新世纪高职高专教改项目成果教材

高等职业教育技能型紧缺人才培养试用

软件技术基础

——离散数学、数据结构、C++编程实训

来可伟　编

高 等 教 育 出 版 社

内容提要

本书是教育部新世纪高职高专教育人才培养模式和教学内容体系改革与建设项目成果，是组织有关教育部高职高专教育专业教学改革试点院校编写的。

本教材系统地介绍了离散数学、数据结构、C++语言编程三个领域的知识。在理论方面，将离散数学作为编程技术必要的数学常识进行了深入浅出的介绍，以培养学员将实际问题抽象为数学表达式的抽象思维能力。在实验技能方面，采用基于案例（case–based）的方式，通过完整的编程全过程练习，使学员能按现代软件工业一线编程人员的要求掌握编程的基本技能，养成良好的规范化作业的习惯。

教材对各种常见数据结构采用了基于 C++语言类模板的实现方法，不但方法新颖、充分体现基础理论对编程的指导作用，而且有很高的实用价值。

本书适合于高等职业学校、高等专科学校、成人高校、示范性软件职业技术学院、本科院校举办的二级职业技术学院，也适合本科院校、继续教育学院、民办高校及技能型紧缺人才使用，还可供计算机专业人员和爱好者参考使用。

图书在版编目(CIP)数据

软件技术基础 / 来可伟编．—北京：高等教育出版社，2004.4

ISBN 7–04–014765–3

Ⅰ．软… Ⅱ．来… Ⅲ．软件—基本知识Ⅳ．TP31

中国版本图书馆 CIP 数据核字（2004）第 026944 号

策划编辑 冯 英 责任编辑 关 旭 封面设计 王凌波
版式设计 胡志萍 责任校对 杨雪莲 责任印制

出版发行 高等教育出版社 购书热线 010–64054588
社 址 北京市西城区德外大街 4 号 免费咨询 800–810–0598
邮政编码 100011 网 址 http:// www.hep.edu.cn
总 机 010–82028899 http:// www.hep.com.cn

经 销 新华书店北京发行所
印 刷

开 本 787×1092 1/16 版 次 年 月第 1 版
印 张 16.25 印 次 年 月第 次印刷
字 数 390 000 定 价 26.60 元（含光盘）

出 版 说 明

为认真贯彻《中共中央国务院关于深化教育改革全面推进素质教育的决定》和《面向 21 世纪教育振兴行动计划》，研究高职高专教育跨世纪发展战略和改革措施，整体推进高职高专教学改革，教育部决定组织实施《新世纪高职高专教育人才培养模式和教学内容体系改革与建设项目计划》（教高[2000]3 号，以下简称《计划》）。《计划》的目标是："经过五年的努力，初步形成适应社会主义现代化建设需要的具有中国特色的高职高专教育人才培养模式和教学内容体系。"《计划》的研究项目涉及高职高专教育的地位、作用、性质、培养目标、培养模式、教学内容与课程体系、教学方法与手段、教学管理等诸多方面，重点是人才培养模式的改革和教学内容体系的改革，先导是教育思想的改革和教育观念的转变。与此同时，为了贯彻落实《教育部关于加强高职高专教育人才培养工作的意见》（教高[2000]2 号）的精神，教育部高等教育司决定从 2000 年起，在全国各省市的高等职业学校、高等专科学校、成人高等学校以及本科院校的职业技术学院（以下简称高职高专院校）中广泛开展专业教学改革试点工作，目标是：在全国高职高专院校中，遴选若干专业点，进行以提高人才培养质量为目的、人才培养模式改革与创新为主题的专业教学改革试点，经过几年的努力，力争在全国建成一批特色鲜明、在国内同类教育中具有带头作用的示范专业，推动高职高专教育的改革与发展。

教育部《计划》和专业试点等新世纪高职高专教改项目工作开展以来，各有关高职高专院校投入了大量的人力、物力和财力，在高职高专教育人才培养目标、人才培养模式以及专业设置、课程改革等方面做了大量的研究、探索和实践，取得了不少成果。为使这些教改项目成果能够得以固化并更好地推广，从而总体上提高高职高专教育人才培养的质量，我们组织了有关高职高专院校进行了多次研讨，并从中遴选出了一些较为成熟的成果，组织编写了一批"新世纪高职高专教改项目成果"教材。这些教材结合教改项目成果，反映了最新的教学改革方向，很值得广大高职高专院校借鉴。

新世纪高职高专教改项目成果教材适用于高等职业学校、高等专科学校、成人高校及本科院校举办的二级职业技术学院、继续教育学院和民办高校使用。

高等教育出版社

2002 年 11 月 30 日

前　言

长期以来，在初级和中级计算机教育，如高等职业教育的计算机专业以及非信息专业的计算机教育中，通常只教授编程语言，不教授相关的理论知识，这是一个弊病，因为程序编写仅仅是软件技术领域的一小部分，而软件技术是一项具有数学严密性的技术。科学技术的知识结构一般包括三个方面的内容：基础理论、技术基础理论和专业技术。对于软件技术，可以说离散数学是其基础理论，数据结构与算法设计是其技术基础理论。若对软件技术赖以发展的这些数学基础理论和专业理论的系统缺乏了解，所进行的编程实践就会成为盲目的实践，所编写的程序也许有漂亮的外观，但内部的数据和控制逻辑却是杂乱无章的。软件技术又是一项具有工程严密性的技术，如果没有形成规范化编程作业的观念、技能和作业方式，所编写的程序就会漏洞百出，缺乏文档，难于更新与维护。因此，近年来各方面一直在努力加强以上两方面的教学，本教材就是我们努力的结果。正如教材副标题“离散数学、数据结构、C++编程实训”所表明的，本书强调两点：一是从指导编程作业的目的出发，掌握离散数学和数据结构的一些常识；二是通过完整的编程作业训练，逐步掌握科学的、严密的程序设计方法，养成规范化作业的习惯。

本书主要面向信息专业高等职业教育，也可应用于非信息专业的其他工程专业本科计算机教育。本教材最初是以网络多媒体教材形式提供的，以便于学员自学一些较难理解的概念和编写程序的具体操作步骤。这次改编，又结合近年来的教学经验，在形式和内容上都作了多处重要修改，原有的多媒体素材经修改后作为书后配盘提供给读者。

虽然本书强调学习基本数学概念，但学习本书并不要求具备高深的数学预备知识，只要具备高中程度的数学基础即可。非信息专业的人员经过努力，都能掌握计算机程序开发的基本理论和方法。学习计算机程序编制并不需要特殊才能，只要按照科学的、严密的方法和步骤，养成规范化的编程作业习惯，成为一个好的编程人员并不困难。另外，本书还设计了一些选读内容，目的是为了扩大读者的知识面，这些章节以*号标识，学员可根据自身情况有选择地阅读这些带*号的章节。

本书由张志浩教授审阅，根据他的意见，书中做了多处重要修改，特此致谢。

本书的多媒体版本（书后配盘）得到了同济大学网络学院和同济大学高等技术学院的大力支持，特此致谢。

目　录

第1章 绪 论

1.1 本课程的知识结构

工程技术学科的知识组成一般都包括3个方面的内容：基础理论、技术基础理论和专业技术。软件技术学科的3个知识层次分别为离散数学、数据结构与算法设计以及用某种编程语言进行编程作业。

计算机所能处理的信息称为数据。在计算机发展初期，计算机主要用于数值计算，处理的数据主要是数值数据，如整数、分数、有理数、无理数、实数、复数等。随着计算机科学和技术的发展，所处理的数据对象的概念随之拓宽，文字、图形、图像、语音、视频等人类感官所能感受的各种信息，都可成为数据对象，这就是非数值计算领域。

离散数学是现代数学的一个重要分支，与许多我们所学过的数学不同，计算机在非数值计算领域所处理的很多数据都具有离散性的特点。所以，对离散数学的学习和研究是计算机软件技术的数学理论基础。

用计算机程序解决实际问题时，必须解决两个方面的问题：

- 把要解决的问题表示为计算机所能处理的数据；
- 将解决问题的过程表示为计算机所能执行的步骤。

前者属于数据结构所研究的范畴，后者属于算法设计所研究的范畴。因此可以说，对数据结构和算法设计的学习和研究是计算机软件技术的技术基础理论。

程序是计算机解决实际问题的基本形式，因此，能用某种计算机语言熟练地编程是软件专业人员必须掌握的专业技术。

没有理论指导的实践是盲目的实践，如果对软件技术赖以发展的基础理论缺乏系统的了解，就不能成为一个合格的软件技术人员。所以，本教材强调从掌握基础理论的有关常识开始学习软件技术，但考虑到一般软件技术人员从事实际工作的需要，教材在阐述有关理论时将把重点放在一些与程序设计有密切关系的基本概念以及如何应用其指导程序设计上。同时，为兼顾知识的完整性，教材对相关概念也有必要的简介，但将其列为选读内容，使学员有充分的学习自主权。

下面简要介绍组成软件技术学科的3个领域中本教材所涉及的内容。

1.2 离散数学

离散数学是指导程序设计的主要基础理论。数学永远是工程学科生存和发展的基础，计算机程序设计更不例外。长久以来有一种误解，认为掌握了编程语言，就等于掌握了软件开发技术的全部。因此，许多有着漂亮界面的程序，其内部的数据组织却是杂乱无章的，其内部的控制逻辑也是混乱或者不严密的，其原因就是编写人员不注意应用，或不知如何应用指导程序设

计的数学理论。所以，本教材强调在学习编程前应首先了解基本的数学理论。

离散数学是现代数学的一个重要分支，它是研究如何描述和处理各种离散性数据对象的科学，涉及领域十分广泛。因此，并不仅仅只有软件技术人员才需要学习离散数学。在国外大学中，集合论、数理逻辑、图论已经是从事社会科学、教育学、商业、艺术等非理工类专业学生必须掌握的数学知识。本书用了一些源自日常生活的有趣例子来说明严密的数学思维过程，这种思维方式对于解决任何问题都是十分重要的。

本教材根据软件开发的一般需要，着重介绍命题逻辑、集合论、图论与树几个方面的内容，其中命题逻辑是关于如何将待解决的问题表示为程序中的逻辑运算的，集合论、图论与树则是关于如何将待处理的数据对象表示为计算机所能处理的形式的。

为了便于学员理解，教材对有关数学概念的阐述尽量通过实例进行深入浅出的说明，学员只要有高中数学基础，就可以掌握这些概念。结合实例阐述概念不但使学员容易理解概念，而且可以对如何应用这些概念去解决实际问题有直观的认识。

1.3 数据结构与算法设计

数据是信息的载体，在计算机科学中，数据一般指计算机程序能识别和处理的符号。数据一般又分为数值性数据和非数值性数据两大类，数据结构是数据（包括数值性和非数值性）之间内在关系的数学描述。本教材应用面向对象的概念来研究数据结构。从面向对象的观点来看，具有相同性质的数据抽象为数据对象，而那些具体的数据则是数据对象的成员或元素，数据结构描述的就是数据成员以及成员之间的关系。这种关系又分为逻辑关系和物理关系。数据结构的逻辑关系是面向问题的，是数据功能的描述，如存放学生数据的表格需要哪些栏目；数据结构的物理关系描述数据在计算机中的具体表达方式，如用整数还是用字符表示等。

虽然数据结构和算法设计研究领域的内容十分丰富，本教材只着重介绍其中应用最广的一种线性数据结构——序列（list），另外也将简要介绍根树和图。本教材对所介绍的数据结构的实现完全基于 C++语言的类模板（class template）和运算符重载（operator overload）方法。用类模板编写数据结构的最大好处是使关于算法的数学描述和具体程序能完全对应起来，教材中提供的范例程序不仅使学员可以通过观察程序的实际运行过程来学习有关数学概念，还可以直接应用这些范例程序解决各种实际问题，这也是本教材的特点之一。

1.4 C++语言

计算机编程语言一般可分为常规编程语言和人工智能编程语言，常规编程语言又分为面向过程的和面向对象的。面向过程语言有 FORTRAN、BASIC、C 等，面向对象语言则包括 C++、SmallTalk 等。

面向对象编程的基本出发点是尽可能地按照人类的思维方式分析和解决问题。可以将其特点概括为 3 条：

- 封装性
- 聚集性

- 继承性

客观世界是复杂的，而计算机处理信息的能力是有限的。为了能在计算机能力所及的范围内尽可能如实地反映客观世界，需要对事物进行抽象。抽象是一种有目的的“取主去次”的过程，将那些对当前问题来说最重要的信息与其他无关紧要的信息分开，并依据这些主要特性对事物进行分类，这就是对象的封装性。对象的封装使人们可以把握事物的共性，以区分不同类的事物。对于复杂的事物，可用两种方法来分解其复杂性，一种是用对象的聚集性把它分解为比较简单的类；另一种是用对象的继承性把它分解为抽象程度较低的子类。

通过学习 C++语言可以掌握面向对象程序设计的方法与技术。C++语言是由美国 Bell 实验室（AT&T 实验室的前身）的 B. Stroustrup 在 C 语言的基础上开发的，是目前应用最广的面向对象编程语言之一。C++的名称总是给人以误导，总以为学习 C 语言是学习 C++语言的基础，其实两者的设计理念完全不同。C 语言是面向过程的，C++是面向对象的。教学实践表明，许多学过面向过程编程语言（如 C、BASIC）的学员，在对面向对象方法的理解和掌握上往往比没有学过任何编程语言的学员更困难。因此建议在使用本教材前不必先学习其他任何编程语言。本教材采用的是基于案例的教学方法，是以没有任何编程基础的学员为对象的，入门不难，也有利于深造。

1.5 学习要求和方法

虽然本教材涉及传统上由不同课程讲述的内容，但并不是简单地把几本教材合并为一本，而是力求从内容编排上将数学概念与其实践应用有机结合起来，以充分体现理论对实践的指导意义。具体方法是把离散数学、数据结构学科领域中对程序设计最具指导意义的一些基本概念抽取出来，围绕理论和程序设计的关系进行阐述。同时，在程序设计的教学内容中加强了关于如何应用这些概念的论述。近年来的教学实践证明，增加关于离散数学和数据结构常识的教学内容以后，不仅没有增加学习难度，反而使随后的程序设计教学变得比较容易，学员所设计的程序质量也得到了很大提高。

教材副标题“离散数学、数据结构、C++编程实训”表明，本教材强调要同时进行两个方面的教学：一是要求学员掌握一点离散数学和数据结构的常识，能自觉地应用理论指导编程实践；二是要求学员尽可能按照软件工业一线编程人员的要求完成编程作业，从中掌握科学的、严密的程序设计方法，养成规范化作业的概念和技能。具体要求如下：

- 在基础理论方面，要求学员熟悉离散数学的基本概念，培养一定的抽象思维的能力，着重掌握应用这些基本概念解决实际问题的方法。在编排形式上，对于一些重要的或者难于理解的概念，专门设立了标题为“导读”的内容，进行专门讲解。教材特地提供了一个名词索引，将概念出现的章节一一列出来，以便于掌握概念之间的内在联系。
- 在实践技能方面，采用了基于案例的方法，目的是使学员完整地掌握开发软件的方法、步骤、技术，养成实施规范化工程作业的习惯。为此，教材中特别以“规范化编程实践”的标题列出了一些建议遵守的准则。此外，还对编程中经常易犯的错误以“经验与提醒”的形式作了特别提示。

对有关数学的习题，教材大部分都给出了参考答案和解题提示；对有关编程作业的习题则

提供了范例程序。软件技术领域的许多问题都存在不止一种解决方法，因此希望学员不要束缚自己的思路，要充分发挥自己的分析能力，这对于解决实际的能力的培养尤为重要。

通过团队协作分析和解决问题是将来从业所必须具备的能力，因此建议学员通过小组讨论和交流，对习题进行分析。无论是对培养个体应用知识的能力，还是对培养团队工作精神来说，这都是一种好方式。

本教材建议教学时数为 60。建议学时分配为：用 18 学时学习离散数学理论部分；30 学时用于学习 C++语言和进行编程作业的实践训练（数据结构的内容已穿插在其中）；最后用 12 学时集中完成课程作业。大量的编程练习对学习软件技术来说十分关键，因此课外练习编程时间应不少于 60 学时。

1.6 书后配盘

在书后配盘中有本教材的网络多媒体版本。将光盘放入计算机光驱中，点击光驱设备符号或者用鼠标右键点击光驱设备符号，打开右键菜单，从中选择“自动播放”，教材首页便可自动在浏览器中打开。

将光盘中全部内容拷贝到 Web 服务器中，还可以成为一个网站。

与书面教材相比，光盘有以下优点：多媒体形式中有许多便于自学的素材，如在计算机上进行编程作业过程的视频演示以及对一些数据结构的工作原理的动画描述等；多媒体教材最大的优势还在于它提供了一种立体化的知识结构，通过点击网页中的链接，可将有关概念融会贯通。需要注意的是，该多媒体教材要求学时较多，凡光盘内容超出本教材之处均作为选学内容。

1.7 软件技术教育

为了帮助学员理解如何学习软件技术，下面对软件技术教育做些介绍。技术（technique）这个词在有些词典里的意思是指完成技术工作细节所必须具备的能力。例如，机械加工业的一线工作人员必须具备按统一的工艺流程加工零件的技能，才能生产出合格产品。程序编写员就是软件行业的一线工作人员，他们必须掌握的技术就是按规范化的作业方法，编写标准化的程序代码，提供规范化的文档和进行规范化的测试。

很久以来一直有一种误解，以为技巧在计算机软件开发中起着重要作用。诚然，在计算机发展的早期，巧妙的设计可以克服资源（内存和 CPU）不足所带来的一些限制。但是，现在的程序设计已有完善的理论作为指导，软件开发也正在以越来越大的工业化规模进行。在这种形势下，对编写程序的个体来说，重要的不再是如何巧妙地利用计算机资源，而是用规范化的作业方式，编写出高度规范化的程序代码。

现在很多书中把软件技术和软件工程混为一谈，但它们是有区别的。第一，像所有其他工程一样，软件工程是组织和管理群体的方法，注重的是完成任务的群体活动，包括任务分解、活动组织和协调；而软件技术是软件开发一线编程人员所必须具备的、用规范化作业方式编写出高度规范化的程序代码的技能。第二，软件技术和软件工程是由不同的职能人员去实施的。对一般软件技术人员来说，他们的职责在大多数情况下只是按设计说明编写程序代码，而不是

从头策划一个项目。第三，软件技术和软件工程的区别还体现在两者质量保证措施不同。软件技术所关注的是具体程序代码的质量（即数据定义和算法的正确性），而软件工程关注的是软件开发全过程的质量。因此，软件技术教育的重点就是使从事软件开发的所有个体都充分认识软件开发作业的工程性质，摆脱手工作坊式的软件开发模式，掌握从事规范化编程作业的观念和技能。

下面摘录了一篇关于印度软件业的报导，从中可以进一步体会什么是个人应掌握的软件技术，什么是软件工程应该考虑的问题，从而更有目的地学习软件技术。

看看印度软件人(原载 2000 年 7 月 17 日文汇报，原著者：戴表)

（这是一位中国软件公司老板对印度软件人所下的评语。从中我们或许可以知道一些近年来印度软件业迅速在世界崛起的原因）

我在工作中，接触到印度软件人和他们开发出来的软件。从整个体系架构上看，他们的作品非常清晰，按照我们的要求实现了全部功能，而且相当稳定，但是打开具体代码一看，拖沓冗长。我们自己的一些程序员据此说他们水平低。但是，印度人能把软件整体把握得很好，并弄出相当好的设计文档。我们对某些特定的开发工具可能是精通的，却无法保证把一个软件稳当、完整地开发出来。

举个简单的例子：软件中需要一个列表，用来表示我们处理的事务。该表在业务繁忙的时候将变得很大。中国人总是用双向链表，抱着《数据结构》书在那里写链表的类，而我们雇的那些印度人根据情况，有时就抛弃链表，直截了当开一个大数组。为什么印度人不用链表？他们说，你们的设备最少都装备了 512MB 内存，占用一些没什么，而数组方式访问方便、效率高。他们做事情就这么简单明了，跟哥伦布竖鸡蛋一样。

印度人并不随意马虎

我对印度软件业的几点感受是：

1．流程重于项目。

2．质量监察独立于研发部门，专门检查开发部门的开发流程是不是按照既定流程走。如果质量管理人员觉得流程不对，他会直接上报高层，项目肯定就此停止。

3．所有的材料(包括草稿)都有文档。

4．准备工作做得很充分，详细文档要求达到拿到这个文档就可以编码的程度。一般写文档时间占 60%，而编码时间相对较少。

5．有各种详细的 Review（同行评审），包括项目组内、项目组之间、客户的……

6．计划详细，几十天的大项目竟能精确到小时级。

让中国高手惊呆了

印度人做事的总体风格很书面、很理论。我们招聘印度人，给应聘者出了一份与国内差不多的试卷，有基本概念和编程题目。等到他们完成后，我们这些自认的中国高手简直惊呆了！他们做的编程题目简直像是有统一答案，程序结构、注释、变量命名就不用说了，连表达方式都极其类似！反观所谓的中国高手，每个人都有自己的一套。到了新的岗位，先把前任的程序贬损一通，然后自己再开发有更多问题的代码来代替。我曾经统计了一下我的公司，一个软件中有 4 个以上的 CSocket 版本。每个人都觉得别人做得差，自己再搞一套。中国人就是这个样子，还振振有辞“我们这样有创造性”。

不依赖任何一个个人

印度软件公司的编程人员流动率(包括内部项目之间的流动)高达30%,但他们可以让高中生编代码,可以想见其整体的文档水平之高。他们的产品不依赖任何一个个人,谁都可以立即辞职,而产品的开发还会正常进行。

习 题

1. 为什么说数学是工程技术的基础?
2. 软件技术和软件工程有什么区别?
3. 说说自己心目中的软件开发人员应该具备什么素质?
4. 印度软件人的故事给自己学习软件技术以什么启示?

第2章　命 题 逻 辑

知识点：

命题逻辑是谓词逻辑的子集，它们都是指导程序设计的重要理论。从便于学习的角度出发，一般教材都是从命题逻辑开始介绍数理逻辑，就像在学习实数概念以前先学习自然数的概念一样。本章的学习目的为：

① 熟悉命题逻辑的基本概念并能熟练进行命题逻辑演算。

② 能对实际问题进行抽象，建立命题模型。既能将自然语言表示为命题公式，又能用自然语言描述命题表达式。

数理逻辑是用数学方法研究人们思维活动的一门学科，它将研究对象的形式关系抽象为可运算的符号体系，因此数理逻辑又称为符号逻辑，它是计算机应用和理论研究的基本工具，内容很丰富。本教材针对非计算机专业工程技术人员设计程序的需要，着重介绍命题逻辑的一些最基本的概念，以便能应用它们来设计程序逻辑控制语句（见 6.4.7 节）。

2.1　命题逻辑的基本概念

所有的数学学科都是建立在定义、定律、定理的体系基础上的。由于数学学科本身的特点，阐述定义、定律和定理的语句显得要比一般自然语言中的语句难于理解，因此，除了反复阅读有关文字叙述以外，更重要的是通过将概念应用于实际问题去逐渐加深理解。特别是命题逻辑，与日常生活有密切关系，所以一定要多联系实际来理解有关概念。

2.1.1　命题

语句是人们进行思维推理、表达内容的基本形式，在命题逻辑中称这种表达判断的陈述句为命题。命题是命题逻辑中的一个语法单位。

定义 2-1：

表示判断的陈述句称之为命题，其内容的真实性必须是惟一确定的，称为命题的值或真值。

定义 2-2：

原子命题是不能再分解的命题。真值确定的原子命题称为命题常量，真值可变化的原子命题称为命题变元。

命题不同于自然语言之处在于，它必须有惟一确定的真值。所谓真值是指命题语句所陈述的内容为真或为假。当一个命题为真时，用 T（或 TRUE）表示；若命题的值为假，则用 F（或 FALSE）表示。为了便于程序设计，一般也用二进制数 1 表示命题为真，0 表示命题为假。

命题一般用大写字母或带下标的大写字母来表示，如 P、P_1 等。命题变元没有确定的真假

值，在命题演算中可以代表任何命题。

例 2-1：

以下是命题的例子。

A：地球是球形的(T)

P_1：太阳绕着地球转(F)

Q_2：3+3=6 (T)

而 $x+3>0$ 不是命题，因其没有确定的真假值。

例 2-2：设 P 为命题变元，在没给它赋值之前，它没有值。

如果令

P：月球上没有生物

则 P 的值为真。

而如果令

P：地球是方的

则命题变元 P 的值为假。

导读：

① 若将命题变元比做普通数学里的变量，则命题常量就相当于普通数学中的常量。

② 所有的数学学科都是有共性的，通过对比以往所学过的数学概念，可以帮助理解离散数学的有关概念。

2.1.2 复合命题

学习数学最重要的就是学习如何进行运算。如普通数学用加、减、乘、除运算符把运算数组成数学运算式，用根号表示开方运算。在命题逻辑中，原子命题通过命题运算符构成的逻辑运算式，称为复合命题。主要的命题运算符有 6 个：否定、合取、析取、异或、蕴含、等价。

复合命题的值与组成它的原子命题的值有关，它们之间的取值关系，也就是运算符的运算规则可用真值表来表示，它可直观地描述命题运算的结果，是进行命题运算的重要工具。

定义 2-3（否定运算符¬）：

设 P 为一命题，则命题¬P 称为“P 的否命题”，读作“非 P”，符号¬称为否定运算符。规定当且仅当 P 是假时¬P 是真。

命题¬P 取值的规则如表 2-1 所示。

表 2-1 表示命题否定运算规则的真值表

P	¬P
0	1
1	0

例 2-3：

（1）给定命题 A：今天气温低于摄氏 30 度

则命题¬A 的意思是：今天气温不低于摄氏 30 度。

（2）给定命题 B：机器运行不正常

则命题$\neg B$的意思是：机器运行正常。

否定运算符的作用只是否定所依附的语句，而不管其原意如何。因此表示否定内容的语句不一定就是否定命题，如例 2-3 中命题 B 本身表达的是一个否定的内容，而其否定式表示的却是一个肯定的内容。

定义 2-4（合取运算符$\wedge$）：

设 P、Q 为两个命题，则命题 $P\wedge Q$ 称为“ P、Q 的合取 ”，读作“ P 与 Q ”，符号$\wedge$称为合取运算符。合取运算的规定是：当且仅当 P 和 Q 同时是真时，$P\wedge Q$ 为真。

命题 $P\wedge Q$ 的取值规则如表 2-2 所示。

表 2-2 表示命题合取运算规则的真值表

P	Q	$P\wedge Q$
0	0	0
0	1	0
1	0	0
1	1	1

例 2-4：给定命题

A：地球是方的

B：今天晴天

则命题 $A\wedge B$ 的意思是：地球是方的且今天是晴天。

本例特意用上述不成立的事实组成命题，以强调复合命题关注的只是形式。

定义 2-5（析取运算符$\vee$）：

设 P、Q 为两个命题，则命题 $P\vee Q$ 称为“ P、Q 的析取 ”，读作“ P 或 Q ”，符号$\vee$称为析取运算符。析取运算的规定是：当且仅当 P 和 Q 中至少一个为真时，$P\vee Q$ 为真。

命题 $P\vee Q$ 的取值规则如表 2-3 所示。

表 2-3 表示命题析取运算规则的真值表

P	Q	$P\vee Q$
0	0	0
0	1	1
1	0	1
1	1	1

例 2-5：给定命题

A：地球是方的

B：今天晴天

则命题 $A\vee B$ 的意思是：地球是方的或者今天是晴天。

与前例一样，本例子旨在说明复合命题关注的只是形式，所以也有意选取了上述似乎并不成立的事实组成命题。

定义 2-6（异或运算符∇）：

设 P、Q 为两个命题，则命题 $P\nabla Q$ 称为“ P、Q 的异或 ”，读作“ P 异或 Q ”，符号∇称为异或运算符。异或运算的规定是：当且仅当 P 和 Q 中只有一个为真时，$P\nabla Q$ 为真。

命题 $P\nabla Q$ 的取值规则如表 2-4 所示。

表 2-4 表示命题异或运算规则的真值表

P	Q	$P\nabla Q$
0	0	0
0	1	1
1	0	1
1	1	0

例 2-6：设有命题

Q：A 是男性

R：A 是女性

则异或命题 $R\nabla Q$ 的意思是：A 不是男性就是女性。

自然语言中的“或”有两种解释：一种是相容的，即两者可同时为真（原文为 inclusive OR）；一种是不相容的，即两者不可同时为真（原文为 exclusive OR）。有的教材将前者称为“简单或”、“可兼或”。析取运算符表示的是相容的情况，异或运算符表示的则是不相容的情况。

定义 2-7（蕴含运算符→）：

设 P 和 Q 为两个命题，则命题 $P\rightarrow Q$ 称为“P 蕴含 Q”，符号→称为蕴含运算符，P 称为前提，Q 称为结论。蕴含运算的规定是：当且仅当 P 是真和 Q 是假时，$P\rightarrow Q$ 为假，其余情况 $P\rightarrow Q$ 皆为真。

命题 $P\rightarrow Q$ 的取值规则如表 2-5 所示。

表 2-5 表示命题蕴含运算规则的真值表

P	Q	$P\rightarrow Q$
0	0	1
0	1	1
1	0	0
1	1	1

蕴含命题相当于语句“如果……就……”，但仅表示充分条件，而不是必要条件。所以 P 取假时，不论 Q 为何值，$P\rightarrow Q$ 总为真。只有 P 为真而 Q 为假时，$P\rightarrow Q$ 才为假。蕴含是运算符中最难理解的，希望以下两个例子有助于对它的理解。

例 2-7：用命题表示语句

如果天晴就去郊游

解：它可以用蕴含命题表示为：天晴→郊游。

显然，如果天晴且去郊游了，该复合命题为真。如果天晴却没有去郊游，则该复合命题为假。但是如果下雨，不论是否去郊游，都无从判断该命题的真假。既然无法否定原命题，所以规定当 P 为假时，无论 Q 为何值，$P\rightarrow Q$ 总是为真。

例 2-8：用命题表示语句

只有天晴才去郊游

解：用命题表示为：郊游→天晴。

本题虽然与上题只有一字之差，但意思全变了。上题表示充分条件，而本题为必要条件，即天晴时并不一定去郊游。但是，如果换个叙述法，本例句子与“如果去郊游一定是晴天”可

以说是等价的，所以可用命题表示为“郊游→天晴”。

定义 2-8（等价运算符↔）：

设 P 和 Q 为两个命题，则命题 $P\leftrightarrow Q$ 称为“P 等价于 Q”，符号↔称为等价运算符。等价运算的规定是：当且仅当 P 和 Q 的值相同时，$P\leftrightarrow Q$ 为真。

命题 $P\leftrightarrow Q$ 的取值规则如表 2-6 所示。

表 2-6 表示命题等价运算规则的真值表

P	Q	$P\leftrightarrow Q$
0	0	1
0	1	0
1	0	0
1	1	1

例 2-9：设有命题

P：a 是 1 个等边三角形

Q：a 的 3 个内角都是 60°

因为 P 是 Q 的充要条件，所以命题 $P\leftrightarrow Q$ 的意思是：如果 a 是一个等边三角形，则它的 3 个内角都是 60°；或者是，如果一个三角形的 3 个内角都是 60°，则它是一个等边三角形。

等价命题相当于“当且仅当”，但是“当且仅当”是数学术语，自然语言中没有与之相当的词汇，一般也是用“如果……那么……”来表示其等价关系，如上面例子所示。

因此遇到自然语言中的“如果……那么……”的叙述，就要根据上下文之间的关系和有关专业知识，才能判断是蕴含式，还是等价式。

蕴含与等价的最大差别在于，对于蕴含来说，话不能倒过来说。例如有命题

$\neg$郊游 $\rightarrow \neg$天晴

其意思是：如果没有去郊游那就不是天晴。

可是如果将句子反过来说：因为不是天晴，就没有去郊游，那就错了。

但是对于等价来说，反过来叙述的事实也是成立的。例如，将上述命题改为

$\neg$郊游 $\leftrightarrow \neg$天晴

则句子“如果没有去郊游，那就不是天晴”和句子“如果不是天晴，就不去郊游”都是成立的。

上述分析表明将自然语句命题化时，应考虑到所有情况，判定是充分条件还是必要条件，才能确定正确的运算表达式。

导读：

初次接触离散数学的学员一般都会对这里所讨论的蕴含和等价运算感到困惑，原因是其取值规定与人们的常识有所不同。其实，类似的问题在传统数学的学习中也是有过的，例如，对于只学习过初等数学的人来说，负数开方的运算似乎违背常识，但是你所要做的只是记住 -1 的开方等于 i，并按此规定进行有关的运算，就能掌握关于复数的知识。因此，学习命题逻辑的最好方法就是记住关于命题运算的取值规定，按规定进行运算，不要囿于过去的知识而妨碍对新知识的理解。

2.2 命题公式与真值表

2.2.1 命题公式

命题公式是由命题常量和命题变元按一定规则组成的表达式，也就是说，命题公式是用命题运算符以及括号将命题变元和命题常量组成的运算式。如给定命题 P、Q，则用命题运算符连接起来的式子$(\neg P)$、$(P\wedge Q)$、$(P\vee Q)$、$(P\nabla Q)$、$(P\rightarrow Q)$、$(P\leftrightarrow Q)$就是命题公式，这些式子还可进一步组成更复杂的命题公式，如$\neg(P\wedge Q)$、$\neg(P\vee Q)\rightarrow(P\wedge Q)$等。

在普通的数学式子中，运算遵循先乘除后加减以及括号中的式子优先运算的规则，这就是所谓的运算符优先级问题。命题运算符同样也有优先级的规定。

定义 2-9（命题运算符的优先级）：

（1）$\neg$

（2）$\wedge$，$\vee$，∇，$\rightarrow$，$\leftrightarrow$

该定义表明除了“非”运算外，其他 5 个运算符的优先级都是相同的，因此需要用括号来清楚地表明运算顺序，否则运算就无法进行，如

$$P\rightarrow(R\rightarrow Q)\rightarrow(P\rightarrow Q)\wedge(P\rightarrow R)$$

是无法进行运算的，是错误的。

而

$$(P\rightarrow(R\rightarrow Q))\rightarrow((P\rightarrow Q)\wedge(P\rightarrow R))$$

才是有意义的。

2.2.2 真值表

数学中的变量值是可以改变的，命题变元的值也是可变的，因此包含变元的命题公式取值就有多种可能性。命题公式所有变元的一组确定的数值称为公式的一组真值指派，列出公式所有真值指派及公式的相应值的表格称为真值表。2.2.1 节中各种运算符的运算规则就是用真值表来说明的。构造真值表的一般方法为：

（1）找出给定命题公式中所有命题变元 P_1，P_2，…，P_n，列出其可能的赋值共 2^n 个。

（2）按从内到外的顺序写出命题公式的各层次。

（3）对于每组赋值，计算命题公式各层次的值，直至最后计算出整个命题公式的值。

例 2-10：构造公式

$$(P\wedge\neg Q)\vee(\neg P\wedge Q)$$

的真值表。

解：公式$(P\wedge\neg Q)\vee(\neg P\wedge Q)$的真值表的构造过程如表 2-7 所示。

表 2-7 公式$(P\wedge\neg Q)\vee(\neg P\wedge Q)$的真值表

P	Q	$\neg P$	$\neg Q$	$P\wedge\neg Q$	$\neg P\wedge Q$	$(P\wedge\neg Q)\vee(\neg P\wedge Q)$
0	0	1	1	0	0	0
0	1	1	0	0	1	1
1	0	0	1	1	0	1
1	1	0	0	0	0	0

2.2.3 永真式、永假式及可满足公式

给定一命题公式，若对其所包含的命题变元指派不同的值，其值亦随之变化。根据不同情况可将命题公式分为：永真式（有的教材称重言式）、永假式（有的教材称矛盾式）和可满足公式。

定义 2-10：

（1）当且仅当在任何一组变元指派下，由 n 个命题变元 $P_1, P_2, \cdots, P_n$ 组成的命题公式 $A(P_1, P_2, \cdots, P_n)$ 都是一个为真的复合命题，则为一个永真式。

（2）当且仅当在任何一组变量指派下，由 n 个命题变元 $P_1, P_2, \cdots, P_n$ 组成的命题公式 $A(P_1, P_2, \cdots, P_n)$ 都是一个为假的复合命题，则为一个永假式。

（3）若存在至少一组变量指派，使由 n 个命题变元 $P_1, P_2, \cdots, P_n$ 组成的命题公式 $A(P_1, P_2, \cdots, P_n)$ 是一个为真的复合命题，则为一个可满足公式。

例 2-11：

（1）$(P\wedge(P\rightarrow Q))\rightarrow Q$ 是永真式，因为其真值表各项都为 1。

（2）$\neg(P\rightarrow Q)\wedge Q$ 是永假式，因为其真值表各项都为 0。

（3）$P\wedge(Q\vee R)$ 是可满足公式，因为其真值表中至少有一项为 1。

用列真值表的办法可验证一个命题公式是永真式、永假式还是可满足公式，学员可对例 2-11 自行列真值表作为练习。另一种证明方法是用命题演算的方法。

导读：

不太严格地讲，永假式好比数学里没有解的方程式，可满足公式好比至少有一个解的方程式，而永真式则好比有任意解的方程式。

2.3 命题演算

数学运算的方式一般有两类：

- 式子演算题
- 应用题

前者用数学公式和定理等对已有的数学表达式进行求解、进行化简或者证明一些表达式彼此相等；后者需要先把实际问题抽象为一个数学表达式，然后求其解。

本节讨论命题演算问题，下一节讨论如何用命题逻辑解决实际问题，即命题逻辑建模的问题。

2.3.1 命题公式的化简

命题演算的一项重要内容就是运用运算定理对复杂表达式进行化简。本节将介绍一些用于命题逻辑推理的定律以及推理的方法。

定义 2-11（等价式）：

如两个命题 P 和 Q 含有相同的命题变元，且对任意一组指派其值都相同，则称其为等价式，

记为 $P \Leftrightarrow Q$。

例 2-12：试证

$$\neg P \vee Q \Leftrightarrow P \rightarrow Q$$

证：列出等价式两边命题的真值表如表 2-8 所示。由表 2-8 可知，对于 P 和 Q 的所有可能取值，$\neg P \vee Q$ 与 $P \rightarrow Q$ 的值完全相同，所以两式是等价式。

表 2-8 （$\neg P \vee Q$）和（$P \rightarrow Q$）的真值表

P	Q	$\neg P$	$\neg P \vee Q$	$P \rightarrow Q$
0	0	1	1	1
0	1	1	1	1
1	0	0	0	0
1	1	0	1	1

要特别注意符号⇔不是命题运算符，与符号↔有本质上的差别。符号↔的两边是同一个命题的前提与结论，往往包括不同的变元；而符号⇔的两边是包括相同变元的两个命题。

命题公式化简运算的目的一般有两个：

- **把复杂的公式化简。**
- **将式中的蕴含运算符→代换成合取和析取运算符（也称为命题公式的范式化。关于范式化的严格数学定义，可参见有关命题逻辑的专门教材，这里不做深入介绍）。**

表 2-9 列出了命题化简运算常用的等价公式（其中编号是为了在命题演算中引用方便），以下是一些化简运算的例子。

表 2-9 等价命题公式（编号是为了在命题演算中引用方便）

编 号	运 算 律	数 学 式
E1	双重否定律(对合律)	$\neg\neg P \Leftrightarrow P$
E2	等幂律	$P \wedge P \Leftrightarrow P$ $P \vee P \Leftrightarrow P$
E3	交换律	$(P \wedge Q) \Leftrightarrow (Q \wedge P)$ $(P \vee Q) \Leftrightarrow (Q \vee P)$
E4	结合律	$(P \wedge Q) \wedge R \Leftrightarrow P \wedge (Q \wedge R)$ $(P \vee Q) \vee R \Leftrightarrow P \vee (Q \vee R)$
E5	分配律	$P \wedge (Q \vee R) \Leftrightarrow (P \wedge Q) \vee (P \wedge R)$ $P \vee (Q \wedge R) \Leftrightarrow (P \vee Q) \wedge (P \vee R)$
E6	吸收律	$P \wedge (P \vee Q) \Leftrightarrow P$ $P \vee (P \wedge Q) \Leftrightarrow P$
E7	德·摩根（DeMorgan）律	$\neg(P \vee Q) \Leftrightarrow \neg P \wedge \neg Q$ $\neg(P \wedge Q) \Leftrightarrow \neg P \vee \neg Q$
E8	同一律	$P \vee \mathrm{F} \Leftrightarrow P$，$P \vee 0 \Leftrightarrow P$ $P \wedge \mathrm{T} \Leftrightarrow P$，$P \wedge 1 \Leftrightarrow P$
E9	零一律	$P \vee \mathrm{T} \Leftrightarrow \mathrm{T}$ (或者 $P \vee 1 \Leftrightarrow 1$) $P \wedge \mathrm{F} \Leftrightarrow \mathrm{F}$ (或者 $P \wedge 0 \Leftrightarrow 0$)
E10	否定律	$P \vee \neg P \Leftrightarrow \mathrm{T}$ (或者 $P \vee \neg P \Leftrightarrow 1$) $P \wedge \neg P \Leftrightarrow \mathrm{F}$ ($P \wedge \neg P \Leftrightarrow 0$)
E11	蕴含等值式	$(P \rightarrow Q) \Leftrightarrow \neg P \vee Q$
E12	等价等值式	$(P \leftrightarrow Q) \Leftrightarrow (P \rightarrow Q) \wedge (Q \rightarrow P)$

例 2-13：化简下列各式：

（1）$(P\wedge(P\rightarrow Q))\vee R$

（2）$(P\wedge(Q\rightarrow R))\rightarrow S$

解：

（1）$(P\wedge(P\rightarrow Q))\vee R$

$\Leftrightarrow(P\wedge(\neg P\vee Q))\vee R$ 蕴含等值式(E11)用于$(P\rightarrow Q)$

$\Leftrightarrow((P\wedge\neg P)\vee(P\wedge Q))\vee R$ 分配律(E5)用于 $P\wedge(\neg P\vee Q)$

$\Leftrightarrow(0\vee(P\wedge Q))\vee R$ 否定律(E10)用于 $P\wedge\neg P$

$\Leftrightarrow(P\wedge Q)\vee R$ 同一律(E8)用于 $0\vee(P\wedge Q)$

（2）$(P\wedge(Q\rightarrow R))\rightarrow S$

$\Leftrightarrow\ \neg(P\wedge(\neg Q\vee R))\vee S$ 蕴含等值式(E11)用于 $Q\rightarrow R$ 和$(P\wedge(Q\rightarrow R))\rightarrow S$

$\Leftrightarrow\ (\neg P\vee(Q\wedge\neg R))\vee S$ 德·摩根律用于$\neg(P\wedge(\neg Q\vee R))$

$\Leftrightarrow\ (\neg P\vee S)\vee(Q\wedge\neg R)$ 交换律

$\Leftrightarrow(\neg P\vee S\vee Q)\wedge(\neg P\vee S\vee\neg R)$ 分配律

例 2-13（2）中最后的表达式 $(\neg P\vee S\vee Q)\wedge(\neg P\vee S\vee\neg R)$称为合取范式，其特点是括号中只有析取运算符，括号之间只有合取运算符。反之，如括号中只有合取运算符、括号之间只有析取运算符，则称为析取范式。将命题表达式转换为合取范式也是化简运算的内容之一。

除等价关系外，命题公式之间还可以有蕴含关系，当且仅当 $P\rightarrow Q$ 为永真式时，称“命题 P 蕴含命题 Q”，记作 $P\Rightarrow Q$，也称为“重言蕴含式”。重言蕴含式也有相应的演算公式，称为推理定律。表 2-10 列出的就是一些主要的推理定律。

表 2-10 一些命题推理定律

编 号	定 理 名	数 学 式
T1	附加定律	$P\Rightarrow(P\vee Q)$ $Q\Rightarrow(P\vee Q)$
T2	化简定律	$(P\wedge Q)\Rightarrow P$ $(P\wedge Q)\Rightarrow Q$
T3	假言推理定理	$((P\rightarrow Q)\wedge P)\Rightarrow Q$
T4	拒取式	$((P\rightarrow Q)\wedge\neg Q)\Rightarrow\neg P$
T5	析取三段式	$((P\vee Q)\wedge\neg P)\Rightarrow Q$
T6	假言三段式	$((P\rightarrow Q)\wedge(Q\rightarrow R))\Rightarrow(P\rightarrow R)$
T7	等价三段式	$((P\leftrightarrow Q)\wedge(Q\leftrightarrow R))\Rightarrow(P\leftrightarrow R)$

* 2.3.2 命题推理规则与方法

推理也是逻辑命题演算的重要内容。在数理逻辑中，推理是指由一些假设成立的命题按一定规则推断出新命题的过程。对于给定命题公式 H_1，H_2，…，H_n，如按一定规则能从它们推断出新命题公式 C，则称公式 C 是前提集合$\{H_1, H_2,\cdots, H_n\}$的结论，记作 $H_1\wedge H_2\wedge\cdots\wedge H_n\Rightarrow C$ 或 H_1，H_2，…，$H_n\Rightarrow C$。若推理每一步所得结论都是根据推理规则得到的，该推理就是有效的。推理的有效与否与前提的真假无关，结论是否有效也与它自身的真假无关。也就是说：推理过程关

注的只是形式，而无视它是否合理。推理的基本方法有真值表法、直接证明法和间接证明法。

1．真值表法

做出命题公式 $H_1\wedge H_2\wedge\cdots\wedge H_n\rightarrow C$ 的真值表，如果它永远为真，则 $H_1\wedge H_2\wedge\cdots\wedge H_n\Rightarrow C$ 成立。

例 2-14：证明拒取式

$$(P\rightarrow Q)\wedge\neg Q\Rightarrow\neg P$$

证：

要证明$(P\rightarrow Q)\wedge\neg Q\Rightarrow\neg P$，需要列出$((P\rightarrow Q)\wedge\neg Q))\rightarrow\neg P$ 的真值表，如表 2-11 所示。由其可知，无论给 P 和 Q 以何种真值指派，$((P\rightarrow Q)\wedge\neg Q))\rightarrow\neg P$ 总为真，所以$(P\rightarrow Q)\wedge\neg Q\Rightarrow\neg P$ 成立。

表 2-11 $(P\rightarrow Q)\wedge\neg Q\rightarrow\neg P$ 的真值表

P	Q	$P\rightarrow Q$	$\neg Q$	$(P\rightarrow Q)\wedge\neg Q$	$\neg P$	$((P\rightarrow Q)\wedge\neg Q))\rightarrow\neg P$
0	0	1	1	1	1	1
0	1	1	0	0	1	1
1	0	0	1	0	0	1
1	1	1	0	0	0	1

2．直接证明法

利用推理规则和等价命题公式、命题推理定律，将给定前提 $H_1\wedge H_2\wedge\cdots\wedge H_n$ 转换为与结论相同的表达式。命题推理常用的规则见定义 2-12。

定义 2-12：

（1）前提引入规则（P 规则）

在推导的任何一步均可引入前提。

（2）结论引用规则（T 规则）

推导过程中任何一步所得结论均可作为后继推导的前提。

（3）置换规则

在推导过程中，命题公式的子公式可被其等价公式所置换。

（4）代入规则

在推导过程中，命题变元可被其等价公式所置换。

（5）附加前提规则（CP 规则）

欲证明$(A_1\wedge\cdots\wedge A_n)\Rightarrow(A\rightarrow B)$，只需证明$(A_1\wedge\cdots\wedge A_n\wedge A)\Rightarrow B$ 即可。

例 2-15：试证

$$\neg(P\wedge\neg Q),\neg Q\vee R,\neg R\Rightarrow\neg P$$

证：

$\neg(P\wedge\neg Q)\wedge(\neg Q\vee R)\wedge\neg R$

$\Rightarrow\neg(P\wedge\neg Q)\wedge\neg Q$　　析取三段式(T5)用于$(\neg Q\vee R)\wedge\neg R$

$\Rightarrow(\neg P\vee Q)\wedge\neg Q$　　德·摩根律用于$\neg(P\wedge\neg Q)$

$\Rightarrow\neg P$　　析取三段式(T5)

证毕

注意，上面推导过程的第一步和第三步中，因为所应用的析取三段式是蕴含公式，前式与后式之间是蕴含关系，所以用符号“$\Rightarrow$”来连接。而第二步应用的 DeMorgan 律是等价公式，

所以用符号“$\Rightarrow$”连接前后两个式子。

由于有多种推理规则和等价式可供应用，同一问题往往可有多种证明形式，学员可尝试本题的其他证明形式。

3．间接证明法(归谬法、反证法)

通过证明($H_1\wedge\cdots\wedge H_n\wedge\neg C$)是假的来证明($H_1\wedge\cdots\wedge H_n$)$\Rightarrow C$，即若在给定前提中引入与结论相反的附加前提后，能证明其值为假，则可证明原式值为真。

例 2-16：试证

$$\neg(P\wedge\neg Q),\neg Q\vee R,\neg R\Rightarrow\neg P$$

证：反证法的要点是将原式中结论的否定命题引入前提做附加命题，然后通过证明该新命题为永假式来证明原式为永真式。具体步骤如下：

$\neg(P\wedge\neg Q)\wedge(\neg Q\vee R)\wedge\neg R\wedge\neg\neg P$	将结论$\neg P$ 的否定$\neg\neg P$ 引入附加前提
$\Leftrightarrow\neg(P\wedge\neg Q)\wedge(\neg Q\vee R)\wedge\neg R\wedge P$	双重否定律(E1)用于$\neg\neg P$
$\Leftrightarrow(\neg P\vee Q)\wedge(\neg Q\vee R)\wedge\neg R\wedge P$	德·摩根用于$\neg(P\wedge\neg Q)$
$\Rightarrow(\neg P\vee Q)\wedge\neg Q\wedge P$	析取三段式(T5)用于$(\neg Q\vee R)\wedge\neg R$
$\Rightarrow\neg P\wedge P$	析取三段式(T5)用于$(\neg P\vee Q)\wedge\neg Q$
$\Rightarrow 0$	否定律(E10)用于$\neg P\wedge P$

因为结果为永假，所以原结论成立。

2.4 命题模型

命题模型（也称命题逻辑模型）是由命题构成的对自然语句描述的事实的抽象，它可将若干不同的处理与各种条件关联起来，表述概念、进行推理和判断等，因此它是应用命题逻辑解决实际问题的重要工具。

对于命题模型的应用，应掌握两方面的内容：一是将自然语言表示为命题公式，即命题建模；二是用自然语言描述命题公式的含义。这对设计程序和理解程序均有重要意义。

2.4.1 命题建模

1．命题建模的步骤

- 根据给定条件，将自然语句符号化，建立若干原子命题。
- 将原子命题按题意组合成复合命题。
- 简化复合命题。

2．自然语言符号化的一般规则

- “……是……”、“……做……”的叙述句一般可用原子命题表示。
- 用逗号“，”、“和”、“且”、“而”、“但是”等并列联结词连接的两个句子表示的是一个合取命题。
- 对于用“或者”连接的两个句子，需要分析，是简单的析取命题，还是异或命题。
- 对于一个用“如果……就……”表示的命题，应尝试将句子反向陈述，若正向、反向命题均成立，则其为一个等价命题，否则就是一个蕴含命题。

以下是命题模型应用的一些例子。

例 2-17：两个双动开关分别装在楼上和楼下，用于控制同一个灯，即，无论扳动哪个开关都能使关着的灯打开，而无论扳动哪个开关又都能使开着的灯关掉。用命题表示其控制逻辑。

解：由题意可设 3 个命题

P：灯亮

Q：开关 1 在上位($\neg Q$：开关 1 在下位)

R：开关 2 在上位($\neg R$：开关 2 在下位)

则以下命题公式表示其输出项

$$(Q\wedge\neg R)\vee(\neg Q\wedge R)\leftrightarrow P$$

开关 1 在上位和开关 2 在下位的组合时灯是亮的（两者为与的关系），当开关 1 在下位和开关 2 在上位时灯也是亮的（两者为与的关系），而两个组合之间是或的关系，所以用析取运算符来连接它们。两个命题都是灯亮的充要条件，所以它们和命题 P 用一个等价运算符来连接。也就是说，如果把话反过来说也是成立的。即如果灯是亮的，则不是开关 1 在下位和开关 2 在上位，就是开关 1 在上位和开关 2 在下位。

例 2-18：用命题公式表示以下描述的更新记录的条件：

若已售出货和收款但没收佣金，或已收款但既没有售出货也未收佣金，则更新记录。

解：根据题意可建立以下原子命题

P：售货

Q：收款

R：收佣金

则语句“已售出货和收款但没收佣金”可表示为

$$(P\wedge Q)\wedge\neg R$$

语句“已收款但既没有售出货也未收佣金”可表示为

$$Q\wedge\neg(P\vee R)$$

因为，这两个语句之间是“或”的关系，最后得命题

$$((P\wedge Q)\wedge\neg R)\vee(P\wedge\neg(Q\vee R))$$

经化简得到

$$P\wedge\neg R$$

如果用语句来表示就是：如果售出货但没有收佣金就更新记录。

本例说明了逻辑建模对程序设计的意义。设计程序需要将用户要求抽象为命题公式，但直接得到的式子往往比较复杂和条理不清，通过化简可以理顺众多需求之间的关系，使控制逻辑最优化。关于应用命题逻辑设计程序控制语句的具体方法，将在 6.4.7 节中予以详细讨论。

例 2-19：用命题公式设计一个向客户进行广告促销程序的条件：

(1) 若年龄大于 25 岁，或既有家庭财产保险又有人寿保险，则投送广告。

(2) 若月收入超过 2 000 元且有家庭财产保险，或年龄超过 25 岁且有人寿保险，则投送广告。

(3) 若小于 25 岁且无家庭财产保险，则投送广告。

解：首先根据题意建立以下原子命题

A：超过 25 岁

B：月收入超过 2 000 元

C：有家庭财产保险

D：有人寿保险

则上述语句可分别用以下命题来表示

(1) $(A\vee(C\wedge D))$

(2) $((B\wedge C)\vee(A\wedge D))$

(3) $\neg(A\vee C)$

由于 3 个语句并列或的关系，所以可合并为一个复合命题

$$(A\vee(C\wedge D))\vee((B\wedge C)\vee(A\wedge D))\vee(\neg(A\vee C))$$

然后对其进行简化，结果为

$$(A\wedge D)\vee(B\wedge C)\vee(\neg A\wedge\neg C)$$

学员可自己练习将其转换为自然语言描述。

* 2.4.2 命题模型推理

导读：本节为学科完整性和扩大知识面而设。

逻辑模型也是判定事实的重要工具。用命题模型表示概念时，其前提和结论都是确定的；而在用命题模型进行推理时，一般只有前提，结论是未知的，需要通过命题演算得出，或需要通过命题演算加以确定。本节是一个关于命题模型推理的例子，习题中给出了另外一个有趣的例子。

例 2-20：南洋某岛上有两个部落，其中一个总是说实话，而另一个则永远说谎话。当探险家问某岛民该岛上是否有金子时。答曰：“当且仅当我说的是实话时，此地有金子。”试问：

(1) 该人属于哪个部落

(2) 该处是否确有金子

解：根据题意可设下列命题

P：该人属于说实话部落

Q：岛上有金子

则该岛民的话可用命题 $P\leftrightarrow Q$ 来表示。

根据定义 2-8，命题 $P\leftrightarrow Q$ 的取值规则可用表 2-12 表示。

表 2-12 命题 $P\leftrightarrow Q$ 的真值表

P	Q	$P\leftrightarrow Q$
0	0	1
0	1	0
1	0	0
1	1	1

由表 2-12 得知语句“当且仅当我说的是实话时，此地有金子”有两种情况：

(1) 假定该人属说实话的部落，即 P=1，则其话必为真，即 $P\leftrightarrow Q$ 的值也为 1。由真值表可知，此时 Q 必为 1，表示岛上确有金子。

(2) 假定该人属说假话的部落，即 P=0，则其话必为假，即 $P\leftrightarrow Q$ 的值也为 0。由真值表可知，此时 Q 也为 1，表示岛上确有金子。

因此，结论是：(1) 该处确有金子。(2) 无法确定其属于何部落。

习 题

知识点：

命题逻辑建模是程序设计的理论基础，也是本教材引入离散数学的目的之一。因此要求学员熟练掌握：

① 将自然语言表达为命题公式。

② 将命题公式用自然语言来描述。

③ 命题公式简化演算。

1. 已知命题 P=1，Q=0,计算下列各命题的值。

(1) $P\wedge Q$ (2) $P\vee Q$ (3) $P\wedge\neg Q$ (4) $\neg P\wedge Q$

(5) $\neg(P\vee Q)$ (6) $\neg(P\wedge Q)$ (7) $\neg(\neg P\wedge Q)$ (8) $\neg(P\vee\neg Q)$

2. 写出下列公式的真值表。

(1) $\neg(\neg P\wedge Q)$ (2) $\neg(P\vee\neg Q)$

(3) $\neg(P\vee Q)\wedge\neg(P\wedge Q)$ (4) $\neg(P\vee\neg Q)\wedge R$

(5) $P\rightarrow(Q\vee R)$ (6) $(P\vee Q)\rightarrow(P\rightarrow Q)$

3. 判断下列各对公式是否等价。

(1) $\neg(P\wedge\neg Q)$ $\neg P\vee Q$

(2) $\neg(P\vee\neg Q)\wedge\neg(P\vee Q)$ $P\vee(P\wedge Q)$

(3) $\neg(P\vee\neg Q)\wedge\neg(P\vee Q)$ $(\neg P\wedge Q)\wedge(\neg P\wedge\neg Q)$

(4) $P\vee(\neg Q\wedge R)$ $(P\vee\neg Q)\wedge(P\vee R)$

4. 化简以下各式：

(1) $((A\rightarrow B)\leftrightarrow(\neg B\rightarrow\neg A))\wedge C$

(2) $(A\wedge B\wedge C)\vee(\neg A\wedge B\wedge C)$

(3) $(\neg A\wedge B\wedge C)\vee(A\wedge\neg B\wedge C)\vee(A\wedge B\wedge\neg C)\vee(A\wedge B\wedge C)$

5. 证明以下等价关系(不构造真值表)：

(1) $A\rightarrow(B\rightarrow A)\Leftrightarrow\neg A\rightarrow(A\rightarrow\neg B)$

(2) $\neg(A\leftrightarrow B)\Leftrightarrow(A\vee B)\wedge\neg(A\wedge B)$

(3) $\neg(A\leftrightarrow B)\Leftrightarrow(A\wedge\neg B)\vee(\neg A\wedge B)$

(4) $(A\rightarrow P)\wedge(B\rightarrow P)\Leftrightarrow(A\vee B)\rightarrow P$

6. 给定命题及其值如下

A：Mary 去看电影。(1)

B：Mary 的妹妹去看电影。(0)

C：天下雨。(0)

用复合命题表示下列句子，并计算其值。

(1) 天下雨，Mary 去看电影。
(2) Mary 去看电影或者天下雨。
(3) Mary 和妹妹去看电影。
(4) Mary 或她妹妹去看电影。
(5) 天下雨，但 Mary 和妹妹仍去看电影。
(6) 如果妹妹不去，Mary 也不去看电影。
(7) 虽然天不下雨，但 Mary 和妹妹未去看电影。
(8) 尽管下雨，但 Mary 仍去看电影，其妹妹却没有去。

7. 某商品保证书中有以下条款，用命题公式来表示它们。
(1) 商品使用不当或者自行拆卸不在保修范围内。
(2) 只有原包装没有打开的商品才能退货。
(3) 退货期为 30 天。
(4) 综合上述条款的保证条件。

8. 某旅游合同中有以下句子
T：早餐包括在房费中
S：卧铺另加价
W：游客可选景点 A 而不另收费
用自然语言描述下列命题公式的内容。
(1) $T\vee\neg S$　(2) $\neg S\wedge\neg W$　(3) $\neg(S\wedge T)$
(4) $\neg(W\vee T)$　(5) $T\rightarrow(S\vee\neg W)$　(6) $(S\wedge T)\rightarrow\neg W$

9.（选作）历史上有一个有趣的逻辑推理题。古罗马有个富商，他有一个才貌双全的女儿叫鲍西亚，每天向她求婚的人不计其数。为了选出最聪明的人作女婿，富商想出一个方法。他请工匠做了金、银、铅 3 个匣子，并对求婚者说：“我将女儿的肖像放在其中一个匣子里，在 3 个匣子上各刻了一句话，但其中只有一句是真的。谁能根据匣子上刻的话猜出肖像在哪个匣子里，就可以做我的女婿。”3 个匣子上刻的话分别是：

金匣子：肖像在铅匣子中。
银匣子：肖像在金匣子中。
铅匣子：肖像不在此匣中。

许多求婚者都失败了，但据说最后有个叫巴萨尼奥的年轻人成功了。你能用命题演算求出巴萨尼奥的答案是什么吗？他的答案是万无一失的吗？

第3章　谓 词 逻 辑

知识点：

① 命题逻辑是谓词逻辑系统的一个子系统，谓词逻辑包括变元的概念和数量的推理机制，表述和推理能力均比命题逻辑强大得多。虽然需要专门的人工智能编程语言（如 Prolog）才能全面实现谓词推理机制，但它对普通程序控制语句的设计也有指导意义（参见 6.4.7）。

② 本章是为学员完整了解数理逻辑学科内容而设的，但是内容难度稍大，教学时可酌情选读。

命题逻辑可根据原子命题的真假确定复合命题的真假，也可判定在命题范围内的推理所得结论是否有效。但是，命题逻辑无法有效地表达数量的概念，如以下事实或推论，就无法用命题逻辑表示：

(1) **所有的自然数都是实数。**

(2) **有些大学生是运动员，所以有些运动员是大学生。**

以上两个推理都是有效的，但却无法用命题逻辑加以证明，而谓词逻辑引入变元的概念和数量的推理机制，可有效地处理这类问题。由于命题逻辑是谓词逻辑系统的一个子系统，命题公式是谓词公式的特例，因此谓词逻辑的许多定义和运算法则均与命题逻辑相似。通过对比有关概念间的相似性学习本章，将会起到事半功倍的作用。

3.1　谓词命题和谓词公式演算

3.1.1　谓词和个体词

定义 3-1（谓词命题）：

谓词逻辑用 $Y(x)$**表示** x **是** Y **的命题，其中** x **称为个体词，表示判断对象；** Y **称为谓词，表示判断对象的性质。**

个体词通常用小写字母表示。若个体词任指某类对象中的一个，则称为个体变元；反之，若其内容为一个确定的对象，则称为个体常元；谓词的首字母通常用大写字母表示。如语句“ x is a Dog（x **是一只狗）”可表示为**

$$\mathrm{Dog}(x)$$

而表达式

$$\mathrm{Red}(x)$$

用语句描述，就是“ x is Red（x **是红色的）”。**

说明：

① 从现在起，将一律用英语描述事实，以便于转换为命题，因为英语的判断句一定是“x is y”的形式，而汉语就不一定了。如语句“x is larger than y”，用汉语表示就是“x 大于 y”，其中没有“是”动词，在转换为命题时不容易把握词语之间的关系。

② 在谓词意义比较明确时，为简单起见，还经常只用谓词的首字母表示命题，如上述命题 Red(x)就可简略表示为 $R(x)$。

上述命题只涉及两个变元，但一般命题涉及对象不止两个，所以谓词公式可推广到多个变元的一般情况。

定义 3-2（谓词的元）：

由 n 个个体变元 $a_1, a_2, \ldots, a_n$ 组成的命题中的谓词称为 n 元谓词

$$P(a_1,a_2,\ldots,a_n)$$

其中变元的顺序一般是不可交换的。

如语句“x is Larger than y”可表示为命题

$$L(x, y)$$

其中 L 是二元谓词。

语句“x is Between y and z”可表示为命题

$$B(x, y, z)$$

其中谓词 B 涉及 3 个个体变元，是三元谓词。

3.1.2 量词

谓词逻辑一个最重要的特点是能用量词对命题涉及的个体数量进行限定。量词是谓词逻辑中的一种短语，其作用为限定其后某表达式中某个变元的范围，以表明指认的变元是这个范围内的全体或某一个。量词有两种：全称量词和存在量词。

定义 3-3：

(1) 全称量词：用符号$\forall x$ 表示，意思是变元 x 的所有个体，或者任何一个。

(2) 存在量词: 用符号$\exists x$ 表示，意思是至少存在变元 x 的一个个体。

定义 3-4：

(1) 紧接于量词之后的原子公式或出现在量词后面括号内的公式叫做量词的辖域。

(2) 在量词辖域内的变元称为约束变元；当公式前无量词时，其中的变元称为自由变元。

(3) 量词是由内层到外层作用的，变元只受离其最近的量词的约束，称为最近作用原则。

例 3-1：

(1) 命题“所有的自然数都是实数”可表示为

$$(\forall x)N(x)\rightarrow R(x)$$（其中 N 表示自然数，R 表示实数）

用语言来描述，就是“对于所有 x，若其为自然数，则是实数”。注意这里量词$\forall$限定的只是 $N(x)$。

(2) 命题“有一些自然数是素数”可表示为

$$(\exists x)(N(x)\wedge P(x))$$（其中 P 表示素数 prime number）

用语言来描述，就是“至少有一个 x，它既是自然数又是素数”。注意这里量词$\exists$限定的是两个

表达式 $N(x)$和 $P(x)$，若只限定一个，意思就不对了。

为简单起见，式中$(\forall x)$和$(\exists x)$外的括号一般可以省略。

导读：

命题逻辑是对全部对象而言的，相当于谓词逻辑中用全称量词限定所有变元的情况，所以被认为是谓词逻辑的子系统。

上述例子说明，在同一个公式中，一个变元既可约束出现，又可自由出现。为避免概念上的混乱，一般要用改名规则对其加以改写。如例 3-1 中(1)可改写为：$\forall yN(y)\rightarrow R(x)$，其限定关系就很清楚了，意思是对于所有是自然数的 y 来说，它们都是实数。公式 $R(x)$中的 x 可取任何值，所以是自由变元。

定义 3-5（改名规则）：

(1) 将变元在量词及其辖域中的所有出现均一起改名，其余部分不变。

(2) 新名称应是公式中未出现过的符号。

例 3-2：(1) 给定命题

$$\forall x(P(x)\rightarrow R(x))\vee\forall x(P(x)\rightarrow Q(x))$$

其中第一个量词$\forall x$ 的辖域为 $P(x)\rightarrow R(x)$，第二个量词$\forall x$ 的辖域为 $P(x)\rightarrow Q(x)$，其中的 x 都是约束变元。因此，需要对公式进行改名，结果为

$$\forall x(P(x)\rightarrow R(x))\vee\forall y(P(y)\rightarrow Q(y))$$

(2) 给定命题

$$\forall x((P(x)\wedge\exists xQ(x,z))\rightarrow\exists yR(x,y))\vee Q(x,y)$$

其中第 1 个量词$\forall x$ 的辖域为$(P(x)\wedge\exists xQ(x,z))\rightarrow\exists yR(x,y)$，第 2 个量词$\exists x$ 的辖域为 $Q(x,z)$，第 3 个量词$\exists y$ 的辖域为 $R(x,y)$，其中的 x, y 都是约束变元。但是，在 $Q(x,z)$中的 x 只受离其最近的量词$\exists x$ 的约束，而不受$\forall x$ 的约束。最后一项 $Q(x,y)$中的 x, y 又都是自由变元。因此，需要对公式进行改名，结果为

$$\forall u((P(u)\wedge\exists vQ(v,z))\rightarrow\exists sR(u,s))\vee Q(x,y)$$

3.1.3 谓词公式演算

定义 3-6（谓词命题函数）

当命题中所有对象都是变元时，有

$$P(x_1, x_2, \cdots, x_n)$$

P 称为命题函数。命题函数不是命题，但如果将其中的变元代以确定的个体时，就表示一个具体的命题。

不包括命题运算符的命题函数 $P(x_1, x_2, \cdots, x_n)$称为谓词逻辑的原子公式。与逻辑命题公式合成法则相似，可用第 2.1.2 节所讨论的 6 种逻辑运算符将原子公式连接成复杂的谓词公式。例如，若给定原子谓词公式

$$P(x)、Q(x)$$

则用逻辑运算符组成的以下式子也都是谓词公式

$$\exists xP(x)\rightarrow Q(x)$$

$$\exists x P(x) \wedge \exists y Q(y)$$

$$\forall x(\neg P(x)) \vee \forall y Q(y)$$

谓词公式可包含有命题变元和个体变元。所谓的对谓词公式赋值，即用确定的个体取代个体变元，确定的命题取代命题变元。

一个谓词公式经赋值后，便成为有确定真值的命题。与命题公式类似，谓词公式也分为永真式、永假式和可满足公式。但是，由于谓词公式的复杂性和解释的多样性，目前还没有一个可行的算法来判断一个谓词公式是可满足的还是不满足的。

与命题公式类似，两个谓词公式之间也存在着等价或者蕴含的关系。

定义 3-7：

两个谓词公式 A 与 B 是等价的，当且仅当对 A 和 B 的任一组赋值，两者的值都相同，记作 $A \Leftrightarrow B$。

例 3-3：

(1) 若用谓词公式取代表 2-9 中命题等价公式中的命题变元，即可得到谓词公式的等价公式。例如，用 $\exists x P(x)$ 取代命题逻辑的否定律 $P \vee \neg P \Leftrightarrow 1$（表 2-9 公式 E10）中的 P，得到的式子：

$$\exists x P(x) \vee \neg \exists x P(x) \Leftrightarrow 1$$

就是谓词公式否定律的表达形式之一。

(2) 若在表 2-9 中命题等价公式两边同时加上同一量词，也可得到谓词公式的等价公式。例如，在命题逻辑的蕴含等值式 $(P \rightarrow Q) \Leftrightarrow \neg P \vee Q$（表 2-9 公式 E11）两边同时加上量词 $\forall x$，得到的式子：

$$\forall x(P(x) \rightarrow Q(x)) \Leftrightarrow \forall x\,(\neg P(x) \vee Q(x))$$

就是谓词公式蕴含等值式的表达形式之一。

(3) 同理，若用谓词公式取代表 2-10 命题推理定律中的变元，或者在其两边同时加上同一量词，即可得到对应的谓词推理定律。例如，在假言推理定律 $((P \rightarrow Q) \wedge P) \Rightarrow Q$（表 2-10 公式 T3）中两边同时加上量词 $\exists x$，得到的式子

$$\exists x((P(x) \rightarrow Q(x)) \wedge P(x)) \Rightarrow \exists x Q(x)$$

就是谓词推理的假言推理定律的表达形式之一（下面的例 3-4 中将应用到假言推理定律的另一个形式）。

例 3-3 说明命题演算中的等价式和蕴含式（表 2-9 和表 2-10）都可以推广到谓词逻辑中使用。但是随着两边添加量词的不同，或者代换的谓词公式不同，每个命题等值公式或者命题推理定律在谓词演算里都可以有多种表现形式，无法简单地用一个表来概括它们，因此这里也就不一一列举了。

此外，谓词逻辑中还有一些专门对数量关系进行推理的规则，表 3-1 列出了其中主要的一些。

表 3-1 一些有关量词的推理规则

定 律 名	数 学 式
量词的转化律	$\neg \exists x A(x) \Leftrightarrow \forall x \neg A(x)$ $\neg \forall x A(x) \Leftrightarrow \exists x \neg A(x)$
量词 $\forall$ 对 $\wedge$ 的分配律	$\forall x(A(x) \wedge B(x)) \Leftrightarrow \forall x A(x) \wedge \forall x B(x)$
量词 $\exists$ 对 $\vee$ 的分配律	$\exists x(A(x) \vee B(x)) \Leftrightarrow \exists x A(x) \vee \exists x B(x)$

可以简单地将上述等价式理解为量词对逻辑非、合取和析取运算的分配律，其作用是将量词深入到各谓词变元前。

命题演算中的推理规则（定义 2-12），如 P 规则、T 规则、CP 规则，均可在谓词演算中应用，此外，还有以下一些专门关于量词的演算规则：

定义 3-8：

(1) **全称特指规则**(US 规则)

$$\forall xA(x)\Rightarrow A(y)$$

若个体域中所有个体都具有性质 A，则个体域中任一个体 y 必具有性质 A。

(2) **存在特指规则**(ES 规则)

$$\exists xA(x)\Rightarrow A(c)$$

若个体域中存在具有性质 A 的个体，则个体域中必有某个个体 c 具有性质 A。

(3) **全称一般化规则**(UG 规则)

$$A(x)\Rightarrow \forall yA(y)$$

若个体域中任一个体都具有性质 A，则个体域中全体个体都具有性质 A。

(4) **存在一般化规则**(EG 规则)

$$A(c)\Rightarrow \exists xA(x)$$

若个体域中某个个体 c 具有性质 A，则个体域中存在具有性质 A 的个体。

上述规则里的 c 表示它不是变元，而是一个确定的个体。 以下是谓词公式推理的例子。

例 3-4：试证

$$P(x),\forall xQ(x)\Rightarrow \exists x(P(x)\wedge Q(x))$$

证：

$P(x)\wedge\forall xQ(x)$ （原式左边）	
$\Rightarrow P(x)\wedge Q(x)$	全称特指规则（US）用于$\forall xQ(x)$
$\Rightarrow \exists x(P(x)\wedge Q(x))$	存在一般化规则(EG)

证毕

例 3-5：试证

$$\forall x(P(x)\rightarrow Q(x))，\exists xP(x)\Rightarrow \exists xQ(x)$$

证：

$\forall x(P(x)\rightarrow Q(x))\wedge \exists xP(x)$	
$\Rightarrow \forall x(P(x)\rightarrow Q(x))\wedge P(c)$	存在特指规则（ES）用于$\exists xP(x)$
$\Rightarrow (P(c)\rightarrow Q(c))\wedge P(c)$	全称特指规则（US）用于$\forall x(P(x)\rightarrow Q(x))$
$\Rightarrow Q(c)$	假言推理定律(T3)（注 1）
$\Rightarrow \exists xQ(x)$	存在一般化规则(ES)

证毕

注 1 该结果是用谓词公式 $P(c)$和 $Q(c)$分别取代假言推理定律$(P\rightarrow Q)\wedge P\Rightarrow Q$（表 2-10 公式 T3）中的变元 P 和 Q 得到的。

3.2 谓词模型

与命题模型类似，谓词模型（也称谓词逻辑模型）是由谓词公式构成的、对自然语句描述的事实的抽象。它的作用是使若干不同的处理与各种条件关联起来，表述概念，进行推理和判断等。因此它是阐明问题和设计程序的重要工具。具体来说，谓词模型可以用于：

- 表述概念。如表示“偶数是可以被 2 整除的自然数”的数学概念。
- 进行推理和判断。如证明定律等。

谓词逻辑具有与自然语言几乎同样的表达和判定能力，谓词模型的表述和推理能力也比命题模型强大得多。因此，它不仅在计算机学科中，而且在许多自然学科，如数学、物理学中，都被广泛用于叙述概念以及进行定理和定律的证明等。

3.2.1 谓词建模

谓词建模（用谓词模型表述概念）的步骤和命题建模的步骤很相似，都有以下步骤：

- 根据给定条件，将自然语句符号化，建立若干原子公式。
- 将原子公式按题意组合成复合公式。
- 简化复合公式。

不同的是，在谓词公式中应考虑量词。以下就是谓词建模时处理量词的一些规则。

- 若句子中有“每个”、“所有”等字样，一般应该用全称量词$\forall x$加以限定。
- 若句子中有“某个”、“一个”等字样，一般可用存在量词$\exists x$加以限定。
- 若句子中有“并非所有”的字样，则既可用存在量词$\exists x$加以限定，也可用全称量词$\forall x$的否定加以限定。

注意，如果是“仅有一个”或是“有一个且仅有一个”则不能用“任一个”量词。以下是谓词建模的一些例子。

例 3-6：用谓词公式表达

(1) A 物体是红色的。

解：上述命题的英语句子是：A is Red。**所以，用谓词公式表示就是：**

$$R(A)$$

(2) B 物体不是红色的。

解：该命题是前一个命题的否定式，只要在原命题前加上逻辑非运算符号即可。即$\neg R(B)$。

(3) A 物体是红色的，而 B 物体不是红色的。

解：两个句子表示一个并列事实，所以用一个合取命题表示为 $R(A)\wedge\neg R(B)$。

例 3-7：用谓词公式表达概念

如 m 是奇数，则 $m+1$ 不是奇数。

解：这里首先要定义一个命题：m **是奇数**

$$O(m) \qquad (\text{m is a Odd number})$$

则不是奇数的命题可表示为其否定式

$$\neg O(m)$$

但是上述句子不能表述为 $O(m)\rightarrow\neg O(m+1)$，因为 m 和 $m+1$ 对于命题 $O(m)$来说没有区别，均指任何一个个体。因此，还需要定义一个数学函数，以表示 m 和 $m+1$ 之间的关系

$$f(x)=x+1$$

这表明无论 x 是什么数，$f(x)$都比 x 大 1。若 x 为奇数，则 $f(x)$为偶数。因为“若……则……”相当于一个蕴含关系，所以上述句子最后可表示为

$$O(m)\rightarrow\neg O(f(m))$$

其意思为，若 m 为奇数，则比 m 大 1 的函数 $f(m)$就不是奇数。

导读：

① 什么样的概念要用函数而不是谓词命题描述？一般来说，凡是涉及数学运算的问题一般用函数表示，因为它无法用“x is y”的句型来描述。

② 在谓词建模中经常会用到这种用函数表示概念的方法。

例 3-8：用谓词公式表达物理定律

质点在平衡力系作用下，保持其静止或匀速直线运动状态不变。

解：先设以下原子公式

$M(P)$：P 是质点（P is a Mass）。

$B(f)$：f 是平衡力系（f is a Balanced system）。

$L(P)$：P 做匀速直线运动（P is Linearly moving）。

$S(P)$：P 保持静止（P is Static）。

$A(f, P)$：f 作用在 P 上（f is Acting upon P）。

则该定律可表示为复合公式

$$\forall P\forall f((M(P)\wedge B(f)\wedge A(f,P))\rightarrow(S(P)\vee L(P)))$$

如果用语句描述，上式相当于：“对于一切质点，在任何一个平衡力系的作用下，处于静止状态或者保持匀速直线运动。”当人们用自然语言表述这一概念时，一般已习惯于省略其中的“一切、任何一个”等字样，而用谓词逻辑的数学方法表述时，则要求表达更严密、更精确。

*3.2.2 谓词推理

与命题模型类似，用谓词模型表示概念时，其前提和结论均为确定的；而用谓词模型进行推理时，一般只有前提，而结论是未知的，需要通过谓词演算得出，或需要通过谓词演算加以确定。

复杂的事实，很难一眼辨别其真伪，而谓词演算由于其推理过程严密而精确，很容易在一些似是而非、类似绕口令的叙述中，抽取出事实真相，起到去伪存真的作用。因此谓词模型广泛用于人工智能程序和理论证明等。以下是谓词推理的一个例子。

例 3-9：给定以下条件

(1) 任何人若不喜乘机动车，就喜骑自行车

(2) 如某人喜步行，则他必不喜欢乘机动车

试证：因为有人不喜欢骑自行车，所以必有人不喜欢步行。

证：由题意可设以下原子公式

$W(x)$：x 喜欢步行（x is like Walking）

$B(x)$：x 喜欢骑自行车（x is likes Biking）

$R(x)$：x 喜乘机动车（x is like Riding）

于是上述句子可以分别表示如下：

任何人若不喜乘机动车，就喜骑自行车

$$\forall x(B(x)\vee R(x))$$

如某人喜步行，则他必不喜欢乘机动车

$$\forall x(W(x)\rightarrow\neg R(x))$$

有人不喜欢骑自行车

$$\exists x\neg B(x)$$

把上述命题合取起来，就是论证前提

$$\forall x(W(x)\rightarrow\neg R(x)),\ \forall x(B(x)\vee R(x)),\ \exists x\neg B(x)$$

而欲证的结论为：$\exists x\neg W(x)$

证明过程如下：

$\forall x(W(x)\rightarrow\neg R(x))\wedge\forall x(B(x)\vee R(x))\wedge\exists x\neg B(x)$

$\Rightarrow\forall x(W(x)\rightarrow\neg R(x))\wedge\forall x(B(x)\vee R(x))\ \wedge\neg B(c)$ 存在特指规则(ES)用于$\exists x\neg B(x)$

$\Rightarrow\forall x(W(x)\rightarrow\neg R(x))\wedge(B(c)\vee R(c))\wedge\neg B(c)$ 全称特指规则(US)用于$\forall x(B(x)\vee R(x))$

$\Rightarrow(W(c)\rightarrow\neg R(c))\wedge(B(c)\vee R(c))\ \wedge\neg B(c)$ 全称特指规则(US)用于$\forall x(W(x)\rightarrow\neg R(x))$

$\Rightarrow(W(c)\rightarrow\neg R(c))\ \wedge R(c)$ 析取三段式(T5)用于$(B(c)\vee R(c))\ \wedge\neg B(c)$

$\Rightarrow\neg W(c)$ 拒取式(T4)

$\Rightarrow\exists x\neg W(x)$ 存在一般化规则(EG)

证毕

习　题

知识点：

本章为选读内容。有兴趣的学员可将以下两点作为练习目的：

① 将自然语言表达为谓词公式。

② 用自然语言描述谓词公式。

1. 已知公式 $P(x)=1$，$Q(x)=0$，计算下列各命题的值：

(1) $P(x)\wedge Q(x)$　(2) $P(x)\vee Q(x)$　(3) $P(x)\wedge\neg Q(x)$

(4) $\neg P(x)\wedge Q(x)$　(5) $\neg(P(x)\vee Q(x))$　(6) $\neg(P(x)\wedge Q(x))$

(7) $\neg(\neg P(x)\wedge Q(x))$　(8) $\neg(P(x)\vee\neg Q(x))$

2. 写出下列公式的真值表：

(1) $\neg(\neg P(x)\wedge Q(x))$　(2) $\neg(P(x)\vee\neg Q(x))$

(3) $\neg(P(x)\vee Q(x))\wedge\neg(P(x)\wedge Q(x))$　(4) $\neg(P(x)\vee\neg Q(x))\wedge R(x)$

(5) $P(x)\rightarrow(Q(x)\vee R(x))$　(6) $(P(x)\vee Q(x))\rightarrow(P(x)\rightarrow Q(x))$

3. 将以下语句表达成谓词合成公式：

(1) 某篮球国手比所有的人都高。
(2) 任何一个人一定比某个人高。
(3) 可以用分数表示的数是有理数。
(4) 2 不能整除任何一个非偶数。
(5) 乘积为零的 2 个实数中必有一个为零。
(6) 通过两点有且仅有一条直线。
(7) 存在实数 x、y、z，使得 $xy > x+z$。
(8) 当且仅当 $2<|x|<5$ 时 $4< x^2 <25$。

第4章 集 合 论

知识点：

本章学习要求包括：

① 熟悉集合的基本概念，能熟练进行集合运算、简化集合表达式。

② 能进行集合建模，将用自然语言描述的问题转换为集合表达式，或用自然语言描述集合表达式的意义。

本教材之所以强调学习软件技术应从了解离散数学的常识开始，是因为计算机所处理的数据大部分是离散的非数值数据。分类是处理离散的非数值数据最基本的方法之一，集合论（set theory），或称集合逻辑就是关于事物分类的数学方法。熟悉集合论这一工具，对学习程序设计语言、数据结构、数据库系统原理有重要意义。另外，在其他科学领域也经常用到集合论的知识。

4.1 集合的概念

4.1.1 集合

从某种意义上来说，自然科学就是对事物进行分类的科学。集合是对事物进行分类的一种数学工具，是具有相近性质的离散型数据的简单罗列。例如，学员名单是集合，商品目录也是集合。

定义 4-1：集合

具有共同属性的互不相同对象的全体称为集合（set），组成集合的对象个体即为其元素(element)。

定义 4-2：

(1) 有限集：集合由有限元素组成。有限集中可分辨元素的个数称为集合的基数。

(2) 无限集：集合元素数量可为任意多个。

(3) 全集：集合包含的研究对象的所有元素，记为 U。

(4) 空集：集合不包含任何元素，记为$\varnothing$ 。

(5) 有序集：集合中的元素总是按给定顺序排列。

(6) 无序集：集合中的元素排列顺序是无关紧要的。

集合一般用大写字母或带下标的大写字母表示；元素用小写字母或带下标的小写字母表示。集合的表达式是

$a \in A$，读作“a 属于集合 A”或者“a 在集合 A 中”。

如果某元素 b 不在集合 A 中，则记为

$b \notin A$，读作“b 不属于集合 A”或者“b 不在集合 A 中”。

4.1.2 集合的描述方法

常用的描述集合的方法有列举法和描述法。但是，表示集合的方法并不限于这两种，例如，6.1.2 还介绍了用语法（grammar）表示集合的方法。

例 4-1：列举法

集合列举法在{ }中一一列出其元素，元素之间用逗号分开，如

$C = \{2,8,5\}$

$B = \{$张同学,李同学,王同学,X 同学$\}$

$\mathbf{N} = \{1,2,3,\cdots\}$ (N 表示自然数的集合)

$A = \{a_0, a_1, a_2, \cdots, a_{n-1}\}$ 。

说明：

因为 C++语言对于元素的编号方法是从 0 开始至 $n-1$，为了能将有关数学概念与程序语句直接对应，在对集合元素编号时，教材均沿用从 0 至 $n-1$ 的约定。

列举法因须将元素一一列举，因此一般只适用于有限集。对于元素数量较多的集合或无限集，则可用描述法。描述法可采用自然语言，但更常用严密的数学表达式，包括数学运算式、逻辑运算式以及集合定义等。

例 4-2：以下是用自然语言说明的集合

(1) N 是自然数集。

(2) Z 为非负整数集。

(3) B 为信息专业全体学生的集合。

以下是用数学表达式说明的集合

(4) $P = \{ x \mid (x \in N) \wedge (x > 1) \}$

其意思是集合 P 中的所有元素 x 必须同时满足 2 个条件：属于自然数集 N 和大于 1。

(5) $Q = \{ x \mid x = y + z, y \in \{1,3,5\}, z \in \{2,4,6\}\}$

其意思是集合 Q 的所有元素 x 都是 y 和 z 的和，而 y 和 z 本身则是用列举法定义的集合。

(6) 小于 10 的素数的集合。因为 10 以内的素数个数有限，可用列举法表示为 $P=\{2, 3, 5, 7\}$。

也可以用描述法表示为 $P = \{ p \mid (p \leq 10) \wedge (p$ 为素数$)\}$。

(7) 能被 5 整除的数。因为能被 5 整除的数有无穷多个，只能用描述法 $X = \{ x \mid x = 5i, i \in N$ (N 是自然数) $\}$，该式通过说明 x 是自然数 i 的 5 倍来描述 x 能被 5 整除的性质。

严格说来，定义 4–1 只是集合概念的说明，目前在理论上对集合尚无严格的定义，它可以是任何对象的任何种类的汇集。以下是关于集合概念的进一步说明：

- 虽然这里集合的定义要求其元素是互不相同、可分辨的，但一般集合也可包括重复元素，如 $S = \{a, a, b, c\}$ ，学员名册里可有重名的学生。
- 虽然这里将集合定义为具有共同属性的对象的全体，但理论上对集合的元素并没有限制。它可以是任何对象的任何种类的汇集。例如，在集合 $s = \{$ Smith,《集合论》，computer $\}$中，3

个元素分别表示人、书名和物，并没有什么共同的地方，但该集合在理论上仍然成立。

- 集合的成员也可以是另一个（或另一些）集合。例如，集合$\{\{a,b,c\},\ d\}$中有两个元素，$\{a,\ b,\ c\}$和 d。以下的例子进一步说明集合作为集合元素和普通元素的区别。

例 4-3：给定 2 个集合

$$S_1 = \{a,\ b\}，\ S_2 = \{\{a,\ b\}\}$$

其中

$$a \in S_1$$

但是 $a \notin S_2$，因为对 S_2 来说，$\{a,\ b\}$是一个元素

4.1.3 集合间的关系

比较大小是一种最基本的数学运算，子集概念可以说是集合上的比较运算。

定义 4-3：

(1) 设 P、Q 为任意两个无序集合，当且仅当 P 与 Q 有完全相同的元素时，集合 P 与 Q 被认为相等，记作 $P=Q$。

(2) 设 P、Q 为任意两个集合，若对于所有 $p\in P$，必有 $p\in Q$，则称集合 P 是集合 Q 的子集，或称集合 Q 包含集合 P，记作 $P\subseteq Q$ 或 $Q\supseteq P$。子集关系又称为集合的包含关系。

(3) 设 P、Q 是任意两个集合，若对于所有 $p\in P$，必有 $p\in Q$，但反之不然，则称集合 P 是集合 Q 的真子集，记为 $P\subsetneq Q$。

例 4-4：给定以下集合

$$A = \{1,\ 2,\ 3\}，\quad B = \{1,\ 2,\ 3,\ 4\}，\ C = \{1,\ 3,\ 2\}$$

则它们之间有以下关系

（1）$A = C$，因为 A 和 C 有完全相同的元素（因为是无序集合，不考虑元素的顺序）。

（2）$A\subseteq B$，因为凡在集合 A 中的元素必在集合 B 中。

（3）$A\subsetneq B$，因为凡在集合 A 中的元素都在集合 B 中，而 B 中的元素 4 不在集合 A 中。

相等的集合可互为子集，但相等的集合彼此不是真子集；而真子集一定是子集。集合的包含有以下性质：

(1) 空集是所有集合的子集，即对于所有集合 A，$\varnothing\subseteq A$。

(2) 自反性，所有集合都是它们自己的子集 $A\subseteq A$。

(3) 可传递性，如 $A\subseteq B$，$B\subseteq C$，则 $A\subseteq C$。

以上 3 个性质非常直观，很容易理解，就不再赘述了。

4.2 集合运算

4.2.1 集合的运算

在数学中，数的运算法则是比其表示形态更本质的东西。因此，学习任何数都离不开对其运算法则的理解和应用。对集合也是这样，集合的基本运算有 4 种：

定义 4-4：

(1) 集合 P 与 Q 中所有可分辨元素所组成的集合，称为 P 与 Q 的并集，记作 $P\cup Q$。

$$P \cup Q = \{ x \mid (x \in P) \vee (x \in Q) \}$$

(2) 由集合 P 与 Q 共有元素所组成的集合，称为 P 与 Q 的交集，记作 $P \cap Q$。

$$P \cap Q = \{ x \mid (x \in P) \wedge (x \in Q) \}$$

(3) 集合 P 与 Q 的差集是由属于 P 但不属于 Q 的元素所组成的集合，记作 $P \setminus Q$。

$$P \setminus Q = \{ x \mid (x \in P) \wedge (x \notin Q) \}$$

(4) 设有集合 P，P 的补集可定义为全集 U 与 P 的差集，记作 $\sim_U P$，本书记作 P'，$P' = U \setminus P$。

上述集合运算符的优先级以补运算为最高，其余各运算符都是平级的，所以与命题运算一样，当有运算式中包括多个集合时，必须用括号来说明运算顺序。

例 4-5：设有集合

$P = \{1, 2, 3, 4\}$

$Q = \{2, 3, 5, 6\}$

全集 $U = \{1, 2, 3, 4, 5, 6, 7, 8\}$

则

(1) $P \cup Q = \{1, 2, 3, 4, 5, 6\}$，因为集合元素必须是可分辨的，所以同时属于两个集合的元素，如 2、3，在并集里只出现一次。

(2) $P \cap Q = \{2, 3\}$。

(3) $P \setminus Q = \{1, 4\}$，$Q \setminus P = \{5, 6\}$。注意，一般情况下，$A \setminus B \neq B \setminus A$。

(4) $P' = U \setminus P = \{5, 6, 7, 8\}$。

另外，有的教材中还提到对称差运算，但它可视为上述基本运算的组合。

定义 4-5：

集合 P 与 Q 的对称差$\oplus$定义为 $P \oplus Q = (P \setminus Q) \cup (Q \setminus P)$。

由定义可知，所谓对称差，就是那些只在集合 P 或 Q 里、但不同时在两个集合中的元素。

例 4-6：

$$\{a, b\} \oplus \{a, c\} = \{b, c\}$$

$$\{a, b\} \oplus \varnothing = \{a, b\}$$

$$\{a, b\} \oplus \{a, b\} = \varnothing$$

4.2.2 集合的运算定律

集合的运算定律反映集合的恒等性，即相同的集合可用不同方法描述。利用集合运算定律，可将复杂的集合描述简化为简单的集合描述。如表 4-1 所示，主要的集合运算定律包括交换律、结合律、分配律、德·摩根律、吸收律等。集合运算定律均可根据各种运算的定义加以验证，有的定律还可以用文氏图直观地验证（文氏图的概念属选读内容，可参阅书后配盘）。

关于运算定律的若干说明：

(1) 集合是离散数学所处理的对象，所以具有和整数、实数等一样的共性，服从交换律、结合律、分配律等“数”的基本运算规律。

(2) 交换律还可推广到多个集合的情形，如$(A \cap B) \cap C = (B \cap A) \cap C = C \cap (B \cap A)$。

但是，交换只能改变同种算符(交或并)两边集合的次序。以下是一个错误应用交换律的例子：

$$(A \cup B) \cap C \neq (A \cup C) \cap B$$

表 4–1 集合的运算定律

运 算 律	数 学 式
交换律	$A \cap B = B \cap A$，$A \cup B = B \cup A$
结合律	$(A \cap B) \cap C = A \cap (B \cap C)$，$(A \cup B) \cup C = A \cup (B \cup C)$
分配律	$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$，$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
德·摩根律	$(A \cap B)' = A' \cup B'$，$(A \cup B)' = A' \cap B'$
吸收律	$A \cup (A \cap B) = A$，$A \cap (A \cup B) = A$
同一律	$A \cup \varnothing = A$，$A \cap U = A$
互补律	$A \cup A' = U$，$A \cap A' = \varnothing$
对合律	$(A')' = A$
等幂律	$A \cup A = A$, $A \cap A = A$
零一律	$A \cup U = U$, $A \cap \varnothing = \varnothing$

(3) 结合律也可推广到由同种算符(交或并)连接的多个集合之间，但如果括号内的集合包括补集符号时，结合律不成立。

(4) 分配律还可推广到一般情形。若将 n 个集合 $A_1, A_2, \cdots, A_n$ 的并集与交集分别定义为

$$\bigcup A_i = A_1 \cup A_2 \cup \cdots \cup A_n \text{ 和 } \bigcap A_i = A_1 \cap A_2 \cap \cdots \cap A_n$$

则有

$$B \cap (\bigcup A_i) = \bigcup (B \cap A_i) = (B \cap A_1) \cup (B \cap A_2) \cup \cdots \cup (B \cap A_n)$$

和

$$B \cup (\bigcap A_i) = \bigcap (B \cup A_i) = (B \cup A_1) \cap (B \cup A_2) \cap \cdots \cap (B \cup A_n)$$

(5) 德·摩根律是数理逻辑中最有用的定律，在命题逻辑和集合论里中都有德·摩根律，且形式十分相似。它就像普通数运算中的“去括号”运算，可简单记为“当把括号外面的取反符号移入括号内时，括号内的变量和运算符都取其反”。这里的“取反”对于命题变元来说，可理解为取其逻辑非；对于逻辑运算来说，合取和析取则互为“反”运算。对于集合变量来说，“取反”可理解为取其补集，而对于集合运算来说，并与交则互为“反”运算。

以下是运用定律化简集合运算的若干例子。

例 4-7：

(1) $(A \cup (B \cap C)) \cap (A \cup B)$

$=A \cup ((B \cap C) \cap B)$ 分配律

$=A \cup ((C \cap B) \cap B)$ 交换律

$=A \cup (C \cap (B \cap B))$ 结合律

$=A \cup (C \cap B)$ 等幂律

(2) $(B \cap A') \cup (A \cup C)'$

$=(A' \cap B) \cup (A \cup C)'$ 交换律

$=(A' \cap B) \cup (A' \cap C')$ 德·摩根律

$=A' \cap (B \cup C')$ 分配律

4.3 集合模型

与命题模型类似，对集合模型（集合命题模型）的研究和应用也包括以下两个方面：

- 集合建模：将对问题的自然语言描述转换为集合表达式。
- 集合命题推理：利用命题语句表述概念，进行推理和判断等。

4.3.1 集合建模

用集合对事物进行分类是集合模型最常见的应用。集合建模首先将问题的自然语言描述转换为集合表达式，然后再用集合运算符表示有关集合间的关系。

定义集合表达式的问题可归结为给出元素范围或给出元素属性的问题。根据情况可用列举法或描述法。用自然语言描述集合关系的情形虽然一般都比较复杂，但可以按下列规则转换为集合运算符：

(1) 用含有“非”字或其他否定语气的词汇来描述补集。

(2) 用“或”字或其他扩大集合的条件表达式来描述并集。

(3) 用“和”字或者其他限制条件的表达式来描述交集。

(4) 用“不含”、“去除”等限制条件来表示差集。

集合建模要处理的另一方面则是解读命题模型，即用自然语言描述集合表达式的内容。

例 4-8：在使用因特网上的搜索程序时，用户可将各种查找条件组合起来，以便迅速发现有关的信息而又不包括无关的信息。查找条件的组合正是利用了集合运算的原理。例如，google网提供了以下查找选项：

- 搜索结果：包括以下全部子词、包括以下完整字句、包含以下任何一个子词、不包括以下子词。
- 子词位置：网页内的任何地方、网页的标题、网页的内文、网页内的网址、在网页的链接内。试用集合表达式描述下述搜索条件：

(1) 网页的标题内包括以下全部子词。

(2) 网页的内文里不包括以下任何一个子词。

(3) 网页内任何地方都不包括以下子词。

解：先设以下基本集合：

A：网页的标题。

B：网页的内文。

C：网页内的网址。

D：在网页的链接内。

以及 3 个子词集合：$X1$、$X2$、$X3$，则上述搜索条件可以分别表示为：

(1) 网页的标题内包括以下全部子词：$A \cap (X1 \cap X2 \cap X3)$。

(2) 网页的内文里不包括以下任何一个子词：$B \setminus X1 \setminus X2 \setminus X3$。

(3) 网页内的网址或者网页的链接内包括子词 $X1$ 或者 $X2$：$(C \cup D) \cap (X1 \cup X2)$。

本例说明，可利用集合分类事物来组合和优化查询条件，在浩瀚的因特网上迅速捕获所需信息。

例 4-9：给定集合定义

$U = \{ x \mid x$ 是元件$\}$

$W = \{ x \mid x$ 在保修期中$\}$

N = { x | x 是新元件}

S = { x | x 在使用中}

则下列集合表达式可用语言描述为

(1) $N \cap W$　　在保修期内的新元件。

(2) $N \cap S$　　使用中的新元件。

(3) $W \cap (U \setminus S)$　　保修期内，但是尚未使用的元件。

(4) $N \cup W$　　新的或在保修期中的元件。

(5) $((U \setminus N) \cap W) \cap (N \cap S)$　　在保修期内的非新元件或者使用中的新元件。

*4.3.2 集合命题推理

涉及数量的推理方法主要有谓词逻辑和集合命题，但谓词逻辑无法在 C++等一般的编程语言中直接实现，因此用集合论进行推理对于程序设计来说有重要的理论和实际意义。

用集合表示的逻辑命题，称为集合命题或集合语句。常见的集合命题有两种等价的描述模式：

(1) 用子集符号，或用语句“ ___是___的子集 ”、“ ___不是___的子集 ”等表示。以下是一些常见的子集方式命题：

- 所有 S 是 P：$S \subseteq P$，集合 S 中的每个元素都是集合 P 的元素。
- 没有 S 是 P：$S \subseteq P'$，集合 S 中的每个元素都是 P 的补集的元素。

(2) 用集合的相等符号，或者用语句“ ___等于__的集合”、“ ___不等于__的集合”等表示。如上述子集命题可用对应的等式命题表达如下：

- 所有 S 是 P： $(S \cap P') = \varnothing$，$S$ 与 P 的补集之交集为空集。
- 没有 S 是 P： $(S \cap P) = \varnothing$，即 S 与 P 的交集为空集。
- 一些 S 是 P： $(S \cap P) \neq \varnothing$，一些元素在 S 与 P 的交集中。
- 一些 S 不是 P：$(S \cap P') \neq \varnothing$，一些元素在 S 与 P 的补集的交集中。

例 4-10：给定以下集合：

U = { x | x 是顾客 }

C = { x | x 是优先的顾客 }

P = { x | x 已付款顾客 }

S = { x | x 得到账单的顾客 }

下列自然语句的两种模式的命题表示为：

(1) 每个优先的顾客都已付款，可用子集形式表示为 $C \subseteq P$，或表示为集合相等形式 $(C \cap P') = \varnothing$

(2) 每个得到账单的优先顾客都已付款，可用子集形式表示为 $(C \cap S) \subseteq P$

(3) 一些已付款的优先顾客得到了账单，可用集合相等形式表示为 $P \subseteq (C \cup S)$

学员可自行写出（2）和（3）的另一种表示形式作为练习。

集合命题与集合描述的重要区别在于，集合命题有真值，可用于断定事物的真与假，而集合描述只命名集合。若一个命题为真，则其否定为假，反之亦然。

集合命题的真实性取决于两个因素：

- 各组成集合的定义。
- 全集的定义。对给定某一个全集为真的命题，对给定的另一个全集则可能为假。

例 4-11：给定以下集合定义

$U=\{a,b,\cdots,j\}$字母表中前 10 个字符

$A=\{a,b,c,d,e\}$，$B=\{a,c,e,g,i\}$

$C=\{a,c,e\}$，$D=\{a,b,c,e\}$

则下列命题为真

$A\subseteq(B\cup C')$　　$B\neq C$

$D\subsetneq A$　　$(A\cap C)\subseteq D$

下列命题为假

$D\subseteq(C\cap B)$　　$(A\cup C)\subseteq B$

例 4-12：设某人拥有的所有 CD 唱片的集合为 S，其中

M：是现代音乐 CD

C：古典音乐 CD

T：今年新购买的 CD

G：是柜中收藏的 CD

且已知

(a) 所有的现代音乐 CD 都是今年才买的。

(b) 所有的古典音乐 CD 都放置在柜中。

(c) 柜中没有今年新购的 CD。

试证以下论断是否正确。

(1) 柜中没有现代音乐 CD。

(2) 所有的古典 CD 都是今年买的。

解：

首先将已知条件表示为下列值皆为真的集合命题

(a) $M\subseteq T$　　(b) $C\subseteq G$　　(c) $T\subseteq G'$

(1) 由给定条件 $M\subseteq T$ 和 $T\subseteq G'$，可得 $M\subseteq G'$，即所有现代音乐 CD 不在柜中，论断正确。

(2) 由给定条件 $C\subseteq G$ 可得 $C\cap G'=\varnothing$，而 $T\subseteq G'$，则 $G\cap T=\varnothing$，即没有古典 CD 是今年买的，原论断值为假。

另外，集合论中有一些命题既不取决于全集的定义，也与其各组成集合的定义无关。这些语句称为逻辑真（对其否定则称为逻辑假），它们的真或假只取决于运算的定义和关系描述。以下就是集合命题推理中常用的一些逻辑真语句。

(1) $A=A$　　(2) $A\subseteq A$

(3) $A\subseteq U$　　(4) $\varnothing\subseteq A$

*4.4 集合基数推理

有限集中可分辨元素的个数称为集合的基数（cardinal number），用$|A|$表示。对集合中元素数量进行定量推理也是集合模型应用的一个重要方面。以下是基数运算的一些基本法则。

定义 4-6：有限集合的基数运算法则（应用这些运算法则的前提是集合中没有重复的元素）：

(1) $|P\cup Q|=|P|+|Q|-|P\cap Q|$。

(2) $|P \oplus Q| = |P|+|Q|-2|P \cap Q|$。

(3) $|P \cap Q| \leq \min(|P|,|Q|)$。其中，min(…)的意思是取两者中的最小值。

(4) $|P-Q| \geq |P|-|Q|$。该公式还可推广到多个集合的情形。

例 4-13：

设全班 50 个学生中，有 27 人在第一次考试中得 A，23 人在第二次考试中得 A，但有 15 人两次考试都没得 A，问两次都得 A 的学生有多少？

解：设

A 为第一次考试得 A 学生的集合，则$|A|=27$。

B 为第二次考试得 A 学生的集合，则$|B|=23$。

而全体学生的集合 U 的基数等于 50，即$|U|=50$。

曾经得到过 A 的学生就是 A 与 B 的并集$(A \cup B)$。

两次都没得 A 的学生就等于 $U \setminus (A \cup B)$，$|U \setminus (A \cup B)|=15$，则$|A \cup B|=35$。

两次考试都得 A 的学生的集合是 $A \cap B$，$|A \cap B|=|A|+|B| \setminus |A \cup B|=27+23-35=15$。

答：有 15 个学生两次考试都得 A。

例 4-14：

为开发某软件调研了 30 个用户，其中有 15 人希望增加 A1 功能，8 人希望增加 A2 功能，6 人希望增加 A3 功能，3 人希望增加所有 3 项功能。试求：多少人希望增加功能，又有多少人没有要求增加功能。

解：设

$A1$ 为希望增加 $A1$ 功能用户的集合，$|A1|=15$。

$A2$ 为希望增加 $A2$ 功能用户的集合，$|A2|=8$。

$A3$ 为希望增加 $A3$ 功能用户的集合，$|A3|=6$。

因为有 3 人希望增加所有 3 项功能，所以$|A1 \cap A2 \cap A3|=3$。

将公式（1）应用到 3 个集合，得到以下式子

$$|A1 \cup A2 \cup A3| = |A1|+|A2|+|A3|-|A1 \cap A2|-|A1 \cap A3|-|A2 \cap A3|+|A1 \cap A2 \cap A3|$$
$$= 32-|A1 \cap A2|-|A1 \cap A3|-|A2 \cap A3|$$

因为

$$|A1 \cap A2| \geq |A1 \cap A2 \cap A3|，|A1 \cap A3| \geq |A1 \cap A2 \cap A3|，|A2 \cap A3| \geq |A1 \cap A2 \cap A3|$$

所以 $|A1 \cup A2 \cup A3| \leq 32-3-3-3 = 23$，即至多有 23 人希望增加功能，而没提要求的用户不少于 7 人。

习　题

知识点：

本章练习的重点是：

- 熟悉集合的基本运算法则。
- 将自然语言表达为集合公式和用自然语言描述集合公式。

1. 写出下列集合的表达式：

(1) 小于 20 的素数的集合。

(2) 正奇数的集合。

(3) 所有一元二次方程解的集合。

(4) 某电视台制作一个为时 15 分钟的节目，其中包括戏剧、音乐与广告。以 5 分钟为一基本单位，求可能的节目安排方案的集合。

2. 计算下列各式：

(1) 给定：$A=\{a,b,c\}$, $B=\{a,1,2,3\}$, $U=\{1,2,3,a,b,c\}$

求 $A \cup B$，$A \cap B$，$A \setminus B$，$B \setminus A$, A', B'，$A' \cup B'$，$A' \cap B'$。

(2) 用上式给定集合验证德·摩根律。

3．试求集合 $A=\{\varnothing, 1, a\}$ 的所有子集和真子集。

4. 化简下列各式：

(1) $(A \setminus B) \cup B$

(2) $(A \cap B) \cup (A \cap B')$

(3) $((A \cup B \cup C) \cap (A \cup B) \setminus (A \cup (B \setminus C)) \cap A)$

5. 利用定义

$U=\{x \mid x$ 是图书馆藏书$\}$

$R=\{x \mid x$ 是参考书$\}$

$T=\{x \mid x$ 是教科书$\}$

$F=\{x \mid x$ 是小说$\}$

$C=\{x \mid x$ 是计算机类书$\}$

将下列集合描述转换为集合表达式：

(1) 非小说类书。

(2) 参考书与非小说。

(3) 既非参考书，又非小说。

(4) 计算机方面的教科书和参考书。

(5) 非计算机类的教科书和参考书。

6. 利用基本集合定义

$S=\{x \mid x$ 是售货员$\}$

$F=\{x \mid x$ 是百货部$\}$

$T=\{x \mid x$ 是服装部$\}$

$M=\{x \mid x$ 完成了定额$\}$

$B=\{x \mid x$ 得到了奖金$\}$

将下列集合描述转换为集合表达式：

(1) 百货部的每个售货员都完成了定额。

(2) 服装部的一些售货员没有完成定额。

(3) 百货部和服装部完成定额的售货员都得到了奖金。

7. 利用下列基本集合定义

P：价格 12 000 元以下

C：笔记本型

G：液晶显示器

W：3 年保修期

A：有内置网卡

描述下列集合表达式：

(1) $P \cap C$　(2) $A \cup G$　(3) $G \setminus A$　(4) $P \cap (G \cup W)$　(5) $P \setminus (G \cup A)$

8. 微机的 Windows 操作系统提供了查找文件或文件夹的功能，可根据用户指定文件类型、文件生成日期、大小查找位于一个和多个磁盘上的文件。若给定下列集合定义：

U：我的电脑里的全部文件

C：储存在 C 盘上的文件

D：储存在 D 盘上的文件

T：指定类型的文件

W：包含指定文字的文件

B：在给定日期 b 以前生成的文件

A：在给定日期 a 以后生成的文件

L：大小至少为某数值的文件

S：大小至多为某数值的文件

用自然语言描述下述集合表达式：

(1) $T \cup W$

(2) $B \cap A$

(3) $(C \cup D) \cap W$

(4) $(U \setminus C) \cap S \cap L$

9. （选做）基数计算

(1) 全部 50 个学生，26 人在第一次考试得 A，21 人在第二次考试得 A，但有 17 人从来没得过 A，问有多少人 2 次考试都是 A？

(2) 已知两次考试中得 A 的学生数相等，但其中 40 名学生只得过一个 A，4 名没得过 A，分别求：只有第一次考试得 A 的学生数、只有第二次考试得 A 的学生数、两次考试都得 A 的学生数。

第5章 图 论

知识点：

① 图论中所谓的图并非几何意义上的图形，而是指事物之间的联系。所以，培养图论建模的抽象思维能力是本章学习的一个重点。通过学习能对实际问题进行抽象，建立图或树的模型是本章学习的第一点要求。

② 本章首次引入了算法的概念，通过设计算法而不是应用计算式来解决问题是图论的一大特点。因此，熟悉图论（树）的一些常用的算法是本章学习的第二点要求。

5.1 图与树

图论（包括树）是离散数学中的重要分支，在计算机科学、物理学、化学、交通运输、管理科学等许多领域均有广泛的应用。图论的内容十分丰富，本章着重介绍与程序设计有关的图和树的一些最基本的概念。

图论里所说的图并非几何意义上的图形，而是指事物之间的联系。如在体育比赛的循环赛中，每个球队都要与其他各队比赛；又如在学校中，每个教师可讲授若干门课，而多个教师又可在同一时间对不同班级教授同一课程。如何用数学来描述这样相互之间有错综复杂关系的数据，就是图论研究的内容之一。

5.1.1 图

定义 5-1：

图（graph）是一个二元组 $G=(V, E)$**，其中**

(1) V **是一有限非空集合，**$V=\{v_0, v_1, \ldots, v_{n-1}\}$**，称为** G **的结点或顶点（vertex）。**

(2) E **是一有限非空集合，**$E=\{e_0, e_1, \ldots, e_{m-1}\}$**，称为** G **的边（edge）或弧（arc），其元素** e **是由** V **中元素组成的对偶，**$e_i=(v_j, v_k)$**。如对偶是有序的，则称为有向边，否则就是无向边。**

定义 5-2：

(1) **有向图（directed graph，digraph）：全部边均为有向边的图。**

(2) **无向图：全部边均为无向边的图。**

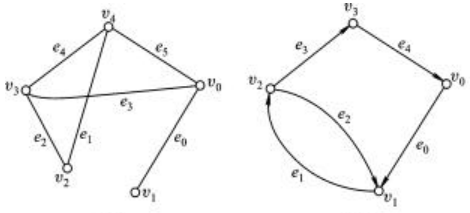
(3) **混合图：兼有有向边和无向边的图。**

图还可用图解法表示，将图表示为图形，有助于直观地理解图论的许多定义和概念。具体方法是：

- **用直线或弧线表示边，用箭头表示有向边的方向，边上可以加注描述边的数据，如边的名字、长度等。在保持结点和边关系不变的情况下，边的位置、大小和形状都是无关的。**
- **用圆圈或者方框表示结点，圆圈和方框中内容是描述结点数据的，如名字等。为作图方**

便起见，常常只用小圆圈或者黑点表示结点，甚至什么图形都不用，只用边的交点表示图的结点。所有这些图示方法在后续内容中基本上都有介绍。

例 5-1：图 5-1(a)的边没有方向，为无向图，图 5-1(b)为有向图，箭头表示边的方向。它们可以用定义式分别表示如下：



(a) (b)

图 5–1 图的图形表示法

图 5-1(a)中 $V=\{v_0,v_1,v_2,v_3,v_4\}$

$$E=\{(v_0v_1),(v_0v_3),(v_0v_4),(v_2v_3),(v_2v_4),(v_3v_4)\}$$

为方便起见，也可用名字指代边，如 e_0, e_1, e_2, …

图 5-1(b)中 $V=\{v_0,v_1,v_2,v_3\}$

$$E=\{(v_0v_1),(v_1v_2),(v_1v_3),(v_2v_3),(v_3v_1)\}$$

5.1.2 图的性质

定义 5-3：

(1) 无向图中，顶点 u 关联的边数称为 u 的度（或度数），记为 $\deg(u)$。

(2) 有向图中，以顶点 u 为起点的边数称为 u 的出度，记为 $\deg^+(u)$，以顶点 u 为终点的边数称为 u 的入度，记作 $\deg^-(u)$。出度与入度的和仍称为顶点 u 的度，记为 $\deg(u)$。

定理 5-1：

在一个有 n 个顶点和 m 条边的(n, m)图中，其顶点度的总和等于其边数的 2 倍。

定理 5-2：

在一个有 n 个顶点和 m 条边的(n, m)图中，顶点度为奇数的顶点数必有偶数个。

上述定理无论对有向图还是无向图都是成立的。

例 5-2：

（1）图 5-1(a)有 5 个顶点、6 条边，其顶点度的总和=1+2+3+3+3=2×6，其中奇数度的顶点有 4 个。

（2）图 5-1(b)有 4 个顶点、5 条边，其顶点度的总和=2+2+3+3=2×5，其中奇数度的顶点有 2 个。

定义 5-4：

(1) 多重边：两个顶点间一条以上的边称为多重边，也称为平行边。

(2) 多重图：含有平行边的图。

(3) 自回路：两个组成顶点为同一顶点的边称为自回路。

(4) 简单图：不含平行边和自回路的图称为简单图（本教材后续章节中将只考虑简单图）。

显然，有向图中才会有多重边或者平行边，如图 5-1(b)中有平行边 e_1 和 e_2，因此是一个多重图。图 5-2 是有自回路的图的例子。

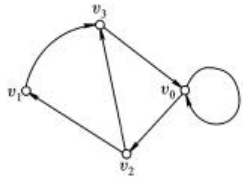


图 5-2 含自回路的图

5.1.3 完全图和子图

定义 5-5（完全图）：

(1) 若一个简单无向图的任意两顶点之间有且仅有一条边，则称为完全无向图。

(2) 若一个简单有向图的任意两顶点之间有且仅有一条边，则称为完全有向图。

定理 5-3：

在一个(n, m)完全图中，顶点数和边数之间存在以下关系：$m = n(n-1)/2$

由定义可知，完全图中每个顶点和其他所有顶点都有边相联，所以可利用组合公式计算出其边数 $m = C_n^2 = n(n-1)/2$，这就是定理 5–3 中公式的来由。若给定顶点集合，无向图有惟一的完全图；但对于有向图，由于边的方向不同，它可有多个完全图。

例如，图 5-1（a）就不是完全图。若要使其成为完全图，它应该有 5×(5-1)/2=10 条边，还须增加 4 条边 v_1v_2, v_1v_3, v_2v_0, v_1v_4。图 5-1（b）也不是完全图，因为它有多重边，不满足两顶点之间有且只有一条边的规定。

定义 5-6：

对于图 $G=(V, E)$和 $G_1=(V_1, E_1)$，有

(1) 若 $V_1 \subseteq V$，$E_1 \subseteq E$，则称 G_1 为 G 的子图。

(2) 若 $V_1 \subseteq V$，$E_1 \subsetneq E$，则称 G_1 为 G 的真子图。

(3) 若 $V_1=V$，$E_1 \subseteq E$，则称 G_1 为 G 的生成子图。

例 5-3：

图 5–3 是对上述定义的图解。图 5-3(b)是图 5-3(a)的生成子图，图 5-3 (c)是图 5-3 (a)的真子图。它直观地表明：生成子图的边可少于原图，而顶点数与原图一定相同，而真子图的边一定比原图少，顶点则少于和等于原图。联想关于子集的定义，可以帮助理解子图的定义。

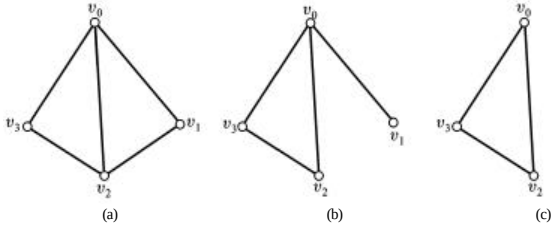


图 5–3 图的生成子图和真子图

在图的生成子图中，有一类非常重要的子图，就是图的生成树。生成树对于解决与图有关的问题来说有重要意义，详见后续章节关于克鲁斯克尔算法和狄克斯特算法的内容。

5.1.4 图的同构

判断两个数是否相等是数学的基本运算，图也是一种广义的数，图的相等称为同构。判断两图同构的必要条件为（但不是充分条件）：

(1) 顶点数相同。

(2) 边数相同。

(3) 度数相同的顶点数相同。

例 5-4：

尽管图 5–4 中的两个图看起来很不相同，但它们是同构的，因其均有 6 个顶点、9 条边，顶点的度数均为 3。而且，其顶点和边一一对应。

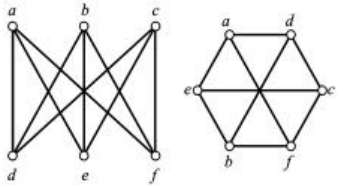


图 5–4 同构图示例

在解决图论有关问题时，经常需要在保持图同构的情况下改变图的图形，以便判定图的性质，但由于改变后的图形往往会与原图看上去相差甚远，进行图的同构变换是一个比较复杂的运算，有关变换理论和方法，可参看有关图论的专门论著。这里主要是利用人的视觉判断能力进行同构变换。下节将介绍图的同构变换的一个重要应用：判断一个图是否平面图。

5.1.5 平面图

在用图解决自然图形问题时，图的边经常会交叉。若经过适当同构变换可以消除相交，则称为平面图。平面图在印刷电路板和集成电路的设计中有重要应用。如果一个电路是平面图，则可将其制作为一层的印刷电路板；若交叉不可避免，则必须制作成多层印刷电路板。以下是平面图的有关概念和定理。

定义 5-7（面）：

在一个图中，若由边所包围的一个区域中既不包含图的顶点、又不包含图的边，则称为图的一个面；而包围该面的诸边所构成的回路称为该面的边界。若该面的面积有限，则称为有限面；反之则称为无限面。

例 5-5：

图 5–5 所示之图共有下列 4 个面。其中前 3 个为有限面。

(1) 左面的白色三角形区。

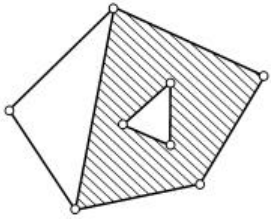


图 5–5 图的面

(2) 右边的白色小三角形区。

(3) 介于四边形和小三角形之间的阴影区。

(4) 还有一个无限面：整个图形外部的区域。

定理 5-4：

平面图满足欧拉公式 $n - m + r = 2$，其中 n 为顶点数，m 为边数，r 为面数。

利用欧拉公式，可判定一个连通图是否为平面的，但它只是必要条件，而不是充分条件。也就是说，平面图必须满足欧拉公式，但满足欧拉公式的并不一定就是平面图。

例 5-6：

在图 5–6(a)中，初看有一些交叉边，但经过同构变换后的图 5–6 (b)就是一个平面图，其顶点数 n=5，边数 m = 9，面数 r = 6（5 个有限面，1 个无限面），$n–m+r$=5-9+6=2。

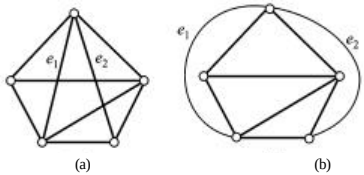


图 5–6 平面图的判定

5.1.6 有权图和网络

在实际应用中，经常在图的边上附加一些信息。这个附加的信息称为边的权，或权重(weight)。权一般定义为边集上的非负实函数。边上带权重的图称为有权图，也称网络(network)。如高速公路网可视为以距离作为边的权重的图，其他如供电网、供水网、燃气输送管道等，也可视为有权图。复杂问题中的权函数往往包括多个变量。有权图的应用是图论研究的重要领域。

例 5-7：

图 5–7 所示的例子为一个因特网，能根据路径的权重来选择数据传送的最佳路径。其权函数中包括了多个变量：途径的路由器数量（Hop count）、网络线路的带宽（Bandwidth）、传送过程中的延时（Delay）、线路的可靠性（Reliability）和线路上的负载（Load）以及费用（Cost）等。

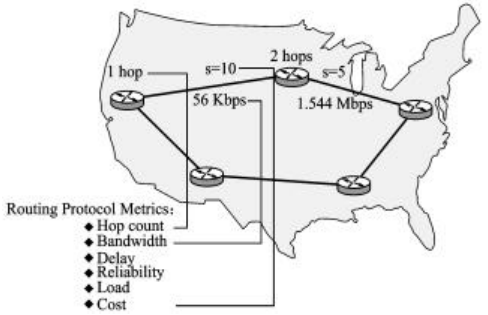


图 5–7 多变量权函数

5.1.7 树和根树

树是一种特殊的图。对图进行一些运算，如发现通路的结果所生成图的子集就是树。本节

介绍有关树的基本概念。

定义 5-8（树）：

(1) 不包含回路的图称为树，由有向图衍生的树称为有向树，而由无向图衍生的称为无向树。

(2) 树中度数为 1 的结点称为叶结点，度数大于 1 的结点称为分支结点。

(3) e(边数)= v(结点数)−1。

图 5–8 直观地描述了树与图的区别。

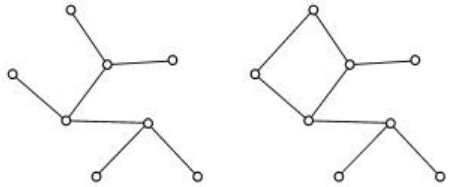


图 5–8 树与图的区别

在树中应用最多的是所谓根树，以下是关于根树的有关概念。

定义 5-9（根树）：

(1) 若一有向树中只有一个入度为零的结点，而其余结点的入度皆为 1，则称为根树。

(2) 入度为零的结点称为根（或根结点）。

(3) 那些出度为零的结点称为叶（或叶结点）。

(4) 除了根和叶，其余的结点为分枝结点。

(5) 根树中边的方向始终是从上一级结点指向下一级结点，为简便起见一般不再用箭头明示。

为方便作图起见，一般都将树根画在最高层，以便于往下展开子树。以下用图 5–9 定义一些描述根树常用的术语。

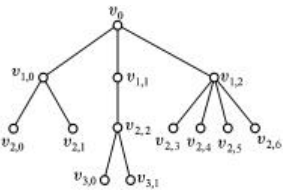


图 5–9 根树

(1) 根结点：入度为零的结点，如图 5–9 中 v_0。

(2) 子结点与父结点：若有边从结点 a 指向 b，则称 a 是 b 的父结点，而 b 即是 a 的子结点，如图 5–9 中 $v_{1,0}$ 是 $v_{2,0}$，$v_{2,1}$ 的父结点，而 $v_{2,0}$，$v_{2,1}$ 则是 $v_{1,0}$ 的子结点。

(3) 祖先与后裔：若从结点 a 有路可达结点 b，但 a 又不是 b 的父结点，则称 a 为 b 的祖先，而 b 即为 a 的后裔。如图 5–9 中 $v_{1,1}$ 是 $v_{3,0}$、$v_{3,1}$ 的祖先，而 $v_{3,0}$、$v_{3,1}$ 则是 v_0 的后裔。

(4) 兄弟：同一父结点的子结点彼此为兄弟，如图 5–9 中的 $v_{1,0}$、$v_{1,1}$、$v_{1,2}$。

(5) m–叉树：若一个树中所有分叉结点的出度皆不大于 m，则称该树为 m–叉数。若树中每个分支结点的出度都恰好等于 m，则称为完全 m–叉树。

(6) 层：规定根树的层数为 0，如结点 a 至根的距离为 n，则 a 就在第 n 层上。

(7) 树高：树的高度等于其叶结点层数的最大值。如图 5–9 中树的高度为 3。

(8) 有序树：若各层上的结点按一个约定的顺序排列，则称为有序树。

5.1.8 二叉树

导读：

本节首次应用了递归（recursion）形式来定义二叉树，这是理解的一个难点，但它又是计算机学科经常使用的一种数学方法。

当树的叉数 m 等于 2 时，称为二叉树，它在计算机学科中有广泛应用。如图 5-10 所示，二叉树的特点是每个结点最多只有两个子结点，并且二叉树是有序的，其子结点有左右之分，子树也有左右之分，顺序不能颠倒。定义 5-10 是二叉树的数学定义。

定义 5-10（二叉树）：

二叉树（binary tree）是结点的一个有限集合，该集合或者为空，或者是由一个根结点加上称为左子树和右子树的两个分支、互不相交的二叉树组成。

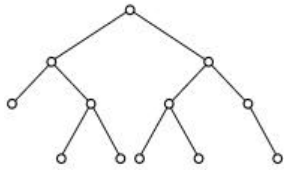


图 5–10 二叉树示例

注意定义 5-10 采用了递归形式，这为实现二叉树的数据结构提供了数学理论基础。所谓递归是一种循环推理方式，其中每次循环都会导致一个与现有表达式结构相似、但复杂程度降低的表达式，直至最后得到一个不能继续展开的最简式。例如，自然数的阶乘可定义为：$n!=n\times(n-1)!$。等号右边的$(n-1)!$在形式上与左边的 $n!$完全相同，但复杂程度低一级。这就是递归形式的运用。

根据递归概念，可以如此理解定义 5–10：二叉树是由两个比它小的二叉树分别组成左右分支构成的，而每个分支又是由比它还小的二叉树分别组成左右分支构成的，余此类推，直至分支为空，即到叶结点为止。这就是一种递归形式。在计算机学科中，常用递归形式定义概念、数据结构、算法等，关于递归概念还将在 10.3 节予以详细论述。

5.2 图的运算

2.1.2 节指出，学习数学最重要的就是学习如何进行运算。和迄今为止所学过的数学（包括普通数学和刚学过的命题逻辑、集合逻辑）不同，图论中的运算无法用简单的运算式来表示，而要经过一系列运算步骤才能完成。对这些步骤的描述称为“算法（algorithm）”，算法是用数学方法精确定义的解决问题的规则或步骤。设计算法也是学习软件技术的重要内容。

图论的主要运算是用算法发现各种各样的路，它是应用图来解决实际问题的基础。本节首先介绍关于路的概念，然后讨论关于一些具有重要应用价值的路的算法。

5.2.1 图的连通性

定义 5-11：

图 $G=(V=\{v_0, v_1, \ldots, v_{n-1}\}, E=\{e_1, e_2, \ldots, e_{m-1}\})$**中**

(1) 首尾相接的边的有序集合$\{(v_0,v_1), (v_1,v_2), \ldots, (v_{k-1},v_k)\}$，称为联结顶点 v_0 到 v_k 的路，该

集合中边的数量称为路的长度。

(2) 若一条路的最后一条边与其第一条边首尾相接，即 $v_0=v_k$，则称为回路，否则称为开路。

(3) 顶点 $v_0, v_1, v_2, \ldots, v_k$ 各不相同的开路亦称为通路; $v_0, v_1, v_2, \ldots, v_k$ 各不相同的回路则称为圈。

(4) 若通路所有的边各不相同，则称为简单通路；反之，有边重复出现的路称为复杂通路（本教材后续章节中将只考虑简单通路）。

(5) 若回路的所有边各不相同，则称为简单回路；反之，有边重复出现的路称为复杂回路（本教材后续章节中将只考虑简单回路）。

定义 5-12：如果图中存在一条从顶点 u 至 v 的路时，则

(1) 对无向图，则称 u，v 之间是连通的，并约定顶点与其自身是连通的。

(2) 对有向图，则称 u，v 之间是可达的，并约定顶点与其自身是可达的。

(3) 若一个图中任意两顶点都是连通的，则称该图为连通的，否则就是不连通的（本教材后续章节中将只考虑连通图）。

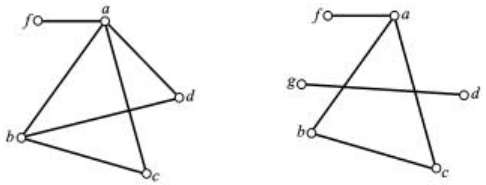
例 5-8：

（1）图 5–11(a) 中所有顶点之间都有边相连，为连通图。

（2）图 5–11(a) 中$\{(b, c), (c, a), (a, f)\}$是从顶点 b 至 f、长度为 3 的路。

（3）图 5-11(a) 中顶点 f 至 c 有 2 条路，f-a-c 和 f-a-b-c，两顶点间距离按其中短的计算，为 2。

（4）图 5–11(b) 中顶点 d 只与顶点 g 连通，它们与其他顶点不连通，所以是一个不连通图。



(a) 连通图　　(b) 不连通图

图 5–11　图的连通性

5.2.2 欧拉回路

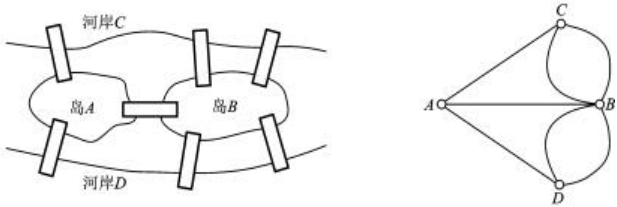
欧拉回路是数学家欧拉在研究著名的德国哥尼斯堡（Koenigsberg）七桥问题时发现的。如图 5–12(a)所示，流经哥尼斯堡的普雷格尔河中有两个岛，两个岛与两岸共 4 处陆地通过 7 座桥彼此相联。7 桥问题就是如何能从任一处陆地出发，经过且经过每个桥一次后回到原出发点。这个问题可抽象为一个如图 5–12(b)所示的数学意义上的图，其中 4 个结点分别表示与 4 块陆地对应，如结点 C 对应河岸 C，结点 A 对应岛 A 等，而结点之间的边表示 7 座桥。欧拉由此提出了著名的欧拉定理。

定义 5-13：

(1) 欧拉路：通过图中所有边的简单路。

(2) 欧拉回路：闭合的欧拉路。

(3) 欧拉图：包含欧拉回路的图。



(a) 7桥地理位置示意　　(b) 抽象为数学上的图

图 5–12 哥尼斯堡 7 桥问题

定理 5-5：欧拉定理

一个连通图为欧拉图的充分必要条件是它的每个结点的度数皆为偶数。

根据欧拉定理，7 桥问题中 4 个顶点的度数皆为奇数，所以不可能经过且只经过每个桥一次回到出发点。定义 5-13 无论对有向图还是无向图均适用，但对于有向图，其中的边和路都是有单一方向的。

若一个连通图的所有结点的度数皆为偶数，则可用弗劳利（Fleury）算法求出其欧拉回路。

算法 5-1：弗劳利算法

(1) 选择任意一个结点作为起点，选择一条从它开始的边，将其加到解的集合中。

(2) 从该边的终端，选择与之连通的下一条边作为新的起点，将已经过的边从图中删除。

(3) 重复步骤(1)和(2)，如若经过一个结点的所有边均被删除，则将该结点也删除。

(4) 直到所加入的边的终点与起点重合，形成回路为止。

例 5-9：图 5-13 为某地区的道路图，试设计一条能不重复地经过所有道路的环行公交线路。

解：该问题的实质就是求图 5-13 的欧拉回路。

根据欧拉定理首先判定图中确实存在欧拉回路。因为所有结点的度数皆为偶数，所以符合条件。

用弗劳利算法求解图 5–13 中欧拉回路的过程如表 5-1 所示。

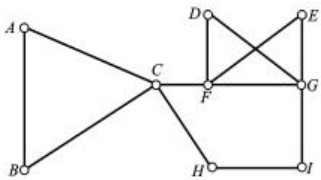


图 5–13 求解欧拉回路

表 5-1 用弗劳利算法求解例 5-9 的欧拉回路步骤

步　骤	当前选择边	欧拉回路解集
0	*CH*	*CH*
1	*HI*	*CH, HI*
2	*IE*	*CH, HI, IE*
3	*EF*	*CH, HI, IE, EF*
4	*FD*	*CH, HI, IE, EF, FD*
5	*DG*	*CH, HI, IE, EF, FD, DG*

续表

步　骤	当前选择边	欧拉回路解集
6	*GC*	*CH, HI, IE, EF, FD, DG, GC*
7	*CB*	*CH, HI, IE, EF, FD, DG, GC, CB*
8	*BA*	*CH, HI, IE, EF, FD, DG, GC, CB, BA*
9	*AC*	*CH, HI, IE, EF, FD, DG, GC, CB, BA, AC*

最后求得的线路是：*CH, HI, IE, EF, FD, DG, GC, CB, BA, AC*。

显然，对于给定的图，随起始边选择的不同，其欧拉回路也不同。读者可以尝试选择另一条边为起始边，寻找另一条欧拉回路。

5.2.3 哈密顿回路

哈密顿（Hamilton）回路（或称 Hamilton 图）也是一种很有意义的图。它与欧拉图正好互相呼应，欧拉回路要求通过每条边一次且仅一次，哈密顿回路则要求通过每个顶点一次且仅一次。哈密顿图的一个重要应用是所谓货郎担问题（traveling salesperson problem, TSP），要求在图中发现经过所有顶点且总距离最短的路线。注意这里所说的距离是指路径上所有边的权的总和，而不是定义 5–11 中所说的路的长度。

与欧拉图不同，迄今为止还没有发现一个能简单判定哈密顿图的充要条件。而且，从算法设计理论来说，没有有效方法可求得该问题的精确解（参见 5.3.3）。但是有些算法，如最近邻居算法（nearest neighbor algorithm）、最佳边算法（best-edge algorithm），可给出足够好的结果。

算法 5-2：nearest neighbor algorithm

(1) 从任何结点开始，将其加入到解的集合中。

(2) 从与该结点连接的边中选择最短的那条边的结点加入到解的集合中，这就是所谓的最近邻居。若同时有多条边距离相等，则可任选一条。

(3) 从上述运算所选的最近邻居出发，重复上述过程，但应避免已选择过的结点，以免形成回路。

(4) 当所有结点都加到解的集合中后，将最后加入的结点与起始结点连接，就得到哈密顿回路。

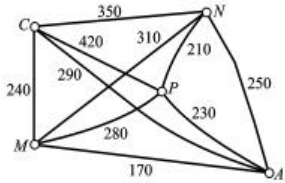


图 5–14　求解 TSP 问题

例 5-10：图 5-14 表示的是若干城市之间的距离，某公司欲从 *P* 地出发，在其各周边城市举办产品展销会，求解一条能不重复地经过各地的最短路径。

解：哈密顿问题之所以又名货郎担问题，就是来源于这样流动销售的问题。该问题可以用最近邻居算法求解，求解步骤如表 5-2 所示。

表 5-2　用最近邻居法求解例 5-10 的步骤

迭 代 次 数	最 近 邻 居	与最近邻居连接各边长度
0	*P*	*PN*=210, *PA*=230, *PM*=280, *PC*=420
1	*N*	*NA*=250, *NM*=310, *NC*=350
2	*A*	*AM*=170, *AC*=290
3	*M*	*MC*=240
4	*C*	

最后求得的哈密顿回路是 $P–N–A–M–C–P$，总距离为 210+250+170+240+420=1 290。

5.2.4 生成树和最小费用生成树

生成树（Spanning Tree）是图中的一种特殊通路，在实际应用中有广泛意义。例如，若将道路网表示为一个图，则生成树就表示从某地出发、到达所有其他各地且不绕圈子的直达路径。所谓直达即不经过同一条边两次。

用不同的遍历方法，可从图得到不同的生成树，从不同的顶点出发，也得到不同的生成树。但是，一个连通图的生成树一定是原图的极小连通子图，包含原图所有顶点和 $n–1$ 条边；而遍历不连通图或有向图得到的一般都是由若干生成树组成的一个生成森林。

定义 5-14：

(1) 若一个图的生成子图是一个树，则称为该图的生成树。

(2) 在一个加权图的所有生成树中，其边权之和最小的为最小费用生成树（minimum-cost spanning tree, MST）。

(3) 最小生成树不是惟一的。

求有权图的最小费用生成树有着重要的工程实际意义。例如，要在 n 个地点之间开通道路，或架设通信线路，如何使总造价最少，就是这类的问题。最小生成树可用克鲁斯克尔算法或者 Prim 算法求得。

算法 5-3：克鲁斯克尔（Kruskal）算法

(1) 给定有 n 个顶点的有权图 $G=(V, E)$，定义一个边和顶点皆为空集的图 $T=(\varnothing, \varnothing)$为解的集合。

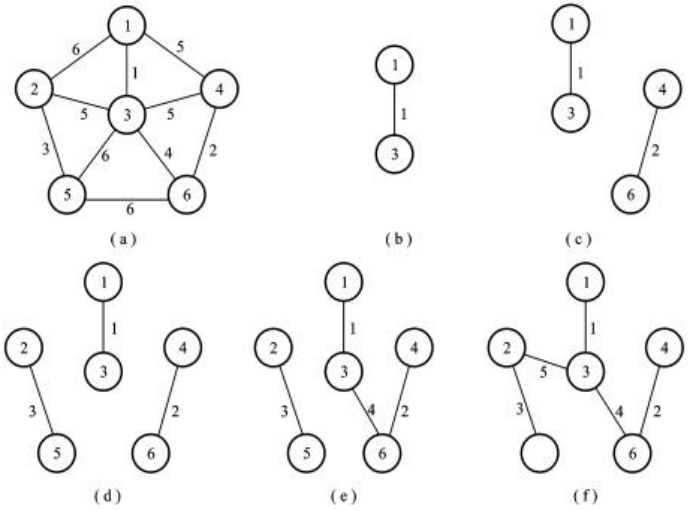


图 5–15 用克鲁斯克尔算法求最小生成树

(2) 依次在集合 E 中选取权值最小的边，将其加到 T 中，若该边加入后，图中出现回路，则将其删除，选取下一个权值最小的边；

(3) 重复步骤(2)，直到所有顶点都在同一个连通分量上时为止。

例 5-11：图 5-15（a）中结点表示的是某地新建的一些居民小区，边上的权重表示这些居民小区之间的距离，电力公司需要在这些小区之间架设输电线路，设计一个可使总长度最短的线路架设方案。

解：该问题实质上是求图 5-15 的最小生成树的问题。

图 5–15 (b)～(f)就是根据克鲁斯克尔算法求其最小生成树的过程：依次加入边(1, 3)（权重 1)、边(4, 6)（权重 2)、边(2, 5)（权重 3)、边(3, 6)（权重 4)、边(2, 3)（权重 5)，最后得到图(5)，即为所需要的线路架设方案，总线路长度为 15。

5.2.5 狄克斯特算法

在图形应用中，经常还需要求从求图中某个结点至其余各结点的最短路径，如对于一个物流配送系统计算从配送中心到各订货点的最短路径。狄克斯特（Dijkstra）算法可用于此目的。

算法 5-4：狄克斯特算法

若给定带权有向图 $G=(V, E)$和源顶点 v_0，构筑一个源集合 S，将 v_0 加入其中。

(1) 对差集 $V \setminus S$ 中各顶点 v_i，逐一计算从 v_0 至它的距离 $D(v_0, v_i)$，若该两顶点之间没有边，则其距离为无穷大。求出其中距离最短的顶点 w，将其加入到集合 S 中。

(2) 重新计算 v_0 至差集 $V \setminus S$ 中各顶点的距离 $D(v_0,v_i) = \mathrm{Min}(D(v_0,v_i), D(v_0,w)+C(w,v_i))$。其中 $C(w, v_i)$是顶点 w 与 v_i 之间边上的费用。

(3) 重复步骤(1)和(2)，直至所有的顶点都加到集合 S 中为止。

例 5-12：图 5-16(a)为某物流配送系统，边上数字是各地间距离，配送中心位于结点 1 处，试设计从结点 1 至其他各结点路线最短的送货路线。

解：该题的实质就是求从结点 1 至其余各结点的最小费用生成树。

图 5–16 的(b)～(e)即为采用狄克斯特算法求解过程的图示，计算步骤和中间数据如表 5-3 所示。

表 5-3 例 5-12 求解步骤

迭代次数	加入解集的结点	生成树	顶点 1 至其余各顶点距离			
			$D(2)$	$D(3)$	$D(4)$	$D(5)$
0	1		10	∞	30	100
1	2	图(b)	10	60	30	100
2	4	图(c)	10	50	30	90
3	3	图(d)	10	50	30	60
4	5	图(e)	10	50	30	60

最后求得从结点 1 至其余各结点的路径为：1→2（距离 10），1→4（距离 30），1→4→3（距离 50），1→4→3→5（距离 60）。

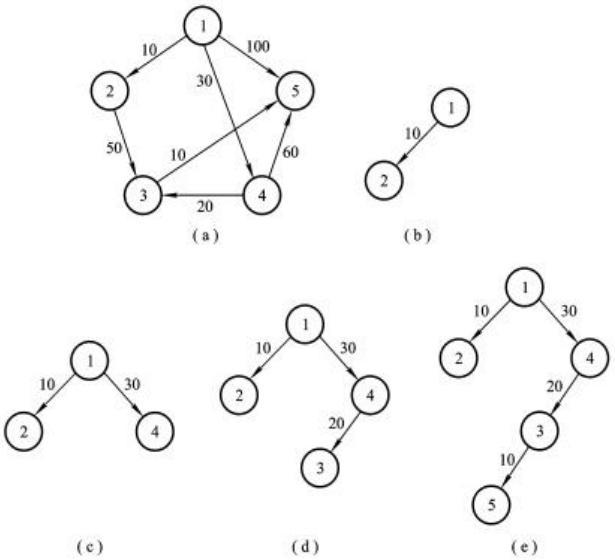


图 5–16 狄克斯特算法过程示例

由例 5-11 和例 5-12 可知，狄克斯特算法和最小生成树的区别在于：

(1) 最小生成树是对全图而言的，而狄克斯特算法是对某个结点而言的。

(2) 最小生成树是连接所有结点的最短路径，但如果从某结点出发，沿着最小生成树到另一个结点的路径不一定是最短的。而在狄克斯特树中，从根结点到各叶结点的路径都是最短的。

若将狄克斯特算法依次应用于每个顶点，最后可得到任意两个顶点之间的最短路径，这就是通常所说的任意顶点对之间最短路径问题（all-pairs shortest paths, APSP）。求 APSP 还有一个更直接的 Floyd 算法，这里不再详细介绍。

5.2.6 图的遍历

所谓遍历（traverssal）即从某结点出发，按特定顺序依次访问其他结点。遍历方法有 2 种：

（1）广度优先遍历（width first traversal）

算法 5-5：图的广度优先遍历

① 从某顶点开始，将其标识为已访问过。

② 然后逐个访问该顶点的邻接顶点中尚未遍历过的顶点，直到所有邻接顶点都被标识为已访问过为止。

③ 然后对这些邻接结点逐个重复过程(2)，直到所有顶点都没有可访问的邻接顶点为止，这次遍历结束，生成一个广度优先遍历树。

④ 检查图中是否还有没访问过的顶点，若有，则从中任选一个结点再次开始步骤(1)～(3)，直到图中所有顶点都被访问过，遍历结束。

一般来说，遍历结果往往产生不止一个遍历树，形成的多个树称为广度优先生成森林（width-first spanning forest），而且从不同顶点开始遍历得到的结果也各不相同。

（2）深度优先遍历（depth-first traversal）

算法 5-6：图的深度优先遍历

① 从某结点 S_0 开始，将其标识为已访问过，并令访问深度 $h=0$。

② 选取该顶点的某个邻接结点 $S_{1,0}$，将其标识为已访问，令访问深度 $h=h+1$。

③ 从 $S_{h,0}$ 开始，选择它的某个邻接结点 $S_{h+1,0}$，重复过程(2)。

④ 若在某个访问深度 h 上，再无可访问的邻接结点，则回溯到上一访问深度 $h-1$，选择该层的下一个邻接结点 $S_{h-1,1}$，重复过程(2)和(3)。

⑤ 若层次 $h-1$ 上的所有邻接结点都按上述模式被访问过，则继续回溯到更上一级 $h-2$，重复过程(2)和(4)，直到回溯到 $h=0$ 的层上，所有邻接结点都被访问过，这次遍历结束，生成一个深度优先遍历树。

⑥ 继续检查图中是否还有没访问过的顶点，若有，则从中任选一个再次开始步骤(1)～(5)，直到图中所有顶点都被访问过以后，整个图的遍历结束。

与广度优先遍历一样，深度优先遍历结果也往往产生不止一个遍历树，形成深度优先生成森林（depth-first spanning forest），而且从不同顶点开始遍历得到的结果也各不相同。

例 5-13：

（1）对于图 5–17(a)，第一次从结点 D 开始进行广度遍历，第二次从结点 E 开始，得到广度优先遍历森林如图 5–17(b)所示。

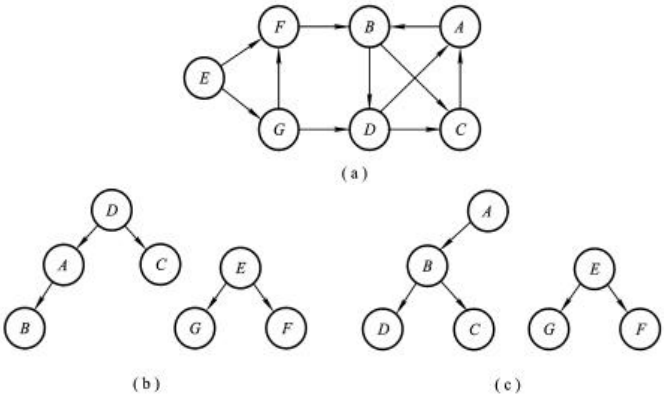


图 5–17 图的遍历

（2）第一次从结点 A 开始，第二次从结点 E 开始，得到的深度优先生成森林如图 5–17(c)所示。

5.2.7 树的遍历

树也是图，其遍历方法自然也分为广度优先和深度优先，但对于树，由于其结点的排列有规则性，其遍历结果产生惟一的结点序列，而不像图那样可能有多种结果。

(1) 广度优先遍历

自根部向下，依次访问完第一层的所有结点后，然后逐层访问各子树上同一层次的所有结点，依此类推，直至最大深度上的结点都访问完毕为止。

(2) 深度优先遍历

深度优先又分前序和后序。

(a) 前序遍历是先访问根结点、然后以递归方式依次访问各子树，对每个子树都一直访问到最底层的子结点后，再访问另一子树。

(b) 后序遍历则是先访问各子结点，最后是根结点。

例 5-14：对于图 5–18 所示的树，进行各种遍历。

(1) 由广度优先遍历得到的结点序列为：*A*-*B*-*C*-*D*-*E*-*F*-*G*-*H*-*I*-*J*-*K*-*M*-*N*；

(2) 由前序深度优先遍历得到的结点序列为：*A*-*B*-*E*-*F*-*C*-*G*-*M*-*N*-*D*-*H*-*I*-*J*-*K*。

5.2.8 二叉树的遍历

遍历二叉树结点的方法有 3 种：

(1) 前序（遍历）法（preorder traversal）

从根结点开始，先访问结点本身，然后前序遍历其左子树，最后前序遍历右子树。

(2) 中序（遍历）法（inorder traversal）

从根结点开始，先中序遍历其左子树，访问结点本身，最后中序遍历其右子树。

(3) 后序（遍历）法（postorder traversal）

从根结点开始，后序遍历其左子树，然后后序遍历其右子树，最后访问结点本身。

例 5-15：对图 5–19 中所示二叉树采用 3 种遍历法的结果如表 5-4 所示。

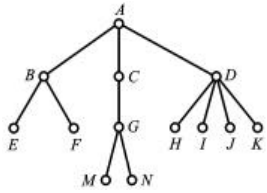


图 5–18 广义树的遍历

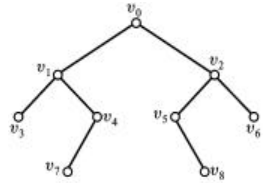


图 5–19 二叉树遍历示例

表 5-4 图 5-19 所示二叉树的 3 种遍历结果

遍历顺序	0	1	2	3	4	5	6	7	8
前序法	v_0	v_1	v_3	v_4	v_7	v_2	v_5	v_8	v_6
中序法	v_3	v_1	v_7	v_4	v_0	v_5	v_8	v_2	v_6
后序法	v_3	v_7	v_4	v_1	v_8	v_5	v_6	v_2	v_0

注意：由于树的定义是递归的，所以遍历方法也采用了递归形式。以前序法为例，其详细过程为：首先访问整个树的根结点 v_0，然后进入其左子树，访问该左子树的根结点 v_1，由于它下面还有左子树，所以继续往下访问 v_3；由于 v_3 没有左子树，则回溯到 v_1，转入其右子树，访问 v_4，然后访问其左子树 v_7，对 v_0 左子树的访问到此结束。然后一直回溯到根结点 v_0 处；进入其右子树 v_2，依次访问 v_5 和 v_8 后，进入 v_2 右子树访问 v_6，整个遍历结束。

读者可自己演绎一下中序法和后序法的详细步骤作为练习。

5.3 图论建模

图是一种描述数据对象间关系的数学工具。图的应用首先是与自然图形密切相关的。本章已经介绍了足够多的关于自然图形的例子，如欧拉回路（例 5-9）、货郎担问题（例 5-10）、克鲁斯克尔算法问题（例 5-11）、狄克斯特算法问题（例 5-12）等。但是，还有很多问题乍看起来与图形毫无关系，但仍需要用图论方法来解决。因此，本节着重讨论如何用图和树来描述事物间相互关系的问题。

5.3.1 用图表示网络关系

一般来说，当所描述事物之间有纵横交错的关系时，可用结点表示事物、用边表示事物之间的关系以及伴随这些关系的信息，从而形成一个描述事物之间网络关系的图，而问题的解决就相当于发现图中的路。

例 5-16：某地有 8 名患者感染了 S 病毒，当地卫生防疫部门将他们隔离，希望能就此阻止 S 病毒在当地继续传播。根据对这 8 名患者进行的流行病学调查，发现了他们彼此之间的接触史如表 5-5 所示，试问病毒最先是由谁开始传播的，是否所有被感染的患者都被隔离？

表 5-5 患者之间的接触史

被隔离患者	与被隔离患者有过接触者
A	*D, J*
B	*C, F, I*
C	*F*
D	*C*
F	*L*
I	*B, F*
J	*A, C, F*
L	*C*

解：该问题可以用有向图来描述：用结点表示患者，将患者之间的接触看成有向边。则上述关于患者接触关系的描述可用图 5–20 的有向图表示。

因为图中所有结点的入度均不为 0，也就是说他们之中没有人可能是最初的病毒感染源，在当地一定至少还有一名没有被发现的带毒者。

这种用顶点表示活动而用边表示活动之间的顺序关系的图，一般称之为 AOV（activity on vertices）图，其中一个重要应用是决定活动进行的顺序。

例 5-17：许多工程活动，如项目计划、生产流程、施工过程，都可用图表示。管理科学里的 CPM（critical path method）方法和 PERT（program evaluation and review technology）方法就是典型的例子。它们在由活动组成的网络中找出费用（时间或成本）最高的路径，称为关键路径。然后，通过缩短关键路径的总时间，降低其总费用，使整个项目的完成时间缩短，总成本降低（关于 CPM 和 PERT 的详细介绍，可参看管理学科的有关教材）。表 5-6 列出了某生产过程各工序之间的顺序关系以及它们所需的时间和成本，试求其关键路径。

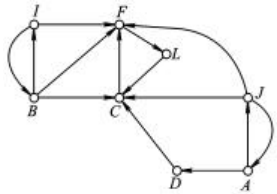


图 5–20 表示患者接触史的 AOV 图

表 5-6 工序之间的顺序关系

工 序	耗时(天)	成 本	上 道 工 序
A	6	100	–
B	4	115	*A*
C	5	95	–
D	7	87	*C*
E	4	64	*B, D*
F	8	75	*C*
G	14	228	–

解：由所给数据绘出 PERT 网络图，其步骤是：

① 从表 5-6 中找出没有任何上道工序的 3 个工序，*A*、*C*、*G*，将它们作为边，它们的起点就是 PERT 的起点。

② 找出以 *A* 为上道工序的工序 *B*，将 *A* 的终点作为边 *B* 的起点，画出边 *B* 来。

③ 用类似的方法，从边 *C* 的终点画出边 *D* 和 *F*。

④ 因为边 *E* 的上道工序为 *B* 和 *D*，所以边 *B* 和 *D* 的终点汇合为一个，成为边 *E* 的起点。

⑤ 边 *E*、*F*、*G* 都没有后续工序，它们的终点就是整个 PERT 的终点。

所得 PERT 图如图 5–21 所示，其中为每个结点编上了号码。

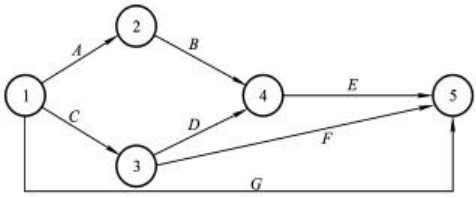


图 5–21 PERT 网络示例

要发现整个工程的关键路径，需一一求出从结点 1 至结点 5 的所有通路，结果如表 5-7 所示。由该表得知：其中耗时最多的时间关键路径为：*C*、*D*、*E*，共需时 16 天；成本最高的费用

关键路径为：*A*、*B*、*E*，总成本为 279。

表 5-7 图 5-21 中结点 1 至结点 5 各路径程度

路 径	时 间	成 本
A, B, E	6+4+4=14	100+115+64=279
C, D, E	5+7+4=16	95+87+64=246
C, F	5+8=13	95+75=170
G	14	228

这种用顶点表示状态、用边表示状态之间的活动以及伴随这些活动的费用、时间等，一般也称为 AOE（Activity on Edge）图。

5.3.2 用树表示分类的层次关系

集合是一种对事物进行分类的数学工具，树也是一种对事物进行分类的方法。两者的区别在于，由集合表达的分类之间只有包含与被包含（子集）的关系，由树表达的分类还可以表达事物之间的从属关系，或者表示一种共性与个性的关系。以下就是用树进行分类的一些例子：

例 5-18：如图 5–22 所示，微机 Window 系统中的文件夹组织是相当典型的树形结构，可使用户分门别类地存放各类文件。如在“我的文档”下创建一个文件夹“doc”专门存放 Word 程序生成的文档文件，创建另一个文件夹“graph”专门存放图形文件。在每个文件夹中还可进一步创建分类文件夹，如文件夹“report”专门存放请示报告，文件夹“notice”存放各类通知和告示。采用这种树形结构可将内容繁杂的文件有条理地组织起来，能在需要的时候迅速找到它们。设想如果将所有文件都放在一个文件夹、即一个简单的集合中，管理不可能方便有效。

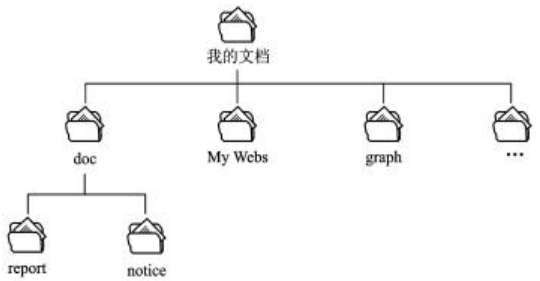


图 5–22 用树形结构组织文件夹

例 5-19：用户界面的作用是使用户可用交互方式执行程序的有关功能。为方便用户使用，用户界面最常见的形式就是将有关功能分类组织为一个层次结构，即通常所说的菜单树（menu tree）。图 5–23 就是 Word 程序的菜单树结构示意。在主菜单下有文件、编辑、视图等栏目，分别表示 Word 软件的主要功能。每个大栏目下有表示具体功能的小栏目，如在“文件”栏目下有新建、打开、关闭、保存、打印等具体功能；在“编辑”栏目项下，有“剪切”、“复制”、“粘贴”等具体功能。

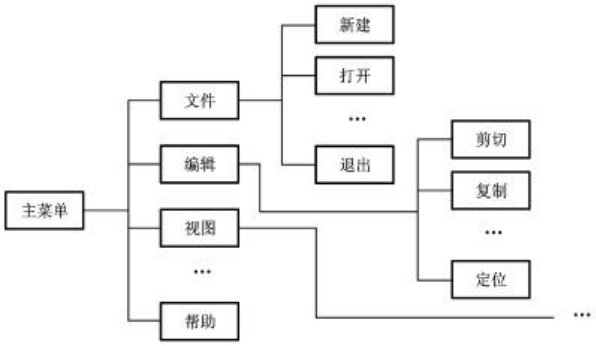


图 5–23 基于菜单树的用户界面

5.3.3 搜索树

解决复杂问题的常用方法之一是将问题分解，发展出若干初步解决方案，然后再对每个初步解决方案进行分解，发展出更详细的解决方案，余此类推，直到最后分解为可以直接解决的问题为止。如果把原问题表示为树的根，初步解决方案就相当于其下的子结点，而逐步发展出来的各级解决方案就是后裔子结点，最后的解决方案就相当于叶结点。这样，原问题的解决就可归结为在该树中搜索一条由根至叶结点的路径问题。

例 5-20：例 5–10 中对图 5–14 采用最近邻居法求出其哈密顿回路，但该法不能确保求出的解就是最佳解。如需求最佳解，必须一一列出从起点 P 出发、不重复经过各结点回到 P 点的所有可能路径，结果可用图 5–24 所示的搜索树（search tree）表示。由该图知：从顶点 P 出发，经过其他所有顶点后再回到原出发顶点 P，共有 24 条可能的回路，逐一求出这些路径的长度，其中最小者就是最佳路径。

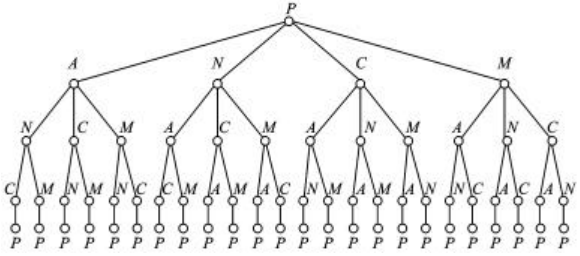


图 5–24 表示图 5-14 中所有可能回路的搜索树

像这样列出所有可能的解，然后从中搜索最佳解的方法，在软件技术中称为 brute force 算法，其字面意思是“用蛮力”，意思是全凭力气来求解。中文不妨称其为“穷举法”。从数学上来说，对于 n 个结点的图，可能路径的总数$= (n-1)!$。这里结点数 $n = 5$，所以共有 24 条可能的

路径。当图中包含结点数较多时，这种方法就非常不实际了。例如，当 $n = 11$ 时，可能的选择将有 10！=3628800 个，若用人工计算每条路径长度需时 1 分钟，则计算完所有可能路径长度大约需时 6.904 年。

图论的许多问题都类似这样，无法求得最佳解。因此，最近邻居法、克鲁斯克尔算法和狄克斯特算法等对于解决图的问题十分重要，这些算法虽然不能确保每次都可求得理论上的最佳解，但能确保有效地得到一个足够好的结果，这也是图论运算的一个特点。

需要特别注意的是，不要把搜索树与图的生成树混淆起来。生成树是图的子图，而搜索树表示所有可能的选择。为帮助学员进一步理解搜索树的概念，以及如何应用它解决实际问题，下面再举一个例子。

例 5-21：假定现需要设计一个计算机程序与人对弈常见的井字棋游戏的程序，如何才能计算出可取胜的棋局呢?

各方可能采取的布局可以用图 5–25 的搜索树表示。若黑方先下，则它可在 9 种可能的布局中任选一个，而应对黑方的每种布局，白方则可在 8 种可能的布局中任选一个，但此时要判断哪个布局更好一些，对双方来说形势都不很明朗。然后，当黑方布第二个子时，它可在 7 种可能的布局中选一个，此时布局的优劣已初现端倪，其中以布局(2)最好，它既为己方最后形成三子连势提供了可能，而且又使对方已经布下的子决无形成三子连势的可能；布局(3)虽然增加了己方取胜的可能，但并没有减少对方取胜的可能，比布局(2)稍逊一筹；而布局(7)不好也不坏；最差布局当属布局(1)。一般搜索树都用分值来定量评价各结点所表示选择的优劣，而包含分值最高的结点的路径就是易取胜的路径。设计下棋程序的一个重要任务就是要在搜索树中发现一条能最终导致取胜布局的路径。

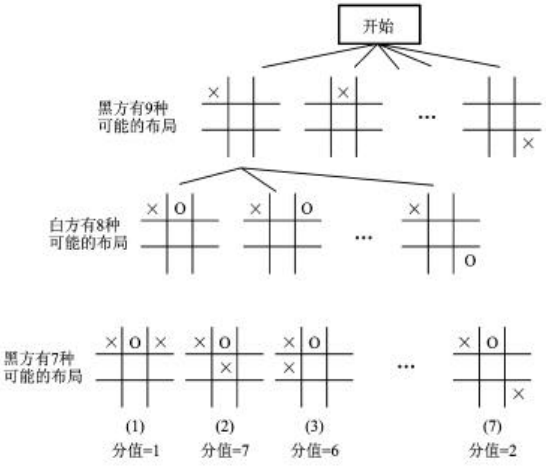


图 5–25　表示下棋双方可能的布局的搜索树（局部）

显然不可能在这里完全画出该搜索树，因为最终将有 9! = 362 880 个叶结点，即 362 880

条可能的路径，一般的计算机也不可能一次求出该搜索树所有的分支。因此，一般的下棋程序采取的方法是对某个布局只生成 2–3 层搜索树，然后从中选取一条相对较好的路径。

用搜索树计算取胜路径是设计所有游戏程序（包括各种棋类）的基本原理。当计算机有足够大的内存容量和足够快的计算速度时，计算机程序就足以能在游戏中战胜最好的人类选手。这方面最著名的例子当数 1997 年 5 月 IBM 公司的深蓝（Deep Blue）计算机战胜国际象棋大师卡斯帕罗夫（Garry Kasparov）的故事。卡斯帕罗夫号称人类有史以来最伟大的棋手，在国际象棋棋坛上他独步天下，曾连续十二年获国际象棋大师称号。“ Deep Blue* ”是美国 IBM 公司生产的并行超级计算机 IBM RS/6000 SP，重 1 270 kg，有 32 个微处理器，每秒钟可计算 2 亿步棋，存储有一百多年来优秀棋手的对局两百多万个。可以说已将所有的可能性“一网打尽”，难怪卡斯帕罗夫要输棋。

习 题

知识点：

图论的应用十分广泛，但就本课程学习来说，需要重点掌握的内容为：

① 图的图形表示。

② 用常用算法发现一些重要的通路。

③ 图论建模，尤其是非自然图形的建模。

1. 设 $V=\{u,v,w,x,y\}$

(1) $E=\{(u,v),(u,x),(v,w),(v,y),(x,y)\}$

(2) $E=\{(u.v),(v,w),(w,x),(w,y),(x,y)\}$

做出 $G=(V, E)$的图解并求各顶点的度数。

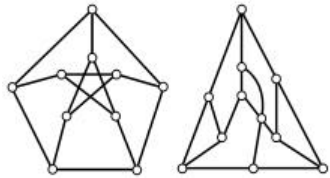
2. 说明题 2 图中的两个图是同构的。

思考：能否再想出一个与它们同构的图形？

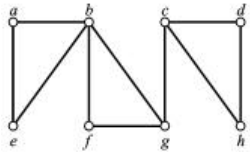
3. 求题 3 图中从 a 到 h 的所有通路，其中最短的是哪一条？

4. 求题 4 图中顺序经过 a、b、c、d，然后返回 a 的最短路径。

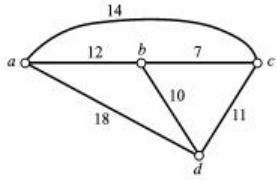
5. 列出题 5 图所示二叉树的结点的 3 种遍历结果。



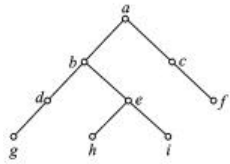
题 2 图



题 3 图



题 4 图



题 5 图

6. 有 6 个城市 $C1, C2, \ldots, C6$，每两个城市之间的直达航空旅行的费用如题 6 表所列，表中符号“–”表示该两个城市之间没有直达航班。

(1) 试求连接所有 6 地的最便宜的路径；

(2) 求从 $C1$ 到其他各地的最便宜的路径。

题 6 表　各城市之间的航空费用

费用 城市 / 城市	$C1$	$C2$	$C3$	$C4$	$C5$	$C6$
$C1$	0	50	–	40	25	10
$C2$	50	0	15	0	–	25
$C3$	–	15	0	10	20	–
$C4$	40	0	10	0	10	25
$C5$	25	–	20	10	0	55
$C6$	10	25	–	25	55	0

7. 在开发办公自动化系统时，要根据组织内部的业务流程来决定信息流动方向。所谓业务流程就是公文批转程序。例如，某些由下而上呈报的文件需要先由部门经理批准，然后转呈上级主管批准；而另一些自上而下颁发的文件，则应由主管部门经理批准后转职能部门执行。题 7 表是已知某公司内的文件批转程序，现假设有一份文件得到了所有 5 位主管的批准，问它是按什么顺序流转的？

题 7 表　某公司内部公文流转顺序

主 管 人 员	文件转发方向
公司经理（company president, CP）	MD
营销主任（marketing director, MD）	-
销售经理（sales manager, SM）	MD, PM
生产主管（production manager, PM）	CP
财务主管（financial officer, FO）	SM, MD, PM

8. 根据教学计划，学员必须按照题 8 表中给定的顺序选课，才能保证学习任一门课程时具备了必须的背景知识。试问哪些课是另一些课的基础？

题 8 表　选课顺序关系

课 程 代 号	课 程 名 称	先 修 课 程
$C1$	高等数学	
$C2$	离散数学	
$C3$	程序设计语言	$C1$, $C2$
$C4$	数据结构	$C2$, $C3$
$C5$	形式语言与自动机	$C2$
$C6$	编译原理与方法	$C4$, $C5$
$C7$	操作系统	$C8$, $C4$
$C8$	计算机原理	$C9$
$C9$	汇编语言	$C1$

9. 一个有名的智力游戏题为：一个人要携带一只狗、一只羊和一袋白菜过河。由于船很小，每次除人外只能再带一件物品，但不能将狗和羊单独留在一起，也不能将羊和白菜单独留在一起。试求一个能携带这 3 样物品安全过河的步骤。

提示：

一般人会凭感觉来解决这个问题，但是只有应用数学理论解决问题，方法才是系统的，答案才是严密的。

第 6 章　C++编程作业入门

知识点：

本章学习目的包括：

① 掌握 VC++编译器的使用方法。

② 熟悉 C++语言的基本词法和句法（除类外），能够熟练编写简单程序。

③ 尽管编程理论、方法和工具已有很大发展，编程作业在很大程度上仍然是一种工匠性（state of art）的活动，因此应投入足够的时间进行艰苦的练习。

所有编程语言的学习都包括两方面的内容：学习该编程语言本身，包括词法和句法；学习某种语言编译器产品的使用方法。学习一种语言，无论是自然语言还是计算机语言，最好的学习方法都是通过实际应用去掌握它。因此，在介绍 C++的语法以前，教材将先介绍用 Windows 操作系统下的 Microsoft 公司 Visual C++（简称 VC++）进行编程作业的基本方法，学员在学习语法概念的同时就可将它们编成简短的程序，通过实际运行程序来理解和掌握有关的语法概念，而不是死记硬背语法条文。

另外，正如本教材在开始所指出的，编程作业的规范化是软件技术学习的重要内容。因此从开始学习编程就应注意培养良好的编程习惯。

6.1　程序编写作业概述

6.1.1　编程语言

语言（language），包括自然语言和编程语言，都定义为字符串的集合，但常用的列举法和描述法这两种描述集合的方法（参见 4.1.2）都不适合于语言的描述。首先，几乎所有的语言都有数量无限的字符串（即句子），其次，这两种方法无法描述所有句子的性质。对语言来说，最关注的问题还是：

① 给定语言的描述，怎样自动生成句子。

② 给定语言的描述，确定一个字符串是否是语言（即合法的句子）。

因此，应有一种描述方法能解决上述问题，这就是语法（grammar）。语法是一种较自然的描述语言的方法，一般包括两个方面：词法（lexicon）是关于词汇性质的规定，句法（syntax）是关于程序语句结构的规定，只有由合法的词汇按合法的句法构成的句子才能被程序识别和执行。语言依其语法的类型被分为以下几种：

- Type–1 语言是上下文有关（context-sensitive）语言，即一个句子成分的展开与其上下文的内容有关，如自然语言。
- Type–2 语言是上下文无关（context-free）语言，其句子成分的展开只考虑其形式，而

与内容无关。

- Type-3 语言又称为正则语言(formal language)。大多数非人工智能编程语言都用 type-3 语法描述其词法，用 type-2 语法描述其句法。其语法规则要比任何一种自然语言简单得多，词汇量非常有限，程序语句只能由语言本身专用的关键词（keyword）和编程员定义的标识符（identifier）按规定的句型组成，而规定句型的种类也十分有限，因此学起来并不困难。但是，编程语言也有其特点，其一，必须严格按规定格式书写，才能被计算机识别和处理；其二，必须按严密的数理逻辑组织其语句，程序才能正常运行并产生预期的计算结果。只要把握这两个特点，学习编程就容易多了。

编程语言也是语言，其学习与汉语、英语等自然语言的学习存在共同性。最重要的一点就是，不要等到掌握了全部语法以后才“开口说话”。只要掌握了基本词汇和基本句型，就可以开始造句，并通过造句来加深对语法概念的理解。所以，建议一边阅读教材，一边在计算机上实际编写程序，通过观察程序语句的实际运行情况来理解语法概念，不实际动手练习是绝对学不好编程的。

根据用途，编程语言分为用于数值计算和数据处理语言，如 FORTRAN、BASIC、C、C++、Pascal 等；用于人工智能的语言，如 LISP 和 PROLOG 等；还有许多其他的专用编程语言等。本教材将介绍采用 C++语言编写软件程序的方法和步骤。C++是一种面向对象的编程语言（关于什么是面向对象编程的问题将在后续章节中予以详细介绍），由美国 AT&T 公司的 B. Stroustrup 在 20 世纪 80 年代开发，现已经成为应用最广的编程语言。

由于篇幅所限，本教材不能介绍关于 C++的全部概念，但只要掌握以上两个特点，从学习语法和应用数理逻辑入手学习，再通过大量的编程实践，掌握编程技术应该说入门不难，深造也不难办到。

6.1.2 编译器与编译作业流程

要使用 C++语言编写的程序作业能够最终在计算机上运行，在计算机上必须有相应的 C++编译器或者说编译程序（compiler）。

图 6-1 所示，编程作业的一般步骤为：

① 用文本编辑程序编写一个程序文件（也称为源代码）。一般 C++编译器都有集成的源代码文件编辑功能，也可利用 Windows 系统提供的记事本（notepad）程序进行编辑。C++程序一般都采用模块化的结构，将源代码根据功能分成多个文件。

② 由编译器检查源代码有无语法错误，若有则通知编程员进行必要的修改。经修改无误后，编译器将源代码变成机器内部的二进制文件(称为目标代码)，这步工作叫做编译(compile)。

③ 当组成一个程序的所有源代码文件模块都分别通过编译、生成目标代码以后，编译器就可将它们组合为一个可运行的程序（称为可执行代码），这步工作叫做连编（link）。

④ 运行可执行代码，并继续检查错误并修改程序，直到达到预期的设计目的。

现有许多厂家提供不同操作平台上的 C++编译器。在微机 Windows 操作系统下运行的 C++编译器有 Microsoft 公司的 Visual C++(简称 VC++)编译器，还有 Turbo C++编译器，Borland C++编译器等，另外还有在 UNIX 操作系统和 Linux 操作系统下运行的 C++编译器。因此，学习 C++语言编程时，除了学习语言本身外，还要学习某个编译器产品的具体使用方法。一般来说，伴

随每个 C++语言编译器产品，一般有两本资料：

- Programming References（编程手册）
- User's Guide （用户手册）

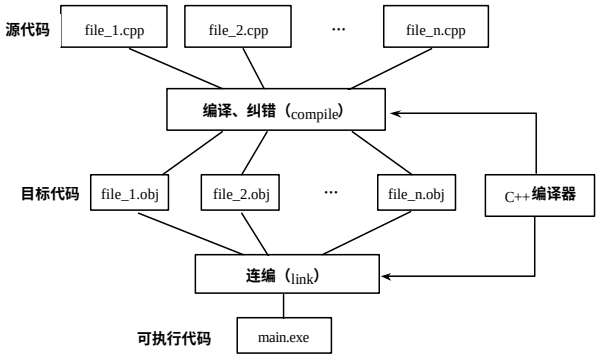


图 6-1 程序编写作业流程

前者介绍 C++语法规则，后者是使用该产品的具体方法，包括如何创建项目、如何编辑、如何编译、如何调试等。在学习编程时必须仔细阅读这两本手册。

本教材将主要介绍使用 Microsoft 公司 Visual C++（简称 VC++）编译器进行编程作业的具体方法和步骤，但考虑到教材应具有普遍性，教材避免在程序中使用 VC++特有的一些内容，如 Windows 下用户界面的编程。所有的范例程序都在命令行环境（即通常所说的 DOS 界面）中运行。因此它们可用任何其他 C++编译器编写和运行，方便了学员的学习。

由于篇幅关系，本教材只能介绍 C++的主要内容，但所包括的内容可满足编写一般应用软件的需要。学员如有兴趣深入学习 VC++编程，特别是 Windows 环境下用户界面的开发方法，可另行参看 VC++的随机文档光盘或者专门参考书。

6.2 用 VC++编译器进行编程作业

导读：

在书后配盘的对应章节中有下述内容的动画演示。

本节介绍用 VC++编译器编写、运行 C++程序的作业过程。

6.2.1 建立 VC++项目

上一节已提到，从编写程序到运行过程，要在多个文件模块上分别执行多个操作步骤，因此，一般的编辑器都用一个 Project（项目）将所有文件组织到一个集成环境中，以便统一进行管理。下面是建立 VC++项目的步骤：

① 启动 VC++编译器：点击 Windows 屏幕左下角的【开始】按钮，从【程序】项中可以

看到【Microsoft Visual C++】的选项，点击它就可使 VC++编译器开始运行。

② 当 VC++编译器启动以后，可看到图 6-2 所示的 VC++编译器环境，其中分为 3 个区域。左面的是文档管理区，目前暂时是空白的，在项目创建好以后，该窗口中会列出项目中包括的各种文件。右面是文件编辑区，用于编辑源程序。屏幕下方是系统提示区，用于显示工作信息，包括编译源程序时发现的错误、连编过程中发现的错误等。

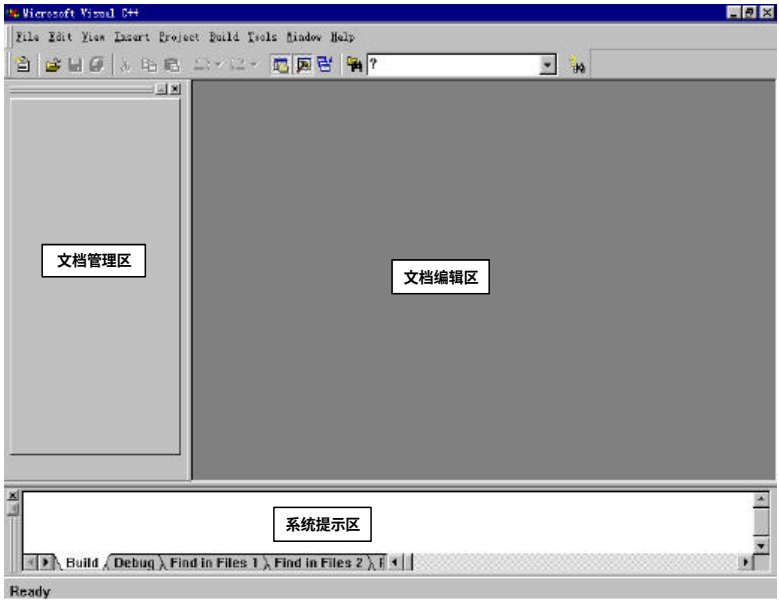


图 6-2 VC++编译器环境

③ 点击【File】菜单项，打开下拉式菜单，选择【New】，屏幕出现图 6-3 所示对话框。点击其中【Projects】标签，在窗口中选择条目“Win32 Console Application”，然后在 Location 窗口中选择一个存放源代码的文件夹，在 Project Name 中输入欲建立的项目名称，例如，project_1；最后点击【OK】按钮。

④ 接下来，屏幕出现图 6-4 所示的 Win32 Console Application 对话框，选择条目“An empty project”，然后点击【Finish】按钮。系统显示图 6-5 所示的 project 创建成功的信息，点击【OK】按钮继续。

说明：

为便于学习，本教材仅介绍 Console Application 类型的程序，它采用命令行窗口（即一般所说的 DOS 窗口）作为数据输入、输出界面。若采用 Window 式的对话框作为输入、输出界面，程序会漂亮得多，但编程也会复杂得多，而且还需用到 VC++特有的一些 Window 函数（非标准的 C++语言），在学习阶段不必在这方面花太多时间。

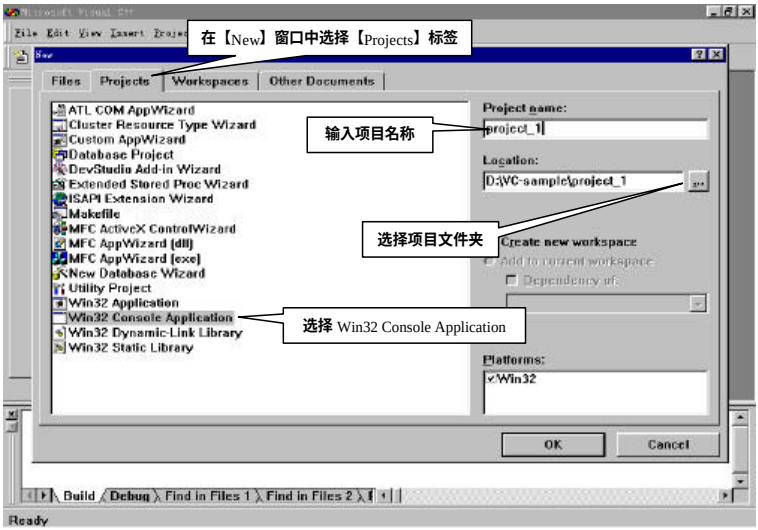


图 6-3 创建“Win32 Console Application

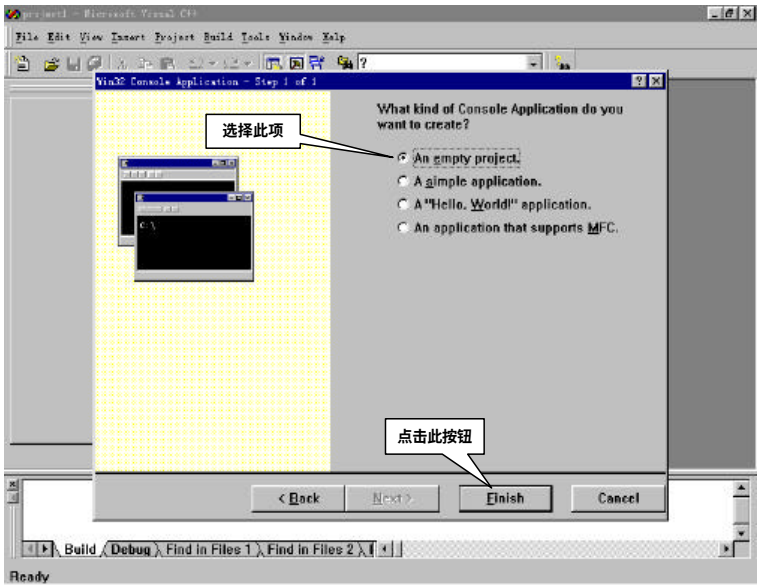


图 6-4 选择“An empty project”

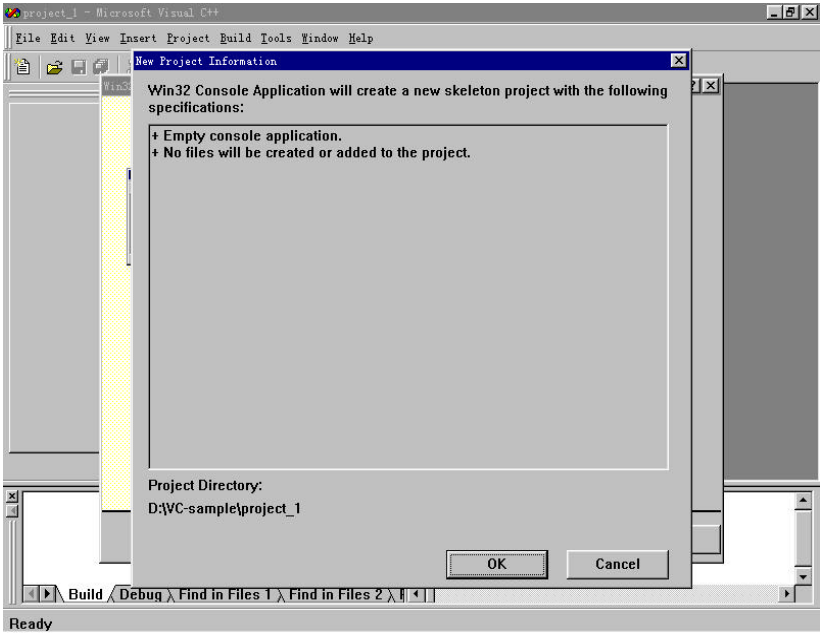


图 6-5 项目创建成功信息

6.2.2 编辑源代码文件

项目建立以后，便可编辑源代码文件（或称源程序），即用编程语言书写的文本文件。具体操作如下：

① 点击窗口左上角的【File】项，打开下拉式菜单，选择【New】子项，屏幕上出现图 6–3 所示的对话框，但这次应点击【File】标签，然后在下方窗口中选择条目“ C++ Source File ”(图 6-6)，然后在 File 窗口中输入欲编写的程序名，如 sample_1（不必加扩展名，系统会自动加上相应的扩展名 cpp)，同时在窗口右上角【Add to project】的小方框中打勾，最后点击【OK】按钮。屏幕会在右面编辑区中显示该程序文件的内容，如图 6-7 中所示，开始时文件是空白的。学员可将例 6-1 所示的程序输入其中。

例 6-1：一个简单的 C++程序。

```
// sample_1.cpp   using VC++ compiler
#include <stdio.h>
void main()
{
    int  a, b, c;
    double  x , y, z;
```

```
    a = 2;
    b = 3;
    c = a + b;
    x = 3.45;
    y = 9.71;
    z = x*y;
    printf("Sample_1.cpp\n");
    printf("a=%d b=%d a+b=%d\n", a, b, c);
    printf("x=%f y=%f x*y=%f\n", x, y, z);
}
```

② 输入完毕后关闭文件的方法是，点击【File】菜单项，打开下拉式菜单，选择【Close】。若文件经过修改，则屏幕会出现一个对话框，询问是否要保存该文件。编程员可根据情况点击按钮【YES】或者【NO】。如果回答 YES，文件会保留所作的修改，形成新版本；如果回答 NO，文件则恢复到未经修改的老版本。

若需接着进行编译、连编和运行等操作，就不必关闭文件，只要点击【File】菜单项、打开下拉式菜单、选择【Save】或点击菜单条上表示保存文件的磁盘快捷按钮即可。最后在关闭项目时，再一起关闭所有打开的文件。

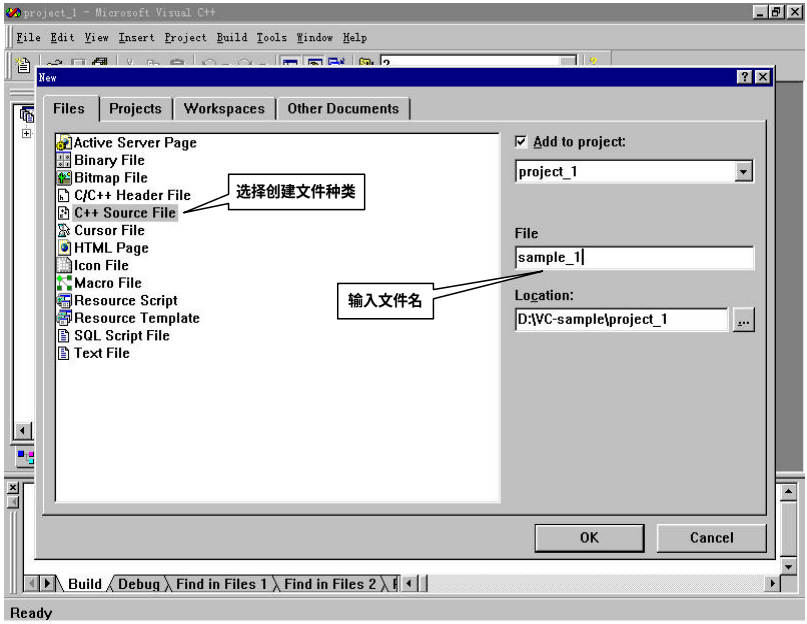


图 6-6 创建源程序文件

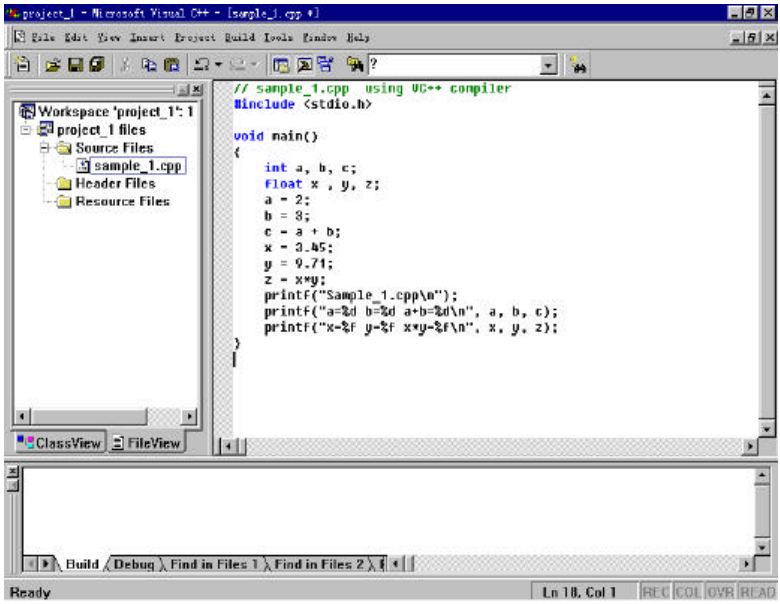


图 6-7 输入源程序文件

在连编时，VC++编译器会自动将同一个项目中的文件模块组合为一个程序，所以不要将程序文件存放在项目以外的文件夹中，而应用项目集中处理属于同一个程序的各个模块文件。

6.2.3 编译和查错

编辑完毕，点击菜单上的【Build】选项，打开下拉式菜单，选择第一个条目 Compile sample_1.cpp，程序就开始检查程序的语法，并将诊断信息显示在屏幕下方的系统提示区中，如图 6-8 所示。若用鼠标条点击某条诊断信息，VC++编译器便会自动打开对应的源程序文件，并用箭头指出错误语句所在位置。

为了体会编译器如何报告语法错误以及如何进行纠错作业，学员可在输入程序时人为地制造一些错误。例如，图 6-8 所示即为语句 c = a + b 后面少输入了符号“ ;”所引起的错误。

根据编译器提供的诊断信息，准确判断程序错误的性质和位置是学习编程最困难的环节之一，因为程序提供的信息往往不能切中要害。例如图 6–8 中系统显示错误的箭头指在 x = 3.45 这一行上，诊断信息是 missing ';' before identifier 'x'，但实际上错误是由上一条语句 c = a+b 后面缺少符号' ;'所引起的（符号' ;'是表示语句结束的标点符号，详见 6.3.2 节）。因此当有多条诊断信息时，可先修改前几条错误，然后尝试重新编译。随着这些错误得到纠正，许多误报的信息也许就自动消失了。

改正以后重新编译，若没有错误，则编译器显示目标程序 sample_1.obj 已生成的信息，如图 6-9 所示。

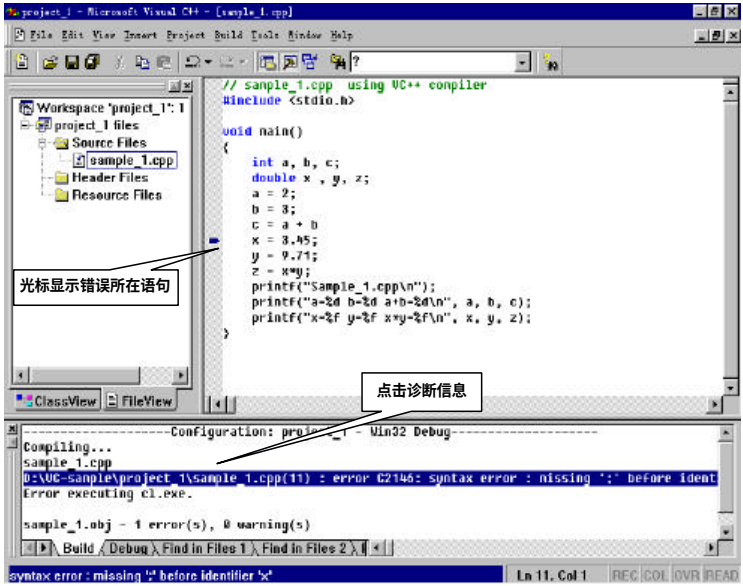


图 6-8 编译器指出源代码错误

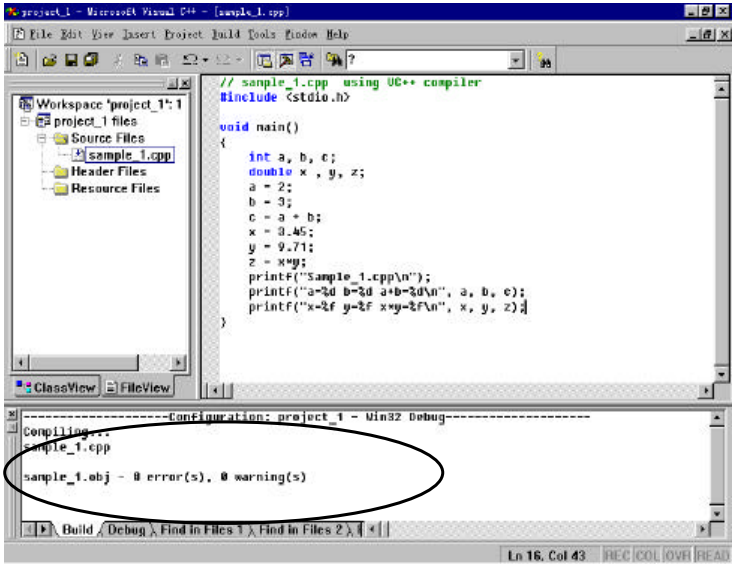


图 6-9 编译器显示成功生成目标代码的信息

6.2.4 连编和运行程序

当所有源代码文件编译纠错完毕并生成目标代码以后，就可将它们连编为一个可执行代码文件。具体步骤是点击菜单上的【Build】选项，打开下拉式菜单，点击选项【Build sample_1.exe】，VC++编译器就开始将目标文件和系统库函数连编为一个完整的执行程序，若无错误，VC++编译器显示成功生成执行代码文件 sample_1.exe 的信息，如图 6-10 所示。

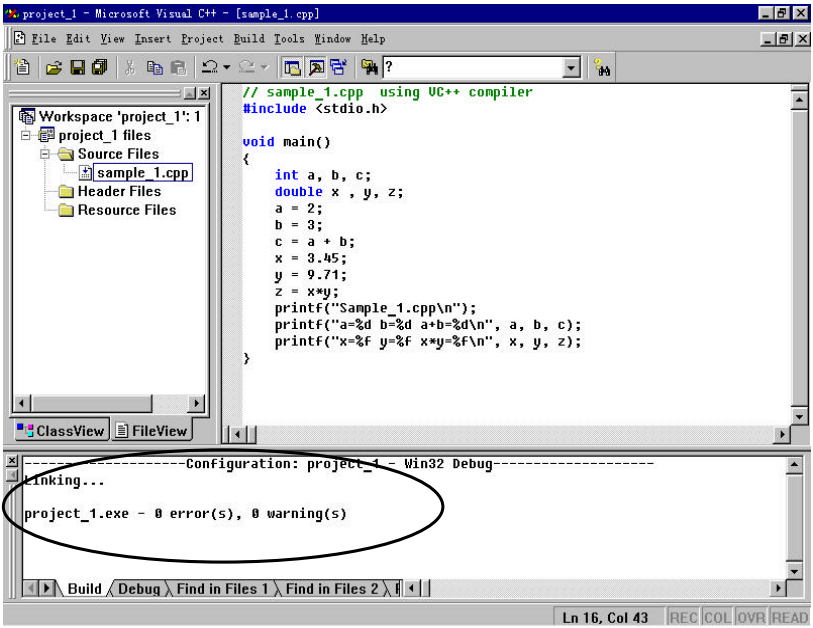


图 6-10 编译器显示成功生成可执行代码信息

点击菜单上的【Build】选项，打开下拉式菜单，点击选项【Eexcute sample_1.exe】，程序即开始运行，屏幕上会弹出一个命令行窗口，将程序计算结果输出（如图 6-11 所示）。运行完毕，按屏幕指示“Press any key to continue”按任意一键后，命令行窗口关闭，系统回到 VC++编辑器环境中。

可执行代码可脱离编辑器的环境单独运行，具体方法是在 Windows 桌面上，打开“我的电脑”，找到可执行代码所在的文件夹，点击文件 sample_1.exe，程序就可运行。可执行代码还可拷贝到其他微机上运行。

能够运行只是测试程序的第一步，它仅说明程序的语法没有错误，要使程序产生预期的结果，还需要用数据检测程序内部计算过程是否正确，发现和纠正程序设计中的逻辑错误和数据错误。关于如何测试程序的问题，后续章节中还将予以详细讨论。

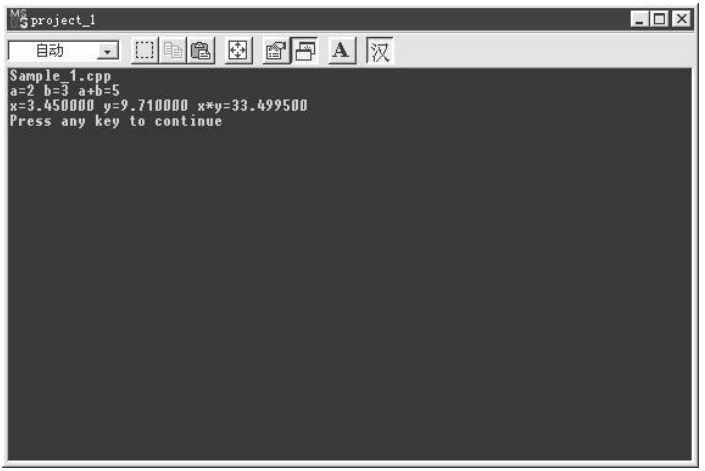


图 6-11 程序在命令行窗口中运行

6.2.5 项目的关闭和再打开

关闭项目的方法是，点击【File】菜单项，打开下拉式菜单，选择【Close Workspace】，屏幕会出现一个对话框，询问是否要关掉所有窗口，回答 Yes 后，编译器先关闭所有打开的文件，最后关闭项目窗口，退回到 Windows 的桌面上。

在关闭文件过程中，若有文件曾被修改过但没有保存，VC++编译器会用对话框逐个提示是否要保存这些文件。编程员可根据情况点击按钮【Yes】或【No】。

再次打开项目的方法是，点击【File】菜单项，打开下拉式菜单，选择【Open Workspace】，屏幕会出现一个用于打开文件的对话框，已建立的项目会有一个与之同名的文件夹（图 6-12），进入该文件夹，可看见一个与项目同名的、后缀为 dsw 的文件 project_1.dsw（见图 6-13），点击该文件，就可进入项目中。

点击窗口左面文档管理区中 Source Files 文件夹前的加号，VC++编译器会在文件夹图符的下方列出该文件夹中的文件。如点击 sample_1.cpp，该文件便在右面的编辑区中打开以供编辑。

6.2.6 向项目中添加文件和从项目中删除文件

（1）添加文件

当把一个程序文件，如教材提供的范例文件从其他地方拷贝到项目文件夹中后，它们并不能自动成为项目的组成部分。因此，不会出现在 VC++编译器的文件管理窗口中，也不会被连编到程序中。若要把这些从其他文件夹拷贝来的文件加入项目中，可以用鼠标右键点击编译器左方文件管理窗口中的 Source Files 或者 Head Files 的文件夹，并在打开的右键菜单中选择“Add Files to Folder”（图 6-14），然后在随后打开的文件选择窗口中选择 cpp 文件或 h 文件，即可将其加入到项目中。

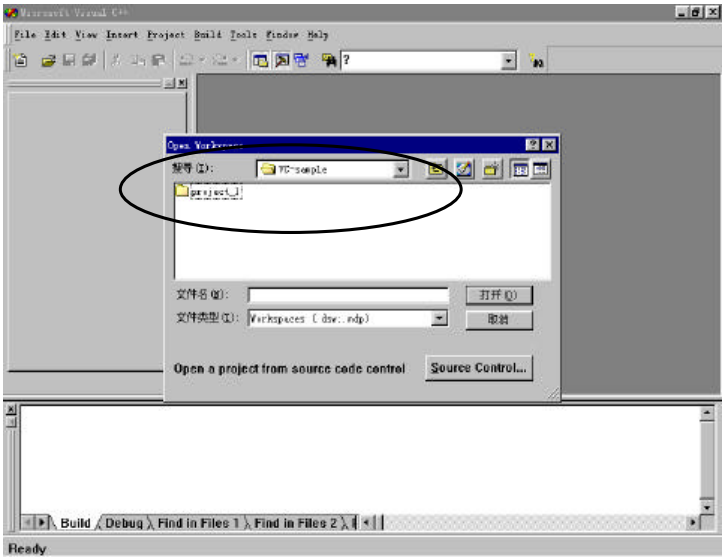


图 6-12 选择项目文件夹

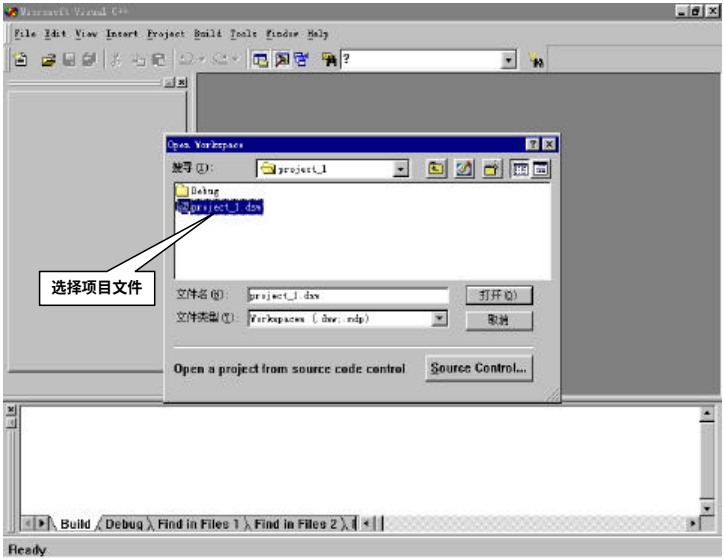


图 6-13 打开项目文件

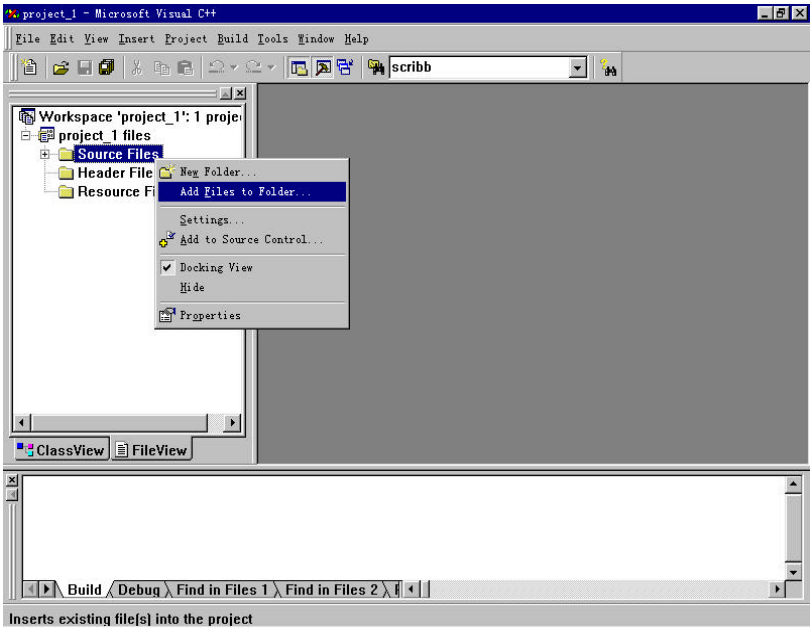


图 6-14 向项目中添加文件

(2) 删除文件

用鼠标点击文件名，使其被选中，然后按键盘上【delete】键，便可把文件从项目中删除，但是文件并没有在物理上被删除，仍在该项目文件夹中，只是不再能从 VC++编译器中打开，也不会在连编时被包括进去。若欲将这些文件永久删除，可用 Windows 的资源管理器的删除文件的功能。

6.3 C++语言词法概要

导读：

① 为便于学员将来能读懂 C++的语法规则，教材中在介绍 C++语法概念与术语时将同时标出其原文。

② 由于篇幅关系，教材在这里仅介绍 C++语法的主要内容。关于完整的 C++语法规则，可参看光盘中的语法附录或者 VC++的编程手册（Programming Guide）。

编程语言与自然语言都是语言，学习上都是有共性的。学员们对学习外语已经有了相当多的经验，因此教材的编排充分考虑了如何利用学员学英语的经验来帮助学习 C++语言。具体来

说，教材将首先在 6.3 节中集中讲解 C++语言的词法，它好比学英语要从了解名词、形容词、动词的性质开始，然后，将在 6.4 节讲解 C++的句法，即各种程序语句的编写方法，它好比学英语要学主动句、被动句等句型一样。希望学员能利用这些基本的学习规律来对编程语言进行理解和掌握。

6.3.1 基本词汇

在任何语言中，词汇都是构成语句的基本元素。C++程序的词汇称为基本词汇（Token），是程序语句的最小组成单位，相当于一般语法里所说的单词。下面是关于 Token 使用的一些约定：

① 在程序中，Token 之间是用“空格”分隔开来的，以便于编译器逐个识别它们。这里所说的空格是广义的。对 C++程序来说，以下字符均被认为是空格：

- 键盘上的空格键
- 键盘上的 Tab 键
- 键盘上的 Enter 键
- 打印时另起一行的控制符（line feed）
- 打印时另起一页的控制符（form feed）
- 程序注释语句（comments）

② 重复的空格都只作为一个空格处理。以下语句对程序来说都是等价的：

```
int i;   （句首无空格）
    int i;（句首有若干空格）
int     i;（句中有若干空格）
```

③ Token 是大小写有关的，所以 int 和 Int 被认为是不同的词汇。

Token 共有 6 类，下面分别介绍之。

6.3.2 标点符号

标点符号（punctuator）的种类和在程序句子中的位置对于程序语句的解读和执行来说至关重要。表 6-1 列出了 C++程序中常用的一些标点符号。

表 6-1 C++程序中的标点符号

!	%	^	&	*	(	)	–	+	=	{	}	|	~
[	]	\	;	'	:	"	<	>	?	,	.	/	#

其中符号“;”是标明句子结束的重要符号。它使 C++中的长语句可跨行书写或将若干短语句写成一行。如下面例句所示：

```
int x, y;
x=5;    y=x+6;
```

其他符号如 !、%、^、&、*、–、+、=、|、~、/、(、)、[、] 等主要是表示运算，但在语法上把它们定义为标点符号，后续章节中会陆续介绍它们的作用。

各种括号[ ]、( )、{ }必须成对出现，但是可以嵌套，即括号中还有括号。

6.3.3 关键词

关键词（keyword）是编程语言本身专用的词汇。系统根据这些词汇在程序语句中的位置来识别程序语句的结构，所以编程员不能使用与这些词汇相同的名字来命名变量、常数和函数等。表 6-2 列出了后续章节中应用较多的一些 C++关键词（但并不是 C++关键词的全部）。

表 6-2 C++程序中的关键词

break	case	char	class
const	continue	delete	do
double	else	enum	extern
float	for	friend	int
long	new	operator	private
protected	public	return	short
signed	static	switch	template
this	typedef	unsigned	virtual
void	while		

6.3.4 标识符

标识符（identifier）是以字母开头、包括字母、数字和下划线在内的字符串。它在程序中用来作为：

- 变量名。
- 数据类型名。
- 函数名。
- 类、类的数据成员、类的成员函数名。

关于标识符的使用，有以下约定：

① 标识符由数字字符集合和非数字集合中的字符组成，但必须以非数字字符开头。

非数字字符集合 = a b c d e f g h i j k l m n o p q r s t u v w x y z _
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

数字字符集合 = 0 1 2 3 4 5 6 7 8 9

以下为合法标识符的例子：

a12_3　　BcD_456　　_variable

以下为不合法标识符的例子：

5abc　　（以数字开头）

xy$12　　（$不在合法字符集里）

② 标识符的最大允许长度一般为 32 个字符，但这在不同产品中是不同的，具体数值列于编译器的编程手册。

③ 标识符在其有效范围内不能重名，不论它们是分别作为类型不同的变量名还是函数名。这里所说的有效范围指同一个函数，或同一个程序。即，同一个函数体内不能有重名的变量，

在同一个程序里不同模块之间不能有重名的变量和函数。

规范化编程实践–1：给予标识符有表现力的名称。

软件技术中需要遵守的一个好习惯是给标识符富有说明力的名字，以增加程序的可读性。下面就是关于命名的常用约定：

- 用名词作为变量和数据类型名称。
- 变量名首字母小写。
- 数据类型名首字母大写。
- 函数名一般为动词加上名词的短语，首字母大写。
- 长名可由若干英语词汇连接而成，并将每个词汇的首字母大写以区分词的起始。（注意：过去曾流行过用下划线来区分词，但在 C++里已不再采用这种形式）
- 不要只用几个字符，或除自己外谁也不懂的缩写词汇，最好也不要用汉语拼音。如 studentAge、className 等是形式较好的名称，而 sa、cName、banji 则是表现力差的名称。

6.3.5 常数

常数（literal）指程序中不变的成分，即通常说的常数。C++语言提供的基本数据类型只有 4 种：

- 整数
- 浮点数
- 字符
- 字符串

其中，整数和浮点数根据在计算机内部表示时所占用的字节数的不同，分别细分出更多类型。

（1）整数常数

即不带小数点的数。除十进制数外，还包括八进制、十六进制数等常数。定义一些常用整数常数的关键词及其取值范围如表 6-3 所示。

表 6-3 常用的整数常数类型

关 键 词	类 型	取 值 范 围
char	字符	–128~127
int	整数	–32 768~32 767
long	长整数	–2 147 483 648~2 147 483 647

以下是整数常数定义的例句：

```
int x =-43;
int b = 0147;  （八进制常数必须总是以 0 开头，每位数字的值为 0~7）
int c = 0xAb9;  （十六进制常数必须以 0x 开头，每位数字的值为 0~F，字符可大写，也可小写）
long d = 483648;
```

常数的大小实际上与硬件平台 CPU 的位数有关。现在一些 32 位 CPU 的计算机上，int 的

字长有 4B，而 long 整数则长达 8B，其所表示的数值范围也相应扩大。

（2）浮点数常数

即通常意义上带小数点的数。定义一些常用浮点数常数的关键词及其取值范围如表 6-4 所示。

表 6-4 常用的浮点数常数类型

关 键 词	类 型	有 效 数 字
float	浮点数	3.4E±38 (7 位有效数字)
double	双精度浮点数	1.7E±308 (15 位有效数字)
long double	长浮点数	1.2E±4 932 (19 位有效数字)

以下是定义浮点数常数的一些例句：

```
float x =1.03;
double y = -76.95;
```

浮点数还可以表示为指数形式，如以下例句所示：

```
float u = 18.46e-2;          (=0.1846)
double  z = 18.46E1 ;        (=184.6)
```

其中，e 或者 E（大小写无关）为 10 的指数，如 18.46e-2 相当于 $18.46\times10^{-2}=0.184\ 6$。

不同的 C++编译器因其浮点数内部表示方法不同，其取值范围也不同。一般可从系统头文件 float.h 中查到各种类型数据所能表达的最大值。

（3）字符常数

包括单个的可见字符或用于控制打印格式等的不可见字符。定义字符常数的关键词及其取值范围如表 6-5 所示。

表 6-5 字符常数类型

关 键 词	类 型	取 值 范 围
char	字符	–128~127
unsigned char	无符号字符	0~255

字符常数的值用成对的单引号把字符括起来表示，或用其 ASCII 码值表示，以下是定义字符常数的例句：

```
char ch = 'x';
unsigned char b = 48;        (48 是字符 0 的 ASCII 码值)
unsigned char d = 0x07;      (16 进制表示的 ASCII 码值)
```

标点符号中的单引号和双引号和一些不可见的打印格式控制符虽然也是字符常数，但要用特别方法表示，即用打印格式控制符（escape sequence）表示（一种用反斜杠符号加上其值的表示方法）。表 6-6 列出了表示的一些打印格式控制符。

（4）字符串常数

即多个字符的组合。用数组（array）或指针（pointer）变量定义（关于数组和指针的概念详见 6.4.8），其值则用双引号表示。定义字符串常数的语句格式如下述例句所示：

表 6-6 常用的打印格式控制符的表示方法

打印格式控制符	ASCII 码值	表示的字符	作 用
\'	0x27	'	单引号
\"	0x22	"	双引号
\a	0x07	BEL	铃声
\b	0x08	BS	Backspace 键
\f	0x0C	FF	form feed 另起一页
\n	0x0A	LF	line feed 另起一行
\r	0x0D	CR	Enter 键
\\	0x5C	\	反斜杠

```
char a[4]= "AbE";
char B[  ]= "This is a string";
char*    aString = "1234";
```

字符串常数所占据的字节数量等于其可见字符数+1，这是因为字符串常数后面需要有一个不可见的字符串终结符'\0'。以变量 a 为例，它有 3 个可见字符，但实际占据 4 B。字符串常量 B 虽然只包括 16 个字符，其实际长度为 17。

在定义字符串常数时不必写出结尾符，系统会自动加进去，但在计算字符串长度时该考虑到这不可见词尾，否则程序运行时会引起不可预见的错误。

6.3.6 运算符

运算符（operator）将运算数连接起来，指定其计算的顺序，形成运算式（expression）。C++中的 operator 有 2 大类：

（1）一元运算符（unary operator）

表 6-7 是 C++语言中常用的一些一元运算符。

表 6-7 C++语言中常用的一元运算符

一元运算符	说 明
-	取负运算符
!	逻辑非运算（logical NOT）
++	递增运算符（incremental operator）
--	递减运算符（decremental operator）
*	取值运算符（pointer-dereference operator）
&	取地址运算符（address-of operator）
new	动态分配给一个或多个变量内存空间
delete	收回分配给一个或多个变量的内存空间

一元运算符只涉及一个运算对象，其中有的必须位于运算对象之前，如表示负数的–号、表示逻辑非运算的！号；另一些则既可在前面，也可在后面，如++和--。下面分别介绍它们的意义：

- 运算符“ !”表示逻辑非运算，它和表 6-8 中的二元逻辑运算符&&和||一起，正好对应于 3 种基本的命题逻辑运算：非运算、合取运算和析取运算。它们主要用在程序控制语句中（见 6.4.7 和 6.4.8）。
- 运算符++和--是加 1 和减 1 的简写形式，它们既可出现在变量名之前，也可出现在变量名之后。例如：

a++；相当于 a = a+1；

--b； 相当于 b = b-1；

- 这里的运算符*不是乘号，而是“ 取值运算符 ”，它出现在单个变量名前面（出现在两个变量之间，表示乘法运算的运算符*是二元运算符）。运算符&可视为运算符*的逆运算，其作用是获取变量的地址，关于这两个运算符详细内容见 6.4.8 和 6.4.10。
- 关于运算符 new 和 delete 将在 6.4.8 中和有关 class 的内容中（7.1.3）予以详细介绍。

（2）二元运算符（binary operator）

表 6-8 是 C++语言中常用的一些二元运算符。

表 6-8 C++语言中常用的二元运算符

二元运算符	说 明
+	加（addition）
-	减（minus）
*	乘（multiply）
/	除（divide）
%	整除（modulus）
<	小于（less than）
>	大于（greater than）
<=	不大于，小于等于（not greater than）
>=	不小于，大于等于（not less than）
==	等于（equal to）
!=	不等于（not equal to）
&&	逻辑与运算（logical AND）
‖	逻辑或运算（logical OR）
&	字位与运算（bitwise AND）
^	字位异或运算（bitwise exclusive OR）
|	字位或运算（bitwise inclusive OR）
=	赋值运算符（assignment）
*=	乘赋值符（assign product）
/=	除赋值符（assign quotient）
%=	整除赋值符（assign modulus）
+=	加赋值符（assign sum）
-=	减赋值符（assign difference）
<<=	左移位运算符（assign left shift）
>>=	右移位运算符（assign right shift）
&=	字位与赋值符（assign bitwise AND）
^=	字位异或赋值符（assign bitwise XOR）
|=	字位或赋值符（assign bitwise OR）

表 6-8 中的运算符大致可分成以下几类：

- 数学运算符，如+、–、*、/ 等，这些都是大家有所了解的，只有运算符%不太熟悉，它的作用是取余数。例如，在下述例句中

int a = 5 % 3; （变量 a 的值为 2，因为 5 除以 3 得余数 2）

int b = 10 % 5; （变量 b 的值为 0，因为 10 可以被 5 整除）

运算符+=、*= 等实际上是这些数学运算符的简略形式。例如：

语句 a += b; 相当于 a = a+b；

语句 c *=d; 相当于 c = c*d；依此类推。

- 比较运算符，如<、>、<=、>=、==、!=等，用于比较两个运算数的大小或相等。
- 逻辑运算符 &&和|| 用于进行逻辑与、或的运算（参见 6.4.6、6.4.7 节）。
- 字位运算符，如&、^、| 等，就是把逻辑运算应用到二进制数的字位（bit）上，字位逻辑运算规则如表 6-9 所示。

表 6-9 字位逻辑运算规则

E1 的值	E2 的值	E1&E2 的值	E1^E2 的值	E1|E2 的值
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	0	1

除此以外，C++还有许多特有的运算符，由于它们在本教材中应用较少，而且较难理解，就不在这里深入涉及了。

当一个运算式中包括多个运算符时，优先级高的运算先进行，即通常所说的“先乘除、后加减”规则。优先级相同的运算符则按“自左至右”的顺序进行运算。括号（）可用于提高运算符的优先级，如括号中的加、减运算会先于括号外的乘除运算执行。

导读：

本节从语法角度介绍了 C++语言中的各种运算符，但是其中许多运算符在初学编程时很少遇到。因此，不一定现在就要完全弄清每个运算符的作用。随着编程实践的发展，自然会逐个熟悉起来。

6.3.7 注释

注释（comment）是对程序语句的说明，与程序执行无关。长的注释可能延续多行，但程序都把它作为一个空格字符对待，这就是为什么把注释归类为 Token 的原因。在程序中加入详尽的注释是规范化编程必须的实践，对提高程序的文档性、可读性和可维护性来说是必不可少的。

在 C++程序中，表示注释的规则是：

① 以符号//开头，直到该行结束。

② 介于符号/*和*/之间。

③ 注释可加在程序语句中的任何位置。

④ 注释不能嵌套。

下面就是关于注释的一些例句：

```
// sample comments
int  i;       /* array member index */
float x;    // a variable
int /* array counter */ i;
```

不合法句子：

```
int i; /* array /* conter */ index */
```

规范化编程实践–2：程序中应有足够详细的注释。

一般建议在程序中以下地方加注释：

① 文件开头：包括文件名（这是对文件内容的大致描述）、文件的创建者、创建日期、文件各次修改记录（包括修改者，修改日期）、其他必要的说明。

② 函数：包括对该函数功能的描述、调用参数的描述、函数返回值的描述等。

③ 变量的说明。

④ 复杂控制逻辑的说明，例如，当成对的两个{}距离较远时，可在结束行表明与哪个语句配对。

⑤ 其他有必要说明之处。

学员可通过观察本教材提供的范例程序、VC++中各种范例程序来学习注释的写法。

6.4 C++语言句法概要

教材编排说明：

① 正如下面将要指出的那样，C++语句必须组成函数才能被执行。但是在解释 C++语言的句法时，主要是介绍单个的语句，而不是完整的可直接运行的程序。因此，凡是课文中所说的“例句”都不是完整的程序例子，必须将它们编写为完整的函数，才能观察其运行效果。

② 教材中给出编号的例子，如例 6-1 等则是完整的函数，可以直接用在程序中。

编程语言是编程员与计算机交流的符号，是在数据对象上所执行的一系列运算的说明。将词汇按句法组织起来，便成为程序语句。C++程序语句主要有两大类：用于描述程序所处理的数据对象的定义类语句和对数据对象进行运算的可执行类语句。下面对其句法逐一予以介绍。

6.4.1 定义语句

定义语句（definition）语句的作用是定义程序所涉及的数据对象，包括变量、常数以及编程员自定义的数据等。在 C++里，任何标识符都必须先定义，然后才能在程序中使用，所以定义语句总是位于程序文件和函数体的开头。

定义语句的种类有：

- 预处理宏指令（preprocessor macro）
- 变量（variable）
- 常数（literal）
- 数据类型（typedef）
- 数组（array）
- 函数（function）
- 类、类的成员（class、class member）

下面分别介绍之：

（1）预处理宏指令

预处理宏指令中最常用的是将系统库函数包括进来的 include 语句，它一般位于程序文件的最开始处。其语句格式如下述例句所示：

```
#include <stdio.h>
```

该定义语句以标点符号＃开始，后面是关键词 include，括号< >中是系统库函数文件名。C++语言提供了数量可观的可供编程员调用的库函数，但若将它们都包括在用户程序中，用户程序势必变得庞大无比，于是程序一般只包括那些需要的库函数，而不包括其他函数。为此，系统把库函数分门别类地定义在若干库函数文件里，用户程序只需包括那些它们需要的文件即可。例如，上面的系统文件 stdio.h 就是提供基本的输入、输出函数的，如程序 smaple_1.cpp 中的屏幕打印函数 printf(…)。文件扩展名 h 的意思是 head file，通常称为头文件。

表 6-10 列出了常用的一些系统函数头文件的名称及其内容。

表 6-10 常用的系统函数头文件名称及内容

文 件 名	内 容
stdio.h	输入、输出函数
iostream.h	流输入、输出函数
ctype.h	字符种类判断函数，如 isdigit、isascii 等
math.h	各种数学函数
string.h	字符串运算函数，如 strlen、strcpy、strcat 等
stdlib.h	字符处理函数，如 tolower、toupper、itoa、atoi、atof 等
limits.h	整数的表示范围
float.h	浮点数的表示范围

其他一些常用的预处理宏指令将在后续内容中分别予以介绍（见 7.2.5 节）。

（2）变量和常数

以下是变量和常量定义语句格式的示例：

```
// 定义变量
int a, sum;
double x, y = 1.23;
// 定义常数
```

```
const int b = 25;
const double e = 5.73;
```

上述例句表明：

- 定义变量的语句以数据类型开始，然后是编程员命名的变量。
- 定义常数的语句在句子前要加上关键词 const，句子后面用符号 = 给常数赋值。

常量的值在程序运行中不可改变，而变量的值可以通过运算改变。变量在定义中可以被赋值，也可不赋值。

（3）数组

将在 6.4.3 节中予以详细论述。

（4）函数

将在 6.4.4 节中予以详细论述。

（5）类型定义语句

C++直接提供的基本数据类型名称只有 4 种，但允许编程员为已有的数据类型另起别名。起别名的目的一般都是增加程序的文档性。例如，C++中没有逻辑变量，而用 int 来表示逻辑变量。为了增加程序的文档性，编程员可用定义一个 BOOL（布尔）类型，以区分用于逻辑运算的 int 变量和用于算术运算的 int 变量。如下面例句所示：

```
typedef int BOOL;
BOOL  u = 1, v;
int   a, b;
v = !u;
```

由例句可知，类型定义语句以关键词 typedef 开始，后面是系统认可的数据类型，包括 4 种基本的数据类型以及编程员定义的类等，最后是新数据类型的名字。

（6）类、类成员

typedef 语句只能为已有的数据类型另起一个别名，而类允许编程员根据需要将数据和数据上的运算组合为一个有机整体，表达高层次的抽象概念，它才是真正意义上的新数据类型。正是这种定义类的机制使得 C++成为一种面向对象的编程语言，关于它的详细论述见第 7 章。

6.4.2 数据类型的转换

C++语言支持 3 种数据类型：基本类型、导出类型和类。基本类型已在 6.3.5 节予以详细论述。不同基本类型变量的转换必须用调用函数的方式进行。

① 用函数 float()或者 int()转换整数和浮点数，如以下例句所示：

```
int i = 3, j;
float x, y = 3.55;
x = float(i);
j = int(y);
```

不能直接对不同类型的变量进行赋值，以下语句是错误的：

```
x = i;
j = y;
```

② 字符和整数值的转换应调用系统头文件 stdlib.h 里定义的专用函数进行。因此首先应在程序中用 include 语句将该头文件包括到程序中。具体应用这些函数的方法如以下例句所示：

```
#include <stdlib.h>
int i;
float x;
char a = 'x', b='y';
i = atoi(a);
x = atof(b);
```

6.4.3 导出数据类型

导出数据类型由基本数据类型发展而来，包括枚举（enum）、数组、指针和引用（reference），下面是关于枚举和数组的介绍。关于指针类型参看 6.4.8 节，关于引用类型参看 7.1.5 节。

（1）枚举

用以对若干正整数常量起别名以增强程序的可读性，其语句格式如以下例句所示：

```
enum Days
{
    saturday,                  // saturday = 0 by default
   sunday  =  0,               // sunday = 0 as well
    monday,                    // monday = 1
    tuesday,                   // tuesday = 2
    wednesday,                 // etc.
    thursday,
    friday
}   date;                      // Variable
```

其中 Days 是新的数据类型，date 是该类型的一个变量，saturday 等名字是变量 date 可取的值，与整数相当。

若没有特别说明，enum 类型的变量值从 0 开始，如上述定义中的 saturday＝0，然后依次递增。但可以根据需要规定其中一些常数的值，如上述定义中特别规定了 sunday＝0，从而使表示周末两天的值都为 0。

（2）数组

数组是同一类型的数（基本类型或者类）的集合，语句以数据类型名开始，后跟变量名，变量名后括号[ ]中的数字表示其元素数量，也称为数组的长度。以下是数组定义语句的例句：

```
// 定义一个有 5 个浮点数的数组
float   z[5];
// 定义一个有 10 个字符的数组，即字符串
char    s[10];
```

数组具有以下性质：

- 它所包含的元素个数，即数组的长度必须在定义时予以确定。

- 用表达式“数组变量名[下标变量]”可对数组中指定元素进行访问，所谓访问指对它赋值或将其值赋给其他变量。下标变量的取值范围是从 0~*n*-1，*n* 是数组的长度。以下就是应用下标运算符的一些例句：

```
z[0] = 1.25;  z[1] = 2.36;
float  x = z[0];
```

- 数组还可以是多维的，如语句：

```
float b[10][10];
```

定义的是一个 10×10 的二维数组。

数组常与迭代语句配合使用，使程序结构变得紧凑，详见 6.4.6 节关于程序控制语句的讨论。

> **经验与提醒：**
>
> 若用下标运算符访问数组中的元素，下标值超过数组的定义长度，程序会出现严重错误，甚至死机。因此在访问数组元素时，一定要先检查下标值是否小于等于 *n*–1 和大于等于 0，然后才可以应用下标运算符，*n* 是数组的定义长度。

6.4.4 函数和函数调用机制

在 C++语言中，语句不能独立存在，语句必须组织为函数。函数是程序的最小逻辑单元。

（1）函数调用语句

用于定义一个函数的具体内容，即其内部的具体的程序语句组成。下面是函数定义语句格式示例：

```
// 有返回值的函数
int  Add(int x, int y)
{
     int   z;
     z = x+y;
     return z;
}

// 没有返回值的函数
void PrintResult(int a)
{
      printf("Result=%d\n",a);
};
```

例句表明，函数定义语句包括彼此用空格隔开的 4 部分：函数返回值类型、函数名、调用参数表、函数体。

- 函数返回值类型：函数返回值是函数与调用它的外部程序的输出接口，函数通过它将处理结果提交给外部程序。定义 int Add()说明函数 Add 执行完毕后将返回给调用它的程序一个类

型为 int 的数值，用以参与后续运算。

返回值的类型可以是系统基本数据类型，更多时候则为编程员定义的类。C++要求函数一般都有返回值，为此函数体中最后一个语句必须是“return 变量名”的形式。但是，如果确实没有什么值需要返回时，可将返回值的类型定义为 void，函数体中最后的 return 语句也随之省略。例如，以上例句中的函数 PrintResult 的作用只是打印数据，不必向调用它的外部程序返回任何数据，所以返回类型为 void。

- 函数名：一般为动词和名词组成的短语，以充分描述函数的功能（参见 4.3.4 节中规范化编程实践–1）。
- 括号( )中是调用参数表：它是函数与外部程序的输入接口，列出外部程序在调用它时必须向它提供的一系列数据及其类型，彼此用逗号隔开。在函数被调用时，外部程序必须向它提供与参数表规定的数量、类型完全一致的实际数值。
- 括号{ }中为函数体：由定义语句和可执行语句组成。

（2）函数调用

即函数的调用，一般来说，函数被其他程序调用的形式有两种：

- 以变量形式出现在运算式中，该变量的类型即函数的返回数据类型。
- 作为一个独立语句。

例 6-2：函数调用的两种形式。

```
//  main.cpp,  主函数模块
//   被调用函数的声明语句
int   Add(int, int);
void PrintResult(int);
void main()
{
      int  a=2, b=3,c;
      //  函数 Add 作为一个变量出现在运算式中
      c  =  (Add(a, b)＋4)*7;
      //  函数 PrintResult 作为一个独立语句
       PrintResult(c);
};
```

例 6-2 表明，当一个函数要调用其他函数时，必须在其函数定义语句前用函数声明语句（function declaration）列出被调用的函数（本节稍后将对函数声明语句加以解释），在函数体中可以把被调用的函数作为变量写入运算式中，或者作为一个单独的语句。

图 6-15 是对程序模块化和例 6-2 中函数调用过程的描述：当程序运行遇到某个函数表示式时，将转到被调用函数中，执行其中的程序语句，完成有关的计算。当被调用函数执行完最后一个语句后，程序又回到原先调用它的程序中进行执行后续语句。如果函数有返回值，则通过 return 语句将计算结果返回给调用它的程序，参与那里的运算。例如语句 c = Add(a,b) 中变量 a 和 b 就是传送给子函数 Add 的参数，在调用函数 Add 前，它们已被赋予实际数值，否则被调用函数 Add 是无法运行的。函数 Add 中语句 return z 的作用就是把计算结果返回给外部程序，

并继续在语句 c = Add(a,b)+4 中继续参与其余的运算。

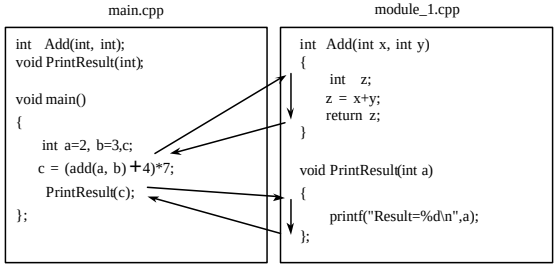


图 6-15 程序模块化和函数调用过程示意

函数内部相对于程序其他部分应该是完全屏蔽的，函数一般应该只用其返回值作为对外部程序的惟一数据输出接口，若需要返回的值不只一个，应将有关数据组合成一个能表达更高层次概念的类，作为一个有机整体一起返回（详见第 7 章关于类的论述）。函数应该只使用调用参数作为它与外部程序惟一的数据输入接口，而不应通过外部程序定义的全局变量来交换信息。如下面的程序语句就是错误的，因为函数中使用了在函数体外面定义的全局变量 x 进行运算。

```
float x = 2.3;
float f()
{
    float  y;
    y = x + 3.0;
    return y;
}
```

任何 C++程序均须有且只有一个名为 main 的主函数。函数 main 通过调用其他函数（统称为子函数）完成整个程序的运行。如果没有主函数的调用，子函数不能单独被计算机运行。子函数互相之间也可调用，子函数也可被自己调用，称为递归调用，但主函数不能被子函数调用。

正如下一节将要讨论的那样，程序语句能直接进行的运算只有基本的算术和逻辑运算，而计算机程序要解决的实际问题往往很复杂，解决问题的过程也很复杂。人类解决复杂问题常用的方法是将一个大而复杂问题分解成若干子问题，把解决问题的过程也对应地分解为若干子过程，通过子过程逐个解决子问题，最后解决整个问题。这就是 C++程序以函数为基本逻辑单位的原因，它将解决特定子问题的有关语句组织起来，可以说是子过程的抽象，主函数通过调用子函数逐个解决子问题，直到解决整个问题。

将一个复杂的程序分解为函数还可以大大增加程序的可读性。一个由大量的基本程序语句组成的程序是很难看懂的，但若将程序适当地分解成函数，并恰当地给每个函数命名，使其充分概括其所执行的子过程的意义，就能很好地反映出程序的内在逻辑。

规范化编程作业一般都要求安排主函数 main 单独占用一个模块，而将子函数按功能和调用关系组合在不同模块里。划分程序模块的另一个方法是按 class 分块（详见 7.2.4 节）。

（3）函数声明语句

根据程序模块化的原则，函数与调用它的函数往往不在同一个程序模块文件里，而 C++要求运算式中只能包括被定义过的标识符，但在一个包括多个文件的程序中，定义又是不能重复的（详见 6.4.1 节），因此除了函数定义语句以外，常常还需要用到函数的声明语句，其作用是在函数被调用的文件模块里（函数本身定义不在该模块中）对之加以声明。

函数声明语句的格式与定义语句相同，但只有 3 部分：返回值类型、函数名和调用参数表，但不包括函数体。其参数表也可简化，只要列出参数个数和类型即可，而不必有参数名字。如图 6-15 中主函数 main 前的语句 int Add(int, int)中只列出数据类型。

函数的定义语句在程序全部模块文件里只能出现一次，而函数的声明语句可有多个，在调用其语句的模块里均应有其声明语句，否则程序在编译时会报错。

此外，若函数定义语句与其声明语句不完全一致，在连编时会认为程序有错。例如，若将图 6–15 的模块 main.cpp 中函数声明语句 Add(int, int)中的参数类型改为（int, float），将调用参数 b 的类型改为 float，虽然编译时不会报错，但在连编时 VC++编译器会显示以下信息：

```
sample_1.obj: error LNK2001: unresolved external symbol "double __cdecl Add(int, float)" (?Multiply@@YANNN@Z)
Debug/project_1.exe : fatal error LNK1120: 1 unresolved externals
Error executing link.exe.
```

即编译器认为在模块文件 main.cpp 中所声明的函数 Add(int, float)实际上是一个没有定义过的函数。

从上面的讨论，可将在程序中应用函数的一些规则总结如下。

规范化编程实践–3：程序应模块化，安排程序主函数为一个单独模块，其余函数按功能分成若干模块。

规范化编程实践– 4：函数只能用调用参数和返回值作为其与外部程序的数据交换接口，不能在函数体中直接使用外部程序定义的全局变量。

规范化编程实践–5：不要定义整个模块或者整个程序共用的全局变量。

导读：程序模块化与 VC++的动态链接库(DLL)

编译器在连编（link）时，一般是将所有模块组合为一个可执行代码整体，这种办法称为静态连编（static linking）。动态连编（dynamic linking）可将其中一些函数组成为一个独立于主程序的库，称为动态链接库（dynamic -link library，DLL）。当主程序运行需要某个函数时，就将该函数从库中调入内存中运行，运行完毕将其从内存中清除。采用 DLL，除可减少程序所占用内存空间外还有以下优点：DLL 可独立于应用程序另行编译和 link，可被单独更新；若干程序可共享一些 DLL，因此可将一些常用的函数标准化后供不同用户程序共享；DLL 还可在不同语言编写的程序之间共享，实现跨语言平台的应用。这是程序模块化的另一大优点。

虽然 DLL 并非 C++语言本身的功能，而是 Windows 系统的特点，但考虑到微机上普遍用 VC++开发 Windows 下的应用程序，特在书后配盘中提供了一个生成 DLL 库的例子。有兴趣者可参看 VC++的用户手册或专门教材深入学习关于 DLL 的知识。

6.4.5 运算式

运算式（expression）语句是程序可执行语句的一种。它用 operator（运算符）将 operand（运算数）连接，指定其计算的顺序。这里所说的运算数并非只是纯数学意义上的数，而是泛指数据对象，计算也并非只是数学运算，而是广义地指计算机对数据进行的各种处理，包括逻辑运算等。

从语法定义上来说，C++语言中运算式的主要类型包括：

- primary expression，基本运算式，用以构成其他所有运算式。
- unary expression，由一元运算符后接运算式构成。
- binary expression，由二元运算符和运算式构成。
- postfix expression，由基本运算式后接 post operator（后缀运算符）构成。

其中 primary expression（基本运算式）就是编程员定义的变量、常数和函数，它们与运算符以及括号有限次的组合构成了更广义的运算式。unary expression 由一元运算符与一个运算式构成。binary expression 则由两个运算式和一个二元运算符构成。这两种运算符已在 6.3.6 详细介绍，本节不再重复。下面着重讨论 postfix expression，它由基本运算式后接 post operator（后缀运算符）构成。后缀运算符的意思是，运算符必须出现在变量名的后边。表 6-11 是 C++语言中一些主要的后缀运算符：

表 6-11 C++语言中的主要后缀运算符

操 作 符	功 能
[ ]	数组成员访问运算符（subscript operator）
( )	函数调用运算符（function-call operator）
++	增量运算（postfix increment operator）
--	减量运算（postfix decrement operator）
.	类成员访问运算（member-access operator）
->	类成员访问运算（member-access operator）

其中数组成员访问运算符[]、函数调用运算符 ()、++和––等，已在有关章节中介绍。运算符“ . ”和“ -> ”用于对类（class）的成员进行操作，详见 7.1.4 节。

程序中的运算式一般有 4 种形式：

① x = (y + z)*(a – c)/d; 虽然它与普通算式相似，但符号 = 不能理解为“等于”，而是“赋值”，即把右边运算式的计算结果赋予左边的变量。

② x = x + 20; 表达式左边的变量 x 本身也参与右边运算式进行运算，然后将计算结果作为左边变量 x 的新值赋予它。

③ x++; 只适用于增量和减量运算，变量自己与数字 1 进行加运算，然后把计算结果赋给自己。

④ 附于变量之上，形成另一个类型的变量。例如，当下标运算符[ ]加在字符串变量 char x[5]后面所形成的运算式 x[1]、x[j]等表示的不再是字符串，而是数组中的其中一个元素，一个字符。

C++语言中还有一些其他的运算式，如带条件运算符的运算式，常量表达式等，可参考书

后配盘中的 C++语法。

6.4.6 程序控制语句

运算式语句是按它们在文件中排列的顺序执行的，而程序控制语句可根据给定逻辑控制程序语句的执行顺序，从而使程序具有进行判断和执行复杂运算的能力。C++中用于控制程序执行逻辑的语句有选择（selection）和迭代（iteration）两大类，每类均有若干种句型。

在介绍语句之前，首先简要介绍描述程序执行逻辑的工具：流程图（flowchart）。在程序设计中，流程图和其他类似工具是理解和阐明程序逻辑流程的重要工具，采用流程图还能使我们不必考虑程序内部的细节就能分析程序的逻辑。流程图中的一些常用符号已经标准化，如图 6-16 所示，下面的论述将用它们描述各种控制语句的逻辑流程。

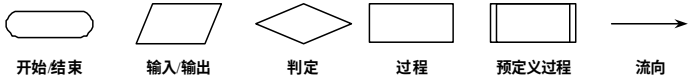


图 6-16 flowchart 基本图符

（1）选择（selection）语句

它有两种形式。第一种为 if 语句（if–statement），用于在两段程序中择其一而执行，其控制流程如图 6-17 所示。

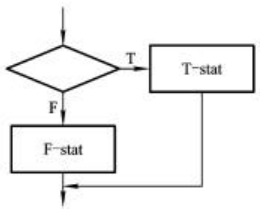


图 6-17 if 语句的控制逻辑

if–statement 语句的格式是：

```
if (x !=1) {
    x = x + 2;
    y = y + 3;
}
else
    x = x - 2;
```

图 6-17 中菱形表示的判定条件由 if 语句中括号(…)中的逻辑运算式表示；当判定条件为真时，执行符号（…）后面的语句，也就是图 6-17 中的 T-stat 方框中的语句（T-stat 为 TRUE–statement 的简写）；当判定条件为假时，程序执行关键词 else 后面的语句，即图 6-17 中 F-stat 方框中的语句（F-state 是 False–statement 的简写）。

括号{ }是标点符号的一种，用于表示复合语句的开始和结束（参见 6.3.2 节）。当其中只有一个语句时，括号{ }可省略。以上例句中 else 后面只有一个语句，因此没有括号{ }。

if-语句可简化为 if (expression) TRUE–statement。即当 expression 的值为真时，执行 TRUE–语句，反之则直接执行下一个程序语句。如下面例句所示。

```
if (x < 0) {
    printf("x < 0\n");
    x = - x;
}；
```

if-语句还可以嵌套，即 T–statement 或者 F–statement 本身也可以是一个 if-语句。

```
if (x<0)
    printf("x < 0\n");
else if (x == 0)
    printf ("x == 0\n");
else
    printf ("x > 0\n");
```

上述多级判断的过程一般可表示为图 6-18 所示的树形结构。

第二种 selection 语句是 switch 语句（switch-statement）。它能在多个分支程序段中择其一而执行，其控制逻辑如图 6–19 所示。

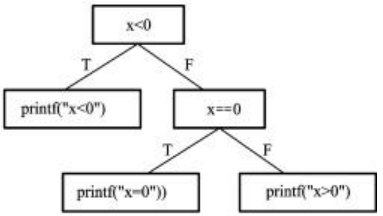


图 6-18 用树表示嵌套的选择逻辑

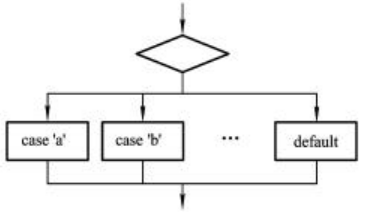


图 6-19 switch 语句的控制逻辑

switch 语句的格式是：

```
switch (ch) {
    case 'a': printf("ch==a\n"); break;
    case 'b': printf("ch==b\n");
    case 'c': printf("ch==c\n"); break;
    case 'd': printf("ch==d\n"); break;
     default:
          printf("ch != a-d\n");
};
```

图 6-19 中菱形表示的判定条件由关键词 switch 后面括号（）中变量的值表示，程序根据其值从 case 表示的分支中选择一个符合条件的语句运行。例如，上述例句的执行过程如下：若变量 ch 值等于字符'a'，则执行 case 'a'语句，结果是在屏幕上打印出'ch==a'；若变量值等于字符'c'，则执行 case 'c'语句，结果在屏幕上打印出'ch==c'。余类推之。若 ch 不等于 a、b、c、d 中的任何一个，则屏幕上将打印出'ch != a–d'。

应用 switch-statement 应注意以下几点：

① 用于选择分支的变量须为整数型变量或者相当于与整数型的变量，如字符等。

② 在每个 case 语句最后，一般都应有语句 break。其作用是告诉程序在执行完这一选择后，直接跳过其余 case 语句到句末。若无 break 语句，则程序将顺序执行下一个 case。例如，在例句中有意略去了 case 'b' 语句后的 break，因此若当 ch 的值等于 b 时，程序实际上会打印出两个

语句，'ch==b'和'ch==c'。

③ default 语句是必不可少的，否则程序会被判定有语法错误。若在该选择项下确实不需执行任务，则简单地给一个语句结束符；即可。

（2）迭代、循环语句（iteration–statement）

它又分为 3 种：while 语句（while–statement）、do while 语句（do–while statement）和 for 语句（for–statement），下面分别介绍之：

① while 语句的控制逻辑如图 6-20 所示。

while 语句的格式是:

```
int j = 1, sum = 0,
while (j < 5) {
        sum = sum + j;
        j++;
};
```

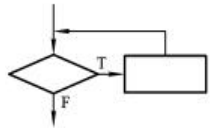


图 6-20 while 语句的控制逻辑

图 6-20 中菱形表示的判定条件由语句中 while 后面的括号（）中的逻辑运算式表示，称为循环条件。语句执行时，首先对循环条件进行判定，如其为真，程序就执行括号 { }中的语句，即图 6-20 中的方框，通常称为循环体；执行完毕后返回句首，再次计算循环条件并判断其真假。上述过程将反复进行，直到循环条件为假时，循环结束。若一开始时循环条件即为假，循环将一次也不执行。

以上例句的执行过程是：第一次，j 值为 1，运算式 j<5 的值为真，括号{ }中的复合语句将被执行，结果使 j 值变为 2；然后回到句首再次执行运算式 j<5，结果仍为真，于是括号{ }中的语句将再被执行一遍；如此继续直到 j=5 时，循环停止。上述语句执行的结果使变量 sum = 1+2+3+4 = 10。

② do-while 语句的控制逻辑如图 6-21 所示。

do-while 语句格式是：

```
int j = 1, sum = 0;
do {
        sum = sum + j;
        j++;
}   (j <5);
```

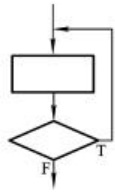


图 6-21 do-while 语句的控制逻辑

图 6-21 中菱形表示的判定条件由语句最后括号（…）中的逻辑运算式表示，称为循环条件。语句执行时，首先执行关键词 do 后面括号{ }中的语句，即图 6-21 中方框表示的循环体；执行完毕，计算循环条件，若其值为真，则程序返回句首，再次执行循环体；若其值为假，则语句结束。与 while 语句不同的是，即使一开始循环条件即为假，循环体也会被执行一次。

以上例句的执行过程是：第一次执行完括号{ }中的复合语句后，j 值为 2，运算式 j<5 的值为真，于是回到句首再次执行括号{ }中的复合语句；如此继续直到 j = 6时，循环停止。上述语句执行的结果使变量 sum = 1+2+3+4 = 10。

③ for 语句的控制逻辑同 while 语句（见图 6–20），其语句格式是：

```
for (int j = 1, sum = 0; j<5; j++) {
```

```
        sum = sum + j;
};
```

语句以关键词 for 开始，括号( )中包括 3 个部分：第 1 部分是对变量赋初值，被赋值的变量一般为逻辑运算式中的变量，通常称为循环变量，但也可包括括号{ }中循环体中的变量；第 2 部分是循环条件；第 3 部分是运算式，与第一部分一样，它可包括对循环变量的运算，也可包括对循环体中变量的运算。

语句的执行过程为：首先执行括号( )中第 1 部分，对有关变量赋初值；然后计算第 2 部分的循环条件，若其值为真，则执行循环体；然后返回句首括号( )中，执行第 3 部分中的运算式；然后重新计算第 2 部分的循环条件，若其值为真，则继续执行循环体，若其值为假，则跳过循环体，语句执行结束。和 while 语句一样，如果一开始循环条件即为假，循环体将一次也不执行。

以上例句的运行过程如下：循环开始，变量 j 被赋初值 1，循环条件 j<5 为真，循环体被执行一次；然后执行运算式 j++使其值为 2，循环体 j<5 判断结果为真，于是再次执行循环体；上述过程反复进行直到 j=5 时为止，循环共执行 4 次。

当循环体的复合语句中只有一个语句时，括号{ }可略去，如以上例句可采用更紧凑的形式：

```
for (int j = 1, sum = 0; j<5; j++)
        sum = sum + j;
```

for 语句中的循环变量可以递增，也可递减。下面这个句子与上面执行结果相同。

```
for (jnt j = 5, sum = 0; j >0; j = j - 1)
        sum = sum + j;
```

与 while 语句相比，for 语句用循环变量控制迭代，循环的步长是固定的，而 while 语句中循环的步长可灵活变化。另一方面，for 语句的句子形式更紧凑。

所有的循环语句都可以嵌套，即其循环体中还可包括另一些循环语句。

```
float    x[10][10];
for (int j = 0; j < 10; j++)
    for (int k = 0; k < 10; k++)
            x[j][k] = …;                    // 对该元素进行赋值
```

经验与提醒：

应特别注意不要将程序控制语句逻辑判断式中的双等于符号“==”误写为单等于符号“=”，编译器是无法发现这种错误的，通过程序运行检测它也十分困难。

6.4.7 应用数理逻辑设计程序控制语句

从以上关于程序控制语句的介绍可知，控制语句中的逻辑运算式是控制程序正确运行、取得预期结果的关键，而第 2 章讨论的命题逻辑和第 3 章讨论的谓词逻辑正是指导设计程序控制逻辑的数学理论基础。本节将介绍如何应用命题逻辑和谓词逻辑指导设计程序控制语句。

用数理逻辑设计程序控制逻辑的一般步骤是：

① 将实际问题抽象为逻辑表达式（命题逻辑或者谓词逻辑）。

② 对逻辑表达式进行简化。

③ 将逻辑表达转换为 C++语言的逻辑运算式，其规则为：

- 非运算对应逻辑运算符!
- 合取运算对应逻辑运算符&&
- 析取运算对应逻辑运算符||

例 6-3 详细说明了应用命题逻辑和谓词逻辑设计程序控制语句的方法和步骤。

例 6-3：软件设计是一项要求严密的工程性任务，除了确保程序本身的正确性以外，还应考虑到用户在使用中可能发生的各种问题（参见 9.3.4 节）。例如，如果程序要求输入一个浮点数，而用户由于按键操作失误，很可能在数字中夹杂着字母。如果程序将这类输入作为数来进行运算，问题小则产生错误计算结果，严重时还会引起程序崩溃。但是，系统提供的输入函数（参见 6.4.10 节）并不具备检测功能，软件设计人员必须自行设计有关程序来保证系统只接受格式正确的输入数据。具体方法如下：不直接用格式输入语句读入数字，而是将输入数据作为字符串读入，用一个程序检查其中是否有数字字符，然后再用系统函数 atoi()或 aotf()将字符串转换成整数或浮点数。本例应用谓词逻辑设计一个函数，用以检测输入数据是否全是数字字符。

解：因为数字字符和非数字字符均有多个，因此可用谓词公式来描述判定字符是否合法数字字符的条件。首先设原子公式如下：

IsDigit（x） x **是数字字符**

IsDot（x） x **是小数点**

然后用上述公式组成复合公式表示字符是合法数字的条件：

IsDigit（x）∨ IsDot（x）

如果用自然语言来描述该公式的意思，就是“若 x 是数字字符或者小数点，则其为合法的数字”。

在将上述谓词公式转换为 C++程序时，可用以下函数实现基本谓词公式：

```
// 判定字符是否数字，对应谓词公式 IsDigit()的函数
int IsDigit（char x）
{
    if (x <=57  &&  x >=48)
        return 1;
    else
        return 0;
};
```

其中语句 if (x <=57 && x >=48) **语句中的判定条件也是应用命题逻辑得出的。首先设基本命题：**

P：**字符** x **的** ASCII **码值大于等于** 48（**对应数字** 0）

Q：**字符** x **的** ASCII **码值小于等于** 57（**对应数字** 9）

字符 x 为数字的条件为既大于等于 48，又小于等于 57，所以可用复合命题表示为：

P∧Q

根据转换规则，与合取运算符∧对应的 C++运算符为&&，所以当运算式 (x <=57 && x >= 48)为真时，使函数返回 1，否则返回 0。

```
// 判定字符是否小数点，对应谓词公式 IsDot()的函数
int IsDot（char x）
{
    return x==46;
};
```

其中语句 return x==46 直接将逻辑运算式 x==46 结果作为函数的返回值，46 是小数点字符的 ASCII 码值。当 x==46 时，该运算式值为 1（逻辑真），也是函数 IsDot 所需要的返回结果。

用户输入的数字可视为一个由多个字符组成的字符串，所以可用以下函数来检测其中是否有非数字字符。

```
// 判定一个输入字符串是否合法的数字
#include <string.h>  // 包括函数 strlen 定义的头文件（参见 6.4.1 节）
int IsValidDigit(char* s)
{
    int isValid = 1;
    for (int j = 0, k = strlen(s); isValid && j < k; j++)
        isValid = IsDigit(s[j]) || IsDot(s[j]);
    return isValid;
};
```

函数中 for 语句的循环条件也是应用命题逻辑得出的，如果用自然语言来描述就是：

① 当字符数组元素的下标变量等于数组的长度 k 时，检测结束（记住：数组下标的记数是从 0 开始到 n–1）。

② 当检测到非数字字符时，检测结束。

学员可自己练习应用命题逻辑得出上述循环条件表达式。

程序执行逻辑的设计是软件技术的重要内容，好的程序逻辑应该是完全结构化的，满足以下条件：

- 程序执行逻辑只有顺序、选择和迭代 3 种，不用跳转语句控制程序的执行。
- 循环语句和迭代语句不用任何跳转语句（如 goto、break 等）来跳出循环体，而以循环条件为惟一的循环停止条件。
- 选择、迭代等语句中的复合语句内部结构对外是完全屏蔽的，外部程序不用跳转语句进入循环体中。

> **规范化编程实践–6：在程序中不使用任何跳转语句，如 goto、break 等（但 switch 语句中的 break 语句不包括在内）。**

6.4.8 指针变量

导读：

指针的概念是 C++语言学习中的一大难点。不仅应仔细阅读，更重要的是通过编写程序的实践进行理解。

（1）C++变量中所存储的实际内容（或者说其实际值）的两种情形

① int a = 2; **变量 a 的实际内容就是一个值为 2 的整数，这种变量称为静态变量。**

② int* b; **变量 b 中目前并没有内容，必须用运算符 new 赋值：**

```
b=new int;
```

现在变量 b 中的值是另一个整数型变量在内存中的地址。但是那个变量目前还没有值，欲给其赋值，则应该用以下语句：

```
*b = 2;
```

变量 b 前面的运算符*在 C++语法的原文中为 Pointer-Derefernce operator，本教材根据其实际作用称之为“取值运算符”，其作用是获取变量 b 所指代的变量的实际值。

（2）指针类型属于 C++的导出类型

① 指针变量所指代的本体变量是在程序运行过程中生成的，所以指针变量又称为动态变量。

② 指针变量的值是内存中的地址，它既可以指代该地址上的那个变量，也可以指代以该内存地址为起点的一系列同类型的变量。例如：

```
b = new int [5];
```

现在变量 b 的值就是连续 5 个整数型变量在内存中地址的起始位置。

（3）应用指针变量的规则

① 应用前一定要用运算符 new 给它赋值，否则会引起程序运行错误。

② 若指针变量指代的是单个数值，则可用取值运算符*获得其指代的本体变量的真实值。如下面语句所示：

```
int* b = new int;
*b = 2;                    // 对指针变量 b 所指代的本体变量赋值
int   a = *b;              // 将指针变量 b 所指代的本体变量的值赋给变量 a
```

③ 若指针变量指代的是一系列同类型的变量，则可用下标运算符[ ]访问这一系列元素中指定的一个，如下面语句所示：

```
int* b = new int [5];
b[0] = 1; b[1] = 2; b[2] = 3; b[3] = 4; b[4] = 5;
```

该形式与访问数组中元素的表达式完全相同。这并不是巧合，因为表示数组的变量，如定义 int c[5]; **中的变量 c 的值实际上也是连续 5 个整数在内存中的起始地址。**

因此，数组变量的实质与指针变量一样，均表示内存地址。C++中其他两种数据类型整数和浮点数，还有编程员自己定义的类（详见第 7 章），则都是表示实际数值。

弄清变量的内容究竟是地址还是实际数据非常重要。一方面，一般的运算符不能应用于内容为地址的变量，如不能用赋值运算符“=”直接对字符串进行赋值。另一方面，有些运算需

要对变量的地址进行。如 C++中有关从文件和键盘输入数据的函数就是直接将输入内容传送到变量的地址中，因此，对于整数、浮点数类型的变量来说，必须用取地址运算符&获取其地址，才能将输入数据传送给它们（详见 6.4.10 节关于输出、输入语句的内容）。

（4）指针变量的性质

① 数组和指针的异同。普通数组型变量所指代变量的数量是在定义时确定的，因此称为静态数组；而用指针变量和运算符 new，可在程序运行过程中按需要长度产生一个动态的数组。如下面语句所示：

```
float*      x;
int *       p;
char *    s;
// 动态生成数组
x = new float[20];                  // x 成为有 20 个浮点数的数组
p = new int [3];                    // p 成为有 3 个整数的数组
s = new char [256];                 // s 成为有 256 个字符的数组
```

然后就可像数组一样用下标运算符访问其中的元素：

```
x[0] = 1.0;   x[1] = 2.0;
p[0] = 1;   p[1] = 3;   p[2] = 5;
s[0] = 'a';   s[1] = 'r';   s[2] = 'r';   s[3] = 'a';    s[4]= 'y';   s[5] = '\0';
```

经验与提醒：

字符串必须以字符'\0'结尾（参见 6.3.5 节），若为常数，系统会自动提供结尾符；若为变量，编程员必须明确给其提供结尾符，否则程序运行时，由于无法检测到一个字符串是否结束而出现不可预计的错误。

② 采用指针变量生成数组的另一个优点是可以重新指定数组的长度，但在重新指定前必须用运算符 delete 释放其所占内存空间，然后重新用运算符 new 给它分配新的长度。下面几个例句的作用就是给变量 p 重新分配内存空间：

```
delete [ ] p;                       // 释放 p 所占内存空间
p = new int [5];                    // 重新给 p 分配空间，使它成为 5 个整数的数组
// 重新给 p 的元素赋值
for (int i=0; i < 5; i++)
      p[i] = i*2 - 1;
```

③ 指针变量的另一个重要作用是作为函数调用参数把一个数组传递给一个函数，或作为返回值向调用它的程序传递一个数组(因为函数的调用参数和返回值都只能是一个单独的数值，因此不能在其定义语句中包括运算符[ ])。下面例句说明了如何用指针变量作参数把一个数组传递给函数：

```
// 求数组中元素的和
// 注意，必须将数组的长度也同时传递给函数
```

```
float Sum(float* element, int length)
{
    float sum = 0.0;
    for (int j =0; j < length; j++)
        sum = sum + element[j];
    return sum;
}
```

④ 指针变量对类（class）类型的数据有重要意义。因为类类型的变量，即类的实例都是有生命周期的，并不是从程序一开始就存在并保持到程序结束，所以，一般无法在程序中预先定义足够数量的静态变量，必须在程序中动态生成。因此，绝大多数类的实例都是用指针变量表示（详见第 7 章和 8.1 节关于类实例管理的内容）。

⑤ 指针还可以是多重的。多重指针相当于一个多维数组。下述语句中的 y 就是一个多重指针变量。

```
int **    y;
```

在应用变量 y 以前，首先为 y 分配内存空间：

```
y = new int* [2];
```

该语句执行的结果使变量 y 成为一个有两个元素的数组，每个元素都是一个 int 类型的指针。要应用每个元素，则还需要用下列语句为其分配内存空间：

```
y[0] = new int [3];
y[1] = new int [4];
```

最后才能对每个元素赋值：

```
y[0][0] = 3;   y[0][1] = 4;   y[0][2]=5;
y[1][0] = 1;   y[1][1] = 3;   y[1][2]=4;    y[1][3]=5;
```

导读：

① 这里特地将两个动态数组的长度设为不同，当然它们可以是相同的。

② C++还可以通过对指针的直接运算对其元素进行访问，虽然其运算式比较紧凑，但没有通过下标运算符[ ]来得直观，程序的可读性也差。这里不再介绍，有兴趣者可参看有关 C++语法的专门论著。

6.4.9 字符串的运算

字符串即字符型变量的数组，在程序中常用于存储文字信息。正如上节所指出的那样，数组变量的值是其地址，因此不能用一般的运算符对其进行运算，如不能直接用赋值运算符 = 对字符串赋值，不能直接用运算符 < 比较大小等，而必须通过调用系统函数来完成有关运算。

下面是字符串运算的一些特点：

- 字符串必须以字符'\0'结尾，在对字符串变量赋值时，一定要给它一个结尾符，否则程序无法识别字符串的结尾，运行时会引起数组下标越界、非法内存的错误。

- 字符串的长度可用系统函数 strlen()获得，其返回值等于其中可见字符的个数，但不包括结尾符。也就是说，字符串实际长度会比函数 strlen()返回的值大 1。
- 若字符串还没赋值，或没有结尾符，则函数 strlen()无法给出字符串的正确长度。
- 可用下标运算符[]对字符串中元素进行访问。

常用的字符串运算函数有以下一些，使用时必须包括系统头文件 string.h。

（1）size_t strlen（const char* s）

返回类型 size_t 是 VC++编译器定义的一个相当于整数的数据类型，函数返回值是字符串 s 的长度。以下是应用函数 strlen 的程序语句实例：

```
char buffer[60] = "How long am I?";
int    len;
len = strlen( buffer );
```

变量 len 的值将等于 14。

（2）strcpy(char* strDestination, const char *strSource)

strncpy(char *strDestinaton, const char *strSource, size_t count)

函数 strcpy 将字符串 strSource 拷贝到 strDestination 中，但编程员必须自己确保 strDestination 的长度不小于 strSource，否则会出现运行错误；

函数 strncpy 中的参数 count 是一个整数类型的变量，规定了从 strSource 拷贝到 strDestination 中字符的个数，显然，count 必须小于 strDestination 的长度。以下例句分别说明了上述两个函数的调用方法与结果：

```
char string[80];
strcpy( string, "Cats are lovely animal" );
```

语句执行后，string 的值为"Cats are lovely animal"

```
strncpy( string, "Dogs", 4 );
string[4] = '\0';
```

语句执行后 string 的值为"Dogs"。

导读：

使用函数 strncpy()时，编程员必须给字符串变量 string 中加上结尾符'\0'，而函数 strcpy 会自动将字符串"Cats are lovely animal"后面的不可见结尾符一起复制到目的变量中。

（3）strcat(char *strDestination, const char *strSource);

strncat(char *strDest, const char *strSource, size_t count);

函数 strcat 将字符串 strSource 接续到 strDestination 后边；strncat 的作用是将字符串 strSource 中头 count 个字符接续到 strDestination 后边，若 count 大于 strSource 的长度，则最多只接续 strSource 长度中的字符。但是，对于两个函数来说，接续后的字符串长度均不能超出 strDestination 的定义长度。以下例句分别说明了上述两个函数的调用方法与结果：

```
char string[80] = "This is the initial";
char suffix[] = " extra text to add to the string...";
```

strcat(string, "string! ");

strncat(string, suffix, 19);

结果使 string 的值为“ This is the initial string! extra text to add”

（4）int strcmp(const char *string1, const char *string2);

int strncmp(const char *string1, const char *string2, size_t count);

函数 strcmp 对字符串 string1 和 string2 逐个字符按其 ASCII 码值进行比较，若遇到相同字符则继续比较下一字符，若 string1 中字符值小于 string2 中的字符值，函数停止比较，并返回<0；反之，若 string1 中字符值大于 string2 中的字符值，函数也停止比较，并返回>0；若函数最后返回=0，则表示两个字符串完全相同。strncmp 的作用基本上相同，但只比较两个字符串的前 count 个字符。以下例句分别说明了上述两个函数的调用方法与结果：

char string1[] = "ABC";

char string2[] = "ABD";

char string3[] = "ABB";

strcmp(string1, string2) 的值<0

strncmp(string1, string2, 2) 的值=0

strcmp(string1, string3) 的值>0

6.4.10 数据的输入输出函数

用户可采用两种方式将需处理的数据输入计算机程序：

- 由用户在键盘上手工输入。显然，这种方式只适用于数据量较少的情况。
- 需要输入的数据量较大时，用户可将数据写入一个文件，由计算机从该文件读入。

与之对应，计算机将计算结果提交给用户的方式也有两种：

- 直接显示在屏幕上。这种方式显然只适用于数据量较少的情况。
- 当需要输出的数据量较大时，计算机可将数据写入一个文件。用户可将文件打印出来或再用其他程序进行处理。

C++语言为这两种输出、输入方式提供了若干输入、输出函数，使用时必须包括系统头文件 stdio.h。

在对文件进行输入、输出操作以前，必须先打开文件，操作完毕后应关闭。下面首先介绍打开和关闭文件的有关函数。

（1）打开文件的函数

FILE *fopen(const char *filename, const char *mode)；

其中：

- FILE 是一个系统定义的指针变量，指向打开的文件。
- *filename 是表示文件名称的字符串。
- *mode 说明文件的访问许可的常数。

*mode 一般包括：

"r" 只读。

"w" 从头写入（文件原有内容会被覆盖）。

"a" 在原有文件后面继续写入。

"r+" 读、写。

"w+" 从头开始读写。

"a+" 读文件，并在文件后面继续写入。

若语句执行成功，则返回一个指向文件的指针变量，用在后续的打印语句中。若文件不存在，则返回一个空指针，空指针在系统中用常数 NULL 表示。如下面例句所示，在应用该函数打开文件时必须总是检查其结果是否成功，以免后续程序使用空指针引起严重错误。

```
FILE* fin;
char*   inputFile  ="input.txt";
if ((fin = fopen(inputFile, "r") == NULL)
      printf("File %s open failure\n", inputFile);
```

（2）关闭文件的函数

```
int fclose(FILE *fp)；
```

其中，*fp 即表示文件的指针。

若语句执行成功，则返回 0；若返回 EOF（文件尾记号）表示执行失败。

（3）打印函数

```
int printf( const char *format [, argument]... );   //  打印到屏幕上
int fprintf( FILE *fp, const char *format [, argument]... ); //  打印到文件中
```

其中：

- 字符串*format 主要是关于被打印数据格式的说明，另外也可包括辅助性文字说明。数据格式的说明包括以下内容：

%[width] [.precision] type

type 是数据类型说明符（一些主要数据类型说明符如表 6-12 所示）；width 是数据的全部位数；precision 是小数点后的位数，例如，%6.2f 表示打印 6 位浮点数，小数点后 2 位。若无 width 和 precision 说明，则系统按默认格式打印。表 6-12 列出了一些主要的读入、写出格式控制符。

表 6-12 数据输出、输入语句中常用格式控制符

格 式	说 明
d	整数
ld	长整形数
x	16 进制数
c	字符
f	浮点数（float）或者双精度数（double）
s	字符串

- argument 即被打印的数据变量名，其后的省略号表明其数量是可变的。argument 前后方括号[ ]的意思是 argument 这一成分可省略。注意变量数据类型与其格式说明必须一一对应，否则会引起程序运行错误。若无数据需要打印，则打印语句只打印字符串*format 的内容。

- 若 printf 语句执行成功，则返回所打印的字符数；若返回一个负值，则表明程序执行有错。

例 6-4：应用打印格式。

```
void  UsePrintFormat  ()
{
      char    ch = 'h';
      char *string = "computer";
      int     a = -9234;
      double   x = 251.7366;
     FILE*  fout=fopen("output.txt", "w");
      // 打印整数
      printf( "Integer formats: Decimal: %d Justified: %.6d\n", a, a);
      fprintf(fout, "Integer formats: Decimal: %d Justified: %.6d\n", a, a);
      // 打印字符
      printf("Character formats: %10c %5hc\n", ch, ch);
      fprintf(fout, "Character formats: %10c %5hc\n", ch, ch);
      // 打印字符串
      printf("String formats: %25s %25.4hs\n", string, string);
      fprintf(fout, "String formats: %25s %25.4hs\n", string, string);
      // 打印浮点数
      printf( "Real number formats: %f %.2f %e %E\n", x, x, x, x );
      fprintf(fout, "Real number formats: %f %.2f %e %E\n", x, x, x, x );
};
```

上述函数执行后，在屏幕上和文件里会显示如下结果：

```
Integer formats: Decimal: -9234    Justified: -009234
Character  formats:        h        h
String formats:     computer         comp
Real  number  formats: 251.736600   251.74   2.517366e+002 2.517366E+002
```

数据文件可用任何文本编辑程序生成，如写字板。在 VC++**集成环境中，也可创建和编辑数据文件。具体方法与创建源代码文件类似：打开菜单【**File**】下的【**new**】选项，点击窗口中【**Files**】的标签，然后在其下方窗口中将文件类型定为【**text File**】（最后一个选项）。与创建源代码文件类似，只需给文件名即可，系统会自动加上** txt **的后缀名。输入完毕不要忘记保存文件（不必关闭）。**

（4**）字符读取函数**

```
int fgetc(FILE *fp)；          // 从文本文件里读取一个字符
int getc(FILE *fp)；           // 从文本文件里读取一个字符
int getchar()；                // 从键盘上取一个字符
```

若执行成功，返回从文件读入的一个字符；若返回 EOF**（文件尾记号），则表示执行失败或已经读到文件结束处。**

例 6-5 设计了一个函数 ReadLine（），它既可用于从屏幕读入一行字符串，又可用于从文件中读入一行字符，函数返回值为读入的字符串的长度。

例 6-5：

```
#include <stdio.h>
// 用指针把字符串变量 s 输入给函数，字符串的长度 length 也必须同时输入函数
// 以避免使用字符串时，发生下标值变量越界的错误
int    ReadLine（FILE* fp, char* s, int length)
{
      int   j, ch;
      for(  j = 0; (j < length - 1) && ((ch = getc(fp)) != EOF) && (ch != '\n'); j++ )
            s[j] = char (ch);                    //（注 1、注 2）
// 必须在字符串最后放上结尾符
       s[j] = '\0';
       return j;
};
```

程序注释：

注 1　注意这里必须有 3 个结束循环的条件：字符个数不能超过字符串定义的长度；遇到字符串结束符'\0'；遇到文件结束符 EOF。如果少一个条件，程序便会在运行时出错。

注 2　由于读入函数的返回值是整数，所以应该用类型转换将其转换为字符型变量，否则得到的将是字符的 ASCII 代码。

例 6-6 综合应用了例 6-5 的函数 ReadLine（）和例 6-3 中的函数 IsValidDigit（）来检查用户输入数据是否合法的数字。

例 6-6：（可从配盘中下载本范例的项目文件）

```
const   int   strLength  =  256;
// 被调用函数的声明
int         IsValidDigit(char*);
int         ReadLine(FILE*,  char*,  int);
void       CheckInput()
{
        char      fileName[strLength],  buffer[strLength];
        FILE*   fp;
        printf（"Input a data file name: "）;
       //  检查读入字符串长度是否为 0，然后再执行打开文件的操作
      if ( ReadLine(stdin,  fileName,  strLength) > 0)   //  注 1
            //  对文件打开与否进行检查，如打开失败，则停止执行后续语句
           if ( (fp  =  fopen(fileName, "r" )) != NULL ) {
                 // 从打开文件中读入一行字符串
                ReadLine(fp,  buffer,  strLength);
```

```
            // 检查读入的字符串中是否有非数字字符
            if (!IsValidDigit(buffer))
                printf ("There is a non-digit character in input %s\n", buffer);
            fclose( fp );
        }
        else
            printf ("File %s open error, check if the file exists.\n", fileName);
};
```

程序注释：

注 1　**表示键盘输入的文件指针为** stdin，**表示屏幕输出的文件指针为** stdout。

规范化编程实践–7：**在输入数据时应提示用户。**
规范化编程实践–8：**在输出数据时应说明。**
规范化编程实践–9：**对所有可能出错的操作，应进行检查其运行结果后再执行后续程序。**
规范化编程实践–10：**对操作失误，应尽量给出明确的诊断信息。**

（5**）按格式读入数据的函数**

```
int fscanf( FILE *fp, const char *format [,argument ]... );        // 从文件中读取
int scanf( const char *format [,argument ]... );                   // 从键盘上读取
```

其中：

- *fp **是指向打开文件的指针。**
- *format **是数据格式说明，其表示方法与打印格式的表示方法相同。**
- argument **是变量的存放位置，注意变量数据类型与其格式说明必须一一对应，否则会引起程序运行错误。输入语句将输入值直接发送到变量的内存位置中，所以在调用时，对于整数、浮点数和字符变量，应采用取地址运算符&的方法获得其地址，才能用在打印语句中；而字符串变量本身的值就是地址，所以可直接用在打印语句中（参见** 6.4.10 **节）。**
- **函数返回值表示成功读入的变量个数，**0 **表示没有读入任何值，**EOF **表示读入错误或遇到文件结束。**

例 6-7 **中的函数先向一个文件里写入字符串、整数等各种类型的数值，然后按给定格式读出这些数据，打印在屏幕上。**

例 6-7：

```
#include <stdio.h>
const int strLength = 256;
void    UseInputFormat()
{
    FILE *fp;
    int k;
    float x;
    char fileName[strLength], buffer[strLength]；
```

```
    char c;
    // 在屏幕上打印一个提示语句，要求用户输入文件名
    printf( "Enter the name of output file: " );
    scanf("%s", fileName);
    fp = fopen(fileName, "w+" );
    if( fp == NULL ) {
        printf( "The file %s open error\n" , fileName);
        return ;
    }
    // 向文件中写入各种格式的数据
    fprintf( fp, "%s   %d   %f   %c", "a-string", 6517, 3.14159, 'A' );
    fclose( fp);
    // 重新打开输出文件，读取刚才写入的数据
    fp = fopen( fileName, "r" );
    //   依次读取刚才写入的数据，注 1
    fscanf( fp, "%s %d %f %c", buffer，&k, &x, &c );
    fclose( fp );
    // 将读入的数据打印在屏幕上
    printf(  "The content in file %s:\n", fileName);
    printf( "%s\n", buffer );
    printf( "%d\n", k );
    printf( "%f\n", x );
    printf( "%c\n", c );
};
```

程序执行后，会按用户输入的名字生成一个文本文件，同时在屏幕上会打印出以下内容：

```
a-string
6517
3.141590
A
```

与写入该文件里的内容完全相同。

程序注释：

注 1　**输入语句将输入值发送到变量的内存地址，所以对于这里的变量** k**（整数）、**x**（浮点数）和** c**（字符），都应采用取地址运算符**&**的方法获得其地址，才能用在调用语句中；而字符串变量** buffer **本身就是地址，所以不必再取其地址（见** 6.4.8 **节）。**

此外需要特别注意的是，程序只能按读取函数中说明的格式处理读入的数据，因此，如果输入数据的格式不准确，例如，本应读入一个整数，却读入一个字符串，程序仍然会按整数处理它。结果轻则程序计算结果不正确，严重时会使程序崩溃，甚至死机。因此，如果需要用交互式从屏幕输入整数或浮点数数据时，一般都按字符串格式读入，然后检验其格式，确保它们

都是由数字字符组成后，再用函数 atoi()或 atof()将字符串转换为所需的整数或浮点数。本章 6.4.7 节中给出了一个可供参考的对字符串进行检验的函数。

除了以文本格式输入、输出数据以外，C++语言还可采用二进制的格式输入和输出数据，有兴趣学习的学员可参看有关的语法规定。

导读：

C++还提供了另一种以 stream（流）方式输入、输出数据的函数。但是，它们对数据格式的说明没有这样直观，格式控制也没有这样方便，这里不再介绍，有兴趣的学员可参看有关 C++语法的专门论著。

习　题

知识点：

本章习题的目的是：

① 熟悉 VC++软件的使用方法。

② 通过编写一些程序，掌握 C++的基本词法和基本句法。

③ 通过犯错误进行学习是一种好方法，一次成功对学习编程来说并不见得是好事情。所以习题中有意安排了人为制造的一些错误，用以学习如何理解编译诊断信息并指导纠错。

④ 培养规范化作业的概念和习惯要从头开始，不要认为练习简单，不值得费劲。要严格按照教材中给出的规范化编程实践原则进行编程作业练习。

1．按照 6.2 节中给出的步骤创建 project_1，将范例程序 sample_1.cpp 输入计算机、编译并运行。为了学习使用编译器，建议学员有意在程序中制造一些错误。例如：

① 略去某个语句后面的句尾号“ ;”。

② 改变某个变量的名字，使其成为没有定义过的变量。

然后通过编译器给出的诊断信息，学会正确判定错误的位置和性质。

2．按图 6–15 的例子创建模块化的程序来创建 project_2。包括两个文件模块：main.cpp 和 module_1.cpp，然后编译、运行。为了学习模块化编程作业，建议学员有意在程序中制造一些错误。例如：

① 在主函数模块中略去函数的声明语句。

② 使函数的声明语句与定义语句不同。

③ 使函数定义语句与调用语句不同。

然后通过阅读编译器给出的编译和连编诊断信息，学会正确判定错误的位置和性质。

3．创建模块化的程序 project_3，分别用 6.4.10 节中的输出输入语句的函数例子编写几个子函数放在一个模块文件里，另外编写一个主函数模块调用它们。

4．创建 project_4，用子函数实现以下功能：

① 从数据文件将若干整数读入一个数组中。教材提供了两个数据文件，一个包括若干整数，另一个包括若干浮点数，可从书后配盘下载。

② 分别定义 3 个函数，各用一种循环语句求上述数组中数的平均值。

思考：能否将许多整数直接相加求平均值？若不能，用什么方法可求出其平均值？（提示：整数直接相加可能超出系统中所能表示的最大整数值。）

③ 分别用两种方法确定数组的大小，从中理解数组和指针的区别。

- 定义一个长度足够大的静态数组。
- 用指针生成长度合适的动态数组。（提示：可设计一个函数，利用文件结束符号 EOF 作为判断循环结束的条件先数出文件中数据的个数，将其返回给外部程序生成一个动态数组后，再调用①中的函数，将数据读到数组中。）

5．在上题基础上，增加两个求数据最大值、最小值的函数，将结果打印到一个文件里。

6．6.4.7 节中给出的检查字符串的函数 IsValidDigit()并不完善，根据下面列出的检测算法改进该函数，准备在后续课程的练习中使用。

① 首先检测字符串首是否有空格，若有则跳过空格，直到遇到第一个非空格字符为止。

② 第一个非空格字符只能是数字、负号、或小数点。

③ 若第一个非空格字符是数字和负号，则其余的字符只能是数字或者小数点，且小数点只能出现一次。

④ 若第一个非空格字符是小数点，则其余字符只能是数字。

要求学员写出完整的函数，并用各种情况反复进行测试。

第 7 章　用类编写面向对象的程序

本章导读：

C++语言能以面向对象方式表达高层次抽象概念，因此应用 C++的类概念编程不是一个简单的语法规则问题，而是根本改变设计观念的问题。本章一方面从人类抽象思维方式的高度，另一方面用实际案例（case）论述如何应用类编写面向对象程序的问题，目的是使学员掌握面向对象程序设计的理念、方法和过程。

学习 C++语言的基本语法后，即可编写一般程序了，但那样编出来的程序并不是面向对象的，与普通编程语言 BASIC、C 等所编写的程序没有本质上的区别。普通编程语言能直接处理的仅有 3 种最基本的数据类型：整数、实数和字符，数组也仅是单一类型数据的集合，而数学和工程学科里涉及的数据要复杂得多。C++中的类能用数据的运算来定义数据，将复杂的数据组织为一个逻辑整体。这使得程序可对数据按人们所认识的数据的本来面貌去分析和处理。因此，理解和应用类的概念是学习 C++语言的关键。

7.1　C++语言中类的概念

7.1.1　概述

在初等数学里，数是用其形态来定义的，如自然数、整数、分数等用其直观表示形态来定义的；而从数学理论的角度来看，“数”是用其运算法则定义的，具体表示形态并不是其本质。如复数可有多种表示形态：实部虚部法、三角函数法、指数法，但无论如何表示，都能按定义的复数运算规则进行加、减、乘、除、乘方等运算。因此，对于复数，运算比表示形态更本质。前几章对离散数学的介绍中也反复强调：学习离散型数据，最重要的是掌握其运算法则。

用运算法则而不是形态来定义数是面向对象方法程序设计的一个基本原则，它通过以下两方面来定义数据类型：

- 能施行于该数据上的运算集
- 数据值的允许集

C++语言中的类提供了一种实现上述概念的机制。类的作用是组织有关数据成为一个表达高层次抽象概念的逻辑单元，在程序中相当于一个数据类型。下面的案例 Case-1 定义了一个数学意义上的复数，教材将围绕该案例系统介绍如何定义类，如何具体编写其程序，以及如何应用类的概念设计 C++程序。对该案例的讨论将贯穿若干章节，这里是其第一部分，复数类 Complex 的基本定义。

教材编排说明：

对学习编程技术来说，最重要的是掌握如何将程序语句组织成为内在逻辑严密的程序。因此，从本章起教材将采用基于案例的方法介绍 C++程序设计和编程作业的过程。本教材中一般的例子只是为了说明一个概念，对它的讨论一般只限于其所在小节。而 case 一般多为一个完整的程序，包括的内容要比一般例子丰富的多，因此对 case 的讨论也会贯穿若干章节。如果对一个 case 的讲解贯穿多个小节，教材将用 Case-1-1、Case-1-2 这样的编号表明所讨论的问题是同一个案例的延续。

Case-1-1：这是关于复数类案例的系列讨论的中的第一部分，Complex 类的定义语句。

```
// complex.h 用 C++的 class 定义的复数
class  Complex  {
public:
// 对数据成员赋初值的构造函数
     Complex();                              // 默认构造函数
     Complex(float, float);
// 获取数据成员值的函数
     float   Real()   { return real; };      // 获取复数的实部
     float   Image()    { return image; };   // 获取复数的虚部
     float   Module();                       // 获取复数的模
     float   Angle();                        // 获取复数的角度
// 修改数据成员值的函数
     void    AssignReal(float)；             // 给复数实部赋值
     void    AssignImage(float)；            // 给复数虚部赋值
// 功能性函数
     Complex   operator  +(Complex);         // 复数的加运算
     Complex   operator  -(Complex);         // 复数的减运算
     Complex   operator  *(Complex);         // 复数的乘运算
private:
     // 类的数据成员
     float  real;                            // 复数的实部
     float  image;                           // 复数的虚部
};
```

下面是应用以上定义进行运算的程序例句：

```
float   r,  m;
Complex a(2.5,7.9), b(-2.3,4.5), c, d;

c = b + a;                    // 复数相加
d = a - b;                    // 复数相减
```

```
r = a.Real();                    // 获取复数的实部
b.AssignImage(3.91);             // 对复数的虚部重新赋值
m = c.Module();                  // 获取复数的模
```

该类定义比较复杂，这里暂不作详细解释，后续教材会完整地介绍如何具体对其编程并在程序中应用。

本章一开始就给出这个例子的目的是要说明：

① 面向对象程序设计的核心思想是按人类所认识的事物的本来面貌去处理它。例如，人们对复数的认识是它有多种表示形态，能进行加减等数学运算，类做到了这些。

② 面向对象的概念要求以同样的方法处理概念上相同的事物。既然复数和整数、浮点数一样可进行加减等运算，就应该能用同样的运算符进行复数的运算。类也做到了这一点，它使所定义的复数运算形式与它的数学概念完全对应。

③ 由类定义的“数”的特性完全由其运算集所描述。例如，Complex 类在内部数据并不重复的情形下，可方便地提供复数的另一种表示形态——模和角度（为简单起见，这里没有提供复数的第 3 种表示法——指数表示法，但要增加是很容易的）。

为了进一步说明类所定义的“数”的特性完全由其运算所决定，而不是由内部的数据形态所决定的，下面的例 7-1 给出了一个二维矢量的类定义：

例 7-1：

```
// 用 C++的类定义平面解析几何二维矢量
class   Vector2D  {
public:
// 给数据成员赋初值的构造函数
      Vector2D()    { };
      Vector2D(float,  float);
// 获取数据成员值的函数
      float    X()    { return x; };          // 获取矢量的 x 分量
      float    Y()    { return y; };          // 获取矢量的 y 分量
// 修改数据成员值的函数
      void    AssignX(float)；                // 给 x 分量赋值
      void    AssignY(float)；                // 给 y 分量赋值
// 类的功能性函数
     Vector2D&  operator  +(Vector2D);    // 矢量的加运算
     Vector2D&  operator  -(Vector2D);    // 矢量的减运算
      float          operator*  (Vector2D);    // 矢量点乘得到一个普通数
private:
      // 类的数据成员
      float  x;        // x 分量
      float  y;        // y 分量
};
```

虽然例 7-1 中 Vector 类与 Complex 类一样，也包括两个类型为 float 的数据成员，但因其运算集不同，所以表达的是与复数概念完全不同的二维矢量的数学概念。

从下一节开始，教材将系统地讨论如何定义类，如何具体编写类的程序等问题。

7.1.2 类定义

class 定义语句的一般格式是：

```
class identifier /* 类名 */ {
public:
    member_functions; // 成员函数
private：
    data_members;   // 数据成员
    member_functions;   // 成员函数
protected：
    data_members;   // 数据成员
    member_functions;   // 成员函数
};
```

其中：

- class、public、private、protected 是关键词。
- identifier 是编程员定义的 class 的名字。
- data_members 部分是数据成员的定义。数据成员描述了类定义所表达的事物或者概念的固有性质，private 部分为私有数据成员，只能参与类自己函数的运算；在 protected 部分为保护数据成员，除了参与类自己函数的运算外，还可参与其子类函数的运算（关于子类的概念将在 7.2.3 节中讨论）。定义数据成员的语句格式与定义普通数据变量的一样（见 6.4.1 节）。数据成员可以是系统固有的数据类型、编程员定义的数据类型或其他已定义类的实例。
- member_functions 部分一般为成员函数的声明语句，在 public 部分中的称为公有成员函数，可被任何外部程序调用；在 private 部分中的称为私有成员函数，只能被类自己的成员函数调用；在 protected 部分中的称为保护成员函数，除了能被类自己的成员函数调用外，还能被其子类的函数所调用。成员函数声明语句的格式与普通函数的声明语句一样，只包括返回数据类型、函数名、参数表，不包括函数体。成员函数的定义语句，即其具体的函数程序一般都放在另外的文件模块里（详见 7.2.4 节）。

> **导读：**
>
> 为便于理解，这里对类定义语句格式的描述并未严格按照 C++语法中的形式。关于类定义更严密的形式语法描述，可参看书后配盘。

上面介绍的是类定义语句的一般格式，并不是所有的类的定义中都包括所有这些成分。许多定义只有 public 和 private 部分，没有 protected 的成分；成员函数只有 public 是必须的，许多类没有 protected 的数据成员或者成员函数；有些类甚至连 private 的数据成员都没有。

类具有面向对象的封装性（encapsulation），外部数据只能通过类提供的公有成员函数完成

有关运算，不需要也不能够直接对类内部的私有成员进行操作。因此，类至少应包括以下几类成员函数：

- Constructor 和 destructor，给数据成员赋初值的构造函数和释放动态实例占用内存空间的析构函数。
- 获取数据成员值的函数。
- 给数据成员赋值的函数。
- 类的功能性函数，这些函数描述类与其他事物的相互关系、相互作用。在 9.3.4 节将详细讨论如何应用系统分析和数据建模方法设计这些功能性函数的问题。

虽然 C++语法并不禁止 public 的数据成员，许多教材中也用 public 的数据成员来编程，但严格说来，这是违背面向对象原则的。因此，本教材将不定义 public 的数据成员列为规范化编程实践必须遵守的准则。

规范化编程实践–11：在类中不应定义公有数据成员。

在类定义中，一般只有对成员函数的声明，并不包括其函数体，因此需要另外的成员函数定义语句来说明函数内部的具体程序代码。对于一些比较简单的成员函数，也可在类定义中直接包括其程序代码，即将其函数体也包括在类的定义中（参见 Case–1-1）。

7.1.3 构造函数和析构函数

类相当于一个数据类型，用它生成的实际数据称为实例（instance）。由于类中往往包括多个数据成员，类的构造函数（constructor）的作用就是对类的数据成员赋初值生成类的实例。定义构造函数的规则是：

- 构造函数的名字必须与类的名字相同。
- 构造函数没有返回类型，不需要返回值。
- 根据需要，一个类可以定义名字相同，但调用参数不同的多个构造函数，分别以不同方法生成类的实例。

因为类定义中一般只包括成员函数的声明，因此还需另外给出成员函数的定义语句。Case-1-2 给出了 Complex 类的构造函数的定义语句：

Case-1-2：这是关于复数类案例的系列讨论的中的第二部分，Comple 类的构造函数定义。

```
// Complex 的默认构造函数
Complex:: Complex()
{
    real = 0; image = 0;
}
// Complex 的构造函数
Complex:: Complex(float x, float y): real(x)
{
     image = y;
}
```

由上述例句可知：

① 成员函数定义语句格式与普通函数定义语句格式大致相同，但应在句首加上类的名字和符号“ ::”。

② 构造函数有两种方法对类的数据成员进行赋值：

- 在函数定义行中，在函数名后以 data_name(data_value)的形式赋值，如语句：

Complex（float x, float y）:real（x）的作用就是将输入参数 x 的值赋予数据成员 real。

- 在函数体中用运算式予以赋值，如上述函数体中的 image = y；

为方便起见，类一般可有若干个构造函数，分别用不同方法生成实例。但是，以下两种构造函数是每个类都必不可少的：

① 默认构造函数（default constructor），默认构造函数即无任何调用参数的构造函数。其作用是为类的实例预定相应的内存空间。一般在默认构造函数中都会进行一些对数据成员设初值的工作。如果没有必要进行任何工作，则可以使函数体为空，即只有两个括号，其中没有语句。

② 拷贝构造函数或复制构造函数（copy constructor），作用是生成一个与已知实例完全相同的新实例，使程序可用赋值语句简单地将一个实例的值赋予另一个实例。如果类的数据成员中没有动态变量，系统一般会自动生成这个构造函数。但是，如果有动态变量，则编程员必须自己定义它，在拷贝构造函数中逐个对数据成员进行复制（具体例子参见 8.1.4 节）。

经验与提醒：

若类的数据成员中有字符串变量，尽量用长度固定的静态数组，不要用 pointer 类型的动态数组，可避免自行定义析构函数、赋值函数等许多麻烦。

在类中不仅可有同名的构造函数，而且其他成员函数也可同名，成员函数还可以和系统运算符如+、–等同名。这种现象称为“成员函数的重载”，将在 7.1.6 节中详细论述。

与其他数据类型一样，类的实例可以是静态变量，但在实际应用中以动态变量为多。因为：

- 类的实例数量变化大，往往难以预先确定
- 类的实例是有生命周期的，并不是所有实例都是一次生成的，程序根据需要在运行过程生成一些实例的同时又会消除一些实例。

与 C++的其他数据类型一样，类的动态实例也是用运算符 new 生成，但其实质是调用类的构造函数。动态实例所占内存空间也用运算符 delete 释放，但其实质是调用类的析构函数。下面例句说明了如何生成类的动态实例：

```
Complex      a(2.5,7.9), b(-2.3,4.5)；// 生成有实际值的实例
Complex       c, d;                   // 只生成实例，但没有数据
Complex*      p;                      // 先定义一个动态实例的指针
p = new Complex [2];                  // 生成关于复数实例的动态数组，长度为 2
p[0] = a;                             // 给数组中的元素赋值
p[1] = b;
```

类的析构函数 destructor 的名字也必须与类名称一样，但在前面加了一个运算符~。析构函数没有返回值，也没有任何调用参数。调用析构函数是通过应用运算符 delete 来实现的，如下

面例句所示。释放了内存的动态数组还可以再次被分配新的长度。

```
delete p; // 释放动态数组的内存空间
p = new Complex [5]; // 重新生成关于复数实例的动态数组，长度为 5
```

如果类中无动态数据成员，编程员无需自己定义析构函数，编译器一般会自动为其生成默认的析构函数。但是，如果类中包括动态变量，就需要编程员自己定义析构函数，逐个释放指针变量所占据的内存空间（参见 8.1.4 节）。

7.1.4 成员函数的定义与调用

与构造函数一样，成员函数在类定义里只是声明语句，因此还需提供包括具体程序语句的成员函数的定义语句。

与构造函数一样，若成员函数的函数体非常简单，也可直接将其定义而不是将声明放在类的定义中，这种形式也称为行内函数（inline function）。例如，Case-1-1 的 Complex 类定义中，就包括了成员函数 Real（）和 Image（）的函数体，因此不必再另外提供其定义。

成员函数定义语句格式与一般函数的定义语句格式相似，以函数返回值开头，后面是函数名，函数名后是参数表和函数体。不同的是，要在函数名之前应加上其所属的类名称以及符号“::”，Case-1-3 详细给出了 Complex 类中各成员函数的定义：

Case-1-3：这是关于复数类案例的系列讨论的中的第 3 部分，Complex 类的成员函数定义。

```
// 求复数模时所用的开方函数 sqrt（）的定义在系统头文件 math.h 中
#include <math.h>
// 获取复数的模
float Complex:: Module()
{
      return  sqrt(r*r + i*i) ;
};
// 获取复数的角度
float Complex:: Angle()
{
      return  atan(i/r);
}
// 给复数实部赋值
void   AssignReal(float  r)
{
     real  =  r;
};
// 给复数虚部赋值
void   AssignImage(float  i)
{
      image  =i;
```

```
};
// 复数的加运算，实部、虚部分别相加
Complex    Complex:: operator +(Complex c)
{
    real = real + c.real;   // 成员函数可以直接引用自己类的实例的数据成员
    image = image + c.image;
    return *this; // 注 1
};
// 复数的减运算，实部、虚部分别相减
Complex    Complex:: operator -(Complex c);
{
    real = real - c.real;
     image = image - c.image;
    return *this; // 注 1
};
// 复数的乘运算
// 复数乘法公式（a + i b）*（c + i d）= (ac – bd) + i (bc + ad)
Complex    Complex:: operator *(Complex c);
{
     return Complex(real*c.real – image*c.image, image*c.real + real*c.image);
};
```

程序注释：

注 1 this **是** C++**中专门用于指代实例本身的指针变量，这里用取值运算符*****得到该实例的值作为函数返回值（参见** 6.4.7 **节关于取值运算符的有关内容）。**

（1**）成员函数引用另一个同类实例的数据成员的格式（有两种）**

① 当实例为静态变量时，用“ 实例名.数据成员名 ”的形式。

② 当实例为指针变量时，用“ 实例名->数据成员名 ”的形式。

但是，只有类自己的成员函数才能引用其 private **的数据成员，只有子类的成员函数才能引用其父类** protected **的数据成员。**

（2**）成员函数调用语句的格式（有两种）**

① 当实例为静态变量时，用“ 实例名.函数名 ”的形式。

② 当实例为指针变量时，用“ 实例名->函数名 ”的形式。

（3**）类的成员函数的在程序中被调用方式（与普通函数一样，也有两种）**

① 作为一个数据类型为返回值的变量用在运算式中。

② 作为一个独立的程序语句。

以下例句说明了成员函数的调用格式和调用形式：

```
float     r, m;
Complex a(2.3, 0.5);
```

```
r = a.Real();                    // 获取实例 a 的实部
m = a.Module();                  // 获取实例 a 的模
a.AssignImage(0.25);             // 重新给复数虚部赋值
Complex* y;                      // 定义 y 为一个动态实例
y = new Complex;                 // 为 y 分配空间
if (y != NULL) {
      y->AssignReal(5.7);        // 对 y 的实部赋值
      y->AssignImage(7.31);      // 对 y 的虚部赋值
};
```

经验与提醒：

使用实例的指针变量前，一定要检查其是否为空指针，以免程序运行中出现不测。

7.1.5 引用数据类型和左值成员函数

引用（reference）类型是一种导出类型（关于数据类型分类参见 6.4.2 节），它对定义类的成员函数有重要意义，主要体现在以下方面：

① 函数的返回值一般是函数与外部程序的输出接口，返回的数值只能出不能进。因此，一般函数调用时只能出现在赋值运算符 = 的右边，这类函数称为“右值函数”。

根据类的封装性原则，外部程序不能对类的数据成员直接进行运算，因此类一般需要两类成员函数，一类专门用于获取数据成员的值，如 Complex 类中的成员函数 Real()和 Image()；另一类则专门用于将值赋给数据成员，如 Complex 类中的成员函数 AssignReal()和 AssignImage()。

若将成员函数的返回类型定义为引用类型，则函数调用既可出现在赋值运算符的右边，又能放在赋值运算符 = 的左面，即通过运算式直接对数据成员进行运算。这类函数称为“左值函数”。左值函数既不违背面向对象封装性的原则（即一切运算通过成员函数进行），又能方便地用普通运算式而不是函数对类的数据成员进行运算，对于那些用于需要进行数学运算的类来说将是很大的方便。下述 Case–1-4 中的 Comple 类中就采用了引用类型的左值成员函数：

Case-1-4：这是关于复数类案例的系列讲解的第 4 部分，左值成员函数的应用。

```
class    Complex {
public:
// 对数据成员赋初值的构造函数
      Complex()   { };
     Complex(float,  float);
// 兼有获取值和赋予值功能的左值函数
      float&    Real()   { return real; };       // 获取复数的实部
      float&    Image()     { return image; };// 获取复数的虚部
// 获取数据成员值的函数
     float      Module();                        // 获取复数的模
```

```
        float     Angle();                    // 获取复数的角度
    // 其他运算函数
        Complex   operator +(Complex);        // 复数的加运算
        Complex   operator -(Complex);        // 复数的减运算
        Complex   operator *(Complex);        // 复数的乘运算
    private:
    // 类的数据成员
        float real;                           // 复数的实部
        float image;                          // 复数的虚部
    };
```

上述定义表明，引用类型的定义格式是在数据类型名后加上符号&。任何数据类型，包括 C++语言的 3 种基本类型和用户自己定义的 class 都能成为引用类型的数据。

Case-1-4 中 Complex 类的改进定义中，成员函数 Real()和 Image()均为左值函数，既能用于获取数据成员的值，又能通过它直接对数据成员赋值。因此，原先专门用于对数据成员赋值的成员函数 AssignReal()和 AssignImage（）就省略了。

下面就是应用以上左值成员函数进行运算的程序语句示例：

```
Complex   a(2.5,7.9), c;
c. Real() = a.Real();      // 使 c 的实部直接等于 a 的实部
a. Image() = 4.38;         // 重新对 a 的虚部赋值
```

但是需要注意的是，被定义为左值函数的只能是那些直接将数据成员作为返回数据的函数，如 Complex 类中的函数 Module()就不能被定义为左值函数，因为它返回的是一个计算结果。

② 在调用 C++函数时，函数会首先对输入参数进行拷贝，然后用其副本在函数内进行运算。这样可避免调用函数改变输入函数参数本身的值，而函数一般也只能用其返回值作为对外部程序惟一的输出接口。

若需要返回的值不只一个，一般做法是将有关数据组合为一个能表达更高层次概念的类作为一个有机整体返回。但是，若需要返回外部程序的数据不止一个，它们之间又没有任何有意义的联系，通过引用类型的参数返回数据就是一种变通的方法。例 7-2 说明了如何用引用类型做函数调用参数，以便在函数中改变其值。这是一个很有用的函数，在后续章节里还会多次用到它。

例 7-2：用引用类型对换两个参数的值。

```
void Swap(float& a, float& b)
{
      float    c;
    c = a; a = b; b = c;
};
```

该程序执行的结果将使变量 a 和 b 的值对换。

③ 由于引用类型的参数在函数内部是不制作副本直接使用的，因此还有另一个用途。当函数的参数本身结构复杂、占据内存空间很大时（如一个内部结构很复杂的类），采用引用类型

的参数，函数可直接利用参数的正本进行运算，而不必再在函数体内制作其副本，从而提高了程序运行效率。

导读：

① 关于引用类型，本教材主要从它在程序中作用的角度加以论述，与一般关于 C++语言教科书的关注角度不同。教学实践证明，这样的论述更容易理解，对在程序中正确应用更有指导意义。

② C++像 C 语言一样，当函数调用参数类型为指针变量时，其值也可在函数体内被改变。但是，引用类型变量用点运算符“.”而不是运算符“->”引用成员函数，程序形式更为简单和直观，可读性更强。所以本教材建议，只有数组才使用指针变量作参数，而对类的实例如果欲在函数体内改变其值，则一律用引用类型作为函数调用参数。

7.1.6 成员函数的重载与运算符成员函数

类中同一名称的成员函数可有多个，这种现象称为成员函数的重载。

（1）重载的 3 种情况

① 同一个类中成员函数互相重载，前提是这些函数的参数个数和类型必须有所相同，并足以使其互相区别。例如，Complex 类中有两个构造函数，一个无参数，一个有参数，因此可相互区分。更一般的情况是，成员函数 A(float)和成员函数 A(int)为参数个数相同而类型不同；成员函数 B(int)和成员函数 B(int, int)为参数个数不同，都是合法的重载函数。

② 子类中的函数还可以重载父类中的函数，详见 7.2.3 节关于类的继承性的论述。

③ 类的成员函数还可以重载 C++语言的许多基本运算符，例如 Complex 就重载了运算符“+”和“–”，这类成员函数称为运算符函数或运算符重载函数。

运算符函数是 C++程序的重要特点，它使程序可直接应用运算符对类的实例进行对应的数学运算，使程序语句与其所表达的数学概念看起来非常一致。下面例句中就直接用运算符+、–对 Complex 类的实例进行对应的数学运算：

```
Complex a(2.5,7.9), b(-2.3,4.5), c, d;
c = b + a;  // 两复数直接相加
d = a - b;    // 两复数直接相减
```

实际上，几乎所有的 C++运算符都允许被重载，表 7-1 为编程中经常重载的一些算术和逻辑运算符（关于完整的可重载运算符介绍可参看 C++的专门教材）。

表 7-1 C++语言中可被重载的运算符

Operator	名 称	类 型
delete	delete	–
new	New	–
!=	Inequality	Binary
*	Multiplication	Binary
+	Addition	Binary
-	Subtraction	Binary
/	Division	Binary

续表

Operator	名 称	类 型
<	Less than	Binary
<=	Less than or equal to	Binary
=	Assignment	Binary
==	Equality	Binary
>	Greater than	Binary
>=	Greater than or equal to	Binary
[ ]	Array subscript	–

（2）关于运算符重载的几点说明

① 运算符函数的具体程序是根据类的数据成员和程序要求而确定的。如，复数的加运算函数就是按复数运算的数学概念定义的。

② 运算符 delete **和** new **的重载实际上是用构造函数和析构函数来实现的，与一般运算符的重载定义格式不同。**

需要注意的是，因为实例对运算符函数都是以运算式形式调用的，因此只有其值不是地址的类的实例才能出现在运算式中；对于类型为指针变量的类的实例，必须以取值运算符*加在实例名称前的形式出现在运算式中，如下面例句所示：

```
Complex  a(2.3, 5.1), b, * c;
c = new Complex(3.24, 1.56)；
b = a + *c;
```

导读：

在 C++**语法中还有其他两种以** friend **方式重载运算符的方法，但严格说来，它们都不完全符合面向对象封装性的概念，所以这里不再详述。有兴趣的学员可参看有关** C++**语言的专门教材。**

7.1.7 案例——用类的运算符函数解线性方程组

在可以重载的系统运算符中，下标运算符[]的重载无疑最有应用价值，特别是对于解决数学问题。Case-2**以求解线性方程组为例说明了运算符函数对**C++**程序解决数学问题的重要意义。**

Case -2**：编写一个应用运算符函数解线性方程组的** C++**程序。**

解：三元一次线性方程组的数学表达式为

$$\begin{cases} a_{00}x_0+a_{10}x_1+a_{20}x_2=b_0 \\ a_{01}x_0+a_{11}x_1+a_{21}x_2=b_1 \\ a_{02}x_0+a_{12}x_1+a_{22}x_2=b_2 \end{cases}$$

方程组的解可用克莱姆法则求得

$$x_0=\frac{\Delta_0}{\Delta},\quad x_1=\frac{\Delta_1}{\Delta},\quad x_2=\frac{\Delta_2}{\Delta}$$

其中，$\Delta=\begin{vmatrix} a_{00} & a_{10} & a_{20} \\ a_{01} & a_{11} & a_{21} \\ a_{02} & a_{12} & a_{22} \end{vmatrix}$，$\Delta_i$ 是用常数项 b_0、b_1、b_2 分别置换行列式中第 i 列的结果。即

$$\Delta_0=\begin{vmatrix} b_0 & a_{10} & a_{20} \\ b_1 & a_{11} & a_{21} \\ b_2 & a_{12} & a_{22} \end{vmatrix}，\Delta_1=\begin{vmatrix} a_{00} & b_0 & a_{20} \\ a_{01} & b_1 & a_{21} \\ a_{02} & b_2 & a_{22} \end{vmatrix}，\Delta_2=\begin{vmatrix} a_{00} & a_{10} & b_0 \\ a_{01} & a_{11} & b_1 \\ a_{02} & a_{12} & b_2 \end{vmatrix}$$

下面详细介绍解该线性方程组的 C++程序的设计方法与过程：

（1）定义一个用于表示列的 Column 类

```
class Column {
public:
    Column() { };
    float & operator[ ] (int j) { return e[j]; }; // 返回列中第 j 个元素
private:
    float   e[3];
};
```

（2）定义一个表示行列式的 Matrix 类

```
class Matrix {
public:
    Matrix() { };
    float   Delta()； // 求行列式的值
    Column&  operator[ ] (int j) { return column[j]; }；// 返回第 j 列
private:
    Column      column[3];
};
```

利用上述类定义，有：

- 方程组的系数行列式可以用 Matrix 的实例来表示，即：

```
Matrix   m；
```

对 m 应用一次下标运算符[]，可获得其中的一列，即 m[0]表示行列式中的第一列，m[1]是第二列。对它们再应用一次下标运算符[]，就可方便地获得列中的各元素，如 m[0][0]为 a_{00}、m[0][1]为 a_{01}、依此类推。

- 方程组中常数项可以用 Column 的实例表示为：

```
Column   b;
```

其中，b[0]、b[1]和 b[2]正好与方程组中的常数项 b_0、b_1、b_2 对应。

（3）定义求行列式值的函数

首先用运算符%定义一个能使数组下标值周而复始变化的函数：

```
int Increment(int i)
{
```

```
    return ( 3 + i) % 3;
}
```

然后定义求行列式的值的成员函数如下：

```
float Matrix::Delta()
{
    float d = 0, right, left;
    for (int i = 0; i < 3; i++) {
        right = 1;
        left = 1;
        for (int j = 0; j < 3; j++) {
            right = right*column[Increment(i + j)][Increment(j)]; // 注 1
            left = left*column[Increment(i + j)][2 - j]; // 注 2
        }
        d = d + right - left;
    }
    return d;
}
```

程序注释：

注 1　**根据三阶行列式值的计算的对角线法，变量** right **是行列式中从左上至右下方向元素的乘积，**$a_{00}a_{11}a_{22}$、$a_{10}a_{21}a_{02}$、$a_{20}a_{01}a_{12}$**。为了能用一个循环变量控制运算下标值的变化，所以定义了函数** Increment**，利用取余数运算符**%**使下标的变化始终不超过** 0~2 **的范围（关于取余数运算符%的性质，详见** 6.3.6 **节）。**

注 2　**根据三阶行列式值的计算的对角线法，变量** left **是行列式中从右上至左下方向元素的乘积，**$a_{20}a_{11}a_{02}$、$a_{10}a_{01}a_{22}$、$a_{00}a_{21}a_{12}$**，为了能用一个循环变量控制运算下标值的变化，所以也用函数** Increment **使下标的变化始终不超过** 0~2 **的范围。**

（4**）定义辅助函数**

辅助函数包括一个读取方程组系数值并将其赋给系数行列式和常数项列的函数，一个解方程组的函数。

```
// 读取方程组系数的函数
void   ReadData(Matrix& m, Column& b)   // 注 1
{
    float   x0, x1, x2, b;
    for (int j = 0; j < 3; j++) {
        printf("顺序输入第%d 个方程的各项系数和常数项，在数据之间留下空格\n", j);
        scanf("%f  %f %f %f", &x0, &x1, &x2, &b);
        m[0][j] = x0; m[1][j] = x1; m[2][j] = x2;  b[j] = y; // 注 2
    }
}；
```

程序注释：

注 1 这里必须引用类型变量做函数调用参数，否则在函数中对行列式的赋值不会返回到外部程序中。

注 2 因为 m[0][0]、m[0][1]、b[0]等都是引用类型的左值函数，所以可直接对它们赋值。

```
// 解方程组的函数
Column  SolveEquations(Matrix& M, Column& b)
{
    Column    x; // 将方程组的解也定义为一个 Column，整体返回
    Matrix    m; // 做中间变量
    float     d = M.Delta(); // 求原系数行列式的值
    for (int j = 0; j < 3; j++) {
        m = M; // 保留原系数行列式，下一次循环还要用
        // 用常数项元素代换系数行列式中的一列中的元素
        m[j] = b;
        x[j] = m.Delta()/d;
    }
    return x;
};
```

（5）定义主函数并用上述定义和函数组成一个完整的程序

```
void main()
{
    Matrix     M;
    Column   b, x;
    ReadData(M, b);
    // 必须先检查系数行列式的值是否为 0
    // 否则在函数 SolveEquations()中除数将为 0，引起程序运行错误
    if (M.Delta() == 0)
        printf("The equation has no solution\n");
    else {
        x = SolveEquations(M, b);
        printf("solution is  x0= %f  x1=%f  x2=%f\n", x[0], x[1] , x[2]);
    }
}；
```

（6）准备测试数据

例如，对于方程组

$$\begin{cases} x_0 + 1.2\,x_1 + 1.3\,x_2 = 1.0 \\ x_0 - 1.5\,x_1 + 3.4\,x_2 = 2.5 \\ 2.5x_0 + 3.2\,x_1 + 4.6\,x_2 = 2.0 \end{cases}$$

所需输入的数据文件内容如下（注意用空格隔开各数据）：

```
1     1.2   1.3    1.0
1   −1.5   3.4    2.5
2.5   3.2   4.6   −2.0
```

所得解为 x_0= 8.18, x_1 = −2.82, x_2= −2.92

案例 Case-2 对引用类型作函数返回值和运算符重载的意义是很好的说明。如果没有这些函数，就需要 18 个成员函数分别用于获取 9 个行列式元素的值、或者对它们赋值，程序也不可能有如此紧凑的形式，其数学意义也不可能像这样明了。

学员可采用普通数组编写一个求解上述线性方程组的程序，通过比较来体会应用类概念分析和解决问题的实质。这种比较对于那些曾学过面向过程编程语言，如 C 语言的学员来说尤其重要，它将有助于他们理解面向对象编程方法和普通编程方法之间的区别，尽快在程序设计观念上有一个根本的转变。

7.2 面向对象程序设计方法与 C++的类

7.2.1 人的抽象思维方法

使计算机按照人类思维的方式处理数据始终是计算机科学与技术发展的终极目标，C++和其他面向对象编程语言的出现朝着这个目标的实现迈进了一大步。面向对象方法的核心是按人类抽象思维的方式去描述计算对象，使计算机能按人类所认识事物的本来面貌去处理它。正如教材一开始所指出的，任何科学技术的应用都不能离开基础理论的指导作用。如果对面向对象编程方法发展哲理缺乏充分理解，而仅仅将其作为程序语句进行学习，就不能掌握其精髓。因此，在进一步讨论面向对象的程序设计思想和方法以前，首先应对人类自身抽象思维方式有所认识。

人类抽象思维的方法主要有：分类、演绎、综合、外推。

（1）分类（classification）

分类根据事物的共同点和差异点，将事物分为不同的种类，确定其从属关系。分类是逻辑方法中的“比较–分类法”的一个方面，也是自然科学研究的基本方法。运用分类的目的是使客观对象系统化，比较是分类的前提，分类是比较的结果。

（2）综合（synthesis）

综合将事物的各个方面、各个部分、各个片断有机地组织成更高层次的整体概念。分类和综合是逻辑方法中的“分析–综合法”的两个方面。分析是将整体分解成部分，将复杂事物分解为简单要素，将动态过程分解为片断。综合则将对象的各个方面、各个部分、各个片断有机地组织成更高层次的概念。

（3）演绎（deduction）

演绎用事物的已知共性推论事物的个性，是逻辑学中“归纳–演绎法”的一个方面。归纳是从特殊的、个别的事物中抽取其未知的共同属性，而演绎则是用事物的已知共性推论个别事物的未知特殊性。

（4）外推（abduction）

外推可视为演绎的逆过程，是从结论和知识得到原始事实的过程。由于其原始事实的得出在某种程度上缺乏确定性，所以也有人称之为“不确定性外推”。

上述抽象思维方法是面向对象程序设计中识别和定义类的基本指导思想。在 C++中用类描述运算对象就体现了分类的思想；类中可包括多种类型的数据，这是一种综合机制；类还可衍生出子类，从而可由事物的共性推论事物的个性，是一种演绎的机制。

导读：

① 归纳（induction）不同于综合，综合是从同一命题的各个方面中抽象出更高层次的完整概念，而归纳是从特殊到一般，是从已知论及未知。

② 外推属于人工智能研究的内容，用一般的程序设计语言是难以实现的。

7.2.2 C++的类的聚集机制

在 C++中，用类描述运算对象体现了分类的思想。类又是一种综合机制，将有关数据组合在一起表示一个新的逻辑概念。具体来说，C++语言的类的数据成员可包括：

- 各种基本数据类型的变量
- 其他类的实例
- 类自己的实例

例如，下面例 7-3 中的二叉树类 BiTree 就是一个既包括类自身的实例，又包括另一个类 NodeData 实例的聚集。

例 7-3：二叉树的类定义举例。

```
// NodeData 是关于树结点的类，为简单起见目前只有结点名字一个数据成员
// 在实际应用中可根据需要增添其他数据成员
const int nameLength = 256;
class   NodeData {
public:
    NodeData() { name = NULL; };
    NodeData(char*);
    char*   Name() { return name; };
private:
    char    name[nameLength];
}
// 二叉树类的定义
class BiTree {
public:
    BiTree();                   // 默认构造函数
    BiTree(NodeData*);          // 构造函数
    ~BiTree();                  // 析构函数
```

```
    BiTree*&    LeftNode();     // 返回左子树
    BiTree*&    RightNode();    // 返回右子树
    NodeData*   Node();         // 返回结点数据
private:
    NodeData*       node;
    BiTree  *leftNode;  *rightNode;
};
```

因为二叉树的数学定义采用了递归形式（参见 5.1.8 节），树的根结点被视为包括左子树和右子树的一个结构。因此，若将二叉树定义为一个类，应有两个数据成员是它自己的实例。因为子树可能为空，所以用指针变量比较合适。当指针变量为空值时，表明没有该分支。关于二叉树类的 C++程序还将在第 10 章中详细讨论。

7.2.3 C++的类的继承机制

C++中的类是对数据进行分类的机制，类可衍生出子类，C++语法中称导出类（derived class），从父类和子类之间的继承关系中，可由事物的共性推论事物的个性，是一种演绎的机制。子类由之衍生的类称为基类（base class），也称为父类。这就是类的继承机制，它使程序既能方便地表达事物分类之间的共性，又能方便地表达和处理事物个体之间的差异。

定义父类和子类时，应从对事物共性和个性的分析出发，采用自上而下的方式，先定义父类，然后再定义各子类。这个父类可能对应一个实际事物，但是在大多数情形下只是表达一种抽象概念。父类提供那些对所有子类来说都具有共性的数据和运算，子类则考虑那些描述其个性的数据和操作。下面的例 7-4 就是 C++类继承机制的一个例子（该例子中的有关定义还将被用在第 11 章的课程作业一中）。

例 7-4：在设计一个管理学校教师和学生的程序时，初步分析确定需要定义两类对象：教师和学生。教师类的基本数据包括姓名、性别、年龄、所在系等；学生类的基本数据包括姓名、年龄、所在班级等。显然，这两个类有一些共同特性。因此，应该把其共同的特性抽取出来，定义为一个父类。以下就是将教师和学生两类事物中共性抽取出来所定义的一个父类：

```
// information about a person
class Person {
public：
// constructors
    Person（）；
    Person（char*, char*, char）;
// 获取数据成员值的函数
    char*   ID() { return id; } ;           // 返回编号
    char*   Name() { return name; } ;       // 返回姓名
    char    Gender() { return gender; };    // 返回性别
// 修改数据成员值的函数
    void    ModifyID(char*);                // 修改编号
```

```
    void    ModifyName(char*);          // 修改姓名
    void    ModifyGender(char);         // 修改性别
protected：                             // 注 1
    char    id[20];                     // 编号，注 2
    char    name[8];                    // 姓名
    char    gender;                     // 性别，M （男）F（女）
}
```

程序注释：

注 1　**父类的数据成员的访问性一般都被定义为** protected**，以便于子类能直接访问它们（参见** 7.1.2 **节关于成员属性可访问性的论述）。**

注 2　**类中的数据成员一般都是事物的固有属性，如人员的姓名、年龄等。但是，类的数据成员的取值必须能将类的各个实例惟一确定地区分开来。虽然习惯上多以姓名来区分人员，但是众所周知，名字相同的人员还是有的。因此，学校里往往采用学号来管理学生，用职工编号来管理教师。而我国还为每个公民指定了惟一的编号，即身份证号。所以，在** Person **类中特地加入了编号这个数据成员，虽然它不是教师和学生这两类事物的固有属性。**

经验与提醒：

一般多定义字符串长度为偶数，而且尽可能地等于 2 **的倍数，以便于计算机的内存管理。上述定义中，编号为** 18 **位，加上结尾符，共需** 19 **个字节，所以长度定为** 20**；而姓名的每个汉字占两个字节，加上结尾符，共需** 7 **个字节，所以长度定为** 8**。**

注意，上述 Person **类的成员函数中目前只有获取数据成员值和修改数据成员值的成员函数，没有功能性函数，因为目前只是孤立地看待一个事物。类的功能性函数是事物之间相互关系和相互作用的抽象描述，应将类放在一个系统中进行观察，才能确定其功能并定义相应的功能性函数。在第** 9 **章介绍系统分析和数据建模方法时，还将围绕如何定义** Person **类的功能性函数的问题展开深入讨论。**

上面的 Person **类只概括了学生和教师两类事物的共性。因此，还需要用子类来分别描述学生类和教师类特有的属性，例如，学生类有班级名，教师类有系名。下面就是这两个子类的定义：**

```
// student.h
class Student: public Person {
public:
    Student(char*, char*, ch, char*);
    char* ClsName();
private:
    // 学生所在班级的名称
    char    clsName[strLength];
};
```

```
// teacher.h
class Teacher: public Person {
public:
    Teacher(char*, char*, ch, char*);
    char* DeptName( );
private:
    // 教师所属系的名称
    char    deptName[strLength];
};
```

由上述定义可知，子类定义语句的格式和一般类的定义语句格式基本相同，但是在类名后面多了其父类的名称，中间用符号“ :”隔开，父类名前还有限定词 public。

导读：

父类名前的限定词也可以为 private。限定词不同，子类对父类不同的继承机制也随之不同。本教材只考虑 public 的继承机制，关于 private 继承机制可参考有关 C++语言的专门论著。

子类和父类的继承关系体现在两个方面：

（1）在数据成员方面

子类自动继承其父类的全部数据成员。从概念上来说，Student 和 Teacher 都是 Person，因此父类 Person 中定义的关于个人信息的有关数据对它们都适用，若以 Person 为基类，各子类都可自动继承它，不必再自行定义。

子类还可有自己的数据成员，如子类 Student 增加了描述其所在班级的数据成员 clsName，子类 Teacher 则增加了描述其所属部门的数据成员 deptName。

（2）在成员函数方面

子类可用 3 种方式继承父类的公有函数：

- 直接使用父类的公有函数。父类 Person 中定义的函数 ID()、Name()等可直接被各子类的实例调用。
- 子类重载父类中同名函数。例如，由于某种需要，子类的某个函数需要执行与父类中同名函数内容略有不同的功能，则子类可重新定义在父类中已定义过的函数，同时在父类的函数前冠以限定词 virtual，意思是虚函数。
- 父类和子类的函数名字相同，但父类无法提供函数的实际程序，而由子类提供其实际内容。这类函数在父类中称为纯虚函数，必须在其声明语句后加上“ =0 ”的内容。

另外，子类还可定义自己的成员函数，如子类 Student 和 Teacher 中都定义了自己的成员函数。

导读：

① 有纯虚函数的类称为虚基类（ virtual base class），只能用于定义子类，不能生成实例。

② 子类对象可继承不只一个父类，而同一个父类也可有多个不同的子类。由于篇幅所限，教材不讨论这些较深入的议题，学员可通过 C++的专门论著了解这方面的内容。

下面是子类构造函数定义语句格式的示例，语句中必须包括对其父类构造函数的调用。因

此，在子类构造函数的参数表中，既应有给父类中数据成员赋初值的数据，还应有给自己数据成员赋初值的参数。

```
// Student 的构造函数
Student:: Student(char* id, char* name, char gender, char* clsName): Person(id, name, gender)
{
// 这里特地将参数名 clsName 和数据成员名取得相同，以便于确定它们之间的对应关系
// 同时在程序语句中用 this-> clsName 形式表示类的数据成员
// 其中 this 是专门指代类实例本身的一个指针，
而指针变量引用数据成员应采用运算符“ ->”
    strcpy(this->clsName, clsName);
};

// Teacher 的构造函数
Teacher:: Teacher(char* id, char* name, char gender, char* deptName): Person(id, name, gender)
{
// 这里特地将参数名 deptName 和数据成员名取得相同，以便于确定它们之间的对应关系
// 同时在程序语句中用 this-> deptName 形式表示类的数据成员
// 其中 this 是专门指代类实例本身的一个指针，
而指针变量引用数据成员应采用运算符“ ->”
    strcpy(this->deptName, deptName);
};
```

7.2.4 按 C++的类划分程序模块

规范化编程实践-12：按类划分程序模块。

第 6 章提到应将 C++程序模块化。模块化方便了程序的编写和修改，每个模块文件可单独修改、编译，否则一个小小的修改就需要将整个程序从头至尾编译一遍，显然是效率不高的方法。

模块化还增加了程序的可读性。分模块存放以后，在主文件中只有主函数 main()，由与其直接有关的变量和函数调用语句组成，层次较低的语句并不出现，因此整个程序的内在逻辑很清楚。7.1.7 节中 Case–2 的主函数就是一个很好的例子。

按类可很自然地将程序划分成合理的模块。这样做还有一个新的优点，就是可以方便地增加新类，以便于程序扩充。

按类对程序模块化的具体方法为：

① 将类的定义存放在 head file（头文件）里，并专门用 h 作为文件的扩展名，如 person.h 的意思就是关于 person 类的定义（有的编译器也用*.hpp 作为头文件的扩展名）。具体做法是在 VC++编译器的集成环境中，在创建新文件时选择 C/C++ Head File 条目，并以待定义的类名为文件名，如定义 Person 类的头文件就冠以 Person 的名字，系统会自动将文件的扩展名定为*.h，

并将文件存放在项目（project）的文件管理窗口中 Head Files 的文件夹中。

② 定义类的成员函数程序代码的文件名字与头文件名字相同，但采用*.cpp 的扩展名，如 person.cpp 就是 person 类成员函数的程序。具体做法是在 VC++编译器的集成环境中，在创建新文件时选择 C++ Sourse Files 条目，并以待定义的类名为文件名，系统会自动将文件的扩展名定为*.cpp，并将文件存放在 project 文件管理窗口中 Sources Files 的文件夹中。

③ 在类的 cpp 文件模块开头应采用 preprocessor macros（预处理宏指令，参见 6.4.1 节）语句#include "class_name.h"，其作用为通知编译器在编译时将对应的头文件包括到文件模块中，否则编译器会认为该文件中类的成员函数是未定义过的。

④ 整个程序将由一个主模块 main.cpp 和多个子模块 class_1.cpp，class_2.cpp，…等组成。与每个程序文件配套的各有一个头文件，class_1.h，class_2.h，…等。

⑤ 其他普通函数仍按其功能相近组合成合适的模块，若其中某个程序需要调用某个类的成员函数，则应该用 preprocessor macros（预处理宏指令，参见 6.4.1 节）语句#include "class_name.h"将定义该类的头文件包括到本程序文件中。

图 7-1 是一个包括 student 和 teacher 类的程序模块化结构示意图。如图所示，主函数模块中同时包括头文件 student.h 和 teacher.h 的语句，而它们各自又包括了头文件 person.h。这样，当编译器处理主函数模块时，就会将头文件 person.h 包括两次，从而引起编译错误（认为有同

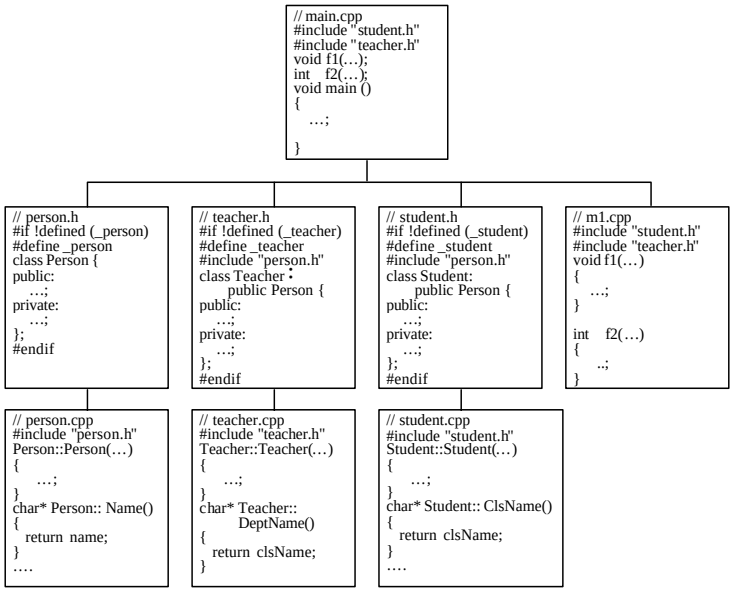


图 7-1 基于类的模块化程序

名的标识符)。为防止这种现象，类的头文件一般都应加上以下一段 preprocessor macros（预处理宏指令)。

```
// student.h     student 类的头文件
#if !defined (_student)          // 使其和类同名
#define _student
class Student {
    …;
};
#endif
```

其大概的意思为，若迄今为止还没遇到过类 Student 的定义，就给它定义一个内部标识 _student。若遇到这个标识，则说明前面的程序已处理过该头文件，不要再 include 第二次，从而避免编译器认为程序有语法错误。

VC++编译器的集成环境还为按类模块化的程序提供了方便的管理工具。图 7-2 显示了 VC++项目中管理模块文件的窗口，只要按上述方法创建类的头文件和 cpp 文件，就能自动排列在对应的文件夹下以方便编程员查看（但如果用 Windows 中的文件管理器查看，所有头文件和 cpp 文件实际上在同一个项目文件夹中，以方便编程员进行文件复制的工作)。图 7-3 显示了 VC++项目中管理类的窗口，各个类中的数据成员和成员函数一目了然，其中有加锁标志的是私有数据成员，形象地表明了外界对它们是不可访问的。

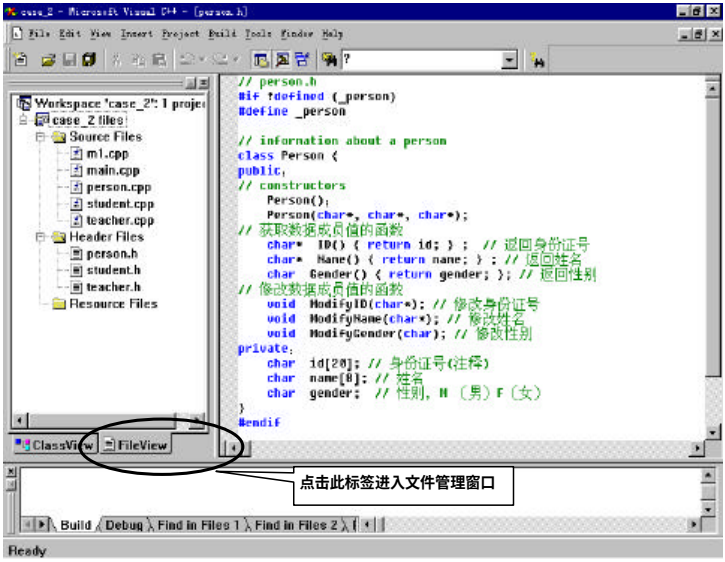


图 7-2 用 VC++编译器管理文件模块

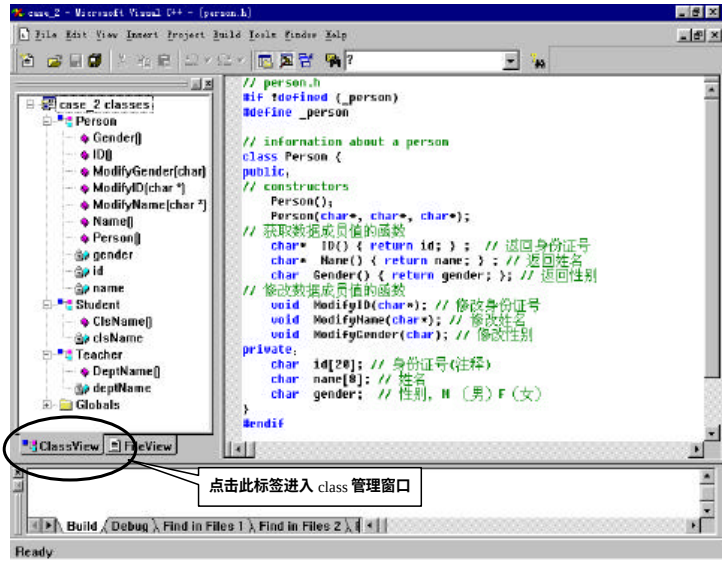


图 7-3 用 VC++编译器管理类定义

习 题

知识点：

本章提供了 Case–1 和 Case–2 两个完整程序，通过将其输入计算机、编译并运行的练习，达到以下目的：

① 进一步熟悉 VC++编译器的使用方法。

② 掌握用类进行面向对象编程的概念。

③ 在编程作业方面掌握按类模块化编程的作业方式。

1．Case–2 **完整地介绍了一个程序的设计步骤和过程，并给出了全部程序代码。学员可以创建一个项目，按** 7.2.4 **节中模块化的原则创建以下模块文件：**column.h、matrix.h、matrix.cpp、misc.cpp(**包括函数** ReadData（）**和** SolveEquations（）、main.cpp(**只包括主函数**)，**并用实际数据检验程序。**

2．**利用** Case–1 **给出的复数类定义编写一个复数计算器程序。**

教材中的 Case-1-1 **至** Case-1-4 **完整地介绍了** Complex **的定义及其过程，但是要形成一个完整程序，还需要以下辅助性的函数：**

// **主函数：用** switch **语句提供用户选择功能**

void main()

{

```
    char choice;
    while {
        PrintMenu(); // 打印屏幕指令
        scanf ("%c", & choice)；
        switch (choice) {
        case '+': Addition（）；// 进行加法运算的函数
                break;
        case '-': Substraction（）；// 进行减法运算的函数
                break;
        //  要求学员仿照上述段落写出执行乘法和其他运算的语句
            …;
        default: ;
        }
    } while (choice != '0');
}
```

下面是主函数中所调用函数的示例：

```
// 打印屏幕指令，指示用户如何选择功能
void PrintMenu()
{
    printf("input choice:\n");
    printf("0: exit\n");
    printf("+: add two complex number\n");
    printf("-: two complex number substraction\n");
    printf("*: two complex number mulitply\n");
}
// 执行加法运算的函数
void Addition（）
{
    float r1, r2, i1, i2;
    printf("input the first complex number real part and image part\n");
    scanf("%f    %f", &r1, &i1);
    printf("input the second complex number real part and image part\n");
    scanf("%f    %f", &r2, &i2);
    Complex c1(r1, i1), c2(r2, i2), c3;
    c3 = c1 + c2;
    printf("The sum complex real=%f image=%f\n", c3.Real(), c3.Image());
};
```

以下是具体作业要求：

① 利用 Case7-1 **所提供的类定义和上述函数，编写一个复数计算器，其功能包括：**

- **两个复数相加。**
- **两个复数相减。**
- **两个复数相乘。**
- **显示复数的模和角度。**

学员可仿照上述范例程序自行完成减法运算函数和乘法运算函数的编写工作。

② 注意使程序模块化，整个程序应包括以下模块文件：complex.h、complex.cpp、main.cpp、misc.cpp（包括函数 Addition（）、Substraction（）等）。

③ （选做）在上述程序中增加一个类似计算器的记忆功能，从而使程序可以做连续运算。

提示：在主函数 main（）中定义一个 Complex 类的变量实例，如 Complex result，将其初始值设为复数的 0。并将各函数的定义修改为需要调用参数，并且有返回值，如：

```
Complex  Addition（Complex）；
Complex  Substraction(Complex);
```

每当调用函数时，就将变量 result 作为函数的输入参数，参与函数中的有关运算，最后再将函数返回值赋予该变量。如下面例句所示：

```
result=Addition(result);
```

3. 类一般都包含比较复杂的数据成员，因此在应用类的实例进行复杂运算以前，首先应对类实例生成情况和数据成员取出情况进行检验。本习题的目的就是检验例 7–4 中 Student 类和 Teacher 类实例生成的情况，并通过练习理解继承机制的工作情况。具体练习包括以下内容。

① Person 类成员函数的定义：

```
// Person.cpp   person 类的成员函数定义
#include <string.h>
Person:: Person()
{
// 给字符串初始化一个结尾符，可避免出现运行错误
      id[0] = '\0';
      name[0] = '\0';
      gender = ' ';
}
```

经验与提醒：对于字符串变量，使用前一定要初始化，给它一个结尾符。

```
Person:: Person(char* a, char* n, char g): gender(g)
{
      strcpy(id, a);
      strcpy(name, n);
}

// 修改身份证号
void  Person:: ModifyID(char* newID)
{
      strcpy(id, newID);
};
// 修改姓名
void  ModifyName(char* newName)
{
      strcpy(name, newName);
};
// 修改性别
void  ModifyGender(char newGender)
```

```
{
    gender = newGender;
};
```

读者可以仿照它完成 Student **类和** Teacher **类的成员函数定义。**

② 编写辅助函数。从文件里读取数据并生成 student **实例的函数。它以** Student **类的指针作为返回数据类型，以熟悉关于动态实例的概念和具体使用方法。**

```
Student* ReadStudent(FILE* fin)
{
    Student* s = NULL;
    char     id[strLength], name[strLength], clsName[strLength];
    char     gender;
    if (fscanf(fin, "%s %s %c %s", id, name, &gender, clsName) > 0)
        s = new Student(id, name, gender, clsName);
    return s;
}
```

以下函数用于打印实例中的数据：

```
void   StudentDataIO()
{
    Student* s = NULL;
    char     fileName[256];
    FILE*    fin;
    printf("Input the student data file name : ");
    scanf("%s", fileName);
    if ((fin = fopen(fileName, "r"))==NULL) {
        printf("Input file %s open error\n", fileName);
        return;
    }
    else if ((s = ReadStudent(fin))== NULL) //  注 1
        printf("No student instance is generated\n");
    else
        printf("Student ID = %s  Name =%s Gender=%c className=%s\n",
            s->ID(), s->Name(), s->Gender(), s->ClsName()); //  注 2
    fclose(fin);
};
```

程序注释：

① 对于指针，一定要检查变量值不为空才能用它进行运算，否则会引起程序运行错误。

② 变量 s **是** Student **类的实例，它可以直接应用父类** Person **的成员函数** ID、Name（）、Gender（）**获取和父类共有的数据成员，应用自己的成员函数** ClsName（）**获取自己特有的数据成员。**

学员可以仿照上述函数编写读取和打印 Teacher **实例数据的函数。**

③ 编写一个主函数 main()**，把上述函数组合为一个可运行的程序。**

④ 程序应包括以下模块：main.cpp、person.h、student.h、teacher.h、student.cpp、person.cpp、teacher.cpp、misc.cpp。

第 8 章　用类模板实现线性数据结构

本章知识点：

① 进一步学习 C++语言，掌握类模板的概念及应用。类模板是 C++语言的精髓，熟练掌握并应用其概念对编写高质量的程序非常重要。

② 线性数据结构，特别是序列（List）是应用最多的数据结构。本章介绍两种序列的模板，已足以解决许多实际问题，应通过练习熟练掌握。

8.1　类模板

8.1.1　数据结构和离散数学

程序所处理的类的实例往往有多个，对这些类实例进行规范化管理是程序必不可少的功能，也是维持数据一致性和完整性的重要手段。虽然数组除用以组合整数、浮点数、字符变量外，也可将某个类的实例组合在一起加以管理，但用数组来管理实例很不方便。第一，使用数组时，程序必须同时考虑许多有关问题，如记住数组的长度和实际元素数量等，以免数组下标值出现错误。第二，管理实例比管理单纯的数复杂得多，如经常需要对实例按某种顺序排序，或按给定的数据成员值查找一个和多个实例等。

根据面向对象的原则，显然应将数组、数组的长度以及有关的运算组合为一个逻辑单元，专门起管理类实例的作用。起管理类实例作用的类（以下简称实例管理类）对所有的类都是需要的，若针对每个特定的类都分别定义一个实例管理类，如对 Student 类定义一个管理类，对 Teacher 类又定义一个管理类，显然不是一种有效的办法。尽管这些类管理的对象不同，但其运算具有许多共性，如均需插入和删除元素，可能都要进行排序、查找等运算。根据面向对象继承性的原理，应该用一个父类来概括其共性。这个概括管理实例功能共性的数据对象，从数学上来说就是数据结构的概念。

程序中的实例即实际事物在程序中的表示。实际事物之间的关系是复杂的，数组所表示的关系可以说是其中最简单的一种，其特点是各元素成一种互相平等的、无内在结构的线性排列，与之对应的数学概念是集合。根据离散数学中的图、树等概念，可定义更复杂的数据结构。这就是为什么说离散数学是程序编制或软件技术的数学理论基础的理由。

从理论上说，程序中的数据结构应为普遍适用的、与所需管理的特定的数据类型无关，但用普通的编程语言不能办到这一点。因此，许多教科书都采用所谓的伪语言（peudo-language，一种介于自然语言和编程语言的中间物）来描述数据结构，但是这样的描述与实际可运行的程序之间还有相当大的差别。

本教材中所有的数据结构均用类模板（class template）和运算符重载方法编写，是一种真

正与具体数据类型无关、普遍适用的数据结构，所有范例程序均可直接运行。这些范例不但有助于对数据结构理论概念和具体的程序代码的理解，而且还可直接用于解决实际问题，有重要实用价值。

本章主要介绍线性数据结构中的序列和矢量矩阵（包括），前者主要用于离散数据对象的管理，后者主要用于数值计算。为兼顾知识完整性，本章也介绍了其他的一些线性数据结构，如队列和堆栈等，但仅列为选读内容。

8.1.2 固定长度的 List 模板

序列是线性数据结构中最简单的，也是应用最广泛的。序列可进行的主要运算包括：随机访问指定位置上的元素，所谓访问就是读取其值或改变其值；在指定位置上插入或删除元素等。此外，它还可根据需要提供一些辅助性的功能，如查找、排序等。下面的序列模板一给出了 C++ 中类定义语句的格式，该模板包括了序列上的一些基本运算，如插入元素、删除元素、访问元素等。

List 模板一：以下为定义序列模板一的头文件 list_1.h **中的全部内容。**

```
// list_1.h（可从书后配盘的对应章节中下载）
template <class T, int n> class List1 {
public:
     List1(): size(0), length(n) { };
    int    Size() { return size; }; // List 中实际存储元素数量
    int    Length() { return length; }; // List 的总长度
    int    Append(T); //  在 List 尾部添加一个元素
    int    Insert(int, T); //  在设定位置插入一个元素
    int    Delete(int);   //  删除指定位置上的元素
    int    Delete(T);   //  根据元素值删除元素
    T&   operator [ ](int j) { return element[j]; }; //  访问序列中第 j 个元素
// 以下排序和检索函数将在 8.3 节中予以讨论，
    void BubbleSort(); //  关于该函数程序的讨论见 8.3.1 节例 8-3
    void Search(); //  关于该函数程序的讨论见 8.3.2 节例 8-7
    void BinaryInsert(); //  关于该函数程序的讨论见 8.3.3 节例 8-9
// 将数据成员定义为 protected，以便子类的成员函数可直接引用它们
protected:
     T     element[n];
    int    length;      // List 的总长度
    int    size;        // List 中实际存储元素个数
};
// 操作成功则返回元素的数量，不成功则返回–1
template <class T, int n>
int List1 <T, n>:: Append(T t)
```

```
{
    if (size >= n ) // 序列已满
        return -1;
    element[size++] = t;
    return size;
};
// 操作成功则返回元素的数量，不成功则返回–1
template <class T, int n>
int List1 <T, n>:: Insert(int i, T t)
{
    if (i >= n – 1 || i < 0) // 序号错误
        return –1;
    for (int j = size; j > i; j--) // 将从 j 开始至序列尾部的元素依次向后移一个
        element[j] = element[j – 1];
    element[i] = t; // 将新元素插在序号 j 处
    size++;
    return size;
};
// 根据序号删除元素
int List1 <T, n>:: Delete(int i)
{
    if (i >= n – 1 || i < 0) // 序号错误
        return – 1;
    for (int k = i; k < size – 1; k++) // 将从 j 开始至序列尾部的元素依次向前移一个
        element[k] = element[k + 1];
    size--; // 元素数量减 1
    return size;
};
// 根据元素值删除元素，注 1
template <class T, int n>
int List1 <T, n>:: Delete(T t)
{
    int deleted = 0;
    for (int i = 0; i < size; ) { // 根据元素值查找元素
        if (element[i] == t) { // 如果找到
            deleted = 1;
            for ( ; i < size – 1; i++) // 将从 j 开始至序列尾部的元素依次向前移一个
                element[i] = element[i + 1];
```

```
            }
            i++;
        }
        if (deleted)
            return --size;
        return –1;
    };
```

由 List 模板一可知，定义类模板的语句格式与普通类的定义语句格式基本相同，不同之处有以下几点：

① 定义中数据类型用符号代表。

② 句首加模板参数表，其形式为：

template <class T, int n>

模板参数一般有两种，或只有下列两种之中的任意一种：

- 数据类型说明，如定义中的 class T 表示符号 T 将来应该被一个数据类型代换，包括基本数据类型，如整数、浮点数、类等。
- 变量，如定义中的 int n 表示 n 将来应该被一个具体的整数值代换。

③ 一般的类，定义在头文件（*.h 文件）中，程序在源代码文件（*.cpp 文件）中，但是类模板的头文件中必须包括其成员函数的程序代码。（注意：这只是 VC++产品本身的缺陷，而不是模板本身的问题。其他 C++产品在采用模板时仍然可以像普通类一样，将其头文件和程序分置于不同文件模块中）。

④ 模板成员函数定义语句格式与普通类的成员函数定义语句基本相同，但在句首也要加上模板参数表。在类名后面则要用括号< >将参数包括进去，并使 template <class T, int n> 中的参数 T、n 和下面句首 List1 <T, n>中的 T、n 等参数一一对应。

下面是关于 List 模板一定义中注释的讲解：

注 1　使用该根据元素值删除元素的函数时应注意以下两个问题：

- 如果用它存储整数，则它和上面根据序号删除元素的函数无法区分，编译器会报错（见 8.1.3 节“导读”中有关解释）。
- 如果用它存储类，则在该类中必须定义运算符函数 operator==()，否则程序无法进行比较两个实例的值的运算。

> **导读：**
>
> 许多教科书将 List 称为“表”，但汉语的“表”字很容易使人联想到“表格”，一种二维的结构。本教材认为称 List 为“序列”更合适。

8.1.3 模板的实例化

类模板就像一个由符号组成的抽象的 class 定义，而模板的实例化（instantiation）就是将其中的符号代之以具体的数据类型，给其中的变量以具体数值，使之成为一个实际的类定义或一

个实际变量。

在程序中对类模板实例化的方法有以下几种：

① 直接从模板生成变量实例：

```
// 定义一个存储 10 个整数的 List 变量
List1 <int, 10> iList10;
// 定义一个存储 20 个浮点数的 List 变量
List1 <float, 20> fList20;
```

由于变量只是在其被定义的函数内有效，所以这种方法一般适用于那些一次性使用的变量。

② 用 typedef 语句由模板定义一个新数据类型，再用该数据类型定义若干变量：

```
// 定义一个新数据类型
typedef List1<float, 30> Flist30;    // 长度为 30 的浮点数 List
// 再由该数据类型定义变量
FList30   fL1, fL2;   //   fL1 和 fL2 为实际变量
```

这种方法产生一个数据类型的声明，在整个程序范围内有效，因此适合于程序中多处需要同类变量的情况。

③ 模板也可以被别的模板继承，模板也可作为别的模板的数据成员如例 8-1 所示：

例 8-1：

```
class    Vector3: public List1<float, 3> {
public:
    Vector3();
    Vector3& operator +(Vector3&);
    Vector3& operator –(Vector3&);
};
```

例 8-1 用继承模板的方法定义了一个有 3 个浮点数的矢量子类，除了继承父类模板 List1 中的成员函数外，子类里增加了进行矢量加减运算的函数。关于矢量和矩阵的详细论述见 8.2 节。

④ 模板产生的类还可再被用于对模板进行实例化。例 8-2 用上述的类 Vector3 对 List1 再次实例化，生成一个有 3 个列矢量（即有 3×3 个元素）的矩阵类：

例 8-2：

```
class    Matrix33: public List1<Vector3, 3> {
public:
    Matrix33();
    Matrix33& operator +( Matrix33&);
    Matrix33& operator –( Matrix33&);
};
```

关于继承模板和用模板作数据成员的更多例子，参见 8.2 节矩阵模板和第 10 章树和图的模板。

导读：

类模板中只有抽象的数据类型的符号，只有当它们被代之以实际的数据类型以后，编译器才会生成实际的代码。因此，随着用于实例化的数据类型不同，编译器可能产生不同的诊断信息。初学者往往会对这种现象感到困惑。例如，当用 List **模板一生成一个存储整数的序列时，编译器会认为函数** Delete(int)**和** Delete(T)**为同一函数，因为其中的** T **被整数取代以后，两个函数完全相同，违反了函数重载的规则。但是，如果用** List **模板一生成一个存储浮点数的序列，编译器就认为两个函数是不同的，为完全合法的定义。**

8.1.4 长度可自动改变的 List 模板

List **模板一的长度是固定的，但实际应用一般都希望可根据需要动态地改变序列长度。**List **模板二就是用动态数组实现的、长度可自由变化的** List**。由于** element **是动态数组，可随着元素数量变化自动改变其长度，因而再也不必顾虑长度问题，使其应用十分灵活方便。**

List 模板二：以下为定义 List **模板二的头文件** list_2.h **中的全部内容。**

```
// list_2.h    （对应光盘中 list_2.h 文件）
# include <stdlib.h> //  定义空指针值 NULL 的头文件
template <class T> class List2 {
public:
      List2();
//  因为类中包括动态变量，必须定义析构函数（见 7.1.3 节）。
       ~List2();
       int      Size() { return size; };
//  因为类中包括动态变量，必须定义运算符函数 operator=()
//  才能对实例进行赋值运算（参见 7.1.3）
      List2&      operator=(List2&);
       int          Append(T);
       int          Insert(int, T);
       int          Delete(int);
       int          Delete(T);
       T&         operator [ ](int i) { return element[i]; };
//  根据需要可增加排序和检索函数的声明
protected:
       T*           element;
      int           size;      //  元素个数
};

template <class T>
List2<T>:: List2()
```

```
{
    element = NULL;
    size = 0;
}
```

经验与提醒：

若类中包括指针变量，在构造函数中务必要给它赋值或设为空指针 NULL。

```
template <class T>
List2<T>:: ~List2()
{
    if (element)
        delete [] element;
}
// 因为类中包括动态变量，必须自定义赋值函数
template <class T>
List2<T>& List2<T>:: operator=(List2& l)
{
    size = l.size; // 使序列长度等于输入序列
    if (size = 0) // 序列长度为 0
        element = NULL;
    else { // 序列长度不为 0
        element = new T [l.size]; // 生成动态数组
        for (int j = 0; j < size; j++) // 逐个拷贝元素
            element[j] = l.element[j];
    }
    return *this;
}

template <class T>
int List2<T>:: Append(T    t)
{
    T*    tmp = new T [size + 1]; // 生成一个新的动态数组
    for (int i = 0; i < size; i++)  // 将原数组中元素拷贝到新数组中
        tmp[i] = element[i];
    tmp[i] = t;   // 使新数组最后一个元素等于插入元素
    if (element) delete [] element;   // 释放原数组的内存空间
    element = tmp;    // 使原数组等于新数组
    size++;     // 将序列长度增加 1
```

```
    return size;
};

template <class T>
int List2<T>:: Insert(int j, T  t)
{
    if (j > size)  // 如插入序号大于序列长度，附加在序列尾部
        return Append(t);
    T*  tmp = new T  [size +1]; // 生成一个新的动态数组
    for (int i = 0; i < j; i++)   // 拷贝前 j – 1 个元素
        tmp[i] = element[i];
    tmp[i] = t;  // 使第 j 个元素等于新元素
    for (   ; i < size; i++) // 拷贝其余的元素，j 的数值接续上式，不必另行赋值
        tmp[i+1] = element[i];
    if (element) delete [] element;  // 释放旧数组的空间
    element = tmp; // 使数据成员等于新数组
    size++;   // 将序列长度增加 1
    return size;
};
// 根据序号删除元素
template <class T>
int List2<T>:: Delete(int j)
{
    if (j > size || j < 0) // 序号错误
        return – 1;
// 将从第 j +1 个开始的元素依次往前移动
// 然后将序列长度减 1，多余的空间将来总有机会释放
// 所以不必像增加元素那样，生成一个新的动态数组后逐一拷贝各元素
    for (int i = j; i < size – 1; i + +)
        element[i] = element[i + 1];
    size--;   // 将序列长度减 1
    return size;
};
// 根据元素值删除元素，注 1
template <class T>
int List2<T>:: Delete(T x)
{
    int deleted = 0;
```

```
        for (int i = 0 ; i < size;  ) {
            if (element[i] == x) {
                deleted = 1;
                for (   ; i < size - 1; i++)
                    element[i] = element[i+1];
            }
            i++;
        }
        if (deleted)
            return --size;
        else
            return –1;
};
```

程序注释：

注 1 **与** List **模板一一样，使用根据元素值删除元素的函数，需要注意以下两个问题：**

- **如果用它存储整数，则它与上面根据序号删除元素的函数无法区分，编译器会报错。**
- **如果用它存储类，则在该类中必须定义运算符函数** operator==()**，否则程序无法进行比较两个实例的值。**

List **模板二每次插入元素时都要将全部元素拷贝一遍，这就增加了运算时间。计算机学科领域著名的“** time trade off space law **”（时间换空间定律）说的就是：若一个算法要节省运行时间，就要多占用内存空间；而反过来，若要少用内存空间，就势必要多耗费计算时间。**

8.1.5 List 模板三

List **模板一和模板二中存储的都是变量的实际值。当数据是有复杂内部成员的类的实例时，将实例一一拷贝到其中不但会降低程序的运行效率，而且还要耗用大量的内存。所以，更好的方法是仅存储实例的地址，即指针变量，而变量本体仍保存在它们原来的内存位置。**

理论上，模板可以用任何数据类型实例化，因此，只要用指针类型对 List **模板一和二进行实例化，它们就可以存储指针变量。如语句：**

```
typedef   List1<Student*>  StudentList;
```

所定义的 StudentList **就是一个存储** Student **类实例指针的序列。**

但是，如果一个模板既用于存储静态变量，又用于存储指针，就无法在模板中提供可供所有子类统一继承的函数，因为有许多运算是不能实施在指针变量上的，如比较大小等。因此本教材定义了一个专门用于指针的 List **模板三，并约定，今后凡是管理变量本体（如整数、实数或类）的序列，都用** List **模板一或二，而管理类的实例指针的序列都用** List **模板三。**

List 模板三：以下为定义 List **模板三的头文件** list_3.h **中的全部内容。**

```
// list_3.h   （对应 list_3.h 文件可从光盘上下载）
# include <stdlib.h> //  定义空指针值 NULL 的头文件
template <class T> class List3 {
```

```
public:
    List3();
    ~List3();
     int        Size() { return size; };
     int        Append(T*);
     int        Insert(int, T*);
     int        Delete(int);
     int        Delete(T*);
    List3&      operator=(List3&);
     T*&        operator [ ](int i) { return element[i]; };
    void        BubbleSort(); // 关于该函数程序的讨论见 8.3.1 节例 8-5
protected:
     T**        element;
     int        size;
};

template <class T>
List3<T>:: List3()
{
    element = NULL;
    size = 0;
}

template <class T>
List3<T>:: ~List3()
{
    if (element)
        delete [] element;
}

template <class T>
List3<T>& List3<T>:: operator=(List3& l)
{
    size = l.size;
    if (size = 0)
        element = NULL;
    else {
        element = new T* [l.size];
```

```
        for (int j = 0; j < size; j++)
            element[j] = l.element[j];
        }
        return *this;
}

template <class T>
int List3<T>:: Append(T*  t)
{
    T**  tmp = new T* [size + 1]; // 生成一个新的动态数组
    for (int i = 0; i < size; i++)  // 将原数组中元素拷贝到新数组中
        tmp[i] = element[i];
    tmp[i] = t;  // 使新数组最后一个元素等于插入元素
    if (element) delete [] element;  // 释放原数组的内存空间
    element = tmp;   // 使数据成员等于新数组
    size++;   // 序列长度增加 1
    return size;
};

template <class T>
int List3<T>:: Insert(int j, T*  t)
{
    if (j > size)  // 如插入序号大于序列长度，附加在序列尾部
        return append(t);
    T**  tmp = new T*  [size +1]; // 生成一个新的动态数组
    for (int i = 0; i < j; i++)   // 拷贝前 j –1 个元素
        tmp[i] = element[i];
    tmp[i] = t;  // 使第 j 个元素等于新元素
    for (    ;i < size; i++) // 拷贝其余的元素，j 的数值接续上式，不必另行赋值
        tmp[i+1] = element[i];
    if (element) delete [] element;  // 释放旧数组的空间
    element = tmp; // 使数据成员等于新数组
    size++;  // 将序列长度增加 1
    return size;
};
// 根据序号删除元素
template <class T>
int List3<T>:: Delete(int j)
```

```
{
    if ( j > size || j < 0)
        return –1;
    for (int i = j; i < size –1;  i++)
        element[i] = element[i+1];
    size--;   //  将序列长度减 1
    return size;
};
//  根据指针值删除元素，注 1
template <class T>
int List3<T>:: Delete(T* x)
{
    int deleted = 0;
    for (int i = 0 ; i < size;  ) {
        if (element[i] == x) {
            deleted = 1;
            for (  ; i < size  - 1; i++)
                element[i] = element[i+1];
        }
        i++;
    }
    if (deleted)
        return --size;
    else
        return –1;
};
```

程序注释：

注 1　**因为这里是根据元素的指针值进行删除作业，所以在使用上不像** List **模板一和二那样对数据类型有限制，在类中也无需定义运算符函数** operator = = ( )**。**

该基于指针的模板特别适合于用生成子类的方法进行实例化。因为 List **模板中成员函数程序均基于指针形式，任何由它衍生出来的子类均可应用模板中的成员函数，模板的适用范围大而灵活。该模板在初始化时，可直接用类的静态变量形式，而不必用其指针形式。如下述语句：**

```
typedef List3<Student> StudentList;
```

生成一个管理 Student **类实例指针的序列。**

8.1.6 Linked List

上述 3 **种** List **模板在每次插入和删除元素时都要移动后续的一系列元素，当元素数量较大时，程序运行效率将大大降低。**Linked List **的基本思路是使** List **元素中包括两部分内容，一部**

分是实际的数据，另一部分是指向下一元素的指针。然后如图 8–1 所示，将这些单元像链条那样一个一个串接起来，其中最后一个元素中向上的箭头表示其值应设为 NULL。这也是传统上将这种方法称为链表的原因。

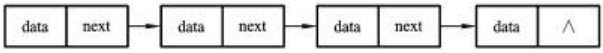


图 8–1 用首尾相接的元素组成 Linked–List

采用这种链接结构，当插入元素时，只要断开链接，插入一个新的单元即可，而不必移动其他元素（即进行元素数据的拷贝）。当删除元素时，只需将其从链接中摘除，并将其前后元素直接链接起来即可，不必移动其他元素。但是，在另一方面，这种结构使得序列的排序和查找操作都比较复杂，因此它比较适合于那些在一个固定位置上进行插入和删除运算的操作。它通常应用于一些特殊序列，如 Queue 和 Stack（详见 8.3 节）。

导读：

① 虽然还有很多其他 List 的实现方法，但只要掌握本教材所介绍的以数组形式实现序列的方法，就足以解决一般实际问题，在阅读有关编程的参考书时不要为众多的方法所困惑。

② 在书后配盘中有关于 linked list 的工作原理的动画描述。

8.2 矢量、矩阵和线性方程组的 C++模板

8.2.1 矢量和矩阵的数学概念

数值计算曾经是计算机程序的惟一应用，现在也仍然是计算机程序应用的重要领域。矢量、矩阵、线性方程组等都是数值分析中常用的数据结构，本节介绍专门用于表示矢量和矩阵运算的模板。

下面首先简要介绍关于矢量和矩阵的数学概念。n 个有序的数 $a_0, a_1, ..., a_{n-1}$ 所组成的数组称为 n 维向量，这 n 个数称为该向量的 n 个分量，第 i 个数 a_i 称为第 i 个分量。

n 维行向量可写成

$$\boldsymbol{a} = (a_0, a_1, \ldots, a_{n-1})$$

n 维列向量可写成

$$\boldsymbol{a} = \begin{pmatrix} a_0 \\ a_1 \\ \mathrm{M} \\ a_{n-1} \end{pmatrix}$$

按矩阵的有关规定，分别称为行向量和列向量，也就是行矩阵和列矩阵，并规定行向量与列向量都按矩阵的运算规则进行运算。

由 $m \times n$ 个数 $a_{i,j}(i = 0, \Lambda\ , n-1;\ j = 0, \Lambda\ , m-1)$ 排列成的 m 行 n 列的数表称为 m 行 n 列矩阵，简称 $m \times n$ 矩阵，记为 $\boldsymbol{A} = (a_{i,j})$

$$\boldsymbol{A}=\begin{pmatrix} a_{0,0} & a_{1,0} & \cdots & \cdots & a_{n-1,0} \\ a_{0,1} & a_{1,1} & & & a_{n-1,1} \\ \cdots & \cdots & & & \cdots \\ a_{0,m-1} & a_{1,m-1} & \cdots & \cdots & a_{n-1,m-1} \end{pmatrix}$$

一般也将其视为 n 个 m 维列矢量组成的矢量组。

$$\boldsymbol{A}=\begin{pmatrix} a_{i,0} \\ a_{i,1} \\ a_{i,2} \\ \mathrm{M} \\ a_{i,m-1} \end{pmatrix}, \quad i=0,\Lambda\ ,n-1$$

导读：

更多的行列式、矩阵和矢量的性质、运算等数学概念，可参看线性代数方面的教材。

8.2.2 矢量和矩阵的模板

由上述定义可知，矢量就是数的线性排列，序列就是它们最自然的表示方式。矩阵又可视为矢量的线性集合，所以可用序列的数组表示。下面就是矢量和矩阵的模板：

矢量模板：以下为定义矢量模板的头文件 vector.h 中的全部内容。

```
// vector.h    （可从光盘的对应章节中下载）
template  <class T, int n > class Vector {
public:
    Vector() { size = n; };
     int   Size() { return size; };
      T&        operator[ ](int j) { return element[j]; };
     Vector&  operator +(Vector&);
     Vector&  operator – (Vector&);
private:
      T      element[n];
      int size;
};
// 数学上，矢量相加就是对应元素相加
template <class T, int n>
Vector<T, n>& Vector<T, n>:: operator+(Vector<T, n>& v)
{
    for (int i=0; i< size; i++)
          element[i] = element[i] + v.element[i];
     return *this;
```

```
};
// 数学上，矢量相减就是对应元素相减
template <class T, int n>
Vector<T, n>& Vector<T, n>:: operator-(Vector<T, n>& v)
{
    for (int i=0; i < size; i++)
        element[i] = element[i] – v.element[i];
    return *this;
};
```

矩阵模板：以下为定义矩阵模板的头文件 matrix.h **中的内容**

```
// matrix.h （可从光盘的对应章节中下载）
# include "vector.h"
template <class T, int m, int n> class Matrix {
public:
    Matrix() { size = m; };
    Vector<T, n>& operator[ ](int i) { return column[i];};
    Matrix&  operator +(Matrix&);
    Matrix&  operator –(Matrix&);
    int Row()  { return column[0].Size ( ); };
    int Size()  { return size; }
protected:
// 将矩阵定义为列矢量的序列，
    Vector<T, n> column[m];
    int  size;
};
// 矩阵相加就是对应列矢量相加
template <class T, int m, int n>
Matrix<T, m, n>& Matrix<T, m, n>:: operator+(Matrix<T, m, n>& x)
{
    for (int i = 0; i < size; i++)
        column[i] = column[i] + x.column[i];
    return *this;
};
// 矩阵相减就是对应列矢量相减
template <class T, int m, int n>
Matrix<T, m, n>& Matrix<T, m, n>:: operator– (Matrix<T, m, n>& x)
{
    for (int i = 0; i < size; i++)
```

```
        column[i] = column[i] - x.column[i];
    return *this;
};
```

导读：

① 将矩阵模板与 7.1.7 节加以比较，可发现模板给编程带来的便利性。

② 第 11 章的课程作业二有应用矩阵模板的例子。

8.3 排序和检索

8.3.1 气泡法排序和函数模板

排序（sort）是将 List 中的元素，依据某些数据成员的值按某种线性顺序排列，如逐渐增大（称为升序）或逐渐减小（称为降序）。排序所依据的数据成员称为关键字（keys）。排序对数据应用有重要意义，如下一节所要讨论的那样，在一个有序 List 中可高效地进行检索。

本节介绍一种最简单的排序方法：气泡法（bubble sorting）。例 8-3 是用于 List 模板一的排序函数。

例 8-3：List 模板一用排序函数

```
// 注 1
template <class T, int n>
void  List1<T, n>:: BubbleSort()
{
    for (int i = 0; i < size – 1; i++)
            for (int j = size – 1; j > i; j--) // 逐个将元素与其后一个元素进行比较
                if (element[j] < element[j –1])   // 注 1
                    // 如后面元素大，则对换两个元素的值
                        Swap(element[j], element[j –1]);   // 注 2
};
```

该函数的工作过程如下：

Step 1：循环变量 i 从序列第一个元素开始，i = 0；循环变量 j 从序列尾部 j = size – 1 的元素开始，然后以递减方式向序列头部移动，直到 j = i + 1 = 1 时为止。在循环体中，元素 element[j] 依次与前一个进行比较，若它比前一个元素“轻”（即其值小），则将它与前一个元素对换。循环完成后，整个序列中最轻的元素将被放在 element[0]的位置。

Step 2：由于第一个元素已是序列中最小的，所以循环变量从 i = 1, 即第二个元素开始，循环变量 j 仍然从序列尾部 j = size – 1 开始，以递减方式向序列头部移动，直到 j = 2 时为止。在循环体中，元素 element[j]依次与前一个进行比较，若它比前一个元素“轻”，则将它与前一个元素对换。循环完成后，将整个序列中最轻的元素放在 element[1]的位置。element[1]一定是比 element[0]“重”的元素。

Step 3：以上运算将一直进行到 i = size – 2 时，即倒数第二个元素时为止，这时所有的元素都将排列有序。

程序注释：

注 1 虽然这里把排序函数 BubbleSort 单独抽出来进行讲解，但是它应该是与其他函数一起被包括在模板定义头文件 list_1.h 中的。

注 2 程序在这里对两个实例的值进行比较运算，如果不是基本数据类型，而是用户自定义的类，则该类中必须包括运算符函数 operator < (…)的定义，否则类的实例将无法进行比较大小的运算。

注 3 程序在这里互换两个实例值。在 7.1.5 节中讨论引用类型时，例 7–2 介绍过 Swap 函数的作用是对换输入函数的两个参数的值。但一般函数仅能针对特定类型的变量，而模板无法预先知道将来实际变量为何种类型。要使模板的成员函数能适用于将来实例化时遇到的任何数据类型，函数也必须是与具体数据无关、普遍适用的形式。为此，C++语言提供了一种函数模板（function template），其作用就是定义一个与具体数据类型无关的函数。

下面的例 8-4 就是例 8-3 排序程序中所用 Swap 函数的模板：

例 8-4：对换参数 a 和 b 的值的函数模板

```
template <class T>
void Swap(T& a, T& b)
{
    T c;
   c = a; a = b; b = c;
};
```

将 Swap()函数模板与例 7–2 的 Swap()函数进行比较可知：函数模板定义语句格式与普通函数定义语句的格式基本相同，不同之处在于：

① 其中本来是具体数据类型的地方现在被符号所代替。

② 在函数模板定义语句句首与类模板一样，增加了一个模板参数表。

应用函数模板的方法是直接将具体的数据代入函数调用语句中即可，如例 8-3 所示。

例 8-5：List 模板三用排序函数

```
// 注 1
template <class T>
void   List3<T>:: BubbleSort()
{
   for (int i = 0; i < size – 1; i++)
            for (int j = size – 1; j > i; j– –)
                  if (*element[j] < *element[j –1])   // 注 1
                        Swap(element[j], element[j –1]);
};
```

程序注释：

注 1 该排序函数应该是与 List 模板三的其他函数一起被包括在头文件 list_3.h 中的。

注 2 该函数与例 8-3 基本相同，只是不能直接比较两个元素，因为它们是指针，而不是元素的真实值，所以需要用取值运算符*获取变量本身的值后进行比较。

如欲应用该函数对某个类,如 Student 类进行排序,则该类中则必须定义运算符函数 operator < (…),其作用是根据对排序关键字值的比较确定实例值的大小顺序。例 8-6 以例 7–4 中 Student 类为例，说明了如何定义单关键字排序的运算符函数 operator<()，如何定义多关键字排序的运算符函数 operator<()，并说明了如何应用例 8-5 所示的排序函数对 Student 类的实例进行排序。

例 8-6：对 Student 类的实例进行排序的方法示例。

① 首先用继承 List 模板三的方法生成一个 StudentList：

```
class StudentList: public List3<Student> {
public:
    StudentList();
    Student*  SearchByID(char*);   //  对该函数程序的讨论见 8.3.2 节例 8-8
};
```

该定义中所包括的查询函数 SearchByID()将在 8.3.2 节中讨论。

② 单关键字排序：只要在 Student 类中定义一个根据学生编号进行单关键字排序的运算符 operator<()函数，就可以直接继承例 8-5 所示 List 模板三的排序函数：

```
int   Student:: operator <(Student& s)
{
     return (strcmp(id, s.id) < 0); //  令 id 值小的实例为小
}
```

上述运算符函数是用单个关键字进行排序，但实际应用中经常需要对实例按多种顺序排序，如对 Student 类的实例，可先按班级排列学生记录，而在同一班级中再按姓名排列。这样对多个属性值进行排序称为多关键字排序。

③ 多关键字排序：实现多关键字排序的关键是使函数中的运算符函数 operator<(…)能按关键字确定实例大小。工作原理与比较字符串大小的函数 strcmp 类似，先比较第 1 个关键字，若不等，就以它们的大小确定实例的大小；若相等，则比较第 2 个关键字。若不等，就以它们的大小确定实例的大小；若相等，则比较第 3 个关键字。依此类推，直到所有关键字都比较完毕。下面就是一个对学生先按班级、后按姓名排序的运算符 operator<(…)函数实例。定义了该函数后，就可以直接用 List 模板三中的排序函数 BubbleSort()进行多关键字排序。

```
int Student：operator <(Student& s)
{
    if (strcmp(clsName, s.clsName) < 0) //  如果班级名字“小”，则认为实例“小”
        return 1;
    else if (strcmp(clsName, s.clsName) == 0) //  如班级名字“相同”，则比较学员名字
        return (strcmp(name, s.name) < 0);
    else
        return 0;
};
```

8.3.2 对分检索法

检索（search）也称为查找，在序列中查找符合给定条件的元素。有效进行检索的前提是元素必须有序，否则就只能逐个比较每个元素，直到发现符合条件的元素为止，这种方法也称为顺序检索法或者穷举法。当元素数量很大时，顺序检索法的效率显然很低。为此，现已发展出了许多更有效的检索方法。本节介绍的对分检索法（binary search）就是其中一种，应用该法的前提是搜索对象必须是一个有序 List。例 8-7 给出了 List 模板一上的检索函数。

例 8-7：

```
// 根据给定的数据值进行查找元素
// 如找到，则返回该元素的序号；如没有找到则返回–1
template <class T, int n>
int List1<T, n>:: BinarySearch(T& t)
{
    int left = 0, right = size – 1;
    while ( left <= right) {
        int middle = (left + right)/2;   // 求出序列中间位置
        // 将输入元素与中间位置上的元素进行比较
          if (t < element[middle])        // 若小于中间位置上的元素，注 1
              right = middle – 1;         // 将序列尾部移至中间位置，重新计算中间位置
          else if (element[middle] < t) // 若大于中间位置上的元素
              left = middle + 1;          // 将序列头部移至中间位置，重新计算中间位置
          else
              return middle;              // 若等于中间位置上的元素，查询成功
          };
     return –1;
};
```

该函数工作过程如下：

① 将给定的数据与序列中间位置上，middle = (left + right)/2 的元素进行比较，若比它小，则只搜索序列前半部分，即以从头至 middle 部分为新的搜索范围；若比它大，则只搜索序列后半部分，以从 middle 至尾的部分为新的搜索范围。

② 对缩小了搜索范围的序列重复上述过程，找出新的中间位置，将该位置上的元素值与给定值进行比较，并根据结果确定是取前半段还是后半段。

③ 重复上述过程，直到某次搜索的序列中间元素的值等于给定值时，将其序号返回，检索成功。若不能发现这样的元素，则返回–1。

程序注释：

注 1 与例 8-3 的排序函数一样，如果对类实例进行查找，类定义中必须有重载运算符< 的成员函数 operator < (…)。但是，如果序列中存储的不是基本数据而是类的实例时，一般不会根据元素值来进行检索，而是根据类的数据成员中的某些关键字进行检索。

如果需要对某个类的序列进行按属性值的检索，则该序列应该用继承模板的方法产生，以便子类中增加自己的成员函数，例 8-6 定义的 StudentList 就是这样一个例子。例 8-8 就是该 StudentList 类中所定义的按编号进行检索的函数 SearchByID()的程序代码。

例 8-8：StudentList 类检索函数的代码

```
Student   StudentList::SearchByID(char*  id)
{
    int left = 0, right = size – 1;
    while ( left <= right) {
          int middle = (left + right)/2;
           int d = strcmp(id, element[middle].ID());
           if  (d < 0)
                 right  =  middle – 1;
           else  if  (d > 0)
                left  =  middle  +  1;
            else
                  return  element[middle];
            };
     return NULL;
}
```

为了进一步改进对分检索的效率，还可采用按黄金分割比例（0.618）分段的方法，以加快锁定目标段落的速度。用顺序查找法时，若有 N 个数据，则在最坏情况下（即要查找的元素在序列最后），需进行 N 次比较运算。对分检索使检索次数在最坏情形下仍然能减少为 $\log_2 N$。例如，若有 100 个数据，最坏情况下顺序查找法需进行比较运算 100 次，而对分检索法只需进行 7 次即可。若有 1 000 个数据，最坏情况下顺序查找法需进行比较运算 1 000 次，而对分检索法只需进行 10 次即可，由此可见顺序查找法的效率之不高。

导读：

算法设计理论将运算次数与数据数量的关系称为时间复杂度。顺序检索法的检索次数与数据个数成简单正比关系，所以其时间复杂度为 $O(N)$。对分查找的算法复杂度为 $O(\log N)$，意思是它与数据个数的对数成正比。BubbleSort 的复杂度为 $O(N^2)$，意思是它与数据个数的平方成正比。$O(N)$、$O(\log N)$和 $O(N^2)$是衡量算法有效性的 3 种最主要的复杂度数量级。

8.3.3 插入排序

上述导读关于算法复杂度的分析说明，排序是一种时间复杂度很高的运算，但是对分查找的时间复杂度要比气泡法低一个数量级。因此，与其以后进行复杂的排序操作，不如从一开始就使元素保持有序。插入排序（insertion sort）方法的产生正是基于了这种想法，其实质是按序插入，从一开始就将元素按其大小插入到序列中对应位置。这样，序列自始至终都是有序的，不必屡屡进行排序作业。例 8-9 给出了一个用于 List 模板一的对分插入排序函数。

例 8-9：List **模板一上的插入排序函数。**

```
template <class T, int n>
int List1<T, n>:: BinaryInsert(T t)
{
    if (size = = length)      // list is full
          return 0;
    int left = 0, right = size – 1;
    while ( left <= right) {
        int middle = (left + right)/2;
        if (t < element[middle])
              right = middle – 1;
         else
              left = middle + 1;
      }
    for (int j = size; j > left; j– –)
          element[j] = element[j – 1];
     element[left] = t;
     return ++size;
};
```

关于序列上的排序，还有其他算法，如 Quick Sort、Heap Sort、Merge Sort **等。但是，它们对算法时间复杂度并没有数量级的根本改进，即使这些算法对时间复杂度有所改进，但都不同程度地降低了算法的稳定性。所以本教材不再深入涉及。**

导读：

① 以上范例程序只适用于 List **模板一和二。**List **模板三中存储的是指针，其排序和检索函数程序与此基本相同，惟一不同的是必须用取值运算符*****将指针变量转换为本体变量后才能进行比较运算，即将语句** if (t < element[middle])**改为** if (t < *element[middle])**，才能进行比较运算。因为数组元素现在只是变量的地址，对其不能进行比较大小的运算。**

② 这里介绍的排序和查找算法已能满足大多数应用程序的需要。此外还有许多其他排序和查找算法，欲深入了解，可参看专门论述数据结构和算法设计的教材。

8.4 队列和堆栈

List **是集合最基本的形式。在实际应用中还发展出了一些特殊的** List**，如队列（**Queue**）和堆栈（**Stack**）。**

Queue **的特点为：元素总是固定地从** List **一端进入（通常也称为尾部），从另一端取出（通常也称为头部），因此通常也叫先进先出（**First-In, First-Out**，**FIFO**）。**

Stack **的特点为：元素总是固定地从同一端进入和删除，因此通常也叫先进后出（**First-In,

Last-Out，FILO)。

由于这两种序列运算上的特点，用 7.1.5 节所介绍的 Linked List 的结构来实现它们比较有效。下面就详细介绍用 Linked List 编写的这两种序列的类模板。

8.4.1 Stack 模板

Stack 的主要功能包括：将一个元素放入堆栈顶部，将顶部的元素取出，查看其顶部元素但不取出。因此可以简单地用 linked List 的方法来实现。图 8–2 为 Stack 的结构示意图。

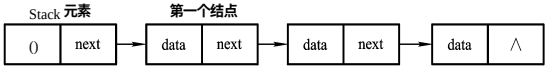


图 8–2 用 Linked–List 做 Stack

与其功能对应，在 Stack 的模板中只定义了 3 个成员函数。

Stack 模板：

```
// Stack.h      (程序文件可从光盘上对应章节下载)
template <class T> class Stack {
public:
     Stack();
     void  Push (T*);    //  将元素放入堆栈顶部
      T* Pop();           //  取出顶部元素
     T*  Peek();         //  查看顶部元素
protected:
      Stack  *next;
      T*  data;
};

template <class T>
Stack<T>:: Stack()
{
     data  =  NULL;
     next  =  NULL;
};
```

构造函数生成一个元素，它不存储数据，只起纯粹的链接作用，称之为 stack 元素。其指针 next 指向 Stack 中第一个结点。最初，指针 next 的值为空指针，说明 stack 为空。

```
template <class T>
void Stack<T>:: Push(T* d)
{
    Stack  *tmp  =  new  Stack;   //  生成新结点
      tmp->data = d;     //  使新结点的数据等于插入的数据
      tmp->next = next;   //  将新结点的 next 指针指向原先第一个结点
```

```
        // 将原先指向第一个结点的指针指向新结点，新结点被接入 stack 中
        next = tmp;
};

template <class T>
T* Stack<T>:: Pop()
{
        if (!next)  // 若没有下一个结点，Stack 为空
                return NULL;
        T* d = next->data; // 取出第一个结点中的数据
        Stack *tmp = next; // 记住第一个结点
        next = next->next; // 将指向第一个结点的指针指向其下一个结点，摘出原先第一个结点
        delete tmp; // 释放原先第一个结点所占内存
        return d; // 将原先第一个结点中的数据返回
};

// 当函数返回空指针时，表明 Stack 为空
template <class T>
T* Stack<T>:: Peek()
{
        if (!next) // 若没有第一个结点，Stack 为空
                return NULL;
        return next->data;  // 返回第一个结点中的数据
};
```

关于 Stack 在程序中应用的例子可参看 10.3.2 节。

8.4.2 Queue 模板

Queue 的主要功能包括：将一个元素放入队列尾部，将队列头部的元素取出，查看其头部元素但不取出。因为插入元素的操作总是在一端，而取出元素的操作总是在另一端，所以在元素中定义了 2 个指针，分别指向其前后环节，使插入和取出元素的操作变得非常简单。这种结构也称为 double– linked list。图 8–3 为该结构的示意图。

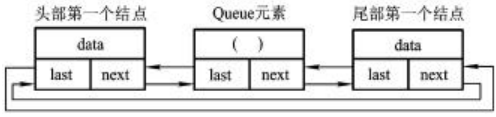


图 8–3 double–linked list

根据要求的功能，在模板定义中也只包括了 3 个成员函数。

Queue 模板：

```
// Queue.h (程序文件可从光盘上对应章节下载)
template <class T> class Queue {
public:
      Queue();
      void  Put(T*); //  将一个元素放进队列尾部
       T*    Get();    //    取出队列头部的元素
       T*    Peek();   //  查看队列头部的元素，但是并不取出
private:
      T*  data;
     Queue  *next,  *last;
};

template <class T>
Queue<T>:: Queue()
{
     data  =  NULL;
      next  =  this;
      last    =  this;
};
```

构造函数生成一个元素，称之为 Queue **元素。它不存储数据，只起纯粹的链接作用，其指针** last **指向** Queue **头部第一个结点，**next **指向** Queue **尾部第一个结点。最初，指针** next **和** last **都分别指向** Queue **元素自己，说明** Queue **为空。**

```
template <class T>
T* Queue<T>:: Get ()
{
      if  (last  = = this) //  如指向第一个结点的指针指向 Queue 元素自己，说明 Queue 为空
            return  NULL;
      Queue*  p  =  last; //  记住 Queue 中头部第一个结点
      T*  d  =  p->data;    //  取出头部第一个结点中数据
       //  将原先指向头部第一个结点的指针指向其下一个结点，使其成为新的头部第一个结点
       last  =  p –>last；
        //  将新的头部第一个结点向后指针指向 Queue 元素，原先头部第一个结点被摘除
        last–>next = this;
       delete  p; //  释放原先头部第一个结点的内存
       return  d;    //  返回原先头部第一个结点中的数据
}
//  查看头部元素，当函数返回空指针时，表明 Queue 为空
template <class T>
```

```
T* Queue<T>:: Peek()
{
    if (last = = this)
        return NULL;
    return last –>data;
};
// 将元素放入 Queue 尾部
template <class T>
void Queue<T>:: Put(T* d)
{
    Queue* tmp = new Queue(); // 生成一个新结点
    tmp –>data = d; // 使新结点数据等于插入值

    next –>last = tmp; // 将原先尾部第一个结点向后的指针指向新结点
    tmp –>next = next; // 将新结点的向前指针指向原先的尾部结点
    tmp->last = this; // 将新结点的向后指针指向 Queue 元素
    next = tmp; // 将 Queue 元素向前指针指向新结点，新结点完全接入 Queue 中
};
```

导读：

各种 C++编译器均提供类库（class library），这就是与数据类型无关的通用模板，例如，VC++提供了 3 种类模板，其有关定义均在系统头文件<afxtempl.h>中：

① List，用 doubly linked list 方式实现的序列，但不能以序号方式获取元素，它可在任何位置插入和删除元素。

② Array，长度可变的序列，可通过序号获取元素。

③ Map（也称为 dictionary），以关键字排序的对象集合。

用户可用它们生成自己所需的序列。

注意，这些模板在不同的编译器中是不同的。与之比较，本教材提供的模板的最大优点是与具体编译器无关，更具有普遍意义，应用起来也更加方便。

习 题

知识点：

① List 是最简单、最基本、也是应用最广的数据结构，也是本教材关于数据结构常识教学内容的重点，本章介绍的 3 种 List 模板对今后实际编程工作也有重要的指导意义，因此本章安排了较多的关于 List 的练习。希望能认真完成这些练习，为后续学习和今后应用打下扎实基础。

② 类模板和函数模板是 C++最重要的功能，应通过练习逐渐掌握模板的概念并能应用于编程。

1. 在计算机上完成 List 模板一的程序，包括排序和检索功能。然后用第 6 章练习中的整数和浮点数数据进行检验。

2. 在计算机上完成 List 模板三的程序，包括排序和检索功能。然后用第 6 章练习中的整数和浮点数数据进行检验。

3. 用 List 模板三定义管理 Student 类实例的 StudentList，然后从文件读入数据，生成若干实例存放于该序列中：

① 实现对学生按姓名排序的功能。

② 按范例程序 8-6 实现先按班级排序、后按姓名排序的功能。

下面是将 student 实例的数据从文件读入到序列中的函数示例：

```
void TestStudentList( )
{
    char    fileName[256];
    StudentList   sList; //  管理学生实例的序列
    //  读入数据到序列中
    ReadStudentList(fileName, sList); //  注 2
    //  打印读入的数据
    for (int j = 0; j < sl.Size(); j++)
        printf("record %d Student ID = %s  Name =%s Gender=%c className=%s\n",
            j, sl[j]–>ID(), sl[j]->Name(), sl[j]–>Gender(), sl[j]->ClsName()); //  注 1
};
```

程序注释：

注 1　List 模板三中存储的是指针，在引用数据成员和调用成员函数时需用“->”的形式。

注 2　函数 ReadStudentList（…）的作用是将数据读入管理实例的序列中，并记下数据文件名，下面是其程序代码：

```
void ReadStudentList(char* fileName, StudentList& sList)
{
    FILE* fin;
    Student*    s;
    printf("Input student data file name: ");
    scanf("%s", fileName):
    if ((fin = fopen(fileName, "r"))==NULL) {
        printf("Input file %s open error\n", fileName);
        return;
    }
    //  读入数据
    while ((s = ReadStudent(fin)) != NULL) //  注 1
        sList.Append(s);
    fclose(fin);
}
```

程序注释：

注 1　函数 ReadStudent()的作用是读入有关学生的信息，生成一个动态实例，具体程序见第 7 章习题 3。

4. （选做）完成 stack 模板的程序，用它生成一个整数堆栈，用若干实际数据检验其工作情况。

5. （选做）完成 Queue 模板的程序，用它生成一个浮点数堆栈，用若干实际数据检验其工作情况。

第 9 章　编程作业全过程

知识点：

本章的目的是使学员对编程作业的全过程有比较全面的了解，能够实施以下任务：

① 用面向对象方法分析系统，用 Class Diagram 建立数据模型。

② 能根据数据模型设计 C++程序。

③ 能根据 UCD 设计用户界面。

④ 能解决程序编写作业各阶段中所遇到的问题，对程序进行比较彻底的检测。

9.1　软件系统开发过程

9.1.1　系统分析、系统设计和系统实施

实际工程问题往往涉及多个事物以及它们之间复杂的相互作用，为了能用计算机程序处理整个系统，必须将实际系统抽象为数据模型。模型的概念对工程技术人员来说并不陌生，模型是人们为了理解事物作出的一种抽象。模型有很多种，例如，物理模型是简化了的或者按比例缩小了的实物。数学模型则是符号和逻辑的组合，如各种数学公式。

计算机程序处理事物时需要两种模型：描述被处理对象本身的模型和描述处理它们过程的模型。前者也称为“数据模型”，后者则称为“过程模型”或者“动态模型”。

数据模型应具有充分的表现能力和足够的抽象程度，它必须能略去无关紧要的细节而又不至于影响把握事物的实质；它既要能充分反映相同事物之间的共性，同时又可反映同类事物中相互的差异。最重要的是，计算机系统所使用的数据模型必须是可计算的，也就是说它必须是能用计算机表示和处理的。

“数据模型必须可用计算机处理”听起来似乎很简单，但实现起来并不容易。众所周知，计算机语言所能处理的数据对象种类简单而有限，但客观世界中的事物都很复杂。要将其表示为计算机所能处理的简单形式，必须遵循科学的方法。

将纷杂的事实抽象为能用计算机进行处理的数学表达（即所谓数据模型）的过程也称为系统分析（system analysis）。数据模型是与具体编程语言无关的，设计处理这些数据模型的具体程序的过程就是系统设计（system design）。设计出来的程序还要经由系统实施（system implementation），或者说经过具体编程的过程，才能最后变成可在计算机上实际运行的系统。因此，完整的编程作业过程应包括 3 个阶段，所要解决的问题分别为：

- 系统分析：确定系统所要执行的功能（functional aspect）。
- 系统设计：确定实现所需功能的技术（technical aspect）。
- 系统实施：具体实现或制作的过程（procedural aspect）。

如图 9–1 所示，三者之间并不是一个顺序关系，而是一个互相交错的网状关系，编程人员要把实施过程中暴露出来的问题向分析人员和设计人员反馈，而分析和设计人员要根据意见修改设计，一个好的程序必须经过多次这样的反复才能完成。

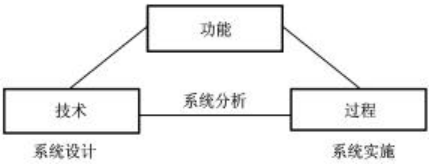


图 9–1 软件开发的 3 个方面（或 3 个阶段）

系统分析和设计方法有多种，最早出现并沿用至今的是面向过程的结构化 SADT（structural analysis and design technique）方法，目前发展最快、应用最多的则是面向对象的 OOAD（object-oriented analysis and design）方法。面向过程的方法的要点是将处理问题的过程分解为若干子过程（procedure），分别处理不同的数据。这种技术最大的问题是，数据与处理它的运算是分离的，数据管理机制十分复杂。举一个最简单的例子，在面向过程的 C 程序中用数组管理数据时，对记录数组长度变量的操作和对数组元素的操作是分别进行的，而在面向对象的 C++ 程序中，List 类作为一个逻辑整体，它在管理元素的同时也自动管理长度。

很多教材都把系统分析和系统设计归入软件工程的范畴，但作为一线编程人员，了解系统分析的基本方法、特别是数据建模的方法仍然非常必要。因为即使是编写一个很简单的应用程序，也需要对问题进行分析和抽象。就像定义二叉树那样由纯数学概念发展而来的简单的类，也需要应用数据建模的知识。

9.1.2 UML 方法

导读：

关于各种面向对象分析和设计方法的全面介绍，可参看有关论述软件工程的专著。

OOAD 方法起源于 20 世纪 50 年代后期，最初是由挪威国防研究所（norwegian defense research establishment）开发的，到 20 世纪 90 年代初，已产生了相当数量的面向对象的编程语言，如 C++、SmallTalk 等，以及许多分析设计方法，如 OMT、OSA、IDEF1x 等。它们的基本概念大致相同，只是描述模型的术语、图形符号等各有不同。经过努力，最终在 1994—1995 年间，由 Object Management Group 把各种方法统一为一个标准：UML（unified modeling language）。UML 采用模型描述系统需求的不同方面，能与各种编程语言(如 C++、Java、Visual Basic)直接联系起来。UML 采用的主要模型有图 9–2 所示的一些。

图 9-2 中：

- class diagram（类图），描述系统中事物对象以及事物之间的相互关系或者作用。
- use case diagram（用况图），描述用户在系统上执行哪些操作或者进行哪些运算。
- sequence diagram（顺序图），描述事件（即对象之间、对象与外界的相互作用）发生的顺序。

• state diagram（状态图），描述一个对象的生命周期等。

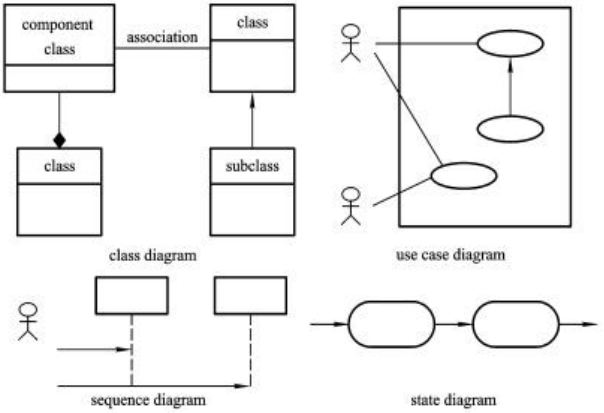


图 9–2 UML 方法主要模型

教材将着重介绍用类图分析事物，用用况图分析程序功能的系统分析方法以及将类图转换为 C++程序和根据用况图设计程序用户界面的系统设计方法。

9.2 用类图建立数据模型

导读：

因为许多书中对 UML 模型中的术语的翻译各不相同，本教材对类图中的有关术语将统一采用下述的名称：

- class：类
- object：类的实例，简称实例
- subclass：子类
- superclass：超类
- attribute：类的属性，简称属性
- key attribute：标识属性
- operation：类的运算，简称运算
- inheritance：类的继承，简称继承
- aggregation：聚集类，简称聚集
- association：类的关联，简称关联
- association class：关联类
- multiplicity：关联的参与度，简称参与度
- accessibility, visibility：可访问性

9.2.1 类和实例

- 类是若干具有相同性质的实例的抽象或概括
- 实例可以是一个物理实体，也可以是抽象概念，它是类所描述的概念的具体化。
- 外部世界根据运算区分不同的类。
- 类可以有属性。
- 类可以组成聚集。
- 类可以衍生子类。

识别和定义类的基本原则是将属性与运算组成一个逻辑单元，外部只能看见用运算定义的类，至于它包括哪些属性以及它如何实现有关运算，只有在类的内部才能看见。

9.2.2 属性

（1）属性的特性

属性是描写类性质的数据，具有以下特性：

- 属性必须是类中所有实例共有的性质，而不是其中个别实例的性质。
- 属性必须是单纯的数据，不包含内部复杂结构；类的属性为其他类或自己的实例的情形叫做对象的聚集（详见 9.2.9 节）
- 各属性名在类中必须是惟一的，不能重名。
- 类可有任意多个属性，也可以根本没有属性。
- 标识属性是这样一组属性，其值能惟一地区分各个实例。在类图中标识属性用字母 K 加以标识。

一般来说，属性都是实例固有的性质，但有时候则需要为编写程序增加另外的属性。如教师、学生等实例的属性本来只有姓名、年龄、性别等数据。但由于姓名往往重复，因此需要另外增加可作为标识属性的属性，如“编号”以便能惟一确定地区分不同的教师或者学生实例。

属性和标识属性并不是必须的，类可以没有属性，因为识别类的最重要的标准是事物之间的相互作用，即它们的运算，而不是其内部的数据。另外，因为 C++语言有多种方法可以把实例惟一地区分开来，如用指针来记录实例，所以对于没有属性的实例，也有方法将它们惟一确定地区分开来。

（2）属性的可访问性

根据封装性的原则，将对象属性的可访问性分成：

- public（公有），其值可被外部改变，在类图中用符号+标识。
- private（私有），其值不能被外部所改变，在类图中用符号–标识。
- protected（保护），其值可有选择地被外部所改变，在类图中用符号 # 标识。

对类的属性赋予初始值即产生特定的实例，开始其生命周期，随着其属性值的变化，进入不同的中间状态，直至最后状态或消亡。

9.2.3 运算

（1）运算的特性

类与其他类相互作用中表现出来的性质由其运算描述，运算具有以下特性：

• 运算必须是类中所有实例所共有的一种功能，它描述了类与类之间的相互关联或者说相互作用，因此，可从类的关联出发定义其运算。类可有任意个运算或没有任何运算。

注意，一般不把对属性的访问操作定义为运算，因为将类图中的类映射为 C++的类时，属性将被映射为 C++类的数据成员，随之也会定义获取其值和修改其值的成员函数（详见 9.3.1 节）。

（2）运算的可访问性

根据封装性的原则，对象的运算可访问性也被分成：

• public（公有），可被其他类使用，在类图中用符号+标识。

• private（私有），只能被类自己使用，在类图中用符号–标识。

• protected（保护），可有选择地被某些类使用，在类图中用符号 # 标识。

9.2.4 类的图形表示

如图 9–3 所示，类图用方框表示类的定义，其中包括三部分内容，分别为类的名字、属性和运算。实例也用方框表示，但在名字下多了一条下划线，并在其中列出属性的具体数值。为方便作图，也可简单地用一个方框表示类，然后另外用文档说明属性和运算。

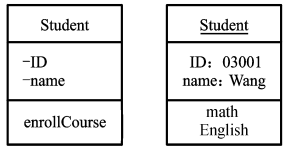


图 9–3 类定义和实例定义的图示

为了文档的可读性，对类、属性、运算一般按以下约定命名：

• 类名字为大写字母开头的英文名词，如果是词语组合，则后续每个词首字母均大写。

• 属性名字为小写字母开头的英文名词，如果是词语组合，则后续每个词首字母均大写。

• 运算名字一般为小写字母开头的“动词+名词”组合，后续每个词首字母均大写。

• 所有的命名要有充分的表现力或者描述力，能够使其含义不言自明。

导读：

除上述约定外，为便于系统设计并将类图模型与 C++程序直接对应起来，本教材中对所有类图模型都将用英语词汇命名。

9.2.5 关联

关联是类与类实例之间的相互作用或者相互关系的描述，实例与外部世界交换数据的相互作用，如给属性赋值或者获取属性值等一般不用关联来表示。

关联有以下特性：

• 关联是有方向性的。方向只反映分析问题的视点，从不同着眼点描述同一件事，最后产生的 C++类定义应该是等价的。

• 关联根据其涉及的类的数量，分为：

二元：只涉及两个类。

多元：同时涉及多个类。

返身：只发生在同类的实例之间，其例子可见图 9-9。

- 两个类之间可以有不止一个关联，但是不能违背冗余性约束（见 9.2.7 节）。
- 关联必须满足各种约束条件（详见 9.2.7 节）。

类图用连线表示关联，用连线上方的箭头表示其方向，如图 9–4 所示。

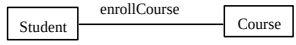


图 9–4 关联的图示

图 9-4 描述的是 Student（学生）和 Course（课程）之间的相互作用，Student 的实例可以注册（enrollCourse）具体的课程。

> **导读：**
>
> 在程序设计中，一般将多元关联转换为二元关联来处理，而自身关联可认为是二元关联的特例，所以本教材后续章节中所讨论的关联都是指二元关联。

9.2.6 关联类

伴随关联往往会产生一些附加信息，例如，当学生注册一门课程后就产生了 Score（成绩）这个新的信息。它不属于发生关联的两个类中的任何一方，而是描述关联本身的。类图用关联类来描述这种带有附加信息的关联。如图 9–5 所示，类图中关联类也用方框表示，放在表示关联的连线下方，并用虚线将它们与对应的关联连接起来。

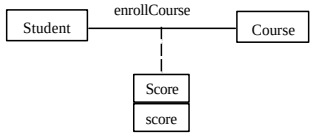


图 9–5 关联类

关联类可以有属性和标识属性，但是不能再与其他类有关联。

9.2.7 关联的约束

定义关联的约束的目的是保证数据模型具有完整性、一致性和最小冗余性。关联必须满足的最重要的约束条件是参与约束（participation constraint），它是关于双方类的实例可以参与该关联次数的规定，也称为参与度（multiplicity）。

表 9-1 列出了关联的几种主要参与度及其表示方法，例 9-1 讨论了几种常见的参与度。

表 9-1 关联的参与度

UML 图中标识	描 述	UML 图中标识	描 述
*	任意次或者没有	0..1	最多只有一次
1..*	至少一次	min..max	min、max 表示范围的上下限，如 2…6
1	有且只有一次		

例 9-1：几种常见关联的参与度。

① 图 9–6 表示一个国家（Country）有且只有一个首都（Capital）。这种双方都只参与关联一次的情况简称为 1:1 的参与度。

② 图 9–7 表示一个 Teacher 可以教多门 Course(课程),而一门课只能由一个 Teacher 来教。这种一方只参与一次，另一方参与多次的情况简称为*:1 的参与度（与之对称的另一种情况

是 1:*)。

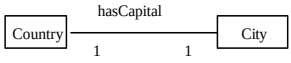


图 9–6 参与度示例 1

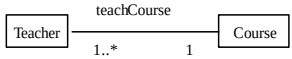


图 9–7 参与度示例 2

③ 图 9–8 表示每个 Customer（客户）可购买多种 Product（产品），而每种产品也会有多名客户购买。这种双方均参与多次的情况简称为*:*的参与度。

④ 参与关联中有一方参与度为 0..*，即不是其中每个实例都总是参与该关联的。这种情况也称“条件性（conditional）”关联。图 9–9 提供了一个条件性关联的例子，它同时也是一个返身关联，其参与度 0..1 的意思是有的 Person 可能不参与 isMarriedTo 的关联。但要注意的是，这里所说的可能不参与并不等于永远不参与，那些暂时没有参与的实例也许会在以后某个时候参与。如果总是有固定的一些实例不参与某个关联，则该关联就不能成立，因为关联必须是类全体实例都具有的性质。

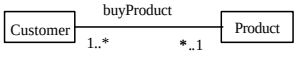


图 9–8 参与度示例 3

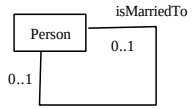


图 9–9 参与度示例——条件性关联

此外，经常考虑的还有冗余约束（redundancy constraint）。类图中如果存在冗余的关联，将给数据的一致性、完整性的维护造成困难。满足冗余约束的条件是类图中没有由非条件性关联构成的环路（见例 9-2），至于为什么环路会产生冗余的理论不在这里详述。

例 9-2：存在冗余的类图

图 9–10 中 3 个关联都是非条件性的，它们形成一个环路，从而造成了数据冗余，必须删除其中一个，如删除 Teacher 和 Student 之间的关联。删除以后，关于教师教学生的信息仍然可以通过运算得出（详见 9.3.5 节）。

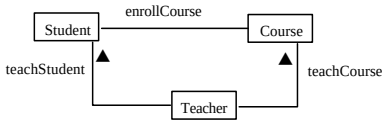


图 9–10 冗余的关联环路

其他约束条件还有基数约束（cardinality constraints），规定类实例的数量；并发约束（Co–concurrence Constraints），规定两个类中最多各有多少个不同的实例可以参与关联等。用户可根据问题的复杂程度确定究竟要设定多少约束。

9.2.8 继承

在 UML 的类图中，衍生子类（subclass）的类称为超类（superclass）。在子类和超类之间存在以下关系：

- 子类是超类的特例，超类是子类的概括或普遍化。
- 子类继承其超类的属性和运算。

- 子类可代替其父类，但父类不能替代子类。
- 继承关系可以是多层的，多层的继承关系可用一个树形结构表示，称为继承树。
- 继承关系是可传递的，即子类除了继承其父类的特性外，还自动继承了继承树中所有比它层次高的超类的特性。
- 对象不能成为它自己的父类或子类。
- 在父类对象和子类对象之间不能有关联。

类图用带箭头的连线表示继承关系，箭头从子类指向超类。图 9–11 中的继承树表示 Person 可被认为是 Student 和 Teacher 的概括，而 Student 又可进一步分为 Undergraduate（本科生）和 Graduate（研究生）。

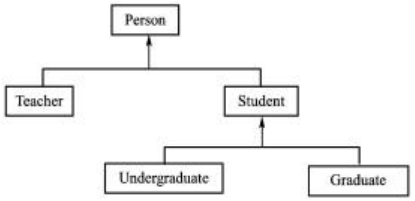


图 9–11 多层次的继承

9.2.9 聚集

若某个类的属性包括其他类的实例，则该类称为聚集，被包含的类称为成员（component）。聚集表达的是部分与整体的关系，聚集可以是多层次的，聚集还可包括类本身的实例。

聚集中所包含的各成员实例的数量称为聚集度（aggregation index）。聚集度一般有两种情况，其表示方法类似关联中的参与度的表示方法。

- 如果成员类实例的数量是可变的，则表示为 1..*，或者 min..max。
- 如果成员类实例的数量是确定的，则聚集度为一个确定的数值。

类图用带菱形的连线表示聚集关系，菱形所指为聚集对象。图 9–12 为表示计算机组成聚集的类图。

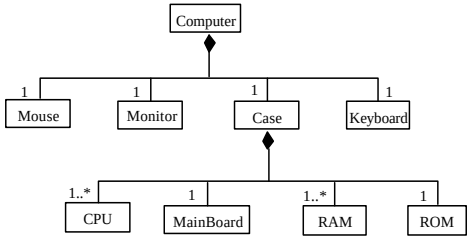


图 9–12 多层次的聚集

9.2.10 案例——学籍管理系统的数据模型

导读：

Case-3 是关于一个学籍管理系统的，它将贯穿用在本章和第 11 章中讲解编程作业的全过程。本节给出该案例讨论的第一部分：系统的数据模型，后序章节将用 Case-3-2、Case-3-3 这样的编号表明本案例讨论的后续部分。

Case-3-1：

第 11 章的课程作业一要求通过编写一个学籍管理系统掌握编程作业的全过程。系统分析和数据建模就是这个过程的第一项工作。图 9-13 给出了该学籍管理系统的完整的数据模型。该模型表明，学籍管理系统包括 3 个主要的类：Student、Teacher 和 Course。模型中存在一个继承关系，Student 类和 Teacher 类可以被认为是 Person 类的子类。在 Student 类和 Course 类之间关联并有关联类。在 Teacher 类与 Course 类之间也有一个关联。

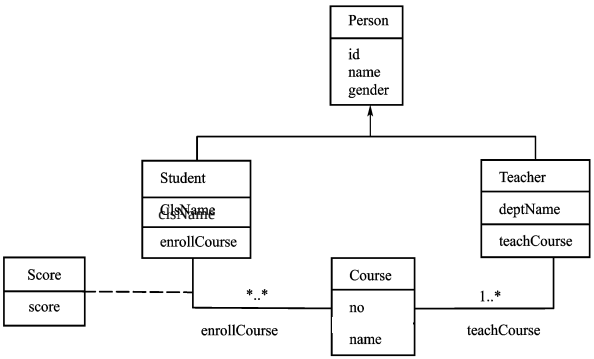


图 9-13 学籍管理系统的数据模型

9.3 由数据模型设计 C++程序

根据数据模型设计具体程序的过程属于系统设计（system design）的范畴。类图描述的数据模型是一种不依赖于物理实现的逻辑模型。采用不同的映射机制，可将它转换为不同编程语言的程序或数据库程序，它也可用任何一种支持面向对象的编程语言来实现。这里主要介绍从类图产生 C++程序的设计规则。

9.3.1 类的映射规则

类图中的类直接对应于 C++中的类，因为 C++中的类的概念实际上就是由面向对象方法而来的。具体的转换规则是：

（1）属性表示为 C++中类的私有数据成员。

（2）根据 7.1.1 节定义类的原则，对每个数据成员定义必要的数据获取函数和赋值函数。

（3）为类定义若干构造函数。

（4）将关联表示为 C++类中描述对象之间相互作用的功能性函数，具体规则详见 9.3.4 节。

由 C++程序中类定义产生的实例就是类图中的实例。

9.3.2 继承的映射规则

对象的继承直接对应于 C++中的父类和子类的定义（参见 7.2.3 节）。例如，图 9–11 中所示 Person 与 Student、Teacher 的继承关系的 C++定义可以示意性表示如下：

```
class Person {
// 省略其中具体函数和数据定义
    …;
};

class Teacher: public Person {
    …;
};
class Student: public Person {
    …;
};

class Undergraduate: public Student {
    …;
};


class Graduate: public Student {
    …;
};
```

9.3.3 聚集的映射规则

因为聚集度一般有两种情况：

- 成员类实例数量是确定不变的，如二叉树每个分支结点总是包括两个子结点。
- 成员类实例数量是不确定的，如一般树中的分支结点各有不同数量的子结点。

与之对应，聚集的映射规则也有两条：

- 当成员类实例数量确定不变时，用确定数量的类的实例作数据成员。
- 当成员类实例数量不确定时，则用类的序列作数据成员。

例 9-3 是这两种映射规则具体应用的例子。

例 9–3：如图 9–14 所示，二叉树可视为由 BiTree 类本身实例作成员的聚集，因为聚集度 2 是确定的，所以二叉树的 C++定义中用两个 BiTree 的实例作数据成员。下面就是二叉树的 C++

类定义示意。

图 9–14 二叉树和多叉树的数据模型

```
class BiTree {
public:
    BiTree();
    BiTree*&  LeftNode() { return leftNode; };
    BiTree*&  RightNode() { return rightNode; };
protected:
    // 这里需要增加关于结点数据的定义，注 1
    BiTree    *leftNode, *rightNode; // 注 2
}
```

程序注释：

注 1 这里没有关于结点数据成员的定义，因为其类型尚未确定。

注 2 因为在左右子树可能为为空，所以用指针类型变量作数据成员，当其值等于 NULL 时，即表示子树为空。

一般树，或者说多叉树，也是一个包含本身实例的聚集，因其聚集度是可变的，所以可用 List 的模板定义一个 GTree 类的序列作它自己的数据成员。下面即为多叉树的 C++定义示意。

```
# include "list_3.h" // 注 1
class GTree {
public:
    GTree();
    GTree*& operator[ ](int);   // 注 2
protected:
    // 这里需要增加关于结点数据的定义，注 3
    List3<GTree>    nodes; // 注 1
};
```

程序注释：

注 1 这里用 List 模板三生成一个存储子树指针的序列做数据成员。

注 2 因为子树存储在一个序列中，用下标运算符函数可方便地获取各子树。

注 3 这里也未定义有关结点的数据成员，因为不能确定其类型。

导读：

上述二叉树和多叉树的 C++定义只是示意性的，因为没有关于结点数据类型的定义，无法直接应用于实际问题。更好的方法是将其定义为与结点具体数据类型无关的类模板，这些类模板将在第 10 章中详细介绍。

9.3.4 关联的映射规则

关联是关于事物相互作用的描述，将其映射为 C++代码的工作包括：

- 根据关联的参与度在其中一个类中引入关联标识属性。
- 或者根据关联的参与度引入第 3 方 C++的类。
- 用 C++类的功能性函数表示关联所描述的类的运算，函数的输入参数和返回值需根据关联的参与度确定。

以上映射规则的实施都与关联的参与度有关，例 9-4 给出了 4 种常见参与度的映射规则，分别与例 9-1 中的 4 种情况对应：

例 9-4：关联的映射规则

（1）参与度为 1:1 的关联（与例 9-1（1）的情况对应）

将其映射为 C++程序的规则是：

- 将其中任一方 C++类中的标识属性引入对方 C++类定义中作关联标识属性。
- 或者将任一方 C++类的指针变量引入对方类中作关联标识属性。

以下就是由该规则之一产生的图 9–6 中 Country 类和 City 类的 C++定义：

```
class City {
public:
    City(char*);
    char* Name() { return name; } :
private:
// 为简单起见，只给该 class 定义一个属性
    char name[256]; // (K)城市名称
};

class Country {
public:
    Country(char*, char*);
    char* Name() { return name; };
    // 描述关联 hasCapital 的功能性函数
    char*     HasCapital() { return capitalName;};  // 注 1
private:
    char name[256]; // (K)
    char      capitalName[256]；// (RK)  from class City
}
```

程序注释：

注 1　这里说明了功能性函数 HasCapital()的定义方法：

- 使其与原图中关联同名。
- 使其返回值等于类中的关联标识属性。

下面以实际数据为例说明为什么要这样定义功能性函数以及如何编写其具体程序。表 9-2(a)是 City 类实例的数据、表 9-2(b)是 Country 类实例的数据。

表 9-2(a) City 类实例的数据

属性名	name
属性值	Beijing
	Washington
	Tokyo

表 9-2(b) Country 类实例的数据

属性名	name	capitalName
属性值	China	Beijing
	U.S.	Washington
	Japan	Tokyo

描述关联 hasCapital 的功能性函数 HasCaptial()的作用是返回某个国家的首都是哪个城市的信息。由表 9-2 可知，因为关于 Country 的 hasCapital 的数据就在 Country 类实例的数据成员中，因此简单地将 Country 类实例中关联属性 capitalName 的值返回去就可以实现该功能。

导读：

上述映射规则说的是可将任意一方类的标识属性引入对方，所以也可反过来，将 Country 的 name 作为关联标识属性引入到 City 类中，但那样函数 HasCapital()的程序就比较复杂，关于其具体程序的编写方法可参看关于 1:*关联的规则。

以下是根据映射规则之二产生的 Country 类的 C++定义。

```
class Country {
public:
    Country(char*, City*);
    char* Name() { return name; };
    //  描述关联 hasCapital 的功能性函数
    City*      HasCapital() { return capital; };  //  注 1
private:
    char       name[256]; // (K)
//  引入 City 类的指针做关联标识属性
    City*      captial；// (RK)
};
```

程序注释：

注 1 这里功能性函数 HasCapital()返回的是 City 类实例的指针，和第一种方法相比，结果的可读性不强，但优点是通过实例的指针可以获取关于该 City 实例的全部信息。反之，如果仅知道关联标识属性，则还必须通过查询 City 类的序列才能找到有关实例。两种方法各有优缺点，可以根据具体程序需求确定究竟采用何种映射规则。

虽然规则建议了两种映射关联的方法，但这里和下面的讨论中主要还是用引入关联标识属性的方法为例，因为比较直观、便于理解。

（2）参与度为 1:*（或*:1）的关联（与例 9-1(2)的情况对应）

将其映射为 C++程序的规则是：

- 将参与度为多次一方类的标识属性引入参与度为 1 的类中作关联标识属性。
- 或者将参与度为多次的类的指针变量引入参与度为 1 的类中作数据成员。

以下就是根据该规则之一产生的图 9–7 中 Teacher 类和 Course 类的 C++定义。

```
# include "list_3.h"   // 用 List 模板三定义 Teacher 类的序列
class Course {
public:
    Course(int, char*, char*);
    int   No(); // 获取课程编号
    char*    Name(); // 获取课程名字
    char*    TeacherID();   // 获取授课教师 ID
private：
    int       no；// (K)，课程编号
    char    name[256];   // 课程名字
    char   teacherID[20]; // (RK) from class Teacher
}

typedef List3<Course> CourseList;
class Teacher {
public:
    Teacher(char*, char*, char);
    char* ID() { return id; };
    char* Name( ) { return name; };
    // 用于描述关联 teachCourse 的功能性函数
    CourseList*   TeachCourse(CourseList&); // 注 1
private:
    char id[20]; // (K) 教师身份证号
    char name[8];   // 教师姓名
    char gender;
};
```

程序注释：

注 1　下面以实际数据为例说明为什么要这样定义功能性函数以及如何编写其具体程序。表 9-3（a）是 Teacher 类实例的数据、表 9-3（b）是 Course 类实例的数据：

表 9-3(a)　Teacher 类实例的数据

属性名	Id	name	gender
属性值	320105195001015610	Wang	M
	230116196311174477	Zhang	F
	120120197009083311	Li	M

表 9-3(b) Course 类实例的数据

属性名	no	name	teacherID
属性值	1001	High Math.	320105195001015610
	1005	Discrete Math.	320105195001015610
	1011	English	230116196311174477
	2012	Computer Hardware	120120197009083311
	2013	Network	120120197009083311

描述关联 teachCourse 的功能性函数 TeachCourse()的作用是返回某个教师教哪些课程的信息。由表 9-3 可知，因为关于教师 teachCourse 的信息被包括在若干 Course 类实例的数据成员中，所以要用存储所有 Course 的实例的序列作为函数 TeachCourse 的输入参数（该序列用 List 模板三生成），以便从中检索出所有由某个教师所教的课程，存放在一个序列中作为函数的返回值。下面即为描述该关联的功能性函数 TeachCourse()的程序代码。

```
CourseList* Teacher::TeachCourse(CourseList& cl)
{
    CourseList* tmp = new CourseList; // 生成一个动态序列做函数返回值
    for (int j = 0; j < cl.Size(); j++)
        if (strcmp(cl[j]–>TeacherID(), id) == 0)  // 注 1
            tmp–>Append(cl[j]);
    return tmp;
}；
```

程序注释：

注 1 根据教师编号对 Course 类进行检索，如果 Course 类实例中的数据成员 teacherID 的值与当前 Teacher 类实例的 id 相同，说明该课程由该教师所教，则将该实例指针加入到作为函数返回值的序列中。

（3）参与度为 *:* 的关联（与例 9-1（3）的情况对应）

将其映射为 C++程序的规则是：

- 引入第 3 方 C++的类，将两方面类的标识属性均引入这个第 3 方 C++类定义中做标识属性。
- 或者引入第 3 方 C++的类，将两方面类的指针变量都引入第 3 方类中作数据成员。

以下就是根据该规则之一产生的图 9–8 中 Customer 类、Product 类和 Order 类的 C++定义。

```
# include "list_3.h"   // 用 List 模板三生成 Order 类的序列
// 引入的第 3 方 C++类
class Order {
public:
    Order(int*, int*);
    int     CustomerNo();
    int     ProductNo();
private:
```

```
    int      customerNo;   // (RK) from class Customer
    int      productNo;    // (RK) from class Product
};

typedef List3<Order> OrderList;   // Order 类的序列

class Customer {
public:
    Customer(int, char*);
    int        No();
    char*      Name();
    // 用于描述关联 buyProduct 功能性函数
    OrderList*   BuyProduct (OrderList&); // 注 1
private:
    int      no; // (K)  客户编号
    char   name[8]; //  客户姓名
    …；// 其他属性
}；

class Product {
public:
    Product(int, char*, float);
    int        No(); //   获取 product 编号
    char*      Name(); //   获取 product 名称
    float      Price()；// 获取 product 单价
private：
    int      no; // product 编号  (K)
    char   name[256]; // product 名称
    float   price; // product   单价
};
```

程序注释：

注 1　**下面以实际数据为例说明为什么要这样定义功能性函数，以及如何编写其具体程序。表** 9-4（a）**是** Customer **类实例的数据、表** 9-4（b）**是** Product **类实例的数据，表** 9-4（c）**是** Order **类实例的数据。**

表 9-4(a)　Customer **类实例的数据**

属性名	no	Name
属性值	10210	Zhu
	31305	Chen
	52607	Dong

表 9-4(b)　Product **类实例的数据**

属性名	no	name	price
属性值	1001	TV	500.00
	1005	Wash Machine	1000.00
	2012	DVD	350.00

表 9-4(c) Order 类实例的数据

属性名	productNo	customerNo
属性值	1001	10210
	1001	31305
	1005	31305
	1005	52607
	2012	53607
	2012	10210

描述关联 buyProduct 的功能性函数 BuyProduct ()的作用是返回某个 Customer 所购买的全部 Product 的信息。由表 9-4 可知，因为关于 buyProduct 的信息被包括在若干 Order 类实例的数据成员中，所以要用存储所有 Order 类的实例的序列作为函数 BuyProduct 的输入参数（该序列用 List 模板三生成），以便从中检索出所有由某个客户所购买的各种 Product，存放在一个序列中作为函数的返回值。下面即为描述 Custormer 与 Product 关联的功能性函数 BuyProduct()的程序代码：

```
OrderList*    Customer::BuyProduct(OrderList&  ol)
{
      OrderList*   tmp  =  new  OderList;   //  生成一个关于 Order 类的序列
      for (int j = 0; j < ol.Size(); j++)
            if (ol[j]–>CustomerNo() ==no) //  注 1
                  tmp–>Append(ol[j]);
      return tmp;
}；
```

程序注释：

注 1 根据 Customerno 对 Order 类实例进行检索，如果其 customerNo 与当前 Customer 实例的 no 相同，说明该 Product 为其所购买，即将该 Order 的实例加入到作为函数返回值的序列中。

（4）条件性关联（与例 9-1(4)的情况对应）

如果参与关联的类中有任一方是条件性的，则映射规则是：

- 引入第 3 方的 C++类，将两方面类的标识属性都引入这第 3 方 C++类定义中作标识属性。
- 或者引入第 3 方的 C++类，将两方面类的指针变量都引入这第 3 方类中作数据成员。

以下就是根据该规则之一产生的图 9–9 中 Person 类和 Marriage 类的 C++定义。

```
//  引入的第 3 方 C++类
class Marriage {
public:
      Marriage(Person*,  Person*);
       char*    HusbandID() { return husbandID; };
       char*    WifeID() { return wifeID; };
private:
```

```
    char*    husbandID;
    char*    wifeID;
};

List3 <Marriage> MarriageList;

class Person {
public:
    Person(char*, char*, char);
    char* ID();
    char* Name（）；
    char Gender（）；
// 用于描述关联 isMarriedTo 功能性函数，注 1
    char*      IsMarriedTo(MarriageList&);
private：
    char id[20]; // (K)
    char name[8];
    char gender;
};
```

程序注释：

注 1　**下面以实际数据为例说明为什么要这样定义功能性函数，以及如何编写其具体程序。表** 9-5（a）**是** Person **类实例的数据、表** 9-5(b)**是** Marriage **类实例的数据（因为这是一个返身的关联，所以包括引入的第** 3 **方类，总共** 2 **个类。）**

表 9-5(a)　Person **类实例的数据**

属性名	id	name	gender
属性值	320106197801035610	Zhu	M
	230116198008174477	Chen	F
	120120198507133115	Dong	M

表 9-5(b)　Marriage **类实例的数据**

属性名	husbandID	wifeID
属性值	320106197801035610	230116198008174477

描述关联 isMarriedTo **的功能性函数** IsMarriedTo ()**的作用是返回关于某个** Person **的配偶的信息。由表** 9-5 **可知，因为关于** isMarriedTo **的信息被包括在第** 3 **方类** Marriage **中，所以要用存储所有** Marriage **类的实例的序列作为函数** IsMarriedTo **的输入参数（该序列用** List **模板三生成），以便从中检索出与当前** Person **为配偶的另一个** Person **的信息。下面即为功能性函数** IsMarriedTo()**的程序代码：**

```
char* Person::IsMarriedTo(MarriageList& ml)
{
```

```
    for (int j= 0; j < ml.Size(); j++)
// 如当前 person 实例的 ID 与 marriage 中元素男方实例的 ID 相等
        if (strcmp (ml[j]–>HusbandID(), id) = =0)
            return ml[j]–>WifeID(); // 查询成功，返回女方 ID
// 如当前 person 实例的 ID 与 marriage 中元素女方实例 ID 相等
        else if (strcmp (ml[j]–>WifeID(), id)
            return ml[j]–>Husband ID ()；// 则返回男方 ID
    return NULL； // 没有发现，则返回空指针
};
```

9.3.5 关联类的映射规则

不论该关联的参与度是什么，其映射规则都是：

- **将两方面类的标识属性都引入关联类中作关联标识属性。**
- **或者将两方面类的指针变量引入关联类中作数据成员。**

例 9-5：关联类的映射规则示例。

根据映射规则之一，由图 9–5 **中** Student **类、**Course **类和** Score **类的产生如下** 3 **个** C++**类定义：**

```
class Score {
public:
    Score(int, char*, int);
    char*     StudentID();
    int       CourseNo();
    int       Score(); // 不能与类名相同，所以全部大写加以区分
private:
    int       courseNo；//课程编号
    char      studentID[20]; // 学生身份证号
    int       score; // 分数
};

List3 <Score> ScoreList;

class Student {
public:
    Student(char*, char*,char);
    char*   ID();
    char*   Name();
    char    Gender();
// 描述关联 enrollCourse 的功能性函数，注 1
```

```
    ScoreList*    EnrollCourse(ScoreList&)；
private：
    char    id[20]; // 身份证号
    char   name[8]; //  姓名
    char    gender；// 性别
};

class Course {
public:
    Course(int, char*);
    int            No();
    char*      Name();
private:
    int   no; //  课程编号
    char name[256]; //  课程名字
};
```

程序注释：

注 1　**下面以实际数据为例说明为什么要这样定义功能性函数，以及如何编写其具体程序。表** 9-6（a）**是** Student **类实例的数据、表** 9-6(b)**是** Course **类实例的数据，表** 9-6(c)**是** Score **类实例的数据。**

表 9-6(a)　Student **类实例数据**

属性名	id	name	gender
属性值	320106198501035610	Zhu	M
	230116198508174477	Chen	F
	120120198507133115	Dong	M

表 9-6(b)　Course **类实例数据**

属性名	no	name
属性值	1001	Math.
	1005	English
	2012	Computer Hardware

表 9-6(c)　Score **类实例数据**

属性名	courseNo	studentID	score
属性值	1001	320106198501035610	87
	1001	230116198508174477	75
	1005	230116198508174477	67
	1005	120120198507133115	92
	2012	120120198507133115	83
	2012	320106198501035610	79

描述关联 enrollCourse 的功能性函数 EnrollCourse ()的作用是返回某个 Student 实例所学课程、成绩等信息。由表 9-6 可知，这些信息在第 3 方类 Score 中，所以要用 Score 类实例的序列作为函数 EnrollCourse 的输入参数（该序列用 List 模板三生成），以便从中检索出与当前 Student 实例有关的信息，下面即为功能性函数 EnrollCourse()的程序代码：

```
ScoreList* Student:: EnrollCourse(ScoreList& sl)
{
    ScoreList* tmp = new ScoreList; // 因为一个 student 可能学多门课程，所以返回一个序列
    for (int j = 0; j < sl.Size(); j++)
        if (strcmp(sl[j]–>StudentID(), id) == 0) // 注 1
            tmp–>Append（sl[j]）;
    return tmp;
};
```

程序注释：

注 1 如当前 student 实例的 id 与 Score 实例中的 studentID 相同，说明他选修了该课程，将有关实例指针加入到作为函数返回值的序列中。

导读：

① 即使同样使用 C++语言，从同一个数据模型设计出来的程序也会因人而异。这里介绍的只是认为较好的一种方法，它能确保数据满足完整性、一致性和最小冗余性的要求（关于其理论证明就不在此详述）。但是，这并不是惟一的方法，学员可从经验和其他教材发展自己的设计方法。

② 由于篇幅关系无法对所有可能的关联参与度的情况一一列举，但本着上述 5 条原则，绝大多数的实际应用问题可得到解决。

9.3.6 通过计算获取冗余信息

9.2.7 节指出，类图中的无条件性关联不能形成环路，以避免破坏数据的一致性和完整性。但是，如果确有必要获取冗余的信息，如欲在图 9–10 中获得关于 Teacher 教授 Student 的数据，则可通过在 Teacher 类中定义功能性 TeachStudent()获得。例 9-6 给出了该函数的代码。

例 9-6：获取 Teacher 教授 Student 信息的函数。

```
StudentList* Teacher::TeachStudent(CourseList& cl, ScoreList& scl, StudentList& sl)
{
    StudentList*  tmp = new  StudentList；// 存储教师所教学生的序列
    for (int j =0; j < cl.Size(); j++)  // 在课程序列中检索当前教师所教课程
        if (cl[j]–>TeachID(), id) == 0)  // 如教师 ID 等于课程实例中的 RK
            for (int k = 0; k < scl.Size(); k++)  // 在成绩序列中查找学该课程的学员
                if (cl[j]–>CourseNo() == scl[k]–>CourseNo()) // 如学员 ID 等于课程实例 ID
                    for (int m = 0; m < sl.Size(); m++) // 在学员序列中查找有该 ID 的学员
                        if (strcmp(sl[m]–>ID(), scl[k]–>studentID()) == 0)// 如找到
```

```
                tmp–>Append(sl[m]);// 将学生实例加入结果序列
        return tmp;
    }
```

因为教师教学生的信息与所有 3 个类的实例都有关，所有必须用 3 个序列作函数的参数。设计该函数的基本思路是：在各个 Course 实例中，根据 Teacher 的 ID 找到他所教授的课程，然后用该课程编号在 Score 的实例中，找到学习该课程的学生；然后再用该学生的 ID 在 Student 的序列中，查找 Student 的实例，将其加入到最后要返回的序列中。

9.4 数据模型的一致性和完整性

9.4.1 数据模型的概念一致性

数据模型是计算机系统描述数据对象、它们之间的相互作用，以及它们与周围事物的相互作用的数学模型。然而对同一个系统，因分析者观察角度不同，可能产生不同的数据模型。如果这些模型都正确描述了事物的本质，最后产生的数据定义应完全一致。也就是说，模型在一定程度上应该和分析人员的主观因素无关，这就是数据模型所要求的一致性和完整性。

一致性和完整性是对数据模型正确性的重要检验，如果在进行系统分析时，不同的人产生了不同的数据模型，导致不同的数据定义，则说明这些数据模型中有些是不完善、不正确的，也可能都不正确。

下面的例 9-7 说明了设计得好的数据模型具有内在的一致性。

例 9-7： 图 9–15 是关于图的数据模型之一，它表明图是 Vertex 类的聚集，Vertex 有个返身 Association 海个顶点可与多个顶点(包括自己)connect，但也可不与任何顶点 connect，因此这是一个条件性的关联。

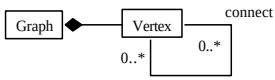


图 9–15 图的数据模型之一

根据 9.3.4 节例 9-4（4）中关联的映射规则，对于这种情形在设计 C++类定义时，要引入第 3 方类，其中用来自 Vertex 类的两个指针变量做数据成员，结果生成如下 Vertex 类和 Arc 类的定义：

```
// 图的顶点类
class Vertex {
public:
    Vertex();
private:
    // 在这里定义结点数据
};

// 引入的第 3 方类
class Arc {
public:
```

```
        Arc(Vertex*, Vertex*);
        Vertex*   Head() { return head; };
        Vertex*    Tail() { return tail; };
    private:
        Vertex *head, *tail; // (RK)
    };
```

由于 Graph **是一个包括可变数量顶点的聚集，所以要用一个** Vertex **的序列来表示它（参见** 9.3.3 **节），得到** Graph **类的** C++**定义如下：**

```
    # include "list_3.h"
    typedef List3<Arc> ArcList;
    typedef List3 <Vertex> VertextList;
    class Graph {
        Graph();
        VertexList&    Vertices() { return vertices; };
        ArcList&       Arcs() { return arcs; };
    private:
        // 注 1
        VertexList    vertices;
        ArcList        arcs;
    };
```

程序注释：

注 1　**虽然类图中只画出了** Graph **是** Vertex **的聚集，但是因为第** 3 **方类** Arc **类的存在是以** Vertex **类的存在而存在的，所以也必须同时被包括在** Graph **的聚集中。**

对于 Graph**，如把顶点和边都视为类，则还有第二种分析方法，图** 9–16 **就是这种分析方法所产生的数据模型。**

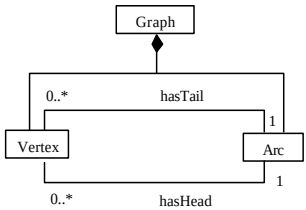


图 9–16　**图的数据模型之二**

该模型表明，图是顶点和边的聚集，在顶点（Vertex**）与边（**Arc**）之间有两个条件性关联，每条边（**Arc**）只能有一个头和一个尾，而每个顶点可以是多条边的头和尾，因为有的顶点可能不作边的头，有的顶点可能不作边的尾，所以这两个关联都是条件性的。**

根据 9.3.4 **节例** 9-4（2）**中的关联影射规则，从图** 9-16 **所示的数据模型可得到以下** C++**类定义：**

```
    class Vertex {
    public:
        Vertex();
    private:
        // 在这里添加关于顶点性质的数据成员定义
    };
```

```
class Arc {
public:
    Arc (Vertex*, Vertex* );
    Vertex*        Head() { return head; };
    Vertex*        Tail() { return tail; };
private:
    Vertex     *head, *tail; //  边的头和尾
    //   在这里添加关于边性质的数据成员定义
};
```

Graph 类的定义也包括了 Vertex 类和 Arc 类的序列：

```
typedef List3 <Vertex>   VertextList;
typedef List3 <Arc>      ArcList;

class Graph {
public:
    Graph();
    VertexList&    Vertices() { return vertices; };
    ArcList&       Arcs() { return arcs; };
private:
    VertexList    vertices;
    ArcList       arcs;
};
```

例 9-7 中由两个不同的数据模型得到的 C++类定义可以说是完全一致的，它表明两个模型都正确地反映了图的概念，具有内在的一致性。

9.4.2 数据一致性和完整性的动态维护

应用上述基于关联参与约束的设计准则可以保证数据的静态完整性、一致性和最小冗余性，但对象实例并不是一开始就有的，也不是永远存在的，它们均有生命周期。每个实例均在程序运行的某一时刻被生成，然后在某一个时刻被删除，或保留到最后把有关信息输出给用户。然而，关联的存在是以实例的存在为前提的。如果一方类的实例被删除（或增加时）而不将参与该关联另一方类的实例也随之删除（和增加），就会违背该关联的参与约束。

这就是说，数据的完整性和一致性需要动态维护。所谓维护，即根据关联的参与度同时增加或删除双方类的相关实例。

本节给出了各种参与度下动态维护数据完整性和一致性的规则，在编写程序时必须根据这些规则进行必要的操作。

（1）参与度为 1:1 的关联数据一致性维护规则，与例 9-4(1)的映射规则对应。

- 当对象一方生成（删除）实例时，另一方也要随之生成（删除）对应的实例。

例如，在例 9–4(1)中，如果将 Country 类的实例（Japan，Tokyo）删除，则要在 City 类中

将实例（Tokyo）删除。反过来，如果在 Country 类中新增一个实例 France，则必须同时在 City 类中相应地增加实例 Paris。

（2）参与度为 1:*的关联数据一致性维护规则，与例 9-4(2)的映射规则对应。

① 当参与度为*的一方删除某个实例时，应在参与度为 1 的对象中删除所有相关的实例。

② 当参与度为*的一方增加一个实例时，应在参与度为 1 的对象中增加至少一个相关的实例。

③ 在参与度为 1 的对象中增加实例时，在参与度为*的一方中必须有与之对应的实例。

④ 在参与度为 1 的对象中删除实例时，则应在参与度为*的一方中检查与之对应的实例，若未参与同样的关联，则必须同时予以删除。

例如，在例 9-4（2）中：

- 若将 Teacher 的实例（320105195001015610, Wang, M）删除，则必须将 Course 类中与它有关的两个实例（1001, High Math, 320105195001015610）和（1005, Discrete Math, 320105195001015610）同时删除。
- 若增加一个 Teacher 的实例 Jiang，则必须在 Course 类中同时增加一门由他所教的课程。
- 若增加一个 Course 的实例 Physics，如果该课程由教师 Qian 教授，但 Qian 不在已有 Teacher 的实例中，则在将 Physics 加为 Course 实例的同时，还应将 Qian 加为 Teacher 的实例。
- 若删除 Course 的实例 English，该课程由教师 Zhang 教授，经检查她除了教 English 外，并未教任何其他课程，因此在将 English 从 Course 中删除的同时，还要将 Zhang 从 Teacher 的实例中删除。如果是删除 High Math，因为教 High Math 的教师 Wang 还教授其他课程，则只要将实例 High Math 从 Course 中删除即可。

（3）引入第 3 方类表示关联的数据一致性维护规则，例 9-4(3)、例 9-4(4)和例 9-5 等 3 种情况都属于这一类。

与上述两种情况不同，因为这种情况中的关联的参与度有多种情况，无法一一列出其具体规则，只能提出维护数据一致性的一般原则：

- 当参与行为任一方删除（增加）实例时，首先要查询该实例是否参与关联，并找出另一方类中对应的实例，然后根据关联的参与约束和条件性，分别确定在 3 个类中应采取的删除（或者增加）实例的措施。

例如，在例 9-4（4）中，假定要将实例 Dong 从 Person 类中删除，首先必须查询第 3 方类 Marriage，发现他并没有参与 isMarriedTo 的关联，所以要进行的操作只是将该实例从 Person 类中删除。但是，如果要将实例 Zhu 或 Chen 从 Person 类中删除，采取的措施包括：在 Marriage 类中删除与它们有关的实例，然后再将实例从 Person 类中删除。

又如在例 9–5 中，假定要从 Course 中将 English 实例删除，首先在 Score 类的实例中查出选修这门课的学生有 2 人，但他们还选修其他课程，所以只需要进行 2 项操作：将实例 English 从 Course 类中删除，在 Score 类中将与 English 有关的 2 个实例删除。

总之，维护数据动态完整性和一致性是一项细致的工作，是建立数据模型和设计程序非常重要的内容。对软件技术人员来说，要考虑的问题包括：

- 在建立数据模型时写出详尽的数据一致性维护规则，为程序设计提供指导。
- 用大量数据进行比较彻底的运行测试。

实践经验表明，软件系统运行后要求维护和修改最多的就是与数据动态一致性和完整性有关的问题，其次是与用户界面有关的问题。如果不建立维护数据完整性和一致性的概念，编写的程序自然免不了漏洞百出。

9.5 用户界面的设计

9.5.1 用户界面的作用

本章一开始就指出，编程作业的任务包括两个方面：一是把处理的数据表示为计算机程序所能处理的形式——数据模型；二是把系统处理数据的功能表示为计算机所能执行的形式——函数。但是，针对数据类定义的成员函数一般都比较简单，一般无法直接用它们完成系统要求的复杂功能。因此，程序设计首先需要将系统功能逐层分解为子任务，直到能用类的成员函数完成这些子任务为止。其次，用户必须能以交互方式调用这些函数去执行有关的功能，系统则必须将数据处理的结果以用户能方便阅读的形式提供给用户，这些都是用户界面的任务。

常用的用户界面形式有：

① 菜单（menu）：菜单可以图形方式提供，如 Windows 环境下运行的各种程序，用户用鼠标或其他方式（如触摸屏幕）通过选择菜单图形启动相应的函数。菜单也可以是字符方式，用户只要按提示文字输入单个数字或字母就可执行有关操作，如自动提款机。

② 对话框或表格填写方式:典型的就是网络搜索程序，用户在系统提供的表格中填入数据，点击各种选项，将命令和数据提交系统执行有关运算。

③ 命令行语言：用户输入按某种格式组成的语句，如 UNIX 操作系统下，微机的 DOS 界面中，用户输入各种命令。

④ 自然语言：这实际上是命令行语言的发展。命令行语言与编程语言一样同属形式语言的范畴，对句法和词法都有严格的要求。相信学员们对此已深有体会。因此，掌握命令行语言对一般人来说并不容易，而使用自然语言可使未经专业训练的一般用户也能方便地使用程序。

⑤ 直接操作可视符号和图形：最典型的就是游戏程序，用户通过操纵图形（游戏中的角色）把数据和命令传达给系统，使系统执行有关运算。

无论哪种形式的用户界面，其作用都是:

- 提供用户操作指令，使用户可以有所选择。
- 得到用户的输入（包括数据和指令），并将其转换为程序所能接受的形式。例如，用户在屏幕上用鼠标点击某个图形，或移动游戏中的某个角色后，由用户界面经过复杂的运算将其转换为具体数值，作为调用函数的参数。
- 将结果提供给用户，如果是一般的计算程序，则用户界面必须把结果打印为与纸面形式相同的报表形式。如果是涉及图形的程序，用户界面则还应绘出相应的图形。

正如下面所要讨论的那样，用户界面是确保程序能可靠稳定运行的重要因数。但是对用户界面的设计往往容易重外表而轻内部逻辑。许多程序之所以无法应付用户操作中可能发生的各种误操作、漏洞百出，一个很重要的原因就是用户界面设计得不够严密。

9.5.2 UCD

用户界面的设计首先取决于用户如何使用程序，因此设计用户界面的前提是对系统功能进行分解，将复杂任务逐级分解为层次较低的子任务。因此下面首先介绍 UML 中的用况图（Use Case Diagram，UCD）。它以一种层次方式分解和描述系统的所有功能，是设计用户界面的重要工具。

导读：

因为许多书中对 UML 模型中的术语的翻译各不相同，本教材对 UCD 图中的有关术语将统一采用下述的中文名称。

- system：系统
- actor：使用者
- use cases, services：服务
- extend：扩展
- include：包括

（1）UCD 的组成

Case-3-2：学籍管理系统的 UCD

图 9–17 为 9.2.10 节图 9-13 所示的学籍管理系统的 UCD 的例子，它表明该学籍系统有 3 个主要功能：管理学生、管理教师、管理课程。其中，教务管理人员和教师将使用所有 3 种服务，而学员只使用两种服务。

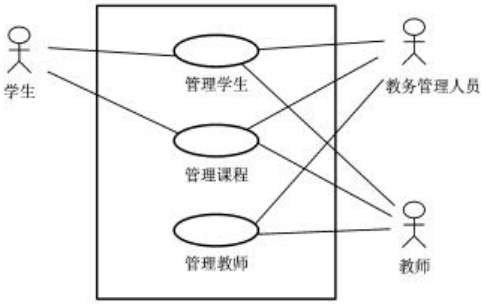


图 9–17 学籍管理系统用况图

图 9-17 表明，UCD 的组成包括 4 种主要成分：

- 系统。图中方框部分。表示程序和系统本身。
- 使用者。图中的人形图符，一般表示使用系统的人员。但是，如果系统与之交互的是另一个系统，如某个数据库系统、某个程序，则也将其视为使用者（一个关于非人类使用者的例子参见第 11 章的课程作业二）。

• 服务。图中的椭圆图形。表示系统为使用者提供的服务。系统可提供的服务很多，但在系统方框中只列出最高层次的服务。例如，管理课程包括若干更具体的服务，如登记学员选课、登记课程成绩等。这些服务将用另外的子图表示。

• 连线。图中的连线表示使用者使用服务的情况，同时也表明服务分解的情况。

（2）服务的分解

UCD 中列出的最高层的服务往往都是比较复杂的，因此需要进一步分解以说明其具体实现途径。UCD 提供了两种分解功能的方法：

• 扩展（extend）说明某个服务的完成具体可通过完成某个子服务来实现。图 9-18 表明管理教师的服务可分解为增加教师、删除教师或者教师列表等子服务。这些子服务既有共性（即都是属于管理课程的范围），又互不相同。执行它们中的任意一个都可被视为执行管理教师的服务。如果用逻辑运算来描述，extend 表示的是一种“或”的关系

• 包括（include）说明完成某个服务的前提是完成其所有子服务。图 9-19 表明增加教师的服务可分解为以下子服务：检查教师序列确定新增实例不在序列之中，检查课程序列看该教师所教的课程是否已有他人在教，只有这些条件都确定后，才能增加教师，并随后增加他（她）所教的课程（因为课程和教师之间关联的参与约束为 1:*，根据 9.4 节提出的维护数据一致性、完整性的规则，在增加 Course 类实例的同时，必须根据情况增加对应的 Teacher 类实例。完成所有这些子服务是执行增加教师服务的必要条件，或者说 include 表示的是一种“与”的关系（但是要说明的是，UCD 图中并不表示子服务的执行顺序）。

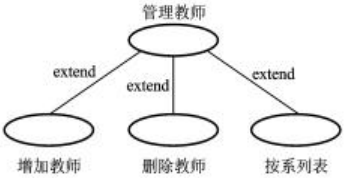


图 9-18 管理教师服务的分解子图

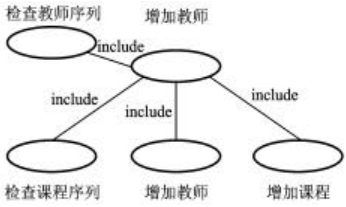


图 9-19 增加教师服务的分解子图

（3）设计 UCD 的步骤

设计 UCD 的一般步骤是：

• 确定系统的边界，即确定系统功能覆盖的范围。如果执行者是人类，要确定哪些任务是交由系统完成的；如执行者是另一个计算机程序，则要确定它们各自都执行哪些任务。

• 列出主要的执行者，即系统的主要用户。注意这里所说的执行者并不一定是亲自使用系统的人员。例如，图 9-17 的学籍管理系统 UCD 中，学员使用系统选修课程，但是他们并不一定亲自去使用系统，大多数情况下是由教务管理人员将学生选课数据输入计算机。

• 确定执行者使用系统的目的，即使用系统要执行的所有服务。

• 对服务进行分析，确定它们之间的层次，然后列出 3~9 项作为系统的主要服务或者功能。

• 用子图描述各主要服务，使服务之间关系形成一种层次结构。

9.5.3 设计用户菜单

导读：

虽然现在已经有各种界面开发工具使编程人员能方便地编写出漂亮、美观的用户界面，但是这些工具并不能自动保证菜单内部逻辑的严密性。因此，设计性能良好，能应对各种意外情况的用户界面仍然是编程人员的职责，希望学员今后不论学习什么界面设计工具，始终要清醒地认识这一点。

菜单式用户界面是采用最多的用户界面形式，它用把可供选择的功能选项分门别类的组成一个层次结构，也就是通常所说的菜单树(menu tree)**。图** 9-20 **就是菜单树的示意图。它提供了一种将** UCD **映射为** C++**程序的最自然的机制，因为从概念上来说，它描述的是系统功能的“或”分解，与** UCD **中的** extend **分解结构是相似的，而从实现上来说，它和** C++**语言中的表示多项选择的** switch **语句相似，都是多叉树的结构。根据它可以很自然地自上而下地设计出各级菜单子函数。**

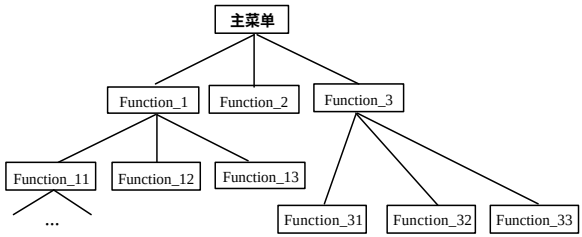


图 9–20 **菜单树**

在设计函数时要尽可能地从高层次功能概念出发，然后逐步细化，最后细化为可以用类的成员函数完成的任务。一个好的菜单程序的设计应该有很好的可读性，读起来就像是读程序或者算法的说明一样。

下面就是用以实现图 9–20 **所示菜单树中函数的示意（并非实际可运行程序）。**

```
void Main ()
{
      char    choice;
     do  {
          MainMenu(); //  打印主菜单指令的函数
          scanf("%c", &choice);
          switch (choice) {
          case '1': Function_1( ...);
                    break;
          case '2': Function_2( ...);
```

```
                    break;
            case '3': Function_3( …);
            default: printf（"the input %c is not legal choice\n", choice）;
            }；
        } while （choice != ' '）;
}
```

下面是打印主菜单指令的函数 MainMenu()**的程序代码示例：**

```
#include <stdlib.h> //  定义系统函数 system()的头文件
void MainMenu()
{
    system("cls"); //  系统函数，用于清屏，每次可以从窗口顶端开始显示
    printf（"User Main Menu: \n"）;
    printf ("0: exit the system \n");
    printf ("1: Execute Function 1\n");
    printf ("2: Execute Function 2\n");
    printf ("3: Execute Function 3\n");
}
```

在函数 Function_1()**中则用** switch **语句表示其子菜单：**

```
void Function_1(… )
{
     char    choice;
    do {
        SubMenu1(); //  打印子菜单–1 的函数
        scanf("%c", &choice);
        switch (choice) {
        case '1': Function_11( ...);
                    break;
        case '2': Function_12( ...);
                    break;
        case '3': Function_13( …);
         default: ;
        }
    } while (choice != '0');
}
```

函数 Function_3()**内部结构与其类似，函数** SubMenu1 **（）的内容与打印主菜单的函数** MainMenu **类似。**

因此可以说，UCD **是指导用户菜单设计的方法和工具，而** switch **语句则为具体实现用户菜单的工具。以下是设计用户菜单的主要步骤：**

- 将用户使用程序的情况表示为 UCD。
- 将 UCD 转换为菜单树，将系统的主要服务用主菜单表示，而将表示服务分解的子服务用子菜单表示（表示 include 关系的子服务将转换为一系列函数的顺序调用）。
- 用 switch 语句将菜单选项与程序函数对应起来。

9.5.4 验证用户输入

用户界面是计算机程序和用户进行交互的惟一通道，因此，除了按用户指令执行程序的功能性函数外，一个完善的用户界面还必须具有以下功能：

- 内在逻辑结构组织合理，能引导用户一步一步自然地进行选择。因此指令和屏幕提示应尽可能详尽，如应明确指出输入数据的格式、允许取值的范围等。
- 对用户的各种误操作有预防措施，使其不至于引起程序的误动作。
- 尽可能地精确诊断用户操作失误和问题的所在和性质，给出尽可能详尽的解释。
- 能为用户提供完善的帮助措施，如命令、词汇的解释等。

下面着重讨论如何防止用户误操作的问题。

（1）验证用户输入的指令

在凡是需要用户输入指令的地方，都需要进行验证。对于命令行方式的界面，对用户输入指令语句的检测需要用到类似编译器的命令解释程序。因此，一般都尽可能地采用菜单方式，用户只要根据系统提示输入一、二个数字或者字符即可，大大减少了输入指令错误的可能。但是，即使这样，对输入内容进行检查和防止程序误操作仍是不可缺少的工作。具体措施包括：

- 在 switch 语句中安排 default case，当输入了合法选择集以外的指令时，程序就会执行这里的语句。如果在这里能安排明确指出操作错误的内容，则更好。例如，图 9-20 所示函数中的打印语句不仅告诉用户输入指令是错误的，而且还将用户输入数据同时也打印出来。这个措施看来不起眼，但符合上述对用户错误给出尽可能详尽解释的原则，对用户是很有帮助的。
- 尽可能按字符格式处理输入数据。例如，在 switch 语句中，choice 被定义为字符型变量，程序按字符格式对读入的用户指令进行处理。若用户输入合法选择集以外的字符，程序会执行 default case 中的语句。但是，若将变量 choice 定义为整数型（一般学员多会这样做），当用户输入字母（如 a、b）而不是数字时，由于系统读取数据语句不能处理字母，程序尚未运行到 default case 处，就已经出现错误了。

（2）验证用户输入数据

对用户数据的检验包括两个方面的内容：

- 数据格式的检验。系统提供的输入函数（参见 6.4.9 节）不能对输入数据进行格式检验。例如，如果读取数据语句中标明的是输入一个浮点数，则不管用户输入的是数字还是字母，输入函数都按数字来处理它。如果用户输入的不是数字，而是字符，那么，小则产生错误计算结果，严重的还会引起程序崩溃。因此，一般做法都是一律把输入数据当作字符读入，然后用程序检查其格式是否正确，最后再用系统函数 atoi()或 atof()将字符串转换成整数或浮点数。
- 数据值域的检查。任何数据都有一个合理的取值范围，即其值域。例如，输入 Person 类实例的性别数据时，除了在提示中指明其合理取值范围是大小写 M、m 和大小写 F、f 外，还

要对读入数据进行检验，剔除不合格的输入数据，使其不要进入程序。否则会影响按性别进行检索。又例如，课程成绩不能为负数。这些人类已习以为常的繁琐小事，对界面设计来说却是非常重要的。如考虑不周密，则往往会因用户一个小小的输入数据错误而引起程序的严重错误。1972 年 6 月的《计算机世界》上一则题为“操作员手指的一滑使税收损失 30 万美元”的报道就是一个绝好的反面例子。该报道称，1972 年 6 月美国某市在使用某个程序计算税率时，操作员的右手小指无意中误按了 p 字母键，使计算机将 950 美元的价格误认为是 7000950，从而导致了巨大的财政损失。

9.5.5 输出数据的可读性

指导用户使用程序的有关指令，将运算结果和运行过程中各种诊断信息以易读、易懂的方式提交用户，也是用户界面的重要任务。下面就是一些有用的原则：

- 界面风格要一致。就以 DOS 界面下菜单为例，如果主菜单规定了输入字符‘0’是停止程序的命令，各级子菜单也要以字符‘0’为退出本级菜单、返回上级的命令。风格一致的菜单可以减少用户的困惑，使系统学习变得十分容易。
- 伴随指令提供详细的提示，如要求用户输入某个数据时，要注明输入的是什么数据，用什么格式，合法的值域范围是什么等。
- 提供尽可能完善的帮助功能。和指令中的提示不同，帮助功能是对系统功能、使用方法的完整的、系统的介绍。
- 详细说明输出数据。将最后结果打印为格式美观的报表形式固然重要，而小处也不应无视。例如，在打印变量的语句中加上变量名字或说明对编程人员来说是举手之劳，但对程序用户来说却方便不少。
- 诊断用户操作错误，并且尽可能确切地指出错误的性质以及纠正的措施。相信绝大多数学员在使用 VC++编译器时，一定深感确切指出错误性质对用户来说是多么重要。

用户界面的设计不仅是技术问题，还涉及人类工程学（ergonomics）、心理学等多方面的知识，希望学员从学习编程伊始，就养成考虑到各种细枝末节之处的观念，除了阅读关于界面设计的专著外，还要通过观察用户界面设计得比较成功的商品软件来逐渐掌握用户界面设计的艺术。

> **导读：**
> 对于复杂程序和系统，还需要用 UML 建立数据处理过程的动态模型，才能设计出逻辑严密的程序代码，有兴趣要深入了解者可参考关于系统分析和设计以及关于 UML 的专门论著。

9.6 程序的检测

9.6.1 程序错误的种类和原因

编写程序工作只是整个软件开发过程的一小部分，一般来说，程序测试工作要占整个软件开发工作量的 1/2~2/3，这还没有将编写过程中的编译、纠错等工作算进去。根据错误在程序开

发过程中发生的阶段，大致可将其分为：编译错误、连编错误和运行错误。

常见编译错误包括两类，一类是语句的句法和词法有错，如不完整的句子、不配对的括号、没有定义过就出现在计算式中的变量、函数调用语句中的参数与定义不符等。另一类是由键盘操作失误或不仔细引起的错误，如该大写的地方没有大写、变量名字写错等。编译错误的发现是最简单的，编译器一般都能指出错误发生的位置，但是对错误的解释却往往言不及义。因此，学习解读编译诊断信息是编程人员必须掌握的基本技能。

常见的连编错误大致有两类。对第一类错误，VC++编译器会显示如下的信息：

```
Linking...
main.obj : error LNK2001%: unresolved external symbol "某个函数名字 ..."
(??A?$List3@H$09@@QAEAAHH@Z)
Debug/program.exe : fatal error LNK1120: 1 unresolved externals
Error executing link.exe.
```

造成这种错误的原因可能有两个：被调用的成员函数仅在头文件的类里予以声明，但没有在 cpp 文件里定义实际代码，成员函数调用语句和定义在名字、参数类型、数量方面不一致。

对第二类错误，VC++编译器会显示如下的信息：

```
Linking...
student.obj : error LNK2005:"某个变量名..." (?MaxLength@@3HA) already defined in xxx.obj
Debug/program.exe : fatal error LNK1169: one or more multiply defined symbols found
Error executing link.exe.
```

造成这种错误的原因是不同文件模块里有相同名字的全局变量。本教材在 6.4.3 节的编程规范化实践–5 已规定不要定义全局变量，但学过其他编程语言的学员常常还是会犯这类错误。

对于连编错误的发现比较困难，系统一般只能指出发生错误的模块，但错误的发生往往不在这个模块里，而是在别的模板里。举例来说，对一个 class 没有定义默认构造函数，系统指出的是调用该默认构造函数的模块有错，而这默认构造函数的调用往往又是隐形的，并没有一个明确调用它的语句。学习发现连编错误也是编程人员必须掌握的基本技能。

程序经过编译、连编无误后能够运行后，就要使用人工或自动手段反复运行它，以检验它是否满足规定的需求，清楚了解预期结果与实际结果之间的差异。运行错误主要来自两个方面：

① 程序语法设计错误。这类语法错误是编译器不能诊断出来的。最常见的就是内存使用错误。其标志是程序停止运行，系统显示图 9–21 所示的提示窗口。

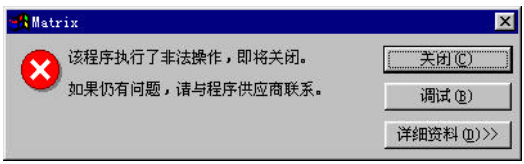


图 9–21 内存错误提示窗口

造成这种现象的主要原因为：

- 对指针变量没有分配内存空间就给它赋值。
- 调用成员函数的指针变量值为空。
- 数组元素的下标值超出其定义长度或者为负值。
- 字符串没有结尾符。

另外常见的错误还有将程序控制语句中的逻辑判断式中的==运算符误写为=运算符，编译器对这种错误是发现不了的，而且这种错误也不会引起程序停止运行，但却会使程序表现异常，得不到预期的结果。这种错误的检查即使对有经验的编程人员来说也颇费周折，因此要特别注意。

② 程序算法设计错误，不能得到预期的结果。对于这种错误可用已知结果的数据进行检验或用系统提供的调试（debug）功能，以步进方式运行和跟踪程序的执行过程，查看中间数据。关于 VC++编译器提供的调试功能和使用方法简介详见 9.6.3 节。

9.6.2 程序运行检测步骤

程序检测应自下而上进行，先从各个类开始，然后是子函数，最后才是整个程序。

- 类是 C++程序最基本的构造元素，因此首先应对定义的各个类进行测试。检查类实例生成与数据成员获取的正确性又是检测中最基础的工作，不要等最后程序运行结果不对了，才发现根本原因是数据没有正确地进去。检查类实例生成情况非常简单，也非常方便。只要写一小段读取数据文件，打印类实例的程序即可。例如，第 7 章习题中就安排了读取数据，生成 Student 类和 Teacher 类实例以及输出实例数据的练习。

在确保数据能正确地输入实例和从实例中取出后，可编写一些简单函数，逐个测试类的功能性函数。

如果程序中有若干类似的类，可以先编写一个类进行彻底的测试，确保其中的算法以及输出输入功能正确无误后，再编写其他类似的类就会有经验可以借鉴，要比全部编写完所有的类再来逐个进行测试效率高的多，发生错误的机会要少得多。

- 单元测试。由下而上对菜单树中的子功能函数进行检测，看其能否正确运行和实现既定功能。如果不进行单元测试就进行集成测试，会使本来很容易发现的问题复杂化，增加开发的困难。
- 集成测试。对整个系统的功能逐项进行测试，以检验模块之间的接口。一定要确保各个模块都能正常运行后，再运行整个程序，进行集成测试。

在有条件地方，最好安排专人与编程人员进行背靠背的测试，这样就容易发现那些在编程人员看来是习以为常的问题，更全面地考验编程人员的设计思想。

9.6.3 用 VC++编译器的调试功能跟踪程序运行过程

所有的 C++编译器都提供能跟踪程序运行过程、检查程序运行错误的调试功能。本节对 VC++编译器的调试功能做一简介，以便学员了解调试的工作原理。

VC++编译器调试功能的具体使用步骤如下：

① 在源代码中设置断点（break point）。编程员估计错误可能发生在哪些语句并在那里设

置断点，程序运行到此地就会停下来，编程人员随后可使程序以“步进”方式运行，以便观察运算过程，查看各变量的值，甚至可以给变量重新赋值，观察新的计算结果。设置断点的方法如下：

打开要检测的源代码文件，将光标移至要检查的语句前，然后打开 VC++编译器主菜单【Edit】的下拉式菜单，选取其中的【Breakpoints】项（图 9-22），然后在系统打开的对话框“Breakpoints”中点击标签【Location】，点击“break at”下方的右向箭头，系统便显示光标所停的行号，点击该行号，该行就进入对话框下方的“breakpoints”列表中，同时在源代码文件中用红点标出断点所在的语句（见图 9-23）。

如果要对程序中某个变量值的变化过程进行跟踪，则可以点击对话框“Breakpoints”中标签【Data】，然后在“Enter the expression to be evaluated”下方输入变量名。

② 断点设置好以后，点击主菜单【Build】下的【Start Debug】选项，在该菜单旁边还会弹出一个小菜单（见图 9-24），选择跟踪方式后，程序即开始运行并进入调试窗口。窗口中显示主函数的源代码，光标指出当前正在执行的语句。程序运行到哪个文件的某个语句处，哪个文件的源代码文件就会自动打开在调试窗口中（图 9-25）。

③ 选择跟踪程序运行的方式：

• Go：从当前语句一直执行到断点，或者需要用户交互输入的地方，或者直到程序结束。若一开始就选此项，则程序从开始一直运行到所遇到的第一个断点。

• Step Into：使程序以步进方式运行。若遇到某个函数调用语句时，此种执行方式使程序跟

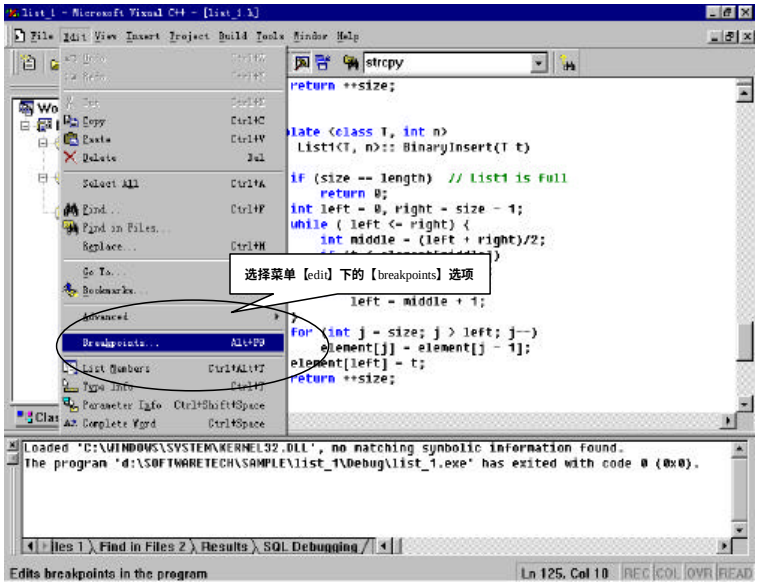


图 9-22 选择菜单上设置断点功能

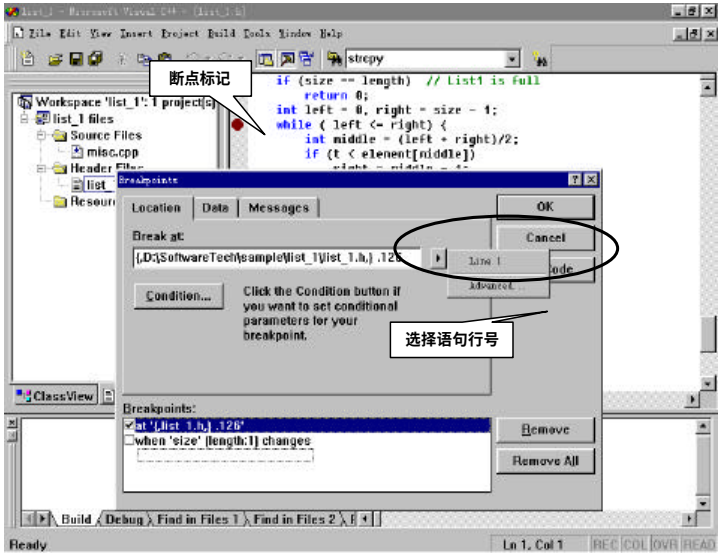


图 9-23 在对话框中选取设定断点所在行号

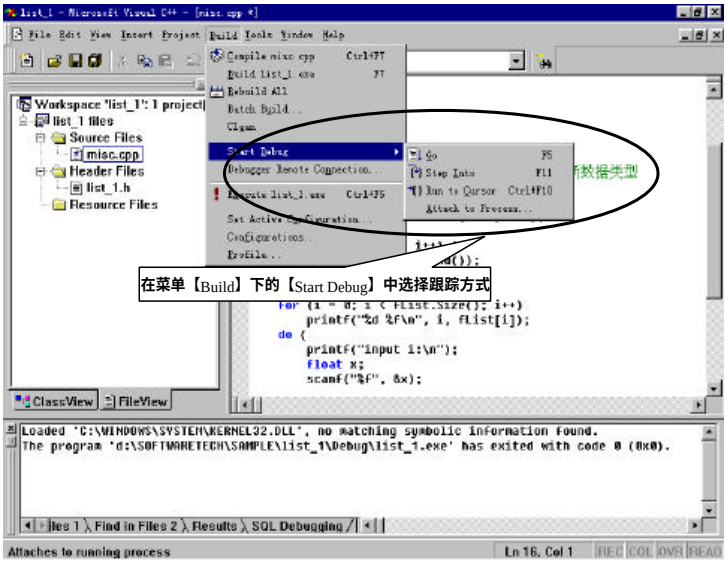


图 9-24 选择跟踪方式

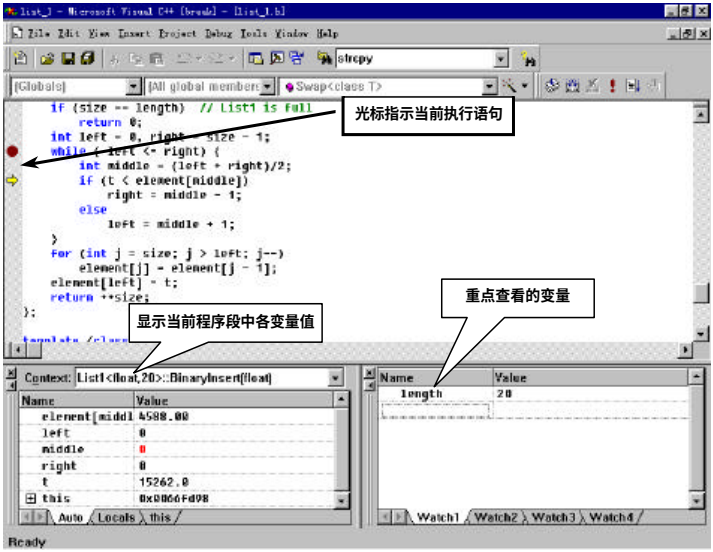


图 9-25 源代码查看窗口和变量查看窗口

踪进入被调用函数中继续进行。若一开始就选此项，则程序从主函数的第一条语句开始执行。

• Run to Cursor：编程员可以把光标移至程序中某个语句处，然后选择此项执行方式，程序便从光标原来所在位置一直运行到这里（如中间遇到断点则执行断点）。若一开始就选此项，则程序从主函数的第一条语句开始执行到光标所在语句。该选项相当于临时增加新断点。

• Step Over：若遇到某个函数调用语句时，此种执行方式使程序跟踪跳过被调用函数，在下一个语句继续。

• Step Out：当跟踪进入某个函数中后，此种执行方式可以使跟踪很快执行完函数中其余的语句，回到函数被调用的程序模板中继续进行。

④ 观察变量值。如图 9-25 所示，在调试窗口左下方“context”窗口中，系统会显示当前所在程序段中的各变量的名字和它们的值，随着程序用“Step Into”方式被执行，可以清楚地观察到这些变量值变化的情况。如果变量较多，编程员可以在调试窗口右下方 Watch 窗口中“name”栏目下方输入特别关注的一些变量名，从而可更方便地查看这些变量值的变化情况。

⑤ 如要停止调试，可以选择主菜单【debug】下的【Stop Debugging】或者【Restart】。但是，用调试功能以步进方式运行程序有时会显得特别繁琐，费时较多。而且有很多时候系统报告的错误位置也并不准确。因此有经验的编程人员有时会采用另一种简便方法，在程序中认为可能是引起错误的地方安排一些打印中间结果的语句，根据中间结果检查可能存在的程序错误。检查完毕，一般也并不删除这些打印语句，而是用注释号将其作为注释。一旦以后要再次检测，可以很方便地将它们恢复出来。

导读：

① 关于 VC++编译器调试功能更详细的使用方法，可参考用户手册或者其他专门参考书。

② 关于其他 C++编译器调试功能的使用方法，原理基本相同，但具体操作指令可能不同。

9.6.4　测试数据

用实际数据对程序进行彻底的测试是确保程序正确性的重要措施。因此，在设计程序时就要考虑到如何进行测试，制备测试数据，而不是在程序编制完以后再来找测试数据。用于测试的数据必须满足以下条件：

- 与实际情况完全一致，不要随意。在编程中最常见的弊病就是为图方便，总是随便输点什么简单数据进去，这就使很多问题无法暴露，而问题发现得越晚，纠正它所付出的代价也就越大。
- 数据要能涵盖所有的程序控制逻辑分支。测试数据的制备要与程序设计同时进行，这样程序设计才能比较全面地考虑所有可能的情况，而最后的测试也能对程序进行有效的检验。
- 数据要包括可能的错误，以检验程序的容错性。一个好的程序不仅能对正确的输入数据给出正确的结果，还要能对错误的输入数据有耐受力，首先程序不会因此崩溃，其次不会产生严重的后果。因此测试数据不仅要包括正确的数据集，还要包括各种可能的错误。

9.7　编程作业的文档工作

软件是程序以及开发、使用和维护程序所需要的全部文档。软件技术人员的职责不仅是编写程序，还必须提供所有与程序有关的文档。由于企业一般都有自己的文档规范，国家对软件工程中的许多文件格式也有标准，所以教材在这里不作深入讨论。但是就本课程的教学目的来说，希望学员能通过课程作业，初步学会以下文档的编制工作：

- 系统分析说明。
- 数据模型。
- UCD。
- 程序设计说明书（包括模块说明）。
- 程序源代码（包括注释）。
- 测试用例或测试数据。
- 程序使用说明。

习　题

知识点：

① 练习用类图建立数据模型和定义 C++类。

② 练习用 UCD 分解系统功能和设计用户界面。

1. 图 9-13 给出了 Case-3 的学籍管理系统的数据模型，在其基础上写出各个类的详细定义、然后将其映射为 C++类定义、详细列出维护数据动态完整性和一致性的规则。

2. 图 9-17、图 9-18、图 9-19 给出了 Case-3 的学籍管理系统的部分的 UCD，在其基础上完成全部的 UCD。

3. 根据下面的描述建立系统的数据模型，并写出 C++的类定义。

- 在一个车间里（machine shop）里有两种工人：操作工（operator）和维修工（maintainer）。
- 车间有若干设备（equipment）。
- 每个操作工只在一台设备上工作。
- 每个维修工负责若干设备的维修工作。

第 10 章　树和图的 C++模板

知识点：

① 数学概念上，树是图的子集，而在程序实现上，实现树运算的数据结构和函数是实现图的运算的基础，许多函数程序都需要应用树的数据和函数，所以本章重点介绍根树的模板。

② 关于图的算法和程序都比较复杂，视情况可将其作为选读内容。

③ 本章还比较深入地讨论了递归概念，作为对教材中迄今为止所涉及的各种递归概念的一个总结。

第 8 章介绍了线性数据结构，它仅为元素简单的罗列，并不反映它们之间的内在联系。较复杂的数据结构，如树和图，除了有存储数据对象的作用以外，往往还能反映它们相互关系或作用，因此对解决实际问题有重要意义。本章将介绍树和图的 C++模板程序。

由于树和图的概念和程序中大量应用了递归的概念，因此，本章还专门用一节总结递归概念。

许多关于数据结构的教材对二叉树讨论较多，而对普通根树介绍较少，但对于解决工程问题来说，还是根树应用较多，所以本章将以介绍根树为主，二叉树的内容视情况可作为选读。

本教材用 C++的类模板方法提供的树和图的模板程序与具体数据类型无关，光盘中有可运行的模板程序供学员下载，不仅可方便学习，还可直接用于解决各种实际问题。

10.1　根树模板

10.1.1　根树的数据模型和 C++模板

5.1.7 节介绍了根树的数学概念，9.3.3 节中给出了根树的数据模型，并由该数据模型生成了 C++类定义。这说明即使是对树这样的纯数学概念，数据模型仍然是计算机程序设计的基础。在前述内容的基础上，本节给出一个与结点数据类型无关的根树模板：

根树模板：

```
// GTree.h , 对应文件可从光盘上下载
#include "list_3.h"
template <class T> class GTree {
public:
      GTree(T*): data(t) {};
      T*            Data() { return data; };
     GTree*         operator[ ](int j) { return nodes[j]; };
```

```
    int             Nodes() { return nodes.Size(); };
    int             Insert(T, T*);   // 根据结点数据值在某结点下插入一个结点
    List3<T>*       DepthFirst();   // 深度优先遍历函数，注 2
    List3<T>*       WidthFirst();   // 广度优先遍历函数，注 2
    List3<T>**      AllPaths(int& no);   // 计算从根结点至所有叶结点的路径，注 3
protected:
    T*              data; // 注 1
    List3<GTree>    nodes;
    // 注 4
    List3<T>*       Leaves();                  // 返回所有叶结点
    void  Path(List3<T>**, int&);              // 返回从当前结点至各后裔叶结点的路径
};
```

程序注释：

注 1　该模板定义与 9.3.3 节中给出的根树的类定义结构完全相同，但是采用了模板的形式，用指针形式表示结点数据，这样模板就可适用于任何的类。

注 2　深度优先和广度优先遍历的结果均生成一个关于结点数据的序列。

注 3　函数 AllPaths 的作用是返回从根结点至各叶结点的路径，每条路径用一个结点的序列 List3<T>表示，而二重指针表示返回的是一个由若干序列组成的数组，参数 no 为引用型，以便返回该数组长度。

注 4　这两个私有函数是供函数 Paths 调用的内部函数。Leaves 的作用是列出所有的叶结点，函数 Path 的作用是用类似深度优先的方式列出从当前结点至各后裔叶结点的路径。

导读：

该模板中多处应用了 List 模板，表明 List 是一种最基本的数据形式，而该树模板本身又会被更复杂的数据结构——图所继承或包含。注意这种在一个模板中引用其他模板的方法，能方便地用简单的数据结构组成复杂的数据结构。

以下是模板成员函数的定义：

```
// 插入结点的函数示例
// 根据结点值查找插入结点
template <class T>
int GTree<T*>:: Insert(T    t,    T*    child)
{
    int  done  =  0;
    if  (data  ==  t)  {   //  如当前结点值等于给定值，将新结点插入其子结点序列中
          nodes.Append(new  GTree<T>(child));
          done  =  1;
     }
     else
```

```
// 若当前结点值不等于给定值，则用递归方式对每个子结点调用本插入函数
// 若有某个子结点插入成功，则返回 1 并停止执行循环。
        for (int i = 0; i < nodes.Size() && !done; i++)
            done = nodes[i]->Insert(t, child);
    return done;
};
```

10.1.2 根树的广度优先遍历函数

在 5.2.7 节中讨论了根树广度优先遍历的数学概念，下面是根树模板中广度优先遍历成员函数的程序：

```
template <class T>
List3<T>* GTree<T>:: WidthFirst()
{
    List3<T>* tmp = new List3<T>; // 生成一个序列保存访问过的结点数据，
                                  // 并作为函数返回值
    List3<GTree> nodeList; // 用以记录各层访问过的结点
// 首先将当前结点数据分别加入结果序列
    tmp->Append(data);
// 将当前结点加入访问过的结点序列中
    nodeList.Append(this);
  int i = 0, j = 1;
  int k, l, m, sum;
   do {
       sum = 0;
     for (k = i; k < j; k++) {
         l = nodeList[k]->nodes.Size();
        sum = sum + l;  // 注 1
       for (m = 0; m < l; m++) {
        // 将各子结点中的数据记录到序列 tmp 中，
        // 并将结点本身记录到序列 nodeList 中
            tmp->Append(nodeList[k]->nodes[m]->data);
            nodeList.Append(nodeList[k]->nodes[m]);
        }
      }
     i = j; // 注 2
     j = j + sum;
  } while (sum != 0); // 注 3
   return tmp;
```

```
}
```

程序注释：

注 1 **变量** sum **起一个累加器的作用，累计子结点的总数。**

注 2 **改变下一次循环的起点和终点。新起点等于上一层的终点** j**，新的终点等于上一层的终点** j **加上本层所有的结点数量** sum**。**

注 3 **当所有结点都没有子结点时，就到了树的最底层，整个遍历结束。**

为便于理解上述函数，特对图 5-18 **所示根树进行广度优先遍历，步骤及中间数据如表** 10-1 **所示。**

表 10-1 **对图** 5-18 **所示根树进行广度优先遍历的过程**

A	i=0，j=1，**加入** *A* **的** 3 **个子结点，** l = 1, sum = 3											
A	*B*	*C*	*D*	i=1，j=4，**依次扫描结点** *B*、*C*、*D*								
A	*B*	*C*	*D*	*E*	*F*	**加入** *B* **的** 2 **个子结点，** l = 2，sum = 2						
A	*B*	*C*	*D*	*E*	*F*	*G*	**加入** *C* **的** 1 **个子结点，** l =1，sum = 2+1=3					
A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*	*K*	**加入** *D* **的** 4 **个子结点，** l = 4，sum = 2+1+4=7	
A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*	*K*	i=4，j=4+7，**依次扫描结点** *E*~*K*	
A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*	*K*	*M*	*N* **加入** *G* **的** 2 **个子结点，** l=2, sum=2
A	*B*	*C*	*D*	E	*F*	*G*	*H*	*I*	*J*	*K*	*M*	*N* i=11，j=11+2，**扫描结点** *M* **和** *N*
A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*	*K*	*M*	*N* **全部结点扫描完毕，遍历结束**

10.1.3 根树的深度优先遍历函数

在 5.2.7 **节中讨论了根树深度优先遍历的数学概念，下面是根树模板中深度优先遍历成员函数的程序：**

```
template <class T>
List3<T>* GTree<T>:: DepthFirst()
{
    List3<T>* tmp = new List3<T>; // 保存访问过的结点数据，并作为函数的返回值
      tmp->Append(data);     // 将本节数据加入保存结果的序列 tmp 中
     // 然后用递归方式依次对本结点的子结点调用本函数；
    for (int i = 0; i < nodes.Size(); i++) {
          List3<T>*  subl = nodes[i]->DepthFirst();
          //  将子结点返回的顶点序列 subl 接续到保存结果的序列 tmp 中
        for (int j = 0; j < subl->Size(); j++)
              tmp->Append((*subl)[j]->data);
         delete subl;   // 释放子结点返回序列所占内存空间，以便访问下一子结点
     }
     return  tmp;
}
```

10.1.4 求根树中所有路径

在应用根树解决问题时，经常需要求根树中从根结点至各叶结点的路径（见 10.5 节图的算法），下面为根树模板中求从根结点至各叶结点路径函数的程序：

函数 AllPaths 返回从根结点至各个叶结点的所有路径，因为每条路径用一个结点的序列 List3<T>表示，所以用一个二重指针表示返回的结果中包括多个序列，同时再用一个引用类型的参数 no 返回该数组的长度（关于多重指针的概念可参考 6.4.7 节，关于引用类型参数的概念可参看 7.1.5 节）：

```
template <class T>
List3<T>** GTree<T>:: AllPaths(int& no)
{
    no = Leaves()->Size(); // 求出根树的所有叶结点，有多少叶结点就有多少路径
    List3<T>** paths = new List3 <T>*; // 根据结点数量生成一个二重指针的数组
    for (int i = 0; i < no; i++) // 为每个叶结点生成一个动态序列，保存从根结点到它的路径
        paths[i] = new List3<T>;
    int index = 0;
    Path(paths, index); // 注 1
    return paths;
}
```

程序注释：

注 1 函数 Path()采用递归形式，以深度优先方式从当前结点的左边第 i 个子结点开始发现各条路径，下面为其程序：

```
template <class T>
void GTree<T>:: Path(List3<T>** paths, int& i)
{
    int leaves = Leaves()->Size(); // 找出当前结点所有的叶结点
    if (nodes.Size() == 0) {    // 注 1
        paths[i]->Append(data);
        i++;
    }
    else {
        for (int j = i; j < i + leaves; j++) // 注 2
            paths[j]->Append(data);
        // 对各子结点递归调用本程序。
        for (j = 0; j < nodes.Size(); j++)
            nodes[j]->Path(paths, i);
    }
};
```

程序注释：

注 1 若本结点的子结点数为 0，则结束本序列，将该结点的数据加入到存储路径的第 i 个序列中，并将路径序列指数 i 移至下一个；

注 2 若本结点有子结点，则将本结点加入到从 i 开始至 i+leaves 的路径序列中，其中 leaves 是本分支下面所有的叶结点数；

下面是计算所有叶结点数量的函数 Leaves()的程序，其算法类似深度优先遍历：

```
template <class T>
List3<T>* GTree<T>:: Leaves()
{
    List3<T>* tmp = new List3<T>;   // 序列用于存储查找到的叶结点，并作为函数的返回值。
   if (nodes.Size() == 0) // 若当前结点无子结点，则为叶结点，将其加入到返回序列中
        tmp->Append(data);
    else
   // 若当前结点有子结点，则对每个子结点递归调用 Leaves 函数
      for (int i = 0; i < nodes.Size(); i++) {
          List3<T>* l = nodes[i]->Leaves();
   // 将从子结点返回的叶结点序列接续到本结点的叶结点序列后面
          for (int j = 0; j < l->Size(); j++)
              tmp->Append((*l)[j]);
          delete l;
        }
    return tmp;
};
```

10.2 二叉树

10.2.1 二叉树的数据模型和 C++模板

5.1.8 **介绍了二叉树的数学概念，**9.3.3 **中给出了二叉树的数据模型，并由该数据模型生成了** C++**类定义。在前述内容的基础上，本节给出一个与结点数据类型无关的二叉树模板：**

二叉树模板：

```
// BiTree.h （对应文件可从光盘上下载）
#include "list_3.h"   // 注 1
template <class T> class BiTree {
public:
    BiTree();
    BiTree(T*);
    ~BiTree();
```

```
    BiTree*     LeftNode() { return leftNode; };
    BiTree*     RightNode() { return rightNode; };
    T*          Data() { return data; };
    List3<T>*   PreOrder();      // 前序法遍历，注 2
    List3<T>*   InOrder();       // 中序法遍历，注 2
protected:
    T*          data;   // 注 1
    BiTree      *leftNode, *rightNode;
    int  Height(); // 求树的高度
};
```

程序注释：

注 1　**与根树一样，在二叉树的结点中也采用指针变量，从而使其可适用于任何数据类型。**

注 2　**结点遍历的结果生成的是一个关于结点数据的序列。**

10.2.2　二叉树的遍历算法

二叉树的递归数据结构是由二叉树的数学定义发展而来的，使得二叉树上的各种运算也可方便地采用递归形式实现。例如，5.2.8 **节用递归方法定义了树结点的前序遍历法，从根结点开始，先访问结点本身，然后前序遍历其左子树，最后再前序遍历其右子树。**

下面就是二叉树模板中成员函数 PreOrder()**的程序：**

```
// 前序法遍历
template <class T>
List3<T>* BiTree<T>:: PreOrder()
{
    List3<T> *tmp, *l = new List3<T>; // 用一个序列保持访问过结点的数据，作函数返回值
    if (data)
        l->Append(data); // 首先将本结点数据加入到返回序列中
    if (leftNode) {          // 如有左子结点，递归调用 PreOrder()函数
        tmp = leftNode->PreOrder();
        for (int i = 0; i < tmp->Size(); i++)      // 将左子结点返回序列接续到返回序列后
            l->Append((*tmp)[i]);
    }
    if (rightNode) {          // 如有右子结点，递归调用 PreOrder()函数
        tmp = rightNode->PreOrder();
        for (int i = 0; i < tmp->Size(); i++)      // 将右子结点返回序列接续到返回序列后
            l->Append((*tmp)[i]);
    }
    return l;
};
```

该程序表明，采用递归算法的函数程序的可读性相当好，可与算法描述很好地对应。除了其中那些判断是否有左、右子树、将访问结果记入已访问队列中等辅助性操作外，主要操作步骤与算法描述完全一致。

10.2.3 二叉检索树简介

导读：

传统的数据结构教材常用大量篇幅讨论二叉树的算法。二叉树的主要用途是进行检索，其原理相当于序列的对分查找。在计算机硬件性能已大大提高的今天，对于解决一般工程问题，二叉树用于检索的优势与序列相比已不明显，而其程序编制又十分繁琐，所以这里将这部分内容列为选读，以便学员对数据结构领域知识有较全面的了解。

二叉树的主要用途是进行检索，在树的结点中可存放实际的记录，也可存放索引记录。检索用的二叉树也称二叉检索树，它必须满足两个条件：第一，它必须是有序的，结点值的顺序必须与中序遍历的顺序保持一致；第二，它必须是高度平衡二叉树，或简称为平衡二叉树，又名 AVL-Tree（其名得自于它的发明者姓氏的首字母），其树中任意两个子树的高度差不大于 1。这样，左右子树中所包含结点数相近，对检索来说其效率最高。

维护二叉树为有序的平衡二叉树的关键是其插入和删除结点的运算：在平衡二叉树中插入结点时，应根据其结点数据值确定其插入位置；因为应始终保持两个子树的高度差不大于 1，因此插入后还应动态重组树的结构。同样，删除结点后，也应动态重组树的结构，以始终保持两个子树的高度差不大于 1。由于这两个操作均十分复杂，用途又十分有限，这里不再详述。有兴趣者可参看光盘中的有关章节。

10.3 递 归

10.3.1 递归的数学概念

知识点：

递归是数学中的一个重要方法，也是离散数学的研究领域之一。理解递归概念既是学习本章的关键，对提高程序设计能力也有重要意义。

迄今为止，教材已在多处涉及递归的概念，包括递归的数据成员定义、递归的函数程序等，本节将对递归概念做总结性的介绍。

递归是一种重要的数学方法，主要应用于以下 3 个方面：

（1）概念

数学上常用的阶乘函数、幂函数等，其定义都是递归的。例如，阶乘可定义为：

$$n! = 1 \text{ (当 } n = 0 \text{ 时)}，n! = n(n-1)! \quad \text{（当 } n>0 \text{ 时）}$$

（2）数据结构

如二叉树和根树的 C++类模板定义。

（3）算法或解题步骤

上面关于二叉树和根树的函数大多采用了递归形式。另外还有不少需要用递归方式解决的问题，其中一个最著名例子就是汉诺塔（Tower of Hanoi）的故事。

例 10 -1：汉诺塔的故事。

如图 10–2 所示，有 3 个圆柱 A、B、C。A 柱上有几个圆环（图示为 4 个），要求将它们全部移到 C 柱上。每次只能移动一个，可用 B 柱作为过渡，但无论在哪个柱子上，小环都不能放在大环下面。

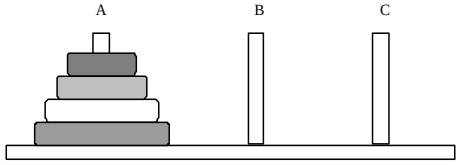


图 10–2　汉诺塔

解：解题算法如下：

① 若 A 柱上的圆环只有一个，即 n=1 时，可将它直接移动到 C 柱上。否则，执行后续步骤。

② 用 C 柱作为过渡，将 A 柱上 n–1 个圆环移动到 B 柱上。

③ 将 A 柱上最下面的大环移动到 C 柱上。

④ 用 A 柱作过渡，将 B 柱上的 n–1 个圆环移动到 C 柱上。

上述算法描述可直接转换为程序语句如下：

```
// move n disks from A to C
// using B as intermedia position
void Hanoi(int n, char A, char B, char C)
{
    if (n == 1)
       // if n ==1, move the disk from A to C
         printf("move %c to %c\n", A, C);
    else {
        // move n-1 disks to B
       // using C as intermedia position
         Hanoi(n - 1, A, C, B);
      // move the disk from A to C
          printf("move %c to %c\n", A, C);
      // move the disk from B to C
        // using A as intermedia position
        Hanoi(n -1, B, A, C);
```

```
    }
}
```

当 n=4 时，用语句 Hanoi(4, 'A', ' B', 'C')调用该函数，打印出以下关于操作步骤的说明：

```
move A to B
move A to C
move B to C
move A to B
move C to A
move C to B
move A to B
move A to C
move B to C
move B to A
move C to A
move B to C
move A to B
move A to C
move B to C
```

导读：

① 学员可尝试按上述步骤玩一下该游戏。

② 光盘中对应章节中有上述过程的动画演示和程序下载。

从上述介绍可归纳出递归的一些特点：

- 一个复杂问题，若能够分解为几个解法相同或类似、但复杂程度降低的子问题，而且如果能解决这些子问题，原来的问题也就解决了。这种问题可用递归方法解决。
- 递归算法必须有结束条件，否则便会陷入无穷循环。当分解后的子问题可直接解决时，就停止问题分解。这些可直接求解的问题称为递归结束条件，如阶乘函数中的递归结束条件是 0！=1。汉诺塔问题中，步骤①，即当 A 柱上只有一个圆环时，可将它直接移动到 C 柱上，也是递归结束条件。
- 递归定义的函数可简单地用递归过程求解。

*10.3.2 递归算法的化解

导读：

过去由于计算机资源的限制，许多关于数据结构和算法的教材中常常主张将递归算法非递归化，但现在一般已不再强调非要将递归算法化解。

虽然用递归方法描述算法简明、扼要，可读性强，但递归本身并不是一种高效的算法，因

为多次调用函数和传递参数，所消耗的系统资源很多。因此，如果需处理的数据对象数量很大，如一个很大的树，就应考虑将递归算法化解为非递归的算法。

将递归程序非递归化的方法主要有 2 种：

（1）用循环语句代替递归

例 10-2：

递归形式的阶乘函数：

```
int   Factorial(int  n)
{
      if (n == 0)
            retrurn 1;
      else
            return n*(n – 1);
}
```

非递归形式的阶乘函数：

```
int Factorial（int n）
{
      int x = 1;
      for (int j = 2, ; j <= n; j++)
            x = x*j;
      return x;
}
```

（2）使用堆栈化解递归

下面是用堆栈方法实现的二叉树的前序遍历函数：

例 10-3：

```
#include "stack.h"   //  堆栈的模板
#include "list_3.h"
template <class T>
List3<T>* BiTree<T>:: PreOrder()
{
    List3<T> *l = new List3<T>; //  用 List 记录访问过的结点
      Stack<BiTree> stack;        //  用堆栈来记录分支结点
    BiTree<T>* t = this; //  从根结点开始
     while  (t)  {
           l->Append(t->data);
          //  如有左子树，压入堆栈后沿左子树向下
          if  (t->leftNode) {
                stack.Push(t);    //  注 1
```

```
            t = t->leftNode;
        }
        // 若无左子树，但有右子树，压入堆栈后沿右子树向下
        else if (t->rightNode) {
            stack.Push(t); // 注 1
            t = t->rightNode;
        }
        // 如堆栈空，遍历完成
        else if (!stack.Peek())
            t = NULL;
        // 如堆栈不空
        else if (stack.Peek()->rightNode == t) { // 在右子树中
            // 沿右子树返回至分支结点
            while (stack.Peek()->rightNode == t)
                t = stack.Pop();
            t = stack.Pop();
        }
        else {      // 在左子树中
            // 沿左子树返回至有右子树的分支结点
            do {
                t = stack.Pop();
            } while (t && !t->rightNode);
            if (t)
                t = t->rightNode;
        }
    }
    return l;
};
```

程序注释：

注 1　这里用堆栈保存分支结点，从左子树返回时，可转向右子树；而从右子树返回时，则一直返回到最上层的分支结点。堆栈的作用是先进后出，所以首先进入的是较上层结点，而首先返回的则是较下层的结点。

两种化解递归方法中，前者只适用于所谓的尾递归（tail recrusive）类型的问题，后者则适用于所有的递归问题。但是，如何用堆栈化解递归，却无一般规律可循。不过一般可通过用堆栈保存以下信息使递归化解：

- **程序调用参数的当前值。**
- **程序中本地变量的值。**
- **返回地址，也就是当前程序调用完以后应返回的控制点等。**

10.4 图的 C++模板和程序

导读：

多数关于数据结构的教材均用矩阵表示图，一些采用面向对象方法的教材也是如此，但矩阵表示并不真正符合面向对象概念，因为矩阵表示不能反映图的本质概念。本节采用面向对象分析方法，从图的数学概念出发所定义的图的模板，不但与图的数学概念完全对应，而且有关函数也与算法描述完全对应，真正体现了面向对象方法的特点和优点。

第 5 章介绍了关于图的基本数学概念，在图上可进行的数学运算包括发现通路、环路、计算最短路径等。这些运算一般均比较复杂，必须借助于计算机进行。在数据结构中，图是应用最广的，但其算法也最复杂。寻求一些问题的最佳算法至今仍是研究的内容。

10.4.1 图的数据模型和 C++模板

9.4.1 **节里讨论了图的两种数据模型，并得到了两种完全相同的图的类定义。在前述内容的基础上，本节给出图的模板定义：**

图的模板：

// graph.h **（对应文件可从光盘上下载）**

```
#include "list_3.h"
template <class T> class Vertex {
public:
      Vertex(T*    d):data(d)  {};
       T*          Data() { return data; };
protected:
       T*         data;
};
template <class T, class V> class Arc {
public:
      Arc(T*    d):data(d)  {} ;
      T*         Data()  {  return  data;  };
      V*         Head() { return head; };
      V*       Tail()  {  return  tail;  };
protected:
       T*         data;
      V        *head,  *tail;  //
};

template <class V, class A> class graph {
```

```
public:
    graph() {};
    int         AddVertex(V* v) { return vertices.Append(v); };
    int         AddArc(A* a) { return arcs.Append(a); };
    List3<V>& Vertices() { return vertices; };
    List3<A>& Arcs() { return arcs; }
protected:
    List3<V>    vertices;
    List3<A>    arcs;
};
```

通过定义不同的顶点和边类，由该模板可方便地生成各种图，包括有向图、无向图、AOV 图或者 AOE 图等。

10.4.2 无向图的最小费用生成树和克鲁斯克尔函数

构筑最小费用生成树的克鲁斯克尔算法已在 5.2.4 节中予以详细解释，因为求解最小费用生成树时要根据边上的权重计算路径长度，所以本节定义了一个实际的边和顶点，用继承模板的方法生成一个实际的图类，并以其为例介绍克鲁斯克尔函数的程序。该例也是对图模板的使用方法的一个说明：

例 10-4：克鲁斯克尔函数。

```
// 用顶点模板生成一个用整数编号的顶点类
typedef   Vertex<int>    vertex;
// 定义一个用整数编号、带权重的边的数据类
class ArcData {
public:
    ArcData(int n, float x): no(n), w(x) {};
    int         No() { return no; };
    float       W() { return w; };
private:
    int         no;
    float       w;
};
// 用 ArcData 类对边的模板进行实例化
typedef Arc<ArcData, vertex> arc;
// 定义边和顶点的序列
typedef List3<vertex>          vList;
typedef  List3<arc>            aList;
// 定义生成树
typedef GTree<vertex>  spanTree;
```

```
// 用继承模板的方法定义一个实际的图
// 因为所有数据成员均在 graph 的模板中定义过了，这里只需定义有关函数即可
class Graph: public graph<vertex, arc> {
public:
    Graph() {};
// 加入一个编号为 n 的顶点
    int         addVertex(int n);
// 加入一个顶点编号为 v1、v2，自己编号为 aNo，权重为 w 的边
    int         addArc(int v1, int v2, int aNo, float w);
// 求最小费用生成树
    Graph*    Kruskal();
private:
// 对边按权重进行排序的函数
    void          SortArcs();
// 检测图中是否有环路的函数
    int         HasLoop();
    int         CheckLoop(int, int*, spanTree*);
// 根据边的号码查找边元素，避免重复加入
    arc*        SearchArc(int);
// 根据顶点编号查找顶点元素，避免重复记入
    vertex*    SearchVertex(int);
// 根据顶点实例 pointer 查找其在序列中序号
    int         SearchVertex(vertex*);
};
// Kruskal 函数程序
Graph*    Graph::Kruskal()
{
// 虽然 Kruskal 算法的结果是一个生成树，但由于无法预先知道其根结点是哪一个
// 所以只能用一个图来保存结果
    Graph* g = new Graph; // 保存结果的图
     SortArcs(); // 注 1
    int   index = 0;
     while ( g->Arcs().Size() < vertices.Size() - 1 ) { // 注 2
          g->AddArcs(arcs[index]); // 依次将权值最小的边加入图中
          g->AddVertices(arcs[index]->Head()); // 将该边的顶点也加入图中
          g->AddVertices(arcs[index]->Tail());
        if (g->HasLoop()) // 检查图中是否有环路，如果有环，则将新加入的边删除，注 3
              g->Arcs().Delete(arcs[index]);
```

```
            index++; // 考虑下一条边
        }
        return g;
};
```

程序注释：

注 1 **因为每次都要寻找权重最小的边，所以需先对** arc **数据进行排序，可仿照** BubbleSort **函数编写此函数。**

注 2 **有** n **个顶点的最小生成树只有** n–1 **条边，所以可通过计算图** g **中的边数来控制循环的结束。**

注 3 **因为最小费用树中不能有环路，所以用辅助成员函数** HasLoop()**用于检测是否有环。若有环，则将新加入的边删除，继续考虑下一条边。**

函数 HasLoop()**与生成深度优先遍历树的函数** SpanTree **非常相似（参见** 10.1.4 **节），在深度优先遍历中，若遇到已访问过的顶点，则说明有环路。具体程序如下：**

```
int Graph:: HasLoop()
{
    int nodes = vertices.Size();
// 用于记录访问过的顶点号的数组
    int* visited = new int [nodes];
    for (int i = 0; i < nodes; i++)
        visited[i] = 0;  // 初始化该数组
    int hasLoop, index = 0;
    do {
        //
        spanTree* tree = new spanTree(vertices[index]);
        hasLoop = CheckLoop(index, visited, tree);
        if (!hasLoop)
            for (i = 0; i < nodes; i++)
                if (!visited[i])
                    index = i;
        delete tree;
    } while (!hasLoop && i < nodes);
    return hasLoop;
}

int Graph::SearchVertex(vertex* v)
{
    for (int i = 0; i < vertices.Size(); i++)
        if (vertices[i] == v)
```

```
                return i;
        return -1;
};

int Graph::CheckLoop(int index, int* visited, spanTree* tree)
{
    visited[index] = 1;
    int hasLoop = 0;
    for (int j = 0; j < arcs.Size() && !hasLoop; j++) {
        if (arcs[j]->Head() == vertices[index]) {
            int t = SearchVertex(arcs[j]->Tail());
            if (!visited[t]) {
                tree->Insert(vertices[index], vertices[t]);
                hasLoop = CheckLoop(t, visited, tree);
            }
            else
                hasLoop = 1;
        }
    }
    return hasLoop;
};
```

导读：

关于有向图的狄克斯特算法，由于程序太长，这里不再介绍。有兴趣者可从光盘上下载程序文件运行。

习　题

知识点：

根树的定义和程序将用在图的有关函数中计算生成树，但有关根树的函数比较复杂，所以通过实际运行有关程序来理解其工作原理是个好方法。

1. **任何算法的设计都是以人工操作过程为基础的，计算机程序无非是人工计算过程的再现。因此，建议用教材中关于根树的模板编写一个可运行的程序，在程序中适当位置加入一些打印中间数据的语句，通过观察程序运行过程来帮助对有关程序的理解。**

具体作业步骤是：

① 定义一个结点数据类型为字符或整数的根树，例如可用以下定义的树表示图 10–1**。**

```
typedef GTree<char>  charTree;
```

② 定义从文件读取结点数据的函数。建议数据文件采用以下格式：首行为根结点值，其余各行成对列出“父结点值 子结点值”。如图 10–1 所示根树的数据文件为：

```
A
A B
A C
A D
B E
…
```

③ 读入数据后，用函数 Insert（T, T*）将各子结点插入指定位置。因为树中存储的是数据的指针，如果用字符型变量作为结点数据时，必须另外生成一个字符型动态变量作为函数输入参数，如下面例句所示：

```
charTree  t(new char('A'));  // 生成一个值等于 A 的动态字符型变量，将其指针作为根结点
t.Insert('A', new char('B')); // 生成一个值等于 B 的动态字符型变量，将其指针作为子结点
t.Insert('A', new char('C')); // 生成一个值等于 C 的动态字符型变量，将其指针作为子结点
```

④ 在打印结点数据时，应将取值运算符加在函数 Data（）前面，才能得到结点数据值，如以下例句所示：

```
fprintf(" The tree root data=%c \n", *t.Data());
```

⑤ 用函数 AllPaths()列出树中所有路径，检查树的结构是否与图 10–1 一致。

2.（选做）用教材提供的克鲁斯克尔函数编写一个程序，用 5.2.4 节例 5–13 的数据进行计算，下面是打印图中数据的函数语句，供编程时参考：

```
void PrintGraph(Digraph& d)
{
    char fileName[256];
    FILE *fp;

    printf("Output file name: ");
    scanf("%s", fileName);
    if ((fp = fopen(fileName, "w")) = = NULL)
        printf("File %s open error\n", fileName);
    else {
        fprintf(fp, "%d\n", d.Vertices().Size());
        for (int i = 0; i < d.Vertices().Size(); i++)
            fprintf(fp, "%d\n", d.Vertices()[i]->No());
        for (i = 0; i < d.Arcs().Size(); i++)
            fprintf(fp, "arc No=%d  head No=%d  tail No=%d  Weight=%f\n",
                d.Arcs()[i]->No(),
                d.Arcs()[i]->Head()->No(),
                d.Arcs()[i]->Tail()->No(),
                d.Arcs()[i]->W());
        fclose(fp);
    }
}
```

3.（选做）用图的模板定义一个 PERT 图，编写一个计算 PERT 图关键路径的程序，用 5.3.1 节例 5–19 的数据进行计算。

第 11 章　课 程 作 业

本教材的书名中含“C++编程实训”，就是强调应尽可能按实际工程作业的要求练习编程作业的全过程。本章安排两个课程作业，对任务进行详细讲解，并提供主要函数的程序，要求学员在此基础上编写出完整的程序。

两个课程作业各有特色，代表两类比较典型的工程实际问题。作业一涉及数据类较多，用户界面设计较复杂，但数据运算较简单，基本上只是定义、修改和存取数据。作业二数据类不多，但数据运算较复杂，对函数程序设计要求较高。

11.1　课程作业一——学籍管理系统

本节是 Case-3 系列讲解的第 3 部分，9.2.10 节的 Case-3-1 给出了该学籍管理系统的数据模型，9.5.2 节的 Case-3-2 给出了该学籍管理系统部分的 UCD，本章将要求学员根据下面的步骤完成该学籍管理系统编写工作的全过程。

11.1.1　根据系统数据模型设计 C++类定义

学籍管理系统的完整数据模型见图 9–13，其中 Student 等类的定义已在 7.2.3 节中给出过，现要求学员完整地写出以下 5 个 C++类的定义：

- Person
- Student
- Teacher
- Course
- Score

11.1.2　定义管理实例的序列

这步工作非常重要，可有效管理类的实例。而且，若无序列类，有些类的功能性函数就无法实现。序列用继承 List 模板三的方法产生，并在各序列子类中分别定义以下查询函数：

- StudentList：根据班级名称查找学生实例的函数 StudentList* SearchByCls (char* clsName)，因为一个班级有多个学生，所以函数返回值为一个序列。
- TeacherList：根据系名查找教师实例的函数 TeacherList* SearchByDept (char* deptName)，因为一个系有多名教师，所以函数返回值为一个序列。
- CourseList：根据课程编号和名字查找一个课程实例的函数，Course* Search (int cNo, char* cName)。
- ScoreList：根据课程编号和学生姓名查找一个成绩实例的函数，Score* Search (int cNo, char* sID)。

11.1.3 完成 UCD

根据用户使用情况作出的系统的部分 UCD 如图 9–17、图 9–18、图 9–19 所示，要求学员继续完成以下服务的分解子图，并最终完成系统全部的 UCD 图。

- 管理学生。
- 管理课程。

11.1.4 设计菜单函数

UCD 完成以后，即可根据它设计系统的菜单树，图 11–1 提供了一种建议的菜单树设计，学员也可自行设计菜单树。

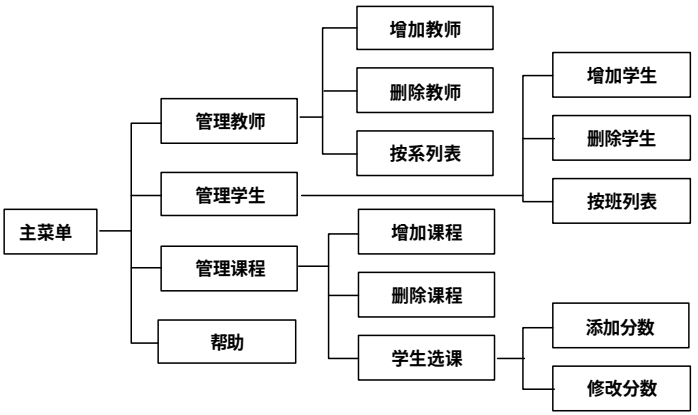


图 11–1 学籍管理系统的菜单树

下面给出程序主函数、ManageTeacher 函数等两个菜单函数的程序示例，学员可仿照范例完成其他菜单函数的程序。

```
// 程序的主函数
void Main ()
{
    StudentList    sList;    // 管理学生实例的序列
    TeacherList    tList;    // 管理教师实例的序列
    CourseList     cList;    // 管理课程实例的序列
    ScoreList      scList;   // 管理成绩实例的序列
    // 用于存储数据文件的名称，注1
     char*    fileNames[4];
    for (int i = 0; i <4; i++)
        fileNames[i] = new char [256];
```

```
    Initial(fileNames, sList, tList, cList, scList); // 注2
    char    choice;
    do {
        MainMenu(); // 这是一个打印菜单指令的函数，参见9.3.3节
        scanf("%c", &choice);
        switch (choice) {
        case '1': ManageTeacher(tList, cList);   // 注3
                  break;
        case '2': ManageStudent(sList, scList); // 注3
                  break;
        case '3': ManageCourse(cList, scList, tList, sList); // 注3
                  break;
        case '4': Help(); // 打印帮助文件
        default: printf（"the input %c is not legal choice\n", choice）;
        }；
    } while （choice != ' '）;
    // 以下函数将程序处理过的数据保存到数据文件中，注4
    SaveWorkSet(fileNames, sList, tList, cList, scList);
};
```

程序注释：

注 1 **程序开始时，应将数据从数据文件中读入程序，而在系统退出时一定要将处理过的程序保持在原来的数据文件中，所以应记录各个文件的名称。因为是整进整出，所以统一用一个数组记录。**

注 2 **每个程序开始运行均需进行许多准备工作，函数** Initial()**的作用是对这些工作进行整合，以避免列举一大堆零碎的语句。**

注 3 **根据维护数据一致性完整性的原则，在增加（或删除）一方类的实例时，应同时考虑随之增加（或删除）另一方类的实例，所以这里将有关类的序列均作为参数输入子函数。**

注 4 **每个程序退出时均需进行许多整理工作，函数** SaveWorkSet()**的作用是对这些工作进行整合，以避免列举一大堆零碎的语句。**

函数 Initial()**在本程序中的任务主要是读取数据文件，下面是其程序示例：**

```
void Initial(char** fileName,
             StudentList&  tList, TeacherList& sList, CourseList& cList, ScoreList& scList)
{
    ReadStudentList(fileNames[0], sList); // 注 1
    ReadTeacherList(fileNames[1], tList);
    ReadCourseList(fileNames[2], cList);
    ReadScoreList(fileNames[3], scList);
};
```

程序注释：

注 1 函数 ReadStudentList()**的作用是将学生数据读到序列中，同时记录数据文件的名字。具体程序见第** 8 **章习题** 3**。学员很容易仿照它自行编写出其他几个读取数据的函数。**

下面是子菜单函数 ManageTeacher()**程序示例，其设计方法与设计主菜单的方法完全相同。**

```
void ManageTeacher(TeacherList& tList, CourseList& cList)
{
    char choice;
    do {
        ManageTeacherMenu();
        scanf("%c", &choice);
        switch (choice) {
        case '1': AddTeacher(tList, cList);
                  break;
        case '2': DeleteTeacher(tList, cList);
                  break;
        case '3': ListTeacher(tList);
                  break;
        default: ;
        };
    } while (choice != '0');
}
```

11.1.5 设计交互式数据输入函数

下面为函数 AddTeacher()**的程序示例，注意它在用户界面严密性方面采取的各种措施。要求学员仿照它完成其他交互式数据输入函数的设计。**

```
void AddTeacher(TeacherList& tList, CourseList& cList)
{
    char    id[20];
    char    name[8];
    char    gender;
    char    deptName[256];
    int     cNo;
    char    cName[256];
    int     valid = 0, interrupt = 0;
// 该循环控制条件可用语言描述如下：
// 当输入数据不是有效数据和用户没有输入中断指令时，保持循环
    while (!valid && !interrupt) {
        printf("Input new teacher's ID[18 digits], hit [Enter] key to exit : ");
```

```
        if (scanf("%s", id) == EOF)   // 注 1
            interrupt = 1;
        else if ((valid = IsValidID(id)) == 1) // 注 2
            if (tList.SearchByID(id) != NULL) {   // 检查该教师实例是否已经存在,注 3
                printf("The teacher with ID =%s is already in system\n", id);
                valid = 0;
            }
    }
    valid = 0;
// 该循环控制条件可用语言描述如下：
// 当输入数据不是有效数据和用户没有输入中断指令时，继续循环
    while (!valid && !interrupt) {
        printf("Input new teacher's name[< 6 letters] , , hit [Enter] key to exit : ");
        if (scanf("%s", name) == 0)
            interrupt = 1;
        else
            valid = IsValidName(name); // 注 2
    }
    valid = 0;
    while (!valid && !interrupt) {
        printf("Input new teacher's gender['M' or 'F'] , , hit [Enter] key to exit : ");
        if (scanf("%c", &gender) == EOF)
            interrupt = 1;
        else
            valid = IsValidGender(gender);
    }
    valid = 0;
    while (!valid && !interrupt) {
        printf("Input new teacher's department name[< 256 letters] , hit [Enter] key to exit : ");
        if (scanf("%s", deptName) == EOF)
            interrupt = 1;
        else
            valid = IsValidDeptName(deptName); // 注 2
    }
    while (!interrupt) {  // 注 3
        printf("Input the course number taught by the teacher, hit [Enter] key to exit : ");
        if (scanf("%d", &cNo) == EOF)
            interrupt = 1;
```

```
            else if (IsValidCourseNo(cNo)) {
                printf("Input the course name [< 256 letters] , hit [Enter] key to exit : ");
                if (scanf("%s", cName) == 0)
                    interrupt = 1;
                else if (IsValidCourseName(cName)) {
                    Course* c = cList.Search(cNo, cName); // 检查该课程是否已经分配他人
                    if ( c==NULL) { // 返回空指针，说明此课程为新课程
                        tList.Append(new Teacher(id, name, gender, deptName));
                        cList.Append(new Course(cNo，cName, id));
                    }
                    else
                        printf("The course %d %s is already assigned to teacher %d\n",
                            c->No(), c->Name(), c->TeacherID());
                }
            }
        }
};
```

程序注释：

注 1 **对于交互方式输入数据的操作，应有能随时中断操作的出口。如果不用变量** interrupt **控制循环的执行，则用户必须输入所有有效数据后才能退出操作。**

注 2 **用户界面一定要检查输入数据的有效性，特别是对以交互方式输入的数据。函数** IsValidID()、IsValidName()**等就是这个作用。关于如何检查输入数据有效性的讨论，参见** 6.4.2 **例** 6-3 **和第** 6 **章习题** 6。

注 3 **除了数据本身必须有效以外，还必须保证增加的新实例以前没有定义过，所以需在管理** teacher **实例的序列中进行检索，当返回一个空指针时，表示序列中没有它，才是有效数据，这也是数据有效性检查的另一方面。序列** TeacherList **的查询函数** SearchByID()**程序的编写可参见** 8.3.2 **关于** StudentList **的查询函数完成。**

注 4 **因为一个教师可教多门课，所以采用一个循环使用户能输入多个课程名称。**

11.1.6 划分程序模块

程序应合理模块化，建议程序由以下模块文件组成：

- list_3.h **模板文件。**
- **对每个类各定义一个头文件和源代码文件。**
- **对各类的序列类各定义一个头文件和源代码文件。**
- **将主菜单与下级子菜单放在一个文件里。**
- **将管理学生、管理教师、管理课程、管理成绩的有关文件分别放在** 4 **个模块里。**
- **其他辅助函数为一个模块。**
- **主函数模块。**

11.1.7 测试程序

9.6.2 节中指出，若程序中有若干类似的类，可先编写一个类进行彻底的测试，确保算法以及输出输入功能正确无误后，再编写其他类似的类，会比一次性编写完所有的类再逐个进行测试效率高。

因为学籍管理系统中各个类的定义均很相似，管理它们的函数功能也很相似，因此建议学员先实现一个类，如 Student 类的各种功能，取得经验后再编写其余各类的程序（在书后配盘中提供了可实现管理 Student 类功能的完整程序，学员可以其为基础完成整个作业）。

利用所在班级有关数据准备 4 个数据文件，要包括足够数量的数据，以确保反复运行后数据仍然能保证完整性和一致性：

- student.txt
- teacher.txt
- course.txt
- score.txt

11.1.8 编写完整的文档

文档是规范化编程作业的重要组成部分，本课程作业要求：

- 写出完整的开发文档，包括系统分析、数据模型、程序使用手册等。
- 在程序中加上完整的注释语句。

11.1.9 其他要求

① 按 9.3.5 介绍的方法增加一个查询教师所教学生的服务。

② 在上述系统中为每个班级指定一个班主任，其关系是一个班级只有一个班主任，但不是每个教师都是班主任。然后，进行以下工作：

- 画出类图。
- 定义对应的类和函数。
- 在 UCD 里和菜单里添加对班主任进行管理的服务。

11.2 课程作业二——五子棋游戏

本作业要求编写一个 DOS 界面下运行的五子棋游戏，可两人对弈，也可以是人与机器对弈。

11.2.1 程序工作原理分析

图 11–2 是下棋程序工作原理示意。用户界面将棋盘和棋子的内部数据转换为外部形式——棋盘和棋子的图形，供用户观看。用户用不同方式将移动棋子的数据输入程序，对于 DOS 界面，可用键盘直接输入棋子在棋盘中的 x、y 坐标；若为 Windows 界面，则可用鼠标点击，然后由程序将鼠标点击屏幕的位置换算为棋盘中的 x、y 坐标。程序将用户移动棋子的数据转换为其内部数据表示，对棋局输赢作出判断，当一方获胜时停止运行。

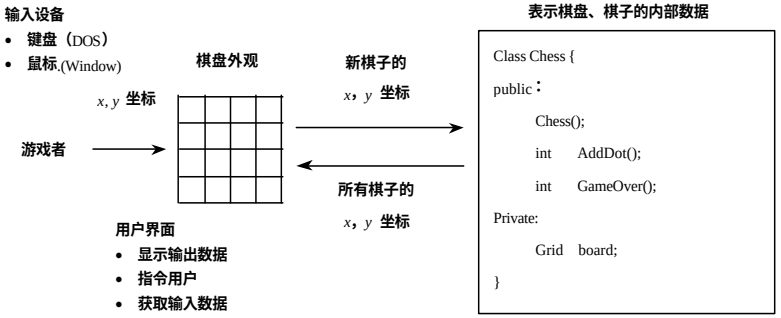


图 11–2 下棋程序的工作原理

如果是人机对弈，则由一个能生成棋子布局的程序或系统代替游戏一方产生移动棋子的数据。对于下棋程序，其工作过程仍然相同。因此，下棋程序的基本功能就是实现两人对弈的程序。

本作业要求学员根据上述分析画出系统的 UCD 图。

下面首先实现两人对弈的程序，然后再介绍由计算机产生棋子移动数据的算法。

11.2.2 数据建模

根据以上分析，程序中最重要的对象为：棋盘（Grid）、棋子（Dot）、和整个游戏（Chess）。下面是它们的定义：

（1）棋盘与棋子的定义

因为棋盘是二维的，所以最直接的一种表示方式就是用矩阵表示棋盘，用矩阵中的元素表示棋子。

可用整数表示棋子种类，但为了增加程序可读性，可定义一个枚举类型（参见 6.4.2 语法）：

```
enum Dot { Blank, Black, White };
```

棋盘可用矩阵模板直接产生：

```
typedef Matrix<Dot, 10，10> Grid;
```

（2）游戏的定义

```
class Chess {
public:
    Chess();
    int       LegalMove(int, int); //  检验棋子移动的合法性的规则
    void      AddDot(int, int, Dot); //  向棋盘中添加棋子
    Grid&     Board() { return board;};
    int       Win(); //  判断是否有赢局
    int       Full();   //  判读棋盘是否已满
private:
    Grid      board;
};
```

上述类定义与游戏种类无关，甚至可将其定义为一个模板。函数的具体程序决定其究竟为何种游戏。例如，如果使函数 LegalMove()执行象棋规则，就是象棋游戏；若执行五子棋规则，就是五子棋游戏。这体现了设计数据模型最重要的原则，即尽可能从高层次的一般概念出发，使定义的数据有普遍性。

下面为有关成员函数的程序：

```
Chess :: Chess()
{     for(int i =0; i < board.Size() ; i++ )
            for(int j = 0; j < board[i].Size() ; j++ )
                  board[i][j] = Blank;    // 将棋盘初始化为全部空格
}
// 检查棋子位置是否合法
// 合法则返回 1，否则返回 0
int Chess::LegalMove(int x, int y)
{
     return (board[y][x] == Blank); // 若不等于空，说明已有棋子
}
// 向棋盘中添加一个棋子
void Chess::AddDot(int x, int y, Dot d)
{
     board[y][x] = d;
}
```

要求学员自行完成函数 Full()的程序(提示：有若干方法可判断棋盘是否满。如，可设一个计数器，记下棋子的总数，当棋子总数等于棋盘格数时，棋盘就满了。另一种方法是对棋盘进行扫描，如所有棋盘格均有子时，棋盘就满了)。

11.2.3 函数 Win()的实现

成员函数 Win()的任务是判断是否一方赢棋。对五子棋来说，五子连成一线为取胜条件。判断五子一线的基本算法如下：从棋盘左上角开始先按列，后按行向下扫描，跳过空格，当遇到非空格时，则检查其东、东南、南、西南 4 个方向上各有多少与它相同的棋子，当累积到 5 个时，则有一方获胜。根据上述分析，可以设计出函数 Win()的程序代码如下所示：

```
int Chess:: Win()
{
     int size=board.Size()-1;
     int k = 0;
     for(int i = 0; i < size && k!=5; i++ )
          for(int j = 0; j < size && k!=5; j++ )
               if (board[i][j] != Blank) {
                    int jj = j, ii = i;
```

```
                k = 1;
                while (board[i][jj] == board[i][jj+1] && jj < size )
                    { k+=1; jj++; } // 东向扫描
                if (k!=5) {
                    jj = j, ii = i;   k = 1;
                    while (board[ii][jj] == board[ii+1][jj+1] && ii <size && jj<size)
                        { k+=1; ii++; jj++; } //东南扫描
                }
                if (k!=5) {
                    jj = j, ii = i;   k = 1;
                    while (board[ii][j] == board[ii+1][j] && ii < size )
                        { k+=1; ii++; } // 南向扫描
                }
                if (k!=5) {
                    jj = j, ii = i;   k = 1;
                    while (board[ii][jj] == board[ii+1][jj-1] && ii < size && jj >0)
                        { k+=1; ii++; jj--; } // 西南扫描
                }
            }
        if (k == 5) // 如果一方有五子连线，则获胜，游戏结束
            return 1;
        return 0;
    };
```

导读：

为了便于理解，这里只应用了最明显的赢棋判断条件。实际上，还有两种判断赢棋的条件：一方形成“活四（四子连线且两头皆为空格）、或者一方形成“双活三（三子连线，且两头皆为空格）”。可定义两个判断上述条件的函数，然后用逻辑“或”运算把三个判定条件组合起来作为最终判定赢棋的条件。

11.2.4 显示棋盘和棋子的函数

虽然是 DOS **命令行界面，但是利用一些特殊字符，也能得到比较美观的棋盘和棋子图形，下面就是在屏幕上画棋盘格与棋子的程序：**

```
// 首先定义棋子的图符
char*      DotIcon[3]={"●", "○", "+" };
// 画棋盘与棋子的函数
void DrawBoard(Grid& g)
{
```

```
    system("cls");   // 清屏
    printf("Welcome our chees word!\n");
    int size=g.Size();
    printf("    0 1 2 3 4 5 6 7 8 9 10\n");
    for(int i = 0; i < size; i++ ) {
        printf("%2d ",i);
        for(int j = 0; j < size; j++ ) {
            if (i == 0 && j == 0 && g[i][j]==Blank) printf("┌");   // 棋盘左上角
            else if (i == size-1 && j == 0 && g[i][j]==Blank) printf("└"); // 棋盘左下角
            else if (i == 0 && j == size-1 && g[i][j]==Blank) printf("┐"); // 棋盘右上角
            else if (i == size-1 && j == size-1 && g[i][j]==Blank) printf("┘"); // 棋盘右下角
            else if (i == 0 && g[i][j]==Blank)  printf("┬"); // 棋盘上边线
            else if (i==size-1 && g[i][j]==Blank)    printf("┴"); // 棋盘下边线
            else if ( j == 0 && g[i][j]==Blank) printf("├"); // 棋盘左边线
            else if (j == size-1 && g[i][j]==Blank)   printf("┤"); // 棋盘右边线
            else printf("%s", DotIcon[g[i][j]]); // 棋子或者空格
        }
        printf("\n");
    }
}
```

11.2.5 主函数控制逻辑

下面是程序主函数示意，注意其中并没有验证输入有效性的措施，学员必须自行完成它们。

```
void main() {
    int         x, y, player;// 默认黑棋先手
    int         computerPlayer;
    int          win;
    Chess       game;
// 首先确定游戏玩法
    printf("How many players [1: humen to computer; 2: two people :?\n");
    scanf("%d", &computerPlayer);
    if (computerPlayer == 1) {
        game.AddDot(5, 5, Black);
        player = White;
    }
    else
        player = Black;
    do {
        DrawBoard(game.Board());
```

```
            if (player == Black)
                if (computerPlayer == 1) {
                    // 如是人－机对弈，由计算机产生棋着
                    GenerateMove(game.Board(), x, y);
                    game.AddDot(x, y, Black);
                }
                else { // 从键盘读取黑方输入
                    do {
                        printf("Black Input x, y: ");
                        scanf("%d %d", &x, &y);
                    } while (!game.LegalMove(y, x));
                    game.AddDot(y, x, Black);
                }
            else { // 白方下子
                do {
                    printf("White Input x, y: ");
                    scanf("%d %d", &x, &y);
                } while (!game.LegalMove(y, x));
                game.AddDot(y, x, White);
            }
            if ((win = game.Win()) == 0)
                player = !player;
        } while( !win && !game.Full()); // 若有人赢或者棋盘满，则游戏结束
        if (!win)
            printf("The game is even\n");
        else if (player == Black )
            printf("^-^ Black wins !^-^\n");
        else
            printf("^-^ White wins !^-^\n");
    }
```

导读：

一个设计的好的程序，用户界面和内部的运算应该彼此完全独立，从而可以很方便地从一个系统移植到另一个系统。在书后配盘里有本作业的 Windows **界面的版本，它与这里给出的程序基本上相同，不同的只是函数** DrawBoard **（）里面画图部分的内容以及读取用户输入数据部分的内容。**

11.2.6 产生棋着的算法

在 5.3.3 **节图论应用中介绍了搜索树在游戏程序中的应用。对于五子棋，搜索算法与判断**

输赢的算法基本相同。具体步骤为：

① 从棋盘左上角开始先按列、后按行向下扫描。

② 遇到己方的子，则依次在其东、东南、南、西南 4 个方向上检查连子个数，找出连子最多的方向，判断该方向前后是否有空格，如果有空，则本次扫描有效，记下该空格位置。

③ 若本次扫描有效，则与前次扫描结果进行比较，如果优于前次，则记下这次棋局。

④ 全部扫描结束，所记下的最佳棋局就是最后产生的棋着。

```
// 产生棋局的函数程序示例
void GenerateMove(Grid& g, int& x, int& y)
{
    int size= g.Size();
    int xx, yy, xxx, yyy;
    int k, merit, maxMerit = 0;
    int ii, jj;
  for(int i = 0; i < size; i++ )
      for(int j = 0; j < size; j++)
        if ( g[i][j] == BLACK) {
            // search east
            jj = j;
            while (g[i][jj] == BLACK && jj < size) {
                k++; jj++;
            };
            // front is blank
            if (jj < size && g[i][jj] == BLANK) {
                merit = k;
                xx = jj; yy = i;
            }
            // rear is blank
            else if (j > 0 && g[i][j-1] == BLANK) {
                merit = k;
                xx = j - 1; yy = i;
            }
            else
                merit = 0;
            // search southeast
            ii = i; jj = j;  k = 1;
            while (g[ii][jj] == BLACK && ii < size && jj < size) {
                k++;   ii++;   jj++;
            };
```

```
// front is blank
if (g[ii][jj] == BLANK) {
      xxx = jj;   yyy = ii;
}
// rear is blank
else if (i > 0  && j > 0 && g[i- 1][j - 1] == BLANK) {
      xxx = j - 1; yyy = i - 1;
}
else
      k = 0;
// better than east
if ( k > merit) {
      merit = k;
      xx = xxx; yy = yyy;
}
// search south
ii = i; k = 1;
while (g[ii][j] == BLACK && ii < size ) {
      k++; ii++;
};
// front is blank
if (g[ii][j] == BLANK) {
      xxx = j;    yyy = ii;
}
// rear is blank
else if (i > 0  && g[i- 1][j] == BLANK) {
      xxx = j;   yyy = i - 1;
}
else
      k = 0;
// better than southeast
if ( k > merit) {
      merit = k;
      xx = xxx; yy = yyy;
}
// search southwest
ii = i; jj = j; k = 1;
while (g[ii][jj] == BLACK && jj > 0 && ii < size ) {
```

```
                    k++; ii++; jj--;
                };
                // front is blank
                if ( g[ii][jj] == BLANK) {
                    xxx = jj; yyy = ii;
                }
                // rear is blank
                else if (j < size - 1 && i > 0 && g[i - 1][j + 1] == BLANK) {
                    xxx = j + 1; yyy = i - 1;
                }
                else
                    k = 0;
                // better then south
                if ( k > merit) {
                    merit = k;
                    xx = xxx; yy = yyy;
                }
                // if better than previous
                if (merit > maxMerit) {
                    maxMerit = merit;
                    x = xx;
                    y = yy;
                };
            }; // end if == black
    };
```

该范例程序表明算法设计并不神秘，最重要的是考虑问题要周全，许多对人类来说不言而喻的小问题，在设计计算机程序时都必须一一考虑到，考虑周全的方法就是要对问题建立完善的逻辑模型。

11.2.7 其他要求

若学员有兴趣，可进行以下工作：

① 改进判定赢局的函数，具体方法见 11.2.3 **导读框中的讨论。**

② 产生棋局算法的完善程度决定了下棋程序的设计水平，本书中所介绍的是一种最简单的方法。可以在其基础上改进产生棋局的算法，如：

- **增加考虑棋局上活四和活三的数量。**
- **不但考虑己方得分，还要考虑对方的失分。如是否可以减少对方的活四或活三的数量。**

③ 在学员中进行程序下棋比赛，看谁的算法最完善。

附录一　名 词 索 引

附录二　离散数学部分习题参考答案和提示

第二章　参考答案和提示

1.（1）0　（2）1　（3）1　（4）0　（5）0　（6）1　（7）1　（8）0

3.（1）是　（2）否　（3）是　（4）是

4.（1）C　（2）$B\wedge C$　（3）$(B\wedge C)\vee(A\wedge C)\vee(A\wedge B)$

6.（1）$A\wedge C$　0　（2）$A\vee C$　1　（3）$A\wedge B$　0　（4）$A\vee B$　1

（5）$C\rightarrow(A\wedge B)$　0　（6）$\neg B\rightarrow\neg A$　0　（7）$\neg C\rightarrow(\neg A\wedge\neg B)$　0　（8）$C\rightarrow(A\wedge\neg B)$　1

7. 设原子命题

A：恰当使用　B：自行拆卸过　C：在保修范围内

D：退货　E：未打开原包装　F：30 天之内

（1）$A\wedge\neg B\rightarrow C$　（2）$E\rightarrow D$　（3）$F\rightarrow D$　（4）上述各式的合取

导读：

在命题建模时，同一个问题描述产生多种不同的模型是很自然的。例如，对题（1），另一种表达方式是$\neg A\vee B\rightarrow\neg C$，它与 $A\wedge\neg B\rightarrow C$ 显然是等价的，因此，如果在练习中提出了其他不同的表达式，可以通过验证其等价性来判断它们的正确性。

8.（1）早餐包括在房费中或者卧铺不另加价。

（2）卧铺不另加价但不能选景点 A 而不另收费。

（3）要么卧铺不另加价，要么早餐不包括在房费中。

（4）选景点 A 另收费，但早餐不包括在房费中。

（5）如早餐包括在房费中，则卧铺要另加价或者景点 A 要另收费。

（6）如卧铺另加价、早餐也包括在房费中，则景点 A 另收费。

注意：这里有意将题（4）和题（6）设计得不合理，目的是希望学员严格按运算规则做题，而不是凭感觉做题。另外，要思考这个问题，应该如何改动命题，才能使之成为合理的条件？

9. 根据题意可设下列命题为推理前提：

G：肖像在金盒子中

S：肖像在银盒子中

则$\neg G\wedge\neg S$ 表示肖像在铅盒子中（既不在金盒子中也不在银盒子中）。盒子上的 3 句话分别可以表示为：

（1）$\neg G\wedge\neg S$

（2）G

（3）$G \vee S$（肖像不在此盒中，即肖像在金盒子中或者在银盒子中）

列出其真值表如附表 2-1 所示。

由于 3 句话必为两假一真，从真值表得知符合这种条件有两种可能：表中第一行和表中第二行，对应第一种情况，$S=1$,即肖像在银盒子中，对应第二种情况，$(\neg G \wedge \neg S)=1$,即肖像在铅盒子中。

概念辨析：看来巴萨尼奥并不一定能得到其心上人。本题说明直觉是靠不住的,只有用严格的数学方法才能得到正确的结论。

附表 2-1

G（语句 2）	S	$G \vee S$（语句 3）	$\neg G \wedge \neg S$（语句 1）
0	1	1	0
0	0	0	1
1	1	1	0
1	0	1	0

第三章 参考答案和提示

导读：

和命题建模一样，同一个问题描述产生多种不同的谓词模型也是很自然的。因此，这里给出的答案都只是若干可能的表达式中的一种。

3.（1）定义谓词公式

H（x，y）表示“x 比 y 高”。

某篮球国手为常数 c

则语句可以表示为：$\forall y\, H(c, y)$。

（2）定义谓词公式

H（x，y）表示“x 比 y 高”

则语句可以表示为：$\forall x \exists y H(x, y)$。

（3）定义谓词公式

$F(x)$ 表示 x 是分数

$R(x)$ 表示 x 是有理数

则语句可以表示为：$\forall x(F(x) \rightarrow R(x))$。

（4）定义谓词公式

$M(a, b)$表示 a 可以被 b 整除

$E(x)$表示 x 是偶数

则语句可以表示为：$\forall x(\neg E(x) \rightarrow \neg M(x, 2))$。

（5）定义谓词公式

$R(x)$表示 x 是一个实数

$Z(x)$表示 x 为 0

和数学函数

$f(x，y) = xy$

则语句可以表示为：$\forall x\forall y((R(x) \wedge R(y) \wedge Z(f(x, y))) \rightarrow (Z(x) \vee Z(y))$

（6）定义谓词公式

$P(x)$ 表示 x 是一个点

$L(x, y)$ 表示经过点 x, y 有一条直线

$N(x, y)$ 表示 x 不等于 y

则语句可以表示为：$\forall x\forall y((P(x) \wedge P(y) \wedge N(x, y)) \leftrightarrow L(x, y))$。

注意：因为两点必须不相等才能有一条直线，所以要增加命题 $N(x，y)$

（7）定义数学函数

$f(x, y) = xy$

$g(x, y)=z + y$

谓词公式

$R(x)$表示 x 是实数

$L(x, y)$表示 x 比 y 大

则语句可以表示为：$\exists x\exists y\exists z(R(x)\wedge R(y)\wedge R(z)\wedge L(f(x, y), g(x,z)))$

（8）定义数学函数

$f(x)$ 表示 x 的绝对值 $|x|$

$g(x)$ 表示 x 的平方

定义谓词公式

$L(x,y)$ 表示 x 比 y 大

则语句可以表示为：$\forall x\ ((L(f(x), 2)\wedge L(5, f(x)))\Leftrightarrow(L(g(x), 4)\wedge L(25, g(x))))$。

第四章　参考答案和提示

1.（1）{2,3,5,7,11,13,17,19}

（2）$\{x|x = 2n - 1, n\in$**N**（**N** 是自然数）$\}$

（3）$\{x|\ ax^2+bx+c=0, a\neq0, a,b,c\in$ **R**（**R** 是实数）$\}$

（4）首先定义集合

d（drama，戏剧时间）

m（music，音乐时间）

c（commercial，广告时间）

则节目组合可以用集合表示为

$\{\{d,m,c\},\{d,c,m\},\{c,d,m\},\cdots\}$ (顺序不同的安排应视之为不同的元素)

元素数量＝3!＝6

2.（1）$A \cup B$＝$\{a,b,c,1,2,3\}$，$A \cap B$=$\{a\}$，$A\backslash B$=$\{b,c\}$，$B\backslash A$=$\{1,2,3\}$

A'=$\{1,2,3\}$, B'=$\{b,c\}$，$A' \cup B'$=$\{b,c,1,2,3\}$, $A' \cap B'$=$\varnothing$

（2）因为用上述集合元素计算得：$(A \cup B)'=\varnothing$　　$A' \cap B'=\varnothing$, 所以 $(A \cup B)'=A' \cap B'$

同法可证 $(A \cap B)'=\{1,2,3,b,c\}$,　$A' \cup B'=\{1,2,3,b,c\}$，所以 $(A \cap B)'=A' \cup B'$

（3）试求集合 $A=\{\varnothing, 1, a\}$的所有子集和真子集。

子集：$\varnothing;\{\varnothing\};\{1\};\{a\};\{\varnothing,1\};\{1,a\};\{\varnothing,a\};\{\varnothing,1,a\}$

真子集：$\varnothing;\{\varnothing\};\{1\};\{a\};\{\varnothing,1\};\{1,a\};\{\varnothing,a\}$

4. **注意：证明过程不是惟一的，但结果应该相同。**

（1）原式＝$A \cup B$

（2）原式＝$A \cup (B \cup B') = A \cap \varnothing = A$

（3）因为 $(A \cup B \cup C) \subseteq (A \cup B)$，$(A \cup (B \backslash C)) \subseteq A$，**原式＝** $(A \cup B) \backslash A = B \backslash A$

5.**（1）非小说类书** $(U \backslash F)$

（2）参考书与非小说 $R \cup (U \backslash F)$

（3）既非参考书，又非小说 $(U \backslash R) \cap (U \backslash F)$

（4）计算机方面的教科书和参考书 $C \cap (T \cup R)$

（5）非计算机类的教科书和参考书 $(U \backslash C) \cap (T \cup R)$

6.**（1）百货部的每个售货员都完成了定额** $S \cap F \subseteq M$

（2）服装部的一些售货员没有完成定额 $(S \cap T) \backslash M \neq \varnothing$

（3）百货部和服装部完成定额的售货员都得到了奖金 $(F \cup T) \cap (M \cap S) \subseteq B$

7.**（1）价格** 12 000 **元以下的笔记本计算机。**

（2）有内置网卡或液晶显示器的计算机。

（3）不带内置网卡的液晶显示器计算机。

（4）价格 12 000 **元以下，保修** 3 **年或者液晶显示器的计算机。**

（5）价格 12 000 **元以下，但是不包括液晶显示器或者有内置网卡的。**

8. **用自然语言描述下述集合表达式：**

（1）$T \cup W$**：或者按类型查找或者按包含文字查找。**

（2）$B \cap A$**：在给定日期** a **与** b **之间生成的文件。**

（3）（$C \cup D$**）** $\cap$ W**：查找** C **盘或** D **盘上包含指定类型的文件。**

（4）（$U \backslash C$**）** $\cap$ **（**$S \cap L$**）：查找** C **盘外，大小介于** S **和** L **之间的文件**

9.**（1）设**

F**：只有第一次考试得** A **的学生，**$|F|=26$

S**：只有第二次考试得** A **的学生，**$|S|=21$

N**：从来没有得过** A **的学生，**$|N|=17$

两次都得 A **的学生**$= |F \cap S| = |F|+|S|-|F \cup S| = |F|+|S|-(|U|-|N|)=26+21-33=14$**。**

（2）设

F**：只有第一次考试得** A **的学生,** $|F|$

S**：只有第二次考试得** A **的学生，**$|S|$

N**：从来没有得过** A **的学生，**$|N|=4$

只得过一个 A **的学生＝**$|P \cup Q| = 40 = 2|F| - |P \cap Q|$

从来没得过 A **的学生＝**$|P \oplus Q| = 2|F| - 2|P \cap Q| = 4$

合并两式得 $2|F| - 2(2|F| - 40) = 4$, **解之得**$|F| = 38$**。**

答：只有第一次考试得 A **的学生数** 38**、只有第二次考试得** A **的学生数** 38**，**2 **次考试都得** A **的学生数** 36**。**

第五章　参考答案和提示

3．8 种，最短路径为 *abgch*。

4．*a,b,c,d,a*

5．（1）前序：*abdgehicf*

（2）中序：*gdbheiacf*

（3）后序：*gdhiebfca*

7. 该问题可用有向图来描述，用边的方向表示公文流转的去向。如果一份文件得到了所有 5 位主管的批准，它只能从财务主管开始，流转顺序是：FO（财务主管）→SM（销售经理）→PM（生产主管）→CP(公司经理)→MD(营销主任)。

8. 用结点表示课程、用边表示它们之间的先修要求，做出 AOV 图，结果表明：高等数学（*C*1）是除离散数学（*C*2）、形式语言与自动机（*C*5）以外所有课程的基础，离散数学（*C*2）是除高等数学（*C*1）、计算机原理（*C*8）和汇编语言（*C*9）以外各门课程的基础，但高等数学和离散数学的学习可彼此独立进行。

9. 将解题过程中的各种状态列于附表 2-2。从初始状态开始，一一列出由此可以到达的状态 0~状态 3（包括允许的和不允许的）。从而形成搜索树的 3 个分支，但是其中只有状态 1 是允许的，人可以将羊留在对岸，并返回继续将狗或者白菜带过河，达到状态 10 和状态 11。从状态 10，人可以将白菜或者羊又带回来。也就是说，从状态 10 可发展出两个状态 100,101。但其中 101 显然是不可取的，因为这又回到状态 1 了。同理，从状态 11 可发展出两个状态 110,111。其中 110 也是不可取的。

附表 2-2

状　　态	河的一岸	河的另一岸
0	人、羊、狗、白菜	–
1	狗、白菜	人、羊
2	狗、羊	人、白菜
3	羊、白菜	人、狗
10	狗	人、白菜、羊
11	白菜	人、狗、羊
100	狗、人、羊	白菜
101	狗、人、白菜	羊
110	白菜、人、狗	羊
111	白菜、人、羊	狗
…	…	…

依此类推，如能找到一条达到最后状态（羊、狗、白菜皆在对岸）的路径，问题就解决了。思考：把用该方法得到的答案与直觉加以比较，从中体会数学理论为什么是重要的。

附录三　如何阅读用形式文法描述的 C++语法规则

教材中对 C++语法的介绍只涉及与课程有关的一些重点，但 C++语言有着很丰富的内容和很强大的功能。在书后配盘中包括了用形式文法描述的 VC++语言的全部内容。为帮助学员能读懂语法描述，下面对形式文法的基本概念做一简单介绍。

定义：形式文法包括 4 项要素：

（1）终止符（terminal）的集合 T。

（2）非终止符（nonterminal）的集合 N。

（3）产生式的集合（production）P。

（4）在集合 N 中的一个特殊符号，起始符 S。

其中：

- 非终止符是用于描述句子结构的中间符号，在某种意义上相当于通常所说的句子成分，如主语、谓语、宾语等。
- 终止符是作为这些句子成分的具体词汇，如主语具体为 I、We 等，谓语则具体为 do、is 等。
- 产生式描述语言构成的一般形式为$a \rightarrow b$。其中a和b为非终止符或终止符，它表明为了生成最后形式的句子，需将a 转换为b。
- 起始符是该语言中所有句子的起始点。

给定语法后，即可按以下方法生成该语言中的句子。

（1）从起始符开始。

（2）如果当前的终止符和非终止符某个部分与某个产生式的左边内容相同，则代之以该产生式右边的内容。

（3）重复以上过程，直至所有的非终止符都被终止符所代替或没有产生式可应用时为止。前者得到该语言的一个合法句子，后者则需要从起始符开始，重新尝试应用不同的产生式。

例 1：

T=(a, the, dog, cat, chases, meets, runs)

N=(sentence, noun-phrase, noun, article, verb-phrase, verb)

P=(sentence->noun-phrase verb-phrase, [规则 P1]

noun-phrase->article noun,[规则 P2]

noun-phrase->noun,[规则 P3]

article->a,[规则 P4]

article->the,[规则 P5]

noun->dog,[规则 P6]
noun->cat,[规则 P7]
verb-phrase->verb noun-phrase,[规则 P8]
verb-phrase->verb，[规则 P9]
verb->meets,[规则 P10]
verb->chases,[规则 P11]
verb->runs [规则 P12]
)

S=(sentence)

下面是用产生式 P 生成一个句子的过程：

sentence->noun-phrase verb-phrase（应用规则 P2）
sentence->article noun verb-phrase（应用规则 P4、P6）
sentence->a dog verb-phrase（应用规则 P9）
sentence->a dog verb（应用规则 P12）
sentence->a dog runs（全部为终止符，代换结束）

至此，所有非终止符都转换为终止符，一个合法的句子产生。

将上述产生式以各种方式组合运用，可得到构成该语言集合的一系列句子，如：

a dog meets the cat
dog chases cat
the cat runs
...

例 2：

下面是用形式文法描述的关于 C++中语法词 Token 的语法规则：

token:
punctuator
keyword
identifier
constant
operator
comments

它相当于上述定义中的产生式$a \rightarrow b$。冒号前面是非终止符、冒号后面各行是并列的展开式，包括非终止符和终止符。习惯上一般将终止符用粗体表示，非终止符则用斜体表示。

上述语法规则的意思是，*Token* 这个语法成分可进一步展开为 *keyword*、*identifier* 等语法成分中的某一种，而它们又是非终止符，还可进一步展开。例如，关于 *identifier* 的语法规则是：

identifier:
nondigit
identifier nondigit
identifier digit

nondigit: one of

a b c d e f g h i j k l m n o p q r s t u v w x y z _
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

digit: one of

0 1 2 3 4 5 6 7 8 9

注意该规则用了递归形式，意为：*identifier* **这个语法成分可进一步展开为** *nondigit* **或另一个** *identifier* **后跟一个** *nondigit*（**非数字**）**或另一个** *identifier* **后跟一个** *digit*（**数字**）**。只有当** *identifier* **展开为** *nondigit* **并进而展开为终止符，即** 26 **个大小写字母加上下划线中的一个时，才能产生一个合法的** *identifier*（**标识符**）**。这就是语法规定变量名必须是以非数字字母开头的字符串的原因。**

参 考 文 献

1 M A Ellis, M B Stroustrup. The Annotated C++ Reference Manual. Addison-Wesley, 1990

2 H R Lewis, C H Papadimitriou. Elements of The Theory of Computation. 2ed. Prentice-Hall International Inc, 1998

3 L C Liu. Elements of Discrete Mathematics. 2ed. McGraw-Hill Book Company, 1977

4 J R Norman. Object-Oriented Systems Analysis and Design. Prentice-Hall International Inc, **北京：清华大学出版社，**1997

5 L T Pirnot. Mathematics All Around, Addison Wesley Longman Inc.**北京：机械工业出版社，**2002

6 E Stiller, C LeBlanc. Poject-Based Software Engineering: An Object-Oriented Approach. Addison-Wesley, 2002

7 B Stroustrup. The Design and Evolution of C++. **北京：机械工业出版社，**2002

8 **卜月华. 图论及其应用. 南京：东南大学出版社，**2000

9 **耿素云，屈婉玲，张立昂. 离散数学. 第** 2 **版. 北京：清华大学出版社，**1999

10 **屈婉玲，耿素云，张立昂. 离散数学题解. 北京：清华大学出版社，**1999

参 考 网 站

1 http://www.uml.org.**关于** Unified Modeling Language **的标准**

2 http://www.omg.org.Object Management Group **网站，关于面向对象概念和方法**