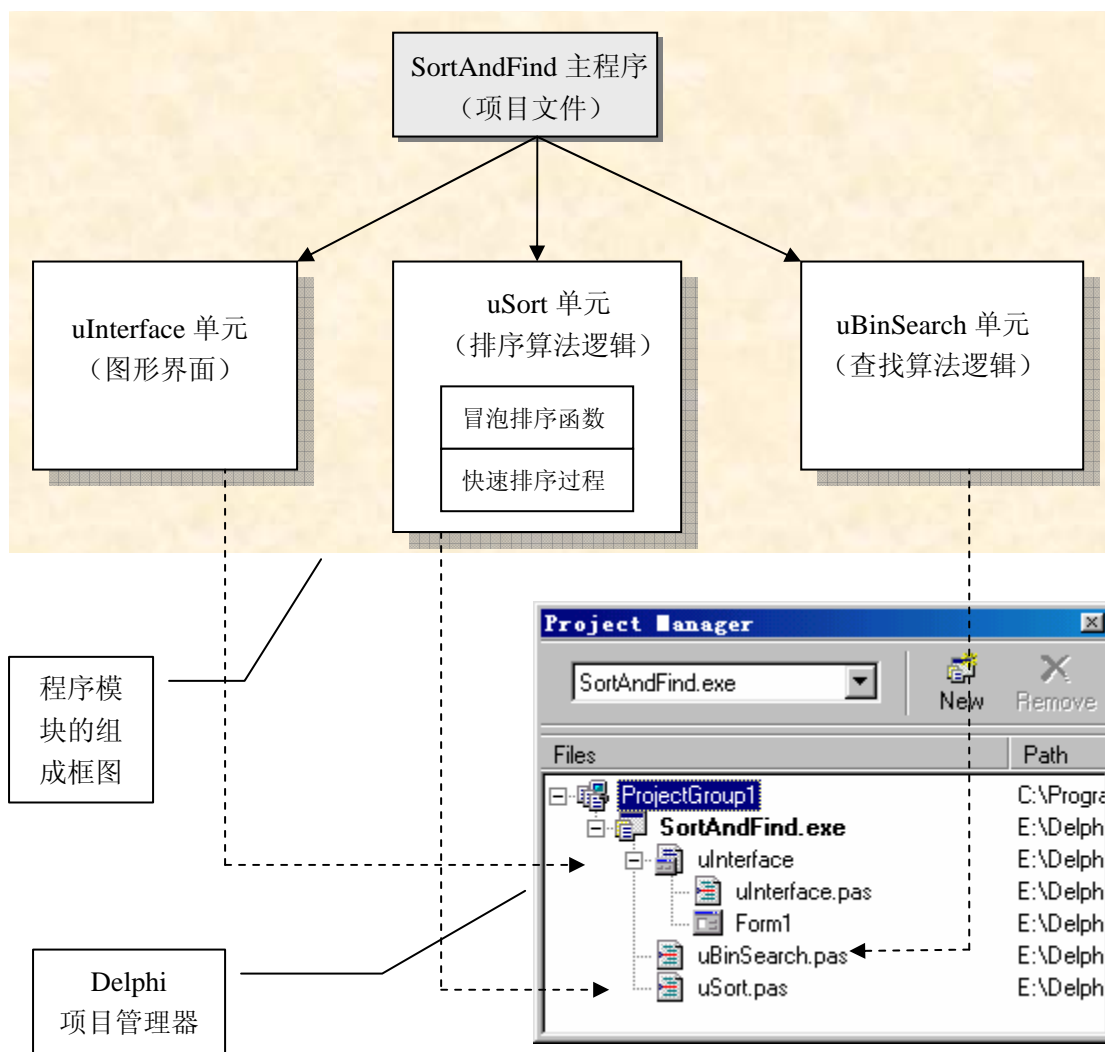


图 6-10 范例程序的组成框图和对应的 Delphi 项目

2. 各程序单元代码分析

我们先编写程序的核心逻辑单元，即两个算法单元。

(1) 算法逻辑单元

首先是作为排序算法模块的 uSort 单元，该单元中包含了一个冒泡排序函数 BubbleSort 和一个快速排序过程 QuickSort。冒泡排序函数接受一个 TIntArray 类型的数组参数 a，并对该数组各元素进行冒泡排序，最后返回一个排好序的 TIntArray 类型数组。而快速排序过程，虽然也接受一个 TIntArray 类型的数组参数 a，但是该参数是作为变量参数而不是值参数传递的，因此会影响到实参的值。也就是说，作为实参的数组传入该过程排序后，其内部排列的元素值由原来的无序排列变成了有序排列。

在实际设计中，为了方便客户程序调用，应该将两种排序算法都统一成一种形式（比如：都设计成过程，都使用变量参数）。我们在范例程序实现中有意不这样做，目的是为了让读者有机会回顾前面第 5 章学过的函数与过程及参数方面的内容。在后面的调用排序例程的用户界面操作程序中，我们还会向读者演示对于函数和过程、值参数和变量参数的不同处理方法。

至于排序算法，我们在上一章中已经讲过，读者应该可以自己读懂程序。需要指出的是，

TIntArray 类型是我们在作为用户界面模块的 uInterface 单元中声明的整数数组类型:

```
type
  TIntArray = array[0..9] of integer;
```

为了使用该标识符,我们需要在 uSort 单元中引用 uInterface 单元,为此在 uSort 单元的接口部分加入以下代码:

```
uses uInterface;
```

同时考虑到本单元的目的是提供两种排序算法,因此还要在单元的接口部分公布冒泡排序函数 BubbleSort 和快速排序过程 QuickSort,以便其他单元的程序调用。最后实现的完整代码如示例程序 6-2 所示。

示例程序 6-2 作为排序算法模块的 uSort 单元源代码

```
unit uSort;
{排序}

interface

uses uInterface;

function BubbleSort(a:TIntArray):TIntArray;
procedure QuickSort(var a:TIntArray; s, t:integer);

implementation

function BubbleSort(a:TIntArray):TIntArray;
var
  i, j, temp:integer;
begin
  for i:=low(a) to high(a)-1 do
    for j:=low(a) to (high(a)-1)-i do
      if a[j]<a[j+1] then
        begin
          //交换两个数据
          temp:=a[j];
          a[j]:=a[j+1];
          a[j+1]:=temp;
        end;
  result:=a;
end;

procedure interchange(var a:TIntArray; i, j:integer);
```

```
var
    tempChange : integer;
begin
    tempChange := a[i];
    a[i] := a[j];
    a[j] := tempChange;
end;

procedure QuickPass(var a:TIntArray;s,t:integer; var sign : integer);
var
    i, j, temp : integer;
begin
    i:=s;
    j:=t;
    temp:=a[s];
    while i<j do
    begin
        while (i<j) and (a[i]>=temp) do inc(i);
        while (i<j) and (a[j]<=temp) do dec(j);
        if i<j then
            interChange(a,i,j);
            if (a[t]>temp) and (i=t) then //划分元素在末端情况处理
            begin
                a[s]:=a[i];
                a[i]:=temp;
                sign := i;//划分元素的位置 sign
                exit;
            end;
        end;
        a[s]:=a[i-1];
        a[i-1]:=temp;
        sign := i-1; //划分元素的位置 sign
    end;

procedure QuickSort(var a:TIntArray;s,t:integer);
var
    sign : integer;
begin
    if s<t then
    begin
        QuickPass(a,s,t,sign);
        //快速排序的递归实现
        QuickSort(a,s,sign-1);
        QuickSort(a,sign+1,t);
```

```
    end;  
end;  
  
end.
```

uBinSearch 单元与 uSort 单元的结构基本一样，该单元包含了一个折半查找算法的实现函数 binsearch，该算法我们也在前面一章的数组一节中讲过，这里不再赘述。请读者自己阅读示例程序 6-3 所示的代码。

示例程序 6-3 作为查找算法模块的 uBinSearch 单元源代码

```
unit uBinSearch;  
{折半查找}  
  
interface  
  
uses uInterface;  
  
function binsearch(a:TIntArray; k:integer):integer;  
  
implementation  
  
function binsearch(a:TIntArray; k:integer):integer;  
var  
    l,h,mid:integer;  
    found:boolean;  
begin  
    l:=low(a);  
    h:=high(a);  
    found:=false;  
    while(l<=h) and not found do  
    begin  
        mid:=round((l+h)/2); //round 为取整函数  
        if k<a[mid] then  
            l:=mid+1  
        else  
            if k=a[mid] then  
                found:=true  
            else  
                h:=mid-1;  
        end;  
        if found then  
            result:=mid  
        else  
            result:=-1;  
    end;  
end;
```

```
end;
end.
```

(2) 用户界面单元

最后，我们来设计带窗体的单元 `uInterface`，作为用户操作的图形界面。在这个界面单元的设计中，我们主要考虑的问题来自输入和输出两个方面：

- **输入** 如何产生一组用于输入的随机数据？如何显示这组未排序的输入数据？
- **输出** 如何显示已作排序处理输出的数据？如何显示数据查找的结果？

对于随机数据，可以由 Delphi 提供的随机函数 `random` 产生：

```
for i:=0 to 9 do
begin
    a1[i]:=random(255); //为数组元素随机产生 0~255 之间的随机整数值
    ...
end;
```

由于输入和输出数据都无需用户手工修改，所以在 Windows 窗体中，我们可以使用标签 (`TLabel`) 来显示只读的数据。考虑到该程序是“排序和查找算法演示程序”，为了加强演示效果，我们不妨将用于排序的数据转换成不同颜色的对应色块。这样一来，未排序数据的色块，颜色杂乱排列；而已排序数据的色块，颜色由浅到深有序排列，一眼就可以看出来。这也可以体现图形化界面的优势，便于我们对不同数据排序前后的排列变化有直观地了解。最后我们决定在图形界面上使用带标签的编辑框 (`TLabelledEdit`) 控件来显示数据，该控件是编辑框和标签控件的组合，我们可以用其标签部分显示数据的数值，用其编辑框部分来显示对应的色块。

```
//edt1[i]是 TLabelledEdit 类型的数组元素
//利用 TLabelledEdit 控件来显示未排序数组 a1 的数值
edt1[i].EditLabel.Caption:=inttostr(a1[i]);
//利用 TLabelledEdit 控件的颜色来图形化显示未排序数组 a1 的数值
edt1[i].Color:=rgb(a1[i],255,255);
```

上面代码中，使用 `rgb` 颜色函数将 0~255 之间的整数值转换成不同深浅度的颜色。具体地说，`rgb(a1[i],255,255)` 中，`a1[i]` 值越大，`edt1[i]` 的颜色越浅，`a1[i]` 值越小 `edt1[i]` 的颜色越深（参见图 6-12 所示的实际运行界面）。

Delphi 作为强大的 Windows 应用程序可视化开发工具，在开发图形界面方面可谓得天独厚。我们既可以通过 Delphi 集成环境中的组件面板，往空白窗体上拖放控件，组成我们需要的设计界面；也可以在程序中动态创建和设置需要的控件。

首先，我们通过 Delphi 组件面板，向空白窗体拖放 1 个编辑框 (`TEdit`) 控件、5 个按钮 (`TButton`) 控件和 3 个标签 (`TLabel`) 控件，它们的布局如图 6-11 所示。另外，在窗体上我们还留出了输入数据显示区和输出数据显示区，以便在程序中动态创建和布置显示数据的控件。

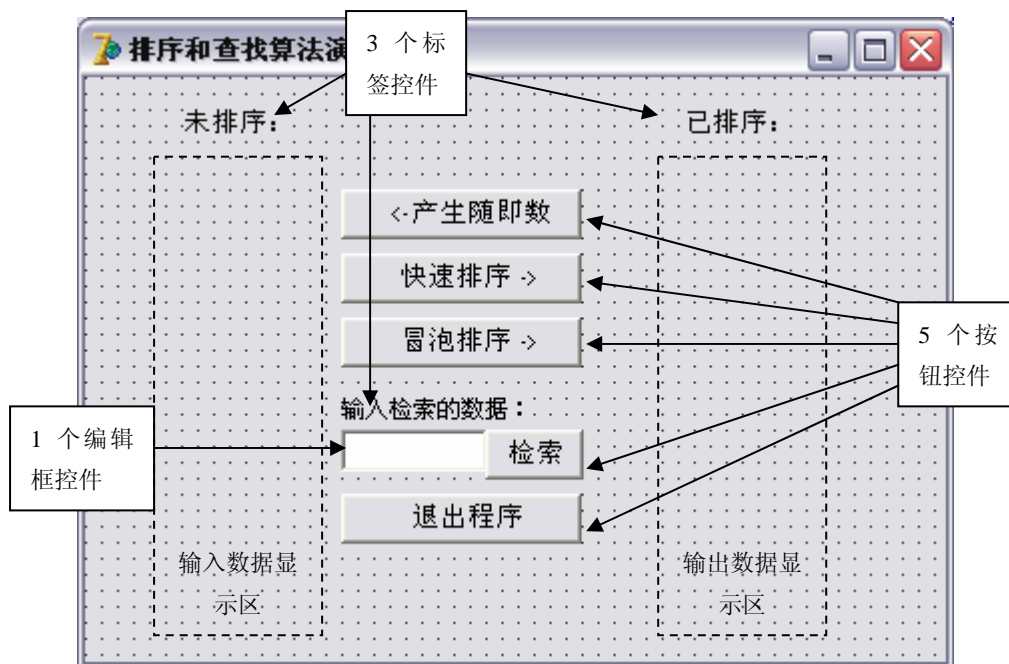


图 6-11 范例程序可视化设计的界面

需要动态创建的控件是用于显示数组数据的 `TLabelledEdit` 控件。为了使他们一一对应于数组元素,我们也可以声明一个 `TLabelledEdit` 类型的数组,该数组元素就是这些 `TLabelledEdit` 控件。用于排序的整数数组类型 `TIntArray` 和用于显示排序数据的 `TLabelledEdit` 数组类型 `TEditArray` 在 `uInterface` 单元接口部分声明如下:

```
type
  TIntArray = array[0..9] of integer;
  TEditArray = array[0..9] of TLabelledEdit;
```

于是,我们在 `TForm1.FormCreate` 过程(即面向对象程序设计所称的 `TForm1` 类的 `FormCreate` 方法)中写下以下代码,以便在 `Form1` 窗体创建的同时,在窗体上动态创建用于显示输入数据的带标签的编辑框(`TLabelledEdit`)控件:

```
edt1[i]:=TLabelledEdit.Create(self); //创建控件对象实例
edt1[i].Parent:=self; //指明放置该控件的容器对象为 Form1
edt1[i].Left:=50; //设置控件左边距
edt1[i].Top:=24*i+30; //设置控件上边距
edt1[i].Height:=20; //设置控件高度
edt1[i].Width:=60; //设置控件宽度
edt1[i].LabelPosition:=lpLeft; //设置控件的标签在编辑框的左边
```

这些代码创建了 i 个带标签的编辑框控件对象,并为其进行布局、大小等属性设置。通过写代码动态创建的控件和可视化设计时拖放到窗体的控件都能一样很好地运行,最后实际运行界面如图 6-12 所示,读者可以将其和设计的界面做一个效果对比。

尽管从组件面板拖放控件到窗体比较简单,但动态创建控件在编程时操控更为灵活。比如,可视化拖放到窗体的控件就不能作为控件数组管理。而我们动态创建的控件则可以通过

控件数组方便地与排序数组建立对应关系。例如，在每次排序、查找处理完毕后，我们都需要在一组用于显示输出数据的带标签的编辑框中显示已排序数组 `a2`。有了控件数组 `edt2`，这项工作就简单多了，我们仅需写一个简单的循环语句，就建立了控件数组 `edt2` 与已排序数组 `a2` 的对应关系，从而实现了数据的转换和显示。这段程序写在 `TForm1.Display` 过程中，并在排序和查找操作中调用，其代码如下所示：

```
//显示排序结果
procedure TForm1.Display;
var
  i:integer;
begin
  for i:=0 to 9 do
  begin
    //我们用 edt2 的数组元素对象来显示排序结果，即数组 a2 的值
    edt2[i].EditLabel.Caption:=inttostr(a2[i]);
    edt2[i].Color:=rgb(a2[i],255,255);
  end;
end;
```

用户在图形界面上对“排序和查找算法演示程序”的操作主要是通过点击按钮实现的。用户点击按钮时触发一个点击按钮事件，导致程序执行该事件的对应代码。因此，我们需要做的就是为这些点击按钮事件编写对应的执行代码。在 Delphi 集成环境中，我们选中窗体上的按钮，用鼠标双击该按钮，Delphi 就自动生成一个对应点击按钮事件的过程，我们随即可在这个过程中加入对应的程序代码。范例程序中诸如 `TForm1.btnXXXClick` 这类的过程就是这样编写的。有关 Windows 控件编程的更多内容，后面第 10 章 会详细讲解。

`uInterface` 单元的完整代码如示例程序 6-4 所示。通过源代码我们可以看到，在 `uInterface` 单元的第 1 行是单元头，它定义了单元名为 `uInterface`。接着是 `interface` 部分，它含有 `uses` 和 `type` 子句。

`uses` 子句列出了在 `uInterface` 单元中使用的其它单元名称。因为 Windows 界面需要各种 VCL 组件、例程及系统函数的支持，所以它引用了不少外部单元。当我们从组件面板拖放控件到窗体时，Delphi 会自动在窗体单元中加入相应的单元引用，但动态创建控件时就需要我们手工加入这些单元引用。至于如何确定该控件来自那个单元，我们可以查看 Delphi 的在线帮助。

`type` 子句中，有我们声明的自定义数组类型以及自动生成的 `TForm1` 类型，`TForm1` 类型表示用于创建窗体对象的窗体类（在后面几章的面向对象程序设计和 Windows 程序设计部分会详细介绍类和窗体）。

在 `implementation` 部分同样也出现了 `uses` 子句，并列出了引用的单元 `uBinSearch` 和 `uSort`。为什么这两个单元不在 `Interface` 部分的 `uses` 子句中引用呢？这是因为这两个单元在自己的 `Interface` 部分已经引用了 `uInterface` 单元，如果 `uInterface` 单元的 `Interface` 部分中再引用他们，将导致循环引用的错误。所以，我们将引用放到了 `uInterface` 单元的 `implementation` 部分。

在 `implementation` 部分声明了两个 `TIntArray` 类型的变量 `a1` 和 `a2`，分别用于存放未排序的和已排序的数据，同样还声明两个 `TEditArray` 类型的变量 `edt1` 和 `edt2`，分别用于显示未排序的和已排序的数据。

随机函数为数组 a1 产生一组未排序整数，该组整数我们通过控件数组 edt1 显示在界面上。如果用户此时点击冒泡排序按钮，被触发的按钮事件将执行 TForm1.btnBSortClick 过程，该过程调用了冒泡排序函数 BubbleSort，并将排序后的结果赋值给数组 a2，同时通过控件数组 edt2 将结果显示在界面上。

同样，用户点击快速排序按钮将执行 TForm1.btnQSortClick 过程。不同的是该过程所用的快速排序过程 QuickSort 传递的是作为变量参数的数组。如果直接将数组 a1 作为实参传递，那么排序结果会改变数组 a1 原来的值。所以我们将数组 a1 赋值给 a2，再将 a2 作为实参传递。这样，排序结果就直接反映在数组 a2 中，不会影响到数组 a1。

当用户点击检索按钮时，将执行 TForm1.btnFindClick 过程。这段程序首先检查用户有没有输入要检索的数据。如果有输入，则检查检索的数据是否已排序，因为折半查找需要事先排序。对于已排序的数据则调用 binsearch 函数，并在输出显示的控件数组 edt2 中，用红色标记出对应检索出数据的那个控件。在如图 6-12 所示的运行界面上，我们可以看到，位于已排序那一组带标签的编辑框中，用红色标记出的标签为 210 的控件正是我们要搜索的数据 210。

示例程序 6-4 作为用户界面模块的 uInterface 单元源代码

```
unit uInterface;

interface

uses

  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ExtCtrls;

type

  TIntArray = array[0..9] of integer;
  TEditArray = array[0..9] of TLabelledEdit;

  TForm1 = class(TForm)
    btnRandom: TButton;
    btnBFind: TButton;
    btnFind: TButton;
    btnExit: TButton;
    edtFind: TEdit;
    Label1: TLabel;
    btnQFind: TButton;
    Label2: TLabel;
    Label3: TLabel;
    procedure btnRandomClick(Sender: TObject); //随机输入数据
    procedure btnBFindClick(Sender: TObject); //冒泡排序
    procedure btnFindClick(Sender: TObject); //检索
    procedure btnExitClick(Sender: TObject); //退出程序
    procedure Display; //显示结果
    procedure btnQFindClick(Sender: TObject); //快速排序
```

```
    procedure FormCreate(Sender: TObject); //初始化
private
    { Private declarations }
public
    { Public declarations }
end;

var
    Form1:TForm1;

implementation

uses uBinSearch, uSort;

var
    Sorted : boolean; //已排序标记
    a1,a2:TIntArray;
    edt1,edt2:TEditArray;
{$R *.dfm}

//随机输入数据
procedure TForm1.btnRandomClick(Sender: TObject);
var
    i:integer;
begin
    for i:=0 to 9 do
    begin
        a1[i]:=random(255); //随机产生新的一组数组值
        //我们用 edt1 数组元素对象来显示未排序数组 a1 的值
        edt1[i].EditLabel.Caption:=inttostr(a1[i]);
        edt1[i].Color:=rgb(a1[i],255,255);
    end;
end;

//冒泡排序
procedure TForm1.btnBSortClick(Sender: TObject);
begin
    a2:=BubbleSort(a1); //调用冒泡排序函数
    Sorted := true;
    Display;
end;

//快速排序
procedure TForm1.btnQSortClick(Sender: TObject);
```

```
begin
    a2:=a1;
    QuickSort(a2,low(a2),high(a2)); //调用快速排序过程
    Sorted := true;
    Display;
end;

//检索查找
procedure TForm1.btnFindClick(Sender: TObject);
var
    tempInt : integer;
    tempPosition : integer;
begin
    //如果没有没输入要检索的数据，就退出。
    if edtFind.Text='' then
    begin
        ShowMessage('没输入要检索的数据! ');
        exit;
    end;
    //折半查找，如果没有排序，需要先进行冒泡排序。
    if not Sorted then
    begin
        a2:=BubbleSort(a1) ;
        Sorted := true;
    end;
    Display;
    tempInt := strtoint(edtFind.Text);
    tempPosition := binsearch (a2,tempInt);

    if (tempPosition = -1) then
        ShowMessage(' 无此数据! ')
    else
        edt2[tempPosition].Color:=clred; //用红色标记出检索出的数据

end;

//退出程序
procedure TForm1.btnExitClick(Sender: TObject);
begin
    close;
end;

//显示排序结果
procedure TForm1.Display;
```

```
var
  i:integer;
begin
  for i:=0 to 9 do
  begin
    //我们用 edt2 的数组元素对象来显示排序结果，即数组 a2 的值
    edt2[i].EditLabel.Caption:=inttostr(a2[i]);
    edt2[i].Color:=rgb(a2[i],255,255);
  end;
end;

//初始化
procedure TForm1.FormCreate(Sender: TObject);
var
  i:integer;
begin
  for i:=0 to 9 do
  begin
    a1[i]:=random(255); //为数组元素随机产生 0~255 之间的整数
    //创建 edt1 数组元素对象，并初始化其在窗体的位置
    //该数组元素对象为 TLabeledEdit 类型的控件，我们用其显示未排序数组 a1
    edt1[i]:=TLabelledEdit.Create(self);
    edt1[i].Parent:=self;
    edt1[i].Left:=50;
    edt1[i].Top:=24*i+30;
    edt1[i].Height:=20;
    edt1[i].Width:=60;
    edt1[i].LabelPosition:=lpLeft;
    //利用 TLabeledEdit 控件来显示未排序数组 a1 的数值
    edt1[i].EditLabel.Caption:=inttostr(a1[i]);
    //利用 TLabeledEdit 控件的颜色来图形化显示未排序数组 a1 的数值
    edt1[i].Color:=rgb(a1[i],255,255);
    //创建 edt2 数组元素对象，并初始化其在窗体的位置
    //该数组元素对象为 TLabeledEdit 类型的控件，我们用其显示已排序数组 a2
    edt2[i]:=TLabelledEdit.Create(self);
    edt2[i].Parent:=self;
    edt2[i].Left:=300;
    edt2[i].Top:=24*i+30;
    edt2[i].Height:=20;
    edt2[i].Width:=60;
    edt2[i].LabelPosition:=lpLeft;
  end;
  Sorted := false;
end;
```

end.

另外，uInterface 作为窗体单元还带有一个由 Delphi 自动维护的 uInterface.dfm 文件，该文件的内容用于规范窗体以及其中各控件的属性。这也是窗体单元与普通单元的重要区别之一。

3. Program 主程序代码分析

作为一个 Windows 应用程序，“排序和查找算法演示程序”的 Program 主程序可以直接由 Delphi 自动为我们生成。因此，我们不需要编写代码。

前面我们提到，程序的基本结构是一个 repeat 循环结构，然后等待用户进行输入或操作。Delphi 在自动为我们生成 Program 主程序时已经考虑到了这个问题。只不过在 Windows 应用程序中，主程序创建主窗体后就进入了窗体的消息处理循环中，以便接受用户事件和系统事件所产生的消息，并调用对应的处理程序，直到应用程序结束为止。

在 Delphi 集成开发环境中，我们选择主菜单中的 Project | View Source 菜单项，即可打开 Program 主程序，其代码如示例程序 6-5 所示。

示例程序 6-5 Program 主程序 SortAndFind 的源代码

```
1: program SortAndFind;
2:
3: uses
4:   Forms,
5:   uInterface in 'uInterface.pas' {Form1},
6:   uBinSearch in 'uBinSearch.pas',
7:   uSort in 'uSort.pas';
8:
9: {$R *.res}
10:
11: //主程序代码实现
12: begin
13:   Application.Initialize;
14:   Application.CreateForm(TForm1, Form1);
15:   Application.Run;
16: end.
```

分析主程序 SortAndFind 的源代码可知，代码第一行指定了程序名，它由关键字 Program 声明，后面跟着主程序（也是项目）的名字 SortAndFind。关键字 Program 告诉编译程序，这是主程序。接着，在第 3 行使用了 uses 语句，引用了主程序所用到的所有单元。本例中包含了 Forms 单元，表示它支持窗体操作；另外 3 个是我们为这个应用程序所创建的程序单元，并使用 in 关键字，标明单元所对应的文件名称（可以包含文件的路径）。第 9 行的 {\$R *.res} 是一个伪指令，这是一种特殊的编译指令，它被括在注解中，但是伪指令名的前面加有“\$”符号。最后，就是 Program 的主体部分了，其代码包含在 begin 和 end 之间。对照上面的代码，可以看出这主要是用于对应用程序进行初始化，并创建 Form1 窗体。

4. 范例程序源文件

现在,如果我们打开这个范例程序文件所存放的文件夹,可以发现其中除了项目(主程序)文件 SortAndFind.dpr,单元文件 uInterface.pas、uSort.pas 和 uBinSearch.pas 之外,还有一大堆其他文件,如:与 SortAndFind 有关的 SortAndFind.res、SortAndFind.cfg、SortAndFind.dof 文件,与 uInterface 有关的 uInterface.dfm、uInterface.dcu、uInterface.ddp、uInterface.~dfm、uInterface.~pas 文件,以及编译后生成 SortAndFind.exe 可执行文件,等等。除了那些带有“~”符号的文件是系统自动生成的临时备份文件外(这些文件与去掉“~”号的文件一一对应),那么其它的文件到底是做什么用的呢?

一个 Delphi 应用程序的项目包含了各种源文件,这些文件分别对应着项目中所包含的窗体、单元、资源、选项等,所有这些信息都通过不同的源文件保存在磁盘中。在设计程序时,Delphi 创建了这些文件的大部分,但有一些文件也可以借助其它工具来创建或其它途径获得,如资源文件、帮助文件等。下面是程序设计时,Delphi 应用程序项目所包含的主要文件类型及其说明:

- **项目文件(.dpr)** Delphi 项目文件,用于保存窗体、单元等的信息,以及程序运行的初始化代码等,这种文件实际上就是包含了 Delphi 源代码的主程序文件。
- **项目配置文件(.cfg)** 保存项目配置,其文件名与项目名相同。
- **资源文件(.res)** 该二进制文件包含项目的图标,由 Delphi 不断更新和创建,用户一般不需要修改。其文件名与项目名相同
- **选项文件(.dof)** 含有当前项目选项设置的文本文件,其文件名与项目名相同。
- **单元文件(.pas)** Delphi 源代码文件,用于保存程序单元源代码,可以是与窗体有关的单元或是独立的单元。
- **窗体文件(.dfm)** 保存窗体或数据模块及其组件属性的二进制文件。其文件名与窗体单元名相同
- **类型库(.tlb)** 一种系统自动创建的或由类型库编辑器为 OLE 服务器端应用程序建立的文件。
- **包文件(.dpk)** 一种定义组件包的项目源代码文件。

而下面是应用程序编译后,Delphi 自动创建的文件:

- **执行文件(.exe)** 为程序编译后的可执行文件。
- **编译单元文件(.dcu)** 是在编译单元过程中产生的文件,即单元文件的编译版本。该文件会联接到最终的执行文件中。
- **动态链接库文件(.dll)** 为设计动态链接库时创建的文件。
- **ActiveX 文件(.ocx)** 一种特殊的 DLL,含有 ActiveX 构件或窗体。

5. 运行、测试程序

程序编译、生成之后,我们就可以在 Delphi 的集成开发环境中调试和运行了。有关程序的调试我们将在第 9 章中详细讲解。

现在,从 Delphi 集成开发环境的主菜单中选择 Run 菜单项或者直接按 F9,程序便运行起来了,其效果如图 6-12 所示。

窗体左边是一组带标签的编辑框,显示随机产生的未经排序的 10 个整数,数值越大,对应的色块颜色越浅。用户可以通过点击按钮来选择何种排序操作:冒泡排序还是快速排序。排序完毕后,数据按从大到小的降序顺序显示在右边的一组带标签的编辑框中,其对应的色块颜色由浅至深显示排序的效果。接着,就可以进行检索工作了,例如,在中间的编辑框中输入待检索的数据 210,点击检索按钮后,程序便给出查找的结果,用红色突出显示在对应的带标签的编辑框中。

通过对这个范例程序的剖析和研究，我们不仅学到了结构化程序设计的基本方法，还了解了一个完整的 Delphi 应用程序的开发过程，并在具体应用中对前面所学的知识进行了回顾和总结。同时也为我们进一步学习 Windows 程序设计奠定了基础。

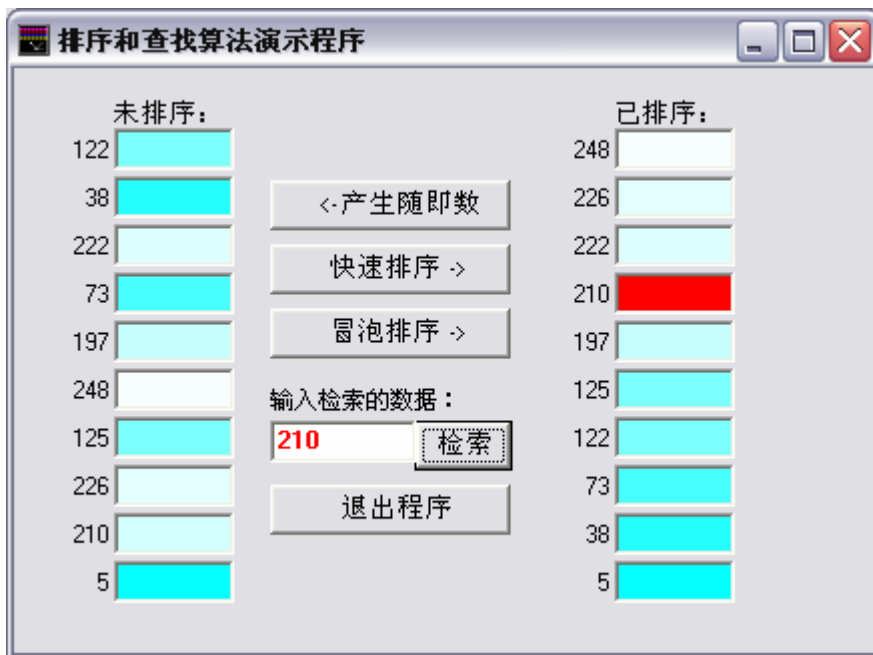


图 6-12 数据检索应用程序运行图

6.4 本章小结

- 一个 Delphi 应用程序可由多个称为单元的源代码模块组成。使用单元可以把一个大型程序分成多个逻辑相关的模块，并用来创建在不同程序中使用的程序库。每个 Delphi 应用程序都有一个首先执行的 Program 主程序，Program 作为主程序块将激活其他执行各种任务的所需的二级程序块——Unit 单元。
- Program 主程序就是 Delphi 中的项目文件，它是一个特殊的源代码文件。
- Delphi 应用程序中的单元 (unit) 实际上就是一个程序模块，因此单元是程序模块化的基础。单元一般由类型、常量、变量以及例程组成。将不同的单元引用、汇集，最终构成一个大的应用程序。
- Delphi 单元以单元头开始，接下来是 interface (接口部分)、implementation (实现部分)、initialization (初始化部分) 和 finalization (结束部分)。其中，初始化部分和结束部分是可选的。
- uses 子句用于声明引用的单元。uses 子句可以出现在项目文件中、单元文件的接口或实现部分。引用单元时要注意避免循环引用不当导致的语法错误。
- 标识符的作用范围就是它的生命期，因此标识符的作用范围决定了它的可访问性。标识符范围从它被声明开始，随包围声明的代码段结束而结束。
- 结构化程序设计要求程序具有合理的结构以保证和验证程序的正确性。结构化程序设计的三种基本结构是：顺序结构、选择结构和循环结构。其特征是：一个入口；一个出口；结构中每一部分都应当有被执行到的机会；没有死循环。
- 正确构造结构化程序的三条规则是：规则 1、任何矩形框可用两个顺序的矩形框代替；规则 2、任何矩形框可用任何简单控制结构代替；规则 3、规则 1 和规则 2 可

根据需要按任何顺序使用任意次。

- 结构化程序设计遵循模块化、自顶向下逐步求精的设计方法。
- 程序的基本结构是一个 `repeat` 循环结构，在循环中等待用户进行输入或操作。在 Windows 应用程序中，主程序创建主窗体后就进入了窗体的消息处理循环中，以便接受用户事件和系统事件所产生的消息，并调用对应的处理程序，直到应用程序结束为止。
- Delphi 应用程序项目所包含的文件类型主要有：项目文件（.dpr）、项目配置文件（.cfg）、资源文件（.res）、选项文件（.dof）、单元文件（.pas）、窗体文件（.dfm）、类型库（.tlb）、包文件（.dpk）。应用程序编译后，Delphi 自动创建的文件执行文件（.exe）、编译单元文件（.dcu）、动态链接库文件（.dll）、ActiveX 文件（.ocx）。

6.5 本章习题

复习题

1. 一个 Delphi 应用程序通常由哪两类不同的源代码模块组成？他们之间的关系是怎样的？
2. 编写 Program 主程序代码时，我们要注意哪些？
3. Delphi 单元的基本框架是怎样构成的？分析其各部分的组成和作用。
4. 结构化程序有哪些基本特征？
5. 构造正确的结构化程序有哪些规则？
6. 简述结构化程序设计的方法。
7. Delphi 应用程序项目所包含的主要文件类型有哪些？

测试题

8. 程序单元的结构是_____。
 - A、单元名称，引用，interface，类型声明，变量声明，implementation
 - B、单元名称，interface，类型声明，变量声明，引用，implementation
 - C、单元名称，interface，引用，类型声明，变量声明，implementation
 - D、单元名称，interface，类型声明，引用，变量声明，implementation
9. 下列文件中哪种被删除后不会影响到正常编程_____。
 - A、*.pas
 - B、*.dpr
 - C、*.dfm
 - D、*.dcu
10. 以下说法不正确的是_____。
 - A、单元的 Interface 部分用于声明常量、类型、变量、过程、函数等，这些声明对于其它使用了该单元的单元、项目、库、包等是可用的。
 - B、单元的 implementation 部分是代码实现部分，不能用于声明常量、类型、变量、过程、函数等。
 - C、单元的 initialization 部分和 finalization 部分是可选的，他们可有可无。
 - D、只要不出现语法错误，单元还可以互相引用。

11. Delphi 单元以单元头开始, 接下来是_____。
- A、interface 部分、implementation 部分、initialization 部分和 finalization 部分
 B、interface 部分、uses 部分、type 部分和 implementation 部分
 C、interface 部分、uses 部分、type 部分、var 部分和 implementation 部分
 D、接口部分、引用部分、声明部分和实现部分

12. 结构化程序设计采用了_____的设计原则。

- A、模块化、自顶向下逐步求精
 B、模块化、自下而上, 逐步积累
 C、反复迭代, 逐步扩展
 D、层次化, 逐层细化

13. 下面程序几段有关单元引用的代码中, 出现循环引用错误的是_____。

A、

Unit Unit1;	Unit Unit2;	Unit Unit3;
Interface	Interface	Interface
uses Unit2;	uses Unit3;	const c=1;
const a='b';	const b='c';	Implementation
Implementation	Implementation	const d=2;
begin	begin	begin
...
end.	end.	end.

B、

Unit Unit1;	Unit Unit2;	Unit Unit3;
Interface	Interface	Interface
uses Unit2;	uses Unit3;	const c=1;
const a='b';	const b='c';	Implementation
Implementation	Implementation	uses Unit1;
begin	begin	const d=2;
...	...	begin
end.	end.	...
		end.

C、

Unit Unit1;	Unit Unit2;	Unit Unit3;
Interface	Interface	Interface
uses Unit2;	Implementation	const c=1;
const a='b';	uses Unit3;	Implementation
Implementation	const b='c';	uses Unit1;
begin	begin	const d=2;
...	...	begin
end.	end.	...
		end.

D、

```

Unit Unit1;           Unit Unit2;           Unit Unit3;
Interface             Interface             Interface
uses Unit2;          uses Unit3;          uses Unit1
const a='b';         Implementation     const c=1;
Implementation       const b='c';         Implementation
begin                begin                const d=2;
...                  ...                  begin
end.                  end.                  ...
                    end.

```

14. 设有若干个过程(Procedure), 且它们之间的嵌套关系如下:

```

Procedure P0;
  Procedure P1;
    Procedure P11;
      begin
        ...{P11 代码}
      end;
    Procedure P12;
      Procedure P121;
        begin
          ... {P121 代码}
        end;
      begin
        ... {P12 代码}
      end;
    begin
      ... {P1 代码}
    end;
  Procedure P2;
    begin
      ... {P2 代码}
    end;
  begin
    ... {P0 代码}
  end;

```

关于这些子程序之间的相互调用, 下面说法不正确的是_____。

- A、P1 和 P2 之间, P11 和 P12 之间可以互相调用, 但可能要用到 Forward 指示字。
- B、P0 可以调用 P2, P1 可以调用 P11
- C、P121 可以调用 P11
- D、P0 可以调用 P12

15. 有如下程序:

```
program test ;
```

```

{$APPTYPE CONSOLE}
uses
  SysUtils;
var
  x:real;
  a,b,c:integer;
procedure p1(a,b,c:real);
var
  x:real;
begin
  x:=a+b+c;
  writeln(x);
end;

begin
  x:=3;
  b:=4;
  readln(a);
  p1(a,a,b);
end.

```

则下面的说法中正确的是：_____

- A、主程序 test 中 x 的作用范围包含过程 p1
- B、过程 p1 中 x 的作用范围包含主程序 test
- C、主程序 test 中 x 的作用范围和过程 p1 中 x 的作用范围相同
- D、主程序 test 中 x 的作用范围不包含过程 p1

16. 下列哪种文件类型中包含有 Delphi 应用程序的 program 主程序_____。

- A、.pas
- B、.dcu
- C、.dfm
- D、.dpr

17. 与非结构化方法编写的程序相比，结构化程序设计的优点是_____。

- A、可避免死循环
- B、结构清晰，程序更容易理解，
- C、容易保证程序和算法的正确性
- D、以上都是

练习题

18. 指出以下程序段的错误：

```
//----- ex1 单元-----
```

```
unit ex1;
interface
  function exam1(x,y:Integer):integer;
  ...
implementation
  uses ex2;
  function exam1(x,y:Integer):integer;
  begin
    ...
  end;
  ...
end.

//----- ex2 单元-----
unit ex2;
interface
  procedure exam2(i,j:Integer);
  var
    x:integer;
implementation
  procedure exam2(i,j:Integer);
  begin
    ...
    x:=exam1(i,j);
    ...
  end;
  ...
end.
```

19. 设计一个电话号码管理程序。要求实现新增、修改、查询以及按照不同的顺序显示等功能。（提示：使用记录类型的数组存储电话号码）
20. 用结构化设计方法设计一个大赛评分应用程序。要求包含：
- ◆ program 主程序。
 - ◆ 选手检录单元：录入、存储参赛选手的编号、姓名等信息。
 - ◆ 评分统计单元：10 名评委打分，去掉一个最高分、去掉一个最低分，求出平均分作为该选手的最后得分。
 - ◆ 分数显示单元：提供 2 种分数显示选择，即按照选手编号顺序或按照分数高低顺序显示选手成绩。