APPLIED C

应用C语言

中国科学院成都计算机应用研究所情报室

第一部分　设计应用系统

如果一个程序员仅仅具有在教室中学到的使用C语言的经验，那么一旦他投身入实际问题的C程序设计中，就会很快发现，课堂中学习的程序和专业人员编写的程序的重要区别就在于程序的大小。几乎没有例外，专业性的软件开发总是需要一个庞大的好些人参加的长时期计划。把最终形式的源代码写下来可能需要一摞纸。对于这种规模的应用问题，计划阶段所需做的事比起在编两页代码前先思考一下应当使用的算法要多得多。

那些已在其它语言中获得过专业经验的打算使用C的程序员，大抵都熟悉计划过 程并了解它的重要性。然而，他们也可能知道,在实践中,人们往往对关键性的第一步强调不够。结果是程序的规范说明具有二义性且不完整，用户接口太复杂且不一致，或者根本不适用。编出的代码也就很苯拙，难于阅读、调试和维护，开发周期中有重大的设计修改时情况更是如此。

对任何应用系统都要细心周密地计划，这一点至关重要。无论是程序员自己或他人编写的规范说明，都应当是清晰且无二义性的。计划实施伊始时，就要在计划中考虑到用户以及机器。在表述与审查设计思想时，使用模型常常比用文字好得多。在编写任何代码之前必须形成一个完整的（尽管不是定版的）兰图。

下面几章讨论程序的规范说明和用户接口设计，这是对一个应用系统作出计划时要做的两件最重要的事。这一部分包括论编程风格的一章，该章讨论了许多有关如何产生出清晰易读的程序代码的问题。

《应用C语言》

Bonnie Derman著，赖少庆译

目　　录

序　言

第一部分　设计应用系统

第二部分　选择数据类型

第三部分　代码组织

第四部分　为特定任务编码

第五部分 生成可执行代码

第一章　程序的规范说明

导　　论

为什么要写规范说明？

软件生产中一个最重要的但又是最易被忽视的问题就是规范说明。除开那些最简单的程序不论，对所有的程序而言，精心的设计总是可以减少错误，减轻修改和维护工作，避免产生全然不可理解的软件。另一方面，除了那些最复杂的及最形式化的程序之外，对于所有其它程序，这一计划过程往往又都差不多被忽视了。

只要你不是从最初的设想到最终的软件产品一直单枪匹马地干，你就总需要与他人交流思想，以便获得对你们的程序的一致认可的共同概念。凡是曾经在专业软件生产部门中编写过程序的人就会知道，这样的交流通讯往往会出现毛病：人们时时改变自己的初衷，在同一问题的看法上大相径庭，对程序的诸多方面疏于考虑，而且经常是缺少关于设计决定的纪录作为参考来评判是非。程序的规范说明（“spec”）就是用来缓解这些问题的。程序规范说明反映出对于程序的功能及局限性的精心规划设计。如规范说明写得好，它就可以清楚地描述出程序应当是什么样子。规范说明为程序的设计者、编程者、维护者，以及可能的使用者提供了一个共同的参照物。

本章内容组织

这一章讨论两个规范说明文档：需求文档和软件规范。这些文档满足两个不同（尽管是相关的）目的，并面向两类不同的对象。需求文档表明程序的目的，规定系统必须完成的那些功能，它是用来帮助用户的。软件规范是规定如何实现系统的第一个步骤。它把系统的每一段落规定为一个抽象软件部件，目的是为了帮助程序员。有许多系统都还没有复杂到要求这两种不同规范说明文档的程度。不过，尽管实际使用的文档可能是合二为一的，二者的目的却应当断然分开，因为它们代表了设计中的两个不同阶段，为了强调这些文档的不同作用，我们将在本章中分别地讨论它们。

需　求　文　档

生成一段软件的第一步是规定它应当做什么，这样的一个规范说明称为需求文档或者功能性说明。我们将使用前者以避免混淆，因为术语“功能说明”常常用来指规范说明中的另一个部分，关于它，本章以后将要讨论。

需求文档描述程序是做什么的，它通常用自然语言写成。它列举出程序应当满足的那些性质以及限制。必须把这类需求规定得足够清楚，以便能检查它们。例如，“容易使用”，就是一个不好的需求说明；而“用户将通过菜单选择发出一切命令”这样的说明，则是一个很好的需求例子。

需求文档的目的是帮助用户。虽然在很多实际场合中，在软件编成，调试完并投放市场之前，程序员是并不接触用户的，但是在设计程序的最初一些步骤中就产生一个面向用户的文档却具有很多优点。由于软件产生于用户的需要（而不是因为程序员有编程的能力），因此满足用户的需求是软件开发的第一准则。面向用户的方案使得你在考虑具体实施之前先考虑

他们的需求。对需求文档的审查使你能在编码之前查找并改正许多设计上的毛病以满足用户的需求。采用面向用户的，功能性的需求文档方案或许是对软件计划作整体性全面考虑的最好捷径。仅当你确定了程序将要做什么之后，你才可能确定程序应当怎样去做。

有位作者为需求文档规定了以下几条准则。

需求文档应当：

1. 只规定外部的系统行为，这就是说，应从用户观察系统的角度来描述它。

2. 规定出关于实现的限制，这就是说要描述出硬件施予的限制，例如速度，容量等约束条件。

3. 易于修改，这条准则意味着：使用表格或略图作为表达工具，比满纸的文字语句要好。

4. 应是系统维护人员的参照工具。它应当包括一个精心组织的切合实用的目录和索引。

5. 记录关于系统生命周期的展望，这就是说要反映出对未来系统可能作的修改的预测和计划。

6. 描述出对错误状态的可接受响应的特征，指明在什么情况下一个简单的错误消息就足够了，在什么情况下系统如何从错误状态中恢复过来。

根据这些准则，另一位作者为需求文档勾画出了一个轮廓，内容如图 1 —1 所示。

导　论

对系统的需求

文档其余部分的结构

所用的表示方法

硬　件

所用特殊硬件的说明，或者

系统可在其上运行的最小配置及随选设备。

概念模型

系统提供的主要服务，以及它们之间的关系。用图形表示最好。

功能需求

为用户提供的服务，这一段包括对用户接口性质提出的全部需求。为此，画出很有用的屏幕布置图象，它可以清楚准确地指明系统以什么样子呈现在用户面前。

数据库需求

系统所用数据的逻辑组织及这些数据之间的关系。

非功能需求

这是相对于功能性需求而言的，它指明系统运行应满足的约束条件。

有关维护的信息

系统得以建立的基本前提

可以预见的系统变化（硬件的替换，用户需求的变更等）

专用语汇

文档所用技术术语的定义索引

图 1 — 1

在需求文档中，信息应一次一点地逐款表述出来，以便于理解。不要采取报流水账的方式，最好把一件件准确定义的条款列成醒目的表格。如能附上图形说明就再好不过了。

需求文档是关于程序将要“做什么”而不是“如何做”的声明，尽管从某种程度讲，实现是与功能不可分离的，但在系统开发的初步阶段应尽可能把这两者区别开来以提供最大限度的灵活性。一上来就假定某种特定的实现会全无必要地限制你的设计。无论代码是由你自己编还是由其它人写，都要尽可能为程序员的创造性留有充分的用武之地。在需要对实现规定一些特殊要求的时候，请务必记住这一点。

概念模型

概念模型是系统的最高层次抽象。原则上说，这类模型可以用文字表达，也可用形式化的逻辑表达式描述，不过，采用图形方法来刻画往往更方便，更容易理解。通常，由箭头连接的带标号框块所构成的简单框图，就足以描述清楚系统的主要部件和这些部件之间的以及它们和用户之间的关系。图 1 — 2 画出了处理地址薄这一简单应用问题的概念模型。

如果必要，概念模型可进一步细化，即将其中的每一部件精细为一个概念模型。例如，你可以为图 1 — 2 中的“显示姓名和地址”部件设计出第二层概念模型。该模型可以包括“显示整个地址薄”部件、“显示单个姓名/地址的细节”部件以及诸如此类的其它部件。一般而言，系统越复杂，这种多层分解的模型方法越有用。

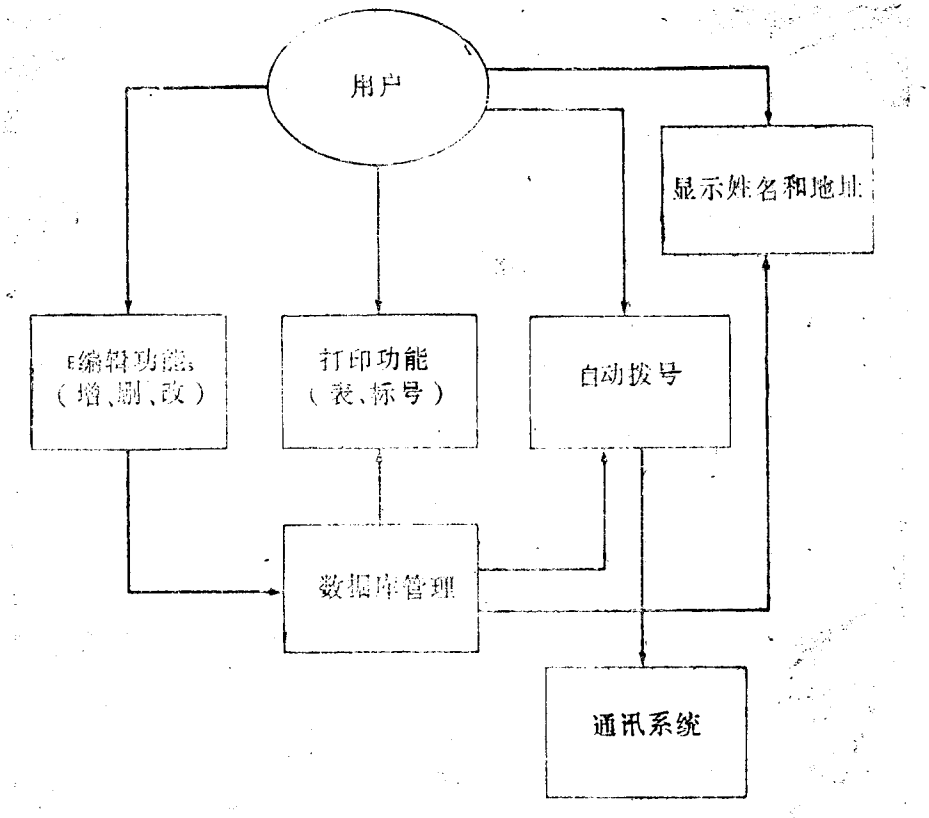

图 1 — 2

功能需求

功能需求部分指明系统为用户提供的每种服务。理想的情况是，这一需求应当是完整的

——它应该罗列出为用户提供的全部服务，应当是一致的——这些需求之间不能相互冲突。但在实践中，在软件开发的这一早期阶段，完整性和一致性均难达到。然而，这两者毕竟是我们追求的有用目标。

为避免用自然语言写规范说明时可能产生的许多问题，就不要用自由格式的分段落写法，而应把每一要点分别列出。要按功能组织需求文档。或许可以把概念模型中勾画出的每一部件列为一项，作为文档的各主要条款。不管是编写程序规范说明的哪个部分，都不要报流水账，这样才能获得清晰性。冗长累赘是产生二义性的源泉。图1—3以图表格式为地址薄（即图1—2中的模型）的“编辑功能”部件定义出功能需求。你可能会注意到，功能需求涉及到用户“选择”不同的随选功能。当然，用户挑选这些随选功能的方法也必须载入文档中，但为简洁起见，我们假定，早些时候已经在需求文档中写下了这些方法。

编辑功能

用户可从主显示屏中挑选如下编辑功能的任意一个

增加一项

删除一项

修改一项

Ⅰ、增加一项

程序显示一个空白模板，以及如下一些指示。（可能还有另一些指示）：

如所有信息均正确，则按ENTER键

如欲取消正要增加的项，按ESC键

A、如用户按ENTER键

1. 模板消逝
2. 新的信息增加到数据库中
3. 用户返回到显示屏
4. 在主显示屏上显示出新信息

B、如用户按ESC键

1. 模板消逝
2. 不在数据库中增加新信息
3. 用户返回主显示屏

Ⅱ、删除一项

程序提示用户选择要删除的项

A、如用户选择了一项：

1. 程序显示确认消息，指示用户按ENTER键作删除，或按ESC键取消删除动作。
2. 如用户按ENTER键

a 确认信息消逝

b 该项从数据库中清除

c 用户返回主显示屏

d 该项从主显示屏中清除

3. 如用户按ESC键

a 该项不从数据库中消除

b 用户返回主显示屏

B、如用户按ESC键

1. 提示信息消逝

2. 用户返回主显示屏

Ⅲ、修改一项

程序提示用户选择要修改的项

A、如用户选择了一个项

1. 程序显示包含这一项的信息的模板，并为用户给出如何修改该信息的指示。

2. 如用户按ENTER

a 模板消逝

b 修改了的信息增加到数据库中

c 用户返回主显示屏

d 在主显示屏上显示修改了的信息

3. 如用户按ESC键

a 模板消逝

b 不在数据库中增加修改了的信息

c 用户返回主显示屏

B、如户按ESC键

1. 提示消逝

2. 用户返回主显示屏

图 1—3

数据库需求

需求文档也应描述系统所必须使用的数据库。这一描述应包括所要用的数据项和这些项的逻辑关系。

对于这部分需求文档，表格常常是有用的。你可以为每一组相关的数据项构造出一张表。图 1—4 示明这样的一张表，它描述了在图 1—2 中给出模型的地址薄应用问题的数据

LName	FName	Street	City	State	Zip	Phone
Derman	Bonnie	1 Elm	Boston	MA	02115	(617) 926-2625
Farlow	Tim	2 Main	Boston	MA	02114	(617) 549-4829
Gregory	Keith	10Cross	Hartford	CT	06604	(203) 268-8243
Graneau	Justin	5Main	L.A.	CA	90016	(818) 238-4958
Pendzick	Rick	12Birch	Austin	TX	78766	(512) 234-4593
Shane	Art	5Stepney	Closter	NJ	07092	(201) 589-6843

图 1—4（注：表中提供的数据仅只是为说明问题用的）

非功能需求

系统的非功能需求是对程序性能的限制和约束，其中包括速度、存贮和操作系统诸因素，此外还有一些更细的考虑，例如可用的字符集、图形能力等。

非功能需求一般与硬件能力的关系特别密切。如果是为一个费时长久的应用项目作设

计，那么很可能，当项目完成时，硬件的能力将会有所扩充。非功能需求经常“预支”比当前可用的更大的硬件能力。如果你确实作了这种“预支”，就一定要明确地讲清楚你关于未来硬件能力所作的假定。

非功能需求常常体现为与功能需要的代价交换，其中最值得注意的是执行速度与存贮空间的代价交换。应当清楚地指明这些代价交换，以供实现阶段参考。或许可以把这些代价交换排列成矩阵形式。此矩阵的行上列出运行速度和可维护性等程序性质，列上写下容量极限等其它一些需求。这样，矩阵行列交叉处的元素就是释明相应的代价交换，以及互相冲突的需求和程序性质的注记。

需求验证

真正有用的需求文档应接受可能的用户或能起到可能的用户作用的其它人的验证。正如一个程序的β版本（见后文）应当接受调试和评估一样，需求文档也应接受评估，以证实它是否满足真正的用户需求。一个完整的需求文档不能有内部矛盾，应当描述出为用户提供的全部功能，在权衡用户需求和系统能力时应尽可能现实些。建立一个能清晰、完整、现实地定义出程序应当做什么的需求文档通常能大大减轻程序的修改和维护工作。

软 件 规 范

程序设计中的下一步是作软件规范，也称为设计规范。软件规范是为程序员写的。其作用是为程序实现提供初步设想。

软件规范是确定程序应如何完成任务的最初一步。软件规范为需求文档中规定的每项服务或每组服务，引入一个抽象软件部件。一般说来，编写软件规范是产生程序的函数和模块的第一步。比起需求文档，软件规范对系统的行为作了更精细的分解，它规定了每个程序过程所完成的动作以及这些动作的后果。

规范说明中的形式化方法

软件规范为程序员编写代码提供了指南。因此，它在建立程序必须遵从的条条准则时，必须尽可能地清晰，无二义性。不少软件工程师都发现，对于很大的系统，特别是对那些把软件规范当合同使用的系统，有必要使用形式化的规范说明语言。自然语言天生具有二义性，因此为了消除软件规范中的二义性，设计者们开发出了若干种形式化的方法。一般地说，这些形式化语言均为采用逻辑符号的高级结构化语言，有些类似于伪码。形式化的规范说明方法有两个主要优点：由于使用了形式语言，规范说明只能以一种方式解释，给出的定义因而更加严格。不完全性很容易隐藏于自然语言之中；而在形式化的结构中它就无藏身之地了。在那些特别强调系统应完全符合软件规范的场合，可以用形式化的规范说明作为测试软件的依据。

尽管现在我们讨论问题时不需要很高的形式化水准，但我们仍将介绍一些编写形式化规范说明的一般概念。提出形式化方法是为了鼓励你清晰地思考，以交流重要的软件设计问题。本节论述软件规范的过程中，所举例子将用到一定程度的形式化方法。一些更严格的形式化方法已被开发出来，但由于它们过于复杂，此处不拟采用。

软件规范的各个部分

软件规范常由两部分构成：描述系统约束条件的非功能规范，以及功能规范，后者描述了每个程序模块的作用，或许还要说明模块所用数据类型。我们将首先讨论软件规范的非功能部分，然后讨论功能规范。我们把数据类型的说明作为一个单独论题处理，放在第三段中

讨论。

非功能规范

软件规范经常包括一个对程序的非功能性质的说明：程序必须在何种约束条件下运行。软件规范中规定的约束条件与需求文档中规定的约束条件间的区别就在于阅读对象不同。软件规范是为程序员编写的。

功能规范

程序的功能描述是软件规范的主要构成部分。功能描述明确了每个模块对其环境的影响：它的输入，输出，及其对全局信息的影响。此处所说的“模块”是指在系统设计的这一阶段所能区分开来的任意过程。在最终的产品中，模块往往会被进一步划分，在规范中被确认为模块的某些过程可能会变成模块中的单个函数。

你可以通过分解概念模型来辨认出功能规范的各模块。至于作到什么详细程度，那就要根据应用问题的复杂性，根据你感觉到你所需要的特殊程度，由各人自己来作判断了。一旦你把你的概念模型细化到了最详细的层次，就可以把最底层的每一框当作一个模块。

模块的功能规范常常提供某些有关实现的指示，在一种极端情况下，这种规范就是为过程编写的伪码。而在另一个极端，功能规范把过程当作一个黑匣子，只描述它的外部影响。

如果必要，功能规范也可包括数据抽象。数据抽象定义了程序中使用的数据类型，办法是规定出可以在这些类型上施行的运算以及这些运算的性态行为。虽然也可以给出任何数据类型的数据抽象，但我们认为这样的形式化水准已超出了我们的需要。不过，如果你的程序要定义新的数据类型，利用数据抽象来刻画它倒是满有用的。

过程抽象

对每个过程所作的功能规范常被称为过程抽象，它至低限度应当对过程的输入、输出和效果作出规定。最详尽的过程抽象可以是为所说明过程编写的实用伪码。

I/O规范

根据黑匣子的观点，刻画一个过程只需要指明它的输入、输出和效果，因此这样给出的规范也可以叫做I/O规范。I/O规范指出，给定了输入约束条件，过程的输出就应满足输出约束。图1—5是一个函数的I/O规范，该函数返回其输入值的阶乘。

```
FACT ( X:integer ) —> integer
        input  0 < X < 8
        output FACT = X!
```

图1—5

图1—5中的规范描述了该过程为其环境施加的影响。图中第一行为该过程起了个名字（FACT），规定了输入类型（整数）和输出类型（整数）。第二行指出输入约束条件：输入必须大于0小于8（8！超出了许多机器所允许的最大整数范围）。第三行指出输出约束条件：输出必须是输入的阶乘。请注意，这里面不包含任何有关如何计算此阶乘的细节。

如果给此过程提供合法的输入，则这一I/O规范完全确定了该过程的效果。然而该规范还不能算完整的，因为它未能确定在给过程提供不合法输入时会产生什么后果。任何过程抽象都应当规定：在给它予任何可能的不合法输入时，过程所产生的影响。图1—6是上述过程的一个修正了的规范，它对错误情况也作了描述。

```
FACT ( X:integer ) —>integer
        N-state
          input  0 < X < 8
          output FACT = X!
        E-state
          input  X <= 0  or  X >= 8
          output ERROR CODE
```

图 1 — 6

图 1 — 6 中的I/O规范就完整了。第 1 行与图 1 — 5 中的第 1 行相同。第 2 行 指明，本行以后的描述是针对正常状态，也就是合法输入的。第 3 行和第 4 行 直 接 从图 1 — 5 中照搬。第 5 行指明一个错误状态，即输入不合法。在这个实例中，只描述了一个 E—state(错误状态)，因为只可能有一类错误。（我们假定，输入不满足输入约束（＞0 且＜8 ）时，返回一个ERROR-CODE（错误码）。

对于那些可能会有多类输入和输出的过程，其I/O规范可能变得与 过程的实际代码一样大、一样复杂。因此I/O规范一般只对简单的过程有用。对于较复杂的模块，操 作规范更易写易懂。

操作规范

I/O规范代表了认识过程的最抽象的观点。走另一极端的过程抽象则可以由执行 过程的伪码构成。这种接近于实现的规范称为操作规范。操作规范与实际代码的不同之处在于，它是为谋求清晰性而不是为追求效率设计的。在某种程度上说，它定义了实现，只不过它通常远不及实际代码具体。图 1 — 7 展示了图 1 — 5 和图 1 — 6 中说明的阶乘函数的操作规范。

```
FACT ( X:integer ) —> integer
if X <= 0 return ERROR CODE
if X >= 8 return ERROR CODE
if X <= 2 return X
return ( X * FACT(X-1))
```

图 1 — 7

图 1 — 7 表明了在一个过程抽象中访问过程的用法。在本例中，这种访问是递归的，不过，一个过程抽象也可以访问不同的函数。当然，任何被访问的函数也必须有自己的过程规范。利用这种访问可以把过程抽象写得更简洁，使其既易写又易懂。

图 1 — 7 中的操作规范隐含指定了一种特殊的实现。另外，还可以利用迭代法实现上述过程，此时，其操作规范如图 1 — 8 所示。

```
FACT ( X:integer ) —> integer
if X <= 0  return ERROR CODE
if X >= 8  return ERROR CODE
let FACT = 1
for all i, 2 <= i <= X do
    FACT = FACT * i
```

图 1 — 8

图1—7和1—8中的操作规范是有实用价值的合法代码，它们各为程序员限定了一种特殊的实现。对这个过程而言，我们宁肯要I/O规范，有它也足够了：它提供了所有必要信息而复杂性又最低，同时，也没有对程序员作不必要的限制。不过，对于更复杂的过程，I/O规范则常常是累赘的，用操作规范却能更清晰地定义过程。

混合型规范

当你处理现实世界中的大量应用问题时就会发现，某些地方宜用I/O规范，某些地方宜用操作规范，甚至还有些场合，用自然语言来描述最合适。在编写程序规范说明时，混合使用这几种方法是最合理最有效的办法。我们一开始就说过，程序规范的最基本目标是清晰性。因此，能为程序的任一部分提供最清楚形象的方法就是描述该程序部分的正确方法。

数据抽象

对于程序定义的新数据类型，在你的软件规范中纳入一个描述它的数据抽象，是会有益的。数据抽象通过指明数据的范围和在数据上执行的操作的性状行为来定义一个数据类型。它既避免在实现这一数据类型时把程序员限制得太死，又能完整地描述出该数据类型所必备的属性。图1—9给出了一个整数队列的数据抽象。

```
queue of I
    Interface
        create -> queue of I
        put ( queue of I, I) - >queue of I
        get ( queue of I ) -> I
        is_empty ( pueue of I ) -> boolean
        remove ( queue of I ) -> queue of I
    Axioms
        1.  is_empty ( create ) = true
        2.  is_empty ( put(q,v))= false
        3.  get ( create ) = ERROR
        4.  get ( put(q,x))= if is_empty ( q ) then x else get(q)
        5.  remove(create)= ERROR
        6.  remove ( put(q,x))= if is_empty (q) then create else put
                                ( remove(q), x)
```

图1—9

在图1—9中，接口段列出了每个函数及其输入输出。而公理段则定义了每个函数的行为。

公理1和2定义了is—empty操作，公理1指出，新产生的队列总是为空，公理2指出，由put操作产生的队列非空。

公理3声明，试图从空队列中获取一个项是错误的。

公理4递归地指明get函数返回最先加入到队列中去的项。

公理5和公理6定义移去操作，它给出把输入队列的第1项移去后留下的队列。（即remove函数是get函数的“剩余”）。公理5指出对一空队列施行移去操作是错误的。公理6与公理4类似，它递归地说明，被移去的项就是最先加入队列中的项。

图1—9中的规范完整地无二义性地规定了队列是什么东西，如何去操作它，但却丝毫

未涉及如何实现它的问题。

结　　论

编写程序规范说明是软件设计中最早的、也常常是最易遭到忽视的阶段。本章提出了一些编写程序规范说明的理由与建议。

除了最简单的例子，在所有的应用问题中，精心的计划都能降低代价，减少错误，减轻修改和维护工作。程序规范说明这种系统化方法就是用来为应用系统作计划，用来交流应用问题的目标和方法的。

在编写规范说明时应仔细思考程序的功能和局限性。单是这种思维过程就能减少出错及修改的机会。高质量的规范说明能清楚明瞭地刻画出程序的面貌，为设计师，程序员，维护人员和可能的用户提供一个参照基础。

程序规范说明可分为两大构成部分：需求文档和软件规范。需求文档是为用户写的，它描述出程序将要作的事，它尽可能不涉及程序的内部实现，只是描述那些用户要看到的东西。另一方面，软件规范则是为程序员所用的。它指明软件应满足的约束条件，为系统的每一部件附上一个设计准绳。软件规范的功能性部分至少要说清楚每个过程的输入、输出和全局性影响；最详细的规范能指明实现过程的方法。

书写程序规范说明的详细程序不一，所用方法的形式化水平也不一。究竟要用多少形式化方法，要写到哪种详细程序，取决于应用问题的性质以及建立此应用系统的过程中产生出的通讯交流的需要。

第二章　设计用户接口

导　　论

软件开发是建筑在一门科学上的艺术。这门科学就是对控制结构和数据结构的理解，而这些简单或复杂的控制结构则是由计算机的工作运转生发出来的。但是这门科学又尚未（或许永远不会）发展到这样的地步，以至于任何程序设计任务，除了最简单的以外，均有一个显而易见的最佳的现成解决。正是在这种背景下，这门艺术得以诞生。

在程序设计中，没有一件事比设计用户接口更成其为一门艺术。全部软件的目的就是处置数据或设备，而用户接口的目的却是与一个独特的最难捉摸的“设备”：人，打交道。在用户接口中发生的数据处置不外乎就是接受键入字符和在屏幕上显示字符串这些事。编写这些例程并不困难，难就难在决定如何使用这些程序：决定接受哪个按键、把哪个串显示到屏幕上，何时显示，等等。

因此，本章将不讨论这些例程的实现。实现问题，特别是商品化工具的用户接口实现问题，将在第9章《接受用户输入》中讨论。本章只对用户接口的难点：设计问题，作一定性讨论。

人们在着手设计用户接口时会发现，软件设计的一般方法和工具已不敷使用。这一点毫不奇怪。结构程序设计法并未对发生在计算机外的事发过议论。我们总是假定程序有适当的输入，并且得到了合理的输出。有了这些也就认为要做的事完成了。然而，任何程序都仅仅是一个更大系统的一部分：程序的输入必得从某个地方来且输出到某个地方去。获取输入并提供输出就正是用户接口的功能。

目前，还不存在普遍接受的专门用于生成用户接口的过程，也没有保证能适用于每个程序的用户接口设计的严格规则。确切地说，这一领域中的知识和经验还只是体现为一大堆技术。正如其它任何技术一样，这些技术中能表述为切合实用的严格法则的东西很少。不过，也还是可能寻求某些在任何场合下均必须优先考虑的技术：对中介的认识。深刻地理解了用户接口为何物，就能更好地设计（并实现）它。而达到这种理解就正是本章的目的。

什么是用户接口

“用户接口”不是指程序的一个部份，而是指对程序的视域，也就是作为用户的人对程序的视域。大多数程序确实有特殊的例程直接负责处理输入和输出，但由此却往往导致一种误解，认为这些例程就构成用户接口。用户接口产生于特殊的输入和输出例程，也产生自程序的整体组织中。一个简单的例子可以阐明这一观点。

考虑一个称为DELPHI的假想数据检索程序。DELPHI接受用户的问题，分析它并编排一个响应屏幕，然后显示这一响应屏幕。对于当前的讨论，只需要程序操作的三个细节：

1．有时候，编排数据的过程需要相当长的时间。

2．键盘输入是由操作系统接受到的，该输入存于缓冲器中，当需要时传送给程序。

3．用户按ESC键退出程序。

我们考察DELPHI的两种不同的逻辑结构。在第一种方案中（见图2—1），一个最高层的循环包含4个例程调用。