面向对象程序设计系列教材

面向对象程序设计高级教程

陈 奇

高等教育出版社

面向对象程序设计系列教材

面向对象程序设计高级教程

陈　奇

高等教育出版社

图书在版编目(CIP)数据

面向对象程序设计高级教程/陈奇. —北京：高等教育出版社,2001.7
高等学校本专科教材. 计算机及相关专业用
ISBN 7-04-007922-4

Ⅰ.面… Ⅱ.陈… Ⅲ.面向对象语言—程序设计—高等学校—教材 Ⅳ.TP312

中国版本图书馆 CIP 数据核字（2001）第 035046 号

面向对象程序设计高级教程
陈奇

出版发行	高等教育出版社		
社　　址	北京市东城区沙滩后街 55 号	**邮政编码**	100009
电　　话	010-64054588	**传　　真**	010-64014048
网　　址	http://www.hep.edu.cn http://www.hep.com.cn		
经　　销	新华书店北京发行所		
印　　刷	国防工业出版社印刷厂		
开　　本	787×1092　1/16	**版　　次**	2001 年 7 月第 1 版
印　　张	21.75	**印　　次**	2001 年 7 月第 1 次印刷
字　　数	530 000	**定　　价**	23.50 元

本书如有缺页、倒页、脱页等质量问题，请到所购图书销售部门联系调换。

前　言

怎样学好一种语言？怎样学会用正确的方法用这种语言编程？这恐怕是所有渴望成为一个优秀软件工程师的人都必须面对的问题。古人云：学而不思则罔，思而不学则怠。这是一个很好的经验教训，不过还应该再补充一点：实践。学好语言最重要的法宝是编程，再编程。

接触过不少同学，他们非常热爱计算机，非常渴望成为一个卓越的程序员，他们也很努力，但效果却不如人意，有些人甚至走了很长的弯路。自己当初也经历过种种曲折，很希望做点什么对这些同学有所助益。

本书是讲述面向对象程序设计方法与C++的，但我们不希望把它仅仅写成一种语言教材。学好语言不等于学好了编程，而不以编程为目的的语言学习也是不可能真正学好语言的。本书在内容取舍、教材结构及习题设置方面都力求体现这一点。

20世纪90年代初以前，我一直是用C语言写软件，用Pascal讲课。当时我们开了一门叫做高级程序设计的课程，力图告诉学生怎样才算是好程序，怎样才是一个高的编程境界。多年后，一些学生回忆起来，仍觉得获益良多。通过那门课的教学，我们（我和我的学生们）明白了一个道理，编程就像武侠小说里的练武，光是一招一式是不够的，要成为一个武林高手，就要掌握关键，就要追求那种境界。

当时，我们在课上读了不少非常优美的程序，也写了一些所谓的大作业（一千行开外，四、五千行以内），好像也挺优美的。但一到写实际的程序（一般在数万行以上），最终总是写得很蹩脚。当时用的是结构化编程，我自己的体会是很有道理的，但总是写不好。一方面得承认自己练得不够纯熟，一方面好像总觉得结构化程序设计本身是一套境界不够高的“拳”，坦率地说，我一直没有在用C语言编写大程序上体会到多少内在的乐趣。20世纪80年代末开始，我们接触到了面向对象程序设计，进而又接触了C++，当我们开始尝试用C++以面向对象方式编程时，各种条件都还比较差，国内教材几乎没有，当时流行的C++编译器都还只有C版本。但即使在那样的条件下，当在写一个C++的软件系统时，我第一次在复杂程序组织方面体会到一种令人惊奇的美妙感觉。很久以后，我才逐渐明白了面向对象思想巨大的魅力所在，这是一种令人振奋的体验。

面向对象真的那么好吗？国外有人说，C++的好处你得写五万行以上的程序才能体会到。这也许说得有些过，但确实，结构化程序设计和面向对象程序设计，都是程序越来越复杂的产物。在本书中，我们给出了一个较大的程序实例，展示了采用结构化的C与面向对象的C++不同的实现方案。但限于篇幅，我们仍没有机会接触真正复杂的程序，希望读者能多加一些想像去体会面向对象在复杂程序组织中的意义。

本书不打算写成一本手册，所以读者不会在其中查到C++的全部细节。我们认为目前主要的C++编程环境提供的联机帮助远比课本要好得多。本书的目的是想帮助读者掌握面向对象编程与C++的关键。以前接触过一些“精通”C++的学生，令人惊奇的是，他们对最关键、最重要的C++基本成分居然不得要领，甚至全然不知，而且更无法想像的是，这种情况还比较普遍，我们希望在“学”这一环节上，能给读者正确的引导。

本书的前两章首先回顾了结构化程序设计与C的要点，并突出了C的重要难点。尽管本书假设读者都是学过C的，但根据我们的经验，许多学过C，甚至用C写过不少程序的读者，对

C语言的一些基本方面仍然缺乏透彻的认识。C++是C的发展,C几乎是C++的一个真子集,C方面未能有透彻掌握的人是不可能真正掌握C++的。如果C语言与结构化编程都掌握得很好的读者可以跳过这两章。

第三章用较少的篇幅专门引入面向对象的基本认识,希望能使读者对面向对象有一个清楚的基本认识。

第四章介绍C++的基础部分,这样安排是想让读者不要一下子被C++丰富的细节搞糊涂,从而能够有重点地先把C++过一遍。这样做的另一个重要考虑是希望读者可以尽早进行比较全面的C++编程,而不是学了C++的一鳞半爪只是做一些"造句"式的编程。如果是课堂教学,一般来说这时就可以进入比较大的综合程序编写了。

第五章介绍20世纪90年代中期才成熟的先进的面向对象分析(OOA)与设计(OOD)体现UML,这是一个非常复杂的体系,本书未能详尽阐述,仅希望读者除了面向对象编程(OOP)外,建立初步的OOA、OOD的概念。教学安排上可以配合我们已经开始的综合程序的设计。

第六章讨论C++的一些高级问题,根据进度的编排,我们希望本章是随着读者逐渐深入运用C++而进行的。相信这样边用、边学、边思考的方式会有助于读者比较彻底地掌握C++语言与编程。

第七章介绍标准模板库(STL)。STL是1998年C++国际标准正式认可的,也是对C++多年发展中一个缺憾的弥补。与C的标准库函数类似,STL极大地增强了C++预先具有的能力。STL是一个庞大复杂的类库,一章的篇幅希望能使读者有个初步掌握,以便在编程实践中逐步运用。

本书是一般的程序设计教材,按理说,不应该讨论具体的某种编程环境,但从实践角度出发,任何大而复杂的程序一般都需要有良好的界面。C++在语言一级并未有界面的具体标准,选择一个目前业界最普遍的环境是一个合理的作法。除此以外,相信对Windows与MFC的初步掌握,有助于我们理解其他类似的系统。这里要强调的一点是,读者千万不要喧宾夺主,全面掌握MFC不是本书学习的目的。事实上,很多优秀的程序员非常希望写不依赖于某种环境的程序,在这种意义上,选择MFC或其他非标准类库,是由于C++没有提供标准界面接口无可奈何的结果。

本书的基本部分是我对讲授面向对象程序设计课程多年心得的总结,要感谢我的所有学生,本书是我们共同努力的结晶。本书也是我们在承担国家面向对象软件开发项目中技术与经验的积累。正是这些富有挑战性的软件开发,使我对面向对象编程与C++的精髓有了一定的理解。感谢师长们多年来给我的机会。

本书的具体编写还得到了我的同事与学生的大力支持与协助,他们是王硕苹、马明、阮志华和董颖涛。在此表示我衷心的感谢。

编写一本C++教材是我的宿愿,但由于我一直忙于其他工作,使本书的编写比较仓促。书中错误与疏漏之处在所难免,恳请读者指正。

编 者

2001年5月于老和山下

内 容 提 要

本书针对学过一门结构化语言的读者，以C++为背景语言，全面介绍了面向对象程序设计的一些概念和方法。主要内容包括：结构化程序设计，C语言回顾，面向对象程序设计基础，C++语言基础，面向对象的分析和设计，C++深入论题，标准模板库STL，Windows编程和MFC，一个综合实例研究。全书系统地介绍了C++的要点与关键，深入分析了C++中的难点，并对重要的实现机制进行了必要的讨论，力求帮助读者在全面掌握面向对象程序设计方法与C++编程精髓的基础上，进入一种良好的编程境界。

本书强调实例分析与概念理论相结合，通过一个贯穿全书的实例——图书馆管理系统，比较完整地展示了采用结构化编程语言C与面向对象编程语言C++的不同实现方案，具有较强的实用性。本书可作为高等学校计算机或相关专业的教材或参考书，也可供对计算机有较高要求专业的研究生使用。对于希望深入掌握面向对象程序设计方法与C++编程技术的读者，本书也不啻是一本较好的参考书。

责任编辑　倪文慧
封面设计　李卫青
责任印制　杨　明

目　录

第一章　结构化程序设计

程序设计主流已经从结构化程序设计过渡到了面向对象程序设计。有人错误地以为这两者是完全对立的，其实不然，面向对象程序无论从发展的历史上看还是从其内在的意义上看，都是结构化程序设计的发展。在一定意义上讲，可以认为面向对象程序设计更彻底地实现了结构化程序设计的理想。为了更好地讨论面向对象程序设计，我们有必要首先来回顾一下结构化程序。

与此同时，目前面向对象程序设计的主流语言C++又恰恰是从结构化程序设计的主流语言C发展过来的，为后面讲述的方便，我们也一并回顾一下C语言中的难点、要点或一些容易忽略而又很有意义的问题。已经熟悉这些内容的读者可以跳过本章。

1.1　软件和编程

计算机是20世纪人类伟大的成就之一。计算机在诞生以来的50多年中的发展速度是令人震惊的，不仅在于其迅速提高的速度和容量，更在于其应用的广阔性。计算机为什么会有如此超乎人们想像的今天呢？一个重要的奥秘就是“软件”。

什么是软件？认真一想一定会发现有些难以回答，软件是程序、工具还是一类机器？或者是机器中的一种组成部分？把软件看作是一类机器或机器部件是有道理的，但它和传统机器的最大区别就在于它的“软”，即容易改变。计算机的一个基本设计就是将“机器”分成了两部分：一部分是具有普遍性的支撑平台，人们抽象出了基本的逻辑结构和物理结构，即计算机的硬件；另一部分是可以随时根据用户的需要定制和改变的，即软件。制造和出售硬件的厂商在某种程度上不用关心他们制造的“机器”将会用来干什么，但在这种机器上附着的软件为它提供了无限的可能。软件这种非常好的“可塑性”和“可变性”——有点像橡皮泥——也使得计算机不仅可以被用于越来越广泛、几乎不受限制的领域，而且可以被不断地迅速改进。这一切应该是人类制造工具史上前所未有的。

什么是好的软件呢？更稳定、更快、功能更强大？这些都很重要，但最重要的是软件应该有很好的适应性，应该容易适应需求的变化，容易移植、改错及扩展。简单地说，软件最重要的特性是“柔性”，只要有柔性，一个不稳定的程序可以改稳定，一个不快的程序可以改快，一个功能不够的程序可以扩展功能。有生命力的软件一定不断有版本升级。

怎样又快又好的捏制这样的橡皮泥呢？这就是程序设计需要研究的内容了。计算机诞生的半个多世纪也是程序设计技术和艺术发展的半个多世纪。

软件可以划分成不同的层次，如图1.1所示。

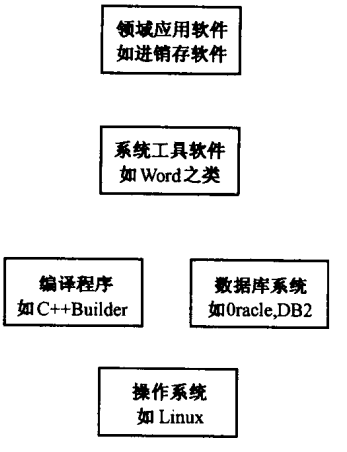


图1.1　软件的层次

不同层次的软件开发有很大的不同。操作系统层指操作系统以及和硬件密切相关的各种外挂的设备驱动程序，如 Linux、Windows 2000 等就属于这一层。由于数据库服务的特殊性，也可以将数据库系统单列一层，或者和 Web 服务、网络代理服务等一起列在操作系统附加的服务层。编译程序也是非常基本的系统软件，一般可以认为它与数据库系统处于同一层上。系统工具软件是指和特定用户应用关系不大的一类软件，如编译器、文档编辑器 WPS 2000 和 Word、浏览器 Netscape、图片显示工具 ACDSee、文件压缩工具 WinZip 等等。应用层的软件指和某类或某个特定的用户密切相关的、适用面相对比较窄的、针对特定应用需求的软件，如图书馆管理软件、飞机自动驾驶控制软件、气象预报软件、股票分析软件、企业 MIS 软件等等。

一般来说，越高层的开发中用的软件技术含量越低；越高层的软件越和用户的应用要求有关；越高层的软件越可以采用一些“预制件”；越高层的软件界面部分所占比越高，越可以多采用所谓可视化编程技术以提高开发效率。

这里必须对有志于软件开发的读者强调一点：软件开发应该指上面所有这些层次。软件专业知识越扎实，从软件技术上讲适应的开发范围越广。当然，对于应用层的软件开发往往还需要掌握相应的专业知识，专业知识掌握的程度将极大地影响软件的设计，但其中涉及的软件技术问题需要的仍然是软件专业的水平。

现在软件编程方面存在许多认识上的误区，举例如下：

误区之一：学完 X 语言(如 C)的程序设计课(或更正确地称之为程序设计基础)就会编程了。

其实大多数语言讲授的都是一些基本的词法、语法和简单的程序，打个比方就相当于小学学习的认字和语法，读一些短文甚至只是短句。一般课程里的上百行甚至是几十行的程序(当然是一种练习)，最多只相当于小学的造句，而目前实用软件动辄上万行，几十万甚至几百万、几千万行程序，那才是真正的“短篇小说”、“中篇小说”和“长篇小说”。

误区之二：对于数据库应用软件，采用最新的可视化编程技术和第四代语言开发，就不需要多少深入的计算机基础了。

对于一些规模小、要求低的系统也许是这样，但如果软件系统复杂到一定的程度，对系统的并发性、稳定性、鲁棒性要求又较高，没有扎实的软件基础是完全不可能胜任开发的。

误区之三：采用 C/C++ 这样的语言现在已不时兴了，真正的编程都是像 PB，Delphi，VB 等可视化工具。

实际上，前已有述，软件是多层次的，这些工具一般都是面向比较高层的，特别是基于数据库应用的软件开发的。从软件专业的角度讲，能胜任较低层次的软件开发是软件开发能力的基础。在开始学习编程的阶段，不应该把重点放在熟悉那些可视化编程工具等高层次的技巧上，而应该扎扎实实学好语言本身，学会直接采用这种语言“手工”编程部分的基本功。对于计算机专业的学生这一点尤其重要，因为即使有一天编程完全“自动”了①，那么这种“自动”的机器由谁来造呢？

1.2 结构化程序设计

结构化程序设计缘于 20 世纪 60 年代后的所谓“软件危机”，软件界的人士经过痛苦的经

① 自动程序生成一直是计算机界的梦想，一部分人也在为此努力。最近几年可视化编程的进展可以认为是这一努力的一部分。在欧洲，正在尝试一些不同的方案，但遗憾的是我们离彻底解决问题还很远(如果是可能达到的话)。

验终于认识到,不能再将程序看作是少数天才凭借灵感的"写作"过程,软件开发在更大程度上应该是一种工程。软件是许多人经过一个互相协作的比较漫长的过程而不断完善的相当复杂的产品。今天,软件界认识到,软件的开发实际上不仅仅是"程序员"完成的,和其他产品一样,它需要市场分析人员、管理人员、设计人员、编程人员、测试人员、销售人员、技术支持人员,甚至用户共同协作来完成。如果这个软件真有生命力,那我们就需要不断地改进它。

前已有述,在这一点上,软件的改进在频度、深度和广度方面都要超过一般的硬件产品。由于今天实用的软件往往规模庞大、结构复杂,即使是编程阶段也需要很多人的协作,另一方面,由于不断改进的循环开发过程、人员流动等因素,今天的开发人员就需要去理解昨天的程序,需要和昨天的开发人员"合作"。这样巨大的规模、非常的复杂性、人和人之间多方面的合作构成了今天软件开发的必须面对的基本问题。

结构化程序设计成了解决这些问题的一个基本方面。结构化程序设计就其思想基础而言,实际上基于20世纪伟大成就"三论"之一——系统论。事物都可以看作系统,系统是由子系统组成的,系统都有一定的目的(功能),系统是运动变化的等等。软件作为一种人造的复杂系统,自然可以以系统论的观点去分析、研究和设计。在20世纪60年代以前,程序编写都被看作一个写作过程,信马由缰是基本的编程方式,在程序中大量出现的goto语句成了这种方式的一个典型表现。在这种情况下,最终程序常常表现为一团乱麻就不足为奇了。

事实上,结构化程序设计不仅从系统论中找到思想的灵感,也从硬件的发展中找到了可以效仿的榜样。早期的电子产品都是要用分离元件的,在一块电路板上把线连来拉去,后果是,过一段时间就很难理解这团乱麻,一旦出现问题就更难确定问题何在。随着元件的逐渐增多(几十万、几百万甚至几千万晶体管),这一问题就更加突出。随着小规模集成电路、大规模集成电路乃至超大规模集成电路的诞生,分而治之成了一个自然的对策,更好的结构、模块化、封装使今天的硬件在更为复杂的同时可能反而更容易理解了。

结构化程序设计最早的代表性争论就是关于goto语句的,它可以(或应该)去掉吗?一个基本认识就是:goto语句要为一团乱麻式的程序负很大的责任。E.W.Dijkstra早在1965年一次学术会议上指出"程序的质量与程序中所包含的goto语句的数量成反比","可以从高级语言中取消goto语句"。当然,需要改进的远不止goto语句,甚至也远不止是在程序的代码编写(所谓coding)阶段。所以结构化程序设计实际上应该包括程序设计思想、程序设计方法、程序代码组织等等方面。伴随结构化程序设计同时兴起的是软件工程思想和技术,本书不是一本专门的软件工程教材,但也不可避免地会对这些软件工程的内容有所涉及。

1.3 层次树状的结构

结构化程序设计的观点是,整个程序结构应该是层次树状的。按系统论的思想,整个程序(系统)应该是由子程序(子系统)组成的,每一个子程序可以由更下一层的子程序组成。例如,一个简单的图书馆管理程序的层次树状结构图如图1.2所示。

这样的层次树状结构使整个程序有一个清楚的脉络,不再是一团乱麻。我们可以充分利用分而治之的策略思考、分析、编辑、调试整个程序。

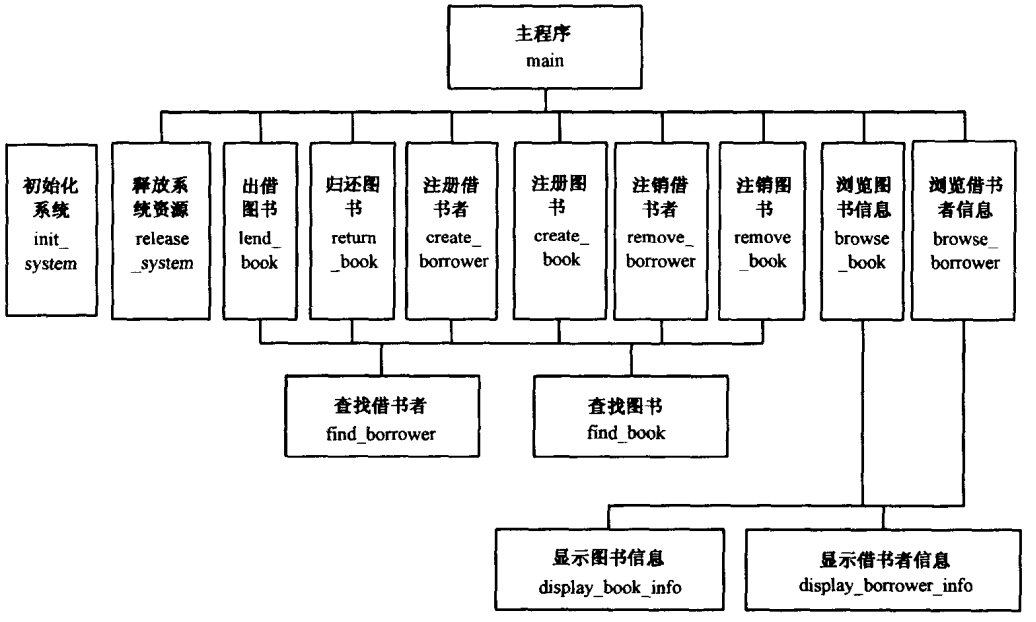


图 1.2　简单图书馆管理系统的层次树状结构图

1.4 模 块 化

结构化程序设计在总体结构上表现为层次树状的结构，而在局部组织上则表现为模块化。模块化思想的缘起是非常早的，最早人们发现使用早期 FORTRAN 语言编程中存在变量名混淆的现象，于是在 20 世纪 50 年代末，软件界提出了“分隔”(barrier)的概念，很容易理解，这实际上就是 Algol、Ada 和 Pascal 一类语言中的 begin/end，或 C/C++ 一类语言中的{}。其实这应是对程序的一种划分技术，是分而治之的实现手段。我们通过对程序的划分，将变量局部于这种划分内，程序块的相对独立性才成为可能。同时，现代的结构化程序设计语言尽管保留了 goto 语句，但一般都不允许 goto 语句转入和转出程序块，这也是使程序块成为相对独立的一种保证。

当然这种独立性是相对的，函数或过程一般说来相对独立性比较强，如果只是函数或过程中的程序块独立性就会弱一些。所以在结构化程序设计中，比较狭义上的模块应该是指函数或过程。而减少模块之间的耦合度是结构化程序设计的基本要求。

怎样才能减少函数之间的耦合度，实现程序的模块化呢？在形式上简单地说有两条：限制 goto 语句的使用和尽量不用全局变量。

关于 goto 语句的早期争论，极端的一派主张完全禁止 goto 语句的使用，对于这个主张的实现，关于“只要三种控制结构就具有完全的表达能力”的证明为其提供了支持①。理论上讲 goto 语句可以不用，但是否就应该完全不用呢？可以想像，这一点很难被老一代的程序员所接受，他们提出的理由，除了感情方面的原因之外，至少还是有一些地方是合理的。争论的结果是，

① 1966 年，Bohm 和 Jacopini 首先证明，只要三种控制结构，即：顺序、选择和循环结构就能够实现任何单入口、单出口的程序。也就是说，采用 goto 语句的程序理论上都可以只用上述 3 种控制结构实现，goto 语句是可取消的。

新一代的结构化编程语言(如:Pascal、Ada、C 等)仍然保留了 goto 语句,但在语法上加了若干避免引起混乱的限制。同时要求编程者只在非常有限的几种特殊情况下才可考虑使用 goto 语句。

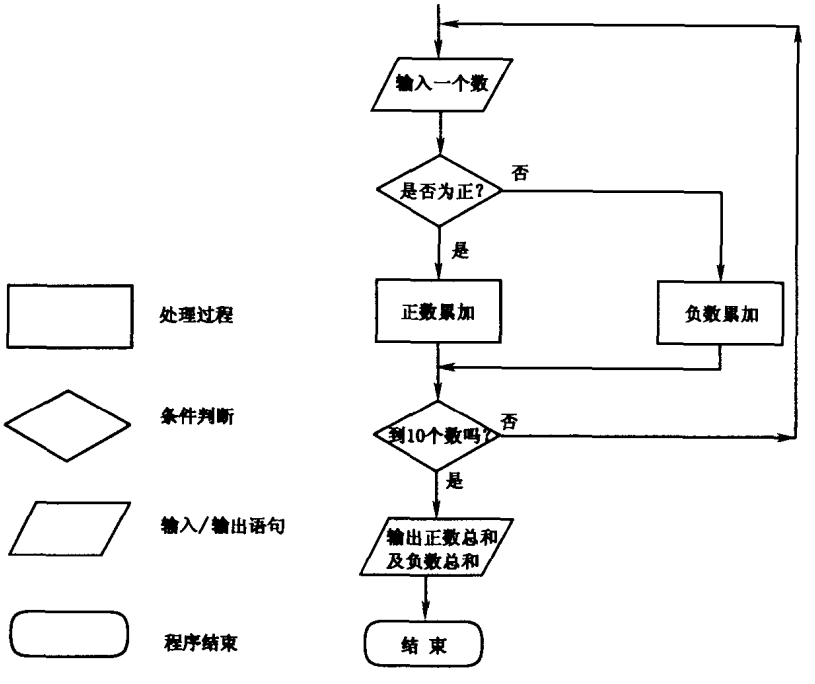
保证函数或过程模块化的一个基本要求是,尽量避免使用全局变量。也就是说,函数中出现的变量只能是局部变量或者是参数。全局变量的使用将会使函数之间潜在着错综复杂的联系,尽量避免使用全局变量意味着,一个函数所有要和外部交换的信息都必须通过参数或返回值进行。

上面的两点实际上是强调了模块化在形式上应该有的保证,但更重要的是在模块的设计和划分上的合理性,模块应该以什么为标准来组织呢? 我们可以从系统论中得到启发:系统都有一定的目的和功能,所以模块自然应该按功能来组织。为了保证模块之间较小的耦合度,模块内就应该有较高的聚合度(密切的内部联系),按功能划分显然是最合理的选择。

模块化的思想中实际上还包含有"封装"的思想,一个模块实际上是一个功能的封装体,它所用到的数据封装于模块的内部,它通过一些数据(参数和返回值)和外界联系。这些早期的封装思想在面向对象的程序设计中将会被极大地发展。

1.5 自顶向下方法

在结构化程序设计的观点看来,整个程序的结构是层次树状的,而它的开发方式则是自顶向下、逐步求精的,即所谓的 Top-Down 方式。自顶向下的设计方式实际上在很大程度上借鉴了其他工程设计的经验和方式,比较典型的是建筑设计的方法。在 20 世纪 70 年代软件工程



(a) 流程图使用的基本框　　(b) 流程图示例

图 1.3 标准的程序框图和示例

的初创期,相当多的软件工程学者都在建筑工程建造的方式中汲取灵感。一些传统的软件工程学者认为,理想的软件开发应该是这样的:由一些天才的软件设计人员完成高层和关键技术设计,然后再由一些具有非常丰富的经验和创造性的设计人员完成详细的下层设计,这一设计过程是自顶向下的,也就是说先完成上层的,再细化完成下层的,逐层往下。所有的这些设计并不直接编程,而是通过所谓的"蓝图语言"来完成,美国国家标准局制定的程序框图标准就可以算作是一种蓝图语言。标准的程序框图和示例如图 1.3 所示。

随着结构化程序设计的广泛使用,程序设计框图又暴露出这样的缺点,即图中完全无约束的箭头恰恰迎合了 goto 语句使用的要求。于是又产生了类语言的蓝图表示方案,如图 1.4 所示。

```
begin
   working : = true;
   initialize;
   while working do
   begin
      getacommand;
      if "valid command" then processcommand
   end // of while
end
```

图 1.4 一种类程序语言的蓝图表示示例

不管采用哪一种蓝图语言,早期相当多的软件工程学者认为,理想的状况下用蓝图语言描述清楚程序所有的细节后,剩下的工作只需一些能准确理解蓝图的技术人员,不折不扣地把蓝图转化为具体的源程序就可以了。20 世纪 70 年代以后有不少软件是以这样的方式生产出来的,这种方式比起 20 世纪 60 年代信马由缰的开发方式是一个巨大的进步。但事物总是不断向前发展的,随着软件开发技术的发展,人们也发现了这种类似建筑工程的软件开发方式的局限性(我们将在后续章节中进行讨论)。但需要强调的是,尽管有一定的局限性,并不意味着我们不应该去学习和掌握这种方法。

1.6 软件开发的瀑布模型

在结构化程序设计风行的时候,提出过多种软件开发方法与软件开发模型。其中最典型、影响最大的是"瀑布式生命期"(Waterfall Life Cycle)模型。瀑布模型如图 1.5 所示。

瀑布模型把软件系统的生产过程分为软件计划、需求分析、设计、编码、测试和维护 6 个阶段。其中:

(1)软件计划。在设计任务确定前,首先要进行调研和可行性分析,了解工作范围和所花代价,作出合理的计划。

(2)需求分析。具体调查分析用户要求,并用需求规格说明书表达出来。需求说明书中通常包括功能需求、性能需求、环境需求与限制等内容。

(3)软件设计。需求分析阶段解决做什么,软件设计阶段就是解决怎么做。该阶段还可再分为总体(概要)设计和详细设计。

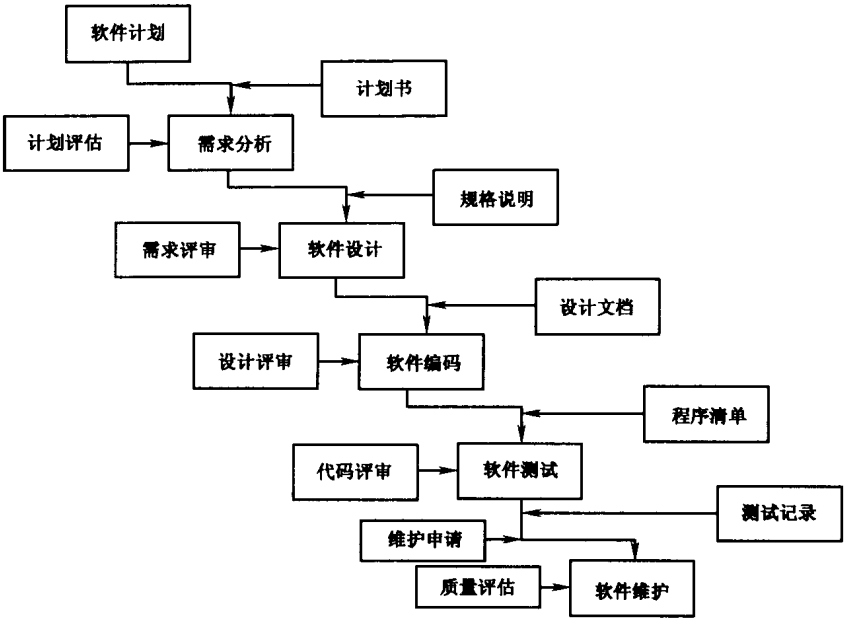


图 1.5 软件开发的瀑布模型

(4)软件编码。使用具体的软件开发环境和语言实现上述设计。

(5)软件测试。该阶段发现与排除软件中存在的错误,并解决性能、可靠性等方面的问题。测试通常分单元测试(模块测试)与综合系统测试。

(6)软件维护。对已交付的软件进行排错、修改和扩充。

通过对各种软件开发进行调查,我们对工作量得到如下统计:

软件计划与需求分析:20%,软件设计:20%,软件编码:20%,模块测试:20%,综合与系统测试:20%。

习 题 1

1. 为什么要提出结构化程序设计思想?
2. 用 C 语言写的程序一定是结构化的吗?
3. 我们打算为一个书店编写图书销售程序,请按照瀑布模型完成编码前的工作。
4. 你能否举一个例子说明,有时候完全消除程序的全局变量即使不是完全不可能的,也是非常困难的。
5. 你认为一个优秀的程序员应该具备哪些素质?

第二章　C 语言回顾

2.1　C 语言基础

众所周知，C 语言是应编写一个高度可移植的、开放的操作系统 UNIX 的需要而“顺便”创造出来的。严格说来，C 语言是一种中级语言①，具有许多不能称为高级语言的语言特性。但 C 语言在实践中却表现出了强大的生命力，并最终成为结构化程序设计的主流语言。

导致 C 语言成功的主要原因归纳如下：

(1)强大和灵活的表达能力。作为一个面向操作系统编写的语言，C 语言支持位、字节和指针的直接操作。它的类型概念也相当灵活，这些都是它的“中级语言”特性，但也是它强大能力的基础。

(2)很好的标准化程度和可移植性。在各种不同平台上的 C 语言之间保持了相当兼容的水平。多年来，UNIX 操作系统上配备的 C 语言一直被作为 C 语言的公认标准(见 Brian W. Kernighan 和 Dennis M. Ritchie 合著的《The C Programming Language》，1978 年 Prentice-Hall 出版)。尽管以后出现了一大批 C 语言编译器，但由于它们都有公认的效仿对象，所以仍然保持了惊人的兼容性。1987 年 12 月，美国国家标准化协会(ANSI)公布了 C 语言的新标准——87 ANSI C。

(3)广泛的用户群。Unix 从诞生之日起就成了美国各大学计算机系的首选教学系统，伴随其中的 C 语言在美国计算机专业群体中有最广泛的接收面。C 语言的成功本身又加强了这一趋势。

(4)语言简洁。与 C 语言强大的功能相比，C 语言的设计是相当简洁的，它仅有 32 个关键字(其中 27 个来源于 Kernighan 和 Ritchie 的公认标准，另 5 个是 ANSI 标准化委员会增加的)。

(5)紧凑高效。C 语言编译产生的代码效率一般要高于其他高级语言编译产生的代码，而且 C 语言的一些低级特性为编程者改善特定程序段的效率提供了可能。也许在今天这已不是最重要的特性，但对一些效率要求比较高的领域(如实时系统)仍有重要的意义。

本书假设读者已有 C 语言的学习基础，因此本章仅对 C 语言中的一些关键的、必须掌握的内容进行强调，完全没有学过 C 的读者，建议先阅读其他 C 语言教材。

2.1.1　编码

冯·诺依曼在设计计算机时有一个基本的想法：将数据和操作统一采用二进制表示，存放在统一的介质中。在这样的设计下，如果我们在计算机的内存上任取一个字节，比如值为 0x20，它可能代表如下内容：数字 32，一个编码为 0x20 的指令，一个指令中 0x20 所指定的硬件端口，一个 ASCII 编码中代表的空格符，一幅黑白照片中比较暗的一种灰，等等。具体的意义

① 语言的所谓高级、中级和低级并不是指它的功能强弱、先进落后。这里所谓的高和低是指离硬件的远和近。简言之，硬件对于编程者越透明即越高，当然越高级的语言也就越缺乏操作硬件的能力。