目　录

第1章 预备知识

21世纪人类将进入信息化社会，信息化社会的基础之一是计算机。计算机作为信息处理的工具，在人类科技活动史上获得了空前最快速度的发展。

计算机的出现将人类的创造性思维推向一个更高的阶段。思维活动可以利用语言来形式化，而语言层次可以离开意识层次相对独立地活动。计算机语言作为人和计算机之间进行意识交流的工具，人通过计算机语言将意识活动交给计算机进行独立的加工，产生进一步的思维活动，所以可以认为计算机是人类思维的工具。计算机思维是一种物化的思维，是人脑思维的进一步延伸。

在计算机语言层次，人与计算机的意识活动的交流是通过程序设计这个环节来完成的。1976年N. Wirth出版了一本题名为《Algorithms + Data Structure = Programs》的著作，提出了程序是算法和数据结构的结合的观点，也就是说程序设计主要包括两方面的内容：行为特性的设计和结构特性的设计。行为特性的设计是指完整地描述问题求解的全过程，并精确定义每个解题步骤，这一过程即是算法的设计；而结构特性的设计是指在问题求解的过程中，计算机所处理的数据之间的联系，及这些联系的表示方法。随着人们对程序设计概念的认识的加深，进一步将文档也作为软件的一个部分。

在这一章预备知识里，将首先讨论算法这个组成程序的基石，然后介绍关于逻辑代数的基础知识，最后结合程序设计语言的发展介绍程序设计思想的沿革，从而得出C++的出现是历史的必然的结论。关于数据结构的知识将随着语言的介绍逐步展开。

1.1 算 法

1. 算法的概念

粗略地说，算法是解决问题的具体步骤。解决任何一个问题，小到冲制一杯速溶咖啡，大至联合国一次安理会议，都必须有一个确定的步骤。例如冲咖啡时必须先准备好杯子，再准备开水，开水冲入咖啡后对不同的客人还会有是否加糖及是否加牛奶等不同的步骤。当然这是一个原始的算法，对这个算法的描述采用了自然语言的方法。

在欧几里得的《几何原本》里，阐述了求两个数的最大公因子的过程，也称欧几里得算法。

欧几里得算法：给定两个正整数m和n，求它们的最大公因子，即同时能够整除m和n的最大正整数。

Step1. 以n除m，并令r为所得余数(显然 $n > r \geqslant 0$)，

Step2. 若 $r = 0$，算法结束，n即为m和n的最大公因子，

Step3. 置 $m \leftarrow n$，$n \leftarrow r$，返回Step1。

在描述欧几里得算法的时候，我们采用了一种比自然语言更抽象一些的方法，或者称

为半自然语言的方法。同时欧几里得算法的描述与冲制速溶咖啡的算法相比，具有完全的确定性。在欧几里得算法的描述中引入了一些变量，如 m、n、r，引入了关系运算符 =、≥，还引入了记号 ← 表示将该符号右边的变量中的值送到左边的变量中去，所以← 也称为赋值号。在算法的描述乃至程序的设计中，变量是个很重要的概念。所谓变量是指一块在计算机内存中分配的存储空间，或者说，每个变量名代表了内存中一个确定的存储区。赋值是指将赋值号右边的变量中存储的值送到左边的变量所代表的存储单元中去。

还需要注意的是，在算法中不仅各步骤间的顺序是重要的，在每步内的动作次序同样也是重要的。例如在欧几里得算法的 Step3 中，“置 m←n，n←r ”绝不能写成“置 n←r，m←n”，因为这样做的结果是在 m、n 和 r 中最后都存储了同一个值 r，即都变成了 Step1 除法运算的余数。

由于算法是对实际运算的抽象表述，同时算法描述的往往是比冲制咖啡复杂得多的运算，所以一个算法的每一步的含义（或者说是作用）往往不是一目了然的。学习一个算法的最好方法莫过于将该算法所允许的实例代入到算法中，按算法的步骤去执行算法，这种由人来执行算法的方法称为“人工模拟”，人工模拟也是开发和检验算法的一种重要手段，通过人工模拟能了解算法每一步的实际含义。算法中一般都包含了“如果…则”的步骤，规定对不同的实例以不同的途径处理，所以为深刻弄懂一个算法，一般还需要设计多个实例来反复体会算法中各种不同处理途径的作用。

对欧几里得算法，第一个实例是：

m = 12 和 n =4，

Step1. m/n = 12/4 = 3 …… 0，

Step2. 由于余数 r = 0，算法结束，4 即为 12 和 4 的最大公因子。

由于这个实例没有“走”到 Step3，所以由这个实例不能全面了解欧几里得算法，第二个实例是：

m = 48 和 n = 14，

Step1. m/n = 48/14 = 3 …… 6（r=6），

Step2. 余数 r 不为零，进入下一步，

Step3. 置 m←n、n←r，即 m = 14、n = 6，返回 Step1。

Step1. m/n = 14/6 = 2 …… 2（r=2），

Step2. 余数 r 不为零，进入下一步，

Step3. 置 m←n、n←r，即 m = 6、n = 2，返回 Step1。

Step1. m/n = 6/2 = 3 …… 0，

Step2. 由于余数 r = 0，算法结束，n 的现值 2 为 48 和 14 的最大公因子。

对欧几里得算法的跟踪应当可以告一段落了，但再深入地想一想，欧几里得算法的上述表述是否尚有可改进之处呢？下一个实例是：

m = 14 和 n =48，

Step1. m/n = 14/48 = 0 …… 48，

Step2. 由于余数 r = 48，进入 Step3，

Step3. 置 m←n、n←r，即 m = 48、n = 14，返回 Step1。

我们可以发现，当 m 小于 n 时，经过 Step1、Step2 和 Step3 后，仅将 m 和 n 互换了一

次。所以，可以为欧几里得算法增加 Step0：

Step0. 如果 m<n，则 m ↔ n。

这样尽管欧几里得算法由三步变为四步，但对 m<n 的实例，实际减少了算法的执行时间。而按 m 和 n 之间大小关系的出现概率考虑，m<n 及 m>n 出现的概率可能各为一半。

至此可以给算法一个更精确的定义：一个算法是一个有穷规则的集合，其中的规则规定了解决一个特定问题的运算序列。作为一个算法必须具备以下五个特性：

1）有穷性　一个算法必须总是在有穷步之后结束。

欧几里得算法满足这个条件。因为在 Step1 以后，r 的值肯定小于 n，所以如果 r 不等于零，则在经过 Step3 将 r 的值赋给 n 以后，重新进行 Step1 时 n 的值已经减小，正整数的递减序列必然最后要终止。所以对任何给定的 m 和 n 的值，Step1 只会执行有穷次。对足够大的 m 和 n，可能算法执行的次数相当之多，但即使达到天文数字，仍然是有穷的。

如果一个计算不具有有穷性但具有算法其他特性，则可称其为计算方法。例如对无穷调和级数的和的计算就不是一个算法而是一个计算方法，在确定了项数或确定了对精度的要求以后，才能满足有穷性的要求，也就是说，这时对无穷调和级数的和的计算才可能成为一个算法。

2）确定性　算法的每一个步骤都必须有确定的定义。

欧几里得算法的每一步都是确定的。例如在 Step1 中，除法的算术运算法则保证了两个正整数相除的步骤，而结果的商和余数都是确定的。

按照确定性的要求，冲制一杯速溶咖啡不是一个算法，因为其中若干步骤不具备确定性，例如“加一些牛奶并轻轻搅拌”，“一些”是多少、何谓“轻轻”等等都是不确定的说明。

3）能行性　算法的能行性是指算法中有待实现的每一个步骤都必须是基本的。

欧几里得算法涉及到的运算包括整数的表示、整数的除法、整数是否为零的判断及整数的赋值，这些运算都是基本的、能行的。

非能行的一个例子是：“如果 2 是使方程 $x^n+y^n=z^n$ 有正整数解 x、y、z 的 n 中最大的整数，则执行 Step4 ”，这一步骤就不是基本的能行的。

4）一个算法有 0 个或若干个输入，算法的输入是算法执行的初始数据。

欧几里得算法需要两个正整数 m 和 n 作为初始数据。

5）一个算法有一个或多个输出，作为算法执行的结果。

欧几里得算法的结果是正整数 m 和 n 的最大公约数。

2. 算法的基本结构和表示

对算法和程序设计方法的理论研究及程序设计实践指出，算法的基本组成结构只需要有三种，第一种是顺序结构，第二种是选择结构，第三种是循环结构。或者说，任何一个算法，无论其多么简单或多么复杂，都可由这三种结构组合和构造而成。

前面在讨论欧几里得算法时，已经有了变量的概念和赋值运算的概念，并介绍了算法的一种类似于自然语言的表示方法。算法的作用在于记录及交流人的解决问题的思想，同时算法也是作为编制计算机程序的前导步骤。为使算法的描述更简捷和清晰，人们研究和创造了多种表示算法的方法，本书主要介绍算法的伪码表示法和流程图表示法。伪码是一种用以表示算法的代码，伪码采用了类似于程序设计语言的语句表示方法，但伪码不是任何

一种程序设计语言，不涉及程序设计的细节。流程图是一种图语言，用流程图可以直观地了解算法的结构。

下面分别讨论顺序结构、选择结构和循环结构的含义，以及其伪码表示法和流程图表示法。讨论将结合算法的举例进行。

1）顺序结构

在伪码表示法中，每一个步骤称为一条语句。根据需要可将多条语句包容在 BEGIN 和 END 之间组成语句块（简称块），即组织成下列形式：

```
BEGIN
   语句 1
   语句 2
     .
     .
     .
   语句 n
END
```

顺序结构中，各语句或块是按照在算法中排列的先后次序执行的。

【例 1.1】求两个数的绝对值之和。

算法 1：

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* 输入：任意两个数                                                          *
* 输出：输入的两个数的绝对值之和                                            *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
num1 ← 第一个输入的数
num2 ← 第二个输入的数
num1 ← num1 的绝对值
num2 ← num2 的绝对值
sum ← num1 + num2
输出 sum 的值
```

流程图表示中所用的符号已由国家标准局制定了国家标准（见图 1.1）。

算法 1 用流程图表示为图 1.2。

2）选择结构

选择结构根据某种条件选择性地执行算法的某一部分。选择结构最常见的是判断算法中所说明的条件是否成立，如果条件成立执行某个语句或语句块，否则执行另一个语句或语句块，后一个语句（或语句块）在某些情况下可以省略。

选择结构用伪码表示为：

```
IF 〈条件〉
   〈语句或语句块 1〉
[ELSE
   〈语句或语句块 2〉]
```

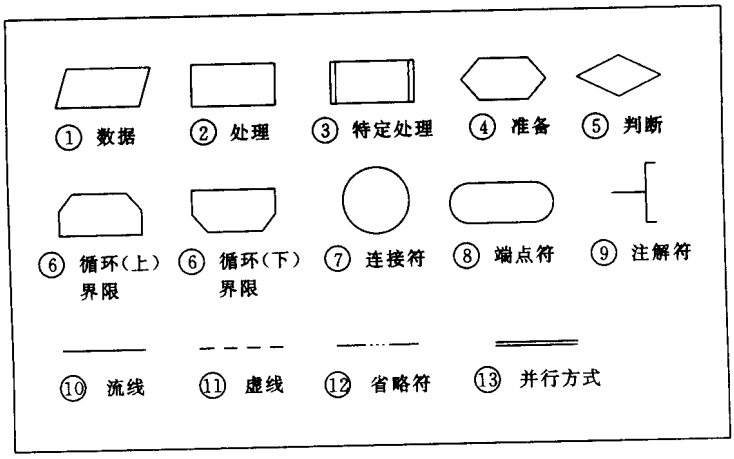


图 1.1　标准流程图的规定符号

无论是执行了〈语句或语句块 1〉,还是执行了〈语句或语句块 2〉都完成了选择结构的执行。上列表示还说明,如果我们在条件不成立时不要求做任何动作,可以省略 ELSE 的那一部分。

选择结构的流程图表示见图 1.3。

【例 1.2】输入三个数,求其中最大的数。

算法 2:

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* 输入:任意三个数                                                          *
* 输出:输入的三个数中的最大的数                                            *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
num1 ← 第一个输入的数
num2 ← 第二个输入的数
num3 ← 第三个输入的数
IF (num1 < num2)
   IF (num2 < num3)
     max ← num3
   ELSE
     max ← num2
ELSE
   IF (num1 < num3)
     max ← num3
   ELSE
     max ← num1
输出 max 的值
```

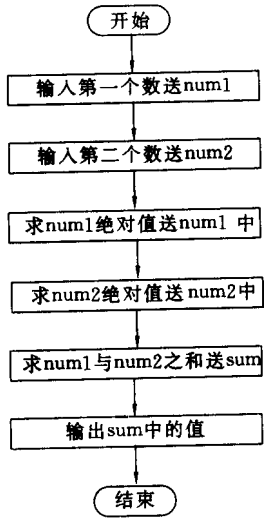
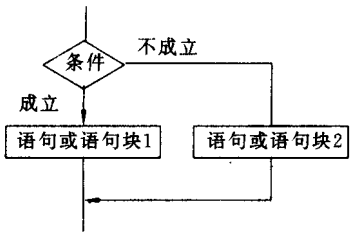


图 1.2　算法 1 的流程图

算法 2 用流程图表示为图 1.4。

3) 循环结构



循环结构根据某种条件重复性地执行算法所规定的某一部分。在这一部分执行完成后，再一次判断所说明的条件是否成立，如果条件成立再一次执行这一部分，如此循环重复直致条件不成立为止，从而达到重复性地执行算法所规定的某一部分的目的。

图 1.3　选择结构的流程图

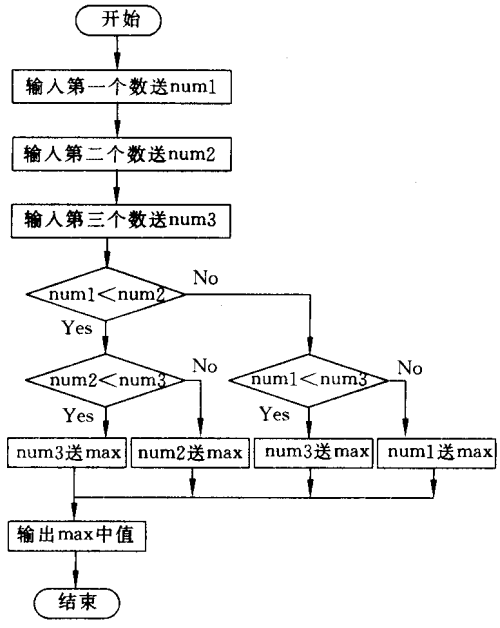


图 1.4　算法 2 流程图

循环结构用伪码表示为：

```
WHILE〈条件〉
   BEGIN
    〈语句或语句块〉
   END
```

循环结构的流程图表示见图 1.5。

【例 1.3】设计一个算法，统计输入的一组数据中非负整数的个数。

算法 3：

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* 输入：有限个数的一组输入数据                                                  *
* 输出：统计这组输入数据中的非负整数的个数                                      *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
1  number ← 输入数据的个数
2  count ← 0                    // 非负整数的个数，先预置为 0
```

```
3  WHILE (number ≠ 0)
     BEGIN
4      num ← 输入的下一个数据
5      IF (num ≥ 0)
6          count ← count + 1
7      number ← number − 1
     END
8  输出 count 的值
```

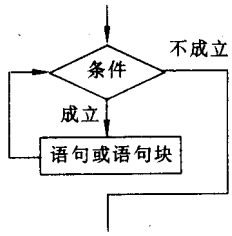


图 1.5　循环结构的流程图

算法 3 和算法 2 不同，算法 2 中预先确定了输入数据的个数，而算法 3 并没有限定输入数据的个数，也就是说该算法能够灵活地处理不同个数的输入数据，所以定义了一个变量 number，该变量作为算法所要求的第一个输入量，代表算法当前这次执行所处理的输入数据的个数，从第二个输入量开始才是算法所处理的输入数据。在算法执行中，每处理了一个输入数据，将 number 的值减 1，一直到所有的输入数据处理完成为止。

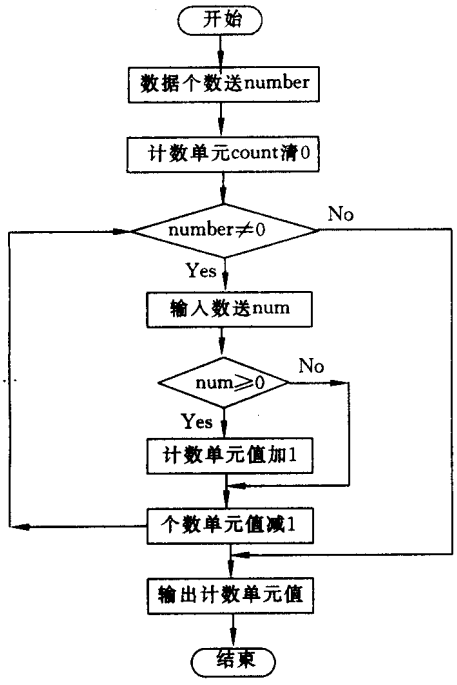


图 1.6　算法 3 的流程图

算法 3 用“//”的方法引入了对算法某个步骤的注释。为提高算法(以及程序)的可读性，在其适当的地方加以注释是十分必要的，应有意识地逐步培养这种良好的习惯。

算法 3 用流程图表示为图 1.6。

和前面几个算法相比，这个算法显然要复杂一些。为了学习这个算法让我们用人工模拟的方法，将该算法所允许的实例代入到算法中去，按算法的步骤人工执行一次。为跟踪算法的动态执行过程，需要将算法中的可执行语句及包括判断的语句按顺序编号，同时为记录算法的当前执行的以及下一步将执行的语句，以及记录算法中的变量的值的变化过程，需要画出一张算法的动态跟踪表。跟踪表的第一列是执行的步数，第二列和第三列分别是这一步所执行的语句的编号及下一步将执行的语句的编号，第四列是当前这一步的注记，以后的各列是算法中所用的各个变量。算法每执行一步，跟踪表就增加一行，记录在这一步执行的语句标号，展望下一步将执行的语句，同时记录这一步对算法中变量的值作用结果，随着算法的逐步执行，跟踪表逐行增加，直至过程结束。算法 3 的跟踪表见表 1.1，在人工模拟前这张表是一张空表，设处理 5 个数：6，−17、0、3 和 −10，表中是人工模拟后跟踪的结果。

表 1.1 算法 3 的跟踪表

步	当前语句	下一语句	注 记	number	count	num
1	1	2	处理 5 个数	5		
2	2	3			0	
3	3	4	输入第一个数			
4	4	5				6
5	5	6				
6	6	7	循环变量减 1		1	
7	7	3		4		
8	3	4	输入第二个数			
9	4	5				−17
10	5	6				
11	6	7	循环变量减 1			
12	7	3		3		
13	3	4	输入第三个数			
14	4	5				0
15	5	6				
16	6	7	循环变量减 1		2	
17	7	3		2		
18	3	4	输入第四个数			
19	4	5				3
20	5	6				
21	6	7	循环变量减 1		3	
12	7	3		1		
23	3	4	输入第五个数			
24	4	5				−10
25	5	6				
26	6	7	循环变量减 1			
27	7	3	循环结束	0		
28	3	8				
29	8		输出最大的数			

3. 算法的基本分类

算法是解决问题的方法，不同的领域有各自的算法。如果根据问题的领域来区分，算法可分为数值问题的算法和非数值问题的算法。数值问题算法是指解决传统数学问题的算法，例如解方程的算法、解方程组的算法、以及积分算法和微分算法等等，随着计算机在数值计算方面应用和研究的发展，求各种数值问题的近似解的算法也获得迅速的发展和应用。数值问题的算法是数学研究的一个重要方面，本书将在第 9 章给出几个数值问题算法的例子。和数值问题的算法相比，计算机科学往往对非数值问题的算法给与了更多的重视，

计算机非数值应用领域同样也是个相当宽广的领域，在各个不同的方向上有不同的非数值问题的算法，例如在图论中有涉及图的各种处理的图算法，在规划问题中也有各种规划的算法。所以算法的研究基本上是依赖于具体的研究领域的。

尽管不同的领域有各自的算法，但从算法所采用的方法或思路上看，最基本的大致可分成以下几种：直接法、枚举法、递推法、递归法、回溯法、数字模拟等等。下面先简单介绍直接法、枚举法、递推法和递归法的基本思想，其他一些方法将随着课程学习的深入逐步展开。

1）直接法

直接法是指根据问题所给的条件，直接通过计算来得到解答。本章的算法 1 是直接法的一个例子，通过计算直接得到两个数的绝对值之和。

2）枚举法

枚举法又称穷举法。枚举法通过逐一考察问题的所有可能解，来找出问题的真正的解。由枚举的方法可见，枚举法的条件是问题的可能解必须是有限的，而且这些可能解必须是已知的。

【例 1.4】给定一个正整数，确定它的整数的立方根是否存在，如存在则找出这个立方根。

显然，一个正整数的整数立方根如存在的话，肯定在这个数和零之间，而这之间的正整数的个数是有限的，所以可以设计一个基于枚举法的算法。

算法 4：

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* 输入：输入的一个正整数                                                  *
* 输出：这个正整数的立方根(如果这个立方根存在的话)                        *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
number ← 输入一个正整数
result ← 0                          // 正整数的立方根，先预置为 0
WHILE (result ≤ number)
   BEGIN
      IF (result×result×result = number)
         BREAK
      ELSE
         result ← result + 1
   END
IF result ≤ number
   result 是 number 的立方根，输出 result 的值
ELSE
   输出 number 没有立方根的信息
```

3）递推法

递推法是从已知的初始条件出发，逐次推出中间结果，每一步在理想状态下应逐步接近问题的最后解。下面举一个简单的递推法的例子。

【例 1.5】计算一个正整数的阶乘，即计算 N!。

根据数学知识 N! = 1×2×3×・・・×N,所以可以从 1 开始,得到 1 的阶乘后乘 2 得到 2!,再和 3 相乘得 3!,从而最后得到 N!。

算法 5:

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* 输入:输入一个正整数                                                    *
* 输出:这个正整数的阶乘                                                  *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
number ← 输入一个正整数
result ← 1                    // 阶乘的初始值必须预置为 1
current ← 1
WHILE (current ≤ number)
   BEGIN
     result ← result × current
     current ← current + 1
END
输出 result
```

阶乘的计算是递推法求解的一个典型的例子。其他的非数值算法中递推求解的例子还有求级数的和等等。在实际的计算中往往要注意递推中结果值的增长情况,例如阶乘的增长就非常之快,10! = 3628800,13! 就超过 C++ 中整数的最大允许范围。

递推法在数值算法中又称迭代法,迭代法经常用于求近似解的问题,根据对前一步结果的误差的不同处理方法,迭代法又有逼近迭代和试探迭代等不同的方法。数值计算中要注意解的稳定性问题,即在迭代中每一步的解应越来越接近真正的解,否则迭代不会成功。

4)递归法

一个直接或间接调用自身的算法称为递归算法,一个使用函数自身给出定义的函数称为递归函数。在有些问题的算法描述中,递归法往往比非递归法直接易懂。

【例 1.6】将计算一个正整数的阶乘的问题用递归法求解。

阶乘函数的递归定义为:

$0! = 1$

$N! = (N-1)! \times N$

第一式给出阶乘函数的初始值,初始值是非递归定义的。每个递归函数都需要给出非递归定义的初始值,否则这个函数无法计算出来。第二式用较小自变量的函数值来表达较大自变量的函数值,在定义式两边都引用了阶乘记号,所以是一个递归定义。

递归算法的描述和处理将在本书的第 6 章介绍。

由上面的简单介绍已经可以看出,算法是解决问题的基础,为解决一个问题可能设计出不同的算法。那么什么是一个好的算法呢?衡量算法的质量指标包括算法的正确性、可读性、健壮性以及高的执行效率和低的存储空间要求。正确性是指算法对合法的输入应当能产生所要求的输出;可读性将使算法易于交流和理解;健壮性是指算法应在一定的范围内能对不合法的输入作必要的反应,而不致产生一些莫明其妙的结果;高的执行效率和低的存储空间要求是对算法在时间和空间的效率要求。本书将陆续对这些问题展开讨论。

1.2 逻辑代数基础

在算法的选择结构和循环结构中，都要求对给出的条件进行测试，以便根据条件是否成立来执行算法的不同动作。

这种“条件”在逻辑上被称为“命题”。一般而言，逻辑是指思维的规律和客观的规律性，因此逻辑将研究正确推理的规则。所谓命题在逻辑中是指可以决定真假的陈述句，符合事实的陈述句称为真命题，违反事实的陈述句称为假命题。现在逻辑已成为计算机科学领域中的一个重要的研究内容。

现代逻辑源于古代数学家和哲学家的工作，德国著名数学家莱布尼兹(G. W. Leibniz)提出了所有论证都能通过计算来解决的思想，1854 年英国数学家布尔(G. Boole) 发表了一部重要的著作《思维的规律》，实现了莱布尼兹的思想。其基本点是用符号来表达语言和思维的逻辑性，将逻辑看成是由变量和运算符构成的系统，在这个系统中表达式只能取值“0”或者“1”。布尔成功地将逻辑归结为一种代数演算，即所谓的逻辑代数(或称布尔代数)。命题逻辑是逻辑代数的一个特例。本书不准备全面介绍逻辑代数以及命题逻辑，只对程序设计所涉及的逻辑表达式及其求值方法的问题进行讨论。

在逻辑代数中，只有两个常量：逻辑真(TRUE)和逻辑假(FALSE)，每一个逻辑常量、逻辑变量以及逻辑表达式也只能取这两个值之一，即不是逻辑真就是逻辑假。在程序设计语言中，一般用 0 值表示逻辑假，用 1 值(或任一非 0 值)表示逻辑真。

最基本的逻辑运算符(逻辑连接符)有三个：

1) 逻辑非 NOT

2) 逻辑与 AND

3) 逻辑或 OR

如果 a 是一个逻辑量，NOT a 表示取这个逻辑量当前值的相反的值。即如果 a 的值是逻辑真(非零值)，NOT a 表示逻辑假(零值)；反之、如果 a 的值是逻辑假(零值)，NOT a 表示逻辑真(非零值)。逻辑非(NOT) 的运算规律可以用逻辑非真值表(表 1.2) 来表示。

表 1.2 逻辑非真值表

操作数	NOT 操作数
0	1
1	0

逻辑与 AND 是对两个逻辑量进行的操作，在计算机中将以两个操作数作为操作对象的操作称为双目操作，相应的运算符称为双目运算符。如果 a,b 是两个逻辑量，a AND b 表示将这两个逻辑量进行逻辑与操作，操作的结果得到一个逻辑值。随 a 和 b 各自值的不同有四种组合，只有当 a 和 b 的值是都是逻辑真(非零值)时，a AND b 的值才是逻辑真(非零值)，其他三种组合的运算结果均为逻辑假(零值)。逻辑与(AND) 的运算规律可以用逻辑与真值表(表 1.3)来表示。

表 1.3 逻辑与真值表

第一操作数	AND	第二操作数	结果
1		1	1
1		0	0
0		1	0
0		0	0

逻辑或 OR 也是对两个逻辑量进行操作的双目操作。如果 a，b 是两个逻辑量，a OR b 表示将这两个逻辑量进行逻辑或操作，操作的结果得到一个逻辑值。随 a 和 b 各自值的不同也有四种组合，只有当 a 和 b 的值是都是逻辑假(零值)时，a OR b 的值才是逻辑假(零值)，其他三种组合的运算结果均为逻辑真(非零值)。逻辑或(OR)的运算规律可以用逻辑或真值表(表 1.4)来表示。

表 1.4 逻辑或真值表

第一操作数	OR	第二操作数	结果
1		1	1
1		0	1
0		1	1
0		0	0

逻辑常量和逻辑变量是逻辑表达式的基本组成部分，简单逻辑表达式用逻辑运算符组合起来能组成复杂的逻辑表达式。在逻辑运算中，三个逻辑运算符的优先级不同，其中逻辑非(NOT) 的优先级最高，逻辑与(AND) 次之，逻辑或(OR) 最低。复杂逻辑表达式的运算根据逻辑运算符优先级的高低按从左到右的次序进行，用括号可以强制运算的次序。任何一个逻辑表达式的结果只能取两个逻辑值之一，即不是逻辑真就是逻辑假。

例如 lvar1，lvar2，lvar3 是三个逻辑变量，它们的值分别是 1，0，1，用逻辑运算符和括号构成下列逻辑表达式：

NOT lvar1 AND (lvar2 OR lvar3)

结果为逻辑值 0。

如果 A 和 B 是两个逻辑表达式，对逻辑表达式 A AND B 以及 A OR B 求值时，可能有两种不同的方法。最简单的方法是对两个逻辑表达式分别求值，然后根据逻辑运算符 AND 或 OR 的运算法则求整个表达式的值，这种方法的运算效率不是最高的。例如对 A OR B 求值时，如果已经求出逻辑表达式 A 的值是逻辑真，那么无论逻辑表达式 B 的值是逻辑真或逻辑假，A OR B 的值一定是逻辑真。所以在程序设计语言中往往采取一种更有效的求值方法，用前面介绍的选择结构可以表述成：(其中 EVALUATE 表示求值过程)

对 A OR B 求值

```
IF (EVALUATE(A))
  result ← TRUE
```

```
    ELSE
      result ← EVALUATE(B)
对 A AND B 求值
    IF (EVALUATE(A))
      result ← EVALUATE(B)
    ELSE
      result ← FALSE
```

在复杂的逻辑表达式求值时，往往还会用到逻辑代数中的德.摩根(De Morgan)定律：

NOT (A OR B) = NOT A AND NOT B

NOT (A AND B) = NOT A OR NOT B

1.3 程序设计语言

1. 程序设计语言的发展

算法是问题求解过程的描述，由执行者(人或计算机)对算法的执行称为计算。为使计算机能够执行算法必须将算法以计算机能够接受和处理的方法表示出来。程序设计语言是描述算法的手段同时也是计算机能够接受和处理的算法表示方法，用程序设计语言将算法表示出来并交给计算机去执行的过程称为程序设计。计算机的发展导致了程序设计语言的发展和程序设计方法的研究。

当第一台电子数字计算机 ENIAC(Electronic Numerical Integrator And Computer) 在 1945 年哇哇堕地的时候，并不存在现代意义上的程序设计语言。ENIAC 存储容量只有 20 个字长为 10 位的十进制数，它的程序是“外插型”的，即是用线路连接的方式实现的。为由计算一个问题转为计算另一个问题，必须在线路板上花上几天的时间重新连结线路。

在 ENIAC 紧张设计的同时，本世纪最伟大的应用数学家之一 John Von Neumann(冯.诺依曼)对电子数字计算机发生了兴趣并投身到设计者行列，冯.诺依曼与 ENIAC 的研究组紧密合作提出了一种全新的存储程序通用电子数字计算机方案 EDVAC(Electronic Discrete Variable Automatic Computer)，这就是人们目前通称的冯.诺依曼结构。

EDVAC 方案明确规定了新机器有五个构成部分：计算器、逻辑控制装置、存储器、输入部件及输出部件，并描述了这五个部分的职能和相互关系。与 ENIAC 相比，EDVAC 方案有两个非常重大的改进，一是为充分发挥电子元件的作用而采用二进制，二是提出了“存储程序”的概念，就是将程序的指令用代码的形式输入到计算机的存储器中，用记忆数据的同一装置存储执行运算的指令，一条程序指令完成后自动执行下一条程序指令，用“条件转移”指令自动确定作业的顺序。存储程序和程序控制的概念是计算机史上的一个里程碑，它使得全部运算成为真正的自动过程。尽管从 EDVAC 以来计算机已经发生了天翻地复的变化，但存储程序和程序控制的概念仍然是当今计算机设计和操作的基本准则。

最原始的程序设计语言称为机器语言。根据存储程序的概念，程序指令也作为数字存储程序在计算机中，所以程序指令也是二进制形式的，而且程序指令必然和计算机的型号(或系列)相关的。设一台计算机采用两字节的指令系统，每条指令包括操作码和操作数两

部分，取数指令是二进制 1010 0000，加法指令是二进制 0000 0100，存数指令是二进制 1010 0010，停止指令是 1111 0100。要实现两个数 7 和 10 相加的运算 7 +10，若 7 已存放在存储器中，相加以后的和要放回到存储器中去，每一个常量和变量都在内存的一个确定地址中。实现上述要求的机器语言的程序段为：

地址	M	说 明
10H	1010 0000	将第一个操作数送入累加器 AL 中
11H	0001 0111	第一个操作数的地址是 17H（十六进制）
12H	0000 0100	将累加器 AL 中的数与第二个操作数相加
13H	0000 1010	第二个操作数 10（十进制）
14H	1010 0010	将相加的结果送回到存储器
15H	0001 1000	运算结果的地址（十六进制的 18H）
16H	1111 0100	停止
17H	0000 0111	第一个操作数 7(十进制)
18H		

机器语言程序设计要求程序员用数字代码和地址编写程序，这就必然要考虑许多与机器有关的细节，而且机器语言的程序都是二进制代码，程序的阅读和修改相当麻烦，迫使人们去设计程序设计的更直观更简单的方法。

首先的改进程序设计语言的一个思路是，既然字符能通过 ASCII 码等编码方法在计算机内部用二进制代码表示，那么程序的指令码就可以用英语的单词(或单词的简写) 来表示，再由计算机将字符表示的程序指令转化为对应的二进制代码。按照这个思路在 1950 年代早期出现了符号机器语言，称为汇编语言(Assembly Language)。

汇编语言用指令的英语助记符代替了机器语言中的操作码，地址和数据除用二进制、八进制或十六进制表示外也可以用符号名表示。为了把这种符号程序翻译成等价的机器语言程序，需要一个称为汇编程序的翻译程序。汇编程序与用汇编语言编制的程序是两个不同的概念，汇编程序是计算机系统程序的一个组成部分，其作用是将用户用汇编语言编制的程序翻译成(或称汇编成)机器可以执行的机器语言代码。汇编指令格式一般是 [指令助记符] [操作数]。例如传送指令的指令助记符是 MOV，加法指令的指令助记符是 ADD，停止指令的指令助记符是 HLT，加法运算 7+10 的汇编程序段写成(其中 M1 和 M2 是存储单元地址)。

```
MOV AL, [M1]
ADD AL, 0AH
MOV [M2], AL
HLT
```

由于汇编语言的代码和机器语言指令存在一一对应的关系，所以采用汇编语言来编制程序同样是一件繁琐的工作，需要有极大的耐心和高度集中的注意力，程序员要涉及许多硬件的细节，例如寄存器和存储器地址分配等等。这些与算法的实现往往很少相关，所以无论是机器语言还是汇编语言都是面向机器的语言。

1950 年代中期，出现了面向问题的高级程序设计语言，高级语言的问世是程序设计语言方面的真正的突破。