算法＋数据结构＝程序

〔瑞士〕N. 沃思 著

曹德和　刘塘年 译

丘玉圃 校

科学出版社

1984

内 容 简 介

本书从结构程序设计的观点出发，论述了数据结构和算法结构.

全书共分五章. 第一章介绍基本的数据结构. 第二章介绍数组与文件的各种分类算法，并对其优缺点进行了比较. 第三章讨论递归算法，并仔细论述了回溯算法的设计. 第四章讨论动态数据结构，如线性表、树(包括检索树、均衡树、B 树)和杂凑表，以及在它们之上的各种操作算法. 最后一章简要地介绍形式语言的概念，并为一个小的简单程序设计语言 PL/0 编制了编译程序. 各章都附有习题. 全部程序都是用 PASCAL 语言书写的.

虽然本书起点较高，但内容深入浅出，因此对不同水平的软件工作者都有参考价值. 本书可以作为计算机学科有关专业的教材和参考书.

Niklaus Wirth
ALGORITHMS + DATA STRUCTURES = PROGRAMS
Prentice-Hall 1976

算法+数据结构=程序

〔瑞士〕N. 沃思 著
曹德和 刘椿年 译
丘玉圃 校
责任编辑 杨家福 刘晓融
科学出版社出版
北京朝阳门内大街 137 号
中国科学院印刷厂印刷
新华书店北京发行所发行 各地新华书店经售
*
1984 年 10 月第 一 版 开本：850×1168 1/32
1984 年 10 月第一次印刷 印张：13 5/8
印数：0001—25,700 字数：358,000
统一书号：15031・602
本社书号：3715・15—8
定价：3.10 元

目　　录

序　言

近年来，计算机程序设计这一课题已被认为是一门可以给予科学解释和描述的学科．掌握这门学科对于成功地进行工程设计是基本的和重要的；这门学科已经从一种技巧发展成一门学问．最早对这一发展作出杰出贡献的是 E. W. 戴克斯特拉 (Dijkstra) 和 C. A. R. 霍尔 (Hoare) 两人．戴克斯特拉的“Notes on Structured Programming”(结构程序设计札记)[1] 一文，开创了把程序设计看作一门科学的新观念，对人类智力提出了挑战，揭开了程序设计“革命”的新篇章．霍尔的“Axiomatic Basis of Compute Programming”(计算机程序设计公理化基础)[2] 一文清楚地表明，可以在数学推理的基础上对程序进行严格的分析．这两篇论文都令人信服地证明，如果使程序员认识到他们迄今不曾了解而又常常直接采用的方法和技术，就能防止许多程序设计的错误．这些论文都把注意力集中到程序的构成和分析方面，说得更明确点，就是集中到由程序正文所代表的算法的结构上．然而，十分清楚的是，对程序构造进行系统而科学的研究，首先是对包含复杂数据集合的大型复杂程序而言的．因此，程序设计的方法学必然包含数据结构的一切方面．其实，程序就是在数据的某些特定的表示方式和结构的基础上对抽象算法的具体表述．霍尔通过“Notes on Data Structuring”(数据结构札记)[3] 一文所做的一个突出贡献，就是澄清了关于数据结构的术语和概念方面的杂乱局面．该文说明，不了解施加于数据上的算法就无法决定如何构造数据，反之，算法

1) 见 Dahl, Dijkstra and Hoare, Structured Programming, New York, Academic Press, 1972, pp.1—82．中译本：陈火旺、吴明霞、丁宪理译，结构程序设计，科学出版社，1980，第 1—105 页．

2) 见 *CACM*, **12**, **No.** 10 (1969), 576—583.

3) 见《结构程序设计》，第 106—215 页．

的结构和选择却常常在很大程度上依赖于作为基础的数据结构. 简而言之，程序的构成与数据结构是两个不可分割地联系在一起的问题.

然而，本书却以数据结构一章开始. 这是因为: 首先，人们的直观概念是数据先于算法——你总得先有对象才能对它们施加运算;其次，更直接的理由是，本书假定读者已熟悉计算机程序设计的基本概念. 不过，导论性的程序设计课程传统上都是明智地集中讨论运行于相对简单的数据结构上的算法. 因此，我们设置导论性的数据结构一章看来是可取的.

全书，尤其是第一章，我们采用霍尔[1]所阐述的理论和术语，并用程序设计语言 PASCAL[2] 来实现. 这一理论的本质就是: 数据首先是代表客观现象的抽象，最好表示成不必在普通程序设计语言中实现的抽象结构. 在构造程序的过程中，随着算法的精细化，数据表示方法也逐步求精，越来越符合所使用的程序设计系统的限制[3]. 因此，我们为所谓基本结构的数据结构假定了一些基本构成原则. 最重要的一点是，已知这些结构是很容易在现有计算机上实现的，因为只有这样，它们才能作为分子出现在数据描述逐步求精过程的最后一步，而被用作实际数据表示中的真正基本元素. 这些结构是: 记录、数组(大小固定的)和集合. 毫不奇怪，这些基本的构成原则是与数学上同样基本的概念相对应.

这种数据结构理论的基石在于把基本结构和"高级"结构区别开来. 基本结构是分子(它们自己又是由原子构成的)，是高级结构的组成成分. 基本结构的变量只改变自己的值，从不改变其结构和取值的集合. 因此，它们占用的存贮空间的大小是不变的. 而"高级"结构的特点是在程序执行过程中既改变值又改变结构. 因此，它们的实现需要更为复杂的技术.

1) "Notes of Data Structuring".

2) N. Wirth, "The Programming Language Pascal"(程序设计语言 Pascal), *Acta Informatica*, **1**, No. 1 (1971), 35—63.

3) N. Wirth, "Program Development by Stepwise Refinement"(逐步求精的程序编制), *CACM*, **14**, No. 4(1971), 221—227.

顺序文件——或简称序列——在这样分级时是作为混合型结构出现的，它确实改变自己的长度；但这种结构上的改变是很简单的. 因为实际上所有的计算机系统中顺序文件都是真正基本的，所以，在第一章，我们把它放在基本结构部分.

第二章处理分类算法. 文中列举了为达到同一目的的各种各样的方法. 对其中某些算法的数学分析表明了这些方法的优点与缺点，使程序员认识到为给定问题选择好的解决方法而进行分析的重要性. 数组分类法与文件分类法（常常称为内部分类与外部分类）的区分表明，数据的表示方法对于选择合适的算法和算法的复杂性确有重要影响. 本书之所以为分类算法安排了如此大的篇幅，正因为它是一种理想的媒介，可以用来说明许多程序设计原理和在大多数其它应用中发生的情况. 常有这样的情形：似乎一个人只从分类算法中选取例子就可以构成完整的程序设计课程.

另一个通常被导论性程序设计课程忽略，而在许多算法求解概念中起着重要作用的论题是递归. 因此，第三章专门讨论递归算法. 递归作为重复（迭代）计算方法的推广，在程序设计中是一个重要的、有力的概念. 可惜，在许多程序设计教材中，所举的例子都是只用简单的迭代即可满足要求的情况. 本书第三章与此不同，集中考虑了几个例题，此时递归是最自然的求解办法；而若用迭代，产生出来的程序将是繁复而不清晰的. 回溯算法是递归的一种理想应用. 但是最明显应该使用递归的是那些运行在已递归定义的数据结构上的算法. 这些情况在最后两章处理，为此第三章提供了良好的背景.

第四章处理动态数据结构. 这种数据在程序执行时会改变自己的结构. 本章说明递归数据结构是常用的动态结构的一个重要子集. 在这些情况下，虽然递归定义既自然又可行，但通常在实际中并不使用，而是强令程序员使用显式指示或指针变量的办法使他明白实现这种定义的方法. 本书采用了这种方法并反映了这种技巧的现状：第四章专门讨论使用指针的程序设计、表、树以及包含着更复杂的数据网的例子. 本章还介绍了通常所谓的“表处理”

(这一名称并不太恰当)；还为树组织，特别是检索树安排了相当多的篇幅．最后描述了散列表，亦称"杂凑"代码，它常常比检索树更受欢迎．这就提供了对经常遇到的应用问题中两种根本不同技术进行比较的可能性．

最后一章包括形式语言定义和分析问题的简明引论，还在一台简单计算机上为一种简单语言构造了编译程序．写这一章的动机是三重的．首先，一个成熟的程序员对程序设计语言的编译过程中的基本问题和技术至少应有某种程度的理解．第二，为便于操作而需要一个简单输入或控制语言定义的应用问题在不断增多．第三，形式语言在符号序列上定义了一种递归结构；因此，它们的处理程序就是有效应用递归技术方面的很好例子，而为了得到大型或超大型程序的显明结构，这种递归技术是极为重要的．如果选取的语言的编译程序太长，就不能有效地包括在一本不是专为编译程序专家而写的书中；而如选取的语言过于简单，又不足以作为一个有效的例子．作为折中，我们选取了 PL/0 语言作为样板．

程序设计是一种构造性的技术．怎样开展这种构造性的又具有创造力活动的教学呢？一个方法是从许多实例中总结出基本的构成原则，并以系统的方式来展现它们．不过程序设计这一领域极为纷繁多变，常常包含着复杂的脑力活动．因此，以为能够把它压缩成一种纯粹"开药方"式的训练是错误的．我们所能采用的教学方法只能是仔细地选择和描述标准的例子．自然，我们不能指望每一个人都能从例子中学到同样多的东西．这种方法的特点就是把许多东西留给学生，让他们通过勤奋地学习和直觉去体会．对于相对来说又复杂又长的程序，这一点尤如此．这种程序出现在本书中不是偶然的．实用中较长的程序才是"正常"的情况，而且它们更适用于展示难以捉摸的然而是重要的东西——风格和整齐的结构．我们也打算用它们作为读程序的练习，通常由于强调写程序而过于忽略了读程序．这正是本书采用完整的长程序作为例子的最初动机．读者将通过程序的逐步发展过程得到一个程序选

展中的各个阶段，从而看出随着细节的逐步精细化，这一发展过程是显然的．我认为把程序表为充分考虑细节的最终形式是很重要的，因为程序设计中，错误正是隐藏在细节之中．尽管对于学者们来说，纯粹描述算法原则及其数学分析可能具有刺激性和挑战性，但这对于实际工程人员似乎是不实在的．因此，我严格遵循这一原则：将程序的最终形式以某一语言表述出来，以便确实能在计算机上执行．

当然，这就要求找到一个描述形式，该形式既便于机器执行，又充分独立于机器．在这方面，无论是广泛使用的语言，还是抽象的记号，都被证明是不合用的．PASCAL 语言却正好适用，它就是为着这个目的而研制出来的，所以本书也就采用了它．凡熟悉其他高级语言如 ALGOL 60 或 PL/1 的程序员很容易理解这种程序，因为，随着课文的进展很容易理解 PASCAL 的记号．然而，这并不是说预先做些准备就没有好处．为此，《Systematic Programming》（系统化程序设计）[1]一书提供了理想的背景，因为它也是基于 PASCAL 记号的．然而，本书并不打算作为 PASCAL 语言的使用手册，已有更适于这一目的的材料[2]．

本书既是苏黎士联邦技术学院（ETH）几门程序设计课程的缩写本，但与此同时又作了精心改进．本书的许多思想和观点都和我在 ETH 的同事们讨论过．我特别要感谢 H. Sandmayr 先生，是他仔细地阅读了手稿；感谢 Heidi Theiler 小姐，是她认真耐心地打印了课文．还应指出，IFIP（国际信息处理联合会）2.1 和 2.3 工作小组所举行的会议给了我很大影响，特别是这一期间我与 E. W. 戴克斯特拉和 C. A. R 霍尔进行过许多值得纪念的讨论．最后，还应指出的是，ETH 慷慨地为我提供了工作条件和计算设备，否则，本书的准备工作将无法进行．

N. 沃思

1) N. Wirth (Englewood Cliffs, N. J.: Prentice-Hall, Inc., 1973).

2) K. Jensen and N. Wirth, "PASCAL——User Manual and Report" (PASCAL——用户手册与报告) *Lecture Notes in Computer Science*, Vol. 18 (Berlin, New York; Springer-Verlag, 1974).

第一章　基本数据结构

1.1 引　言

现代数字计算机原是作为能方便而快速地进行复杂、耗时计算的工具而发明的．可是，现在它在大多数的应用中，存取大量信息的能力却起着支配作用，从而，这个能力被认为是计算机的首要特征，而它的计算能力，即执行算术运算的能力在许多情况下几乎已经变成无足轻重的了．

对所有要处理大量信息的情况而言，这些信息在某种意义上代表着对现实世界的一部分抽象．计算机所用的信息是由一个从现实世界选出的数据集合组成的，这一数据集合被认为与要解决的问题有关，并相信由此可以导出所需要的结果．数据是在如下意义上代表现实抽象的：现实对象的某些性质和特征由于同特定的问题不相干而被忽略．因此，抽象亦即是对事实的简化．

我们可以取一位雇主的人事文件为例．在这个文件中，每一位雇员由一组或者是雇主所需，或者是其会计程序所需的数据来代表（即抽象）．这组数据可能包括雇员的身份，如姓名和薪额．但极不可能包括像头发的颜色、体重和身高这类不相干的数据．

无论使用计算机与否，解决一个问题总是要选择一种对现实的抽象，亦即要定义一个代表现实情况的数据集合．这一选择必须受所要解决的问题的制约．然后要选择表示这一信息的方法．这一选择受解决问题的工具所制约，亦即受计算机所提供的方便条件制约．在大多数情况下，这两步选择并非全然无关．

数据表示方法的选择通常是很困难的，而且不是由使用的设备所能唯一决定的．它必须依据欲在数据上施行的运算来考虑．数的表示方法是一个很好的例子，数本身是被描述客体性质的抽

象．若欲施行的运算只是(或至少"几乎只是")加法，则把数 n 表示成 n 条笔划是一个好办法．用这种表示，加法规则确实是既明显又简单．罗马数字是建立在同一简单性原则上的，对于小的数字来说，加法规则具有类似的直接性质．另一方面，若用阿拉伯数字表示法，则(对小的数字)加法规则就很不直观，必须逐一记住．但是，若考虑到大数的加法或乘除法时，情况就倒过来了．在阿拉伯数字表示法中，把这些运算分解成较简单的运算就要容易得多，因为阿拉伯数字表示法具有建立在数字进位基础上的系统构造原则．

众所周知，计算机使用基于二进数字(0和1)的内部表示法．这一表示法并不适合于人类，因为它通常包含大量的数位，但它却最适合于电路，因为0和1这两个值可以方便而稳定地由电流、电荷、磁场的存在与不存在来加以表示．

从这个例子我们还可看出，表示法这一问题通常要分为几个细致的步骤．给定一个表示方法的问题，比如物体的位置，第一个决策可能导致选择一对实数，比如说笛卡尔坐标或极坐标．第二个决策可能导致选择浮点表示，每一个实数 x 由两个整数来表示，一个表示小数 f，另一个表示对某一底的指数 e(例如 $x=f\cdot 2^e$)．因为知道数据要存到计算机里去，所以第三个决策可能导致采用二进制的位置表示法来表示整数，最后一个决策可能是用磁存贮器件的磁通量的方向来表示二进制数字．显然，在这一系列步骤中，第一步主要受问题状况的影响，而后面几步则越来越依赖于计算工具及其技术．所以，很难要求程序员去决定用什么样的数表示法，甚至决定存贮器特性．这些"低级决策"可以留给计算机设备的设计者们，他们最了解当前的技术状况，能够作出一个明智的选择，使这个选择适用于所有(或几乎所有)的应用问题，其中包括数的应用问题．

在这个意义上，程序设计语言的作用是明显的．一种程序设计语言代表一台能够理解该语言术语的抽象计算机，这些术语可以体现实在机器所用的对象在某种程度上的抽象．这样，只要数

是该语言范围的基本对象，这种“高级”语言的程序员就不必(也不允许)插手数的表示法问题.

这种语言提供了大多数数据处理问题所共有的一组方便的抽象，使用这种语言的重要性主要在于结果程序的可靠性. 用我们熟悉的数、集合、序列以及重复等概念进行推演来设计程序比用二进位、“字”、转移等要容易. 当然，不论数据是数、集合还是序列，一台实在的计算机总要把所有的数据都表示成一大堆二进位. 但这与程序员不相干，他不必为自己所选择的抽象概念表示方法的细节而操心，只要他能确信计算机(或编译程序)所选择的相应表示方法适用于他的目的.

抽象概念与给定的计算机越接近，工程师或语言的实现者就越容易做出表示方法的选择，并且使一种选择能适用于所有(或几乎所有)可以想象到的应用问题的可能性也就越大. 这一事实限制了对给定的实在计算机进行抽象的程度. 例如，在通用语言中，把几何对象也包括在基本数据项目中是没有意义的，因为它们的合适表示方法具有固有的复杂性，这一表示方法将在很大程度上依赖于欲施行于这些对象上的运算. 而这些运算的性质和出现频率是通用语言及其编译程序的设计者所不了解的，他所作的任何选择都可能对某些潜在的应用问题不合适.

在本书中，这些考虑决定了选择何种记号系统来描述算法及其数据. 显然，我们希望使用熟悉的数学概念如数、集合、序列等，而不使用像二进位串这样的与计算机有关的实体. 但同样清楚的是我们也希望使用那些已知确实存在有效编译程序的记号系统. 使用紧密面向机器和依赖于机器的语言也同样是不明智的，因为这无助于用抽象记号描述计算机程序，而这种抽象记号要为具体表示方法问题留出充分选择余地.

程序设计语言 PASCAL 设计的目的就是试图在这些极端之间寻求折中，本书将一直使用它[1-3, 1-5]. 这个语言已成功地在几台计算机上实现了，并已证明这一记号系统充分接近于实际机器，其选用的特性及其表示方法可以清楚地加以解释. 这个语

言也充分接近于别的语言，特别是 ALGOL 60，所以这里所教的课程内容也同样适用于别的语言.

1.2 数据类型的概念

在数学中，习惯按照某些重要特性对变量进行分类. 实型、复型与逻辑型变量之间，表示单个值、值的集合或集合的集合的变量之间，函数、泛函、函数集合之间，诸如此类的东西是明显区分开来的. 在数据处理中，分类的概念即使不是更为重要的，至少也是同等重要的. 我们将遵循这一原则：每一个常量、变量、表达式或函数均属于某一类型. 这个类型本质地刻画了常量所属的、变量或表达式所能取的、函数所能生成的值集合的特征.

在数学文献中，变量的类型不必参照上下文，通常仅从其印刷铅字的字面上就可判定；而这在计算机程序中却行不通. 因为在计算机设备中，通常只使用一种字体（即拉丁字母）. 因此广泛采用如下的规则：通过常量、变量或函数的说明规定其相应的类型，并且说明总是在正文上先于该常量、变量或函数的使用. 考虑到编译程序必须选择对象在计算机存贮器内的表示方法，这一规则显得特别有意义. 显然，分配给一个变量的存贮容量必须根据该变量可能取值范围的大小来选择. 倘若编译程序已知这一信息，就可避免所谓的动态存贮分配. 这常常是获得一个算法高效实现的关键.

这样，本书所用的，并体现在程序设计语言 PASCAL 中的类型概念的主要特征如下[1-2]：

1. 数据类型确定了常量所属的值的集合，确定了变量、表达式所能取的值的集合，也确定了运算符、函数所能生成的值的集合.

2. 常量、变量或表达式所表示的值的类型，从其书写形式或其说明即可推知，而不需执行计算过程.

3. 每一运算符或函数要求固定类型的变元，并产生固定类型的结果. 如果一个运算符允许几种类型的变元（如运算符“+”用

于整数相加，亦用于实数相加)，则结果的类型可由特定的语言规则来确定.

因此，编译程序可以使用这些类型信息来检查各种结构的相容性和合法性. 例如，不必执行程序即可发现把布尔(逻辑)值赋给算术(实)变量的错误. 程序正文中的这种冗余技术协助程序编制是极为有用的，必须把它看成是良好的高级语言优越于机器代码(或符号汇编代码)的主要一点. 显然，不管程序是用带有类型概念的高级语言写的，还是用无类型的汇编代码写的，最后总要把数据表示成一大堆二进制数字. 对于计算机来说，存贮是没有明显结构的一大堆均匀的二进制位. 但是，恰恰是这种抽象结构使得程序员能在单调的计算机存贮中识别出它的意义.

本书提供的理论和程序设计语言 PASCAL 规定了定义数据类型的某些方法. 在大多数情况下，新的数据类型是用前面已定义过的数据类型定义的. 这种类型的值通常是前已定义的成分类型的成分值的集成物，并称这样的值为构造的. 如果只有一种成分类型，亦即所有成分均为同一成分类型，则它称为基类型.

属于类型 T 的不同值的个数称为 T 的基数. 类型为 T 的变量 x 记为 $x:T$，为表示变量 x 所需的存贮量，基数提供了一种度量.

因为成分类型又可以是构造的，所以可以建立完整的层次结构，但是，显然一个结构的最终成分必须是不可分的. 因此，还有必要提供引入这种基本的、非构造类型的表示法. 一种直接的方法是枚举构成该类型的值. 例如，在一个有关平面几何图形的程序中，可以引入一个称为 *shape* (形状)的基本类型，它的值可用标识符 *rectangle* (长方形)，*square* (正方形)，*ellipse* (椭圆)，*circle* (圆)来表示. 但是，除这种由程序员定义的类型外，还必须有某些预先已定义的标准类型. 它们通常包括数和逻辑值. 如果在各个值之间存在一种次序，这种类型称为有序的或标量. 在 PASCAL 中，所有非构造类型都假定是有序的，在显式枚举的情况下，假定值就是依它们的枚举序列排序的.

有了这些，就有可能定义基本类型，并建立集成的、构造的类

型至任意嵌套级别．在实用上，只用一种普遍的办法将成分类型组合到一个结构中是不够的．考虑到表示法与使用中的实际问题，通用程序设计语言必须提供几种构造方法．在数学的意义上，它们可能是完全等价的，它们的区别在于构造它们的值和选择它们的值的成分时所用的运算符．这里提供的构造方法是数组、记录、集合和序列(文件)．更为复杂的结构通常不作为"静态"类型定义，而代之以在程序执行过程中"动态"地生成，在执行过程中，它们可能有不同的大小和形式．这样的结构是第四章的内容，第四章还包括表、环、树和一般的有限图．

把变量与数据类型引进程序中是供计算时使用的．为此必须使用一组运算符．就像数据类型一样，程序设计语言提供一定数量的基本的、标准(原子)的运算符，以及提供若干构造方法．有了这些方法就可以用基本运算符定义合成运算．运算的合成问题常被认为是程序设计技巧的核心．不过，今后将会表明，数据的恰当合成同样是基本的和重要的．

最重要的基本运算符是比较和赋值，即检验相等(或在有序类型的情况下检验次序)和强令相等．在本书中，这两个运算符之间的基本差别由其记号上的明显不同而加以强调（尽管不幸的是在FORTRAN 和 PL/1 这样广泛使用的程序设计语言中，由于使用等号作为赋值号而造成了模糊）．

检验相等：$x = y$

对 x 赋值：$x := y$

这些基本运算符是对所有数据类型定义的，但应注意到，如果数据是大型的、高级构造的，那末它们的执行可能包含着相当大量的计算工作．

除了检验相等(或次序)和赋值之外，一类基本的和隐含定义的运算符就是所谓转换运算符．它们把数据类型映射到别的数据类型上．对于构造类型它们尤其重要．结构值通过所谓构造器由其成分值生成，而其成分值通过所谓选择器被提取．这样，构造器和选择器就是将成分类型映射到构造类型或者将后者映射到前