目　录

第 1 章　绪论 …… (1)
1.1　问题求解 …… (1)
1.2　算法表达中的抽象机制 …… (1)
1.3　抽象数据类型 …… (4)
1.3.1　抽象数据类型的基本概念 …… (4)
1.3.2　使用抽象数据类型的好处 …… (6)
1.3.3　数据结构、数据类型和抽象数据类型 …… (6)
1.4　用 C++ 描述数据结构与算法 …… (7)
1.4.1　变量、指针和引用 …… (7)
1.4.2　函数与参数传递 …… (8)
1.4.3　C++ 的类 …… (9)
1.4.4　类的对象 …… (10)
1.4.5　构造函数与析构函数 …… (11)
1.4.6　运算符重载 …… (11)
1.4.7　友元函数 …… (11)
1.4.8　内联函数 …… (12)
1.4.9　结构 …… (12)
1.4.10　联合 …… (12)
1.4.11　异常 …… (12)
1.4.12　模板 …… (13)
1.4.13　动态存储分配 …… (13)
1.5　算法复杂性分析 …… (15)
1.5.1　算法与程序 …… (15)
1.5.2　算法复杂性的概念 …… (15)
1.5.3　算法复杂性的渐近性态 …… (16)
习题一 …… (18)
第 2 章　表 …… (20)
2.1　ADT 表 …… (20)
2.2　用数组实现表 …… (21)
2.3　用指针实现表 …… (25)
2.4　用间接寻址方法实现表 …… (31)
2.5　用游标实现表 …… (34)
2.6　循环链表 …… (41)
2.7　双链表 …… (45)
习题二 …… (50)
第 3 章　栈 …… (54)
3.1　ADT 栈 …… (54)

3.2 用数组实现栈 …… (55)
3.3 用指针实现栈 …… (58)
3.4 等价类划分问题 …… (61)
习题三 …… (63)
第4章 队列 …… (66)
4.1 ADT 队列 …… (66)
4.2 用指针实现队列 …… (66)
4.3 用循环数组实现队列 …… (69)
4.4 电路布线问题 …… (74)
习题四 …… (77)
第5章 串 …… (79)
5.1 ADT 串 …… (79)
5.2 用数组实现串 …… (80)
5.3 用指针实现串 …… (85)
5.4 串的块链表示法 …… (86)
5.5 串的堆结构 …… (86)
5.6 模式匹配 …… (86)
5.6.1 朴素的模式匹配算法 …… (86)
5.6.2 模式匹配的 KMP 算法 …… (88)
习题五 …… (93)
第6章 排序与选择 …… (95)
6.1 简单排序算法 …… (95)
6.1.1 冒泡排序 …… (95)
6.1.2 插入排序 …… (96)
6.1.3 选择排序 …… (97)
6.1.4 简单排序算法的计算复杂性 …… (97)
6.2 快速排序算法 …… (98)
6.2.1 算法基本思想及实现 …… (98)
6.2.2 算法的性能 …… (100)
6.2.3 随机快速排序算法 …… (100)
6.3 合并排序算法 …… (101)
6.3.1 算法基本思想及实现 …… (101)
6.3.2 消除递归 …… (101)
6.3.3 自然合并排序 …… (103)
6.4 线性时间排序算法 …… (103)
6.4.1 计数排序 …… (103)
6.4.2 桶排序 …… (105)
6.5 中位数与第 k 小元素 …… (106)
6.5.1 平均情况下的线性时间选择算法 …… (106)
6.5.2 最坏情况下的线性时间选择算法 …… (107)
习题六 …… (109)
第7章 树 …… (111)
7.1 树的定义 …… (111)

7.2 树的遍历 …… (113)
7.3 树的表示法 …… (115)
7.3.1 父结点数组表示法 …… (115)
7.3.2 儿子链表表示法 …… (116)
7.3.3 左儿子右兄弟表示法 …… (116)
7.4 二叉树 …… (117)
7.5 ADT 二叉树 …… (119)
7.6 二叉树的实现 …… (120)
7.6.1 二叉树的顺序存储结构 …… (120)
7.6.2 二叉树的结点度表示法 …… (121)
7.6.3 用指针实现二叉树 …… (122)
7.7 线索二叉树 …… (126)
习题七 …… (128)
第 8 章 图 …… (131)
8.1 图的基本概念 …… (131)
8.2 抽象数据类型 ADT 图 …… (134)
8.3 图的表示法 …… (135)
8.3.1 邻接矩阵表示法 …… (135)
8.3.2 邻接表表示法 …… (136)
8.3.3 紧缩邻接表 …… (136)
8.4 用邻接矩阵实现图 …… (137)
8.4.1 用邻接矩阵实现赋权有向图 …… (137)
8.4.2 用邻接矩阵实现赋权无向图 …… (140)
8.4.3 用邻接矩阵实现有向图 …… (141)
8.4.4 用邻接矩阵实现无向图 …… (141)
8.5 用邻接表实现图 …… (141)
8.5.1 邻接表基类 …… (141)
8.5.2 用邻接表实现有向图 …… (143)
8.5.3 用邻接表实现无向图 …… (144)
8.5.4 用邻接表实现赋权有向图 …… (145)
8.5.5 用邻接表实现赋权无向图 …… (148)
8.6 图的遍历 …… (149)
8.6.1 图的搜索游标 …… (150)
8.6.2 广度优先搜索 …… (152)
8.6.3 深度优先搜索 …… (154)
8.7 最短路径 …… (155)
8.7.1 单源最短路径 …… (155)
8.7.2 所有顶点对之间的最短路径 …… (159)
8.8 最小生成树 …… (160)
8.8.1 最小生成树性质 …… (160)
8.8.2 Prim 算法 …… (161)
8.8.3 Kruskal 算法 …… (163)
8.9 图匹配 …… (166)
习题八 …… (168)

第 9 章　集合 …… (170)
9.1　以集合为基础的抽象数据类型 …… (170)
9.1.1　集合的定义和记号 …… (170)
9.1.2　定义在集合上的基本运算 …… (171)
9.1.3　集合的简单表示法 …… (171)
9.2　字典 …… (179)
9.2.1　实现字典的简单方法 …… (179)
9.2.2　用散列表实现字典 …… (180)
9.3　有序字典 …… (189)
9.3.1　有序字典的定义 …… (189)
9.3.2　用数组实现有序字典 …… (189)
9.3.3　用二叉搜索树实现有序字典 …… (190)
9.3.4　AVL 树 …… (199)
9.3.5　红黑树 …… (209)
9.4　优先队列 …… (218)
9.4.1　优先队列的定义 …… (218)
9.4.2　用字典实现优先队列 …… (219)
9.4.3　优先级树和堆 …… (219)
9.4.4　用数组实现堆 …… (221)
9.4.5　可并优先队列 …… (224)
9.5　并查集 …… (229)
9.5.1　并查集的定义及其简单实现 …… (229)
9.5.2　用父亲数组实现并查集 …… (230)
习题九 …… (233)
第 10 章　算法设计策略 …… (236)
10.1　递归与分治策略 …… (236)
10.1.1　递归的概念 …… (236)
10.1.2　分治法的基本思想 …… (238)
10.1.3　二分搜索技术 …… (239)
10.1.4　棋盘覆盖问题 …… (240)
10.2　动态规划 …… (242)
10.2.1　矩阵连乘问题 …… (243)
10.2.2　动态规划算法的基本要素 …… (247)
10.2.3　最大子段和问题 …… (250)
10.3　贪心算法 …… (256)
10.3.1　活动安排问题 …… (256)
10.3.2　贪心算法的基本要素 …… (258)
10.3.3　哈夫曼编码算法 …… (261)
10.4　回溯法 …… (265)
10.4.1　回溯法的算法框架 …… (265)
10.4.2　符号三角形问题 …… (270)
10.4.3　圆排列问题 …… (273)
10.4.4　连续邮资问题 …… (275)
10.4.5　回溯法的效率分析 …… (277)

10.5 分支限界法 …… (280)
10.5.1 分支限界法的基本思想 …… (280)
10.5.2 装载问题 …… (282)
10.5.3 批处理作业调度问题 …… (290)
习题十 …… (295)
参考文献 …… (299)

第1章 绪 论

1.1 问题求解

用计算机解决一个稍复杂的实际问题，大体都要经历如下的步骤：

（1）将实际问题数学化，即把实际问题抽象为一个带有一般性的数学问题。这一步要引入一些数学概念，精确地阐述数学问题，弄清问题的已知条件和所要求的结果，以及在已知条件和所要求的结果之间存在着的隐式或显式的联系。

（2）对于确定的数学问题，设计求其解的方法，即所谓的算法设计。这一步要建立问题的求解模型，即确定问题的数据模型并在此模型上定义一组运算，然后借助于对这组运算的执行和控制，从已知数据出发导向所要求的结果，形成算法并用自然语言来表述。这种语言不是程序设计语言，不能被计算机所接受。

（3）用计算机上的一种程序设计语言来表达已设计好的算法。换句话说，将非形式自然语言表达的算法转变为用一种程序设计语言表达的算法。这一步称为程序设计或程序编制。

（4）在计算机上编辑、调试和测试编制好的程序，直到输出所要求的结果。

上述问题求解的过程中，求解问题的算法及其实现是所关心的核心内容。本章着重考虑第（3）步，而且把注意力集中在算法表达的抽象机制上，目的是引入一个重要的概念——抽象数据类型；同时，为大型程序设计提供一种相应的自顶向下逐步求精的模块化方法，即运用抽象数据类型来描述程序的方法。

1.2 算法表达中的抽象机制

算法是一个运算序列。这个运算序列中的所有运算定义在一类特定的数据模型上，并以解决一类特定问题为目标。这个运算序列应该具备下列4个特征：

（1）有限性，即序列的项数有限，且每一项运算都在有限时间内完成；

（2）确定性，即序列的每一项运算都有明确的定义，无二义性；

（3）可以没有输入运算项，但一定要有输出运算项；

（4）可行性，即对于任意给定的合法的输入都能得到相应的正确的输出。

这些特征可以用来判别一个确定的运算序列是否称得上是一个算法。

算法的程序表达，归根到底是算法要素的程序表达，因为一旦算法的每一项要素都用程序清楚地表达，整个算法的程序表达也就不成问题。

很明显，算法有如下3要素：

（1）作为运算序列中各种运算的运算对象和运算结果的数据；

（2）运算序列中的各种运算；

（3）运算序列中的控制转移。

这3要素依序分别简称为数据、运算和控制。

由于算法层出不穷，变化万千，各种运算所作用的对象数据和所得到的结果数据名目繁多，不胜枚举。最简单最基本的有布尔值数据、字符数据、整数和实数数据等；稍复杂的有向量、矩阵、记录等数据；更复杂的有集合、树和图，还有声音、图形、图像等数据。

同样，由于算法层出不穷，变化万千，各种运算的种类五花八门、多姿多彩。最基本最初等的有赋值运算、算术运算、逻辑运算和关系运算等；稍复杂的有算术表达式和逻辑表达式等；更复杂的有函数值计算、向量运算、矩阵运算、集合运算，以及表、栈、队列、树和图上的运算等；此外，还可能有以上列举运算的复合和嵌套。

控制转移相对单纯。在串行计算中，它只有顺序、分支、循环、递归和无条件转移等几种。

最早的程序设计语言是机器语言，即具体的计算机上的一个指令集。当时，要在计算机上运行的所有算法都必须直接用机器语言来表达，计算机才能接受。算法的运算序列包括运算对象和运算结果都必须转换为指令序列，且每一条指令都以编码（指令码和地址码）的形式出现。这与用高级程序设计语言表达的算法相差甚远。对于没受过程序设计专门训练的人来说，一份程序恰似一份“天书”，让人看了不知所云，可读性极差。用机器语言表达算法的运算、数据和控制十分繁杂琐碎，因为机器语言所提供的指令太初等、原始。机器语言只接受算术运算、按位逻辑运算和数的大小比较运算等。对于稍复杂的运算，都必须一一分解，直到到达最初等的运算才能用相应的指令替代之。机器语言能直接表达的数据只有最原始位、字节和字 3 种。算法中即使是最简单的数据如布尔值、字符、整数和实数，也必须一一地映射到位、字节和字中，还得一一分配它们的存储。对于算法中有结构的数据的表达则要麻烦得多。机器语言所提供的控制转移指令也只有无条件转移、条件转移、进入子程序和从子程序返回等最基本的几种。用它们来构造循环、形成分支、调用函数和过程都得事先做许多准备，还需要许多经验和技巧。

直接用机器语言表达算法有许多缺点：

（1）大量繁杂琐碎的细节牵制着程序员，使他们不可能有更多的时间和精力去从事创造性的劳动，执行对他们来说是更为重要的任务，如确保程序的正确性、高效性。

（2）程序员既要驾驭程序设计的全局，又要深入每一个局部直到实现的细节，即使智力超群的程序员也常常会顾此失彼，屡出差错，因而所编出的程序可靠性差，且开发周期长。

（3）由于用机器语言进行程序设计的思维和表达方式与人们的习惯大相径庭，只有经过较长时间职业训练的程序员才能胜任，使得程序设计曲高和寡。

（4）因为它的书面形式全是“密”码，所以可读性差，不便于交流与合作。

（5）因为它严重地依赖于具体的计算机，所以其可移植性和可重用性差。

克服上述缺点的出路在于程序设计语言的抽象，让它尽可能地接近于算法语言。为此，人们首先注意到的是可读性和可移植性，因为它们相对地容易通过抽象而得到改善。

汇编语言实现了对机器语言的抽象。它将机器语言的每一条指令符号化：指令码代之以记忆符号，地址码代之以符号地址，使得其含义显现在符号上而不再隐藏在编码中，让人望“文”生义。同时，只要该计算机配备了汇编语言的一个汇编程序，汇编语言就摆脱了具体计算机的限制，可在具有不同指令集的计算机上运行，这无疑是机器语言朝算法语言靠拢迈出的一步。但是，它离算法语言还太远，以致程序员还不能从分解算法的数据、运算和控制，直至细化到汇编可直接表达的指令等繁杂琐碎的事务中解脱出来。

高级程序设计语言的出现使算法的程序表达产生了一次飞跃。诚然，算法最终要表达为

具体计算机上的机器语言才能在该计算机上运行，得到所需要的结果。但汇编语言的实践启发人们，表达成机器语言不必一步到位，可以分2步走或者可以筑桥过河，即先表达成一种中间语言，然后转成机器语言。汇编语言作为一种中间语言，并没有获得很大成功。原因是它离算法语言还太远。这便指引人们去设计一种尽量接受算法语言的规范语言，即所谓的高级程序设计语言。程序员可以用它方便地表达算法，然后借助于规范的高级语言到规范的机器语言的“翻译”，最终将算法表达为机器语言。而且，由于高级语言和机器语言都具有规范性，这里的“翻译”完全可以机械化地由计算机来完成，就像汇编语言被翻译成机器语言一样，只要计算机配备一个编译程序。上述两步，前一步由程序员去完成，后一步可以由编译程序去完成。在规定清楚它们各自该做什么之后，这两步是完全独立的。它们各自该如何做则互不相干。前一步要做的只是用高级语言正确地表达给定的算法，产生一个高级语言程序，后一步要做的只是将第一步得到的高级语言程序翻译成机器语言程序。至于程序员如何用高级语言表达算法和编译程序如何将高级语言表达的算法翻译成机器语言表达的算法，显然毫不相干。处理从算法语言最终表达成机器语言这一复杂过程的上述思想方法就是一种抽象。汇编语言和高级语言的出现都是这种抽象的范例。与汇编语言相比，高级语言的巨大成功在于它在数据、运算和控制3方面的表达中引入许多使之十分接近算法语言的概念和工具，大大地提高抽象地表达算法的能力。在运算方面，高级语言除允许原封不动地运用算法语言的四则运算、逻辑运算、关系运算、算术表达式、逻辑表达式外，还引入强有力的函数与过程等的工具，并让用户自定义。这一工具的重要性不仅在于它精简了重复的程序文本段，而且在于它反映出程序的2级抽象。在函数调用级，人们只关心它能做什么，不必关心它如何做。只是到定义函数时，人们才给出如何做的细节。用过高级语言的读者都知道，一旦函数的名称、参数和功能被规定清楚，那么，在程序中调用它们便与在程序的头部说明它们完全分开。可以修改一个函数甚至更换函数体而不影响调用该函数。如果把函数名看成是运算名，把参数看成是运算的对象或运算的结果，那么，函数调用和初等运算的引用就完全一样。利用函数以及函数的复合或嵌套，可以很自然地表达算法语言中任何复杂的运算。

在数据表示方面，高级语言引入了数据类型的概念，即把所有的数据加以分类。每一个数据（包括表达式）或每一个数据变量都属于其中确定的一类。称这一类数据为一个数据类型。因此，数据类型是数据或数据变量类属的说明，它指示该数据或数据变量可能取的值的全体。对于无结构的数据，高级语言除提供标准的基本数据类型——布尔型、字符型、整型和实型外，还提供用户可自定义的枚举类型、子界类型和指针类型。这些类型（除指针外），其使用方式都顺应人们在算法语言中使用的习惯。对于有结构的数据，高级语言提供了数组、记录、有限制的集合和文件等标准的结构数据类型。其中，数组是科学计算中的向量、矩阵的抽象；记录是商业和管理中的记录的抽象；有限制的集合是数学中足够小的集合的势集的抽象；文件是诸如磁盘等外存储数据的抽象。人们可以利用所提供的基本数据类型（包括标准的和自定义的），按数组、记录、有限制的集合和文件的构造规则构造有结构的数据。此外，还允许用户利用标准的结构数据类型，通过复合或嵌套构造更复杂更高层的结构数据。这使得高级语言中的数据类型呈明显的分层。高级语言中数据类型的分层是没有穷尽的，因而用它们可以表达算法语言中任何复杂层次的数据。

在控制方面，高级语言通常提供表达算法控制转移的如下方式：

（1）默认的顺序控制；

（2）条件（分支）控制；

（3）选择（情况）控制；

（4）循环控制；

（5）函数调用，包括递归函数调用；

（6）无条件转移。

以上算法控制转移表达方式不仅覆盖了算法语言中所有控制表达的要求，而且不再像机器语言或汇编语言那样原始、那样烦琐、那样隐晦，而是与自然语言的表达相差无几。

高级程序设计语言是对机器语言的进一步抽象，它所带来的主要好处是：

（1）高级语言更接近算法语言，易学、易掌握，一般工程技术人员只需要几周时间的培训就可以胜任程序员的工作；

（2）高级语言为程序员提供了结构化程序设计的环境和工具，使得设计出来的程序可读性好，可维护性强，可靠性高；

（3）高级语言不依赖于机器语言，与具体的计算机硬件关系不大，因而所写出来的程序可植性好、重用率高；

（4）由于把繁杂琐碎的事务交给了编译程序去做，所以自动化程度高，开发周期短，且程序员得到解脱，可以集中时间和精力去从事对于他们来说更为重要的创造性劳动，提高程序的质量。

1.3 抽象数据类型

1.3.1 抽象数据类型的基本概念

与机器语言和汇编语言相比，高级语言的出现大大地简便了程序设计。但算法从非形式的自然语言表达形式转换为形式化的高级语言表达，仍然是一个复杂的过程，仍然要做很多繁杂琐碎的事情，因而仍然需要进一步抽象。

对于一个明确的数学问题，设计它的算法，总是先选用该问题的一个数据模型。接着，弄清该问题所选用的数据模型在已知条件下的初始状态和要求的结果状态，以及这2个状态之间的隐含关系。然后探索从数据模型的已知初始状态出发到达要求的结果状态所必需的运算步骤。把这些运算步骤记录下来，就是求解该问题的算法。

按照自顶向下逐步求精的原则，在探索运算步骤时，首先应该考虑算法顶层的运算步骤，然后再考虑底层的运算步骤。所谓顶层的运算步骤是指定义在数据模型级上的运算步骤，或称宏观步骤。它们组成算法的主干部分。这部分算法通常用非形式的自然语言表达。其中，涉及的数据是数据模型中的一个变量，暂时不关心它的数据结构；涉及的运算以数据模型中的数据变量作为运算对象，或作为运算结果，或二者兼而为之，简称为定义在数据模型上的运算。由于暂时不关心变量的数据结构，这些运算都带有抽象性质，不含运算的细节。所谓底层的运算步骤是指顶层抽象的运算的具体实现。它们依赖于数据模型的结构，依赖于数据模型结构的具体表示。因此，底层的运算步骤包括2部分：一是数据模型的具体表示；二是定义在该数据模型上运算的具体实现。可以把它们理解为微观运算。于是，底层运算是顶层运算的细化；底层运算为顶层运算服务。为了将顶层算法与底层算法隔开，使二者在设计时不会互相牵制、互相影响，必须对二者的接口进行一次抽象。让底层只通过这个接口为顶层服务，顶层也只通过这个接口调用底层的运算。这个接口就是抽象数据类型。其英

文术语是 Abstract Data Types，简记 ADT。

抽象数据类型是算法设计和程序设计中的重要概念。严格地说，它是算法的一个数据模型连同定义在该模型上并作为该算法构件的一组运算。这个概念明确地把数据模型与该模型上的运算紧密地联系起来。事实正是如此。一方面，如前面指出过的，数据模型上的运算依赖于数据模型的具体表示，因为数据模型上的运算以数据模型中的数据变量作为运算对象，或作为运算结果，或二者兼而为之。另一方面，有了数据模型的具体表示，有了数据模型上运算的具体实现，运算的效率随之确定。于是，就有这样一个问题：如何选择数据模型的具体表示使该模型上的各种运算的效率都尽可能地高？很明显，对于不同的运算组，为使该运算组中所有运算的效率都尽可能地高，其相应的数据模型的具体表示将是不同的。在这个意义下，数据模型的具体表示又反过来依赖于数据模型上定义的那些运算。特别是，当不同运算的效率互相制约时，还必须事先将所有的运算相应的使用频度排序，让所选择的数据模型的具体表示优先保证使用频度较高的运算有较高的效率。数据模型与定义在该模型上的运算之间存在着的这种密不可分的联系是抽象数据类型的概念产生的背景和依据。

应该指出，抽象数据类型的概念并不是全新的概念。它实际上是熟悉的基本数据类型概念的引伸和发展。用过高级语言进行算法设计和程序设计的人都知道，基本数据类型已隐含着数据模型和定义在该模型上的运算的统一。事实上，大家都清楚，基本数据类型中的逻辑类型就是逻辑值数据模型与 3 种逻辑运算（或、与、非）的统一体；整数类型就是整数值数据模型与 4 种算术运算（加、减、乘、除）的统一体；实型和字符型等也类同。每一种基本类型都连带着一组基本运算。只是由于这些基本数据类型中的数据模型的具体表示、基本运算和具体实现都很规范，都可以通过系统内置而隐蔽起来，使人们看不到它们的封装。许多人由于习惯于在算法与程序设计中用基本数据类型名和相关的运算名，而不问其究竟，所以没有意识到抽象数据类型的概念已经孕育在基本数据类型的概念之中。

回到定义算法的顶层和底层的接口，即定义抽象数据类型。根据抽象数据类型的概念，对抽象数据类型进行定义就是约定抽象数据类型的名字，同时，约定在该类型上定义的一组运算的各个运算的名字，明确各个运算分别要有多少个参数，这些参数的含义和顺序以及运算的功能。一旦定义清楚，在算法的顶层就可以像引用基本数据类型那样，十分简便地引用抽象数据类型；同时，算法的底层就有了设计的依据和目标。顶层和底层都与抽象数据类型的定义打交道。顶层运算与底层运算没有直接的联系。因此，只要严格按照定义办，顶层算法的设计和底层算法的设计就可以互相独立、互不影响，实现对它们的隔离，达到抽象的目的。

在定义了抽象数据类型之后，算法底层的设计任务就可以明确为：

(1) 对于每一个抽象数据类型赋予其具体的构造数据类型，或者说，对于每一个抽象数据类型名赋予其具体的数据结构；

(2) 对于每一个抽象数据类型上所定义的每一个运算名赋予其具体的运算内容，或者说，赋予其具体的函数。

因此，算法底层的设计就是数据结构的设计和函数的设计。用高级语言表达，就是构造数据类型的定义和函数的说明。

不言而喻，由于实际问题千奇百怪，数据模型千姿百态，问题求解的算法千变万化，所以抽象数据类型的设计和实现不可能像基本数据类型那样规范。它要求算法设计和程序设计人员因时因地制宜、自行筹划，目标使抽象数据类型对外的整体效率尽可能地高。本书在介

绍各种抽象数据类型时会给出一些范例，供设计和实现时参考选用。

1.3.2 使用抽象数据类型的好处

使用抽象数据类型将给算法和程序设计带来很多的好处，主要有：

(1) 算法顶层的设计与底层的实现分离，使得在进行顶层设计时不考虑它所用到的数据和运算将如何分别表示和实现；反过来，在进行数据表示和底层运算实现时，只要定义清楚抽象数据类型而不必考虑它将在什么场合被引用。这样做，算法和程序设计的复杂性降低了，条理性增强了，既有助于迅速开发出程序的原型，又有助于在开发过程中少出差错，保证编出的程序有较高的可靠性。

(2) 算法设计与数据结构设计隔开，允许数据结构自由选择，从中比较，优化算法和提高程序运行的效率。

(3) 数据模型和该模型上的运算统一在抽象数据类型中，反映了它们之间内在的互相依赖和互相制约的关系，便于空间和时间耗费的折衷，灵活地满足用户的要求。

(4) 由于顶层设计和底层实现的局部化，在设计中出现的差错也是局部的，因而容易查找，也容易纠正。在设计中常常要做的增、删、改也都是局部的，因而也都很容易进行。因此，可以肯定，用抽象数据类型表述的程序具有很好的可维护性。

(5) 编出来的程序自然地呈现模块化，而且抽象数据类型的表示和实现都可以封装起来，便于移植和重用。

(6) 为自顶向下逐步求精和模块化提供一种有效的途径和工具。

(7) 编出来的程序结构清晰、层次分明，便于程序正确性的证明和复杂性的分析。

1.3.3 数据结构、数据类型和抽象数据类型

数据结构、数据类型和抽象数据类型，这3个术语在字面上既不同又相近，反映出它们在含义上既有区别又有联系。

数据结构是在整个计算机科学与技术领域中广泛使用的术语。它用来反映数据的内部构成，即数据由哪些成分数据构成，以什么方式构成，呈什么结构。数据结构有逻辑上的数据结构和物理上的数据结构之分。逻辑上的数据结构反映成分数据之间的逻辑关系；物理上的数据结构反映成分数据在计算机内的存储安排。数据结构是数据存在的形式。

数据是按照数据结构分类的，具有相同数据结构的数据同属一类。同一类数据的全体称为一个数据类型。在高级程序设计语言中，数据类型用来说明数据在数据分类中的归属。它是数据的一种属性。这个属性限定了该数据的变化范围。为了解题的需要，根据数据结构的种类，高级语言定义了一系列的数据类型。不同的高级语言所定义的数据类型不尽相同。简单数据类型对应于简单的数据结构；构造数据类型对应于复杂的数据结构；在复杂的数据结构里，允许成分数据本身具有复杂的数据结构，因此，构造数据类型允许复合嵌套；指针类型对应于数据结构中成分数据之间的关系，表面上属简单数据类型，实际上都指向复杂的成分数据即构造数据类型中的数据，因此单独划出一类。

数据结构反映数据内部的构成方式，它常常用一个结构图来描述：数据中的每一项成分数据被看做一个结点，并用方框或圆圈表示，成分数据之间的关系用相应的结点之间带箭头的连线表示。如果成分数据本身又有它自身的结构，则结构出现嵌套。

由于数据类型是按照数据结构划分的，因此，一类数据结构对应着一种数据类型。一个

数据变量，在高级语言中的类型说明必须是该变量所具有的数据结构所对应的数据类型。

最常用的数据结构是数组结构和记录结构。

数组结构的特点是：

（1）成分数据的个数固定，它们之间的逻辑关系由成分数据的序号（数组的下标）来体现。这些成分数据按照序号的先后顺序一个挨一个地排列起来。

（2）每一个成分数据具有相同的结构（可以是简单结构，也可以是复杂结构），因而属于同一数据类型（相应地是简单数据类型或构造数据类型）。这种同一的数据类型称为基类型。

（3）所有成分数据被依序安排在一片连续的存储单元中。概括起来，数组结构是一个线性的、均匀的、可随机访问其成分数据的结构。由于这种结构有这些良好的特性，所以常被人们所采用。

记录结构是另一种常用的数据结构。它的特点是：

（1）与数组结构一样，成分数据的个数固定，但成分数据之间没有自然序，它们处于平等地位。每一个成分数据被称为一个域并赋予域名。不同的域有不同的域名。

（2）不同的域允许有不同的结构，因而允许属于不同的数据类型。

（3）与数组结构一样，可以随机访问其成分数据，但访问的途径是靠域名。在高级语言中记录结构对应的数据类型是记录类型。记录结构的数据的变量必须说明为记录类型。

抽象数据类型的含义在上一段已作了专门叙述。它可理解为数据类型的进一步抽象，即把数据类型和数据类型上的运算捆在一起，进行封装。引入抽象数据类型的目的是把数据类型的表示和数据类型上运算的实现，与这些数据类型和运算在程序中的引用隔开，使它们相互独立。对于抽象数据类型的描述，除了必须描述它的数据结构外，还必须描述定义在它上面的运算（函数）。抽象数据类型上定义的运算，以该抽象数据类型的数据所应具有的数据结构为基础。

1.4 用 C++ 描述数据结构与算法

描述数据结构与算法可以有多种方式，如自然语言方式、表格方式等。在本书中，采用C++语言来描述数据结构与算法。C++语言的优点是类型丰富、语句精炼，具有面向过程和面向对象的双重特点。用C++语言来描述算法可使整个算法结构紧凑，可读性强。在本书中，有时为了更好地阐明算法的思路，还采用C++与自然语言相结合的方式来描述算法。本节对C++语言的若干重要特性作简要概述。

1.4.1 变量、指针和引用

1. 变量

变量是程序设计语言对存储单元的抽象，它具有以下属性：

变量名（name）：变量名是用于标识变量的符号。

地址（address）：地址是变量所占据的存储单元的地址。变量的地址属性也称为左值。

大小（size）：变量的大小指该变量所占据的存储空间的数量（以字节数来衡量）。

类型（type）：变量的类型指变量所取的值域以及对变量所能执行的运算集。

值（value）：变量的值是指变量的所占据的存储单元中的内容。这些内容的意义由变量的类型所决定。变量的值属性也称为右值。

生命期（lifetime）：变量的生命期是指在执行程序期间变量存在的时段。

作用域（scope）：变量的作用域是指在程序中变量被引用的语句范围。

2. 指针变量

C++中的指针变量是一个 T * 类型的变量。其中，T 为任一已定义的类型。

指针变量用于存放对象的存储地址。例如：

```
int n=8;
int * p;
p=&n;
int k= * p;
```

其中，p 是一个指向 int 的指针。通过间接引用指针存取指针所指向的变量。

3. 引用

在 C++中，引用是变量的一个替代名。引用的定义与变量的定义很相似，但引用不是变量。

T & 表示对一个类型为 T 的变量的引用。例如

```
int i=5;
int &j=i;
i=7;
cout<<i<<endl;
cout<<j<<endl;
```

其中，j 是对变量 i 的一个引用。当 i 的值改变时，j 的值也跟着改变。因此，上面的输出语句输出的 i 和 j 的值都是 7。

1.4.2 函数与参数传递

1. 函数

C++中有 2 种函数：常规函数和成员函数。不论哪种函数，其定义都包括 4 个部分：函数名、形式参数表、返回类型和函数体。函数的使用者通过函数名来调用该函数。调用函数时，将实际参数传递给形式参数作为函数的输入。函数体中的处理程序实现该函数的功能。最后将得到的结果作为返回值输出。例如，下面的函数 max 是一个简单函数的例子。

```
int max(int x,int y)
```

```
{
    return x>y? x:y;
}
```

其中，max 是函数名；函数后圆括号中的 int x 和 int y 是形式参数；函数前面的 int 是返回类型；花括号内是函数体，它实现函数的具体功能。

C++ 中函数一般都有一个返回值。函数的返回值表示函数的计算结果或函数执行状态。如果所定义的函数不需要返回值，可使用 void 来表示它的返回类型。函数的返回值通过函数体中的 return 语句返回。return 语句的作用是返回一个与返回类型相同类型的值，并中止函数的执行。

2. 参数传递

在 C++ 中调用函数时传递给形参表的实参必须与形参在类型、个数、顺序上保持一致。参数传递有 2 种方式。一种是按值传递方式。在这种参数传递方式下，把实参的值传递给函数局部工作区相应的副本中。函数使用副本执行必要的计算。因此函数实际修改的是副本的值，实参的值不变。另一种是按引用传递方式。在这种参数传递方式下，需将形参声明为引用类型，即在参数名前加上符号“&”。当一个实参与一个引用类型结合时，被传递的不是实参的值，而是实参的地址。函数通过地址存取被引用的实参。执行函数调用后，实参的值将发生改变。例如

```
void Swap(int &x,int &y)
{
   int temp=x;
   x=y;
   y=temp;
}
```

函数调用Swap(x,y) 交换变量 x 和 y 的值。

在 C++ 中数组参数的传递属特殊情形。数组作为形参可按值传递方式声明，但事实上采用引用方式传递。实际传递的是数组第一个元素的地址。因此在函数体内对于形参数组所作的任何改变都会在实参数组中反映出来。

若传递给函数的值参是一个对象（作为类的实例），在函数中就创建了该对象的一个副本。在创建这个副本时不调用该对象的构造函数，但在函数调用结束前要调用该副本的析构函数撤消这个副本。若采用引用方式传递对象，在函数中不创建该对象的副本，因而也不需撤消副本，但函数将改变引用传递的对象。

1.4.3 C++的类

C++ 的类（class）体现了抽象数据类型（ADT）的思想，它将说明与实现分离。

C++ 的类由 4 个部分组成：

(1) 类名；

(2) 数据成员；

(3) 成员函数；

(4) 访问级别。

对类成员的访问有 3 种不同的级别：公有（public）、私有（private）和保护（protected）级别。在 public 域中声明的数据成员和函数成员可以在程序的任何部分访问；在 private 和 protected 域中声明的数据成员和函数成员构成类的私有部分，只能由该类的对象和成员函数，以及被声明为友员（friend）的函数或类的对象对它们进行访问。此外，在 protected 域中声明的数据成员和函数成员还允许该类的子类访问它们。下面是 C++ 中定义的矩形类 Rectangle 的例子。

```
class Rectangle {
public:
  Rectangle(int,int,int,int);        //构造函数
  ~Rectangle( );                     //析构函数
  int GetHeight( );                  //矩形的高
  int GetWidth( );                   //矩形的宽
private:
  int x1,y1,h,w;
  //(x1,y1)是矩形左下角点的坐标;
  //h 是矩形的高;w 是矩形的宽。
}

Rectangle::GetHeight( ) {return h;} //返回矩形的高
Rectangle::GetWidth( ) {return w;} //返回矩形的宽
```

1.4.4 类的对象

下面的代码段说明了如何声明类 Rectangle 的对象以及如何调用其成员函数。

```
Rectangle r(0,0,2,3);
Rectangle s(0,0,3,4);
Rectangle *t=&s;
if (r.GetHeight( )*r.GetWidth( )>t->GetHeight( )*t->GetWidth( ))
  cout≪"矩形 r";
else cout≪"矩形 s";
cout≪"的面积较大。"≪endl;
```

类对象的声明与创建方式类似于变量的声明与创建方式。对一个对象成员进行访问或调用，可采用直接选择（·）或间接选择（→）来实现。

1.4.5 构造函数与析构函数

C++类的构造函数（Constructor）用于初始化一个对象的数据成员。构造函数名与它所在的类名相同。构造函数必须声明为类的公有成员函数。构造函数不可有返回值也不得指明返回类型。例如，类 Rectangle 的构造函数可定义如下：

```
Rectangle::Rectangle(int x=0,int y=0,int height=0,int width=0)
:x1 (x), y1 (y), h (height), w (width)
{ }
```

可用如下方式声明 Rectangle 的对象 r、s 和 t。

```
Rectangle r(0,0,2,3);
Rectangle * s=new Rectangle(0,0,3,4);
Rectangle t;
```

析构函数（Destructor）用于在一个对象被撤消时删除其数据成员。析构函数名也与它的类名相同，并在前面加上符号“~”。

1.4.6 运算符重载

C++允许为用户定义的数据类型重载运算符。

下面的代码段实现对类 Rectangle 的运算符 == 的重载。

```
bool Rectangle::operator==(const Rectangle &s)
{
   if (this== &s) return true;
   if ((x1==s.x1)&&(y1==s.y1)&&(h==s.h)&&(w==s.w)) return true;
   else return false;
}
```

其中，用到 C++中的保留字 this。在类的成员函数内部，this 表示一个指向调用该成员函数的对象的指针，因此该对象也可用 * this 来表示。

经重载运算符==后，即可用运算符==来判定 2 个 Rectangle 对象是否相同。

1.4.7 友元函数

在类的声明中可使用保留字 friend 来定义友元函数。友元函数实际上并不是这个类的成员函数。它可以是一个常规函数，也可以是另一个类的成员函数。如果想通过这个函数来存取类的私有成员和保护成员，就必须在类的声明中给出该函数的原型，并在前面加上 friend。

1.4.8 内联函数

在函数定义前加上一个 inline 前缀，该函数就被定义成一个内联函数。内联函数的保留字 inline 告诉编译器在任何调用该内联函数的地方直接插入内联函数的函数体。

1.4.9 结构

在 C++ 中，结构（Struct）与类的区别是，在结构中默认的访问级别是 public，而在类中默认的访问级别是 private。除此之外，struct 与 class 是等价的。

1.4.10 联合

联合（Union）是一种结构。在 C++ 中，联合可以包含变量和函数，还可以包含构造函数与析构函数。因此，可以用联合来定义类。C++ 的联合保留了所有 C 的特性，其中最重要的是让所有数据成员共享相同的存储地址。与结构类似，联合的默认访问级别是 public。

在使用 C++ 的联合时应注意：联合不能继承其他任何类型的类；联合不能是基类，不能包含虚成员函数；联合不能含有静态变量；如果一个对象有构造函数与析构函数，那么它不能成为联合的成员；如果一个对象重载了运算符“=”，它也不能成为联合的成员。

1.4.11 异常

C++ 的异常（Exception）提供了一种处理错误的简洁的方法。当程序发现一个错误，就引发一个异常，以便在程序最合适的地方捕获异常并进行处理。在 C++ 中，异常是一个对象，是从基类 exception 派生出来的。程序通过 throw 来引发异常。例如：

```
class error { };
void f(void)
{
    //..............
    throw error( );
}
```

throw 语句类似于 return 语句，但它描述函数的异常终止。异常处理程序通常用一个 try 块来定义。在引发异常之前，程序一直执行 try 块体。在 try 块体之后有一个或多个异常处理程序。每一个异常处理程序由一个 catch 语句组成。这个语句指明欲捕获的异常以及出现该异常时要执行的代码块。当 try 引发了一个已定义的异常时，控制就转移到相应的异常处理程序中。

```
void g(void)
{
  try {
    f( );
```