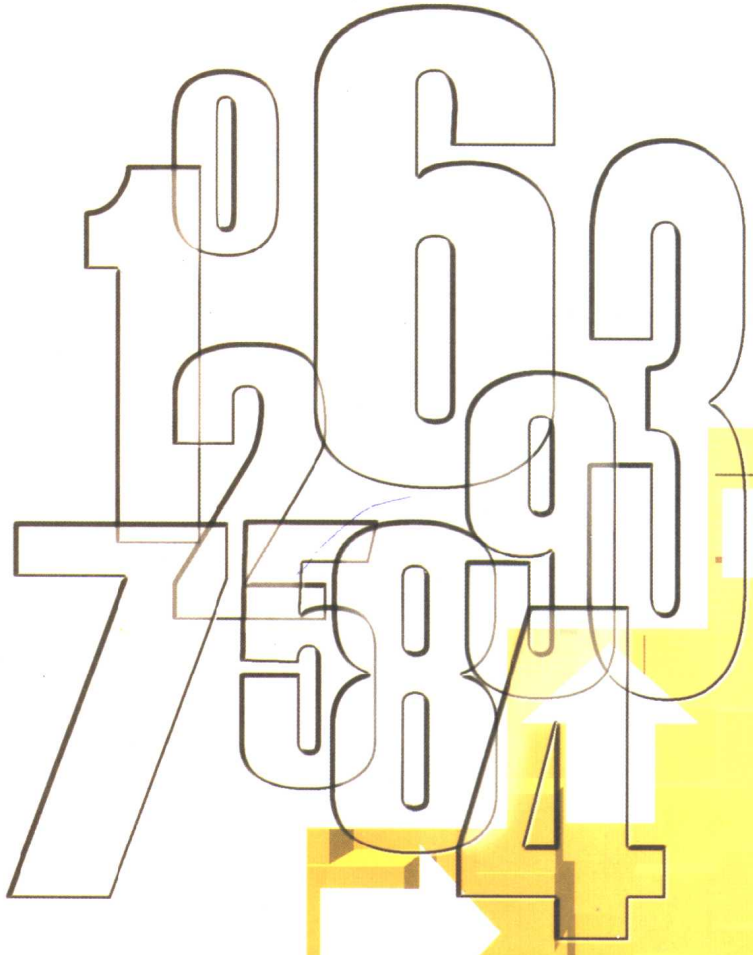
21世纪 高等学校本科系列教材

计算机软件技术基础

(7)

唐培和 薛弘晔 谢维成 编著



重庆大学出版社
新疆大学出版社

计算机软件技术基础

唐培和　薛弘晔　谢维成　编著

重庆大学出版社
新疆大学出版社

图书在版编目(CIP)数据

计算机软件技术基础/唐培和主编. —乌鲁木齐:新疆大学出版社;重庆:重庆大学出版社,2001.8

土木工程专业、电气工程及其自动化专业本科系列教材

ISBN 7-5631-1381-9

Ⅰ.计... Ⅱ.唐... Ⅲ.程序设计-高等学校-教材 Ⅳ.TP311

中国版本图书馆 CIP 数据核字(2001)第 053478 号

计算机软件技术基础

唐培和 薛弘晔 谢维成 编著

责任编辑 谭 敏 王军凤

*

重庆大学出版社
新疆大学出版社 出版发行

新 华 书 店 经 销

四川自贡新华印刷厂印刷

*

开本:787×1092 1/16 印张:21.25 字数:530 千

2001年8月第1版 2001年8月第1次印刷

印数:1—6000

ISBN 7-5631-1381-9/TP·4 定价:30.00 元

前言

按照教育部[1997]155号文件以及教育部工科计算机基础课程教学指导委员会的指示精神，工科非计算机专业计算机基础教学应按三个层次组织教学，即计算机文化基础、计算机技术基础和计算机应用基础。对于计算机软件技术基础所涉及的内容也明确地指出："软件技术基础包括计算机软件中的一些基本概念；人机交互基本工具（含高级语言）的有关知识；结构化程序设计及面向对象程序设计的概念与方法；以及软件重用、软件集成、软件工程等基本思想。在实践环节上，要使用90年代先进的软件开发环境与平台，让学生能熟练地使用这种工具"。并进一步指出该课程的目的是"使学生掌握计算机软件技术的基础知识、基本思想和基本方法；培养学生利用计算机处理问题的思维方式和利用计算机软硬件技术与先进工具解决本专业与相关领域中一些问题的初步能力"。这是编写本书总的指导思想。

长期以来，我们只给工科非计算机专业的学生讲授一门程序设计语言，主要介绍语言的词法、语法及其语句的使用，存在的最大问题是学了语言仍不会设计程序。目前有许多介绍某种程序设计语言的书籍都以"……语言程序设计"命名，但往往还是以介绍语言的词法、语法和语言的结构为核心，涉及程序设计原理、技术、方法等方面的内容并不多。另外，即使在语言课中讲授一些程序设计的内容，也是从某种具体语言的角度而不是从程序设计的角度来介绍的。这就造成学生把某种具体语言的知识学到了，但不能把所学到的语言知识贯通起来，不能利用所学到的知识进行程序设计，这是与学习程序设计语言的目的相悖的。

从程序设计的角度看，语言的词法、语法和语言结构是比较简单的，掌握它们相对来说比较容易，而学习程序设计相对来说较难一些，这是因为：程序设计对人的逻辑思维和形象思维以及对问题的描述和表达的能力要求较高，并且严格。而人们通常所接受的训练主要是各个单项的训练。人们缺乏的就是程序设计所需要的综合能力。所以程序设计的重点应该是培养学生的逻辑思维、形象思维、分析和表达、描述问题和解决问题的综合能力。

本课程的基本定位是：结合本专业的实际需要，能开发出一定规模（几百行到几千行左右）的应用程序。围绕着这一目标来组织教学内容（知识点）。我们认为对于非计算机专业的学生来说，学完一门高级语言（如 C、Pascal 等），若编不出一定规模的程序是没有意义的；当然，超过一定规模的程序要求非计算机专业的学生来掌握，既是困难的，也是没有必要的（可由计算机专业人才来解决）。要具备开发一定规模程序的能力，只学一门语言课显然是不够的。

该课程内容庞杂、学时有限，历来是一个难于解决的问题。程序设计语言、算法与数据结构、软件开发与软件工程、多媒体与用户界面、操作系统、数据库、网络、工具与环境等都是需要涉及到的知识点，加上最新技术的发展，其中的名词术语就有一千余条。当前国内有三种处理办法：一为浓缩型，将以上内容的每一部分浓缩后成为一“拼盘式”的教程。这种教程对教师、学生要求较高且易流于“不深不透”，经实践证明并不合适。二是概括型，围绕实际问题的应用开发展开，能覆盖多少知识点就覆盖多少。这种方式的不足是不够系统和全面的，可能出现明显的薄弱环节。三是分散型，一种技术一门课，连硬件技术课程一起，共 20 几门任由学生选择。这种方式虽然能够“学以致用”，但学时偏多，也缺乏精练的系列教材。我们认为该课程应围绕着非计算机专业的学生进行程序设计或软件开发所需要的知识（概念、原理、方法、技术、经验等）来组织教材内容，既不能简单地“浓缩”加“拼盘”，也不能过分地强调计算机学科的理论性和系统性，毕竟教材的对象是非计算机专业的学生。另外，限于篇幅，本教材不专门介绍某一门具体的高级程序设计语言（如 C、PASCAL 等）。

我们认为，教材内容与教学内容不应该完全一致。教材注重系统全面，而教学采用概括型，可以应用开发实例作为驱动，覆盖不到之处让学生自学。从“以教为主”到“教与自学

相结合”，并逐步过渡到“以学为主”。“以学为主”更需要系统全面的参考教材，教材应该“宽编窄用”。但篇幅不能过大，内容叙述强调整体性，着重原理和关键机制，强调程序设计的概念与方法，强化程序思维，代码细节能省就省或用伪代码描述（虽然不同的程序设计语言，它们在词法、语法及语言结构上有许多不同之处，但用不同的程序设计语言编写程序的设计思想和方法却具有共性）。这样虽有“不够明晰”之嫌，却能给教师、学生留下一定的余地。实验时学生可用 C、PASCAL、FORTRAN 或其他语言实现。

建议采用本教材的学校和教师，期末不一定安排笔试，可以大作业的方式评定成绩，因为我们的最终目的是要求学生具备一定的程序设计或软件开发能力，而不仅仅是掌握一定的知识。大作业题目可指定，也可自定，互不相同，最后的源代码最低不得少于 400～500 行。大作业所需课时较多，课内安排一部分，课外学生自己安排一部分。

本书由唐培和同志任主编，薛弘晔同志任副主编，全书共有 14 章，从内容上来说可划分为 3 个部分：程序设计理论、程序设计技术、程序设计语言与环境。其中第 1、2、3、4、5、6、8、9、13、14 章由广西工学院计算机系唐培和编写；第 7、11 章由陕西工学院计算机系薛弘晔编写；第 10、12 章由四川工学院谢维成编写；附录一、二、三、四由唐培和提供，附录五由谢维成编写。全书最后由唐培和负责统稿。

在这里，我们要感谢北京航空航天大学的杨文龙教授，他在百忙之中详细审阅了本书的初稿，并提出了很多好的意见和建议。在本书的编写过程中，借鉴了前辈们许多有益的经验与思想（见参考文献），在此一并表示感谢。

由于作者水平有限，本书在结构和内容选取上肯定还有不少值得商榷之处；加上时间匆忙，错漏难免，敬请读者批评指正。

唐培和

2001 年 5 月 1 日于柳州

目录

第 1 章
程序设计导论

到目前为止，使用计算机的方式不外乎两种，一种方式是利用现成的软件（工具），通过其提供的命令或手段，控制计算机的运行，以达到我们使用计算机的目的；另一种方式是用户按照自己的需求开发软件或设计程序，然后交给计算机运行，以获得问题的解。前者简单、容易，但有一定的局限性，它难以满足用户的特殊要求；后者有一定的难度，它要求用户有一定的软件开发或程序设计能力。

在用户掌握了一门程序设计语言后，自然对程序设计或软件开发感兴趣。本章从程序设计的概念、特点和范畴出发，力图使读者对程序设计有一个正确的认识。

1.1 程序设计的概念

如前所述，我们使用计算机的基本手段之一是给计算机设计程序。程序是有序的计算机指令的集合（当然严格地说不能叫集合），即若干指令的序列。计算机一步一步地执行了这个指令序列，就完成了我们希望它做的事情。编制程序使计算机得以实施正确动作的全部工作就叫程序设计。通俗地说，程序设计就是按照一定的要求给计算机编排一个合理的动作序列。

编制程序的工作并不是计算机出现以后才有的，也不是惟有计算机才使用程序。早在几千年前的宗教仪式、会议开始时会议主席宣布的议程、生产作业计划、产品零件制造的工艺规程、科研计划乃至每周必行的课程表都是程序。有的是顺序程序，如科研规划，顺序地执行一次就完了。有的是循环程序，如课程表，它被循环反复执行多次。有的是通用程序，如宗教仪式，几代人不变。

在计算机出现之前，程序的执行者是人，如果程序有错，执行中可以补救。计算机程序的执行者是毫无灵活性的机器，它只能接受无二义性的形式符号所组成的程序正文，即形式语言。无论您的意思多么明确，只要最后传达到机器上的程序正文稍有差错，机器立即作出错误的解释。请记住："机器永远按您给它的信息（程序和数据）行事"。

程序正文一方面描述的是计算机将要执行的一步一步的动作；另一方面描述的是被计算的数据对象如何一步一步地变成解。在这个意义上，程序是对确定的计算任务、被处理的数据对象与处理规则的描述。程序设计就是按计算机能实施的处理规则完成问题的求解。

1.2 程序设计的特点

为了对程序设计(Programming)有清晰的理解,我们从宏观上考察程序设计的某些特点。

1.2.1 构造性

程序设计如同作家写文章,音乐家写乐谱,工程师绘制蓝图,计划工作者编制计划。它们共同的特点是,先于实施去"构造"程序。即把要求解的问题,通过语言媒介(计算机语言)构造出一个程序实体。这个程序实体付诸实施(即上机运行)就可体现解题的立意。

构造性使得编制程序时比较自由。只要能得出正确的解,怎么构造都行。因而决定了程序设计结果的多样性,即不同的人为解决同一问题编制的程序,其程序正文各不相同,然而,程序的功能却是等价的。

构造性决定了程序设计的工程性质。在桥梁工程上,很难说张工程师的焊接结构比李工程师的铆接结构一定要好多少,往往是在满足使用技术指标前提下各有优缺点。在不同环境条件下,评价优缺点的尺度又不同,例如,当焊接工匮乏,不能按时完工时,即使铆接结构造价高也是较好方案。因而,程序设计的工程性质决定了程序(或软件)难于建立统一的、定量的程序质量评价标准。

构造性还决定了程序在投入使用前要先作验证(Verification)或测试。正如工程师绘制的产品图纸要进行样机试制,作曲家的乐谱要彩排一样。它不像数学公式推导,只要证明每步推导是正确的,其结果立即可用。计算机科学家力图按数学方法推导程序,以减少验证和测试的费用。在这方面虽取得不少成果,但还很难达到实用阶段。

1.2.2 严谨性

原则上,要做好任何事情都必须准确无误,然而用词不当的文章、印错的乐谱、有错的图纸、欠周密的计划却比比皆是。这实质上是人们在处理复杂事物时不可避免的现象。为了准确,往往采用渐近修正的方式。在信息领域里,如果处理的环境可以接受,那么带有一定错误的信息就不认为是个问题。例如,我们很难找到没有口误的教师、没有印刷错误的讲义。但绝大多数情况下不影响教学,因为学生有判断力。计算机世界就不是这样。计算机只能接受准确无误的信息。通常情况下计算机几乎没有判断力,稍一疏忽不是没有结果就是结果有错。程序设计中,为一点微小的差错付出了极大的代价的事是经常发生的。

严谨性决定了程序设计不能用自然语言作为人-机通信的媒介。无论是汉语还是英语、法语,其语法都不是很严格的,它往往要取决于上下文,甚至语用环境才能排除多义性。例如,我们在报纸上看到这么一个小标题:

"马副总理十八日赴美谈纺织品贸易"

就是一个多义的标题。对于关心时事政治的人,它是明确的:中国目前没有姓马的副总理,能涉及纺织品贸易谈判的只能是东南亚的马来西亚,不会是非洲的马达加斯加或马里。对于其他的人只有看了下面的正文:

"新华社吉隆坡十七日电　马来西亚……"

才能理解标题的含义。计算机却没有这样的能力，目前多半使用的是上下文无关的形式语言系统。它无法补充缺损信息、去掉冗余信息、将暂时不懂的信息搁置起来，待下文或经过推理后予以理解和补充。我们只好严格遵守语法的规定来构造程序。

程序设计的问题总是用非形式化的自然语言陈述出来的。程序设计的任务就在于将问题的非形式描述转变为形式化的描述，最后以形式语言实现形式化的描述。

1.2.3　抽象性

客观世界的问题是极其多样的，然而基于 Von Neumann 原理的计算机只会传数、取数、比较、四则运算、逻辑运算等。程序设计就是把客观世界问题的求解过程映射为计算机的一组动作。用计算机能接受的形式符号记录我们的设计，然后运行实施。动作完成了，得出的数据往往也不是问题解的形式，而是解的映射。例如，在交通控制程序中用高级语言输出的红、绿、黄信号灯多半是 1,2,3 这样的数字符号。

程序设计用以表示设计的符号语言以及数据和解的表示都是着眼于实现的某种约定。不记住这些约定很难把解题的意图告诉机器；反过来，有了程序正文也很难读懂。机器语言时代的程序员从一大片 0 和 1 的组合符号中，要看出哪些组合代表哪些操作，哪些组合代表存储地址，哪些组合代表数据，否则无法作程序设计。这种及时区分组合符号，看出它抽象含义的能力，的确给程序员心灵造成巨大负担和创伤。计算机语言学利用抽象原理提供了日益脱离最低层实现的语言工具，我们今日使用的高级程序设计语言，面向解题过程（算法）去设计程序，不再面向机器去安排存储，也不直接利用操作码。这对于较小程序，其数据加工的流程是清晰的，也不难看出程序的逻辑含义；程序一大就不行了。基于过程实现的语言工具仍然不能清晰地展示程序逻辑。所以，阅读长篇的 BASIC、FORTRAN、COBOL、C、PASCAL 程序并不比文言文、甲骨文好懂。

面向问题程序设计语言的出现，使我们能把精力集中于问题的形式规格说明，我们只要形式地指明，为解这个问题要做什么就可以了。语言的软件系统自动地完成实现。这样，程序正文可清晰地展示程序逻辑，设计程序就很方便了。可惜，面向问题的语言在自动实现的效率上还不能满足一般应用要求。我们仍处在过程语言的后期。一方面我们可以面向对象、面向问题设计程序，一方面仍要人工补足抽象要求的实现，否则无法运行。至于 PASCAL、C 及其以前的语言仍然是面向过程的，用这些语言的程序员仍要完成问题解到过程实现的映射，要有很强的抽象能力。

1.2.4　叠加性

我们知道，今天能用高级语言编写程序，主要是因为计算机里有该语言的编译程序。如果没有系统程序支持，只能用机器语言编程序。多数初学者有这样的心理，他用 BASIC、FORTRAN、PASCAL 或 C 编程序，从数据定义开始，直到所有子程序、所有语句编完为止。总好像对可调用的机器里已有的过程或函数感觉不够完美，总不放心。造成这种心理与我们对程序开发环境、对程序设计不完全理解有关。事实上计算机应用不能永远从“燧木取火”开始。一定要利用已有的数据包、程序库、软件。正如为了吃顿面条，人们不必从种麦子开始，收割、磨面、擀面、切细到煮面一样，而多数人是去买切面或挂面，甚至买方便面或下饭馆。因而要学会尽可能少编程序且取得最大效益。正确利用已有程序还可以减少编程差错。

为了有利于叠加，自己设计的程序应尽可能分割为独立的小块（模块），功能明确而单一，以便增删或为别的程序所引用。

1.3 程序设计概念的范畴

提起程序设计，许多人就想到能以某种语言写出程序正文就算完了。事实上用计算机语言写出源程序仅仅是程序编码。为了能顺利地编码，程序的逻辑结构一定要能反映计算要求，计算要求必须建立在正确的计算模型上。因此，程序设计从定义问题就开始了。此外，编码之后要上机测试，如果程序不正确，则这个程序将毫无存在的意义。所以，程序设计概念范畴从定义问题开始，直到交出一个可用的程序为止。它包括：

- 定义问题　建立程序的规格说明。
- 设　计　建立程序系统结构及其算法实现，通常又分为总体设计与详细设计。
- 程序编码　构成计算机能接受的程序实体。
- 程序测试　检验程序的正确性。

如果是软件还应该交出各阶段的说明文件（文档）。例如，本程序适用范围、使用资源的说明、测试报告、用户使用手册等等。

近代的软件越来越大。人们为了对付日益复杂的程序设计问题，研究了许多不同的程序设计方法，各种不同的算法实现以及测试技术。这些内容将在随后的章节里介绍。

1.4 程序设计准则

对程序设计有了初步概念之后，就可以进一步考察程序设计的原理和方法了。但是原理和方法都要以我们追求的目标为前提。因此，本节研究一个好的程序（或软件）应该遵循哪些准则。

“怎样才算一个好程序?”，随着软硬件技术的发展，它的概念也是发展的。例如，20 年前一个简练的、占用存储空间小、运行快的程序就算好程序，今天看来就不一定了。如果它除了设计者能读懂、能修改以外，谁也看不懂，那不应该叫好程序。再如，20 年前 1 000 个高级语言语句的程序算大程序，今天看来它是一个一般的小程序。1984 年就已经出现了 4 000 万个语句的软件系统。

再者，程序设计的工程性质决定了我们难于定量地制定程序的质量标准。即使是在大型机上运行得很好的程序，拿到小型或微型机上，它的运行时间或许就难以容忍。因此，以下给出的准则是定性的。它们是：

- 正确性（Correctness）
- 可靠性（Reliability）
- 简明性（Simplicity）
- 有效性（Efficiency）
- 可维护性（Maintainability）

- 适应性(Flexibility)

对以上准则各种文献说法不一,但大同小异。为了说明它们的准确含义,分述如下。

1.4.1　正确性

正确性是判定程序质量永恒不变的准则。谁也不会去设计一个得不出正确结果的程序,这是显而易见的事。然而错误的程序比比皆是,甚至使用多年的软件还在陆续不断地发现程序错误。广泛使用的 Windows 操作系统也是如此。造成这种现象的原因是由于检验程序正确性的基本手段是测试(Testing)。正如 E. W. Dijkstra 指出的,"测试只能指出程序的错误,而不能证明程序没有错误"。

要想把一个不大的程序所有可能发生的值验证一遍,其次数即使是每秒千万次的机器几十年也做不完。然而舍此又无法验证它的完全正确性。

正确性的概念是:程序本身具备且仅具备程序"规格说明"中所列举的全部功能。因而验证这些功能的背景条件是否正确,不是正确性考核的范畴。

程序正确性的理论研究,一直是计算机理论界的一个重要课题。特别是 20 世纪 60 年代初,人们发觉"软件危机"以后,程序正确性证明更引人注目了。以后相继取得重大突破。到 70 年代末,一般的程序都可用证明的方法证明其逻辑的正确性。不幸的是,这些证明方法异常繁琐,把程序设计工作量增大了若干倍,而且复杂一点的程序还是无法证明的。因而,程序证明还很难达到实用的程度。但是这些研究是非常有意义的,特别地,表现在人们对程序的本质有了深刻的理解,并提出了"结构化程序设计"的完整思想。我们不能随便编一个程序,指望程序正确性证明理论会证明它是正确的。如果程序按"结构化程序设计"的要求去"构造",设计的程序是可以证明的。70 年代结构化程序设计风靡一时,它对程序设计方法学、程序设计语言产生了巨大的影响,这种影响沿袭至今。

1.4.2　可靠性

可靠性指的是"程序在多次反复使用过程中不失败的概率"。逻辑上正确的程序不一定可靠,可靠性一般对程序所在的计算机系统而言。如果发生一个临界值使程序遇到异常情况,程序若不能恢复到正常状态,则下次使用时就导致失败。核电厂、宇宙飞船的计算系统都需要绝对可靠。

可靠性没有度量的标准。多方检查、反复测试可以及早发现不可靠之处。然而可靠性只能在程序设计建立系统时考虑,不能到系统完成之后再补加上去。而实际情况恰恰相反,可能因某种原因将程序系统作了增删,增删部分往往使原有经过验证的部分变得不可靠。这种情况在利用早期语言(如 FORTRAN)时更为严重,因为这些语言的编译匹配检查能力很差。

一般说来,提高可靠性的方法是增加冗余度。可用多种不同的方法指明同一重要的特性。例如,计算中可能用到下列语句:

```
k = ......;
if ( k >= 10 && k < 20 )
    ......;
if ( k >= 20 && k < 30 )
    ......;
```

```
if ( k >= 30 && k < 50 )
    ......;
......;
```

当 k 值小于 10 或大于 50 时,程序没有明确规定如何执行。程序执行时将按从上到下原则去执行后面的语句。为了确保可靠,则增加冗余信息:

```
k = ......;
if ( k < 10 || k >= 50 )
    ......;
if ( k >= 10 && k < 20 )
    ......;
if ( k >= 20 && k < 30 )
    ......;
if ( k >= 30 && k < 50 )
    ......;
......;
```

这就防止了程序出现某种错误的情况的发生。

1.4.3 简明性

程序的抽象性质要求程序简明易读,这样人们才能读懂它。只有读懂了才能进行维护、修改。所以有些文献叫可理解性(Understandability),或可读性(Readability)。易读是可理解的前提。我们写一个程序只写一次,一旦投入使用,读程序何止几十次?所以尽可能写得简明一些,哪怕费点事也要这样做。

简明性和程序设计语言的表达能力有关,同时和程序设计风格有关。推行一种约定的风格,大家都按这种风格行事,则程序就易读好懂了。这里有一个非常典型的例子,它来自美国计算机科学家 D. E. Knuth 于 1968 年撰写的系列丛书《The art of programming》,我们不妨仔细读一读:

```
a = a - b;
b = a + b;
a = b - a;
```

其实,该程序段的功能与下面的程序段完全等价:

```
temp = a;
a = b;
b = temp;
```

哪一段程序更简单明了,相信大家一目了然。为什么出现这种情况呢?众所周知,早期的计算机速度低,存储容量小,评判一个程序的优劣,很自然地把运行效率放在首位。为了提高效率,众多的程序员竞相追求“技巧”,即使能在程序中减少一条语句,节省一个存储单元,也会觉得很高兴。即使在学术界,当时多数人也把程序设计当成是艺术而不是科学。有些学者认为,既然对同一给定的问题可以设计出多种不同的程序,其间的差异将主要取决于程序员的创造力和风格,就如同文艺创作中的情况一样。D. E. Knuth 不仅撰写了《The art of programming》一

书，在 1974 年荣获计算机领域的最高奖图灵奖时，还以“Computer Programming as An Art”为题发表过演讲。这都是上述观点的反映。

简明性不等于简单性。问题本来就很复杂，我们不可能使它简单。但程序结构清晰，编排得体，容易看懂还是能做到的。最重要的是不要人为地增加复杂性。例如，下面两个程序片段都是解决同一问题的。请分析一下第二段程序这样写到底有什么好处？

```
int  a[10][10];
for (i = 1; i < 10; i++ )
    for ( j = 1; j < 10; j++ )
        a[i][j] = (i/j) * (j/i);
```

也可以换一种写法：

```
int  a[10][10];
for ( i = 1; i < 10; i++ )
   {
    a[i][i] = 1;
    for ( j = 1; j < 10; j++ )
       if ( i ! = j )
          a[i][j] = 0;
   }
```

复杂性是一个更难于度量的概念。在算法设计的讨论中，复杂性专指时间和空间的复杂程度，即时空效率。这里说的复杂性是程序系统结构的复杂性。总的说来，计算机软件是在处理复杂性问题。近来对复杂性方面也提出了一些度量准则。如 MaCabe 和 Hastead 量度等等，但它们不是本书的内容，所以不作讨论。

1.4.4　有效性

程序实施就要占用一定的时间和空间资源。高效的程序占用空间很少、运行时间很短。这当然是我们所需要的。一般来说，时空效率总是人们做事情追求的目标。

对于应用程序，有效性不仅取决于硬件，还取决于软件环境（即支持程序运行的各种软件）。我们应在硬件、软件条件相同的情况下研究程序设计的有效性。

计算机发展的早期，计算机硬件设备价值昂贵，而且运算速度低，存储空间小，有效性成了程序设计的主要准则。有时为了争取小的存贮空间或简化几个运算步骤，把程序搞得晦涩难懂。在当时情况下这样做也是值得的，虽然使用了较多的人工去琢磨巧妙的程序，但从高档机转到低档机并少付机时费和人力的劳务费，这还是有利的。为了提高时间效率，数值分析这门学科研究出许多成功的快速算法，而且多年来为争取时空效率积累了一批优秀的程序。这些程序充分地利用了语言特征，算法巧妙、程序结构考究、运行快、空间利用合理，把程序设计技巧提高了一大步。至今其中的一些程序还可以作为程序设计训练的范例。

然而，当今计算速度已较二三十年前提高了几个数量级，内存增大了若干倍，外存容量几乎达到无限。常见的计算问题几乎不存在时间不能容忍和内存放不下的问题，特别是软件技术的进展和海量存储的出现，使存储空间问题已经全部转化为存取时间长短的问题。因而，空间效率更为次要。只有在特殊情况下才去考虑时间和空间效率，例如机载或弹载的计算机系

统要考虑空间问题,超大型科技计算题目、人工智能以及复杂的实时控制课题仍有时间效率问题。

1.4.5 可维护性

软件凝聚着人们的智力,开发一个大型应用软件系统代价十分昂贵。开发工作量以数百至数千人年计,相应的软件生存期长达 10 ~ 15 年,因此可维护性十分重要,它直接影响到维护费用。可维护性不好的软件甚至导致生存期过早结束。

软件的维护可分为:

- 校正性维护。这是开发的继续,在使用中经常可以发现原有程序系统的小错。好的软件在做这种维护时,不应引起难于控制的派生错误。
- 适应性维护。在较长的使用期内,可能因新的技术出现、使用要求有所变动,例如,老式设备退役,原控制系统软件要适应新设备的要求;又如新的计算机硬件代替原有硬件或软件要适应新的使用方式等等,都需要进行适应性维护。
- 完善性维护。多次使用同一个系统,自然会加深对该系统的认识。在不推翻原设计前提之下,进一步完善的办法不断出现,这是人们不安于陈规的劳动习惯所致,因而完善性维护是必然的。

软件的可维护性要求程序系统模块化和局部化。即某部分程序中有更改不影响到其他部分。即使有影响,其影响参数应置于显式的控制之下。

1.4.6 适应性

计算机硬件发展迅速,更新快,这自然要求软件系统有较好的适应性。例如,光盘代替磁盘,激光打印机代替机械的点阵式打印机等,要求更改后相应软件不作修改或仅作极少量修改。由于开发软件耗费的人力巨大,因而推广、移植较之重新开发所取得的社会效益大。在不同机型上移植软件,也要求软件有好的适应性。

计算机的普适性以及使用时的专用性,决定了机器的大、中、小、微,型号繁多,系统配置的可选项目也多,因而机器指令条数、系统软件的功能差异很大。无法统一也不必要统一。因此,软件的推广、移植天生就有很大的困难。

若软件开发时尽可能远离机器的特性,则适应性就可大为提高。高级语言软件就比汇编语言软件移植性好(后者的移植基本上是重写而不是修改!)。所以,近代软件的开发从使用汇编语言逐步走向高级语言。高级语言的标准化、规范化就有着更大的意义。高级语言提供了抽象的语义而掩蔽了其实现细节。

要使整个系统适应性好,则要尽可能提高程序的通用性。把不通用的部分尽可能集中在某一局部(模块)。例如,前述求单位矩阵的程序若写为:

```
#include  "stdio.h"
main()
   {
      float   a[100][100];
      int      n;
      scanf("%d", &n);
```