

目 录

第 1 章 程序设计基础知识	1
1.1 计算机简介	1
1.2 程序设计语言	5
1.3 算法的描述	7
1.4 计算机软件	11
习题 1	13
第 2 章 FORTRAN90 基础知识	14
2.1 FORTRAN 语言的发展	14
2.2 源程序及其构成	15
2.3 语言元素	18
2.4 程序单元概念	30
习题 2	33
第 3 章 基本语句	34
3.1 类型说明语句	34
3.2 算术表达式和赋值语句	37
3.3 输入与输出语句	45
3.4 输入与输出编辑符	50
习题 3	55
第 4 章 选择结构程序设计	57
4.1 关系表达式与逻辑表达式	57
4.2 IF 语句	61
4.3 IF 结构	62
4.4 CASE 结构	67
4.5 程序举例	70
习题 4	73
第 5 章 循环结构程序设计	75
5.1 引言	75
5.2 无循环变量的 DO 结构	76
5.3 带循环变量的 DO 结构	82
5.4 DO WHILE 结构	88
5.5 DO 结构嵌套	90
5.6 隐含 DO 循环	94
5.7 程序举例	95
习题 5	100

第 6 章 程序单元和过程	103
6.1 概述	103
6.2 函数子程序	104
6.3 子例子程序	108
6.4 虚实结合	109
6.5 子程序的嵌套调用	112
6.6 模块	116
6.7 递归过程	121
6.8 其它部分	126
6.9 程序举例	132
习题 6	137
第 7 章 数组	139
7.1 一维数组	139
7.2 多维数组	148
7.3 数组运算	153
7.4 数组作为过程变元	158
7.5 程序举例	166
习题 7	172
第 8 章 字符型数据	175
8.1 字符型数据基础	175
8.2 字符型数据的输入与输出	178
8.3 字符型数组	180
8.4 用于字符处理的内在函数	182
8.5 字符型数据作为过程的变元及函数值	184
8.6 程序举例	187
习题 8	190
第 9 章 派生类型和指针结构	192
9.1 数据结构与派生类型	192
9.2 指针	200
9.3 链表	206
9.4 二叉树	217
9.5 指针与数组	222
9.6 指针与过程	225
习题 9	229
第 10 章 数据文件	231
10.1 文件概述	231
10.2 对文件的基本操作	232
10.3 顺序文件的存取	236
10.4 直接文件的存取	239
10.5 INQUIRE 语句	241

10.6 无格式文件	243
习题 10	245
第 11 章 FORTRAN 77 的过时特性	246
11.1 过时的书写格式	246
11.2 过时的控制转移方式	247
11.3 数据说明的过时形式	249
11.4 程序中的过时功能	250
11.5 其它过时语句	252
附录 I ASCII 字符集	255
附录 II FORTRAN90 的语法描述	256
附录 III FORTRAN90 的内在过程	264
主要参考文献	269

第 1 章 程序设计基础知识

本书主要讨论用高级语言 FORTRAN90 编写程序的若干问题。由于程序运行的载体是电子计算机,这就需要先了解有关计算机组成等常识,同时也要了解用计算机解题的步骤以及计算机软硬件的简单知识,为以后使用 FORTRAN90 进行程序设计打下基础。

1.1 计算机简介

世界上第一台计算机于 1946 年诞生于美国。在半个世纪的发展过程中,计算机已经历了巨大的变化。按其电子器件的发展来划分,计算机已经换了四代:第一代电子管计算机,第二代晶体管计算机,第三代中小规模集成电路计算机,第四代大规模和超大规模集成电路计算机。近十几年来,人们一直致力于研究第五代计算机。在计算机刚刚诞生的二三十年内,其运算速度每 5~8 年就提高 10 倍,而体积和成本却降低为 1/10。计算机的发展非常迅速,特别是微型计算机问世以来,更加快了计算机的普及范围与速度,也导致了学习计算机浪潮的到来。

1.1.1 计算机的应用与特点

世界上第一台计算机(名叫 ENIAC)最初主要是出于战争的需要,用于计算火炮弹道,属于科学计算范畴。随着计算机硬件和软件的发展,计算机已从单一的科学计算领域,很快发展到国民经济的各个部门乃至家庭。可以说,世界上任何一个领域都已使用或可以使用计算机,计算机的应用范围是十分广阔的。但为了便于了解计算机的用途,可将计算机的应用领域概括成五个主要方面:科学计算、信息处理、实时控制、计算机辅助设计和人工智能。

1. 科学计算

科学计算又称数值计算,早期的计算机主要用于科学计算。例如工程设计、天气预报、地震预测、卫星轨道计算、炮弹弹道计算等等,这些都属于科学计算范畴。1948 年,美国原子能研究中有一项计划,要做 900 万道运算,需要 1500 名工程师计算一年。当时利用了一台初期的计算机,仅用了 150 小时就完成了。这大大地加快了科学研究的速度,使科学家从繁杂的数值计算中解放出来。

另外,由于计算机的出现,导致计算天文学、计算化学、计算生物学、计算医学等学科的产生与发展,使许多古老的学科焕发了青春。

2. 信息处理

信息处理又称数据处理。由于计算机的普及,管理工作也实现了电脑化。人们将用于管理的庞大的程序和数据存入计算机中,管理人员通过终端就可以进行查询、统计、打表等各项管理工作,计算机还可以对数据进行分析 and 汇总,供管理人员分析决策时使用。

人类已经进入了信息时代,其标志之一就是用计算机存储和处理各种信息,运行在计算机上的各种管理软件已在发挥着巨大的作用。例如企业管理系统、情报检索系统、学生成绩

管理系统等等。由于计算机网络系统的发展也十分迅速,这些信息处理软件也可在网络环境中运行,使办公系统实现自动化。

3. 实时控制

实时控制又称过程控制。计算机在机械加工、自动生产线的控制、整个生产过程乃至整个工厂的控制方面有广泛的应用,在化工、石油、造纸、钢铁等企业已广泛地使用了计算机。现代交通控制、高射炮瞄准、飞行器飞行等也都需要使用计算机控制技术。

在生产流程控制中,可将各个环节的测量、调整装置与计算机相连,计算机可以根据事先编好的程序对生产过程进行自动控制。这不但可以提高控制的精度,也可以节省人力。例如某年产 500 万吨钢的钢厂需要职工 15000 人,利用计算机自动化生产后仅需 4000 人。

4. 计算机辅助设计(CAD)

用计算机辅助人们进行设计工作,如设计飞机、房屋、汽车、桥梁等,这可以提高设计速度和节省人力。

以飞机设计为例,人工设计飞机的速度慢,成功率也较低。因为仅仅依靠设计者的经验和头脑很难对多个设计方案进行比较,加上飞机零部件极多,哪一方面出现问题都可能导致设计的失败。利用 CAD 技术,输入一个原始方案后,计算机首先计算出它的性能,然后调整各个设计参数,优化设计方案。经过这样的多次反复过程,选出最佳方案。这显然比人工设计效率高得多。

随着 CAD 技术的发展与应用范围的扩大,又派生出许多新技术分支,如 CAM(计算机辅助制造)、CAI(计算机辅助教学)等。

5. 人工智能

主要研究用计算机来“模仿”人的智能,让计算机具有“推理”和“学习”的能力。例如,人机博弈系统、专家系统、机器人等。

计算机之所以有如上诸多的用途,是因为它具有以下特点:

1. **运算速度快。**ENIAC 每秒运算的次数为 5000 次,而巨型机的运算次数已达每秒几十亿次以上。普通计算机也达每秒几百万次。这种运算速度是传统的计算工具无法相比的。

2. **精确度高。**一般计算机能提供十几位以上的有效数字。

3. **具有“记忆”和逻辑判断能力。**它能把计算步骤、原始数据、运算结果存储在计算机内,它还能进行逻辑判断,并根据判断结果自动决定以后执行哪个命令。

4. **执行步骤由程序控制。**计算机内部操作能按预先编好的程序步骤执行,不需要人工干预。

综上所述,可以认为电子计算机是一个以高速进行操作、具有内部存储能力、由程序控制操作过程的自动电子装置。电子计算机简称计算机,俗称电脑。

1.1.2 计算机的硬件组成

现在的计算机均属于冯·诺依曼型计算机,它具有两个主要特点。一是采用“存储程序”的工作原理,即在计算机算题之前,预先编好程序存入计算机存储器,计算机自动按这些程序工作。二是采用二进制编码,指令和数据均以二进制形式存储,这种编码方式能简化计算机的结构。

计算机系统由硬件系统和软件系统两部分组成。

计算机硬件是指其机械部分和电子部分,由运算器、控制器、存储器、输入设备和输出设备五大部分组成(见图 1-1,带箭头的实线表示数据流,带箭头的虚线表示控制命令,虚线框中间的是主机),其中输入设备和输出设备合称输入输出设备,因此也可以说计算机由四部分组成。

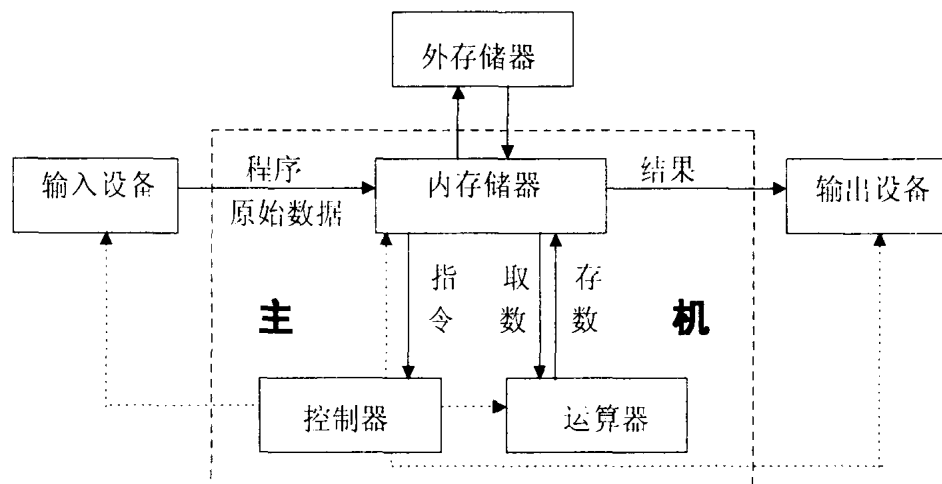
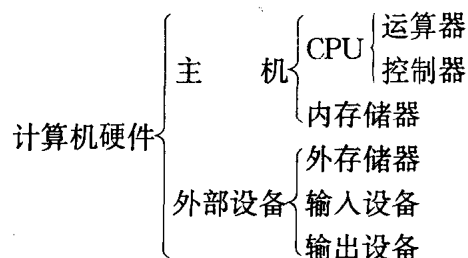


图 1.1 计算机硬件的基本结构

存储器又分成内存储器(或称主存储器)和外存储器(或称辅存储器)。运算器和控制器合称中央处理机(简称 CPU),CPU 与内存储器又合称主机。外存储器、输入设备、输出设备合称外部设备。因而也可把计算机看作是由主机和外部设备组成:



下边简单介绍计算机的五个部件。

1. 运算器

它主要进行算术运算和逻辑运算。运算器的主要技术指标是机器字长和运算速度。机器字越长,能表示的数的范围也就越大,有效数字的位数就越多。运算速度可用每秒中加法指令运算次数、加法和乘法等指令平均运算次数、主频等多种方法表示。

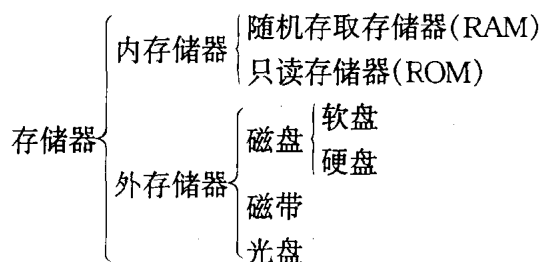
2. 控制器

控制器是分析和执行指令的部件,它是统一指挥和控制计算机各个部件按时序协调操作的中心枢纽。

3. 存储器

它用于存放各种信息。例如从输入设备送入的指令程序和原始数据,从运算器中得到的中间结果等。

存储器有多种分类方法,常见的分类方法如下:



内存储器在主机之内,它与 CPU 之间有信息传递的直接通道,这样内存储器可以随时与 CPU 交换数据。外存储器不能直接与 CPU 交换数据,因为它们之间没有信息传递的直接通道。CPU 若想使用外存储器中的数据,必须利用内存储器作为中间媒介。CPU 与内存储器交换速度快,但内存容量较小;CPU 与外存储器交换速度慢,但外存容量很大。因而只有最常用的信息才放在内存中,其它信息可先放在外存上,需要时必须先换到内存中。

信息存储的最小单位是二进制位,8 个二进制位为 1 字节(Byte,用 B 表示),常用的存储容量单位还有千字节(KB)、兆字节(MB)和千兆字节(GB)等。其中 $1\text{KB} = 2^{10}\text{B} = 1024\text{B}$, $1\text{MB} = 2^{10}\text{KB} = 1024\text{KB}$, $1\text{GB} = 2^{10}\text{MB} = 1024\text{MB}$ 。

4. 输入设备

把信息(程序和数据)送入计算机的部件。常用的输入设备有键盘、鼠标、数字化仪、光笔、摄像机等。

5. 输出设备

将计算机中的信息输出的部件。常用的输出设备有显示器、打印机、绘图仪等。

外存储器属于输入输出型部件,它既可以将信息送入计算机,也可以从计算机中取出信息。常用的外存有磁盘机、磁带机等。

1.1.3 微型计算机

如果将计算机按照价格和运行速度等分型,可分成巨型机、大型机、中型机、小型机和微型机(简称微机)。由于微机价格便宜,又能满足企事业等单位的一般需求,其普及面最广。

微型机是由微处理器、存储器、输入/输出接口和系统总线等组成的计算机。由于微机内有许多输入/输出接口,用户可根据需要配置相应的外部设备,再配上适当的软件。就构成了相应的微型计算机系统。

微型机可分成单片机、单板机和个人计算机等几类。目前国内流行的微机是 IBM-PC 系列机,它由 IBM 公司推出。实际上绝大多数用户使用的都是 IBM-PC 系列的兼容机。IBM-PC 机也是由五种部件构成,但根据它们的组装情况,该机硬件可分为主机箱、键盘和显示器三部分。

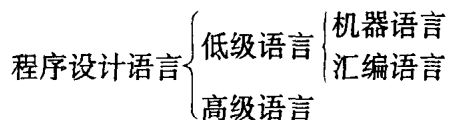
主机箱中除 CPU 和存储器以外,还有配合 CPU 工作的协处理器,在箱中的系统板上还有输入/输出控制部分、总线及扩展槽等。键盘是终端的输入设备,它是由标准的英文打字机键盘演变而来的。显示器有单色和彩色显示器之分,目前一般都使用分辨率较高的彩色显示器。

IBM-PC 机自 80 年代初问世以来发展极快,硬件和软件都在不断地发展中。现在的高档 586 微机无论其运算速度、内外存容量及所携带的外设等都具有相当的规模,软件配备也在日新月异,其适用性很强,在生产实践中使用很广。

1.2 程序设计语言

计算机系统中如果只有硬件还不能被一般用户所使用,必须配有相应的计算机软件。对于科技工作者来说,最常用的软件就是程序设计语言系统。程序设计语言的主要用途是用来编写程序。当人们希望计算机为自己做一些事情时,必须把要完成的这些工作逐项分解,用程序的形式告诉计算机每一步应该怎样去做。因而人们使用程序设计语言的目的是为了产生程序,即产生计算机能识别的操作步骤。

程序设计语言可按如下方式分类:



在上一节中我们说过计算机的控制器是分析和执行指令的部件,而指令就是计算机所能做的操作的最小单位,例如一次加法运算或一次数据传递都可以用一条指令完成。在设计和制造硬件时,就已经决定了该机能识别和执行哪些指令。一台计算机所能识别的一组指令的集合称为该机的指令系统。显然,指令系统中的指令越多,其处理能力越强,用起来越方便。但指令系统太大会使硬件线路复杂化,从而导致计算机价格昂贵。

由于计算机中的信息全用二进制形式表示,其指令和数据也应是二进制形式。早期的计算机仅能识别这种由一条条二进制指令所编写的程序。而这种完全用 0 和 1 二进制代码来书写指令,且这些代码不须翻译就可直接被机器所接受的语言就称机器语言。机器语言书写的程序占用内存空间小,执行速度快,能充分利用硬件资源且可直接被计算机识别。但由于它必须使用绝对地址,程序出错时极难修改,易读性和通用性极差,很难被一般人员所接受。

为了弥补机器语言的弱点,人们用一些比较直观、易记、易写的助记符代替其中的二进制序列,如用 ADD 表示加法指令操作符,用 MOV 表示传递指令操作符,这样机器指令中的操作符、操作数及地址等均用助记符来代替,从而形成了汇编语言。例如 IBM-PC 机 8088 汇编中的一条加法指令可写为:

ADD FIRST, 200H

它表示把存储器中名字为 FIRST 的那个单元之中的数据取出,加上十六进制数 200H(转换成二进制数为 100000000)之后,再存入 FIRST 单元之中。这里的 FIRST 是一个助记符,表示内存中某一个存储单元,可以由用户在使用它之前定义。ADD 也是一个助记符,表示两个数相加的操作符。

汇编语言采用助记符以后,程序的易读性及易修改性比机器语言有了很大的改善,但增加了一件工作:必须将用汇编语言编写的源程序翻译成用机器语言写的二进制代码才能被计算机执行。做这件翻译工作的是汇编程序(见图 1-2)。

无论是机器语言还是汇编语言,它们都具有一个特点:面向机器。所谓面向机器是指其语言与具体的机器有关,换一种机型就不可使用,即其缺乏通用性。例如,我们用 8088 汇编语言编写的程序只能在 IBM-PC 系列机上使用,放到 VAX 机或其它型号的机器上均不能使用。我们把这种与机器有关的语言称作低级语言。机器语言和汇编语言都是低级语言。

由于低级语言与具体的机器关系密切,使用它的人必须了解机器许多内部结构,如寄存器有哪些,它们的名字及功能等,这就使得对计算机内部结构了解不多的人无法有效地使用低级语言。为了解决这一问题,使那些对计算机内部构造并不熟悉的绝大多数人都能编写

计算机程序,人们又发明了高级语言。

高级语言是面向问题计算过程的程序设计语言,它非常接近于人们的习惯,易于人们理解。高级语言源程序不再以指令为组成单位,一般它都是由语句序列组成,而高级语言中的一条语句往往要对应多条指令。

高级语言的主要特点是其独立于具体的计算机,使用方便,通用性强,易于记忆和修改,用户学习高级语言时并不需要学习计算机的内部结构。高级语言的出现,大大地加快了计算机的普及速度,也增加了它的应用范围。

用高级语言编写的源程序必须经过编译(图 1-3)或解释(图 1-4)之后才能被计算机执行。编译程序是把源程序全部编译成用机器语言书写的程序。这种方法的优点是程序执行速度快,一次编译可多次执行。其缺点是源程序修改后必须重新编译,而编译要占据相应的机时。解释程序是逐句逐段地翻译源程序,一边翻译,一边执行。其优点是源程序的修改和扩充较方便,但解释方法工作速度慢,程序被重复执行时也要重复解释。

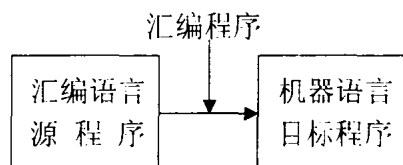


图 1-2 汇编程序工作图

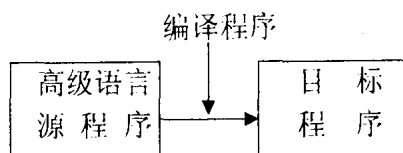


图 1-3 编译程序工作图

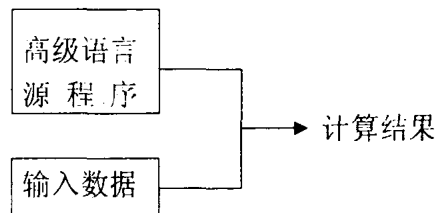


图 1-4 解释程序工作图

国内外使用的高级语言达数百种之多,比较流行的几种如下:

FORTRAN——世界上出现最早的高级语言,主要适用于科学计算。

BASIC——易学易用的语言,适用于初学者。最近几年出现的新版 BASIC 语言可用于科学计算、数据处理及绘图等,其功能大大地增强了。

PASCAL——最早出现的结构化语言,由于其结构严谨、数据类型丰富而最适用于计算机专业教学。

C——近些年最受欢迎的语言之一,由于用它设计出著名的 UNIX 操作系统而闻名。它主要适用于系统软件设计。近些年由于 Borland 等公司将它配上集成环境和引入面向对象的技术而扩大了应用范围。

LISP——是函数式语言,其程序和数据的表示形式是等价的,用递归程序设计作为基础,主要用于人工智能等领域。

PROLOG——以逻辑程序设计为基础,属于描述性语言,主要用于人工智能及专家系统等领域。

其它流行的高级语言还有 COBOL、ADA、ALGOL、PL/1 等。每一种高级语言都在不断地发展着。如 FORTRAN 语言在国内曾流行的版本就有 FORTRAN66、FORTRAN77 等。本书主要介绍 FORTRAN 语言的最新标准——FORTRAN90 的使用方法。

人们使用程序设计语言的主要目的是为了编写出反映人们特定意图的程序,但不能把

程序设计单纯地理解成编程序。实际上,编程序只是程序设计若干步骤中的一步,我们为了编写出合乎要求的程序,要经过分析问题、建立数学模型、设计算法和画出流程图、编制程序、上机调试、维护等多个阶段。

分析问题是对所求问题进行分析,弄清什么是已知条件,要解决什么问题,最后得到什么样的结果等。为了解答未知问题,我们需要对它进行数学表示,从而找出相应的数学模型。由于程序设计是用计算机解决实际问题,因此我们需要在已知数学模型的基础上,提出相应的计算机化的算法。这种算法可以用自然语言表示,也可以用流程图类的图示描述。完成了以上各步以后,才能根据算法编写出相应的程序,经过上机调试之后得到正确的程序投入使用。程序使用中还需要做相应的维护工作。

在初学程序设计语言阶段,由于例题及习题都比较简单,分析问题和建立数学模型等步骤都非常简单,甚至是一目了然的。人们往往不注重这些步骤,也不习惯用流程图描述算法之后再编程,实际上这样做是得不偿失的。随着要解决的问题的复杂化,如果不进行相应的算法描述,编程时会非常困难。因此,在程序设计中,不能忽视程序设计的各个中间步骤。

随着要解决的问题的增大,软件编程量也会增大,其产生的代码会大幅度地增长。这样,程序中出现错误的可能性也增大。在实际的软件开发中,大程序中的错误很难避免,往往会出现软件开发进度比预计拖延,需要的经费不断上涨等问题,从而导致软件危机。为了解决这种危机,人们提出了开发软件的软件工程法。

软件工程是指导计算机软件开发和维护的工程化方法。它采用工程的概念、原理、技术和方法来开发和维护软件,能把经过时间考验而证明正确的管理技术和当前能够得到的最好的技术方法结合起来。软件工程把软件生存周期分成软件定义、软件开发和软件维护三个时期,而软件定义又分成问题定义、可行性研究和需求分析三个阶段,软件开发又分成一般设计、详细设计、编码和单元测试、综合测试四个阶段。软件工程法是目前开发大型软件的有效方法。

1.3 算法的描述

用计算机解决实际问题的主要工作之一是算法的描述。我们从算法的基本概念出发,引出算法的图示描述法。

1.3.1 算法概述

人们每做一件事情都会有具体的工作步骤与方法,这种具体的工作步骤或解题方法就是算法。也可以说,算法就是为解决某一特定问题而采用的具体工作步骤或方法。由于我们讨论的算法最终要反映成计算机程序,因此这里所说的算法实际上是指计算机算法,即算法中的每步都应能被计算机处理,此外还要充分运用计算机本身的特性。

[例 1-1] 设计 $S = \sum_{i=1}^4 a_i$ 的算法。

算法一:

在内存中设一累加用的存储单元 S。

第 1 步:将存储单元 S 清零。

- 第2步:输入第1个数 a_1 。
- 第3步:把 a_1 加到存储单元 S 中。
- 第4步:输入第2个数 a_2 。
- 第5步:把 a_2 加到存储单元 S 中。
- 第6步:输入第3个数 a_3 。
- 第7步:把 a_3 加到存储单元 S 中。
- 第8步:输入第4个数 a_4 。
- 第9步:把 a_4 加到存储单元 S 中。
- 第10步:把存储单元 S 中的结果输出。

这种算法是正确的,但它并未充分利用计算机解题方法的特点。如果按算法一方式,4个数相加需要10个步骤完成,以后每增加一个加数要增加两个步骤。由此推论:10个数相加需要22步,100个数相加需要202步。当加数增多时,程序长度会不断膨胀。1.1节中曾指出计算机的特点之一是其具有逻辑判断能力,我们可以利用这一特性将算法改写如下:

算法二:

在内存中设一累加的存储单元 S 和一计数用的单元 I。

- 第1步:将累加单元 S 清零。
- 第2步:将计数单元 I 清零。
- 第3步:输入一个数 A。
- 第4步:把 A 加到累加单元 S 中。
- 第5步:计数单元 I 的值增加1,即 $I+1 \Rightarrow I$ 。
- 第6步:若 $I < 4$ 则转去执行第3步,否则,继续执行第7步。
- 第7步:输出存储单元 S 中的结果。

算法二就是计算机程序设计时常用的算法。假设要做100个数相加时,只需把第6步中的“ $I < 4$ ”改成“ $I < 100$ ”,算法步骤并不增加。即100个数相加的算法步骤与4个数相加的算法步骤是相同的(但其执行次数并不相同)。算法二是一种比较通用的算法,其第6步使用了计算机中具有“逻辑判断后转移”的功能,使其输入数据 A 并把 A 加到 S 中这几步骤重复使用,从而简化了算法的步骤,使算法的描述更具有实用性。

除了算法中各步应能被计算机准确执行这一特点之外,算法中往往还有若干个输入量和若干个输出量,这些输入量就是程序的原始或中间输入数据;这些输出量就是程序的计算结果,它表示问题得到解答结果或问题未得到解答。算法另一个重要特性就是算法的有穷性,即一个算法应在有穷步之内结束。假如把上述算法二中第6步改成:转去执行第3步。这样算法每到第6步就回到第3步,第7步无法执行到,算法将无法终止。这种算法是不符合要求的。

1.3.2 算法图示表示法

以上介绍了用自然语言文字表示算法的方法。实际上算法的最终表示是程序设计语言表示法,即用程序设计语言书写的程序来表示算法,这是我们要做的主要工作。由于程序设计语言是一种形式化语言,比较抽象难懂,其规定又过细,在算法设计中直接编出程序尚有相当的困难。用自然语言表示算法十分适合人们的习惯,但书写起来极为不方便,算法表

示(如程序流向)不直观。现在用得较多的是伪码表示法和图表示法,伪码是一种介于自然语言和程序设计语言之间的伪码语言。在高级语言教材中一般均使用图示法。

图示法中用得较多的是程序流程图(又称程序框图)和 N-S 图(又称 N-S 结构化流程图或盒图)。

1. 程序流程图

程序流程图一般采用以下具有特定含义的流线和图框

来表示算法(见图 1-5,为了使用方便,各基本符号均带有流程线):

- (1)箭头(流程线):表示流程路线。
- (2)椭圆框(起止框):表示流程的起点或终点。
- (3)方框(处理框):表示一般的处理或运算。它有一个入口,一个出口。
- (4)菱形框(判断框):表示逻辑判断。它有一个入口,两个出口,根据逻辑判断选择一个出口。
- (5)平行四边形框(输入输出框):表示计算机输入或输出数据。只有一个入口,一个出口。
- (6)圆框(连接点):表示将两个流程中各自的某一点连接起来。当一个流程图在一页纸内画不下时可以用“连接点”表示从本页某一点(出口点)连接到下页某一点(入口点)。

例 1-1 算法的流程图如图 1-6 所示。

2. 结构化程序设计与 N-S 图

为了降低程序和程序设计的复杂性,提高程序的易读性,以便在适当的时间内得到一个结构良好、易于理解的程序, E. W. Dijkstra 等人提出了结构化程序设计思想。由于对结构化程序设计众说纷纭,尚没有一个被人们普遍接受的定义。其中一个较为流行的说法是:结构化程序设计是一种程序设计技术,它采用自顶向下逐步求精的设计方法和单入口单出口的控制结构。

结构化程序设计主要有以下几个特点:

- (1)自顶向下逐步求精的程序设计方法符合人类解决复杂问题的普遍规律。因为它采用先全局后局部、先整体后抽象、先抽象后具体的逐步求精过程,提高了软件开发的成功率。
- (2)不使用 GOTO 语句,仅使用单入口单出口的控制结构(基本控制结构有三种),使程序的静态结构和它的动态执行情况比较一致。开发时比较容易保证程序的正确性,出错时

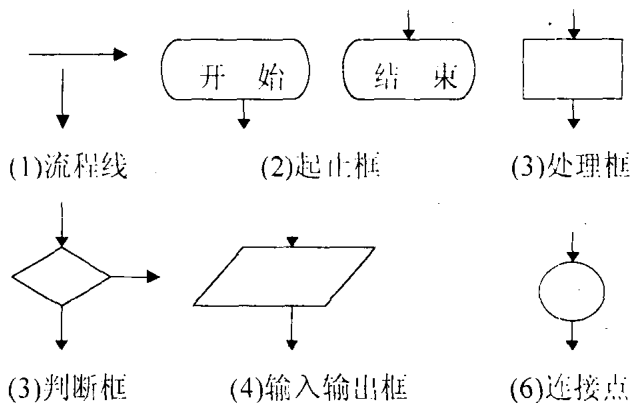


图 1-5 流程图基本符号

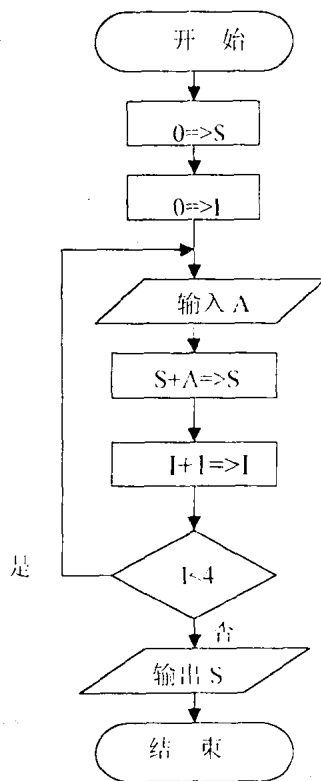


图 1-6 例 1-1 的流程图

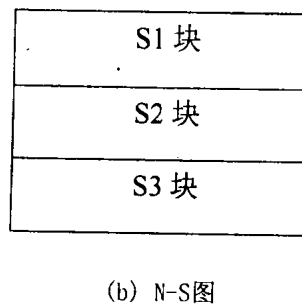
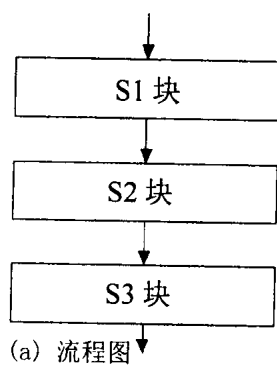


图 1-7 顺序结构

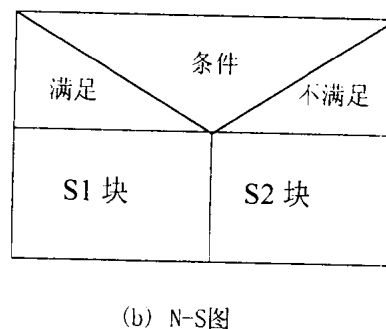
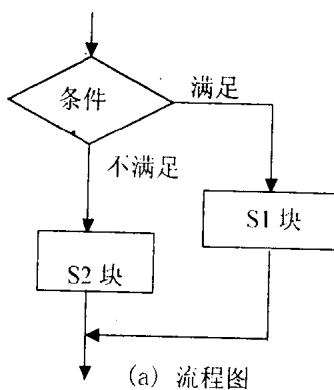


图 1-8 选择结构

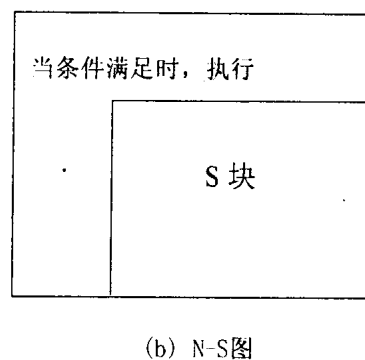
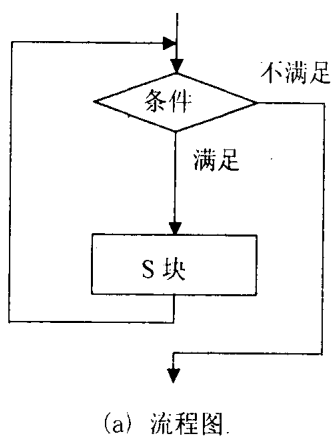


图 1-9 循环结构:“当型循环”

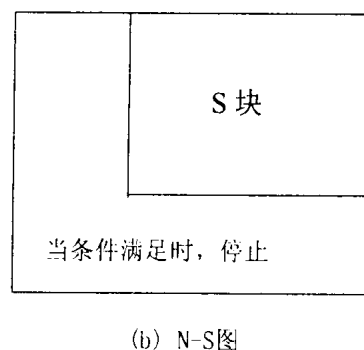
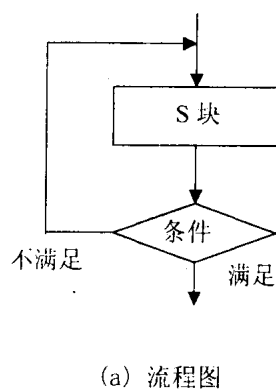


图 1-10 循环结构:“直到型循环”

也容易纠正。

(3)把程序的清晰性看作第一位,效率、存储空间等都在次要位置,由于程序的逻辑结构清晰,有利于程序正确性证明。

三种控制结构分别是顺序结构、选择结构和循环结构。它们突出了单入单出性,即每种结构无论其内部构造如何,它们与外界的联系仅靠一个入口和一个出口,这样万一程序出现错误时,通过这种结构的划分比较容易找到错误所在。修改某一个结构内部时,其它结构不必做无谓的变动。

这三种控制结构可以用程序流程图描述,也可以用N-S图描述(见图1-7至图1-10)。

例1-1算法二的N-S图用传统的程序流程图描述算法时,在算法比较复杂的情况下,若不小心就会绘出非结构化的流程结构,使算法的编程实现出现困难、甚至错误。I. Nassi 和 B. Shneiderman 提出的盒式图(称N-S)既能明确描述作用域、容易实现嵌套关系,也无法出现任何转移控制,这就使N-S图的结构化特性非常明显,在结构化程序设计中得天独厚。

例1-1的算法二用N-S图描述时如图1-11所示。

把这三种基本结构复合之后,可形成许多较复杂的结构化控制结构,如多种条件选择结构等等。

1.3.3 算法描述举例

[例1-2] 分别用流程图和N-S图,描述求下式的算法:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^{n-1}}{(n-1)!}$$

本例的流程图见图1-12,N-S图见图1-13。

通过本例的学习,就可以对简单循环结构的算法有一个初步的了解。熟悉和掌握算法的图形表示,对于以后研究复杂结构的算法设计,进而编写出结构优良、易读性强、能解决实际问题的程序是十分有意义的。

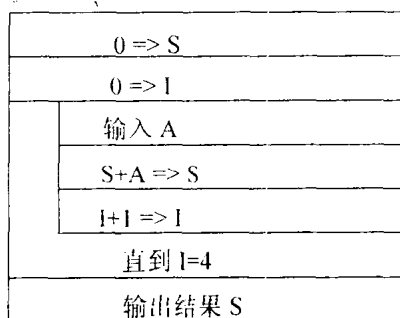


图1-11 例1-1算法二的N-S图

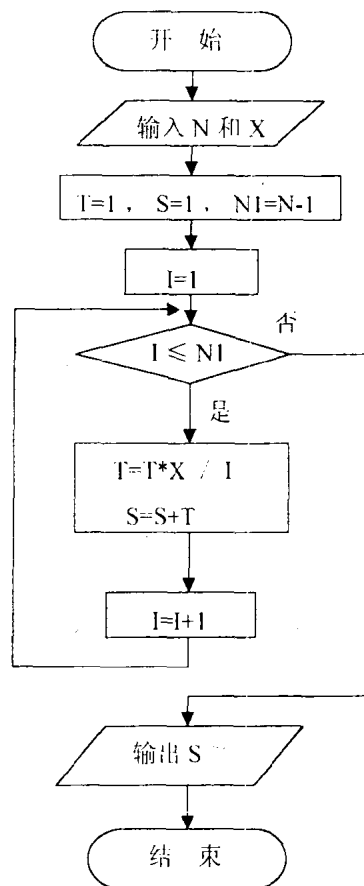


图1-12 例1-2的流程图

1.4 计算机软件

我们使用计算机一般是指使用计算机系统,计算机系统由硬件系统和软件系统组成。

硬件系统指计算机的五个组成部件:运算器、控制器、存储器、输入设备和输出设备。显然,没有计算机硬件就谈不上计算机。但只有硬件的计算机是十分不好用的,一般用户根本无法使用它,只有计算机专家中的极少部分人才会使用。为了让计算机适合绝大多数用户,给计算机系统配备了许多软件,这些软件不但对硬件功能进行了扩充,而且为用户提供了十分友好的使用界面。我们实际上是通过这些软件来使用计算机硬件资源的。

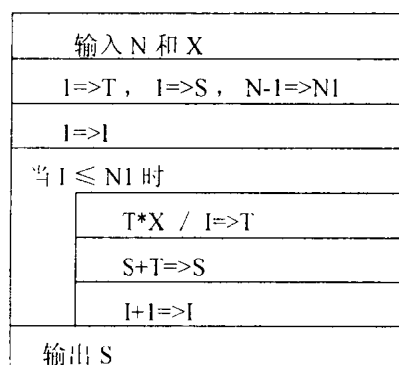


图 1-13 例 1-2 的 N-S 图

软件是指计算机系统中各种程序及其有关文档资料的总称。软件可分成系统软件与应用软件两大类。

系统软件(又称系统程序)是指为其它软件服务的软件,它可以使用和管理系统中各种软、硬件资源。例如微机上的磁盘操作系统 MS-DOS 就属于系统软件,凡是在该系统下应用的其它软件必须在 DOS 管理下运行。系统软件一般有三种:面向用户的软件、面向计算机维护人员的软件和面向计算机本身的软件。各种语言类编译或解释程序、用户修改源程序的编辑程序等都属于面向用户的软件,它们主要为用户提供用程序设计语言编写自己的源程序,并进而形成可执行程序的各种支持工具、库程序、软件包等。在一种高级语言加工程序中,或者提供用于编译方式的编译程序,或者提供用于解释方式的解释程序,有的系统中同时提供该种语言的编译系统和解释系统。

系统软件的数量与质量是衡量计算机系统性能的重要指标之一。系统软件数量越多、功能越强、效率越高、使用越方便,越能吸引更多的计算机用户,购买该计算机的用户也就越多。因此从某种意义上来说,系统软件的优劣决定着硬件系统销售的命运,系统软件中最重要的一种是操作系统。

应用软件是为了解决某些具体问题而编写的各种程序及其文档资料的总称,应用软件又称应用程序。应用软件具有专用性,例如会计帐务处理程序、人事档案管理程序、工厂生产数据管理程序等等。应用软件可以分成两类。一类是不分业务、行业的基本上可以通用的应用软件,例如 CAD 软件、科技计算软件包等等。这类软件虽然有某种特定的用途,但不同行业的人又可共用它。如微机上的 AutoCAD 系统,机械设计人员可用它绘制各种机械零件图,化工技术人员可用它绘制地面管线图,而石油测井人员又可用它绘制各种测井曲线图。该软件虽然满足了不同行业技术人员的需要,但它又把用户局限到图形处理这一领域,故划归应用软件的第一类。应用软件中另一类是按业务划分的软件,如学生学籍管理、医院医疗管理、交通控制系统等等。

系统软件和应用软件的划分有时并不是绝对的,对某些软件来讲很难找到一个严格的界限。例如各种标准程序库等,可以看作是应用软件(应用软件第一类),也可以看作是计算机厂家提供的系统软件,因为用户稍加修改或者是完全不修改就能将它们用到自己编写的程序中。另外中文文字处理软件 WPS、CCED 等,既可以把它划归应用软件第一类,更多的教材中已把它们当作系统软件处理。对于使用计算机的人员来讲,熟悉软件分类是为了

熟悉各类软件的功能与目的,更好地发挥它们的作用。

习 题 1

- 1-1 计算机的特点是什么?
 - 1-2 前四代计算机是如何划分的?
 - 1-3 计算机的主要应用领域有那些? 其特点是什么?
 - 1-4 电子计算机的五个组成部分及其主要功能是什么?
 - 1-5 画出计算机硬件基本结构图。
 - 1-6 内存与外存的区别是什么? 常见的外存设备有哪几种?
 - 1-7 为什么计算机中要用二进制形式表示?
 - 1-8 用于表示存储大小的 B、KB、MB 和 GB 的意义分别是什么? 它们之间的关系是如何换算的?
 - 1-9 什么是高级语言? 什么是低级语言? 高级语言一定比低级语言好吗? 为什么?
 - 1-10 解释汇编程序、编译程序和解释程序。
 - 1-11 什么是系统软件,什么是应用软件,它们之间的划分是否是绝对的?
 - 1-12 用自然语言描述求两个数中最大值的算法。
 - 1-13 分别用流程图和 N-S 图描述求 $n!$ 的算法。
 - 1-14 用流程图描述求下述分数序列前 10 项之和的算法:
$$\frac{2}{1}, \frac{3}{2}, \frac{5}{3}, \frac{8}{5}, \frac{13}{8}, \frac{21}{13}, \dots$$
- 要求用“当型循环”和“直到型循环”分别实现。
- 1-15 用 N-S 图描述习题 1-14 的算法(也用两种循环方法实现)。

第2章 FORTRAN90 基础知识

上一章中介绍了计算机的应用与特点、计算机的组成以及计算机算法等知识。实际上,算法的具体实现一般要借助于一种语言,研制出计算机能识别并可执行的程序,经调试、运行得到所需要的结果。本书涉及的高级语言是 FORTRAN 语言,本章先介绍有关 FORTRAN90 的基础知识。

2.1 FORTRAN 语言的发展

若想让计算机按人们的意愿解决问题、处理问题,需要进行相应的程序设计,这就需要学习程序设计语言。世界上最早出现并得到广泛应用的高级语言是 FORTRAN 语言。

FORTRAN 是 FORmula TRANslation 的缩写,意为“公式翻译”,最初是为数值计算而设计的。第一个 FORTRAN 语言文本是 1954 年提出的,当时由于种种原因,并未马上得到公众的认同,直到 1956 年才开始真正使用。随后经不断发展,形成了多种版本。

1958 年出现了 FORTRAN II。1962 年出现了 FORTRAN IV。

美国国家标准化协会 ANSI(American National Standards Institute)对于 FORTRAN 语言标准的形成起到了主要的作用。1966 年,ANSI 以 FORTRAN IV 为基础制定了美国标准文本,即 ANSI 1966 FORTRAN,简称 FORTRAN66。1972 年,国际标准化组织 ISO(International Standard Organization)宣布,将 FORTRAN66 作为 ISO 当时的 FORTRAN 标准文本。

1976 年,ANSI 对 FORTRAN IV 提出了修订文本,经广泛征求意见,于 1978 年推出了新的标准 ANSI X3.9-1978 FORTRAN,即人们常说的 FORTRAN77。

1991 年,ANSI X3.198-1991 FORTRAN 问世,其国际标准文件号是 ISO/IEC 1539:1991,我国国家标准是 GB/T 3057-1996,人们俗称为 FORTRAN90。

在 FORTRAN90 中,除保持了 FORTRAN77 的全部优点外,又加进了许多具有现代特性的功能,为 FORTRAN 语言注入了新的活力。它主要体现在以下几个方面:

1. FORTRAN90 的结构特性更好,时代性更强,它新增了许多具有现代特性的语句与项目,程序的易读性与可维护性更好。

2. 它的算法功能比 FORTRAN77 又有了更大的加强。仅以数组为例,引进了数组直接运算、数组直接赋值、数组与标量的运算等概念,大大地简化了程序的编写。

3. 在子程序调用中,设计了许多先进手段,如关键字参数调用、可选参数调用、类属过程调用等,这增强了子程序的功能。此外,还增加了递归调用的功能。

4. FORTRAN90 增加了运算符超载、赋值号超载功能,扩展了 FORTRAN 语言的操作功能。

5. 在 FORTRAN90 中,信息管理功能得到了空前的提高,它引进了字符种别的概念,使 FORTRAN 能给各国文字(包括汉字)、各种学术符号(如化学符号、数学符号等)赋以不同的种别参数,从而可以对之进行运算。另外,FORTRAN90 还按数据结构需要,设置派生类

型、指针以及模块等,可以很方便地操作各种数据结构。

6. FORTRAN90 的语法结构更为严谨规范,从高层语法到底层语法,都有统一的结构,使编程和调用都很规范,也提高了程序的可读性。

7. FORTRAN90 在发展 FORTRAN77 的同时,也屏弃了一些过时的特性。FORTRAN77 一行分为四个区的书写格式,给编程带来许多的不便,在 FORTRAN90 中不再采用该格式;还有一些程序功能或由于其作用已过时,或由于在 FORTRAN90 中已有其它方法来代替,因此不再提倡。这些内容主要有:语句函数、假定长数组说明、RETURN 语句、DATA 语句、EQUIVALENCE 语句、COMMON 语句、GOTO 语句、PAUSE 语句、STOP 语句等。有关 FORTRAN77 的过时特性将在第 11 章中具体介绍。

2.2 源程序及其构成

2.2.1 几个简单的 FORTRAN90 程序

下面通过几个简单例子来了解一下 FORTRAN90 程序,以便初步掌握有关 FORTRAN 语言的知识。

[例 2-1] 已知华氏温度与摄氏温度之间的换算公式为:

$$T_c = \frac{5}{9}(T_h - 32)$$

现输入某一华氏温度 T_h , 请计算出相应的摄氏 T_c 。

```
PROGRAM H_ TO_ C
```

```
! Given the Fahrenheit temperature, to calculate the Centigrade
```

```
REAL::TC, TH
```

```
READ *, TH
```

```
TC = 5 * (TH - 32) / 9
```

```
PRINT *, TH, TC
```

```
END PROGRAM H_ TO_ C
```

下面对该程序进行简单的说明:

第一行是主程序语句,标志主程序单元开始,PROGRAM 是关键字,意为“程序”,关键字后空一格写主程序名。程序名最好起与主程序完成的功能相关的名字,以便于记忆和阅读。

第二行以“!”开始为注释行,它只起注释作用,对程序运行结果没有影响。本程序中注释行的意思为“给定华氏温度,计算摄氏温度”。

第三行为说明部分,在本程序当中,说明部分只有一个语句。其中 REAL 是关键字,意为“实型”,::称为双隔号,由两个“:”连成,:后的 TH 和 TC 被说明为实型变量,各变量间用逗号“,”分隔开。

第四行是读语句,READ 为关键字,* 是表控格式,后面跟着变量。当程序执行到该语句时,将等待用户从键盘输入一个实型数据。

第五行是赋值语句,FORTRAN90 中只有赋值语句没有关键字。有关这方面的知识在下一章详细介绍。

第六行是输出语句,将 TH、TC 的值按表控格式输出。

第七行是主程序单元结束语句,END PROGRAM 是关键字,空一格后写本程序名,此名必须与程序开始处主程序语句中的主程序名一致。若不写程序名,也可以正确执行,但最好写上程序名,以使程序清晰、易读性好。

[例 2-2] 输入 M 个整数,将其相加,打印出它们的和。

```
PROGRAM CAL_M_SUM
  IMPLICIT NONE
  INTEGER::N,M
  REAL::T,A
  N=0;T=0
  PRINT *, 'Input number of data:'
  READ *, M
  DO
    READ *, A
    T=T+A
    N=N+1
    IF(N>=M)EXIT
  END DO
  PRINT *, T
END PROGRAM CAL_M_SUM
```

本程序与例 2-1 中的程序相比主要多了 DO-END DO 结构,这是一个不带循环变量的 DO 结构(称循环结构)。它表示要重复执行 DO-END DO 之间的语句序列,直到 N 大于或等于 M 时为止。EXIT 表示跳出循环。在本程序的第四行,将两个语句写在一行上,但中间应使用分号“;”分隔。另外,程序中第二行的语句 IMPLICIT NONE 表示不使用隐式说明,这有利于程序的易读性和可维护性。FORTRAN90 的每一个程序单元中一般均有此句。

[例 2-3] 求 $\sum_{i=4}^8 i!$

该问题涉及五个阶乘的相加,可用子程序实现求阶乘。

```
FUNCTION FACTOR(N) RESULT(FAC_RESULT)
  IMPLICIT NONE
  INTEGER::N,FAC_RESULT,I
  FAC_RESULT=1
  DO I=1,N
    FAC_RESULT=FAC_RESULT*I
  END DO
END FUNCTION FACTOR
PROGRAM CAL_FACTOR
  IMPLICIT NONE
```

```

    INTEGER::FACTOR,S=0,I
    DO I=4,8
        S=S+FACTOR(I)
    END DO
    PRINT *, 'S=',S
END PROGRAM CAL_FACTOR

```

本程序由两部分组成:一个主程序(名为 CAL_FACTOR),另一个是函数子程序(名为 FACTOR)。

子程序第一行的 FUNCTION 是关键字,表明为函数子程序;FACTOR 是任取的函数子程序名;RESULT 也是关键字,表示函数的结果;FAC_RESULT 是任取的变量名,存放函数结果的值,即函数值。经过 DO 结构运算后,FAC_RESULT 中存放的值恰好是 N 的阶乘值,即函数值。

主程序的第五行中的 FACTOR(I)是调用函数子程序,将主程序中变量 I 的值传给子程序 N,即当 I 是 4 时,N 得到的值也是 4,在子程序中计算出 4 的阶乘,将结果存放在 FAC_RESULT 中传回主程序。主程序 DO 结构中的 S=S+FACTOR(I)语句要执行 5 次,每次执行时 I 值是不同的,这可以求出 4 至 8 的阶乘之和。

主程序、子程序中第二行的作用是一样的,即声明在本程序单位中,不使用隐式说明。

通过以上几个程序的学习,对于 FORTRAN90 程序可以有一些简单的了解,具体的编程方法在后续的章节中会详细介绍。

2.2.2 FORTRAN90 程序的构成

通过分析例 2-1 至例 2-3,可以了解 FORTRAN90 程序的构成及程序中包含的主要成分。

1. FORTRAN90 程序是一种分块形式的程序,整个程序由若干个程序单元块组成,各单元都有相似的语句组织形式,其中主程序单元起整体控制作用,各子程序单元各自完成问题的一个子算法。编程时,先把求解的问题分解为若干相对独立的子算法,每个子算法编写成独立的子程序单元。每个程序只有一个主程序单元,程序的执行从主程序单元开始,依次调用各子程序单元,形成程序的整体运行。

按现代编程要求,应尽量多使用子程序单元,这样有利于程序的维护。

2. 无论是主程序单元,还是子程序单元,都是独立的程序单位,应该独立编写,它们的形式相似。例如,主程序单元和函数子程序单元的一般形式为:

主程序单元	函数子程序单元
PROGRAM 主程序名	FUNCTION 函数名(参数表) RESULT(结果变量)
程序体	程序体
END PROGRAM 主程序名	END FUNCTION 函数名

3. 各程序单元的程序体形式上相同,都由若干行组成。FORTRAN90 中的程序行主要包括:

(1)说明语句行。主要用于说明变量的类型、属性等。例如:

```

INTEGER::X,Y           ! 说明 X,Y 为整型变量

```


REAL,DIMENSION(1:10)::A,B !说明 A,B 都是具有 10 个元素的一维实型数组。

(2)可执行语句行。主要有输入语句、赋值语句、输出语句等。一般一个程序行只书写一个可执行语句。FORTRAN90 允许在一行中书写多个语句,但每个语句间应该用“;”分隔。例如:

A=4; B=B+1; D=0

是一个合法的 FORTRAN90 程序行。

(3)注释行。如果某个程序行是以“!”号开始,则“!”号以后的部分被认为是注释内容,它只起到对读程序人员的提示作用,而对程序的执行结果不起任何作用。另外,在某语句的后面出现“!”号,那么“!”号后面的内容也是起注释作用(! 出现在字符串中时例外)。

除此而外,某程序行的第一列的字符是“C”或“*”时,该行也被认作注释行。但在 FORTRAN90 中不提倡这种使用方法。

2.3 语言元素

FORTRAN90 的语言元素是指在程序设计中用到的基本成分,例如字符集、常量、变量、记号以及其它的基本结构等。只有了解并掌握了这些基本要素,才可能正确自如地使用该种语言。

2.3.1 FORTRAN90 字符集

在 FORTRAN90 程序中,允许使用的字符包括字母、数字、下划线及一些规定的专用字符。具体如下:

1. 26 个英文字母

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

在大多数处理系统中允许使用小写字母,除了字符串内的字符外,小写字母等价于相应的大写字母。

2. 10 个阿拉伯数字

0 1 2 3 4 5 6 7 8 9

3. 下划线

_ (注意与“-”是不同的)

下划线可以作为有效字符出现在名字中。

4. 21 个特殊字符(见表 2-1)

特殊字符用于运算符符号、括号、分隔和定界其它词法记号等各种形式。特殊字符 \$ 和? 没有规定用法。

5. 其它字符

具体的处理系统字符集中还可以有附加字符,但只可以出现在字符常量、字符串编辑描述符、注释和输入/输出记录中。本书一般不涉及这类字符。

在 FORTRAN90 中,大小写是等价的。也就是说编译系统会自动将大、小写字母

认为具有相同的意义(除了在字符型常量、字符串编辑描述符、注释和输入/输出记录中外)。本书中,关键字、变量及语法结构中用到的字母,一般均采用大写方式。

表 2-1 特殊字符

字符	字符名	字符	字符名
	空格	:	冒号
=	等号	!	叹号
+	加号	"	引号或双撇号
-	减号	%	百分号
*	星号	&	英语的 and
/	斜杠	;	分号
(左圆括号	<	小于
)	右圆括号	>	大于
,	逗号	?	问号
.	小数点或句号	\$	货币号
'	撇号		

2.3.2 记号

在 FORTRAN90 中,将字母、数字或专用字符的基本有效的序列称为记号,它包括标号、关键字、名字、常数、运算符和定界符。例如:

$A * \sin(X) + 5.3$

这个表达式包括 6 个记号,即 A、*、sin、X、+、5.3。

在字符串内部,不能随意地添加空格。例如,字符串“ABC”与字符串“A B C”是不一样的,前者有 3 个字符,后者有 5 个字符。另外,在记号内部,也不许随意地使用空格。例如, sin(X)不能写成 S I N(X),后者是错误的书写形式。但一般来讲,记号与记号之间的空格可以随意地使用。例如, $A * \sin(X) + 5.3$ 与 $A * \sin(X) + 5.3$ 在语法上是相当的。在这种上下文中,多个空格在语法上与一个空格的作用相同。

一个记号与相邻关键字、名字、常数或标号之间,应当用一个或多个空格分开,以避免错误,同时也使结构清晰。例如:

```
IMPLICIT NONE
END DO
PROGRAM MAIN
END INTERFACE
```

2.3.3 基本类型常量

常量是程序执行过程中不能变化的量。FORTRAN90 中的常量的类型分为基本类型和派生类型两种。基本类型是 FORTRAN 语言本身包含的数据类型。在每一基本类型中,依据不同的处理系统又细分为不同的种别,每一种别都与一个整数相对应,此整数则称为类型种别参数。这是标识和区别所用的各种种别的一种方法。

基本数据类型有五种:整型、实型、复型、字符型和逻辑型。前三种属于数值类型,后两

种为非数值类型,主要用于文字处理和控制。

1. 整型常量

整型常量又叫整数,整数值集合是数学整数的子集。一个处理系统一般提供一种或几种表示方法,此方法定义整型数据值的集合。每种方法是用称为种别类型参数的类型参数值来表征的。一种表示方法的种别类型参数是由内在询问函数 `KIND` 送回的。种别值中提供了给定范围,其提供的最小范围是由内在函数 `SELECTED_INT_KIND` 送回的。表示十进制幂的范围是由内在函数 `RANGE` 送回的。

缺省种别是指有符号或无符号的整数值。下列 6 个数都是合法的整数:

```
0          +456          -123
34_2       34_SHORT      12345678998765432_8
```

在下划线后面的量就是种别参数。其中, `SHORT` 应是一个符号常数,其值必须是整型的有效种别类型参数。没有下划线的数都具有缺省种别。缺省种别整数的范围一般取决于所用计算机的字长。对于 n 位(二进制位)字长的计算机而言,其数据表示的范围一般为 $-2^{n-1} \sim +2^{n-1} - 1$ 。例如对于 32 位机,缺省种别整数的表示范围应该是 $-2^{31} \sim +2^{31} - 1$ 。

为了保证计算机内数的范围够用,就要通过给定种别参数值来约束,同时最好将该种别参数值用符号常数来代表。例如,若要求数的范围在 -999999 到 $+999999$,可通过语句把 `K6` 作为具有适当值的常数而建立起来:

```
INTEGER,PARAMETER::K6=SELECTED_INT_KIND(6)
```

这样, `K6` 被说明成符号常数,可以将它作为种别常数来使用。例如, -123456_K6 、 $+10_K6$ 、 235_K6 。

`SELECTED_INT_KIND(6)` 是内在查询函数的调用,它返回范围在 $-10^6 \sim +10^6$ 之间(不包括两端点值)的所有整数的种别值。这样说明了种别参数后,提高了程序的可移植性。

在编程时,最好都使用种别值,因为它指出一个值所占据的存储单元的字节数。在 `FORTRAN90` 标准中,其用法比较灵活。例如,一个处理系统可能只有 4 字节整数的硬件,而用这种硬件可以支持种别值为 1、2 和 4 的整数(使硬件具有 1、2、4 字节整数的处理系统,放宽了可移植性)。新标准只是指出对于处理系统上的任意两个种别值 K 和 L ,如果 $K < L$,则种别 K 所能表示的数的范围就小于或等于种别 L 所能表示的数的范围。种别值不能是负的。

在一个给定的处理系统中,一给定数据的种别参数值可由 `KIND` 内在函数得到。

`KIND(0)` 为缺省值,其返回值是整型的标准种别参数

`KIND(2_K6)` 为上例说明的种别参数

在 `FORTRAN90` 中,也可以表示二进制、八进制、十六进制形式的无符号整数,其表示形式如下:

二进制数: `B'101101'` 或 `B"101101"`

八进制数: `O'76210'` 或 `O"76210"`

十六进制数: `Z'1FA2'` 或 `Z"1FA2"`

其中, `B`、`O`、`Z` 分别是二进制、八进制、十六进制的标志,其后所跟的数应放在一对单撇号或一对双撇号(引号)中。

整型的类型说明符是关键字 INTEGER。

2. 实型常量

实型常量又叫实数,它具有数学实数的近似值。处理系统必须提供两种或更多种的近似方法,这些方法定义实型数据值的集合。每种方法具有一种表示形式,此形式用称为种别类型参数的值来表示。近似方法的种别类型参数是由内在询问函数 KIND 送回的。在该种别值中,提供给定的精度和给定的幂的范围,提供的最小十进制精度由内在函数 SELECTED_REAL_KIND 送回。

实数最常见的两种表示形式是小数形式和指数形式。

所谓小数形式是指由十个数字、小数点及数符组成的数。如:0.45、-24.0、.4242、1234.56、1.、12.0 等。

而对于一个绝对值相对较小或很大的实数,常用指数形式表示,但它与数学上的表示有所不同。因为在一个 FORTRAN 程序行中,所有内容都只能写在一条水平线上,不能写成上标或下标的形式,因此用字母 E 代替了底数 10。例如:数学上的指数可表示成 $-1.3E35$ 。

指数形式表示的实数由数字部分和指数部分组成,数字部分可以是小数或整数。以字母“E”为分界线,其前的部分为数字部分,其后的部分是指数部分。指数部分由 1~3 位整数及数符组成。例如:

5.23443	E	-22	385	E	6
数字		指数	数字		指数
部分		部分	部分		部分

另外,12E7、0.45E15、-2.97E-10 等均是实数有效的指数表示形式。

一个实数可以表示为小数形式,也可以表示成指数形式。例如,在一个程序中写 123.5、1.235E2、0.1235E3 或 12350E-2 等,作用是相同的。但在计算机输出一个数值时,将按一种“规格化的指数形式”输出。常用的有两种“规格化的指数形式”:

(1) 数字部分的值小于 1,而且小数点后的第一位数字不等于 0,例如 0.743643E-12, 0.9876E22。而 21.835E-12, 0.00023E12 就不属于规格化的指数形式。

(2) 数字部分有一位(而且只能有一位)非零的整数,即小数点前仅有一个非零的数字。例如 7.123E24, -9.345E-12。而 0.2834E-34, 345.2E7 则不属于这种规格化的指数形式。

一种计算机系统只采用上述两种“规格化的指数形式”中的一种。

一般实数的表示范围大约是 $-10^{38} \sim +10^{38}$,有效位为 7~9 位。但也有的系统允许实数的范围大致在 $-10^{75} \sim +10^{75}$,因而一般处理系统指数最多为两位数字。有的计算机系统允许数的范围较大,可以超过 10^{100} ,因此,指数可以是三位数字。所以,应该了解所用计算机系统的规定。

有时为了能得到理想的范围和有效数字,要求说明种别类型参数值。例如:

```
REAL,PARAMETER::LONG=SELECTED_REAL_KIND(8,88)
```

如此说明的符号常数 LONG 提供了至少 8 位的精度,以及 $-10^{88} \sim +10^{88}$ 的值的范围。那么,以此说明的实型常量

```
4.37_LONG
```

```
457.12345E55_LONG
```

就可以在 LONG 指定的精度及数值范围内进行移植。

对于任意的一个实数,可用以下两个询问函数来测定它的精度及范围:

PRECISION(4.37 LONG) 测得值至少是 8,表示至少八位有效数字

RANGE(4.37 LONG) 测得值至少是 88,表示数的范围至少为 $-10^{88} \sim +10^{88}$ 。

实型的类型说明符是关键字 REAL。

3. 复型常数

复型常数又叫复数,它具有数学复数的近似值。在科学和工程计算中,复数被广泛地用到。复数是实数的有序对,将两个实数中间用逗号分隔,然后再放在一对括号中就构成了一个 FORTRAN 语言的复数。例如:

(1.25,0.4)

(1E5,38)

(1.0-5,4.2)

其中,第一个实数称为复数的实部,第二个实数称为复数的虚部。在 FORTRAN 语言中,复数的实部及虚部也可由整数构成。例如:

(3-2,5.0)

(4,7)

(15-4,24-8)

等均是合法的复数。但对于不同的构成方式,最终的复数种别类型是不一样的,具体有以下几种组合:

(1)若实部和虚部都是实数时,则该复数的种别类型参数值是具有较大十进制精度那部分的种别类型参数值;若精度相同,则它是由处理系统确定的两部分之一的种别类型参数值;若一部分具有与复数不同的种别类型参数值,则该部分转换到复数的那种类型,即由低精度向高精度转化。

(2)若实部和虚部均为有符号整数时,则复数的精度和范围就是缺省,且该常数被默认为复型的。

(3)若实部和虚部其中之一是整数,则复数的种别就是另一部分的种别,且将该整数的种别转化到与另一部分相近的种别类型参数值。

对于复数,KIND、PRECISION 和 RANGE 这几个函数仍可以使用。

复型的类型说明符是关键字 COMPLEX。

4. 字符型常量

字符型常量又叫字符串或字符常数,其缺省种别由一对单撇号或一对双撇号之间的字符序列组成。例如:

"ABC D"

'CHINA'

'4653374'

这里所用的字符序列并不仅仅局限于前面提到的 FORTRAN 字符集中的字符,处理系统可以接受和识别的任何字符都是允许的,但一些控制字符不要放在其中,以免出现不易预料的结果。在字符串内大小写字母均可使用,但系统对它们按不同的字符进行识别。单撇

号和双撇号只起定界作用,它们不是字符串的组成部分。

当在某字符串内部具有单撇号时,例如要将 I'm a student 作为一个字符串来处理,为了区分“'”是字符串内容还是定界符,可采用两种方式,即

"I'm a student" 或

'I' 'm a student'

前者用双撇号作定界符,后者用单撇号作定界符,而将字符串的单撇号用两个单撇号表示,系统会自动处理其为字符串的一个单撇号。

对于含有一对撇号的字符串,也可采用类似的方法。

'That was a "story"' 其值为 That was a "story"

"This sounds 'theta'" 其值为 This sounds 'theta'

在字符串内部的空格是有效字符,并占有一个字符的位置。

字符串内字符的个数称为字符串的长度。字符串 "" 和 "" 的长度为 0。

处理系统支持相应字符集的非默认字符常量的例子是:

CYRILLIC_

MAGYAR_

NIHONGO_

HINDI_

HANZI_

其中,CYRILLIC、MAGYAR、NIHONGO、HINDI 和 HANZI 是有名常量,其值分别是对应 Cyrillic(俄语或其它斯拉夫语言)、Magyar(匈牙利语)、Nihongo(日语)、Hindi(印地语)和 Hanzi(汉语)字符的种别类型参数。注意 Hanzi 和 Nihongo 的字库非常大(表意文字符号字符集),而 Cyrillic、Hindi 和 Magyar 是字母字符,字库并不太大。没有这些字库的处理系统是不能使用这些字符集中的字符的。

一般处理系统缺省的字符种别参数是 KIND('A')返回值。若使用的字符集限于字母、数字、下划线和特殊字符,则国际上的可移植性是可保证的。

若使用非标准种别字符时,应将其种别参数放在字符常数的前面,以便于简化编译程序分析语句的工作。例如,某处理系统可以用种别参数值 1 来支持汉字字符,在这种情况下,一个汉字字符常数可被书写成:

1_ "五笔字型输入法" 或

HANZI_ "五笔字型输入法"

一个字符常数需要写成多行源程序时,有一条特殊的规则:不仅每一续行都不能尾随注释,而且每一续行必须以续行标识符(&)作为开头,任何尾随 & 号之后或者前导 & 号之前的空格都不是字符常数的组成部分,& 号本身不是常数的组成部分。其它的字符包括空格都是字符常数的有效组成部分。例如:

STRING= 'On the day I visited the school, an exam was held. &

& The boys and girls on the shooting range, preparing &

& for the exam, were standing or lying on their stomachs.'

在任何一台机器的解释系统中,对字符都有特定的顺序序列。对于 FORTRAN90,要求计算机对字符的排序序列满足下列条件,即字符的序号值应具备的条件:

- (1)对于 26 个大写英文字母 A<B<...<Z
- (2)对于 10 个数字 0<1<...<9
- (3)空格<A<...<Z<0 或
空格<0<...<9<A
- (4)若处理系统支持小写字母,则 a<b<...<z
- (5)若处理系统支持小写字母,则
空格<a<...<z<0 或
空格<0<...<9<a

从以上规则可见,没有关于数字是否在字母之前或之后的规则,也没有关于专用字符或下划线位置的规则,只有空格在部分序列中占前的规则。在实际处理系统中,字符的排序是按其码值进行的,在 IBM-PC 机中一般按 ASCII 方式排序。

字符型的类型说明符是关键字 CHARACTER。

5. 逻辑型常量

逻辑型常量只有真(.TRUE.)和假(.FALSE.)两个值。

对于逻辑型数据,处理系统必须提供一种或几种表示方法。每种方法用称为种别类型参数的值来表示。表示方法的种别类型参数由内在询问函数 KIND 送回。

逻辑常量通常只用来给逻辑型变量置初值,以及得到某一逻辑表达式的值。

缺省种别具有与处理系统有关的种别参数值。实际有效值可通过 KIND(.TRUE.)得到。至于其它固有类型,种别参数值可以用一个跟在下划线后的整型常数指出。例如:

.FALSE._2

.TRUE._SHORT

逻辑型的类型说明符是关键字 LOGICAL。

2.3.4 名字

在 FORTRAN90 中使用的名字被用于标识一个程序成分,如程序单元、有名变量、有名常量、虚元或派生类型等。

对于名字有下列要求:

- (1)名字的长度不能超过 31 个字符。
- (2)名字的组成成分可以是字母、数字和下划线。
- (3)名字的第一个字符必须是字母。
- (4)要特别注意,在名字中不能出现空格。

在 FORTRAN90 中,对名字没有其它限制,也没有保留字。例如,下面的名字都是合法有效的:

A

NAME_LENGTH (单下划线)

S_P_R_E_A_D_O_U_T (两个连续的下划线)

WAITER_ (尾部下划线)

B123

INTEGE

Y5

而下列名字是无效的:

1A (第一个字符不是字母)

THIS NAME (含有一个空格)

A * BC \$ (含有非法字符 * 和 \$)

在 FORTRAN90 中,提倡使用具有一定含义的名字,以提高程序的清晰度及可读性,这样也可以帮助记忆。例如,用 PRIME 代表素数,用 FACTOR 代替阶乘等。

2.3.5 变量

变量是指在程序运行过程中其值可以发生变化的量。每个变量是以变量名来代表的。变量名是名字的一种,其命名规则应满足 2.3.4 节中提到的要求,且提倡使用具有一定意义的变量名。例如:

INTEGRAL_RESULT (积分结果)

NUMBER_OF_DAYS (天数)

FIRST (第一个)

NEXT (下一个)

有些常用符号如 θ 、 α 、 β 等无法直接书写,可将其读音或英文名字写出。如写成 THE-TA、ALPHA、BETA 等。

FORTRAN90 变量的基本类型与常量一样也有五种,即整型变量、实型变量、复型变量、逻辑型变量和字符型变量。

在 FORTRAN90 中,每个变量在使用前都应该在说明部分中说明其类型、属性、种别等,不提倡在 FORTRAN77 中常用的隐式说明。为了抑制隐式说明发生作用,应该在程序说明部分一开始就写出下一语句:

IMPLICIT NONE

以此向系统声明在本程序单位内,隐式说明不起作用。

FORTRAN90 中的五种变量的类型说明关键字是:

说明整型变量的关键字:INTEGER

说明实型变量的关键字:REAL

说明复型变量的关键字:COMPLEX

说明逻辑型变量的关键字:LOGICAL

说明字符型变量的关键字:CHARACTER

说明变量的格式一般是将关键字放在该句的最前面,待说明的变量放在后面,中间用双分隔符::隔开,对于同时说明多个变量时,各变量间用“,”分隔。如:

INTEGER::I,J,K

REAL::X,Y,Z

用这两条说明语句分别说明 I、J、K 为整型变量,可以用来存放整型数据,X、Y、Z 为实型变量,可以用来存放实型数据。

在 FORTRAN90 中,在变量的说明部分有了许多新的功能,例如:

(1)在变量说明的同时,可以给变量置初值:

```
INTEGER::I=5,J=126
```

```
REAL::X=7.2,Y=48.257,Z,W=774.2
```

(2)在说明变量类型的同时,也可以说明其种别:

```
REAL(KIND=2)::X,Y
```

这样,不仅说明 X、Y 是实型变量,并且说明其种别参数是 2。

种别是 FORTRAN90 的新概念,即一个数据,不仅有一个类型,而且在同一类型下可再细分成若干种别。这一方面可以提高数据存储效率,节约内存,同时也有助于程序的移植。

(3)在说明变量的同时,还可以说明变量的属性。属性是被说明对象的所属性质。一个对象被说明具有某一属性时,就使该对象具有某种附加功能、特殊的使用方式与适用范围。属性的种类有很多,有关属性的内容将在以后相应的章节中分别介绍。这里先介绍两种最常见属性的说明格式。其一是:

```
INTEGER,PARAMETER::I=5,J=24
```

PARAMETER 是符号常数属性,经这样说明后,在本程序单位内,I 就相当于整数 5,J 就相当于整数 24。它们的作用与整型常数的作用一样,I、J 的值不能被修改。其它类型的符号常数也采用相同的格式说明。第二种属性说明是:

```
REAL,DIMENSION(1:10)::A
```

DIMENSION 是数组属性,经这样说明后,在本程序范围内,A 代表一个实型一维数组,它的下标以 1 为下界,10 为上界,共有 10 个元素。

2.3.6 派生数据类型

前面介绍了 FORTRAN90 有五种基本数据类型:整型、实型、复型、逻辑型及字符型。编程时,可以根据需要而定义新的数据类型,这就是派生类型。在一个派生类型中,可以含有多种基本类型,使之可表现的内容更丰富。

例如,在一个学校内,每个学生都可由系、班级、姓名、学号来唯一地确定,这样,可以将学生(STUDENT)定义成含有系(DEPARTMENT)、班(CLASS)、姓名(NAME)、学号(NUMBER)这四个分量(或称成员)的一个派生结构。其具体结构为:

```
TYPE STUDENT
  CHARACTER(LEN=20)::DEPARTMENT
  CHARACTER(LEN=10)::CLASS
  CHARACTER(LEN=15)::NAME
  INTEGER::NUMBER
END TYPE STUDENT
```

可见,派生类型的定义是由关键字 TYPE 开始,其后跟所说明类型的类型名,下边是各个成员的说明。类型说明由关键字 END TYPE 结束,其后可有选择地跟所定义类型的类型名。上例的 STUDENT 作为一个新的派生类型,可以像五种基本类型那样来说明新的变量。例如,若声明 PERSON 具有 STUDENT 类型,可用下语句说明:

```
TYPE(STUDENT)::PERSON
```

对于具有 STUDENT 类型的变量 PERSON,可采用下述方法赋值:

```
PERSON = ("COMPUTER", "92-2", "LI LIN", 21)
```

当然还可以采用其它的方法赋值,以后会介绍。

对于派生结构内的每一个成员,可以通过在结构名后加百分号(%)然后加成员名的办法来使用,其用法就跟简单变量一样。例如 PERSON%NAME、PERSON%CLASS 等,其类型与相应成员的类型一致。

2.3.7 基本类型数组

前面提到的派生类型是一种复合类型,该类型可由多个不同类型或相同类型的量复合而成。在 FORTRAN90 中支持的另一个复合对象就是数组。数组是具有相同类型的元素的有序组合。数组在使用前必须进行说明,此处只介绍说明基本类型数组的方法。

例如,在某一程序单元内要用到具有 50 个元素的整型数组 X,可以使用下列语句:

```
INTEGER, DIMENSION(1:50)::X
```

这个数组的元素依次为 X(1)、X(2)、X(3)、...、X(50)。这些元素均都是整型,它们的名字都是 X,针对每一个元素,可依据不同的序号(称为下标)来唯一地识别。因此也将数组元素称为下标变量。

数组在使用前必须说明,一般说明的内容包括数组的名称、类型、维数(即每个元素下标的个数)及每一维的上下限(即下标的起止号)等。例如:

```
REAL, DIMENSION(-4:5)::A
```

说明了一个一维的实型数组 A,共有 10 个元素,分别是 A(-4)、A(-3)、...、A(0)、...、A(5)。而下面的说明:

```
INTEGER, DIMENSION(5,3)::B
```

说明了一个二维的整型数组 B。由于各维的下界缺省,其隐含值为 1。该语句与下述语句是等价的:

```
INTEGER, DIMENSION(1:5,1:3)::B
```

FORTRAN90 规定数组的维数最多可达七维。

数组是一些具有相同类型元素的有序集合。既然有序,其元素在内存中的存放就应有一定的规律。实际上数组的存放并不像我们所书写的那样是一个矩阵形式或一个立体形式,它的元素在内存中占有一片连续的存储单元,按一定规律排列。FORTRAN 语言规定,对于二维及多维数组,始终是最后一个下标先得到一个值,然后再从外向内依次变化,即最后一个下标变化慢、第一个下标变化快。

例如,对于如下说明的一个二维数组:

```
INTEGER, DIMENSION(3,2)::A
```

其元素的有效顺序为 A(1,1)、A(2,1)、A(3,1)、A(1,2)、A(2,2)、A(3,2),即按列优先的次序存放。对于

```
REAL, DIMENSION(2,2,2)::B
```

其元素存放顺序为 B(1,1,1)、B(2,1,1)、B(1,2,1)、B(2,2,1)、B(1,1,2)、B(2,1,2)、B(1,2,2)、B(2,2,2)。

一个数组中的元素的个数称为它的大小;数组的维数称为它的秩;每维中元素的个数称为维的长度,长度序列被称为形。

例如,对于如下说明的数组:

```
INTEGER, DIMENSION( -3:4, 7:15, -4:8, 4)::SUM
```

数组的秩是: 4

维的长度依次为: 8, 9, 13, 4

该数组 SUM 的大小为: $8 \times 9 \times 13 \times 4 = 3744$ 个元素

数组 SUM 的形为: (8, 9, 13, 4)

派生类型也可以包含数组分量。例如:

```
TYPE STUDENT
  CHARACTER(LEN=15)::DEPARTMENT
  INTEGER, DIMENSION(4)::CLASS
  REAL, DIMENSION(4,30)::SCORE
END TYPE STUDENT
```

在该派生结构中,分量 CLASS 是一维数组,SCORE 是二维数组。

数组的引用一般是通过指定下标值的办法来实现。下标值的指定方法可用整数或整型算术表达式的形式。但无论如何,所得到的值不能越过相应维的上下界,否则会出现越界错误或出现不可预料的结果。当然,下标的个数必须与数组的秩相等。

假设下面用到的数组已在不同程序单元中经过了相应的说明。则可按以下方式引用:

```
A(10)           ! A 是一维数组
B(3 * I, J)      ! B 是二维数组, I, J 为整型变量
STU%SCORE(I, K) ! STU 是 STUDENT 类型, I, K 为整型变量
```

在 FORTRAN90 中,允许对数组名进行运算。假设有

```
REAL, DIMENSION(3,3)::A
```

那么, $3 * A$ 就相当于将数组 A 中的每个元素的值取出均乘以 3 后所形成的新数组,而 $\text{SQRT}(A)$ 则相当于将数组 A 的每个元素的值取出都做平方根运算后所形成的新数组。这些对于某些工程运算是很方便的。

在 FORTRAN90 中,允许对数组片段进行操作。所谓数组片段是指原数组(称为父数组)中某些连续元素的重新组合。数组片段也是一个数组。设 A、B、C 分别是一维、二维、三维数组。则:

A(I:J)是数组 A 的数组片段,也是一维数组,共有 $J - I + 1$ 个元素。

B(K,1:N)是数组 B 的数组片段,它是个一维数组,相当于 B 数组 K 行的第 1 列至第 N 列的 N 个元素。

列元素的一个重新组合。

虽然数组片段也是一个数组,但对其内部单个元素的操作不能通过片段标识符来进行。

例如:写 B(K,1:N)(L)这是不正确的,而应使用该数组在原数组中的标识符,即 B(K,L)。

2.3.8 字符子串

字符型常量又叫字符串。一个字符串中的一部分(字符串中相邻的几个字符)称为子字符串或字符子串。为了说明字符子串的有关特点,首先建立一个字符型数组 ROW:

```
CHARACTER(LEN=1), DIMENSION(80)::ROW
```

ROW 是一个具有 80 个元素,且每个元素的长度为 1 的一个字符型数组。对于该数组的每个元素,可采用指定其下标序号的方法来引用,例如 ROW(I)。然而,若采用下述说明方法更合适一些:

CHARACTER(LEN=80)::ROW

它说明了 ROW 是可存放 80 个字符的一个字符型变量。如果要表示该字符型变量的第 I 个字符,其形式为 ROW(I:I)。这就是一个字符的字符子串的表达方法。字符子串的形式为:在一个字符变量(或字符型数组元素)名字的后面有一对括号,括号内写两个整数(或整型表达式),两者之间用冒号分隔。例如:

ROW(4:7)

则表示 ROW 字符串中从第 4 个字符到第 7 个字符组成的一个子串。

字符子串表达的一般形式为:

字符变量名(或字符数组元素名)($i_1:i_2$)

其中, i_1 是整数或整型表达式,那么,该字符子串的长度为 $i_2 - i_1 + 1$ ($i_2 \geq i_1 \geq 1$)。同时规定,视情况不同,可省写 i_1 或 i_2 :

- (1) 当 i_1 不写时,则表示 $i_1 = 1$;
- (2) 当 i_2 不写时,则表示 i_2 的值为字符变量的总长度;
- (3) 当 i_1, i_2 均不写时,则该子串就是表示字符变量本身;
- (4) 无论 i_1, i_2 哪个不写,但在表达子串时,冒号不能省。

对于上面说明的字符变量 ROW,若 $1 \leq I \leq 80$

ROW(:I) 相当于 ROW(1:I)

ROW(I:) 相当于 ROW(I:80)

ROW(:) 相当于 ROW(1:80)

可以将字符长度说明与数组说明结合起来组成特定长度的字符数组,例如:

CHARACTER(LEN=80), DIMENSION(25)::ROWC

这个数组 ROWC 具有 25 个元素,每个元素可以存储 80 个字符的信息。假设每个元素用来存放一行字符,可以利用这样说明的数组来存储 25 行(每行 80 个字符)信息。利用 ROWC(I)来引用第 I 行的信息;如果要引用第 I 行的第 J 个字符,则可以将数组下标和字符子串的引用结合起来:

ROWC(I)(J:J)

2.3.9 数据对象和子对象

在介绍有关对象的内容之前,先介绍一下有关标量的概念。标量是一个非数组的数据。标量可以是基本类型或派生类型。

为了简单,可以只考虑基本类型的数组和标量变量的子串。在前面曾提到过,派生类型可以有数组分量,例如:

TYPE STUDENT

...

INTEGER, DIMENSION(4)::CLASS

...

END TYPE STUDENT

而数组又可以是派生类型的,例如:

TYPE(STUDENT),DIMENSION(30)::STU

一个单独的结构(例如 STU(3))总是一个标量,但它内部可以含有数组分量(例如在刚刚说明的派生类型中的 STU(3)%CLASS)。派生类型可以有其它派生类型的分量。

数据对象(常常缩写为对象)都具有一些性质(例如类型、秩和形等),这些性质确定了数据的特性和对象的使用方法,它们被称为数据对象的属性。在同一程序单元内,一个有名数据对象禁止多次被说明为某一种或几种属性。例如变量 A 若在一个程序单元中被说明为实型后,就不允许再次对它进行类型说明。有名数据对象的类型或者可以用其名字的首字母隐式说明(即隐含说明,在 FORTRAN90 中已不提倡使用),也可以用类型说明语句显式说明。附加的属性也可用单独的说明语句来说明,所有的属性也可以包含在类型说明语句中。例如:

INTEGER::INCOME,EXPENDITURE

说明 INCOME 和 EXPENDITURE 两个有名的数据对象为整型。

REAL,DIMENSION(10:20)::X,Y,Z

说明具有名字 X、Y 和 Z 的三个数据对象,它们都是具有默认的实型种别的一维数组,且其下界为 10,上界是 20,其大小为 11。

子对象是某些有名对象的一部分,可由其它部分引用和独立地说明(只是变量的说明)。这些包括数组的一部分(数组元素和数组片段)、字符串的一部分(字符子串)和结构的一部分(成员)。子对象是它们自己的数据对象,但子对象只能由子对象标志符所引用。该标志符由对象名及其后的一个或多个限定词组成。后继的每一个限定词限定了由前面的名字或标志符所确定的对象的一部分。例如派生类型数组片段 STU(13:18)及 STU(3)%CLASS 均为 STU 的子对象。

由一个名引用的对象是:

有名标量 (标量对象)

有名数组 (数组对象)

由子对象标识符引用的子对象是:

数组元素 (标量子对象)

数组片段 (数组子对象)

结构分量 (标量或数组子对象)

子串 (标量子对象)

由此可见,FORTRAN90 中的术语“数组”和“变量”比在 FORTRAN77 中具有更广泛的含义。术语“数组”用于非标量的对象,包括数组片段或某一结构的数组分量。术语“变量”用于没有指明为常量的有名对象和这种对象的任何一部分,其中包括数组元素、数组片段、结构的分量和子串。

2.4 程序单元概念

程序单元是 FORTRAN 程序的基本成分。程序单元可以是主程序、子程序、模块或块

数据程序单元。一个子程序可以是函数子程序或子例行子程序。模块包含的说明是对其它的程序单元形成可访问性。块数据程序单元用来对有名公用区中的数据对象赋初值(在 FORTRAN90 中不提倡使用)。

2.4.1 主程序

主程序是不包含 SUBROUTINE、FUNCTION、MODULE 或 BLOCK DATA 语句作为其第一条语句的程序单元。

主程序的格式为：

```
PROGRAM 程序名
  [说明部分]
  [可执行部分]
  [内部过程]
END PROGRAM [程序名]
```

主程序的可执行部分不能包含有 RETURN 语句或者 ENTRY 语句。

程序名对可执行程序是全局的,而且不得与该可执行程序中的任何其它程序单元名、外部过程名或公用块名相同,也不得与主程序内的任何局部名相同。

下面看一个主程序的例子：

```
PROGRAM MAIN                ! 主程序开始语句
  IMPLICIT NONE              ! 说明部分
  REAL,DIMENSION(10,10)::A,B
  CALL FIND                   ! 执行部分
  ...
CONTAINS
  SUBROUTINE FIND             ! 内部过程
  ...
END SUBROUTINE FIND
END PROGRAM MAIN              ! 主程序结束语句
```

在主程序的作用范围内的说明不得包含 OPTIONAL 语句、INTENT 语句、PUBLIC 语句或它们的等价属性,在主程序内 SAVE 语句不起作用。

在主程序中的可执行部分说明了程序执行过程期间主程序的动作。它是由若干可执行语句构成的,绝大多数可执行语句都有关键字。常见的可执行语句有赋值语句、IF 结构、DO 结构、CASE 结构以及 READ 语句和 PRINT 语句等。

说明部分为可执行部分确定了环境。常见到的类型说明语句有: INTEGER、REAL、COMPLEX、LOGICAL、CHARACTER、TYPE(类型名)等,它们确定所列出的实际的类型和其它属性。

主程序内的任何内部过程的定义必须跟在 CONTAINS 语句之后。

2.4.2 子程序

子程序是可以完成某一独立算法或功能的程序单元,但它功能的具体体现要通过主程序(或子程序)的调用来实现。子程序又称为过程,按子程序单元与主程序的位置关系分为内部过程和外部过程。外部过程独立成为一个程序单元,处在主程序单元之外,与主程序单元分别编译。如果子程序不是在主程序之外单独编写,而是在主程序单元之内,作为主程序内包含的一个过程,则称其为内部过程。

内部过程可以出现在主程序、外部过程或模块内。内部过程不能出现在其它过程中。除去内部过程名不应是全局的以外,内部过程和外部过程是一样的。内部过程不能包含 ENTRY。

2.4.3 模块

模块也是一种在程序单元之外独立编写的程序单元。它有独特的形式,即模块程序单元内没有可执行语句,除了说明语句外,最多包含内部过程。模块的主要作用是供其它程序单元引用。

模块的一般形式为:

```
MODULE 模块名  
  [类型说明语句]  
  [CONTAINS  
    内部过程]  
END MODULE [模块名]
```

一个程序单元如果引用模块,实际上就是把该模块内的全部语句复制到本程序单元中,并且所有与模块中的名字相同的变量等,彼此取值相同,共享存储单元。因此模块起共享及复制的作用。

模块的引用采用下面的形式:

```
USE 模块 1, 模块 2, ..., 模块 n
```

2.4.4 块数据程序单元

块数据程序单元对有名公用块内的数据对象提供初值。其形式为:

```
BLOCK DATA 程序名  
  [说明部分]  
END [BLOCK DATA [程序名]]
```

由于块数据程序单元赖以支持的 COMMON 语句和 DATA 语句在 FORTRAN90 中已成为过时特性,因此,在 FORTRAN90 中,不提倡使用块数据程序单元。

本章叙述的内容大部分将在以后的章节中详细介绍。

习题 2

- 2-1 FORTRAN 的含义是什么? 其第一个版本是哪一年推出的?
- 2-2 FORTRAN 语言源程序一般由哪几部分构成?
- 2-3 FORTRAN90 字符集包含哪些字符? 希腊字母和罗马字母是否包含其中?
- 2-4 FORTRAN90 有几种基本类型的常量? 都是什么?
- 2-5 请简单描述一下名字和记号的相同处和不同处。
- 2-6 什么是程序? FORTRAN 语言有几类程序单元? FORTRAN90 中常使用哪几类?
- 2-7 FORTRAN90 有几种基本类型的变量? 请分别写出它们的语句说明符。
- 2-8 请说出 ANSI 和 ASCII 的中文含义。
- 2-9 FORTRAN90 的实型常量有几种表示方法? 试举例说明。
- 2-10 FORTRAN 90 的复型常量是如何表示的?
- 2-11 FORTRAN90 的逻辑型值有几个? 各表示什么意义?
- 2-12 什么是字符串和字符子串? 分别采用什么样的表示方法?
- 2-13 什么是数组的秩、维长和形? 数组的大小是什么? 怎样计算数组的大小?
- 2-14 有了数据类型以后, 为什么还要引入类型种别的概念?
- 2-15 FORTRAN90 的过时特性是什么含义? 程序中是否能使用这些过时特性? 如果不能使用, 请简述其理由; 如果能使用, 请说明在程序设计时我们应如何对待这些过时特性。

第 3 章 基本语句

前面已经介绍了 FORTRAN90 的语言组成元素及有关程序单元等概念,也了解了程序是由语句组成。FORTRAN90 的程序具有强大的功能,语句很多,但支持程序的最基本的语句主要有四种:类型说明语句、赋值语句、输入语句、输出语句。运用这些基本的语句,就可以编写一些简单的程序。

3.1 类型说明语句

FORTRAN90 中的变量通常要进行类型说明,然后才能使用。类型说明语句的一般形式是:

类型说明(种别说明),属性说明 1,属性说明 2,...,属性说明 n::变量名表

例如:

INTEGER(KIND=2),DIMENSION(1:15)::A

说明变量 A 的类型是整型、种别是 2、属性是 DIMENSION(1:15),即 A 被说明为种别参数为 2 的一维整型数组,具有 15 个元素。这里的种别说明和属性说明不是必选的,有时也可以省略。类型说明省略了种别说明和属性说明后具有最基本形式:

类型说明::变量名表

3.1.1 类型说明

程序中每个数据都有也只有一个数据类型。一般而言,在 PROGRAM 语句和可执行部分之间的语句为变量的说明语句。每个说明由指定的 FORTRAN90 中内部类型的关键字,后跟两个冒号及由逗号隔开的变量名表组成。内部类型为系统内部配置好的类型。FORTRAN 中有五种内部类型:整型、实型、复型、逻辑型和字符型。在 FORTRAN90 中还可以利用系统内部类型自行设计出新的数据类型,即派生类型。

1. 变量类型说明

说明上述五种基本类型数据的关键字分别是 INTEGER、REAL、COMPLEX、LOGICAL 和 CHARACTER(2.3.5 节),说明派生类型数据的关键字是 TYPE。变量类型一经说明,在执行部分中就必须遵守该类型书写格式及使用规则。否则,就要出错。例如,要说明 I,J,K 是整型变量,说明语句的最基本形式是:

INTEGER::I,J,K

变量表前用双分隔号“::”隔开,变量表有多个变量时用逗号“,”分开,语句的最后没有标点符号。要说明 X 和 Y 是实型变量,说明语句的最基本形式是:

REAL::X,Y

要说明 A 是复型变量,说明语句的最基本形式是:

COMPLEX::A

对于几个类型相同但形不同的实体,可以方便地使用一条语句来说明,如:

INTEGER::A,B,X(10),Y(3,3)

REAL::M(2,4),N

数据在机器内的存储方式不同,一个变量是整型还是实型应根据题意要求来说明。整型数运算速度快,在机内存储没有误差,但能表达的值的范围较小;实型数能表示小数、分数及不同的精度,表达的值的范围大,但数的外部表示与机器存储会有误差。例如送入一个实数 10.2,可能会在机器内表示为 10.1999998。因此在程序中能用整数的地方,尽量使用整数;在使用实数的地方,尽量避免把两个实数作相等或不相等的比较。例如 A 是实型时,如果在 A 等于 10.2 时输出 A 的值,不相等时不输出,这时不能写成下面的语句:

```
IF (A==10.2) PRINT *,A
```

因为当理论上 A 的值是 10.2 时,实际上机内 A 的值可能是 10.1999998,故关系表达式的值可能是假,从而不执行输出语句。遇到这类实型数做相等比较时,可改用下述语句

```
IF (ABS(A-10.2)<1.0E-6) PRINT *,A
```

就可以达到 A 和 10.2 做相等比较的目的。

2. 淘汰隐式说明

FORTRAN 以前的版本规定,程序单元中未经说明的变量具有隐含的类型。即凡是名字以 I 至 N 开头的变量都是整型变量,名字以其它字母开头的变量都是实型变量,这称作 I-N 规则。而 FORTRAN90 不提倡使用这种隐式说明。为了抑制这种隐式说明发生作用,应该在程序说明部分一开始就写出语句:

```
IMPLICIT NONE
```

向系统声明不使用隐式说明。这是 FORTRAN90 编程的一种良好的习惯。

3. 说明语句内给变量赋初值

程序中常需要对一些变量预先设置初值,FORTRAN90 规定可以在说明变量时使变量初始化。它的形式为:

类型说明::变量名 1=a1,变量名 2=a2,...

在类型说明语句的变量表中,把要置的初值写在指定变量名后,例如:

```
REAL::A=3.8,B=4.5
```

使实型变量 A 有初值 3.8,B 有初值 4.5,也可以给被说明的部分变量赋初值,例如:

```
REAL::X=1.2,Y=2.5,Z
```

既说明了变量类型,也为 X、Y 置了初值。

3.1.2 种别说明

FORTRAN90 中提供了对可移植数据精度和范围进行选择的机制,即种别参数。它提供了对每种内部数据类型的不同机器表示进行选择的参数化方式。种别参数为整数。

一个数据通常在内存中占有一定数目的存储单元,但同一类型内数的大小与精度要求不同。因此它们要求的存储单元数量也不相同。按照变量表达的值范围与精度范围,把同一类型划分成几个种别,不同种别分配不同数目的内存单元。具体的种别划分视具体的计算机处理系统而定。因此程序员在使用变量时,可按变量表达值的范围、精度要求范围查阅上机手册,以确定每种类型的可用种别参数及每种类型的缺省种别参数。种别参数在所有情况下均为任选。

1. 种别的说明方法

种别说明的关键字是 KIND, 后跟“=”号及种别参数, 写在类型关键字后括号内。例如, 说明语句:

```
REAL(KIND=2)::X
```

说明变量 X 是实型, 种别参数是 2。

一个变量一定有一个种别, 如果变量的类型说明语句中没有种别说明符, 如:

```
REAL::A,B
```

则表示变量 A、B 的种别缺省(指采用系统规定的标准值)。

2. 有关种别的函数

FORTRAN90 中的内在函数(系统中已配置好可直接引用的函数称为内在函数)有一部分是专用于种别选择的。例如:

KIND(X) 函数: 返回变元 X 的种别参数值, 如 KIND(X) 值为 2, 则 X 的种别参数是 2, 当 KIND 函数的自变量取 0 时, 返回标准种别参数, 也即缺省种别参数。

KIND(0) 返回值是整型的标准种别参数。

KIND(0,0) 返回值是实型的标准种别参数。

KIND(.FALSE.) 返回值是逻辑型的标准种别参数。

KIND('A') 返回值是字符型的标准种别参数。

SELECTED_REAL_KIND(n,m) 函数: 用来产生一个种别值, 它表示某一精度和范围。其中 n 是指十进制有效位的位数, m 指明值范围内以 10 为底的幂次。例如:

SELECTED_REAL_KIND(16,50) 返回一个表示 16 位精度、值范围在 $-10^{50} \sim 10^{50}$ 之间的实型数的种别参数。

SELECTED_INT_KIND(6) 返回一个能存储 $-10^6 \sim 10^6$ 的整型种别参数。

3. 常数种别的表示

程序中的数值型常数、逻辑型常数的种别用后缀法表示, 即后加一下划线, 再跟种别参数; 若是字符型常数, 则用前缀法, 把种别参数列在字符常数之前, 中间用下划线连接。例如:

15_2	表示种别为 2 的整型数 15
14.56_3	表示种别为 3 的实型数 14.56
.FALSE._4	表示种别为 4 的逻辑型常数假
5_'abc'	表示字符串 'abc', 其种别参数是 5

注意: 复型常数的种别由两个实数成分的种别来标明; 字符常数的种别与其它四种内部类型常数的种别标记方法不同, 前者作为前缀, 后者作为后缀。

3.1.3 属性说明

属性是被说明对象的所属性质。一个对象被说明具有某一属性时, 就使该对象具有某种附加功能、特殊的使用方式与适用范围。一个对象可以没有附加的属性说明, 也可以有多个属性说明。属性关键字写在说明语句种别说明符之后, 双分隔号之前, 各属性关键字间用

逗号分开,次序任意。

属性关键字如下:

PARAMETER	DIMENSION	PUBLIC	INTENT
PRIVATE	OPTIONAL	POINTER	SAVE
TARGET	EXTERNAL	ALLOCATABLE	INTRINSIC

现在我们只介绍几个属性关键字的作用,其它的将在有关章节中叙述。

1. PARAMETER 属性

PARAMETER 属性也称常量名属性,用于命名常量。说明的形式类似于变量,即在类型说明符后加上该关键字,在双括号之后是常量名、等号、常量值,常量名的命名方法与变量名一样,但在程序中不能改变常量的值。例如

```
INTEGER, PARAMETER::N=5
```

说明了 N 是常量名,在程序中代表整型常数 5。常量名被说明后,在程序中就不可再改其值。如要修改,系统会给出出错信息。尽可能地把常量说明成参数形式是一种良好的编程习惯,使用参数代替对应常量可使程序更易读,且程序可以很容易地进行修改特定值。

2. DIMENSION 属性

DIMENSION 也称数组属性。要说明一个数组,需要在说明语句中附加数组属性关键字。数组属性关键字是:

```
DIMENSION(数组形状说明)
```

这里的数组形状说明是指数组的维数以及每维下标的变化界限。例如:

```
INTEGER, DIMENSION(1:10)::X
```

说明 X 是整型一维数组,下标的下界是 1,上界是 10。

3. INTENT 属性

INTENT 也称意图属性,只用于子程序中,说明子程序中虚元的使用意图:该虚元把值从主调程序中传进来(IN),或者把子程序内求得的值传到主调程序去(OUT),或两种作用均有。例如在子程序中写说明语句

```
INTEGER, INTENT(IN)::X
```

表示 X 是一个虚元,用来把主调程序内的一个值传递到该子程序内。子程序内若有

```
INTEGER, DIMENSION(1:10), INTENT(OUT)::A
```

则说明了 A 是一维数组、虚元,用于向主调程序传递值。

3.2 算术表达式和赋值语句

许多程序的主要功能在于计算,而计算是通过表达式的求值来实现的,例如下面语句

```
Y = (SIN(X) + COS(X)) * L/T + Q * Q * R + 0.5
```

的作用是求出表达式 $\frac{(\sin x + \cos x)L}{T} + Q^2 \cdot R + 0.5$ 的值并赋给变量 Y。

因此表达式是 FORTRAN 程序中不可缺少的成分。从上面语句可以看到,在表达式中包含常数(0.5)、变量(X, Q, L, T, R)、函数(SIN, COS)和运算符(如 +, *, /),它们是算术表达式的基本成分,其中常数和变量前面已介绍过。

3.2.1 算术表达式

1. 算术运算符

算术运算符共有 5 个

+	加(或正号)	-	减(或负号)
*	乘	/	除
**	乘方		

注意:乘号是用“*”而不是用“×”,以免与字母 X 混淆。乘号是不能省的,例如不能用 AB 代表 $A * B$,除号必须用“/”。用两个星号代表乘方运算, 3^2 在 FORTRAN 中用 $3 * * 2$ 表示。两个运算符不能相邻,例如 $\frac{A}{-B}$ 不能写成 $A / - B$,而应写成 $A / (- B)$ 。

2. 算术运算符的优先顺序

**	最高
* 和 /	次之
+ 和 -	最低

同一优先级的运算按从左到右的原则(左结合)。例如表达式

$3.5 + 2 * A / B + V * * 2$

↑ ↑ ↑ ↑ ↑

④ ② ③ ⑤ ①

要先进行乘方运算,然后再进行乘法,第三步是除法运算,最后是加法运算(先左后右)。 $-B * * 3$ 相当于 $-(B * * 3)$, $X / Y * Z$ 相当于 $(X / Y) * Z$ 。对于 $3 * * 2 * * 3$,是先计算 3^2 还是先计算 2^3 呢? FORTRAN 规定,对连续的乘方运算,采用先右后左的方式(右结合),即先进行 2^3 运算,得结果 8,然后再进行 3^8 运算,得 6561。该表达式相当于 $3 * * (2 * * 3)$ 。

3. 内在函数

在解题中常会遇到一些专门的运算,如求 \sin 和 \cos 、平方根、对数,以及求几个数中的最大数和最小数等。在 FORTRAN 中,用户要想求某数的函数值,通常不必自己编写程序,只需直接使用 FORTRAN 系统提供的内在函数。例如写出 $SQRT(4.0)$,就会计算出 $\sqrt{4}$ 的值。写出 $SIN(0.0)$ 就能得到 0 的正弦值。

这种 FORTRAN 语言内配置的函数称内在函数。内在函数因其功能不同分为三类:基本函数、变换函数和查询函数,它们的全部名称及功能见书后附录。本章只介绍几个常用的内在函数。

(1) 基本函数

数学上的各种初等函数都可在基本函数中找到。具体引用时,只要把 X 代以实际的自变量即可。例如,有关函数的值如下:

$ABS(-5.6) = 5.6$	$COS(1.0) = 0.5403023$
$SIN(1.0) = 0.8414710$	$TAN(1.0) = 1.557408$
$MAX(3, 1, 10, 5) = 10$	$MIN(3, 1, 10, 5) = 1$
$SQRT(4.0) = 2.0$	$LOG(3.0) = 1.986123$

$$\text{LOG10}(100.0) = 2.0$$

$$\text{MOD}(9,6) = 3$$

$$\text{SIGN}(-3.0, 2.0) = 3.0$$

$$\text{EXP}(3.0) = 20.08554$$

$$\text{MOD}(1,3) = 1$$

$$\text{SIGN}(3.0, -2.0) = -3.0$$

使用内在函数应注意:

1° 每一个内在函数要求一个或几个自变量,自变量用括号括起来。大部分函数只有一个自变量(如 SIN、LOG 等),有的函数要求二个自变量(如 MOD、SIGN 等),有的函数则要求不少于 2 个自变量(如 MAX、MIN)。一般根据函数功能便可判断自变量的个数。如果函数有一个以上的自变量,请注意自变量的顺序对运算结果有无影响,如 MOD(A1,A2),A1 是被除数,A2 是除数,A1 和 A2 的顺序不能颠倒。而 MAX(A1,A2,A3)求解 A1、A2、A3 中的最大值,与自变量顺序无关。

表 3-1 常用基本函数

函数名	含义	数学符号	FORTRAN 举例
ABS	求绝对值	$ a $	ABS(A)
COS	余弦值	$\cos x$	COS(X)
SIN	正弦值	$\sin x$	SIN(X)
TAN	正切值	$\tan x$	TAN(X)
ATAN	反正切值	$\tan^{-1} x$	ATAN(X)
ACOS	反余弦值	$\cos^{-1} a$	ACOS(A)
MAX	取最大值	$\max(a_1, a_2, a_3)$	MAX(A1, A2, A3)
MIN	取最小值	$\min(a_1, a_2, a_3)$	MIN(A1, A2, A3)
SQRT	平方根	\sqrt{a}	SQRT(A)
LOG	自然对数	$\log_e a$	LOG(A)
LOG10	常用对数	$\log_{10} a$	LOG10(A)
EXP	求指数函数的值	e^x	EXP(X)
MOD	求余数	$a_1 - \text{int}(\frac{a_1}{a_2}) \times a_2$	MOD(A1, A2)
SIGN	符号	$ a_1 $ (若 $a_2 > 0$) $- a_1 $ (若 $a_2 < 0$)	SIGN(A1, A2)

2° 求三角函数时,自变量必须以弧度为单位。如果原来的单位为角度,需先经转换,再把值填入函数名后边的括号内。

3° 函数对自变量的类型是有要求的。例如 SIN、LOG、EXP、SQRT 等要求自变量为实型(也可为复型),但不能是整型。

4° 自变量可以是常数、变量或表达式。例如 SIN(1.0)、SIN(Y)、SIN(A/3.0+0.5)都是合法的,甚至可以嵌套使用,如 SQRT(SQRT(625.0))。

(2) 转换函数

常用的转换函数主要有:

INT(X) 把实型或复型的值转换成整型。如 INT(2.1)的函数值是整数 2。复数

取 INT, 则取其实际部再转换成整型, 如 $\text{INT}((1.2, 3.5))$ 的函数值是整数 1。

$\text{REAL}(X)$ 把整型或复型的 X 的值取出转换成实型。如 $\text{REAL}(5)$ 的值为 5.0。

$\text{COMPLEX}(X, Y)$ 分别把 X, Y 作为实部和虚部, 合成一个复型数。如 $\text{COMPLEX}(2.3, 4.2)$ 的值为 $(2.3, 4.2)$, $\text{COMPLEX}(1.5)$ 的值为 $(1.5, 0.0)$ 。

数值类型转换函数也可用来在同一数据类型内从一种别转换成另一种别或指定数据类型间转换结果的种别参数。如 $\text{INT}(X, \text{KIND}=\text{SHORT})$ 把实数值转换成种别为 SHORT 的整数。若 Z 为复数, $\text{REAL}(Z, \text{KIND}(Z))$ 的值具有与 Z 同样种别并等于 Z 的实部的值。

$\text{INT}(A, \text{KIND})$ 把 A 的值取出转换为整型, 其种别参数是 KIND 。

$\text{REAL}(A, \text{KIND})$ 把 A 的值取出转换成实型, 以 KIND 的值作为种别参数。

$\text{COMPLEX}(A, B, \text{KIND})$ 把 A, B 分别作为实部和虚部, 合成一个复型数, 以 KIND 的值作该复型数的种别参数。

例如 $\text{INT}(A, 3)$ 表示把 A 的值取出转为整型, 其种别参数是 3。

FORTRAN 可以对数组作函数运算, 即函数的自变量还可以是数组名。对某个数组作函数运算相当于对数组内每个元素施加该函数操作。查询函数也有多种, 如前面介绍的 SELECT_INT_KIND , SELECT_REAL_KIND 等。其它函数将结合各章内容介绍。

在引用函数时应注意几点:

1° FORTRAN90 中绝大多数内在函数是类属函数 (指不单适用于一种类型自变量, 也可适用于其它类型的自变量, 函数值的类型、种别与自变量的类型、种别相同)。例如, 取整函数 $\text{INT}(X)$, 当自变量 X 是实型、复型时都可使用。这一点与以前版本的 FORTRAN 中有些函数名只能适用某一种类型的自变量不同。

2° 自变量可以是表达式, 计算机先求得表达式的值, 然后计算函数值。例如, $\text{ABS}(X)$ 中的 X 可以用表达式替换, 可有 $\text{ABS}(B * 3 + \text{SIN}(A + B))$ 等形式。

3° 某些函数的自变量不止一个且要满足一定的次序。例如, 求余数函数的引用 $\text{MOD}(A, P)$ 有两个自变量, 且 A, P 次序一定, 函数的值是 A 除以 P 后所得的余数。因此引用时要使用两个自变量, 且第一个是被除数, 第二个是除数。

4° 有些函数自变量是可选的。可根据情况决定是否要填入。例如, INT 函数的一般形式为

$\text{INT}(X, \text{KIND})$

它有两个自变量, 其中第二个自变量规定种别参数是可选的, 引用时可写成 $\text{INT}(X, 3)$, 当不需要规定种别时, 引用时可写成 $\text{INT}(X)$ 。

4. 算术表达式

(1) 表达式

表达式是由运算符和括号将各运算元素 (也称操作数, 例如常数、变量、函数、数组元素都是基本的运算元素) 连结起来的有意义的式子。算术表达式中各运算元素都必须是算术量, 用算术运算符把它们连接起来。例如 R 是一个数值变量, 则 $2 * 3.14159 * R$ 是一个算术表达式。算术表达式的值是一个数值。最简单的表达式只有一项, 它只是一个常数或变量名。

表达式中运算符运算的优先次序是:

① 括号 → ② 求函数值 → ③ $*$ 或 $/$ → ④ $+$ 或 $-$

同一层内运算符自左向右依次执行, 括号嵌套时先计算出最内层括号中的值, 逐步外推。例如, $\text{SIN}(X + Y) * 2$ 的运算次序是, 先进行括号内的运算符, 然后进行正弦的运算,

计算出 $\sin(X+Y)$ 的值后,才进行乘方的计算,求出 $\sin^2(x+y)$ 的值。

书写表达式时应注意以下几点:

- 1° 两算术操作数相乘, * 号不能省略。
- 2° 两表达式相除时,分子和分母上的表达式要分别加上括号。
- 3° FORTRAN 中没有方括号和花括号,只有圆括号,可以把圆括号多层套用。
- 4° 底是负数时,幂只许是整型;底是零时,幂不得为零。
- 5° 数学上无意义的算式不能写成表达式,如分母为零,负数开平方等。

表 3-2 是一些数学表达式表示成算术表达式时正确和错误的写法。

表 3-2 数学表达式表示成 FORTRAN 算术表达式

数学表达式	正确表达式	错误表达式
$a \cdot b$	$A * B$	AB
$a \cdot (-b)$	$A * (-B)$	$A * -B$
$\frac{a \cdot b}{c \cdot d}$	$(A * B) / (C * D)$	$A * B / C * D$
$(a \cdot b)^2$	$(A * B) * * 2$	$A * B * * 2$
a^{m+n}	$A * * (M + N)$	$A * * M + N$
$\sin 6x$	$\text{SIN}(6 * X)$	$\text{SIN}6X$ 或 $\text{SIN}(6X)$
$a[b - (c + d)]$	$A * (B - (C + D))$	$A * [B - (C + D)]$
$(a + b)(c + d)$	$(A + B) * (C + D)$	$(A + B)(C + D)$
$1.2 \log 67$	$1.2 * \text{LOG}10(67.0)$	$1.2 * \text{LOG}(67)$
$a \cdot e^x$	$A * \text{EXP}(X)$	$A * E * * X$
$\cos(3x)^3$	$\text{COS}((3 * X) * * 3)$	$\text{COS}(3 * X) * * 3$

(2) 表达式的类型与种别

在表达式运算时必须注意类型问题。如果运算量类型相同,则运算后所得的数据类型为同一类型。例如,两个整型数运算后得整型数,两个实型数运算后得实型数,两个复型数运算后得复型数。特别注意,由于两个整型数运算后仍得整型数,因此有可能把小数部分截去,造成误差。例如 $5/2$ 结果不是 2.5,而是 2, $1/2$ 的计算结果不是 0.5,而是 0。

如果运算量类型不同(称作混合运算),系统一般先把它们转换成相同的类型然后再进行计算。转换的规则是将较低级的类型转换为较高级的类型。数据类型隐含的高低次序如下:

复型	高
实型	↑
整型	低

即整型最低,复型最高,故整型与实型的运算结果为实型, $4 + 3.5$ 得 7.5。实型与复型的运算结果为复型,其余类推。

两个不同种别参数数据的运算,结果值取精度或允许值范围较大的那个参数的种别。设种别参数 5 比 4 高,4 比 2 高,则

3.1_4 + 4.5_5 得 7.6_5
 2.1_4 + (6.5,3.8)_2 得 (8.6,3.8)_4

(3)在表达式计算中要注意由于有效位数的限制而引起的误差

用计算机进行数值计算,有时会产生一些微小的误差,这是由于计算机用以存放数据的存储单元是由有限的字节(或二进制位)组成的,因此无法容纳无限多位数,也就是说其有效位数是有限的(一般为十进制数的7至9位)。例如1.0/3.0的结果在内存中不是0.333333333333...,而是0.3333333(假定使用的计算机系统的有效位数为7位),与理论值有些小误差,这在程序设计中应事先估计到。此外,还应当考虑到表达式计算过程中会不会产生“溢出”。所谓“溢出”是指计算出的数值超出了存储单元所允许的数的范围。如果发生“溢出”,有的计算机系统此时使程序停止运行,按出错处理;有的则将溢出的数按该计算机系统可表示的最大实数处理,但它已无意义了,因为它产生了潜在的错误使下边的运算也是错误的。因此书写表达式时,要避免两个很大的数相乘,或一个很大的数除以一个很小的数。

3.2.2 赋值语句

赋值语句的作用是将一个表达式的值赋给一个变量(或一个数组元素等)。语句的一般形式是:

$$V=e$$

V是一个符号名,可以是变量名、数组名、数组元素名、数组片段名、结构成分名等。但只能是一个名字,不允许出现表达式。“=”称为赋值号。e是表达式,其内容是待计算的值。因而以下赋值语句

$$A=4.2$$

$$B=A$$

$$Z=(-X**2)**3$$

$$GL=.TURE.$$

$$G2='ABC'$$

都是正确的。对赋值语句说明如下:

1° 赋值语句的功能是先计算右边表达式e的值,然后将它赋给赋值号左边的变量名V(或数组元素名、数组名等)。如果左边的变量名V(或数组元素名、数组名等)原来有值,则由新值将其覆盖。由此可见,赋值语句具有先计算、后赋值双重功能。

2° 赋值号“=”形式上与等号相同,但意义不同。赋值号的作用是把其右边e的值存入到左边V的存储单元中,从而使V获得新值,并不要求赋值号两端形式上相等。假设A=5,B=4,则以下两个语句的作用是不同的。

$$A=B$$

$$B=A$$

第一个语句的作用是将B的值赋给变量A,因此赋值后A和B都是4,第二个语句的作用是将A的值赋给变量B,赋值后A和B都是5,见图3-1。

赋值号左边只能是变量名(或数组元素名、数组名等),不能是表达式,它并不考虑数学习惯。例如,在数学中

$$X-2=A+B$$

是合法的,但作为赋值语句是非法的。因为赋值语句的一般形式要求V是一个变量名等,

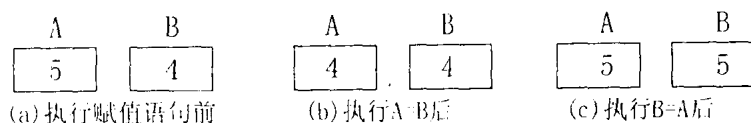


图 3-1 A=B 和 B=A 的不同

这里却写成一个表达式。反之

$$X = X + 2$$

在数学中是错误的,但它是一句正确的赋值语句,因为它符合赋值语句的一般形式。这句赋值语句作用是取出变量 X 所在单元中的数值,加上 2,再存入变量 X 中,且覆盖原来的值。如果 X 原来为 8,执行该赋值语句后,X 值变为 10,再执行一次这个语句,X 值是 12。

3° 赋值语句左端的变量和右端的表达式都有自己的类型、种别。逻辑型、字符型的赋值语句要求“=”两边类型相同,即 V 与 e 都是逻辑型或都是字符型,不允许两端类型不一致。对于数值型赋值,不区别整型、实型和复型,允许“=”两端的数值类型不同。例如 V 是整型时,e 可以是实型;V 是实型时,e 可以是整型等等。执行赋值语句后,V 的类型不变,系统自动把 e 的值转化为 V 的类型后再赋给 V。例如,设 A 为整型、B 为实型、C 为复型,则执行赋值语句序列:

$$B = 3$$

$$A = B$$

$$C = 1.5$$

之后,A 中值为 3,B 中值为 3.0,C 中值是(1.5,0),即 $1.5 + 0i$ 。使用这一规则要注意的是,当 V 是整型、e 是实型时,因为要把 e 的值化为整型后赋给 V,就会把尾数丢掉。假设 A、B 被说明为整型,执行赋值语句;

$$A = 3.9$$

$$B = -2.1$$

之后,A 的值是 3,B 的值是 -2。

在种别方面除字符型外,任何类型的 e、V 种别允许不同,执行赋值语句后,V 的种别不变,系统自动把 e 的值的种别转化为 V 的种别之后再赋给 V。

3.2.3 应用举例

[例 3-1]有一直流电路,电压 $U = 200$ 伏,电阻 $R_1 = 20$ 欧、 $R_2 = 50$ 欧、 $R_0 = 100$ 欧,求电路的等效电阻 R 和总电流 I(见图 3-2)。

从物理学可知:

并联电阻 R_{12} 的值为:

$$R_{12} = \frac{R_1 R_2}{(R_1 + R_2)}$$

总电阻: $R = R_0 + R_{12}$

电 流: $I = \frac{U}{R}$

程序如下:

```
PROGRAM MAIN
  IMPLICIT NONE
  REAL::I,R0,R1,R2,R12,R,U
  R0=100.0; R1=20.0; R2=50.0
  U=200.0
  R12=R1 * R2/(R1 + R2)
  R=R0 + R12
  I=U/R
  WRITE( *, *) 'R=',R
  WRITE( *, *) 'I=',I
END PROGRAM MAIN
```

运行结果:

R= 1.1428571E+02

I= 1.7500000

以上是我们所用的 FORTRAN90 系统 FTN90 输出的结果。在不同的计算机处理系统上,实数输出的表示方式(如所占的位数等)可能有所不同。

[例 3-2] 有一圆柱和圆球,已知半径 $r=1.5$,圆柱高 $h=3.0$,求圆周长、圆面积、圆球表面积、圆球体积、圆柱体积。

从数学知识可知:

圆周长 $L=2\pi r$

圆面积 $S=\pi r^2$

圆球表面积 $SQ=4\pi r^2$

圆球体积 $VQ=\frac{4}{3}\pi r^3$

圆柱体积 $VZ=\pi r^2 h$

程序如下:

```
PROGRAM MAIN
  IMPLICIT NONE
  REAL::PI=3.14159,R=1.5,H=3.0
  REAL::L,S,SQ,VQ,VZ
  L=2.0 * PI * R
  S=PI * R * * 2
  SQ=4.0 * PI * R * * 2
  VQ=4.0/3.0 * PI * R * * 3
  VZ=PI * R * * 2 * H
  WRITE( *, *) 'L=',L
  WRITE( *, *) 'S=',S
  WRITE( *, *) 'SQ=',SQ
```

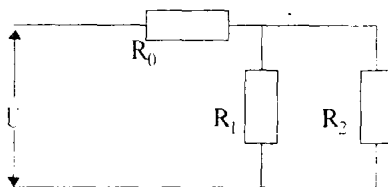


图 3-2 例 3-1 的电路图

```
WRITE( *, * ) 'VQ=', VQ
WRITE( *, * ) 'VZ=', VZ
END PROGRAM MAIN
```

运行结果如下:

```
L=    9.4247704
S=    7.0685778
SQ=   28.2743111
VQ=   14.1371555
VZ=   21.2057343
```

3.3 输入与输出语句

输入输出语句是高级语言最常用的功能,因此需要先有一定的认识。

3.3.1 输入和输出的概念

所谓输入与输出是以计算机内存为主体而言的。将外部介质上的数据通过计算机的输入设备送到计算机内存称为输入;将计算机内存的数据通过输出设备送到外部介质上,称为输出。外部介质可以是卡片、纸带、显示器、键盘、磁盘、磁带等。计算机运行一个程序需要原始数据,运行得到的结果,要输出到显示器或其它输出介质上。向计算机提供原始数据时,可以在程序中用赋值语句直接提供。但当数据量很大或有多组实验数据时,这种方法很不方便,会使程序增长,而且每次运行数据有变化时需要反复修改程序。因此,为了使程序具有通用性,常使程序和数据分开,在每次运行程序时,由计算机输入设备将数据送入到计算机内。

FORTRAN 输入和输出语句为:

READ 语句(输入语句、也称读语句),用于输入。

PRINT 语句和 WRITE 语句(输出语句、也称写语句),用于输出。

一般情况下,执行读语句或写语句必须具备三个条件:即设备、格式、对象,缺少任何一个都无法作正确的读写。在输出方面,通常我们只讨论显示输出(以终端显示器作为输出设备)和打印机输出。在输入方面,我们也只讨论键盘输入。从键盘上,敲入一组字符并按回车键,这叫做输入一个“记录”。输出一行数据称作输出一个记录。输入与输出均以记录为单位,即一次输入或输出一行。

3.3.2 输入语句

1. 基本格式(自定格式)输入

输入语句又称读语句,其最基本的形式为:

READ(部件号,格式说明)输入表

例如:

```
READ(5, '(I7)') X, Y
```

关键字 READ 指示机器要读入数据,5 是部件号,假定代表终端;I7 是编辑符,括号之外要用撇号括起来,编辑符 I7 规定每行读入一个整数,I 表示整型数据编辑,7 表示该数要

占7列(7个字符)位置;X和Y是输入表中的两个变量名,指明读入的数据存放地点,终端上第一行输入的数据送给X,第二行数据送给Y。若输入为

2134↵

865↵

时,X得值2134,Y得值865,这里“↵”表示回车键。

当输入设备为系统隐含指定的设备时,可以有两种方法使输入语句简化。一是把部件号用*代表,如上述输入语句可写成:

READ(*,'(I7)')X,Y

表示数据由系统隐含指定的设备输入。另一简化方法是把部件号省缺,只写出格式和对象。语句的一般形式为:

READ'(I7)',X,Y

表示系统隐含指定的设备中按I7格式输入两数,分别赋给X和Y。数据输入格式同前面一样,仍分两行,但在关键字READ后紧跟格式说明,在格式说明后用一个逗号将格式说明与输入变量表隔开。

注意:要输入的对象应写在输入表中,彼此用逗号分开,它们只能是变量名、数组名、数组片段名、结构成员名,不能是表达式(上例中是X,Y)。

2. 表控格式输入

如果没有特别的输入格式要求,一般可以使用机器提供的标准格式:表控格式。使用表控格式时,只需在读语句的格式说明的位置上写一个'*',机器会按机内制定的表控格式读入,不再需要写明编辑格式。每个输入的数用数学上习惯写法,各数据间用逗号或空格分开。

表控格式输入的一般形式为:

READ(部件号,*)输入表

缺省部件号的读语句的一般形式为:

READ*,输入表

显然,当部件号5代表系统隐含指定的设备时,下面3个语句等价:

READ(5,*)输入表

READ(*,*)输入表

READ*,输入表

它们表示在系统隐含指定的设备上按表控格式输入。假设想让 $X=1, Y=2, Z=3, W=4$,当变量为整型时,执行到语句:

READ*,X,Y,Z,W

之后,计算机会等待键盘输入数据。这时可打入

1,2,3,4↵

X、Y、Z和W中的值将分别取1、2、3和4。当变量均是实型时,可以打入

1.0,2.0,3.0,4.0↵(或1.,2.,3.,4.↵)

当变量是复型时,打入:

(1,0),(2,0),(3,0),(4,0)↵

每个复型变量输入一对括号,括号内先写实部,后写虚部,中间用逗号分开,每对括号间用逗号分开,则X值为 $1+0i, \dots, W$ 值为 $4+0i$ 。需要说明的是:

1° 输入的数据应当和 READ 语句中变量的类型相适应。假设 X、Y 和 Z 都为整型时, 当执行语句

READ *, X, Y, Z

时, 输入以下数据:

2.0, 4.5, 6. ✓

是不正确的。因为 X、Y、Z 均需要整型数据, 而输入给它们的是实型数。

2° 输入数据的个数应当和 READ 语句中变量个数相等。如果输入一个记录中的数据个数少于所需求的个数时, 则 READ 语句可从下一个记录中继续读数, 直到满足所需的数据为止; 如果一个记录中数据的个数多于 READ 语句中变量的个数, 多余的数据不起作用。假设 A、B、C、D、E、F 为实型变量, 语句

READ *, A, B, C, D, E, F

用以下几种输入方法都是可以接受的:

① 11.2, 21.5, 32.4, -10.1, 0.83, -7.21 ✓

② 11.2, 21.5, 32.4 ✓

③ 11.2 ✓

21.5, 32.4, -10.1 ✓

0.83 ✓

-7.21 ✓

④ 11.2, 21.5, 32.4, -10.1, 0.83, -7.21, 101.85, 19.31 ✓

在第①种情况中, 一个记录提供了 READ 语句所需的全部数据。第②种情况中, 第一个记录只提供了三个数据, 未满足所需的数据个数, 计算机等待键盘再输入下一个记录, 输入第二个记录后 READ 语句执行完毕。第③种情况是, 直到第四个记录输入后, READ 语句才执行完毕。第④种情况下, 前 6 个数据分别送入给 A、B、C、D、E、F, 而 101.85 和 19.31 是多余的数据, 不被读入。

3° 每一个 READ 语句都从一个新的记录开始读数。换句话说, 不能从一个记录的中间开始读数。例如: 设 I、J、K、M、N 为整数, 输入语句序列为:

READ *, I, J, K

READ *, M, N

如果输入以下几个记录

5, 6, 11, 14 ✓ (第 1 个记录)

3 ✓ (第 2 个记录)

1, 12, 21 ✓ (第 3 个记录)

得到的结果并不是 I=5, J=6, K=11, M=14, N=3。第一个 READ 语句执行完毕时 14 未被读入, 接着应执行第二个 READ 语句, 但它需从一个新的记录读数, 因此第二个记录的 3 输入给 M, 因为 N 未得到值, 便从下一个记录中继续读数, 将 1 输入给 N, 后面的 12, 21 未被读入, 不起作用。

4° 在输入数据时, 用“/”表示数据结束, READ 语句不再继续读数。假如设 A、B、C 均为整型, 它们已分别有值 3、4 和 5。执行语句:

READ *, A, B, C

如输入:

8,9/10✓

在分别将 8 和 9 读到 A 和 B 中后,遇到斜杠,表示不再输入数据给该 READ 语句中的变量,因此 C 未被赋值(仍保留原来的值 5)。

5° 如果 READ 语句中有几个连续的变量需要赋予同一个值,可以用重复因子 r,表示一数据重复出现 r 次,例如对上面的 READ 语句可用下面形式的输入:

3 * 5✓ (它等效于 5,5,5✓)

如果输入:

3 * ✓

即输入三个空格,A、B、C 这三个变量未被输入新值,它们则保留原来的值。

6° 由于空格是分隔两个数据的符号,因此在一个数据的中间不能插入空格。表控输入使用比较方便,建议初学者尽量使用这种输入方式。

7° 输入的数据必须是常数,不能是变量和表达式。

3. 带控制信息表的读语句

带控制信息表的读语句其一般形式是:

READ(说明符 1,说明符 2,...,说明符 n)输入表

这里的说明符是指一些附加功能(如指示读入是否出错,出错如何处理,读到文件结尾又如何处理等等),这些功能每一个可写成一个控制说明符,彼此用逗号分开,全部控制说明符用一对括号括起,写在读关键字与输入表之间,称为控制信息表。控制说明符的一般形式是:

关键字 = 指定参数

例如:

READ(UNIT=5,FMT='(4I3)',IOSTAT=M)A,B,C,D

其中 UNIT=5(部件号)是必需的,用来指定输入设备,UNIT 是关键字,在“=”号后的 5 是部件号,当然也可是一个整型表达式。当使用系统隐含设备时,用 * 代表部件号。

FMT='(4I3)')(格式说明)也是必需的,用来指定编辑格式,关键字是 FMT,在“=”号后填写所需的格式说明。FMT='(4I3)'表明在一行内输入 4 个整数,每个数占 3 位,即输入数据形式为:

8 12 27 48✓

IOSTAT=M(M 是一个整型变量名)这个说明符是可选的,状态关键字是 IOSTAT,在“=”号后为一个任意的整型变量名(也可以是整型数组元素名)。READ 语句执行完后,整型变量 M 的指示状态如下:

$$M = \begin{cases} 0 & \text{(读正常)} \\ \text{正整数} & \text{(读出错)} \\ \text{负整数} & \text{(遇文件结束)} \end{cases}$$

控制信息表内各说明符位置可任意排列,但通常把 UNIT 放在第一项,FMT 放在第二项,这时可把“UNIT=”与“FMT=”省略,只写部件号及格式说明。

其它说明符还有:

[NML=]名字表名

REC= 整数变量

ERR= 标号

END= 标号

ADVANCE= 字符表达式

SIZE= 整型变量

EOR= 标号

这些将在以后的章节将作相应的介绍。

3.3.3 输出语句

1. 基本形式(自定格式)输出

输出语句又称作写语句,其最基本的形式为:

WRITE(部件号,格式说明)输出表

设备、格式和输出对象是三个必备条件。部件号与格式说明的作用与写法同读语句,输出表除了列出变量名、数组名、数组片段名、结构成员名外,还允许是表达式。输出表中各项彼此用逗号分隔。设 $X=1.2, Y=2.3$, 则

WRITE(6, '(2F6.2)')X, Y

中的 6 代表设备号(一般代表打印机或显示器,这里设为显示器),2F6.2 为格式说明,F 表示作实型编辑,F 后的 6 表示每个数占 6 列(小数点也占一列),小数点后边的 2 表示小数部分占 2 列,F 前的 2 表示重复系数。该语句表示在一行内输出 2 个实数,它们分别是 X、Y 的值,每个值占 6 列,小数部分占 2 列,如数据不足 6 列,左侧用空格填足。屏幕显示如下:

1.20 2.30

当设备是系统隐含指定的设备时有两种简化方法。一是部件号用 * 代表,语句的关键字为 WRITE。另一个方法是省略掉设备号,此时关键字换用 PRINT,其一般形式为:

PRINT 格式说明,输出表

用 PRINT 语句实现上述 WRITE 语句的输出:

PRINT '(2F6.2)', X, Y

则在系统隐含指定的设备上输出:

1.20 2.30

注意:缺省设备的写语句关键字是 PRINT,而不是 WRITE。如仍用 WRITE,则会出错。

2. 表控格式输出

写语句的表控格式也用 * 表示,即按系统隐含指定的格式输出:例如

WRITE(6, *)X, Y

' PRINT *, X, Y

分别是在部件号 6 和系统隐含指定的设备上按表控格式输出 X、Y 的值。

说明:

1° PRINT 语句的输出设备由计算机系统隐含指定,而用 WRITE 语句则可以在希望的任何设备上输出,它的用途比 PRINT 语句广泛。

2° 若 WRITE 语句中括号内有两个星号“*,*”,第一个 * 指出输出的设备,表示在系统隐含指定的设备(打印机或显示器)上输出,此时的 WRITE 和第一个星号的作用就相当于 PRINT。第二个 '*' 指出输出格式,表示按系统隐含指定的格式输出,与 PRINT 语句中

“*”的作用相同。例如语句 `WRITE(*,*)X,Y` 和语句 `PRINT *,X,Y` 的作用相同。

3° 如果输出语句中不出现输出表列,则意味着输出一个“空白行”,实现“隔行打印”。如 `PRINT *` 和 `WRITE(*,*)` 都表示输出一个空白行。

4° 不同的系统表控输出有不同的规定。有的规定每个整型量的输出占 13 列,实型量的输出占 18 列;有的规定每个整型量的输出占 10 列,实型量的输出占 16 列。在我们所用的 FTN90 系统中,一个实型量的输出占 11 列(一位小数点,7 位小数),超过 11 列用实数的指数形式表示;整数按实际长度输出(不超过最大值范围),各数之间用一个空格分开;字符型数据之间无分隔符号。

3. 带控制信息的输出语句

带控制信息的输出语句的一般形式为:

`WRITE(说明符 1,说明符 2,...,说明符 n)输出表`

控制信息表内各说明符的写法和功能与输入语句控制信息表基本相同。例如 $X=5.9$, $Y=7.8$ 时

`WRITE(UNIT=6,FMT='(1X,2F8.2)',IOSTAT=M)X,Y`

表示在部件号 6 的设备上按每个数据占 8 列、小数部分占 2 列的格式在一行内输出 X 和 Y 的值。当输出正常时, M 为零;输出出错时, M 为正数。其数据所占的格式是:

5.90 7.80

上面语句等价于:

`WRITE(6,'(1X,2F8.2)',IOSTAT=M)X,Y`

此时 `UNIT` 处于第一项,`FMT` 处于第二项,这两项关键字可以省略。若使用表控格式,该语句可改写成:

`WRITE(6,*,IOSTAT=M)X,Y` 或 `WRITE(*,*,IOSTAT=M)X,Y`

3.4 输入与输出编辑符

人们总是希望对各种类型的数据都能按要求实现输入与输出。FORTRAN 语言中输出格式功能非常强,与之相应的格式编辑符就比较多,用户可以根据需要输入和输出各种不同格式的数据。由于格式编辑符多,使用时需遵守的规则也很多,稍不注意,就会因不匹配而出错,因此先介绍几个常用的编辑符及其使用的规则。编辑符描述中大写字母是关键字,给出编辑类型,需要照写;小写字母供程序员按需要填写,方括号[]内的内容表示是可选项。

3.4.1 常用编辑符

1. 整型编辑符: `Iw[.m]`

`I` 编辑符专门用于整型数据的格式说明, w 表示字段总宽度, m 表示至少有 m 位数字。如果该整数的实际位数(包括符号)不足 w 位,则左面补以空格。如果多于 w ,则无法正确表示出该数,以 w 个“*”表示格式不够宽。当选用 `Iw.m` 格式时,如果该整数实际位数超过 m 位,则按实际长度打印,不足 m 位时用前置零填满。注意: m 位数字不包括负号。

设 $I=12345$, $J=-24$, $K=24688$, $L=43216$, $N=-275$ 。有以下语句:

① `PRINT '(I5,I6,I4)',I,J,K`

② PRINT '(I10.4,I10)',L,L

③ PRINT '(I10.4,I10)',N,N

执行第①条语句后输出结果为

12345 -24 * * * *

其中,K 的实际长度为 5 位,但格式编辑符为 I4,只提供 4 列位置,无法容纳 24688 这 5 位数字,则输出 4 个星号,表示出现“字段不宽度”的错误。第②条语句输出结果为

43216 43216

L 的实际位数超过 $m(m=4)$ 则按实际位数输出。执行第③条语句后输出结果为

-0275 -275

N 的实际位数不足 $m(m=4)$ 位,则前置 1 个零(负号不包含在 m 位内)。

在 I 编辑符中,显然有 $m \leq w$ 。如果输出的是负数, m 应小于 w 。

2. 实型编辑符:Fw.d

F 编辑符用于实型数据的格式说明,它使数据以“小数形式的实数来表示”。其中 w 表示字段总宽度(包括小数点 1 列), d 为该实数小数位数。设有实型数 $X=10.56$, $Y=-2.7$,语句

PRINT '(F10.3,F8.4)',X,Y

的输出格式为:

10.560 -2.7000

数据不足 w 位时用前置空格补全,小数位数少于 d 位时,用尾零补足。如小数位数超过 d 位,则对小数后第 $d+1$ 位产生截断并进行四舍五入。如整个数位超过 w ,则输出 w 个星号。

当 Fw.d 用在读语句中作格式说明时,输入数可以用上述格式,中间没有逗号;也可以按数学习惯输入,各数间用逗号分开。例如,输入形式为:

10.56, -2.7

但每个数连同后随逗号不能超过 w 列,否则出错。

说明:应该恰当选择 w 和 d 的值。如果要输出的数值比较小(如 $A=3.1415925$, $B=0.000125$),若 d 选小了,如用(F10.1,F10.2),对后面小数进行四舍五入就会丢掉一些有效数字而产生较大的误差,即“小数印丢”。如果要输出的数值比较大,而所选 w 不够大或 d 太大时,会出现“字段不够宽”的错误。如 $C=21345678.12$,用 F10.6 输出,为保证 6 位小数 w 应至少选择 15,但仅规定了 $w=10$,显然不够,整个字段以“*”充满,这便是“大数印错”。

3. 实型编辑符:Ew.d[Ee]

E 编辑符也用于实型数据的格式说明,数据按指数形式输出。 w 为字段宽度, d 为尾数部分列数。整个数是一个规格化的小数后跟 10 的 n 次幂,它的指数部分通常占 2 列。如果希望指数部分放大,占 4 列或 5 列,就在可选 E 项之后的 e 处填上 4 或 5, e 规定该实型数输出时指数部分所占列数。设实型数 $X=212.345$,则语句 PRINT '(E12.3)',X 的输出结果为

0.212E+03

而语句 PRINT '(E12.3E4)',X 的输出结果为

0.212E+0003

可以看出,输出是以指数形式表示的实数。如果不规定指数部分的位数,指数部分一般

占4列,其中“E”占1列,符号占1列,指数占2列,(Ew.d中的)d指出数据的小数部分中小数的位数,一律以标准化的指数形式表示。w的值应该不小于d+6,因为有d位小数,再加一个“小数点”,小数点前有一位非零数字,还有四位是指数部分,所以共需要d+6列。如果输出的是负数,则应满足 $w \geq d+7$ 。

当Ew.d用于输入时,输入数据也有两种形式。一是严格按上述格式输入,另一个是仍按数学习惯书写实数,各数间用逗号分隔,其规则同使用Fw.d一样。

说明:

1° 实型数据不能用I型编辑符输入与输出,同样整型数据不能用F或E编辑符。

2° 一个实型数据可以用F编辑符输入与输出,也可以用E编辑符输入与输出,程序员可根据需要选F型或E型编辑符进行输入与输出。

3° 有的系统采用了两种标准化指数形式输出。如果表控格式输出的实数超过11列时用指数形式输出:其小数部分的小数点前有且仅有一位非零数字;如果直接采用E格式输出:其小数部分的小数点前没有非零数字,小数点后第一位是非零数字。例如程序段

```
Y = -123.45678912      ! Y是实型变量
PRINT *, Y             ! 表控输出
PRINT '(E15.6)', Y     ! 自定格式输出
```

的输出结果为

```
-1.2345679E+02
-0.123457E+03
```

4. 复型编辑

FORTRAN中复型数据是用两个实型编辑符(例如2Fw.d或2Ew.d[Ee])来分别描述该复型的实数部分与虚数部分的。设X是复型变量,其值为(2.8,4.6),可用下述语句输出X的值:

```
PRINT '(2F4.1)', X
```

输出结果为:

```
2.8 4.6
```

5. 字符型编辑:A

A用来编辑字符型变量,它不关心该字符串的长短如何。例如要输出:“There are 5 books.”字样,输出表共有3项,字符串'There are ',整型变量I值,字符串'books.'。在格式说明中有两个字符型编辑符A,一个整型编辑符,语句应写成:

```
PRINT '(A,I2,A)', 'There are ', I, ' books.'
```

6. 其它编辑符:X,/和:

X编辑符表示输出空格,为了避免两个相邻数据紧连在一起,可以用X编辑符在数据之间插入一些空格。它的一般形式为nX,n为重复系数,是希望插入的空格个数。

设X=1.1,Y=1.2,语句

```
PRINT '(F3.1,5X,F3.1)', X, Y
```

的输出格式为:

```
1.1    1.2
```

“/”编辑符的作用是:结束正在输出的记录并开始下一行的输出。如语句

```
PRINT '(F3.1/F3.1)', X, Y
```

的输出格式为:

1.1

1.2

在“/”后输出的数据 Y 放在下一行输出。

“:”编辑符的作用是:当输入表、输出表输完后,阻止后面的编辑符起作用。该编辑符在输入表/输出表未结束之前,不起作用。假设 $X=1, Y=2, Z=3$, 要想在输出数据之间用“+”号连接,可用如下语句实现:

```
PRINT '(I1:"+",I1:"+",I1:"+")',X,Y,Z
```

这时输出格式为:

1+2+3

输出 X 的值 1 后,由于输出尚未结束,“:”编辑符不起作用,遇 '+' (+号前后每两个撇号起一个撇号的作用)就打印加号。而输出 Z 的值 3 后,输出表已输出完毕,“:”编辑符就起作用,阻止后面的加号输出。

3.4.2 有关一个记录的编辑格式说明

1. 一个记录的编辑格式由它所需的各种类型编辑符组合在一起,彼此用逗号分开,并用括号括起来,再在外面加上单撇号(或双撇号)作定界符。例如:

```
'(编辑符 1,编辑符 2,...,编辑符 n)'
```

语句

```
PRINT '(F3.1,' AND ',F3.1)',X,Y ! 假设 X=1.1,Y=1.2
```

中,字符串 ' AND ' 用单撇号(由于出现在格式串之内,用两个连续的单撇号表示,也可以用双撇号)作定界符,输出结果为:

1.1 AND 1.2

如果要输出带撇号的字符常数,可以用两个连续的撇号作为一个字符“'”(撇号)。例如想输出:

THIS IS MY SISTER'S BOOK.

可写语句

```
PRINT '(A)', 'THIS IS MY SISTER'S BOOK.'
```

2. 编辑符的类型必须与输入(或输出)表中对应位置数据的类型一致,否则出错。如:

```
'(I4,F5.1,E10.3)'
```

规定一行内输入或输出三个数。如果一行输完后,输出(输入)表内还有等待输出(输入)的变量名,则按此行数据格式在下一行继续输出(输入),直到输完为止。例如:

```
PRINT '(I4,F5.1,E10.3)',A,B,C,D,E,F,G,H
```

则按如下顺序打印各项数据:

A	B	C
D	E	F
G	H	

其中第一列均为整数形式,第二列均为实数形式,每行格式一样。为使编辑符与对应位置上变量的类型一致,A、D、G 应是整型变量,B、C、E、F、H 应是实型变量,否则出错。

3. 记录编辑格式中如有相连的几个相同的编辑符,则可简化为一个编辑符,但在前边需要加上重复系数,例如:

```
PRINT '(F6.1,F6.1,I4,I7,I7,I7)',A,B,C,D,E,F
```

也可写成:

```
PRINT '(2F6.1,I4,3I7)',A,B,C,D,E,F
```

4. 相同的编辑字符串连续出现时,可简化为一串,用括号括起,前置重复系数。例如:

```
PRINT '(F5.1,2I4,F5.1,2I4,E10.3)',A,B,C,D,E,F,G
```

也可写成:

```
PRINT '(2(F5.1,2I4),E10.3)',A,B,C,D,E,F,G
```

5. 在读写语句中可没有变量名表,但不可没有记录格式说明,此时的写语句输出一空行,读语句执行时则读入一空行(即按一下回车键)。例如下列语句是合法的:

```
WRITE(10,'(3F5.1)',IOSTAT=L)
```

```
PRINT '(4I3)'
```

```
PRINT *
```

```
READ *
```

3.4.3 应用举例

[例 3-3] 已知三角形的三条边长 a 、 b 和 c ,求三角形面积。

求面积公式为: $\text{area} = \sqrt{s(s-a)(s-b)(s-c)}$, 其中: $s = \frac{1}{2}(a+b+c)$ 。

程序如下:

```
PROGRAM MAIN AREA
```

```
IMPLICIT NONE
```

```
REAL::A,B,C,S,AREA
```

```
READ *,A,B,C
```

```
S=(A+B+C)/2.0
```

```
AREA=SQRT(S*(S-A)*(S-B)*(S-C))
```

```
WRITE(*,'(3(A,F6.3),A,F8.3)')A='A,' B='B,' C='C,&
```

```
' AREA=' ,AREA
```

```
END PROGRAM MAIN AREA
```

运行结果为:

4.567,8.123,6.013✓

A= 4.567 B= 8.123 C= 6.013 AREA= 13.546

[例 3-4] 读入 5 个实数,要求打印出绝对值最大和绝对值最小的那两个数的绝对值。

```
PROGRAM MAIN
```

```
IMPLICIT NONE
```

```
REAL::A,B,C,D,E,BIG,SMALL
```

```
READ *,A,B,C,D,E
```

```

BIG = MAX(ABS(A),ABS(B),ABS(C),ABS(D),ABS(E))
SMALL = MIN(ABS(A),ABS(B),ABS(C),ABS(D),ABS(E))
PRINT '(2(A,F8.2))','MAX= ',BIG,' MIN= ',SMALL
END PROGRAM MAIN

```

运行结果为:

123.7, -21.56, 468.1, 1482.6, -331.7✓

MAX= 1482.60 MIN= 21.56

程序中 ABS 为绝对值函数,自变量 A、B、C、D 和 E 均为实型,因而 ABS(A)、ABS(B)、ABS(C)、ABS(D)和 ABS(E)的函数值也为实型。MAX 为求两个或两个以上变量中的最大值。今有 5 个实型自变量,则 MAX 的函数值也是实型。MIN 是求最小值的函数,情况与 MAX 类似。

习 题 3

3-1 分别为下列变量写出说明语句:

- (1) X、Y 是整型,种别参数是 2。
- (2) W 是一个一维数组、实型,种别参数是 4(W 的数组属性是 DIMENSION(1:8))。
- (3) C 是一个复型变量,种别参数取标准值。
- (4) A = REAL(30_3) + REAL((1,2)) (A 是实型)
- (5) C = 3 (C 是整型)

3-2 写出下列数学式子的 FORTRAN 表达式:

- (1) $(x-y)^4$
- (2) $2a+5$
- (3) $\frac{a+b}{c \cdot d}$
- (4) $(1.5+a)(2+b)$
- (5) $\cos^2 3x$
- (6) $\frac{1}{p} \sin(2p+x)$
- (7) $a \cdot b + c \cdot d$
- (8) $e^x \cdot \frac{1}{y}$
- (9) $10^{-3} \cdot e^{(x+2)}$
- (10) $\frac{12.7 + \log_{10} 18}{\sqrt{x+y}}$

3-3 指出执行下列赋值语句后,左边变量的数值及应采用的种别(假定种别参数愈大,精度与表达值的范围愈大)。

- (1) I = 30_1/18_1 (I 是整型)
- (2) I = INT(3.9) (I 是整型)
- (3) A = REAL(30_3) + REAL((1,2)) (A 是实型)
- (4) C = 3 (C 是整型)
- (5) X = (0.3_3 * 2.0 + 1.5_4/0.3)/0.2 (X 是实型)

3-4 有一表控输入语句:

```
READ *, A, B, C
```

如果输入以下记录,问 A、B、C 的值各为多少?

- (1) 87, 65, -173✓

(2)87 65 -1 73✓

(3)87✓

65 73 64✓

(4)87,65,73,84✓

3-5 假设以下两个 READ 语句中的变量均为实型:

READ *,X,Y,Z,A

READ *,C,F,G,H,Z

输入数据为

16.8 21.78 -23.4✓

67.31,21.3 6, 560.3 72,123.7,-56.5,10.6✓

21.3 6, 560.3 72,123.7,-56.5,10.6✓

时,各变量的值是多少?

3-6 写出用下列编辑描述符输出时的结果。

机内数值	编辑符	机内数值	编辑符
234	I6	-8760	I8
-6721	I4	84	I6.4
212.6364	F8.2	1.321	F10.8
2E5	F15.3	621.4	F6.3
6.52	E10.2	-1.5E-2	E8.1
12.46	E14.2E3	12345.6545	E18.3E4

3-7 设实型变量 X 的初值为 0.1,Y 的初值为 0.2,常数 a 的值是 0.3,编程求这三个数之和及 $(X+Y)/a$ 之商。

3-8 按可读性要求编一程序,求二次方程的根: $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ 。对读入的 a、b、c,求两个实根(假设 $b^2 - 4ac \geq 0$)。

3-9 编一程序,用表控格式、缺省设备号的读语句读入三个数 0.1、0.2、3,分别赋给实型变量 X、Y 和整型变量 Z,再用自定义形式输出 X、Y、Z,令 X 值宽 10 列,小数部分占 2 列,Y 是指数型实数,宽 12 列,小数占 3 列,Z 的宽度是 7 列。分两种情况编程输出:

(1)在一行中输出;

(2)分两行输出,一行输出两个实数,一行输出一个整数。

3-10 已知梯形的上底 $a=112$ 公分,下底 $b=240$ 公分,高 $h=150$ 公分,求梯形面积(米² 以为单位表示,取一位小数,对第二位小数进行四舍五入)。分两种情况编程:

(1)用格式输入和格式输出;

(2)用表控输入和表控输出。

3-11 有一扇形如图 3-3。已知其半径 r 及圆心角 θ 。弦长 $b=2r\sin \frac{\theta}{2}$,拱高 $h=2r\sin^2 \frac{\theta}{4}$,弧度 $a=\frac{2\pi\theta}{360}$,面积 $s=\frac{1}{2}ar^2 - a\sqrt{r^2 - a^2}$ 。编一程序,对任意给定的 r 及 θ ,输出 h 、 b 及 s 的值。

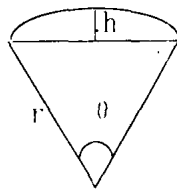


图 3-3 习题 3-11 图

第 4 章 选择结构程序设计

选择结构的逻辑功能如图 1-8 所示。在 FORTRAN90 中,可以用 IF 语句、IF 结构和 CASE 结构来实现选择结构程序设计。

4.1 关系表达式与逻辑表达式

选择结构是通过条件判断来决定程序下一步应该做什么。这就需要“条件”的值有两种可能:条件满足和条件不满足。FORTRAN 语言中的“条件”是一个逻辑表达式。我们先来讨论逻辑表达式中最简单的形式:关系表达式。

4.1.1 关系表达式

关系表达式是形如 $X < 0$ 和 $Y \geq Z - 6$ 这样的式子,其一般形式为:

 <算术表达式><关系运算符><算术表达式>
或 <字符表达式><关系运算符><字符表达式>

其中关系运算符有 6 种:

$>$	(.GT.)	大于	\geq	(.GE.)	大于或等于
$<$	(.LT.)	小于	\leq	(.LE.)	小于或等于
$=$	(.EQ.)	等于	\neq	(.NE.)	不等于

括号内的关系运算符是 FORTRAN77 及以前版本使用的,现已不提倡使用。每个关系表达式中的关系运算符只准许出现一次,其两侧或者都是算术表达式,或者都是字符表达式。复型数据只能使用“=”和“/=”两种关系运算符。以下都是合法的关系表达式:

$(A + B) > (6 * \text{SQRT}(2.8 + X))$

$5.0 - Y \leq \text{LOG}(X) / \text{SIN}(X)$

$'A' < 'B'$

$'ABC' \geq '0123'$

关系表达式的运算次序是:先分别计算两侧算术表达式或字符表达式的值,然后按关系运算符进行比较。因此关系表达式

$(A + B) > (6 * \text{SQRT}(2.8 + X))$

也可写成

$A + B > 6 * \text{SQRT}(2.8 + X)$

算术表达式加括号与不加括号的作用是相同的。

关系表达式的值(比较的结果)不是一个数值,而是一个逻辑值,即“真”或“假”。例如当 $A = 5.5, B = 1.6$ 时,关系表达式

$A > B$

的值为“真”。如果 A 的值为 0.5,该关系表达式的值为“假”。为了便于理解,可以将“真”理

解为“条件满足”,将“假”理解为“条件不满足”。

注意:

1° 若算术关系表达式两端的算术表达式的类型不同,在执行时会自动转换成同一类型(转换的规律与混合运算的转换规律相同,即低级的向高级的转换)。如果 Z 为实型变量,则对关系表达式

$$Z \geq 5 - 2$$

系统会先计算出 $5 - 2$ 的结果 3,并将它转换成实数 3.0 后再与 Z 做比较。

2° 关系表达式中只允许有一个关系运算符,不可随意使用数学中的表示方法。例如:

$$3 < X < 7$$

是不正确的。

3° 在用“=”或“/=”对两个实数进行比较时,有时会出现一些小误差。例如,将 10 个 0.1 累加,结果可能不等于 1,这是由于实数在内存中以二进制形式存放引起的误差。0.1 是不可能用有限位的二进制数准确地表示的,因此 $10 * 0.1 = 1$ 或 $1.0 / 3.0 * 3 = 1$ 这种关系表达式的值可能不是“真”而是“假”。编程时应估计可能出现的误差。如果从理论上计算出 A 应等于 B,这时在程序中应使用 $ABS(A - B) < \epsilon$ 来代替条件判断 $A = B$;用 $ABS(A - B) > \epsilon$ 来代替条件判断 $A \neq B$ 。即当 $A - B$ 的绝对值小于某一个很小的正数 ϵ 时,便认为此两数近似相等;当这两个数的差的绝对值大于或等于这个很小的正数 ϵ 时,才认为此两数不相等。 ϵ 的大小应根据实际问题的需要来选取,一般情况下可选 $\epsilon = 10^{-6}$ 。

4.1.2 逻辑表达式与逻辑赋值

1. 逻辑常数

逻辑型常数只有两个:

.TRUE. “真”,表示满足逻辑条件。

.FALSE. “假”,表示不满足逻辑条件。

注意:.TRUE. 和 .FALSE. 两边各有一点,必不可少,否则会被系统误作变量名。

2. 逻辑变量

逻辑型变量用来存放逻辑值。因此,逻辑变量的取值也只有两种可能:.TRUE. (真) 或 .FALSE. (假)。

逻辑型类型说明的关键字是 LOGICAL,说明逻辑型变量的类型说明语句的一般形式为:

LOGICAL[(KIND= 整型常数表达式)][,属性说明 1, ..., 属性说明 n]::变量表

“(KIND= 整型常数表达式)”说明这些逻辑型变量的种别参数。各属性说明说明各变量的属性,个数不限。例如

LOGICAL(KIND=2), DIMENSION(1:8)::L

说明 L 是一个一维逻辑型数组,有 8 个元素:L(1)、L(2)、……、L(8),它们都用于存放种别参数为 2 的逻辑型数据。

被说明的变量表内列出要说明为逻辑型的对象,可以是一般变量名、数组名、结构成员名等,彼此用逗号分开,并可以对变量置以逻辑型初值。例如要说明 L1 和 L2 为逻辑型变量,并且置 L1 初值为真(.TRUE.)、L2 初值为假(.FALSE.),可写成:

LOGICAL::L1=.TRUE.,L2=.FLASE.

3. 逻辑表达式

逻辑表达式由逻辑操作数和逻辑运算符按一定的语法规则构成。它主要用于选择结构的条件判断中。

(1) 逻辑操作数

参与逻辑运算的逻辑操作数有四种:逻辑常数、经 LOGICAL 语句说明的变量、关系表达式和逻辑表达式。

(2) 逻辑运算符

逻辑运算符共有五个:

.AND. (逻辑与) .OR. (逻辑或) .NOT. (逻辑非)

.EQV. (逻辑等) .NEQV. (逻辑不等)

表 4-1 为以上五种逻辑运算的逻辑值表。

表 4-1 逻辑运算表

A	B	.NOT. A	.NOT. B	A. AND. B	A. OR. B	A. EQV. B	A. NEQV. B
真	真	假	假	真	真	真	假
真	假	假	真	假	真	假	真
假	真	真	假	假	真	假	真
假	假	真	真	假	假	真	假

例如:

$X < 0$

表示当 $X < 0$ 时,该逻辑表达式的值为真;

.NOT. .FALSE.

该逻辑表达式的值恒为真;

.NOT. A. AND. $X - Y > G$

当逻辑变量 A 为假,而且 $X - Y > G$ 时,该逻辑表达式的值为真;

$A \neq D$.OR. .NOT. $U = V$ 当 $A \neq D$ 或者 $U \neq V$ 时,该逻辑表达式的值为真;

$L1$.EQV. $A + B \leq X * 2$ 当逻辑变量 $L1$ 为真且 $A + B \leq X^2$ 时,或者 $L1$ 为假且 $A + B > X^2$ 时,该逻辑表达式的值为真。

表 4-2 逻辑表达式中各运算符的运算优先次序

运算符类别	运算符(或称操作符)	优先级
括号	()	1
算术运算符	* (乘方)	2
	*, /	3
	+, -	4
关系运算符	>, >=, <, <=, =, /=	5
逻辑运算符	.NOT.	6
	.AND.	7
	.OR.	8
	.EQV., .NEQV.	9

逻辑值的相等比较或不等比较只能用逻辑运算符.EQV. 或.NEQV., 而不能用关系运算符=和/=. 逻辑值不做大于或小于等类的比较。

(3)逻辑表达式的运算次序

在一个逻辑表达式中往往包含算术运算符、关系运算符和逻辑运算符,它们的运算优先次序见表4-2。

例如,若 $A=1.5, B=2.0, C=1.2, D=7.5, X=3.0, Y=5.0, L1=.TRUE.$, 有逻辑表达式:

$$A > 2.8 * B .AND. X \geq Y .OR. L1 .AND. .NOT. (3.6 - C) * 2 \geq D / 2.5$$

		①2.4	
②5.6		②4.8	②3.0
③.FALSE.	③.FALSE.	③.TRUE.	
		④.FALSE.	
⑤.FALSE.		⑤.FALSE.	
⑥.FALSE.			

运算次序为:

- ①先算出括号内的值, $(3.6 - C)$ 的值为 2.4。
- ②算出算术表达式 $2.8 * B$ 、 $(3.6 - C) * 2$ 和 $D / 2.5$ 的值, 它们分别为 5.6、4.8 和 3.0。
- ③算出关系表达式 $A > 2.8 * B$ 、 $X \geq Y$ 和 $(3.6 - C) * 2 \geq D / 2.5$ 的值, 它们分别是.FALSE. (假)、.FALSE. (假) 和.TRUE. (真)。
- ④进行.NOT. 运算, .NOT. $(3.6 - C) * 2 \geq D / 2.5$ 的值为.FALSE. (假)。
- ⑤进行.AND. 运算, .OR. 前面和后面的两个逻辑表达式的值均为.FALSE. (假)。
- ⑥最后进行.OR. 运算, 得到整个逻辑表达式的值为.FALSE. (假)。

注意:

1°两个逻辑操作数不可直接相邻, 中间必须有逻辑运算符隔开。如不等式 $0 < X < 1$ 应写成逻辑表达式 $X > 0 .AND. X < 1$, 若写成 $X > 0 X < 1$ 便是错误的, 写成 $0 < X < 1$ 也是不对的。

2°两个逻辑运算符也不可直接相邻, 中间必须有逻辑操作数隔开, 只有.NOT. 例外。如 $A .AND. .NOT. B$, 此时.NOT. B 是作为一个逻辑操作数跟在.AND. 后面。

4. 逻辑赋值

逻辑赋值的一般形式为

$$V = e$$

V 可以是逻辑型的变量、数组元素、数组、数组片断等。若 V 是逻辑型的变量或数组元素, 则 e 必须是一个逻辑表达式。例如:

$$A = X > Y$$

$$B3 = U .OR. G .AND. Z - 8 < = 2.2$$

$$F = .TRUE.$$

4.1.3 逻辑型数据的输入输出

1. 用表控格式输入输出逻辑型数据

假设 L1 和 L2 已说明为逻辑型变量,有以下语句:

```
READ *,L1,L2
```

```
PRINT *,L1,L2
```

如果输入:

T,F✓

则 L1 的值为.TRUE.,L2 的值为.FALSE.。执行输出语句时,用 T 代表.TRUE.,用 F 代表.FALSE.。以上语句的输出为:

T F

用表控格式输出逻辑值时,T 和 F 所占的字段宽由具体的计算机处理系统规定。

2. 用逻辑型格式输入输出逻辑型数据

用 L 编辑描述符规定逻辑数据输入输出的格式。例如:

```
PROGRAM TEST
```

```
IMPLICIT NONE
```

```
LOGICAL::A,B,C,D
```

```
READ '(L5,L3)',A,B
```

```
C=.TRUE.
```

```
D=.FALSE.
```

```
PRINT '(2L3,2L2)',A,B,C,D
```

```
END PROGRAM TEST
```

运行结果如下:

F T✓

F T T F

则执行 READ 语句之后,A 被赋值.FALSE.,B 被赋值.TRUE.。

4.2 IF 语 句

了解了关系表达式、逻辑表达式以后,就可以学习选择结构程序设计了。实现选择结构最简单的方法就是 IF 语句。IF 语句的一般形式是:

```
IF (e) S
```

其中 IF 是关键字,e 是一个逻辑表达式,S 是一条可执行语句。

IF 语句的逻辑功能如图 4-1 所示。例如:

```
IF(X>0) PRINT *,X
```

假设 X 的值为 2,则逻辑表达式 $X>0$ 的值为.TRUE.(真),这时输出 X 的值,然后执行 IF 语句的下一条语句;若 X 的值为 0,则逻辑表达式 $X>0$ 的值为.FALSE.(假),这时不执行后面的 PRINT 语句,直接执行 IF 语句的下一条语句。

注意:IF(e)后面只能跟一个可执行语句,而不能跟可执行语句的序列。下面的写法是错误的:

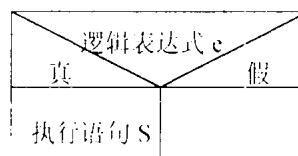


图 4-1 IF 语句的逻辑功能

IF (A<B) T=A;A=B;B=T

它的原意是:如果 A 小于 B,则使 A 和 B 的值互换。但在编译时将显示出“语法错误”的信息。如果一个条件满足后要执行多条语句时,可用 4.3 节介绍的 IF 结构实现。对以下分段函数:

$$Y = \begin{cases} 0 & (x \leq 0) \\ 1 & (x > 0) \end{cases}$$

可以使用 IF 语句来表达:

Y=0

IF(X>0)Y=1

这里先置 Y=0,下面的 IF 语句在 X>0 成立时执行 Y=1;如果 X≤0,则不执行 Y=1,这正反映了上述函数的要求。但其写法不够明显,缺乏易读性,我们要尽量避免这种书写方式,努力把程序写得朴实、简明。改写为

IF(X<=0) Y=0

IF(X>0) Y=1

之后,则非常易读,且不易出错。但这种写法也不该提倡,使用 4.3 节介绍的 IF 结构实现才是最佳的方案。

使用 IF 语句有两点注意:

1° 语句 S 不能是另一个 IF 语句、DO 结构、主程序结束语句、函数结束语句、子程序结束语句等。

2° 无论语句 S 执行与否,都要执行 IF 语句下面的语句,因此要防止下一个语句赋的值覆盖 IF 语句取得的结果,如上面的分段函数用以下两个语句

IF(X<=0) Y=0

Y=1

实现时,不管 X≤0 还是 X>0,Y 的值最后都是 1,这显然是错误的。

4.3 IF 结构

对于多分支的选择结构,或者虽然是二分支但每种选择要执行多条语句时,用 IF 语句来实现就有困难了,而用 IF 结构能够较好地解决这个问题。

4.3.1 IF 结构的一般形式

[例 4-1] 已知 X,求 Y 和 Z 的值,其计算公式分别为:

$$Y = \begin{cases} \sin x & (x > 0) \\ 0 & (x = 0) \\ \lg x & (x < 0) \end{cases}$$

$$Z = \begin{cases} \sqrt{x} & (x > 0) \\ 0 & (x = 0) \\ \cos x & (x < 0) \end{cases}$$

程序如下:

```

PROGRAM YZ_ X
  IMPLICIT NONE
  REAL::X,Y,Z
  PRINT *, 'INPUT X:'
  READ *, X
  IF(X>0) THEN                                ! IF - THEN 语句
    Y= SIN(X)                                ! IF - THEN 块
    Z= SQRT(X)                                !
  ELSE IF(X= =0) THEN                         ! ELSE IF 语句
    Y= 0                                     ! ELSE IF 块
    Z= 0                                     !
  ELSE                                         ! ELSE 语句
    Y= TAN(X)                                ! ELSE 块
    Z= COS(X)                                !
  END IF                                     ! END IF 语句
  WRITE( *, * ) 'X=',X, ' Y=',Y, ' Z=',Z
END PROGRAM YZ_ X

```

在此程序中,如果输入的 X 值是 0.0,因其未经过任何运算,不会有舍入误差,而且 0.0 转换成二进制存入内存时也不会有转换的误差,故在 ELSE IF 语句中判断 X 是否等于 0 时,可以不用“ABS(X)<1E-6”,而直接用“X= =0”做条件。

IF - THEN 语句标志着一个 IF 结构(又称块 IF 结构)的开始,是 IF 结构的唯一入口。END IF 语句标志着一个 IF 结构的结束,是 IF 结构的唯一出口。ELSE IF 语句在 IF 结构中可以多次出现,也可没有。ELSE 语句在 IF 结构中也可以没有,如出现只能出现一次,而且必须在所有的 ELSE IF 语句之后。

IF 结构的一般形式为:

```

IF(e1) THEN                                ! IF - THEN 语句
  块 1                                       ! IF - THEN 块(或称 IF 块)
ELSE IF(e2) THEN                             ! ELSE IF 语句
  块 2                                       ! ELSE IF 块
ELSE IF(e3) THEN
  块 3
...

```

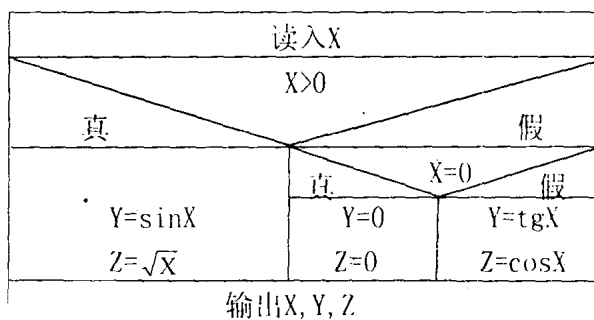


图 4-2 例 4-1 流程图

ELSE IF(e_n) THEN	
块 n	
ELSE	! ELSE 语句
块 n+1	! ELSE 块
END IF	! END IF 语句

其中 $e_1, e_2, e_3, \dots, e_n$ 是逻辑表达式, 指出各种条件, 块 $i(i=1, 2, \dots, n)$ 是一组语句, 其内容是当逻辑表达式 e_i 为真时要执行的程序段。块 $n+1$ 也是一组语句, 如果 e_1 至 e_n 均为假时执行此块中的各条语句。

4.3.2 IF 结构的执行过程

IF 结构的执行过程为:

1. 执行 IF-THEN 语句, 检查 e_1 是否为真。若 e_1 为真, 则执行块 1, 然后转语句 END IF 处出口。
2. 若 e_1 为假, 则执行第一个 ELSE IF 语句, 检查 e_2 是否为真。若 e_2 为真, 则执行块 2, 然后转 END IF 语句处出口。
3. 若 e_2 为假, 则执行第二个 ELSE IF 语句, 检查 e_3 是否为真。若 e_3 为真, 则执行块 3, 然后转 END IF 语句处出口。
- ...

$n+1$. 若 e_n 为假, 则执行块 $n+1$, 然后执行出口语句 END IF 语句; 如果 IF 结构中没有 ELSE 语句, 则直接转出口。

由上可见, 执行 IF 结构时, 自上而下顺次检查每块前面的条件, 满足条件时就执行该条件下面的块, 然后立即转向出口, 以后即使还有满足条件的块, 也不予执行。因此, 一个 IF 结构中最多只能执行一个块, 也可能一块也不执行(例如, 如果 e_1 至 e_n 均为假, 且没有 ELSE 块时就属于这种情况)。

4.3.3 IF 结构的取名

IF 结构可分无名与有名两种, 前面讲的一般形式是无名结构。当 IF 结构太多或彼此嵌套时, 为了阅读时清晰, 可以给它取一个名。IF 结构的取名与变量的取名方法相同, 要尽量反映本 IF 结构的算法。此时 IF 结构的一般形式为:

IF 结构名: IF(e_1) THEN

块 1
ELSE IF(e_2) THEN [IF 结构名]
块 2
... ..
ELSE IF(e_n) THEN [IF 结构名]
块 n

ELSE [IF 结构名]

块 $n+1$

END IF IF 结构名

假设其 IF 结构名为 FIRST, 则入口语句为:

FIRST: IF(e_1) THEN

出口语句为:

END IF FIRST

入口、出口写结构名时, 其结构名字应该一致。IF 结构名与入口语句间要用冒号分隔, 出口语句(END IF 语句)与 IF 结构名之间要空一格, 不能有冒号。ELSE IF 语句及 ELSE 语句后可以写结构名, 也可以不写, 如果写结构名, 其结构名一定要与入口处的结构名一致。

4.3.4 IF 结构中块的缺省及 IF 结构的嵌套

1. IF 结构中块的缺省

在 IF 结构中, IF-THEN 块、ELSE IF 块、ELSE 块都可以缺省, 缺省有两种形式: 一种是该类语句出现, 但后面没有其相应的语句; 另一种是连该类语句一起省略。这两种缺省情况执行的结果可能不同, 要仔细分辨。

(1) 只有一个 IF-THEN 块

IF(e_1) THEN

语句块 1

END IF

若 e_1 值为真, 执行语句块 1 后转出口; 若 e_1 值为假, 则直接转出口。

(2) 缺省 ELSE IF 块

1° 与 ELSE IF 语句一同缺省, 如:

IF(e_1) THEN

语句块 1

ELSE

语句块 2

END IF

当 e_1 为真时执行语句块 1; e_1 为假时执行语句块 2。

2° ELSE IF 语句保留, 如:

IF(e_1) THEN

语句块 1

ELSE IF(e_2) THEN

ELSE

语句块 3

END IF

当 e_1 为真时执行语句块 1; e_1 为假且 e_2 为真时直接转出口; e_1 与 e_2 皆为假时执行语句块 3。

(3)缺省 ELSE 块

```
IF( $e_1$ ) THEN
    语句块 1
ELSE IF( $e_2$ ) THEN
    语句块 2
END IF
```

当 e_1 为真时执行语句块 1; e_1 为假且 e_2 为真时执行语句块 2; e_1 与 e_2 皆为假时直接转出口。

(4)缺省 IF - THEN 块

```
IF( $e_1$ ) THEN
    语句块 1
ELSE IF( $e_2$ ) THEN
    语句块 2
...
ELSE
    语句块  $n+1$ 
END IF
```

当 e_1 为真时,直接转出口;只有当 e_1 为假时,才会一块块检查下去。

注意:IF - THEN 语句不可缺省。另外,缺省 IF - THEN 块和保留 ELSE IF 语句且缺省 ELSE IF 块这两种缺省方式看起来很不舒服,建议通过适当的方法改写成其它的缺省方式。

2. IF 结构的嵌套

IF 结构的任一语句块(IF - THEN 块、ELSE IF 块、ELSE 块)中都可以嵌入另一个结构,如另一 IF 结构、后面将讲到的 CASE 结构、DO 结构等。前提是必须把整个结构完整地嵌在 IF 结构的某一块中,绝不可跨越两块。

[例 4-2] 求一元二次方程 $aX^2 + bX + c = 0$ 的根。

程序如下:

```
PROGRAM ROOT
    IMPLICIT NONE
    REAL::A,B,C,D,X,X1,X2,R,T
    PRINT *, 'INPUT A,B,C:'
    READ *, A,B,C
    D=B*B-4*A*C
    FIRST: IF (ABS(D)<1E-6) THEN
        X=-B/(2*A)
        PRINT *, 'X=', X
    ELSE
    NEXT: IF (D>0) THEN
        X1=(-B+SQRT(D))/(2*A)
        X2=(-B-SQRT(D))/(2*A)
```

```

        PRINT *, 'X1 = ', X1, ' X2 = ', X2
    ELSE
        R = -B/(2 * A)
        T = ABS(SQRT(-D)/(2 * A))
        PRINT *, 'X1 = ', R, '+', T, 'i'
        PRINT *, 'X2 = ', R, '-', T, 'i'
    END IF NEXT
END IF FIRST
END PROGRAM ROOT

```

在 IF 结构嵌套时,为了清晰区分内层与外层结构,一般应对内、外层结构分别取名,并且内层结构要比外层结构后退几格,即写成“锯齿形”,也称“嵌套型”。

上面程序中内层 IF 结构 NEXT 全部嵌入外层 IF 结构 FIRST 的 ELSE 块中,因此是合法的嵌套。

在某些场合,IF 结构嵌套是必需的,但如果嵌套过多,阅读时就要一层层地记住前面各层的条件,容易出错,也不易维护,因此应尽量减少嵌套。方法是把条件分细,列成多句 ELSE IF 语句。例如上面程序中双重嵌套结构可以改写为更具可读性的结构:

```

IF(ABS(D)<1E-6)THEN
    X = -B/(2 * A)
    PRINT *, 'X = ', X
ELSE IF(D>0)THEN
    X1 = (-B + SQRT(D))/(2 * A)
    X2 = (-B - SQRT(D))/(2 * A)
    PRINT *, 'X1 = ', X1, ' X2 = ', X2
ELSE
    R = -B/(2 * A)
    T = ABS(SQRT(-D)/(2 * A))
    PRINT *, 'X1 = ', R, '+', T, 'i'
    PRINT *, 'X2 = ', R, '-', T, 'i'
END IF

```

4.4 CASE 结 构

CASE 结构也是用来形成多分支的,但它仅能按某一个量的划分来选择不同的处理块。

4.4.1 CASE 结构的一般形式

CASE 结构的一般形式是:

```

SELECT CASE(情况表达式)
CASE(情况选择器 1)
    语句块 1

```

```

CASE(情况选择器 2)
    语句块 2
...
CASE(情况选择器 n)
    语句块 n
CASE DEFAULT
    DEFAULT 块
END SELECT

```

结构的第一句是 CASE 结构的入口,标志 CASE 结构的开始。其中情况表达式可以是整型、逻辑型或字符型表达式(注意:不能是实型或复型表达式)。

CASE(情况选择器 i)等一类语句称情况语句,可以有多句。情况选择器只是写出情况表达式所可能取的某一个值或某一组值,其类型与情况表达式类型应一致。当情况选择器内有多个值时,一般把各值一一列出,彼此用逗号分开,如果这些值是连续的,也可用起始值与终止值域表示,形式是

始值:终值

如 CASE(1,2,3)也可写成 CASE(1:3)。

如果值域为从某值开始后所有的值,可写成

始值: 如果值域为某值前(含该值)所有的值,可写成
:终值

每个 CASE 语句后跟随的语句块,称 CASE 块,当选择器中的某个值与情况表达式值一致时被执行。

情况选择器也可为表达式,但不能有重复值,即同一个值不能出现在两个不同的情况选择器中。

当情况表达式是字符型时,情况选择器的值也应是字符型,且种别参数必须与之一致,但长度可以不同。

DEFAULT 也是一种情况选择器,后随语句块。CASE DEFAULT 最多可以有一句,写在所有 CASE 语句之后。

END SELECT 语句是 CASE 结构的出口语句,标志结构结束。

例如,一年中十二个月的天数用 CASE 结构可以表示如下:

```

SELECT CASE(MONTH)
CASE(2)
    DAYS= 28
CASE(4,6,9,11)
    DAYS= 30
CASE(1,3,5,7,8,10,12)
    DAYS= 31
END SELECT

```

4.4.2 CASE 结构的执行过程

CASE 结构的执行过程为:

1. 计算情况表达式的值。
2. 计算第一个 CASE 语句情况选择器内各表达式的值,如其中某一个值与情况表达式值相等,则执行语句块 1,然后转出口,否则转下一步。
3. 检查下一个 CASE 语句的情况选择器,满足则执行语句块 2,然后转出口;否则再查下一个 CASE 语句的情况选择器,如此反复,直到全部情况选择器值都检查完。
4. 如果所有情况选择器中都没有情况表达式的值,若有 CASE DEFAULT 语句,则执行 DEFAULT 块后转出口,否则直接转出口。

4.4.3 CASE 结构的命名

CASE 结构也分成无名与有名两种,无名的一般形式如前。有名的 CASE 结构名与一般变量的取名规则相同,使用结构名的方法与标识 IF 结构名的方法基本相同。有名 CASE 结构的一般形式为:

```
结构名:SELECT CASE(情况表达式)
        CASE(情况选择器 1) [结构名]
            语句块 1
        ...
        CASE(情况选择器 n) [结构名]
            语句块 n
        CASE DEFAULT      [结构名]
            语句块 n+1
    END SELECT 结构名
```

其中入口语句、出口语句及 CASE 语句的结构名要一致,CASE 语句的结构名可以省略。

4.4.4 非数值型的情况表达式

1. 字符型情况表达式

编一程序,根据考试成绩的等级打印出对应的百分制分数段。用 CASE 结构实现时其主要部分如下:

```
SELECT CASE(GRADE)
    CASE('A')
        PRINT *, '90 --- 100'
    CASE('B')
        PRINT *, '80 --- 89'
    CASE('C')
        PRINT *, '70 --- 79'
    CASE('D')
        PRINT *, '60 --- 69'
```

```

CASE DEFAULT
PRINT *, '<60'
END SELECT

```

2. 逻辑型情况表达式

当情况表达式为逻辑表达式时, CASE 语句中的情况选择器必须为逻辑值, 即只能为 .TRUE. 或 .FALSE.。

设有分段函数

$$Y = \begin{cases} x+5 & (x \geq 0) \\ -x & (x < 0) \end{cases}$$

用 CASE 结构可编写如下:

```

SELECT CASE(X >= 0)
CASE(.TRUE.)
Y = X + 5.0
CASE(.FALSE.)
Y = -X
END SELECT

```

CASE 结构也可以嵌套, 并可与其它结构嵌套, 嵌套的形式和规则与 IF 结构相同。

4.5 程序举例

[例 4-3] 输入三个数 A、B、C, 把它们按从大到小的顺序输出出来。

程序如下:

```

PROGRAM SEQUENCE
IMPLICIT NONE
INTEGER::A,B,C
PRINT *, 'INPUT A,B,C:'
READ *, A,B,C
IF(A>B)THEN
IF(B>C)THEN
PRINT *, A,B,C
ELSE
IF(A>C)THEN
PRINT *, A,C,B
ELSE
PRINT *, C,A,B
END IF
END IF
ELSE
IF(B<C)THEN
PRINT *, C,B,A

```

```

ELSE
  IF(A>C)THEN
    PRINT *,B,A,C
  ELSE
    PRINT *,B,C,A
  END IF
END IF
END IF
END PROGRAM SEQUENCE

```

运行结果如下:

INPUT A,B,C:

3,1,2↙

3 2 1

[例 4-4] 有一函数

$$Y = \begin{cases} 0 & (x_0 \leq x < x_1) \\ \sqrt{x - x_1} & (x_1 \leq x < x_2) \\ kx & (x_2 \leq x < x_3) \\ \frac{e^{-x}}{x} & (x_3 \leq x) \end{cases}$$

程序如下:

```

PROGRAM FUNY
  IMPLICIT NONE
  REAL::K,X0,X1,X2,X3,X,Y
  PRINT *, 'INPUT K,X0,X1,X2,X3:'
  READ *, K,X0,X1,X2,X3
  PRINT *, 'INPUT X:'
  READ *, X
  IF(X>=X0)THEN
    IF(X>=X3)THEN
      Y=EXP(-X)/X
    ELSE IF(X>=X2)THEN
      Y=K*X
    ELSE IF(X>=X1)THEN
      Y=SQRT(X-X1)
    ELSE
      Y=0.0
    END IF
    WRITE(*,*)'X=',X,' Y=',Y
  ELSE

```

```

WRITE( *, * )'X<',X0
END IF

```

```

END PROGRAM FUNY

```

先输入 K、X0、X1、X2 和 X3 的值,在计算公式中它们是常数。然后再输入 X 的值,X 的值应大于等于 X0,每次可取不同的值。

运行记录如下:

```

INPUT K,X0,X1,X2,X3:

```

```

2.0,10.0,20.0,30.0,40.0✓

```

```

INPUT X:

```

```

15.0✓

```

```

X= 15.0000000 Y= 0.0000000E+00

```

再运行一次,结果如下:

```

INPUT K,X0,X1,X2,X3:

```

```

3.0,-20.0,-10.0,5.0,28.0✓

```

```

INPUT X:

```

```

22.0✓

```

```

X= 22.0000000 Y= 66.0000000

```

[例 4-5]输入一个数,判断它是否能被 3 整除,并打印出相应的信息。

1. 用 CASE 结构来实现

```

PROGRAM JUDGE

```

```

IMPLICIT NONE

```

```

INTEGER::N,M

```

```

READ( *, * )N

```

```

M=MOD(N,3)

```

```

SELECT CASE(M)

```

```

CASE(0)

```

```

WRITE( *, '(I6,2X,A)') N,'CAN BE DIVIDED EXACTLY BY 3.'

```

```

CASE DEFAULT

```

```

WRITE( *, '(I6,2X,A)') N,'CAN NOT BE DIVIDED BY 3.'

```

```

END SELECT

```

```

END PROGRAM JUDGE

```

运行结果如下:

```

36✓

```

```

36 CAN BE DIVIDED EXACTLY BY 3.

```

2. 用 IF 结构来实现

只需把原 CASE 结构改为如下结构即可:

```

IF (M == 0) THEN

```

```

WRITE( *, '(I6,2X,A)') N,'CAN BE DIVIDED EXACTLY BY 3.'

```

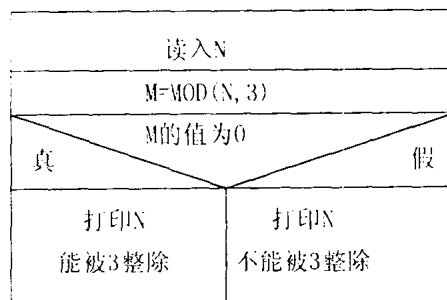


图 4-3 例 4-5 流程图

ELSE

WRITE(*, '(I6,2X,A)') N, 'CAN NOT BE DIVIDED BY 3.'

END IF

习题 4

4-1 写出下列逻辑表达式的值,假设 $A=2, B=7.5, C=-3.6, L1=.TRUE., L2=.FALSE.$ 。

(1) $A-7 < B-6.5$

(2) $\text{NOT}.L2. \text{OR}. B-A \leq C/2. \text{AND}. C > -3.6$

(3) $L2$

(4) $L1. \text{EQV}. \text{NOT}. L2. \text{AND}. L1. \text{OR}. 3 * A \leq 4 - B$

(5) $\text{ABS}((C-A) - (B * A - C)) < 1E-6). \text{OR}. L1$

(6) $(A + C > B. \text{AND}. C * 2 > 10.0). \text{NEQV}. (L1. \text{OR}. L2)$

4-2 输入一个学生的成绩 G 。如果 $G \geq 90$, 则输出“VERY GOOD”; 如果 $80 \leq G < 90$, 则输出“GOOD”; 如果 $60 \leq G < 80$, 则输出“PASS”; 如果 $G < 60$, 则输出“FAIL”。

4-3 有一函数:

$$Y = \begin{cases} \frac{40}{15} + x & (0 \leq x < 15) \\ 50 & (15 \leq x < 30) \\ 50 - \frac{10}{15}(x - 30) & (30 \leq x < 45) \\ 40 + \frac{20}{30}(x - 45) & (45 \leq x < 75) \\ 60 - \frac{10}{15}(x - 75) & (75 \leq x < 90) \end{cases}$$

编写程序, 输入一个 X 之后, 打印出其相应的 Y 值。

4-4 有一函数:

$$Y = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1}(x - x_1) + y_1 & (x_1 \leq x < x_2) \\ \frac{y_3 - y_2}{x_3 - x_2}(x - x_2) + y_2 & (x_2 \leq x < x_3) \end{cases}$$

输入 $x_1, x_2, x_3, y_1, y_2, y_3$, 再输入 x , 要求打印出相应的 y 值。

4-5 为优待顾客, 商店对购货额为 1000 元以上的(含 1000 元, 下同), 八折优惠; 500 元以上、1000 元以下, 九折优惠; 200 元以上、500 元以下的, 九五折优惠; 100 元以上、200 元以下的, 九七折优惠; 100 元以下的, 不优惠。请编出程序, 在输入一个购货款额后, 打印出应收货款。要求使用 ELSE IF 块。

4-6 输入一个整数, 打印出它是奇数还是偶数。

4-7 输入一个数 M , 判断它能否被 7、11、17 整除。如果能被其中某个数整除, 则输出“ M 能被 7(或 11、17)整除”; 如果不能被其中任何一个数整除, 则输出“ M 不能被 7、11 和 17 整除”。

4-8 输入四个数 A, B, C, D , 按由小到大的顺序把它们打印出来。

4-9 有一方程 $AX^2 + BX + C = 0$, 其中 A、B 和 C 的值由键盘输入, 请编写程序, 输出以下情况时方程的解:

- (1) $A=0, B \neq 0$
- (2) $A=0, B=0, C=0$ (X 为任意值)
- (3) $A=0, B=0, C \neq 0$ (X 无解)
- (4) $A \neq 0, B^2 - 4AC > 0$
- (5) $A \neq 0, B^2 - 4AC = 0$
- (6) $A \neq 0, B^2 - 4AC < 0$

4-10 以市中心为圆心, 半径 20 公里以内(包括 20 公里), 每亩地价 2 万元, 20 公里以外的, 每亩 1 万元。请编写程序, 输入某一点的 x, y 坐标, 求出该点的每亩地价。

4-11 编写货物征税程序。10000 元和 10000 元以上的货物征税 5%; 5000 元到 9999 元的货物征税 4%; 1000 元到 4999 元的货物征税 3%; 1000 元以下的免税。输入货款, 求税金(保留两位小数)。

4-12 有一地区的建筑规划是从市中心向外逐渐增加建筑高度。以市中心为坐标原点, 对该地区的任一处(x, y), 规定: 在 |x| 与 |y| 均小于 10 公里范围内, 建筑高度 H 不超过 20 米; |x| 与 |y| 有一个在 10 公里到 20 公里之间, 另一个不超过 20 公里的, H 不超过 30 米; |x| 与 |y| 有一个在 20 公里到 30 公里之间, 另一个不超过 30 公里的, H 不超过 50 米; |x| 与 |y| 至少有一个在 30 公里以上(含 30 公里)的, H 不超过 100 米。输入一个(x, y)值, 求出在该处盖房最高可以为多少米。要求用逻辑变量。

4-13 给出一个年号(如 1996), 判断它是否闰年。是闰年时, 输出“××××年是闰年”的信息, 否则输出“××××年不是闰年”的信息。判断是否闰年的标准是:

- (1) 年数不能被 4 整除的不是闰年;
- (2) 年数能被 4 整除但不能被 100 整除的是闰年;
- (3) 能被 100 整除但不能被 400 整除的不是闰年, 能被 400 整除的是闰年。

4-14 读入整型变量 N 的值, 若 $N=1, 2, 3, 5$, 则 $Y=1$; $N=4, 8$, 则 $Y=2$; $N=6, 7$, 则 $Y=3$; 其它情况则 $Y=0$ 。打印 Y 值。要求用 CASE 结构。

4-15 设变量 COLOR_LIGHT 是字符型变量, 表示灯光色彩。用 CASE 结构编程, 当该变量值是红色('RED')时, 输出停止字样(STOP); 当变量值是黄色('YELLOW')时, 输出等待字样(WAIT); 当值是绿色('GREEN')时, 输出通行字样(PASS)。

4-16 有四个圆, 圆心分别为(2, 2), (2, -2), (-2, 2), (-2, -2), 圆半径为 1。输入的坐标点在圆上或圆内时, 相应的 H 值为 10; 坐标点在圆外时, 相应的 H 值为 0。输入一个坐标点(x, y), 求相应的 H。要求用逻辑型情况表达式的 CASE 结构。

第 5 章 循环结构程序设计

循环结构是结构化程序设计的三种基本结构之一。FORTRAN90 中由于它由以 DO 开头、以 END DO 结束的程序段组成,故又称为 DO 结构(或 DO 块)。另外,循环就意味着重复执行某一段程序,因而也可以称作重复结构。

5.1 引言

循环是程序设计中一个十分重要的概念和手段。在程序设计中,常常遇到一些需要重复执行多次的问题。例如要打印 $\sin X$ 的值,每 10° 为一个间隔, X 的范围为 $0^\circ \sim 360^\circ$ 。若采用顺序结构实现,其程序应为:

```
PROGRAM PROG1
  IMPLICIT NONE
  REAL::PI
  PI=3.14159/180.0
  PRINT *,SIN(0)
  PRINT *,SIN(10 * PI)
  PRINT *,SIN(20 * PI)
  ...
  PRINT *,SIN(360 * PI)
END PROGRAM PROG1
```

这个程序很长,输出的部分就占 37 行。若采用循环方法,则很容易使程序缩短。从以上 37 个输出语句看,自变量的变化是每次增加 10° ,其它的都不变。这样可用一个变量作为自变量,每次使其增加 10° ,直到 360° 为止。用循环方法改写的程序为:

```
PROGRAM PROG2
  IMPLICIT NONE
  REAL::X,PI
  X=0
  PI=3.14159/180
  DO
    PRINT *,SIN(X * PI)
    X=X+10
    IF(X>360)EXIT
  END DO
END PROGRAM PROG2
```

在该程序中,DO 和 END DO 之间的三个语句是循环时要执行的部分,叫做循环体。该程序中若自变量超过 360° 时,就跳出循环体,循环结构结束。

在实际应用中,有许多计算公式是有规律的,均可采用循环的方法进行计算。例如:

$$e \approx 1 + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{n!}$$

$$\sin x \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots + (-1)^{n-1} \frac{x^{2n-1}}{(2n-1)!}$$

由于 FORTRAN90 中不提倡语句带有标号,也不提倡使用 GO TO 语句,因而排除了用 IF 语句和 GOTO 语句构成循环的情况。这样在循环程序段中均由 DO 结构实现,其 DO 结构共有三种形式:

1. 不带循环变量的 DO 结构
2. 带循环变量的 DO 结构
3. DO WHILE 结构

5.2 无循环变量的 DO 结构

5.1 节所列举的循环的例子属于不带循环变量的 DO 结构,该结构的特点是 DO 语句只有关键词 DO,后边既无循环变量控制,也没有条件控制。

5.2.1 无循环变量 DO 结构的一般形式

该 DO 结构的一般形式是:

[结构名:] DO	! DO 语句
循环体	! DO 块
END DO [结构名]	! END DO 语句

方括号内的内容是可选项,也就是说在 DO 结构中,根据需要可以加上结构名。该结构名对 DO 结构起标识作用,以提高程序的可读性。

在 DO 结构中,DO 是关键字,称为 DO 语句,表示 DO 结构的入口;END DO 也是关键字,称为 END DO 语句,表示 DO 结构的出口;在这两语句之间的语句序列称为 DO 块,也叫循环体。

DO 块可以是一条或多条简单的可执行语句,也可以包含完整的 IF 结构、CASE 结构或另一个 DO 结构(称为 DO 嵌套)等。

5.2.2 无循环变量 DO 结构的执行过程

对于无循环变量 DO 结构,其执行过程可通过下面的例子来说明:

```
DO
  READ *, X
  SUM = SUM + X
  PRINT *, SUM
END DO
```

当控制从 DO 语句进入循环体以后,将顺序地执行 DO 块中的各个语句。本 DO 结构先从键盘输入一个数放入变量 X 中,然后将它加到用于累加的变量 SUM 中,接着输出 SUM 值,然后再重复执行循环体,从键盘再输入数据、累加、输出,……,一直进行下去。

由这个执行过程可见,该 DO-END DO 结构本身只起到一个引导循环的作用,并不能终止循环。因此,这种 DO 结构是一个无休止的循环,称为死循环。它虽然在语法上没有毛病,但缺少实用性。在实用中,必须使其在某种情况下停止循环。这就要用到下面介绍的 EXIT 语句。

5.2.3 EXIT 语句和 CYCLE 语句

EXIT 语句和 CYCLE 语句是专门用在循环结构中的,都对循环有控制作用。

1. EXIT 语句

EXIT 语句的一般格式为:

EXIT [DO 结构名]

该语句的作用是停止循环,将控制转移到当前循环(或指定结构)之外。EXIT 的英文意思是“出口”,因而也称其为循环出口语句。

当单独使用 EXIT 语句时,将无条件地中止循环,因此是没有实际意义的。一般的用法是有条件地控制循环出口。其格式为:

IF(逻辑表达式)EXIT [DO 结构名]

它的执行过程是当逻辑表达式的值为假时,继续执行循环体(去执行该 IF 语句下边的语句);当逻辑表达式的值为真的时候,停止正在进行的循环,将控制转移到 EXIT 后指定的结构之外;若没有指定结构名,则跳出当前循环。

对于 5.2.2 节中的 DO 结构进行如下改进:

```
DO
  READ *, X
  IF(X >= 99999)EXIT
  SUM = SUM + X
  PRINT *, SUM
END DO
```

通过这样改进后,该结构就是有条件地执行了。当输入的数大于或等于 99999 时,就终止循环,进行循环结构后面的操作。

含有 EXIT 语句的 DO 结构的简单形式为:

```
DO
  块 1
  IF(逻辑表达式)EXIT
  块 2
END DO
```

其中块 1 和块 2 可以是一条或多条可执行语句,当然也可以有一个是空块(没有任何语句)。该 DO 结构的执行过程如图 5-1 所示。

2. CYCLE 语句

CYCLE 语句的一般格式为:

CYCLE [DO 结构名]

其作用是在循环执行到该语句时,跳过循环体在它后面的那些语句,再从循环体的第一条语句开始执行。含有 CYCLE 语句 DO 结构的简单形式为:

DO

块 1

IF(逻辑表达式)CYCLE

块 2

END DO

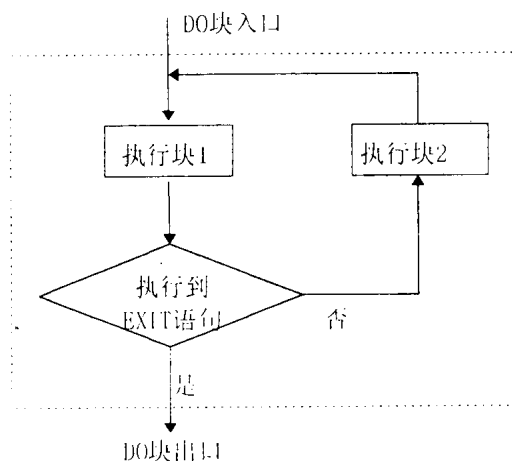


图 5-1 含有 exit 语句的 DO 结构的执行过程

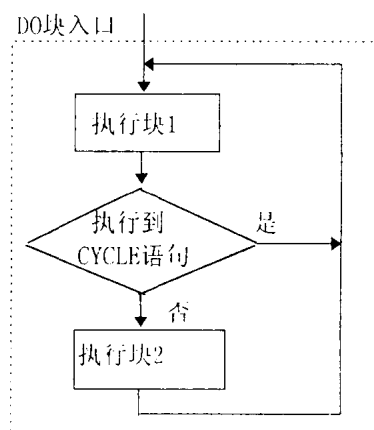


图 5-2 含有 cycle 语句的 DO 结构的执行过程

其执行过程如图 5-2 所示,块 1、块 2 也可以有一个是空块。

从 CYCLE 语句的执行功能看,它只能引导循环的执行方向,并不能终止循环的执行。CYCLE 语句的使用一般也是有条件的,常用的格式为:

IF(逻辑表达式)CYCLE [DO 结构名]

它的执行机理是,当逻辑表达式的值为真时,执行 CYCLE 功能;若逻辑表达式的值为假,则继续执行该语句下面的内容。

使用了 CYCLE 语句之后,使 DO 结构的应用更加灵活多变,可以有条件地选择某次循环执行的部分,它是一条很有用途的语句。例如:

DO

READ *,X

IF(X<=0)CYCLE

IF(X>999999)EXIT

SUM=SUM+X

END DO

在该结构中,由于使用了 CYCLE 语句,就可以实现只往 SUM 中累加输入的小于或等于 999999 的正数,不累加负数。

[例 5-1] 求 $1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$ 的值。

解决这类问题的关键是找出多项式的通项。对于本题,很容易地看出通项是 $a_i = \frac{1}{i}$ 。

这样本例变成求 $\sum_{i=1}^n a_i$ 的值。只要给出 n 的值,编写一个能循环 n 次的 DO 结构,每次循环时累加一个 a_i 即可(i 由 1 变化到 n ,每一次 i 加 1)。

由分析可知,程序需要几个变量:累加的项数 I (整型),存放通项值的 T (实型),存放总和的 S (实型),和待输入的总项数 N (整型)。

具体程序如下:

```
PROGRAM MAIN
  IMPLICIT NONE
  INTEGER::N,I=1
  REAL::T,S=0.0
  READ *,N
  DO
    T=1.0/I
    S=S+T
    I=I+1
    IF(I>N)EXIT
  END DO
  PRINT *, "SUM=",S
END PROGRAM MAIN
```

在本例中,对变量 I 、 S 进行类型说明的同时,分别被赋了初值。当然,也可在说明部分只说明变量类型,在执行部分再赋初值。

程序的运行结果如下:

15✓

SUM= 3.182290

在程序当中,使用变量 S 来存放累加后的和,程序的 7、8、9 和 10 行是需要重复执行的语句,称为“循环体”。第 10 行是控制循环的语句,当累加次数大于 N 时停止循环。可以通过程序执行时各变量的值的变化来看一看循环执行情况(称为“走程序”,见表 5-1)。

表 5-1 例 5-1 的循环执行过程

第几次循环	S 的原值	I 的原值	T 的值	S 的新值	I 的新值	I>15	还执行循环否
1	0	1	1	1	2	否	执行
2	1	2	1/2	$1 + 1/2$	3	否	执行
3	$1 + 1/2$	3	1/3	$1 + 1/2 + 1/3$	4	否	执行
...
14	$1 + 1/2 + \cdots + 1/13$	14	1/14	$1 + 1/2 + \cdots + 1/14$	15	否	执行
15	$1 + 1/2 + \cdots + 1/13 + 1/14$	15	1/15	$1 + 1/2 + \cdots + 1/14 + 1/15$	16	是	不执行

该程序比较简单,但却包含了一般循环程序的基本内容,即一些变量初值的设定(如 I、S)与循环体语句的关系以及循环中终止条件等。

对于该程序,如果将变量 S 的初值改为

S=1.0

那么,循环体内的语句改为下列顺序才能保证程序的正确执行:

I=I+1

T=1.0/I

S=S+T

IF(I>=N)EXIT

读者可以自己分析一下这样改动后程序的执行情况,思考循环终止条件采用“>=”来代替原来的“>”的原因是什么。

[例 5-2] 求一批数中负数的个数及负数的总和。要求数据一个一个地由键盘输入,当输入的数据为 0 时终止循环。

假设读入的数据放在 X 中,负数的个数用 NUM 统计,而负数的总和放在 SUM 当中。程序如下:

```
PROGRAM COUNT
  IMPLICIT NONE
  REAL::X,SUM=0.0
  INTEGER::NUM=0
  DO
    READ *,X
    IF(X==0)EXIT
    IF(X>0)CYCLE
    NUM=NUM+1
    SUM=SUM+X
  END DO
  PRINT *,"NUMBER=",NUM
  PRINT *,"SUM=",SUM
END PROGRAM COUNT
```

在本程序中,用 EXIT 语句和 CYCLE 语句联合控制循环的执行。当读入的 X 值等于 0 时,终止循环。否则再判断 X 是否大于 0,若大于 0,则再输入一个数。只有当输入的数小于 0 时,才使 NUM 加 1,然后将 X 累加到 SUM 中。

[例 5-3] 用迭代法求方程 $X^2 + 4X + 1 = 0$ 的根。

迭代法的具体操作过程如下:

- (1)先将求 X 的式子写成 $X = F(X)$ 的形式。令 $X = (-X^2 - 1)/4$
- (2)大致估计出 X 的范围,给 X 指定一个初值 X_0 ,把它带入上式等号右边,求出 X 的第一次近似值 X_1 。
- (3)再将 X_1 带入上式,求出第二次近似值 X_2 。这样一次又一次地将求出的新值又作为

下一次的初值代入 $F(X)$, 这就是迭代法。

(4) 当前后两次迭代求得的值达到一定精度, 即 $|X_{n+1} - X_n| \leq \epsilon$ (ϵ 为一个给定的误差值) 时, 认为 X_{n+1} 就是 X_0 附近的一个近似根。如果迭代达到一定次数仍达不到给定的精度, 则认为迭代可能是发散的(不收敛), 就不再迭代下去了, 并给出相应的提示信息。

```
PROGRAM ITERATION
  IMPLICIT NONE
  INTEGER::I,M      ! M用于控制迭代的次数
  REAL::X0,X,E
  READ *,X0,E,M
  I=1
  DO
    X=(-X0*X0-1)/4
    IF(ABS(X-X0)<=E)EXIT
    X0=X
    I=I+1
    IF(I>=M)THEN
      PRINT *, 'DIVERGENT!'
      EXIT
    END IF
  END DO
  IF(I<M)PRINT '( "I=",I4,"X=",F12.7)',I,X
END PROGRAM ITERATION
```

若初值为 2, $\epsilon = 10^{-6}$, 给定的控制迭代次数为 40 时, 程序运行结果为:

2.0, 1E-6, 40 ✓

I= 10 X= -0.2679493

由此可知, 求出的近似根为 -0.2679493。共进行了 10 次迭代就满足了要求。若让 X 的初值 $X_0 = -1.0$ 时, 经 9 次迭代, 可得近似根 -0.2679492。但当 $X_0 = 5.0$ 时, 第一次迭代后得 $X_1 = -6.5$, 第二次后得 $X_2 = -10.8125$, 第三次后得 $X_3 = -29.4775391 \dots$, 其结果并不能收敛于某一常数, 这时的迭代是发散的。

由此可见, 在使用迭代法时, 应该先判断 $X_{i+1} = F(X_i)$ 是否收敛。一般采用下面的方法判断: 若 $F(X)$ 具有一阶连续导数, 且对所有的 X , 若有 $|F'(X)| \leq P < 1$ (P 为常数), 那么, $X_{i+1} = F(X_i)$ 对于任意的 X_0 均收敛, 且 P 越小, 收敛速度越快。因此, 选择适当形式的 $F(X)$ 是必要的。当然 $P > 1$ 时, 也不一定绝对是发散的, 只是敛散性需用其它方法来判断。本例中由于不能保证对任意的 X_0 都使 $P < 1$, 因此对于有的 X_0 , 迭代是收敛的; 而当 X_0 取 5.0 时, 迭代是发散的。

本例的程序中, DO 结构内使用了两个 EXIT 语句。第一个 EXIT 语句用于控制当满足精度时退出循环, 而第二个 EXIT 语句是用于控制当循环次数较大(达到了限定的次数)后, 迭代精度仍没有满足要求时, 打印出“发散”的信息后, 也退出循环。

通过以上几个例子, 可以看到一般 DO 结构的使用方法, 以及编制一般循环程序应采取的步骤。

5.3 带循环变量的 DO 结构

前面所述的 DO 结构,适用于事先无法准确知道循环次数的情况,它是根据给定的逻辑表达式的值是否为“真”来控制循环是否继续执行。如果我们已知循环的次数(或者可以计算出循环的次数),就可以使用带循环变量的 DO 结构来处理。这种循环的总体结构与无循环变量的 DO 结构相似,入口有 DO 语句,出口有 END DO 语句,对于该结构,只是在入口 DO 语句的后面增加一个循环控制变量以及该变量的初值、终值与每次循环后控制变量增加的量,以此来规定循环的次数。

5.3.1 带循环变量 DO 结构的一般形式

带循环变量 DO 结构的一般形式是:

```
[DO 结构名:] DO[,] $v = u_1, u_2$  [,  $u_3$ ]  
      循环体  
      END DO[DO 结构名]
```

其中:DO 是关键字,表明 DO 结构的入口; v 是循环控制变量(又简称为循环变量),可以是整型或实型; u_1 是循环控制变量的初值; u_2 是循环控制变量的终值; u_3 是循环控制变量的增量(又称步长),当 $u_3 = 1$ 时,可以省写“ u_3 ”。

u_1 、 u_2 、 u_3 可以是整型、实型的常数、变量或表达式,当其为表达式时,其中涉及的变量应具有具体的值, u_1 、 u_2 、 u_3 的符号可正、可负。

由于实型数据在内存中进行数据转换时,可能出现误差,因此,在使用 v 、 u_1 、 u_2 、 u_3 这些变量时,最好以整型为主,防止因取舍误差而导致循环次数与实际设想不符。

以下是合法的 DO 结构:

```
DO I=1,100,2  
  S=S+I  
END DO
```

其中 I 是循环变量,其初值是 1,终值是 100,步长是 2,实际上循环体被执行 50 次。

下列语句均为合法的 DO 语句:

```
DO I=3,19,2  
DO I=19,3,-2  
DO K=M+1,M*L,J      ! 其中 M,L,J 应有确切的值  
DO A=1.2,14.7,0.1    ! 虽合法,但不提倡用小数步长
```

当循环变量的增量是 1 时,可以省写步长。例如:

```
DO M=3,7              相当于 DO M=3,7,1  
DO K=2,SQRT-REAL(N)) 相当于 DO K=2,SQRT-REAL(N)),1
```

5.3.2 带循环变量 DO 结构的执行过程

带循环变量 DO 结构的执行过程为：

(1) 先分别计算表达式 u_1 、 u_2 、 u_3 的值，当这些值的类型与循环变量类型不一致时，将它们转化为与循环变量 v 相同的类型。例如，若 u_1 、 u_2 、 u_3 的值为实型， v 为整型，则将 u_1 、 u_2 、 u_3 转化为整型。此时， u_1 、 u_2 、 u_3 的值计算出来以后，在循环执行过程中不能再发生变化。

(2) 循环变量 v 得到初值 u_1 ，就相当于执行该赋值语句：

$$v = u_1$$

(3) 计算出循环次数 R ，其计算公式为：

$$R = \text{INT}((u_2 - u_1 + u_3)/u_3)$$

下面给出几种情况来看一下计算循环次数的方法(设 I 、 J 、 K 为整型变量， X 、 Y 、 Z 为实型变量)：

① DO I = 1, 100, 2

$$R = \text{INT}((100 - 1 + 2)/2) = 50$$

② DO X = 1.2, 54.3, 1.4

$$R = \text{INT}((54.3 - 1.2 + 1.4)/1.4) = 38$$

若循环变量为整型，即

DO J = 1.2, 54.3, 1.4

首先应将 1.2、54.3 和 1.4 分别转化为整型 1、54 和 1。则：

$$R = \text{INT}((54 - 1 + 1)/1) = 54$$

可见，由于控制变量初值、终值和步长的类型与循环变量类型的不一致，导致循环次数的巨大差异。因此，应尽量使循环变量与 u_1 、 u_2 、 u_3 具有相同的类型。

③ DO K = 15, 2, 1

$$R = \text{INT}((2 - 15 + 1)/1) = -12$$

若改为 DO K = 15, 2, -1，则 $R = \text{INT}((2 - 15 - 1)/(-1)) = 14$ 。也就是说，当循环变量的初值大于终值时，应使步长为负值才能保证循环正常进行。

(4) 检查循环次数。若 $R \leq 0$ ，跳过循环体，从循环出口语句 END DO 退出该层循环；若 $R > 0$ 时，执行循环体的内容。

(5) 执行完循环体后，循环变量增加一个步长 $v = v + u_3$ ，循环次数减 1， $R = R - 1$ 。

(6) 返回(4)，重复(4)、(5)步骤。

以上描述过程见图 5-3。

从框图可见，当每次循环执行到循环体最后时，要进行两个主要的功能，即循环变量 v 增加一个步长 u_3 和循环次数 R 减 1。这是处理系统自动进行的。

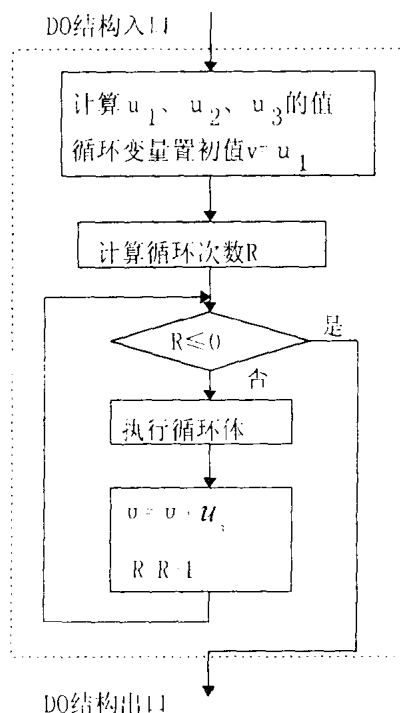


图 5-3 带循环变量 DO 结构的执行过程

若有下面的循环:

```
DO I=3,7,2
  M=3*I
  PRINT *,I,M
END DO
```

该 DO 结构循环执行的次数计算为:

$$R = (7 - 3 + 2) / 2 = 3 \text{ (次)}$$

在执行循环中,各变量值的变化情况如表 5-2 所示。

表 5-2 上例中循环各变量值的变化情况

第几次循环	I 值	M 值	下次循环 I 值	下次循环 R 值	下次循环是否执行
1	3	9	5	2	执行
2	5	15	7	1	执行
3	7	21	9	0	不执行

从该表可见,循环终止的条件是:循环计数变量 $R \leq 0$;在一般情况下,也可以理解为循环变量的值超过(不等于)终值。当执行完第三次循环后,I 的值为 9,已超过 7,因而循环终止。

5.3.3 带循环变量 DO 结构的有关规定

1. 循环变量在循环体内可以被引用,但不能被赋以新值,也就是说,循环变量在循环体内不能出现在赋值号的左边。例如,下面的用法是错误的:

```
DO K=1,50,4
  K=K*2+5
  PRINT *,K
END DO
```

循环变量在循环体内没被引用时,它仅起控制循环执行次数的作用。例如:

```
DO I=1,30,2
  PRINT *, '-----'
END DO
```

该 DO 结构的作用就是输出 15 行横线。

2. 循环变量的初值 u_1 、终值 u_2 和步长 u_3 只是在循环入口有效(用于计算循环的次数),一旦进入循环体,它们的值(即使是变量)与循环次数不再有关系。若想通过在循环体内修改 u_1 、 u_2 、 u_3 的值来改变循环的次数,是不可能的。例如:

```
K=30
DO I=1,K,2
  ...
  K=60
  ...
```

END DO

设计者最初的设想是通过在循环体内使 K 值改变来增加循环次数,但实际上该 DO 结构正常的执行次数仍然是 15 次。

3. 在带循环变量 DO 结构中仍可使用 EXIT 语句与 CYCLE 语句,在程序执行到 EXIT 语句时,无论循环次数 R 是否小于或等于 0,都必须退出 DO 结构。这种未执行完全部循环次数而脱离循环的,称为循环的“非正常出口”;而执行完全部循环次数从 END DO 退出循环的,称为循环的“正常出口”。

4. 从“正常出口”脱离循环体时,循环变量的值要超过循环变量的终值;从“非正常出口”脱离循环体时(一般是指从 EXIT 语句脱离循环的),循环变量保持“当前值”。常常利用这种特点来进行问题的求解。

[例 5-4] 判断一个整数 N 是否为素数。

素数是指只能被 1 和它本身整除的数。换句话说,如果 N 不能被 2 到(N-1)之中的所有整数整除,则 N 就是素数。只要能被 2 到 N-1 之中的任何一个数整除,N 就不是素数。

本例中循环的初值(2)、终值(N-1)和步长(1)是已知的,因此可用带循环变量的 DO 结构进行处理。

```
PROGRAM PRIME
  IMPLICIT NONE
  INTEGER::N,I
  READ *,N
  DO I=2,N-1
    IF(MOD(N,I) == 0)EXIT
  END DO
  IF(I >= N)THEN
    PRINT *,N,'IS A PRIME NUMBER.'
  ELSE
    PRINT *,N,'IS NOT A PRIME NUMBER.'
  END IF
END PROGRAM PRIME
```

本例中用到了从“非正常出口”退出循环时,循环变量的值保持“当前值”这个特点来判断 N 是否为素数。当从正常出口退出循环时,循环变量的值应超过终值 N-1。由于此 DO 结构中步长为 1,故循环变量 I 的值应是终值(N-1)加上步长(1),即为 N(满足 $I \geq N$ 的条件)。这时,表明 N 不能被 2 至 N-1 之中的所有整数整除,则 N 是素数;否则,若从非正常出口退出循环时,循环变量一定小于 N。

实际上,判断 N 是否为素数,不必将 N 被 2 到(N-1)除,只要被 2 到 \sqrt{N} (若 \sqrt{N} 不是整数,取其整数部分)除即可。采用这样作法,可对程序稍加改动,再说明一个整型变量 M,用它来存放 \sqrt{N} 的整数部分。程序主要改动部分如下:

```
...
M=SQRT(REAL(N))
DO I=2,M
  IF(MOD(N,I) == 0)EXIT
```

```

END DO
IF(I>M)THEN
...

```

这样改动后,可使循环执行的效率大为提高。例如若判断 101 是否为素数,按原来的方法最多需执行循环 100 次,而采用改进的方法,循环最多只需执行 9 次。另外,也可以用 $N/2$ 作为判断素数的终值。

5. 循环变量在循环体内不能被重新赋值,但在循环体外,循环变量可以被重新赋值。两个并列的 DO 循环可以使用相同的循环变量。例如:

```

DO I=4,15
...
END DO
...
DO I=10,17
...
END DO

```

是合法的循环结构。

6. 在 DO 结构中包含的 IF 结构和 CASE 结构等必须是完整的结构,不允许出现结构间的交叉。以下的交叉是非法的:

```

DO I=11,12
...
IF(e1)THEN
...
END DO
...
END IF

```

7. 循环可以不经 END DO(使用 EXIT 语句)而退出循环,但不能从循环体外转入循环体内。例如以下结构

```

DO I=1,N
...
10 A=A+1
...
END DO
...
GO TO 10

```

是非法的。由于 FORTRAN90 不提倡使用 GO TO 语句,因此这种错误较易避免。

下面再通过两个例子熟悉一下带循环变量的 DO 结构。

[例 5-5] 编程序求 100~200 之间有多少个各位数字之和等于 10 的整数? 请将这些数一一输出。

本题的关键是将 100~200 之间的三位数的各位数字求出,然后再求其和,当其和为 10 时,输出该数,并记录出现一次。对于三位数,求其个位、十位和百位数字的方法如下:

百位数字: $I1 = N/100$ (利用整数相除得整数的特点)
 十位数字: $I2 = (N - I1 * 100)/10$ 或 $I2 = \text{MOD}(N/10, 10)$
 个位数字: $I3 = N - I1 * 100 - I2 * 10$ 或 $I3 = \text{MOD}(N, 10)$

程序如下:

```
PROGRAM N100_200
  IMPLICIT NONE
  INTEGER::N, I1, I2, I3, I, M=0
  DO N=100,200
    I1 = N/100
    I2 = (N - I1 * 100)/10
    I3 = MOD(N,10)
    I = I1 + I2 + I3
    IF(I = 10) THEN
      M = M + 1
      PRINT *, N
    END IF
  END DO
  PRINT *, 'TIMES=', M
END PROGRAM N100_200
```

满足条件的数共有 10 个。

[例 5-6] 求 $N! = 1 \times 2 \times 3 \times \cdots \times N$ 的值。程序如下:

```
PROGRAM CALCULATE_FACTORIAL
  IMPLICIT NONE
  INTEGER::N, I, FACTORIAL_N
  READ *, N
  FACTORIAL_N = 1
  DO I=1, N
    FACTORIAL_N = FACTORIAL_N * I
  END DO
  PRINT '("FACTORIAL OF", I4, " IS ", I5)', N, FACTORIAL_N
END PROGRAM CALCULATE_FACTORIAL
```

在进行连乘问题的计算时,存放连乘积的变量的初值应为 1。当程序运行时若输入的值 5,输出的结果为:

FACTORIAL OF 5 IS 120

5.4 DO WHILE 结构

无循环控制变量 DO 结构中,控制循环终止是用有条件的 EXIT 语句;而带循环控制变量的 DO 结构中,循环一般是由循环的次数来控制。本节介绍的循环,控制条件在 DO WHILE 语句中。

5.4.1 DO WHILE 结构的一般形式

DO WHILE 结构的一般形式为:

```
[结构名:]DO WHILE(逻辑表达式)  ! DO WHILE 语句
      循环体                      ! DO WHILE 块
EDN DO [结构名]                  ! END DO 语句
```

结构名的用法与 5.2 节中提到的类似。在该结构中,DO WHILE 是关键字,称为 DO WHILE 语句,表示循环的入口;END DO 也是关键字,称为 END DO 语句,是循环的正常出口;DO WHILE 语句中的逻辑表达式是控制循环的条件;DO WHILE 语句和 END DO 语句之间的部分构成了参与循环的内容,称为 DO WHILE 块,也叫循环体。

5.4.2 DO WHILE 结构的执行过程

在介绍 DO WHILE 语句执行过程前,先了解一下计算 7! 的一段程序:

```
...
I=0; FAC=1
DO WHILE (I<7)
  I=I+1
  FAC=FAC*I
END DO
...
```

当控制到达循环入口时,首先判断逻辑表达式 $I < 7$ 的值是否为真,若为真则进入循环体;若为假则退出循环。当执行完一次循环体后,控制又返回到 DO WHILE 语句检验其后的条件,直到其后的逻辑表达式的值为假时退出循环。

DO WHILE 结构的执行过程如图 5-4 所示。

从 DO WHILE 结构的执行过程可见,当控制第一次被引导至 DO WHILE 语句处时,若其后的条件不满足,则控制不会被引导到循环体(循环体一次也不被执行)。

[例 5-7]求 N 个整数中奇数的平均值。用 DO WHILE 结构实现如下:

```
PROGRAM ODD_AVER
  IMPLICIT NONE
```

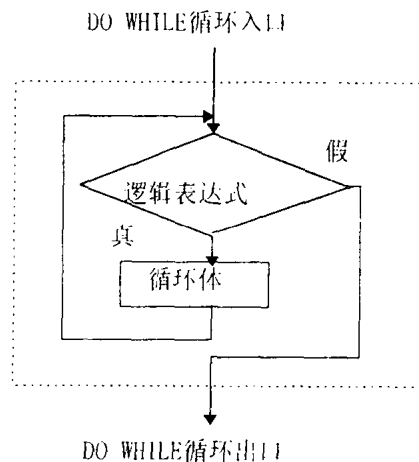


图 5-4 DO WHILE 循环的执行过程

```

INTEGER::X,N,I=0,M=0
REAL::AVER,SUM=0
READ *,N
DO WHILE(I<N)
    READ *,X
    IF(MOD(X,2)/=0)THEN
        SUM=SUM+X
        M=M+1
    END IF
    I=I+1
END DO
AVER=SUM/M
PRINT *,AVER
END PROGRAM ODD_AVER

```

读者可以思考一下,若将 SUM 和 AVER 均说明成整型变量,结果会有什么不同?

[例 5-8]有一批实数,用零作终止标记。请编程序找出这批数中的最大值及最小值,并求出该最大值是这批数中的第几个数(若最大值数多次出现,只打印出第一个的位置)。

对于这个问题的关键是要记录最大值出现的位置号。因此从第一个数开始就要记录输入变量的个数。

具体程序如下:

```

PROGRAM MAX_MIN
    IMPLICIT NONE
    REAL::X,BMAX,BMIN
    INTEGER::I,IMAX
    READ *,X
    BMAX=X           ! 先输入一个数给 BMAX、BMIN 置初值
    BMIN=X
    I=1
    READ *,X
    DO WHILE(X/=0)
        I=I+1
        IF(X>BMAX)THEN
            BMAX=X
            IMAX=I
        ELSE IF(X<BMIN)THEN
            BMIN=X
        END IF
        READ *,X
    END DO

```



```

PRINT '("The Max Number is",F6.2,4X,"Position is:",I2)',BMAX,IMAX
PRINT '("The min number is",F6.2)',BMIN
END PROGRAM MAX_MIN

```

程序中 BMAX、BMIN 用于存放最大值及最小值,用 I 统计输入数据的个数,IMAX 记录最大值第一次出现的位置号。

5.5 DO 结构嵌套

在一个 DO 结构的循环体内又包含有另一个完整的 DO 结构,称为 DO 结构的嵌套。

下面的程序段就是一个嵌套的 DO 结构,I 是外层 DO 结构的循环变量,J 是内层 DO 结构的循环变量。为了便于阅读,对每个 DO 结构都给出了结构名,名为 II 的结构是外层 DO 结构,名为 JJ 的结构是内层 DO 结构。

```

...
II: DO I=1,2
JJ: DO J=5,7
    PRINT *,I,J
END DO JJ
END DO II
...

```

该程序段的执行结果为:

```

1 5
1 6
1 7
2 5
2 6
2 7

```

它共执行 6 次循环。对于外层 DO 语句,计算循环次数为 $RI=2$ (次),内层 DO 语句,计算循环次数为 $RJ=3$ (次)。对于内循环体执行的循环总次数为 $RI \times RJ=6$ (次)。

由于输出的刚好是内外循环的循环变量的值,因此,可以清楚地看到嵌套 DO 结构的执行情况。

图 5-5 给出了嵌套循环执行的流程图。其中 RI、RJ 分别是外层和内层的循环次数,I、J 分别是外层和内层的循环变量。

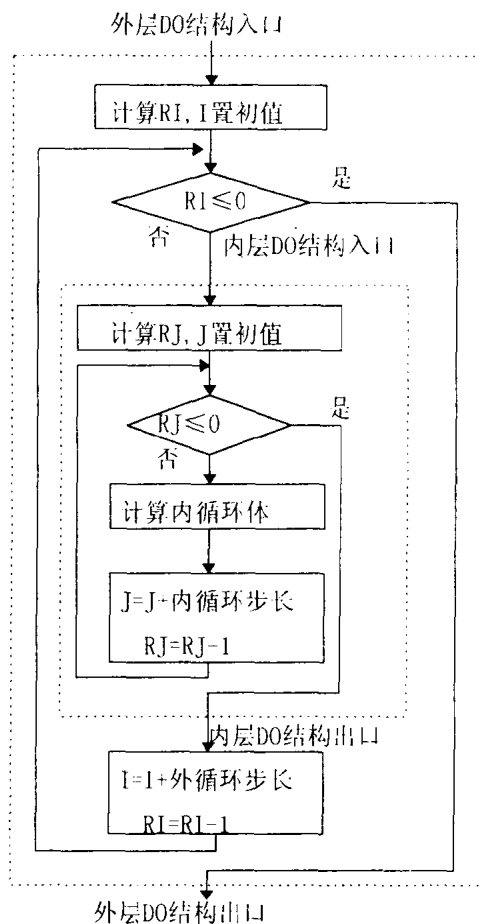


图 5-5 嵌套 DO 循环的执行过程

5.5.1 嵌套 DO 结构的执行过程

DO 结构可以有多种嵌套,这里介绍二重嵌套的执行过程。对于多重嵌套,其基本原理相同。

1. 当控制进入到外层 DO 结构后,先计算出外层 DO 结构的循环次数 RI,外层循环变量得到初值。

2. 若 $RI \leq 0$,则结束外循环的执行,当然也不能进入内循环;若 $RI > 0$,执行外层结构的 DO 块内的语句。

3. 当遇到内层 DO 语句时,控制进入内层 DO 结构;先算出内层 DO 结构的循环次数 RJ,内层循环变量得到初值。

4. 对于内层 DO 结构的执行过程与 5.3.2 节中介绍的一样;若 $RJ > 0$,顺次执行内循环的各语句;当 $RJ \leq 0$ 时,从“正常出口”退出内循环。

5. 继续执行内循环 DO 结构后面的外循环体内的其它语句。

6. 外层 DO 结构循环变量加一个步长,循环次数 RI 减 1。

7. 重复 2~6 步的算法,直到 RI 等于 0,结束全部循环。

从以上的分析可见:

1° 外循环体共执行 RI 次,内循环共执行 $RI * RJ$ 次。可见,若最初的 $RI \leq 0$,则内外循环体将一次也不执行。

2° 在每执行一次外循环的过程中,内循环体执行了 RJ 次,即在每一次外循环过程中,外循环变量的值保持不变,而内循环变量共变化了 RJ 次。

下面通过一个例子熟悉一下嵌套 DO 循环的使用方法。

[例 5-9] 求出 20~50 之间的全部素数。

在例 5-4 中,给出了判断某一整数 N 是否为素数的程序,若求某一范围内所有素数的问题,则应使用嵌套的 DO 结构。

```
PROGRAM PRIME20_50
  IMPLICIT NONE
  INTEGER::N,I,M,SWITCH
NN:  DO N=20,50
      M=SQRT(REAL(N))
      SWITCH=1
II:   DO I=2,M
          IF(MOD(N,I) == 0)SWITCH=0
      END DO II
      IF(SWITCH == 1)PRINT *,N
  END DO NN
END PROGRAM PRIME20_50
```

程序的执行结果为:

23
29
31

37
41
43
47

本程序中判断 N 是否为素数时,变量 SWITCH 起“开关”作用。在判断某一整数 N 是否为素数前,先将开关置为 1。而在判断过程中,如果 N 能被某个数整除时,则将开关置为 0。一旦 SWITCH 在内循环中被置成 0 以后,就不能再改变了。因而,在内循环结束之后,可以通过检测开关变量 SWITCH 值的办法来确定 N 是否为素数。

可以将外循环的 DO 语句改写为:

```
NN: DO N=21,49,2
```

这样可使外层 DO 结构的循环次数由 31 次减为 15 次。另外,内层 DO 循环也可以改由 DO WHILE 结构实现。

5.5.2 嵌套 DO 结构的有关规定

1. 各种 DO 结构都可以嵌套,但无论何种嵌套,内层 DO 结构必须完整地嵌入到外层 DO 结构之中,两者不能出现交叉。如:

```
SUM=1;I=1
II:  DO
      FAC=1
      I=I+1
JJ:   DO J=1,I
      FAC=FAC * J
      END DO JJ
      SUM=SUM + FAC
      IF(I=5)EXIT
      END DO II
PRINT *,SUM
...
```

这是无循环变量 DO 结构与带循环变量 DO 结构的嵌套,用以实现计算下式的值:

$$1! + 2! + 3! + 4! + 5!$$

由于带循环变量 DO 结构完整地出现在无循环变量的 DO 结构中,因而这是一个合法的二重嵌套 DO 结构。但下面的写法由于出现了两个 DO 结构的交叉,是属于非法的:

```
II:  DO
      ...
JJ:   DO J=1,I
      ...
      END DO II
      ...
```

END DO JJ

2. 对于带循环变量 DO 结构, 并列的 DO 结构可以用同一个变量名作循环变量, 而嵌套的 DO 结构则不能使用相同的循环变量名。因为在嵌套的 DO 结构中, 若使用了相同变量名, 执行内循环时要对内循环变量赋值, 这就相当于对外层循环变量重新赋值, 这是不允许的。

3. 对于嵌套的 DO 结构, 无论循环次数是否为 0, 只要控制执行到 EXIT 语句, 都将转到循环体外。如果在 EXIT 后没有指明结构名时, 就是转出当前所在的循环(最近一层循环); 若指定了结构名时, 则转出该结构名所代表的循环体。例如:

```
...
II: DO I=3,50
JJ: DO J=2,1-1
    IF(MOD(I,J) == 0)EXIT
    END DO JJ
    IF(J >= I)PRINT *,I
END DO II
...
```

执行内层 DO 结构中的 EXIT 语句, 只退出内层 DO 结构。若将该条语句改为

```
IF(MOD(I,J) == 0)EXIT II
```

将直接退出结构名为 II 的 DO 结构, 即直接退出外层循环。

4. 可以将控制从循环体内转到 DO 结构外, 但不允许将控制从 DO 结构外转到 DO 结构内。由于 FORTRAN90 不提倡使用 GO TO 语句, 因此容易避免这种错误。

[例 5-10] 求出全部的水仙花数。所谓水仙花数是个三位数, 其各位数字的立方和等于该数。例如 $153 = 1^3 + 5^3 + 3^3$, 因而 153 是水仙花数。

一般这类问题的求解是采用穷举法, 即将所有可能的数字组合全部找到, 然后选出符合条件的组合。本例的研究对象是三位数, 可以假设其百位、十位、个位数字分别是 I、J 和 K。将 I、J、K 的所有组合找到, 再看哪些符合水仙花数条件, 输出满足条件的各个数。

```
PROGRAM FLOWER
IMPLICIT NONE
INTEGER::I,J,K,M,N
II: DO I=1,9
JJ: DO J=0,9
KK: DO K=0,9
    M=I*100+J*10+K
    N=I**3+J**3+K**3
    IF(M==N) PRINT *,M
END DO KK
END DO JJ
END DO II
END PROGRAM FLOWER
```

程序的运行结果为:

153

370

371

407

该程序用到了三重嵌套 DO 结构,其各层 DO 结构的循环次数分别为 9 次、10 次、10 次。对于内循环体,其总的循环次数为 $9 \times 10 \times 10 = 900$ 次。实际上该问题也可采用一重 DO 结构实现(习题 5-14),其循环次数仍然是 900 次。

5.6 隐含 DO 循环

隐含 DO 循环实际上是带控制变量的 DO 结构。但简化成只有 DO 结构的第一句,且把 DO 关键字隐去,成为 $v = u_1, u_2, u_3$ 的形式。这种结构不能独立存在,主要用于输入输出语句,以改善输入输出的形式。常常出现于数组的输入输出中。

隐含 DO 循环的形式为:

$(list, v = u_1, u_2, u_3)$

左括号相当于 DO,右括号相当于 END DO。而 v, u_1, u_2, u_3 分别相当于带循环变量 DO 结构中的循环变量、循环变量初值、终值和步长。当步长是 1 时,可将其前边的逗号省略。list 是参与循环的项,可以是一个或多个常量、变量、数组元素等,甚至可以是另外的隐含 DO 循环(称为隐含 DO 循环嵌套)。例如:

READ *, (A(I), I=1,5)

它表示要输入数组 A 的 5 个元素,相当于

READ *, A(1), A(2), A(3), A(4), A(5)

就是说,可以在一个记录内将 A 数组的 5 个数据一次输入。它不同于:

DO I=1,5

READ *, A(I)

END DO

这种显式的 DO 结构虽然也是要求输入数组 A 的 5 个元素,但它每个记录只能输入一个数据,全部数据输入完后需要 5 行。又如:

PRINT '(5I5)', (A(I), I=1,5)

可以实现将数组 A 的 5 个元素在一行内输出。若采用

DO I=1,5

PRINT '(5I5)', A(I)

END DO

则每行只能输出一个元素,输出要占 5 行。这是很难看的。若采用

PRINT '(5I5)', A(1), A(2), A(3), A(4), A(5)

的形式,虽然可以每行输出 5 个数据,但书写太麻烦。当数组元素较多时,更是如此。

可见,隐含 DO 循环在某些输入输出语句中是很有用的。

下面给出几种常用隐含 DO 循环的形式:

1. 按指定步长读写。例如:

```
PRINT *,(X(J),J=1,19,2)
```

则输出 X(1)、X(3)、X(5)、...、X(19)。

2. 隐含 DO 循环与普通变量混合使用。例如:

```
PRINT *,A,B,(X(I),I=1,3),C
```

相当于

```
PRINT *,A,B,X(1),X(2),X(3),C
```

有时候说明了一个较大的数组,但只给前 N 个元素输入值,可写为

```
READ *,N,(A(I),I=1,N)
```

这种用法方便灵活,常被用到。

3. 用于输出一些特殊符号。例如:

```
PRINT *,(' ',I=1,80)
```

可输出一行横线。

4. 隐含 DO 循环可以并列出现。例如:

```
READ *,(A(I),I=1,4),(B(I),I=1,4)
```

相当于

```
READ *,A(1),A(2),A(3),A(4),B(1),B(2),B(3),B(4)
```

5. 隐含 DO 循环可嵌套使用。例如:

```
PRINT '(5I5)',((A(I,J),J=1,5),I=1,3)
```

对于嵌套的隐 DO 循环,其最外层的一对括号相当于外层的 DO 语句及 END DO 语句,而稍内的一对括号相当于内层隐 DO 循环的 DO 语句和 END DO 语句,它的执行过程与前面介绍的嵌套 DO 结构相似。该输出语句相当于:

```
PRINT '(5I5)',A(1,1),A(1,2),A(1,3),A(1,4),A(1,5),A(2,1),A(2,2),&  
A(2,3),A(2,4),A(2,5),A(3,1),A(3,2),A(3,3),A(3,4),A(3,5)
```

表示每行输出 5 个数据,共占 3 行。隐含 DO 循环在输入输出时用法灵活,功能较强,是非常有用的输出方法。

5.7 程序举例

[例 5-11]验证角谷猜想。“角谷猜想”是日本数学家角谷静夫提出的。其思路为:对于任意一个自然数,若为偶数把它除以 2;若为奇数把它乘以 3 后再加 1。如此经过有限次运算后,最终能得到自然数 1。试编程序,输入一个自然数验证之。

```
PROGRAM SUPPOSE
```

```
IMPLICIT NONE
```

```
INTEGER::N,I=0
```

```
READ *,N
```

```
DO
```

```
IF(MOD(N,2) == 0) THEN
```

```
    N=N/2
```

```
ELSE
```

```

        N=N*3+1
    END IF
    PRINT '(3X,I5)',N
    I=I+1
    IF(N==1)EXIT
END DO
PRINT *, 'I=',I
END PROGRAM SUPPOSE

```

当输入 10 时,输出结果为:

```

5
16
8
4
2
1

```

I= 6

本例中使用了不带循环变量的 DO 结构。程序中的变量 I 用来统计计算的次数,而每次计算都将变化后的自然数打印出来,可以清楚地看到程序的验证过程。

[例 5-12]利用下式求 π 的值:

$$2 \times \frac{2}{\sqrt{2}} \times \frac{2}{\sqrt{2+\sqrt{2}}} \times \frac{2}{\sqrt{2+\sqrt{2+\sqrt{2}}}} \times \dots$$

直到误差小于 10^{-6} 为止。程序如下:

```

PROGRAM PI- CALCULATE
    IMPLICIT NONE
    REAL::PI1,PI2,T
    PI2=2.0
    T=0.0
    DO
        PI1=PI2
        T=SQRT(2.0+T)
        PI2=PI1*2.0/T
        IF(ABS(PI2-PI1)<1E-6)EXIT
    END DO
    PRINT '(A,F10.6)', 'PI=',PI2
END PROGRAM PI- CALCULATE

```

该程序中, PI2 比 PI1 多乘了一个通项 $2.0/T$, 其中 T 的值为通项的分母

$\sqrt{2+\sqrt{2+\sqrt{2+\sqrt{\dots}}}}$ 。当 $|PI2-PI1|<10^{-6}$ 时循环终止。程序的输出结果为
PI= 3.141592

[例 5-13]用牛顿迭代法求方程 $f(x) = x^3 - 4x - 1 = 0$ 在 $x = 1.5$ 附近的一个实根,直到误差 $|X_{n+1} - X_n| \leq \epsilon = 10^{-5}$ 时为止。

已知牛顿迭代公式为:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

其迭代过程为:对于所给初值 x_1 ,依据给定曲线的函数关系 $f(x)$ 得到 $f(x_1)$ 及 $f'(x_1)$,由迭代公式求出 x_2 :

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$$

然后再由 x_2 点得到 $f(x_2)$ 及 $f'(x_2)$,求出 x_3 (图 5-6)。如此一直求下去,直到满足误差要求为止。

迭代问题的求解应采用循环的方法。由于本问题的循环次数未知,因此只能采用不带循环变量的 DO 语句。在本程序中,用 F 代表 $f(x)$,用 F1 代表 $f'(x)$,N 代表迭代次数,X 为旧值,X1 为迭代的新值。程序如下:

```
PROGRAM CALCULATE_ROOT
  IMPLICIT NONE
  REAL: :F,F1,X,X1
  INTEGER: :N=0
  READ *,X
  DO
    X1=X
    F=X**3-4*X-1
    F1=3*X*X-4
    X=X1-F/F1
    N=N+1
    PRINT *,N,X,X1
    IF(ABS(X-X1)<1E-5)EXIT
  END DO
END PROGRAM CALCULATE_ROOT
```

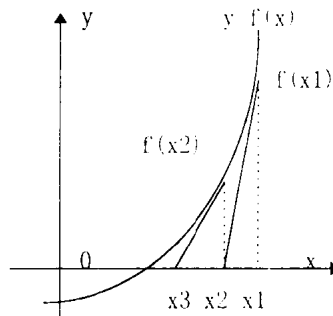


图 5-6 牛顿迭代的几何意义图

程序运行后,当输入 X 的初值为 1.5 时,运行结果为:

```
1  2.8181818  1.5000000
2  2.3082724  2.8181818
3  2.1359091  2.3082724
4  2.1151984  2.1359091
5  2.1149075  2.1151984
6  2.1149075  2.1149075
```

当程序迭代 6 次后,得到了满足精度要求的一个根 2.1149075。当给定的初值为 0 时,迭代 3 次,可得到在 0 附近的一个根 -0.2541017;当给定初值为 -1.5 时,可得在 -1.5 附

近的一个根为 -1.8608059。如果不需要查看迭代的中间结果,可将语句

```
PRINT *,N,X,X1
```

移到 DO 循环之外(X1 不必输出)更好一些。

[例 5-14] 验证哥德巴赫猜想。

哥德巴赫猜想的内容为:一个大的偶数(≥ 4)可以表示成两个素数之和。请编程序,将 10~20 间的全部偶数表示成两个素数之和。

注意:每个偶数只表示成一种组合即可。例如 10 可以表示成 $10 = 3 + 7$ 、 $10 = 5 + 5$ 和 $10 = 7 + 3$ 。我们只取第一种组合。

对于这个问题的分析思路为:若想把一个偶数 N 表示成两个素数之和,首先从 2 开始找到一个小的素数 I,若 $N - I$ 也是素数,则找到了答案;若 $N - I$ 不是素数,则继续增大 I,再找到稍大一点的素数 I,直到 $N - I$ 也是素数为止。

```
PROGRAM GUESS_PRIME
  IMPLICIT NONE
  INTEGER::N,I,J,I1,K
  LOGICAL::A,B
NN:   DO N=10,20,2
II:   DO I=2,N-2
      A=.TRUE.
JJ:   DO J=2,I/2
      IF(MOD(I,J)==0)A=.FALSE.
      END DO JJ
      IF(A)THEN
        I1=N-I
        B=.TRUE.
KK:   DO K=2,I1/2
      IF(MOD(I1,K)==0)B=.FALSE.
      END DO KK
      IF(B)THEN
        WRITE(*, '(I4,"=",I4,"+",I4)')N,I,I1
        EXIT II
      END IF
    END DO II
  END DO NN
END PROGRAM GUESS_PRIME
```

程序运行的结果为:

10= 3+ 7

12= 5+ 7

14= 3+ 11

$$\begin{aligned} 16 &= 3 + 13 \\ 18 &= 5 + 13 \\ 20 &= 3 + 17 \end{aligned}$$

[例 5-15]用辗转相除法求自然数 A 和 B 的最大公约数和最小公倍数。

求 A 和 B 的最大公约数 GCD 的辗转相除法为：首先将 A 除以 B, 得余数 R, 再用余数 R 去除原来的除数, 得新的余数, 重复此过程, 直到余数为 0 时为止。此时的除数就是 A 和 B 的最大公约数。最小公倍数 LCM 可用最初的 A、B 的积除以最大公约数而得。

例如求 24 和 18 的最大公约数：

$A=24, B=18$, 余数 $R=\text{MOD}(24, 18)$, 其值为 6 (不等于 0); 将 B 的值送给 A, R 的值送给 B, 再求余数 $R=\text{MOD}(18, 6)$, 其值为 0。此时的除数 6 就是最大公约数。

最小公倍数可用 A 和 B 原有值的积除以最大公约数： $24 \times 18 / 6 = 72$ 。

PROGRAM GCD_LCM

IMPLICIT NONE

INTEGER::A,B,L,GCD,LCM,M,R

PRINT *, 'INPUT A,B:'

READ *, A,B

$L=A*B$

IF(A<B)THEN

M=A; A=B; B=M

END IF

DO

R=MOD(A,B)

IF(R==0)THEN

GCD=B

EXIT

END IF

A=B; B=R

END DO

$LCM=L/GCD$

WRITE(*, '("THE GCD IS", I5/"THE LCM IS", I5)')GCD,LCM

END PROGRAM GCD_LCM

运行结果为:12,38✓

THE GCD IS 2

THE LCM IS 228

[例 5-16]用梯形法计算定积分 $\int_a^b \sin x dx$ 的值, 假设总区间数由键盘输入, $0 \leq a \leq b \leq \pi$ 。

求这个定积分, 在几何意义上就是求由曲线 $f(x) = \sin x$, X 轴, $X=a$ 及 $X=b$ 所围成的曲边梯形的面积。现在可将该面积划分成许多小曲边梯形, 然后由若干小直角梯形来近似代替每一个小曲边梯形。最后以这些小梯形面积的和来近似代替定积分的值。

小梯形的面积为:

$$(\text{上底} + \text{下底}) \times \text{高} / 2$$

设将区间 $[A, B]$ 划分成 N 等分,即划分为 N 个曲边梯形。那么每个小区间的宽度为:

$$H = (B - A) / N$$

若第 i 个小区间的左端点为 x_i ,右端点为 x_{i+1} ,则其面积为

$$S_i = (f(x_i) + f(x_{i+1})) \times H / 2$$

由于已知总区间数,故可用带循环变量的DO语句。其程序为:

```
PROGRAM TEGERAL_SINX
  IMPLICIT NONE
  REAL::A,B,H,S,F0,F1,X
  INTEGER::N,I
  PRINT *, 'INPUT A,B,N:'
  READ *, A,B,N
  H = (B - A) / N
  X = A; S = 0
  DO I = 1, N
    F0 = SIN(X); F1 = SIN(X + H)
    S = S + (F0 + F1) * H / 2
    X = X + H
  END DO
  PRINT *, S
END PROGRAM TEGERAL_SINX
```

当求 $[0, \pi]$ 区间积分时,积分的精确值为2。本程序中输入的 N 值不同时,其结果也不相同。若输入为0,3.1415926,50时,输出为1.9993415;输入为0,3.1415926,100时,输出为1.9998347;输入为0,3.1415926,1000时,输出为1.9999754。

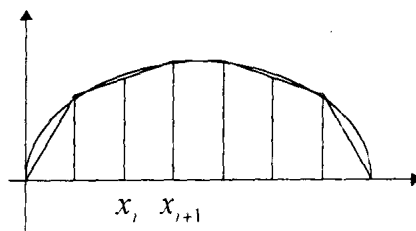


图 5-7 例 5-16 图

习题 5

5-1 编程序验证公式:

$$1^3 + 2^3 + \cdots + n^3 = (1 + 2 + \cdots + n)^2$$

5-2 编一个程序,把一批非零整数中的偶数、奇数的和分别计算出来,用零作终止标记。

5-3 某单位排队形,开始排成3路纵队,末尾多出了2人。后改成5路,末尾又多出了3人。最后改成7路纵队,正好没有余数。编程序求出该单位至少有多少人。

5-4 分别用三种循环结构编程求下式的值:

• 100 •

$$2^0 + 2^1 + 2^2 + 2^3 + \cdots + 2^{63}$$

5-5 有一分数序列,求其前 20 项的和:

$$\frac{2}{1}, \frac{3}{2}, \frac{5}{3}, \frac{8}{5}, \frac{13}{8}, \frac{21}{13}, \cdots$$

5-6 一个数列,它的头三个数是 0、0、1,第四个数是前三个数之和,以后每个数都分别是其前三个数之和。请编程序输出该数列,直到第 20 个数为止。

5-7 编程序计算下式的值:

$$\sum_{i=1}^8 \sum_{j=2}^7 (i+j)(i-j)$$

5-8 用不带循环变量的 DO 结构求下列各式的值:

$$(1) e \approx 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots + \frac{1}{n!}$$

$$(2) \sin x \approx \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots + (-1)^{n-1} \frac{x^{(2n-1)}}{(2n-1)!}$$

5-9 求 50 以内的勾股数,即求满足 $A^2 + B^2 = C^2$ 的 A,B,C。要求 A,B 和 C 均小于或等于 50,且没有重复解。例如: A=3,B=4,C=5 及 A=4,B=3,C=5 即是重复解。

5-10 由于“角谷猜想”尚未得到证明,因此无法判断验证时的循环次数。实际上对有些自然数的验证次数会很大,用例 5-11 的方法需要输出许多中间结果。请改写例 5-11,当验证次数达到 M 次时不再验证,并输出“循环次数等于 M”之类的信息。

5-11 数学灯谜。有算式

$$\begin{array}{r} \text{A B C D} \\ -) \quad \text{C D C} \\ \hline \text{A B C} \end{array}$$

其中 A、B、C、D 均为一位非负整数。要求找出 A、B、C、D 的各个值。

5-12 雨淋湿了一道算术题,9 个数字现只能认清 4 个(第一个数字虽模糊不清,但肯定不是 1)。

$$[\square \times (\square 3 + \square)]^2 = 8 \square \square 9$$

请编程序找出那些看不清的数字。

5-13 Bessel 函数 $J_n(x)$ 有以下的递推关系:

$$J_0(x) = \frac{\sin x}{x}$$

$$J_1(x) = \frac{\sin x}{x^2} - \frac{\cos x}{x}$$

$$J_n(x) = \frac{2n-1}{x} J_{n-1}(x) - J_{n-2}(x) \quad (n > 1 \text{ 时})$$

编写程序,利用递推关系,由任意的 n 和 $x \neq 0$ 求 $J_n(x)$ 。

5-14 用一重循环改写例 5-10。

5-15 已知

$$y_n = \underbrace{\cos(\cos(\cos(\cdots(\cos(x))\cdots)))}_{n \uparrow \cos}$$

计算 y_n 的值,直到 $|y_n - y_{n-1}| < 10^{-6}$ 为止,并输出此时共作了多少次 cos 计算。

5-16 计算下列各式的值:

$$(1) (1 + \frac{1}{1.3}) \times (1 + \frac{1}{3.5}) \times (1 + \frac{1}{5.7}) \times \dots \quad (\text{误差为 } 10^{-4})$$

$$(2) (\frac{2}{3} \times \frac{4}{3}) \times (\frac{4}{5} \times \frac{6}{5}) \times (\frac{6}{7} \times \frac{8}{7}) \times \dots \times (\frac{2n}{2n+1} \times \frac{2n+2}{2n+1}) \times \dots \quad (\text{误差为 } 10^{-5})$$

(3) 在一个程序中分别计算出给定误差小于 0.01、0.001、0.0001 时 π 的值,并输出累计的项数:

$$\frac{\pi}{2} \approx 1 + \frac{1}{3} + \frac{1}{3} \cdot \frac{2}{5} + \frac{1}{3} \cdot \frac{2}{5} \cdot \frac{3}{7} + \frac{1}{3} \cdot \frac{2}{5} \cdot \frac{3}{7} \cdot \frac{4}{9} + \dots$$

5-17 找出 2~1000 间的全部同构数。所谓同构数是指这样一个数,它出现在它的平方数的右端,例如 5 的平方数是 25,且 5 出现在 25 的右端,则 5 是同构数;又如 25 的平方是 625,且 25 出现在 625 的右端,则 25 也是一个同构数。

5-18 用牛顿迭代法求方程

$$x^3 + 9.2x^2 + 16.7x + 4 = 0$$

在 $x=0$ 附近的根,迭代精度为 10^{-5} 。若在 M 次内还未达到精度,则输出发散信息。

5-19 计算定积分 $\int_0^2 (x^2 + 4x + 7) dx$ 。要求用梯形法,计算精度为 10^{-3} 。

5-20 编程序求 3 至 39 之间满足下列条件的各组素数:每组有三个素数,第二个比第一个大 2,第三个比第二个大 4。例如 5、7、11 就是满足条件的一组。

要求:

① 输出满足条件的所有的解;

② 求出满足条件的素数共有多少组。

5-21 已知 $N(N>3)$ 是正整数,它可以写成三个正整数 N_1 、 N_2 和 N_3 之和的形式:

$$N = N_1 + N_2 + N_3.$$

请编程序求出满足上式的全部组合,并当 N_1 、 N_2 和 N_3 中至少有两个素数时输出“YES”,否则输出“NO”。

5-22 所谓互质数(即互素的数),是指这两个数除了 1 以外再没有其它公因数。如 14 和 9, 187 和 79 均为互质数。

任意两个互质的自然数,把它们各自经过若干次加减后,总可以找到结果为 1 的数值。例如,对于互质数 14 和 9,有相加的方法: $14 + 14 = 28$, $9 + 9 + 9 = 27$, 因 $28 - 27 = 1$, 目的达到。

请编程序对任意两个互质数验证上述结论,并输出验证的中间结果。

第 6 章 程序单元和过程

一个 FORTRAN 程序可以由一个以上的程序单元组成,这些程序单元分别称为主程序单元和子程序单元。每个程序单元都可由 FORTRAN 编译程序对它单独进行编译(进行语法检查,生成目标代码),然后再连接在一起,形成可执行程序。前几章介绍的 FORTRAN 程序一般只由一个主程序单元组成,所有的功能都在主程序单元内完成,这不符合程序结构化的要求。本章介绍的程序单元和过程是解决这一问题的一种较好的方法。

6.1 概述

过程是可供调用的一个子程序单元,可供程序中任何需要它的地方调用。这既能避免重复性编程劳动,又能节省编程时间与程序的存储空间。但一个 FORTRAN 程序必须有一个主程序单元,而且只能有一个主程序单元。一个程序中可以根据需要包含任意数量的子程序单元,子程序单元可以被主程序单元调用,也可以被其它子程序单元调用,甚至可以递归调用。主程序单元一般由 PROGRAM 语句开头,以 END 语句结束。

FORTRAN 程序中的过程有两种:函数子程序过程(简称函数子程序或函数)和子例子程序过程(简称子例子程序或子例行子程序)。这两者的形式、功能与调用方法都有所不同,但总称为过程子程序,也简称为过程或子程序。

在一些书上,子程序又称为辅程序,函数子程序又称作函数辅程序,子例子程序又称作子例辅程序。

由于子程序单元是独立的程序单元,因此可将一些常用的标准程序编写成子程序单元放在程序库中供使用者调用。我们已经多次调用过机内的内在过程,例如下列赋值语句:

$$X = \text{ABS}(4.5) + \text{LOG10}(80.5)$$
$$Y = \text{COS}(3.5) - \text{SIN}(2.6)$$

这里的 ABS、LOG10、COS、SIN 等为已编好的内在函数子程序,分别用来求实数 X 的绝对值、常用对数、正弦值和余弦值。自变量 X 在子程序编写时没有具体值,使用时要赋给具体的值,我们把这种自变量称为虚元(或称哑元、形参),在调用时必须把实际的值代入 X,该实际的量就称为实元(或称实参)。虚元和实元又统称为过程变元(简称变元)。例如求 X 的绝对值的函数为 ABS(X),求 4.5 的绝对值时应使用 ABS(4.5),4.5 就是实元。用实元 4.5 替代虚元 X 的过程称作“虚实结合”。对过程有以下说明:

1° 过程作为子程序不能独立运行,只供其它程序单元调用。

2° 过程的调用方法是书写过程名,后跟用一对圆括号括起来的实元表。

3° 过程单元中虚元的个数可以有多个,它们都要写在一对圆括号内(称作虚元表),列在过程名之后。虚元可分为必选虚元与可选虚元,如 INT(A,KIND)的第一个虚元 A 是必写的,称作必选虚元;它的第二个虚元 KIND 是可选的,称作可选虚元。引用时,必选虚元一定要有实元相对应,可选虚元可以没有相对应的实元;有时也可以没有虚元,此时只在函数名后写一对空括号。

4° 当一个过程被调用时,调用它的主程序或子程序称作主调程序,该过程被称作被调程序。

5° 主调程序和被调程序间的数据传递主要是通过虚实结合实现。因为即使主调程序与被调程序使用了相同的变元名,其数据传递仍然按虚实结合进行,与实元名、虚元名是否相同无关。

6.2 函数子程序

函数子程序一般用来计算一个值,这个值可以用在表达式中。像一般变量一样,函数子程序也有一个名字,且函数值要具有一定的类型和种别。

6.2.1 函数子程序的编写方法

FORTRAN 提供了由程序员自己定义函数的手段,即程序员可以根据需要编写一段程序,并将它组成一个函数,形成一个独立的程序单元。这种函数称为外部函数,它的引用方式与系统内已有的内在函数的引用方式完全相同。

[例 6-1] 编写求 $\sum_{i=1}^n i$ (即 $1+2+3+\cdots+n$) 的值的函数子程序。

```
FUNCTION ISUM(N) RESULT(ISUM_RESULT)
  IMPLICIT NONE
  INTEGER::N, ISUM_RESULT, I
  ISUM_RESULT = 0
  DO I = 1, N
    ISUM_RESULT = ISUM_RESULT + I
  END DO
END FUNCTION ISUM
```

第一行的关键字 FUNCTION 表示函数开始, ISUM 是函数子程序名, N 是虚元, 关键字 RESULT 后边的 ISUM_RESULT 是用于存放函数值的结果变量。DO 结构结束后, 变量 ISUM_RESULT 的值便是前 N 个自然数的和。若求 1 至 100 之间自然数的和, 主程序单元应为:

```
PROGRAM MAIN_ISUM
  IMPLICIT NONE
  INTEGER::ISUM, S
  S = ISUM(100)
  PRINT *, S
END PROGRAM MAIN_ISUM
```

运行结果如下:

5050

由此可见,函数子程序必须提供一个值,这个值赋给结果变量,成为函数的值。结果变

量名在函数中说明,它同其它局部变量一样使用,但这个结果变量的值要作为函数值返回到主调程序中。上例的主程序单元中要调用函数子程序 ISUM,该函数子程序名 ISUM 要用说明语句说明类型,其类型就是函数子程序中 ISUM_RESULT 的类型,而函数调用 ISUM(100)就是 N 取 100 时的 ISUM_RESULT 的值,即前 100 个自然数的和。

应当注意到:引用函数子程序时写出的是函数名,但函数的返回值却是函数的结果值。每一个函数子程序中都要使用一个结果变量。

为了能正确地编写函数子程序,现对函数子程序的编写规则作如下说明:

1. 函数子程序用函数语句开始,该语句的一般形式是:

[RECURSIVE] FUNCTION 函数名(虚元表) RESULT(结果名)

其中:

1° 关键字 FUNCTION 表明该程序单元是函数子程序单元。

2° 函数名和普通变量名的取法相同。它在过程中只是一个名,不允许用说明语句说明其类型等,也不允许在函数过程体内出现。

3° 虚元表内的虚元彼此用逗号分隔,都要在说明语句中说明类型。如果没有虚元,函数名后是一对空括号。

4° RESULT 是关键字,其后括号内的结果变量用于存放计算的结果,其变量名通常写成“函数名_结果”的形式,它必须在说明语句中说明类型,在程序执行部分中至少赋值一次,在函数被引用时它的值就是函数值。函数结果名不可列入虚元表。

5° 如果起始语句为由关键字 RECURSIVE 开头,则函数子程序是递归的。例如有两个虚元的递归函数 AF 的函数语句为:

RECURSIVE FUNCTION AF(X,Y) RESULT(AF_RESULT)

2. 函数结束语句是函数过程结束的标志,并具有将控制返回到主调程序中的作用,它的一般形式是:

END [FUNCTION [函数名]]

其中 END FUNCTION 是关键字,函数名是函数语句中的函数名,函数名也可以省略。

3. 函数过程体是函数语句和函数结束语句之间的程序段,可分为说明部分和执行部分。说明部分中应对程序体内一切变量、数组等实体(包括变元)做出说明,同时还要说明函数结果变量名,但不可说明函数名。执行部分中必须对函数结果名至少赋值一次,作为函数值。

4. 虚元可以是变量名、数组名、过程名、指针等。在说明虚元时还有一个可选属性 INTENT,用来说明该虚元是从主调程序传入值,还是向主调程序传送值。例如:

INTENT(IN) ! 表示函数开始时虚元从实元中获得值

INTENT(OUT) ! 表示函数结束时把虚元的值传送给实元

INTENT(INOUT) ! 表示虚元既要从实元中获得值,又要向实元传送值

例 6-1 中说明了三个变量 ISUM_RESULT、I 和 N, ISUM_RESULT 是函数结果名,故不可有 INTENT 属性说明;I 是 DO 结构的循环变量,也不能有 INTENT 属性说明;N 的值未定,必须从主调函数中获得值,因此列为虚元,其使用意图是从主调程序中引进值,因此可以说明属性 INTENT 是(IN)。这样,函数内的说明语句可改写为:

INTEGER, INTENT(IN)::N

INTEGER::ISUM_RESULT, I

由于主程序中没有虚元,因此它不应有 INTENT 属性说明。

6.2.2 函数调用

函数子程序的调用形式与内在函数的引用形式相同,函数子程序的调用应出现在表达式中,其一般形式是:

函数名(实元表)

如例 6-1 中由语句 $S = \text{ISUM}(100)$ 完成对函数子程序的调用。但函数的类型必须在调用程序中说明,并要同被调函数结果变量的类型相同。

一般而言,在进行函数调用时,虚实结合应遵循以下原则:

实元与虚元个数、类型、位置一一对应,但实元与虚元名字可以不同。有时上述原则也可以灵活运用。例如虚元表中有可选变元时,其实元表可能不选某些变元,使实元的个数比虚元少。又如,当虚元为简单变元时,与之结合的实元可以是常数、简单变量、数组元素或表达式。若实元是表达式时,系统先计算表达式的值,而后用该值与虚元结合,求得函数值。只有虚元具有属性 INTENT(IN) 时,实元可以取上述多种形式。如果虚元属性为 INTENT(OUT),与之结合的实元只能是变量或数组元素等,不许是表达式或常数。如果函数无虚元,通过函数名后跟一对空括号实现。下面再看一个例子。

[例 6-2] 编写函数子程序,求 $\frac{x}{1} + \frac{x^2}{1+2} + \frac{x^3}{1+2+3} + \cdots + \frac{x^n}{1+2+3+\cdots+n}$ 前 n 项的和, x 和 n 的值在调用时给定。程序如下:

```
PROGRAM MAIN_SUM1
  IMPLICIT NONE
  REAL::SUM1,X,S
  INTEGER::N
  READ *,N,X
  S=SUM1(X,N)
  PRINT *, 'S=',S
END PROGRAM MAIN_SUM1

FUNCTION SUM1(X,N) RESULT(SUM1_RESULT)
  IMPLICIT NONE
  REAL::SUM1_RESULT,X
  INTEGER::N,I,J,T
  SUM1_RESULT=0
  DO I=1,N
    T=0
    DO J=1,I
      T=T+J
    END DO
    SUM1_RESULT=SUM1_RESULT+X**I/T
  END DO
```

```
END FUNCTION SUM1
```

运行结果如下:

30,2.5✓

S= 3.2631311E+09

6.2.3 接口块

一个过程接口块用于通知编译程序主调程序调用过程时所需要的接口信息,主要包括过程中各个虚元的类型、属性等。接口块写在主调程序的说明部分,一般应写在类型说明语句之前,其内容是被调用过程中的有关说明部分。

功能简单的程序,不必写接口块,例如上面求前 n 个自然数之和的程序。但也可以增加接口块使调用关系更清晰。

[例 6-3] 用接口块实现例 6-1 的程序

```
PROGRAM EXAM_INTERFACE
  IMPLICIT NONE
  INTERFACE
    FUNCTION ISUM(N) RESULT(ISUM_RESULT)
      INTEGER::ISUM_RESULT
      INTEGER,INTENT(IN)::N
    END FUNCTION ISUM
  END INTERFACE
  INTEGER::S
  S=ISUM(100)
  PRINT *, 'S=', S
END PROGRAM EXAM_INTERFACE

FUNCTION ISUM(N) RESULT(ISUM_RESULT)
  IMPLICIT NONE
  INTEGER::ISUM_RESULT, I
  INTEGER,INTENT(IN)::N
  ISUM_RESULT=0
  DO I=1,N
    ISUM_RESULT=ISUM_RESULT+I
  END DO
END FUNCTION ISUM
```

其中 INTERFACE 语句到 END INTERFACE 语句之间的部分就是接口块。显然,接口块的内容就是被调用过程的变元及函数返回变量等说明部分的拷贝。使用接口块应注意:

1. 接口块的位置应在主调程序的说明部分中。含有接口块的过程一般格式如下:

PROGRAM(或 FUNCTION、或 SUBROUTINE)程序名

接口块
主调程序内变元说明
执行语句

END PROGRAM(或 FUNCTION、或 SUBROUTINE)

接口块的一般形式是:

INTERFACE [类属名或赋值规格说明]
 函数子程序语句(或子例子程序语句)
 被调用过程各变元及函数结果值的说明
 函数结尾语句(或子例子程序结尾语句)
END INTERFACE

由此可见:接口块的起始语句包括关键字 INTERFACE,之后是操作的类属名或赋值规格说明,这两项为选择项。接口块由含 END INTERFACE 的语句结束,在它们之间只能有过程的虚元的说明和结果变量的说明,不允许出现任何可执行语句。在 INTERFACE 与 END INTERFACE 语句间,可以说明多个被调用过程的接口,这些接口由各个过程开始语句与结束语句定界,其排列先后次序任意。

2. 遇到下列几种情况时,主调程序必须有接口块:

- (1)实元是关键字变元时;
- (2)实元是缺省的可选变元时;
- (3)一个外部函数使用系统中的内部运算符扩展了原有的功能时;
- (4)外部过程扩展了赋值号的使用范围时;
- (5)使用一个类属名调用过程时;
- (6)如果调用一个外部过程,该过程的结果为数组(只有函数如此);或虚元有一个假定形状数组、一个指针变量、一个目标变量;或者过程是一个函数,其函数结果值是一个字符型,且其长度不是常数,也不是假定长度(*)时。

6.3 子例子程序

过程的另一种形式是子例子程序,它和函数子程序主要有两方面不同:

1. 函数子程序必须有返回值,即在 FUNCTION 语句中一定有 RESULT 关键字,并在它后边括号内有一个结果变量,它表示函数的返回值。而子例子程序本身并不返回值,它的计算结果全都通过虚实结合传递给主调程序。有时它所做的操作可以与算术等运算无关。

2. 函数子程序的调用就像通常的函数引用一样,一般出现在表达式中。而子例子程序必须通过 CALL 语句来调用,其一般形式为:

CALL 子程序名(实元表)

从形式上看,子例子程序的结构除了第一句和最后一句外,其余部分的编写形式与函数编写形式完全相同。子例子程序语句的一般形式是:

[RECURSIVE] SUBROUTINE 子程序名(虚元表)

其中 RECURSIVE 是可选的,当子程序有递归功能时才用到。SUBROUTINE 是关键字,子程序名只是一个名,并不代表返回任何值,故子程序语句中不需要 RESULT 部分。虚元表的写法与函数相同。如果子程序没有虚元,可写成

SUBROUTINE 子程序名()

或

SUBROUTINE 子程序名

子程序结束语句的一般形式为:

END [SUBROUTINE [子程序名]]

方括号的内容可省略,子程序名必须与子例子程序语句中的子程序名一致。该语句的作用是把控制返回主调程序,同时也表示子程序的定义结束。

子程序语句和子程序结束语句之间的部分是子程序体,它的写法也是先写说明部分后写执行部分,与函数体和主程序的规定基本相同。几乎所有的函数都可改写成子例子程序。如:

[例 6-4] 将例 6-1 改由子例子程序实现。

```
SUBROUTINE ISUM(N,ISUM_ VALUE)
  IMPLICIT NONE
  INTEGER,INTENT(OUT)::ISUM_ VALUE
  INTEGER,INTENT(IN)::N
  INTEGER::I
  ISUM_ VALUE=0
  DO I=1,N
    ISUM_ VALUE= ISUM_ VALUE + I
  END DO
END SUBROUTINE ISUM
```

由于虚元变量 ISUM_ VALUE 是在子程序中被赋值,并且通过虚实结合将值传到主调程序中,故可以将它的 INTENT 属性说明为 OUT。由于该程序不属于必须写接口条件中的任何一条,因此主程序中可以不写接口块。主程序简单地编写如下:

```
PROGRAM EXAM_ SUB
  IMPLICIT NONE
  INTEGER::ISUM_ X,X
  READ *,X
  CALL ISUM(X,ISUM_ X)
  PRINT *, 'SUM=', ISUM_ X
END PROGRAM EXAM_ SUB
```

6.4 虚实结合

按照虚实结合的一般原则,要求实元和虚元的个数相等,相应位置上的类型应该一致,否则就会出错。但在使用某些现代特性时,这个原则也是可以灵活的。主要表现在:关键字变元、可选择变元、虚元改名三种形式上。另外,数组也可以作为虚元。

6.4.1 关键字变元

通常在调用过程中要记住虚元名及其位置,而使用关键字变元后,就不必记住虚元的次序了,填写的实元次序可以是任意的。

关键字变元是调用过程的一种现代形式,它的一般形式为:

虚元名 = 实元表达式

例如对于子例子程序语句(对函数子程序语句一样可用):

```
SUBROUTINE SUM(M,SUM_ VALUE1,SUM_ VALUE2)
```

主程序调用语句使用关键字变元时形式如下:

```
CALL SUM(M=N,SUM_ VALUE2=SUMY,SUM_ VALUE1=SUMX)
```

由此可见,有了关键字变元,就无需把变元按虚元的顺序排列,但必须知道虚元的名字。该语句表明虚元 M 与实元 N 结合,虚元 SUM_ VALUE1 与实元 SUMX 结合,虚元 SUM_ VALUE2 与实元 SUMY 结合,这三个关键字变元的位置可以任意。当用部分变元的关键字调用子程序时,第一个关键字变元前面的所有变元都必须与相应的虚元一一对应,一旦使用一个关键字变元,其后面的变元也必须使用关键字变元。例如对子例子程序语句:

```
SUBROUTINE HF(A,B,M,N)
```

如果希望 A 与 5 结合,B 与 25 结合,M 与 10 结合,N 与 1 结合,主调程序调用时可以采用如下几种形式中的任意一种:

```
CALL HF(5,25,10,1)
```

```
CALL HF(B=25,M=10,A=5,N=1)
```

```
CALL HF(5,25,N=1,M=10)
```

但不允许用如下调用形式:

```
CALL HF(25,5,N=1,M=10)
```

```
CALL HF(5,25,M=10,1)
```

请读者自己指出其中的错误。

主调程序中如采用关键字变元调用过程,就必须写出被调子程序的接口块。假设 HF 过程为:

```
SUBROUTINE HF(A,B,M,N)
```

```
  INTEGER,INTENT(IN)::A,B,M,N
```

```
  PRINT *,A*B+M*N
```

```
END SUBROUTINE HF
```

主调程序必须写出该子程序的接口块:

```
INTERFACE
```

```
  SUBROUTINE HF(A,B,M,N)
```

```
    INTEGER,INTENT(IN)::A,B,M,N
```

```
  END SUBROUTINE HF
```

```
END INTERFACE
```

6.4.2 可选择变元

FORTRAN90 子程序的虚元表中可包含可选择项。因此在调用时可以根据实际的需

要,只对虚元表中部分虚元作虚实结合。可选择变元必须在过程中被说明成是可选择的,即要求它具有 OPTIONAL 属性,同时在过程中需要使用 PRESENT 函数。在调用过程时,其主调程序中应加入被调过程的接口块。调用时可选变元可以与实元结合,也可在实元表中不出现,但必选变元不可省。使用可选变元应注意以下两点:

1° OPTIONAL 属性:若一个虚元的类型说明语句中有 OPTIONAL 属性,便是可选变元,没有该属性的是必选变元。

2° PRESENT(X)函数:用来检查它的自变量 X 是否在程序执行部分中出现。当 X 出现时,函数值为“真”;当 X 不出现时,函数值为“假”。

[例 6-5] 编写一个函数子程序。根据需要,既能求出圆柱的底面积值,也能求出圆柱的体积值。此时可将高度 H 列为可选择变元。

```
FUNCTION SS(R,H) RESULT(SS_RESULT)
  IMPLICIT NONE
  REAL,INTENT(IN)::R
  REAL,OPTIONAL,INTENT(IN)::H
  REAL::SS_RESULT
  REAL::TEMP_H
  IF (PRESENT(H)) THEN
    TEMP_H=H
  ELSE
    TEMP_H=1.0
  ENDIF
  SS_RESULT=3.141593 * R * * 2 * TEMP_H
END FUNCTION SS
```

在子程序中通过 PRESENT 函数测试变元 H 是否出现。当 H 出现时把 H 中的值赋给 TEMP_H,若 H 不出现,则让 TEMP_H 的值为 1.0。故当 H 不出现时,就是求圆柱底面积。

6.4.3 通过接口块更改虚元名称

在 FORTRAN90 中允许改变关键字变元名称,名称的改变在接口块中进行,所以主调程序要改变虚元名称必须写出接口块。

在例 6-5 的中,如果调用函数时希望把函数子程序中虚元名 R、H 改为物理意义很明确的 RADIUS、HEIGHT(半径、高度),只需要在主调程序中改写接口块(若原来没有接口块则增加接口块),使其虚元表中用新的虚元名代替原名,但要保持新、旧虚元名的对应位置不变。例 6-5 中改虚元名后的主程序如下:

```
PROGRAM EXAM_SS
  IMPLICIT NONE
  INTERFACE
    FUNCTION SS(RADIUS,HEIGHT) RESULT(SS_RESULT)
```

```

      REAL,INTENT(IN)::RADIUS
      REAL,OPTIONAL,INTENT(IN)::HEIGHT
      REAL::SS,RESULT
      END FUNCTION SS
END INTERFACE
REAL::R1,R2,H1
READ *,R1,H1
READ *,R2
PRINT *,SS(RADIUS=R1,HEIGHT=H1)
PRINT *,SS(RADIUS=R2)
END PROGRAM EXAM. SS

```

6.4.4 虚数组

虚元也可以是一个数组名。如果被调过程中某个数组中各元素值需由主调程序给出,或过程中计算出的某个数组各元素值要传递给主调程序,该数组应列为虚元。数组名写入虚元表中时,实元应是形状相同的数组或其成分。虚数组通常可分为常数组,假定大小数组,可调数组等各种情况。当函数返回结果为数组,或过程虚元为一个假定形状数组时,主调程序要增加接口块。虚数组详细的使用方法将在后面的章节中介绍。

6.5 子程序的嵌套调用

FORTRAN 语言允许子程序的嵌套调用(也称多层调用),即在调用一个子程序的过程中该被调子程序又调用了另一个子程序。图 6-1 中主程序调用子程序 A 称为第一层调用,被调用的子程序 A 调用下属子程序 B,称为第二层调用,依次类推。

6.5.1 嵌套调用

在嵌套调用中,函数子程序和子例子程序均可以调用下属的函数和子例子程序,并不要求被调用的过程与自己是同类的子程序。

图 6-1 描述的是两层嵌套(连主程序共有 3 层程序单元)。其执行过程是:

- ①执行主程序的第一个可执行语句,顺次往下执行;
- ②遇到调用下层子程序 A 操作语句后,流程转入子程序 A 的第一条可执行语句,并将主程序中调用时的实元与子程序 A 的虚元进行虚实结合;
- ③顺次执行子程序 A 的各个语句;
- ④遇到调用下层子程序 B 的语句时,流程转入子程序 B 的第一条可执行语句,并将该层调用时的实元与子程序 B 的虚元进行虚实结合;
- ⑤顺次执行子程序 B,由于 B 未调用其它子程序,则执行子程序 B 的全部操作,直到遇到子程序 B 的结束语句;

⑥返回到上层子程序调用语句处,即返回子程序 A;

⑦继续执行 A 中未执行的语句,直到该子程序结束;

⑧返回到主程序调用子程序 A 的地方;

⑨继续执行主程序中剩余部分,直到主程序结束。

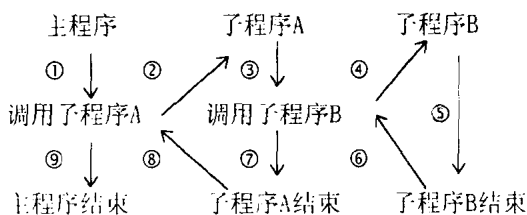


图 6-1 子程序嵌套调用

[例 6-6] 用弦截法求方程 $x^3 - 5x^2 + 16x - 80 = 0$ 的根。方法如下:

(1) 设 $f(x) = x^3 - 5x^2 + 16x - 80$ 。

(2) 输入精度要求 ϵ 和两个不同的 x 值: x_0 和 x_1 。

(3) 用弦截法公式:

$$x_{k+1} = x_k - \frac{f(x_k)}{f(x_k) - f(x_0)} (x_k - x_0)$$

求出 x_2 的值。该值实际上是过 P_0 和 P_1 两点的弦线与 X 轴的交点的值;利用上式求出 x_3 ,该值实际上是过 P_0 和 P_2 两点的弦线与 X 轴的交点的值;...;这样求下去,可得到 x 的序列 $x_0, x_1, x_2, \dots, x_k, \dots$,从而获得满足精度要求的根。

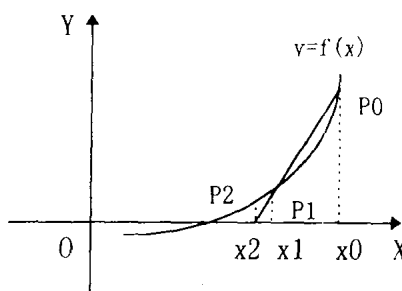


图 6-2 弦截法示意图

为了尽快得到满足要求的根,可以使用快速弦截法。其迭代公式为:

$$x_{k+1} = x_k - \frac{f(x_k)}{f(x_k) - f(x_{k-1})} (x_k - x_{k-1})$$

用该式求 x_2 时与普通弦截法一样,但再利用上式求 x_3 时,该值实际上是过 P_1 和 P_2 两点的弦线与 X 轴的交点的值(不再是 P_0 和 P_2 两点的弦线与 X 轴的交点的值),以后总是用最新求出的两个点做弦线取出新的 x 点。如此求下去,可得到 x 的序列 $x_0, x_1, x_2, \dots, x_k, \dots$,直到 $|x_{k+1} - x_k| < \epsilon$ 为止,这时用 x_{k+1} 作为方程 $f(x) = 0$ 的近似根。

根据上述思路画出程序的 N-S 流程图(图 6-3),并分别用以下函数来实现各部分功能:

(1) 用函数 $F(X)$ 求 x 的函数值 $f(x) = x^3 - 5x^2 + 16x - 80$ 。

(2) 用函数 $NEW_XPOINT(X1, X2)$ 求 $(X1, f(X1))$ 和 $(X2, f(X2))$ 的弦线与 x 轴交点的坐标 X 。

(3) 用函数 $ROOT(X1, X2, E)$ 求满足精度的实根。执行 $ROOT$ 过程中要用到过程 NEW_XPOINT ,而执行 NEW_XPOINT 过程中要用到函数 F 。

程序如下:

```
PROGRAM NESTED_PROCEDURE
```

```
IMPLICIT NONE
```

```
REAL :: X1, X2, X, E, ROOT
```



```

      READ *,X1,X2,E
      X=ROOT(X1,X2,E)
      PRINT *, 'X=' ,X
END PROGRAM NESTED_ PROCEDURE
FUNCTION F(X) RESULT(F_RESULT)
  IMPLICIT NONE
  REAL,INTENT(IN)::X
  REAL::F_RESULT
  F_RESULT=((X-5)*X+16)*X-80
END FUNCTION F
FUNCTION NEW_XPOINT(X1,X2) RESULT
(XPOINT_RESULT)
  IMPLICIT NONE
  REAL,INTENT(IN)::X1,X2
  REAL::XPOINT_RESULT,F
  XPOINT_RESULT=X2-F(X2)/(F(X2)-F(X1))*(X2-X1)
END FUNCTION NEW_XPOINT
FUNCTION ROOT(XK1,XK2,E) RESULT(ROOT_RESULT)
  IMPLICIT NONE
  REAL,INTENT(IN)::XK1,XK2,E
  REAL::ROOT_RESULT,NEW_XPOINT,X1,X2,X
  X1=XK1; X2=XK2
  DO WHILE (ABS(X2-X1)>E)
    X=NEW_XPOINT(X1,X2)
    X1=X2; X2=X
  END DO
  ROOT_RESULT=X2
END FUNCTION ROOT

```

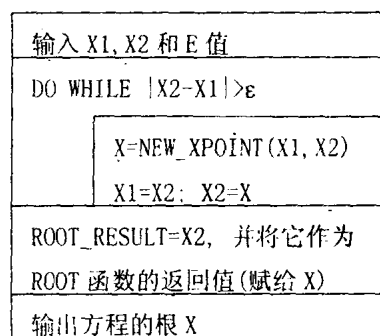


图 6-3 用弦截法求方程的根

运行结果如下:

2,6,1E-5

X= 5.0000000

该程序输入相应的数据之后,先调用了函数子程序 ROOT,在该子程序中循环调用函数子程序 NEW_XPOINT,不断地求过 $y=f(x)$ 上两点的弦线与 X 轴的交点,使其不断地向方程的根逼近。图 6-3 描绘了本程序的示意流程,实际上每个程序单位应单独画一个 N-S 图。程序中共有四个程序单位:一个主程序,三个子程序。其主要的嵌套调用关系如图 6-4 所示。

主调程序调用函数子程序时,要在主调程序中对被调函数的类型进行说明。

6.5.2 虚过程

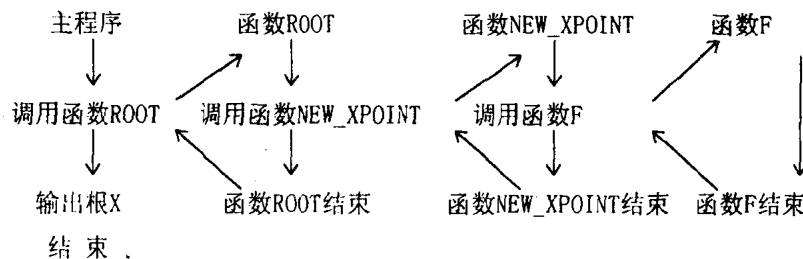


图 6-4 例 6-6 中各函数之间的调用关系

如果虚元是一个过程名时,作为虚元的过程称为虚过程。只有嵌套调用至少两层以上时才能用虚过程。虚过程的使用增加了程序的功能,使程序更灵活。

假设主程序调用过程 A,A 中又调用过程 B(函数或子例子程序),但过程 B 的功能可以根据需要随时更换,如这次调用的 B 是 SIN,下次调用的 B 又是 COS,或其它外部函数过程,此时可将 B 列为虚过程。具体使用虚过程的方法是:

在过程 A 中任意虚拟一个过程名(假设是 P)并把 P 列入 A 的虚元表中,在过程 A 的执行部分调用过程 P,则 P 就是虚过程。主程序调用 A 时,若实元表内用 SIN 与 P 结合,则 A 中调用 P 就相当于调用了调用 SIN;若实元表内用 COS 与 P 结合,则 A 中就是调用 COS;若用外部函数与 P 结合,则 A 中就是调用该函数子程序。如果 P 是子例子程序,虚实结合时要用子例子程序名作为实元。使用虚过程时,在主调程序 A 中要对实过程名作特殊说明,以便让编译系统清楚该实元不是一般的变量,而是一个子程序。这可以用接口块来实现。当使用内在基本函数作实元与虚过程结合时,只许用基本函数的专用名,不可用类属名。

[例 6-7] 设有一个过程 PROC,在调用它的时候,每次实现不同的功能。输入 a 和 b 两个数,第一次调用时求出 a、b 之和,第二次调用时求 a、b 之差。程序如下:

```

PROGRAM ACTUAL_PROCEDURE
  IMPLICIT NONE
  INTERFACE
    FUNCTION SUM(X,Y) RESULT(SUM_RESULT)
      INTEGER,INTENT(IN)::X,Y
      INTEGER::SUM_RESULT
    END FUNCTION SUM
    FUNCTION MINU(X,Y) RESULT(MINU_RESULT)
      INTEGER,INTENT(IN)::X,Y
      INTEGER::MINU_RESULT
    END FUNCTION MINU
  END INTERFACE
  INTEGER::A,B
  READ *,A,B
  CALL PROC(A,B,SUM)
  CALL PROC(A,B,MINU)

```

```

END PROGRAM ACTUAL_ PROCEDURE
SUBROUTINE PROC(A,B,FUN)
  IMPLICIT NONE
  INTERFACE
    FUNCTION FUN(X,Y) RESULT(FUN_ RESULT)
      INTEGER, INTENT(IN)::X,Y
      INTEGER::FUN_ RESULT
    END FUNCTION FUN
  END INTERFACE
  INTEGER, INTENT(IN)::A,B
  PRINT *,FUN(A,B)
END SUBROUTINE PROC
FUNCTION SUM(X,Y) RESULT(SUM_ RESULT)
  IMPLICIT NONE
  INTEGER, INTENT(IN)::X,Y
  INTEGER::SUM_ RESULT
  SUM_ RESULT=X+Y
END FUNCTION SUM
FUNCTION MINU(X,Y) RESULT(MINU_ RESULT)
  IMPLICIT NONE
  INTEGER, INTENT(IN)::X,Y
  INTEGER::MINU_ RESULT
  MINU_ RESULT=X-Y
END FUNCTION MINU

```

运行结果如下：

```

6,2✓
8
4

```

其中 SUM、MINU 均为外部函数，用接口块进行说明。在主程序中第一次调用函数 PROC 时，除了将 A、B 两实元传给 PROC 的虚元 X、Y 外，还将函数名 SUM 传给虚过程 FUN，执行 PROC 可以调用函数 SUM(A,B) 求得 A、B 之和。同理第二次调用函数 PROC 时，将函数名 MINU 传给虚过程 FUN，执行 PROC 过程中便完成了调用函数 MINU(A,B)，求得 A、B 之差。

6.6 模 块

各过程间共享数据的一种方法是通过虚实结合实现，但它并非总是最有效的。模块是提供共享常量、变量、类型说明和过程的一种有效的方法；它可把过程放入不同的程序单元都可调用的“库”。模块也是一种独立编写的程序单元，它具有其独特的形式。模块包含程

序说明语句及过程(在块内定义的过程),但它不能直接运行。其它程序单元引用模块时,实际上就是把该模块内的全部语句复制到本程序单元中,并且所有与模块中的名字相同的变量彼此取值相通,共享存储单元,所以引用模块起两个作用:共享与复制。

6.6.1 模块定义

模块程序单元的一般形式是:

```
MODULE 模块名                ! MODULE(模块开始)语句
  类型说明部分
  [CONTAINS
    内部过程子程序 1
    ...
    内部过程子程序 n]
END MODULE [模块名]
```

MODULE 是关键字,MODULE 语句标志模块程序单元开始。模块名通常加后缀“_MODULE”(也可以不加该后缀),以增加可读性。例如:

```
MODULE A_MODULE
```

MODULE 语句下边是一些说明部分(类型说明语句、派生类型定义及接口块等),但不能有执行部分。从 CONTAINS 语句开始及以后的各过程是可选的。当模块有内部过程时,必须把整个过程完整地写入,各内部过程次序可以任意。一个程序中可以有多个模块程序单元,每个模块都要独立编写。

假设有几个子程序或函数中都用到了同一些变量,只需把它们放在一个模块中,这些子程序或函数通过引用该模块,便相当于将该模块移到自己的说明部分中,从而实现共享数据。模块中也可定义派生类型,以便其他过程使用它来说明该派生类型的变量。

模块名对程序来说是全局的,不能与本程序内的任何程序单元名、外部名相同,也不得与模块内的任何局部名重名。

6.6.2 模块的引用

USE 语句是一个模块引用的说明,通常在程序单元说明部分的最前面。USE 语句一般形式为:

```
USE 模块 1,模块 2,...,模块 n
```

程序单元有此语句,就相当于把模块 1、模块 2、...、模块 n 的语句内容都移植到本程序单元内,从而达到共享模块内容的目的。

[例 6-8] 编程序读入 3 个实数,调用函数 AVER3 求它们的平均值,调用函数 MAX3 求它们的最大值。

方法 1:通过虚实结合传递数据。

```
FUNCTION AVER3(A,B,C) RESULT(AVER_VALUE)
  IMPLICIT NONE
  REAL::A,B,C
```

```

    REAL::AVER_VALUE
    AVER_VALUE = (A + B + C)/3.0
END FUNCTION AVER3
FUNCTION MAX3(A,B,C) RESULT(MAX_VALUE)
    IMPLICIT NONE
    REAL::A,B,C
    REAL::MAX_VALUE
    MAX_VALUE = A
    IF(B > MAX_VALUE) MAX_VALUE = B
    IF(C > MAX_VALUE) MAX_VALUE = C
END FUNCTION MAX3
PROGRAM AVER_MAX_1
    IMPLICIT NONE
    REAL::A,B,C
    REAL::AVER3,MAX3
    READ *,A,B,C
    PRINT *,AVER3(A,B,C),MAX3(A,B,C)
END PROGRAM AVER_MAX_1

```

运行结果如下:

28.0, 75.4, - 23.6 ✓

26.6000004 75.4000015

从上面各程序单元中可以看出,在说明部分都有这样两个语句:

```
IMPLICIT NONE
```

```
REAL::A,B,C
```

并且 A、B、C 在主程序中作实元,与函数中的 3 个虚元传递数据。这也可以通过共享模块内数据来实现。

方法 2: 用模块来实现数据共享。

```

MODULE EXAM_MODULE
    IMPLICIT NONE
    REAL::A,B,C
END MODULE EXAM_MODULE
FUNCTION AVER3() RESULT(AVER_VALUE)
    USE EXAM_MODULE
    REAL::AVER_VALUE
    AVER_VALUE = (A + B + C)/3.0
END FUNCTION AVER3
FUNCTION MAX3() RESULT(MAX_VALUE)
    USE EXAM_MODULE
    REAL::MAX_VALUE

```

```

MAX_ VALUE = A
IF(B>MAX_ VALUE)MAX_ VALUE = B
IF(C>MAX_ VALUE)MAX_ VALUE = C
END FUNCTION MAX3
PROGRAM AVER_ MAX_ 2
  USE EXAM_ MODULE
  REAL::AVER3,MAX3
  READ *,A,B,C
  PRINT *,AVER3(),MAX3()
END PROGRAM AVER_ MAX_ 2

```

程序中各单元通过语句 USE EXAM_ MODULE 使变量 A、B 和 C 成为共享数据。在主程序中输入 3 个数放入 A、B 和 C 后,函数 AVER3 和 MAX3 中的 A、B 和 C 也就具有了同样的值。因此不必再把 A、B、C 作为变元,使 AVER3 和 MAX3 成为无参数函数。当然,若在某程序单元中改变了共享变量的值,其它程序单元中相应的共享变量的值也要发生同样的变化。

应从以下几个主要方面理解模块的概念:

- 1° 通过模块共享可以取代程序单元间的虚实结合,使子程序减少或不用虚元。
- 2° 允许共享模块中的部分变量,其形式为:

```
USE 模块名,ONLY:实体名表
```

即在引用模块时只允许模块内实体名表中列出的实体与本程序单元共享,与其它实体没有共享关系。若在主程序单元中有

```
USE EXAM_ MODULE,ONLY:A,B
```

则在主程序单元中,变量 A、B 通过模块与其它子程序共享。由于 C 不再是共享变量,因此 C 仍需通过虚实结合与各被调过程传递数据。

3° 若模块中实体名与程序单元中的变量不同名,可以通过 USE 语句对模块中的实体改名。假设使模块中实体名 A 与程序单元中变量 X 共享,USE 语句可以写成:

```
USE EXAM_ MODULE,X=>A
```

在本程序单元中将模块中的 A 改名为 X。

- 4° 可以对模块内说明的变量、派生类型及其成员的使用范围进行限制。例如:

```

TYPE,PRIVATE::DATA_ TYPE
  INTEGER::M,N
  REAL::X,Y
END TYPE DATA_ TYPE

```

或

```

TYPE DATA
  PRIVATE
  INTEGER::A
  ...
END TYPE

```

在这两种情况下表示模块中该派生类型、派生类型的内部各成员是“专用的”，模块以外的程序单元不能使用。

5° 模块的内部过程也可供引用模块的各程序单元使用。例 6-8 的方法 2 也可以将函数 AVER3 和 MAX3 放入模块内。其方法如下：

```
MODULE EXAM_MODULE
  IMPLICIT NONE
  REAL::A,B,C
  CONTAINS
    FUNCTION AVER3() RESULT(AVER_VALUE)
      REAL::AVER_VALUE
      AVER_VALUE = (A + B + C)/3.0
    END FUNCTION AVER3
    FUNCTION MAX3() RESULT(MAX_VALUE)
      REAL::MAX_VALUE
      MAX_VALUE = A
      IF(B > MAX_VALUE) MAX_VALUE = B
      IF(C > MAX_VALUE) MAX_VALUE = C
    END FUNCTION MAX3
  END MODULE EXAM_MODULE

PROGRAM AVER_MAX_3
  USE EXAM_MODULE
  READ *, A, B, C
  PRINT *, AVER3(), MAX3()
END PROGRAM AVER_MAX_3
```

每个模块中的函数要完整地书写，函数之间的排列次序任意。本例中的函数没有虚元，因为模块说明部分的 A、B、C 是共享变量，不需要虚实结合。由于主程序已通过 USE 语句将这两个函数的定义包括进来，因此不再需要对调用的函数的类型进行说明的语句

```
REAL::AVER3, MAX3
```

否则就属于重复说明。

6° 通常用模块来建立新的数据类型及规定新类型的操作、赋值等。例如：

```
MODULE TYPE_A
  TYPE REC_TYPE
    INTEGER::NO
    REAL::A,B,C
    CHARACTER(LEN=20)::NAME
  END TYPE
END MODULE
```

当某程序需要使用类型 REC_TYPE 时，便可以用 USE TYPE_A 引用它。

6.7 递归过程

递归是控制程序流程的一种方法,它是 FORTRAN90 新增加的功能。递归调用是指过程内直接或间接地调用自己,能进行递归调用的过程称为递归过程。递归过程可以是递归函数,也可以是递归子例子程序。

6.7.1 递归函数

[例 6-9] 用递归方法求 $n!$

对于非负整数 n ,其阶乘的递归定义如下:

$$n! = 1 \quad (n=0)$$

$$n! = n \cdot (n-1)! \quad (n>0)$$

用这个定义去计算 $4!$ 。 $n=4$ 时,由定义得到 $4! = 4 \times 3!$ 。要完成计算,就要求 $3!$ 的值,而 $3! = 3 \times 2!$, $2! = 2 \times 1!$, $1! = 1 \times 0!$ 。所以 $4! = 4 \times 3 \times 2 \times 1 \times 0!$,再用定义的第一行得: $0! = 1$,这个过程叫递推。由于此时 $0! = 1$ 已求出,便可求出 $1! = 1 \times 0!$,再求出 $2! = 2 \times 1! = 2 \times 1 = 2$, $3! = 3 \times 2! = 3 \times 2 = 6$, $4! = 4 \times 3! = 4 \times 6 = 24$ 。这个过程叫回归。由此可见,递归是递推和回归的总称。将上面的过程综合起来可得: $4! = 4 \times 3 \times 2 \times 1 = 24$ 。

由上面的知识我们将求 $n!$ 的过程描述为:

设 $N!$ 的函数程序名为 $FAC(N)$ 。由于 $N! = N \times (N-1)!$,而求 $(N-1)!$ 实际上仍需调用 FAC ,只是自变量从 N 变为 $(N-1)$,即调用函数 $FAC(N-1)$ 。这样,在求函数 $FAC(N-1)$ 的值时,又可通过 $FAC(N-2)$ 求得。由此可有:

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

$$FAC(N-1) = (N-1) \times FAC(N-2)$$

这样层层递推,直到 $FAC(0)$, $FAC(0)$ 的值是 1。即:

$$FAC(0) = 1$$

上述的多层调用又可一层一层向上回归。即:

$$FAC(1) = 1 \times FAC(0) = 1 \times 1 = 1$$

$$FAC(2) = 2 \times FAC(1) = 2 \times 1 = 2$$

$$FAC(3) = 3 \times FAC(2) = 3 \times 2 = 6$$

...

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

最后可求得 $FAC(N)$,即 $N!$ 的值。

递归过程需要在子程序入口语句前加上关键字 `RECURSIVE`,这样才可在过程体中实现递归调用。其它语句的写法没有特殊要求。下面给出 $N!$ 的递归函数子程序。

```
RECURSIVE FUNCTION FAC(N) RESULT(FAC_RESULT)
```

```
  INTEGER, INTENT(IN)::N
```

```
  INTEGER::FAC_RESULT
```

```
  IF (N == 0) THEN
```



```

        FAC_RESULT=1
    ELSE
        FAC_RESULT=N * FAC(N-1)
    END IF
END FUNCTION FAC

```

该递归函数中规定当虚元 N 的值为 0 时函数结果为 1, $N > 0$ 时, 执行

```

        FAC_RESULT=N * FAC(N-1)

```

右边调用函数本身时, 以 $N-1$ 作为实元, 并进入下一层函数子程序, 只是这下一层子程序仍是 FAC。进入 $FAC(N-1)$ 后, 控制又从函数语句开始, 自上而下执行, 当遇到

```

        FAC_RESULT=(N-1) * FAC(N-2)

```

时, 又调用下一层函数 FAC, 直到 N 的值为 1 遇到

```

        FAC_RESULT=1 * FAC(0)

```

时, 因调用 $FAC(0)$ 时使函数值 $FAC_RESULT=1$, 故 $FAC(0)$ 的值为 1, 递推过程结束。然后继续执行 $FAC_RESULT=1$ 以后的语句。遇到调用 $FAC(0)$ 的 END 语句后返回上一层调用子程序 $FAC(1)$ 中, 得 $FAC(1)$ 的值为 1×1 。再往下执行到 END 语句处, 返回再上一层调用子程序 $FAC(2)$ 中, 这样一层一层回归, 最后可求得函数 $FAC(N)$ 的值。

调用递归函数的主程序的写法与调用非递归函数的写法基本相同。因被调用的是递归函数, 最好写出递归函数接口。主程序如下:

```

PROGRAM RECURSIVE_FUN
    IMPLICIT NONE
    INTERFACE
        RECURSIVE FUNCTION FAC(N) RESULT(FAC_RESULT)
            INTEGER, INTENT(IN)::N
            INTEGER::FAC_RESULT
        END FUNCTION FAC
    END INTERFACE
    INTEGER::N
    READ *, N
    PRINT *, FAC(N)
END PROGRAM RECURSIVE_FUN

```

运行结果如下:

```

5✓
120

```

[例 6-10] 有 5 个人坐在一起, 问第 5 个人多少岁? 他说他比第 4 个人大 2 岁, 问第 4 个人多少岁, 他说他比第 3 个人大 2 岁, 问第 3 个人, 又说比第 2 个人大 2 岁, 问第 2 个人, 说比第 1 个人大 2 岁, 最后问第 1 个人, he 说是 10 岁。请问第 5 个人多大。

显然这是一个递归问题, 要求第 5 个人的年龄, 就必须先知道第 4 个人的年龄, 而要知道第 4 个人的年龄, 必须先知道第 3 个人的年龄, 而第 3 个人的年龄又取决于第 2 个人的年

龄,第2个人的年龄取决于第1个人的年龄,而且每个人的年龄比其前1个人的年龄大2。
用公式表述第n个人的年龄如下:

$$age(n) = \begin{cases} 10 & (n = 1) \\ age(n-1) + 2 & (n > 1) \end{cases}$$

求第5个人的年龄的程序为:

```
PROGRAM RECURSIVE_ AGE
  INTERFACE
    RECURSIVE FUNCTION AGE(N) RESULT(AGE_ RESULT)
      INTEGER, INTENT(IN)::N
      INTEGER::AGE_ RESULT
    END FUNCTION AGE
  END INTERFACE
  PRINT *, AGE(5)
END PROGRAM RECURSIVE_ AGE

RECURSIVE FUNCTION AGE(N) RESULT(AGE_ RESULT)
  INTEGER, INTENT(IN)::N
  INTEGER::AGE_ RESULT
  IF(N == 1) THEN
    AGE_ RESULT = 10
  ELSE
    AGE_ RESULT = AGE(N-1) + 2
  END IF
END FUNCTION AGE
```

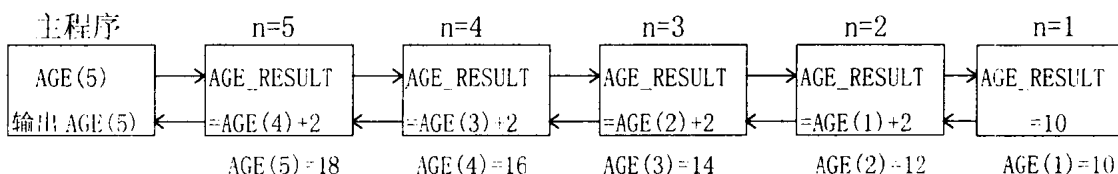


图 6-5 AGE 函数递归调用过程

函数递归调用过程如图 6-5 所示。

6.7.2 递归子例子程序

子例子程序也具有递归的性质,即可以在子例子程序体内又调用自身,它的入口语句前也要写上关键字 RECURSIVE。下面通过一个典型例子,介绍子例子程序的递归方法。

[例 6-11] Hanoi(汉诺)塔问题。有三根柱子 A、B、C,A 柱上有若干个盘子,盘子大小不等,大的在下,小的在上(如图 6-6)。要求把这些盘子从 A 柱子移到 C 柱子上,在移动过程中可以借助 B 柱子,但每次只允许移动一个盘子,且在移动过程中在三根柱子上都必须

保持大盘子在下,小盘子在上。要求程序输出移动的步骤。

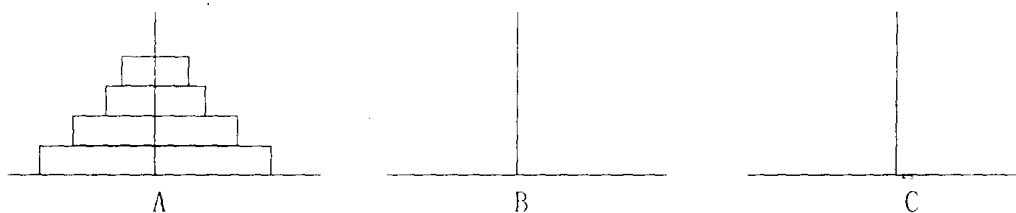


图 6-6 汉诺塔问题

算法分析:

将 n 个盘子从 A 柱移到 C 柱上可以分解为以下三个步骤:

1. 将 A 上 $n-1$ 个盘子借助 C 柱先移到 B 柱上;
2. 将 A 柱上剩下的最大一个移到 C 柱上;
3. 将 $n-1$ 个盘子从 B 柱借助于 A 柱移到 C 柱上。

例如,将 A 柱上三个盘子移到 C 柱上,可以分解为以下三步:

1. 将 A 柱上 2 个盘子移到 B 柱上(借助 C);
2. 将 A 柱上 1 个盘子移到 C 柱上;
3. 将 B 柱上 2 个盘子移到 C 柱上(借助 A)。

其中第 1 步又可用递归方法分解为:

- 1.1 将 A 上 1 个盘从 A 移到 C;
- 1.2 将 A 上 1 个盘从 A 移到 B;
- 1.3 将 C 上 1 个盘从 C 移到 B。

第 3 步可以分解为:

- 3.1 将 B 上 1 个盘子从 B 移到 A 上;
- 3.2 将 B 上 1 个盘子从 B 移到 C 上;
- 3.3 将 A 上 1 个盘子从 A 移到 C 上。

将以上各步综合起来,可得移动步骤为:

$A \rightarrow C, A \rightarrow B, C \rightarrow B, A \rightarrow C, B \rightarrow A, B \rightarrow C, A \rightarrow C$

由此可见上面的步骤可分成两类操作:

1. 将 $n-1$ 个盘子从一柱移到另一个柱上($n > 1$)(这是递归过程);
2. 将 1 个盘子从一个柱上移到另一柱上。

下面给出 FORTRAN 程序,用过程 HANOI 实现第一类操作,过程 MOVE 实现第二类操作:

```
SUBROUTINE MOVE(START,GOAL)
  INTEGER::START,GOAL
  PRINT *,START,'-->',GOAL
END SUBROUTINE MOVE
RECURSIVE SUBROUTINE HANOI(N,A,B,C)
  INTEGER::N,A,B,C
  IF(N==1)THEN
```

```

        CALL MOVE(A,C)
ELSE
    CALL HANOI(N-1,A,C,B)
    CALL MOVE(A,C)
    CALL HANOI(N-1,B,A,C)
END IF
END SUBROUTINE HANOI
PROGRAM MAIN_ HANOI
    IMPLICIT NONE
    INTERFACE
        RECURSIVE SUBROUTINE HANOI(N,A,B,C)
            INTEGER::N,A,B,C
        END SUBROUTINE HANOI
    END INTERFACE
    INTEGER::NUM_DISK
    READ *,NUM_DISK
    PRINT *, 'INPUT THE NUMBER OF DISK:',NUM_DISK
    PRINT *, 'THE STEP TO MOVEING ',NUM_DISK, ' DISK:'
    CALL HANOI(NUM_DISK,1,2,3)
END PROGRAM MAIN_ HANOI

```

程序输出结果如下(为节省篇幅,移盘步骤的格式未完全按机器输出格式列出):

```

INPUT THE NUMBER OF DISK: 4
THE STEP TO MOVEING 4 DISK:
1-->2 1-->3 2-->3 1-->2 3-->1 3-->2 1-->2 1-->3
2-->3 2-->1 3-->1 2-->3 1-->2 1-->3 2-->3

```

6.7.3 间接递归

递归是指一个过程中又调用了该过程本身。当过程 A 调用过程 B,过程 B 又调用过程 A 时,属于间接地调用自己,这称作间接递归。间接递归过程可以是函数,也可以是子例子程序,在间接递归函数和间接递归子例子程序语句前仍需要加关键字 RECURSIVE。

相对而言,递归的概念不太容易理解,间接递归比直接递归又难一些。因此,我们仅通过一个例子介绍一下间接递归的使用方法。

[例 6-12] 用间接递归函数验证例 5-11 介绍的“角谷猜想”。程序如下:

```

PROGRAM INDIRECT
    IMPLICIT NONE
    INTERFACE
        RECURSIVE FUNCTION DECREASE(A) RESULT(DE_RES)
            INTEGER,INTENT(IN)::A

```

```

        INTEGER::DE_RES
    END FUNCTION DECREASE
    RECURSIVE FUNCTION INCREASE(B) RESULT(IN_RES)
        INTEGER, INTENT(IN)::B
        INTEGER::IN_RES
    END FUNCTION INCREASE
END INTERFACE
INTEGER::N,X
READ *,N
X=DECREASE(N)
PRINT *, 'FINAL= ',N,', ',X
END PROGRAM INDIRECT
RECURSIVE FUNCTION DECREASE(A) RESULT(DE_RES)
    IMPLICIT NONE
    INTEGER, INTENT(IN)::A
    INTEGER::DE_RES, INCREASE
    DE_RES=A
    DO WHILE (MOD(DE_RES,2) /= 0)
        DE_RES=DE_RES/2
        PRINT *, 'DECREASE:',DE_RES
    IF(DE_RES/=1) DE_RES=INCREASE(DE_RES)
END FUNCTION DECREASE
RECURSIVE FUNCTION INCREASE(B) RESULT(IN_RES)
    IMPLICIT NONE
    INTEGER, INTENT(IN)::B
    INTEGER::IN_RES, DECREASE
    IN_RES=3*B+1
    PRINT *, 'INCREASE:',IN_RES
    IN_RES=DECREASE(IN_RES)
END FUNCTION INCREASE

```

6.8 其它部分

FORTRAN90 中的过程按其在程序中所处位置可分为外部过程与内部过程。另外，FORTRAN90 还提供了类属过程、重载运算符、自定义运算符、重载赋值号等功能。

6.8.1 外部过程与内部过程

如果一个过程未包含在主程序单元(或过程、模块)中,这个过程就属于外部过程。外部

过程独立成为一个程序单元,与主程序单元分别编译。相反,如果一个过程不是写在主程序(或过程、模块)外,而是写在主程序单元(或某一过程、某一模块)之内,作为主程序(或某一过程、某一模块)内包含的一个过程,则称该过程为内部过程。前面介绍的过程大多数都是外部过程。

1. 内部过程的位置

下边以内部过程 A 和 FA 为例,讲述它们在主程序中的具体位置:

```
PROGRAM MAIN
  (主程序的说明部分)
  (主程序的执行部分)
CONTAINS
  SUBROUTINE A
    (子程序 A 的说明部分)
    (子程序 A 的执行部分)
  END SUBROUTINE A
  FUNCTION FA() RESULT(FA_RESULT)
    (函数 FA 的说明部分)
    (函数 FA 的执行部分)
  END FUNCTION FA
END PROGRAM
```

由此可见,在程序体后再加一条 CONTAINS 语句,其后写出各内部子程序的全部语句(它们仍有各自的开始语句与结束语句),最后再写主程序结束语句。子例子程序 A 和函数子程序 FA 写在主程序的 CONTAINS 语句与主程序结束语句之间,作为主程序 MAIN 的内部过程。当然,如果某个子程序还调用了的其它的过程,也可以把这些被调过程写在该子程序中,成为该子程序的内部过程,写法与主程序的内部过程相似。另外,模块中也可以有内部过程(前面曾给出例子),其形式与主程序的内部过程相似。

2. 内部过程与外部过程的区别

内部过程有两个重要特征:

(1)内部过程通常没有说明语句。它们使用的变量等实体的说明应统一放在包含它的程序单元的说明部分中,因此它们的变量与主调程序同名变量的值是相同的。主调程序内可以直接使用内部子程序变量的值或给它赋值。

(2)内部过程一般没有虚元。主调程序调用时也不需要虚实结合。因为,如果内部过程的变量在主程序中被说明,它们就可以直接引用,不需要通过虚实结合来传递数据。调用内部过程时只要简单地写一个过程名。

在一些特殊场合,内部过程也可保留自己单独说明的少量虚元。如果变量是在内部子程序中单独说明的,它只是局部变量,其作用域只能局限于该内部子程序,主调程序的其它部分不能使用它们。下面通过一个例子熟悉一下内部过程的使用方法。

[例 6-13] 编一过程,求两个数之和。

方法 1: 用外部过程实现。

```
PROGRAM EXTERNAL_PROC
  IMPLICIT NONE
```

```

INTEGER::A,B,SUM
READ *,A,B
CALL ADD(A,B,SUM)
PRINT *, 'SUM=',SUM
END PROGRAM EXTERNAL PROC
SUBROUTINE ADD(A,B,SUM)
  IMPLICIT NONE
  INTEGER,INTENT(IN)::A,B
  INTEGER,INTENT(OUT)::SUM
  SUM=A+B
END SUBROUTINE ADD

```

虽然实元与虚元同名,但它们之间并不能直接传递数据,必须通过虚实结合实现。

方法 2: 用内部过程实现。

```

PROGRAM INTERNAL PROC
  IMPLICIT NONE
  INTEGER::A,B,SUM
  READ *,A,B
  CALL ADD
  PRINT *, 'SUM=',SUM
CONTAINS
  SUBROUTINE ADD
    SUM=A+B
  END SUBROUTINE ADD
END PROGRAM INTERNAL PROC

```

其子程序中并无变量说明,也没有虚元,主调程序中调用过程时只写过程名,而无实元表,因为变量 A、B、SUM 在主程序中说明后便作为全局变量,即在 PROGRAM 和 END PROGRAM 之间有效。故子程序 ADD 中可以直接引用这些变量,其结果自动带回主程序。

使用内部或外部过程时还应注意以下几点:

1° 包含内部过程的主程序(或模块、子程序)中,一定要有 CONTAINS 语句。如不写该语句,则编译程序无法正常编译。

2° 一个内部过程应在包含该过程的主程序或过程中说明,因此这个主程序或过程就可以调用这个内部过程。另外,在包含该过程的主程序或过程中说明的其它内部过程也可以调用该内部过程,但是其它外部过程不能调用该内部过程。

3° 主程序能调用外部过程,外部过程也可以调用外部过程。

4° 模块中包含的内部过程可供主程序和过程引用,这可以通过 USE 语句实现。

6.8.2 类属过程、重载运算符、自定义运算符、重载赋值号

1. 类属过程

类属过程是过程的一种。它允许不同类型的实元与同一个虚元结合,放宽了虚实结合

时类型必须一致的条件。

类属过程的编写方法是:先编写若干个功能相同的过程,它们分别有不同类型(例如整型、实型、复型等)的虚元。然后在主调程序中编写接口,为接口取一个统一的名,接口内分别列出虚元类型不同的过程说明部分的语句,这个统一的接口名就是类属过程名。例如将上面求两数之和的过程改写为类属过程后,其虚元便不只限于实型,只要引用类属过程名,就可任意地把整型、实型、复型实元与虚元结合。

[例 6-14] 编写交换两个存储单元数据的类属过程。

```
SUBROUTINE SWAP_REAL(A,B)
    IMPLICIT NONE
    REAL,INTENT(INOUT)::A,B
    REAL::TEMP
    TEMP=A; A=B; B=TEMP
END SUBROUTINE SWAP_REAL
SUBROUTINE SWAP_INTEGER(A,B)
    IMPLICIT NONE
    INTEGER,INTENT(INOUT)::A,B
    INTEGER::TEMP
    TEMP=A; A=B; B=TEMP
END SUBROUTINE SWAP_INTEGER
PROGRAM SWAP_DATA
    IMPLICIT NONE
    INTERFACE SWAP
        SUBROUTINE SWAP_REAL(A,B)
            REAL,INTENT(INOUT)::A,B
        END SUBROUTINE SWAP_REAL
        SUBROUTINE SWAP_INTEGER(A,B)
            INTEGER,INTENT(INOUT)::A,B
        END SUBROUTINE SWAP_INTEGER
    END INTERFACE
    REAL::X1,X2
    INTEGER::Y1,Y2
    READ *,X1,X2
    READ *,Y1,Y2
    CALL SWAP(X1,X2)
    PRINT *,X1,X2
    CALL SWAP(Y1,Y2)
    PRINT *,Y1,Y2
END PROGRAM SWAP_DATA
```

运行结果如下:

2.34, 5.73✓

4, 7✓

5.7300000 2.3399999

7 4

各过程的说明可以写在一个接口块内,也可写在几个接口块内,但每个接口块类属名都要相同。类属名可以任取,类属过程可以是函数过程也可以是子例子程序过程。

2. 重载运算符、自定义运算符

每一个标准 FORTRAN 运算符,如 +、-、*、/、**、<=、<、==、/=、>=、>等,可以通过恰当的过程扩充其功能,使其具有新的含义,这就是所谓的运算符重载。具有了新的含义的运算符仍符合 FORTRAN 规定的运算优先等级,同时也可以定义新的运算符。

[例 6-15] 假设定义一个字符串运算,让它删除两个字符串的结尾空格后,将这两个字符串连接起来,再假设用符号“+”来完成这个运算。那么,‘JOHN’+‘DOT’的结果应是‘JOHNDOT’。

下面先给出一个普通函数,命名为 CONCAT

```
FUNCTION CONCAT(STR1,STR2) RESULT(CONCAT_RESULT)
  IMPLICIT NONE
  CHARACTER(LEN=200)::CONCAT_RESULT
  CHARACTER(LEN=100),INTENT(IN)::STR1,STR2
  CONCAT_RESULT(1:LEN_TRIM(STR1))=STR1
  CONCAT_RESULT(LEN_TRIM(STR1)+1:LEN_TRIM(STR1)+&
    LEN_TRIM(STR2))=STR2
END FUNCTION CONCAT
```

其中 LEN_TRIM 为内在函数,其功能是取字符串的长度(不计尾部空格)。CONCAT_RESULT(1:LEN_TRIM(STR1))为字符串 CONCAT_RESULT 前半部分的子串被赋以字符串 STR1(去掉尾部空格)的值;同理下一句是 CONCAT_RESULT 后半部分子串被赋以字符串 STR2(去掉尾部空格)的值。故函数的值是 STR1(去掉尾部空格)和 STR2(去掉尾部空格)的并置字符串,下面把这个函数功能定义为重载的运算符“+”,则应在主调程序中编写如下接口块:

```
INTERFACE OPERATOR(+)
  FUNCTION CONCAT(STR1,STR2) RESULT(CONCAT_RESULT)
    CHARACTER(LEN=200)::CONCAT_RESULT
    CHARACTER(LEN=100),INTENT(IN)::STR1,STR2
  END FUNCTION CONCAT
END INTERFACE
```

有了这个接口程序后,“+”运算符就可以用于连接两个字符串,但它仍可以用作两个数值的加法运算。即遇到运算符“+”时,如果操作数为数值,则“+”仍为加法运算;如果不是数值,就不再代表加法,操作含义要到 FUNCTION CONCAT 中找。一般说来定义“+”超载后,如 CH1、CH2 为字符串,则允许写

CH1 + CH2

其值为前后两个字符串(去掉尾部空格)的并置。

一个用于超载二目运算符的函数必须有两个虚元,在使用这个运算符时不能与它原有的运算对象有相同的类型和种类。但有时它也可以带有一个虚元。例如减运算符“-”,它既可作二目运算,又可作一目运算。即使两种都超载,也不会混淆,因为超载函数的变元数不同。

当然也可以定义一个崭新的操作,自定义的运算符可由字母组成。两边用小数点作定界符,以便与一般变量和关键字区别开来。

[例 6-16] 定义一个新运算符,能将一个字符串中小写字母改成大写。

实现此功能的函数为:

```
FUNCTION UPPER_STR(STR) RESULT(UPPER_STRING)
    IMPLICIT NONE
    CHARACTER(LEN=*) ,INTENT(IN)::STR
    CHARACTER(LEN=255)::UPPER_STRING
    INTEGER::I
    CHARACTER::CH
    DO I=1,LEN_TRIM(STR)
        CH=STR(I:I)
        IF(CH>='a'.AND.CH<='z') THEN
            UPPER_STRING(I:I)=ACHAR(ICHAR(CH)-32)
        ELSE
            UPPER_STRING(I:I)=CH
        END IF
    END DO
END FUNCTION UPPER_STR
```

可以用以下主程序使用自定义运算符.UPPER.,但要书写相应的接口程序。

```
PROGRAM MAIN_UPPER
    IMPLICIT NONE
    INTERFACE OPERATOR(.UPPER.)
        FUNCTION UPPER_STR(STR) RESULT(UPPER_STRING)
            CHARACTER(LEN=*) ,INTENT(IN)::STR
            CHARACTER(LEN=255)::UPPER_STRING
        END FUNCTION UPPER_STR
    END INTERFACE
    CHARACTER(LEN=60)::CH1,CH2
    READ(*,'(A)')CH1
    CH2=.UPPER.CH1
    PRINT '(A,/A)',CH1,CH2
END PROGRAM MAIN_UPPER
```

通常程序员定义的单目运算符也都有一个相同的优先级,它比所固有的运算符的等级

要低。程序员定义的二目运算符都有一个相同的优先级,但它比所固有的二目运算符的等级要高。自定义运算符名称可以是任取的名字,但不能与固有的运算符同名,也不能与逻辑常数等同名。

3. 超载赋值

超载赋值是指把赋值的含义扩展到新的数据类型。按规定赋值号两边的类型必须相容,例如可以把数值型数据赋给数值型变量,把逻辑型数据赋给逻辑型变量等。用超载赋值时,可以进行类型不同的数据间赋值,可将一个类型的表达式赋给另一类型的变量。例如把一逻辑型的值赋给一个数值型变量,即把.TRUE.赋给一整型变量得值1,把.FALSE.赋给一整型变量得值0。

实现赋值号超载的方法是:先编一个不同类型变量值之间一一对应关系的子例子程序,而后在主调程序中编写一个接口,接口语句的形式为:

```
INTERFACE ASSIGNMENT(=)
```

[例6-17] 编写一个子例子程序,把逻辑表达式的值超载赋值给整型变量。

```
SUBROUTINE LOG_INT(I,L)
  INTEGER,INTENT(OUT)::I
  LOGICAL,INTENT(IN)::L
  IF(L)THEN
    I=1
  ELSE
    I=0
  END IF
END SUBROUTINE LOG_INT
```

主调程序中程序接口如下:

```
INTERFACE ASSIGNMENT(=)
  SUBROUTINE LOG_INT(I,L)
    INTEGER,INTENT(OUT)::I
    LOGICAL,INTENT(IN)::L
  END SUBROUTINE LOG_INT
END INTERFACE
```

在主调程序中,可以将.TRUE.和.FALSE.赋值给整型变量,编译程序根据赋值运算符两边的类型决定进行普通赋值操作还是超载赋值操作。

进行超载赋值运算时,必须编写成子例子程序形式;要实现超载运算符功能时,必须编成函数子程序形式。否则就会混淆概念,无法达到目的。

6.9 程序举例

[例6-18] 用函数子程序求自然数A和B的最大公约数。

例5-15已经给出了用辗转相除法求两个自然数的最大公约数的程序。本例中用函数子程序法实现,自然数A、B为函数变元,函数的返回结果是A和B的最大公约数。程序如

下:

```
FUNCTION GCD(X,Y) RESULT(GCD_RESULT)
  IMPLICIT NONE
  INTEGER,INTENT(IN)::X,Y
  INTEGER::GCD_RESULT,A,B,T,R
  A=X; B=Y
  IF(A<B)THEN
    T=A; A=B; B=T
  END IF
  R=A
  DO WHILE (R/=0)
    R=MOD(A,B)
    A=B; B=R
  END DO
  GCD_RESULT=A
END FUNCTION GCD
PROGRAM GCD_FUN
  IMPLICIT NONE
  INTEGER::X,Y,GCD
  READ *,X,Y
  PRINT *,GCD(X,Y)
END PROGRAM GCD_FUN
```

[例 6-19] 编写求正整数 M 是否为素数的子例子程序,通过逻辑型变元带回 M 是否为素数的信息。程序如下:

```
SUBROUTINE PRIME(M,PRIME_LABEL)
  IMPLICIT NONE
  INTEGER,INTENT(IN)::M
  LOGICAL,INTENT(OUT)::PRIME_LABEL
  INTEGER::I=2
  PRIME_LABEL=.TRUE.
  DO WHILE(I<=M/2.AND.PRIME_LABEL)
    IF(MOD(M,I)==0)PRIME_LABEL=.FALSE.
    I=I+1
  END DO
END SUBROUTINE PRIME
PROGRAM MAIN_PRIME
  IMPLICIT NONE
  INTEGER::PM
```

```

LOGICAL::PRIME_LOG
READ *,PM
CALL PRIME(PM,PRIME_LOG)
PRINT *,PRIME_LOG
END PROGRAM MAIN_PRIME

```

主调程序调用子程序 PRIME 以后,若实元 PRIME_LOG 为 .TRUE. 表示 PM 是素数,为 .FALSE. 表示 PM 不是素数。该子例子程序也可以由函数实现。

[例 6-20] 斐波纳契数列为 1,1,2,3,5,8,13,21,34,⋯,它可由下列方法定义:

$$\begin{aligned}
 f(1) &= 1 \\
 f(2) &= 1 \\
 f(n) &= f(n-1) + f(n-2) \quad (n > 2)
 \end{aligned}$$

编程求出该数列的前 n 项值。

方法 1: 函数中用循环实现:

```

FUNCTION FIB(N) RESULT(FIB_RESULT)
  IMPLICIT NONE
  INTEGER,INTENT(IN)::N
  INTEGER::FIB_RESULT
  INTEGER::F1,F2,F3,I
  IF(N<=2)THEN
    FIB_RESULT=1
  ELSE
    F1=1; F2=1
    DO I=3,N
      F3=F1+F2
      F1=F2; F2=F3
    END DO
    FIB_RESULT=F2
  END IF
END FUNCTION FIB
PROGRAM FIB_FUN
  IMPLICIT NONE
  INTERFACE
    FUNCTION FIB(N) RESULT(FIB_RESULT)
      INTEGER,INTENT(IN)::N
      INTEGER::FIB_RESULT
    END FUNCTION FIB
  END INTERFACE
  INTEGER::N,I,S

```

```

READ *,N
DO I=1,N
  S=FIB(I)
  PRINT *,S
END DO
END PROGRAM FIB_FUN

```

方法 2: 用递归函数实现:

```

RECURSIVE FUNCTION FIB(N) RESULT(FIB_RESULT)
  IMPLICIT NONE
  INTEGER, INTENT(IN)::N
  INTEGER::FIB_RESULT
  IF(N<=2) THEN
    FIB_RESULT=1
  ELSE
    FIB_RESULT=FIB(N-1)+FIB(N-2)
  END IF
END FUNCTION FIB
PROGRAM FIB_RECU
  IMPLICIT NONE
  INTERFACE
    RECURSIVE FUNCTION FIB(N) RESULT(FIB_RESULT)
      INTEGER, INTENT(IN)::N
      INTEGER::FIB_RESULT
    END FUNCTION FIB
  END INTERFACE
  INTEGER::N,I,S
  READ *,N
  DO I=1,N
    S=FIB(I)
    PRINT *,S
  END DO
END PROGRAM FIB_RECU

```

由于方法 2 每求数列中一个数都要多次递归调用函数,因此运行效率较方法 1 差。但方法 1 也不是最佳方案,因为它每求一个数时并没有充分利用该数是由数列中前两个数之和这一特点,而是每次都从头加起。此处举例仅说明函数的用法,未把效率放在重要位置。

[例 6-21] 编写求 $\int_a^b f(x)dx$ 的近似值的函数,并利用该函数求以下两个定积分值。

$$y_1 = \int_0^4 (x^2 + 3x + 2)dx$$

$$y_2 = \int_0^{\frac{\pi}{2}} x \cdot \sin x dx$$

例 5-16 曾给出了用梯形法求定积分的方法。实际上, n 个梯形的面积之和也可以用下式表示(h 是每个区间的宽度):

$$T_n = h \left[\frac{f(a)}{2} + f(a+h) + f(a+2h) + \cdots + f(b-h) + \frac{f(b)}{2} \right]$$

在求任意函数 F 的定积分值的函数子程序 TRP 中, a 、 b 和 n 的值是输入数据, F 是被积函数名, 函数 TRP 中需要有描述 F 的接口块。函数子程序 TRP 的流程如图 6-7 所示。程序如下:

```
FUNCTION TRP(F,A,B,N) RESULT(TRP_RESULT)
```

```
  IMPLICIT NONE
```

```
  INTERFACE
```

```
    FUNCTION F(X) RESULT(F_RESULT)
```

```
      REAL, INTENT(IN)::X
```

```
      REAL::F_RESULT
```

```
    END FUNCTION F
```

```
  END INTERFACE
```

```
  REAL::TRP_RESULT,&
```

```
    A,B,H,SUM
```

```
  INTEGER::N,I
```

```
  H=(B-A)/N
```

```
  SUM=0.5*(F(A)+F(B))
```

```
  DO I=1,N-1
```

```
    SUM=SUM+F(A+I*H)
```

```
  END DO
```

```
  TRP_RESULT=H*SUM
```

```
END FUNCTION TRP
```

```
FUNCTION F1(X) RESULT&
```

```
  (F1_RESULT)
```

```
  IMPLICIT NONE
```

```
  REAL, INTENT(IN)::X
```

```
  REAL::F1_RESULT
```

```
  F1_RESULT=X*SIN(X)
```

```
END FUNCTION F1
```

```
FUNCTION F2(X) RESULT(F2_RESULT)
```

```
  IMPLICIT NONE
```

```
  REAL, INTENT(IN)::X
```

```
  REAL::F2_RESULT
```

```
  F2_RESULT=X**2+3*X+2
```

```
END FUNCTION F2
```

```
PROGRAM MAIN_TRP
```

函数TRP(F, A, B, N)的说明部分 其中含接口块, 用于说明函数F(X)
H=(A+B)/N
SUM=0.5*(F(A)+F(B))
DO I=1,N-1
SUM=SUM+F(A+I*H)
TRP_RESULT=H*SUM

图 6-7 函数积分流程图

```

IMPLICIT NONE
INTERFACE
  FUNCTION F1(X) RESULT(F1_RESULT)
    REAL, INTENT(IN)::X
    REAL::F1_RESULT
  END FUNCTION F1
  FUNCTION F2(X) RESULT(F2_RESULT)
    REAL::F2_RESULT
  END FUNCTION F2
END INTERFACE
REAL::A1,A2,B1,B2
INTEGER::N1,N2
REAL::TRP
READ *,A1,B1,N1
PRINT *,TRP(F1,A1,B1,N1)
READ *,A2,B2,N2
PRINT *,TRP(F2,A2,B2,N2)
END PROGRAM MAIN_TRP

```

习 题 6

6-1 编写以 A 和 B 作为变元的函数子程序,求 $\cos A \sin B + \sin A \cos B$ 的值。再编写主程序,依次读入 5 对 A、B 的值,并求出相应的函数值。这 5 对数据是:

① $83^\circ, 73^\circ$ ② $15^\circ, 35^\circ$ ③ $44^\circ, 66^\circ$ ④ $3^\circ, 70^\circ$ ⑤ $11^\circ, 27^\circ$

6-2 已知函数

$$f(x) = \begin{cases} 1 + \sqrt{1+x^2} & (x < 0) \\ 0 & (x = 0) \\ 1 - \sqrt{1+x^2} & (x > 0) \end{cases}$$

编写函数子程序及主程序,求 x 分别取值 $-0.5, 0$ 和 12.5 时的函数值。

6-3 编写一个判断素数的函数子程序,在主程序中输入 2 个整数 $M1$ 和 $M2$,并输出 $[M1, M2]$ 上的全部素数。

6-4 编程序求

$$S = a + aa + aaa + \cdots + \overbrace{aa \cdots a}^n$$

的值,其中 a 和 n 都是数字。例如: $3 + 33 + 333 + 3333$ (此时 $a = 3, n = 4$), a, n 由主程序输入。

6-5 编写子程序求二次方程的两个实根,其求根公式为:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

6-6 编程序求的二项式系数:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}, k = 0, 1, \dots, n$$

当求 n 次的二项式系数时, k 从 0 到 n 变化, 主程序通过调用求阶乘的函数子程序求出各项系数。

6-7 写一个类属函数求 $A \times B \times C \times D$ 的积, 允许用整型实元和实型实元与虚元结合, 其函数值与实元具有相同的类型。

6-8 编写一个函数子程序, 求 4 个数之和, 但有时也用它求 3 个数之和(仅 3 个实元)。

6-9 已知双曲正弦函数 $\sinh(x) = \frac{e^x - e^{-x}}{2}$, 其中 e^x 可用以下公式近似表示:

$$e^x \approx 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^m}{m!} = \sum_{n=0}^m \frac{x^n}{n!}$$

先编写求 e^x 值的子程序(x 和 m 由键盘输入); 再编写一个子程序, 调用求 e^x 值的子程序求 $\sinh(x)$ 的近似值。

6-10 编写一个子程序 SORT_4(A,B,C,D), 使其 4 个实元按由小到大的顺序排序。

6-11 编写用牛顿迭代法求方程

$$ax^3 + bx^2 + cx + d = 0$$

在 x_0 附近的一个根的函数。其中 a, b, c, d 及 x_0 的值均在主程序中输入, 输出也在主程序中进行。

6-12 编一主程序, 能调用函数过程, 该过程能求 $(F(x))^3$ 的值, 但有时 $F(x) = x^2 + 1$, 有时 $F(x) = x^2 + 2x + 2$, 试用虚过程实现。

6-13 把一个整数表达式的值超载赋给一个逻辑变量。零为假值, 其它整数为真。

6-14 编写递归函数 $f(n)$ ($n \geq 0$), 其定义如下:

$$f(0) = 0$$

$$f(1) = f(2) = 1$$

$$f(n) = 2 \times f(n-1) + f(n-2) + f(n-3) \quad (n \geq 3)$$

并用主程序调用此函数, 分别求 $n = 10, n = 15$ 时的函数值。

6-15 编写递归函数 GCD(A,B), 其递推关系如下:

$$\gcd(a, b) = b \quad (a \bmod b = 0 \text{ 时})$$

$$\gcd(a, b) = \gcd(b, a \bmod b) \quad (a \bmod b \neq 0 \text{ 时})$$

6-16 用递归方法求 n 阶勒让德多项式的值, 递推公式为:

$$p_n(x) = \begin{cases} 1 & (n = 0) \\ x & (n = 1) \\ ((2n-1)xp_{n-1}(x) - (n-1)p_{n-2}(x))/n & (n > 1) \end{cases}$$

6-17 甲、乙、丙同时开始放鞭炮, 甲每隔 t_1 秒放一次, 乙每隔 t_2 秒放一次, 丙每隔 t_3 秒放一次, 每个人各放 N 个鞭炮, 编写一个子程序求出总共能听到多少次鞭炮声。

6-18 用间接递归法求

$$n \times ((n-1) + (n-2) \times ((n-3) + (n-4) \times ((n-5) + (n-6) \times (\dots))))$$

的值。

第 7 章 数 组

到目前为止,程序中使用的变量都是基本类型的数据(整数值、实数值、逻辑值等),这些变量又称为简单变量。在机器内部,各个简单变量都占据独立的存储单元,彼此之间没有直接联系。除简单变量之外,FORTRAN90 还提供了一种构造类型的数据—数组。

数组是标量数据为元素的集合,其每一个元素都具有同一种数据类型、种别参数和属性。数组中的元素在内存中占据连续的存储单元。本章介绍有关数组的概念及使用方法。

7.1 一维数组

根据数组元素使用下标的个数可分为一维数组和多维数组。一维和二维数组是最常用的数组,一维数组比较简单。

7.1.1 一维数组的概念

FORTRAN 程序中使用数组时,必须先对数组进行说明。说明数组时一般用类型说明语句实现,但需要增加一个数组属性说明标识符: DIMENSION(数组形状说明)。说明一维数组的一般形式为:

类型说明[[[长度说明],[种别说明]]], DIMENSION($d_1:d_2$),其它属性说明::数组名表

例如数组说明

```
INTEGER, DIMENSION(1:30)::A
```

其中(1:30)描述了数组形状,说明该数组是一维数组,维的下界是 1,上界是 30。这表明数组 A 有 30 个元素,即 A(1)、A(2)、…、A(30)。

1. 维界、界偶与维长

上面说明中括号内的 1:30 称为维的界偶。维表明一种排列方式,维界表示在该种方式排列中元素所取的最低序号和最高序号,分别称为维的下界与上界。在数组形状描述时,每一维数都要用一对维界表示,称为界偶,其一般形式是:

$$d_1:d_2$$

这里的 d_1 和 d_2 必须是整型表达式,表达式的值可以是正值,也可以是负值,其中 d_1 代表下界, d_2 代表上界。例如可以有

```
INTEGER, DIMENSION(5:7)::A
```

```
INTEGER, DIMENSION(10+5:50+N)::B
```

这里 N 为一常数名。数组 A 的维下界为 5,上界为 7,数组 A 的元素依次为 A(5)、A(6)和 A(7);若 N 的值为 10,则数组 B 的维下界为 15,上界为 60,各元素的序号自 B(15)、B(16)起直到 B(60)止,共 46 个元素。

维的上、下界值的符号虽然可正可负,但必须满足 $d_1 \leq d_2$ 。例如上述 46 个元素的维界也可定为 $d_1 = -46, d_2 = -1$,这时的说明语句应为

INTEGER, DIMENSION(-46:-1)::B

当 $d_1 < 0$ 且 $d_2 > 0$ 时, 要注意元素序号递增过程中应包括序号 0。例如数组 A 的属性说明改为 DIMENSION(-1:3) 时, 表示 A 共有 5 个元素, 依次为 A(-1)、A(0)、A(1)、A(2) 和 A(3)。

有时也允许 $d_1 = d_2$, 此时数组内只有一个元素。

若 d_1 取 1 时, 可以把界偶 $d_1:d_2$ 简化为只写一个 d_2 , 因此 DIMENSION(1:100) 也可以写成 DIMENSION(100)。

维界一般只能取整常数表达式(即表达式中只有常量), 不得出现整型变量。只有在特殊场合下, 即数组出现在子程序中, 且被作为虚元时, 维界中才允许出现整型变量名。例如, 属性说明可以有:

DIMENSION(N:3), DIMENSION(L:K-2)

这种把整型变量名写入维界表达式的数组称为可调数组。

一维数组中的维长就是数组元素的个数, 它的值等于:

$$d_2 - d_1 + 1$$

例如属性说明中维界偶为 1:100, 其维长为 $100 - 1 + 1 = 100$; 维界偶为 -100:-1 时, 其维长为 $(-1) - (-100) + 1 = 100$ 。

2. 数组元素与下标

一个数组可以存放多个同类型的数据, 可以通过下标访问这些数据。数组元素在程序中可以赋值和引用, 下标则指明了数组元素在数组中的位置。

引用一维数组中的元素的一般形式是:

数组名(下标表达式)

假设数组 A 的维下界为 1, 则 A(5) 表示数组 A 的第 5 个元素。

下标表达式应是整型表达式, 允许出现 +、-、*、/ 等算术运算符。操作数可以是常数、整型变量, 也可以是简单的内在函数或数组元素, 但表达式的值一定要是整数。程序执行时, 系统先计算各下标表达式的值, 而后通过该下标值找出相应位置的元素。下列各元素写法都是合法的:

A(3*2), A(ABS(-3)), A(5+I), A(A(I))

其中 I 为整型, A(I) 的值也是整型且在 A 的维界范围之内。对任意下标表达式而言, 其取值必须在相应的维界偶范围内, 即:

$$d_1 \leq \text{下标表达式的值} \leq d_2$$

7.1.2 一维数组应用

[例 7-1] 输入 10 个整数, 并按输入时的逆序输出, 每行输出 5 个数。

程序如下:

```
PROGRAM NUMBER
  IMPLICIT NONE
  INTEGER, DIMENSION(1:10)::A
  INTEGER::I
```

```

DO I=1,10
  READ *,A(I)
END DO
PRINT '(5I4)',(A(11-I),I=1,10)
END PROGRAM NUMBER

```

程序中说明 A 为 10 个元素的整型数组。在 DO 循环中实现从终端上读入 10 个整数,每次读入一个整数。PRINT 语句中的隐含 DO 循环使输出按题目中要求的进行,即按输入时的逆序输出,且每行输出 5 个数。

[例 7-2] 把 10 个整数中最大的数找出来,并指出它在队列中的位置。

程序中先输出 10 个数到数组 A 中。用变量 MAX 来记录数组中最大的那个数所在的位置,在进行比较之前,先让 MAX=1,表示开始时假设 A(1)为最大。流程图见图 7-1。程序如下:

```

PROGRAM MAXNUM
  IMPLICIT NONE
  INTEGER,DIMENSION(1:10)::A
  INTEGER::I,MAX
  DO I=1,10
    READ *,A(I)
  END DO
  MAX=1
  DO I=2,10
    IF(A(I)>A(MAX)) MAX=I
  END DO
  PRINT *, 'MAX NUMBER IS:',A(MAX)
  PRINT *, 'THE POSSITION IS:',MAX
END PROGRAM MAXNUM

```

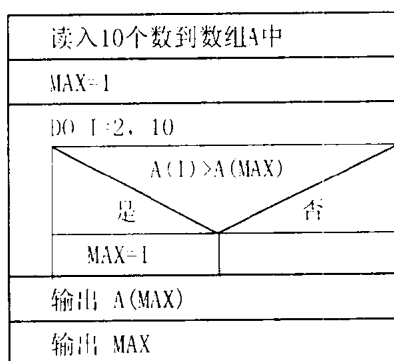


图 7-1 例 7-2 程序的 N-S 图

[例 7-3] 编写程序,从终端输入 30 名学生的成绩,把高于平均分的学生的学号和成绩输出来。

用数组 NUM 来存放学生学号,数组 SC 来存放对应学生的成绩。变量 AVER 存放学生的平均成绩。程序如下:

```

PROGRAM STUSC
  IMPLICIT NONE
  INTEGER,DIMENSION(1:30)::NUM
  INTEGER::I
  REAL,DIMENSION(1:30)::SC
  REAL::SUM,AVER
  SUM=0
  DO I=1,30

```

```

      READ *, NUM(I), SC(I)
      SUM = SUM + SC(I)
    END DO
    AVER = SUM/30.0
    PRINT *, 'AVERAGE = ', AVER
    PRINT *, 'LIST OF SCORES GREATER THAN AVERAGE:'
    DO I = 1, 30
      IF (SC(I) > AVER) PRINT *, NUM(I), SC(I)
    END DO
  END PROGRAM STUSC

```

7.1.3 数组片段

FORTRAN90 规定,在程序执行部分中,引用一个数组或其元素有以下三种形式:

数组名	代表数组中所有元素
数组元素	代表数组中一个元素
数组片段	代表数组中若干个元素,它们可以相连或分离。

其中数组片段又主要有两种表示方法。

1. 连续片段法

数组的连续片段表示一组相连的元素,下标偶对的前一个整数值表示下标的起始位置,冒号后的一个整数值表示下标的终止位置。其一般形式为:

数组名(整型表达式:整型表达式)

例如连续数组片段为

A(3:6)

时,代表数组 A 中 4 个相连的元素: A(3)、A(4)、A(5)和 A(6),它可以像简单变量一样写在赋值语句、输入输出语句等语句中,但它所代表的是数组元素的集合。例如赋值语句

A(3:6) = 0.0

表示把 A(3)至 A(6)这 4 个元素都置 0.0。输出语句

PRINT *, A(3:6)

则输出 A(3)至 A(6)之间的 4 个元素值。

2. 下标三元组法

该方法可以把不连续的元素组成数组片段。它用三个表达式来描述,其一般形式为:

数组名([表达式 1]:[表达式 2][:表达式 3])

其中表达式 1 为被选取元素的起始下标值,表达式 2 的值为被选取元素的终止下标,表达式 3 表示间隔(即起始值到终止值间每次增加的步长)。这三个表达式的值均可以省写。当省略表达式 1 时,表示把维的下界作起始下标;省略表达式 2 时表示以维的上界作终止下标;省略表达式 3 时,表示间隔为 1,这与连续数组片段表示形式的功能等价。下标三元组法中的 [表达式 1]:[表达式 2][:表达式 3]称作三元组表达式。

假设有以下说明:

INTEGER, DIMENSION(6:50)::A

则在程序中可以有以下数组片段:

A(10:30:1) 代表 A(10)至 A(30)中连续的 21 个元素

A(:20:5) 代表 A(6)、A(11)和 A(16)

A(8::10) 代表 A(8)、A(18)、A(28)、A(38)和 A(48)

从程序易读性考虑,还是不省略下标三元组的各个表达式为好。

7.1.4 数组构成器及用途

数组构成器是左右两端用“(/ ”和“ /) ”定界的同类型数据的集合,可以用它给数组赋初值和一般性赋值,它可代替用 DO 循环给每个数组元素依次赋值的繁琐形式。

例如下面就是一个数组构成器的书写形式:

(/ 1,2,3,4 /)

该数组构成器代表 4 个整数值,可以把它赋给具有 4 个元素的一维数组或数组片段。假设数组 A 为实型、维长为 4 且下标从 1 开始,则赋值语句

A = (/ 1.0,2.0,3.0,4.0 /)

是合法的。执行该语句后, A(1)=1.0, A(2)=2.0, A(3)=3.0, A(4)=4.0。

除了由常数直接形成数组构成器之外,还有以下构成数组构成器的方法:

1. 隐含 DO 循环法。其一般形式为:

(/ 隐含 DO 表 /)

2. 数组元素法。其一般形式为:

(/ 数组元素,数组元素,...,数组元素 /)

3. 数组片段法。其一般形式为:

(/ 数组片段 /)

以上数组构成器,可以使用到赋值语句中,给数组或其元素赋值。例如:

A = (/ (SQRT(REAL(I))), I = 1,4 /)

使用的是隐含 DO 循环数组构成器对数组 A 赋值。其中 (SQRT(REAL(I))), I = 1,4 是隐含 DO 表,赋值号右端整个形式就是数组构成器。执行该赋值语句后, $A(1) = \sqrt{1.0}$, $A(2) = \sqrt{2.0}$, $A(3) = \sqrt{3.0}$, $A(4) = \sqrt{4.0}$ 。SQRT 函数的自变量必须是实型数,但此处 I 是整型,所以取自变量为 REAL(I)。又如:

A = (/ V(3),V(6),V(7),V(9) /)

使用的是数组元素法数组构成器对数组 A 赋值。假设已对数组 V 进行了说明且 V 已有定义,其中 V(3)=3.3, V(6)=6.6, V(7)=7.7, V(9)=9.9,那么上述赋值语句执行之后, A(1)=3.3, A(2)=6.6, A(3)=7.7, A(4)=9.9。再如语句:

B = (/ V(3:9:3) /)

使用的是数组片段法数组构成器对数组 B 赋值。假设 V 的意义同上,数组 B 有 3 个元素,执行上赋值语句后, B(1)=3.3, B(2)=6.6, B(3)=9.9。

数组构成器除了通过赋值语句给数组赋值外,还可以在数组说明时直接给数组赋初值。例如:

INTEGER, DIMENSION(2:4)::M = (/ 1,2,3 /)

说明了一维数组 M 是整型,且给它的各元素赋了初值: M(2)=1, M(3)=2, M(4)=3。

7.1.5 数组形式

FORTRAN90 中的数组有 4 种形式,它们各自说明的方式及使用的场合、方法都有其自己的特点。

1. 常数组

所谓常数组是指说明时维界表达式中只出现整型常数或整型常数名,未出现整型变量与其它操作数的数组。例如:

```
REAL,DIMENSION(3:5+7)::A
INTEGER,DIMENSION(0:8)::B
```

这里 A 的上下界分别为 3 和 12;B 的上下界分别为 0 和 8,全都具有整型常数值,因此数组 A 和 B 都属于常数组。

过去所涉及的数组均都是常数组。它既可以用于主程序中,也可用于子程序中;既可以作为实数组,也可以作为虚数组。其使用范围较广,也比较容易理解和使用。

2. 可调数组

说明可调数组时,可以在维界表达式中出现整型变量。例如:

```
REAL,DIMENSION(N:M)::C
```

这里的 N 和 M 为整型变量名,如此说明的数组 C 就是可调数组,N、M 称为可调维。

可调数组只可以在子程序中作虚数组用,虚数组的可调维中需要的变量也必须列入子程序的虚元表中。例如,在子例子程序中可以有:

```
SUBROUTINE SUB(C,M,N)
REAL,DIMENSION(N:M)::C
```

其中第一句是子例子程序的开始语句,子程序名为 SUB,后面括号内是虚元素,第二句说明一个可调数组 C,它必须作为虚数组,故数组名 C 与可调维 N、M 都被列入虚元表中。

可调维数组的可调维 N、M 在定义子程序时没有确定的值,须由调用这个子程序的主调程序通过虚实结合赋给确定的值。例如在主调程序中使用语句

```
CALL SUB(A,10,20)
```

那么虚数组 C 就与实数组 A 虚实结合,且其形状为(10:20)。

3. 假定形状数组

说明假定形状数组时,不出现上下维界,只用一个冒号“:”表示。例如:

```
REAL,DIMENSION(:)::D1
INTEGER,DIMENSION(:,)::D2
```

则 D1 为一维实型假定形状数组,D2 为二维整型假定形状数组。

与可调数组一样,假定形状数组也只能在子程序中作虚数组用。当主调程序调用具有假定形状虚数组的子程序进行虚实结合时,取实数组维的上下界作为假定形状数组维的上下界。例如,如果主调程序中有一个形状为(1:5)的实数组 A 与被调子程序的假定形状数组 D1 相结合,此时 D1 的实际维界也是(1:5)。

说明假定形状数组的另一种方法是只写出维的下(或上)界,不写上(或下)界。例如:

```
REAL,DIMENSION(3:)::D3
```

则 D3 也是一维的假定形状数组,只是其下界是已经确定的,而上界由调用它进行虚实结合

时的实数组确定,原则是所取的上界使维长与实数组的维长相等。

4. 动态数组

动态数组又称可分配数组,它的维界可在程序执行过程中按需要变化,即数组占据的存储空间可以根据需要在程序执行中实施分配。这就需要有动态地分配存储空间和释放存储空间的机制。这种动态数组的说明及其使用方法由以下几步实现:

(1)说明一假定形状数组,同时还必须说明 ALLOCATABLE 属性,表示该数组是可分配存储空间的动态数组。例如:

```
REAL,DIMENSION(:),ALLOCATABLE::AL1,AL2
```

说明 AL1、AL2 都是一维实型动态数组。

(2)用 ALLOCATE 语句分配内存

经过以上说明的动态数组 AL1 和 AL2 并未占据内存空间,需要通过 ALLOCATE 语句给它们分配内存。假如整型变量 M 和 N 均有值的话,分配语句

```
ALLOCATE(AL1(M),AL2(6:N))
```

表示给数组 AL1 分配 M 个单元,该数组的形状是(1:M);给数组 AL2 分配 $N - 6 + 1$ 个单元,该数组的形状是(6:N)。

(3)用 DEALLOCATE 语句释放内存

如果某动态数组的任务已经完成,可以通过释放语句将该数组所占据的存储空间释放(还给系统)。例如释放语句

```
DEALLOCATE(AL1,AL2)
```

表示把动态数组 AL1 和 AL2 所占据的内存全部还给系统,即数组 AL1 和 AL2 的形状不再存在,不能再使用数组 AL1、AL2(或它们的元素、片段)。程序中若想使用已经释放的动态数组,必须再次通过 ALLOCATE 语句重新分配内存之后才能进行。但它已经重新分配了内存空间,与原来的那个动态数组已没有了联系,数组释放内存之前存储的数据已不存在。

释放不再使用的动态数组的内存空间是良好的程序设计习惯,也有利于提高内存的利用率。

分配语句和释放语句的一般形式分别是:

```
ALLOCATE(数组名 1(形状描述),数组名 2(形状描述),...,[STAT=变量名])
```

```
DEALLOCATE(动态数组名 1,动态数组名 2,...,[STAT=变量名])
```

ALLOCATE 语句中的“STAT=变量名”是可选的说明符,STAT 是关键字,变量名必须是整型变量。如果在 ALLOCATE 语句中选择了该项(假设为“STAT=ST”),执行 ALLOCATE 语句后如果分配成功,系统给 ST 置 0;如果出错(例如对非动态数组执行 ALLOCATE,内存空间不够用等等)则给 ST 置某一正整数,程序仍将继续执行。可以在 ALLOCATE 语句之后书写一个 IF 结构,按 ST 等于 0 还是大于 0 采取不同的处理方式。如果 ALLOCATE 语句中无 STAT 项,分配又出错时,程序会自动停止执行。

DEALLOCATE 语句的 STAT 选项的功能与 ALLOCATE 语句类似。

7.1.6 一维数组的输入与输出

一维数组的输入与输出同简单变量一样,也使用读、写语句,过去介绍的读、写语句都可以用来输入输出数组及其成分。在输入输出表中可出现数组有关成分:数组名、数组元素或

数组片段。输入输出时按给出的数组下标的顺序依次进行。

可以从以下几个方面了解对数组的输入与输出：

1. 输入输出表中使用数组名时,表示输入输出的是数组的全部元素。假设有数组说明及输入语句:

```
INTEGER,DIMENSION(2:5)::A
REAL,DIMENSION(1:6)::B
READ *,A,B
```

该输入语句相当于

```
READ *,A(2),A(3),A(4),A(5),B(1),B(2),B(3),B(4),B(5),B(6)
```

2. 输入输出表中使用数组片段时,表示输入输出的是数组片段中所选中的各个元素。如果有语句

```
PRINT *,A(2:3),B(2:6:2)
```

相当于语句

```
PRINT *,A(2),A(3),B(2),B(4),B(6)
```

3. 允许输入输出表中数组成分与非数组成分的实体混合出现。例如

```
READ *,A,N,B(1:4)
```

是合法的读语句。

4. 利用隐含 DO 循环输入输出数组。例如语句

```
PRINT *,(A(I),I=1,3),(B(J),J=1,4,3)
```

相当于语句

```
PRINT *,A(1),A(2),A(3),B(1),B(4)
```

[例 7-4] 已知有 L 个学生,读入他们的学号及其考试成绩,然后按成绩排定名次,并输出排序以后的学号与成绩。

设学生学号放入数组 NUM 中,成绩存入数组 SCORE 中,它们都是整型数组。假设有 4 名学生,有 L=4。为了用 L 说明数组的大小,可以将先 L 说明为整型常数名(值为 4)。程序中先读入学号数组与成绩数组,同一学生的学号及其成绩一起输入。在排名次需要交换成绩时,其相应的学号必须同时交换,使同一个人的学号与其成绩始终保持相同的下标。另外,两个变量中的数据相交换时,需要引入第三个临时变量(程序中为 T),用作交换时的中间工作单元。程序名取 SORT_1。程序如下:

```
PROGRAM SORT_1
  IMPLICIT NONE
  INTEGER,PARAMETER::L=4
  INTEGER,DIMENSION(1:L)::NUM,SCORE
  INTEGER::I,J,T
  READ *,(NUM(I),SCORE(I),I=1,L)
  DO I=1,L-1
    DO J=I+1,L
      IF(SCORE(I)<SCORE(J))THEN
        T=SCORE(I); SCORE(I)=SCORE(J); SCORE(J)=T
        T=NUM(I); NUM(I)=NUM(J); NUM(J)=T
      
```

```

        END IF
    END DO
END DO
PRINT '("NO.",I2,":",I5,5X,"SCORE:",I3)',(I,NUM(I),&
    SCORE(I),I=1,L)
END PROGRAM SORT_1

```

该程序运行结果如下:

```

97031,78✓
97032,97✓
97033,64✓
97034,88✓
NO. 1:97032    SCORE: 97
NO. 2:97034    SCORE: 88
NO. 3:97031    SCORE: 78
NO. 4:97033    SCORE: 64

```

输出中标出了每人的名次、学号与成绩。

由于 READ 语句中的变量采用隐含 DO 循环给出,它实际相当于语句

```

READ *,NUM(1),SCORE(1),NUM(2),SCORE(2),NUM(3),&
    SCORE(3),NUM(4),SCORE(4)

```

根据表控格式的规定,输入的 8 个数据可以打在一行上,也可以按本例的方法敲在多行上。

输出也采用了隐含 DO 循环方式,但 I 每取一个值时,要分别输出一组数据:名次、学号和成绩。由于 I 的取值为 1、2、3、4,故需输出四组数据。PRINT 语句中使用的是格式输出,每组数据占一行。

本例中有 L 个学生,实际上 L 应是可变化的量。由于一般情况下不能用变量作为维界来说明数组,我们采用了用常量名作为数组维界的方法。但每当 L 的值发生变化时,必须修改程序中对 L 的说明语句,需要重新编译、连接才能再次运行。这显然不符合题目的要求。因此需要修改上述程序。

为了让数组的大小与 L 相同,可以将 NUM 和 SCORE 说明为动态数组,每次先读入整数 L,用 L 作为动态数组的边界来给数组 NUM 和 SCORE 分配内存,然后再读入学号数组与成绩数组。数组使用完毕之后,需要用释放语句将其所占的内存释放。程序如下:

```

PROGRAM SORT_2
C      IMPLICIT NONE
A:     INTEGER,DIMENSION(:),ALLOCATABLE::NUM,SCORE
        INTEGER::L,I,J,T
        READ *,L
        ALLOCATE(NUM(1:L),SCORE(1:L))
        READ *,(NUM(I),SCORE(I),I=1,L)
        DO I=1,L-1
            DO J=I+1,L

```

```

        IF(SCORE(I)<SCORE(J))THEN
            T=SCORE(I); SCORE(I)=SCORE(J); SCORE(J)=T
            T=NUM(I); NUM(I)=NUM(J); NUM(J)=T
        END IF
    END DO
END DO
PRINT '("NO.",I2,";",I5,5X,"SCORE:",I3)',(I,NUM(I),&
        SCORE(I),I=1,L)
DEALLOCATE(NUM,SCORE)
END PROGRAM SORT_2

```

7.2 多维数组

一维数组中一般用一对维界来描述数组形状,使用时也用一个下标引用数组元素。实际上,在一个数组的属性说明中,用几对维界来描述数组形状就称该数组为几维数组。一维数组的元素在内存中按下标从小到大排列。对多维数组而言,只有一种排列方式还不能唯一地确定每个元素的位置,需要同时采用几种方式排列。设有一个矩阵如图 7-2 所示,它有 3 行 4 列,共 12 个数。对其中任一个数,都既有行的序号又有列的序号,仅说出它的行号或列号都不能唯一地确定该数的位置。例如要查找按行排序为 3 的数时,由于第 3 行中有 4 个数,不能确定具体的位置;而按列排序方式查找序号为 2 的数时,由于第 2 列共有 3 个元素,也不能确定其位置。因此必须同时给出行序号和列序号才能唯一地确定所找的元素。

1	3	5	7
2	4	6	8
9	7	5	3

图 7-2 3 行 4 列矩阵

为了将图 7-2 中矩阵的数据形象地存入数组中,需要说明二维数组,其说明语句应为:

```
REAL,DIMENSION(1:3,1:4)::A34
```

第 1 维是行序,第 2 维是列序;第 1 维的维界偶是 1:3,第 2 维的维界偶是 1:4。

7.2.1 二维数组的概念

二维数组的说明与一维数组的说明形式类似,只是在数组形状说明中增加了一对维界,其说明语句的一般形式是:

类型说明([长度说明],[种别说明]),DIMENSION($d_{11}:d_{12},d_{21}:d_{22}$),其它属性说明::数组名表
例如:

```
INTEGER,DIMENSION(1:4,1:5)::A
```

```
REAL,DIMENSION(1:7,1:15)::R
```

分别说明了二维整型数组 A 和二维实型数组 R。

二维数组具有两对维界,即两个界偶。它的一般形式是:

$d_{11}:d_{12},d_{21}:d_{22}$

其维界的规定与一维数组一样,不再赘述。

二维数组的大小可由以下公式计算

$$(d_{12} - d_{11} + 1) \times (d_{22} - d_{21} + 1)$$

数组的大小(元素的个数)有时也称数组的体积,它可由数组形状描述。假如有说明

REAL,DIMENSION(2:4,3:6)::A

则数组 A 的大小为 $(4-2+1) \times (6-3+1) = 12$, 即数组 A 有 12 个元素。

二维数组片段的书写与用法与一维数组片段基本相同,只是在书写时增加一维,其一般形式为:

数组名([表达式 11]:[表达式 12]:[表达式 13],[表达式 21]:[表达式 22]:[表达式 23])

如数组片段:

V(2:4:2,1:7:3)

表示这个片段内有元素 V(2,1)、V(2,4)、V(2,7)、V(4,1)、V(4,4)和 V(4,7)。

在下标三元组中,某一维的表达式可以用一个整型向量名代替,该向量内诸元素的值(整型)即代表该维选取的下标值。假设 W 是向量,共有两个元素,它们的值依次取为 5 和 8。则下列二维数组 C 的片段

C(1:4:3,W)

表示被选取的元素分别是 C(1,5)、C(4,5)、C(1,8)和 C(4,8)。

在片段书写格式中,某个下标三元组表达式也可被一整型值代替,它表示在该维始终取该值为下标。例如数组片段

C(1:4:3,6)

第二维的下标三元组表达式为 6,表示第二维的下标始终取 6。该数组片段代表两个元素: C(1,6)和 C(4,6)。

另外,对二维数组而言,前述的可调数组、假定形状数组和动态数组的说明和使用方法等与一维数组一样,只是维数增加一维。

7.2.2 二维数组的存储与输入输出

由二维数组的说明可知,二维数组在其形状描述和构成数组片段时与一维数组相似。所以使用二维数组时,与一维数组有许多相同之处。但在具体进行一个二维数组的输入或输出时却有一个问题,即按什么样的顺序进行输入或输出。例如有一个数组 A 为 3 行 3 列(图 7-3),第一个下标表示行序,第二个下标表示列序。当我们执行读语句(写语句情况类似)

READ *,A

时,需要读入 9 个数。第 1 个数当然是送入 A(1,1),第 2 个数如果采用行优先原则,应赋给 A(1,2);反之,如果数组接收数据时采用列优先原则,第 2 个数应赋给 A(2,1)。

A(1,1)	A(1,2)	A(1,3)
A(2,1)	A(2,2)	A(2,3)
A(3,1)	A(3,2)	A(3,3)

图 7-3 3×4 数组

FORTRAN 语言规定,数组在机内存储时,按列优先存放,即先存完第 1 列内的各个元素,接着再存放第 2 列内诸元素,然后存储第 3 列、第 4 列直到最后一列存放完毕。因此上述输入语句读入的数,应按读入的先后次序先赋给第 1 列内(竖向)各个元素,而后再赋给第 2 列、第 3 列,即

输入顺序为: A(1,1)、A(2,1)、A(3,1)、A(1,2)、A(2,2)、A(3,2)、A(1,3)、A(2,3)、A(3,3)。这就是按列存储方式。

由于二维数组是按列存储的,所以在执行上面读语句 READ *, A 时,如果输入数据为 1.1,2.2,3.3,4.4,5.5,6.6,7.7,8.8,9.9 时,1.1 到 3.3 依次送给第 1 列各元素 A(1,1)、A(2,1)、A(3,1),4.4 到 6.6 依次送给第 2 列各元素 A(1,2)、A(2,2)、A(3,2),7.7 到 9.9 依次送给第 3 列各元素 A(1,3)、A(2,3)、A(3,3)。这时数组 A 及其数据如图 7-4 所示。

按列存储方式与数学中接触的矩阵按行处理的习惯不一致。在给数组输入数据时,应先敲入第 1 列元素的值,再敲入第 2 列、第 3 列的值,这样机内收到的数据才与想象中的相同。例如,解二元一次方程组时,它的系数矩阵是两行两列(用 2×2 的数组 A 存储),设其系数及形状如图 7-5 所示。为了用程序解这个方程组,需要输入方程的系数。当执行语句

READ *, A

时,要输入四个数,此时必须先输入第 1 列的两个数,再输入第 2 列的两个数:

5.0,7.0,6.0,8.0

这样机内收到的系数矩阵符合本来形状。如果不慎按行输入,即输入为

5.0,6.0,7.0,8.0

则处理系统收到数据时,仍把前两个数送给第 1 列,后两个数赋给第 2 列,因此实际收到的系数矩阵如图 7-6 所示。这与想象中的系数矩阵不一致,计算结果当然不会正确。

如果一定要按行输入系数矩阵,可以通过使用隐含 DO 循环来实现。

一个二维数组的元素按列下标排序,其元素下标变化如图 7-4,也即当把元素一般地写成 A(I,J)时,J 先停在 1 上,I 遍历 1 到 3,而后 J 加 1,I 再遍历 1 到 3,直到 J 取到维上界为止。由此,我们看到让 J 作为外层循环变量,I 作为内层循环变量的二重隐含 DO 循环((A(I,J),I=1,3),J=1,3)与直接使用数组名 A 的意义是相同的。即输入语句

READ *, A

与输入语句

READ *, ((A(I,J),I=1,3),J=1,3)

是等价的。

如果在隐 DO 循环中,把 J 作为内层循环变量,I 作为外层循环变量,写成:

((A(I,J),J=1,3),I=1,3)

则相当于元素序列: A(1,1)、A(1,2)、A(1,3)、A(2,1)、A(2,2)、A(2,3)、A(3,1)、A(3,2)、

A(1,1)	1.1
A(2,1)	2.2
A(3,1)	3.3
A(1,2)	4.4
A(2,2)	5.5
A(3,2)	6.6
A(1,3)	7.7
A(2,3)	8.8
A(3,3)	9.9

图 7-4 二维数组存储

A(1,1): 5.0	A(1,2): 6.0
A(2,1): 7.0	A(2,2): 8.0

图 7-5 二元方程组系数矩阵

A(1,1): 5.0	A(1,2): 7.0
A(2,1): 6.0	A(2,2): 8.0

图 7-6 不希望的系数矩阵

A(3,3),这实际是按行优先次序列出各元素。对于如图 7-5 的二维数组,如果使用读语句:

```
READ *,((A(I,J),J=1,2),I=1,2)
```

则可以按行输入数据:

5.0,6.0,7.0,8.0

此时可以把前两个数赋给第一行的两个元素,而后两个数赋给第二行的两个元素,机内收到的数据如图 7-5 所示,这是正确的。

因此,直接使用二维数组名时表示按列优先次序使用各个元素;也可以通过隐含 DO 循环按行(或列)优先次序使用各个元素。当输入(或输出)二维数组时,要仔细体会程序中是按行还是按列优先次序组织的语句。从而决定按什么方式输入(或输出)数组中的数据。

[例 7-5] 某工厂生产了 8 种产品,由 5 位推销员负责推销,图 7-7 是某月各推销员推销产品数量图。第 1 行表示第 1 种产品销售的数量,第 2 行表示第 2 种产品销售数量,……。第 1 列代表第 1 位销售员推销的各种产品的数量,第 2 列代表第 2 位销售员推销的各种产品的数量,……。要求根据输入数据输出以下表格。表格中最后 1 列表示各推销员推销的产品总数,表格中最后 1 行表示各种产品销售总数。

3	3	0	2	5
0	2	0	1	5
10	1	0	1	7
0	1	0	0	6
3	3	2	2	0
0	4	6	8	0
0	4	8	0	0
2	1	4	3	5

图 7-7 推销员推销产品数量

NUM OF PRODUCT									
SALESMAN:	1	2	3	4	5	6	7	8:	TOTAL

1	:	3	0	10	0	3	0	0	2: 18
2	:	3	2	1	1	3	4	4	1: 19
3	:	0	0	0	0	2	6	8	4: 20
4	:	2	1	1	0	2	8	0	3: 17
5	:	5	5	7	6	0	0	0	5: 28

TOTAL:	13	8	19	7	10	18	12	15	

在程序中说明了 8×5 的数组 SALE 来存放以上推销数据。数组 NUM 存放每种产品的推销总数,数组 EVERY 存放每位推销员推销的产品总数。程序如下:

```
PROGRAM PRODUCT
  IMPLICIT NONE
  INTEGER,DIMENSION(1:8,1:5)::SALE
  INTEGER,DIMENSION(1:8)::NUM
  INTEGER,DIMENSION(1:5)::EVERY
```

```

INTEGER::I,J
CHARACTER(LEN=70)::FORM
READ *,((SALE(J,I),I=1,5),J=1,8)
DO I=1,5
    EVERY(I)=0
    DO J=1,8
        EVERY(I)=EVERY(I)+SALE(J,I)
    END DO
END DO
DO I=1,8
    NUM(I)=0
    DO J=1,5
        NUM(I)=NUM(I)+SALE(I,J)
    END DO
END DO
PRINT '(27X,"NUM OF PRODUCT"/8X,"SALESMAN:",8I4,&
      ": TOTAL",/6X,51("-"))',(I,I=1,8)
PRINT '(9X,I4,3X,":",8I4,":",I5)',(J,(SALE(I,J),I=1,8),&
      EVERY(J),J=1,5)
PRINT '(6X,51("-"),/8X,"TOTAL :",8I4)',(NUM(I),I=1,8)
END PROGRAM PRODUCT

```

7.2.3 向量下标

向量下标指定一个下标的序列,该序列中表达式的值组成一个下标序列。假设 U 和 V 分别是形状为(1:3)和(1:5)的一维数组, Z 是形状为(1:5,1:7)的二维数组, U 和 V 的取值分别是:

$$U = (/1,5,3/)$$

$$V = (/2,1,1,5,3/)$$

就可以用 U 、 V 作为向量下标访问数组。例如 $Z(2,V)$ 代表数组 Z 第 2 行中的元素 $Z(2,2)$ 、 $Z(2,1)$ 、 $Z(2,1)$ 、 $Z(2,5)$ 和 $Z(2,3)$, $V(U)$ 代表元素 $V(1)$ 、 $V(5)$ 和 $V(3)$, $Z(U,V)$ 由元素

$$Z(1,2) \ Z(1,1) \ Z(1,1) \ Z(1,5) \ Z(1,3)$$

$$Z(5,2) \ Z(5,1) \ Z(5,1) \ Z(5,5) \ Z(5,3)$$

$$Z(3,2) \ Z(3,1) \ Z(3,1) \ Z(3,5) \ Z(3,3)$$

组成。

在使用向量下标时,作为下标的数组必须有定义(有值),且其值要在相应的数组维界之内。例如访问元素序列 $Z(U,V)$ 时, U 和 V 中各个元素都必须有值,而且 U 的各个元素的值必须在 Z 的第 1 维的维界 1 到 5 之间; V 的各个元素的值必须在 Z 的第 2 维的维界 1 到 7 之间。

7.2.4 n 维数组

通过前边的叙述可知,具有一个下标的数组元素所组成的数组称为一维数组。具有两个下标的数组元素所组成的数组称为二维数组,依此可知任意维数组的定义。FORTRAN90 规定数组的维数最多可用到七维,因此数组元素的下标最多可有七个。通常,我们把二维和更多维数的数组称为多维数组,而一维数组和多维数组统称为 n 维数组。

用类型说明语句和 DIMENSION 属性说明 n 维数组,其一般形式为:

类型说明([(长度说明],[种别说明])),DIMENSION(形状说明),其它属性说明::数组名表
其中形状说明的形式如下:

$$d_{11}:d_{12},d_{21}:d_{22},\cdots,d_{n1}:d_{n2}$$

这里 $d_{11}:d_{12},d_{21}:d_{22},\cdots,d_{n1}:d_{n2}$ 分别为第 1 维、第 2 维、 \cdots 、第 n 维的维界偶。例如说明语句

REAL,DIMENSION(1:2,1:4,1:6)::R

说明了 3 维实型数组 R,它有 3 个界偶,即 R 数组每个元素有 3 个下标,第 1 个下标范围从 1 到 2,第 2 个下标范围从 1 到 4,第 3 个下标范围从 1 到 6。其元素个数为

$$(2-1+1)\times(4-1+1)\times(6-1+1)=48$$

n 维数组($n\geq 2$ 时)与二维数组具有相似的结构和性质,其基本概念都可以从二维数组中引申而来。如数组片段的使用、可分配数组的说明以及数组的形状等都适用于 n 维数组。n 维数组存放时要按第 n 维、第 n-1 维、 \cdots 、第 2 维、第 1 维的次序进行, $n=2$ 时就是按列(第 2 维)存放。

n 维数组输入输出是按其存放次序进行的,即按第 n 维、第 n-1 维、 \cdots 、第 2 维、第 1 维的次序进行。最后一维的下标变化最慢,第一维下标变化最快。例如,下述数组说明

REAL,DIMENSION(1:2,1:3,1:2)::A

表明 3 维数组 A 在内存的存储方式应如图 7-8 所示。

直接用数组名进行输入输出时,其顺序为 A(1,1,1)、A(2,1,1)、A(1,2,1)、A(2,2,1)、A(1,3,1)、A(2,3,1)、A(1,1,2)、A(2,1,2)、A(1,2,2)、A(2,2,2)、A(1,3,2)、A(2,3,2)。如果想按其它顺序输入输出,应使用隐含 DO 循环。

程序设计中比较常用的是一维和二维数组,三维数组用得较少,超过三维的数组几乎不用。因此,应把学习的重点放在一维和二维数组上。

A(1, 1, 1)
A(2, 1, 1)
A(1, 2, 1)
A(2, 2, 1)
A(1, 3, 1)
A(2, 3, 1)
A(1, 1, 2)
A(2, 1, 2)
A(1, 2, 2)
A(2, 2, 2)
A(1, 3, 2)
A(2, 3, 2)

图 7-8 3 维数组存储图

7.3 数组运算

与其它语言不同,FORTRAN90 提供了强大的数组运算功能,它允许把整个数组作为一个操作数进行运算,也允许在赋值语句中对整个数组或其片段进行赋值,就像对一个简单变量的赋值一样。

7.3.1 数组赋值

数组赋值语句的形式与一般变量赋值相似,其一般格式是:

$$A=e$$

这里 A 是一个数组名或数组片段, e 是数组名或数组片段的表达式。但 e 中出现的数组或数组片段的形状必须与 A 相同, 否则不能赋值。形状相同指维数相同, 每维长度相同, 但每维上下界可以不同。例如数组 A 的形状为(3:5,1:2), 数组 B 的形状为(11:13,5:6)时都是 3 行 2 列, 每一维的长度相同, 因而形状相同。赋值语句

$\begin{pmatrix} A(3,1) & A(3,2) \\ A(4,1) & A(4,2) \\ A(5,1) & A(5,2) \end{pmatrix}$	$\begin{pmatrix} 3.1 & 3.2 \\ 4.1 & 4.2 \\ 5.1 & 5.2 \end{pmatrix}$
(a) 数组A	(b) 数组B

图 7-9 数组 A 的形状与数组 B 的形状相同

$$A=B$$

是合法的。处理系统执行上述赋值语句时,并不考虑数组 A 中元素具体的下标与数组 B 中元素下标是否相同。只要它们形状相同,就能把 B 中每一位置上的值赋给 A 的相同位置上的元素。实际上,数组 A 的形状如图 7-9(a)所示,数组 B 的形状与数组 A 的形状相同,其数据如图 7-9(b)所示。执行 A=B 后, $A(3,1)=3.1$ 、 $A(4,1)=4.1$ 、 $A(5,1)=5.1$ 、 $A(3,2)=3.2$ 、 $A(4,2)=4.2$ 、 $A(5,2)=5.2$, 即数组赋值语句相当于给数组中每个相同位置上的元素一一赋值。

假设有另一数组 C, 形状为(1:2,1:3), 即 2 行 3 列, 虽然大小与 A 相等, 但形状不同, 因此不准做 A=C 这样的赋值。

7.3.2 数组表达式

数组表达式 e 中允许使用内部算术运算符, 如 +、-、*、/、** , 允许使用的操作数可以是数组名、数组片段、数组构成器或简单的标量等。这些操作要求参与操作的数组的形状都相同。

1. 数组与数组运算

两个数组进行“+”和“-”这些算术运算时, 其结果是一个形状相同的数组, 它的每个位置上元素的值是参与运算的相同位置上一对元素按照某个具体运算后的结果值。例如有数组 A 和数组 B, 其形状如图 7-10 所示。执行语句

$$C=A+B$$

之后, 数组 C 的结果如图 7-11 所示。数组与数组运算的结果还是数组, 应该用数组存放该结果。

表达式中的数组在进行运算后, 不再保留原来的下标形式, 由系统将它们按维界下界为 1 重新排列。这样参与运算的两个数组只要形状相同, 原来的下标是否相同已不重要。假设上述数组 A 的形状为(1:2,4:6), 数组 B 的形状为(3:4,5:7), A+B 的形状为(1:2,1:3), 在做 C=A+B 这种赋值操作时, C 也应是 2 行 3 列的数组(与 A+B 形状相同)。

2. 数组与标量运算

数组表达式可以与标量做算术运算, 其结果是数组。例如, 当 A 和 B 是形状相同的数组时, 可以有如下合法的赋值语句:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} \quad \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{pmatrix} \quad \begin{pmatrix} a_{11}+b_{11} & a_{12}+b_{12} & a_{13}+b_{13} \\ a_{21}+b_{21} & a_{22}+b_{22} & a_{23}+b_{23} \end{pmatrix}$$

图 7-10 数组 A 和数组 B 的形状

图 7-11 A+B 的结果

$$B = A - 5$$

数组 A 与标量值 5 相减时,实际是把 A 中各个元素的值均减去 5, A-5 的结果的形状与数组 A 相同,即数组 B 也应与 A 有相同的形状。其运算过程如下:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} - 5 = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} - \begin{bmatrix} 5 & 5 \\ 5 & 5 \end{bmatrix} = \begin{bmatrix} a_{11} - 5 & a_{12} - 5 \\ a_{21} - 5 & a_{22} - 5 \end{bmatrix}$$

由于 FORTRAN90 允许标量像上例那样参与数组的赋值操作,因此也可以对数组直接赋某个标量的值。例如可以有如下赋值语句

$$A = 0$$

它表示给数组 A 的所有的元素全部赋值为零,以此代替某些循环赋值的操作。

3. 数组作为内在函数的实元

FORTRAN90 的表达式中允许对数组求基本函数,其函数值仍是一个形状相同的数组。例如执行完语句

$$B = \text{ABS}(A)$$

之后,赋值号左边数组 B 的每个位置上的元素值就是被操作数组 A 对应位置元素取 ABS 函数之后的值,这里 A 和 B 必须是形状相同的数组。如果 A 和 B 都是一维数组,且下标均从 1 开始。则有 $B(1) = \text{ABS}(A(1))$, $B(2) = \text{ABS}(A(2))$, ...。

在数组表达式内,所有的 FORTRAN 内部运算符、内在函数都可使用,数组片段也和数组一样进行运算。这能减轻程序员编程的负担。

7.3.3 数组内在函数

FORTRAN90 提供了丰富的数组操作函数,但它们的函数值可以是数组、标量等,不一定是数组。下边介绍几个常用的数组操作函数。

1. 求数组大小函数(SIZE)

该函数的功能是求数组元素的个数,SIZE(A)的值一定是正整数。如果数组 C 是 3×4 的二维数组,则 SIZE(C)的值是 12,表示 C 有 12 个元素。

2. 求数组的极值等函数

该类函数有:

(1)求数组最大值元素的函数: MAXVAL;

(2)求数组最小值元素的函数: MINVAL;

(3)求数组最大值元素下标的函数: MAXLOC;

(4)求数组最小值元素下标的函数: MINLOC。

函数 MAXVAL 和 MINVAL 的功能比较容易理解。函数 MAXLOC 和 MINLOC 的功能分别是找出数组中最大值或最小值元素的位置,函数值是由各维下标组成的向量形式,但

这里返回的位置已经是经过整理的下标向量(维下界全为 1)。假设 A 是二维数组, MAXLOC(A)的返回值是由最大值元素下标组成的数组构成器。如果 A 的形状如图 7-12 所示, MAXLOC(A)的值是(3,2/),即最大值的元素在第 3 行第 2 列。

假设数组 D 的说明如下:

INTEGER, DIMENSION(-5:1), PARAMETER::D(/1,0,-12,-2,16,-1,8/)

D 的最大值是 16,它的下标是 -1。由于 MAXLOC 是与位置有关的函数,它返回系统将数组 D 按维界下界为 1 重新排列之后的最大元素下标的向量(即最大元素的位置),

因此 MAXLOC(D)的函数值是(/5/),而不是(/-1/)。

3	7	10	0
7	4	7	6
1	18	5	2

图 7-12 3 行 4 列数组 A

11	13	8
12	11	6

图 7-13 2 行 3 列数组 B

3. 求数组各元素之和的函数(SUM)和各元素之积的函数(PRODUCT)

函数 SUM 的功能是求各元素值之和。设数组 A 如图 7-12,则

$$\text{SUM}(A) = 3 + 7 + 1 + (-7) + 4 + 18 + 10 + 7 + 5 + 0 + 6 + 2 = 56。$$

函数 SUM 也可以有 3 个自变量,其一般形式为:

SUM(ARRAY, DIM, MASK)

其中 ARRAY 是被求和的数组名,是必选项。DIM 和 MASK 是可选自变量。MASK 起屏蔽作用,它的值为一个逻辑表达式。凡满足条件的元素求其函数值,不满足条件的元素则被屏蔽在外,不参加求函数值的运算(若没有满足条件的元素,其函数值为 0)。假设只想求正元素值之和,则有

$$\text{SUM}(A, \text{MASK} = A > 0)$$

其函数值为 $3 + 7 + 1 + 4 + 18 + 10 + 7 + 5 + 6 + 2 = 63。$

自变量 DIM 也是可选的,用来指明选中哪一维来求函数值,其取值范围必须在 1 到数组最大维数之间。如果自变量写 DIM=1,在这里表示按第 1 维的变化求和,即分别按列求和。原来数组是几列,其函数值也是一个几列的向量,其第 1 列的值是数组 A 的第 1 列元素之和,第 2 列的值是数组 A 的第 2 列元素之和,……。具体说,如果数组 B 形如图 7-13 所示,则函数 SUM(B, DIM=1)的值是一个向量[1, 27, -2],它们分别是 B 每列上各元素之和;而 SUM(B, DIM=2)是按行下标不变对不同的列(第 2 个)下标求元素和,实际上就是按行分别求和,函数值向量为[-6, 32]。

SUM 的 3 个自变量也可以都用。例如

SUM(B, MASK = A > 0, DIM = 1)

表示按列求正元素值之和,结果向量为[12, 27, 6]。

函数 PRODUCT 的功能是求数组内各元素之连乘积,其一般形式是

PRODUCT(ARRAY, DIM, MASK)

第一个参数 ARRAY 是必选的, DIM 和 MASK 是可选的,其使用方法与函数 SUM 相同。没有满足条件的元素时,其乘积为 1。

4. 求数组维的下界函数(LBOUND)和上界函数(UBOUND)

函数 LBOUND 的值指明数组每维的下界,引用方式为:

LBOUND(ARRAY, DIM)

其中 DIM 是可选项。函数 UBOUND 的值指明每维上界,用法与 LBOUND 相同。

5. 数组转置函数 (TRANSPOSE)

该函数的一般形式为

TRANSPOSE(MATRIX)

它要求数组 MATRIX 的维数为 2, 其形状为 (m, n)。函数的返回值是将自变量数组 MATRIX 行列上相对应的元素互换后的数组, 其维数也为 2, 形状为 (n, m)。

其它数组操作函数可以查附录和 FORTRAN90 有关资料。

7.3.4 屏蔽数组赋值

屏蔽数组赋值是指按照逻辑数组表达式的值屏蔽数组赋值语句中表达式的求值和赋值。屏蔽数组赋值可以由 WHERE 语句或 WHERE 结构实现, 它们分别类似于标量操作时的 IF 语句与 IF 结构。

WHERE 语句的一般形式为:

WHERE(数组关系表达式)数组赋值语句

其中数组关系表达式是以数组名为操作对象的关系表达式。当该表达式为真时, 执行后面的数组赋值语句, 否则不执行数组赋值语句。而所有这种赋值操作, 实际上都是对每个对应位置上元素的操作。

假设 A 和 B 均为数组, 且形状相同, 可以写 WHERE 语句如下:

WHERE(A<0) B=0

执行此语句, 系统会逐一检查数组 A 的各元素值, 当 A 某一位置上的元素值小于 0 时, 它对应位置上 B 的元素值置 0, B 的其它元素保持不变。该 WHERE 语句实际上相当于执行了以下 DO 结构(假定 A、B 都是一维数组, 下标从 1 开始):

DO I=1, SIZE(A)

IF(A(I)<0) B(I)=0

END DO

因此 WHERE 类似于标量操作时的 IF 语句。

而 WHERE 结构类似于标量操作时的 IF 结构, 仅供数组操作分支选择时使用, 但它只能包括两个部分: WHERE 块和 ELSE WHERE 块。WHERE 结构的一般形式为:

WHERE(数组关系表达式)

数组赋值语句

! WHERE 块

...

ELSE WHERE

数组赋值语句

! ELSEWHERE 块

...

END WHERE

假设 A、B、C、D 为形状相同的数组, 则

WHERE(C==0)

C=1

ELSE WHERE

A=B/C

D=B * C

END WHERE

是合法的。为了便于解释,假设这 4 个数组均是一维的,维上、下界均相同。执行这个 WHERE 结构时,逐个检查数组 C 的所有元素 C(I)。当某一 C(I)值等于 0 时,把该元素重新赋值为 1;若 C(I)不等于 0 时,置 $A(I)=B(I)/C(I)$, $D(I)=B(I) * C(I)$ 。

WHERE 结构只能用于数组操作,不允许嵌套,其可选块也比 IF 结构少,写法形式与 IF 结构也有差异。反之,IF 结构只能用于标量运算,不能用于数组操作。

7.4 数组作为过程变元

数组不但可以象变量那样参与算术运算、函数运算和被赋值,还可以在过程调用中充当过程变元(虚元和实元)和作为函数的返回值。FORTRAN 中的过程有两种,一种是函数过程,另一种是子例过程。实际上,FORTRAN90 中不但允许数组作为过程变元,而且数组片段也可以作为过程变元。本节主要介绍用数组元素和数组作为过程变元等方法。

7.4.1 数组元素作为过程变元

数组元素只能作为过程的实元,不能作为虚元。数组元素作为实元时,与之相应的虚元只能是类型与之相同的变量。因此数组元素作为实元时,其虚元的有关规定与第 6 章的叙述相同。

[例 7-6] 编写判断一个整数是否大于零的函数子程序,通过调用它求出 10 个整数中的一维数组中正整数的个数。程序如下:

```
FUNCTION ELEMENT_GT0(ELEMENT) RESULT(A_GT0)
  IMPLICIT NONE
  INTEGER, INTENT(IN)::ELEMENT
  INTEGER::A_GT0
  IF(ELEMENT>0) THEN
    A_GT0=1
  ELSE
    A_GT0=0
  END IF
END FUNCTION ELEMENT_GT0
PROGRAM ARRAY_GT0
  IMPLICIT NONE
  INTEGER, DIMENSION(1:10)::A
  INTEGER::I, NUM=0, ELEMENT_GT0
  READ *, A
  DO I=1,10
    NUM=NUM+ ELEMENT_GT0(A(I))
  END DO
```

```
PRINT *, 'NUM=', NUM
END PROGRAM ARRAY_GT0
```

函数子程序 ELEMENT_GT0 判断虚元 ELEMENT 的值是否大于零。若其值 >0 , 函数则返回 1; 否则返回 0。主程序用数组元素 A(I) 作为实元, 与虚元 ELEMENT 进行虚实结合。由于主程序中的 I 从 1 到 10 进行变化(增量为 1), 因而 DO 结构结束之后 NUM 中的值就是数组 A 中大于零的元素的个数。另外, 主调过程应对被调函数的类型进行说明, 本程序中主程序使用的说明方法是:

```
INTEGER::ELEMENT_GT0
```

7.4.2 数组作为函数过程变元及函数值

就函数调用来说, 一般都需要传递数据。函数中的虚元既可以是一个简单变量(整型、实型、字符型或逻辑型等), 也可是一个数组名。一般而言, 如果函数子程序中所用的某个数组中的元素值需要由主调程序传递而来, 或函数子程序中计算出的某个数组中的值要传递给主调程序时, 该子程序中的数组应列为虚元(又称虚数组), 其数组名要写入虚元表中。而主调程序中与虚数组发生数据传递关系的数组应在函数调用时列为实元(实数组), 放入实元表中。作为虚元的数组必须是数组名, 而作为实元的数组可以是数组名或其成分(如数组片段)。另外, 虚数组的大小不能超过实数组。

当形状相同的实数组与虚数组相结合时, 实数组的第 1 个元素和虚数组的第 1 个元素结合, 实数组的第 2 个元素和虚数组的第 2 个元素结合, ..., 实数组的最后 1 个元素和虚数组的最后 1 个元素结合。

虚数组可区分为常数组、假定大小数组和可调数组等多种情况。另外, 数组还可以作为函数的返回值。下边通过几个例子介绍数组作为虚元和函数返回值为数组的方法。

1. 常数组作虚元

[例 7-7] 自编函数子程序求 3×5 数组中的最大元素值。

虽然该问题可以用内在函数 MAXVAL 得以解决, 但是为了熟悉数组作为变元的使用方法, 我们自编函数 ARRAY_MAXVAL1 完成该功能。程序如下:

```
FUNCTION ARRAY_MAXVAL1(ARR) RESULT(MAX_VAL1)
  IMPLICIT NONE
  INTEGER, DIMENSION(1:3, 1:5), INTENT(IN)::ARR
  INTEGER::MAX_VAL1
  INTEGER::I, J
  MAX_VAL1 = ARR(1, 1)
  DO I = 1, 3
    DO J = 1, 5
      IF (MAX_VAL1 < ARR(I, J)) MAX_VAL1 = ARR(I, J)
    END DO
  END DO
END FUNCTION ARRAY_MAXVAL1
```

```

PROGRAM ARR_MAX1
  IMPLICIT NONE
  INTEGER,DIMENSION(1:3,1:5)::A
  INTEGER::I,J,S,ARRAY_MAXVAL1
  READ *,((A(I,J),J=1,5),I=1,3)
  WRITE(*,'(5I5)')((A(I,J),J=1,5),I=1,3)
  S=ARRAY_MAXVAL1(A)
  PRINT *, 'MAXVAL= ',S
END PROGRAM ARR_MAX1

```

主程序中先按行输入 15 个整数放入 3×5 的数组 A 中,然后按行(列对齐原则)输出一
次,看一看 A 中输入的数据是否与实际数据一致。最后调用函数 ARRAY_MAXVAL1(A)
求出 A 中最大元素的值。

[例 7-8] 假设有两个形状相同(3 行 4 列)的数组 A 和 B,编写求 $A+B$ 的函数子程序,
并把 $A+B$ 的值作为函数值。

假设函数子程序名为 ARRAY_SUM1,主程序调用 ARRAY_SUM1 求两个数组之和。
由于函数 ARRAY_SUM1 的结果值是一个数组,必须在主调程序中编写接口块。程序如
下:

```

FUNCTION ARRAY_SUM1(A,B) RESULT(S_RESULT)
  IMPLICIT NONE
  REAL,DIMENSION(1:3,1:4),INTENT(IN)::A,B
  REAL,DIMENSION(1:3,1:4)::S_RESULT
  S_RESULT=A+B
END FUNCTION ARRAY_SUM1
PROGRAM ARRAY_PLUS1
  IMPLICIT NONE
  INTERFACE
    FUNCTION ARRAY_SUM1(A,B) RESULT(S_RESULT)
      REAL,DIMENSION(1:3,1:4),INTENT(IN)::A,B
      REAL,DIMENSION(1:3,1:4)::S_RESULT
    END FUNCTION ARRAY_SUM1
  END INTERFACE
  REAL,DIMENSION(1:3,1:4)::AA,BB,CC
  INTEGER::I,J
  READ *,AA,BB
  CC=ARRAY_SUM1(AA,BB)
  PRINT '(4F8.2)',((CC(I,J),J=1,4),I=1,3)
END PROGRAM ARRAY_PLUS1

```

主程序在调用函数 ARRAY_SUM1 时把 AA 和 BB 作为实数组,因此实元 AA 与虚数组 A

进行虚实结合,实元 BB 与虚数组 B 进行虚实结合,而把函数的结果赋给数组 CC。

在主程序中,由于说明了函数 ARRAY_SUM1 的接口,故不再需要在说明语句中说明函数名 ARRAY_SUM1 的类型。

2. 假定形状数组作虚元

例 7-7 中的函数子程序 ARRAY_MAXVAL1 由于其虚数组 ARR 是常数组,只能求 3 行 5 列数组的最大值。如果想求另一种形状实数组的最大元素值,必须改写该函数的定义。实际上可以用假定形状数组作虚元,由此编出的过程具有一定的通用性。

使用假定形状数组作虚元时,可与任何形状的维界偶的实元数组结合。假定形状的意思是假定虚数组的形状就是实元数组的形状。假定形状数组虚元一旦与实元相结合,就相应取实元数组的每维维界为自己的维界。

用假定形状数组作虚元改写例 7-7 如下:

```
FUNCTION ARRAY_MAXVAL2(ARR,M,N) RESULT(MAX_VAL2)
  IMPLICIT NONE
  INTEGER,DIMENSION(:,:),INTENT(IN)::ARR
  INTEGER,INTENT(IN)::M,N
  INTEGER::MAX_VAL2
  INTEGER::I,J
  MAX_VAL2=ARR(1,1)
  DO I=1,M
    DO J=1,N
      IF(MAX_VAL2<ARR(I,J))MAX_VAL2=ARR(I,J)
    END DO
  END DO
END FUNCTION ARRAY_MAXVAL2
PROGRAM ARR_MAX2
  IMPLICIT NONE
  INTERFACE
    FUNCTION ARRAY_MAXVAL2(ARR,M,N) RESULT(MAX_VAL2)
      INTEGER,DIMENSION(:,:),INTENT(IN)::ARR
      INTEGER,INTENT(IN)::M,N
      INTEGER::MAX_VAL2
    END FUNCTION ARRAY_MAXVAL2
  END INTERFACE
  INTEGER,DIMENSION(1:3,1:5)::A
  INTEGER::I,J,S
  READ *,((A(I,J),J=1,5),I=1,3)
  WRITE(*,'(5I5)')((A(I,J),J=1,5),I=1,3)
  S=ARRAY_MAXVAL2(A,3,5)
  PRINT *,'MAXVAL=',S
```



```
END PROGRAM ARR_MAX2
```

该程序中的函数 ARRAY_MAXVAL2 的虚数组 ARR 的形状为(:,:), 为假定形状数组。它与实元 A 进行虚实结合, 因此虚数组 ARR 就具有实数组 A 的形状(1:3,1:5), 函数的返回值就是数组 A 中的最大元素的值。假如用形状为(1:4,1:3)的数组 B 作为实元, 虚数组 ARR 的形状也是(1:4,1:3), 这时只要用以下语句

```
S=ARRAY_MAXVAL2(B,4,3)
```

调用函数 ARRAY_MAXVAL2, 函数的返回值就是 4×3 的数组 B 的最大元素值。由此可见, 函数 ARRAY_MAXVAL2 的功能(比如通用性)比函数 ARRAY_MAXVAL1 要强。从这一点来看, 函数 ARRAY_MAXVAL2 要比函数 ARRAY_MAXVAL1 好一些。

由于使用了假定形状数组, 因此需要在主调程序中书写函数 ARRAY_MAXVAL2 的接口, 同时不必在主程序中再对函数名 ARRAY_MAXVAL2 作类型说明。而例 7-7 原来的程序中由于没有接口块, 则需要主调程序中说明函数名的类型, 即在主程序说明部分中有说明语句:

```
INTEGER::ARRAY_MAXVAL1
```

例 7-8 中的函数 ARRAY_SUM1 也存在类似于例 7-7 中通用性不强的问题, 这也可以通过使用假定形状数组来解决。程序如下:

```
FUNCTION ARRAY_SUM2(A,B,M,N) RESULT(S_RESULT)
  IMPLICIT NONE
  REAL,DIMENSION(:,:),INTENT(IN)::A,B
  INTEGER,INTENT(IN)::M,N
  REAL,DIMENSION(1:M,1:N)::S_RESULT
  S_RESULT=A+B
END FUNCTION ARRAY_SUM2
PROGRAM ARRAY_PLUS2
  IMPLICIT NONE
  INTERFACE
    FUNCTION ARRAY_SUM2(A,B,M,N) RESULT(S_RESULT)
      REAL,DIMENSION(:,:),INTENT(IN)::A,B
      INTEGER,INTENT(IN)::M,N
      REAL,DIMENSION(1:M,1:N)::S_RESULT
    END FUNCTION ARRAY_SUM2
  END INTERFACE
  REAL,DIMENSION(1:3,1:4)::AA,BB,CC
  INTEGER::I,J
  READ *,AA,BB
  CC=ARRAY_SUM2(AA,BB,3,4)
  PRINT '(4F8.2)',((CC(I,J),J=1,4),I=1,3)
END PROGRAM ARRAY_PLUS2
```

本例中用 AA 和 BB 作为实数组,求出的是 AA+BB 的值。如果 A1 和 B1 是 5×5 的数组,用 A1,B1,5,5 作为实元表时,虚元 M 和 N 的值都是 5,虚数组 A 和 B 的形状是(1:5,1:5),求出的函数值是 A1+B1 的值,而函数 ARRAY-SUM2 的定义不需作任何改变。

3. 可调维长数组作虚元

编写通用的求数组最大元素值这类函数的另一种方法是使用可调维长数组。可调维长数组(又称可调数组)也可用作虚数组,即它的维界表达式是个尚未定义的整数变量,如 M、N 等,此时必须把虚数组名及其维界中的变量名都列为虚元。例如,求两个数组之和时,虚数组 A 和 B 可以用 $m \times n$ 的可调数组,这时 A、B、M 和 N 都应列入虚元表中,函数中有关语句可书写如下:

```
FUNCTION ARRAY-SUM3(A,B,M,N) RESULT(S_RESULT)
  IMPLICIT NONE
  INTEGER,INTENT(IN)::M,N
  REAL,DIMENSION(1:M,1:N),INTENT(IN)::A,B
  REAL,DIMENSION(1:M,1:N)::S_RESULT
  ...
END FUNCTION ARRAY-SUM3
```

当虚实结合时,若调用语句为

```
CC=ARRAY-SUM3(AA,BB,3,4)
```

实数组 AA 与虚数组 A 相结合,实数组 BB 与虚数组 B 相结合,虚元 M 与实元 3 相结合(即 M 的值为 3),虚元 N 与实元 4 相结合(即 N 的值为 4)。这样,虚数组 A 和 B 的形状都是(1:4,1:3),求出的函数值也是 4×3 的数组。另外,函数的返回值也可以用可调数组。

[例 7-9] 用函数子程序求一个 5×5 整型数组中主对角线上前 M 个元素之积。

我们用函数过程 IPR 来实现其功能,数组 A 采用可调数组。程序如下:

```
FUNCTION IPR(A,N,K) RESULT(A_RESULT)
  IMPLICIT NONE
  INTEGER,INTENT(IN)::N,K
  INTEGER,DIMENSION(1:N,1:N),INTENT(IN)::A
  INTEGER::A_RESULT
  INTEGER::I
  IF(K<=0 .OR. K>N)THEN
    PRINT *, 'ERR:',K
    A_RESULT=-1
  ELSE
    A_RESULT=1
    DO I=1,K
      A_RESULT=A_RESULT*A(I,I)
    END DO
  END IF
```

```

END FUNCTION IPR
PROGRAM ARRAY_FUN
  IMPLICIT NONE
  INTERFACE
    FUNCTION IPR(A,N,K) RESULT(A_RESULT)
      INTEGER,INTENT(IN)::N,K
      INTEGER,DIMENSION(1:N,1:N),INTENT(IN)::A
      INTEGER::A_RESULT
    END FUNCTION IPR
  END INTERFACE
  INTEGER::I,J,M,PROD
  INTEGER,DIMENSION(1:5,1:5)::X
  READ *,((X(I,J),J=1,5),I=1,5)
  READ *,M
  PROD=IPR(X,5,M)
  PRINT *,PROD
END PROGRAM ARRAY_FUN

```

在程序执行时,主程序通过如下语句

```
PROD=IPR(X,5,M)
```

调用函数过程 IPR(A,N,K)。由虚实结合,N 的值为 5,K 的值为 M,于是函数过程中定义了一个 5×5 的整型数组 A,其函数过程 IPR 变成可以对任意大小的整型方阵求其主对角线上前 K 个元素的乘积,也就是求主程序中数组 X 主对角线上前 M 个元素之积。

本程序也可以不写接口块,因为可调数组并不属于必须书写接口程序的条件之一。但有了接口块会使程序更容易理解和使用。

7.4.3 数组作为子例子程序变元

在子例子程序中,除了用变量作虚元外,也可以用数组作虚元。一般说来,若虚元是数组名,与之对应的实元也应是数组名。实际上,无论是函数子程序还是子例子程序,一般只要求虚、实数组具有同样的大小(也可以实数组的比虚数组大,反之则不行),并不要求它们具有同样的维界。假如有下例:

```

PROGRAM MAIN
  INTEGER,DIMENSION(2:14)::AA
  :
  CALL SUB(AA)
  :
END PROGRAM
SUBROUTINE SUB(DA)
  INTEGER,DIMENSION(-5:7):DA
  :

```

END SUBROUTINE

这是合法的。主程序调用 SUB 时,AA(2)和 DA(-5)结合,AA(3)和 DA(-4)结合,⋯,AA(14)和 DA(7)结合。其虚实数组的结合如图 7-14 所示。

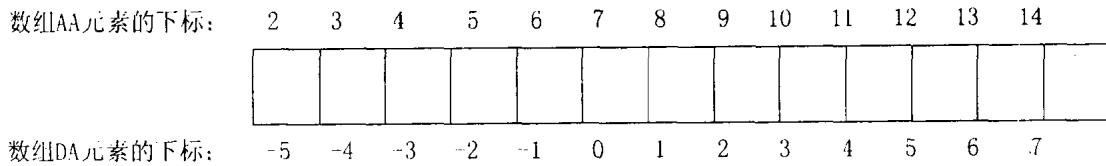


图 7-14 维界不同的数组 AA 和 DA 的虚实结合图

在子例子程序中,可以使用常数组作虚元,也可以使用假定形状数组作虚元,其使用方法完全与函数过程中使用假定形状数组作虚元相同。例如上述例子在子例过程中也可以使用假定形状数组作虚元。程序如下:

```

SUBROUTINE SUB(DA)
  INTEGER,DIMENSION(:)::DA
  :
END SUBROUTINE SUB
PROGRAM MAIN
  INTEGER,DIMENSION(2:14)::AA
  :
  CALL SUB(AA)
  :
END PROGRAM MAIN

```

本程序中主调程序调用子程序时,实元数组 AA 与虚元数组 DA 进行虚实结合,其中 DA 接收 AA 的维下界 2 作为自己的维下界,接收 AA 的维上界 14 作为自己的维上界。子例子程序虚实结合的方法与函数子程序完全相同。

[例 7-10] 用子例子程序改写例 7-9,求一个 5×5 整型数组中主对角线上前 M 个元素之积。程序如下:

```

SUBROUTINE IPR_SUB(A,N,K,A_RESULT)
  IMPLICIT NONE
  INTEGER,INTENT(IN)::N,K
  INTEGER,DIMENSION(1:N,1:N),INTENT(IN)::A
  INTEGER,INTENT(OUT)::A_RESULT
  INTEGER::I
  IF(K<=0 .OR. K>N)THEN
    PRINT *, 'ERR:',K
    A_RESULT = -1
  ELSE
    A_RESULT = 1
    DO I=1,K

```

```

        A_RESULT = A_RESULT * A(I,I)
    END DO
END IF
END SUBROUTINE IPR_SUB
PROGRAM ARRAY_SUB
    IMPLICIT NONE
    INTERFACE
        SUBROUTINE IPR_SUB(A,N,K,A_RESULT)
            INTEGER, INTENT(IN)::N,K
            INTEGER, DIMENSION(1:N,1:N), INTENT(IN)::A
            INTEGER, INTENT(OUT)::A_RESULT
        END SUBROUTINE IPR_SUB
    END INTERFACE
    INTEGER::I,J,M,PROD
    INTEGER, DIMENSION(1:5,1:5)::X
    READ *, ((X(I,J), J=1,5), I=1,5)
    READ *, M
    CALL IPR_SUB(X,5,M,PROD)
    PRINT *, PROD
END PROGRAM ARRAY_SUB

```

用子例子程序实现例 7-9 的功能时, 需要把原函数中存放返回值的变量 A_RESULT 移入虚元表中。由于 A_RESULT 参与虚实结合时只是向实元 PROD 传递计算结果, 因此它的 INTENT 属性应选择 OUT。

7.5 程序举例

数据排序是实际应用中比较常见的问题, 也是程序设计训练中比较有代表性的例题。一般来说, 排序是指将一批数按从小到大(或从大到小)重新排列起来。排序的方法有很多种, 高级语言程序设计教程中一般涉及的排序方法有 4、5 种, 但用这些方法组织的排序问题是很多的。

下边以 6 个数从小到大排序问题为例, 介绍几种常见的排序方法。为了便于讲述, 假设这 6 个数如下:

8, 5, 3, 6, 2, 4

最常用的想法是用第 1 个数分别与第 2 个、第 3 个、…、第 6 个数相比较, 如果次序不对, 则立即交换。如第 1 个数与后边各个数相比较(可能交换)之后的结果为:

5, 8, 3, 6, 2, 4 (第 1 个数与第 2 个数比较, 次序不对, 交换)

3, 8, 5, 6, 2, 4 (第 1 个数与第 3 个数比较, 次序不对, 交换)

3, 8, 5, 6, 2, 4 (第 1 个数与第 4 个数比较, 次序正确, 不交换)

2, 8, 5, 6, 3, 4 (第 1 个数与第 5 个数比较, 次序不对, 交换)

2,8,5,6,3,4 (第1个数与第6个数比较,次序正确,不交换)

经过一轮(5次)比较以后,已经把这6个数中的最小的数2交换到第1个数的位置。然后对第2至第6个数继续采用这种方法(第2轮比较交换),经过4次比较以后,把这5个数中的最小的数3交换到第2个数的位置。这时的数据顺序如下:

2,3,8,6,5,4

继续进行类似的第3轮、第4轮和第5轮比较交换,最后可得数据序列

2,3,4,5,6,8

这就是已排序的数据。一般情况下,这种对 n 个数所做的排序,需要 $n-1$ 轮比较。

上述排序方法在每次比较次序不对时都立即交换数据,称之为顺序交换法,也称为顺序排序法。例7-4采用的就是顺序交换法。如果每一轮比较只是记录最小值的位置,该轮比较结束之后才与最小的数交换,这称为选择排序法。对8,5,3,6,2,4这6个数进行选择排序时,第1轮比较后,发现第5个数最小,把第1个数和第5个数交换:

2,5,3,6,8,4

第2轮比较之后,第2个数和第3个数交换:

2,3,5,6,8,4

继续进行比较与交换,就可以实现排序的目的。如果对 n 个数做选择排序,也需要 $n-1$ 轮比较,但最多只进行 $n-1$ 次交换,极大地降低了交换次数。

另一种比较方法是对相邻的两个数比较:第1个数和第2个数比较,第2个数和第3个数比较,……,第5个数和第6个数比较,如果次序不对立即交换,第1轮比较以后可以把最大的数放到最后一个(第6个数)位置。然后对前5个数采用这种相邻数比较交换的方法,把剩下的5个数中最大的数换到第5个数的位置。继续这种比较与交换,显然也可以实现数据的排序。这种方法是把较大的数不断地换到后边,把较小的数不断地换到前边。如果把这些数竖起来摆放,就是把较大的数不断地换到下边(好像重物下降),把较小的数不断地换到上边(好像气泡上升)。因此,这种方法又称为下沉法或冒泡法。

排序的另一种方法是插入排序法。这种方法是假设已有 $i-1$ 个已排好序的数,然后把第 i 个数插入到适当的位置中。当然,插入第 i 个数时,比它大的数要顺次向后移动。这样,我们就可以把第1个、第2个、……、第 n 个数依次地插入到队列中,从而实现排序。

下边对冒泡排序和插入排序各举一个例子。

[例7-11] 编程序对10个整数按降序(从大到小)的次序进行冒泡排序。程序如下:

```
PROGRAM BUBBLE_SORT
  IMPLICIT NONE
  INTEGER, DIMENSION(10)::A
  INTEGER::I,J,T
  READ *,A
  DO I=1,9
    DO J=1,10-I
      IF(A(J)<A(J+1)) THEN
        T=A(J); A(J)=A(J+1); A(J+1)=T
      END IF
    END DO
  END DO
```

```

END DO
WRITE(*, '(5I5)')A
END PROGRAM BUBBLE_SORT

```

该程序的排序方法已在前面做了介绍,理解它并不困难。但本程序只能对 10 个整数排序,读者可以模仿例 7-4 的做法编写对 n 个整数排序的程序,也可以使用带有假定形状数组虚元或可调数组虚元的子例过程对 n 个整数进行排序。

[例 7-12] 对任意个正整数按升序(从小到大)进行插入排序。程序如下:

```

PROGRAM INSERT_SORT
  IMPLICIT NONE
  INTEGER, DIMENSION(1:100)::ARRAY
  INTEGER::I,X,N=0
  READ *,X
  DO WHILE(X>0. AND. N<100)
    N=N+1
    ARRAY(N)=X
    READ *,X
  END DO
  CALL INSERT(ARRAY,N)
  PRINT *, "THE NEW SEQUENCE:"
  PRINT '(10I5)', (ARRAY(I), I=1,N)
END PROGRAM INSERT_SORT

SUBROUTINE INSERT(A,N)
  INTEGER, DIMENSION(1:N), INTENT(INOUT)::A
  INTEGER, INTENT(IN)::N
  INTEGER::I,J,K,X
  DO I=2,N
    X=A(I)
    J=1
    DO WHILE(J<=I-1. AND. X>A(J))
      J=J+1
    END DO
    DO K=I-1,J,-1
      A(K+1)=A(K)
    END DO
    A(J)=X
  END DO
END SUBROUTINE INSERT

```

由于题中要对任意个正整数排序,在数据没全部输入之前还不能确定究竟有多少个数,因此不能简单地理解为 n 个数排序。主程序中采用了终止标记技术,即输入的数据小于零

时停止输入,这时变量 N 的值就是要排序的数据的个数。因为 N 值不能事先给出,所以无法说明足够大的数组存放要排序的数据。程序中增加了判断 $N < 100$,用它防止 N 值超过数组 ARRAY 的维上界。

过程 INSERT 的功能是对 N 个整数插入排序。虚数组 A 既从实数组 ARRAY 中得到未排序的数据,也把排好序的数据传回实数组,因此它的 INTENT 的属性取 INOUT。过程 INSERT 中让 I 从 2 到 N 做循环,这样,插入排序的问题就转变成了在循环体中把 A(I)插入到 A(1)至 A(I-1)之间的问题。在以 I 为循环变量外层循环体内嵌入了两个并列的 DO 结构,第一个 DO 结构用于寻找 A(I)应该占据的位置 J,第二个 DO 结构把比 A(I)大的数依次后移。最后把 A(I)的值放入 A(J)中,完成对 A(I)的插入。

[例 7-13] N 只猴子选猴王问题。选猴王的方法是:它们围成一圈,从 1 到 M 连续报数,凡报到 M 时,报该数的猴子就从圈中退出,然后剩下的猴子接着(从退出圈中的猴子后边开始)从 1 到 M 报数,直到只剩一个猴子时,该猴子就是猴王。

```
PROGRAM MONKEY
  IMPLICIT NONE
  INTEGER::N,M,KING_MONKEY
  READ *,N,M
  PRINT *, 'The number of monkey:',N
  PRINT *, 'Number off:',M
  PRINT *, 'King:',KING_MONKEY(N,M)
END PROGRAM MONKEY
FUNCTION KING_MONKEY(N,M) RESULT(KING)
  INTEGER,DIMENSION(:),ALLOCATABLE::A
  INTEGER,INTENT(IN)::N,M
  INTEGER::KING
  INTEGER::I,R,P
  ALLOCATE(A(1:N))
  DO I=1,N
    A(I)=I
  END DO
  R=N; I=1; P=0
  DO WHILE(R>1)
    IF(I>N)I=1
    IF (A(I)/=0) THEN
      P=P+1
      IF(P==M)THEN
        A(I)=0
        R=R-1
        P=0
      END IF
    END IF
  END DO
```



```

        END IF
        I=I+1
    END DO
    DO I=1,N
        IF(A(I)/=0)THEN
            KING=I
            EXIT
        END IF
    END DO
    DEALLOCATE(A)
END FUNCTION KING_MONKEY

```

由于猴子数 N 是变量,用它说明数组大小有一定的困难。一般来讲有以下处理方法:

- (1)把数组说明得大一些(如例 7-12),这时有可能浪费数组空间;
- (2)在过程中用假定形状数组作虚元,需要实数组与之虚实结合;
- (3)在过程中用可调数组作虚元,也需要实数组与之虚实结合;
- (4)使用动态数组。

本例中在函数中找猴王,为了避免使用数组作虚元,在函数过程中使用了动态数组。函数中用 R 表示还剩下的猴子数,用 P 表示这一轮已经数到了几个猴子,P 与 M 相等时就应让当前数到 P 的那个猴子退出。

当输入为 10 和 3 时,程序输出结果如下:

```

The number of monkey: 10
Number off: 3
King: 4

```

当输入为 100 和 7 时,程序输出结果如下:

```

The number of monkey: 100
Number off: 7
King: 50

```

[例 7-14] 利用过程求数组片段中各元素的和。

7.4 节讲述了数组元素和数组名作为过程变元的方法。而数组片段作为过程变元时只能作为实元,不能作为虚元。

用数组片段作为实元时,其对应的虚元必须是数组。在过程中对这种虚数组的使用方法与实元是数组时没有什么两样,我们只需要了解数组片段与虚数组的结合方法。本例中主程序有一个 4×5 的数组 A,用语句

```
S=SUM_PART(A(1:3,1:4),3,4)
```

调用函数时,实元 A(1:3,1:4)是数组片段,它对应的虚数组是可调的:ARR(1:M,1:N)。因此数组 A 的前 3 行、前 4 列共 12 个元素与虚数组 A 的 12 个元素相结合。这样,函数 SUM_PART 的返回值就是数组 A 的前 3 行、前 4 列这 12 个元素之和。

```

FUNCTION SUM_PART(ARR,M,N) RESULT(SUM_P)
    IMPLICIT NONE

```

```

INTEGER::M,N
INTEGER,DIMENSION(1:M,1:N),INTENT(IN)::ARR
INTEGER::SUM_P,I,J
SUM_P=0
DO I=1,M
    DO J=1,N
        SUM_P=SUM_P+ARR(I,J)
    END DO
END DO
END FUNCTION SUM_PART
PROGRAM ARRAY_PART
    IMPLICIT NONE
    INTEGER,DIMENSION(1:4,1:5)::A
    INTEGER::I,J,S,SUM_PART
    DO I=1,4
        DO J=1,5
            A(I,J)=5*(I-1)+J
        END DO
    END DO
    WRITE(*, '(5I5)')((A(I,J),J=1,5),I=1,4)
    S=SUM_PART(A(1:3,1:4),3,4)
    PRINT *, 'S=', S
END PROGRAM ARRAY_PART

```

例 7-14 的程序输出结果如下:

```

1   2   3   4   5
6   7   8   9  10
11  12  13  14  15
16  17  18  19  20
S= 90

```

[例 7-15] 利用随机过程产生 10 个处于区间[10,99]上的随机整数。

```

PROGRAM RANDOM_ARRAY
    IMPLICIT NONE
    INTEGER::I
    INTEGER,DIMENSION(1:10)::N_RAN
    REAL::X_RAN
    DO I=1,10
        CALL RANDOM_NUMBER(X_RAN)
        N_RAN(I)=INT(X_RAN*90+10)
    END DO

```

```
END DO
```

```
WRITE( *, '(10I5)')(N_RAN(I), I=1, 10)
```

```
END PROGRAM RANDOM_ARRAY
```

随机子例子程序 RANDOM_NUMBER(X_RAN)能通过虚实结合返回一个 $[0, 1)$ 上的随机实数。如果多次调用该子程序,返回的实数能在 $[0, 1)$ 上均匀分布。DO 结构循环体共执行 10 次,每次产生一个随机整数,放入数组 N_RAN 中。根据过程 RANDOM_NUMBER 的功能,用 X_RAN 作为实元调用该过程之后,有

$$0 \leq X_RAN < 1$$

做不等变换,不等号各边都乘以 90,得

$$0 \leq X_RAN * 90 < 90$$

不等号各边都加上 10,得

$$10 \leq X_RAN * 90 + 10 < 100$$

不等号各边都取整,得

$$10 \leq \text{INT}(X_RAN * 90 + 10) \leq 99$$

因此,用表达式 $\text{INT}(X * 90 + 10)$ 产生的整数在区间 $[10, 99]$ 上。程序运行结果为:

```
21  89  29  99  77  33  75  42  65  38
```

如果再运行一次该程序,发现其结果仍然是上边 10 个数。这说明两次运行得到的随机序列是相同的。为了解决这个问题,可在程序中第 5 行之后加上以下语句序列:

```
INTEGER, DIMENSION(1:8)::SEED
```

```
CALL DATE_AND_TIME(VALUE=SEED)
```

```
SEED(1) = 10000 * SEED(7) + 100 * SEED(6) + SEED(5)
```

```
CALL RANDOM_SEED(PUT=SEED)
```

再运行程序时,其结果为:

```
45  59  38  19  76  86  53  76  49  41
```

再运行一次,其结果会不同:

```
32  70  14  45  44  44  58  39  34  62
```

如果继续运行下去,会得到许多不同的随机整数序列,这是因为我们每次用了不同的种子 SEED 重新启动了由 RANDOM_NUMBER 使用的伪随机数生成器。DATE_AND_TIME 能返回系统日期和实时时钟数据,SEED(5)、SEED(6)和 SEED(7)分别返回的是代表当日时间的小时数、一小时之内的分和一分之中的秒,由它们进行运算赋给 SEED(1)的值每时每刻都是不同的,因而 RANDOM_SEED 中的种子也不相同,最后得到的随机整数序列也不相同。

习题 7

7-1 用类型说明语句说明

(1) 1 个有 10 个元素的整型数组 M。

(2) 1 个二维逻辑型数组 L,第 1 维界偶是 0:7,第 2 维界偶是 7:0。

(3) 1 个字符型数组,一维,下标从 1 到 100,取名为 C。

(4)把下列矩阵定义为数组,名为 R:

$$\begin{bmatrix} 1.1 & 1.2 & 1.3 \\ 1.4 & 1.5 & 1.6 \end{bmatrix}$$

7-2 什么是数组的形状,两个数组满足哪些条件才称为形状相同?

7-3 什么是数组片段,它是怎样书写的,是否要求数组片段的元素都是相连的? A(1:5:2, -2:8:2)表示哪些元素?

7-4 数组构成器起什么作用,形式如何? 能不能用在二维数组上? 设 A 和 B 都是一维实型数组,各有四个元素。编程序给 A 置初值 A(1)=1.1, A(2)=1.8, A(3)=1.3, A(4)=2.4; 用赋值语句给 B 赋值 B(1)=2.1, B(2)=2.2, B(3)=2.3, B(4)=2.4。

7-5 二维数组在内存中的次序是怎样的? 多维数组又是怎样存储的? 设数组形状如习题式),并写出各数的输出格式。

7-6 是否允许把数组名写入数组表达式中? 是否可以对数组名作算术运算,或对数组名求内在基本函数? 两个数组间进行运算或赋值时,应满足什么条件?

7-7 有数组 A、B、C、D、E 和 F 如下:

$$A: \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad B: \begin{bmatrix} -1.1 & -2.1 & -3.1 \\ -4.1 & -5.1 & -6.1 \end{bmatrix} \quad C: \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad D: (1 \quad 2 \quad 3)$$

数组 E 的描述为(1:2,1:3),数组 F 的描述为(1:3)。

问下列数组表达式是否合法? 对不合法的说明理由,对合法的写出计算结果:

$$(1) E = A + B$$

$$(2) E = \text{ABS}(B) + 2$$

$$(3) E = B + C$$

$$(4) C = A + C$$

$$(5) F = D * D$$

$$(6) F = A(1:2,1) + B(1:3,1)$$

$$(7) F = A(1,1:3) + B(1,1:3)$$

7-8 用 WHERE 结构,使习题 7-7 中 A 中元素小于或等于 2 时,置 B 中相应的元素为自身的绝对值,否则 B 中元素值加 1,并输出运算之后 B 的内容。

7-9 将 A 说明为二维整型常数组(2×2),将 B 说明为二维实型假定形状数组, C 为二维实型可调数组, D 为二维整型动态数组。并分别说明它的使用场合(在主程序或过程中)。

7-10 设有二维数组 A(3×6),分别编程实现:

(1)找出 A 的最大值元素及其行、列下标(假设最大值仅出现一次);

(2)找出第 K 列最大值;

(3)把最大值所在的这一行与第一行交换位置。

7-11 用动态数组存储读入的 N 个储款数,统计这些存款总数,而后释放该数组。再用它存储读入的 M 个取款数,统计这些取款总额,求出收支相抵的余额。

7-12 写一个程序,输出杨辉三角形的前 10 行:

$$\begin{array}{ccccccc} 1 & & & & & & \\ 1 & 1 & & & & & \\ 1 & 2 & 1 & & & & \\ 1 & 3 & 3 & 1 & & & \\ 1 & 4 & 6 & 4 & 1 & & \\ & & & & & & \\ & & & & & & \\ & & & & & & \\ & & & & & & \\ & & & & & & \end{array}$$

7-13 将 12 个实型数按降序排列,使用选择排序法。

7-14 输入 5 个学生的学号和 3 门课成绩,按学生的平均成绩从高到低输出成绩单。

7-15 用子例过程将 3×5 的二维数组的每一行按升序排序,要求用冒泡排序法。

7-16 有一个 $m \times n$ 数组,用函数过程求其中所有不相邻元素之和(从第 1 个元素开始)。

7-17 找出一个 $m \times n$ 数组中的马鞍点,所谓马鞍点是指某个元素在所在的行中最小,在所在的列中最大。数组中可能没有马鞍点,也可能有多个马鞍点。

7-18 输入 N 个数,把出现次数最多的那个数找出来,并统计出现的次数。

7-19 将 10 个数的数组 A 中的最大元素与 $A(10)$ 互换,将最小元素与 $A(1)$ 互换。

7-20 “纵横图”是每行、每列及两条对角线的和都相等的方阵。 $n \times n$ 的奇数纵横图的一种生成方法如下($n=5$ 时,见图 7-15):

17	24	1	8	15
23	5	7	14	16
4	6	13	20	22
10	12	19	21	3
11	18	25	2	9

图 7-15 5×5 纵横图

(1) 先把 1 填在第 1 行的中间位置;

(2) 下个数填在刚填入的那个数的右斜上方;

(3) 若右斜上方已出界,可想象把它卷成一个筒后继续填入;

(4) 若右斜上方已填入数,则下个数填在本数的下方。

把 1 至 $n \times n$ 之间的整数全部填完后,即可产生 $n \times n$ 的纵横图。编程序,对奇数 n 输出其纵横图。

第 8 章 字符型数据

FORTRAN90 由于引进了字符种别的概念,使 FORTRAN 能对各国文字(如汉字和日文)、各种学术符号等赋以不同的种别参数,从而可以对其进行操作,这就大大地增强了 FORTRAN 语言在信息管理方面的功能。

8.1 字符型数据基础

本节先介绍字符型常量与变量、字符子串、字符型表达式、字符赋值语句及字符比较等字符型数据的基础知识,为进一步学习字符型其它知识与操作打好基础。

8.1.1 字符型常量与变量

字符型常量是指括在一对单撇号" ' "或一对双撇号" "之间的一串字符,必要时可直接给出其种别参数。字符型常量又称字符串或字符型常数(见 2.3.3)。

用于存放字符型数据的变量就是字符型变量。字符型变量除了具有类型、种别和值之外,它比其它基本类型的量还多一个特性:长度。也就是说,在说明一个变量为字符型变量的时候,还要指明该变量最多能存放多少个字符。

用于说明字符型变量的关键字是 CHARACTER。其说明的基本格式为:

CHARACTER(LEN=n1[,KIND=n2])[,属性说明 1,属性说明 2,...,属性说明 n]::变量表列

格式中的 n1 和 n2 是一个整数或整型表达式,n1 代表被说明变量的长度,n2 代表字符变量的种别参数,各项属性说明是字符型变量有关属性的说明。例如:

CHARACTER(LEN=15,KIND=2)::A,CH

CHARACTER(KIND=3,LEN=8)::VAR,AU

CHARACTER(LEN=7),DIMENSION(1:10)::X,Y,Z

都是合法的说明语句。如果字符长度写在前,种别参数写在后,则可以简写为:

CHARACTER(15,2)::A,CH

当一个字符说明语句只有长度说明而无种别参数说明时,采用以下两种说明方法是等价的:

CHARACTER(LEN=15)::A

CHARACTER * 15::A

在 CHARACTER 后说明的长度为公用长度,如果被说明的变量的长度不全一样时,也可在相应变量的后边指明其具体的长度:

CHARACTER(LEN=10)::A,B * 5,C * 3

依据个别优于一般的原则可知,A 的长度是 10,B 的长度是 5,C 的长度是 3。

可以在字符说明语句中使用 PARAMETER 属性和 DIMENSION 属性来说明字符符号常数(字符符号常量)。例如:

CHARACTER(LEN=10),PARAMETER::NAME='FORTRAN_90'

该语句说明 NAME 是一个字符符号常数,其长度为 10,在程序中代表字符常量 'FORTRAN_90'。又如:

在前面曾提到,对于字符型变量,必须指明其长度,但有时可把长度值写成一个星号,以表示在此处暂时可以不指明其长度。例如可以有以下两个合法的等价说明

CHARACTER(LEN=*)::CH,PARA

CHARACTER*(*)::CH,PARA

但应注意,这种写法在许多处理系统中是有条件的,并非在任何情况下均可如此书写。一般而言,在以下两种状态下均可以采用星号来说明长度:

1° 具有 PARAMETER 属性的字符常量。例如以下两个语句

CHARACTER(LEN=*),PARAMETER::PARA='FORTRAN_90'

CHARACTER*(*),PARAMETER::PARA='FORTRAN_90'

给出的字符串长是确定的,因此字符符号常数 PARA 的长度是可知的(为 10),可用 * 说明。

2° 用字符变量作虚元时

在子程序中,用字符型变量作虚元时可以不指明其具体长度,即用 * 定义长度。这时它可以与任意长度的字符型实元相结合,有利于提高子程序的通用性。

8.1.2 字符子串

一个字符串中连续的一部分称为字符子串或子字符串,而原字符串称为该子串的母串。字符子串的一般形式为:

字符变量名(e1:e2)

其中,e1、e2 是整型表达式,代表该子串在母串中的起始和终止位置。例如,若有一字符变量 NAME,为其赋值为:

NAME='DAQING PETROLUUM INSTITUTE'

那么 NAME(1:6)就是 NAME 的一个子串,其值为 'DAQING',共含有 6 个字符。

如果字符串的长度为 L,e1、e2 的取值应满足

$1 \leq e1 \leq e2 \leq L$

字符子串的长度为 $e2 - e1 + 1$ 。例如 NAME(1:6)的长度为 $6 - 1 + 1 = 6$ (个字符)。另外,当 $e1 = 1$ 、 $e2 = L$ 时,可分别省写 e1 或 e2。例如,NAME(1:6)可简写为 NAME(:6); NAME(7:26)可简写为 NAME(7:); NAME(1:26)可简写为 NAME(:),它实际上就是 NAME 本身。

8.1.3 字符表达式与字符赋值语句

字符表达式是指用字符运算符把字符常数、字符变量或字符数组元素等字符型数据连接起来的有意义的式子。字符表达式具有字符型值。

字符运算符只有一个,就是字符连接符 '//',它是由两个斜杠组合而成的,其作用是将两个字符型数据连接起来,成为一个字符型数据。该运算符是一个双目运算符,在其两侧应各有一个字符型操作数。例如,'HE'//'LLO!' 的值为 'HELLO!', 'FORTRAN'//' '//90' 的

值为 'FORTRAN 90'。

单独一个字符常量、字符变量、字符子串、字符数组元素或字符函数等,都属于字符表达式的特例,可把它看作最简单的字符表达式。

对于已说明的字符变量和字符型数组元素,可以对其进行字符赋值操作。字符赋值语句的格式与前面章节介绍的格式相同:

V = e

只是在此处,赋值号两边涉及的都是字符型的量。

[例 8-1] 本例子用于熟悉字符型数据的赋值操作。

```
PROGRAM CHAR_1
  IMPLICIT NONE
  CHARACTER(LEN=5)::A,B,C
  CHARACTER(LEN=11)::D
  A='CHINA'
  B='JAPAN'
  C=A
  D=A//' '//B
  PRINT *,A,B,C
END PROGRAM CHAR_1
```

程序中用到的 4 个赋值语句均是字符赋值语句,其中最后一个赋值语句是将字符表达式 A//' '//B 的值求出之后赋给变量 D。因此 D 中值是字符串 'CHINA JAPAN'。

下面的赋值操作是非法的:

A = 1234 (1234 不是字符型数据)

C//A = B (赋值号左边是表达式)

在进行字符赋值操作时,有时赋值号两端数据的字符长度不等,这时会遵循以下原则:

- 1° 当赋值号右边表达式中字符个数等于赋值号左边变量中字符长度时,直接赋值。
- 2° 当赋值号右边表达式中字符个数少于赋值号左边变量中字符长度时,变量右端补足空格。例如字符变量 A 的长度为 7 时,执行语句

A = 'HANZI'

之后,A 中值为 'HANZI ',即 A 的尾部有 2 个空格。

- 3° 当赋值号右边表达式中字符个数多于赋值号左边变量中字符长度时,将右边多余的字符截去。例如,执行语句

A = 'OPERATING'

之后,A 中值为 'OPERATI'。

8.1.4 字符关系表达式

与数值型数据一样,字符型数据也能进行关系操作。用关系运算符将字符型操作数连接而成的有意义的式子叫字符关系表达式。例如,'A' >= 'B'、'THE' != 'THESE' 和 '0' > '9' 都是合法的字符型关系表达式。字符型关系表达式的值只有 .TRUE. 和 .FALSE. 这两种可能。

字符数据的比较是按着字符的代码进行的,任一系统中用到的各个字符都要用一个代码来表示。最常用的西文代码有两种,即 ASCII 和 EBCDIC。由于微机系统一般都采用 ASCII 码,因此本书中的讨论都是基于 ASCII(见附录 I)而言的。

ASCII(American Standard Code for Information Interchange)意为美国信息交换标准代码。其基本集有 128 个字符,扩充集有 256 个字符,每个字符用一个字节表示。

在 ASCII 基本集中,10 进制序号为 0~31 及 127 的字符为控制字符,一般在屏幕上不可显示,其它字符均为可显示的字符。在 ASCII 中,一些常用字符的代码值有如下的规律:

' '<'0'<...<'9'<'A'<...<'Z'<'a'<...<'z'

字符比较时要遵循以下规则:

1° 两个字符比较时,以它们的代码值决定大小。例如:

'C'<'D' 的值为真

'9'<'1' 的值为假

2° 两个字符串比较时,是将它们的值从左到右逐个字符进行比较。若所有字符完全相同,则两表达式相等;否则,以第一次出现不同字符的比较结果为准。例如:

'THESE'<'THOSE'

的值为真。因为第 3 个字符 'E'<'O',则前一表达式的值小于后者。

3° 若两个字符串中字符个数不等时,则将较短的字符串后面补足空格后再比较。例如进行以下字符型关系操作时

'where'<'wherever'

先将 'where' 后边补空格成为 'where ' 之后,再与 'wherever' 比较。在可显示字符中,因为空格的码值最小,则表达式 'where'<'wherever' 的值为真。

4° 当字符串中既有字母又有数字时,则应了解系统所用的是哪种编码方法。对于 ASCII 码,有 '0'<'9'<'A'<'a';而对于 EBCDIC 码,有 'z'<'A'<'Z'<'0'。

8.2 字符型数据的输入与输出

字符型数据的输入与输出也可分成表控格式(自由格式)和自定格式两种。

8.2.1 表控格式的输入与输出

[例 8-2] 用表控格式输入和输出字符型数据。

```
PROGRAM CHAR_2
  IMPLICIT NONE
  CHARACTER(LEN=6)::CITY
  CHARACTER(LEN=5)::COUNTRY
  CHARACTER(LEN=12)::NAME
  READ *,CITY,COUNTRY
  NAME=CITY//' '//COUNTRY
  PRINT *,NAME
  PRINT *,CITY,COUNTRY
END PROGRAM CHAR_2
```

在程序运行时,如果输入
'DAQING','CHINA'✓
其输出为
DAQING CHINA
DAQINGCHINA

用表控格式输入字符型数据时,必须输入用单撇号或双撇号做定界符的字符串。如果在一个记录上输入多个字符型常量,各常量之间要用逗号分隔。而输出时,按变量中的值进行输出(没有定界符),一般采用左对齐方式输出。在一个输出语句中输出多个字符型量时,各量之间一般是连续输出的,中间并不留空格。

对于表控输入时,如果输入字符个数与对应变量的长度不一致时,以变量长度为准。若读入的字符的个数少于变量长度时,则在尾部补足空格后送给变量;若读入字符个数多于变量长度时,则将尾部多余的字符截去后再送给变量。

8.2.2 自定格式的输入与输出

字符型数据进行格式输入输出时,应采用字符型格式编辑符 A,并有 A 和 Aw 两种形式。

1. 格式输入

(1) 仅用字符型编辑符 A

例如:

```
CHARACTER(LEN=5)::A,B,C*4  
READ '(3A)',A,B,C
```

若输入的字符串为

A1234B1234C12345

系统会根据各变量实际长度,自动截取相应的字符串。本例中,A 获得的值为 'A1234',B 获得的值为 'B1234',C 获得的值为 'C123',而最后两个字符 '45' 作为多余字符而被截掉。

由此可见,格式输入时字符串两边不应加定界符,系统会依据格式自动截取字符串。若加了定界符,该定界符会被系统当做输入的字符而送入相应的变量中。

(2) 用字符型编辑格式 Aw

A 是字符格式描述符,w 是规定的输入宽度,它必须是无符号整型常数。例如:

```
CHARACTER(LEN=5)::A,B,C*4  
READ '(A5,A3,A6)',A,B,C
```

若输入以下字符串

A1234B12C12345

按输入的格式,分别为变量 A 截取 5 位 'A1234',为变量 B 截取 3 位 'B12',为变量 C 截取 6 位 'C12345'。但 A、B、C 中实际得到的值还应取决于变量本身的长度 L,其实际输入时按以下原则处理:

1° 若 $L=w$,输入的字符个数恰好够用。上例中 A 的值是: 'A1234'。

2° 若 $L>w$,输入的字符不能满足,在后面补足 $L-w$ 个空格后送给变量。上例中,B 的值是: 'B12 '。

3° 若 $L < w$, 输入的字符不能全部接收, 从右边取 L 个字符送给变量(左边的字符去掉)。上例中, C 的值是: '2345'。

2. 格式输出

格式输出仍然有 A 和 Aw 两种形式。

(1) 仅用字符型编辑符 A

仅用编辑符 A 输出数据在前几章已经见过, 这里只举例复习一下。

[例 8-3] 仅用编辑符 A 输出数据举例。

```
PROGRAM CHAR_3
```

```
    IMPLICIT NONE
```

```
    CHARACTER(LEN=5)::A,B,C*4
```

```
    A='A1234'
```

```
    B='B1234'
```

```
    C='C123'
```

```
    PRINT '(A,A,A)',A,B,C
```

```
END PROGRAM CHAR_3
```

程序输出结果为:

A1234B1234C123

它是按字符变量的实际长度输出的, 各字符数据之间也没有空格。

(2) 用字符型编辑格式 Aw

同样使用上例中的变量, 将输出语句改为

```
    PRINT '(A5,A7,A2)',A,B,C
```

则输出结果为:

A1234 B1234C1

对字符型数据采用 Aw 形式输出时, 其实际的输出格式与说明符的宽度 w 和变量的长度 L 都有关系。具体输出时, 按以下原则处理:

1° 若 $w = L$, 其输出按相应的长度进行。上例中, A 的输出是: A1234。

2° 若 $w > L$, 在字符变量值的左边补 $w - L$ 个空格输出(右对齐)。上例中, B 的输出是: B1234。

3° 若 $w < L$, 输出字符变量值中左边的 w 个字符。上例中, C 的输出是: C1。

由以上字符型数据的格式输入输出例子可见, 使用字符型编辑描述符 A 进行输出的方法比较容易掌握。

与数值型数据的输入方式不同, 字符型数据的输入有时采用自定格式输入反而会降低输入工作量(见例 8-4)。

8.3 字符型数组

字符型变量如果具有 $DIMENSION$ 属性, 就成为字符型数组。例 8-4 中数组 A 共有 7 个元素, 由于每个元素可以存放一个字符(因为字符长度说明是 1), 所以数组 A 一共可以存放 7 个字符。该程序输入 7 个字符后按自定格式输出数组 A 的片段。

[例 8-4] 用自定格式输入和输出字符型数据。

```
PROGRAM CHAR_4
  IMPLICIT NONE
  CHARACTER(LEN=1),DIMENSION(1:7)::A
  INTEGER::I
  READ '(7A1)',(A(I),I=1,7)
  DO I=3,1,-1
    PRINT '(7A1)',A(1:7:I)
  END DO
END PROGRAM CHAR_4
```

程序运行结果如下:

FORTRAN✓

FTN

FRRN

FORTRAN

如果改为表控输入:

```
READ *,(A(I),I=1,7)
```

其输入应改为:

'F','O','R','T','R','A','N'✓

由此可见,自由格式要求在各个输入的字符串两边均应加一对撇号。7 个元素需要输入 7 个用撇号括起来的字符串。在字符型数据输入时,自由格式反不如自定格式方便。当然,上述 7 个字符也可以用长度为 7 的字符型变量存放,那样会更简单一些。

[例 8-5] 从键盘上输入 5 个城市的名称(用汉语拼音表示),然后按从小到大的顺序排序。程序如下:

```
PROGRAM CITY_SORT
  IMPLICIT NONE
  CHARACTER(LEN=10),DIMENSION(1:5)::CITY
  INTEGER::I,J
  CHARACTER(LEN=10)::TEMP
  READ *,CITY
  DO I=1,4
    DO J=I+1,5
      IF(CITY(I)>CITY(J)) THEN
        TEMP=CITY(I); CITY(I)=CITY(J); CITY(J)=TEMP
      END IF
    END DO
  END DO
  WRITE(*, '(A)')CITY
END PROGRAM CITY_SORT
```

该程序采用自由格式输入和自定格式输出,其运行结果如下:

'Harbin','Shanghai','Anda','Beijing','Daqing'↵

Anda

Beijing

Daqing

Harbin

Shanghai

[例 8-6] 利用字符型二维数组输出用“*”组成的田字图形。程序如下:

```
PROGRAM TIAN
  IMPLICIT NONE
  CHARACTER(LEN=1),DIMENSION(1:7,1:7)::T
  INTEGER::I,J
  DO I=1,7
    DO J=1,7
      IF(I==1.OR.I==4.OR.I==7.OR.J==1.OR.J==4.OR.&
        J==7) THEN
        T(I,J)='*'
      ELSE
        T(I,J)=''
      END IF
    END DO
  END DO
  WRITE(*,'(7A1)')((T(I,J),J=1,7),I=1,7)
END PROGRAM TIAN
```

程序运行结果如下:

```
* * * * *
*      *      *
*      *      *
* * * * *
*      *      *
*      *      *
* * * * *
```

8.4 用于字符处理的内在函数

FORTRAN90 中提供了许多与字符型操作有关的内在函数,本节介绍其中最常用的函数。

1. 求字符串长度函数(LEN 和 LEN_TRIM)

其一般形式为:LEN(String) 和 LEN_TRIM(String)

其中 String 为字符型常量、变量或数组名等字符串。

函数 LEN 的结果值是 String 中字符个数(包括前置及尾随空格),String 若是数组名,则结果为 String 中一个元素的字符个数。例如:LEN('ABCD')的值为 4。假如有

CHARACTER(LEN=5),DIMENSION(1:20)::A

则 LEN(A)的值为 5。

函数 LEN_TRIM 的值是把字符串去掉尾部空格后的长度。例如,LEN_TRIM('AB C D')的值为 5,LEN_TRIM(' ABC ')的值为 3,LEN_TRIM(' ')的值为 0。

2. 除去字符串尾部空格函数(TRIM)

其一般形式为:TRIM(String)

函数的结果值是去掉 String 中的尾部空格后剩余的字符串。例如,TRIM('A B ')的值为 'A B',TRIM(' ABC ')的值为 'ABC'。

3. 字符与字符序号相互转化函数(ICHAR、IACHAR、CHAR 和 ACHAR)

字符向字符序号转化函数的常用形式为:

ICHAR(CH) 和 IACHAR(CH)

其中 CH 是长度为 1 的字符型常量或变量。其函数的结果值为字符在相应处理系统中的字符序号,若要按照 ASCII 码求其序号,则应使用 IACHAR 函数。例如,IACHAR('A')的值为 65,IACHAR('Z')的值为 90。

在 FORTRAN90 中,也允许 String 所代表的字符长度超过 1,此时只取第一个字符作为有效字符。例如,ICHAR('ABS')的值为 65。

字符序号向字符转化函数的常用格式为:

CHAR(I) 和 ACHAR(I)

其中 I 可以是整型常量、变量或表达式。其函数的结果值为序号 I 所对应的字符,若要按照 ASCII 码求其对应的字符时,应使用 ACHAR 函数。例如,ACHAR(65)的值为 'A',ACHAR(90)的值为 'Z'。

4. 子串位置函数(INDEX)

其一般形式为:INDEX(String1,String2)

其中 String1,String2 均为字符型,且种别参数应该一致。若 String2 是 String1 的一个子串,其函数的结果值是一个正整数,该数表示 String2 在 String1 中最左边的子串的起始位置;若 String2 不是 String1 的子串,则函数结果值为 0。例如,INDEX('FORTRAN','AN')的值为 6,INDEX('FORTRAN','T R')的值为 0。

5. 字符串比较函数(LGE、LGT、LLE 和 LLT)

以字符串是否大于或等于函数 LGE 为例,它的一般形式为:

LGE(String1,String2)

若 String1 \geq String2,其函数结果值为真;否则为假。例如,LGE('ONE','TWO')的值为假。其它三个函数 LGT、LLE 和 LLT 分别表示字符串是否大于、小于或等于、小于函数。例如,LGT('ONE','TWO')的值为假,LLE('ONE','TWO')的值为真,LLT('ONE','TWO')的值也为真。

6. 首、尾部空格调整函数(ADJUSTL 和 ADJUSTR)

其一般形式为:ADJUSTL(String) 和 ADJUSTR(String)

调左函数 ADJUSTL 的函数值是把 String 的首部空格调到尾部,调右函数 ADJUSTR 的函数值是把 String 的尾部空格调到首部。

除以上函数以外,与字符有关的函数还有在字符串中寻找字符集中某一字符的函数 SCAN、证实一组字符是否在字符串中的函数 VERIFY 等(见附录Ⅲ)。

8.5 字符型数据作为过程的变元及函数值

在过程中,除了与主程序单元一样使用字符型数据以外,还可以把字符型数据作为过程的虚元、实元和作为函数子程序的返回结果。

8.5.1 字符型数据作为函数过程变元

由于字符型数据与数值型数据存在差异,因此,它们作为函数虚元的用法也有不同。

1. 字符型变量做虚元

当子程序中虚元是字符型变量时,对应的实元必须是字符型变量、字符型数组元素、字符串常量或字符表达式等。在子程序中出现的字符型虚元,必须在该子程序中用 CHARACTER 语句说明其类型。若采用固定长度说明时,其长度必须小于或等于所对应实元的长度;也可采用假定长度说明(*),在进行虚实结合时,其假定长度虚元变量自动与对应的实元取相同的长度。

[例 8-7] 编写函数子程序,实现字符内在函数 LEN_TRIM 的功能。在主程序中输入一个字符串,对去掉尾部空格部分的字符串中的字符逆序存放后输出。

```
FUNCTION LENTRIM(STRING) RESULT(LENTRIM_RESULT)
```

```
  IMPLICIT NONE
```

```
  CHARACTER(LEN=*),INTENT(IN)::STRING
```

```
  INTEGER::LENTRIM_RESULT,I
```

```
  LENTRIM_RESULT=0
```

```
  DO I=LEN(STRING),1,-1
```

```
    IF(STRING(I:I)/=' ')THEN
```

```
      LENTRIM_RESULT=I
```

```
      EXIT
```

```
    END IF
```

```
  END DO
```

```
END FUNCTION LENTRIM
```

```
PROGRAM CHAR_INVERSE
```

```
  IMPLICIT NONE
```

```
  CHARACTER(LEN=20)::STRING1,C*1
```

```
  INTEGER::I,K,LENTRIM
```

```
  PRINT *, "Input STRING: "
```

```
  READ "(A)",STRING1
```

```
  K=LENTRIM(STRING1)
```

```

DO I=1,K/2
  C=STRING1(I:I)
  STRING1(I:I)=STRING1(K-I+1:K-I+1)
  STRING1(K-I+1:K-I+1)=C
END DO
PRINT *, "INVERSE STRING IS: ",STRING1
END PROGRAM CHAR_ INVERSE

```

程序运行输入

Good morning! ✓

时,其输出为:

INVERSE STRING IS :! gninrom dooG

主程序的作用是输入一串字符,调用函数子程序求出从第一个字符到最后一个非空格字符的长度,并将其按逆序存放后再输出。

2. 字符型数组做虚元

当子程序中的虚元是字符型数组时,对应的实元必须是字符型数组或数组元素等。若虚实数组元素长度一致时,按以前描述的虚实结合方式进行。若长度不一致时,则虚实数组的元素间并不是一一对应结合,而是按数组变元全部元素的所有字符进行一个一个字符地相结合。例如,某一虚元数组有4个元素,每个元素长度是3,则总长度是12个字符;与之对应的实元数组有4个元素,每个元素长度是5,总长度为20个字符。当它们从第一个元素开始结合时,只是实数组的前12个字符参与了虚实结合,而实数组后面的8个字符并未使用。假设虚数组是A,实数组是B,其虚实结合图见图8-1,图中每个方格代表一个字符。

若虚元数组的长度用*号说明,则虚元数组中每个元素的长度与对应实元数组元素的长度一致。

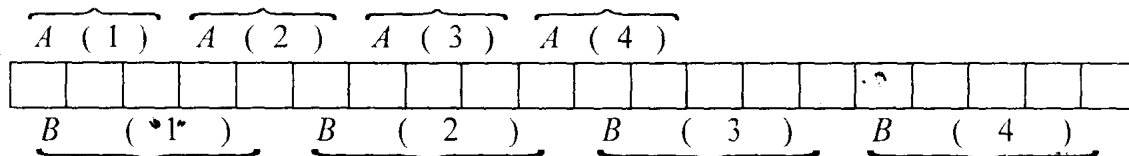


图8-1 字符型数据的虚实结合

8.5.2 字符型数据作为函数过程的返回值

函数子程序返回的结果为字符型数据的例子是十分常见的。我们通过例8-8熟悉字符型数组做虚元和函数返回字符型数据的方法。

[例8-8] 字符型虚实数组元素长度不一致的例子。

```

PROGRAM CHAR_8
  IMPLICIT NONE
  CHARACTER(LEN=5),DIMENSION(1:4)::S
  CHARACTER(LEN=3)::MAX_STRING,R

```



```

S = (/ "POUND", "FRANC", "DINAR", "RUPEE" /)
R = MAX_STRING(S)
PRINT *, "MAX_STRING =", R
END PROGRAM CHAR_8

FUNCTION MAX_STRING(STR) RESULT(R_STRING)
    IMPLICIT NONE
    CHARACTER(LEN=3), DIMENSION(1:4), INTENT(IN)::STR
    CHARACTER(LEN=3)::R_STRING
    INTEGER::I, K=1
    WRITE(*,*)STR
    DO I=2,4
        IF(STR(I)>STR(K))K=I
    END DO
    R_STRING=STR(K)
END FUNCTION MAX_STRING

```

程序输出结果如下:

```

POUNDFRANCDI
MAX_STRING = RAN

```

函数 MAX_STRING 的功能是求数组中元素的最大字符串。但由于实元和虚元的元素长度不同,因此所求出的最大字符串并非是实元数组 S 的元素的最大的字符串。结合图 8-1 来读本程序,不难看出函数的返回值是怎样产生的。

8.5.3 字符型数据作为子例过程变元

函数子程序主要是通过函数 RESULT 后边的变量名返回函数值。但是,如果某些虚元的 INTENT 属性的值为“OUT”或“INOUT”,这些虚元的值在函数中被修改时,与之对应的实元也会跟着发生变化。因此从这个意义上来说,函数子程序也可以带回多个值。但函数子程序的主要功能是返回函数值,所以又把函数子程序通过虚实结合带回值的过程叫做函数子程序的副作用(或称辅作用)。

子例子程序的虚元也可以是字符型变量和数组,其使用方法与函数子程序相似。由于子副作用问题。

[例 8-9] 编一子例子程序,将一字符数组中各元素连接成一个长串。

```

PROGRAM LINKER
    IMPLICIT NONE
    INTERFACE
        SUBROUTINE LINK_STRING(STRING,STR_RES,N)
            INTEGER,INTENT(IN)::N
            CHARACTER(LEN=*),DIMENSION(:),INTENT(IN)::STRING
            CHARACTER(LEN=*),INTENT(OUT)::STR_RES

```

```

        END SUBROUTINE LINK_STRING
    END INTERFACE
    CHARACTER(LEN=15),DIMENSION(1:4)::SS
    CHARACTER(LEN=90)::SS_RES
    READ *,SS
    CALL LINK_STRING(SS,SS_RES,4)
    PRINT *,TRIM(SS_RES)
END PROGRAM LINKER
SUBROUTINE LINK_STRING(String,STR_RES,N)
    IMPLICIT NONE
    INTEGER,INTENT(IN)::N
    CHARACTER(LEN=*),DIMENSION(:),INTENT(IN)::STRING
    CHARACTER(LEN=*),INTENT(OUT)::STR_RES
    INTEGER::I
    STR_RES=TRIM(ADJUSTL(String(1)))
    DO I=2,N
        STR_RES=TRIM(STR_RES)//" "//TRIM(ADJUSTL(String(I)))
    END DO
END SUBROUTINE LINK_STRING

```

程序运行结果如下：

'I am a ',' Chinese. ',' I love my ',' motherland.' ✓

I am a Chinese. I love my motherland.

子程序中使用了星号说明字符型虚元的长度,因而它实际的长度就是相对应的实元的长度。在字符串连接时使用 TRIM(ADJUSTL(String(I)))是为了把数组元素首部和尾部的空格全部去掉。

8.6 程序举例

[例 8-10] 判断一个数的符号。对于正数、负数和零分别给出“Positive”、“Negative”和“Zero”的信息。

```

PROGRAM TEST_SIGN
    IMPLICIT NONE
    INTEGER::NUMBER
    CHARACTER(LEN=8)::SIGN
    READ *,NUMBER
    IF(NUMBER>0)THEN
        SIGN='Positive'
    ELSE IF(NUMBER<0)THEN
        SIGN='Negative'
    ELSE
        SIGN='Zero'
    END IF

```

```

ELSE
    SIGN = 'Zero'
END IF
PRINT *, NUMBER, " is ", SIGN
END PROGRAM TEST SIGN

```

在程序运行时输入

- 23 ✓

后,其输出为:

- 23 is Negative

[例 8-11] 有一篇英文文章每行不超过 40 个字符,共 N 行。编程序统计出文章中英文单词的个数和空格的个数。为了简化问题,若干个连续的非空格字符即被认为是一个单词。

```

PROGRAM WORD BLANK NUM
    IMPLICIT NONE
    CHARACTER(LEN=40)::LINE
    INTEGER::LABEL,NW,NB,N,I,J,K
    NW=0; NB=0
    PRINT *, 'Please input line number:'
    READ *, N
    DO I=1,N
        PRINT *, 'Please input string ', I
        READ '(A)', LINE
        LABEL=0; K=LEN_TRIM(LINE)
        DO J=1,K
            IF(LINE(J:J) == ' ') THEN
                LABEL=0; NB=NB+1
            ELSE IF(LABEL == 0) THEN
                LABEL=1; NW=NW+1
            END IF
        END DO
    END DO
    PRINT " ('WORDS=' , I4, 5X, 'BLANKS=' , I4) ", NW, NB
END PROGRAM WORD BLANK NUM

```

程序运行情况如下:

Please input line number:

3 ✓

Please input string 1

I COME FROM CHINA ✓

Please input string 2

MY NAME IS ZHANG✓

Please input string 3

I AM 30✓

WORDS= 11 BLANKS= 8

[例 8-12] 在同一个坐标系上打印 $\sin x$ 和 $\cos x$ 两条曲线。要求输出 0° 到 180° 的曲线, 每隔 10° 输出一个点。正弦曲线的轨迹用“.”表示, 余弦曲线的轨迹用“*”表示。由于 $\sin x$ 和 $\cos x$ 的值域是 $[-1, 1]$, 为了清楚表示其轨迹必须对其放大一定倍数, 并右移一定的位置。

```
PROGRAM SIN_COS_DRAW
  IMPLICIT NONE
  INTEGER::I, IS, IC
  REAL::R
  CHARACTER(LEN=60)::LINE
  R=3.1415926/180
  DO I=0,180,10
    LINE=' '
    IS=SIN(R*I)*25+30.5
    IC=COS(R*I)*25+30.5
    LINE(IS:IS)='.'
    LINE(IC:IC)='*'
    LINE(30:30)='|'
    PRINT *,LINE
  END DO
END PROGRAM SIN_COS_DRAW
```

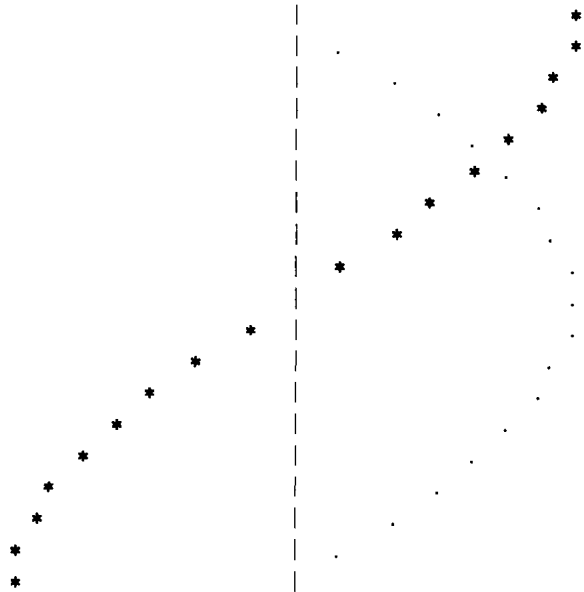


图 8-2 例 8-12 输出结果

对本例稍作修改就可以输出 0° 到 360° 的曲线, 读者可以自己完成。

[例 8-13] 验证一字符串是否为“回文”。所谓字符串回文是指一个字符串从左向右读和从右向左读是一样的。例如单词“radar”是一个回文串, “1991”也是一个回文串, 但“AB-CD CBA”和“ABA”(假设尾部空格不算字符串的一部分)都不是回文的。

```
PROGRAM HUIWEN
  IMPLICIT NONE
  CHARACTER(LEN=80)::LINE
  INTEGER::I, J
  LOGICAL L
  READ "(A)", LINE
  J=LEN_TRIM(LINE)
  L=.TRUE.
  DO I=1, J/2
```

```

        IF(LINE(I:I)/=LINE(J:J))THEN
            L=.FALSE.
            EXIT
        ELSE
            J=J-1
        END IF
    END DO
    IF(L)THEN
        PRINT *,TRIM(LINE),' is a HUIWEN.'
    ELSE
        PRINT *,TRIM(LINE),' is not a HUIWEN.'
        PRINT *,"Character",I,"from the left is ",&
            CHAR_OR_BLANK(LINE(I:I))
        PRINT *,"Character",I,"from the right is ",&
            CHAR_OR_BLANK(LINE(J:J))
    END IF
CONTAINS
    FUNCTION CHAR_OR_BLANK(C) RESULT(CB_RESULT)
        CHARACTER(LEN=5)::CB_RESULT
        CHARACTER(LEN=*),INTENT(IN)::C
        IF(C==' ' )THEN
            CB_RESULT="Blank"
        ELSE
            CB_RESULT=C
        END IF
    END FUNCTION CHAR_OR_BLANK
END PROGRAM HUIWEN

```

习题 8

8-1 已知有下面的说明语句

```
CHARACTER(LEN=2)::A*3,B,C*3,D
```

现在希望把(HOW、(DO、(YOU、(DO(分别放入 A、B、C、D 四个字符型变量中。对于下面的各输入语句,应如何输入数据?

- (1)READ(*,'(A3,A2,A3,A2)')A,B,C,D
- (2)READ(*,'(A5,A4,A3,A2)')A,B,C,D
- (3)READ(*,'(A,A,A,A)')A,B,C,D
- (4)READ(*,'(A3,1X,A2,1X,A3,1X,A2)')A,B,C,D

8-2 若变量 A,B,C,D 说明与 8-1 题相同,并已分别存放了 'HOW'、'DO'、'YOU'、

'DO',按以下各输出语句,输出结果是怎样的?

- (1)PRINT '(1X,A3,A2,A3,A2)',A,B,C,D
- (2)PRINT '(1X,A3,2X,A2,2X,A3,2X,A2)',A,B,C,D
- (3)PRINT '(A3,A3,A4,A3)',A,B,C,D
- (4)PRINT '(1X,4A)',A,B,C,D

8-3 编程序输出下面的图案(程序中不能使用输入语句):

(1)	(2)	(3)	(4)
* * * * *	*	*	* * * * *
* * * * *	* * *	* *	* *
* * * * *	* * * * *	* *	*
* * * * *	* * *	* *	* *
* * * * *	*	*	* * * * *

8-4 按 ASCII 代码字符序列,写出下面字符关系表达式的值。

- (1)'CHINA'.EQ.'china' (2)'THESE'.LT.'THOSE'
- (3)'8088'.GT.'80386' (4)'ABC'.GT.'ABC '
- (5)'ABC'.LE.' ABC' (6)'ABC'.LE.'ABC '

8-5 有一篇文章,要求统计出字符串 'THE' 在文章中出现的次数,并指出它们分别从第几行第几个字符位置开始。

8-6 把 25 个字母 A,B,C,...,X,Y 按如下要求打印出来:从最中间的字母 M 开始打印,然后一左一右依次打印出其它字母,即 MLNKO...

要求:不许使用 READ 语句。

8-7 将一个字符串中的前导空格、后置空格及中间的空格全部去掉后,得到一个新的字符串后输出。

8-8 字母翻译,将一行信息中出现的每个大写字母翻译成字母表中其后继第 4 个字母。若译后的字母超过了大写字母的界限,则回到“A”继续计算后继次序,即 A→E、B→F、C→G、...、V→Z、W→A、X→B、Y→C、Z→D。

8-9 编一个子程序用以将一个字符串插入到另一个字符串中,从第 N 个位置开始插入。(分别用函数子程序和子例子程序实现)

8-10 有一批图书。每本书有:书名(NAME)、作者(AUTHOR)、编号(NUM)、出版日期 DATE)4 项数据,希望输入 N 本书的 4 项数据后按书名的字母顺序将各书按从小到大的次序排列好。然后输入一本书的书名,进行查找。如果查到库中有此书,输出此书的书名、作者、编号和出版日期;如果库中无此书,则输出"NO THIS BOOK!"。

8-11 在同一坐标系中打印 $\sin x$ 和 $\cos(3x+1)$ 两条曲线。

8-12 编写两个子程序及主程序,其功能分别为:

子程序 1:求出字符串 STRING1 中字母、数字出现的次数各是多少?

子程序 2:求出字符串 STRING2 中出现次数最多的字符及其出现次数。

主程序:输入 K 和字符串 A(A 的最大长度为 30 个字符)。主程序中首先求出字符串 A 的有效长度,然后输入 K 的值,当 K=1 时调用子程序 1,当 K=2 时调用子程序 2。

要求子程序中不能有输出语句。

第 9 章 派生类型和指针结构

用高级语言编写的程序一般是对一些数据进行操作之后再产生另外一些数据。这些数据可以是 FORTRAN 系统内部固有的各种类型: INTEGER 型、REAL 型、LOGICAL 型、CHARACTER 型等等,有时数据比较复杂,不能用上述内部类型表示,这需由内部类型导出其它的类型,这就是派生类型。派生类型是 FORTRAN90 新增的主要内容之一。由于派生类型的出现,指针类型才有实际意义,才会引出对二叉树,队列等数据结构问题的讨论。

本章所涉及的知识是新型高级语言的主要特征之一,在 Pascal 和 C 语言中都有与本章内容相似的章节。学好派生类型与指针,对掌握复杂的数据结构问题是十分必要的。

9.1 数据结构与派生类型

派生类型是 FORTRAN90 处理复杂数据问题的有效方法,它与数据结构知识的关系十分密切。

9.1.1 数据结构

数据结构指的是数据的组织方式,即若干个数据之间的连接方式。例如一个二维坐标点的坐标可用两个实型变量 X 和 Y 来描述, X 和 Y 之间就 FORTRAN 语言本身并不预示着它们之间有什么联系,只是程序员把这两个变量理解为一个点的 X 轴和 Y 轴的坐标。如果我们换一种方法来表述二维点的坐标,把两个实型数据(假设仍表示二维点的 X 轴和 Y 轴坐标)结合在一起,形成一个新的数据类型 COORDINATE,那么只要说明一个 COORDINATE 型的变量就可以表示二维点的两个坐标值。这个 COORDINATE 就是一个派生类型,该类型下有两个实型数据,而且这两个实型数据是有关系的,可以表示二维点的两个坐标,也可表示一对兄弟的体重或一个职工的月工资和津贴(假如有这种需要的话)。可以看到这两个实型数据放在一起时,就具有了某种关系,这种关系正表明它是一种简单的数据结构。

第七章介绍过数组,实际上数组也是一种数据结构。设有以下说明语句

```
REAL, DIMENSION(1:100)::A
```

它说明数组 A 中有 100 个实型的元素,数组的引用或者是给出下标来确定某一个元素,或者是给出数组名代表数组的全部元素,或者是通过数组名及相应的三元组表达式代表数组片段。数组是我们已经使用过、但尚未明确指出的一种数据结构。

在实际应用中,并不都像上述几种数据结构中的数据那样都有相同的类型。以学生成绩的表述为例,应当包含学生姓名(字符型)、学号(整型)和考试成绩(实型)等信息。假如共有 30 名学生,表示他们的学习成绩需要有三个数组:一个字符型数组用于存放 30 个学生的姓名,一个整型数组用于存放相应学生的学号,一个实型数组用于存放相应学生的考试成绩。如果把它们按成绩排序的话,这三个数组都要同时调整次序,操作上十分不便。这时我

们可以定义一个派生类型,它由一个字符型数据、一个整型数据和一个实型数据组成,然后说明一个具有这种派生类型为元素的数组,这样从数据的组织上十分方便。

另外,有的数据结构是层次状的。以表示教师信息的结构为例,它可以分成职工号、工作单位、姓名、性别、职称、工资和住址等多项,而工作单位又可细分为系和教研室两项,工资又可分成基本工资、工龄工资和岗位工资三项,这种数据的结构如图 9-1 所示。

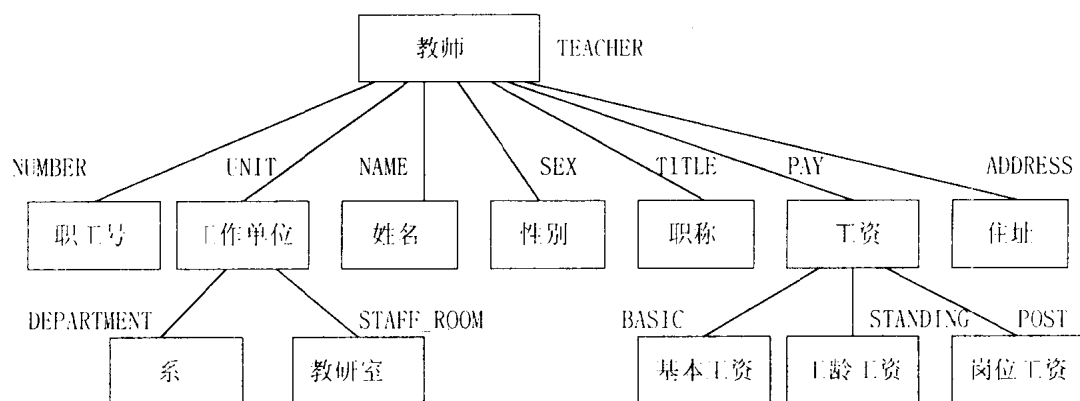


图 9-1 教师数据结构图

这里对数据结构的探讨只是示范性的,我们不需要了解关于它的详细准确的定义,而只是为派生类型的引出做一点准备工作。实际上高级语言的功能之一就是给出各种自然数据抽象的定义方法,也就是给出对各种数据的描述方法,然后再编写出可执行语句序列对这些数据进行相应的操作(算法的计算机表示),如排序、迭代、矩阵运算、建立链表等等。算法和数据结构的有机结合,就可以形成程序。

9.1.2 派生类型的定义

整型、实型、逻辑型、字符型、数组类型等都是 FORTRAN 语言系统内部固有的数据类型,我们只需要使用它们就可以了。然而对于较复杂的实际问题一般也会有比较复杂的数据结构,虽然直接用上述内部类型也可以表示,但或者是操作不便,或者是不易理解数据之间的相互关系,需要寻找较好的表示方法。在 FORTRAN90 中将复杂的数据结构分解成较简单的成员时,可用派生类型反映它。

定义派生类型的一般形式为:

```

TYPE [,存取方式描述:]派生类型名
  成员组 1 类型说明
  成员组 2 类型说明
  ...
  成员组 n 类型说明
END TYPE [派生类型名]
  
```

上述一般形式中,TYPE 和 ENDTYPE 是关键词,分别表示派生类型定义的开始和结束,派生类型名是必须有的、由程序员给出的名字,它必须符合 FORTRAN 的命名规则。为

了明确表示是哪一个派生类型的定义结束,可在 END TYPE 之后加上派生类型名(但不可写错,需要与 TYPE 后边的派生类型名一致)。用这种方法定义的派生类型就象 INTEGER、REAL 等类型一样,可用于说明新的变量、数组等,但具有这种派类型的变量、数组所能做的操作与派生类型有关。存取方式描述是关键字 PRIVATE 等,PRIVATE 表示该类型是专用的,只有 TYPE 块写在模块说明部分时才选用。

TYPE 与 END TYPE 之间的成员类型说明是派生类型组成成员的描述,这些成员的类型可以是以前用过的 FORTRAN 内部类型,如整型、实型、逻辑型、数组类型等,也可以是一个较低层次的派生类型,还可以是本章以后介绍的指针类型等。

实际上,一种派生类型反映了实际应用中的一种数据结构。也可以说,结构化强的数据可用派生类型描述。

9.1.1 节所述的表示二维坐标的类型 COORDINATE 可以定义如下:

```
TYPE COORDINATE
  REAL::X,Y
END TYPE COORDINATE
```

这种派生类型的定义描述了实际应用中的一种二维坐标结构。有了这种类型之后,可以利用它说明 COORDINATE 型的变量,方法如下:

```
TYPE(COORDINATE)::COOR
```

这样,COOR 就是 COORDINATE 这种派生类型的变量,可用 COOR%X 表示其中一个成员(X 轴值),用 COOR%Y 表示另一个成员(Y 轴值)。

用派生类型的说明语句说明变量的一般形式是

```
TYPE(派生类型名)::变量名[,...]
```

9.1.1 节谈到的学生一门课程成绩结构的派生类型可以定义如下:

```
TYPE SCORE_TYPE
  CHARACTER(LEN=10)::NAME      ! 学生姓名
  INTEGER::NUMBER              ! 学号
  REAL::SCORE                  ! 考试成绩
END TYPE SCORE_TYPE
```

如果有 30 名学生的话,可说明一个具有 SCORE_TYPE 型的一维数组来存放学生成绩等信息:

```
TYPE(SCORE_TYPE),DIMENSION(1:30)::STUDENT_SCORE
```

数组 STUDENT_SCORE 的每一个元素存放一名学生的姓名、学号及成绩。

对于图 9-1 的教师(TEACHER)结构,可先定义派生类型 UNIT_TYPE 和 PAY_TYPE:

```
TYPE UNIT_TYPE
  CHARACTER(LEN=12)::DEPARTMENT,STAFF_ROOM
END TYPE UNIT_TYPE
TYPE PAY_TYPE
  REAL::BASIC,STANDING,POST
END TYPE PAY_TYPE
```

工作单位结构对应的派生类型为 UNIT_TYPE,它有两个长度为 12 的字符型成员 DEPARTMENT 和 STAFF_ROOM,分别表示系名和教研室名。工资结构对应的派生类型为

PAY_TYPE 它有三个 REAL 型成员 BASIC、STANDING 和 POST,分别表示基本工资、工龄工资和岗位工资。这样,可以在派生类型 UNIT_TYPE 和 PAY_TYPE 之上构造更高层的派生类型 TEACHER:

```
TYPE TEACHER
  INTEGER::NUMBER
  TYPE(UNIT_TYPE)::UNIT
  CHARACTER(LEN=8)::NAME
  CHARACTER::SEX
  CHARACTER(LEN=15)::TITLE
  TYPE(PAY_TYPE)::PAY
  CHARACTER(LEN=20)::ADDRESS
END TYPE TEACHER
```

说明 EXAM_TEACHER 为 TEACHER 型的变量的方法如下

```
TYPE(TEACHER)::EXAM_TEACHER
```

可以用 EXAM_TEACHER%NUMBER 表示职工号,用 EXAM_TEACHER%UNIT%DEPARTMENT 表示系名等。定义派生类型时应注意以下几点:

1° TYPE 和 ENDTYPE 之间只能有关于结构成员的类型说明语句,不允许有可执行的动作语句。

2° 派生类型成员为字符型时,其长度必须确定,不可用*。

3° 为了便于记忆,派生类型名往往用 _TYPE 作为后缀,如 UNIT_TYPE 等。

4° 派生类型又可称为导出类型。

5° 派生类型的成员还可以是数组或以后将要介绍的指针等。如:

```
TYPE LINE_TYPE                                ! LINE_TYPE 表示直线型
  REAL,DIMENSION(2,2)::COORD                  ! COORD 为直线两端点的坐标
  REAL                                     ::WIDTH    ! WIDTH 表示线的宽度
  INTEGER                                 ::PATTERN    ! PATTERN 表示线型编号
END TYPE LINE_TYPE
```

6° 结构构造函数

FORTRAN90 中,每定义一个派生类型,就建立了一个结构构造函数。以前边定义的派生类型 SCORE_TYPE 为例,系统会自动生成一个名叫 SCORE_TYPE 的结构函数,它有三个虚元自变量 NAME、NUMBER 和 SCORE。假设有变量说明

```
TYPE(SCORE_TYPE)::STU_SCORE
```

那么下边语句就是正确的

```
STU_SCORE = SCORE_TYPE('ZHANG',105,95.5)
```

它将 'ZHANG' 作为 NAME 的实元,105 作为 NUMBER 的实元,95.5 作为 SCORE 的实元,调用结构构造函数 SCORE_TYPE,并把函数值赋给变量 STU_SCORE。

9.1.3 派生类型的简单应用

派生类型的应用是很广泛的。下面通过几个例子介绍派生类型的简单应用,更进一步的应用在以后各节中介绍。

[例 9-1] 输入平面直角坐标系上两个点的坐标值,然后求出它们两点之间的距离。

根据解析几何的知识,平面上任意两点 (x_1, y_1) 和 (x_2, y_2) 之间的距离为:

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

解题程序如下:

```
PROGRAM DISTANCE
  IMPLICIT NONE
  TYPE COORDINATE
    REAL::X, Y
  END TYPE COORDINATE
  TYPE(COORDINATE)::COOR1, COOR2
  REAL::DIST
  READ '(2F6.1)', COOR1, COOR2
  PRINT '(4F8.2)', COOR1, COOR2
  DIST = SQRT((COOR2%X - COOR1%X) * (COOR2%X - COOR1%X) &
    + (COOR2%Y - COOR1%Y) * (COOR2%Y - COOR1%Y))
  PRINT *, 'DIST = ', DIST
END PROGRAM DISTANCE
```

假设两点坐标为(1.1,2.0)和(3.1,4.0),运行结果如下:

```
1.1  2.0 ✓
3.1  4.0 ✓
1.10  2.00  3.10  4.00
DIST =  2.8284271
```

程序中 COOR1 和 COOR2 均为派生类型 COORDINATE 类型的变量,分别用于存放两点的坐标值。DIST 为实型变量,用于存放两点间的距离。由于采用 2F6.1 格式输入数据,每行只能输入两个实型数据(代表一个点的坐标)。求 DIST 值的语句也可写成:

```
DIST = SQRT((COOR2%X - COOR1%X) * * 2 + (COOR2%Y - &
  COOR1%Y) * * 2)
```

[例 9-2] 假设某班有 6 个学生,每个学生的信息包括姓名、学号和一门课程的考试成绩。请将学生的信息按学习成绩从大至小排序。

在程序开始时,先说明符号常数 N 为 6,这样学生人数发生变化时只需改变 N 的值即可。STU_SCORE 是 SCORE_TYPE 型(见 9.1.2)、具有 N 个元素的一维数组,即 STU_SCORE 的每个元素都具有 SCORE_TYPE 类型,它有三个成员:NAME、NUMBER 和 SCORE。程序大致可分成四个部分:①类型和变量等的说明;②输入数据到数组 STU_SCORE 中;③对数组 STU_SCORE 进行排序;④将排好序的数组 STU_SCORE 输出。程序如下:

```
PROGRAM SCORE_SORT
```

```

IMPLICIT NONE
INTEGER,PARAMETER::N=6,LENGTH=10
TYPE SCORE_ TYPE
    CHARACTER(LEN=LENGTH)::NAME
    INTEGER::NUMBER
    REAL::SCORE
END TYPE SCORE_ TYPE
TYPE(SCORE_ TYPE),DIMENSION(1:N)::STU_ SCORE
INTEGER::I,J,TINT
CHARACTER(LEN=LENGTH)::TCHAR
REAL::TREAL
PRINT *, 'INPUT DATA:'
DO I=1,N
    READ '(A,I4,F5.1)',STU_ SCORE(I)
END DO
DO I=1,N-1
    DO J=I+1,N
        IF(STU_ SCORE(I)%SCORE<STU_ SCORE(J)%SCORE)THEN
            TCHAR=STU_ SCORE(I)%NAME
            STU_ SCORE(I)%NAME=STU_ SCORE(J)%NAME
            STU_ SCORE(J)%NAME=TCHAR
            TINT=STU_ SCORE(I)%NUMBER
            STU_ SCORE(I)%NUMBER=STU_ SCORE(J)%NUMBER
            STU_ SCORE(J)%NUMBER=TINT
            TREAL=STU_ SCORE(I)%SCORE
            STU_ SCORE(I)%SCORE=STU_ SCORE(J)%SCORE
            STU_ SCORE(J)%SCORE=TREAL
        END IF
    END DO
END DO
PRINT *, 'SORTED DATA:'
DO I=1,N
    WRITE(*, '(A,I5,F6.1)')(STU_ SCORE(I)
END DO
END PROGRAM SCORE_ SORT

```

运行结果为:

INPUT DATA:

ZHANG HONG 101 78.5✓

```

MA LIN      102 94.0✓
ZHAO YI     103 99.5✓
WANG PING   104 70.0✓
WU BING     105 100.0✓
LI MING     106 80.0✓

```

SORTED DATA:

```

WU BING     105 100.0
ZHAO YI     103 99.5
MA LIN      102 94.0
LI MING     106 80.0
ZHANG HONG  101 78.5
WANG PING   104 70.0

```

由于 FORTRAN90 中可对派生类型变量(或数组元素)直接用赋值语句赋值,故上述程序中二重 DO 循环部分可改用比较简单的方式实现:

```

DO I=1,N-1
  DO J=I+1,N
    IF(STU_SCORE(I)%SCORE<STU_SCORE(J)%SCORE)THEN
      T_STU=STU_SCORE(I)    ! T_STU 应在前边被说明为 SCORE_TYPE 型变量
      STU_SCORE(I)=STU_SCORE(J)
      STU_SCORE(J)=T_STU
    END IF
  END DO
END DO

```

在本例中,由于采用格式输入,STU_SCORE(I)%NAME 对应的格式为 A,而成员 NAME 的字符长度为 10。故 NAME 数据应恰好占 10 个字符(姓名不足 10 个字符时右补空格;长于 10 个时应截去多余的字符)。STU_SCORE(I)%NUMBER 需要 4 位整数,STU_SCORE 用 F5.1 格式,需要 5 位实数。如果输入数据长短与实际不对应,或者会出现无效数据错,或者输入的结果与期望的不一致,最终导致运行结果失败。

如果学生数是一个变量(假设用变量 N 表示),这时可以将数组 STU_SCORE 说明得大一些(例如有 100 个元素)。学生数 N 由键盘输入,但 N 值超过 100 时程序应输出错误信息后结束。

[例 9-3] 假设我们只关心教师的职工号、工作单位、姓名及工资等信息(9.1.2 中 TEACHER 类型的简化形式),说明一个教师结构类型的变量,输入一个教师的全部数据,并求出它的工资总额。

该程序如下:

```

PROGRAM PAY_SUMMATION
  IMPLICIT NONE
  TYPE UNIT_TYPE
    CHARACTER(LEN=12)::DEPARTMENT,STAFF_ROOM
  END TYPE UNIT_TYPE

```

```

TYPE PAY_ TYPE
  REAL::BASIC,STANDING,POST
END TYPE PAY_ TYPE
TYPE TEACHER
  INTEGER::NUMBER
  TYPE(UNIT_ TYPE)::UNIT
  CHARACTER(LEN=8)::NAME
  TYPE(PAY_ TYPE)::PAY
END TYPE TEACHER
TYPE(TEACHER)::EXAM_ TEACHER
REAL::SUM
READ *,EXAM_ TEACHER
SUM= EXAM_ TEACHER% PAY% BASIC + EXAM_ TEACHER% PAY% &
  STANDING + EXAM_ TEACHER% PAY% POST
PRINT '(I4,A13,A13,A9,3F7.2)',EXAM_ TEACHER
PRINT '(A,F8.2)', 'SUM= ',SUM
END PROGRAM PAY_ SUMMATION

```

程序运行结果如下：

```

1001,'COMPUTER','SOFTWARE','LIU FANG',350.0,40.0,276.5✓
1001 COMPUTER    SOFTWARE    LIU FANG 350.00  40.00 276.50
SUM=  666.50

```

由于程序中向 EXAM_ TEACHER 中读入数据时用的是表控格式,故各成员数据之间用逗号分隔,字符型数据需要用单撇号(或双撇号)括起来,如 'COMPUTER' 等。

9.1.2 中曾讲述过,FORTTRAN90 中每定义一个派生类型就建立了一个结构构造函数。为了加强对结构构造函数的理解,例 9-4 给出了使用这种函数的一个例子。

[例 9-4] 结构构造函数举例。

```

PROGRAM STRUCTURE_ CONSTRUCTOR
  IMPLICIT NONE
  INTEGER,PARAMETER::LENGTH=10
  TYPE SCORE_ TYPE
    CHARACTER(LEN=LENGTH)::NAME
    INTEGER::NUMBER
    REAL::SCORE
  END TYPE SCORE_ TYPE
  TYPE(SCORE_ TYPE)::STU_ STRUCT
  STU_ STRUCT=SCORE_ TYPE('ZHANG HONG',101,87.5)
  WRITE(*, '(A,A,A,I5,A,F6.1)') 'NAME= ',STU_ STRUCT%NAME, &
    ' NUMBER= ',STU_ STRUCT%NUMBER, ' SCORE= ',&
    STU_ STRUCT%SCORE

```

END PROGRAM STRUCTURE_CONSTRUCTOR

程序运行结果如下:

NAME= ZHANG HONG NUMBER= 101 SCORE= 87.5

利用结构构造函数 SCORE_TYPE 给 SCORE_TYPE 类型的变量赋值很方便。如:

STU_STRUCT= SCORE_TYPE('ZHANG HONG',101,87.5)

否则必须由以下三个语句才能完成上述赋值功能:

STU_STRUCT%NAME= 'ZHANG HONG'

STU_STRUCT%NUMBER= 101

STU_STRUCT%SCORE= 87.5

9.2 指 针

指针是 FORTRAN90 引入的新概念。通过指针的学习,可以较好地理解通过指针访问数据的方法、动态数据存储概念及其操作方法等问题。

9.2.1 指针变量的基本概念

我们在第 7 章中学习过数组。数组的特点之一就是同一数组中各个元素的类型都是相同的,这些元素在内存中都占具同样大小的存储空间。由于数组元素按下标从小到大在内存中连续存放,这样当数组元素类型及种别参数一定的情况下(元素大小成为已知量),如果再知道数组的起始地址,系统根据下标值就很容易计算出该元素在内存中的起始地址。因而数组用于存储大量同类型数据时,有着元素寻址快等特点。但数组本身也有许多缺点,如在数组中插入或删除一个数据时就不太方便,效率也较低。另外,如果数组元素较多,实际数据项较少时,也存在浪费内存空间的问题。指针在处理数据插入、删除这类操作时非常方便,还可以利用指针形成比较复杂的数据结构,如链表、树等。

指针变量简称指针。说明指针变量时需要指出指针变量的类型,如实型、整型等,另外还要说明它具有 POINTER 属性。例如

REAL, POINTER::PT1, PT2

就说明了两个实型指针变量 PT1 和 PT2。这两个指针变量为实型,可以指向实型的目标变量。目标变量简称目标,它是指针变量指向的对象,它要具有 TARGET 属性。例如,可用以下语句说明 X 和 Y 都是实型目标变量:

REAL, TARGET::X, Y

由于目标变量 X、Y 和指针变量 PT1、PT2 类型相同(都是实型),因而 PT1、PT2 可以指向 X、Y。这种指向关系需要通过指针赋值语句来建立。例如,如果想让 PT1 指向 X, PT2 指向 Y,可以通过以下两个指针赋值语句实现:

PT1= >X

PT2= >Y

此时指针与目标的关系如图 9-2 所示。

这种指向关系一旦建立以后,使用 X 和使用它的指针 PT1 的作用是相同的,使用 Y 和使用它的指针 PT2 的作用也是相同的,即 X 和 PT1 具有同样的值, Y 和 PT2 具有同样的值

(这与其它高级语言指针的概念有所不同)。例如,如果执行语句序列:

$X = -32.33$; $PT2 = 20.22$

那么 $PT1$ 的值也是 -32.33 , Y 的值就是 20.22 。

指针赋值语句的赋值号是 $=>$, 由字符 '=' 与字符 '>' 合成, 读作“指向”。它

只能用于给指针变量赋指针值, 即使指针变量与目标变量之间建立指向关系。

[例 9-5] 指针应用的简单例子。

```
PROGRAM EXAM_POINTER
  IMPLICIT NONE
  REAL, TARGET::X, Y
  REAL, POINTER::PT1, PT2
  X = 10.11; Y = 20.22
  PT1 = >X; PT2 = >Y
  PRINT *, 'X=', X, ' Y=', Y
  PRINT *, 'PT1=', PT1, ' PT2=', PT2
  X = -30.33; Y = -40.44
  PRINT *, 'PT1=', PT1, ' PT2=', PT2
END
```

运行结果如下:

```
X= 10.1099997 Y= 20.2199993
PT1= 10.1099997 PT2= 20.2199993
PT1= -30.3299999 PT2= -40.4399986
```

在上例中先给目标变量分别赋值 10.10 和 20.22。然后通过指针赋值语句 $PT1 = >X$ 和 $PT2 = >Y$ 让指针变量 $PT1$ 指向目标 X , 让指针变量 $PT2$ 指向目标 Y 。这时输出 X 和输出 $PT1$ 的值是相同的。由于实数在计算机中只能近似存储, 故 X 和 $PT1$ 的输出结果是 10.1099997, Y 和 $PT2$ 的输出结果都是 20.2199993。当再用语句 $X = -30.33$ 和 $Y = -40.44$ 修改 X 和 Y 中的值时, $PT1$ 和 $PT2$ 的值也都相应改变了。

目前使用指针时应注意以下几个问题:

1° 如果在图 9-2 状态下, 执行指针赋值语句 $PT1 = >PT2$ 表示 $PT1$ 与目标 X 的关系自动解除, $PT1$ 指向 $PT2$ 所指向的目标, 即 $PT1$ 和 $PT2$ 都指向 Y (见图 9-3)。如果在图 9-2 状况下, 执行赋值语句 $PT1 = PT2$, 其作用是将 $PT2$ 所指向的目标的值赋给 $PT1$ 所指向的目标。这时

X 中的值与 Y 的值相同。如果原来 X 中的值是

10.11, Y 中的值是 20.22, 执行 $PT1 = PT2$ 之后, 关系图如图 9-4(a) 所示; 若执行的赋值语句是 $PT2 = PT1$, 关系同如图 9-4(b) 所示。

2° 可以有多个指针同时指向同一个目标, 但不允许一个指针同时指向多个目标。如图

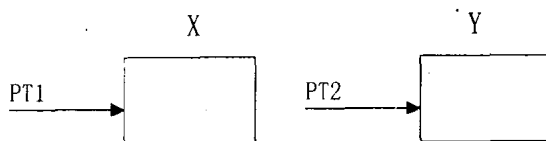


图 9-2 指针与目标关系图

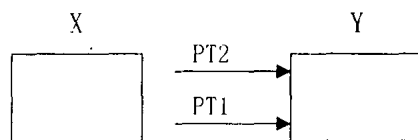


图 9-3 指针 $PT1$ 和 $PT2$ 指向同一个目标 Y

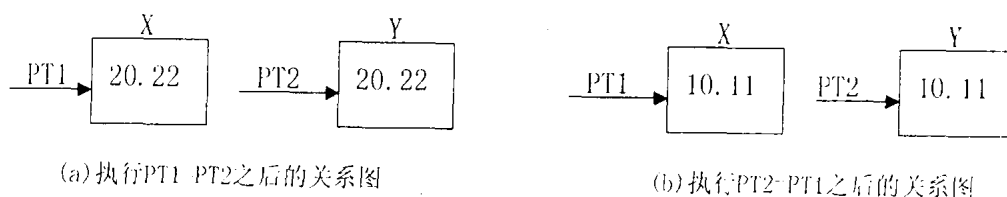


图 9-4 含指针的一般赋值语句

9-3 中, 指针 PT1 和 PT2 都指向目标 Y 是允许的, 而一个指针同时指向多个目标是不可能的。因为给指针和目标建立指向关系的操作需要使用指针赋值语句, 而这种指针赋值语句仅保留指针最后一个指向关系。如指针 PT1 原来指向 X, 再进行指针赋值 $PT1 = > Y$ 时, 是先将 PT1 与目标 X 的指向关系断开之后才又让 PT1 指向目标 Y。故 PT1 不可能同时指向多个目标。这与一个简单变量中不能同时存放多个值是同一个道理。

3° 指针不但可以指向实型、整型目标, 还可以指向字符型和逻辑型目标。例如:

```
CHARACTER(LEN=10), TARGET::CH
CHARACTER(LEN=10), POINTER::PCH
LOGICAL, TARGET::LG
LOGICAL, POINTER::PLG
PCH = > CH
PLG = > LG
```

这样 PCH 指向 CH, PLG 指向 LG。一般情况下, 使用 PCH 和 CH 的作用是相同的; 使用 PLG 就相当于使用了 LG。

9.2.2 指针指向派生类型

指针除了可以指向像实型这样的简单类型目标以外, 更重要的作用是指向派生类型。

[例 9-6] 指针指向派生类型目标举例。

```
PROGRAM POINTER_DERIVED
  IMPLICIT NONE
  INTEGER, PARAMETER::LENGTH=10
  TYPE SCORE_TYPE
    CHARACTER(LEN=LENGTH)::NAME
    INTEGER::NUMBER
    REAL::SCORE
  END TYPE SCORE_TYPE
  TYPE(SCORE_TYPE), TARGET::STU_S1, STU_S2
  TYPE(SCORE_TYPE), POINTER::PT_S1, PT_S2
  PT_S1 = > STU_S1
  PT_S2 = > STU_S2
  READ *, STU_S1, PT_S2
  PRINT *, STU_S1, STU_S2
```

```

PRINT *,PT_S1,PT_S2
PT_S2=PT_S1
PRINT *
PRINT *,STU_S1,STU_S2
PRINT *,PT_S1,PT_S2
END PROGRAM POINTER_DERIVED

```

程序中 STU_S1 和 STU_S2 都是 SCORE_TYPE 型的变量, PT_S1 和 PT_S2 是指向 SCORE_TYPE 型的指针。程序运行后如果输入以下数据:
 'ZHANG HONG',101,78.5
 'ZHAO YI',103,99.5
 指针与目标的关系如图 9-5 所示。

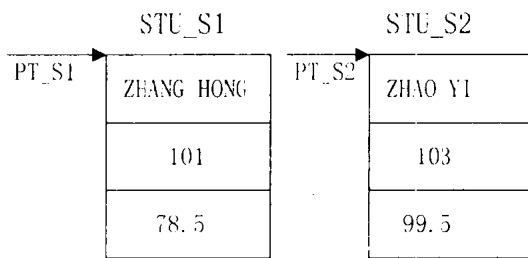


图 9-5 指针指向派生类型目标

语句 PT_S2 = PT_S1 执行完毕之后, PT_S2 所指向的目标 STU_S2 中的值被 PT_S1 中指向的目标 STU_S1 中的值所替换, 指向关系变成图 9-6(a) 的形式。因此, 程序的输出结果(输出结果不含输入部分)为:

```

ZHANG HONG 101 78.5000000 ZHAO YI      103 99.5000000
ZHANG HONG 101 78.5000000 ZHAO YI      103 99.5000000

```

```

ZHANG HONG 101 78.5000000 ZHANG HONG 101 78.5000000
ZHANG HONG 101 78.5000000 ZHANG HONG 101 78.5000000

```

如果将程序中含指针的一般赋值语句 PT_S2 = PT_S1 换成指针赋值语句 PT_S2 => PT_S1, STU_S2 中的值并未发生变化, 而指针 PT_S2 脱离了与 STU_S2 的指向关系, 与 PT_S1 一样也指向 STU_S1。这样指向关系如图 9-6(b) 所示。程序的输出结果也变为:

```

ZHANG HONG 101 78.5000000 ZHAO YI      103 99.5000000
ZHANG HONG 101 78.5000000 ZHAO YI      103 99.5000000

```

```

ZHANG HONG 101 78.5000000 ZHAO YI      103 99.5000000
ZHANG HONG 101 78.5000000 ZHANG HONG 101 78.5000000

```

9.2.3 动态控制

上节的例子中, 指针都是指向已经存在的一个目标。如例 9-5 中 PT1 是指向实型目标 X, 例 9-6 中的 PT_S1 指向 SCORE_TYPE 型的目标 STU_S1 等。实际上, 可以用 ALLOCATE 语句为指针变量分配一块存储空间, 并让该指针指向这块空间, 而且这块空间就以该指针变量命名, 这就是目标的动态控制问题。

目标的动态控制可由 ALLOCATE 语句、NULLIFY 语句、DEALLOCATE 语句和指针赋值语句等 LOCATE 用于把某个指针变量通过动态分配得到的存储空间归还给系统, 这称

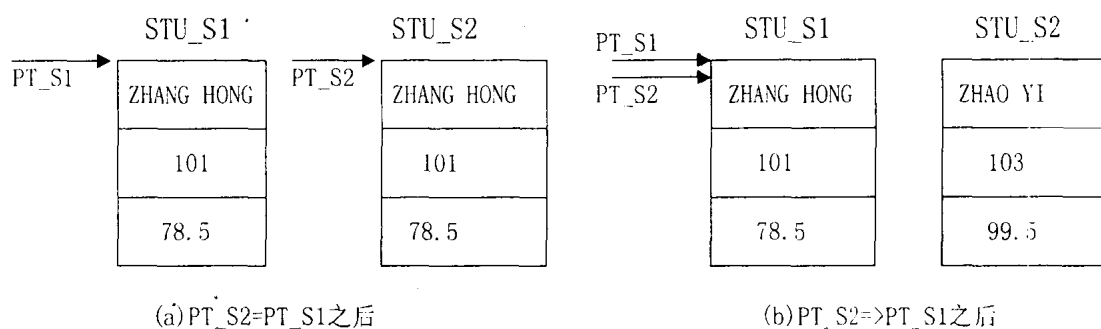


图 9-6 派生类型含指针的一般赋值语句和指针赋值语句的不同

作释放指针所指向的目标;NULLIFY 语句使指针变量指向空,即该指针变量为空指针;指针赋值语句可以使指针变量改变指向的目标。下边通过一个例子说明用于动态控制的前三个语句的用法。

[例 9-7] 目标的动态建立与释放以及指针置空举例。程序如下:

```
PROGRAM POINTER_STUDENT
  IMPLICIT NONE
  TYPE STUDENT_TYPE
    INTEGER::NUMBER
    REAL::SCORE
    TYPE(STUDENT_TYPE),POINTER::NEXT
  END TYPE STUDENT_TYPE
  TYPE(STUDENT_TYPE),POINTER::PT_STU1,PT_STU2
  ALLOCATE(PT_STU1)
  ALLOCATE(PT_STU2)
  READ *,PT_STU1%NUMBER,PT_STU1%SCORE
  READ *,PT_STU2%NUMBER,PT_STU2%SCORE
  NULLIFY(PT_STU1%NEXT)
  NULLIFY(PT_STU2%NEXT)
  PRINT *,PT_STU1%NUMBER,PT_STU1%SCORE
  PRINT *,PT_STU2%NUMBER,PT_STU2%SCORE
  DEALLOCATE(PT_STU1)
  DEALLOCATE(PT_STU2)
END PROGRAM POINTER_STUDENT
```

本例中 PT_STU1 和 PT_STU2 都是指向 STUDENT_TYPE 类型的指针,通过语句 ALLOCATE (PT_STU1)和 ALLOCATE(PT_STU2)建立了两个目标,其名字分别就叫 PT_STU1 和 PT_STU2。如果输入以下数据

101,78.5✓

103,99.5✓

之后,指针与目标的关系如图 9-7 所示。

这时 PT_STU1%NEXT 和 PT_STU2%NEXT 中均无值(无定义),称作指向待定,而这两个成员具有与 PT_STU1、PT_STU2 同样的类型,也是指向 STUDENT_TYPE 类型的指针。因而下述语句

PT_STU2=>PT_STU1%NEXT

从语法上来讲是合法的。但由于 PT_STU1%NEXT 中指向待定,那么一旦执行了这个语句其执行效果难以想象。因此可以通过语句 NULLIFY (PT_STU1%NEXT) 和 NULLIFY (PT_STU2%NEXT) 将其指针值置成空指针,这样再做 PT_STU2=>PT_STU1%NEXT 这种操作时就会

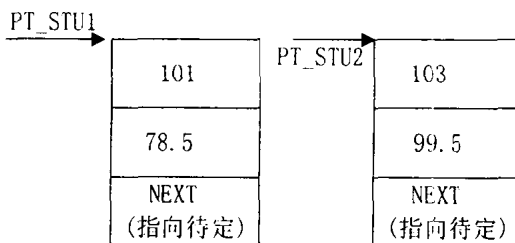


图 9-7 动态分配指针指向的目标关系图

使 PT_STU2 也置空,从而避免出现不可想象的执行结果。另外,有时一个指针不需要它指向任一目标时,也可以将它置空(称作空指针)。指针变量的值为空与指针中值无定义(无具体的指向目标)不同。前者的置空是指指针为空值;而后者的无定义的处理方法与具体的 FORTRAN90 系统有关,可能系统会随便给它一个指针值,也可能由系统给一个缺省的确定值。

空指针用符号 \perp 表示。例 9-7 程序中将两个指针的成员 NEXT 置空后,如图 9-8 所示。

内在函数 ASSOCIATED 用于查询指针指向的问题。该函数有两个虚元,其中第二个虚元是可选的,函数值为逻辑型,或者返回真,或者返回假。

假设 PT1 和 PT2 都是某种类型的指针变量,TR 是该类型的目标变量。那么函数调用

ASSOCIATED(PT1)

表示若 PT1 指向某个实在目标,函数值为真;若 PT1 指向空,函数值为假。

若使用两个虚元,函数调用

ASSOCIATED(PT1,TR)

表示若 PT1 指向目标 TR 时,函数值为真,否则为假。函数调用

ASSOCIATED(PT1,PT2)

若 PT1 和 PT2 指向相同(包括均指向空)时,函数值为真,否则为假。

例 9-7 中程序输出结果如下:

101 78.5000000

103 99.5000000

程序结果输出以后,动态建立的两个由 PT_STU1 和 PT_STU2 所指向的目标没有再保留的必要,可以使用语句 DEALLOCATE(PT_STU1)和 DEALLOCATE(PT_STU2)将 PT_STU1 和 PT_STU2 所指向的目标释放,将它们所占的存储空间归还系统。

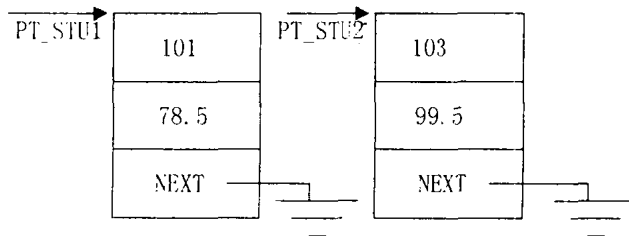


图 9-8 指针置空后关系图

使用动态控制还需要注意以下几个问题:

1° 如果不清楚指针的状态,可以通过 ASSOCIATED 函数来确定。使用该函数时,它的两个实在参数如果是指针时,不能是无定义(指针待定)的。否则测出的函数值无实际意义。例如,如果 PT1 无定义,那么

ASSOCIATED(PT1)

的函数值(假如有该值的话)对用户来讲并无实用价值。因而,当 PT1 无定义时,不允许在程序中出现这样的函数调用。

2° 以图 9-8 为例,这时若执行语句

PT_STU2 = >PT_STU1

PT_STU2 也转去指向 PT_STU1 所指向的目标,而原来 PT_STU2 所指向的目标还占具存储空间,这个目标既未归还给系统(释放空间),又无法再访问它或其成员,因而这种做法是不恰当的。这有两种处理方法。其一是

DEALLOCATE(PT_STU2)

PT_STU2 = >PT_STU1

即先将 PT_STU2 这个目标释放,再修改 PT_STU2 的指向。方法二是引入另一个与 PT_STU2 具有同样类型的指针 PT_STU3,然后做

PT_STU3 = >PT_STU2

PT_STU2 = >PT_STU1

即先让 PT_STU3 指向 PT_STU2 指向的目标,然后再修改 PT_STU2 的指向。以后可用 PT_STU3 访问原来 PT_STU2 指向的目标(见图 9-9)。这种方法在链表等数据结构的操作中经常用到。

3° 仍然在图 9-8 基础上,做以下语句

PT_STU1%NEXT = >PT_STU2

可形成图 9-10 的关系。这样只要知道指针 PT_STU1 就可以用 PT_STU1%NEXT 找到目标 PT_STU2,就好像拉成链一样。这就是下一节要讲的链表。

4° 要避免出现悬挂指针。例如,当指针 P1 和指针 P2 都指向同一目标 T 时,如果执行语句

DEALLOCATE(P1)

则 P1 已被释放,其 P1 指向的目标已不存在,再访问 P1 是非法的。同理,P2 指向的目标也不能再被访问(指针 P2 被悬挂起来),否则会产生不可预料的结果。

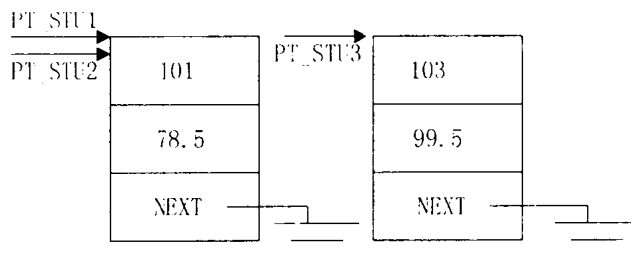


图 9-9 防止目标无法访问的方法

9.3 链 表

链表是指针最常用的编程方法之一。本节通过链表的建立、插入、遍历及删除等操作的例子,比较详细地介绍了有关链表的概念及其相关操作的实现方法。

9.3.1 链表的概念

数组由在内存中连续存放的同类型数组元素所组成。它用于排序、矩阵运算等非常方便,存取速度也很快,且能按下标对数组元素进行直接存取。但数组也有其缺点,如删除数组中某元素的值或在数组中插入一个元素时需要对大量的数组元素进行移动,这限制了数组在某些方面的应用。

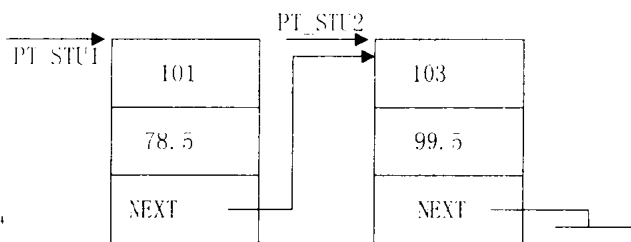


图 9-10 用指针形成拉链的方法

学习了派生类型和指针之后,已经了解了动态数据结构的特点。它可以用 ALLOCATE 函数随时为一个指针变量分配存储空间,用 DEALLOCATE 函数随时释放不再需要的存储空间。这就为链表结构的产生和应用奠定了基础。

链表的特点是由若干个具有相同派生类型的数据拉成链(见图 9-10),这些派生类型数据称作链表的表目(也称结点),链表中第 1 个表目称作头结点,指向头结点的指针称作头指针。该派生类型的成员中至少有一个指向本派生类型的指针,链表中最后一个表目的成员的指针值为空,其余表目的成员的指针均指向下一个表目。为了便于对链表的操作,需要引入另一个派生类型,它用于存放链表的头指针。链表的头指针表示链表的存在与否(头指针为空表示空链表,否则表示链表中至少有一个以上的表目)。

为了讨论问题的方便,我们仍用 § 9.2 节中的派生类型 STUDENT_TYPE 来存放学生的学号(NUMBER)和考试成绩(SCORE)。当然,该类型的成员 NEXT 用于链表拉链。表头指针用 STUDENT_HEAD 型的变量存放,该类型的说明如下:

```
TYPE STUDENT_HEAD
    TYPE(STUDENT_TYPE), POINTER::HEAD
END TYPE STUDENT_HEAD
```

由以上讨论所形成的链表的一般形式如图 9-11 所示。

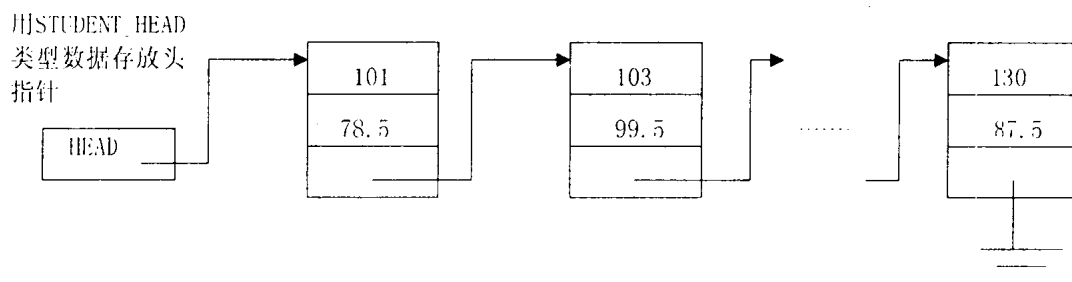


图 9-11 单向链表的一般形式

链表中每个表目由三部分组成:最上栏为成员 NUMBER 的值,中栏为成员 SCORE 的值,最下栏为成员 NEXT 的值。链表的头指针是非常重要的,有了它就可以访问链表中的任何一个表目。最后一个表目的 NEXT 值为空,表示链表的结束。如果让最后一个表目的成员 NEXT 指向头结点,就形成了单向循环链表(见图 9-12)。我们只讨论图 9-11 所示的单向非循环链表(简称单向链表或简称链表)。

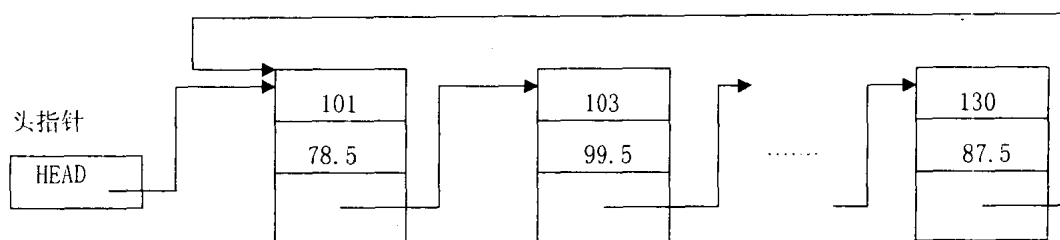


图 9-12 单向循环链表

由于本节中的各个例子均是围绕对学生链表的操作而展开的,其诸项操作又由不同的子程序单元实现。这些子程序单元及主程序单元均需要用到关于派生类型 STUDENT_TYPE 和 STUDENT_HEAD 的定义。因此引入以下模块定义:

```

MODULE STU_LIST
  IMPLICIT NONE
  TYPE STUDENT_TYPE
    INTEGER::NUMBER
    REAL::SCORE
    TYPE(STUDENT_TYPE),POINTER::NEXT
  END TYPE STUDENT_TYPE
  TYPE STUDENT_HEAD
    TYPE(STUDENT_TYPE),POINTER::HEAD
  END TYPE STUDENT_HEAD
END MODULE STU_LIST

```

另外,通过对链表的性质加以分析,其主要的操作有以下几种:

1. 链表的建立;
2. 链表的遍历(显示链表中各个结点中的数据);
3. 删除链表中的一个结点;
4. 在链表中插入一个结点。

我们把以上四种操作分别简称为建立链表、遍历链表、删除链表和插入链表,并将这四种操作的主要工作分别由子例子程序 CREAT_LIST、PRINT_LIST、DELETE_LIST 和 INSERT_LIST 实现。为了便于上机调试,尽快地观察各种操作之后链表中的状况,以上操作均由菜单选项驱动。

[例 9-8]由以下菜单提示用户输入 0 至 4 之间的数:

```

Main Menu
-----
0 - - - - Return
1 - - - - Creat
2 - - - - Print

```

3 - - - - Delete

4 - - - - Insert

=====

Select 0 - - 4:

用户输入 0 时,程序运行结束;输入 1 时,调用子程序 CREAT_LIST 建立学生成绩链表;输入 2 时,调用子程序 PRINT_LIST 显示链表中各个表目的学号及考试成绩;输入 3 时,调用子程序 DELETE_LIST 删除指定的结点;输入 4 时,调用子程序 INSERT_LIST 插入一个结点。每个操作(选 0 除外)结束之后,又显示以上菜单,供用户再选相应的操作编号,一直到用户输入 0 时为止。

完成以上菜单驱动的主程序如下:

```
PROGRAM LIST_STUDENT
```

```
USE STU_LIST
```

```
INTEGER::NB,NUM
```

```
TYPE(STUDENT_TYPE)::TEMP_STU
```

```
TYPE(STUDENT_HEAD)::HEAD_STU
```

```
NULLIFY(HEAD_STU%HEAD)
```

```
FIRST:DO
```

```
PRINT *
```

```
PRINT *,'          Main Menu'
```

```
PRINT *,'- - - - -'
```

```
PRINT *,'  0 - - - - Return'
```

```
PRINT *,'  1 - - - - Creat'
```

```
PRINT *,'  2 - - - - Print'
```

```
PRINT *,'  3 - - - - Delete'
```

```
PRINT *,'  4 - - - - Insert'
```

```
PRINT *,'====='
```

```
PRINT *,'Select 0 - - 4: '
```

```
READ *,NB
```

```
SELECT CASE(NB)
```

```
  CASE(0)
```

```
    EXIT
```

```
  CASE(1)
```

```
    CALL CREAT_LIST(HEAD_STU)
```

```
  CASE(2)
```

```
    CALL PRINT_LIST(HEAD_STU)
```

```
  CASE(3)
```

```
    PRINT *,'Input delete number'
```

```
    READ *,NUM
```

```
    CALL DELETE_LIST(HEAD_STU,NUM)
```



```

CASE(4)
    PRINT *, 'Input insert data:'
    READ *, TEMP_STU%NUMBER, TEMP_STU%SCORE
    CALL INSERT_LIST(HEAD_STU, TEMP_STU)
CASE DEFAULT
    PRINT *, 'Input data error!'
END SELECT
END DO FIRST
END PROGRAM LIST_STUDENT

```

通过这种菜单式的选项驱动,每做完建立链表、删除链表和插入链表中任一操作之后,可以立即选择遍历链表操作显示各结点中的数据。由此检查有关操作正确与否,从而判断实现各个操作的子程序是否正确。

主程序中派生类型变量 HEAD_STU 用于存储表头指针(即 HEAD_STU%HEAD)。TEMP_STU 是 STUDENT_TYPE 型变量,用于存放插入结点的数据。在调用子程序 INSERT_LIST 插入结点之前,先用 READ 语句输入被插入结点的学号及学生成绩,分别暂存到 TEMP_STU%NUMBER 和 TEMP_STU%SCORE 中。而删除结点之前先输入被删除结点的学生学号,放入整型变量 NUM 中,然后再调用子程序 DELETE_LIST 将学号为 NUM 的那个结点从链表中删除。

以下各小节是在模块 STU_LIST 及主程序 LIST_STUDENT 的基础上,讨论各种操作功能的实现过程。

9.3.2 链表的插入及建立

建立链表的过程就是把一个个表目插入链表中的过程。因而建立链表的操作实际上就是不断地调用子程序 INSERT_LIST 的过程,只要将插入链表的操作分析透彻,建立链表操作也就迎刃而解了。

[例 9-9] 编写实现链表插入的子程序。假设插入顺序按学号从小到大进行,即学号小的靠近表头,学号大的靠近表尾。

该子程序如下:

```

SUBROUTINE INSERT_LIST(HEAD_LIST, INSLIST)
    USE STU_LIST
    TYPE(STUDENT_HEAD), INTENT(INOUT)::HEAD_LIST
    TYPE(STUDENT_TYPE), INTENT(IN)::INSLIST
    TYPE(STUDENT_TYPE), POINTER::STU_PTR, STU_PTR1, STU_PTR2
    ALLOCATE(STU_PTR)
    STU_PTR%NUMBER = INSLIST%NUMBER
    STU_PTR%SCORE = INSLIST%SCORE
    STU_PTR1 = >HEAD_LIST%HEAD

```

```

IF (.NOT. ASSOCIATED(STU_PTR1)) THEN
    HEAD_LIST%HEAD = >STU_PTR
    NULLIFY(STU_PTR%NEXT)
ELSE IF (STU_PTR%NUMBER < STU_PTR1%NUMBER) THEN
    HEAD_LIST%HEAD = >STU_PTR
    STU_PTR%NEXT = >STU_PTR1
ELSE
    DO WHILE (ASSOCIATED(STU_PTR1).AND. &
        (STU_PTR%NUMBER > STU_PTR1%NUMBER))
        STU_PTR2 = >STU_PTR1
        STU_PTR1 = >STU_PTR1%NEXT
    END DO
    IF (ASSOCIATED(STU_PTR1).AND. STU_PTR1%NUMBER &
        == STU_PTR%NUMBER) THEN
        PRINT *, 'Number error!'
    ELSE
        STU_PTR2%NEXT = >STU_PTR
        STU_PTR%NEXT = >STU_PTR1
    END IF
END IF
END SUBROUTINE INSERT_LIST

```

该子例子程序一共有两个虚元: HEAD_LIST 和 INSLIST。其中 HEAD_LIST%HEAD 是链表表头(在程序中先用指针 STU_PTR1 指向该表头), INSLIST 是 STUDENT_TYPE 型变量, 表示要插入的结点的数据。为了将它的数据插入链表, 需要说明一个 STUDENT_TYPE 型指针 STU_PTR(用 ALLOCATE 语句分配空间), 并把 INSLIST%NUMBER 中的值先送入 STU_PTR%NUMBER 中, 把 INSLIST%SCORE 中的值先送入 STU_PTR%SCORE 中(为了便于讲解, 假设学号和成绩分别是 100 和 85.0)。这样该插入操作就变成了把 STU_PTR 指向的数据插入链表的问题。这种插入操作分以下几种情况进行:

1. 如果 STU_PTR1 的值为空, 说明链表中还没有结点(是空表)。这时只需要让头指针 HEAD_LIST%HEAD 指向 STU_PTR, 并将 STU_PTR%NEXT 置空即可(插入后如图 9-13(a)所示)。

2. 如果 STU_PTR1 的值不为空, 说明链表中有结点。这时先测试一下 STU_PTR%NUMBER 是否小于 STU_PTR1%NUMBER。如果小于, 说明 STU_PTR 应成为新的表头结点, 其处理方法为:

```

HEAD_LIST%HEAD = >STU_PTR
STU_PTR%NEXT = >STU_PTR1

```

该插入过程见图 9-13(b), 其中虚线表示插入时新建立起的指针, 带十字交叉的线表示原

来的指针连线(下同)。

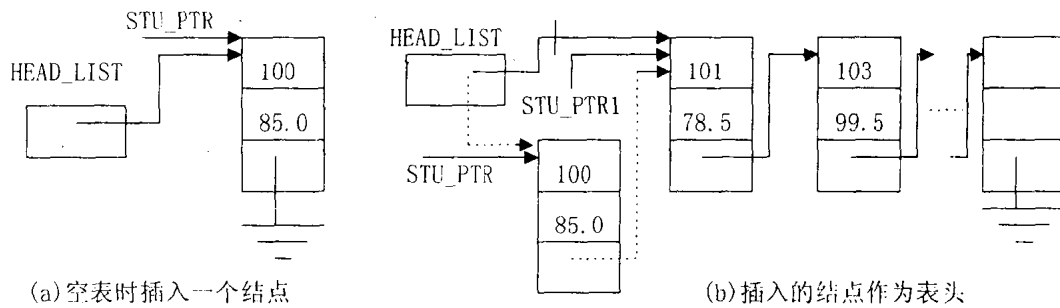


图 9-13 插入的结点作为表头的两种情况

3. 如果以上两种情况都不成立,那么被指入的结点或者应插入到表中央,或者作为表尾。这时应根据学号的大小用循环结构寻找应插入的位置。程序中找到位置是在 STU_PTR2 和 STU_PTR1 之间。这时又分两种情况:一种是 STU_PTR1%NUMBER 的值与插入结点 STU_PTR 的 NUMBER 相等,这时输出错误信息(假设学号应唯一),插入结点操作被拒绝;另一种情况是正常完成插入操作,见图 9-14。

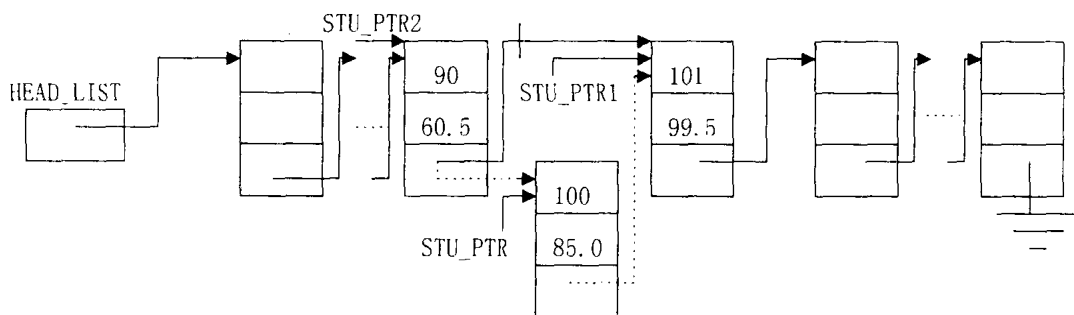


图 9-14 结点插入链表中间的情况

将结点插入表尾的操作与插入链表中间的操作步骤是一样的,只是形成的操作图略有不同。读者可以自己体会这方面的关系。

有了链表插入操作子程序,建立链表子程序就好编写了。

[例 9-10]编写建立链表的子程序,表目中的顺序仍然按学号从小到大进行。

该子程序如下:

```
SUBROUTINE CREAT_LIST(HEAD_LIST)
  USE STU_LIST
  TYPE(STUDENT_HEAD), INTENT(INOUT)::HEAD_LIST
  TYPE(STUDENT_TYPE)::TEMP_LIST
  TYPE(STUDENT_TYPE), POINTER::STU_PTR1, STU_PTR2
  STU_PTR1 => HEAD_LIST%HEAD
  DO WHILE (ASSOCIATED(STU_PTR1))
```

```

    STU_PTR2 = >STU_PTR1
    STU_PTR1 = >STU_PTR1%NEXT
    DEALLOCATE(STU_PTR2)
END DO
NULLIFY(HEAD_LIST%HEAD)
DO
    PRINT *, 'Input a number and a score(if (number<=0) then exit):'
    READ *, TEMP_LIST%NUMBER, TEMP_LIST%SCORE
    IF (TEMP_LIST%NUMBER<=0) EXIT
    CALL INSERT_LIST(HEAD_LIST, TEMP_LIST)
END DO
END SUBROUTINE CREAT_LIST

```

由于建立链表一般是指产生一个新的链表,以示与插入链表的区别。因此在建立链表的实际操作之前,先用一个 DO 循环将原有链表中各个结点(假如有的话)依次用 DEALLOCATE 函数释放之后,再将表头指针置空。当然也可以直接用语句 NULLIFY(HEAD_LIST%HEAD)将表头指针置空,但原来链表若有结点,系统就无法收回这些结点所占内存空间了。

实际建立链表的操作也是由一个 DO 循环完成。在循环体中输入学号和成绩,若学号小于或等于 0,则退出建立链表操作的循环,否则调用 INSERT_LIST 子程序把刚刚输入的学号和成绩作为一个结点插入链表中,然后再去输入下一个学生的学号及成绩,直到输入的学号不是正数为止。该 DO 循环也可以改用 DO WHILE 结构实现,而不使用 EXIT 语句。

9.3.3 遍历链表

[例 9-11] 编写遍历由例 9-9 或例 9-10 所建立的链表的子程序,要求打印出学号、成绩及结点顺序号(从 1 开始)。

该子程序如下:

```

SUBROUTINE PRINT_LIST(HEAD_LIST)
    USE STU_LIST
    TYPE(STUDENT_HEAD), INTENT(IN)::HEAD_LIST
    TYPE(STUDENT_TYPE), POINTER::STU_PTR
    INTEGER::N
    STU_PTR = >HEAD_LIST%HEAD
    N=0
    PRINT *, 'Print student list:'
    DO
        IF (.NOT. ASSOCIATED(STU_PTR)) THEN
            EXIT
        ELSE

```

```

        N = N + 1
        PRINT '(I3,I5,F6.1)',N,STU_PTR%NUMBER,STU_PTR%SCORE
        STU_PTR => STU_PTR%NEXT
    END IF
END DO
END SUBROUTINE PRINT_LIST

```

程序中由整型变量 N 存放结点序号,用指针 STU_PTR 指向表头。只要 STU_PTR 不为空时,则输出结点序号、学号和考试成绩,然后通过语句

```
STU_PTR => STU_PTR%NEXT
```

让 STU_PTR 指向链表中下一个结点。相对链表插入和链表删除而言,遍历链表的程序比较简单。

9.3.4 删除链表

如果链表中某个结点不再适合作为链表的一员时,可将它从链表中摘除。例如学生休学、退学或降级时都会有这种删除操作。

[例 9-12]编写按学号删除链表结点的子程序。

该子程序如下:

```

SUBROUTINE DELETE_LIST(HEAD_LIST, NUM_LIST)
    USE STU_LIST
    TYPE(STUDENT_HEAD), INTENT(INOUT)::HEAD_LIST
    INTEGER, INTENT(IN)::NUM_LIST
    TYPE(STUDENT_TYPE), POINTER::STU_PTR1, STU_PTR2
    STU_PTR1 => HEAD_LIST%HEAD
    IF (.NOT. ASSOCIATED(STU_PTR1)) THEN
        PRINT *, 'List empty! '
    ELSE IF (STU_PTR1%NUMBER == NUM_LIST) THEN
        HEAD_LIST%HEAD => STU_PTR1%NEXT
        DEALLOCATE(STU_PTR1)
        PRINT '(I5,A)', NUM_LIST, ' Deleted! '
    ELSE
        DO WHILE (ASSOCIATED(STU_PTR1).AND.&
            (STU_PTR1%NUMBER /= NUM_LIST))
            STU_PTR2 => STU_PTR1
            STU_PTR1 => STU_PTR1%NEXT
        END DO
        IF (.NOT. ASSOCIATED(STU_PTR1)) THEN
            PRINT '(I5,A)', NUM_LIST, ' not been found! '
        END IF
    END IF
END SUBROUTINE DELETE_LIST

```

```

ELSE
    STU_PTR2%NEXT=>STU_PTR1%NEXT
    DEALLOCATE(STU_PTR1)
    PRINT '(15,A)',NUM_LIST,' Deleted! '
END IF
END IF
END SUBROUTINE DELETE_LIST

```

程序中先让指针 STU_PTR1 指向表头,然后分以下几种情况采取不同的操作步骤(为方便起见,假设被删结点学号为 101):

1. 如果 STU_PTR1 的值为空,说明该链表为空表,这时输出 '空表' 信息后返回。
2. 如果不是空表,测试一下表头结点的学号是否与被删结点学号 NUM_LIST 相等。

若相等则修改头指针 HEAD_LIST%HEAD,并把被删结点所占空间释放。该操作见图 9-15 所示。

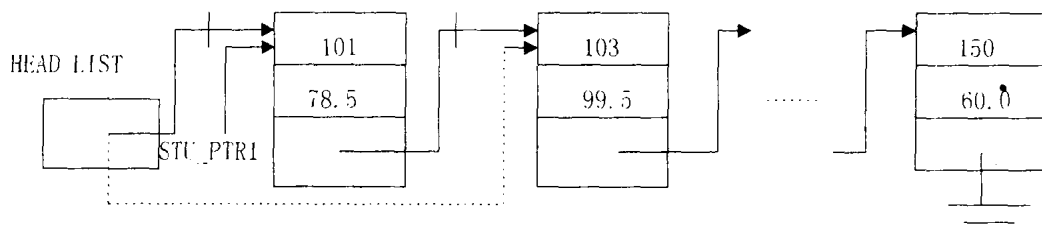


图 9-15 表头结点被删除示意图

3. 如果以上两种情况都不成立,则用 DO 循环在链表中寻找被删结点。若链表中所有结点的学生都不等于 NUM_LIST,则退出循环后 STU_PTR1 一定为空值。这时输出 '结点未找到' 信息后返回。如果 STU_PTR1 不为空,则 STU_PTR1 就是被删结点,而 STU_PTR2 指向 STU_PTR1 的前一个结点,这时可用语句。

• STU_PTR2%NEXT=>STU_PTR1%NEXT

将 STU_PTR1 所指向的结点从链表中摘除,然后再用 DEALLOCATE 函数将摘除的结点空间还给系统。这种操作如图 9-16 所示。

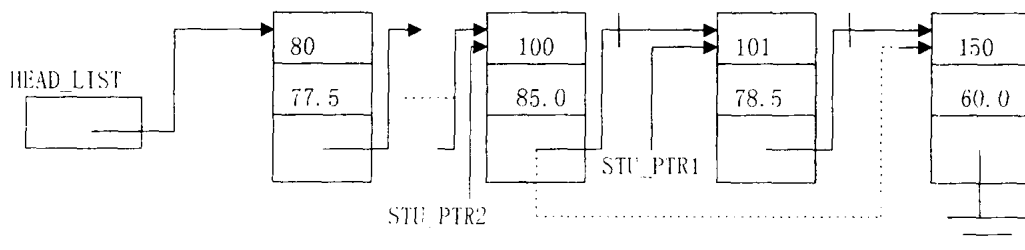


图 9-16 删除链表中间结点示意图

如果将 9.3.1 到 9.3.4 各节例子中的程序及模块 STU_LIST 联系起来考虑,就能实现菜单驱动下的有关链表的诸项操作。

9.3.5 链表插入和链表删除的其它方法

例 9-8 至例 9-12 介绍了有序链表的操作。所谓有序链表是指链表中的各个表目按某个关键词成员(或成员组)的值进行排序,无论是插入还是删除等操作均以维持上述排序顺序为原则。这种有序表的操作除了本节中介绍的建立链表、遍历链表、删除链表及插入链表之外,还有许多其它操作。例如将链表进行逆序排序操作,即表头结点变成表尾,表尾结点变成表头。以例 9-8 至例 9-12 中的链表为例,就是将原有链表按学号从大至小顺序重新链接。另外还有链表检索操作,如查出具具有某个成绩的所有学生的学号等。

除了上述链表之外,还有按其它方式组成的链表。例如,每次插入和删除操作都只在表头结点处进行,即新插入的结点作为表头,删除操作也只是删除表头结点,这种链表叫后进先出链表。如果新插入的结点总是放在表尾,删除操作总是删除表头结点,这就是先进先出链表,自然界中称之为队列(这类似于排队买东西的队列,但不允许“夹塞”)。例 9-13 介绍这种队列的插入和删除操作。

[例 9-13]按先进先出方式改写例 9-9 和例 9-12 中的插入、删除操作过程。

将表目插到表尾处的过程如下:

```
SUBROUTINE INSERT_LIST(HEAD_LIST, INSLIST)
  USE STU_LIST
  TYPE(STUDENT_HEAD), INTENT(INOUT)::HEAD_LIST
  TYPE(STUDENT_TYPE), INTENT(IN)::INSLIST
  TYPE(STUDENT_TYPE), POINTER::STU_PTR, STU_PTR1
  ALLOCATE(STU_PTR)
  STU_PTR%NUMBER = INSLIST%NUMBER
  STU_PTR%SCORE = INSLIST%SCORE
  NULLIFY(STU_PTR%NEXT)
  STU_PTR1 = >HEAD_LIST%HEAD
  IF (.NOT. ASSOCIATED(STU_PTR1)) THEN
    HEAD_LIST%HEAD = >STU_PTR
  ELSE
    DO WHILE (ASSOCIATED(STU_PTR1%NEXT))
      STU_PTR1 = >STU_PTR1%NEXT
    END DO
    STU_PTR1%NEXT = >STU_PTR
  END IF
END SUBROUTINE INSERT_LIST
```

删除表头结点的过程如下:

```
SUBROUTINE DELETE_LIST(HEAD_LIST)
```

```

USE STU_LIST
TYPE(STUDENT_HEAD), INTENT(INOUT)::HEAD_LIST
TYPE(STUDENT_TYPE), POINTER::STU_PTR
INTEGER::NUM_LIST
IF (.NOT. ASSOCIATED(HEAD_LIST%HEAD)) THEN
    PRINT *, 'List empty! '
ELSE
    STU_PTR = >HEAD_LIST%HEAD
    NUM_LIST = STU_PTR%NUMBER
    HEAD_LIST%HEAD = >STU_PTR%NEXT
    DEALLOCATE(STU_PTR)
    PRINT '(I5,A)', NUM_LIST, ' Deleted! '
END SUBROUTINE DELETE_LIST

```

本例中的两个过程与例 9-9、例 9-12 相比比较简单,具体步骤不再多述。但是如果在链表中增加一个表尾指针的话,将表目插入表尾的操作可以更简单一些,插入速度也会更快一些。

9.4 二叉树

树在自然界中是十分常见的,它有树根、树干、树叶等特征。计算机界的树与自然界中的树有相似之处,但也有独自的特点。计算机界中的树代表一种数据结构,其树根在上,树叶在下。图 9-17 就是典型的树。它由 A 至 G 七个结点组成,结点是存放数据的地方。结点 A 叫树根,也称根结点,它是对树进行操作的出发点(类似于链表的表头结点)。结点 B、C、D 是结点 A 的孩子,结点 E、F 是结点 B 的孩子,结点 G 是结点 D 的孩子;结点 A 叫结点 B、C、D 的双亲,同理结点 B 是结点 E、F 的双亲,结点 D 是结点 G 的双亲。从某结点的孩子开始及其分支部分称作该结点的子树,如结点 A 有三棵子树,这些子树的树根分别是 B、C、D,以 B 为树根的子树有三个结点,而以 C 为树根的子树只有树根 C。结点 E、F、C、G 不再有孩子,这些结点没有子树,它们称作树叶。从子树的角度考虑,一棵树是由若干棵子树组成,而每个子树又是由各自的子树组成。因此,树的定义是递归的,也是分层的。以图 9-17 为例,树根为第一层,根的孩子 B、C、D 为第二层,结点 E、F、G 为第三层。树中结点的最大层次叫树的深度,图 9-17 中的树的深度为 3。如果一棵树中任一结点的孩子数最多不超过 2,这种树就叫二叉树。二叉树结点的两个孩子,分别称作左孩子和右孩子。由结点的左孩子作为根的子树称作该结点的左子树,由结点的右孩子作为根的子树称作该结点的右子树。

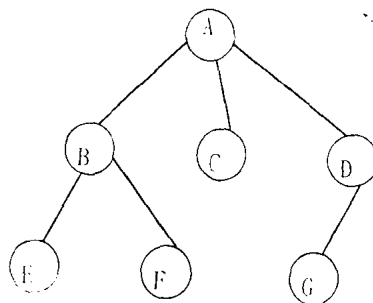


图 9-17 树

考查一棵姓氏二叉树(图 9-18),每个结点由姓氏和两个指针组成,可以定义派生类

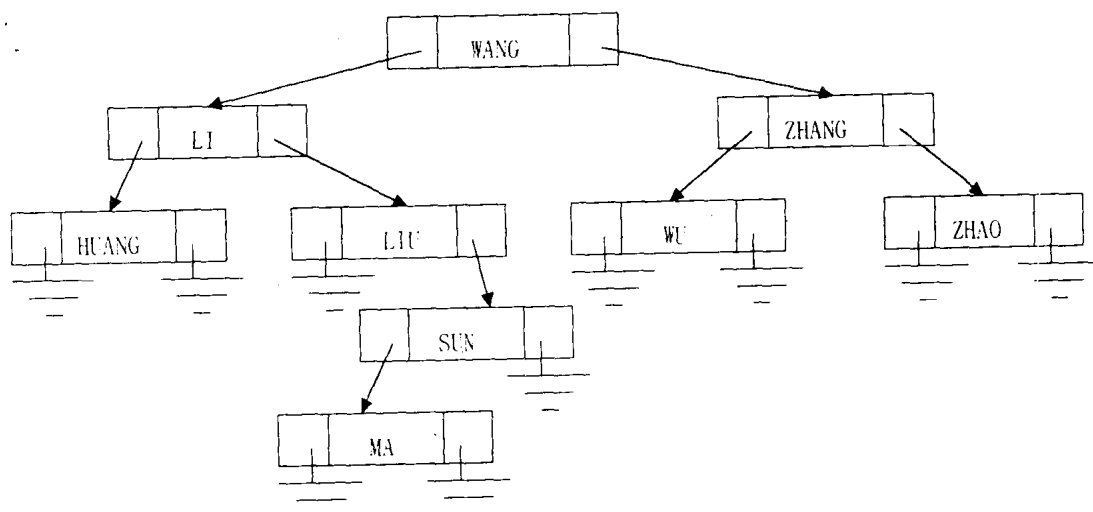


图 9-18 姓氏二叉树

型:

```

TYPE NODE_TREE
  CHARACTER(LEN=6)::SURNAME
  TYPE(NODE_TREE), POINTER::LEFT_PTR, RIGHT_PTR
END TYPE NODE_TREE

```

通过对这种派生类型进行相应的操作,就可以建立二叉树和遍历二叉树中的各个结点。

[例 9-14]编写姓氏二叉树插入结点和遍历结点的子程序。在主程序中多次调用插入结点过程,形成二叉树,然后调用遍历结点过程将该树中各结点姓氏输出。

程序如下:

```

PROGRAM TREE_EXAMPLE
  IMPLICIT NONE
  TYPE NODE_TREE
    CHARACTER(LEN=6)::SURNAME
    TYPE(NODE_TREE), POINTER::LEFT_PTR, RIGHT_PTR
  END TYPE NODE_TREE
  TYPE(NODE_TREE), POINTER::HEAD_PTR
  CHARACTER(LEN=6)::TEMP_SURNAME
  NULLIFY(HEAD_PTR)
  DO
    READ(*, '(A)') TEMP_SURNAME
    IF (TEMP_SURNAME == ' ') EXIT
    CALL INSERT_NODE(HEAD_PTR, TEMP_SURNAME)
  END DO
  PRINT *, 'InOrder:'
  CALL PRINT_IN(HEAD_PTR)

```

CONTAINS

```
RECURSIVE SUBROUTINE INSERT_NODE(HEAD_TREE, &
    INS_SURNAME)
    TYPE(NODE_TREE), POINTER :: HEAD_TREE
    CHARACTER(LEN=6), INTENT(IN) :: INS_SURNAME
    IF (.NOT. ASSOCIATED(HEAD_TREE)) THEN
        ALLOCATE(HEAD_TREE)
        HEAD_TREE%SURNAME = INS_SURNAME
        NULLIFY(HEAD_TREE%LEFT_PTR)
        NULLIFY(HEAD_TREE%RIGHT_PTR)
    ELSE IF (INS_SURNAME < HEAD_TREE%SURNAME) THEN
        CALL INSERT_NODE(HEAD_TREE%LEFT_PTR, &
            INS_SURNAME)
    ELSE
        CALL INSERT_NODE(HEAD_TREE%RIGHT_PTR, &
            INS_SURNAME)
    END IF
END SUBROUTINE INSERT_NODE

RECURSIVE SUBROUTINE PRINT_IN(HEAD_TREE)
    TYPE(NODE_TREE), POINTER :: HEAD_TREE
    IF (ASSOCIATED(HEAD_TREE)) THEN
        CALL PRINT_IN(HEAD_TREE%LEFT_PTR)
        PRINT *, HEAD_TREE%SURNAME
        CALL PRINT_IN(HEAD_TREE%RIGHT_PTR)
    END IF
END SUBROUTINE PRINT_IN

END PROGRAM TREE_EXAMPLE
```

如果输入以下数据：

WANG✓
ZHANG✓
WU✓
LI✓
HUANG✓
ZHAO✓
LIU✓
SUN✓
MA✓
✓

则会建立与图 9-18 完全相同的姓氏二叉树,其输出结果为:

InOrder:

HUANG

LI

LIU

MA

SUN

WANG

WU

ZHANG

ZHAO

本例中主程序分三个部分:

1. 有关类型和变量的说明及赋初值,其中 HEAD_PTR 是根结点的指针,开始时其值为空,表示空树。

2. 在键盘上输入若干个姓氏并调用过程 INSERT_NODE 将它们一个个插入到二叉树中,一直到输入的姓氏为空格时停止。

插入结点过程 INSERT_NODE 是一个递归过程,它有两个虚元。第一个虚元 HEAD_TREE 给出根结点的指针,第二个虚元 INS_SURNAME 是要插入的姓氏数据。该过程首先测试函数 ASSOCIATED(HEAD_TREE)的值。若该值为假,表示树中尚无结点,这时需要为指针变量 HEAD_TREE 分配存储空间,并将其两个指针都置为空,将要插入的结点的姓氏送入 HEAD_TREE% SURNAME 中,插入过程完毕。若 ASSOCIATED(HEAD_TREE)的值为真,表示树中已有结点。这时根据 INS_SURNAME < HEAD_TREE% SURNAME 的值决定插入次序。若该值为真,用语句

CALL INSERT_NODE(HEAD_TREE% LEFT_PTR, INS_SURNAME)

把 INS_SURNAME 插入到以 HEAD_TREE 的左孩子为根的子树中。若 INS_SURNAME 不小于 HEAD_TREE% SURNAME,则用语句

CALL INSERT_NODE(HEAD_TREE% RIGHT_PTR, INS_SURNAME)

把 INS_SURNAME 插入到以 HEAD_TREE 的右孩子为根的子树中。这种插入通过递归调用实现,最后可将 INS_SURNAME 插入到一个合适的位置(结点刚插入时一定是树叶)。

实际上这种插入能形成有序树,它的任一结点姓氏均大于左孩子的姓氏,并小于或等于右孩子的姓氏。

3. 调用递归过程 PRINT_IN 遍历树中结点。该过程只有一个指针虚元 HEAD_TREE,指向树根。如果 HEAD_TREE 的值不为空,则用语句

CALL PRINT_IN(HEAD_TREE% LEFT_PTR)

先遍历左子树,然后访问结点 HEAD_TREE(输出它的姓氏值),最后用语句

CALL PRINT_TN(HEAD_TREE% RIGHT_PTR)

遍历右子树。

本例中,除以上三个部分之外,还有以下两点说明:

1° 程序中的两个过程均在主程序单元之内定义,属于内部过程。如果把它们定义为外部过程,需要定义模块:

```
MODULE BTREE
  IMPLICIT NONE
  TYPE NODE_TREE
    CHARACTER(LEN=6)::SURNAME
    TYPE(NODE_TREE),POINTER::LEFT_PTR,RIGHT_PTR
  END TYPE NODE_TREE
END MODULE BTREE
```

在主程序和两个过程中均用语句

```
USE BTREE
```

引用模块 BTREE。另外,由于两个过程均用指针变量作为虚元,故需要在主程序中使用接口块:

```
INTERFACE
  RECURSIVE SUBROUTINE INSERT_NODE(HEAD_TREE,&
    INS_SURNAME)
    TYPE(NODE_TREE),POINTER::HEAD_TREE
    CHARACTER(LEN=6),INTENT(IN)::INS_SURNAME
  END SUBROUTINE INSERT_NODE
  RECURSIVE SUBROUTINE PRINT_IN(HEAD_TREE)
    TYPE(NODE_TREE),POINTER::HEAD_TREE
  END SUBROUTINE PRINT_IN
END INTERFACE
```

2° 在程序中遍历二叉树时,采用的方法是先遍历左子树,访问根结点,再访问右子树。如果访问顺序不同,会有不同的输出结果。一般情况下,遍历二叉树有三种方式:

a)中序遍历:先中序遍历左子树,然后访问根结点,最后中序遍历右子树。本例中的过程 PRINT_TN 就属于中序遍历。

b)前序遍历:先访问根结点,然后前序遍历左子树,最后前序遍历右子树。

c)后序遍历:先后序遍历左子树,然后后序遍历右子树,最后访问根结点。

前序遍历过程如下:

```
RECURSIVE SUBROUTINE PRINT_PRE(HEAD_TREE)
  TYPE(NODE_TREE),POINTER::HEAD_TREE
  IF (ASSOCIATED(HEAD_TREE)) THEN
    PRINT *,HEAD_TREE%SURNAME
    CALL PRINT_PRE(HEAD_TREE%LEFT_PTR)
    CALL PRINT_PRE(HEAD_TREE%RIGHT_PTR)
  END IF
END SUBROUTINE PRINT_PRE
```

后序遍历过程如下：

```

RECURSIVE SUBROUTINE PRINT_AFT(HEAD_TREE)
  TYPE(NODE_TREE), POINTER :: HEAD_TREE
  IF (ASSOCIATED(HEAD_TREE)) THEN
    CALL PRINT_AFT(HEAD_TREE%LEFT_PTR)
    CALL PRINT_AFT(HEAD_TREE%RIGHT_PTR)
    PRINT *, HEAD_TREE%SURNAME
  END IF
END SUBROUTINE PRINT_AFT

```

在输入不变的情况下,采用三种遍历方式访问结点的顺序如表 9-1 所示。

表 9-1 三种遍历过程输出表

输入数据	中序遍历输出结果	前序遍历输出结果	后序遍历输出结果
WANG	HUANG	WANG	HUANG
ZHANG	LI	LI	MA
WU	LIU	HUANG	SUN
LI	MA	LIU	LIU
HUANG	SUN	SUN	LI
ZHAO	WANG	MA	WU
LIU	WU	ZHANG	ZHAO
SUN	ZHANG	WU	ZHANG
MA	ZHAO	ZHAO	WANG

9.5 指针与数组

FORTRAN90 中由于指针只是变量的一种属性,并不属于独立的类型,故不能把许多指针作为元素而形成指针数组。另外,具有指针属性的变量也不能直接指向数组,而只能指向数组中的某个元素。因此,FORTRAN90 中指针的用法与其它语言略有区别,在指针与数组的关系上也不能直接关联。但可以设法用指针访问数组的元素,从而实现用指针访问数组全体。下边通过三个例子说明指针与数组间的关系。

[例 9-15] 设某学习小组有 6 名学生,每名学生的信息有姓名、学号和成绩三项,用指针法访问学生数组,并将该学习小组全体成员的数据输出。

程序如下：

```

PROGRAM SCORE_INOUT1
  IMPLICIT NONE
  INTEGER, PARAMETER :: N=6, LENGTH=10
  TYPE SCORE_TYPE
    CHARACTER(LEN=LENGTH)::NAME
    INTEGER::NUMBER
    REAL::SCORE
  END TYPE SCORE_TYPE

```

```

TYPE STUDENT_PTR
  TYPE(SCORE_TYPE), POINTER::PTRSTU
END TYPE STUDENT_PTR
TYPE(SCORE_TYPE), DIMENSION(1:N), TARGET::STU_SCORE
TYPE(STUDENT_PTR), DIMENSION(1:N)::STU
INTEGER::I
DO I=1,N
  STU(I)%PTRSTU=>STU_SCORE(I)
END DO
PRINT *, 'INPUT DATA:'
DO I=1,N
  READ *, STU_SCORE(I)
END DO
PRINT *, 'OUTPUT DATA:'
DO I=1,N
  WRITE(*, '(A,I5,F6.1)') STU(I)%PTRSTU
END DO
END PROGRAM SCORE_INOUT1

```

程序运行结果如下:

```

INPUT DATA:
'ZHANG HONG',101,78.5✓
'MA LIN',102,94.0✓
'ZHAO YI',103,99.5✓
'WANG PING',104,70.0✓
'WU BING',105,100.0✓
'LI MING',106,80.0✓
OUTPUT DATA:
ZHANG HONG  101  78.5
MA LIN      102  94.0
ZHAO YI     103  99.5
WANG PING   104  70.0
WU BING     105 100.0
LI MING     106  80.0

```

为了用指针法访问数组元素,定义了只包括一个指向 SCORE_TYPE 型的指针(PTRSTU)成员的派生类型 STUDENT_PTR。STU 是具有该派生类型的一维数组,它的每一个元素都可以指向数组 STU_SCORE 的元素类型。这样通过程序段

```

DO I=1,N
  STU(I)%PTRSTU=>STU_SCORE(I)
END DO

```

让 STU 的第 I($1 \leq I \leq N$)个元素的成员 STU(I)%PTRSTU 指向 STU_SCORE(I)。以后对 STU(I)%PTRSTU 的使用,就是使用了 STU_SCORE(I)。由此可见,FORTRAN90 虽然

不支持指针数组,但通过本例的方法也可以间接地将若干个指针放在同一数组中。

[例 9-16]用动态控制法完成例 9-15 的功能。程序如下:

```
PROGRAM SCORE_INOUT2
  IMPLICIT NONE
  INTEGER, PARAMETER::N=6, LENGTH=10
  TYPE SCORE_TYPE
    CHARACTER(LEN=LENGTH)::NAME
    INTEGER::NUMBER
    REAL::SCORE
  END TYPE SCORE_TYPE
  TYPE STUDENT_PTR
    TYPE(SCORE_TYPE), POINTER::PTRSTU
  END TYPE STUDENT_PTR
  TYPE(STUDENT_PTR), DIMENSION(1:N)::STU
  INTEGER::I
  DO I=1,N
    ALLOCATE(STU(I)%PTRSTU)
  END DO
  PRINT *, 'INPUT DATA:'
  DO I=1,N
    READ *, STU(I)%PTRSTU
  END DO
  PRINT *, 'OUTPUT DATA:'
  DO I=1,N
    WRITE(*, '(A,I5,F6.1)') STU(I)%PTRSTU
  END DO
  DO I=1,N
    DEALLOCATE(STU(I)%PTRSTU)
  END DO
END PROGRAM SCORE_INOUT2
```

本程序中不再说明静态数组目标 STU_SCORE, 而通过 ALLOCATE 语句给 STU(1)%PTRSTU 至 STU(N)%PTRSTU 分配 N 个动态存储空间, 而这 N 个存储空间并不像数组元素那样在内存中连续存放。用 STU(I)%PTRSTU 存放第 I 个学生的有关信息, 应用中可以用 STU(I)%PTRSTU 代表第 I 个学生的全部信息, 也可以用其成员 STU(I)%PTRSTU%NAME、STU(I)%PTRSTU%NUMBER 和 STU(I)%PTRSTU%SCORE 分别表示第 I 个学生的姓名、学号和考试成绩。程序尾部的程序段

```
DO I=1,N
  DEALLOCATE(STU(I)%PTRSTU)
END DO
```

用于释放前边由 ALLOCATE 语句所得到的动态存储空间。由于这种回收过程刚好是在程

序 END 语句之前完成,而一般的 FORTRAN90 系统在程序结束(遇到 END 语句)时,均能自动地回收这些已分配的动态存储空间,因此上述三行语句也可以省略不写。但作为一种良好的程序设计习惯,还是有回收语句序列为好。另外,如果程序不断地申请动态存储空间,考虑到系统空间的承受能力问题,也需要随时释放那些不再需要的动态存储空间。

[例 9-17] 用指针指向含有数组成员的派生类型法完成例 9-15 的功能。

程序如下:

```
PROGRAM SCORE_INOUT3
  IMPLICIT NONE
  INTEGER, PARAMETER :: N=6, LENGTH=10
  TYPE SCORE_TYPE
    CHARACTER(LEN=LENGTH)::NAME
    INTEGER::NUMBER
    REAL::SCORE
  END TYPE SCORE_TYPE
  TYPE STUDENT_ARRAY
    TYPE(SCORE_TYPE), DIMENSION(1:N)::STUARR
  END TYPE STUDENT_ARRAY
  TYPE(STUDENT_ARRAY), TARGET::STU_SCORE
  TYPE(STUDENT_ARRAY), POINTER::STU_PTR
  INTEGER::I
  STU_PTR => STU_SCORE
  PRINT *, 'INPUT DATA:'
  DO I=1,N
    READ *, STU_PTR%STUARR(I)
  END DO
  PRINT *, 'OUTPUT DATA:'
  DO I=1,N
    WRITE(*, '(A,I5,F6.1)') STU_PTR%STUARR(I)
  END DO
END PROGRAM SCORE_INOUT3
```

程序中 STU_SCORE 是派生类型的变量,它的成员只有 N 个元素的一维数组 STU_ARR,该数组元素均具有 SCORE_TYPE 型。用指针变量 STU_PTR 指向目标 STU_SCORE。这样在程序中使用 STU_PTR%STUARR(I)就等价于使用 STU_SCORE%STUARR(I)。

9.6 指针与过程

如果指针变量作为过程的虚元,且该过程是外部过程时,调用它的程序单元需要编写关于该被调过程的接口块。如果函数过程的返回值是指针变量时,也需要使用接口块。下面

分别讨论指针在过程中的使用方法。

9.6.1 函数过程的返回值是简单指针变量

[例 9-18] 利用返回指针变量值的过程求两个整数的最大值。

程序如下：

```
PROGRAM PTR_INTEGER
  INTERFACE
    FUNCTION FUN_PTR(M1,M2) RESULT(PTR)
      INTEGER,INTENT(IN)::M1,M2
      INTEGER,POINTER::PTR
    END FUNCTION FUN_PTR
  END INTERFACE
  INTEGER::X,Y
  INTEGER,TARGET::RM
  READ *,X,Y
  RM=FUN_PTR(X,Y)
  PRINT *,RM
END PROGRAM PTR_INTEGER

FUNCTION FUN_PTR(M1,M2) RESULT(PTR)
  INTEGER,INTENT(IN)::M1,M2
  INTEGER,POINTER::PTR
  INTEGER,TARGET::SM
  PTR=>SM
  IF (M1>M2) THEN
    PTR=M1
  ELSE
    PTR=M2
  END IF
END FUNCTION FUN_PTR
```

函数过程 FUN_PTR 的返回值 PTR 是整型指针。过程中的语句

```
PTR=>SM
```

是不可少的。否则,PTR 并没有指向的目标,导致后边的赋值操作 PTR=M1 和 PTR=M2 的结果是不可预料的。当然,也可以在 IF 结构之前通过语句

```
ALLOCATE(PTR)
```

给 PTR 分配空间。这时关于 SM 的说明及其相关语句 PTR=>SM 均可以去掉。

9.6.2 函数过程的返回值是派生类型指针变量

[例 9-19] 以例 9-15 中的数据为依据,利用函数过程求学生中具有最高成绩的学生的全部信息,并通过函数值返回这些信息。

程序如下：

```
MODULE SCORE_MODULE
  IMPLICIT NONE
  INTEGER, PARAMETER :: N = 6, LENGTH = 10
  TYPE SCORE_TYPE
    CHARACTER(LEN = LENGTH) :: NAME
    INTEGER :: NUMBER
    REAL :: SCORE
  END TYPE SCORE_TYPE
END MODULE SCORE_MODULE

PROGRAM PTR_DERIVED
  USE SCORE_MODULE
  INTERFACE
    FUNCTION MAX_SCORESTU(SCORE, M) RESULT(MAX_SCORE)
      TYPE(SCORE_TYPE), DIMENSION(1:M), TARGET, &
        INTENT(IN) :: SCORE
      INTEGER, INTENT(IN) :: M
      TYPE(SCORE_TYPE), POINTER :: MAX_SCORE
    END FUNCTION MAX_SCORESTU
  END INTERFACE
  TYPE(SCORE_TYPE), DIMENSION(1:N) :: STU_SCORE
  TYPE(SCORE_TYPE), POINTER :: PTR_STU
  INTEGER I
  PRINT *, 'INPUT DATA:'
  DO I = 1, N
    READ '(A, I4, F5.1)', STU_SCORE(I)
  END DO
  ALLOCATE(PTR_STU)
  PTR_STU = MAX_SCORESTU(STU_SCORE, N)
  WRITE(*, '(A, A, I5, F6.1)') 'MAX SCORE DATA: ', PTR_STU%NAME, &
    PTR_STU%NUMBER, PTR_STU%SCORE
  DEALLOCATE(PTR_STU)
END PROGRAM PTR_DERIVED

FUNCTION MAX_SCORESTU(SCORE, M) RESULT(MAX_SCORE)
  USE SCORE_MODULE
  TYPE(SCORE_TYPE), DIMENSION(1:M), TARGET, INTENT(IN) :: SCORE
  INTEGER, INTENT(IN) :: M
  TYPE(SCORE_TYPE), POINTER :: MAX_SCORE
  INTEGER J, L
  L = 1
```

```

DO J=2,M
  IF(SCORE(L)%SCORE<SCORE(J)%SCORE) L=J
END DO
MAX_SCORE=>SCORE(L)
END FUNCTION MAX_SCORESTU

```

函数过程 MAX_SCORESTU 有两个虚元,其中第一个虚元 SCORE 是学生数组,第二个虚元 M 表明 SCORE 的元素个数。函数返回值 MAX_SCORE 是 SCORE_TYPE 型指针。在函数体中求出成绩最好的学生的下标为 L,然后通过语句

```
MAX_SCORE=>SCORE(L)
```

让 MAX_SCORE 指向具有最大成绩的元素 SCORE(L),并将其值返给主调程序 PTR_DERIVED。

本例运行结果如下:

INPUT DATA:

ZHANG HONG 101 78.5✓

MA LIN 102 94.0✓

ZHAO YI 103 99.5✓

WANG PING 104 70.0✓

WU BING 105 100.0✓

LI MING 106 80.0✓

MAX SCORE DATA: WU BING 105 100.0

9.6.3 用虚实结合返回派生类型指针变量

这方面的应用在 9.3 节和 9.4 节中都有相应的介绍。用虚实结合返回派生类型指针变量的值(当然也可以用简单指针变量作为虚元)是用得较多的一种方法。在某些系统中,不允许虚元同时具有 POINTER 和 INTENT 属性。

[例 9-20]用带有指针属性虚元的子例子程序完成例 9-19 的求最大成绩学生信息的功能。

程序如下:

```
MODULE SCORE_MODULE
```

```
...
```

```
END MODULE SCORE_MODULE
```

! 以上部分与例 9-19 的 MODULE SCORE_MODULE 相同

```
PROGRAM PTR_DERIVED
```

```
USE SCORE_MODULE
```

```
SUBROUTINE MAX_SCORESTU(SCORE,M,MAX_SCORE)
```

```
TYPE(SCORE_TYPE),DIMENSION(1:M),TARGET,&
```

```
INTENT(IN)::SCORE
```

```
INTEGER,INTENT(IN)::M
```

```
TYPE(SCORE_TYPE),POINTER::MAX_SCORE
```

```
END SUBROUTINE MAX_SCORESTU
```

```

END INTERFACE
TYPE(SCORE_ TYPE), DIMENSION(1:N)::STU_ SCORE
TYPE(SCORE_ TYPE), POINTER::PTR_ STU
INTEGER I
PRINT *, 'INPUT DATA:'
DO I=1,N
    READ '(A,I4,F5.1)',STU_ SCORE(I)
END DO
CALL MAX_ SCORESTU(STU_ SCORE,N,PTR_ STU)
WRITE( *, '(A,A,I5,F6.1)') 'MAX SCORE DATA: ',PTR_ STU%NAME,&
PTR_ STU%NUMBER,PTR_ STU%SCORE
DEALLOCATE(PTR_ STU)
END PROGRAM PTR_ DERIVED
SUBROUTINE MAX_ SCORESTU(SCORE,M,MAX_ SCORE)
    USE SCORE_ MODULE
    TYPE(SCORE_ TYPE), DIMENSION(1:M), TARGET, INTENT(IN)::SCORE
    INTEGER, INTENT(IN)::M
    TYPE(SCORE_ TYPE), POINTER::MAX_ SCORE
    INTEGER J
    ALLOCATE(MAX_ SCORE)
    MAX_ SCORE = SCORE(1)
    DO J=2,M
        IF(MAX_ SCORE%SCORE < SCORE(J)%SCORE) &
            MAX_ SCORE = SCORE(J)
    END DO
END SUBROUTINE MAX_ SCORESTU

```

子程序中的虚元 MAX_ SCORE 具有 POINTER 属性,通过语句

```
ALLOCATE(MAX_ SCORE)
```

为 MAX_ SCORE 分配存储空间。如果没有这一步,必须在主调程序中用 ALLOCATE 语句为指针变量 PTR_ STU 分配存储空间。否则在子程序中对 MAX_ SCORE 的赋值等项操作的结果是不可预料的。

除了以上与主调程序发生联系的指针的几种用法之外,在过程中还可说明其它的指针变量。由于这些指针变量与其它程序单元没有联系,其用法与主程序单元中的指针变量的用法没有区别。此外,还可以把指针变量放在模块中,这样凡是使用了该模块的程序单元均可共享这些指针变量,达到快速传递数据的目的。如果一个程序单元包含内部过程,该程序单元内说明的指针变量也可被这些内部过程所使用。以上这些方面的用法可通过相应的习题或修改例题而得到训练。

习 题 9

9-1 输入由 20 个字符组成的字符串到字符变量中,然后统计空格、字母、数字和其它字符各有多少个(使用派生类型变量存放这些数值)。

9-2 用含有三个实型数据的派生类型变量存放长方体的长、宽和高,并计算其体积。

9-3 将 10 个 SCORE_TYPE 类型(见 9.1.2)的数据按考试成绩从大至小的顺序排序,要求使用冒泡排序法。

9-4 输入 12 个整数,将其中最小的数与第一个数交换位置,将其中最大的数与最后一个数交换位置。要求编四个子程序:

①输入数据;

②求最小的数及其位置,用派生类型(成员是最小的数和其位置)和数组作为变元;

③求最大的数及其位置,用派生类型(成员是最大的数和其位置)和数组作为变元;

④输出最小的数、最大的数及交换后的 12 个数。

9-5 定义一个返回 COORDINATE 型(见 9.1.2)数据的函数子程序,它有两个虚元,分别表示平面直角坐标系中的 X 轴和 Y 轴坐标。

9-6 写一个函数子程序,实现两个字符串的比较。该函数有两个虚元,均为字符型。如果第一个字符大于第二个字符返回 1,如果第一个字符小于第二个字符返回 -1,两个字符相等时返回 0。要求函数的返回值具有指针类型。

9-7 编写不用操作符“//”实现两个字符串连接的函数。要求函数值具有指针类型。

9-8 改写例 9-2,假设增加学生的年龄信息,考试科目增加到 5 门。

9-9 改写例 9-3,假设有 10 个教师的信息,要求统计每个教师的工资总额,并求出他们的平均工资。

9-10 在上题的基础上,将教师的信息按工作单位从小到大排序(使用顺序交换法)。

9-11 建立和遍历先进先出链表,结点具有例 9-3 中的教师结构类型。

9-12 将上题中建立的链表检索,输出所有 COMPUTER 系的教师的全部信息。

9-13 改写例 9-8 至例 9-12 中的程序,表头不由 STUDENT_HEAD 的变量给出,而直接由 STUDENT_TYPE 型的指针变量给出。

9-14 如果二叉树的头指针由 NODE_HEAD 类型的变量给出,相应的操作程序应如何改写?

假设 NODE_HEAD 的定义如下:

```
TYPE NODE_HEAD
```

```
TYPE(NODE_TREE), POINTER::HEAD
```

```
END TYPE NODE_HEAD
```

9-15 按图 9-1 中给出的教师数据结构,编写某系教师的数据查询程序。要求教师数据存储在一个顺序文件中。

9-16 按图 9-1 中给出的教师数据结构,组织多个链表,每个链表存放一个系的教师的信息。

第 10 章 数据文件

每个程序一般都包括三个主要步骤:输入数据、处理数据和输出数据。输入的数据是由终端键盘输入,输出的数据也是显示在终端显示器上或用打印机打印出来。如果运行时敲错一个数据,全部数据都需要重新输入。程序运行结束后,我们仅能得到屏幕上的显示结果或打印纸上的输出结果,而计算机内并未保留任何数据。若想把这些结果作为另一个程序的输入数据,还需要在键盘上把它们一一敲入,这很不便。

如果先把输入数据形成一个数据文件,存放于外部介质(例如磁盘)上,程序从这些文件上读入数据,其计算后得到的输出结果也存放到数据文件中。这就可以避免因输入错误而进行的重复输入,其计算结果也便于其它程序使用。此外,由于计算机从这些外部介质上的数据文件中存取数据的速度远远快于键盘输入、显示器输出和打印机打印的速度,所以这也有利于加快程序的运行速度。本章主要介绍与数据文件有关的概念及操作方法。

10.1 文件概述

文件是记录的序列,而记录是值(或字符)的序列。FORTRAN 语言对文件的操作一般是以记录为单位的。除特殊说明以外,FORTRAN 程序每执行一次 READ 语句或 WRITE 语句,总是读入或输出一个完整的记录。在分行设备(如终端、打印机等)上,一个记录就是一行,文件中每个记录的长度可以是固定的,也可以是变化的,这与文件的种类及存储方式等性质有关。在终端键盘输入时,按一次回车键就结束一个记录。

FORTRAN 程序(又称源程序)由若干行组成,需要把它存放到文件中才能编译、连接和运行。这种存放源程序的文件称为源程序文件,而存放程序运行时所需要的输入数据和输出数据的文件称为数据文件。

按所用的存储设备的类型,文件分为内部文件和外部文件。如果文件存储在外部设备上(例如磁盘、磁带等),称作外部文件。如果文件的信息存储在计算机内存中,则称之为内部文件。

按文件所使用的外部介质划分,可分成磁盘文件、磁带文件等。为了系统管理外部设备的方便,也把外设当作文件,例如打印机文件、终端显示器文件、键盘文件等。但在高级语言书籍中所谈的文件一般指保存在磁盘或磁带上的那些文件,本书一般指磁盘文件。

按对文件是否能进行读写操作来划分,可分成输入文件、输出文件和输入输出文件。键盘文件是输入文件,打印机文件是输出文件,而磁盘、磁带文件既能作输入操作,也能作输出操作,它们属于输入输出文件。

按文件的存取方式可分为顺序存取文件和直接存取文件。顺序存取文件简称顺序文件,其特点是读(写)文件的记录时,总是从文件的起始位置开始,一个记录接着一个记录读(写)。假设需要读(写)第 n 个记录时,必须按顺序读(写)入前 $n-1$ 个记录。直接存取文件又称随机存取文件,简称直接文件或随机文件。程序中可以根据需要直接存取直接文件中的某个指定的记录,而不必每次都从文件的开始处进行存取。直接文件的存取速度快,但其

记录的长度是固定的、读写时不能使用自由格式；顺序文件的存取速度慢，但记录的长度可变、可以使用自由格式。它们各有所长，不能简单地谁好谁坏。

按文件记录的格式编辑划分，可分为有格式文件和无格式文件。内存中数据是采用二进制内码形式存放的，如果外存中的数据也采用二进制内码方式存放，读写操作时不需要进行格式编辑，这时需使用无格式文件。如果对文件的读写操作使用了有格式（自由格式或自定义格式）的读写语句，外存中的数据不再采用二进制内码形式存放，这时内存和外存之间的数据传递要按所给的格式进行编辑，这种文件是有格式文件。

本章重点讨论有格式的磁盘数据文件的顺序和直接存取方法。

10.2 对文件的基本操作

对文件的基本操作包括打开文件、关闭文件及读、写文件等。

10.2.1 OPEN 语句

向文件输出数据或者从文件中读入数据之前，必须把该文件与一个文件号连接起来。在 FORTRAN 程序中，这可通过 OPEN 语句来实现，通常也把这一操作称为打开一个文件。OPEN 语句的一般形式如下：

OPEN(连接信息表)

其连接信息表中主要包括以下几个说明符：

1. UNIT = ut。此处 ut 是一个文件号，必须是无符号整数。当该说明符是连接信息表的第一个说明符时，“UNIT = ”可以省略。

2. FILE = fe。fe 是字符型数据，代表文件名（不包括尾部空格）。例如语句

OPEN(3, FILE = 'DATA.DAT')

表示把文件号 3 与名为 DATA.DAT 的文件连接起来。

3. STATUS = ss。ss 是由用户给出的字符串，表示文件的状态，可选以下 5 种值之一：'OLD'、'NEW'、'REPLACE'、'SCRATCH' 和 'UNKNOWN'。

'NEW' 表示所指定的文件名是一个以前不存在的新的文件名。如：

OPEN(3, FILE = 'DATA.DAT', STATUS = 'NEW')

使计算机在磁盘上建立一个名为 DATA.DAT 的文件。执行 OPEN 语句成功之后文件中还没有数据，但该文件的状态已变成了 'OLD'。如果在该磁盘上原来已有一个同名的文件，执行该 OPEN 语句时就会出错，需要将 ss 的值改为 'OLD'。

如果 ss 为 'SCRATCH'，表示打开临时文件，将由处理系统为指定的文件号连接一个特殊的“无名”文件。当关闭该文件时，此文件将自动消除。在 OPEN 语句中不能同时指定文件名和 SCRATCH 说明符。

'UNKNOWN' 表示对文件存在状态不清楚，由系统按文件的实际情况处理。例如文件已有，就打开该文件；文件不存在，就着手建立新文件。

'REPLACE' 表示指定的文件不存在时，由系统建立该文件；如已有同名文件，则用此名产生一个新文件（原文件的数据将被删除）。

STATUS 项的缺省值为 'UNKNOWN'。

4. ACCESS = as。as 是字符串,其值可为 'SEQUENTIAL'(顺序的)或 'DIRECT'(直接的)。它指定连接的文件是顺序文件还是直接文件。此说明符的缺省值为 'SEQUENTIAL'。

5. FORM = fm。fm 是字符串,其值为 'FORMATTED' 或 'UNFORMATTED',表明文件是有格式的或无格式的。该说明符对顺序文件的缺省值是 'FORMATTED'(有格式的),对直接文件的缺省值是 'UNFORMATTED'(无格式的)。

6. RECL = rl。rl 是一个其值为正整数的整型表达式,用来指定文件记录的长度,其单位为字节。直接文件中每个记录的长度必须相等,必须选择 RECL 项,顺序文件不需要使用 RECL 说明符。

7. ACTION = an。an 为一字符串,可取值为 'READ'、'WRITE' 或 'READWRITE'。'READ' 表示该文件只能读,禁止写;'WRITE' 表示该文件只能写,不许读;'READWRITE' 表示可以读,也可以写。其缺省值为 'READWRITE'。

8. BLANK = bk。bk 为一字符串,取值可为 'NULL' 或 'ZERO'。'NULL' 表示空格全部忽略不计,'ZERO' 表示数据中的空格作零处理。其缺省值是 'NULL'。

9. IOSTAT = it。it 为整型变量,一执行 OPEN 语句,it 就有了值。执行 OPEN 语句未发生错误时,其值为零;若有错误发生,它便被赋以一个正整数值。

10. ERR = label。若打开文件操作出错,转到标号为 label 的语句处。由于 FORTRAN90 不提倡使用语句标号,而且对出错的判断可以由 IOSTAT 说明符实现,所以本章不使用、也不介绍该语句(包括 READ、WRITE、REWIND、BACKSPACE 等语句)中的 ERR 说明符。各语句中其它与语句标号有关的说明符(如 END=、EOF= 等)也不介绍。

OPEN 语句中各说明符的次序可以任意。但应注意:

1° 文件与文件号在某一程序单元(不一定是主程序)中连接成功,就在该程序的所有单元中都保持这种连接关系,如果以后没有用 CLOSE 语句解除这种连接,也没有新的 OPEN 语句修改其连接特性,则此连接将一直保持到整个 FORTRAN 程序运行结束。

2° 一个文件不能同时与两个文件号连接,一个文件号也不能同时与两个文件连接。

10.2.2 CLOSE 语句

CLOSE 语句又称关闭文件语句,用于解除指定文件号与文件的连接关系。它的一般形式是:

CLOSE(关闭信息表)

关闭信息表主要有下列说明符可供选择:

UNIT = ut

IOSTAT = it

STATUS = ss

其中前两项用法同 OPEN 语句。

STATUS 项用来指定文件关闭后是否保留。ss 是用户指定的字符串,可以是 'KEEP' 或 'DELETE'。'KEEP' 表示文件关闭后继续存在,并可被重新连接。当对应的 OPEN 语句中指定文件具有 'SCRATCH'(清除)状态时,不能使用 'KEEP'; 'DELETE' 表示文件在关闭后不复存在。若该项缺省,除了在 OPEN 语句中标明为 'SCRATCH' 状态的文件之外,其他

文件全部保留。

10.2.3 读写语句

FORTRAN90 读写语句共有 4 种形式:

READ 格式说明,输入表

PRINT 格式说明,输出表

READ(控制信息表)输入表

WRITE(控制信息表)输出表

前两种省略了文件号,表示在系统预先指定的设备上输入或输出,一般微机上隐含指定的输入设备是键盘,隐含的输出设备是终端显示器或打印机。

后两种既适用于在隐含设备上输入或输出,也适用于对文件的输入或输出。控制信息表中的说明符,除 UNIT 项必不可少外,其它说明符皆可根据情况选用,有些还不能同时出现。现将其中主要的几种说明符列举如下:

1. UNIT = ut。ut 为输入或输出文件号。当它为第一项时,可以省写“UNIT =”,只写 ut。若文件号为 *,表示在系统隐含设备上输入或输出。例如:

```
READ(10,FMT='(2F5.1)')X,Y
```

表示从与文件号 10 相连接的文件中按指定格式读出两个数据,分别赋给 X 和 Y。

2. FMT = ft。ft 为格式说明。格式说明与前几章使用的格式说明是一样的,由 * 或一串编辑符组成。格式说明可以有显式说明、星号和字符表达式三种形式。例如:

```
READ(10,FMT='(2F5.2)')X,Y
```

 ! 显式说明的格式用括号括起来,用撇号定界。

```
READ(10,*)X,Y
```

 ! * 表示表控格式。

```
READ(10,FMT=C)X,Y
```

 ! 字符表达式是被赋了显式格式说明的字符变量或其它字符型表达式。本例中假设 C 是字符型变量,且已赋值 C = '(2F5.2)'。

当格式说明为第 2 项时,可以省写“FMT =”。例如:

```
READ(10,'(A3,F5.1)',IOSTAT=M)C,X
```

3. REC = rc。rc 是整型表达式,用来指定直接文件中被读写的记录的记录号。

4. IOSTAT = it。it 为整型变量,当执行 READ 语句或 WRITE 语句时,it 便有了值。若其值为负值,表示遇见文件结束信息;若为正值,表示读或写操作出错;若为零,表示正常。如无这项,读写不正常时程序停止执行。

5. ADVANCE = ae。ae 是级别说明,必须为 'YES' 或 'NO',分别表示推进式或非推进式输入输出,缺省表示 'YES'。该说明符仅用于自定格式的输入输出中。推进式输入输出表示每次读写操作以记录为最小单位,非推进式输入输出表示每次读写操作可以以记录为最小传递单位、也可以以记录中的一部分为传递单位,这可以实现多次读写操作的数据占据同一个记录(终端输入输出时的一行,见例 10-5)。

[例 10-1]从键盘上输入 N 个学生的姓名和考试成绩,把它们写入一个顺序文件中。

```
PROGRAM CREAT_FILE
```

```
IMPLICIT NONE
```

```
CHARACTER(LEN=10)::NAME
```

```

REAL::SCORE
INTEGER::N,I
OPEN(1,FILE='STUDENT.DAT',STATUS='NEW')
READ *,N
DO I=1,N
    READ(*,'(A10,F6.1)')NAME,SCORE
    WRITE(1,'(A10,F6.1)')NAME,SCORE
END DO
CLOSE(1)
END PROGRAM CREAT_FILE

```

程序中的 OPEN 语句用来建立一个新文件 STUDENT.DAT,它与文件号 1 相连接。然后输入学生的人数 N,在 DO 结构中依次输入 N 个学生的姓名(汉语拼音表示)及其考试成绩,并把它写入 1 号文件(STUDENT.DAT)中。然后用 CLOSE 语句关闭该文件,解除了文件号 1 与文件 STUDENT.DAT 的连接关系,该文件中已有了 N 个记录。

如果程序输入如下:

```

4✓
ZHANG HONG  90.0✓
LI NING      88.5✓
ZHAO MIN    100.0✓
WANG QIANG  85.5✓

```

变量 N 的值为 4,程序结束后文件中有 4 个记录,即 4 个学生的数据。如果再一次执行该程序,由于这时文件的状态已是 'OLD',打开文件的操作不会成功。这时有多种解决办法,其中较好的方法有两种:删除文件 STUDENT.DAT 以后再运行该程序,或是将打开文件时的状态改成 'REPLACE'。

[例 10-2]把例 10-1 建立的文件中的数据显示到终端显示器上。

```

PROGRAM READ_FILE
IMPLICIT NONE
CHARACTER(LEN=10)::NAME
REAL::SCORE
INTEGER::IS
OPEN(1,FILE='STUDENT.DAT',STATUS='OLD')
READ(1,'(A10,F6.1)',IOSTAT=IS)NAME,SCORE
DO WHILE(IS/=0)
    WRITE(*,'(A10,F6.1)')NAME,SCORE
    READ(1,'(A10,F6.1)',IOSTAT=IS)NAME,SCORE
END DO
CLOSE(1)
END PROGRAM READ_FILE

```

OPEN 语句用来打开文件 STUDENT.DAT,此时它的状态应是 'OLD',该文件与文件

号 10 相连。接着按例 10-1 中的程序写入时的格式把文件中的数据全部读出并显示到终端屏幕上,最后关闭文件。

10.3 顺序文件的存取

例 10-1 和例 10-2 都是对顺序文件的操作。由 OPEN 语句连接好数据文件以后,文件的指针定位在第 1 个记录之前。这时读(或写)记录时实际上就是对第 1 个记录的读(或写),该读(或写)操作完毕以后,文件指针自动指向第 2 个记录,下次读(或写)操作就是对第 2 个记录的读(或写)操作,然后文件指针自动指向第 3 个记录,……,对第 N 个记录读(或写)操作之后文件指针自动指向第 N+1 个记录,……,直到文件结尾。由于顺序文件的每个记录长度可以互不相同,因此对顺序文件只能按记录号从小到大依次进行读写,不能按记录号直接存取。

如果在实际应用中,对大号记录操作完毕之后又想对较小号记录操作该怎么办呢?我们可以将文件关闭之后再打开(文件指针自动回到第 1 个记录之前,即文件初始点),然后再依次做读(或写)操作,将文件指针移到想操作的记录之前。这显然是一种比较笨的方法。实际上,对大号记录操作完毕之后才想对较小号记录操作这一思想是直接存取操作的请求。在顺序文件中提出直接存取请求显然不能有较方便的方法,但对于一些特殊情况提供了处理方法。

10.3.1 REWIND 语句(反绕语句)

不论当前文件的指针在何处,如果想把文件定位到文件初始点,都可以通过反绕语句实现。反绕语句有两种形式:

REWIND 文件号

REWIND (定位说明符表)

定位说明符表中主要有两个说明符:

[UNIT=]文件号,此为必选项;

IOSTAT=整型变量,此为可选项,用法同 OPEN 语句。

例如:

REWIND 10 ! 使文件号为 10 的文件定位到初始点。

REWIND (2) ! 使文件号为 2 的文件定位到初始点,2 前面省略了 UNIT=。

REWIND (IOSTAT=I,UNIT=12) ! 使文件号为 12 的文件定位到初始点,I 的值表明了该反绕操作是否正确。

[例 10-3]通过例 10-2 检查文件 STUDENT.DAT 中学生的数据后发现第 K1 个记录的学生姓名有错,第 K2 个记录的学生的成绩有误。编写程序修改原来的文件。

```
PROGRAM UPDATE_SEQUENTIAL_FILE
```

```
IMPLICIT NONE
```

```
CHARACTER(LEN=10)::NAME,NEW_NAME,CFMT='(A10,F6.1)'
```

```
REAL::SCORE,NEW_SCORE
```

```

INTEGER::IS,N,I,K1,K2
OPEN(1,FILE='STUDENT.DAT',STATUS='OLD')
OPEN(2,STATUS='SCRATCH')
READ *,K1,NEW_NAME
READ *,K2,NEW_SCORE
N=0
READ(1,CFMT,IOSTAT=IS)NAME,SCORE
DO WHILE (IS==0)
    N=N+1
    IF(N==K1) NAME=NEW_NAME
    IF(N==K2) SCORE=NEW_SCORE
    WRITE(2,CFMT)NAME,SCORE
    READ(1,CFMT,IOSTAT=IS)NAME,SCORE
END DO
REWIND(1); REWIND(2)
DO I=1,N
    READ(2,CFMT)NAME,SCORE
    WRITE(1,CFMT)NAME,SCORE
    WRITE(*,CFMT)NAME,SCORE
END DO
CLOSE(1); CLOSE(2)
END PROGRAM UPDATE_SEQUENTIAL_FILE

```

由于不能对顺序文件同时读和写,因此修改顺序文件时,必须从文件头开始依次读入每个记录,把正确的记录写到另一个临时文件中;读到有错的记录时,则把修改后的内容写到临时文件中。全部修改完后,把临时文件中的内容按次序传送回原文件。上述程序中与文件号 2 连接的文件是一个临时文件。由于在打开文件操作时指定它的 STATUS 为 'SCRATCH',因此不必指定文件名,而且关闭该文件时系统会自动将它删除。

程序中由 N 记录合法记录的个数,读 1 号文件遇到变量 IS \neq 0 时表示遇到文件结束记录。这时由两个 REWIND 语句将两个文件的指针都移回文件初始处,通过带控制变量的 DO 循环(这时已知有 N 个记录)从 2 号文件中顺次读出各个记录并写入 1 号文件中(1 号文件原来的数据被冲掉)。如果某个记录不再写入 1 号文件,实际上就删除了这个记录。因此也可以用这种方法完成删除记录的操作。

10.3.2 BACKSPACE 语句(回退一个记录语句)

BACKSPACE 语句使指定文件号连接的文件在当前位置上回退一个记录。如果当前文件定位在第 N 个记录之后,执行一次 BACKSPACE 语句,就定位在第(N-1)个记录之后,因而使程序能重读(或写)第 N 个记录。这打破了原来顺序文件必须顺序存取,读写过程不能再使用前面记录的限制。如果当前文件定位在文件结束记录之后,则经 BACKSPACE 语

句后,定位退到文件结束记录之前,这又可以重新在文件中写记录。如果当前文件定位在文件初始点,执行 BACKSPACE 语句后文件指针不变。

BACKSPACE 语句有两种形式:

BACKSPACE 文件号

BACKSPACE (定位说明符表) ! 定位说明符表中的各项同 REWIND 语句。

例如:

BACKSPACE 3

将与部件号 3 连接的文件回退一个记录。

[例 10-4]读下述程序,写出输出结果。

```
PROGRAM READ_FILE
  IMPLICIT NONE
  CHARACTER(LEN=10)::NAME
  REAL::SCORE
  INTEGER::IS
  OPEN(1,FILE='STUDENT.DAT',STATUS='OLD')
  READ(1,'(A10,F6.1)',IOSTAT=IS)NAME,SCORE
  DO WHILE(IS/=0)
    WRITE(*,'(A10,F6.1)')NAME,SCORE
    READ(1,'(A10,F6.1)',IOSTAT=IS)NAME,SCORE
  END DO
  BACKSPACE(1); BACKSPACE(1)
  READ(1,'(A10,F6.1)',IOSTAT=IS)NAME,SCORE
  WRITE(*,'(A10,F6.1)')NAME,SCORE
  CLOSE(1)
END PROGRAM READ_FILE
```

假设文件 STUDENT.DAT 中仍然是例 10-1 所输入的那 4 个记录。程序中先用 DO 结构将文件中的 4 个记录读出后在终端上显示出来,这时文件指针指向文件结束记录。接着用两个 BACKSPACE(1)将文件指针倒退两个记录,指向倒数第 2 个记录(第 3 个记录)并将该记录读入后显示出来。本程序运行结果如下:

```
ZHANG HONG  90.0
LI  NING     88.5
ZHAO MIN    100.0
WANG QIANG  85.5
ZHAO MIN    100.0
```

10.3.3 END FILE 语句(结束语句)

END FILE 语句在指定文件当前位置处写上一个文件结束记录。执行该语句后,文件指针定位在文件结束记录之后,不能再读写原来连接的文件。如果一定要进行读写操作,必须先执行 BACKSPACE 语句、REWIND 语句或把该文件关闭再打开之后方能进行。END

FILE 语句的一般形式也有两种:

END FILE 文件号

END FILE (定位说明符表)

定位说明符表各项同 REWIND 语句。例 10-1 和例 10-3 中的 1 号文件都是输出文件,在关闭它们之前均可以用 END FILE(1)明确地写入一个文件结束记录。

10.4 直接文件的存取

对直接文件可以直接存取任一记录。在读写语句中通过指明记录号,就可直接存取该记录内的数据,而不必从文件的开头依次去读写。

对于直接文件,OPEN 语句的连接信息表中必须增加以下两项:

ACCESS='DIRECT',RECL=整型表达式

其中,ACCESS='DIRECT'表示采用直接存取方式打开文件,RECL=描述了直接文件中所有记录的统一长度(该长度可用整型表达式表示)。如果希望直接文件中每个记录的长度都是 60 个字符,则可以在 OPEN 语句中使用说明符

RECL=60

但在具体读写时,只要记录长度不超过 60 个字符即可。直接文件各个记录的长度必须相同,否则难以实施直接存取。

在直接文件的读写语句的信息表中必须有记录说明符,用来指定读写的记录号,其形式为:

REC=整型表达式

[例 10-5]已知某班有 $N(\leq 100)$ 名学生,每个学生选修 3 门课程。编程序将这些学生的姓名及其 3 门选修课的成绩存入直接文件中。

假设每个学生的数据由派生类型 STUDENT_TYPE 组成,全班学生的数据由这种派生类型的数组存放。程序中可以将 N 个学生的数据先读入到该数组中,然后再输出到直接文件中。程序如下:

```
PROGRAM CREATE_DIRECT_FILE
  IMPLICIT NONE
  TYPE STUDENT_TYPE
    CHARACTER(LEN=10)::NAME
    REAL,DIMENSION(1:3)::SCORE
  END TYPE STUDENT_TYPE
  TYPE(STUDENT_TYPE),DIMENSION(1:100)::STU
  INTEGER::N,I,J
  WRITE(*, '(A)', ADVANCE='NO') 'INPUT NUMBER OF STUDENTS: '
  READ(*, *) N
  DO I=1,N
    READ(*, *) STU(I)%NAME, (STU(I)%SCORE(J), J=1,3)
```

```

END DO
WRITE( *, '(4X,A10,3F6.1)') (STU(I)%NAME, &
    (STU(I)%SCORE(J),J=1,3),I=1,N)
OPEN(3,FILE='DSTU.DAT',STATUS='REPLACE',ACCESS='DIRECT',&
    FORM='FORMATTED',RECL=28)
DO I=1,N
    WRITE(3,'(A10,3F6.1)',REC=I)STU(I)%NAME,&
        (STU(I)%SCORE(J),J=1,3)
END DO
CLOSE(3)
END PROGRAM CREATE_DIRECT_FILE

```

程序中第一个 WRITE 语句输出提示 INPUT NUMBER OF STUDENTS: 时,由于使用了说明符 ADVANCE='NO',故采用非推进式输入输出。下一个 READ 语句输入的 N 值与上述提示就占据同一行。为了简化,假设 N 值为 3,其运行结果如下:

```

INPUT NUMBER OF STUDENTS: 3✓
'ZHAO HONG',80.0,90.0,70.0✓
'QIAN MING', 60.5,95.5,100.0✓
'SUN YANG',95.0,55.5,76.0✓
    ZHAO HONG  80.0  90.0  70.0
    QIAN MING  60.5  95.5 100.0
    SUN YANG   95.0  55.5  76.0

```

[例 10-6]将例 10-5 中建立的直接文件按记录号从小到大显示到终端上。程序如下:

```

PROGRAM READ_DIRECT_FILE
    IMPLICIT NONE
    CHARACTER(LEN=10)::NAME
    REAL,DIMENSION(1:3)::SCORE
    INTEGER::IOS,N
    OPEN(2,FILE='DSTU.DAT',STATUS='OLD',ACCESS='DIRECT', &
        RECL=28,FORM='FORMATTED')
    N=1
    READ(2,'(A10,3F6.1)',REC=N,IOSTAT=IOS)NAME,SCORE
    DO WHILE (IOS==0)
        WRITE( *, '(A10,3F6.1)')NAME,SCORE
        N=N+1
        READ(2,'(A10,3F6.1)',REC=N,IOSTAT=IOS)NAME,SCORE
    END DO
    CLOSE(2)
END PROGRAM READ_DIRECT_FILE

```

读直接文件 DSTU.DAT 时,可以用例 10-5 中的派生类型 STUDENT_TYPE 的变量,也可以用本例中的方法用非派生类型变量。因为将派生类型数据写入文件以后,它们的每个成员都按相应的格式进行了转换,例如姓名占 10 个字节、每个成绩占 6 个字节(小数部分占 1 个字节)。将它们再次读入内存时,只要输入表中的变量、输出格式和文件中的数据相吻合即可,不必要使用与输出时同样的派生类型变量。

另外,本例中事先并不知道文件中有多少个记录,因此难以用数组将各个记录全部读入后再输出,而只能采用读入一个记录显示一个记录的方法。

实际上,直接文件的存取操作并不需要像本节的例子中这样按记录号的次序从小到大进行,可以随意给一个记录号并对该记录进行存取。读者在本节两个例子的基础上,自己编写这样的程序并不困难。

10.5 INQUIRE 语句

INQUIRE 语句又称查询语句。按功能分为三种:按文件号查询、按文件查询和按长度查询。

10.5.1 按文件号查询与按文件查询

按文件号查询与按文件查询语句的一般形式都是

INQUIRE(查询说明表)

1. 按文件号查询

其查询说明表中必须有“UNIT=文件号”这项,当它是第一项时,可以省写 UNIT=,其它说明符根据查询选择使用。如查询某文件号是否已经与文件连接,可使用存在说明符 EXIST,查询某文件号与哪个文件连接,使用文件名说明符 NAME。

各种查询都有其专用的说明符,在说明符后写一相应类型的变量名,执行 INQUIRE 语句后,系统返回值给各变量。例如已说明变量 E 为逻辑型,N 为字符型,现要查询文件号 1 是否已与文件连接及连接的文件名,可用查询语句

INQUIRE(1,EXIST=E,NAME=N)

执行该语句后,系统给变量 E 一个逻辑值。如果文件号 1 已经与某个文件相连接,则 E 的值为 TRUE.,否则为 FALSE.。如果文件号 1 已连接,则 N 的值为已连接的文件名,否则其值为 'UNDEFINED'。例如,在执行该查询语句之前已执行 OPEN 语句使文件号 1 与文件 STUD.DAT 连接,则 E 的值为 TRUE.,N 的值为 'STUD.DAT'。

2. 按文件查询

其查询说明表中必须有“FILE=文件名”这项,其它说明符根据查询选择使用。例如查询文件是否已打开(连接)、文件的存取方式、连接的文件号等可分别使用连接(打开)说明符 OPENED、存取方式说明符 ACCESS 和文件号说明符 NUMBER 等。

设已说明 OPD 是逻辑型变量,ACS 是字符型变量,NR 是整型变量,执行查询语句

INQUIRE(FILE='STUD.DAT',OPENED=OPD,ACCESS=ACS,&
NUMBER=NR)

后,若文件 STUD.DAT 已经连接,则 OPD 的值为 TRUE.,否则为 FALSE.。若该文件是

顺序文件则 ACS 的值为 'SEQUENTIAL', 若它是直接文件则 ACS 的值为 'DIRECT', 若它未连接则 ACS 的值为 'UNDEFINED'。若文件已连接, 则 NR 为连接的文件号, 否则为 -1。如果在执行该查询语句之前已执行过语句

OPEN(2, FILE='STUD.DAT', STATUS='OLD', ACCESS='DIRECT')

而且成功了, 则 OPD 的值为 .TRUE., ACS 的值为 'DIRECT', NR 的值为 2。

按文件号查询与按文件查询的说明符项中, 说明符 UNIT 专用于按文件号查询, FILE 专用于按文件查询, 其它说明符项既可用于按文件号查询, 也可用于按文件查询。例如, 若使用查询语句

INQUIRE(FILE='LEV.DAT', EXIST=ET)

则如果文件 LEV.DAT 已连接, 那么 ET 的值为 .TRUE., 否则为 .FALSE.。下面列出其它较常用的说明符的关键字、相应的变量类型和变量的可能取值及其意义。NAMED=逻辑型变量名, 文件有文件名时, 返回值为 .TRUE., 否则为 .FALSE.。

SEQUENTIAL=字符型变量名, 返回值为 'YES' | 'NO' | 'UNKNOWN' (A|B 表示 A 或者 B, 下同), 分别表示允许顺序存取、不允许顺序存取或系统不能确定。

DIRECT=字符型变量名, 返回值为 'YES' | 'NO' | 'UNKNOWN', 分别表示允许直接存取、不允许直接存取或系统不能确定。

FORM=字符型变量名, 返回值为 'FORMATTED' | 'UNFORMATTED' | 'UNDEFINED', 分别表示文件为有格式的、无格式的或系统不能确定。

FORMATTED=字符型变量名, 返回值为 'YES' | 'NO' | 'UNKNOWN', 分别表示文件允许有格式输入输出、不允许或系统不能确定。

UNFORMATTED=字符型变量名, 返回值为 'YES' | 'NO' | 'UNKNOWN', 分别表示文件允许无格式输入输出、不允许或系统不能确定。

RECL=整型变量名, 对于直接文件, 返回值为文件记录长度; 对于顺序文件, 返回值为文件中最大记录长度(有格式文件以字符为单位)。

NEXTREC=整型变量名, 用于直接文件, 返回值为文件中刚刚存取过的记录的下一个记录号。如果文件自连接以来没有被读写过, 则返回值为 1。

BLANK=字符型变量名, 返回值为 'NULL' | 'ZERO' | 'UNDEFINED', 分别表示空格无意义、作零处理或文件未连接。

POSITION=字符型变量名, 返回值为 'REWIND' | 'APPEND' | 'ASIS' | 'UNDEFINED', 分别表示文件被连接且定位在初始点、结束点、位置未变化、没有连接或未被连接成顺序存取文件。

ACTION=字符型变量名, 返回值为 'READ' | 'WRITE' | 'READWRITE' | 'UNDEFINED', 分别表示文件被连接且只读、只写、可读可写或文件未被连接。

READ=字符型变量名, 返回值为 'YES' | 'NO' | 'UNKNOWN', 分别表示允许读、不允许读或系统不能确定。

WRITE=字符型变量名, 返回值为 'YES' | 'NO' | 'UNKNOWN', 分别表示允许写、不允许写或系统不能确定。

READWRITE=字符型变量名, 返回值为 'YES' | 'NO' | 'UNKNOWN', 分别表示允许读写、不允许读写或系统不能确定。

10.5.2 长度查询

长度查询的一般形式是:

INQUIRE(IOLNGTH= 整型变量名)输出表

其功能是查询输出表的长度,一般用在建立无格式直接存取文件之前。建立无格式直接存取文件时,可把查询所得结果作为记录长度(RECL 说明符使用)。例如,执行查询语句

INQUIRE(IOLNGTH=L)A(1:N)

后 L 的值为该输出表 A(1)、A(2)、…、A(N)的长度(字节数)。

10.6 无格式文件

到目前为止,我们涉及的都是有格式文件。计算机内部的数据都是以二进制内码(补码)形式存放。假设某处理系统的整型数据在内存中占 4 个字节,那么整数 654321 在内存中的存放如图 10-1 所示。

如果采用过去的有格式文件的方法把它写到文件中,需要给出适当的数据格式描述,以便系

00000000	00001001	11111011	11110001
----------	----------	----------	----------

图 10-1 整数 654321 的二进制存储方式

统按所给的格式进行数据加工和转换。如果采用格式输出,数据在外存要使用编码(微机一般采用 ASCII 码)形式存放,图 10-1 中的整数 654321 在外存中至少要占 6 个字节(见图 10-2)。

00110110	00110101	00110100	00110011	00110010	00110001
----------	----------	----------	----------	----------	----------

图 10-2 用 ASCII 码方式存放整数 654321

1° 把内存中的数据写到外存文件中,或把外存文件中的数据读入内存时都要进行数据格式转换。当存取的数据量较大时,系统会因数据的格式转换而耗费大量的时间,从而降低系统存取文件的效率。

2° 若采用机器内码方式存储数据,当每个数据的类型、种别等一定时,其数据所占的存储空间是一定的。而用格式输入输出时,数据所占的空间与数据的有效位数及值域有关,同一类型及种别的不同值数据占据的空间大小可能不同,这不利于采用直接存取。

3° 格式方式存储数据时往往要占据较多的空间,如 654321 至少占 6 个字节,比内码方式多 2 字节。为了使得输出数据记录的长度相同,常常需要按最大数据或最长字符串设计输入输出格式,这使得文件中存有大量的类似于空格的无效数据,浪费了许多外存空间。

综合以上原因,需要引入无格式文件。

无格式文件中的数据采用与内存中数据同样的存储方式,即采用二进制内码形式存放数据。这样,系统读写文件时不需要进行格式转换,从而加快了数据存取的速度、也节省了大量的外存空间,直接存取文件的操作也便于组织。

用 OPEN 语句打开无格式文件时,需要使用说明符

FORM = 'UNFORMATTED'

如果是顺序存取文件,省略 FORM 说明符意味着是有格式文件;如果是直接存取文件,省略 FORM 说明符意味着是无格式文件。无格式文件的读写语句中不能使用格式说明符 FMT,对它也不能做非推进式输入输出操作。

[例 10-7]已知有一批学生的姓名及其考试成绩(用成绩 <0 作为输入数据的终止标记),把这些数据写入无格式直接文件中,然后通过记录号查询学生的信息。

```
PROGRAM CREAT_ UNFORMATTED_ FILE
  IMPLICIT NONE
  CHARACTER(LEN=10)::NAME
  INTEGER::I,SCORE,N=0
  OPEN(11,FILE='UNFORM.DAT',STATUS='REPLACE',&
       ACCESS='DIRECT',FORM='UNFORMATTED',RECL=14)
  WRITE(*,'(A)')'INPUT NAME & SCORE(<0--END):'
  READ(*,'(A10,I3)')NAME,SCORE
  DO WHILE (SCORE>=0)
    N=N+1
    WRITE(11,REC=N)NAME,SCORE
    READ(*,'(A10,I3)')NAME,SCORE
  END DO
  WRITE(*,'(A)',ADVANCE='NO')'INPUT RECNO&
    (<1.OR.>N--END)='; READ *,I
  DO WHILE (I>=1.AND.I<=N)
    READ(11,REC=I)NAME,SCORE
    WRITE(*,'('NO=',I3,' NAME=',A10,' SCORE=',I3)')&
      I,NAME,SCORE
    WRITE(*,'(A)',ADVANCE='NO')'INPUT RECNO&
      (<1.OR.>N--END)='; READ *,I
  END DO
  CLOSE(11)
END PROGRAM CREAT_ UNFORMATTED_ FILE
```

打开文件时,需要给出记录长度说明符 RECL=14。由于文件每个记录中有姓名和成绩两个数据:姓名 NAME 被说明为长度为 10 的字符型变量,考试成绩 SCORE 是整型变量,我们用的处理系统整型数据默认的长度是 4 字节,所以记录的长度应选择 14。如果一时无法确定记录的实际长度,可以用以下查询语句查询记录的长度:

```
INQUIRE(IOLength=RL)NAME,SCORE ! RL 应被说明为整型变量
```

在打开文件中使用说明符 RECL=RL 告诉系统记录的长度。

无格式文件固然有许多优点,但其缺点也是比较明显的。例如,它不便于用操作系统命令直接显示其内容,不能用正文编辑程序对它录入、修改和添加数据和不便于用简单直观的方法验证它的数据是否正确等等。因此,我们必须辩证地对待这两种文件。实际应用中,要根据实际问题的需要及特点选择是采用有格式文件还是采用无格式文件。

习题 10

10-1 顺序文件和直接文件有什么区别?这两种文件在 OPEN 语句中的主要区别是什么?读、写文件这两种操作有何异同?

10-2 写出能完成下面各操作的 OPEN 语句(说明项不允许有缺省):

(1)将已存在的文件 MM.DAT 与文件号 12 连接,存取方式是顺序存取。

(2)建立一与文件号 9 连接的有格式直接存取新文件 RLD.DAT。

10-3 写出查询文件号 16 是否连接、与哪个文件连接及是否允许直接存取的 INQUIRE 语句。

10-4 从键盘上输入 5 个字符串(每个字符串最大长度是 20 个字符),将其中的小写字母都转换成大写字母,并把转换后的每个字符串作为一个记录存放到顺序文件 TRA.DAT 中。

10-5 有两个文件 A.DAT 和 B.DAT,各存放一个无序的字符串。编一程序,将这两个文件中的字符串合并(按字符顺序从小到大排列)后输出到一个新文件 C.DAT 中。

10-6 产生 100 个 50 到 1000 之间的随机整数,把它们存放到文件 RAN.DAT 中,然后读出后 50 个数,并选出其中的素数放到文件 PRIME.DAT 中。

10-7 键盘输入 10 个学生的学号、姓名和两门课成绩,将上述数据及每个学生的平均成绩写入文件 STUD.DAT 中。要求分别用顺序文件和直接文件实现。

10-8 将上题文件 STUD.DAT 中平均成绩高于 80 的学生的姓名、学号及平均成绩显示出来。

10-9 将文件 STUD.DAT 中的学生数据按平均成绩进行降序排序,并将已排序的学生数据存到一个新文件 SORT.DAT 中。

10-10 在上题已排序的学生成绩文件 SORT.DAT 中插入一个学生的数据,使插入后的文件内容仍按序排列。要求分别用顺序文件和直接文件实现。

10-11 将满足下列条件之一的记录从上题的文件 SORT.DAT 中删除(分别用顺序文件和直接文件实现):

(1)姓名为 ZHANG LING;

(2)学号为 101 或 107。

10-12 运行例 10-5 中的程序,输入 6 个学生的信息。然后编写一个程序,按记录号查询学生的数据,直到输入的记录号为 -1 时停止。

10-13 编一程序,查询文件 UVW.DAT(假设每个记录由学号和考试成绩两项组成)是否存在、已连接否、与之连接的文件号是什么、直接存取还是顺序存取。根据查询结果,如果文件未连接,则将其与文件号 18 连接,然后读出并显示该文件的内容;如果已连接,则直接进行上述显示输出。

第 11 章 FORTRAN77 的过时特性

FORTRAN77 在科学计算等领域有着十分广泛的用途。但由于它的某些过时特性已不能适应现代程序设计思想的需要,在 FORTRAN90 中已由较先进的现代特性所代替。考虑到现存的大量 FORTRAN77 程序中都使用了这些过时的特性,为了使 FORTRAN90 程序员也能方便地阅读 FORTRAN77 程序,并能将它修改成具有 FORTRAN90 特性的程序,需要介绍 FORTRAN90 中不再提倡使用的这些过时的特性。

11.1 过时的书写格式

FORTRAN77 中与程序书写有关的程序分区、使用语句标号、数据块子程序及程序单元首语句的书写方式等都属于过时特性。

1. 语句行的过时格式

FORTRAN77 的程序行分成语句行和非语句行(注释行)。其中每个语句行有 80 列,划分成四个区。第 1 列至第 5 列是标号区,书写语句标号;第 6 列是续行区,在该列上放一个非零或非空格字符表示本行是上一语句行的继续;第 7 列至第 72 列是语句区,用于书写语句成分;第 73 列至第 80 列是注解区,用作注解用。另外,凡第 1 列为“*”或“C”的行全为注释行,这些行的内容可以占用全行的 80 列。注释行及注解区的内容不参加编译。

FORTRAN77 的书写格式过于严格,程序员更喜欢 FORTRAN90 中那种比较随意、易掌握的书写格式。

2. 过时的语句标号

FORTRAN77 中各语句可以根据需要而设置语句标号。每个标号是 1 到 5 位无符号整数,要放在标号区。标号的主要作用是用于实现程序控制的转移等,这就使程序员可以任意地通过引用标号而使程序流程随意转向,既破坏了程序的整体结构,使程序阅读十分困难,也不便于程序的调试与维护。FORTRAN90 中不提倡使用语句标号,控制转移一律通过控制结构实现。

3. 不应再使用数据块子程序

FORTRAN77 中的数据块子程序专门用于给有名公用区中的变量赋初值。它以 BLOCK DATA 开头,以 END 结束,里面没有可执行语句,全部都是用来说明有名公用区中成员的说明语句和 DATA 语句。假设有两个简单变量 A、B 和一个整型数组 NUM 全都放在一个有名公用区(名为 TABLE)内,要给 A、B 分别置初值 5、8,数组 NUM 中 10 个元素分别赋初值 1、2、…、10,必须使用数据块子程序:

```
BLOCK DATA AA
  DIMENSION NUM(10)
  COMMON/TABLE/A,B,NUM
  DATA A,B/5,8/
  DATA NUM/1,2,3,4,5,6,7,8,9,10/
```

END

这种数据块子程序编写花费大、作用小,而且它所使用的 COMMON 语句、DATA 语句也属于过时特性。因此数据块子程序也应淘汰,它要做的工作可由 FORTRAN90 的模块子程序完成。

4. 主程序与函数子程序语句的落后方式

在 FORTRAN77 中主程序单元的首行语句(PROGRAM 语句)是可选的,这使得主程序单元不能与其它程序单元的形式相统一,阅读时也难以发现主程序单元从何处开始,因此 FORTRAN90 中的主程序应统一写上 PROGRAM 语句。

在 FORTRAN77 中函数子程序语句的一般形式是:

[函数类型] FUNCTION 函数名([虚参表])

这里函数名既作为函数子程序名,又代表函数返回值变量,这种用法欠科学,应使用 FORTRAN90 中具有现代特性的函数子程序语句,把函数名和函数值分开。

11.2 过时的控制转移方式

由于语句标号已是过时特性,使用它会带来程序结构混乱、易读性差、不便于维护等问题,因此与标号有关的以下各语句或其某种使用方法是不提倡使用的:

1. GO TO 语句

GO TO 语句是无条件转移语句,它的形式是:

GO TO n

这里 n 是一个可执行语句的标号。例如:

GO TO 10

就是使控制跳过一段程序,转移到标号为 10 的那个语句处继续执行。例如程序段

```
K=0
10    K=K+1
      PRINT *,K
      GO TO 10
```

将输出 1、2、3、...各个值。

2. ASSIGN 语句

ASSIGN 语句又称标号赋值语句,它能将一个语句标号赋给一个变量供 GO TO 语句使用。例如:

ASSIGN 10 TO M

就使 M 成为标号变量,并且语句标号值为 10,以后再执行

GO TO M

时,就相当于执行了

GO TO 10

3. 算术 IF 语句

算术 IF 语句提供了三个分支转移手段,执行哪个分支取决于算术表达式的值是小于、等于还是大于零。算术 IF 语句的一般形式是:

IF(e)k1,k2,k3

其中 e 是一个算术表达式, k_1, k_2, k_3 分别代表本程序单位中的三个语句标号。当 e 的值小于零时, 控制转向标号为 k_1 的语句处执行; 当 e 的值等于零时, 控制转向标号为 k_2 的语句处执行; 当 e 的值大于零时, 控制转向标号为 k_3 的语句去执行。例如按以下公式计算 y 的值时:

$$y = \begin{cases} x & (x > 1) \\ 0 & (x = 1) \\ x^2 & (x < 1) \end{cases}$$

可写出以下程序:

```
      READ( *, * ) X
      IF(X-1.0) 10, 20, 30
10     Y = X * X
      GO TO 40
20     Y = 0.0
      GO TO 40
30     Y = X
40     PRINT *, X, Y
      END
```

4. 计算 GO TO 语句

计算 GO TO 语句也是用于控制转移。它的一般形式是:

GO TO(k_1, k_2, \dots, k_n), i

其中 i 是一个整形表达式, i 的值应该为 1 到 n , 当 i 的值为 1 时, 转向标号为 k_1 的语句; 当 i 的值为 2 时, 转向标号为 k_2 的语句; ...; 当 i 的值为 n 时, 转向标号为 k_n 的语句。

5. 带标号的 DO 语句与 CONTINUE 语句

FORTRAN77 中 DO 语句的一般形式是:

DO k [,] $i = e_1, e_2$ [, e_3]

其中 k 为语句标号, 与循环终端的语句标号一致, i 是循环控制变量, e_1, e_2, e_3 分别为循环变量的初值、终值和增值。由于有一些语句(如 END 语句、GO TO 语句、DO 语句、END IF 语句等)不能用作循环的终端语句, 因此常使用 CONTINUE 作为循环的终端语句。CONTINUE 语句是可执行语句, 它的作用是使程序流程转到它的下一个语句。例如:

```
      DO 10 N = 1, 100
      X = N/2.0
      PRINT *, X
10     CONTINUE
```

6. 数据传输等语句的 ERR、END 说明符

在 READ 语句、WRITE 语句以及其它数据传输辅助语句(如 OPEN 语句、CLOSE 语句及定位语句等)中有以下两个说明符:

ERR = 语句标号

END = 语句标号

前者表示若执行本语句时出错则转向“=”后的语句标号处执行; 后者表示如执行本语句遇到文件结束信息时转向指定标号处执行。它们的功能可由现代特性的 IOSTAT 说明符来

代替。

11.3 数据说明的过时形式

以下各种数据说明语句或者在 FORTRAN90 中已有更好的说明方法,或者存有这样那样的不足,应予淘汰或不再使用。

1. 隐式说明形式

FORTRAN77 中有两种隐含的数据类型说明形式。其一是 I-N 规则:规定凡名字以 I 到 N 这 6 个字母开头的变量均自动作为整型变量,以其它字母开头的变量作为实型变量。该规则既不利于以物理意义给变量取名,名字写错时也不便于检查。FORTRAN90 中用语句 IMPLICIT NONE 向系统声明不允许用上述隐含说明。

另外一种隐含说明即 IMPLICIT 语句(隐含说明语句),它可以规定以某些字母开头的变量的类型。例如语句

```
IMPLICIT INTEGER(A,B),REAL(C,M)
```

规定了以 A 和 B 开头的变量都是整型变量,以 C 和 M 开头的变量都是实型变量。这种说明不利于程序的阅读及维护,因此应淘汰 FORTRAN77 中的 IMPLICIT 语句。

2. 类型说明语句的过时形式

旧的类型说明语句举例如下:

```
INTEGER X,Y           (说明 X 和 Y 都是整型变量)
```

```
REAL I,K              (说明 I 和 K 都是实型变量)
```

```
CHARACTER * 10 B,A * 5 (说明 B 是长度为 10、A 是长度为 5 的字符型变量)
```

这种说明语句不能表达种别参数以及各种属性信息。为了表示它们的其它属性,还必须使用其它说明语句,如 PARAMETER 语句、DIMENSION 语句等。但这种说明方式使变量及其属性的说明分散在不同的说明语句之中,使程序员很难对一个变量有一个全局的了解,这种旧的类型说明语句形式显然已经过时。

3. 双精度类型

FORTRAN77 中使用实型数据时,为了满足某些计算精度的要求设置了双精度类型。例如语句

```
DOUBLE PRECISION D,F
```

说明了变量 D 和 F 是双精度类型变量。在微机的 FORTRAN77 系统中一般它可具有 15 至 16 位有效数字。

由于 FORTRAN90 中实型数据有若干种别,不同种别采用不同位数的有效数位,双精度型只是实型种别中的一种,因此双精度型的存在已没有意义。

4. 数据赋初值语句

DATA 语句专门用来在编译期间给变量赋初值。例如

```
DATA A,B,C/1.0,2.0,3.0/
```

表示在编译时给 A 赋初值 1.0、给 B 赋初值 2.0、给 C 赋初值 3.0。但 FORTRAN 语言中无法说明 DATA 语句是说明语句还是可执行语句,所以在 FORTRAN90 中不宜再使用,它的功能可以由类型说明语句完成。例如下述类型说明语句就包含了上述 DATA 语句的功能:

```
REAL::A=1.0,B=2.0,C=3.0
```


5. EQUIVALENCE 语句

EQUIVALENCE 语句也叫等价语句,其主要用途是节约内存空间,使一些不交叉使用的变量共用存储单元。例如:

```
EQUIVALENCE(A,B,C)
```

使变量 A、B 和 C 都存储在同一个内存单元中,即 A、B 和 C 的值是相同的(如果它们的类型、种别及属性都相同的话)。若变量 A 的值改变,则 B 和 C 的值也随之变化,反之亦然。

由于 FORTRAN90 中的模块、动态存储分配以及指针等完全可以代替 EQUIVALENCE 语句的功能,EQUIVALENCE 语句没有存在的必要。

6. COMMON 语句

COMMON 语句也叫公用语句,其主要用途是通过在不同的程序单元之间建立公用区来达到变量共享存储单元、从而实现数据传递的目的。假设有以下程序:

```
SUBROUTINE SUB
```

```
COMMON C,D
```

```
PRINT *,C,D
```

```
END
```

```
PROGRAM MAIN
```

```
COMMON A,B
```

```
READ *,A,B
```

```
CALL SUB
```

```
END
```

主程序和子程序中的两个 COMMON 语句说明了一个公用区,使得主程序中的变量 A 与子程序中的变量 C 共用同一个存储单元,主程序中的变量 B 与子程序中的变量 D 占用同一个存储单元。这样,如果主程序读入的值使得 A=1.0、B=2.0 时,则在子程序 SUB 中输出的 C 和 D 的值也分别是 1.0 和 2.0。

上面 COMMON 语句建立的是无名公用区。如果把 COMMON 中的变量划分成许多组,每组取一个名字,这就形成了有名公用区。有名公用区的名字用两个斜杠“/”夹起来。假设主程序中有语句

```
COMMON /D/A,B,/X/U,Y,Z
```

子程序中的公用语句为

```
COMMON /D/C,D,/X/W,G,B
```

它们说明了两个有名公用区 D 和 X,公用区中数据的传递是在同名公用区间进行的。

COMMON 语句可以在主程序与子程序间使用,也可以在子程序单元之间使用。不同的程序单元中同名的有名公用区的长度必须一致,而无名公用区的长度可以不同。

利用公用区传递数据很方便、传递速度也很快,但使用时的限制较多。FORTRAN90 的模块完全包含了 COMMON 语句的全部功能,并且不易出错,故 COMMON 语句应予以淘汰。

11.4 程序中的过时功能

FORTRAN77 中的语句函数、函数的特定名、假定大小数组和子程序的多重入口等都

属于过时的功能。

1. 语句函数

FORTRAN77 中的语句函数的定义形式如下:

函数名(a1,a2,...,an)=表达式

其中 a1,a2,...,an 是函数的自变量。语句函数的定义应写在说明部分之后、执行部分之前,并且仅能供本程序单元引用。引用方式与引用函数子程序类似,也是在表达式中写上:

语句函数名(实元表)

这种语句函数功能不强,而且它既不属于说明部分,也不属于执行部分,其功能也可以由内部函数形式实现,故不提倡使用。

2. 函数的特定名

FORTRAN77 中有的内在函数名只能对一种数据类型适用,因此称此类函数名为函数的特定名。如 IABS、ABS、DABS、CABS 这 4 个取绝对值的函数适用于不同的数据类型,求整数的绝对值用 IABS,求实数的绝对值用 ABS,求双精度数的绝对值用 DABS,求复数的绝对值用 CABS。

这种特定名的使用方法复杂,也不便于记忆,不应提倡使用,实际应用中可以使用内在函数的类属名。函数的类属名能对多种类型的自变量值进行操作。例如求绝对值函数的类属名是 ABS,它可以接受各种数值型的自变量数据,用起来十分方便。

3. 假定大小数组

假定大小数组只能作为子程序中的虚元数组,把数组名写在虚元表中,可以用“*”号作为虚元数组的数组说明符中最后一个维定义符的上界,它的作用是使所定义的虚元数组的大小与相对应的实元数组的大小完全相同。例如:

DIMENSION DA(-5:*)

INTEGER DB(1:3,2:*)

假定大小数组的功能完全可以用 FORTRAN90 中的假定形状数组实现,因此不宜再使用。

4. 多重入口

子程序的多重入口可以使函数子程序或子例子程序的控制从子程序中间指定地方开始执行,但需要在这些入口处使用 ENTRY 语句。ENTRY 语句的一般形式是:

ENTRY 入口名(虚元表)

调用时与调用函数子程序或子例子程序一样,只是需要将函数名或子例子程序名换上入口名。以下是具有多重入口的子例子程序:

SUBROUTINE SUB(X,Y)

...

ENTRY SUBXY(X,Y)

...

ENTRY SUBY(Y)

...

END

该程序共有 3 个入口:SUB、SUBXY 和 SUBY,并且各有自己的虚元表。如果主调程序中调用语句为

CALL SUBY(Y1)

则控制直接从 ENTRY SUBY(Y)语句处进入,而后依次往下执行。

多重入口虽然能缩短程序的长度,但使程序的可维护性大大降低,破坏了程序的结构化,故应予以淘汰。

5. 交错返回

交错返回是子程序有多个返回点。例如:

```
SUBROUTINE SUB(A, *, *)
  IF(A<=0)THEN
    RETURN 1
  ELSE
    RETURN 2
  END IF
END
PROGRAM MAIN
  READ *, X
  CALL SUB(X, * 100, * 200)
100  Y = X * X
      GO TO 300
200  Y = SQRT(X)
300  PRINT *, X, Y
      END
```

如果主程序输入的 X 值大于 0 时,子程序从 RETURN 2 处返回到第 2 个“*”所对应的实元 * 200 所代表的语句,即执行语句 Y = SQRT(X);否则子程序从 RETURN 1 处返回到第 1 个“*”所对应的实元 * 100 所代表的语句,即执行语句 Y = X * X。这种用法既不容易理解,也难以控制程序的流程,应予以淘汰。

11.5 其它过时语句

FORTARN 77 中还有一些语句或语句的部分用法也属于过时的特性。

1. FORMAT 语句

FORMAT 语句又称格式语句,用来提供具体的格式信息。例如语句序列

```
      READ( *, 10)I, X
10    FORMAT(I5, F6.2)
```

等同于语句

```
      READ( *, '(I5, F6.2)')I, X
```

FORMAT 语句既不是说明语句,也不是执行语句,还要使用落后的语句标号,因此应予以淘汰。

2. PARAMETER 语句

PARAMETER 语句一般形式如下:

```
PARAMETER(P1 = C1, P2 = C2, ..., Pn = Cn)
```

其中 P1, P2, ..., Pn 是符号常数名, C1, C2, ..., Cn 是由常数和已有定义的符号常数所组成的常数表达式。

使用符号常数可以增强程序的可读性、可维护性和可移植性,并能减少程序出错的机会。但 FORTRAN90 中进行类型说明时,只需要使用 PARAMETER 属性就可以完成 PARAMETER 语句的功能,因此应淘汰 PARAMETER 语句。

3. DIMENSION 语句

FORTRAN77 中可以使用 DIMENSION 语句对数组进行定义。例如

```
DIMENSION A(120)
```

定义了有 120 个元素的数组 A。但 DIMENSION 语句只能定义数组的大小不能说明数组的类型,不如在 FORTRAN90 的类型说明语句中添加 DIMENSION 属性来说明。

4. RETURN 语句

RETURN 语句用在子程序中,表示返回到调用它的程序单元。FORTRAN77 子程序中的 END 语句同时起到了返回调用程序单元的作用,因此 RETURN 语句已经多余。RETURN 语句的多重返回形式,更加搞乱了程序的结构。因此不应再使用 RETURN 语句。

5. STOP 语句

它的作用是“停止运行”。由于 END 语句已经具有停止程序执行的功能,因此 STOP 语句是多余的,应予以淘汰。

6. PAUSE 语句

它的作用是“暂停执行程序”,程序员通过相应的命令可以使之从断点处恢复执行。这一功能可以由现代特性的输入输出语句代替,所以应予以淘汰。

7. DO 语句中的实型循环变量

DO 语句中的循环变量、循环变量的初值、终值和增值理论上可以是整型值,也可以是实型值。但实型值在计算机中的计算和存储可能有误差,因此实际的循环次数可能与理论值不吻合,故循环变量的实型值应予以淘汰。遇到这种情况,可设法把它们转化为整型后处理。例如程序段

```
DO X=0.0,3.14159,3.14159/18
  PRINT *,X,SIN(X)
END DO
```

可以改写为

```
DO I=0,180,10
  X=I*3.14159/180
  PRINT *,X,SIN(X)
END DO
```

8. 关系运算符

FORTRAN77 中的关系运算符是 .EQ. (等于)、.NE. (不等于)、.LT. (小于)、.LE. (小于或等于)、.GT. (大于)和 .GE. (大于或等于),FORTRAN90 中新设置的关系运算符分别是 == (等于)、/= (不等于)、< (小于)、<= (小于或等于)、> (大于)、>= (大于或等于),这比 FORTRAN77 中用 4 个字符(包括两边的两个点)表示关系运算符更为直观和简捷,因此用 4 个字符表示关系运算符的形式应予以淘汰。

9. Hollerith 数据编辑符与字符串编辑符

FORTRAN77 中 Hollerith 常数可以在格式说明中作为编辑符。例如：

```
      WRITE( *,100)N  
100    FORMAT(2HN=,I5)
```

如果 N 的值是 123,则输出

N= 123

这里的 Hollerith 常数是 2HN=,表示编辑符 'N=',输出语句遇到这种编辑符会照打 Hollerith 常数文字。但它要逐个数字符个数,而且字符个数与 H 前边的数不吻合时,Hollerith 编辑符将出错,因此应予以淘汰。

FORTRAN77 中还有字符串编辑符,即撇号中的内容按原样输出,例如:

```
      WRITE( *,100)N  
100    FORMAT('N=',I5)
```

将输出同样的结果。但这种形式也应该淘汰,最好把输出的字符串放入输出表中。由于 FORMAT 语句也是过时的,则 WRITE 语句应写成以下形式:

```
      WRITE( *,FMT='(A,I5)')'N=',N
```

附录 I ASCII 字符集

[illegible]

附录 II FORTRAN90 的语法描述

为了便于理解和阅读,本附录给出了 FORTRAN90 的语句排列次序,并采用基于 BNF 的方法分程序单元、说明部分、执行部分和动作语句四部分对 FORTRAN90 的主要语法框架进行描述。其中[]表示其中的内容可选,⋯表示前边的语法表述可重复多次,A|B表示必须选 A 或 B 中之一。其中⋯前边是逗号时,表示⋯前边的内容可以重复多次,但最后一次重复之后不带逗号。例如下述描述

对象名,⋯

表示“对象名,”可以重复多次,最后一个对象名之后不带逗号,它和以下描述

对象名[,对象名]⋯

是相同的。有时也用以下方法描述

对象名,对象名,⋯,对象名

1. 语句排列次序

FORTRAN90 各语句的排列次序按下表所示:

PROGRAM语句、FUNCTION语句、SUBROUTINE语句、MODULE语句、BLOCK DATA语句		
USE语句		
FORMAT语句 和 ENTRY语句	IMPLICIT NONE	
	PARAMETER语句	IMPLICIT语句
	PARAMETER语句 和 DATA语句	导出类型定义、接口块、 类型声明语句、语句函数语句、 说明语句
	DATA语句	可执行结构
CONTAINS语句		
函数子程序、子例子程序		
END语句		

2. 程序单元

可执行程序是: 程序单元
[程序单元]

⋯

程序单元是: 主程序
或 子例子程序
或 函数子程序
或 模块
或 数据块

主程序是: [PROGRAM 主程序名]
说明部分
执行部分
[内部子程序部分]
END [PROGRAM [程序名]]

子例子程序是: [RECURSIVE] SUBROUTINE 子程序名[([虚元表])]

[ENTRY 语句]…
 [语句函数语句]…

派生类型定义是: TYPE [[,PUBLIC|PRIVATE]::]派生类型名
 [SEQUENCE][PRIVATE]…
 类型关键字[[,属性说明]…:]成员名, …
 …

接口块是: END TYPE [派生类型名]
 INTERFACE [类属名|OPERATOR(定义操作符)|ASSIGNMENT(=)]
 FUNCTION 语句|SUBROUTINE 语句
 说明部分
 END [FUNCTION [函数名]]|END [SUBROUTINE [子程序名]]
 …
 [MODULE PROCEDURE 过程名, …]
 …

类型说明语句是: END INTERFACE
 类型说明 [[,属性说明]…:]实体名, …

类型说明是: INTEGER [([KIND=]整型表达式)]
 或 REAL [([KIND=]整型表达式)]
 或 DOUBLE PRECISION
 或 COMPLEX [([KIND=]整型表达式)]
 或 CHARACTER [([LEN=]长度值)]
 或 CHARACTER [* 长度值[,]]
 或 CHARACTER [[LEN=]长度值, [KIND=]整型表达式]
 或 CHARACTER [[KIND=]整型表达式, [LEN=]长度值]
 或 LOGICAL [([KIND=]整型表达式)]
 或 TYPE (派生类型名)

属性说明是: PARAMETER
 或 PUBLIC|PRIVATE
 或 ALLOCATABLE
 或 DIMENSION(常数组形状描述)
 或 DIMENSION(假定形状数组描述)
 或 DIMENSION(动态数组形状描述)
 或 DIMENSION(假定大小数组描述)
 或 EXTERNAL
 或 INTENT(IN|OUT|INOUT)
 或 INTRINSIC
 或 OPTIONAL
 或 POINTER
 或 SAVE
 或 TARGET

实体描述是: 对象名 [(数组描述)][* 字符长度][= 初始表达式]
 或 函数名 [(数组描述)][* 字符长度]

属性描述语句是: 访问语句
 或 ALLOCATABLE 语句

或 COMMON 语句
或 DATA 语句
或 DIMENSION 语句
或 EQUIVALENCE 语句
或 EXTERNAL 语句
或 INTENT 语句
或 INTRINSIC 语句
或 NAMELIST 语句
或 OPTIONAL 语句
或 POINTER 语句
或 SAVE 语句
或 TARGET 语句
访问语句是: PUBLIC[[:]模块变量名,...]
或 PRIVATE[[:]模块变量名,...]
ALLOCATABLE 语句是: ALLOCATABLE[[:]数组名[动态数组描述],...
COMMON 语句是: COMMON[/公用块名]/对象名表,[,]/[公用块名]/对象名表]...
DATA 语句是: DATA 对象表/初值表/[[,对象表/初值表/]]...
对象表是: 变量名[,变量名]...
或 隐含 DO 循环表
初值表是: [重复系数 *]常数,...
或 结构构造器
或 字符常数
DIMENSION 语句是: DIMENSION[[:]数组名(数组描述)[,数组名(数组描述)]...
EQUIVALENCE 语句是: EQUIVALENCE(等价对象,等价对象,...)[,等价对象,等价对象,...]...
等价对象是: 变量名
或 数组元素
或 子串
EXTERNAL 语句是: EXTERNAL 外部名[,外部名]...
外部名是: 外部过程名
或 虚元名
或 块数据程序单元名
INTENT 语句是: INTENT(IN|OUT|INOUT)[[:]虚元名[,虚元名]]...
INTRINSIC 语句是: INTRINSIC 内部过程名[,内部过程名]...
NAMELIST 语句是: NAMELIST/名字表名/对象名[,对象名]...[[,]
/名字表名/对象名[,对象名]...]
OPTIONAL 语句是: OPTIONAL[[:]虚元名[,虚元名]]...
POINTER 语句是: POINTER[[:]对象名[(动态形状描述表)]
[,对象名[(动态形状描述表)]]...
SAVE 语句是: SAVE[[:]储存实体[,储存实体]]...
储存实体是: 对象名
或 /公用块名/
TARGET 语句是: TARGET[[:]对象名[(数组描述)][,对象名[(数组描述)]]...
语句函数语句是: 语句函数([虚元表])=表达式

4. 执行部分

执行部分是： 可执行结构
 [执行部分结构]

可执行结构是： 动作语句
 或 CASE 结构
 或 DO 结构
 或 IF 结构
 或 WHERE 结构

执行部分结构是： 可执行结构
 或 FORMAT 语句
 或 DATA 语句
 或 ENTRY 语句

CASE 结构是： [结构名:]SELECT CASE (情况表达式)
 CASE(情况选择器 1) [结构名]
 语句块 1
 ...
 CASE(情况选择器 n) [结构名]
 语句块 n
 [CASE DEFAULT [结构名]
 语句块 n+1]
 END SELECT [结构名]

DO 结构是： [结构名:]DO[[,]循环变量名=循环初值,循环终值[,循环增值]]
 语句块
 END DO [结构名]

 或 [结构名:]DO 语句标号[[,]循环变量名=循环初值,循环终值[,循环增值]]
 语句块
 标号 CONTINUE

 或 [结构名:]DO WHILE(逻辑表达式)
 语句块
 END DO [结构名]

IF 结构是： [结构名:]IF(逻辑表达式 1)THEN
 语句块 1
 [ELSE IF(逻辑表达式 2)THEN [结构名]
 语句块 2]
 ...
 [ELSE IF(逻辑表达式 n)THEN [结构名]
 语句块 n]
 [ELSE [结构名]
 语句块 n+1]
 END IF [结构名]

WHERE 结构是： WHERE(屏蔽表达式)
 [数组赋值语句]
 ...
 [ELSE WHERE
 [数组赋值语句]

...]
END WHERE

5. 动作语句

动作语句是: ALLOCATE 语句
或 赋值语句
或 BACKSPACE 语句
或 CALL 语句
或 CLOSE 语句
或 计算 GO TO 语句
或 CONTINUE 语句
或 CYCLE 语句
或 DEALLOCATE 语句
或 ENDFILE 语句
或 函数结束语句
或 主程序结束语句
或 子例子程序结束语句
或 EXIT 语句
或 GO TO 语句
或 IF 语句
或 INQUIRE 语句
或 NULLIFY 语句
或 OPEN 语句
或 POINTER 赋值语句
或 PRINT 语句
或 READ 语句
或 RETURN 语句
或 REWIND 语句
或 STOP 语句
或 WHERE 语句
或 WRITE 语句
或 算术 IF 语句
或 标号赋值语句
或 赋值 GO TO 语句
或 PAUSE 语句

ALLOCATE 语句是: ALLOCATE(分配对象[(分配形状描述表)][,STAT=整型变量])
对象为变量名或结构成员名

赋值语句是: 变量名 = 表达式

BACKSPACE 语句是: BACKSPACE 外部文件部件号
或 BACKSPACE([UNIT=]外部文件部件号[,ERR=标号]
[,IOSTAT=整型变量])

CALL 语句是: CALL 子例子程序名[([实元表])]

实元表形式是: [虚元名=]实元名

或 表达式

或 过程名

或	* 语句标号
CLOSE 语句是:	CLOSE([UNIT=]部件号[,IOSTAT=整型变量][,ERR=标号] [,STATUS=字符串])
计算 GO TO 语句是:	GO TO (语句标号表)[,]整型表达式
CONTINUE 语句是:	CONTINUE
CYCLE 语句是:	CYCLE [DO 结构名]
DEALLOCATE 语句是:	DEALLOCATE(分配对象[,分配对象]...[,STAT=整型变量])
ENDFILE 语句是:	ENDFILE 部件号
或	ENDFILE([UNIT=部件号[,ERR=标号][,IOSTAT=整型变量])
函数结束语句是:	END [FUNCTION [函数名]]
主程序结束语句是:	END [PROGRAM [主程序名]]
子例子程序结束语句是:	END [SUBROUTINE [子程序名]]
EXIT 语句是:	EXIT [DO 结构名]
GO TO 语句是:	GO TO 语句标号
IF 语句是:	IF(逻辑表达式)动作语句
INQUIRE 语句是:	INQUIRE(查询说明符表)
或	INQUIRE(IOLENGTH=整型变量)输出表
查询说明符是:	[UNIT=]部件号 FILE=文件名
或	IOSTAT=整型变量
或	ERR=标号
或	EXIST=逻辑变量
或	OPENED=逻辑变量
或	NUMBER=整型变量
或	NAMED=逻辑变量
或	NAME=字符变量
或	ACCESS=字符变量
或	SEQUENTIAL=字符变量
或	DIRECT=字符变量
或	FORM=字符变量
或	FORMATTED=字符变量
或	UNFORMATTED=字符变量
或	RECL=逻辑变量
或	NEXTREC=逻辑变量
或	BLANK=字符变量
或	POSITION=字符变量
或	ACTION=字符变量
或	READ=字符变量
或	WRITE=字符变量
或	READWRITE=字符变量
或	DELIM=字符变量
或	PAD=字符变量
NULLIFY 语句是:	NULLIFY(指针[,指针]...)
OPEN 语句是:	OPEN(连接说明符表)
连接说明符是:	[UNIT=]部件号

或 IOSTAT= 整型变量
 或 ERR= 标号
 或 FILE= 文件名
 或 STATUS= 字符变量
 或 ACCESS= 字符变量
 或 FORM= 字符变量
 或 RECL= 整型变量
 或 BLANK= 字符变量
 或 POSITION= 字符变量
 或 ACTION= 字符变量
 或 DELIM= 字符变量
 或 PAD= 字符变量

POINTER 语句是: POINTER[::]对象名[(动态形状描述表)]
 [,对象名[(动态形状描述表)]]...

PRINT 语句是: PRINT 格式[,输出表]

READ 语句是: READ 格式[,输入表]

或 READ(输入输出控制说明符表)[输入表]

输入输出控制说明符是:

[UNIT=]部件号
 或 [FMT=]格式
 或 [NML=]名字表名
 或 REC= 整型变量
 或 IOSTAT= 整型变量
 或 ERR= 标号
 或 END= 标号
 或 ADVANCE= 字符表达式
 或 SIZE= 整型变量
 或 EOR= 标号

RETURN 语句是: RETURN [整型表达式]

REWIND 语句是: REWIND 部件号

或 REWIND([UNIT=]部件号[,IOSTAT= 整型变量][,ERR= 标号])

STOP 语句是: STOP [字符常数]

或 STOP 小于或等于 5 位的无符号整数

WHERE 语句是: WHERE(屏蔽表达式)赋值语句

WRITE 语句是: WRITE(输入输出控制说明符表)[输出表]

算术 IF 语句是: IF(数值表达式)标号,标号,标号

标号赋值语句是: ASSIGN 标号 TO 整型变量

赋值 GOTO 语句是: GO TO 整型变量[[,](标号表)]

PAUSE 语句是: PAUSE [字符常数]

或 PAUSE 小于或等于 5 位的无符号整数

附录Ⅲ FORTRAN90 的内在过程

1. 内在函数

函数分类	函数名称	函数功能	变元类型	结果类型
数值函数	ABS(A)	求 A 的绝对值	整型 实型或复型	整型 实型
	AIMAG(Z)	求复数的虚部	复型	实型
	AINTE(A[,KIND])	截断成整数	实型	实型
	ANINT(A[,KIND])	最接近的整数	实型	实型
	CEILING(A)	大于或等于数 A 的最小整数	实型	默认整型
	CMPLX(X[,Y][,KIND])	把两个(或一个)数转换成复数	X 整型、实型或复型 Y 整型或实型	复型
	CONJG(Z)	求共轭复数	复型	复型
	DBLE(A)	转换成双精度实型数	整型、实型或复型	双精度实型
	DIM(X,Y)	求正差	整型或实型	与变元相同
	DPROD(X,Y)	双精度实数求积	默认实型	双精度实型
	FLOOR(A)	小于或等于原数的最大整数	实型	默认整型
	INT(A[,KIND])	转换成整型数	整型、实型或复型	整型
	MAX(A1,A2[,A3]...)	求最大值	整型或实型	与变元相同
	MIN(A1,A2[,A3]...)	求最小值	整型或实型	与变元相同
	MOD(A,P)	求余数	整型或实型	与变元相同
	MODULE(A,P)	模函数	整型或实型	与变元相同
	NINT(A[,KIND])	最近的整数	实型	整型
	REAL(A[,KIND])	转换成实型	整型、实型或复型	实型
	SIGN(A,B)	传送符号	整型或实型	与变元相同
数学函数	ACOS(X)	求反余弦	实型,且 $ x \leq 1$	实型
	ASIN(X)	求反正弦	实型,且 $ x \leq 1$	实型
	ATAN(X)	求反正切	实型	实型
	ATAN2(Y,X)	求反正切	实型	实型
	COS(X)	求余弦	实型或复型	与 X 相同
	COSH(X)	双曲余弦	实型	与 X 相同
	EXP(X)	指数	实型或复型	与 X 相同
	LOG(X)	自然对数	实型或复型	与 X 相同
	LOG10(X)	常用对数	实型	与 X 相同
	SIN(X)	正弦	实型或复型	与 X 相同
	SINH(X)	双曲正弦	实型	与 X 相同
	SQRT(X)	求平方根	实型或复型	与 X 相同
	TAN(X)	正切	实型	与 X 相同
	TANH(X)	双曲正切	实型	与 X 相同
字符函数	ACHAR(I)	将 ASCII 码转换成字符	整型	字符型
	ADJUSTL(String)	左对齐	字符型	字符型
	ADJUSTR(String)	右对齐	字符型	字符型
	CHAR(I[,KIND])	处理机对照序列中给出字符	整型	字符型
	IACHAR(C)	求某一字符的 ASCII 值	默认字符型且长度为 1	默认整型
	ICHAR(C)	处理机对照序列中某一字符的位置	默认字符型且长度为 1	默认整型
	INDEX(String,SubString[,Back])	指出子串的起始位置	字符型	默认整型
	LEN_TRIM(String)	除掉尾部空格后的字符串长度	字符型	默认整型标量
	LGE(String_A,String_B)	字符串 A 是否大于或等于 B	默认字符型	默认逻辑型
	LGT(String_A,String_B)	字符串 A 是否大于 B	默认字符型	默认逻辑型
	LLE(String_A,String_B)	字符串 A 是否小于或等于 B	默认字符型	默认逻辑型
	LLT(String_A,String_B)	字符串 A 是否小于 B	默认字符型	默认逻辑型
	REPEAT(String,NCOPIES)	重复连接	String 字符型标量 NCOPIES 非负整型标量	字符型标量
	SCAN(String,Set[,Back])	扫描字符串中的字符集合	字符型	默认整型

(续一)

字符查询 函数 种别函数	TRIM(STRING)	除去尾部空格	字符型标量	字符型标量
	VERIFY(STRING,SET[,BACK])	在一个字符串中检查字符集	字符型	默认的整型
	LEN(STRING)	计算字符串的长度	字符型标量或字符型有 值数组	默认的整型标量
逻辑函数 数值查询 函数	KIND(X)	求种别参数	任意内在类型	默认的整型标量
	SELECTED_INT_KIND(R)	给定范围内选取整型种别参数	整型标量	默认的整型标量
	SELECTED_REAL_KIND(P,R)	给定精度和范围内选取实型 种别参数	整型标量	默认的整型标量
	LOGICAL(L[,KIND])	在逻辑种别之间转换	逻辑型	逻辑型
	DIGITS(X)	求有效数字个数	整型或实型标量或 有值数组	默认的整型标量
	EPSILON(X)	与 1 相比几乎可以忽略不计 的数值	实型标量或有值数组	与 X 相同
	HUGE(X)	模型中最大数	整型或实型标量或 有值数组	与 X 相同
	MAXEXPONENT(X)	模型中最大的指数	实型标量或有值数组	默认的整型标量
	MINEXPONENT(X)	模型中最小的指数	实型标量或有值数组	默认的整型标量
	PRECISION(X)	十进制精度	实型或复型标量或 有值数组	默认的整型标量
位处理 函数	RADIX(X)	该模型的基数	整型或实型标量或 有值数组	默认的整型标量
	RANGE(X)	十进制指数范围	整型或实型标量或 有值数组	默认的整型标量
	TINY(X)	模型中最小的正数	实型标量或有值数组	与 X 相同
	BTEST(I,PCS)	位测试	整型	默认的逻辑型
	LAND(I,J)	逻辑与	整型	整型
	IBCLR(I,POS)	位清除	整型	整型
	IBITS(I,POS,LEN)	位析取	整型	整型
	IBSET(I,POS)	置位	整型	整型
	IEOR(I,J)	按位加	整型	整型
	IOR(I,J)	位同或	整型	整型
位查询函数 浮点处理 函数	ISHIFT(I,SHIFT)	逻辑移位	整型	整型
	ISHFTC(I,SHIFT[,SIZE])	循环移位	整型	整型
	NOT(I)	逻辑补码	整型	整型
	BIT_SIZE(I)	模型中位的数目	整型	标量整型
	EXPONENT(X)	模型数的指数部分	实型	默认的整型
	FRACTION(X)	模型数的小数部分	实型	实型
	NEAREST(X,S)	在给定方向上最接近(但不等) 的数	实型且 $S \neq 0$	实型
	RRSPACING(X)	给定数与模型数相对间隔的 倒数	实型	实型
	SCALE(X,I)	实数乘以其基数的整数次幂	X 为实型,I 为整型	实型
	SET_EXPONENT(X,I)	设置数的指数部分	X 为实型,I 为整型	实型
传递函数	SPACING(X)	给定数与模型数的绝对间隔	实型	实型
	TRANSFER(SOURCE,MOLD [,SIZE])	把第一个变元处理成第二个 变元的类型	任意类型的标量或有值 数组,SIZE 为整型标量	与 MOLD 相同
向量和矩阵 乘法函数	DOT_PRODUCT(VECTOR_A, VECTOR_B)	两个秩为 1 的数组的点积秩为 1	的整型、实型、标量 复型或逻辑型有值数组	秩为 1 或 2 的整型、实 型、复型或逻辑型有值 数组
	MATMUL(MATRIX_A,MATRIX_B)	矩阵相乘		与变元相同
数组约简 函数	ALL(MASK[,DIM])	所有元素值均为真时函数值 为真	MASK 是逻辑型	逻辑型
	ANY(MASK[,DIM])	只要有一个元素值为真时函 数值为真	MASK 是逻辑型	逻辑型
	COUNT(MASK[,DIM])	数组中元素值为真的个数	MASK 是逻辑型	默认的整型
	MAXVAL(ARRAY[,DIM][,MASK])	数组中最大值	ARRAY 整型或实型	与 ARRAY 相同
	MINVAL(ARRAY[,DIM][,MASK])	数组中最小值	ARRAY 整型或实型	与 ARRAY 相同
	PRODUCT(ARRAY[,DIM][,MASK])	数组元素乘积	ARRAY 整型或实型	与 ARRAY 相同

(续二)

数组查询 ·函数	SUM(ARRAY[,DIM][,MASK])	数组元素总和	ARRAY 整型实型或复型	与 ARRAY 相同
	ALLOCATED(ARRAY)	数组分配状态	可分配数组	默认逻辑标量
数组构造 函数	LBOUND(ARRAY[,DIM])	数组中维下界	ARRAY 任意类型	默认的整型
	SHAPE(SOURCE)	数组或标量的形	任意类型的有值数组或标量	秩为 1 的默认整型数组
数组变形 函数	SIZE(ARRAY[,DIM])	数组中元素的总数	ARRAY 任意类型	默认的整型标量
	UBOUND(ARRAY[,DIM])	数组的维上界	ARRAY 任意类型	默认的整型
数组处理 函数	MERGE(TSOURCE,FSOURCE, MASK)	屏蔽下做合并	任意类型	与 TSOURCE 相同
	PACK(ARRAY,MASK[,VECTOR])	屏蔽下将一个数组压缩成秩	ARRAY 任意类型	与 ARRAY 相同
数组定位 函数	SPREAD(SOURCE,DIM,NCOPIES)	用增加一维的方式复制数组	SOURCE 为任意类型 DIM,NCOPIES 整型标量	与 SOURCE 相同
	UNPACK(VECTOR,MASK,FIELD)	屏蔽下将秩为 1 的数组拆成另一个数组	任意类型	与 VECTOR 类型相同的数组
数组处理 函数	RESHAPE(SOURCE,SHAPE [,PAD][,ORDER])	改变数组形状	SOURCE,PAD 为任意类型的有值数组 SHAPE,ORDER 为整型	形为 SHAPE 的数组
	CSHIFT(ARRAY,SHIFT[,DIM])	循环移动	ARRAY 任意类型	与 ARRAY 相同
数组定位 函数	ECSHIFT(ARRAY,SHIFT [,BOUNDARY][,DIM])	去尾移动	SHIFT 为整形标量或数组 ARRAY 任意类型	与 ARRAY 相同
	TRANSPOSE(MATRIX)	二维数组转置	SHIFT 为整形标量或数组	与变元相同
变元存在 查询函数	MAXLOC(ARRAY[,MASK])	数组中最大值位置	任意类型秩为 2 的数组 整型或实型	默认的整型、秩为 1 且大于等于 ARRAY 的秩的数组
	MINLOC(ARRAY[,MASK])	数组中最小值位置	整型或实型	同上
指针相连状 态查询函数	PRESENT(A)	变元存在与否	可访问的虚元名	默认的逻辑标量
	ASSOCIATED(POINTER [,TARGET])	相连状态或相连状态比较	POINTER 任何类型指针 TARGET 指针或目标	默认的逻辑标量

2. 内在子程序

子程序名称	子程序功能	变元类型
DATE_AND_TIME([DATE][,TIME] [,ZONE][,VALUES])	得到日期和时间等各项参数	DATE,TIME,ZONE 默认的字符型标量 VALUES 为默认的整型、秩为 1
MVBITS(FROM,FROMPOS,LEN,TO, TOPOS)	按位序列从一个整数拷贝到另一个整数	整型
RANDOM_NUMBER(HARVEST)	返回伪随机数	实型
RANDOM_SEED([SIZE][,PUT [,GET])	重新启动或初始化伪随机数发生器	SIZE 为默认的整型标量 PUT,GET 为默认的整型数组
SYSTEM_CLOCK([COUNT [,COUNT_RATE][,COUNT_MAX])	从系统时钟得到数据	默认的整型标量

3. 内在函数的特定名

特定名	类属名	变元类型
ABS(A)	ABS(A)	默认的实型
ACOS(X)	ACOS(X)	默认的实型
AIMAG(Z)	AIMAG(Z)	默认的复型
AINT(X)	AINT(X)	默认的实型
ALOG(X)	LOG(X)	默认的实型
ALOG10(X)	LOG10(X)	默认的实型
·AMAX0(A1,A2[,A3]···)	REAL(MAX(A1,A2[,A3]···))	默认的整型
·AMAX1(A1,A2[,A3]···)	MAX(A1,A2[,A3]···)	默认的实型
·AMIN0(A1,A2[,A3]···)	REAL(MIN(A1,A2[,A3]···))	默认的整型

(续一)

•AMIN1(A1,A2[,A3]...)	MIN(A1,A2[,A3]...)	默认的实型
AMOD(A,P)	MOD(A,P)	默认的实型
AMINT(A)	AMINT(A)	默认的实型
ASIN(X)	ASIN(X)	默认的实型
ATAN(X)	ATAN(X)	默认的实型
ATAN2(Y,X)	ATAN2(Y,X)	默认的实型
CABS(A)	ABS(A)	默认的复型
CCOS(X)	COS(X)	默认的复型
CEXP(X)	EXP(X)	默认的复型
CHAR(I)	CHAR(I)	默认的整型
CLOG(X)	LOG(X)	默认的复型
CONJG(Z)	CONJG(Z)	默认的复型
COS(X)	COS(X)	默认的实型
COSH(X)	COSH(X)	默认的实型
CSIN(X)	SIN(X)	默认的复型
CSQRT(X)	SQRT(X)	默认的复型
DABS(A)	ABS(A)	双精度实型
DACOS(A)	ACOS(A)	双精度实型
DASIN(X)	ASIN(X)	双精度实型
DATAN(X)	ATAN(X)	双精度实型
DATAN2(Y,X)	ATAN2(Y,X)	双精度实型
DCOS(X)	COS(X)	双精度实型
DCOSH(X)	COSH(X)	双精度实型
DDIM(X,Y)	DIM(X,Y)	双精度实型
DEXP(X)	EXP(X)	双精度实型
DIM(X,Y)	DIM(X,Y)	默认的实型
DINT(A)	AINT(A)	双精度实型
DLOG(X)	LOG(X)	双精度实型
DLOG10(X)	LOG10(X)	双精度实型
DMAX1(A1,A2[,A3]...)	MAX(A1,A2[,A3]...)	双精度实型
DMIN1(A1,A2[,A3]...)	MIN(A1,A2[,A3]...)	双精度实型
DMOD(A,P)	MOD(A,P)	双精度实型
DNINT(A)	ANINT(A)	双精度实型
DPROD(X,Y)	DPROD(X,Y)	默认的实型
DSIGN(A,B)	SIGN(A,B)	双精度实型
DSIN(X)	SIN(X)	双精度实型
DSINH(X)	SINH(X)	双精度实型
DSQRT(X)	SQRT(X)	双精度实型
DTAN(X)	TAN(X)	双精度实型
DTANH(X)	TANH(X)	双精度实型
EXP(X)	EXP(X)	默认的实型
•FLOAT(A)	REAL(A)	默认的实型
IABS(A)	ABS(A)	默认的实型
•ICHAR(C)	ICHAR(C)	默认的实型
IDIM(X,Y)	DIM(X,Y)	默认的实型
•IDINT(A)	INT(A)	双精度实型
IDNINT(A)	NINT(A)	双精度实型
•IFIX(A)	INT(A)	默认的实型
INDEX(STRING,SUBSTRING)	INDEX(STRING,SUBSTRING)	默认的实型
•INT(A)	INT(A)	默认的实型
ISIGN(A,B)	SIGN(A,B)	默认的实型
LEN(STRING)	LEN(STRING)	默认的实型
•LGE(STRING_A,STRING_B)	LGE(STRING_A,STRING_B)	默认的实型
•LGT(STRING_A,STRING_B)	LGT(STRING_A,STRING_B)	默认的实型
•LLE(STRING_A,STRING_B)	LLE(STRING_A,STRING_B)	默认的实型
•LLT(STRING_A,STRING_B)	LLT(STRING_A,STRING_B)	默认的实型
•MAX0(A1,A2[,A3]...)	MAX(A1,A2[,A3]...)	默认的实型
•MAX1(A1,A2[,A3]...)	INT(MAX(A1,A2[,A3]...))	默认的实型

(续二)

·MIN0(A1,A2[,A3]...)	MIN(A1,A2[,A3]...)	默认的实型
·MIN1(A1,A2[,A3]...)	INT(MIN(A1,A2[,A3]...))	默认的实型
MOD(A,P)	MOD(A,P)	默认的实型
NINT(A)	NINT(A)	默认的实型
·REAL(A)	REAL(A)	默认的实型
SIGN(A,B)	SIGN(A,B)	默认的实型
SIN(X)	SIN(X)	默认的实型
SINH(X)	SINH(X)	默认的实型
·SGNL(A)	REAL(A)	双精度实型
SQRT(X)	SQRT(X)	默认的实型
TAN(X)	TAN(X)	默认的实型
TANH(X)	TANH(X)	默认的实型

4. 几点说明

(1) 许多变元关键词的名字指明了它们的用途。例如:

KIND	描述结果的 KIND、标量整型初始表达式
STRING, STRING_A	任意字符串
BACK	指明串扫描是从右向左、逻辑型
MASK	用于变元的屏蔽、逻辑型
DIM	数组变元选择的维、整型标量

(2) 函数的类属名的使用具有通用性,应尽量少用函数的特定名。

(3) 函数名前带·的内在函数的特定名不能用作实元。

(4) 内在函数和内在子程序合称为内在过程。由于翻译的原因,有的资料也把它们分别称作内部函数、内部子例子程序和内部过程(内部子程序)。由于这种称呼与程序单元中内部子程序部分的内部子程序相混淆,因此本书一律采用“内在”来称呼,以示区别。在查阅其它未明确区分这两类称呼的资料时,可以根据上下文确定真正的含义。

主要参考文献

- [1] 潘在元,张素素.FORTRAN90 教程.浙江大学出版社,1993
- [2] Fortran 工作组.标准 Fortran 90 语言程序设计.学苑出版社,1994
- [3] JeanneC. Adams 等. Fortran 90 Handbook. Intertext Publications/MultisciencePressInc. 1992
- [4] 王文才,郭淑分.FORTRAN 9 0 学习指南.国防出版社,1992
- [5] 马瑞民,陈仁华等.微机系统与操作基础教程.石油工业出版社,1995
- [6] 谭浩强,田淑清.FORTRAN 语言程序设计.高等教育出版社,1985
- [7] 秦克诚.FORTRAN 程序设计.电子工业出版社,1987
- [8] 唱江华,马瑞民等.PASCAL 语言程序设计.哈尔滨工程大学出版社,1996
- [9] 谭浩强.C 程序设计.清华大学出版社,1991
- [10] 谭浩强,张基温.TrueBASIC 程序设计.清华大学出版社,1989
- [11] 何新贵等.Fortran90.中国铁道出版社,1994

