



# Fortran 90/95 高级程序设计

周振红 郭恒亮  
张君静 李 强 编 著



 黄河水利出版社

# 前 言

在科学与工程计算领域，有相当多的科技人员对 Fortran 90/95 标准缺乏足够的了解，担心采用新标准会废弃已有的宝贵程序资源，所以仍在使用自己所熟悉的 Fortran 77 进行计算编程。事实上，Fortran 90/95 标准融入了许多现代语言特征，例如模块、接口块、指针、派生类型等，并进一步加强了数组功能，使得程序开发效率和可维护性都得到了极大的提高。

随着 Fortran 标准的不断更新，会逐步淘汰一些过时的语言特征。比如，Fortran 90 标准标明过时的一些语言特征，就从 Fortran 95 标准中去掉了。而且，语言实现机制也在发生变化：Fortran 90/95 标准已带有面向对象的特性，Fortran 2003 则直接支持面向对象的程序设计。所以，眼下从 Fortran 77 转到 Fortran 90/95 已势在必行。

目前，介绍 Fortran 90/95 的书籍还比较少，有的主要讲解的还是 Fortran 77 语法，有的侧重编译器、集成开发环境的介绍，对 Fortran 90/95 新语言特征讲解得不够系统、深入。

针对上述情况，作者在数年《Fortran 90/95 程序设计》教学、科研实践的基础上，对 Fortran 90/95 进行了系统的整理、编撰，并从当前软件开发的实际情况出发，增加了独具特色的语言扩展：模拟 C++ 面向对象程序设计，Visual Studio 6.0 环境下 Fortran 与 C/C++ 的混合编译，动态链接库 DLL 和基于组件对象模型 COM 的组件开发，及其在 Visual C++ 6.0、Visual Basic 6.0 和 Delphi 7.0 中的集成。

下面是作者的联系方式，随时欢迎您的指教。通信地址：郑州大学环境与水利学院，450002。电子信箱：zzh6374@163.com

作者

2005 年 6 月

# 导 读

第一章 Fortran 背景知识。介绍 Fortran 语言发展简史, Fortran 90/95 新的语言特征, 以及 Visual Fortran 编译器的演变。

第二章 Fortran 程序设计基础。介绍程序书写、字符集和标识符、数据类型、声明的有关事项、算术表达式及表控输入/输出语句。

第三章 模块化程序设计——例程和模块。从模块化程序设计出发, 重点讲述例程和模块, 其中例程包括内部例程、外部例程、例程重载、递归例程、例程参数和接口块, 另外也介绍了主程序单元的构造。

第四章 结构化程序设计——控制结构。从结构化程序设计出发, 重点讲述选择结构(包括 CASE 结构)和循环结构。

第五章 数组。重点讲述数组加强功能, 包括数组整体及数组段操作, WHERE 和 FORALL 构造, 动态数组、不同类型数组参数及数组型函数。

第六章 派生类型。讲述派生类型的构造、操作符重载及数据库管理应用。

第七章 指针。介绍指针的基本概念, 讲述指针数组、指针型函数和指针参数的使用及单链表应用。

第八章 模拟 C++面向对象程序设计。综合运用模块、接口块、派生类型和指针, 模拟 C++主要的面向对象特性: 封装、继承和运行时多态。

第九章 格式化输入/输出及文件操作。介绍格式编辑符, 讲述格式化输入/输出语句的使用, 以及内部文件和外部文件的操作。

第十章 Fortran 与 C/C++的混合编译。系统讲述 Fortran 与 C/C++之间调用约定的协调、程序的混合编译, 以及 Fortran 模块数据和例程的传递。

第十一章 Fortran 与 C/C++混成 DLL 并集成到 Win32 应用程序。在混合编译的基础上, 将 Fortran 与 C/C++建造成动态链接库 DLL, 并集成到 Visual C++ 6.0、Visual Basic 6.0 及 Delphi 7.0 中。

第十二章 Fortran COM 组件的创建及其在客户程序中的调用。介绍 COM 对象及组件的基本概念, 讲述在 Fortran COM 服务器向导支持下 COM 组件的创建, 以及 Fortran COM 组件在 Visual C++ 6.0、Visual Basic 6.0 及 Delphi 7.0 客户程序中的调用。

# 目 录

## 前 言

## 导 读

<b>第一章 Fortran 背景知识</b> .....	(1)
第一节 Fortran 语言简史 .....	(1)
第二节 Fortran 90/95 新的语言特征 .....	(2)
第三节 Visual Fortran 编译器的演变 .....	(4)
<b>第二章 Fortran 程序设计基础</b> .....	(5)
第一节 程序书写 .....	(5)
第二节 字符集和标识符 .....	(7)
第三节 数据类型 .....	(8)
第四节 声明的有关事项 .....	(13)
第五节 算术表达式 .....	(15)
第六节 表控输入/输出语句 .....	(17)
<b>第三章 模块化程序设计——例程和模块</b> .....	(21)
第一节 内部例程 .....	(21)
第二节 主程序 .....	(26)
第三节 外部例程 .....	(27)
第四节 接口块 .....	(28)
第五节 模 块 .....	(30)
第六节 例程参数 .....	(32)
第七节 例程重载 .....	(36)
第八节 递归例程 .....	(38)
<b>第四章 结构化程序设计——控制结构</b> .....	(42)
第一节 选择结构 .....	(42)
第二节 循环结构 .....	(49)
<b>第五章 数 组</b> .....	(57)
第一节 数组声明 .....	(57)
第二节 数组存储 .....	(59)
第三节 数组操作 .....	(60)

第四节	数组参数	(70)
第五节	动态数组	(73)
第六节	数组型函数	(76)
第六章	派生类型	(80)
第一节	派生类型的定义	(80)
第二节	派生类型的构造及初始化	(81)
第三节	操作符重载	(84)
第四节	数据库管理应用	(89)
第七章	指针	(98)
第一节	指针的基本概念	(98)
第二节	指针数组	(99)
第三节	指针型函数	(101)
第四节	指针参数	(102)
第五节	单链表应用	(104)
第八章	模拟 C++ 面向对象程序设计	(118)
第一节	C++ 实现的类层次	(118)
第二节	Fortran 90 模拟方法	(122)
第九章	格式化输入/输出及文件操作	(130)
第一节	PRINT 语句	(130)
第二节	格式编辑符	(132)
第三节	READ 语句	(136)
第四节	WRITE 语句	(136)
第五节	内部文件	(137)
第六节	外部文件	(138)
第七节	不换行的读写	(142)
第十章	Fortran 与 C/C++ 的混合编译	(144)
第一节	调用约定的协调	(144)
第二节	Fortran 与 C/C++ 的混合编译	(149)
第三节	Fortran 模块数据和例程的传递	(154)
第十一章	Fortran 与 C/C++ 混成 DLL 并集成到 Win32 应用程序	(160)
第一节	动态链接库 DLL	(160)
第二节	Fortran 与 C/C++ 混成 Win32 DLL	(160)
第三节	DLL 例程在 Win32 应用程序中的集成	(168)

---

第十二章 Fortran COM 组件的创建及其在客户程序中的调用 .....	(176)
第一节 COM 对象及组件 .....	(176)
第二节 Fortran COM 组件的创建 .....	(177)
第三节 COM 组件在客户程序中的调用 .....	(184)
参考文献 .....	(192)

# 第一章 Fortran 背景知识

## 第一节 Fortran 语言简史

要了解 Fortran 90/95, 我们有必要先简单回顾一下 Fortran 语言的发展历史。

1954~1957 年, 世界上第一种高级程序设计语言——Fortran 诞生于 IBM 公司。Fortran 是 IBM Mathematical Formula Translation 的缩写, 其设计目的在于为科研人员提供一种符合数学思维习惯的高级语言, 以满足科学计算的需要。20 世纪 60 年代初, 在国防、教育和科技领域对高性能计算工具的迫切需求下, Fortran 语言蓬勃发展, 成为当时统治计算机世界的高级语言之王。

1962 年, 为了统一不同公司、不同硬件平台上的 Fortran 语言, 人们开始了 Fortran 语言标准化的尝试, 这也是程序设计语言发展史上的第一次标准化历程。1972 年, Fortran 66 标准(标准编号来自标准草案的制定时间)正式发布。但因为标准文档过于简单, 约束力不强, Fortran 66 标准发布后, Fortran 语言的统一问题并没有得到彻底解决。

1978 年, Fortran 语言标准的第一个修订版本正式发布, 这就是所谓的 Fortran 77。Fortran 77 细致地描述了 Fortran 语言的各种特征, 让 Fortran 成了一种真正规范、高效和强大的结构化程序设计语言。此后, 无数性能优异的 Fortran 77 编译器和开发工具的问世更是让 Fortran 77 成为了几乎所有理工科学生的必修课。

尽管 Fortran 77 的影响力一直延续到了今天, 但 Fortran 语言不断变革的历程却从未停止过。为了改变 Fortran 77 那种老式的、从穿孔卡片遗留下来的语言风格, 并给 Fortran 注入更多的现代语言特征, 人们于 1991 年发布了崭新的 Fortran 90 标准。除了自由的代码风格外, Fortran 90 还为 Fortran 语言引入了模块、接口块、自定义(派生)数据类型和运算符、可动态分配和参与复杂运算的数组、例程重载、指针、递归等重要的语法特征。这不但使结构化的 Fortran 语言更趋完善, 也使其具备了少量的面向对象语言特征。

1997 年发布的 Fortran 95 标准在 Fortran 90 的基础上, 吸收了 HPF(High Performance Fortran, Fortran 语言在并行环境下的一个变种)的优点, 提高了 Fortran 语言在并行任务中的表达和计算能力, 并进一步完善了派生类型、指针、数组等要素的相关语法。

2004 年 5 月, 在 ISO、IEC 的联合工作组 JTC1/SC22/WG5 以及美国 Fortran 委员会 NCITS/J3 的共同努力下, 终于推出了 Fortran 2003 标准。Fortran 2003 近乎彻底地解决了 Fortran 语言现代化的问题: 完整的面向对象机制、灵活的语法特征、统一的接口标准等。不过, 支持 Fortran 2003 标准的编译器要在一两年后才能开发出来。

以 Fortran 66 为基准, 我们可以把后续的 Fortran 77/90/95 以及 Fortran 2003 均视为对 Fortran 语言标准的修订。在历次修订中, Fortran 77 和 Fortran 95 是修订幅度相对较小的版本, 而 Fortran 90 和 Fortran 2003 则是锐意变革的“大修”版本。

由于支持 Fortran 2003 标准的编译器还不可利用, 本书从实际情况出发, 重点讲解 Fortran 90 的语法及应用, 必要时也介绍 Fortran 95 语法。

## 第二节 Fortran 90/95 新的语言特征

下面分别介绍 Fortran 90/95 新的主要语言特征。

### 一、Fortran 90

#### (一)自由书写格式

Fortran 90 提供了一种新的自由书写格式, 行中的位置没有特殊意义, 没有保留列, 尾部可以出现注释, 空格在某些情况下是有意义的。例如: PROGRAM 并不等于 PROGRAM。

#### (二)模块

Fortran 90 提供了一种新的程序单元——模块, 其功能远比 Fortran 77 数据块程序单元来得强大。模块包含了数据、例程、例程接口等要素的声明, 别的程序单元引用后, 就可访问其中的数据和例程了。模块要素的可见性可以限制在模块内, 以提供数据抽象, 编写安全、可移植的程序代码。

#### (三)自定义(派生)数据类型和操作符

Fortran 90 允许从固有数据类型和派生类型中定义新的数据类型, 派生类型可以整体访问, 也可直接访问当中的元素; 可以重载固有操作符(如+、\*), 也可定义新的操作符, 以适应派生类型数据操作要求。

#### (四)数组功能加强

在 Fortran 90 中, 固有操作符和相关的固有函数可以直接操作整个数组或数组段。可以对数组整体、部分及隐式赋值(包括可选择性赋值的 WHERE 语句), 声明数组常量, 定义派生类型数组值函数、用数组构造子规定一维数组的值, 利用 ALLOCATABLE 或 POINTER 属性为数组动态分配内存。新的固有例程能够创建和使用多维数组, 支持数组计算(如 SUM 函数累加数组元素的和)。



### (五)例程重载

同 C++ 相似, Fortran 90 也支持例程重载。它通过接口块规定统一的名子,将不同参数表的例程置于接口块内,引用时按参数匹配的原则调用具体的例程。如同固有函数 ABS, 其参数既可以是整数,也可以是实数或复数。

### (六)指针

Fortran 90 指针允许动态访问和处理数据,可以用来创建动态数组和派生类型的动态数据结构(如链表)。指针可以指向固有数据类型或派生类型,一旦和同类型的目标变量相关联,指针可以代替目标出现在表达式和赋值语句中。

### (七)递归

假如 RECURSIVE 关键字添加在例程(FUNCTION 或 SUBROUTINE)原型中, Fortran 90 例程也可递归实现。

### (八)接口块

Fortran 90 程序单元(主程序、外部例程和模块)可以包含接口块,接口块用来:  
①描述外部例程或虚参例程的接口;②为重载的例程规定统一的名称;③定义或扩展操作符等。

### (九)封装机制

类似于 C++ 中的类, Fortran 90 可以将派生类型数据连同其操作例程封装在模块内,通过其公有接口,供别的程序单元使用,以扩展 Fortran 功能,使之适合特殊的应用需求,例如模拟面向对象的程序设计、数据库管理、建立动态数据结构等。

## 二、Fortran 95

### (一)FORALL 语句和构造

在 Fortran 90 中,可以通过数组构造子、RESHAPE 和 SPREAD 固有例程逐元素地构造数组, Fortran 95 的 FORALL 语句和构造则提供了另一种数组操作方式。FORALL 允许通过元素下标对数组元素、数组段、字符子串或指针目标进行操作;类似于隐式 DO 循环, FORALL 构造可使几个数组赋值语句共享相同的下标循环控制表达式。

FORALL 是 WHERE 的一般形式,两者都通过隐式循环对数组进行操作,只不过 FORALL 是使用元素下标, WHERE 则针对整个数组。

### (二)PURE 用户定义例程

在用户定义的例程(子程序或函数)原型前添加 PURE 关键字,向系统表明该用户定义例程没有副作用。例如:在例程内改变值传递参数的值。

### (三)ELEMENTAL 用户定义例程

在用户定义例程原型前添加 ELEMENTAL 关键字,它是 PURE 例程的特殊形式。

当传递数组参数时，每次对一个数组元素进行操作。要使用 ELEMENTAL 例程，须在调用程序中建立其接口块。

#### (四)CPU\_TIME 子程序

该子程序通过参数返回特定 CPU 处理器的时间，单位为秒，参数须是单一的实数。

#### (五)NULL 函数

在 Fortran 90 中不能直接初始化指针为空指针。Fortran 90 指针必须先与目标变量相关联，或动态分配内存，然后才能使用 NULLIFY 函数置空指针。Fortran 95 则可利用 NULL 函数直接初始化指针为空指针。

值得注意的是：Fortran 95 已删除了 Fortran 90 标明为过时的某些语言特征，并新标出了一些过时的语言特征。

### 第三节 Visual Fortran 编译器的演变

当前，国内使用较多的 Fortran 编译器或可视化集成开发环境为 Visual Fortran，它起源于 Microsoft 的 Fortran PowerStation 4.0。这套工具后来卖给了 Digital 公司继续开发，第二个版本称为 Digital Visual Fortran 5.0。Digital 被 Compaq 并购后，接下来的 6.0、6.1、6.5 和 6.6 版本称为 Compaq Visual Fortran。本书使用目前最新的 (Compaq)Visual Fortran 6.6 专业版开发实例。

Visual Fortran 6.6 被组合在 Microsoft Visual Studio 6.0 集成开发环境中。Visual Studio 提供了统一的操作界面，这个界面包括文字编辑器、Project 的管理、调试工具等。而编译器则是使用类似 Plug In 的方法组合到 Visual Studio 中，用户在使用 Visual Fortran 6.6 和 Visual C++ 6.0 时，看到的都是相同的操作界面。

Visual Fortran 6.6 除了完全支持 Fortran 90/95 语法外，扩展部分还提供有完整的 Windows 应用程序开发工具，可以直接建造 Win32 DLL(动态链接库)、基于组件对象模型 COM 的组件等，专业版还内含了 IMSL 数值链接库。另外，它还可以和 Visual C++ 6.0 互相链接，将 Fortran 和 C/C++语言的程序代码混合编译成同一个执行文件(EXE 或 DLL)。

## 第二章 Fortran 程序设计基础

在学习 Fortran 90/95 高级的语言功能之前, 先来了解进行数值计算编程所必备的基础知识: 程序书写、字符集及标识符、数据类型、声明的有关事项、算术表达式以及表控输入/输出语句。

### 第一节 程序书写

#### 一、程序构造形式

我们先来看一个银行存款程序实例。

[例 2-1] 简单 Fortran 程序的构造形式。

---

```
PROGRAM MONEY
! Calculates balance after interest compounded
  REAL  BALANCE, INTEREST, RATE

  BALANCE = 1000
  RATE = 0.09
  INTEREST = RATE * BALANCE
  BALANCE = BALANCE + INTEREST
  PRINT*, 'New balance:', BALANCE
END PROGRAM MONEY
```

---

第一行的 PROGRAM 关键字标识 Fortran 主程序, 后接程序名, 这一行是可选的; 以感叹号开始的第二行是注释, 不参加编译; 第三行中的 REAL 将其后面的变量声明为实型数。这三行为非执行部分, 之后的部分(END 语句之前)为执行部分。

由此给出简单 Fortran 90 程序的构造形式:

[PROGRAM 程序名]

[声明语句]

[执行语句]

END [PROGRAM [程序名]]

方括号内的部分是可选的, END 语句是惟一必须的, 它通知编译器: 程序编译到此结束。END 语句中的程序名可以省略, 但若出现程序名, 必须同时出现 PROGRAM 关键字。

## 二、语句

语句是 Fortran 程序的基本单位, 一条语句可包含 0~132 个字符。Fortran 77 规定, 一条语句的不同部分应从特定的列开始, 这样的书写格式称为固定格式, 相应的程序文件扩展名为 .f 或 .for; 自由格式的 Fortran 90(文件扩展名为 .f90)则无此限制。

除赋值语句外, 所有的语句都从一个关键字开始。比如, 例 2-1 中出现的关键字: PROGRAM、REAL、PRINT 和 END。

一般情况下, 每行一条语句。如一行有多条语句, 它们之间以分号间隔。为清楚起见, 通常将几条简单的赋值语句写在一行上。例如:

```
A = 1; B = 1; C = 1
```

假如一条语句一行写不完, 允许出现续行, 但要求被续行最后的非空白字符为 &。例如:

```
A = 174.6 * &
```

```
(T-1981.2)**3
```

续行从下一行(非注释行)的第一个非空白字符开始; 若下一行的非空白字符为 &, 则续行从该字符后的第一个字符开始。Fortran 90 允许出现多达 39 个续行。

## 三、空白的作用

通常, 空白没有意义, 它不参加编译。适当地运用空白空间, 可以增加程序的可读性, 如程序块中的代码缩进。但在代表有意义字符序列的记号(token)内, 比如: 标号、关键字、变量名、操作符等, 不允许出现空白。例如, INTEGER、BALANCE 和 <= 是非法的(<= 为操作符)。

一般情况下, 记号之间需留有空白, 例如: 30CONTINUE 是非法的, 因为标号 30 和关键字 CONTINUE 是两个独立的记号。而有的记号间的空白是可选的, 例如: END PROGRAM 和 ENDPGRAM, A\*B 和 A\*B 均是合法的。

## 四、注释

Fortran 90 只提供了一种注释方式: 以感叹号开始的语句作为注释, 字符串内的感叹号除外。注释可以是一整行, 也可以是空白行。注释在编译时被忽略。

## 五、固定格式

早期的计算机，还没有使用键盘/显示器作为输入/输出设备，那时的程序是利用穿孔卡片一张张地记录下来，再让计算机来执行。固定格式正是为了配合早期使用穿孔卡片输入程序所发明的格式。见例 2-2 所示。

**[例 2-2] 固定格式编写的程序。**

---

```
C      FIXED FORMAT DEMO
      PROGRAM Fixed
      PRINT*, 'Hello
      $World!'
      PRINT 10
10     FORMAT(1X, 'This program is written in fixed format.')
      END
```

---

在固定格式中，每行有 80 列，这 80 列被分为 4 个区，分别书写不同的内容：

(1)第 1~5 列为标号区。可以写 1~5 位整数作为语句标号，也可以没有标号，标号区中的空格不起作用。标号区中某一行的第 1 列若出现“C”或“\*”字符，则该行被认为是注释行。该程序中的第一行即为注释行，10 为格式标号。

(2)第 6 列为续行标志区。如果在一行的第 6 列上写上一个非空格或非 0 的字符，则该行作为上一行的续行。程序中的“\$”即为续行标志。

(3)第 7~72 列为语句区。语句可以从第 7 列以后任何位置开始书写，但一行只能写一个语句。语句区内的空格(不包括引号内字符串中的空格)在编译时被忽略。

(4)第 73~80 列为注释区。注释区用于程序员书写提示信息，在编译时不予处理。

这里介绍固定格式，只是让大家对 Fortran 77 程序有所了解，建议大家在编写新的程序时一律采取前述的自由格式。

## 第二节 字符集和标识符

### 一、字符集

Fortran 90 字符集由下列字符组成：

- 26 个英文字母(A~Z 和 a~z)；

- 数字 0~9;
- 下划线(\_);
- 特殊字符, 如表 2-1 所示。

表 2-1 字符集中的特殊字符

字符	名称	字符	名称
	空格	:	冒号
=	等号	!	感叹号
+	加号	"	引号
-	减号	%	百分号
*	星号	&	和号
/	撇号	;	分号
(	左括号	<	小于号
)	右括号	>	大于号
,	逗号	?	问号
.	小数点	\$	美元符号
,	省略号		

注: (1)表中的标点符号占一个字节, 即为半角;

(2)字符串中的字符可以是 Fortran 90 字符集之外的字符。

## 二、标识符

在给变量、常量、例程等标识符命名时, 须以字母(A~Z, a~z)开头, 后可接多达 30 个字母(A~Z 或 a~z)、数字(0~9)或下划线(\_)。例如: MASS, rate, Npts, 19J7, Time\_Rate, Speed\_of\_Light。针对标识符命名, 有下列几点值得注意:

- (1)只能以字母开头(3M, \_Right 为无效标识符);
- (2)不能含有空格字符(Time Rate 为无效标识符);
- (3)不区分字母大、小写(Vel, VEL, vel 为同一标识符);
- (4)长度限定为 31 个字符(Fortran 77 为 6 个字符);
- (5)避免与关键字、标准例程重名。

## 第三节 数据类型

程序要和数据打交道, 数据都有一个特定的类型。数据类型含有两层意思: 一是数据可以取哪些值; 二是对数据可以进行哪些运算。例如: 整数取 0、 $\pm 1$ 、 $\pm 2$ 、 $\pm 3$  等, 可以对它们进行算术运算。

Fortran 90 提供了 5 个固有(内建)数据类型, 这些数据类型被分成两大类: 一类是数值型, 包括整型、实型和复数型; 另一类是非数值型, 包括字符型和逻辑型(或

布尔型)。

在固有数据类型之外，还允许用户定义自己的数据类型——自定义数据类型或派生类型。

本节主要讲解 5 个固有数据类型的取值范围(和精度)，派生数据类型将在第六章中讲解。

## 一、整数类型

### (一)整型变量

声明整型变量的一般形式为：

INTEGER I

INTEGER([KIND=]n) I

n 是种类参数，取值为 1、2、4、8。

种类参数(KIND)是 Fortran 90 新添加的特性，它通过规定存储数据所用的内存字节数来控制数据的取值范围，1、2、4、8 为整数在内存中的存储字节数。如果种类参数没有特别规定，则取缺省值；而缺省值受编译器选项影响，若没有编译器选项规定，32 位系统下缺省值为 4。不同种类参数的整数取值范围见表 2-2。

表 2-2 不同种类参数的整数取值范围

参数	整数取值范围
INTEGER(1)	-128 ~ 127
INTEGER(2)	-32768 ~ 32767
INTEGER(4)	-2147483648 ~ 2147483647
INTEGER(8)	-9223372036854775808 ~ 9223372036854775807

Fortran 90 提供的 KIND 函数，用来获取缺省种类参数的值；HUGE 函数则用来获得取值范围的上限；上限加 1 即为取值范围的下限。如下列代码段所示：

```

INTEGER(8) I,Big,Small
Big = HUGE(I)
Small = Big + 1
PRINT*, 'Largest:      ',Big
PRINT*, 'Smallest:     ', Small

```

值得注意的是：在不同的平台(处理器和编译器)下，相同的种类参数可能有不同的取值范围，这极大地影响了程序代码的可移植性。对此，Fortran 90 提供了

SELECTED\_INT\_KIND 函数:

`result = SELECTED_INT_KIND (r)`

result 代表整数  $n$  在范围  $-10^r < n < 10^r$  内的种类参数。例如:

---

```
result = SELECTED_INT_KIND(3)      ! 2
result = SELECTED_INT_KIND(8)      ! 4
result = SELECTED_INT_KIND(10)     ! 8
result = SELECTED_INT_KIND(19)     ! -1
```

---

返回-1 表示没有可用的种类参数。该函数为我们提供了跨平台能力, 比如要在不同的平台下表示  $\pm 10^{10}$  内的整数  $I$ , 可以采取如下的声明方式:

---

```
INTEGER, PARAMETER :: K10 = SELECTED_INT_KIND(10)
INTEGER(K10) I
```

---

先声明依赖于特定平台的种类常数, 再以该常数为种类参数声明变量。

## (二)整型常量

一般的文字整型常量, 其种类参数取缺省值; 但也可显式声明种类参数, 例如:

47\_2, 其一般形式为:

`[s]n[_k]`

其中,  $s$  代表正负号;  $n$  为  $0 \sim 9$  的十进制数(前导 0 被忽略);  $k$  指种类参数; 在  $n$  和  $k$  之间为一下划线。

## 二、实数类型

### (一)实型变量

声明实型变量的一般形式为:

```
REAL A
REAL([KIND=]n) A
DOUBLE PRECISION A
```

种类参数  $n$  为 4、8, 若没有编译器选项规定, 缺省值为 4。双精度实型数相当于 REAL(8), 不能再为它规定种类参数。实数取值范围分别为:

(1)REAL(4), 取值范围为  $\pm 1.1754944\text{E}-38 \sim \pm 3.4028235\text{E}+38$ ;

(2)REAL(8), 取值范围为  $\pm 2.225073858507201\text{E}-308 \sim \pm 1.797693134862316\text{E}+308$ 。

Fortran 90 除了提供与整数对应的 KIND、HUGE 函数外, 还提供了获得实数的



取值范围下限函数 TINY、精度函数 PRECISION、指数范围函数 RANGE，以及获取特定平台下种类参数的函数 SELECTED\_REAL\_KIND：

`result = SELECTED_REAL_KIND (p,r)`

函数返回有效位数为  $p$ (精度)、指数范围为  $10^{-r} \sim 10^r$  的实数种类参数。比如，在特定平台下规定有效位数为 15、指数为 307，相应的种类参数应为 SELECTED\_REAL\_KIND (15,307)。若没有可用的种类参数，函数分别返回-1、-2 和-3：

- 返回-1 表示无法满足规定的有效位数；
- 返回-2 表示无法满足规定的指数范围；
- 返回-3 表示有效位数和指数范围都无法满足。

## (二)实型常量

文字实型常量是数学上实数的近似表达，不带指数的实型常量一般形式为：

`[s]n[k]`

带指数实型常量一般形式为：

`[s]n E[s]m[k]` 或 `[s]n D[s]m`

其中， $s$  代表正、负号； $n$ 、 $m$  代表  $0 \sim 9$  的十进制数， $n$  通常带有小数点， $m$  为指数； $k$  为种类参数(其值一般取 4、8)，若没有规定种类参数，种类参数取缺省值。

例如：1.0E6 和 1.0D6 表示  $1.0 \times 10^6$ ，前者为单精度 REAL(4)，后者为双精度 REAL(8)；1.0E6\_8 则表示双精度数  $1.0 \times 10^6$ ，但以 D 表示的实数不能再带种类参数。

## 三、复数类型

### (一)复数型变量

Fortran 90 支持复数类型及其算术运算。同声明整数、实数类型变量一样，声明复数类型变量的一般形式为：

`COMPLEX X`

`COMPLEX([KIND= $n$ ]) X`

复数类型变量的种类参数为 4、8，若没有编译器选项规定，种类参数缺省值为 8。

### (二)复数型常量

复数类型常量呈现下列形式：

`(r,m)`

其中， $r$  代表复数常量的实部； $m$  代表复数常量的虚部。

复数类型常量的种类参数，取实部和虚部的实数(不计整数)种类参数的极大值。  
例如：

`KIND( (1,1.0) ) = 4`

`KIND( (1_8,2_8) ) = 8`

KIND( (1.0, -1.0\_8) ) = 8

#### 四、逻辑类型

##### (一)逻辑型变量

声明逻辑型(或布尔型)变量 L 的一般形式为:

LOGICAL L

LOGICAL([KIND=]n) L

种类参数 n 为 1、2、4、8, 若没有编译器选项规定, 种类参数缺省值为 4。

##### (二)逻辑型常量

逻辑型常量为.TRUE.(逻辑真)和.FALSE.(逻辑假), 其种类参数取缺省值。当然, 也可显式规定逻辑型常量的种类参数, 比如: .TRUE.\_2, 种类参数规定为 2。

#### 五、字符类型

##### (一)字符型变量

声明字符型变量 C 的形式为:

CHARACTER C

CHARACTER [(LEN=]len) C, CHARACTER\*len C

CHARACTER [(LEN=]len [, (KIND=]n)]) C

CHARACTER [(KIND=n [, LEN=len)] C

字符型有两个可选参数: 长度参数和种类参数。不管是否显式规定, 字符型种类参数总是 1, 即一个字符占一个存储字节。假如两个可选参数都没有给出, 长度和种类参数均取缺省值 1; 若只给出一个参数, 这个参数代表长度; 若给出两个参数, 这两个参数依次为长度和种类参数(种类参数只能取 1); 若采取关键字(KIND = , LEN =)声明形式, 参数的顺序可以任意。例如:

CHARACTER(KIND = 1, LEN = 10) Str

##### (二)字符型常量

与 C 语言不同, 字符常量和字符串常量分别以单引号和双引号作为界定符。

Fortran 中的字符串统一以单引号或双引号作为界定符:

[k\_] '[ch]'

[k\_] "[ch]"

其中, k 为可选的种类参数(值为 1), 后接一下划线; ch 为字符(串), 它可以是 Fortran 90 字符集之内或之外的字符, 字符的个数为字符串长度。

下列为一些有效的字符型常量:

"WHAT KIND TYPE? "

```
'TODAY'S DATE IS: '
```

```
"The average is: "
```

值得注意的是：若要表示和界定符相同的字符，须用两个连续的界定符字符来表示。例如：字符型常量'Fortran's 90'，表示字符串 Fortran's 90，长度为 12。另外，界定符必须统一：要么都用单引号，要么都用双引号。

## 第四节 声明的有关事项

### 一、强制类型声明

Fortran 90 以前版本的变量类型有一个隐含约定，即 I~N 规则：除非特别声明，否则，在程序中的变量名凡以 I、J、K、L、M、N 这 6 个字母开头的都被默认为整型变量，以其他字母开头的表示实型变量。为了确保兼容以前版本的程序代码，Fortran 90 仍然可以使用 I~N 规则，但并不推荐使用，因为这种隐含约定往往会带来严重的程序错误。让我们实际操作一个实例(例 2-3)，以加深认识。

[例 2-3] 隐含约定带来的弊端。

```
PROGRAM Main
  !IMPLICIT NONE
  !REAL Interest,Count
  Interest=0.08
  Count=0.08
  PRINT*, 'Interest = ',Interest      !0
  PRINT*, 'Count = ',Cont            !0.0
END
```

例 2-3 将一个实型常量 0.08 赋值给一个隐含约定为整型的变量 Interest，在变量中仅保存了实型常量的整数部分 0；在引用实型变量 Count 时，错将变量名写成 Cont，Fortran 编译器认为 Cont 是一个新的实型变量，并给出缺省值 0.0。在实际编程中，类似这样的错误并不少见。

如何才能避免产生这类错误呢？Fortran 90 提供了 IMPLICIT NONE 语句(放在程序的开始部分)，以废除隐含约定，强制变量在使用前必须进行类型声明。

在例 2-3 中添加 IMPLICIT NONE 语句，再对程序进行编译，则编译时出现错

误, 并提示要求显式声明 Interest、Count 和 Cont 三个变量; 在程序中添加变量声明语句, 并将 Cont 改为 Count, 此时就会顺利通过编译, 并产生正确的输出结果。

## 二、变量声明及初始化

Fortran 语言、C 语言都属编译型语言, 在编译完成后, 编译器已为所引用的变量开辟了内存空间, 在程序执行的过程中一般不会再申请内存(不包括动态分配), 所以声明部分必须出现在执行部分之前, 而不能将声明语句插在执行部分当中。如果使用 DATA 给变量赋初值(一般的赋值语句是执行语句), 请记住 DATA 也是声明的一部分, 通常跟在它所赋值的变量声明之后、执行语句之前。

Fortran 90 中, 变量声明的一般形式为:

数据类型 [[, 属性>::] 变量列表

属性有 DIMENSION、PARAMETER、TARGET、POINTER、ALLOCATABLE、INTENT 等。在数据类型与变量之间的并列冒号“::”是可选的, 但在声明的同时给变量赋初值的情况下, 符号“::”不能省略。见例 2-4 所示。

[例 2-4] 变量声明及初始化。

```
PROGRAM Main
  IMPLICIT NONE
  INTEGER a
  REAL(8)      :: b=2.0
  COMPLEX(8)   :: c=(1.0,2.0)
  CHARACTER(20) :: str="Fortran 90"
  a=6
  PRINT*, 'a=', a, ', b=', b, ', c=', c, ', str=', str
END
```

Fortran 77 则要使用 DATA 语句来设置初值。DATA 语句的格式是在 DATA 关键字后接上所要设置初值的变量, 然后再用两个斜杠包住所要设置的值。见例 2-5 所示。

[例 2-5] DATA 语句的使用。

```
PROGRAM Main
  IMPLICIT NONE
  INTEGER a
```

```
REAL(8) b
COMPLEX(8) c
CHARACTER(20) str
DATA a,b,c,str /1,2.0,(1.0,2.0),'FORTRAN 77'/
PRINT*,'a=',a,',b=',b,',c=',c,',str=',str
END
```

### 三、常量声明(PARAMETER)

程序中所用到的数据，有些是固定不变的常数，如：圆周率、重力加速度等，这些数据可以声明成常量。见例 2-6 所示。

[例 2-6] 常量声明。

```
PROGRAM Main
  IMPLICIT NONE
  REAL,PARAMETER :: PI=3.14159
  PRINT 10, sin(PI/6.0)
  10 FORMAT('sin( $\pi/6$ )=',F4.2)
END
```

符号常量只能在声明时通过 PARAMETER 属性设置其值，而且只能设置一次。值设置好后，在程序中就不能改变。

## 第五节 算术表达式

在数值计算中，算术表达式具有特别重要的作用。算术表达式由运算符、常量和变量组成，它确定了计算某个数值的规则。算术表达式只计算数值的大小，其结果为一标量。

### 一、运算符及其优先级

Fortran 有 5 种算术运算符，分别是加、减、乘、除和乘幂，按其优先级由低到高排列如下：

+ 加， - 减  
\* 乘， / 除

**\*\*** 乘幂(两个星号之间不能有空格)

**( )** 括号

即乘除运算级高于加减运算,乘幂又高于乘除,而括号总是具有最高优先级。在运算级相同的情况下(不包括乘幂),按“从左至右”的法则;但在连续的乘幂运算中,则遵循“从右至左”的法则。如  $2**3**2$ ,应先算  $3**2$ ,再算左边的乘幂,最后结果是 512,而不是 64。

## 二、整数除法

整数除法应引起大家特别注意,因为这和数学上的习惯不同。编程实现除法运算时,如果是整数之间的除法,那么结果也是整数。这在整除的情况下不会有什么问题,但若不能整除,最后结果只取商的整数部分,小数部分被去掉。比如:  $1/2 + 1/2$ ,结果不是 1 而是 0。所以,编程时若要表示  $1/2$ ,要么用 0.5,要么用 1.0/2.0,一定不能直接使用  $1/2$ 。

## 三、算术表达式中的类型转换

Fortran 并非强类型语言,它允许不同类型的数值型数据(整型、实型和复数型)之间进行算术运算,但不允许在数值型数据与非数值型数据(字符型和逻辑型)之间进行算术运算。当遇到不同类型之间的算术运算时,一般应显式地进行类型转换。假如 A、B 分别是整型和实型,REAL(A)就将整型 A 转换为实型,INT(B)则将实型 B 转换为整型。

当然,系统也有一个自动转换规则,即将低精度类型转换成高精度类型。这意味着:在整型和实型之间进行算术运算时,整型将被转换成实型。如:

$3 + 4.0$

程序执行时,先将整数 3 转换成实数 3.0,然后加上 4.0 得 7.0,其结果是实型数。这条规则对每一个算术运算符都适用。

但要注意一点,数据类型的转换是从左至右进行的,当遇到不同类型时才开始转换。如:

$9/4/3.0$

计算时,先进行  $9/4$  的整数除法运算,得整型数 2,然后整型数 2 被转换成实型数 2.0,再除以实型数 3.0,最后结果是 0.6666667,而不是数学上的结果 0.75。

## 四、赋值语句中的类型转换

表达式对变量赋值时,如果变量与表达式的结果类型相同,则直接进行赋值;如果变量与表达式的结果类型不同,则先进行表达式的类型转换,再进行赋值。见

例 2-7。

[例 2-7] 赋值语句中的类型转换。

```
PROGRAM Main
  IMPLICIT NONE
  INTEGER:: a = 3/2
  INTEGER:: b = 3.0/2
  REAL    :: c = 3/2
  REAL    :: d = 3.0/2
  PRINT*, 'a=',a,', b=',b,', c=',c,', d=',d
END
```

最后的结果：a=1, b=1, c=1.000000, d=1.500000。

由于在赋值语句中，类型转换常常会出现意想不到的情况，给编程带来麻烦，因此建议赋值时最好使用相同类型。

## 第六节 表控输入/输出语句

表控输入/输出语句，即 READ\*和 PRINT\*语句。这里的“\*”表示“表控输入或输出”，要求从系统隐含指定的输入/输出设备(键盘/屏幕)上输入或输出数据。

### 一、表控输入语句

表控输入不必指定输入数据的格式，只需将数据按其合法形式依次输入即可，所以又称为自由格式输入。READ\*语句的一般形式为：

READ\*, 变量列表

其中，变量列表中的变量用逗号分隔。

当程序执行到 READ\*语句时，向设备发出输入数据的指令，这时就可通过键盘输入数据。

输入数据时，如果只有一个数据，可直接输入；如果是多个数据，数据之间一定要设法分隔开，否则，系统会把输入的多个数据当作一个数据来处理。解决此问题的办法有两个：

(1)一种办法是使用多行输入数据，每个数据独自放在一行。

(2)另一种办法是在一行输入多个数据，各个数据之间用分隔符分隔开。允许使用的分隔符有空格、逗号和斜杠(/的作用是终止输入，即输入数据到此结束)。

[例 2-8] 表控输入。

```
PROGRAM Main
  IMPLICIT NONE
  INTEGER a
  REAL b
  CHARACTER(5) c
  LOGICAL d
  COMPLEX e

  PRINT*, "Input a,b,c,d,e:"
  READ*, a, b, c, d, e
  PRINT*, 'a=', a, ', b=', b, ', c=', c, ', d=', d, ', e=', e

END
```

若输入

1,2,abc,.true.,(1,-1)

则输出

a = 1 , b = 2.000000 , c = abc , d = T , e = (1.000000,-1.000000)

如果输入数据个数多于变量个数，多余的数据不起作用；如果输入数据个数少于变量个数(如使用斜杠提前结束输入)，Fortran 并不认为有错误，而是把没有输入数据的数值型变量值设为 0(0.0)，字符型变量值设为依赖于特定系统的字符串。

通常，输入数据的类型要和对应的变量类型一致。但 Fortran 允许将一个整数输入给一个实数，反之亦然；一个整数或实数可以输给一个字符型变量，但一个含有前导字符是字母的字符串不能输给一个整数或实数。

若字符串不含空格，输入时可不加界定符；但若字符串含有空格，输入时必须加界定符单引号或双引号。

如果有多个 READ\*语句出现，Fortran 规定：每一个 READ\*语句在输入数据时，必须从一个新的输入行开始。这就要求输入数据的行数至少应与 READ\*语句的个数相等。

## 二、表控输出语句

通过前面的实例演示，相信大家已经对 PRINT\*语句有了一个感性认识。PRINT\*语句的一般形式为：



PRINT\*, 输出列表

列表内容可以是常量、变量、表达式或字符串, 它们之间使用逗号分隔。

PRINT\*语句使用简单, 非常适合于较少数据的输出, 或跟踪计算过程中变量的变化情况。至于涉及有格式的复杂 PRINT 语句, 留待第九章讲解。

如果程序中有多个 PRINT\*语句, Fortran 规定: 每个 PRINT\*语句都从一个新行开始输出, 即自动换行。若一个 PRINT\*语句无任何输出项, 将在屏幕上输出一空白行。

另外, PRINT\*语句还有计算功能, 可以直接进行表达式的计算。见例 2-9 所示。

[例 2-9] 表控输出。

```
PROGRAM Main
  IMPLICIT NONE
  INTEGER :: a=1,b=5,c=8
  PRINT*, 'max(a,b,c)=' ,MAX(a,b,c)
END
```

## 小 结

- 一个简单的 Fortran(主)程序, 由声明语句和执行语句构成, 并以 END 语句结尾; Fortran 90 语句书写采取自由格式, 即一条语句可从任一系列开始, 并可容纳 0 ~ 132 个字符, 若一行写不下, 可以在续行接着书写, 续行可多达 39 个; 适当的空白空间能够增加程序的可读性, 但记号(token)内不允许出现空格; 注释部分以感叹号标识。
- Fortran 字符集包括 26 个英文字母、数字、下划线及 21 个特殊字符; 标识符名称只能由字母、数字和下划线构成, 并以字母开头, 长度可达 31 个字符。
- Fortran 90 提供的数据类型有整型、实型、复数型、逻辑型和字符型, 并以种类(KIND)参数(存储字节数)控制数据的取值范围和精度。字符型除种类可选参数(值为 1)外, 还有规定字符串长度的可选参数。
- Fortran 90 通过 IMPLICIT NONE 语句, 来强制类型声明。声明变量的通用形式为: 数据类型[[,属性]:]变量列表, 当有属性存在, 或声明变量的同时进行初始化, 声明操作符(:)是必须的。声明常量使用 PARAMETER 属性, 并在声明的同时进行赋值。
- 计算编程应特别注意算术表达式的使用: 算术运算规则及运算符的优先级和数学上的一致; 整数除法若不能整除, 结果只取商的整数部分, 小数点后的部分

被去掉；若算术表达式中的操作数类型不一致，且没有进行显式类型转换，系统自动从左到右、按运算符优先级、将低精度类型转换为高精度类型进行运算。

- 当输入/输出简单的数据时，可以使用 READ\*/PRINT\*语句，按自由格式，分别从键盘输入数据，并在屏幕上输出数据。

## 第三章 模块化程序设计——例程和模块

人们在解决现实世界中的复杂问题时，往往采取“化整为零，分而治之”的指导思想：将一个大的问题，划分成若干子问题，分别进行求解。将这种思想引入程序设计中，就形成了模块化程序设计方法：将一个实际编程问题，划分成若干功能单一的“模块”，分别加以实现。

在 Fortran 语言中，“模块”直接映射为例程(子程序和函数的统称)。Fortran 90 将例程进一步分为内部例程和外部例程，外部例程是单独的程序构造单元，还有两种程序构造单元是主程序和模块，这三种程序单元都可含有其内部例程。通用的例程放置于模块中，作为模块例程专供其他程序单元使用。

### 第一节 内部例程

#### 一、内部函数

##### (一)构造形式

在详细讲解内部函数的语法规则之前，先来看一个用牛顿法解方程的应用实例：方程的一般形式为  $f(x)=0$ ，设根的一个近似值为  $x_i$ ，新的近似值  $x_{i+1}$  表示为：

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

其中， $f'(x)$  是  $f(x)$  的一阶导数，叠代求解至  $f(x)$  接近 0。

假设  $f(x) = x^3 + x - 3$ ，那么  $f'(x) = 3x^2 + 1$ ；根的初值设为 2；叠代控制条件为  $f(x) > 10^{-6}$ ，或者叠代步数不超过 20。程序中，用  $F(x)$  代表  $f(x)$ ， $DF(x)$  代表  $f'(x)$ 。

[例 3-1] 牛顿法解方程。

```
PROGRAM Newton
```

```
  IMPLICIT NONE
```

```
  INTEGER :: Its          = 0           !叠代数
```

```
  INTEGER :: MaxIts       = 20          !最大叠代数
```

```
  LOGICAL :: Converged    = .false.     !是否收敛
```

```
  REAL    :: Eps          = 1e-6        !叠代精度
```

```
REAL      :: X          = 2          !根的初值

DO WHILE (.NOT. Converged .AND. Its < MaxIts)
  X = X-F(X) / DF(X)
  PRINT*, X, F(X)
  Its = Its + 1
  Converged = ABS( F(X) ) <= Eps
END DO

IF (Converged) THEN
  PRINT*, 'Newton converged'
ELSE
  PRINT*, 'Newton diverged'
END IF

CONTAINS

FUNCTION F(X)
  REAL F, X
  F = X ** 3 + X-3
END FUNCTION F

FUNCTION DF(X)
  REAL DF, X
  DF = 3 * X ** 2 + 1
END FUNCTION DF

END PROGRAM Newton
```

由例 3-1 可见，内部函数位于主程序的 CONTAINS 关键字和 END 语句之间，其构造形式为：

```
FUNCTION 函数名( [参数列表] )
  [声明语句]
  [执行语句]
END FUNCTION [函数名]
```

其中，[ ]为可选部分。强制类型声明 IMPLICIT NONE，其作用域为整个程序单元，所以无需在内部函数中再重写此语句。尽管可以在程序单元头部声明内部函数的函数名、参数及其有关变量，但从数据安全性考虑，还是应该在内部函数中予以声明。

值得注意的是：若在内部函数中声明了和全局变量同名的局部变量，全局变量被屏蔽，内部函数引用的是局部变量，这种现象称为同名覆盖。

### (二)全局变量和局部变量

在内部例程中引用全局变量，有时容易引发不易查找的错误。下面来看一个求  $n!$  的应用实例。

[例 3-2] 内部函数使用全局变量。

```
PROGRAM Factorial
  IMPLICIT NONE
  INTEGER I

  DO I = 1, 10
    PRINT*, I, Fact(I)
  END DO

  CONTAINS
    FUNCTION Fact( N )
      INTEGER Fact, N, Temp
      Temp = 1
      DO I = 2, N
        Temp = I * Temp
      END DO
      Fact = Temp
    END FUNCTION Fact
END PROGRAM Factorial
```

该程序输出为：

1	1
3	6
5	120
7	5040
9	362880
11	39916800

```

13  1932053504
15  2004310016
17  -288522240
19  109641728

```

程序错误的原因：I 是一全局变量，当函数 Fact 第一次被引用时，I = 1，该值被传递给虚参 N，相同的 I 在函数循环体中被赋初值 2，此时 2 > N，不执行 DO 循环 (Fact = 1)，当函数 Fact 返回到主程序打印时，I 的值为 2；下一次引用时，I 在主程序 DO 循环中增至 3，依此类推，程序决不会计算偶数阶乘。

正确的方式，应在内部函数中重新声明 I，使之成为一局部变量。在内部例程中声明所有变量应成为一条编程规则，以消除全局变量带来的负面影响。假如需要从程序单元向内部例程传递数据，最安全的方式是利用参数传递。假如需要在多个内部例程中共享大量数据，最好的解决方案是在模块(代替 Fortran 77 中的共用区)中统一声明全局变量，需要访问这些全局变量的内部例程引用该模块即可。

### (三)函数返回值

在 C/C++ 中，函数由 return 语句返回值；但在 Fortran 中，通过将表达式的值赋给函数名来实现。此时，赋值号左边的函数名不能带参数表。例如：

```
F = X ** 3 + X - 3
```

在 Fortran 90 中，函数值也可通过 RESULT 子句来返回。例如：

```

FUNCTION F(X) RESULT(R)
  REAL R, X
  R = X ** 3 + X - 3
END FUNCTION F

```

其中，R 代表函数结果，其数据类型代表函数类型；赋值号左边不再是函数名 F，而是函数结果 R，相应的函数类型声明，也由声明函数名 F 改为声明函数结果 R。

通常情况下，函数执行到最后的 END 语句才返回。但有时执行流程需要提前返回，这时可使用 RETURN 语句。不过，RETURN 语句不应滥用，否则会导致类似 GO TO 语句的问题。

### (四)语句函数

Fortran 77 提供了语句函数，Fortran 90 也予以支持，不过不推荐使用。语句函数有其特殊的规则，请看一个具体的实例：求函数  $f(x) = \frac{x^2}{\sqrt{x^2 + 2x + 1}}$  在  $x = 1, 2, 3$ 、

4、5 处的值。

[例 3-3] 语句函数的使用。

```
PROGRAM Statement_Function
  IMPLICIT NONE
  REAL f,x,y                !函数名 f、形参 x、实参 y 均需声明类型
  f(x) = x**2/SQRT(1.0 + 2.0*x + x**2)    !定义语句函数

  DO
    WRITE(*,'(A)', ADVANCE = 'NO') '输入 x 的值: '
    READ*,y
    IF(INT(y) == 0) EXIT        !0 终止循环
    PRINT*,'f(',y,')=',f(y)
  END DO
END PROGRAM Statement_Function
```

定义语句函数的形式为：

函数名(参数 1, 参数 2, ...) = 函数表达式

从中可以看出，语句函数在形式上和数学上的函数表达式完全一样。

语句函数使用时应注意以下几点：

- 语句函数先定义后使用，且只能用一条语句来定义；
- 定义语句应放在声明部分，且放在语句函数相关的类型声明之后；
- 参数列表可以为空(此时，函数实际为一常量表达式)，但函数名后边的一对括号，无论在定义还是引用时都不能省略。

## 二、内部子程序

子程序和函数的主要差别有下列几点：

- 没有返回值和子程序名关联，因此无需声明子程序类型；
- 通过 CALL 语句调用子程序；
- 在例程原型(头)和 END 语句(尾)中，使用关键字 SUBROUTINE；
- 若子程序参数表为空，子程序名后的一对括号可以省略。

通常，函数通过函数名返回一个值，而子程序通过参数可以返回多个值。实例 3-4 中的子程序，用来实现交换两个数据的算法。

[例 3-4] 通过子程序参数返回多个值。

```
PROGRAM Exchange
IMPLICIT NONE
REAL :: A = 1, B = 5

CALL Swop( A, B )
PRINT*, A, B

CONTAINS
  SUBROUTINE Swop( X, Y )
    REAL Temp, X, Y
    Temp = X
    X = Y
    Y = Temp
  END SUBROUTINE
END PROGRAM
```

---

Fortran 参数传递, 缺省为引用传递(地址传递)。子程序调用时, 实参 A、B 的值被传递给形参 X、Y, 改变后的形参值又被传回调用程序, 因此可以实现交换两个数据的目的。子程序中的 Temp 为一局部变量, 它在主程序中不可访问。

有时, 我们希望参数以值方式传递, 即形参的改变不影响实参。为此, Fortran 90 提供了 INTENT 属性(见本章第六节例程参数)。

内部子程序的构造形式为:

```
SUBROUTINE 子程序名[( 参数表 )]
  [声明语句]
  [执行语句]
END SUBROUTINE [子程序名]
```

## 第二节 主程序

一个完整的程序有且只有一个主程序, 其构造形式为:

```
[PROGRAM 程序名]
  [声明语句]
  [执行语句]
```



```
[CONTAINS  
    内部例程]  
END [PROGRAM [程序名]]
```

主程序使用时应注意以下几点：

- 主程序只有 END 语句是必须的，其他都是可选的；
- 若含有内部例程，则必须有 CONTAINS 关键字；
- 可以有多个内部例程，但内部例程不能再含有自己的内部例程，即不允许内部例程的嵌套；
- 主程序的 END 语句若出现程序名，其前面的 PROGRAM 关键字不能少。

### 第三节 外部例程

通用的例程一般作为外部例程来实现，以便被多个调用程序使用。通常，外部例程位于单独的文件中，除了头、尾部分外，外部例程和主程序在形式上是相同的。其构造形式：

```
SUBROUTINE 子程序名([ 参数表 ])  
    [声明语句]  
    [执行语句]  
[CONTAINS  
    内部例程]  
END [SUBROUTINE [子程序名]]
```

或为：

```
FUNCTION 函数名([ 参数表 ])  
    [声明语句]  
    [执行语句]  
[CONTAINS  
    内部例程]  
END [FUNCTION [函数名]]
```

外部例程和内部例程的主要差别：

- 外部例程可以含有内部例程，而内部例程不能再含有内部例程；
- END 语句中的关键字 FUNCTION/SUBROUTINE，在外部例程是可选的，但在内部例程是必须的。

下面，将实例 3-4 的内部例程 Swop 改为外部例程来实现，如实例 3-5 所示。

**[例 3-5] 外部例程的使用。**

```
PROGRAM Exchange
  IMPLICIT NONE
  EXTERNAL Swop           !声明例程 Swop 为外部的
  REAL :: A = 1, B = 5
  CALL Swop( A, B )
  PRINT*, A, B
END PROGRAM Exchange

SUBROUTINE Swop( X, Y )
  REAL Temp, X, Y
  Temp = X
  X = Y
  Y = Temp
END SUBROUTINE Swop
```

如果无意中用系统的标准例程名作了外部例程名，编译器会优先引用标准例程，而不是用户定义的外部例程。为避免出现这种情况，可在调用程序的声明部分，添加外部例程声明语句(EXTERNAL)，这应该成为一个编程惯例。

假如一个文件包含多个外部例程，可考虑将外部例程移至模块中，从而将外部例程转换为模块例程。

## 第四节 接口块

要正确地调用例程，编译器需要知道例程的有关信息：例程名、参数个数及各自的数据类型等，这些信息的集合称为例程的接口。标准例程、内部例程和模块例程的接口对编译器透明，因此这些接口是显式的；但对外部例程，编译器无从知道其接口信息，即外部例程的接口是隐式的。

在调用程序中声明外部例程时，只提供了外部例程名，此时的接口仍是隐式的。当遇到可选参数这样的复杂情况时，还需要提供进一步的接口信息，编译器才能对外部例程产生正确的调用。为此，Fortran 90 提供了接口块，以向调用程序明确外部例程的接口信息。接口块的构造形式为：

---

INTERFACE

接口体

END INTERFACE

接口体由外部例程头、参数声明和外部例程尾构成。其中，参数名可以和外部例程定义用的参数名不同。不过，为了省事，通常都是将外部例程定义的参数声明部分直接拷贝过来。

[例 3-6] 针对上述 Swop 外部例程，在主程序(调用程序)中添加接口块。

---

PROGRAM Exchange

IMPLICIT NONE

INTERFACE

SUBROUTINE Swop( X, Y )

REAL Temp, X, Y

END SUBROUTINE Swop

END INTERFACE

REAL :: A = 1, B = 5

CALL Swop( A, B )

PRINT\*, A, B

END PROGRAM Exchange

---

有了接口块详细规定，就不需要用 EXTERNAL 语句声明外部例程了。

在例 3-6 中，外部例程的接口信息比较简单，可以不使用接口块；但下列情况下，必须使用接口块：

- 外部例程具有可选参数；
- 例程用来定义操作符重载；
- 外部函数返回数组或变长字符串；
- 外部例程具有假定形状数组、指针或目标参数；
- 例程作参数；
- 例程重载。

除了操作符和例程重载外，其他的只要将外部例程转换为模块例程，就可免去提供接口块的麻烦。所以，模块在 Fortran 90 中具有十分重要的作用。

## 第五节 模 块

### 一、构造形式

前面讲过, Fortran 90 有三种程序单元: 主程序、外部例程和模块, 模块主要用来在程序单元之间共享数据和操作例程。

实例 3-7 将例程 Swop 置于模块中, 出于演示目的, 在模块中还声明了一实型常量 PI。

[例 3-7] 模块的使用。

---

```
MODULE MyUtils
  REAL, PARAMETER :: PI = 3.1415927
CONTAINS
  SUBROUTINE Swop( X, Y )
    REAL Temp, X, Y
    Temp = X
    X = Y
    Y = Temp
  END SUBROUTINE Swop
END MODULE MyUtils

PROGRAM Main
  USE MyUtils           !位于 IMPLICIT NONE、声明语句之前
  IMPLICIT NONE
  REAL :: A = PI, B = PI*2

  CALL Swop( A, B )
  PRINT*, A, B
END PROGRAM Main
```

---

例程 Swop 已转换为模块例程。主程序通过引用(USE)模块, 就可使用当中的模块例程及数据。注意: USE 语句须位于 IMPLICIT NONE 语句之前; 若模块和主程序放于同一个文件中, 模块应位于主程序的前面。

模块的构造形式为：

MODULE 模块名

[声明语句]

[CONTAINS

模块例程]

END [MODULE [模块名]]

模块例程作为模块的内部例程，其形式和主程序、外部例程包含的内部例程是一样的，只不过模块例程还可含有自己的内部例程。内部例程和三种程序单元之间的层次关系如图 3-1 所示，其中 Mod-sub 指模块例程、Int-subs 指内部例程、Ext-sub 指外部例程。

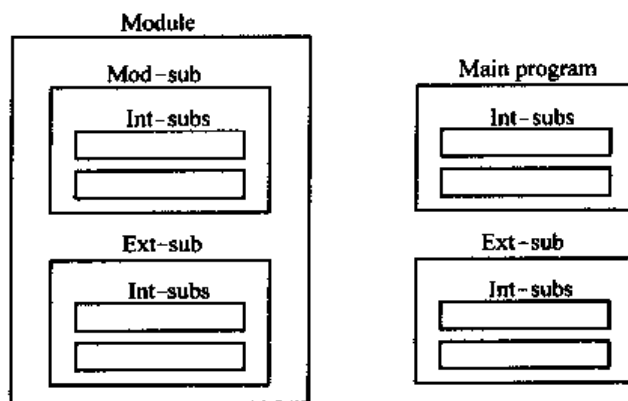


图 3-1 内部例程和程序单元的层次关系

Fortran 77 还有一种数据块程序单元，用来对有名共用区的数据对象(变量)进行初始化。在 Fortran 90 中，数据块程序单元已被模块程序单元所代替。

值得注意的是：模块不仅可供主程序和外部例程引用，还可供其他模块引用。在开发模块库时，应注意模块的层次关系，使得后来开发的模块只引用早期模块。这样，最终的客户程序只引用最新开发的模块即可。

主程序、模块和外部例程三种程序单元之间的关系，如图 3-2 所示。

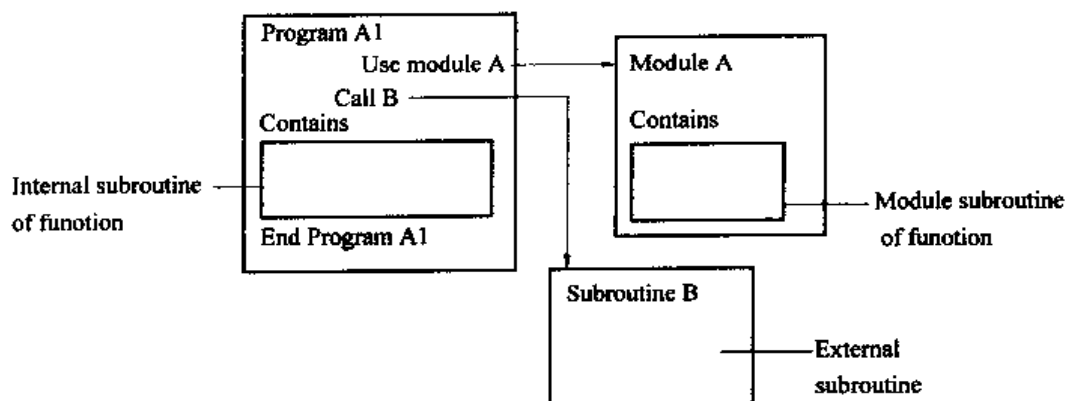


图 3-2 主程序、模块和外部例程之间的关系

## 二、USE 语句

要使用模块中的实体(数据和例程), 须先引用模块。其一般引用形式为:

USE 模块名

有时, 不方便直接使用模块实体名, 比如: 要使用两个独立模块的同名例程, 或模块实体具有一个很长的名称, 这时, 可在客户程序中重新命名模块实体。例如, 模块 YourMod 有一个例程或变量 YourPlonk, 被重命名为 MyPlonk:

USE YourMod, MyPlonk => YourPlonk

其模块引用形式为:

USE 模块名, 重命名列表(新实体名 => 原实体名, 新实体名 => 原实体名)

有时, 某个客户程序只使用模块中的部分实体。这时, 可采取如下的引用形式:

USE YourMod, ONLY : X, Y

只使用模块 YourMod 中的 X 和 Y。冒号后边的实体也可被重命名。

假如某个客户程序要同时引用多个模块, 每个模块都得使用单独的 USE 语句, 并使 USE 语句出现在 IMPLICIT NONE 之前, 而模块的先后次序无关紧要。

## 三、PUBLIC 和 PRIVATE 属性

模块也能够实现信息隐藏: 将模块内部使用的实体隐藏起来, 使外部程序只使用公共部分的实体。模块缺省访问属性为 PUBLIC, PRIVATE 属性将实体访问限于模块内。例如:

REAL, PRIVATE :: X

将实型变量 X 规定为私有的, 即只能在模块内访问。

下面的声明则同时规定变量 X、例程 Swop 为私有的:

PRIVATE X, Swop

也可采取不带实体列表的统一声明形式:

PRIVATE

PUBLIC Swop

除 Swop 外, 其余的模块实体都是私有的。

## 第六节 例程参数

例程在定义时, 列表中的参数只是特定数据类型的占位符(因此称为形参或虚参), 系统不会为它们分配存储单元; 当例程被调用或引用时, 列表中的参数才被分配一定的存储单元, 并接收外部实参传递进来的值; 例程执行完毕, 例程中的参数

所占用的存储空间被系统自动释放，所保存的参数值也随之消失。

例程中的局部变量也有同样的性质：例程执行时“生存”，例程不执行时“消亡”。除非被声明具有 SAVE 属性，在这种情况下，局部变量就是静态变量，从例程被调用到程序结束，一直保存有特定的值。

### 一、参数传递

实参和虚参之间的数据传递有两种方式：一是引用传递，二是值传递。以引用方式传递数据时，实际上是将实参的内存地址传递给虚参，这样例程中虚参的变化就会反映到实参中；而以值方式传递数据时，是将实参值的一份拷贝传递给虚参，虚参的变化不会影响到实参。

在 Fortran 中，参数(包括数组参数)通常是以引用方式传递，即地址传递，但在实参为常量和表达式的情况下，参数是以值方式传递。若将一变量实参括起来，该实参就转化为表达式，表达式以值方式传递。如：

```
CALL Sub((A), B)
```

其中，(A)是表达式，以值方式传递。

通常，函数参数不应发生改变；而子程序参数中，有的需向子程序传入数据，有的需从子程序中传出数据。为确保参数按用户的意愿进行传递，Fortran 90 提供了 INTENT 属性。请看下面的实例：

```
SUBROUTINE SUB( X, Y, Z )  
    REAL, INTENT(IN)      :: X  
    REAL, INTENT(OUT)     :: Y  
    REAL, INTENT(INOUT)   :: Z  
    ...  
END SUBROUTINE SUB
```

其中，INTENT(IN)表示向例程传入数据，拥有该属性的虚参不允许改变；INTENT(OUT)表示传出数据，对应的实参必须是一变量；INTENT(INOUT)表示传进/传出数据，对应的实参也必须是一变量。

假如一个虚参没有规定 INTENT 属性，对应的实参可以是变量，也可以是常量或表达式。

建议所有的虚参都规定 INTENT 属性，特别是函数参数应规定为 INTENT(IN) 属性。

## 二、参数类型匹配

[例 3-8] 参数类型匹配。

---

```
PROGRAM Main
  IMPLICIT NONE
  EXTERNAL Sub
  INTEGER :: X=1
  CALL Sub(X)
  PRINT*,'X=',X
END PROGRAM

SUBROUTINE Sub(A)
  IMPLICIT NONE
  REAL,INTENT(INOUT) :: A
  A=A+1
END SUBROUTINE
```

---

虚参 A 声明为实型，对应的实参声明为整型，实参和虚参类型不匹配导致出现错误的结果。

其实，整数与实数在计算机中是以不同方式表达的。由于例程中虚参为实型，故此，实参传进的整型数 1 被认为是实数 1.0，而实数 1.0 的计算机整型表达为 1065353216。使用 Visual Fortran 自带的工具 BITVIEWER 进行察看，上述情况便会得到证实。

若将实例中的外部例程改为内部例程或模块例程，编译器编译时就会检查出实参和虚参类型不匹配的错误。这也说明编译器一般不会对外部例程的参数类型匹配进行检查，除非在调用程序中建立了外部例程的接口块。

## 三、可选参数

参数列表可能很长，但有时并不是所有的参数都需要传递，在这种情况下，可以规定部分或全部参数具有可选(OPTIONAL)属性。假如参数列表既有必选参数又有可选参数，那么所有的必选参数必须放在可选参数之前，也就是先必选，后可选。例如一个外部例程 Sub 有 6 个参数，最后 4 个是可选的，下面是调用程序中建立的接口块：



---

```
INTERFACE
```

```
  SUBROUTINE Sub( DumU, DumV, DumW, DumX, DumY, DumZ )
```

```
    REAL DumU, DumV, DumW, DumX, DumY, DumZ
```

```
    OPTIONAL DumW, DumX, DumY, DumZ
```

```
  END SUBROUTINE
```

```
END INTERFACE
```

---

下面是不同的调用语句：

---

```
CALL Sub( A, B )                                ! 1
```

```
CALL Sub( A, B, C, D )                          ! 2
```

```
CALL Sub( A, B, DumX = D, DumY = E, DumZ = F ) ! 3
```

---

上列第 1 种调用，只有必选参数被传递；第 2 种调用，2 个必选参数和前面的 2 个可选参数被传递；第 3 种调用，2 个必选参数和后面的 3 个可选参数被传递。

第 3 种调用表明，可选参数可以不按声明的列表次序进行调用。若不按列表次序调用，可选的虚参名必须被引用，即提供关键字参数列表。一旦某个可选参数使用了关键字，其后面的可选参数都得使用关键字。比如，下面的调用就是错误的：

```
CALL Sub( A, B, DumX = D, E, F )
```

这里要特别强调的是：外部例程使用可选参数，需要在调用程序中建立其接口块。

在编写带有可选参数的例程时，通常赋给可选参数一默认值。调用时，若不向可选参数传递数据，就使用默认值。下面来看一个实例：编写一函数，计算  $F(X)=A \cdot X^2 + B \cdot X + C$  的值，其中， $X$  的值是必需的， $A$ 、 $B$ 、 $C$  的值则可有可无。

**[例 3-9] 可选参数的使用。**

---

```
MODULE Mod
```

```
  IMPLICIT NONE
```

```
  CONTAINS
```

```
    REAL FUNCTION Func(X,A,B,C)
```

```
      ! 计算  $FUNC(X)=A \cdot X^2+B \cdot X+C$ 
```

```
      ! A,B,C 不传入，值为 0
```

```
      REAL, INTENT(IN) :: X ! X 值一定要传入
```

---

```

REAL, OPTIONAL, INTENT(IN) :: A,B,C ! A,B,C 可以不传入
REAL RA, RB, RC

RA=0.0;RB=0.0;RC=0.0      !几个简单的赋值语句放在一行
IF ( PRESENT(A) ) RA=A    !检查可选参数是否存在
IF ( PRESENT(B) ) RB=B
IF ( PRESENT(C) ) RC=C
Func = RA*X**2 + RB*X + RC

END FUNCTION
END MODULE

PROGRAM Main
  USE Mod
  IMPLICIT NONE
  PRINT*, Func(2.0, C=1.0)      ! F(2)=0*2^2 + 0*2 + 1 = 1
  PRINT*, Func(2.0, B=1.0, A=2.0) ! F(2)=2*2^2 + 1*2 + 0 = 10
END PROGRAM

```

---

## 第七节 例程重载

前面讲过，程序调用例程时，实参和虚参的类型必须匹配。可是，系统提供的一些标准函数，比如求绝对值函数 ABS，既能接收整数，也能接收实数，甚至是复数，这似乎违反了要求类型匹配的原则，其实这里采用的是例程重载的技巧。

重载的概念在一些现代编程语言中早已存在(如 C++中的函数重载、操作符重载)。在 Fortran 90 中，例程重载是指不同参数列表的例程被赋予相同的名字，例程调用时，编译器会依据所传递的实参类型，按实参和虚参类型匹配的原则，调用或引用相应的例程。例如：一个子程序 Swop(X, Y)，其功能是交换两个数的值。当中的两个参数，可以是实数，也可以是整数。如例 3-10 所示。

[例 3-10] 例程重载。

---

```

MODULE Mod
  IMPLICIT NONE

```

```
INTERFACE Swop    !Swop 是调用时使用的例程名
  MODULE PROCEDURE SwopReals, SwopIntegers
END INTERFACE

CONTAINS

  SUBROUTINE SwopReals( X, Y )
    REAL, INTENT(INOUT) :: X,Y
    REAL Temp
    Temp = X
    X = Y
    Y = Temp
  END SUBROUTINE

  SUBROUTINE SwopIntegers( X, Y )
    INTEGER, INTENT(INOUT) :: X,Y
    INTEGER Temp
    Temp = X
    X = Y
    Y = Temp
  END SUBROUTINE
END MODULE

PROGRAM Main
  USE Mod
  IMPLICIT NONE
  REAL    :: A = 1.0 , B = 2.0
  INTEGER :: I = 1, J = 2

  CALL Swop( A, B )
  CALL Swop( I, J )
  PRINT*, 'A=', A, ', B=', B
  PRINT*, 'I=', I, ', J=', J
END PROGRAM
```

在模块中建立重载例程接口块，只需列出具体的模块例程，并以调用时的例程名命名接口块：

```
INTERFACE Swop
  MODULE PROCEDURE SwopReals, SwopIntegers
END INTERFACE
```

如果具体的例程作为外部例程来实现，在调用程序中建立的接口块应为：

```
INTERFACE Swop
  SUBROUTINE SwopReals( X, Y )
    REAL, INTENT(INOUT)      :: X,Y
  END SUBROUTINE

  SUBROUTINE SwopIntegers( X, Y )
    INTEGER, INTENT(INOUT)   :: X,Y
  END SUBROUTINE
END INTERFACE
```

和一般外部例程接口块相比，以上建立的接口块只是多了个接口名(即调用时的例程名)。

## 第八节 递归例程

在一个例程体内出现直接或间接调用例程自身的语句，称该例程为递归例程。递归的典型例子是计算  $n!$ ，其递归公式为：

$$n! = \begin{cases} 1 & (n=1) \\ n \times (n-1)! & (n>1) \end{cases}$$

Fortran 90 支持递归，但必须添加 RECURSIVE 关键字，在例程是函数的情况下，还须添加 RESULT 子句。下面分别用递归函数和递归子程序求  $n!$ 。

[例 3-11] 递归函数。

```
PROGRAM Main
  IMPLICIT NONE
  INTEGER I
```

```
DO I=1,10
    PRINT*, I,'!=', Factorial(I)
END DO

CONTAINS

RECURSIVE FUNCTION Factorial( N ) RESULT (Fact)
    INTEGER Fact, N
    IF( N == 1 ) THEN
        Fact = 1
    ELSE
        Fact = N * Factorial( N-1 )
    END IF
END FUNCTION

END PROGRAM
```

---

RESULT 关键字在非递归情况下是可选的，换句话说，在函数中采用 RESULT 子句不论在哪一种情况下都是正确的。值得注意的是：在递归函数体内引用函数自身时，用的是函数名(Factorial)；赋值给函数时，左值用的是结果名(Fact)。

下面列出上述函数的非递归形式，以示区别：

---

```
FUNCTION Factor( N ) RESULT (Fact)
    INTEGER Fact, N, I
    Fact = 1
    DO I = 2, N
        Fact = I * Fact    !左、右均用结果名
    END DO
END FUNCTION
```

---

可以看出：递归算法简单直观，容易编写。但递归算法执行效率低，原因是递归包含递推和回归两个过程，系统分别要进行进栈和出栈操作。例如，求 5! 的递推过程为：

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

$$2! = 2 \times 1!$$

$$1! = 1$$

再按上述相反的过程回归计算：

$$2! = 2 \times 1! = 2$$

$$3! = 3 \times 2! = 6$$

$$4! = 4 \times 3! = 24$$

$$5! = 5 \times 4! = 120$$

所以，递归程序设计除了要找出递归表达式外，还要确立递归的终止条件(如： $1! = 1$ )；无限递归没有任何实际意义。

在递归子程序下，同样须添加 RECURSIVE 关键字。我们将上述求  $n!$  的算法，重写为递归子程序(例 3-12)。

[例 3-12] 递归子程序。

---

PROGRAM Main

IMPLICIT NONE

INTEGER F, I

DO I = 1, 10

CALL Factorial( F, I )

PRINT\*, I, '!', F

END DO

CONTAINS

RECURSIVE SUBROUTINE Factorial( F, N )

INTEGER F, N

IF (N == 1) THEN

F = 1

ELSE

CALL Factorial( F, N-1 )

F = N \* F

END IF

END SUBROUTINE

END PROGRAM

---

编写该递归子程序的关键，是将赋值语句  $F = N * F$  置于递归调用语句之后。

另外，不论是递归调用子程序，还是递归引用函数，实参都要递减(比如：实参  $N-1$ )，这样才有可能从未知向已知方向移动，直至达到终止条件。若递归例程是外部例程，须在调用程序中建立其接口块。

### 小 结

- 模块化程序设计思想：将大的程序分解为若干个功能单一的例程。例程可以是包含在程序单元中的内部例程，也可以是作为独立程序单元使用的外部例程。Fortran 90 程序单元包括主程序、外部例程和模块。其中，主程序和外部例程包含的内部例程不能再包含内部例程，而模块中的模块例程允许包含其内部例程，即允许例程嵌套。
- 若外部例程接口信息较简单，可以通过 EXTERNAL 关键字将例程声明为外部的，以防止调用程序使用 and 外部例程同名的标准例程；若外部例程接口信息复杂或某些特殊情况下，必须要建立接口块，以使编译器可以产生正确的调用。编译器自动为标准例程、内部例程和模块例程提供显式接口。
- 模块通常含有全局变量和通用例程，专供其他程序单元使用。在通过 USE 语句引用模块时，可只使用其中的部分实体，还可为模块实体重命名。模块中实体的缺省访问属性为 PUBLIC，PRIVATE 属性则将实体访问属性限定在模块内。
- 实参和虚参间的数据传递必须类型匹配。实参变量以引用方式传递，常量和表达式则以值方式传递，编程中应通过 INTENT 属性明确规定参数的传递方式。Fortran 90 允许声明可选参数，并允许以关键字形式使用可选参数。具有可选参数的外部例程，使用时须在调用程序中建立其接口块。
- 重载例程通过有名接口块来规定，重载的具体例程可以是模块例程，也可以是外部例程，但要求规定不同的接口体。
- Fortran 90 支持递归，即允许例程直接或间接地调用自身。前提：声明例程时要添加 RECURSIVE 关键字，在函数例程情况下，还要添加 RESULT 子句。

## 第四章 结构化程序设计——控制结构

业已证明,任何复杂的算法都可由三种基本结构来实现。这三种基本结构就是顺序结构、选择(分支)结构和循环结构。按此三种结构进行程序设计,称之为结构化程序设计。

早期的 Fortran 程序只有顺序结构和以 IF 语句为代表的选择结构,若要执行稍为复杂的流程,不得不在程序中添加许多 GOTO 无条件转移语句,结果使程序的可读性和可维护性变得很差。后来添加了 IF 块(Fortran 90 还有 CASE 块)和循环结构,从而使 Fortran 程序设计真正跨入结构化阶段。

本章着重讲解选择结构和循环结构。

### 第一节 选择结构

#### 一、关系及逻辑运算符

选择结构中的条件判别式通常由逻辑表达式组成。逻辑表达式中的运算符,包括算术运算符、关系运算符和逻辑运算符。其中,算术运算符已在第二章中介绍过。Fortran 90 的关系运算符和逻辑运算符分别列于表 4-1 和表 4-2。

表 4-1

关系运算符

关系运算符(Fortran 77 对应运算符)	代表的意义
==(EQ.)	相等
/(NE.)	不相等
>(GT.)	大于
>=(GE.)	大于或等于
<(LT.)	小于
<=(LE.)	小于或等于

表 4-2

逻辑运算符

逻辑运算符	含义	优先级
.NOT.	逻辑非	1
.AND.	逻辑与	2
.OR.	逻辑或	3
.EQV.	逻辑等	4
.NEQV.	逻辑不等	4
.XOR.	逻辑异或	4



这三种运算符的优先级由高到低依次为：算术运算符、关系运算符和逻辑运算符。

## 二、IF 语句

IF 语句是最早的选择结构，其构造形式为：

IF(condition) statement

假如条件为真，就执行条件后的那条语句；否则，程序流程跳过 IF 语句，接着往下执行。IF 语句只能写在一行上。

[例 4-1] IF 语句和 IF 块的对比。

---

```

PROGRAM Main      !IF 语句与 IF 块
  IMPLICIT NONE
  LOGICAL L1,L2
  CHARACTER CH1,CH2
  WRITE(*,'(A)',ADVANCE='NO') 'L1=(Y/N):'
  READ*,CH1
  WRITE(*,'(A)',ADVANCE='NO') 'L2=(Y/N):'
  READ*,CH2
  L1=(CH1 == 'Y' .OR. CH1 == 'y')
  L2=(CH2 == 'Y' .OR. CH2 == 'y')
  CALL Proc_Old(L1,L2)
  CALL Proc_New(L1,L2)
  CONTAINS
    SUBROUTINE Proc_Old(L1,L2)
      LOGICAL,INTENT(IN) :: L1,L2
      INTEGER I,J

      IF (.NOT.L1) GOTO 10
        I = 1
        J = 2
        GOTO 30
    10  IF (.NOT.L2) GOTO 20
        I = 2
        J = 3
  
```

```
        GOTO 30
20      I = 3
        J = 4
30      CONTINUE
        PRINT*, 'Old: I=', I, ', J=', J
END SUBROUTINE

SUBROUTINE Proc_New(L1,L2)
    LOGICAL, INTENT(IN) :: L1, L2
    INTEGER I, J

    IF (L1) THEN
        I = 1
        J = 2
    ELSE IF (L2) THEN
        I = 2
        J = 3
    ELSE
        I = 3
        J = 4
    END IF
    PRINT*, 'New: I=', I, ', J=', J
END SUBROUTINE

END PROGRAM
```

---

从例 4-1 中可以看出，IF 语句的功能极为有限，为了执行稍复杂的流程控制，不得不结合使用 GOTO 无条件转移语句，结果使程序代码的可读性大为降低。

这里顺便提一下 Fortran 77 中的算术 IF 语句。算术 IF 语句被 Fortran 90 认为是一种过时的语句，因为它的语意不清晰，不符合现代程序设计语言规范。算术 IF 语句的构造形式为：

IF (算术表达式) 语句标号 1，语句标号 2，语句标号 3  
如果算术表达式的值小于 0，程序执行语句标号 1(后面的语句)；如果值等于 0，执行语句标号 2；如果值大于 0，执行语句标号 3。如下列子程序所示：

---

```
SUBROUTINE Proc(A,B,C)
    ! 求方程  $A \cdot X^2 + B \cdot X + C$  的根, 用算术 IF 语句
    REAL, INTENT(IN) :: A,B,C
    REAL X1, X2, Disc

    IF (A == 0) GOTO 40
    Disc = B * B - 4 * A * C

    IF (Disc) 10, 20, 30
        10 PRINT*, 'Complex roots'
            GOTO 40
        20 X1 = -B / (2 * A)
            PRINT*, 'Two equal real roots:', X1
            GOTO 40
        30 X1 = (-B + SQRT( Disc )) / (2 * A)
            X2 = (-B - SQRT( Disc )) / (2 * A)
            PRINT*, 'X1=', X1, ', X2=', X2

    40 RETURN
END SUBROUTINE
```

---

### 三、IF 块

IF 块的构造形式为:

IF condition THEN

    blockI

[ELSE

    blockE]

END IF

其中, condition 是一个逻辑表达式, 其结果不外乎真或假。blockI 和 blockE 是语句块, 当条件为真, blockI 被执行; 否则, blockE 被执行。ELSE 块是可选的。

现在来看一个实例: 将学生得分划分成 5 个等级(A,B,C,D 和 F), 分别统计每个等级的人数。用 IF 语句(IF 块的简略形式)编写的代码段为:

```
IF (Final >= 90) A = A + 1
IF (Final >= 80 .AND. Final < 90) B = B + 1
IF (Final >= 70 .AND. Final < 80) C = C + 1
IF (Final >= 60 .AND. Final < 70) D = D + 1
IF (Final < 60) F = F + 1
```

---

上述代码段中有 5 个 IF 语句。一个学生的得分只能是上述 5 种情况当中的 1 种，但程序仍然要做 5 次判断，所以算法效率较低。我们用 ELSE IF 块重写上述代码段：

---

```
IF (Final >= 90) THEN
    A = A + 1
ELSE IF (Final >= 80) THEN
    B = B + 1
ELSE IF (Final >= 70) THEN
    C = C + 1
ELSE IF (Final >= 60) THEN
    D = D + 1
ELSE
    F = F + 1
END IF
```

---

程序执行上述代码段时，一旦发现某个逻辑条件为真，就不会再去判断其他的逻辑条件。最不利的情况下(比如得分 50)，系统才要判断 4 次。这里的算法效率明显高于前一种。由此，给出 IF 块更一般的构造形式：

```
IF (logical-expr1) THEN
    block1
ELSE IF (logical-expr2) THEN
    block2
ELSE IF (logical-expr3) THEN
    block3
...
ELSE
```

blockE

END IF

假如 logical-expr1 为真, block1 被执行; 假如 logical-expr1 为假, logical-expr2 被估值, 如果该表达式是真, block2 被执行; 假如没有表达式是真, blockE 被执行。值得一提的是, ELSE IF 结构要求合理排列逻辑条件, 使得一次只能有一个逻辑条件为真。

一个 IF 块可以被命名, 以增加程序的可读性。如:

---

```
GRADE: IF (Final >= 60) THEN
```

```
    PRINT*, 'Pass'
```

```
ELSE GRADE
```

```
    PRINT*, 'Fail'
```

```
END IF GRADE
```

---

IF 块可以嵌套使用, 如求方程  $ax^2 + bx + c = 0$  根的子程序:

---

```
SUBROUTINE Proc(A,B,C)
```

```
! 求方程  $A \cdot X^2 + B \cdot X + C$  的根
```

```
    REAL, INTENT(IN) :: A,B,C
```

```
    REAL X1, X2, Disc
```

```
    Disc = B * B - 4 * A * C
```

```
    Outer: IF (A /= 0) THEN
```

```
        Inner: IF (Disc < 0) THEN
```

```
            PRINT*, 'Complex roots'
```

```
        ELSE Inner
```

```
            X1 = (-B + SQRT( Disc )) / (2 * A)
```

```
            X2 = (-B - SQRT( Disc )) / (2 * A)
```

```
            PRINT*, 'X1=', X1, ', X2=', X2
```

```
        END IF Inner
```

```
    END IF Outer
```

```
END SUBROUTINE
```

---

其中, 距 ELSE 最近的 IF 语句是内层的 IF 语句, 故此, ELSE 属于内层 IF 块。从

中可见，命名 IF 块，可以使代码变得更清晰。

#### 四、CASE 块

CASE 块是 Fortran 90 新添加的分支结构。我们先来看一个实例：从键盘输入一个字符，判断是元音字母，还是辅音字母，当输入“@”时，程序结束。实例如下：

---

```
SUBROUTINE Proc
  CHARACTER CH
  DO
    WRITE(*, '(A)', ADVANCE='NO') 'Input a character:'
    READ*, CH
    IF (CH == '@') EXIT
    IF (CH >= 'A' .and. CH <= 'Z' .or. CH >= 'a' .and. &
      CH <= 'z') THEN
      SELECT CASE (CH)
        CASE ('A', 'E', 'I', 'O', 'U', 'a', 'e', 'i', &
          'o', 'u')
          PRINT*, 'Vowel'
        CASE DEFAULT
          PRINT*, 'Consonant'
      END SELECT
    ELSE
      PRINT*, 'Something else'
    END IF
  END DO
END SUBROUTINE
```

---

若完全用 IF 块来实现上述功能要编写很长的程序代码，且程序的逻辑难以理解。

CASE 块的构造形式为：

```
SELECT CASE (expr)
  CASE (selector1)
    block1
  CASE (selector2)
```

```
        block2  
    [CASE DEFAULT  
        blockD]  
END SELECT
```

其中, `expr` 必须是整型、字符型或逻辑型变量。`Selector` 可以是同一类型、非交叉的值或值范围的列表。如

```
CASE( 'a':'h', 'i':'n', 'o':'z', '_' )
```

这里的冒号被用来规定值的一个范围。若一个范围的上界被省略, 当 `expr` 大于或等于范围的下界时, 当前的语句块被执行; 若下界被省略, 当 `expr` 小于或等于范围的上界时, 当前的语句块被执行。

和 IF 块一样, CASE 块也可被命名。下面用 CASE 块, 改写前面统计学生得分的代码段:

```
SELECT CASE ( INT(Final) )  
    CASE (90:)  
        A = A + 1  
    CASE (80:89)  
        B = B + 1  
    CASE (70:79)  
        C = C + 1  
    CASE (60:69)  
        D = D + 1  
    CASE DEFAULT  
        F = F + 1  
END SELECT
```

在有多个分支的情况下, 使用 CASE 块要比 IF 块方便。

## 第二节 循环结构

### 一、确定性循环

确定性 DO 循环的构造形式为:

```
[name:] DO 变量 = 表达式 1, 表达式 2 [, 表达式 3]  
        语句块
```

**END DO [name]**

在确定性 DO 循环中，变量一般为整型变量，表达式 1、表达式 2 和表达式 3 一般也是整型表达式，其中表达式 3 是可选的。它们表示的意义如下：

- 变量用来控制循环次数，一般称为循环变量；
- 表达式 1 代表循环变量的初值；
- 表达式 2 代表循环变量的终值；
- 表达式 3 代表循环变量的步长，如果步长为 1，表达式 3 可省略。

循环次数可按下面的公式计算：

$\text{MAX}((\text{表达式 2} - \text{表达式 1} + \text{表达式 3}) / \text{表达式 3}, 0)$

系统在进行确定性循环时，先按上述公式计算循环次数。如果循环变量的步长为 0，就会发生除 0 运算的错误。这一点在编程中应引起注意。

下面给出几个 DO 循环的代码段，以加深理解。

```
DO I = 2, 7, 2
```

```
    WRITE(*, '(I3)', ADVANCE = 'NO') I
```

```
END DO !输出: 2 4 6
```

```
DO I = 5, 4
```

```
    WRITE(*, '(I3)', ADVANCE = 'NO') I
```

```
END DO !不执行循环
```

```
DO I = 5, 1, -1
```

```
    WRITE(*, '(I3)', ADVANCE = 'NO') I
```

```
END DO !输出: 5 4 3 2 1
```

```
DO I = 1, 6, -2
```

```
    WRITE(*, '(I3)', ADVANCE = 'NO') I
```

```
END DO !不执行循环
```

假如将 DO 循环简记为：

```
DO I = A, B, C
```

根据具体情况，DO 循环可分为以下 4 种：

(1)  $C > 0$ ,  $A < B$ 。执行循环体，循环变量从 A 开始，第二次循环以后，每次增加一个步长 C，直到循环变量的终值超过 B。



(2)  $C > 0, A > B$ 。不执行循环体。

(3)  $C < 0, A > B$ 。执行循环体，循环变量从  $A$  开始，第二次循环以后，每次减小一个步长的绝对值  $|C|$ ，直到循环变量的终值小于  $B$ 。

(4)  $C < 0, A < B$ 。不执行循环体。

如果在循环结构执行后要引用循环变量，请注意此时循环变量的值：第(1)种情况， $I > B$ ；第(3)种情况， $I < B$ 。

Fortran 77 中的 DO 循环为：

DO 语句标号，循环变量 = 表达式 1，表达式 2 [, 表达式 3]

其中，循环变量、表达式 1、表达式 2 和表达式 3 的意义同上。请看下面的函数：

```

INTEGER FUNCTION FACT(N)
    INTEGER, INTENT(IN) :: N
    INTEGER I
    FACT=1
    DO 10, I=1, N
        FACT=FACT*I
    10 CONTINUE
END FUNCTION

```

这种 DO 循环被 Fortran 90 认为是过时的。

DO 循环要求循环变量为整数，但在科学与工程计算中，经常会遇到非整数的情况。比如，物体在重力作用下做垂直运动，从时刻  $T_0$ (0.0 秒)到时刻  $T_1$ (5.0 秒)，求在每一时间间隔  $DT$ (0.4 秒)下的物体位置。

显然，这里需将时间段划分成多个时段： $(T_1 - T_0) / DT$ 。要计算最后一个时段末的位置，需将该值加 1，即  $(T_1 - T_0) / DT + 1$ 。但循环变量要求是整数，若用 NINT 函数对此取整，则会超出原来的循环范围；若用 INT 截断，又会损失循环步数。Fortran 90 的 SPACING 函数可以很好地解决这一问题：

$INT((T_1 - T_0) / DT + SPACING(DT)) + 1$

即在截断前增加一个  $SPACING(DT)$ 。例 4-2 给出完整的程序代码。

[例 4-2] 非整数循环变量的处理。

```

PROGRAM Vertical
! Vertical motion under gravity
IMPLICIT NONE

```

```
REAL, PARAMETER :: G = 9.8
REAL              dT, S, T, TStart, TEnd
REAL              :: U = 60
INTEGER           I, TRIPS
WRITE(*, '(A)', ADVANCE='NO') 'TStart, TEnd, DT : '
READ*, TStart, TEnd, dT
TRIPS = INT( (TEnd - TStart) / dT + SPACING(dT) ) + 1
PRINT*, 'TRIPS = ', TRIPS
T = TStart
DO I = 1, TRIPS
    S = U * T - G / 2 * T * T
    PRINT*, I, T, S
    T = T + dT
END DO
END PROGRAM Vertical
```

---

DO 循环也可以嵌套使用，见例 4-3。

[例 4-3] DO 循环的嵌套。

---

```
PROGRAM Main
    IMPLICIT NONE
    INTEGER I, J
    DO I = 1, 9
        DO J = 1, 9
            PRINT*, I, ' * ', J, ' = ', I * J
        END DO
    END DO
END PROGRAM
```

---

## 二、非确定性循环

非确定性循环，其循环次数事先无法确定，结束循环条件要在循环过程中进行设置。先来看一个猜数程序(例 4-4)，当中的三个内部过程分别用不同的非确定性循环来实现。

[例 4-4] 三种非确定性循环的对比。

PROGRAM Main

IMPLICIT NONE

INTEGER,PARAMETER :: Weight = 45 !答案

INTEGER :: Guess = 0 !猜测值

!CALL Proc\_1

!CALL Proc\_2

CALL Proc\_3

CONTAINS

SUBROUTINE Proc\_1

DO

WRITE(\*, '("Your guess: ")', ADVANCE = 'NO' )

READ\*, Guess

IF (Guess == Weight) EXIT

IF (Guess > Weight) THEN

PRINT\*, 'Too high. Try again'

ELSE

PRINT\*, 'Too low. Try again'

END IF

END DO

PRINT\*, "You're right!"

END SUBROUTINE

SUBROUTINE Proc\_2

DO

WRITE(\*, '("Your guess: ")', ADVANCE = 'NO' )

READ\*, Guess

IF (Guess == Weight) THEN

PRINT\*, "You're right!"

ELSE IF (Guess > Weight) THEN

PRINT\*, 'Too high. Try again'

ELSE

PRINT\*, 'Too low. Try again'

```
        END IF
        IF (Guess == Weight) EXIT
    END DO
END SUBROUTINE

SUBROUTINE Proc_3
    DO WHILE (Guess /= Weight)
        WRITE(*, '("Your guess: ")', ADVANCE = 'NO' )
        READ*, Guess
        IF (Guess > Weight) THEN
            PRINT*, 'Too high. Try again'
        ELSE
            PRINT*, 'Too low. Try again'
        END IF
    END DO
    PRINT*, "You're right!"
END SUBROUTINE
END PROGRAM
```

---

在例 4-4 中，共出现了 3 种非确定性 DO 循环。

(1)第一种：

```
DO
    IF (logical-expr) EXIT
    block
END DO
```

(2)第二种：

```
DO
    block
    IF (logical-expr) EXIT
END DO
```

(3)第三种：

```
DO WHILE (logical-expr)
    block
END DO
```

第一种循环和第三种循环可以相互转化, 即 DO WHILE 循环等价于:

```
DO
  IF (.NOT.logical-expr) EXIT
  block
END DO
```

Fortran 90 中的 EXIT 命令, 用于跳出 DO 循环。原则上, EXIT 可以放在 DO 循环中的任何位置, 但为使代码更具可读性, 通常将 EXIT 放在 DO 循环头部, 或放在 DO 循环末尾。

Fortran 90 与循环有关的命令, 除 EXIT 外还有 CYCLE。见例 4-5。

[例 4-5] CYCLE 的使用。

```
PROGRAM Main
  IMPLICIT NONE
  INTEGER I
  DO I = 1, 10
    IF ( MOD(I, 5) == 0 ) CYCLE
    PRINT*,I
  END DO
END PROGRAM
```

DO 循环遇到 CYCLE 命令, 其后面的程序代码被跳过, 程序流程回到 DO 循环开头, 接着执行下一次循环。

类似于 IF 块、CASE 块, DO 循环也可以各取一个名字, 以方便代码的阅读。

### 小 结

- 任何算法都可由三种基本结构来实现。选择和循环控制结构的出现, 使 Fortran 真正步入结构化程序设计阶段。
- 一般的选择结构 IF 块的构造形式为 IF THEN [ELSE IF THEN] [ELSE] END IF, 使用时应合理排列逻辑条件, 使得一次只能有一个逻辑条件为真。在只有一个分支的情况下, IF 块简化为 IF 语句。当只对单一的整型、字符型或逻辑型变量/表达式进行判断时, 宜采用简明的 CASE 结构。
- 确定性循环的构造形式为 DO I = A, B, C END DO, 它取消了语句标号, 并以 END DO 代替 CONTINUE; 非确定性循环的构造形式为 DO END DO 和 DO WHILE END DO, 前者须在循环体中增加 IF 语句, 以便在适当条件下跳

出(EXIT)循环。

- 不论是 IF 块、CASE 块，还是 DO 循环块，都可以被命名，以增加程序的可读性。

## 第五章 数 组

处理大量数据时,数组是不可缺少的工具。现代编程语言几乎都提供了数组, Fortran 90 的数组功能比其他语言所无法比拟的。在 Fortran 90 中,不但可以逐个元素地对数组进行操作,还可以对数组整体、数组段直接进行操作; Fortran 90 提供了针对数组操作的构造块和函数;为有效利用内存, Fortran 90 还提供了动态数组;另外, Fortran 90 数组的隐式循环和数组赋值更是别具特色。

### 第一节 数组声明

我们先来看下列数组声明实例:

```
REAL, DIMENSION(15)      :: X          ! 下界缺省为 1
```

```
REAL, DIMENSION(1:5,1:3) :: Y
```

```
REAL, DIMENSION(-4:0,1:3) :: Z
```

- 维(rank)——代表下标个数。X 为一维数组, Y 和 Z 为二维数组。
- 界(bounds)——X 下界 1、上界 15, Y 下界 1 和 1、上界 5 和 3, Z 下界-4 和 1、上界 0 和 3。

- 度(extent)——维上的元素个数。X 度为 15, Y 和 Z 度为 5 和 3。
- 大小(size)——总的元素个数,或特定维上的元素个数。X、Y 和 Z 的大小为 15。
- 形状(shape)——由维和度决定。X 形状为(/15/), Y 和 Z 形状为(/5,3/)。
- 一致的(conformable)——形状相同的数组是一致的或兼容的,一致的数组才能相互赋值。Y 和 Z 形状相同,因此它们是一致的。

Fortran 90 数组声明的一般形式为

```
TYPE, DIMENSION([dl:] du[, [dl:] du]...) :: Arr
```

```
TYPE [::] Arr([dl:] du[, [dl:] du]...)
```

TYPE 代表数据类型, dl 和 du 分别为维的下界和上界, Arr 为数组变量。下列都是合法的数组声明:

```
REAL, DIMENSION(100)          :: R
```

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

```
REAL                          :: T(10,10)
```

```
REAL, DIMENSION(-10: -1)      :: X
```

```

INTEGER, PARAMETER          :: lda = 5
REAL, DIMENSION(0:lda-1)    :: Y
REAL, DIMENSION(1+lda*lda,10) :: Z

```

上列数组声明说明：

- 上、下界可以任意规定；
- 缺省下界为 1；
- 可以省略 DIMENSION 属性,如 T；
- 数组大小可以是 0。

Fortran 77 中, 声明数组分两步进行: 先声明数组类型, 再声明数组维数及大小。即:

```
DataType ArrayName
```

```
DIMENSION ArrayName(Size1, Size2, Size3, ...)
```

下面来看实例: 一个数据文件保存有学生学号、姓名和成绩三列数据, 其行数未知, 要求从该文件中读取全部数据, 并在屏幕上予以显示。例 5-1 和 5-2 分别使用一维数组和二维数组。

[例 5-1] 一维数组的使用。

```
PROGRAM Main
```

```
IMPLICIT NONE
```

```
INTEGER, PARAMETER :: MAX = 100      !最大行数
```

```
CHARACTER(20) NO(MAX),NAME(MAX)     !学号、姓名
```

```
REAL MARK(MAX)                      !成绩
```

```
INTEGER IO,I,N                      !N 代表实际行数
```

```
OPEN (1, FILE = 'DATA.TXT')
```

```
READ( 1, *, IOSTAT = IO ) ( NO(I),NAME(I),MARK(I), I = 1, MAX )
```

```
IF (IO < 0) THEN                    !遇文件尾
```

```
    N = I - 1
```

```
ELSE
```

```
    N = MAX
```

```
END IF
```

```
PRINT*,N
```

```
WRITE(*,'(2A,F4.1)') ( NO(I),NAME(I),MARK(I), I = 1, N )
```

```
CLOSE(1)
```

```
END PROGRAM
```



---

**[例 5-2] 二维数组的使用。**

---

```
PROGRAM Main
  IMPLICIT NONE
  INTEGER, PARAMETER :: MAX = 100    !最大行数
  CHARACTER(20) Stud(MAX,3)          !学号、姓名、成绩，共 3 列
  INTEGER IO,I,J,N
  OPEN (1, FILE = 'DATA.TXT')
  READ( 1, *, IOSTAT = IO ) ( (Stud(I,J), J = 1, 3), I = 1, MAX )
  IF (IO < 0) THEN
    N = I - 1
  ELSE
    N = MAX
  END IF
  PRINT*,N
  PRINT*, ((Stud(I,J), J = 1, 3), I = 1, N )
  CLOSE(1)
END PROGRAM
```

---

将数据文件中的学号、姓名声明为字符数组，成绩声明为实型数组。在从文件中读数据和向屏幕输出数据的过程中，使用了数组隐式循环。READ 语句使用了 IOSTAT 可选参数，该参数若为-1，表示文件指针已到达文件尾，此时引用循环变量，须将其值减 1。

## 第二节 数组存储

在 Fortran 77 中，数组在内存中按列主方式存储。以二维数组为例，先存储第一列，然后是第二列、第三列，直至最后一列；在 Fortran 90 中，有些场合下数组按列主方式存储(简称列主存储)，而另一些场合下并没有规定按什么方式存储(简称自由存储)。

### 一、自由存储

不规定数组在内存中如何存储，便于编写可移植的程序，编译器可以自由地实现存储优化。例如，在分布式计算环境中，一个大的数组可以被存储到 100 个处理器中，每个存储数组中的部分元素。在 High Performance Fortran 中，自由存储被广

泛应用。

## 二、列主存储

在有些场合下,比如:向另一种语言编写的例程传递数组、数组构造、数组输入/输出、系统提供的一些数组函数(TRANSFER, RESHAPE, PACK, UNPACK 和 MERGE)等,则需要明确给出数组的存储方式——列主存储。

# 第三节 数组操作

## 一、数组赋初值

数组赋初值,可使用 DATA 语句,也可省略 DATA,还可使用隐式循环。

### (一)使用 DATA 的一般形式

使用 DATA 的一般形式为:

```
INTEGER A(5)
```

```
DATA A /1,2,3,4,5/
```

!A(1), A(2), A(3), A(4), A(5)的值分别为 1, 2, 3, 4, 5

### (二)使用乘号(\*)

DATA 的数据区中,还可以使用乘号“\*”来表示数据的重复(重复的次数写在前面)。例如:

```
DATA A /5*3/ !A(1), A(2), A(3), A(4), A(5)的值均为 3
```

### (三)使用隐式循环

隐式循环可以用来设置数组的初值。例如:

```
DATA (A(I),I=2,4) /2,3,4/
```

!A(2), A(3), A(4)的值分别为 2, 3, 4

隐式循环可以认为是 DO 循环的简略形式。在隐式循环中,同样可设置循环变量的增量(或步长)。例如:

```
DATA (A(I),I=1,5,2) /1,3,5/
```

!A(1), A(3), A(5)的值分别为 1, 3, 5

隐式循环也可以嵌套使用。例如:

```
INTEGER B(2,2), I, J
```

```
DATA ((B(I,J),I=1,2),J=1,2) /1,2,3,4/
```

### (四)省略 DATA

Fortran 90 还可以省略 DATA 关键字,直接给数组设置初值。例如:

```
INTEGER :: A(5)=(/1,2,3,4,5/)
```

省略 DATA，仍可使用隐式循环。不过，所赋初值的个数要与数组元素的个数相等。另外，在这种情况下隐式循环的构造形式也有所不同：

```
INTEGER I
```

```
INTEGER :: A(5)=(/1,(2,I=2,4),5/)    !A(2), A(3), A(4)的值为 2, A(1)和 A(5)
                                         的值分别为 1 和 5
```

隐式循环除了用来给数组赋初值外，还可用来输出数组。例如：

```
PRINT*,((B(I,J),I=1,2),J=1,2)
```

## 二、数组整体操作

Fortran 90 可以对数组进行整体操作，大大简化了其他语言需要使用循环才能完成的操作。现举几个实例来说明：

```
A = 5
```

其中，A 是任意维数及大小的数组。该语句将数组 A 所有元素的值设为 5。

```
A = (/1,2,3/)
```

其中，A(1)=1，A(2)=2，A(3)=3。所提供的数据个数必须跟数组 A 的大小一样。

```
A = B
```

其中，A 和 B 是形状完全相同的数组。该语句将数组 A 相应位置元素的值设置成同数组 B。

```
A = B + C
```

```
A = B - C
```

```
A = B * C
```

```
A = B / C
```

其中，A、B 和 C 是三个形状完全相同的数组。上述语句分别将数组 B 和 C 相应位置元素的值相加、相减、相乘和相除，得到的结果再赋给数组 A 对应元素。

```
A = Sin(B)
```

数组 A 的每一个元素为数组 B 相应元素的 Sin 值，数组 B 须是实型数。

```
A = B > C
```

A、B 和 C 是三个形状完全相同的数组。不过，A 为逻辑型数组，B 和 C 为同类型的数值型数组。就一维情况而言，该语句相当于下面的代码段：

```
DO I=1,N
```

```
  IF (B(I)>C(I)) THEN
```

```
    A(I)=.TRUE.
```

```

      ELSE
        A(I)=.FALSE.
      END IF
    END DO

```

### 三、数组段操作

除一次性对整个数组进行操作外，还能对数组段进行操作。数组段的下标三元组形式为：

[<bound1>]: [<bound2>] [:<stride>]

数组段起始于下标 bound1，终止于下标 bound2，步长为 stride。如：

A(:)           ! 整个数组  
 A(3:9)       ! A(3) ~ A(9)，缺省步长 1  
 A(3:9:1)     ! A(3) ~ A(9)，缺省步长 1  
 A(m:n)       ! A(m) ~ A(n)，步长 1  
 A(m:n:k)     ! A(m) ~ A(n)，步长为 k  
 A(8:3:-1)    ! A(8) ~ A(3)，步长为-1  
 A(8:3)       ! A(8) ~ A(3)，步长 1，大小 0  
 A(m:)        ! A(m) ~ A(上界)，步长 1  
 A(:n)        ! A(下界) ~ A(n)，步长 1  
 A(:,2)       ! A(下界) ~ A(上界)，步长 2  
 A(m:m)       ! 1 个元素的数组段

数组段的操作语法类似于隐式循环，请看下面的实例：

A(3:5) = 5

其中，将 A(3)、A(4)、A(5) 的值设置为 5，其他值不变。

A(3:) = 5

其中，将 A(3)之后所有元素的值设置为 5，其他值不变。

A(3:5) = (/3,4,5/)

其中，将 A(3)、A(4)、A(5) 的值分别设为 3、4、5，其他值不变(等号左边数组元素的个数必须和等号右边数据个数一样多)。

A(1:3) = B(4:6)

其中，设置 A(1) = B(4)、A(2) = B(5)、A(3) = B(6)(等号两边的数组元素的个数必须一样多)。

A(1:5:2) = 3

其中, 将 A(1)、A(3)、A(5) 的值设为 3。类似于隐式循环, 最后一个数 2 用来设置步长。

$A(1:10) = A(10:1:-1)$

其中, 将 A(1:10) 翻转过来。即将 A(1) 设为原来的 A(10), A(2) 设为原来的 A(9), 依此类推。

$A(:) = B(:,2)$

其中, 假设 A 和 B 分别声明为 INTEGER A(5)、INTEGER B(5,2), 这里将二维数组 B 第 2 列的 5 个元素的值赋给一维数组 A 的 5 个元素(等号两边的数组元素的个数必须一样多)。

#### 四、数组输出

数组元素、数组整体和数组段都可表控输出(PRINT\*)。

设矩阵 A 为  $\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$ , 下列为输出语句:

---

PRINT*, 'Array element	=', a(3,2)
PRINT*, 'Array section	=', a(:,1)
PRINT*, 'Sub - array	=', a(:2,:2)
PRINT*, 'Whole Array	=', a
PRINT*, 'Array Transp"d	=', TRANSPOSE(a)

---

分别在屏幕上输出:

---

Array element	= 6
Array section	= 1 2 3
Sub - array	= 1 2 4 5
Whole Array	= 1 2 3 4 5 6 7 8 9
Array Transposed	= 1 4 7 2 5 8 3 6 9

---

#### 五、WHERE 构造

WHERE 是 Fortran 90 新添加的功能, 用来取出部分数组内容进行操作。上一节是由数组下标值规则地使用一部分数组元素, 而 WHERE 是按逻辑判断, 使用满足

条件的那部分数组元素。

先来看一个实例：假设每年收入 3 万元以下，所得税为 10%；收入为 3 万元到 5 万元之间，所得税为 12%；收入 5 万元以上，所得税为 15%。使用 WHERE 构造计算应上交的所得税。

[例 5-3] WHERE 构造的使用

```
PROGRAM Main
  IMPLICIT NONE
  REAL :: income(10) = (/25000,30000,50000,40000,&
    35000,60000,27000,45000,20000,70000/)
  REAL :: tax(10) = 0
  INTEGER I
  CALL Proc_Where ; PRINT*,tax
  CALL Proc_IF ; PRINT*,tax
  CONTAINS
    SUBROUTINE Proc_Where ! Where 构造
      WHERE (income < 30000.0)
        tax = income * 0.1
      ELSEWHERE (income < 50000.0)
        tax = income * 0.12
      ELSEWHERE
        tax = income * 0.15
      END WHERE
    END SUBROUTINE

    SUBROUTINE Proc_IF ! IF 构造
      DO I = 1, 10
        IF (income(I) < 30000.0) THEN
          tax(I) = income(I) * 0.1
        ELSE IF (income(I) < 50000.0) THEN
          tax(I) = income(I) * 0.12
        ELSE
          tax(I) = income(I) * 0.15
        END IF
      END DO
    END SUBROUTINE
```

---

```
END DO
END SUBROUTINE
END PROGRAM
```

---

从例 5-3 中可以看出, WHERE 构造类似于 DO 循环内嵌 IF 块, 但使用 WHERE 构造的程序代码比较精简。WHERE 构造的语法格式为:

---

```
WHERE (logical - expr1)
    block1
ELSEWHERE (logical - expr2)
    block2
ELSEWHERE (logical - expr3)
    block3
...
ELSEWHERE
    blockE
END WHERE
```

---

如果只有一条执行语句, 那么可以将这条执行语句写在 WHERE 后面, 并省略 END WHERE。此时, WHERE 构造就转化为 WHERE 语句。例如:

```
WHERE (income > 50000) income = 50000
WHERE 语句和 IF 语句在构造形式上是相同的:
WHERE (logical - expr) 执行语句
WHERE 构造也可以被命名。例如:
```

---

```
name: WHERE (income > 50000)
    income = 50000
END WHERE name
```

---

WHERE 构造也可以嵌套使用。例如:

---

```
WHERE (income < 50000.0)
    WHERE(income < 30000.0)
        tax = income * 0.1
```

```
ELSEWHERE
    tax = income * 0.12
END WHERE
ELSEWHERE
    tax = income * 0.15
END WHERE
```

---

## 六、FORALL 构造

FORALL 是 Fortran 95 新添加的功能。它通过隐式循环来使用数组，不过它的功能更为强大。下面先看两个简单实例：

[例 5-4] FORALL 构造的使用。

---

```
PROGRAM Main
    IMPLICIT NONE
    INTEGER I,J,A(5,5)
    CALL Proc_1
    WRITE(*,'(5I)') ((A(I,J),J=1,5),I=1,5)
    CALL Proc_2
    WRITE(*,'(5I)') ((A(I,J),J=1,5),I=1,5)
    CONTAINS
        SUBROUTINE Proc_1
            FORALL(I = 1 : 5 , J = 1 : 5)
                A(I,J) = I * J
            END FORALL
        END SUBROUTINE

        SUBROUTINE Proc_2
            DO I = 1 , 5
                DO J = 1 , 5
                    A(I,J) = I * J
                END DO
            END DO
        END SUBROUTINE
```



---

END PROGRAM

---

例 5-4 中，内部例程 Proc\_1 用 FORALL 构造来实现，Proc\_2 则用对应的 DO 循环来实现。

[例 5-5] FORALL 语句的使用。

---

```

PROGRAM Main
  IMPLICIT NONE
  INTEGER I,J
  INTEGER,PARAMETER :: N = 5
  INTEGER A(N,N)
  CALL Proc_1
  WRITE(*,'(5I)') ((A(I,J),J=1,N),I=1,N)
  CALL Proc_2
  WRITE(*,'(5I)') ((A(I,J),J=1,N),I=1,N)
  CONTAINS
    SUBROUTINE Proc_1
      FORALL(I=1:N,J=1:N, I<J) A(I,J)=1      !上三角
      FORALL(I=1:N,J=1:N, I==J) A(I,J)=2    !对角线
      FORALL(I=1:N,J=1:N, I>J) A(I,J)=3     !下三角
    END SUBROUTINE

    SUBROUTINE Proc_2
      DO I = 1, N
        DO J = 1,N
          IF (I < J) A(I,J)=1                !上三角
          IF (I == J) A(I,J)=2               !对角线
          IF (I > J) A(I,J)=3                !下三角
        END DO
      END DO
    END SUBROUTINE
  END PROGRAM

```

---

例 5-5 中，内部例程 Proc\_1 用 FORALL 语句来实现，Proc\_2 则用对应的 DO

循环结合 IF 语句来实现。

FORALL 语句的一般形式：

FORALL (循环表达式 1 [,循环表达式 2]…[,条件判别式]) 执行语句

FORALL 构造的一般形式：

[name:] FORALL (循环表达式 1 [,循环表达式 2]…[,条件判别式])

FORALL 语句块

END FORALL [name]

其中，循环表达式，相当于 DO 循环中的表达式 1、表达式 2 和表达式 3(代表步长，步长为 1 时可省略)，循环表达式的数量和数组维数相对应；条件判别式，是逻辑表达式，若省略该项，则条件为真(.TRUE.)，条件判别式可使用循环表达式中的循环变量；FORALL 语句块，要被赋值的变量须是数组元素或数组段，且须引用出现在循环表达式中的所有循环变量(或数组下标)，赋值表达式不能是字符表达式；Name 是给 FORALL 构造规定的名字。

FORALL 可以嵌套，还可以在 FORALL 构造中使用 WHERE，不过在 WHERE 构造中不能使用 FORALL。如下面的代码段所示：

---

```

FORALL (I=1:5)  !FORALL 的嵌套及在 FORALL 中使用 WHERE
  FORALL(J=1:2)
    B(I,J)=2
  END FORALL
  WHERE (B(:,I) /= 0)
    B(:,I)=1.0 / B(:,I)
  END WHERE
END FORALL

```

---

## 七、矢量下标

一维矢量作为数组下标，用来规定一个数组段，该数组段中的元素次序未必是线性的。例如：

```
INTEGER, DIMENSION(5) :: V=(/1,4,8,12,10/)
```

```
INTEGER, DIMENSION(3) :: W=(/1,2,2/)
```

A(V)标识了一个非规则数组段：A(1)、A(4)、A(8)、A(12)和 A(10)，如图 5-1 所示：

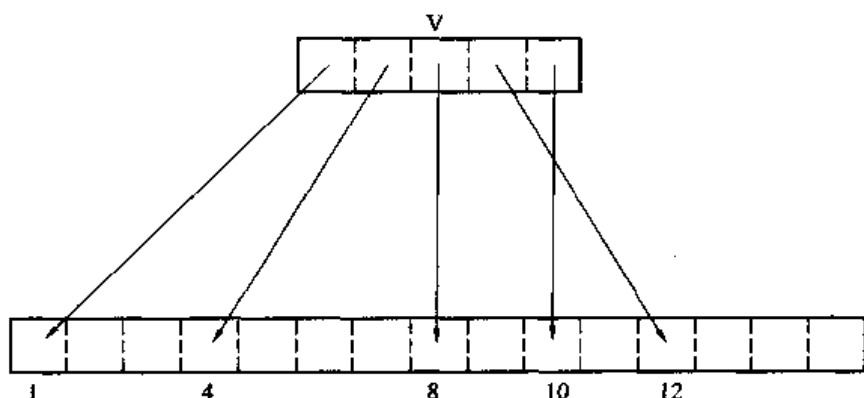


图 5-1 非规则数组段

$$A(V) = 3.5$$

是将 3.5 赋给上列 5 个元素。同样：

$$C(1:3,1) = A(W)$$

是将数组段  $C(1:3,1)$  设为  $A(1)$ 、 $A(2)$ 、 $A(2)$ 。

矢量下标可以用在赋值操作符(=)的任何一边,但为了保持并行计算环境下数组操作的完整性,赋值操作符左边的数组下标必须惟一。因此,对  $A(W)$  进行赋值是非法的( $A(2)$ 被赋值两次)。

一维矢量也可作为二维数组的下标。例如：

$$A(1) = \text{SUM}(C(V,W))$$

值得注意的是,矢量下标效率非常低,不应轻易使用。

## 八、数组标准函数

Fortran 90 提供了许多针对数组操作的标准函数。假设下列数组声明：

`REAL, DIMENSION(-10:10,23,14:28) :: A`

则：

- `LBOUND(SOURCE[,DIM])`——返回指定维的数组下界。如：

$$\text{LBOUND}(A) = (/ -10, 1, 14 /) \text{ (数组)}$$

$$\text{LBOUND}(A,1) = -10 \text{ (标量)}$$

- `UBOUND(SOURCE[,DIM])`——返回指定维的数组上界。如：

$$\text{UBOUND}(A) = (/ 10, 23, 28 /) \text{ (数组)}$$

$$\text{UBOUND}(A,1) = 10 \text{ (标量)}$$

- `SHAPE(SOURCE)`——返回数组形状。如：

$$\text{SHAPE}(A) = (/ 21, 23, 15 /) \text{ (数组)}$$

$$\text{SHAPE}(/4/) = (/ 1 /) \text{ (数组)}$$

- SIZE(SOURCE[,DIM])——返回数组(或指定维)的元素个数。如：

SIZE(A,1) = 21

SIZE(A) = 7245

- RESHAPE(SOURCE, SHAPE)——改变数组形状。如：

B = RESHAPE((/1,2,3,4/),(/2,2/))      ! INTEGER::B(2,2)



• ALL(X)——相当于逻辑与操作。当逻辑数组 X 中的所有元素为真时，该函数返回真；否则，返回假。

• ANY(X)——相当于逻辑或操作。当逻辑数组 X 中有任何元素为真时，该函数返回真；否则，返回假。

- SUM(X)——返回数值型数组所有元素的和。如下列代码段所示：

```
INTEGER, DIMENSION(5,5) :: A
```

```
REAL X(3), Y(3)
```

```
...
```

```
IF (ANY(A > 0)) A = 1
```

```
IF (ALL(A == 0)) A = -1
```

```
Dot = SUM( X * Y )
```

## 第四节 数组参数

、作为例程参数使用的数组有固定形状、假定大小和假定形状数组三种形式，不管是哪一种形式的数组参数，其传递均采取引用方式传递。

### 一、固定形状数组

固定形状数组，其维上具有明确的上、下界，其形状规定采取如下形式：

([dl:] du[, [dl:] du]...)

dl 和 du 分别代表维的下界和上界，若下界省略，其缺省值为 1。维的上、下界取整数。

假如至少有一维的界由非常量表达式表示，这样的数组称为大小可调数组，其实际大小待例程调用时方能确定。非常量表达式中的变量，要么是虚参，要么是公

用区中的变量。如例 5-6 所示。

**[例 5-6]** 固定形状数组(可调大小数组)。

---

```

PROGRAM Main
  IMPLICIT NONE
  REAL,DIMENSION(3,2) :: A1=(/1.0,2.0,3.0,4.0,5.0,6.0/)

  PRINT*, THE_SUM(A1,3,2)
  CONTAINS
    FUNCTION THE_SUM(A, M, N)
      INTEGER M,N,I,J
      REAL A(M, N),THE_SUM,SUMX  ! A 为可调大小数组
      SUMX = 0.0
      DO J = 1, N
        DO I = 1, M
          SUMX = SUMX + A(I, J)
        END DO
      END DO
      THE_SUM = SUMX
    END FUNCTION
  END PROGRAM

```

---

函数 THE\_SUM 中的虚参数组 A 为可调大小数组, 运行时, 可调大小数组 A 的大小由与虚参 M 和 N 对应的实参决定。

## 二、假定形状数组

假定形状数组, 不明确规定维的上界。其形状规定形式为:

([dl]:[, [dl]:]...)

下界 dl 若省略, 缺省值为 1。例 5-7 展示了假定形状数组参数的使用。

**[例 5-7]** 假定形状数组。

---

```

PROGRAM Main
  IMPLICIT NONE
  REAL,DIMENSION(3:5,2:3) :: A1=(/1.0,2.0,3.0,4.0,5.0,6.0/)

```

---

---

```

PRINT*, THE_SUM(A1)
CONTAINS
  FUNCTION THE_SUM(A)
    REAL A(-1:, :), THE_SUM, SUMX    ! A 为假定形状数组
    INTEGER I, J
    SUMX = 0.0
    DO J = 1, UBOUND(A, 2)
      DO I = -1, UBOUND(A, 1)
        SUMX = SUMX + A(I, J)
      END DO
    END DO
    THE_SUM = SUMX
  END FUNCTION
END PROGRAM

```

---

假定形状数组虚参采取和实参数组相同的形状，或者说，实参数组将形状传递给虚参数组，然后两者按列主方式逐个元素进行对应。例 5-7 中，实参数组和虚参数组的对应关系为：

A1: A1(3,2), A1(4,2), A1(5,2), A1(3,3), A1(4,3), A1(5,3)

A: A(-1,1), A(0,1), A(1,1), A(-1,2), A(0,2), A(1,2)

程序中采用 UBOUND 函数，直接获取对应后的虚参数组上界。

若外部例程采用了假定形状数组参数，则须在调用程序中建立其接口块。例如：

---

```

INTERFACE
  FUNCTION THE_SUM(A)
    REAL A(-1:, :)
  END FUNCTION
END INTERFACE

```

---

### 三、假定大小数组

假定大小数组的形状规定为：

([expli - shape - spec,]... [dl:]\*)

expli – shape – spec 代表固定形状规定, dl 指最后一维的下界(缺省值为 1), \* 指最后一维的上界。所谓“假定大小”, 是指虚参数组和实参数组的大小相同。例 5-8 展示了假定大小虚参数组的使用。

[例 5-8] 假定大小数组。

```
PROGRAM Main
  IMPLICIT NONE
  REAL,DIMENSION(3,2) :: A1=(/1.0,2.0,3.0,4.0,5.0,6.0/)
  INTEGER                :: ROW=3, COL=2

  PRINT*, THE_SUM(A1,ROW,COL)
  CONTAINS
    FUNCTION THE_SUM(A,M,N)
      INTEGER I,J,M,N
      REAL A(M, *), THE_SUM, SUMX ! A 为假定大小数组
      SUMX = 0.0
      DO J = 1,N
        DO I = 1,M
          SUMX = SUMX + A(I, J)
        END DO
      END DO
      THE_SUM = SUMX
    END FUNCTION
  END PROGRAM
```

值得注意的是:不能运用 UBOUND 函数来获得假定大小数组最后一维的上界。既然虚参数组(假定大小数组)和实参数组的元素个数相同,就可以在例程中使用代表行、列数的参数,来对数组中的各元素进行循环。

## 第五节 动态数组

有时,数组的实际大小事先无法确定,为适应所有可能的情况,通常声明一个超大的数组,这无疑会浪费内存空间。Fortran 77 还不支持动态数组, Fortran 90 则对动态数组提供了支持,可以在程序执行时,再决定数组的实际大小和为数组动态

分配存储空间，待不需要时，再将动态分配给数组的内存释放掉，从而高效地利用内存资源。动态数组的使用一般要经历三个步骤：

(1)声明动态数组。规定数组的维数，但不给出维上的大小和上、下界(故此，称为延迟形状数组)。如：REAL, DIMENSION(:), ALLOCATABLE :: X。

(2)给动态数组分配内存。如：ALLOCATE( X(N) )。

(3)将分配的内存释放掉。如：DEALLOCATE( X )。

下面给出两个实例，以说明动态数组的使用。

[例 5-9] 动态数组的使用之一。

---

```

PROGRAM Main
  IMPLICIT NONE
  INTEGER Students           !学生人数
  REAL,ALLOCATABLE :: Mark(:) !声明动态数组，学生成绩
  INTEGER I
  WRITE(*,'(A)',ADVANCE='NO') 'How many students: '
  READ*, Students
  ALLOCATE(Mark(Students))   !动态分配内存
  DO I=1,Students
    WRITE(*,'(No. ',I,'"s mark: ')",ADVANCE='NO') I
    READ*,Mark(I)
  END DO
  PRINT*,Mark
  DEALLOCATE(Mark)           !释放内存
END PROGRAM

```

---

根据输入的学生人数，动态设置数组的大小。注意：设置动态数组大小可以使用变量，而声明一般数组大小却要使用常量。

[例 5-10] 动态数组的使用之二。

---

```

PROGRAM Main
  IMPLICIT NONE
  CHARACTER(20), DIMENSION(: , :), ALLOCATABLE :: X, OldX
  CHARACTER(20), DIMENSION(3) :: A
  INTEGER IO,I,J,N

```

---



```

OPEN (1, FILE = 'DATA.TXT')
CALL Proc_1
CALL Proc_2
CLOSE(1)
CONTAINS
  SUBROUTINE Proc_1
    !先确定文件记录数 N, 再重读文件中的数据
    N=0
    DO
      READ( 1, *, IOSTAT = IO ) !将变量列表置空
      IF (IO < 0 ) EXIT
      N=N+1
    END DO
    ALLOCATE (X(N,3))           !动态分配内存
    REWIND(1)                   !定位文件指针到文件头
    READ(1, *) (( X(I,J), J=1,3 ), I = 1, N )
    PRINT*,N
    PRINT*,(( X(I,J), J=1,3 ), I = 1, N )
    DEALLOCATE(X)               !释放内存
  END SUBROUTINE

  SUBROUTINE Proc_2
    !在读文件的过程中, 动态分配内存和释放内存
    ALLOCATE( X(0,3) )          !可分配 0 大小的内存
    N=0
    REWIND(1)
    DO
      READ( 1, *, IOSTAT = IO ) (A(I),I=1,3)
      IF (IO < 0 ) EXIT
      N=N+1
      ALLOCATE( OldX(N-1, 3) )
      OldX = X                  !数组整体赋值
      DEALLOCATE( X )
      ALLOCATE( X(N,3) )
    END DO
  END SUBROUTINE

```

```

      X(:N-1,:) = OldX
      X(N,:) = A(:)
      DEALLOCATE( OldX )

    END DO
    PRINT*,N
    PRINT*,(( X(I,J), J=1,3 ), I = 1, N )
    DEALLOCATE(X)                !释放内存

  END SUBROUTINE
END PROGRAM

```

在例 5-1 和例 5-2 中,采用固定大小数组来读入文件中的三列数据。而在例 5-7 中,则采用了动态数组:例程 Proc\_1 是先确定文件行数(或记录数),再为动态数组分配适当大小的存储单元;例程 Proc\_2 是在确定文件行数的过程中,边分配边释放,直至最后一行。

注意:在使用上, Fortran 90 的动态数组与其他语言(如 Visual Basic)不同,要增加动态数组的大小,须将原来的动态数组释放掉,再重新指定动态数组的大小。

计算机的内存是有限的,用户不可能无休止地请求分配内存,所以 ALLOCATE 分配内存并不总是成功的。鉴于此,系统提供了可选状态参数 STAT,以检查内存配置是否成功。如:

```
ALLOCATE(X(SIZE),STAT=ERROR)
```

当中的 ERROR 为事先声明好的整型变量。如果 ERROR 等于 0,则表示内存配置成功,否则内存配置失败。

与动态分配内存有关的函数还有 ALLOCATED,用来检查一个动态数组是否已配置了内存,其返回值为逻辑真或者假。例如:

```
IF (.NOT. ALLOCATED(X)) ALLOCATE( X(0,3) )
```

另外,动态数组不能作为例程虚参来使用。

## 第六节 数组型函数

在 Fortran 90 中,函数可以返回一个值(标量),也可以返回多个值(矢量)——数组。如例 5-11 所示。

[例 5-11] 函数的返回值为二维字符数组。

```
MODULE Mod
```

IMPLICIT NONE

CONTAINS

```

FUNCTION FileRow(FileName)                !返回文件中的数据行数
  CHARACTER(10), INTENT(IN) :: FileName
  INTEGER FileRow
  INTEGER N, IO
  OPEN (1, FILE = FileName)
  N=0
  DO
    READ( 1, *, IOSTAT = IO )              !将变量列表置空
    IF (IO < 0 ) EXIT
    N=N+1
  END DO
  FileRow=N
  CLOSE(1)
END FUNCTION

```

```

FUNCTION FileData(FileName,N)              !返回二维字符数组
  CHARACTER(10), INTENT(IN) :: FileName
  INTEGER, INTENT(IN) :: N                !文件中的行数
  CHARACTER(10), DIMENSION(N,3) :: FileData, X !二维字符数组
  INTEGER I,J
  OPEN (1, FILE = FileName)
  READ(1, *) (( X(I,J), J = 1, 3 ),I = 1,N) !数组隐式循环
  FileData=X                               !数组整体赋值
  CLOSE(1)
END FUNCTION

```

END MODULE

PROGRAM Main

USE Mod

IMPLICIT NONE

CHARACTER(10), DIMENSION(:,,:), ALLOCATABLE :: X !动态数组

```

      INTEGER N,I,J
      N=FileRow('Data.txt')
      ALLOCATE(X(N,3))
      X=FileData('Data.txt',N)
      PRINT*,N
      WRITE(*,'(3A)') (( X(I,J),J=1,3), I=1, N )
      DEALLOCATE(X)
END PROGRAM

```

模块 Mod 中包含了两个函数：FileRow 返回文件中的行数，FileData 返回二维字符数组。主程序中，声明了二维动态数组 X，依据函数 FileRow 返回的行数动态分配大小，将函数 FileData 返回的数组直接赋给动态数组 X，动态数组使用完后进行释放。

若返回数组的函数为外部函数，须在调用程序中建立接口块。在例 5-11 中，返回数组的函数置于模块中，模块例程无需建立接口块。

## 小 结

- Fortran 90 的数组声明形式为 TYPE, DIMENSION([dl:] du[, [dl:] du]...) :: Arr 或 TYPE [:] Arr([dl:] du[, [dl:] du]...)。
- Fortran 77 的数组按列主方式存储，但在 Fortran 90 中，一些场合下数组按列主方式存储，而另一些场合下并不规定数组按何种方式存储。
- 数组赋初值一般使用 DATA 语句，其中还可使用隐式循环；Fortran 90 允许对数组元素、数组段及数组整体进行操作(包括输出操作)；Fortran 90 的 WHERE 构造相当于 DO 循环内嵌 IF 块，WHERE 语句形式上类似于 IF 语句，但是对数组整体进行操作；Fortran 95 的 FORALL 构造相当于隐式循环结合隐式 IF 语句，是对数组元素进行操作，FORALL 构造允许嵌套；一维矢量可以作为数组下标，用来规定一个数组段，该数组段中的元素次序可以是非线性的；Fortran 90 提供了许多有关数组操作的标准函数。
- Fortran 90 有三种形式的数组参数：固定形状数组、假定形状数组和假定大小数组。固定形状数组包括可调大小数组；假定形状数组和实参数组的形状保持一致，若外部例程含有假定形状数组参数，须在调用程序中建立接口块；假定大小数组和实参数组的元素个数保持相同。
- 动态数组的使用要经历声明、分配和释放三个步骤。声明时，动态数组须是延迟形状数组，并具有 ALLOCATABLE 属性；分配和释放分别使用 ALLOCATE

和 DEALLOCATE 语句。若要增加动态数组的大小，须将原来的动态数组释放掉，再重新指定大小。

- Fortran 90 的函数返回值，既可以是标量，也可以是矢量——数组。若返回数组的函数是外部函数，须在调用程序中建立接口块。

## 第六章 派生类型

在前几章的实例程序中，处理的数据都是 Fortran 90 固有数据类型。在有些场合下，直接处理集合数据会给编程带来很大的便利。数组集合的元素要求类型都是相同的，Fortran 90 允许定义不同类型元素的集合，称之为用户定义类型或派生类型。

### 第一节 派生类型的定义

Fortran 90 允许由固有数据类型或其他派生类型创建新的派生类型。比如要维护学生记录，学生的基本信息可能包括姓名、学号、性别、出生年月、地址、电话号码、成绩等，据此创建的学生数据类型为：

---

```

TYPE Student
  CHARACTER (20) Name           ! 姓名
  CHARACTER (10) No             ! 学号
  LOGICAL Female                ! 性别
  INTEGER BirthDate             ! 出生年月
  CHARACTER (20), DIMENSION(4) :: Address ! 地址
  CHARACTER (10) Telephone      ! 电话号码
  REAL, DIMENSION(20)           :: Marks   ! 各科成绩
END TYPE

```

---

构造派生类型的形式为：

```

TYPE [ [, access] :: ] name
    component - definition
END TYPE [name]

```

其中，access 代表 PRIVATE 或 PUBLIC 关键字，在模块中定义数据类型时使用，缺省为 PUBLIC。若规定为 PRIVATE，则该数据类型只能在模块内使用；name 指派生类型名称，类似于固有数据类型的整型 INTEGER、实型 REAL，类型名称在模块范围内须是惟一的；component - definition 代表派生类型成员声明，在第一个成员声明前还可出现 PRIVATE 或 SEQUENCE 关键字，PRIVATE 规定派生类型的成



```

TYPE(Student) :: stu1 = Student( "Smith, JR", 49 ), stu2
PRINT*, '输入 stu2 的成员: '
READ*, stu2
PRINT*, stu1, stu2, Student( "Bloggs", 50 )
END PROGRAM

```

例 6-1 中，在模块中定义派生类型，在主程序中使用该派生类型，并在声明派生类型变量 `stu1` 时，用派生类型构造子对其初始化。派生类型构造子的形式为：

`d - name (expr - list)`

`d - name` 指派生类型名；`expr - list` 为规定派生类型成员值的表达式列表，表达式的次序、类型、种类数必须和定义派生类型成员时的一致。

派生类型构造子可以出现在初始化中，也可出现在赋值、输入/输出语句中。

Fortran 95 允许在定义派生类型时，直接给出成员的默认值。如：

```

TYPE Student
  CHARACTER (20) NAME
  REAL :: Mark = 60
END TYPE

```

其中，给出了成员 `Mark` 的默认值 60。

须注意的是：初始化时，成员的默认值将被构造子的表达式列表所覆盖。比如：

```
TYPE(Student) :: stu1 = Student( "Smith, JR", 49 )
```

这里，49 覆盖了成员 `Mark` 的默认值 60。

需要说明的是：如果在模块中定义派生类型时将成员声明为 `PRIVATE`，就不能使用构造子进行初始化。如例 6-2 所示。

[例 6-2] 派生类型的私有构造。

```

MODULE Stu_Type
  IMPLICIT NONE
  TYPE Student           !定义 Student 数据类型
    PRIVATE              !将成员的访问权限控制在模块内
    CHARACTER (20) Name
    REAL Mark
  END TYPE

  CONTAINS

```



```

SUBROUTINE Student_(this,n,m)  !相当于 C++中的构造函数
    TYPE(Student),INTENT(out)::this
    CHARACTER(*),INTENT(in)::n
    REAL,OPTIONAL,INTENT(in)::m
    this%Name=n
    IF (PRESENT(m)) THEN
        this%Mark=m
    ELSE
        this%Mark=60
    END IF
END SUBROUTINE

SUBROUTINE Print_Student(this)
    TYPE(Student),INTENT(in)::this
    PRINT*,this
END SUBROUTINE

END MODULE

PROGRAM Main
    USE Stu_Type
    IMPLICIT NONE
    TYPE(Student) :: stu
    CALL Student_(stu, "Smith, JR")
    CALL Print_Student(stu)
END PROGRAM

```

例 6-2 中，由于将派生类型的成员声明为私有的，不能在模块外部对其初始化。所以，程序中设计了一个类似于 C++构造函数的公有例程，以便对其初始化；且使用了可选参数设计构造例程，这样，在初始化时可以灵活决定是否传入可选参数，例如：

```

CALL Student_(stu, "Smith, JR")           !省略可选参数
CALL Student_(stu, "Smith, JR",70.0)      !传入可选参数

```

如果派生类型的成员是公有的，可以通过派生类型变量名直接输出其成员(见例 6-1)；反之，需要设计专门的输出例程(见例 6-2)。

### 第三节 操作符重载

假如有某一派生类型的变量  $a$  和  $b$ ，要执行加法运算，我们自然希望使用“+”运算符，写出表达式“ $a + b$ ”，但编译时会出错，因为编译器不知道该如何完成这个加法操作(Fortran 90 预定义运算符的运算对象只能是固有数据类型)。这时候需要我们自己编写程序说明“+”在作用于该派生类型的变量时，该实现什么样的功能，这就是运算符或操作符重载。

操作符重载是对已有的操作符赋予多重含义，使同一操作符作用于不同类型的数据时产生不同的行为。Fortran 90 的操作符重载，分为赋值操作符重载和其他操作符重载。

#### 一、赋值操作符重载

作为简单的演示，我们来重新定义前面的 Student 派生类型，使得可以实施两种简洁的赋值操作：

一是将姓名字符串直接赋给派生类型变量，如：

stu = "Smith, JR"      !stu 为 Student 派生类型变量

二是从派生类型变量中提取姓名，即将派生类型变量直接赋给姓名字符变量。

如：

StuName = stu      !StuName 为字符类型变量

这里，需要重新定义赋值操作符，使之可以处理字符类型和派生类型组成的混合类型。第一种情况下，需要编写带有两个虚参的子程序例程(Student\_From\_Name)，两个虚参分别为派生类型和字符类型，其顺序必须和赋值表达式中出现的顺序相同。在该例程中，需要显式地将字符参数赋给派生类型的姓名成员；第二种情况是第一种情况的反操作，即子程序例程(Name\_From\_Student)的两个虚参分别为字符类型和派生类型，且将派生类型的姓名成员赋给字符参数。如例 6-3 所示。

**[例 6-3] 赋值操作符重载。**

---

```
MODULE StudentMod
```

```
  IMPLICIT NONE
```

```
  TYPE Student_Type
```

```
    CHARACTER (20) NAME
```

```
    REAL Mark
```

```

END TYPE

INTERFACE ASSIGNMENT(=)      !接口块须以 ASSIGNMENT 命名
    MODULE PROCEDURE Name_From_Student, Student_From_Name
END INTERFACE

CONTAINS

    SUBROUTINE Name_From_Student( String, Student )
        CHARACTER(*),INTENT(OUT)    :: String
        TYPE(Student_Type),INTENT(IN)  :: Student
        String = Student % Name
    END SUBROUTINE

    SUBROUTINE Student_From_Name( Student, String )
        CHARACTER(*),INTENT(IN)      :: String
        TYPE(Student_Type),INTENT(OUT) :: Student
        Student % Name = String
    END SUBROUTINE

END MODULE

PROGRAM Main
    USE StudentMod
    IMPLICIT NONE
    TYPE (Student_Type) :: Student = Student_Type( "Bloggs", 50 )
    CHARACTER (20) StuName

    Student = "Smith, JR"    !
    StuName = Student        !

    PRINT*, Student, StuName

END PROGRAM

```

这里，包含重载例程的接口块必须以 ASSIGNMENT 关键字命名。

赋值操作符的重载用子程序例程来实现；其他操作符的重载则用函数例程来实现。

## 二、其他操作符重载

考虑下面的实例(例 6-4): 在模块 IntegerSets 中定义了一个派生类型 SET(类似于 Pascal 语言中的集合类型), 该类型的数据由函数 BuildSet 产生; 定义了一个集合关系操作符(.IN.), 如果整数 I 是集合 S1 中的成员, 那么表达式 I.IN.S1 返回逻辑真; 操作符(\*)被重载, 以执行交集操作, 例如, S1\*S2 返回集合 S1 和 S2 中共有的数据, 假如操作数为固有数据类型, 操作符(\*)仍执行乘法运算。

[例 6-4] 其他操作符重载。

---

```

MODULE IntegerSets
  IMPLICIT NONE
  INTEGER, PARAMETER :: MaxCard = 100

  TYPE SET
    PRIVATE                                !规定成员在模块外不可访问
    INTEGER Cardinality
    INTEGER, DIMENSION( MaxCard ) :: Members
  END TYPE SET

  INTERFACE OPERATOR (.IN.)!接口块以 OPERATOR 命名, 新定义操作符
    MODULE PROCEDURE MemberOf
  END INTERFACE

  INTERFACE OPERATOR (*) !接口块以 OPERATOR 命名, 重载操作符
    MODULE PROCEDURE Intersect
  END INTERFACE

  CONTAINS
    FUNCTION BuildSet( V ) !函数返回值为派生类型
      TYPE(SET) BuildSet
      INTEGER V(:)          !延迟形状数组参数
      INTEGER J
      BuildSet % Cardinality = 0
      DO J = 1, SIZE( V )

```

```

    IF (.NOT.(V(J) .IN. BuildSet)) THEN
        IF (BuildSet % Cardinality < MaxCard) THEN
            BuildSet % Cardinality = BuildSet %
                & Cardinality + 1
            BuildSet % Members( BuildSet %
                & Cardinality ) = V(J)
        ELSE
            PRINT*, 'Maximum set size exceeded -
                & adjust MaxCard'
            STOP
        END IF
    END IF
END DO
END FUNCTION BuildSet

FUNCTION Card( S )           !函数返回集合中元素的最大序号
    INTEGER Card
    TYPE (SET) S
    Card = S % Cardinality
END FUNCTION Card

FUNCTION Intersect( S1, S2 ) !函数返回交集派生类型
    TYPE(SET), INTENT(IN) :: S1, S2
    TYPE(SET) Intersect
    INTEGER I
    Intersect % Cardinality = 0

    DO I = 1, S1 % Cardinality
        IF (S1 % Members(I) .IN. S2) THEN
            Intersect % Cardinality = Intersect % &
                Cardinality + 1
            Intersect % Members(Intersect % &
                Cardinality) = S1 % Members(I)
        END IF
    END DO
END FUNCTION Intersect

```

```

        END DO
    END FUNCTION Intersect

    FUNCTION MemberOf( X, S )      !函数返回逻辑类型
        INTEGER, INTENT(IN) :: X
        TYPE(SET), INTENT(IN) :: S
        LOGICAL MemberOf
        MemberOf = ANY( S % Members(1 : S % Cardinality) == X)
    END FUNCTION MemberOf

    SUBROUTINE PrtSet( S )          !派生类型数据的输出例程
        TYPE (SET) S
        INTEGER I

        PRINT '(20I4)', (S % Members(I), I = 1, S % Cardinality)
    END SUBROUTINE PrtSet
END MODULE

PROGRAM Main
    USE IntegerSets
    IMPLICIT NONE
    TYPE (SET) S1, S2, S3

    S1 = BuildSet( (/ 1, 2, 3, 4, 5 /) )
    S2 = BuildSet( (/ 2, 4, 6, 8 /) )
    S3 = S1 * S2

    WRITE (*, "(S1      ', I3, ' ELEMENTS: ')", &
        ADVANCE = 'NO') Card( S1 )
    CALL PrtSet( S1 )

    WRITE (*, "(S2      ', I3, ' ELEMENTS: ')", &
        ADVANCE = 'NO') Card( S2 )
    CALL PrtSet( S2 )

```

```

WRITE (*, "('S1*S2 ', I3, ' ELEMENTS: ')", &
      ADVANCE = 'NO') Card( S3 )
CALL PrtSet( S3 )
END PROGRAM

```

结果输出:

```

S1      5 ELEMENTS:    1   2   3   4   5
S2      4 ELEMENTS:    2   4   6   8
S1*S2   2 ELEMENTS:    2   4

```

其中:

(1)派生类型的成员被声明为私有的,以将其可访问性限定在模块内,这样,改变派生类型的定义,不会影响使用它的外部程序(外部程序通过公有例程使用派生类型);

(2)操作符(一元或二元)重载,需要使用函数例程规定操作符如何作用于操作数,并通过以关键字 OPERATOR 命名的接口块,将操作符(\*)与函数例程关联起来:

```

INTERFACE OPERATOR (*)
MODULE PROCEDURE Intersect
END INTERFACE

```

这样,我们就可以用表达式运算  $S3 = S1 * S2$ , 代替函数调用  $S3 = \text{Intersect}(S1, S2)$ , 使得语义更清楚,更符合人们的使用习惯。

(3)重载的操作符,可以是固有操作符(如: \*),也可以是自定义的操作符(如: .IN.). 固有操作符不改变固有类型表达式的运算规则;自定义的操作符须是“.字符序列.”的形式,字符序列至多由 31 个字符组成。

(4)函数例程的返回值也可以是派生类型,如:例 6-4 中的 BuildSet、Intersect 函数。

## 第四节 数据库管理应用

目前,用数据库来管理各种数据已十分普遍。通常,数据库由数据表组成,数据表由行或记录组成,而行又由列或字段组成。Fortran 90 的派生类型数据结构恰巧表达了数据库中的记录,派生类型的成员则表达了记录中的字段。所以, Fortran 90 的派生类型可以直接应用于数据库的记录操作。下面将通过实例(例 6-5)演示数据库记录的追加、更新及显示操作,当中的学生信息数据库用直接访问文件(direct access

files)模拟。直接访问文件的每行记录是定长的,访问时可以在文件中快速定位记录,因而拥有较高的执行速度;但由于采取二进制格式存储,一般的编辑器无法查看。

程序的设计思路:将数据类型定义和数据库的记录操作相关例程统一置于模块中,以便外部程序使用;在主程序中打开/关闭文件,并用 DO WHILE 循环设计一命令菜单,分别用来执行记录的追加、更新、显示及退出操作。

为简便起见,学生记录取姓名和成绩两个字段,据此定义的派生类型为:

```
TYPE StudentRecord
    CHARACTER (NameLen) Name
    INTEGER           Mark
END TYPE StudentRecord
```

### 一、追加记录

算法:先从头到尾遍历文件中的记录,找出文件中的最大记录号(Fortran 90 无法直接定位某一条记录),最大记录号加一即为要追加的记录号;然后循环地从键盘读数据、往文件中写数据,每次读、写完毕将当前记录号加一,以非字母、非空白字符结束读、写。

下列为遍历记录代码段:

---

```
DO WHILE (EOF == 0)
    READ( 1, REC = RecNo+1, IOSTAT = EOF )    !置空变量列表
    IF (EOF == 0) THEN                          !遇文件尾, EOF = -1
        RecNo = RecNo + 1                      !记录号递增
    END IF
END DO
```

---

下列为读、写记录代码段:

---

```
this = StudentRecord( "a", 0 )                !以满足下列循环条件
DO WHILE ((VERIFY( this % Name, NameChars ) == 0))
    PRINT*, "Name (any non-letter/non-blank to end): "
    READ*, this % Name                          !从键盘读数据
    IF (VERIFY( this % Name, NameChars ) == 0) THEN
        PRINT*, "Mark: "
        READ*, this % Mark                      !从键盘读数据
```

---



```

WRITE (1, REC = RecNo) this
RecNo = RecNo + 1           !记录号递增
END IF
END DO

```

这里假定姓名成员由字母和空格组成，并利用 VERIFY 函数验证其合法性，当中的 NameChars 实参是由字母和空格组成的字符串常量。

值得注意的是：一般的键盘输入以空格分割，输入不含空格的字符串时无需加字符界定符。若姓名中含有空格，则输入时必须添加字符界定符。

## 二、更新记录

算法：从头到尾遍历文件中的记录，将其姓名字段与要更新的成绩对应的姓名相比较，若匹配，则从键盘读入要更新的成绩，重写该记录。

用是否到达文件尾和是否找到匹配的记录来控制循环。若找到匹配的记录，则重写该记录，并将“发现匹配记录的逻辑变量”设为.TRUE.；若没有找到匹配的记录，则将记录号加一，比较下一条记录。若循环结束，“发现匹配记录的逻辑变量”仍为.FALSE.，则表明文件中没有与姓名匹配的记录。

下列为更新记录的主要代码段：

```

DO WHILE (EOF == 0 .AND. .NOT. Found)
  READ (1, IOSTAT = EOF, REC = RecNo) this    !从文件中读记录
  IF (EOF == 0) THEN
    Copy = this % Name                        !姓名字段
    CALL StripBlanks( Copy )                  !去掉姓名两边空格
    IF (Item == Copy) THEN                     !Item
      Found = .true.
      PRINT*, 'Found at recno', RecNo, ' Enter new mark:'
      READ*, this % Mark                      !从键盘读入成绩
      WRITE (1, REC = RecNo) this             !重写当前记录
    ELSE
      RecNo = RecNo + 1
    END IF
  END IF
END IF
END DO

```

---

```

IF (.NOT. Found) THEN
    PRINT*, Item, ' not found'
END IF

```

---

其中, StripBlanks 为自定义的子程序例程, 其功能是将传入的字符串两边的空格去掉。

### 三、显示记录

算法: 从头至尾对文件中的记录进行遍历。若文件指针没到文件尾, 显示当前记录, 并将当前记录号加一。下列为显示记录的子程序例程:

---

```

SUBROUTINE DisplayRecords
    RecNo = 1
    EOF = 0
    DO WHILE (EOF == 0)
        READ (1, REC = RecNo, IOSTAT = EOF) this
        IF (EOF == 0) THEN
            PRINT "(A20, I3)", this
        END IF
        RecNo = RecNo + 1
    END DO
END SUBROUTINE DisplayRecords

```

---

例 6-5 为上述数据库操作的完整程序(包括模块和主程序)。

[例 6-5] 数据库管理应用。

---

```

MODULE Student_Records
    IMPLICIT NONE
    INTEGER, PARAMETER :: NameLen = 20
    CHARACTER (*), PARAMETER :: NameChars = &
        " abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"

    TYPE StudentRecord
        CHARACTER (NameLen) Name

```

```

        INTEGER                                Mark
END TYPE StudentRecord

TYPE (StudentRecord),PRIVATE  :: this
INTEGER,PRIVATE                :: EOF, RecNo

CONTAINS

    SUBROUTINE AddRecords                                !追加记录子程序
        RecNo = 0
        EOF = 0

        DO WHILE (EOF == 0)
            READ( 1, REC = RecNo+1, IOSTAT = EOF )
            IF (EOF == 0) THEN
                RecNo = RecNo + 1
            END IF
        END DO

        RecNo = RecNo + 1
        this = StudentRecord( "a", 0 )
        DO WHILE ((VERIFY( this % Name, NameChars ) == 0))
            PRINT*, "Name (any non-letter/non-blank to end): "
            READ*, this % Name
            IF (VERIFY( this % Name, NameChars ) == 0) THEN
                PRINT*, "Mark: "
                READ*,this % Mark
                WRITE (1, REC = RecNo) this
                RecNo = RecNo + 1
            END IF
        END DO
    END SUBROUTINE AddRecords

    SUBROUTINE UpDate                                !更新记录子程序
        CHARACTER (NameLen) Item, Copy

```

```
LOGICAL Found
Found = .false.
EOF = 0
PRINT*, "Update who?"
READ "(A20)", Item
CALL StripBlanks( Item )
RecNo = 1
DO WHILE (EOF == 0 .AND. .NOT. Found)
  READ (1, IOSTAT = EOF, REC = RecNo) this
  IF (EOF == 0) THEN
    Copy = this % Name
    CALL StripBlanks( Copy )
    IF (Item == Copy) THEN
      Found = .true.
      PRINT*, 'Found at recno', RecNo,&
        'Enter new mark:'
      READ*,this % Mark
      WRITE (1, REC = RecNo) this
    ELSE
      RecNo = RecNo + 1
    END IF
  END IF
END DO
IF (.NOT. Found) THEN
  PRINT*, Item, ' not found'
END IF
END SUBROUTINE UpDate

SUBROUTINE DisplayRecords      !显示记录子程序
  RecNo = 1
  EOF = 0
  DO WHILE (EOF == 0)
    READ (1, REC = RecNo, IOSTAT = EOF) this
    IF (EOF == 0) THEN
```

```

        PRINT "(A20, I3)", this
    END IF
    RecNo = RecNo + 1
END DO
END SUBROUTINE DisplayRecords

SUBROUTINE StripBlanks( Str )
    CHARACTER (*) Str
    INTEGER I
    I = 1
    DO WHILE (I < LEN_TRIM( Str ))
        IF (Str(I:I) == " ") THEN
            Str(I:) = Str(I+1:)
        ELSE
            I = I + 1
        END IF
    END DO
END SUBROUTINE StripBlanks
END MODULE

```

#### PROGRAM Main

```

    USE Student_Records
    IMPLICIT NONE
    TYPE (StudentRecord)    Student
    INTEGER                  RecLen
    LOGICAL                  IsThere
    CHARACTER (NameLen)     FileName
    CHARACTER                Ans
    CHARACTER (7)           FileStatus

```

```

    INQUIRE (IOLENGTH = RecLen) Student      !确定记录长度
    WRITE (*, "(File name: ')", ADVANCE = "NO")
    READ*, FileName
    INQUIRE (FILE = FileName, EXIST = IsThere) !确定文件存在

```

```
IF (IsThere) THEN
    WRITE (*, "(File already exists. Erase and &
        recreate (Y/N)? )", ADVANCE = "NO")
    READ*, Ans
    IF (Ans == "Y") THEN
        FileStatus = "REPLACE"
    ELSE
        FileStatus = "OLD"
    END IF
ELSE
    FileStatus = "NEW"
END IF

OPEN (UNIT = 1, FILE = FileName, STATUS = FileStatus, &
    ACCESS = 'DIRECT', RECL = RecLen) !以直接访问方式打开文件
Ans = " "
DO WHILE (Ans /= "Q")                !构建一命令菜单
    PRINT*
    PRINT*, "A: Add new records"
    PRINT*, "D: Display all records"
    PRINT*, "Q: Quit"
    PRINT*, "U: Update existing records"
    PRINT*
    WRITE (*, "(Enter option and press ENTER: )", &
        ADVANCE = "NO")
    READ*, Ans
    SELECT CASE (Ans)
        CASE ("A", "a")
            CALL AddRecords
        CASE ("D", "d")
            CALL DisplayRecords
        CASE ("U", "u")
            CALL UpDate
```

```
END SELECT
END DO
CLOSE (1)                                !关闭文件
END PROGRAM
```

### 小 结

- 派生类型类似于 C 语言中的结构体,可以用来构造不同类型(包括数组和其他派生类型)元素的集合。
- 定义派生类型使用 TYPE 结构块,声明派生类型的变量(包括数组)须在类型名前加 TYPE 关键字,引用派生类型的成员须使用成员操作符%,同一派生类型的两个变量可以相互赋值。
- 派生类型可以通过构造子进行初始化,构造子还可出现在赋值、输入/输出语句中。Fortran 95 允许在定义派生类型时,直接给出成员的默认值;但在初始化时,成员的默认值将被构造子的值所覆盖。在模块中定义派生类型时,只有将成员的访问属性规定为 PUBLIC(缺省),才能在外部程序中使用构造子。
- 通常,将派生类型及其操作例程的定义置于模块中,以便外部程序使用。若在模块中将派生类型规定为 PRIVATE,则派生类型及其成员只能在定义的模块内访问;若只将派生类型的成员规定为 PRIVATE,则派生类型在外部程序中仍可访问,但其成员变得不可访问,这就是所谓的“信息隐藏”。
- Fortran 90 的固有运算(如:赋值、加、减、乘、除等)是针对固有数据类型的,针对派生类型的运算通常要重新进行定义,即对操作符进行重载。赋值操作符重载,须将具体的操作例程(子程序)置于以 ASSIGNMENT 命名的接口块内;其他操作符重载,须将具体的操作例程(函数)置于以 OPERATOR 命名的接口块内。其他操作符可以是固有操作符,也可以是新定义的操作符。新定义操作符须是“.字符序列.”的形式,其中字符序列至多由 31 个字符组成。
- 派生类型可以作为函数例程的返回值。
- 派生类型可以直接表达数据库中的记录,用于数据库的管理操作,如:追加、更新、显示记录等。

## 第七章 指 针

Fortran 90 也提供了指针, 不过 Fortran 90 指针类似于 C++ 中的引用, 但和 C++ 中的指针概念不同。本章着重论述 Fortran 90 指针的基本概念、指针数组、指针参数、指针型函数以及单链表指针应用。

### 第一节 指针的基本概念

在介绍指针基本概念之前, 先让我们来看一个实例例 7-1。

[例 7-1] 指针的声明、赋值、动态分配及运算。

---

```
PROGRAM Pointers
  IMPLICIT NONE
  INTEGER, POINTER :: p, q
  INTEGER, TARGET :: n = 5
  INTEGER :: m

  p => n
  q => p

  ALLOCATE( p )
  p = 4

  m = p + q + n
  PRINT *, "p = ", p, ", q = ", q, ", m = ", m

  DEALLOCATE( p )
END PROGRAM Pointers
```

---

结合例 7-1, 给出 Fortran 90 与指针相关的一些基本概念:

(1) 指针。Fortran 90 的指针变量, 是具有指针(POINTER)属性的变量。程序中,



p 和 q 声明为整型指针变量, 可以指向整型存储单元。

(2)目标。为便于编译器优化, Fortran 90 规定: 能被指针引用的变量必须具有目标(TARGET)属性。n 是具有目标属性的整型变量, 因此它可以作为整型指针 p 或 q 引用的变量。

(3)指针赋值。指针赋值操作符为“=>”。 $p => n$ , 是指针赋值语句, 它将指针 p 与目标 n 相关联(称 p 指向 n), p 作为 n 的别名引用的是同一存储单元的内容;  $q => p$ , 同样是指针赋值, 但代表的意义不同: 赋值号两边均为指针变量, 在这种情况下, 指针 q 间接地与指针 p 的目标 n 相关联, 因此一个目标同时可以有多个关联指针。 $p = 4$ , 是将 p 关联的目标赋以整型数 4(C++语句为:  $*p = 4$ )。

(4)指针关联。Fortran 90 指针有三种关联状态: 关联(associated)、非关联(disassociated)和未定义(undefined)。声明时, 为未定义; 指针赋值或动态分配(ALLOCATE)时, 为关联; 运用 NULLIFY 函数(Fortran 95 为 NULL)或动态释放(DEALLOCATE)时, 为非关联。ASSOCIATED 函数用来检测指针是否关联。

(5)动态分配。ALLOCATE 语句用来为目标动态分配存储单元, 并将指针与动态分配的存储单元目标相关联。因此, ALLOCATE(p)为一整型目标分配空间, 并与指针 p 相关联。此时, 新目标的内容还未定义, 而 p 以前的目标 n 不受影响。 $p = 4$ , 将新目标的内容设为整型数 4。DEALLOCATE 语句用来撤消指针关联, 并释放在 ALLOCATE 语句动态分配的存储单元。

(6)指针表达式。Fortran 90 的指针实际上是对目标变量的引用, 它不具有地址概念。在表达式  $p + q + n$  中, p 和 q 的目标值分别为 4 和 5,  $n = 5$ , 所以表达式结果为 14。

## 第二节 指针数组

Fortran 90 指针毕竟只是一种变量属性, 而不是一种真实的数据类型, 因此不能直接创建指针数组。作为一种替代方式, 我们可以定义包含指针的派生类型, 然后创建该派生类型的数组, 来间接实现指针数组的功能。

在声明具有指针属性的数组时, 须采取延迟形状数组形式。如:

REAL, DIMENSION(100), POINTER :: X !非法

REAL, DIMENSION(:), POINTER :: X !合法

下面利用指针数组, 来解决交错数组(如: 下三角矩阵)的动态分配问题。如例 7-2 所示。

[例 7-2] 指针数组在下三角矩阵中的应用。

---

```

PROGRAM LowerTriangular
  IMPLICIT NONE

  TYPE ROW
    REAL, DIMENSION(:), POINTER :: R    !声明具有指针属性的数组
  END TYPE

  INTEGER, PARAMETER :: N = 4
  TYPE (ROW), DIMENSION(N) :: T    !声明派生类型的数组
  INTEGER I

  DO I = 1, N
    ALLOCATE (T(I) % R(1:I))          !为每行分配不同存储单元
    T(I) % R(1:I) = 1                 !为下三角矩阵每行赋值
  END DO

  DO I = 1, N
    PRINT*, T(I) % R(1:I)              !输出下三角矩阵
  END DO

  DO I = 1, N
    DEALLOCATE (T(I) % R)              !释放存储单元
  END DO
END PROGRAM LowerTriangular

```

---

其输出结果为：

---

```

1.000000
1.000000    1.000000
1.000000    1.000000    1.000000
1.000000    1.000000    1.000000    1.000000

```

---

这里，先创建含有指针成员的派生类型，再创建该类型的数组，然后为数组的

指针成员关联的目标动态分配内存，结果节省了近一半的存储单元。

### 第三节 指针型函数

有时，需要函数返回一个可变大小的数组，可是 Fortran 90 不允许函数返回值有 ALLOCATABLE 属性，但允许有 POINTER 属性。因此，可以用指针数组代替动态数组作为函数的返回值。见例 7-3 中的 Vector 函数所示。

[例 7-3] 函数的返回值为指针数组。

---

```
PROGRAM Sort
```

```
!对任一大小的整型数组按升序排列
```

```
IMPLICIT NONE
```

```
INTEGER, DIMENSION(10) :: X = (/ 3,6,9,-1,56,4,6,0,0,8 /)
```

```
INTEGER, DIMENSION(:),POINTER :: P      !声明与X同维的指针数组
```

```
P=>Vector( X )
```

```
!指针指向函数目标
```

```
PRINT*, P
```

```
!间接释放为函数分配的空间
```

```
DEALLOCATE( P )
```

```
CONTAINS
```

```
FUNCTION Vector( A )
```

```
!函数返回值为指针数组
```

```
INTEGER, DIMENSION(:), POINTER :: Vector
```

```
INTEGER, DIMENSION(:) :: A      !延迟形状数组参数
```

```
INTEGER I, J, T
```

```
ALLOCATE( Vector(SIZE(A)) )      !为函数动态分配空间
```

```
Vector = A                      !相同形状的数组间赋值
```

```
DO I = 1, SIZE(A)-1
```

```
    DO J = I+1, SIZE(A)
```

```
        IF (Vector(I) > Vector(J)) THEN
```

```
            T = Vector(J)
```

```
            Vector(J) = Vector(I)
```

```
        Vector(I) = T
      END IF
    END DO
  END DO
END FUNCTION Vector
END PROGRAM Sort
```

其中，函数 Vector 的返回值规定为具有指针属性的延迟形状数组，虚参 A 接收任意大小的一维数组实参，函数 Vector 将排好序的数组返回调用程序。

调用程序(主程序)将相同维的指针数组指向函数目标，使用完毕，释放指针数组，从而将函数内动态分配的内存释放掉。也就是说，通过释放指向函数目标的指针数组，来间接释放为函数动态分配的内存。

由于函数返回指针，且虚参为延迟形状数组，若函数 Vector 作为外部例程使用，则须在调用程序中建立其接口块。

## 第四节 指针参数

指针作参数大致分两种情况：一是虚参具有目标属性，实参可以是指针，也可以是普通实参；二是虚参具有指针属性，实参必须是指针。

下面用子程序例程实现例 7-3 中的函数例程 Vector，以演示不同的指针参数。见例 7-4、例 7-5 和例 7-6。

**[例 7-4]** 虚参具有目标属性，实参为数组。

```
PROGRAM Sort
  IMPLICIT NONE
  INTEGER, DIMENSION(10) :: X = (/ 3,6,9,-1,56,4,6,0,0,8 /)

  CALL Vector( X )
  PRINT*,X

CONTAINS
  SUBROUTINE Vector( A )
    INTEGER, DIMENSION(:),TARGET :: A
    INTEGER I, J, T
```

---

```

      DO I = 1, SIZE(A)-1
        DO J = I+1, SIZE(A)
          IF (A(I) > A(J)) THEN
            T = A(J)
            A(J) = A(I)
            A(I) = T
          END IF
        END DO
      END DO
    END SUBROUTINE Vector
  END PROGRAM Sort

```

---

例 7-4 的虚参具有目标属性，实参为普通数组。由于参数传递为引用传递，所以例程内虚参的改变会反映在实参上，调用后数组的顺序就变成有序的。

[例 7-5] 虚参具有目标属性，实参为指针。

---

```

PROGRAM Sort
  IMPLICIT NONE
  INTEGER,DIMENSION(10),TARGET :: X = (/ 3,6,9,-1,56,4,6,0,0,8 /)
  INTEGER, DIMENSION(:), POINTER :: P

  P=>X
  CALL Vector( P )
  PRINT*, P
  (略：子程序例程 Vector 同例 7-4)
END PROGRAM Sort

```

---

例 7-5 中，虚参具有目标属性，实参为指针。指针 P 先指向目标 X，再作为实参传入子程序。在这里，指针 P 作为目标 X 的引用或别名，实质传入子程序的仍是 X，只不过要在声明变量 X 中增加 TARGET 属性，以便赋值给指针。

[例 7-6] 实参、虚参皆为指针。

---

```

PROGRAM Sort
  IMPLICIT NONE

```

```
INTEGER,DIMENSION(10),TARGET  :: X = (/ 3,6,9,-1,56,4,6,0,0,8 /)
INTEGER, DIMENSION(:), POINTER  :: P

P=>X
CALL Vector( P )
PRINT*, P

CONTAINS
  SUBROUTINE Vector( A )
    INTEGER, DIMENSION(:), POINTER :: A
    INTEGER I, J, T
    DO I = 1, SIZE(A)-1
      DO J = I+1, SIZE(A)
        IF (A(I) > A(J)) THEN
          T = A(J)
          A(J) = A(I)
          A(I) = T
        END IF
      END DO
    END DO
  END SUBROUTINE Vector
END PROGRAM Sort
```

例 7-6 的实参、虚参皆为指针。调用前，实参指针 P 指向目标 X，即和目标 X 相关联；调用时，实参 P 传入子程序，使得虚参指针 A 间接地与目标 X 相关联，即实参、虚参指针关联的是同一目标。

例 7-4、例 7-5 和例 7-6 的虚参 A 为延迟形状数组，且具有目标或指针属性，若子程序 Vector 作为外部例程使用，则须在调用程序中建立其接口块。

## 第五节 单链表应用

指针的一个重要应用是解决像线性表、树等数据结构的动态存储问题。以线性表为例，若采用数组顺序存储，需要为数组声明足够的长度，才能满足多种情况的要求；在插入和删除操作时，还需要移动大量的元素，导致存储空间的浪费和执行

效率的低下。

若采用链式存储,可以根据实际情况为链表动态分配存储空间;在插入和删除操作时,也无需移动元素,因此链表可以有效地节省存储空间,执行插入、删除等操作时又有较高的执行效率。

### 一、链表结构

利用 Fortran 90 提供的派生类型和指针工具,可以定义单链表结点(或元素)的逻辑结构,如:

```
TYPE NODE
  INTEGER val
  TYPE(NODE),POINTER :: next
END TYPE
```

从构成上看,单链表结点的数据类型由数据域和指针域两部分组成。通常,数据域包含一个或多个数据项(比如:学生记录的学号、姓名、成绩等),为了操作简便,我们以单一的主键(惟一标识一个结点的数据项)代表数据域;指针域则包含指向结点自身的指针,单链表的指针域有一个指针,指向其后继结点,双链表有两个指针,分别指向其前驱结点和后继结点。在此,我们只探讨单链表。

在存储结构上,数组是按照元素下标依次存放在连续的存储单元中,存储位置的相临关系即代表元素的先后关系;而链表结点的存储单元既可以是连续的,也可以是不连续的,元素(或结点)的存储位置通过链接指针来确定,如图 7-1 所示。

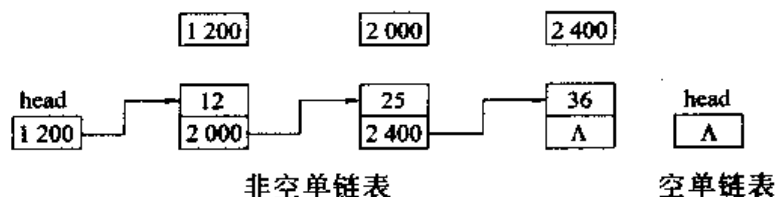


图 7-1 单链表的存储结构示意图

图 7-1 中的非空单链表由三个结点组成,其存储单元的地址编码分别为 1 200、2 000 和 2 400,数据项的值分别为 12、25 和 36。链表头指针 head 指向首结点存储单元 1 200,首结点的指针域指向第二个结点的存储单元 2 000,第二个结点的指针域又指向第三个结点(尾结点)的存储单元 2 400,尾结点的指针域为空。空单链表不包含结点,其头指针为空。

### 二、单链表基本操作

单链表的基本操作,包括链表的创建、链表结点的插入、删除、释放及链表的

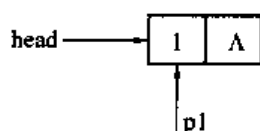
输出。

### (一)单链表的创建及输出

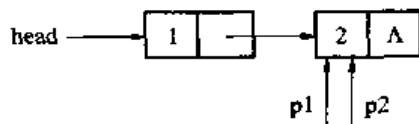
#### 1. 单链表的创建

单链表的创建步骤一般为：

(1)配置首结点 p1，并将头指针 head 指向首结点 p1；



(2)配置下一个结点 p2，并将该结点连接到前一个结点(即将前一个结点的指针域指向该结点)，再移动前一个结点指针 p1，使其指向新结点 p2；



(3)重复步骤(2)，直到所有的结点连接完毕。

创建链表的实现函数为：

---

```

FUNCTION link_create(head, len) !head 结点指针，len 结点个数
  TYPE(NODE),POINTER :: link_create,head,p1,p2
  INTEGER              :: len,i

  PRINT*,"创建链表，输入首结点值==>"
  ALLOCATE( p1 )                !申请首结点
  READ*,P1 % val                 !输入结点值
  NULLIFY(p1 % next)            !首结点指针域置空

  IF (.NOT. ASSOCIATED(head)) THEN
    head => p1                   !头指针指向首结点
  END IF

  DO i=1, len-1
    PRINT*,"创建链表，输入下一个结点值==>"
    ALLOCATE( p2 )                !申请下一个结点
    READ*,P2 % val                 !输入结点值
    NULLIFY(p2 % next)            !新结点指针域置空
  
```



---

```

        p1 % next => p2                !新结点连接到前一个结点
        p1 => p1 % next                !使 p1 指向最后一个结点
    END DO

    link_create => head                !函数返回头指针
END FUNCTION

```

---

## 2. 单链表的输出

算法：从头至尾遍历每一个结点(尾结点的指针域为空)，输出结点值。

实现链表输出的子程序为：

---

```

SUBROUTINE link_print(head)
    TYPE(NODE), POINTER :: head, p    !head 为链表头指针
    p => head                          !指针 p 先指向首结点
    DO WHILE (ASSOCIATED(p))          !非尾结点
        PRINT*, p % val
        p => p % next                  !移动指针，指向下一个结点
    END DO
END SUBROUTINE

```

---

## (二)单链表结点的插入

链表结点插入操作的步骤为：

(1)创建待插入结点 q(值为 x)。如：

```

    ALLOCATE( q )
    q % val = x

```

(2)结点插入。按插入位置的不同，分为空链表、在表头插入和在表中(包括表尾)插入三种情况：

1)空链表

将表头指针指向插入结点，并置空其指针域。如：

```

    head => q
    NULLIFY(q % next)

```

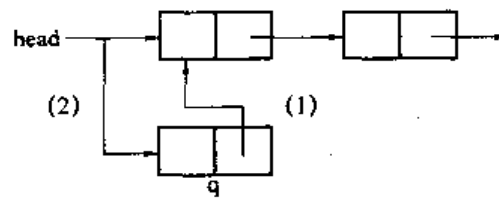
2)在表头插入

先将插入结点的指针域指向原首结点，再将表头指针指向插入结点。如：

```

    q % next => head
    head => q

```

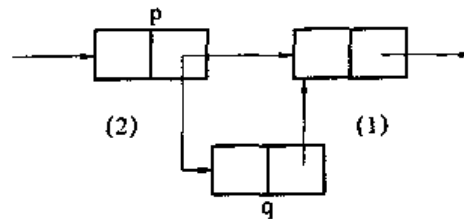


### 3)在表中插入

先找出待插入结点  $q$  的前趋结点  $p$ ，然后将插入结点的指针域指向前趋结点的后继结点，再将前趋结点的指针域指向插入结点。如：

$q \% \text{next} \Rightarrow p \% \text{next}$

$p \% \text{next} \Rightarrow q$



实现链表结点插入的函数为：

```
FUNCTION link_ins(head, i, x)
```

```
  TYPE(NODE),POINTER :: link_ins,head,p,q
```

```
  INTEGER              :: i,x,j
```

```
  ALLOCATE( q )
```

!为插入结点分配内存

```
  q % val = x
```

```
  IF(.NOT. ASSOCIATED(head)) THEN
```

!空链表

```
    head => q
```

```
    NULLIFY(q % next)
```

```
    link_ins => head
```

```
    RETURN
```

```
  ENDIF
```

```
  IF(i==1) THEN
```

!在表头插入结点

```
    q % next => head
```

```
    head => q
```

```
    link_ins => head
```

```
    RETURN
```

ENDIF

p => head

j = 2

DO WHILE (j < i .AND. ASSOCIATED(p)) !寻找第 i - 1 个结点

p => p % next

j = j + 1

END DO

IF(.NOT. ASSOCIATED(p)) THEN !插入的位置在范围之外

PRINT\*, "i out of range !"

link\_ins => head

RETURN

ELSE

q % next => p % next !在表中插入结点

p % next => q

link\_ins => head

END IF

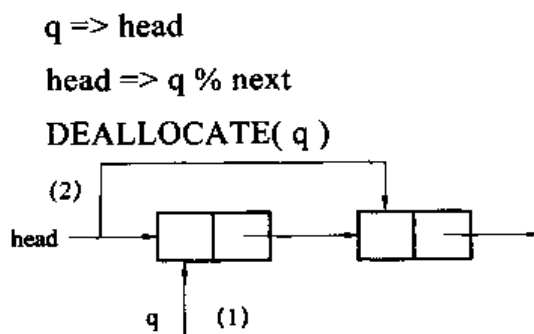
END FUNCTION

### (三)单链表结点的删除

链表结点删除分删除表头结点和表中结点两种情况。

#### 1. 表头结点删除

将临时指针 q 指向表头结点，移动头指针 head 使其指向表头结点的后继结点，释放表头结点。如：



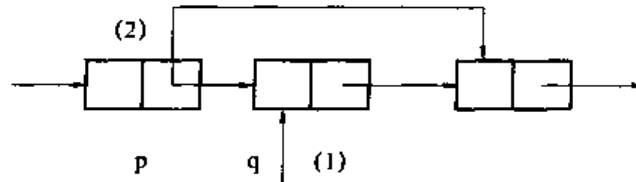
#### 2. 表中结点删除

找出待删除结点的前驱结点 p，将临时指针 q 指向待删除结点，将前驱结点 p 的指针域指向待删除结点的后继结点，释放待删除结点。如：

```

q => p % next
p % next => q % next
DEALLOCATE( q )

```



链表结点删除的实现函数为：

```

FUNCTION link_del(head, i)
  TYPE(NODE),POINTER :: link_del, head, p, q
  INTEGER              :: i, j

  IF(i == 1) THEN                                !删除表头结点
    q => head
    head => q % next
    DEALLOCATE( q )
    link_del => head
    RETURN
  END IF

  p => head
  j = 2

  !寻找待删除结点的前趋结点 p
  DO WHILE(j < i .AND. ASSOCIATED(p))
    p => p % next
    j = j + 1
  END DO

  IF(.NOT. ASSOCIATED(p) .OR. .NOT. ASSOCIATED(p % next)) THEN
    PRINT*, "i out of range !"
    RETURN
  ELSE

```

```

        q => p % next                !删除表中结点
        p % next => q % next
        DEALLOCATE( q )
        link_del => head
    ENDIF
END FUNCTION

```

#### (四)单链表的释放

算法：从头至尾(表尾结点的指针域为空)逐一释放每个结点，在释放过程中不断移动头指针，使其指向待释放结点的后继结点。

链表释放的实现子程序为：

```

SUBROUTINE link_free(head)
    TYPE(NODE), POINTER :: head, p

    DO WHILE( ASSOCIATED(head) )    !逐一释放
        p => head
        head => head % next          !指针下移
        DEALLOCATE( p )              !释放结点
    END DO
END SUBROUTINE

```

在实际编程中，通常将链表结点定义和操作例程统一置于模块中，以方便外部程序的使用。例 7-7 为上述链表操作的完整程序。

[例 7-7] 实参、虚参皆为指针

```

MODULE Link
    IMPLICIT NONE

    TYPE NODE                                !链表的结点定义
        INTEGER val
        TYPE(NODE),POINTER :: next
    END TYPE

```

## CONTAINS

!链表的创建

FUNCTION link\_create(head,len) !head 结点指针, len 结点数

TYPE(NODE),POINTER :: link\_create,head,p1,p2

INTEGER :: len,i

PRINT\*,"创建链表, 请输入结点值==&gt;"

ALLOCATE( p1 ) !申请新结点

READ\*,P1 % val !输入结点值

NULLIFY(p1 % next) !新结点指针域置空

IF (.NOT. ASSOCIATED(head)) THEN

head =&gt; p1 !p1 接入表头

END IF

DO i=1, len -1

PRINT\*,"创建链表, 请输入下一个结点值==&gt;"

ALLOCATE( p2 ) !申请下一个结点

READ\*,P2 % val !输入结点值

NULLIFY(p2 % next) !新结点指针置为空

p1 % next =&gt; p2 !p2 连接到前一个结点

p1 =&gt; p1 % next

END DO

link\_create =&gt; head

END FUNCTION

!链表结点的插入

FUNCTION link\_ins(head, i, x)

TYPE(NODE),POINTER :: link\_ins,head,p,q

INTEGER :: i,x,j

ALLOCATE( q ) !为插入结点分配内存

q % val = x

```

    IF(.NOT. ASSOCIATED(head)) THEN    !空链表
        head => q
        NULLIFY(q % next)
        link_ins => head
        RETURN
    END IF

    IF(i==1) THEN                      !插入在第一个结点位置
        q % next => head
        head => q
        link_ins => head
        RETURN
    END IF

    p => head
    j = 2
    DO WHILE (j < i .AND. ASSOCIATED(p))!寻找 i-1 个结点
        p => p % next
        j = j + 1
    END DO

    IF(.NOT. ASSOCIATED(p)) THEN        !插入的位置在范围之外
        PRINT*, "i out of range !"
        link_ins => head
        RETURN
    ELSE
        q % next => p % next            !结点插入
        p % next => q
        link_ins => head
    END IF
END FUNCTION

!链表结点的删除

```

```

FUNCTION link_del(head, i)
  TYPE(NODE), POINTER :: link_del, head, p, q
  INTEGER               :: i, j

  IF(i == 1) THEN                                !删除头结点
    q => head
    head => q % next
    DEALLOCATE( q )
    link_del => head
    RETURN
  END IF

  p => head
  j = 2
  DO WHILE(j < i .AND. ASSOCIATED(p)) !寻找 i-1 个结点
    p => p % next
    j = j + 1
  END DO

  IF(.NOT. ASSOCIATED(p) .OR. .NOT. &
    ASSOCIATED(p % next)) THEN
    PRINT*, "i out of range !"
    RETURN
  ELSE
    q => p % next
    p % next => q % next
    DEALLOCATE( q )
    link_del => head
  END IF
END FUNCTION

!链表的输出
SUBROUTINE link_print(head)
  TYPE(NODE), POINTER :: head, p

```



```

        p => head
        DO WHILE(ASSOCIATED(p))           !判断是否到表尾
            PRINT*, p % val
            p => p % next
        END DO
    END SUBROUTINE

!链表的释放
SUBROUTINE link_free(head)
    TYPE(NODE), POINTER :: head, p

    DO WHILE( ASSOCIATED(head) )         !逐一释放
        p => head
        head => head % next              !指针下移
        DEALLOCATE( p )
    END DO
END SUBROUTINE
END MODULE

PROGRAM Main
USE Link
IMPLICIT NONE
INTEGER                :: pos,x,len, Ans = 0
TYPE(NODE),POINTER :: head => NULL()!置空头结点, 或创建空链表

DO WHILE (Ans /= 5)
    PRINT*, "1: 创建链表"
    PRINT*, "2: 插入结点(位置从 1 开始)"
    PRINT*, "3: 删除结点(位置从 1 开始)"
    PRINT*, "4: 释放链表"
    PRINT*, "5: 退出"
    WRITE (*, "('Enter option and press ENTER: ')", &
        ADVANCE = "NO")
    READ*, Ans
    SELECT CASE (Ans)

```

```

CASE (1)
  IF( ASSOCIATED(head) ) THEN
    PRINT*,"先释放原有链表，再创建新链表"
    CYCLE
  END IF
  PRINT*,"请输入链表的长度==>"
  READ*,len
  head => link_create(head,len)
  CALL link_print(head)
CASE (2)
  PRINT*,"请输入插入结点位置==>"
  READ*,pos
  PRINT*,"请输入插入的结点值==>"
  READ*,x
  head => link_ins(head,pos,x)
  CALL link_print(head)
CASE (3)
  IF(.NOT.(ASSOCIATED(head))) CYCLE!空链表，跳过
  PRINT*,"请输入删除结点位置==>"
  READ*,pos
  head => link_del(head,pos)
  CALL link_print(head)
CASE (4)
  CALL link_free(head)
END SELECT
END DO

CALL link_free(head)                                !结束前，释放链表
END PROGRAM

```

### 小 结

- Fortran 90 的指针变量，是具有指针(POINTER)属性的变量，可以指向具有目标(TARGET)属性且类型相同的变量。它相当于 C++中的引用，而不代表地址，

对指针的操作实际上是对目标变量的操作。

- 给指针赋值( $\Rightarrow$ ), 将建立指针和目标的关联, 使指针和目标引用同一个存储单元的内容。Fortran 90 指针有三种状态: 关联(associated)、非关联(disassociated)和未定义(undefined)。标准例程 NULLIFY 置空指针, 使之处于非关联状态; Fortran 95 还提供了 NULL 函数, 将指针初始化为空指针。ASSOCIATED 函数用来测试一个指针是否关联, 以及与什么样的目标关联。
- 普通变量是在编译时分配内存, 动态变量则在运行时动态分配内存。Fortran 90 使用 ALLOCATE 语句为指针引用的目标动态分配内存, 并建立起指针与目标的关联; 使用 DEALLOCATE 语句来释放内存。
- Fortran 90 允许延迟形状数组具有指针属性, 即可以声明没有上、下界规定的固定维的指针数组; 要声明固定大小的指针数组, 须借助于派生类型: 通过声明具有指针成员的派生类型的数组, 达到间接声明固定大小指针数组的目的。
- Fortran 90 的函数返回值不能有 ALLOCATABLE 属性, 但允许有 POINTER 属性。借助于指针工具, Fortran 90 函数可以返回任意大小的数组。
- Fortran 90 允许指针作参数, 当虚参具有目标属性时, 实参可以是指针, 也可以是普通实参; 当虚参具有指针属性时, 实参只能是指针。
- 借助于 Fortran 90 提供的派生类型和指针工具, 可以解决像线性表、树等数据结构的动态存储问题, 使有限的内存资源得到高效利用。

## 第八章 模拟 C++ 面向对象程序设计

目前,在大型软件工程开发中,面向对象方法被大家共认为是提高代码重用和可维护性的有效方法。Fortran 90 尽管融入了许多现代语言特征,例如模块、接口块、派生类型、指针等,但没有提供继承和运行时多态语言机制,即不直接支持面向对象编程。2004 年 5 月发布的 Fortran 2003 标准,尽管支持面向对象编程(类似于 C++),但支持 Fortran 2003 标准的编译器还没有开发出来。所以,眼下进行科学与工程计算的面向对象编程有两个选择:一是将其算法转换成 C++ 代码,这会大大增加开发工作量、延长开发周期;二是直接利用 Fortran 90 提供的现代语言特征,来模拟 C++ 面向对象特性,间接实现面向对象编程。

本章将通过和 C++ 的对照实例说明:如何用 Fortran 90 提供的现代语言特征,来模拟 C++ 面向对象的三个主要特性,即封装、继承和运行时多态。

### 第一节 C++ 实现的类层次

尽管 Fortran 90 主要用于科学与工程计算编程,但为了便于说明与理解,我们选择构造一个公司雇员类层次(如图 8-1 所示)。假定某小型公司仅有技术人员和推

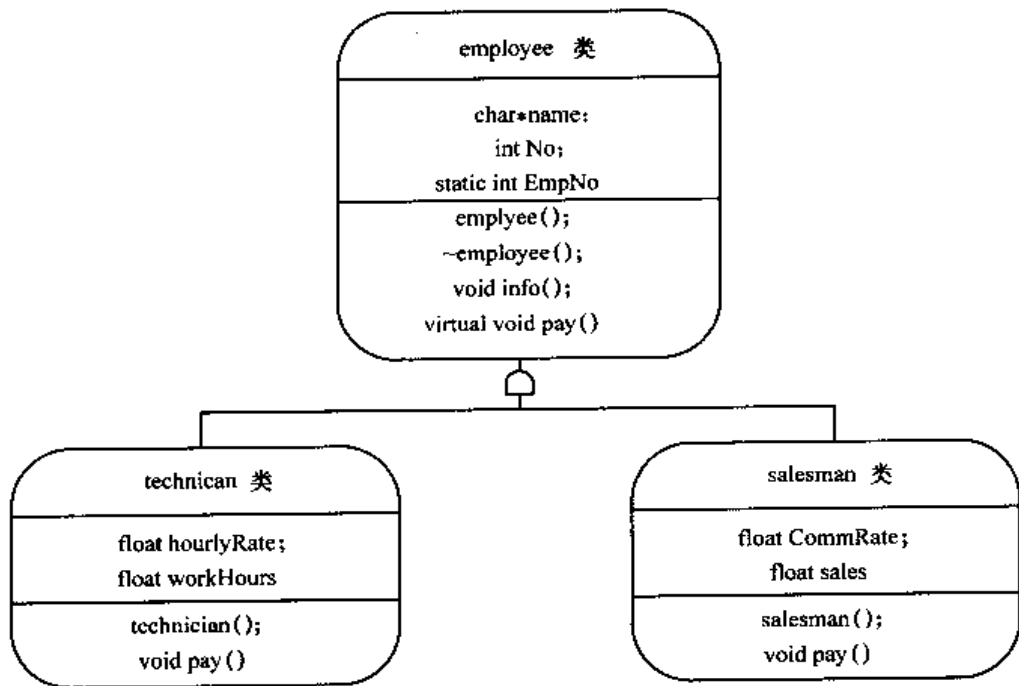


图 8-1 C++ 类层次图

销员两类雇员，每个雇员都包含姓名和编号基本信息，技术人员按每小时 100 元计酬，推销员按当月销售额的 4%提成，要求得出雇员的基本信息和当月酬金。

针对上述问题，我们可以用 C++构造一个雇员类层次：雇员类、技术人员类和推销员类。其中，雇员类为基类，技术人员类和推销员类为派生自雇员类的子类，它们的 C++类声明为：

---

```
//employee.h
class employee {                                //雇员类(基类)
protected:
    char *name;                                //姓名
    int No;                                    //个人编号
    static int EmpNo;                          //公司职员统一编号
public:
    employee(const char *pname);               //构造函数
    ~employee();                               //析构函数
    void info();                              //显示基本信息
    virtual void pay();                       //计算月薪虚函数
};

class technician:public employee {             //技术人员派生类
private:
    float hourlyRate;                         //每小时酬金
    float workHours;                          //当月工作时数
public:
    technician(const char *pname,float hours=0,float rate=100);
    void pay();                               //计算月薪虚函数
};

class salesman: public employee {              //推销员派生类
private:
    float CommRate;                           //按销售额提取酬金的百分比
    float sales;                              //当月销售额
public:
```

---

```
salesman(const char *pname,float sale=0,float rate=0.04);
```

```
void pay();
};
```

---

其中, technician 和 salesman 类继承了 employee 类的数据成员(name、No 和 EmpNo)和函数成员(info、pay), 并在 employee 基类中用虚函数(pay)规定了统一接口, 以提供运行时的多态性。该类层次的实现代码为:

---

```
//employee.cpp
#include<iostream.h>
#include<string.h>
#include"employee.h"

int employee::EmpNo=1000;           //员工起始编号为 1000

employee::employee(const char *pname) {
    name=new char[strlen(pname)+1];    //动态申请内存空间
    strcpy(name,pname);
    No=EmpNo++;                       //员工编号为目前最大编号加 1
}

void employee::info() {
    cout<<"雇员: "<<name<<" 编号: "<<No<<endl;
}

void employee::pay() {}              //虚函数的函数体为空

employee::~~employee() {
    delete name;                     //删除动态分配的内存空间
}

technician::technician(const char *pname,float hours,float rate):employee(pname){
    hourlyRate=rate;                //每小时酬金 100 元
```

```
        workHours=hours;
    }
    void technician::pay() {
        float accumPay=hourlyRate*workHours;    //计算月薪，按小时计酬
        cout<<"属兼职技术人员。每小时酬金："<<hourlyRate<<"；工作时数："
        <<workHours <<"；本月工资："<<accumPay<<endl;
    }

    salesman::salesman(
    const char *pname,float sal,float rate):employee(pname){
        CommRate=rate;                            //销售提成比例 4%
        sales=sal;
    }

    void salesman::pay() {
        float accumPay=sales*CommRate;            //月薪=销售提成
        cout<<"属推销员。提成百分比："<<CommRate<<"；销售额："
        <<sales <<"；本月工资："<<accumPay<<endl;
    }
```

---

外部程序测试代码如下：

---

```
#include<iostream.h>
#include"employee.h"

void main() {
    technician t1("t1",1); salesman s1("s1",100);    //实例化对象
    employee *emp[]={&t1,&s1};                      //声明指向基类的指针数组

    for(int i=0;i<2;i++) {
        emp[i]->info();                            //共享基类的的函数成员
        emp[i] ->pay();                             //虚函数展示运行时多态
    }
}
```

---

从上列代码可以看出：在支持面向对象程序设计的 C++ 中，继承和运行时多态是固有的语言机制，无需额外的代码来实现。和 C++ 不同，Fortran 90 没有提供实现继承和运行时多态的固有语言机制。

## 第二节 Fortran 90 模拟方法

在 Fortran 90 中，主要通过其派生类型、模块、接口块和指针，来模拟面向对象的三个主要特性。

### 一、封装(类)

模块(module)是 Fortran 90 新提供的程序单元，当中定义的数据类型及包含的模块例程可以被其他程序单元所共享，是模拟面向对象主要特性的基本框架。和上列 C++ 的 employee 类相对应的 Fortran 90 模块声明为：

---

```

module employee_class
    implicit none

    type employee                                !定义 employee 类型
    private                                     !缺省为 public
    character*10 name                            !姓名
    integer No                                  !个人编号
    end type employee

    integer,save,private::EmpNo=1000           !公司职员统一编号
    contains
    subroutine employee_(this,n)                !相当于构造函数
        type(employee),intent(out)::this
        character(*),intent(in)::n
        this%name=n
        this%No=EmpNo
        EmpNo=EmpNo+1
    end subroutine employee_

    subroutine emp_info(this)                   !展示基本信息模块例程

```



```

        type(employee),intent(in)::this
        print*,"雇员: ",this%name,"; 编号: ",this%No
    end subroutine emp_info
end module employee_class

```

其中, `private` 将成员可见性限定在模块范围内,从而达到信息隐蔽作用; `intent` 规定了参数传递属性,和接口定义语言 IDL 中的规定相同,共有 `in`、`out` 和 `inout` 三个属性值; `save` 属性相当于 C++ 中的 `static`; “`::`” 操作符用于声明变量,当声明变量同时进行初始化,或变量有属性规定时,此操作符是必须的; “`%`” 是派生类型的成员操作符,相当于 C++ 中的 “`.`” 操作符。

在 C++ 中,成员函数的第一个参数隐含指向当前对象的 `this` 指针。仿照这一惯例,我们将模块例程的第一个参数统一用 `this` 表示。另外, C++ 类的声明和实现代码通常分置于头文件(.h)和实现文件(.cpp)中;但在 Fortran 90 中,派生类型及其操作例程通常统一置于模块中(.f90)。使用上述模块的外部程序如下所示:

```

program employee_test
    use employee_class           !将模块引入主程序单元
    implicit none
    type(employee)::a,b         !声明二个自定义类型变量 a,b
    call employee_(a,"emp1")     !调用构造例程初始化对象 a
    call emp_info(a)             !展示对象 a 的基本信息
    call employee_(b,"emp2"); call emp_info(b)
end program

```

在模块 `employee_class` 中,派生类型 `type(employee)`、模块例程 `employee_` 和 `emp_info` 的访问属性为 `public`(模块缺省),所以外部程序在引用模块后,可以直接使用它们。在使用对派生类型进行操作的模块例程时, Fortran 90 将对象作为例程参数使用;而 C++ 则采取 “对象.成员函数” 的形式。

## 二、继承

我们可以采取 C++ 类组合方式,将基类对象作为派生类的一个数据成员,从而将隶属关系 “`is A`” 转变为包含关系 “`has A`”,见下列技术人员派生类的模块声明所示。

---

```

module technician_class
    use employee_class           !引入基类模块
    implicit none

    type technician              !定义 technician 类型
        private
        type(employee)::emp      !employee 对象成员
        real hourlyRate;         !每小时酬金
        real workHours;          !当月工作时数
    end type technician

contains
    subroutine technician_(this,name,hours,rate) !构造例程
        type(technician),intent(out) ::this
        character(*),intent(in)      ::name
        real,intent(in)               ::hours
        real,optional,intent(in)      ::rate    !可选参数
        call employee_(this%emp,name) !委托给基类例程
        this%workHours=hours
        if(present(rate)) then
            this%hourlyRate=rate
        else
            this%hourlyRate=100
        end if
    end subroutine technician_

    subroutine tech_info(this) !展示基本信息例程
        type(technician),intent(in)::this
        call emp_info(this%emp) !委托给基类例程
    end subroutine tech_info

    subroutine tech_pay(this) !展示当月酬金例程

```

```

type(technician),intent(in)::this
real pay
pay=this%hourlyRate * this%workHours
print*,"属兼职技术人员。每小时酬金：",this%hourlyRate,
    &"；工作时数：",this%workHours,"；本月工资：",Pay
end subroutine tech_pay
end module technician_class

```

在该类层次中，继承主要体现在构造例程 `technician_` 和展示基本信息例程 `tech_info` 上：通过将调用委托给基类的相应例程，克服了重复编写代码的弊端。

在声明 C++ 函数原型时，参数可以给出缺省值；Fortran 90 允许声明可选参数，再配合 `present` 函数，可以达到和 C++ 缺省值同样的效果，且两者遵循相同的声明和使用规则。

类似于 `technician` 类，`salesman` 派生类声明为：

```

module salesman_class
  use employee_class
  implicit none

  type salesman
    private
    type(employee)::emp
    real CommRate;           !按销售额提取酬金的百分比
    real sales;              !当月销售额
  end type salesman

  contains
    subroutine salesman_(this,name,sal,rate)
      type(salesman),intent(out)::this
      character(*),intent(in)::name
      real,intent(in)::sal
      real,optional,intent(in)::rate

      call employee_(this%emp,name)
    end subroutine salesman_
end module salesman_class

```

```
        this%sales=sal
        if(present(rate)) then
            this%CommRate=rate
        else
            this%CommRate=0.04
        end if
    end subroutine salesman_

    subroutine sale_info(this)
        type(salesman),intent(in)::this
        call emp_info(this%emp)
    end subroutine sale_info

    subroutine sale_pay(this)
        type(salesman),intent(in)::this
        real pay
        pay=this%CommRate * this%sales
        print*,"属推销员。提成百分比：",this%CommRate, &
            "; 销售额：",this%sales,"; 本月工资：",Pay
    end subroutine sale_pay
end module salesman_class
```

---

### 三、运行时多态

在 C++ 中，类的继承和运行时多态是统一的语言机制：通过在基类中声明虚函数，就可以在派生类中覆盖或重写对应的虚函数；使用时，将基类指针指向不同的派生类对象，就会引发不同的派生类虚函数，从而表现出不同的多态行为(见前述 C++ 代码所示)。

在 Fortran 90 中，我们可以构造一个辅助多态类进行模拟：该类的作用类似于 C++ 的虚基类，将指向上述两个派生类的指针直接作为其数据成员；然后增加赋值操作，结合“函数重载”，将不同的子类对象赋给相同的“多态类”对象；再增加动态分发机制，以根据不同的子类对象，调用不同的子类例程。其构造代码如下：

---

```

module poly_employee_class                                !定义“多态类”
    use technician_class ; use salesman_class            !引入子类模块
    implicit none

    !对外隐藏模块例程
    private::technician_,salesman_,assign_technician, &
        assign_salesman

    type poly_employee
        private
        type(technician),pointer::pt;                    !子类对象指针
        type(salesman),pointer::ps
    end type poly_employee

    interface new                                          !接口块，以重载例程
        module procedure technician_,salesman_
    end interface
    interface poly
        module procedure assign_technician,assign_salesman
    end interface

    contains
        !将 technician 赋给 poly_employee
        function assign_technician(pt) result(pps)
            type(poly_employee)::pps
            type(technician),target,intent(in)::pt
            pps%pt=>pt                                     !指针赋值
            nullify(pps%ps)                                !将另一个子类指针置空
        end function assign_technician

        !将 salesman 赋给 poly_employee
        function assign_salesman(ps) result(pps)

```

```

        pps%ps=>ps
        nullify(pps%pt)
    end function assign_salesman

    subroutine poly_info(this)      !展示基本信息，建立动态分发机制
        type(poly_employee),intent(in)::this
        if(associated(this%pt)) then
            call tech_info(this%pt)
        elseif(associated(this%ps)) then
            call sale_info(this%ps)
        end if
    end subroutine poly_info

    subroutine poly_pay(this)      !展示当月工资，建立动态分发机制
        if(associated(this%pt)) then
            call tech_pay(this%pt)
        elseif(associated(this%ps)) then
            call sale_pay(this%ps)
        end if
    end subroutine poly_pay
end module poly_employee_class

```

其中，接口块 `new` 和 `poly` 给不同的模块例程赋以相同的名称，但它们的参数表不同，从而对 C++ 中的函数重载提供了支持。在赋值例程中，将传入的子类型变量赋给相应的指针，并置空另一个指针成员；而在动态分发例程中，用 `associated` 函数判断哪一个指针成员有关联，依此建立起动态分发机制。

需要指出的是：Fortran 90 中的指针(`pointer`)只是一对象属性，并非指向对象的地址，其作用类似于 C++ 中的引用。因此，无需对“多态类”的数据成员进行初始化，即不需要创建构造例程。下面列出使用整个类层次的主程序代码：

```

program poly_test
    !重新命名模块例程
    use poly_employee_class,info=>poly_info,pay=>poly_pay
    implicit none

```

!声明子类对象

```
type(technician),target::t1;type(salesman),target::s1
```

```
type(poly_employee)::emp
```

!声明“多态”对象

!实例化对象

```
call new(t1,'JONES',1.0);call new(s1,'SMITH',100.0)
```

```
emp=poly(t1)
```

!虚基类指针指向子类

```
call info(emp);call pay(emp)
```

!调用虚函数

```
emp=poly(s1);call info(emp);call pay(emp)
```

```
end program poly_test
```

C++声明对象的同时，由系统自动调用构造函数完成实例化；Fortran 90 通常需分两步进行：先声明对象，再显式调用构造例程。但是，若定义派生类型时不用 private 声明其数据成员，且实参列表次序和其数据成员的声明次序相同，那么也可以一步完成。如下列代码所示：

```
type(employee) :: a = employee("empl",1000)
```

### 小 结

- 上述实例表明，完全可以用 Fortran 90 提供的现代语言特征：派生类型、模块、接口块、指针等，模拟 C++主要的面向对象特性。值得注意的是：在模拟运行时多态机制时，需设置一“多态类”，其作用类似于 C++的虚基类。
- 模拟继承时，可以采取 C++类组合方式，将基类对象作为子类的一个数据成员。函数成员继承，则通过将子类例程调用委托给基类例程来实现，从而克服了需要复制基类代码的弊端。
- 模拟多态时，将子类对象指针作为“多态类”的数据成员。要使“多态类”产生作用，需进行赋值操作：将子类对象赋给“多态类”的指针成员，并建立动态分发机制，根据传入的不同对象调用不同子类的例程。
- Fortran 90 对 C++面向对象特性的成功模拟，为解决当前数值计算的代码重用和维护性提供了有效途径，并为应对 Fortran 2003 面向对象编程打下了基础。

## 第九章 格式化输入/输出及文件操作

在前面的章节中，我们集中精力于编程解决各种问题上，而没有过多关注输入/输出的外观。在这一章中，将着重探讨格式化输入/输出，包括各种格式编辑符、输入/输出语句以及文件操作。

### 第一节 PRINT 语句

在讲解 PRINT 语句的使用之前，先来看一格式输出实例(例 9-1)。该例输出 Fibonacci 数列：从第三项开始，每一项为前两项之和。

[例 9-1] 输出 Fibonacci 数列。

---

```
PROGRAM RabbitBreeding
  IMPLICIT NONE
  INTEGER Month
  REAL Fn, Fn_1, Fn_2

  ! Format 规定
  10  FORMAT( 'Month', T12, 'Population', T27, 'Ratio' /      &
             5('-'), T12, 10('-'), T27, 5('-') )
  20  FORMAT( I3, T12, F7.1, T27, F6.4 )

  Fn_1 = 1
  Fn_2 = 1
  PRINT 10                                !输出表头

  DO Month = 3, 12
    Fn = Fn_1 + Fn_2
    PRINT 20, Month, Fn, Fn / Fn_1        !输出表中记录
    Fn_2 = Fn_1
    Fn_1 = Fn
```



```

END DO
END PROGRAM

```

程序运行结果为：

Month	Population	Ratio
3	2.0	2.0000
4	3.0	1.5000
5	5.0	1.6667
6	8.0	1.6000
7	13.0	1.6250
8	21.0	1.6154
9	34.0	1.6190
10	55.0	1.6176
11	89.0	1.6182
12	144.0	1.6180

程序中，用 PRINT n 代替 PRINT\*，这里的 n 为语句标号(1 ~ 99999)，FORMAT 语句则规定了输出的格式。

第一个 FORMAT 语句用于输出表头。T12 将制表位移到第 12 列，即后面的输出项从第 12 列开始；撇号(/)终止当前行，并开始一新行；5('-')表示将 '-' 重复输出 5 次。

第二个 FORMAT 语句用于输出表中记录，即输出变量列表。I 为整数编辑符，I3 指整数输出宽度占 3 列；F 为实数编辑符，F7.1、F6.4 指实数输出宽度分别占 7 列和 6 列，而小数点后分别有 1 位和 4 位。

PRINT 语句通常用于屏幕输出，其一般形式为：

```
PRINT fmt [,list]
```

Fmt 可以是下列三种格式中的一种：

(1)FORMAT 格式。通过语句标号引用 FORMAT 格式规定。例如：

```
PRINT 10, X
```

```
10 FORMAT( 'The answer is: ', F6.2 )
```

当中的 10 为语句标号，X 为实型变量(下同)，FORMAT 为格式语句。

(2)自由格式。以星号表示自由格式。例如：

```
PRINT*, 'The answer is: ', X
```

(3)字符串规定格式。字符串即为格式规定。例如：

```
PRINT "( 'The answer is: ', F6.2 )", X
```

List 为输出列表, 当中的项可以是常量、表达式、变量或隐式 DO 循环列表(形如: `variable = expr1, expr2 [,expr3]`)。

## 第二节 格式编辑符

例 9-1 中的 F7.1 为格式编辑描述符(简称编辑符), 它规定了输出/输入的外观。具体讲, 编辑符决定了计算机内部表示的数据如何转换为输出设备及文件中可读的字符串, 或输入设备或文件中的字符串如何转换为计算机内部表示的数据, 共有三类编辑符: 数据编辑符、字符串编辑符和控制编辑符。

### 一、数据编辑符

在下面的行文中, 字母 w、m、d 和 e 代表正整型常量, b 代表空格。假如规定的宽度不够, 输出项以星号(\*)填充。数据编辑符分为整数、实数、复数、逻辑数和字符 5 种编辑符。

#### (一) 整数

整数使用 I 编辑符, 其一般形式为 Iw, 当中的 w 规定了输出宽度, 包括前导的负号所占据的一列。

另一种形式为 Iw.m, 保证有 m 位数字(不包括负号)被输出, 若不够 m 位, 前面填 0。例如: 在 I6.3 编辑符下, -99 被输出为 bb-099。

二进制、八进制和十六进制整数, 分别使用 Bw、Ow 和 Zw 编辑符。例如:

```
READ '(B4)', I
```

将字符串 1111 转换成十进制数 15。

在 Bw、Ow 和 Zw 编辑符下, 同样可以规定最小的位数 m。

#### (二) 实数

实数编辑符包括 F、E、G、EN 和 ES。

##### 1. F 编辑符

F(Fixed point)编辑符的形式为 Fw.d, w 为总的输出宽度(包括负号和小数点列), d 为小数点后的位数。例如: 在 F8.2 编辑符下, -12.345 输出为 bb-12.35, 小数点后的第二位四舍五入。

输入时, 若字符串包含小数点, d 被忽略。例如: 在 F8.2 编辑符下, 1.2345 仍被读为 1.2345; 在 F9.1 编辑符下, 12.345E-2 被读为 0.12345。

输入时, 若字符串不包含小数点, 最右边的 d 位为小数部分。例如: 在 F7.2 编辑符下, -12345 被读为 -123.45。

## 2. E 编辑符

E 编辑符包括 Ew.d 和 Ew.dEe, 两种形式的输入规则与 F 编辑符相同。

输出时, Ew.d 中的 w 规定了输出项的宽度, 包括可能的前导负号列、小数点和 4 列指数部分; d 仍为小数部分的位数; 基数的绝对值小于 1。例如: 在 E10.4 编辑符下, 1.234E+23 输出为 0.1234E+24。

Ew.dEe 中的 e 限定了指数部分的位数, 不足部分填充 0。例如: 在 E11.4E3 编辑符下, 1.234E+23 输出为 0.1234E+024。

## 3. G 编辑符

G 编辑符包括 Gw.d 和 Gw.dEe, 是 F 和 E 编辑符的一般形式。在事先无法确定变量大小时, G 编辑符十分方便, 它依据变量值的大小, 相应地采取 F 编辑符或 E 编辑符。

## 4. EN 编辑符

EN(engineering)编辑符限定指数为 3 的倍数, 基数大于或等于 1 且小于 1000, 其他的规则与 E 编辑符相同。例如: 在 EN10.2 编辑符下, 2.17 输出为 2.17E+00; 2.17E-2 输出为 21.70E-03; 2.17E+5 输出为 217.00E+03。

## 5. ES 编辑符

ES(scientific)编辑符限定基数大于或等于 1 且小于 10。例如: 在 ES10.2 编辑符下, 0.217 输出为 2.17E-01; 21.7 输出为 2.17E+01。

## (三)复数

复数由实部和虚部构成, 因此其编辑符由一对实数编辑符(F、E、EN 或 ES)组成, 实部和虚部可以使用不同的实数编辑符, 它们之间由字符串、控制编辑符分隔。

## (四)逻辑数

L 为逻辑数编辑符, 其一般形式为 Lw。输出时, T 或 F 出现在第 w 列, 即右对齐; 输入时, 可以输入大写或小写的 T 或 F, 也可以输入.TRUE.或.FALSE., 此时 w 被忽略。

## (五)字符

字符编辑符包括 A 和 Aw。

在 A 编辑符下, 若实际字符个数小于声明时的字符个数, 取实际字符个数, 右边填充空格(即左对齐); 若实际字符个数大于声明时的字符个数, 则从左到右取声明时的字符个数。例如:

```
CHARACTER*7 :: C1 = 'SMITH', C2 = 'JOHN SMITH'
```

输出分别为 SMITHbb 和 JOHN SM。

在 Aw 编辑符下, 输出时, 左边的 w 个字符被输出, 不足时左边填充空格。例如:

```
CHARACTER*5 :: C = 'ABCDE'
```

在 A3 编辑符下，输出为 ABC；在 A6 编辑符下，输出为 bABCDE。

Aw 编辑符的输入规则有点特殊。假如 len 代表声明字符变量的长度，如果  $w < \text{len}$ ，最左边的 w 位字符被读入，右边填充  $\text{len}-w$  个空格。例如：在 A3 编辑符下，上述字符变量 C 读入 ABCDE，实际值为 'ABCbb'；

但如果  $w > \text{len}$ ，在 w 个字符中(包括右边的空格)最右边的 len 位字符被读入。例如：输入 ABCDEF，在 A6 编辑符下，C 的值为 'BCDEF'。

值得注意的是：上述整数、实数、复数和逻辑数据的输出，均为右对齐。

## 二、字符串编辑符

可以将字符串常量嵌入 FORMAT 语句中输出。例如：

```
PRINT 10
```

```
10 FORMAT( 'Fortran 90 is the language for me' )
```

这里顺便提一下过时的 H(Hollerith)编辑符：输出字符串时，将 nH 描述符置于字符串首部。例如：

```
PRINT 10
```

```
10 FORMAT( 24HWe must count carefully! )
```

当中的 n 代表输出字符串的字符个数。

使用 H 编辑符时，字符个数 n 的规定必须准确；否则，会出现程序编译错误。

## 三、控制编辑符

控制编辑符可以准确定位输出列，开始一新行，在输入时跳过若干列等。下面结合不同的格式控制，探讨相应的控制编辑符的使用。

### (一)内嵌空格输入及正号输出

在 BN(blanks null)和 BZ(blanks zero)编辑符下，输入列表中的内嵌空格要么作为 0，要么作为空白。例如：

```
INTEGER I1,I2
```

```
READ '(BN, I3, BZ, I3)',I1,I2
```

若输入字符串 1b31b3，I1 和 I2 的值分别为 13 和 103。

有三种编辑符控制正数的正号输出：SP(sign print)输出正号，SS(sign suppress)正号被压缩掉，S 恢复缺省设置。例如：

```
INTEGER :: I = 99
```

```
PRINT '(SP, I3, SS, I3, S, I3)',I,I,I
```

I 被输出三次，其结果为+99b99b99。

## (二)比例因子

比例因子  $kP$  可以应用到 E、F、EN、ES 和 G 编辑符下实数的输入,  $k$  为整数比例因子, 将除指数部分外的实数缩小  $10^k$ 。例如: 在(2P, F3.0)格式下, 1.0 被读为 0.01。

比例因子  $kP$  也对 E、F 和 G 编辑符下实数的输出产生影响, 在 F 编辑符下,  $kP$  将实数扩大  $10^k$  输出; 在 E 编辑符下,  $kP$  将指数缩小  $k$  倍、基数扩大  $10^k$  输出。

## (三)制表位

制表位编辑符 T 包括  $T_n$ 、 $TR_n$  和  $TL_n$ 。 $T_n$  在例 9-1 中已出现过, 它将制表位移到第  $n$  列;  $TR_n$ (或  $nX$ ) 将制表位从当前位置向右移动  $n$  列;  $TL_n$  将制表位从当前位置向左移动  $n$  列(最多移动至第一列)。

输入时, 制表位可以用来跳过若干列或重读数据。例如: 在(I1, 2X, I1)格式下, 输入字符串 1234 被读为整数 1 和 4。

输出时, 制表位可以用来替换输出的数据。例如: 在(I3, TL2, I3)格式下, 整数 911 和 999 输出为 9999。

## (四)另起一行

撇号(/)编辑符在例 9-1 中也出现过, 它的作用是开始新的输出行。撇号可以连续出现, 并可用重复数来标识。例如: 格式规定中的///或 3/。

## (五)冒号编辑符

假如再没有输出项, 冒号(:)编辑符将使格式控制失效。例如:

```
INTEGER :: X(3)=(/10,20,30/), N = 2, I
```

```
PRINT 10, (X(I), I = 1, N)
```

```
10 FORMAT('X1=', I2 : 'X2=', I2 : 'X3=', I2)
```

这里, 根据  $N = 1, 2, 3$ , 分别产生如下的输出:

```
X1=10
```

```
X1=10 X2=20
```

```
X1=10 X2=20 X3=30
```

若第一个 I2 编辑符后没有冒号, 当  $N = 1$  时, 将输出:

```
X1=10 X2=
```

## (六)编辑符重复

如同前面出现的撇号重复一样, 数据编辑符也可出现重复。一个重复次数可以应用到括号包围的一组编辑符, 还可被嵌套使用。例如:

```
3(2F6.2, 2(I2, 3I3))
```

假如格式规定的项数小于 I/O(输入/输出)项数, 就开始一个新行, 相同的格式规定又施加到其余的 I/O 项上。例如, 下列代码将输出 100 个数组元素, 每行 20 个:

```
PRINT 10, (X(I), I = 1, 100)
```

```
10 FORMAT( 20I3 )
```

同样，输入时，每当格式规定被重复，就开始读入一新的行。例如：

```
READ 10, I, J
```

```
10 FORMAT( I1 )
```

若输入两行数据：

```
12
```

```
34
```

I 和 J 分别被读为 1 和 3。

### 第三节 READ 语句

类似于 PRINT 语句，READ 语句的基本形式为：

```
READ fmt [,list]
```

其中，fmt 为语句标号、星号或字符串；list 为输入列表。

在从输入设备读入数据的过程中，有时会出现错误、读到文件尾等情况，引发程序意外地被终止。为了避免这种情形的发生，Fortran 90 提供了更一般的 READ 语句：

```
READ([UNIT=]u,[FMT=]fmt[,IOSTAT=ios][,ERR=errorlabel][,END=endlabel])  
[list]
```

其中，单元号 u 和格式规定 fmt 是必选项，其他为可选项。

这里的单元指 I/O(输入/输出)设备，如打印机、终端、磁盘驱动器等，通过编译器与程序相连接。输出时的终端指屏幕，输入时的终端指键盘，它们被称为标准 I/O 设备或单元(系统允许用户重新指定标准 I/O 设备)。I/O 设备通常有一个与之关联的单元号，其范围为 1~99。上述 READ 语句中的单元规定可以有三种形式：整数表达式、星号(指标准输入设备)和用于内部文件的字符变量。

如果可选项中的 IOSTAT 被规定，ios 须是整型变量。READ 语句执行后，依据到达行末、文件尾的不同情况，ios 获得不同的负值(具体的负值依赖于系统)；当错误发生时，ios 为正值或 0。提供 IOSTAT 参数，可以阻止由于异常使程序意外终止。

### 第四节 WRITE 语句

WRITE 语句的一般形式为：

---

```
WRITE ([UNIT=]u, [FMT=]fmt [,IOSTAT=ios]
[,ERR=errorlabel]) [list]
```

当中各参数的规定与 READ 语句的相同。

值得一提的是：输出设备可以在程序执行期间指定。当开发需要输出大量数据到打印机上时，为了在正式打印前浏览、检查输出的数据，可以选择输出到打印机和终端(屏幕)两种方式。如下列代码所示：

---

```
CHARACTER OutputDevice*3
PRINT*, 'Where do you want the output ("prn" or "con")?'
READ*, OutPutDevice
OPEN( 1, FILE = OutputDevice )
WRITE( 1, * ) 'Output on designated device'
```

---

Prn 和 con 分别代表 PC 打印机和终端。

## 第五节 内部文件

在上述 READ 和 WRITE 语句中，I/O 设备可以是内部文件——字符变量。即可以从内部文件读取数据，也可以向内部文件写数据。通常，内部文件用于字符串和数字之间的转换。例如：下列代码段从内部文件读取数据，将字符串转换成数字：

---

```
INTEGER NYEAR
CHARACTER (30) STRING
STRING = "1984"
READ( STRING, 10 ) NYEAR
10  FORMAT( I4 )
```

---

下列代码段则向内部文件写数据，将数字转换为字符串：

---

```
CHARACTER(50) CAPTION
INTEGER :: YEAR = 1984
WRITE( CAPTION, 10 ) YEAR
10  FORMAT( 'Sales figures for the financial year: ', I4 )
```

---

## 第六节 外部文件

外部文件可以将数据永久地保留在磁盘上, 以方便对数据的更新、检查和分析。外部文件分为顺序访问文件(文本文件)和直接访问文件(随机文件), 直接访问文件要求当中所有的行(记录)是等长的, 顺序访问文件则无此要求。文件中的每一行可以有格式的, 也可以是无格式的。

### 一、顺序访问文件

顺序访问文件只能从头开始读数据, 假如要读取的数据在文件末, 也需要先读过前面不需要的行。所以, 读取顺序访问文件要比读取直接访问文件来得慢。另外, 不能直接替换或删除顺序访问文件中的部分数据。但顺序访问文件是 ASCII 文本文件, 可以使用各种字处理软件进行查看。

例 9-2 将展示如何更新一个顺序访问文件: 先从文件中读取每一行, 将要保留的行写到临时文件(SCRATCH)中, 删除原文件中的所有数据, 再将临时文件中的数据拷回原文件。

[例 9-2] 顺序访问文件操作。

---

```
PROGRAM ExternalFile
  IMPLICIT NONE
  CHARACTER(80) Name, FileName, Ans
  INTEGER :: IO = 0

  WRITE( *, '(A)', ADVANCE = 'NO' ) "Name of file to be updated: "
  READ*, FileName

  OPEN( 1, FILE = FileName )
  OPEN( 2, STATUS = 'SCRATCH' )

  DO WHILE (IO == 0)      ! 没有到达文件尾
    READ( 1, *, IOSTAT = IO ) Name
    IF (IO == 0) THEN
      PRINT '(A)', Name
      WRITE( *, '(A)', ADVANCE = 'NO' ) "Delete (Y/N)? "
```



```

        READ*, Ans
        IF (Ans /= 'Y' .AND. Ans /= 'y') WRITE( 2, '(A)' ) Name
    END IF
END DO

REWIND( 2 )                !定位文件头
CLOSE( 1, STATUS = 'DELETE' ) !关闭文件的同时删除所有数据
OPEN( 1, FILE = FileName )  !打开空文件

IO = 0
DO WHILE (IO == 0)
    READ( 2, *, IOSTAT = IO ) Name
    IF (IO == 0) WRITE( 1, '(A)' ) Name
END DO

CLOSE( 1 )
CLOSE( 2 )                !临时文件被删除
END PROGRAM

```

要对文件中的数据进行操作，首先要打开(OPEN)文件，建立文件与程序之间的连接。OPEN 语句的一般形式为：

OPEN( [UNIT = ]u, speclist )

当中的 u 为单元号，假如不使用有名参数(UNIT=)，单元号必须出现在开头。speclist 为格式规定列表，其中的项大多是可选的：

(1)FILE 参数指文件名，假如 FILE 被省略，则必须提供 STATUS 参数，且其值为 SCRATCH。

(2)STATUS 参数若规定为 SCRATCH，一个临时文件被创建，当文件关闭或程序结束时，临时文件随即消失；若规定为 NEW，不能有同名文件存在；若规定为 OLD，同名文件必须存在；若规定为 REPLACE，文件不存在时创建新文件，文件存在时其内容将被替换。

(3)ACCESS 参数规定了文件读取模式，顺序文件缺省为 SEQUENTIAL。

(4)FORM 参数规定了是否按格式读取，顺序文件缺省为 FORMATTED。

(5)POSITION 参数代表文件指针位置，若规定为 APPEND，新的数据将被追加到文件尾。

通常, CLOSE 语句和 OPEN 语句配套使用, 它使文件与程序断开连接。其一般形式为:

CLOSE( [UNIT = ]u [, STATUS = st] )

其中,u 代表要关闭文件的单元号。STATUS 可选参数可以规定为 KEEP 或 DELETE, 缺省为 KEEP, 即关闭后文件中的数据被保留; 若规定为 DELETE, 关闭时文件中的数据被擦除; 若文件为临时文件, 则 STATUS 只能是 DELETE(缺省)。当程序正常结束时, 不管是否执行了 CLOSE 语句, 所有打开的文件都自动被关闭。

在读写文件时, 文件指针在不断移动。REWIND 语句将文件指针定位到文件的开头, 其使用形式为:

REWIND u

BACKSPACE 语句将文件指针定位到当前行的开头, 其使用形式为:

BACKSPACE u

顺序文件末尾有一个结束标记。计算机系统大多自动追加结束标记, 但也允许用户显式追加结束标记:

ENDFILE u

前面提到: 在 OPEN 语句的 speclist 中, 可选参数 FORM 的缺省值为 formatted, 即顺序文件默认是有格式的。但顺序文件也可以是无格式的, 无格式可以有效地节省存储空间。例如: 整数 2147483647 在 32 位 FTN90 编译器下, 无格式存储占 4 个字节; 而有格式存储要占 10 个字节(即表示该整数需要 10 个字符)。

例 9-3 将一个数组以无格式方式写到外部文件, 然后再从文件中读取。

[例 9-3] 无格式顺序访问文件操作。

---

```
PROGRAM Unformat
  IMPLICIT NONE
  INTEGER I
  INTEGER, DIMENSION(10) :: A = (/ (I, I = 1,10) /), B

  OPEN( 1, FILE = 'TEST', FORM = 'UNFORMATTED' )
  WRITE (1) A

  REWIND (1)      !读前, 文件指针重定位到文件头
  READ (1) B
  PRINT*, B
```

---

```
CLOSE (1)
END PROGRAM
```

---

## 二、直接访问文件

在直接(或随机)访问文件中,可以方便地读取、改写或追加一个数据行。直接访问文件缺省是无格式的(二进制),但要求每行记录长度是等长的。在 OPEN 语句中, RECL 参数规定记录长度;在读取过程中,可由 INQUIRE 语句查询记录长度。

在例 9-4 中,将从键盘读取的名字写到随机文件中,再从文件中读取。

[例 9-4] 直接访问文件操作。

---

```
PROGRAM DirectAccess
  IMPLICIT NONE
  CHARACTER (20) NAME
  INTEGER I,LEN

  INQUIRE (IOLENGTH = LEN) NAME           !确定记录长度
  OPEN( 1, FILE = 'LIST', STATUS = 'REPLACE', ACCESS = 'DIRECT',
        & RECL = LEN )

  DO I = 1, 3
    READ*, NAME
    WRITE (1, REC = I) NAME                 !往文件写数据
  END DO

  NAME = 'JOKER'
  WRITE (1, REC = 4) NAME                   !追加行

  DO I = 1, 4
    READ( 1, REC = I) NAME                 !从文件读数据
    PRINT*, NAME
  END DO

  CLOSE (1)
END PROGRAM
```

---

值得注意的是：在直接访问文件的 READ 和 WRITE 语句中，必须给出行号或记录号(REC)。

## 第七节 不换行的读写

通常，READ 和 WRITE 语句都要求从一个新行开始读和写。Fortran 90 提供了 non-advancing I/O，允许进行不换行读写，即下次读写时文件指针仍保持在行内。

不换行的写语句形如：

```
WRITE (*, '(A)', ADVANCE = 'NO') 'Enter a number: '
```

执行时，光标停在字符串 'Enter a number: ' 之后，而不是在下行的开头。

不换行的读语句有时可以发挥特殊的作用，如统计文本文件中的字符个数。

例 9-5 从键盘读入文件名，然后利用不换行的读语句统计文件中的字符个数。

[例 9-5] 不换行读语句的使用。

---

```
PROGRAM NON_ADVANCE
  IMPLICIT NONE
  CHARACTER ch*1,name*20
  INTEGER :: IO = 0, Num = 0

  WRITE (*, '(A)', ADVANCE = 'NO') 'Input file name: '
  READ*, name

  OPEN( 1, FILE = name )

  DO WHILE (IO /= -1)
    READ (1, '(A1)', IOSTAT = IO, ADVANCE = 'NO') ch
    IF (IO == 0) Num = Num + 1
  END DO

  PRINT*, Num
  CLOSE (1)
END PROGRAM
```

---

在 FTN90 编译器下, 当文件指针在行内时, IOSTAT 参数的值为 0; 指针在文件尾时, 值为-1; 指针在行尾时, 值为-2。

### 小 结

- 格式可以是带语句标号的 FORMAT 语句、星号(\*)表示的自由格式(表控格式)和字符串规定格式。
- PRINT 语句只能将数据输出到屏幕, WRITE 语句可以将输出定向到屏幕、打印机或文件, READ 语句可以从键盘、文件中读入数据。代表屏幕、文件等 I/O 设备单元, 可以是单元号、星号(指标准 I/O 设备)和字符变量(内部文件)。
- 格式可以由编辑描述符控制, 编辑描述符包括数据编辑符、字符串编辑符和控制编辑符。
- OPEN 语句通过单元号建立程序与文件之间的连接, CLOSE 语句则断开连接。
- 内部文件主要用于字符串与数字之间的转换。
- 外部文件分为顺序访问文件和直接访问文件(随机文件), 每种文件又分为有格式的和无格式的。
- 顺序文件的数据缺省是有格式的, 每行不要求是等长的, 对顺序文件的读写须从头依次进行, 可以追加新行, 但不能改写现有的行。利用各种字处理软件可以对顺序文件进行查看。
- 随机文件的数据缺省是无格式的(二进制), 每行必须是等长的, 对随机文件的读写按行号(记录号)进行, 可以追加新行, 也可改写现有的行。随机文件的操作较之顺序文件更有效率。
- 利用 Fortran 90 提供的 non-advancing I/O, 可以进行不换行的读写。

## 第十章 Fortran 与 C/C++的混合编译

如果在 32 位 Windows 系统下,既安装了 Compaq Visual Fortran 6.x(安装时选择 Developer Studio 界面),又安装了 Microsoft Visual C++ 6.0,那么两者共享一个可视化开发环境(Microsoft Visual Studio 6.0)。在该环境下,可以编写混合了 Fortran 与 C/C++的应用程序。编译时,分别采用不同的编译器进行编译;链接程序再将编译产生的不同目标对象文件(OBJ)链接成一个可执行文件(EXE 或 DLL)。这里涉及到两种语言编写的程序相互调用,即混合语言编程问题。由于 Fortran 与 C/C++在堆栈管理、目标例程命名、参数传递等方面所遵循的规则(统称为调用约定)不尽相同,要使混合编程获得成功,首先须全面一致地协调两者使用的调用约定。

Fortran 与 C/C++同属编译型语言,任何一种语言都可用来编写主程序,而对另一种语言编写的例程(子程序和函数的统称)实施调用。C/C++程序还可直接访问 Fortran 90 的模块;反之,C/C++函数也可并入 Fortran 90 模块,作为模块例程供其他程序单元使用。

### 第一节 调用约定的协调

Fortran 使用的调用约定有缺省约定(Default)、C 约定和 STDCALL 约定。在 Visual Fortran 6.x 编译器下,通过 ATTRIBUTES 关键字来规定调用约定相关的属性:作用于整个外部例程的缺省约定、C、STDCALL、REFERENCE 和 VARYING 属性;作用于例程参数的有 VALUE 和 REFERENCE 属性。这些属性对堆栈清理、例程命名、参数传递方面的影响列于表 10-1。

下面从三个方面探讨 Fortran 与 C/C++在调用约定上的协调问题。

#### 一、堆栈管理

Fortran 与 C/C++间的例程调用,通过堆栈进行:先将例程地址、例程参数压入堆栈,然后再弹出例程参数和例程地址。进栈时,例程参数从左到右;出栈时,例程参数从右到左,即先进栈的参数后出栈。

在 C 调用约定下(见表 10-1),当执行流程由被调用例程返回时,由调用程序(Caller)负责出栈、清理堆栈操作,这使得调用程序的目标代码稍大一些,因在每一调用点处都要增加管理堆栈的代码;在缺省约定和 STDCALL 约定下,由被调用例程(Callee)

控制堆栈，管理堆栈的代码驻留在被调用例程中，且只出现一次。

表 10-1 Visual Fortran 6.x 调用约定相关属性的影响

		Default	C	STDCALL	C, REFERENCE	STDCALL, REFERENCE
Stack Cleanup		Callee	Caller	Callee	Caller	Callee
Procedure Naming	Name Translated	_name@n	_name	_name@n	_name	_name@n
	Case of Name	All uppercase	All lowercase	All lowercase	All lowercase	All lowercase
Argument Passing	Scalar	Reference	Value	Value	Reference	Reference
	Scalar [value]	Value	Value	Value	Value	Value
	Scalar [reference]	Reference	Reference	Reference	Reference	Reference
	Array	Reference	Reference	Reference	Reference	Reference
	Array [value]	Error	Error	Error	Error	Error
	Array [reference]	Reference	Reference	Reference	Reference	Reference

注：表中例程参数只列出了数值计算常用的标量(Scalar)和数组(Array)。

正因为 C 约定下由调用程序控制堆栈，调用程序知道有多少参数被传递、占多少字节、驻留在堆栈的什么位置等信息，所以在 C 约定下可以传递可变数量的参数，即可选参数。在 C 约定之外规定外部例程拥有 VARYING 属性，以使 Fortran 不再要求匹配参数个数。例如，要在 Fortran 程序中调用 C++ 函数(其缺省约定为 \_\_cdecl)，函数参数包括可选参数，下列为 Fortran 程序中建立的外部函数接口块：

INTERFACE

REAL FUNCTION Func(X, A, B, C)

!DEC\$ ATTRIBUTES C,VARYING :: Func

!DEC\$ ATTRIBUTES ALIAS:'\_Func' :: Func

REAL :: X, A, B, C

END FUNCTION

END INTERFACE

对应的 C++ 外部函数(.cpp 文件)为：

extern "C"

```

float Func(float X, float A=0.0,float B=0.0,float C=0.0)
{
    float f;
    f = A*X*X + B*X + C;
    return f;
}

```

后面的三个参数为可选参数。在 Fortran 主程序中,可采取如下的形式传递可选参数:

```
PRINT*, Func(2.0, 1.0)
```

在 Visual C++ 6.0 下,利用堆栈传递参数的调用约定有 `__cdecl`(缺省)和 `__stdcall`,它们与 Fortran 的匹配关系列于表 10-2。

表 10-2 Fortran 与 C/C++匹配的调用约定

Fortran	C/C++
缺省约定	<code>__stdcall</code>
C	<code>__cdecl</code>
STDCALL	<code>__stdcall</code>

## 二、例程命名

在编译生成的目标文件(.OBJ)中,像全局变量、模块、外部例程等标识符标识了特定的内存位置,在整个程序作用域内是惟一的,并被各个程序单元所共享。由于 C 语言区分字母的大小写(或大小写敏感),Fortran 不加以区分,C++又添加了特定的修饰,导致编译后的目标标识符发生改变(见表 10-1)。这种命名目标标识符的规则,称为命名约定。

为避免链接失败,必须协调 Fortran 与 C/C++的命名约定,使产生的目标外部例程名保持一致。而不同的调用约定,其命名约定也不相同。表 10-3 列出了不同调用约定下产生的 Fortran、C/C++目标例程名。

C++添加的修饰依赖于特定的系统,如果将其去掉,则 C++与 C 的命名约定相同。要去掉修饰,需在函数中添加 C 链接选项(`extern "C"`),例如:

```
extern "C" {int Sum_Up( int a, int b, int c ); }
```

产生的目标例程名为 `_Sum_Up`,调用约定采取了 C/C++的缺省约定 `__cdecl`。

按照表 10-3,协调 Fortran 与 C/C++的例程命名分大写、小写和大小写混合三种情况。



表 10-3 Fortran 与 C/C++的命名约定

Language	Attributes	Name Translated	Case of Name
Fortran	default	_name@n	All uppercase
Fortran	cDEC\$ ATTRIBUTES C	_name	All lowercase
Fortran	cDEC\$ ATTRIBUTES STDCALL	_name@n	All lowercase
C	__cdecl (default)	_name	Mixed case
C	__stdcall	_name@n	Mixed case
C++	Default	_name@@decoration	Mixed case

注：表中 cDEC\$ 的 c 在固定格式下代表\*和 C，在自由格式下代表!，n 指参数列表所占堆栈大小。

### (一)大写

假如 Fortran 采用缺省约定，其目标外部例程名统一被转换为大写，那么在 C/C++中须采用\_\_stdcall 约定，并用大写命名外部函数。例如：

Fortran：

```
REAL FUNCTION  FFARCTAN( Angle)
  !DEC$ ATTRIBUTES VALUE :: Angle
  REAL( 4 )      :: Angle
```

声明参数 Angle 具有值属性，使其与 C/C++的值传递方式相统一。在这种情况下，产生的目标例程名为\_FFARCTAN@4，数字 4 为值参数 Angle 所占的字节数。

在 32 位系统下，值参数一般占 4 个字节，不足 4 字节的加宽到 4 字节，双精度实型数占 8 个字节；引用参数由于传递的是地址编码，而地址编码一律采用 4 字节的长整型数表示。

C(.c 文件)：

```
extern float __stdcall FFARCTAN( float angle );
```

C++(.cpp 文件)：

```
extern "C" { float __stdcall FFARCTAN( float angle ); }
```

### (二)小写

假如 Fortran 采用 C 约定或 STDCALL 约定，其目标外部例程名统一被转换为小写，那么在 C/C++中须采用缺省约定(\_\_cdecl)或\_\_stdcall 约定，并用小写命名外部函数。例如：

Fortran：

```
REAL FUNCTION  FFARCTAN( Angle)
```

```
!DEC$ ATTRIBUTES C :: FFARCTAN
```

```
REAL( 4 ) :: Angle
```

在这种情况下,产生的目标例程名为\_ffarctan。

C(.c 文件):

```
extern float ffarctan ( float angle );
```

C++(.cpp 文件):

```
extern "C" { float ffarctan ( float angle ); }
```

### (三)大小写混合

假如 C/C++ 的外部函数名采取大小写混合形式,在 Fortran 中若要保持一样的形式,须采用 ALIAS 属性来限定产生的目标例程名。例如:

Fortran:

```
REAL FUNCTION FFARCTAN( Angle)
```

```
!DEC$ ATTRIBUTES C , ALIAS: '_FfArctan' :: FFARCTAN
```

```
REAL( 4 ) :: Angle
```

在这种情况下,产生的目标例程名为\_FfArctan。

C(.c 文件):

```
extern float FfArctan ( float angle );
```

C++(.cpp 文件):

```
extern "C" { float FfArctan ( float angle ); }
```

## 三、参数传递

混合编程中的参数传递包括两方面的内容:参数传递方式和数据类型的对应。

### (一)参数传递方式

参数传递有两种方式:值传递和引用传递。前者传递参数的值,后者传递参数的地址。Fortran 参数传递方式取决于调用约定,见表 10-1。缺省约定(Default)下, Fortran 以引用方式传递所有参数;假如采用 C 或 STDCALL 约定,标量参数(即单个参数)采取值传递,数组参数仍采取引用传递。在调用约定之外,还可以规定参数拥有 VALUE 和 REFERENCE 传递属性,使参数分别以值和引用方式传递,并忽略调用约定对参数传递施加的影响。数组参数只能以引用方式传递,并通过数组名传递数组的首地址。在混合编程中,应通过 VALUE 和 REFERENCE 属性明确规定参数的传递方式,而不是依赖于调用约定的设置。例如:

```
SUBROUTINE TESTPROC( VALPARM, REFPARM )
```

```
!DEC$ ATTRIBUTES VALUE :: VALPARM
```

```

!DEC$ ATTRIBUTES REFERENCE :: REFPARM
INTEGER VALPARM
INTEGER REFPARM
...
END SUBROUTINE

```

将参数 VALPARM 规定为值传递, REFPARM 为引用传递。

在 C/C++中, 标量参数采取值传递, 数组参数采取引用传递。不同于 Fortran 的是, C/C++的调用约定不影响参数的传递方式。标量参数若要采取引用传递, 在 C 中只能采用指针参数; 在 C++中, 可以采用指针参数或引用参数。例如, 与上列 Fortran 子程序对应的 C/C++函数原型声明为:

C(.c 文件):

```
extern void __stdcall TESTPROC( int ValParm, int* RefParm );
```

C++(.cpp 文件):

```
extern "C" { void __stdcall TESTPROC( int ValParm, int& RefParm ); }
```

## (二)数据类型的对应

即便协调了两种语言的调用约定、命名约定和参数传递方式, 但若数据类型不对应, 混合编程仍不会获得成功。Fortran 和 C/C++ 的数据类型不完全一一对应, 但就数值计算常用到的整型和实型而言, 它们有着精确的对应关系。两种语言对应的整型、实型列于表 10-4。

表 10-4

Fortran 与 C/C++对应的数据类型

Fortran	C/C++
INTEGER(2)	short
INTEGER(4)	int,long
REAL(4)	float
REAL(8)	double

## 第二节 Fortran 与 C/C++的混合编译

下面给出一个 Fortran 与 C/C++混合编译的典型实例: 该实例包含一个外部例程, 其功能是对一个二维数组 $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ 每行元素的值进行累加, 数组每一维的大小通过参数传入, 累加和存放在一维数组参数中。

## 一、Fortran 程序调用 C/C++ 函数

Fortran 作主程序，须建立外部例程的接口块。在接口块中明确规定调用约定、目标例程名、参数传递方式及其数据类型：

```
PROGRAM Main
  IMPLICIT NONE
  INTEGER(4),PARAMETER          :: Row = 2, Col = 3
  REAL(8),DIMENSION(Row,Col)    :: Arr = (/1,4,2,5,3,6/)
  REAL(8),DIMENSION(Row)        :: RSum

  INTERFACE
    SUBROUTINE ArrRowSum(A,R,C,S)
      !DEC$ ATTRIBUTES STDCALL,ALIAS:'_ArrRowSum@16' :: ArrRowSum
      !DEC$ ATTRIBUTES VALUE :: R, C
      INTEGER(4)           :: R, C
      REAL(8),DIMENSION(R,C) :: A
      REAL(8),DIMENSION(R)  :: S
    END SUBROUTINE
  END INTERFACE

  CALL ArrRowSum(Arr,Row,Col,RSum)
  PRINT*, RSum
END PROGRAM
```

其中，调用约定采用 STDCALL，该约定也是 Win32 API 函数所采用的约定；长整型参数 R 和 C 代表二维数组的行、列数，在外部例程中不发生改变，故规定为值传递，各占 4 个字节；双精度实型数组参数 A 和 S 为引用传递，地址编码为长整型数，占 4 个字节，整个参数列表占 16 个字节；用别名属性(ALIAS)限定目标例程名为大小写混合形式：\_ArrRowSum@16。

C 语言编写的外部函数(.c 文件)为：

```
#define ROW 2
```

---

```

void __stdcall ArrRowSum(double Arr[][ROW], int Row,
int Col,double RSum[])
{
    int i,j;

    for(j = 0; j < Row; j++)
    {
        RSum[j] = 0.0;
        for(i = 0; i < Col; i++)
            RSum[j] =RSum[j] + Arr[i][j];
    }
}

```

---

C/C++缺省的调用约定为\_\_cdecl, 为和 Fortran 的调用约定保持一致, 在外部函数定义中须添加\_\_stdcall 约定。

Fortran 与 C/C++在数组上的不同之处在于: Fortran 多维数组按列主存储, 元素的下标由 1 开始; C/C++多维数组则按行主存储, 元素的下标由 0 开始。故此, 需变换 C/C++的二维数组形状: 将 Fortran 数组的第一维变成 C/C++数组的第二维; Fortran 数组的第二维变成 C/C++数组的第一维。

C++语言编写的外部函数(.cpp 文件)为:

---

```

const int ROW = 2 ;

extern "C" void __stdcall ArrRowSum(
    double Arr[][ROW], int Row,int Col,double RSum[])
{
    for(int j = 0; j < Row; j++)
    {
        RSum[j] = 0.0;
        for(int i = 0; i < Col; i++)
            RSum[j] =RSum[j] + Arr[i][j];
    }
}

```

---

这里，在例程原型前添加了 C 链接选项：extern "C"，以消除 C++ 编译器为目标例程名添加的特定修饰。这样一来，Fortran 主程序和 C++ 例程的混合编译，就等同于 Fortran 主程序和 C 例程的混合编译。

## 二、C/C++ 程序调用 Fortran 例程

不管是用 C 还是 C++ 编写的主程序，Fortran 外部例程都是相同的：

---

```

SUBROUTINE ArrRowSum(Arr, Row, Col, RSum)
    IMPLICIT NONE
    !DEC$ ATTRIBUTES STDCALL, ALIAS: '_ArrRowSum@16' :: ArrRowSum
    !DEC$ ATTRIBUTES VALUE      :: Row, Col
    INTEGER(4)                  :: Row, Col
    REAL(8), DIMENSION(Row, Col) :: Arr
    REAL(8), DIMENSION(Row)     :: RSum
    INTEGER(4)                   :: I, J

    DO I = 1, Row
        RSum(I) = 0.0
        DO J = 1, Col
            RSum(I) = RSum(I) + Arr(I, J)
        END DO
    END DO
END SUBROUTINE

```

---

可在作为独立程序单元的外部例程中直接规定调用约定、目标例程名和参数传递方式，这些设置和上述 Fortran 主程序接口块的设置相同。

C/C++ 主程序要求调用约定采取缺省的 \_\_cdecl 约定，同时也允许为调用的外部函数规定不同的调用约定。下列为 C 语言编写的主程序(.c 文件)：

---

```

#include <stdio.h>
#define ROW 2

void main()
{

```

```
void __stdcall ArrRowSum(  
    double Arr[][ROW], int Row,int Col,double RSum[];  
    double r[ROW], a[][ROW] = {{1,4},{2,5},{3,6}};  
    int i;  
  
    for(i = 0; i < ROW; i++)  
    {  
        ArrRowSum(a, 2, 3, r);  
        printf("RSum[%d] = %f\n", i, r[i]);  
    }  
}
```

---

以下是 C++语言编写的主程序(.cpp 文件):

---

```
#include <stdio.h>  
const int ROW = 2 ;  
extern "C" void __stdcall ArrRowSum(  
    double Arr[][ROW], int Row,int Col,double RSum[]);  
  
void main()  
{  
    double r[ROW], a[][ROW] = {{1,4},{2,5},{3,6}};  
  
    for(int i = 0; i < ROW; i++)  
    {  
        ArrRowSum(a, 2, 3, r);  
        printf("RSum[%d] = %f\n", i, r[i]);  
    }  
}
```

---

外部函数原型声明有链接选项(extern "C")规定,必须将它放在全局作用域内,即在主函数的前面。

在语法功能上, C++可以采取带类型的常量声明取代 C 语言的字符宏定义;可以在结构块(如实例中的循环块)内声明局部变量,而 C 语言只能在函数的开头集中

声明局部变量。

### 第三节 Fortran 模块数据和例程的传递

传递大批量数据的最简便方式，是使用 Fortran 模块(Module)。另外，Fortran 模块例程也可被 C/C++调用；C/C++函数也可并入 Fortran 模块，作为其模块例程使用。但首先得弄清 Fortran 模块实体(数据和例程)的特殊命名规则。

#### 一、模块数据和例程命名

和其他外部实体的命名规则不同，模块实体命名采取下列形式：

`_MODULENAME_mp_ENTITY [ @stacksize ]`

`MODULENAME` 代表模块名，总是呈现大写形式；`ENTITY` 指模块数据和例程，缺省约定下为 `DATA`；`_mp_` 是模块与其实体之间的分隔符，总以小写形式出现；若模块实体为例程，`stacksize` 指参数表所占字节数。例如：

```
MODULE mymod
  INTEGER a
CONTAINS
  SUBROUTINE b (j)
    INTEGER j
    ...
  END SUBROUTINE
END MODULE
```

在 32 位系统下产生的目标文件中，模块实体名为：

```
_MYMOD_mp_A
_MYMOD_mp_B@4
```

和 Fortran 外部例程一样，调用约定对模块实体命名产生影响。见表 10-5 所列。

表 10-5 调用约定对模块实体命名的影响

Calling Convention	Data Entity	Procedure Entity
Default	<code>_MYMOD_mp_A</code>	<code>_MYMOD_mp_B@4</code>
C	<code>_MYMOD_mp_a</code>	<code>_MYMOD_mp_b</code>
STDCALL	Makes no sense	<code>_MYMOD_mp_b@4</code>



对模块数据规定 STDCALL 约定没有实际意义, 调用约定对模块例程参数传递方式的影响同外部例程的, 见表 10-1。

值得注意的是, 别名(ALIAS)属性对模块例程命名产生的特殊作用。例如:

```
!DEC$ ATTRIBUTES ALIAS : '_Routine' :: ROUTINE
```

产生的目标模块例程名为\_Routine, 前面不再添加模块名和分隔符。

## 二、C/C++访问模块数据和例程

实例: 假设一个学生派生类型或结构体由学号和成绩构成, 要求找出最高分的学生, 并算出该班学生的平均分。

我们设计一个 Fortran 模块, 当中包含一模块例程(函数)和一模块变量(代表平均分), C/C++主程序将一个班级的学生成绩作为实参传入模块函数, 该函数返回最高分的学生, 并将算得的平均成绩存入模块变量, 供 C/C++访问。下列为 Fortran 模块定义:

```
MODULE Examp
```

```
    IMPLICIT NONE
```

```
    REAL :: Aver
```

```
    TYPE Student
```

```
        SEQUENCE
```

```
        INTEGER N
```

```
        REAL Mark
```

```
    END TYPE Student
```

```
CONTAINS
```

```
    FUNCTION FindMax(Stud,M)
```

```
        !DEC$ ATTRIBUTES C, ALIAS:'_FindMax':: FindMax
```

```
        INTEGER M
```

```
        TYPE(Student) Stud(M), FINDMAX
```

```
        INTEGER :: I, J = 0
```

```
        REAL    :: T = 0.0, S = 0.0
```

```
        DO I = 1, M
```

```
            IF(T <= Stud(I)%MARK) THEN
```

---

```

        J = Stud(I)%N
        T = Stud(I)%Mark
    END IF
END DO
FINDMAX % N = J
FINDMAX % Mark = T

DO I = 1, M
    S = S + Stud(I).Mark
END DO
Aver = S/M
END FUNCTION
END MODULE Examp

```

---

模块函数采用 C 调用约定, 和 C/C++ 的缺省约定 `__cdecl` 相匹配, 数组大小参数 `M` 为值传递, 数组参数 `Stud` 则采取引用传递; 别名 (ALIAS) 属性将模块函数目标名限定为 `_FindMax`。

访问 Fortran 模块的 C 主程序 (.c) 为:

---

```

#include <stdio.h>

extern float EXAMP_mp_AVER;
extern struct EXAMP_mp_STUDENT{
    int N;
    float Mark;
};
#define STU struct EXAMP_mp_STUDENT
extern STU FindMax(STU s[],int n);

void main()
{
    STU m,s[3]={{1001,70},{1002,80},{1003,90}};

    m = FindMax(s,3);
}

```

```
printf("%d:%f\n",m.N,m.Mark);  
printf("%f\n",EXAMP_mp_AVER);  
}
```

C++主程序(.cpp)为:

```
#include <stdio.h>  
  
extern "C" float EXAMP_mp_AVER;  
extern "C" struct EXAMP_mp_STUDENT{  
    int N;  
    float Mark;  
};  
#define STU struct EXAMP_mp_STUDENT  
extern "C" STU FindMax(STU s[],int n);  
  
void main()  
{  
    STU m,s[3]={{1001,70},{1002,80},{1003,90}};  
  
    m = FindMax(s,3);  
    printf("%d:%f\n",m.N,m.Mark);  
    printf("%f\n",EXAMP_mp_AVER);  
}
```

### 三、C/C++函数并入模块

下面的实例用 C/C++实现求平方根的函数，结果保存在指针或引用参数中。

C 函数(.c):

```
#include <math.h>  
extern void c_sqrt (float a, float b, float* c)  
{  
    *c = (float) sqrt(a*a + b*b);  
}
```

---

```
}
```

---

C++函数(.cpp):

---

```
#include <math.h>
extern "C" void c_sqrt (float a, float b, float& c)
{
    c = (float) sqrt(a*a + b*b);
}
```

---

在 C++中，用简便的引用参数代替指针参数，其实质都是地址传递。

通过在 Fortran 90 模块程序单元中建立其接口块，将 C/C++函数作为模块例程使用。这里，接口块的作用类似于 C++中的头文件：声明例程原型，真正的例程由 C/C++给出。下列为 Fortran 模块及使用模块的主程序：

---

```
MODULE Cproc
    IMPLICIT NONE
    INTERFACE
        SUBROUTINE C_Sqrt (a, b, res)
            !DEC$ ATTRIBUTES C :: C_Sqrt
            !DEC$ ATTRIBUTES REFERENCE :: res
            REAL a, b, res
        END SUBROUTINE
    END INTERFACE
END MODULE

PROGRAM Main
    USE CPROC
    IMPLICIT NONE
    REAL X

    CALL C_Sqrt (3.0, 4.0, X)
    PRINT*,X

END PROGRAM
```

---

在接口块中规定了和 C/C++缺省约定(\_\_cdecl)相匹配的 C 调用约定, 目标例程名统一被转换为小写, 参数传递规定为值传递; 再用 REFERENCE 属性限定引用传递的参数。

### 小 结

- 在 Visual Fortran 6.x/Visual C++ 6.0 环境下进行 Fortran 与 C/C++的混合编译, 即在源代码级上的混合编程是完全可行的。但要使混合编程获得成功, 须全面一致地协调两种语言的调用约定, 包括堆栈管理、目标例程命名、参数传递方式及其数据类型。
- Fortran 的调用约定有缺省约定、C、C REFERENCE、STDCALL 和 STDCALL REFERENCE; C/C++的调用约定有 \_\_cdecl(缺省)和 \_\_stdcall。Fortran 的调用约定既影响堆栈管理、目标例程命名, 又影响参数传递方式; C/C++的调用约定不影响参数传递方式。
- Fortran 的缺省约定、STDCALL 与 C/C++的 \_\_stdcall 约定相匹配, C 约定则与 \_\_cdecl 匹配; Fortran 的调用约定将目标例程名转换为大写或小写, 要保持大、小写混合须使用别名属性加以限定, C 的目标例程名保持大、小写混合, C++的须使用链接选项去掉名称修饰; 在参数传递上, Fortran 缺省为引用传递, C/C++为值传递, Fortran 可通过 VALUE、REFERENCE 对参数传递加以规定, C/C++要传递引用参数须使用指针或引用, 数组参数在两种语言中只能是引用传递。
- Fortran 程序调用 C/C++函数, 须建立其接口块, 以规定调用约定、目标例程名和参数传递, C++函数原型须添加链接选项来消除其名称修饰; C/C++程序调用 Fortran 外部例程, 直接在例程中规定调用约定、目标例程名和参数传递, C++中带链接选项的函数原型声明须放在全局作用域内。
- C/C++程序可直接访问 Fortran 模块中的数据和例程, 但模块数据和例程具有特殊的命名规则; 通过在 Fortran 模块中建立 C/C++函数的接口块, 可以将 C/C++函数并入模块, 作为模块例程来使用。

# 第十一章 Fortran 与 C/C++混成 DLL 并集成到 Win32 应用程序

Fortran 与 C/C++同属编译型语言,又有相同的集成开发环境 Visual Fortran 6.x/ Visual C++ 6.0,可以混合编译、链接成一个 EXE 应用程序。但作为 Windows 快速应用开发工具的 Visual Basic 和 Delphi,和 Fortran、C/C++差异较大,不能直接调用其计算程序。但可以在 Visual Fortran 6.x/ Visual C++ 6.0 集成开发环境中,将 Fortran、C/C++混合编译、链接成动态链接库(Win32 DLL),再在 Visual Basic 和 Delphi 中对其实施调用。

## 第一节 动态链接库 DLL

动态链接库 DLL(Dynamic-Link Libraries)是 Windows 操作系统特有的产物,Windows 系统的好多功能是由 DLL 实现的,Windows 的历次版本升级主要是通过更新这些 DLL 完成的。

DLL 是包含函数和数据的可执行模块,运行时由调用程序(EXE 或 DLL)将其加载到应用程序(调用进程)的地址空间。DLL 中的函数分为外部函数和内部函数:外部函数用于输出,并由外部程序调用;内部函数只能在定义的 DLL 内使用。尽管 DLL 可以输出数据,但它的的功能通常由 DLL 函数使用。

DLL 提供了一种特有的模块化应用程序开发方式,并通过 DLL 函数实现功能的更新和代码重用。同时,由于 DLL 和使用它的应用程序分开存放,应用程序可以保持较小的规模;当几个应用程序共用一个 DLL 时,内存中只保留 DLL 的一份拷贝,几个应用程序共享这份拷贝,因而可以有效地节省内存空间。

当前的 32 位 Windows 操作系统提供的编程接口 API(Application Programming Interface),就是由多个 DLL 实现的。进行 Windows 应用编程时,经常需要直接使用 Win32 API 中的一些函数,来加快应用程序的开发。

Visual Fortran 6.x/ Visual C++ 6.0 也对 Win32 DLL 提供了支持,可以在该环境下,将 Fortran、C/C++或两者混合建成类似于 Win32 API 的 DLL。

## 第二节 Fortran 与 C/C++混成 Win32 DLL

Fortran 与 C/C++混成 Win32 DLL,首先要解决它们的混合编译问题,包括全面

一致地协调它们使用的调用约定、命名约定、参数传递等，然后再导出 C/C++外部函数或 Fortran 外部例程，并编译链接成 DLL。

Fortran 与 C/C++的混合编译在第十章已进行了较为详细的探讨。下面先考查 Visual Fortran 6.x 和 Visual C++ 6.0 中的外部例程导出方式，然后再探讨单一语言和混合语言编写外部例程的 DLL 创建问题。

### 一、DLL 的导出方式

DLL 文件结构类似于 EXE 文件，但有一个重要的不同：DLL 文件包含一个输出表。输出表列出了 DLL 向外部程序输出的函数，这些函数是 DLL 的入口点，只有输出表列出的函数才能由外部程序访问；输出表没有列出的函数只能在 DLL 内使用。

使用 Visual C++ 6.0 自带的工具 DUMPBIN，结合/EXPORTS 开关，可以对 DLL 的输出表进行察看。

Visual C++ 6.0 和 Visual Fortran 6.x 编译器在导出 DLL 的外部例程时，使用了下列不同的语法格式。

#### (一)Visual C++ 6.0

Visual C++ 6.0 提供了两种方式来导出外部函数。

##### 1. 模块定义文件(.DEF)

编写一个模块定义文件，列出要导出的外部函数，当创建 DLL 时将该文件插入 DLL 工程。当通过序号而不是名称导出外部函数时，选择这种方式。

##### 2. \_\_declspec(dllexport)关键字

在要导出的外部函数原型前添加 \_\_declspec(dllexport)关键字。这种方式简洁明快，下面的 Visual C++导出实例一律使用这种方式。

#### (二)Visual Fortran 6.x

Visual Fortran 6.x 使用导出(DLLEXPORT)属性声明方式。例如：

```
!DEC$ ATTRIBUTES DLLEXPORT :: RoutineName
```

导出命名为 RoutineName 的外部例程。

不论是 Visual C++ 6.0 还是 Visual Fortran 6.x，导出外部例程时都应优先选择标准约定(\_\_stdcall/STDCALL)。因为 Win32 API 函数采用的就是这种约定，Windows 快速应用开发工具(Visual C++、Visual Basic、Delphi 等)在加载 Win32 DLL 时默认的也是这种约定。

### 二、由单一语言例程建成 DLL

在 Visual Fortran 6.x/Visual C++ 6.0 中创建 DLL 工程时，可以选择 Visual Fortran

6.x 的 Fortran Dynamic Link Library, 也可以是 Visual C++ 6.0 的 Win32 Dynamic-Link Library 应用类型。在选择 Visual C++ 6.0 的 Win32 DLL 时, 也不需要添加 DLL 入口点函数DllMain, 因为一般的数值计算例程不需要进行初始化和内存清理操作。

既然 Visual Fortran 6.x/Visual C++ 6.0 集成了 Fortran 和 C/C++ 两种编译器, 那么我们可以在一个 DLL 工程中, 同时添加 Fortran 例程、C/C++ 函数, 并导出各自的外部例程/函数。

例 11-1 分别用 Fortran 例程、C/C++ 函数实现  $c = \sqrt{a^2 + b^2}$  的算法, 其结果通过第三个参数输出。将三个例程混合编译链接成一个 Win32 DLL。

[例 11-1] 由单一语言例程建成 DLL。

Fortran 90 外部例程(.f90):

---

```

SUBROUTINE FSqrt (a, b, c)
    IMPLICIT NONE
    !DEC$ ATTRIBUTES DLLEXPORT, STDCALL :: FSqrt
    !DEC$ ATTRIBUTES VALUE      :: a,b
    !DEC$ ATTRIBUTES REFERENCE :: c
    REAL a,b,c
    c = sqrt(a*a + b*b)
END SUBROUTINE

```

---

在 Fortran 90 外部例程中, 采用了 DLLEXPORT 导出属性声明; 调用约定规定为 STDCALL; 在例程内不改变值的参数 a、b 规定为值传递, 返回结果的参数 c 规定为引用传递; 目标例程为 \_fsqrt@12, 同时还导出了另一个目标例程 fsqrt。

这里出现了一个特殊现象, 即导出属性 DLLEXPORT 对要输出的目标例程名产生影响。现将 DLLEXPORT 属性下不同调用约定产生的影响列于表 11-1(以 FSqrt 例程为例)。

表 11-1 DLLEXPORT 属性下不同调用约定的例程命名

调用约定(, DLLEXPORT)	目标例程
缺省	_FSQRT@12, FSQRT
C	fsqrt
STDCALL	_fsqrt@12, fsqrt

从表 11-1 中可以看出, 不论在哪一种约定下, 总产生一个和源代码例程名相同



的目标例程，并依据不同的调用约定，统一转换为大写或小写。但如果添加别名 (ALIAS)，则只产生一份以别名命名的目标例程。例如，将该例程重新声明为：

```
!DEC$ ATTRIBUTES DLLEXPORT, STDCALL, ALIAS:'_FSqrt@12' :: FSqrt
```

其惟一的一份目标例程为\_FSqrt@12，既符合了 STDCALL 约定的例程命名规则，又保持了大小写混合形式。

C 外部函数(.c):

---

```
#include <math.h>
extern __declspec(dllexport)
void __stdcall CSqrt (float a, float b, float* c)
{
    *c = (float) sqrt(a*a + b*b);
}
```

---

C++外部函数(.cpp):

---

```
#include <math.h>
extern "C" __declspec(dllexport)
void __stdcall CppSqrt (float a, float b, float& c)
{
    c = (float) sqrt(a*a + b*b);
}
```

---

在 C/C++外部函数中，使用了\_\_declspec(dllexport)关键字导出函数；调用约定采用\_\_stdcall；参数 a、b 为值传递，参数 c 为指针/引用传递；目标函数名分别为\_CSqrt@12 和\_CppSqrt@12。

这样整个 Win32 DLL 导出了三个外部例程，分别用 Fortran、C/C++加以实现。

### 三、由混合语言例程建成 DLL

设想在编写 Fortran 或 C/C++例程时，部分功能已由现成的 C/C++或 Fortran 例程实现了，此时就可以在 Fortran 或 C/C++例程中直接调用另一种语言编写的例程。这种情形如同 C++中的虚函数：子类的虚函数可以在基类虚函数的基础上进行功能扩充。

这里的混合语言外部例程(导出例程)，其实现体内调用了另一种语言编写的例

程。下面分 Fortran 例程调用 C/C++函数和 C/C++函数调用 Fortran 例程两种情况进行探讨。

### (一)Fortran 导出例程调用 C/C++函数

DLL 导出的是 Fortran 外部例程，在该例程内调用了 C/C++函数。见例 11-2 所示。

[例 11-2] Fortran 导出例程调用 C/C++函数。

Fortran 90 外部例程(.f90):

---

```

SUBROUTINE F_Call_CSqrt (a, b, c)
  IMPLICIT NONE
  !DEC$ ATTRIBUTES DLLEXPORT, STDCALL,
    ALIAS: '_F_Call_CSqrt@12' :: F_Call_CSqrt
  !DEC$ ATTRIBUTES VALUE :: a,b
  !DEC$ ATTRIBUTES REFERENCE :: c
  REAL a,b,c

  INTERFACE
    SUBROUTINE C_Sqrt(v1,v2,r)
      !DEC$ ATTRIBUTES C, ALIAS: '_C_Sqrt' :: C_Sqrt
      !DEC$ ATTRIBUTES VALUE :: v1,v2
      !DEC$ ATTRIBUTES REFERENCE :: r
      REAL v1,v2,r
    END SUBROUTINE
  END INTERFACE
  CALL C_Sqrt(a,b,c)
END SUBROUTINE

```

---

该 Fortran 90 外部例程中建立了两部分接口：前一部分接口针对要导出的外部例程本身，调用约定采用了 STDCALL 约定；后一部分接口(块)则针对外部例程要调用的 C/C++函数，调用约定采用了 C 约定。

外部例程实现体只对 C/C++函数实施调用。实际编程中，可以根据需要在调用前后增加一些特定的算法语句。被调用的 C 函数(.c):

---

```
#include <math.h>
```

---

```
extern void C_Sqrt (float a, float b, float* c)
{
    *c = (float) sqrt(a*a + b*b);
}
```

---

被调用的 C++函数(.cpp):

---

```
#include <math.h>
extern "C" void C_Sqrt (float a, float b, float& c)
{
    c = (float) sqrt(a*a + b*b);
}
```

---

这里为了方便说明, 将 C 和 C++函数取相同的名称, 并编写了一个 Fortran 外部例程。实际编程中, 一个工程内不允许出现同名的外部例程。

### (二)C/C++函数调用 Fortran 例程

DLL 要导出的是 C/C++外部函数, 在外部函数内调用了 Fortran 例程。从第十章可知, C/C++还可访问 Fortran 90 模块中的数据和例程, 这里的 Fortran 例程包括外部例程和模块例程, 分别见例 11-3 和例 11-4 所示。

[例 11-3] C 导出函数调用 Fortran 外部例程。

C 导出函数(.c):

---

```
#include <math.h>

extern void F_Sqrt (float v1, float v2, float* r);

extern __declspec(dllexport)
void __stdcall C_Call_FSqrt (float a, float b, float* c)
{
    F_Sqrt ( a,  b,  c);
}
```

---

其中, 先声明了要调用的 Fortran 外部例程原型, 使用缺省的调用约定(\_\_cdecl); 要导出的外部函数添加了\_\_declspec(dllexport)关键字, 并使用\_\_stdcall 约定, 在其实

---

现体内对 Fortran 外部例程实施调用。被调用的 Fortran 外部例程(.f90):

---

```

SUBROUTINE F_Sqrt (v1, v2, r)
  IMPLICIT NONE
  !DEC$ ATTRIBUTES C, ALIAS: '_F_Sqrt'      :: F_Sqrt
  !DEC$ ATTRIBUTES VALUE                     :: v1, v2
  !DEC$ ATTRIBUTES REFERENCE                 :: r
  REAL v1, v2, r

  r = sqrt(v1 * v1 + v2 * v2)
END SUBROUTINE

```

---

**[例 11-4]** C++导出函数调用 Fortran 模块例程。

C++导出函数(.cpp):

---

```

#include <math.h>

extern "C" void F_Sqrt (float v1, float v2, float& r);

extern "C" __declspec(dllexport)
void __stdcall Cpp_Call_FSqrt (float a, float b, float& c)
{
    F_Sqrt(a, b, c);
}

```

---

在 C++外部函数中，对要调用的函数和导出的函数都添加了 C 链接选项，以消除 C++特定的命名修饰。

Fortran 模块例程(.f90):

---

```

MODULE SqrtMod
  IMPLICIT NONE

  CONTAINS
    SUBROUTINE F_Sqrt (v1, v2, r)

```

```

      !DEC$ ATTRIBUTES C, ALIAS:'_F_Sqrt' :: F_Sqrt
      !DEC$ ATTRIBUTES VALUE          :: v1, v2
      !DEC$ ATTRIBUTES REFERENCE      :: r
      REAL v1, v2, r
      r = sqrt(v1 * v1 + v2 * v2)

      END SUBROUTINE

      END MODULE

```

在模块例程中，用别名(ALIAS)属性限定编译产生的目标例程名，以覆盖模块例程的特殊命名，简化 C++中的函数原型声明。由于 C++使用了缺省的调用约定(\_\_cdecl)，故这里使用了与之匹配的 C 约定。

#### 四、典型实例

前面探讨了 DLL 集成 Fortran、C/C++单一语言编写的外部例程和混合例程。这里，从数值计算经常要用到数组的实际情况出发，选取第十章混合编译的典型实例，用 Fortran 编写外部例程、集成到 DLL 中(见例 11-5)，并作为在 Visual C++、Visual Basic 及 Delphi 中调用的典型实例。

[例 11-5] Fortran 数组传递导出例程。

```

SUBROUTINE FRowSum(Arr, Row, Col, RSum)
  IMPLICIT NONE
  !DEC$ ATTRIBUTES DLLEXPORT,STDCALL,ALIAS:'_FRowSum@16' ::
FRowSum
  !DEC$ ATTRIBUTES VALUE          :: Row, Col
  !DEC$ ATTRIBUTES REFERENCE      :: Arr, RSum

  INTEGER(4)                      :: Row,Col,I,J
  REAL(8), DIMENSION(Row,Col)    :: Arr
  REAL(8), DIMENSION(Row)         :: RSum

  DO I = 1, Row
    RSum(I)=0.0
    DO J = 1, Col
      RSum(I) =RSum(I) + Arr(I,J)

```

```

        END DO
    END DO
END SUBROUTINE

```

在 Fortran 导出例程的属性声明中, 惟一必须的设置是 `DLL_EXPORT`; 调用约定规定为 `STDCALL`, 是为了和 Win32 API 的调用约定相统一; 别名(`ALIAS`)属性声明, 旨在 `STDCALL` 约定命名基础上保持大小写混合, 并限定只产生一份目标例程; 参数传递属性声明只是为了醒目, 其实在 `STDCALL` 约定下标量参数原本就是值传递, 而数组参数只能以引用方式传递。

### 第三节 DLL 例程在 Win32 应用程序中的集成

当前 Win32 应用程序的快速开发工具主要是 Visual C++ 6.0、Visual Basic 6.0 及 Delphi 7.0, 故此我们具体探讨 DLL 例程在这些开发工具中的集成问题。在集成前, 先应弄清楚 DLL 例程的实现机制、调用约定、参数数据类型及传递方式, 然后才可能声明正确的 DLL 例程原型, 顺利链接或加载 DLL, 对 DLL 例程实施调用。

#### 一、例程、调用约定及参数的对应

上述 DLL 例程混合了 Fortran、C/C++ 语言, 下面以 Fortran 例程作为 DLL 例程, 探讨其在例程类型、调用约定、参数的数据类型及传递方式上, 与 Visual C++、Visual Basic 及 Delphi 的对应关系。

##### (一) 例程类型

在例程实现机制方面, 这几种语言是相同的。子程序名没有返回值, 子程序的算法结果通过参数返回, 所以子程序的参数表中必有引用参数; 函数名有返回值, 其算法结果通过函数名返回, 其参数表中的参数通常为值参数, 即参数值在函数实现体内不发生改变。

在声明 DLL 例程原型时, 例程类型必须对应(见表 11-2)。

表 11-2

Fortran 与其他语言中的例程

语言	子程序	函数
Fortran	SUBROUTINE	FUNCTION
Visual C++	(void)function	function
Visual Basic	Sub	Function
Delphi	Procedure	Function

## (二)调用约定

调用约定匹配是混合编程获得成功的一个首要条件。在 Visual Fortran 6.x 编译器下, Fortran 调用约定有缺省约定、C 约定和 STDCALL 约定。这几个约定都是利用堆栈传递参数,进栈时,例程参数从左到右依次压入堆栈;出栈时,例程参数从右到左依次从堆栈中弹出。在 C 约定下,由调用程序负责堆栈管理,因此可传递可变数量的参数(可选参数);在缺省约定和 STDCALL 约定下,由被调用例程负责堆栈管理。

Visual C++、Visual Basic 及 Delphi 与 Fortran 匹配的调用约定列于表 11-3。

表 11-3 Fortran 与其他语言匹配的调用约定

Fortran	Visual C++	Visual Basic	Delphi
Default, STDCALL	__stdcall	Default	Stdcall
C	__cdecl	No	Cdecl

按理来说, Visual Basic 与 Fortran 的 C 约定匹配的应为 Cdecl 约定,但试验证明这不成立。因此, DLL 输出例程应使用标准约定 STDCALL。

## (三)参数类型

参数的数据类型不对应,有可能在 DLL 例程调用时产生错误的结果。这几种语言的数据类型不完全一一对应,但就数值计算常用到的整型和实型而言,它们之间却存在着精确的对应关系,见表 11-4 所示。

表 11-4 Fortran 与其他语言对应的整型和实型

Fortran	Visual C++	Visual Basic	Delphi
INTEGER(2)	short	Integer	SmallInt
INTEGER(4), INTEGER	int, long	Long	LongInt, Integer
REAL(4), REAL	float	Single	Single
REAL(8)	double	Double	Double, Real

## (四)参数传递方式

参数传递分值传递和引用传递两种方式。值传递,是传递参数的值,例程内虚参值发生改变不影响实参;引用传递,是传递参数存储单元的地址,虚参值的改变直接影响实参。不同语言的默认参数传递方式也不同, Fortran 和 Visual Basic 默认的为引用传递; Visual C++和 Delphi 则为值传递,见表 11-5 所列。

表 11-5 不同语言标量参数的传递方式

语言 [ 属性规定 ]		值传递	引用传递
Fortran	No[Reference]		✓
	[Value]	✓	
Visual C++	No	✓	
	[*,&]		✓
Visual Basic	No[ByRef]		✓
	[ByVal]	✓	
Delphi	No[const]	✓	
	[var,out]		✓

Fortran 的调用约定还对参数传递产生影响：缺省约定下，参数以引用方式传递；C 和 STDCALL 约定下，参数以值方式传递。为使参数传递不受调用约定的影响，可为参数单独规定 VALUE 和 REFERENCE 传递属性，使参数分别以值和引用方式传递。

数组参数情况比较特殊，除了存储方式和元素起始下标不同外，数组都是以引用方式传递数组首地址，即第一个元素的地址。如果 Fortran 例程参数为数组，相应地在 Visual C++ 中可以声明数组参数，也可声明指针参数；但在 Visual Basic 和 Delphi 中，只能声明引用参数，并传递数组的第一个元素。

## 二、Visual C++

Visual C++ 要调用 DLL 例程，先要链接 DLL。链接 DLL 的方式有两种：隐式链接和显式链接。

### (一) 隐式链接(Link Implicitly)

当 Windows 加载应用程序时，一起加载隐式链接的 DLL。大部分应用程序采取隐式链接，因为隐式链接比较容易。要隐式链接到 DLL，调用程序需要下列文件：

(1) 头文件(.h)。头文件包含 DLL 输出函数的原型声明，使用输出函数的每一个源文件(.cpp)都需包含头文件；假如只有一个源文件用到输出函数，可以省略头文件，而在该源文件中直接添加输出函数的原型声明。

(2) 输入库文件(.lib)。建造 DLL 的同时，生成输入库文件。链接程序借助于 LIB 文件将 DLL 和调用程序链接在一起。在 Visual C++ 6.0 下，需在 Project Settings dialog box/ Link/ Object/Library Modules 中，添加输入库文件，并将 LIB 文件存放在当前应用程序路径下。



(3)DLL 文件(.dll)。DLL 库文件包含输出函数的可执行代码,当 Windows 加载调用程序时,要能准确定位 DLL 文件。通常,将 DLL 文件存放在当前应用程序路径或系统路径下。

在 Visual C++ 6.0 下,编写的控制台应用程序(Win32 Console Application)为:

```
#include <iostream.h>
const int ROW = 2 ;
extern "C" void __stdcall FRowSum( //声明输出函数原型
    double Arr[][ROW], int Row,int Col,double RSum[]);

void main()
{
    double r[ROW], a[][ROW] = {{1,4},{2,5},{3,6}};

    for(int i = 0; i < ROW; i++)
    {
        FRowSum(a, 2, 3, r);
        cout<<"RSum["<<i<<" = "<<r[i]<<endl;
    }
}
```

声明输出函数原型时, Fortran 的数组参数也声明为对应类型的数组,但由于多维数组的存储方式不同,须将二维数组的行、列对调。

## (二)显式链接(Link Explicitly)

显式链接指运行时当应用程序调用 DLL 输出函数时,Windows 方加载、链接到 DLL,所以称为动态链接库。显式链接不需要输入库文件和头文件(或输出函数原型声明),但需要声明与输出函数对应的函数指针。显式链接需要采取以下三个步骤:

(1)调用 Win32 API 函数 LoadLibrary,以加载 DLL 并获得模块句柄。

(2)调用 Win32 API 函数 GetProcAddress,以获得输出函数地址。应用程序通过函数指针调用 DLL 输出函数,编译器不产生外部引用,所以没必要链接输入库。

(3)调用 Win32 API 函数 FreeLibrary,以释放模块句柄,将 DLL 从内存中清除。

显式链接实例为:

```
#include <iostream.h>
```

```

#include <windows.h>

const int ROW = 2 ;
typedef void (__stdcall* LPFN)(          //声明输出函数指针
    double Arr[][ROW], int Row,int Col,double RSum[]);

void main()
{
    double    r[ROW], a[][ROW] = {{1,4},{2,5},{3,6}};
    HINSTANCE hDLL;           // DLL 句柄
    LPFN      lpfn;           // 函数指针

    hDLL = LoadLibrary("FArrPass");
    if (hDLL)
    {
        lpfn = (LPFN)GetProcAddress (hDLL, "_FRowSum@16");
        if(lpfn)
        {
            for(int i = 0; i < ROW; i++)
            {
                (*lpfn) (a, 2, 3, r);
                cout<<"RSum["<<i<<" = "<<r[i]<<endl;
            }
        }
        FreeLibrary (hDLL);
    }
}

```

值得注意的是：GetProcAddress 函数的第二个参数指 DLL 入口点，必须和输出函数的目标名完全一致。

### 三、Visual Basic

Visual Basic 6.0 使用 DLL 例程，须先声明例程原型，再在程序中直接调用。

声明例程原型的位置，可以是调用例程的窗体(.frm)，也可以是模块(.bas)。在

窗体中声明私有例程(Public);而在模块中要声明公有例程(模块缺省 Public)。这里选择在模块中声明:

```
Declare Sub FRowSum Lib "FArrPass.DLL" Alias "_FRowSum@16" ( _  
    ByRef Arr1 As Double, ByVal v1 As Long, ByVal v2 As Long, ByRef  
Arr2 As Double)
```

通常使用带别名(Alias)的例程声明语句。此处的别名指 DLL 的入口点,必须和 DLL 输出例程的目标名完全一致;例程(Sub 或 Function)名只是 Visual Basic 调用程序使用的例程标识符,可以随意命名。

当中的 DLL 文件通常置于系统路径下。若不是系统路径,声明中 DLL 须采用绝对路径。

例程中的数组参数,须声明同类型的标量引用参数,调用时实参取数组的第一个元素进行传递。

下列为调用 DLL 例程的窗体加载事件过程:

---

```
Private Sub Form_Load()  
    Dim Arr1(1, 2) As Double  
    Dim Arr2(1) As Double  
  
    Arr1(0, 0) = 1: Arr1(0, 1) = 2: Arr1(0, 2) = 3  
    Arr1(1, 0) = 4: Arr1(1, 1) = 5: Arr1(1, 2) = 6  
  
    Call FRowSum(Arr1(0, 0), 2, 3, Arr2(0))  
    Debug.Print Str(Arr2(0)) + vbLf + Str(Arr2(1))  
End Sub
```

---

#### 四、Delphi

Delphi 加载 DLL 的方式有两种:静态加载和动态加载。前者当应用程序运行时即加载 DLL;后者当调用 Windows API 的 LoadLibrary 函数时方加载 DLL,调用 FreeLibrary 函数来卸载 DLL。

##### (一)静态装载(Static loading)

静态加载比较简便,调用前先声明与 DLL 输出例程对应的外部例程。例如:

```
procedure FRowSum(var a:Double; b,c:Longint; var d:Double);Stdcall;  
external 'FArrPass.DLL' Name '_FRowSum@16';
```

Name 关键字后边的为 DLL 入口点的目标例程名, FRowSum 为调用时例程名,可

以随意命名；在 DLL 例程数组参数位置声明对应类型的变量参数；Delphi 缺省调用约定为 Register，故需显式声明 Stdcall 约定；DLL 文件通常置于当前应用路径下。

上述外部例程声明可直接放置于调用程序单元的 implementation 部分或公共程序单元的 interface 部分。

## (二)动态加载(Dynamic loading)

在 Visual C++中，通过函数指针调用 DLL 输出例程；而在 Delphi 中，则通过例程类型变量来调用。以下为窗体创建事件过程的完整代码：

---

```

procedure TForm1.FormCreate(Sender: TObject);
type
  T_SUB=procedure(var a:Double; b,c:Longint; var d:Double);stdcall;
var
  Sub:T_SUB;
  Handle:integer;
  Arr1,Arr2: array of Double;
begin
  SetLength(Arr1,2);
  SetLength(Arr2,6);
  Arr2[0]:=1;Arr2[1]:=4;Arr2[2]:=2;
  Arr2[3]:=5;Arr2[4]:=3;Arr2[5]:=6;

  Handle := LoadLibrary('FArrPass.DLL');
  if Handle <> 0 then
    begin
      @Sub:= GetProcAddress(Handle, '_FRowSum@16');
      if @Sub <> nil then
        begin
          Sub(Arr2[0],2,3,Arr1[0]);
        end;
        FreeLibrary(Handle);
      end;
      ShowMessage(FloatToStr(Arr1[0])+','+ FloatToStr(Arr1[1]));
    end;
end;

```

---

Delphi 的多维动态数组内部结构比较特殊, 对此应引起特别注意。比如, 声明二维双精度实型数组:

Arr2: array of array of Double;

使用前动态分配内存, 具体规定为  $3 \times 2$  的二维数组:

SetLength ( Arr2, 3, 2 );

若按行主存储, 其顺序应为:

Arr2[0,0], Arr2[0,1], Arr2[1,0], ... , Arr2[2,1]

事实并非如此, 所以不能像 Visual Basic 那样, 用第一个二维数组元素作引用实参。实例中采取的方法: 将 Fortran 的二维数组转换为 Delphi 的一维动态数组, 再用一维动态数组的第一个元素作引用实参。

### 小 结

- 在 Visual Fortran 6.x/Visual C++ 6.0 环境下进行 Fortran 与 C/C++的混合编译, 并链接成混合 DLL 是完全可行的。混合 DLL 可以包含 Fortran 例程和 C/C++函数, 也可以包含 Fortran 与 C/C++混合的例程。
- 混合 DLL 中的 Fortran 外部例程和 C/C++外部函数应分别进行导出: Fortran 外部例程使用 DLLEXPORT 属性, C/C++外部函数使用 \_\_declspec(dllexport) 关键字; Fortran 与 C/C++混合的外部例程, 还须全面一致地协调两种语言的例程类型、目标例程名、调用约定、参数传递方式及其数据类型。
- 建造 DLL 时, 应明确其输出例程的例程类型、目标例程名、调用约定、参数传递方式及其数据类型, 以便在 Visual C++、Visual Basic、Delphi 中声明例程原型、函数指针或例程变量时予以协调。针对 Visual C++中的集成, 还应准备与 DLL 对应的输入库文件(.lib)。
- Visual C++中的隐式链接, 需要声明带 C 链接选项的 DLL 例程原型, 并加入输入库; 显式链接, 需要声明相应的函数指针, 并以函数指针调用 DLL 例程。
- 在 Visual Basic 中声明 DLL 例程原型时, 例程名应与 DLL 例程目标名一致; 在数组参数位置声明同类型的引用参数, 调用时以数组第一个元素进行传递。
- Delphi 的静态加载要求声明 DLL 例程原型, 例程名须与 DLL 例程目标名一致; 动态加载要求声明 DLL 例程类型变量, 并以该变量调用 DLL 例程。在数组参数位置声明同类型的变量参数, 调用时将 Fortran 的二维数组转换为 Delphi 的一维动态数组, 再取一维动态数组的第一个元素进行传递。

## 第十二章 Fortran COM 组件的创建及其在客户程序中的调用

Visual Fortran 6.x 增加了一个重要功能：支持基于 COM 的组件开发。它通过提供 COM 服务器向导，来简化 COM 组件开发：开发者通过交互界面定义 COM 组件相关的类、接口、方法等，然后在生成的骨架文件中添加方法的实现代码即可；相关的底层工作，如创建类型库、类工厂、系统注册等，都由向导自动完成。

本章先简要介绍 COM 对象及组件的基本概念，然后通过一个典型实例来探讨 Visual Fortran 6.6 下 COM 组件的创建问题，再在 Visual C++ 6.0、Visual Basic 6.0 及 Delphi 7.0 下调用该 COM 组件。

### 第一节 COM 对象及组件

微软在开发 OLE 2.0 的过程中，从解决软件模块交互操作出发，定义了一个底层编程模型 COM(Component Object Model)，之后，微软以 COM 为基础平台，对 Windows 操作系统重新进行了架构。COM 作为一种基于对象的编程模型，旨在提高软件的互操作性，不同的开发者在不同的时间使用不同的开发工具开发的组件或组件软件，即使它们运行在不同的操作系统、平台下也可相互调用。

在 COM 下，组件与组件、组件与系统间的交互通过接口(一组成员函数或方法)来实现。接口反映了对象的特性，它由一组相关的方法构成，方法用来实现对象特性的具体功能。一个 COM 类或对象可以支持并实现多个 COM 接口(每个 COM 接口都从基接口 IUnknown 派生而来)。

COM 支持服务器/客户模型，COM 的服务提供者称为服务器，服务使用者称为客户。COM 服务器包含了若干 COM 对象，它可以实现为进程内服务器(DLL)，运行时被加载到应用程序的进程空间；也可以实现为进程外服务器(EXE)，运行在独立的进程，位于与应用程序相同或不同的操作系统，甚至是不同的计算机上。COM 服务器也称为 COM 组件。

值得注意的是：进程内 DLL 组件借助于代理(surrogate)程序，也可运行在与应用程序不同的进程中(Windows 提供的标准代理程序为 DLLHOST.EXE)。

COM 组件或服务拥有以下优点：

(1)COM 服务器是可重用的组件，可以被多个应用程序所共享；在保持组件接

口不变的前提下, 改变接口内部的方法实现, 不会影响使用组件的客户程序。

(2)COM 服务器独立于编程语言, 支持 COM 的任何工具都可创建和访问, 包括 Visual C++ 6.0、Visual Basic 6.0、Delphi 7.0 等。

(3)COM 服务器是自描述的。它使用类型库来描述组件类和接口, 支持 COM 的语言工具大多能够直接读取类型库中的信息, 不再需要了解像调用约定这样的底层细节。

(4)COM 服务器是自注册的。客户程序无需关注服务器位于何处, 因为 COM 运行库自动在系统注册表中定位 COM 服务器。

(5)COM 服务器的版本升级具有鲁棒性(robust versioning)。通过添加新的 COM 接口, 来产生新的组件版本。原有客户可以继续使用旧的接口, 新客户则可使用新接口提供的增强功能。

## 第二节 Fortran COM 组件的创建

为方便用户创建 COM 组件, Visual Fortran 6.x 提供了组件向导(Fortran COM Server Wizard), 该向导完成了创建组件的大部分任务:

- (1)定义组件类、接口使用的全球惟一标识符 GUID(Globally Unique Identifier);
- (2)向操作系统注册组件、类及其接口;
- (3)实现创建 COM 对象的类工厂;
- (4)创建用于描述组件类和接口的接口定义库(IDL)和类型库;
- (5)实现 COM 的基接口 IUnknown 或 IDispatch;
- (6)创建和销毁对象实例数据;
- (7)调用实现接口方法的 Fortran 例程。

用户所要做的: 利用向导提供的交互界面, 定义组件类、接口及其方法, 并在接口方法的实现文件中给出方法的实现体。

下面将第十一章中集成到 DLL 中的典型例程转换为 COM 组件方法, 并创建相应的 COM 组件。

### 一、定义组件类、接口及方法

在 Visual Fortran 6.6 集成开发环境中, 选择创建一个 Fortran COM Server 工程(ArrPassCOM)。在随后出现的交互界面(见图 12-1)中, 规定 COM 服务器的类型, 并设置其基本属性, 包括 COM 类、接口和类实例数据类型名。

上述设置完毕, 在 Fortran COM 服务器向导中展示 COM 组件的层次结构及对应的属性。

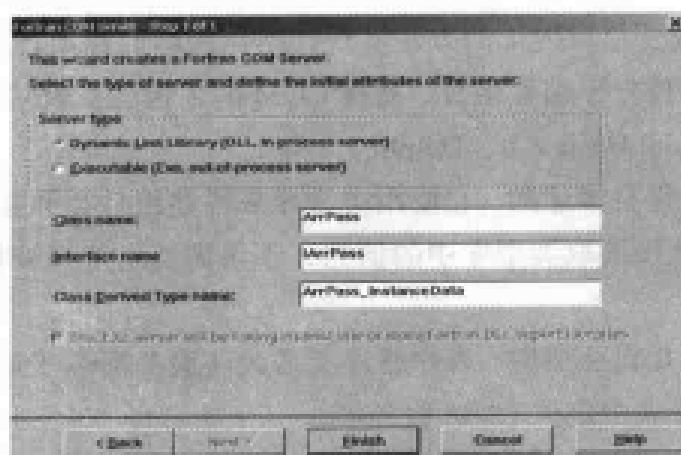


图 12-1 设置 COM 服务器类型及其基本属性

### (一) 组件工程

第一层 ArrPassCOM 为当前建造的 COM 组件工程(见图 12-2), 在右边属性板中对应的属性有类型库 GUID 和类型库版本。保留其缺省值。

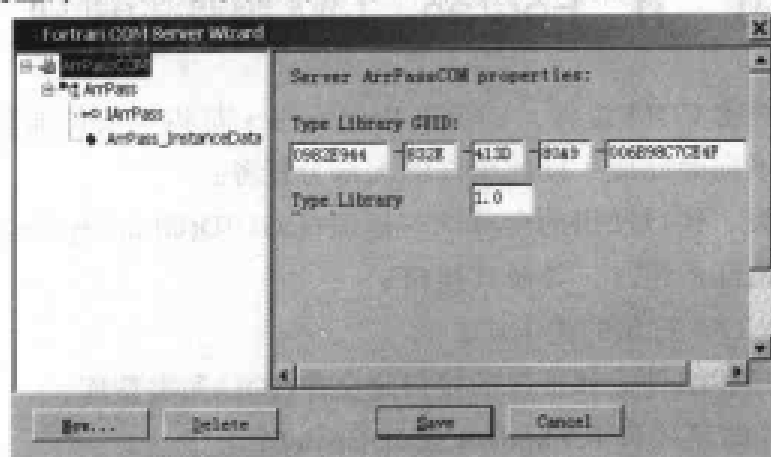


图 12-2 COM 组件属性

### (二) 组件类

第二层 ArrPass 为组件包含的 COM 类(见图 12-3), 其属性包括类的编程 ID(ProgID)、版本号、短名称、线程模型、类 ID(CLSID)等, 保留其缺省设置。

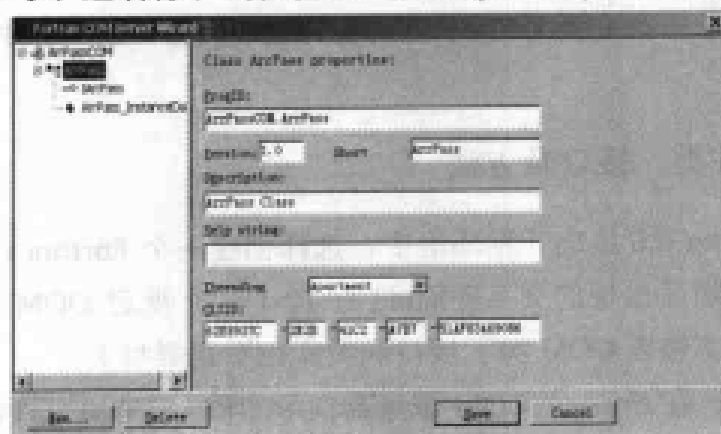


图 12-3 组件类属性



### (三)接口

第三层 IArrPass 为组件类要实现的接口(见图 12-4)。右边属性板中的第一个选项“Dual interface”(双接口),指 VTBL 接口和 IDispatch 接口(也称为自动化接口),分别对应编译时的早绑定和运行时的晚绑定。像 C++等支持 VTBL 接口的语言,可以采取早绑定而获得较高的执行速度;像 VBScript 等脚本语言,只能采取动态绑定(晚绑定)来访问 IDispatch 接口。在实际编程中,COM 接口大多实现为双接口。

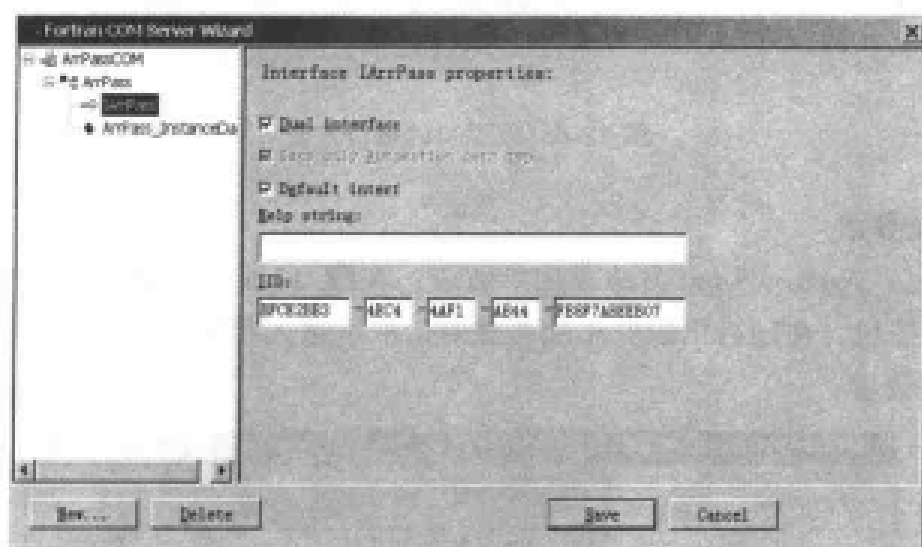


图 12-4 组件接口属性

双接口要求使用与自动化兼容的数据类型(Automation-compatible data types),所以当选取双接口时,第二个选项“Uses only Automation data types”自动被选取。

在 COM 的接口中,总有一个缺省(Default)接口,是客户程序首先要访问的接口。本实例的组件类只实现一个接口(IArrPass),同时它又是组件类的缺省接口。

与接口位于同一层次的 ArrPass\_InstanceData,为 COM 类的实例数据派生类型或结构体。当客户程序创建多个 COM 对象时,每个对象都有自己的实例数据。在本章实例中,这一项只是一个摆设,以满足组件工程编译要求。

### (四)方法

接下来为 IArrPass 接口添加一个方法:在图 12-4 的视图中选中接口(高亮显示),单击“New...”按钮,出现添加方法的对话框(见图 12-5),在该对话框中输入方法名 ArrRowSum。确定后,在向导视图接口下,显示出新添加的方法,如图 12-6 所示。

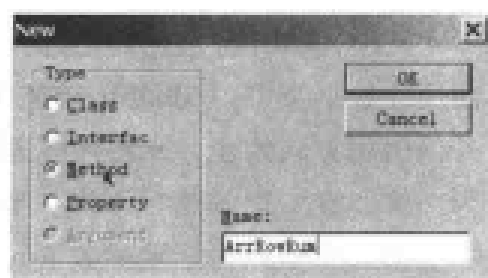


图 12-5 为接口添加方法

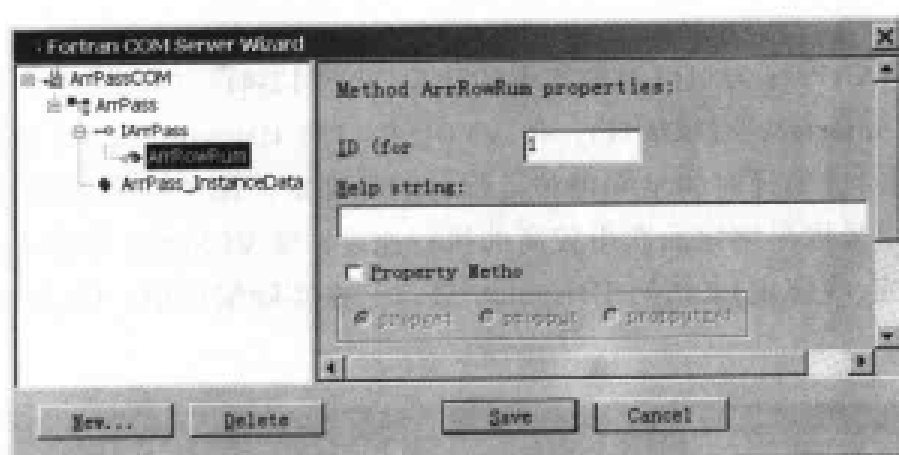


图 12-6 新添加的接口方法

### (五)方法参数

接下来为 ArrRowSum 方法添加参数表：在图 12-6 的视图中选中方法，单击“New...”按钮，在随后出现的添加参数对话框中输入参数名 Arr，并进行图 12-7 所示的属性设置。

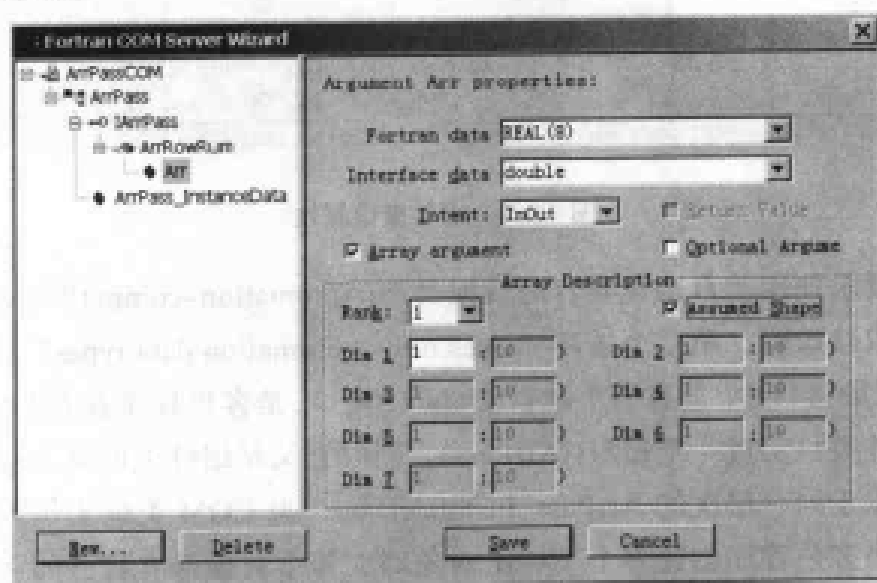


图 12-7 Arr 参数属性

其中，Fortran 数据类型选择 REAL(8)，对应接口数据类型(即接口描述语言 IDL 使用的数据类型)为 double；在 COM 双接口中，数组参数要求是安全数组(SafeArray)，并以引用方式传递，故此将传递属性 Intent 设为 InOut；为操作简便，将数组参数设为一维，在实现方法时再将一维转换为二维。

第二个参数为标量参数 Row，用来传入二维数组的行数，其值在方法内不应发生改变，故将其传递属性 Intent 设为 In，如图 12-8 所示。

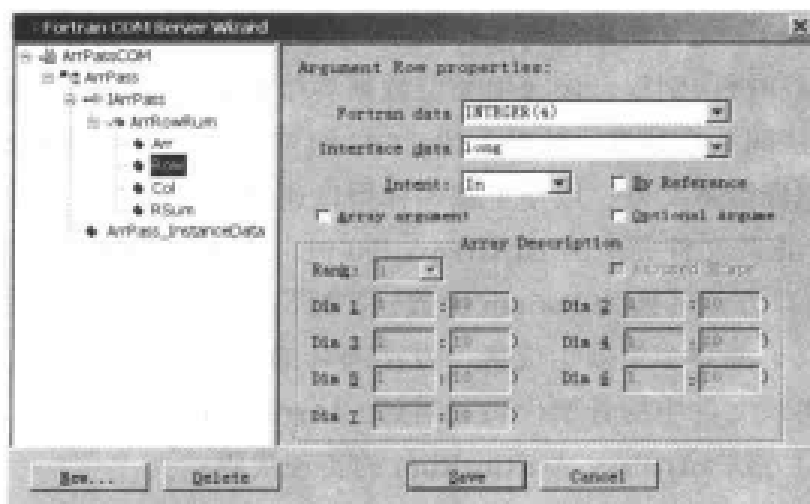


图 12-8 Row 参数属性

第三个参数为标量参数 Col，用来传入二维数组的列数，属性设置同第二个参数；第四个参数为数组参数 RSum，用来传递二维数组每行的和，属性设置同第一个参数。

## 二、实现接口方法

在点击向导中的“Save”按钮或关闭向导后，Visual Fortran 6.6 展示工程视图，同时打开 TODO.TXT 文件，该文件列出由向导生成并添加到工程中的文件。其中，需要用户编辑的文件放在“Source Files”文件夹中；每当用户在向导中修改组件属性，向导都会重新生成相关的文件，这些文件被放在“Do Not Edit”文件夹中。由向导组织的相关文件如图 12-9 所示。

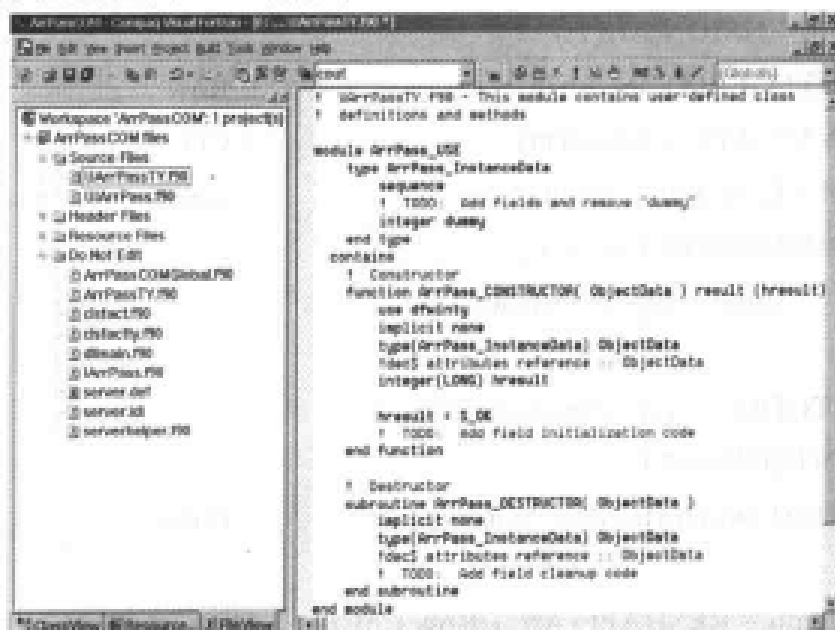


图 12-9 由向导组织的相关文件

“Source Files”文件夹中的 UIArrPassTY.f90 包含了一个派生类型(结构体)的定义及其构造例程和析构例程。该派生类型数据类似于 C++ 中类的数据成员，用于在 COM 对象的方法间共享数据。针对上述 COM 组件示例，派生类型数据没有实际作用，故不编辑但保留该文件，以使工程编译得以通过。

“Source Files”文件夹中的 UIArrPass.f90 包含了接口(IArrPass)方法的实现，向导已将 COM 接口方法转换为 Fortran 函数，并给出了函数骨架。假如在向导中改变接口方法的定义，向导会重新生成接口方法的骨架代码，并放置在 UIArrPass+.f90 文件中，需要将该文件中的骨架代码复制到 UIArrPass.f90 文件中。

下列为接口方法 IArrPass\_ArrRowSum 的实现代码：

---

```
! UIArrPass.f90--This file contains the implementation of the
!
!                               IArrPass methods
!

! IArrPass_ArrRowSum
function IArrPass_ArrRowSum( ObjectData ,&
    Arr, Row, Col, RSum) result (hresult)
    use ArrPass_Types
    implicit none
    type(ArrPass_InstanceData) ObjectData
    !dec$ attributes reference                :: ObjectData
    REAL(8), intent(inout)                   :: Arr
    DIMENSION Arr(1:)
    INTEGER(4), intent(in)                   :: Row
    INTEGER(4), intent(in)                   :: Col
    REAL(8), intent(inout)                   :: RSum
    DIMENSION RSum(1:)
    integer(LONG) hresult

    ! TODO:  Add implementation
    INTEGER(4) I,J
    REAL(8),DIMENSION(Row,Col)              ::Temp

    Temp = RESHAPE(Arr, (/Row,Col/))        !将一维数组转换为二维
    DO I = 1, Row
```

```

        RSum(I)=0.0
        DO J = 1, Col
            RSum(I) =RSum(I) + Temp(I,J)
        END DO
    END DO
! TODO:  End

    hresult = S_OK
end function

```

其中，! TODO 块为添加的具体算法，其他均为向导生成的骨架代码。

C++类的函数成员的第一个参数(隐式参数)为指向当前对象的 this 指针。这里，Fortran 函数的第一个参数为派生类型对象，以此将 Fortran 方法与 COM 对象相关联。

接口方法实现完毕，即可编译、链接组件工程，生成 DLL 文件、类型库文件，同时完成系统注册。其中，由 MIDL 工具编译生成类型库和调度代码的源文件为接口描述库文件：

```

// server.idl : IDL source the server

// This file will be processed by the MIDL tool to
// produce the type library and marshalling code.

import "oaidl.idl";
import "ocidl.idl";

// Interfaces for ArrPass
[
    object,
    uuid(BFCE2BB3-4EC4-4AF1-AB44-FB8F7ABEEB07),
    oleautomation ,
    dual
]
interface IArrPass : IDispatch {
    [id(1)] HRESULT ArrRowRum (

```

```
        [in, out] SAFEARRAY(double)* Arr,  
        [in] long Row,  
        [in] long Col,  
        [in, out] SAFEARRAY(double)* RSum);  
};  
  
// Type library defintion  
[  
    uuid(0982E944-832E-413D-80A9-006B98C7CE4F),  
    version(1.0),  
    helpstring("ArrPass 1.0 Type Library")  
]  
library ArrPassLib  
{  
    importlib("stdole32.tlb");  
    importlib("stdole2.tlb");  
  
    [  
        uuid(62E8937C-2E2B-41C2-A7B7-51AF83469086),  
  
    ]  
    coclass ArrPass  
    {  
        [default] interface IArrPass;  
    };  
};
```

---

当中的接口为双接口(dual)，并由 IDispatch 接口派生。Fortran 数组参数被转换成安全数组，并以指针传递。

### 第三节 COM 组件在客户程序中的调用

下面分别用 Visual C++ 6.0、Visual Basic 6.0 及 Delphi 7.0 建造客户程序，对上述 COM 组件实施调用。

### 一、Visual C++

当在 Visual Fortran 6.6 环境下建造 COM 组件工程时, MIDL 工具将 Server.idl 文件编译成类型库的同时, 还生成了 Server\_i.c 和 Server.h 文件, 这两个文件包含了组件类、接口的定义, Visual C++ 客户程序可以用它们创建组件对象。下列为 Visual C++ 控制台客户程序:

```
//VCClient.cpp
#include <iostream.h>
#include <windows.h>

//Guids, interface declarations, etc.
#include "..\ArrPassCOM\Server_i.c"
#include "..\ArrPassCOM\Server.h"

void main()
{
    cout<<"Initializing COM"<<endl;
    if(FAILED(CoInitialize(NULL)))
    {
        cout<<"Unable to initialize COM"<<endl;
        return;
    }

    //Create COM object, return pointer of interface
    IArrPass* pIArrPass;
    HRESULT hr = CoCreateInstance(CLSID_ArrPass,
                                  NULL,
                                  CLSCTX_INPROC,
                                  IID_IArrPass,
                                  (void**)&pIArrPass);

    if(FAILED(hr))
    {
        cout.setf(ios::hex,ios::basefield);
```

```

        cout<<"Failed to create server instance. HR = "<<hr<<endl;
        CoUninitialize();
        return;
    }
    cout<<"Instance created"<<endl;

    double r[2], a[] = {1,4,2,5,3,6};
    long ix[]={1,2,3,4,5,6};

    //Declare and create safe array
    SAFEARRAY FAR* psa[2];
    SAFEARRAYBOUND sabound[2];
    sabound[0].lLbound = 1;
    sabound[0].cElements = 6;
    sabound[1].lLbound = 1;
    sabound[1].cElements = 2;

    psa[0] = SafeArrayCreate(VT_R8, 1, &sabound[0]);
    psa[1] = SafeArrayCreate(VT_R8, 1, &sabound[1]);

    for(int i=0;i<6;i++)
        SafeArrayPutElement(psa[0], &ix[i], &a[i]);

    //Access interface
    pIArrPass->ArrRowRum(&psa[0], 2, 3,&psa[1]);

    for(int j=0;j<2;j++)
    {
        SafeArrayGetElement(psa[1], &ix[j], &r[j]);
        cout<<"RSum["<<j<<"] = "<<r[j]<<endl;
    }

    //Release interface
    if(pIArrPass)

```



```

plArrPass ->Release();

cout<<"Shutting down COM"<<endl;
CoUninitialize();
}

```

程序中，用 COM API 函数 CoCreateInstance 创建组件对象，并返回接口指针。通过接口指针访问接口方法。

Fortran 组件向导将接口方法中的数组参数转换为安全数组，其数组下界、维数以及元素个数都是明确的，这是 COM 所要求的。当在 Visual C++ 中创建安全数组时，应将数组下界规定为 1，以和 Fortran 数组相统一。另外，应按 Fortran 数组的列主顺序组织对应的一维数组，在 Fortran 组件方法内再将一维数组还原为二维数组。

## 二、Visual Basic

Visual Basic 6.0 对 COM 提供了很好的支持，编写 Visual Basic 客户程序也比较简便。要在 Visual Basic 中使用 COM 组件，首先需要引用组件类型库。当在 Visual Fortran 6.6 环境下编译、建造 COM 组件时，已自动进行了系统注册，所以当打开“工程/引用…”对话框时，可以自列表中选取上述 Fortran 组件类型库(ArrPass 1.0 Type Library)。

在 Visual Basic 工程中添加对组件类型库的引用后，可在对象浏览器中查看组件接口及方法(见图 12-10)。组件的缺省接口(IArrPass)被映射为 Visual Basic 中的类(ArrPass)，接口方法被转换为相应的 Visual Basic 例程，方法参数类型被转换为对应的 Visual Basic 数据类型，尤其是安全数组参数直接转换为对应数据类型的 Visual Basic 动态数组。美中不足的是参数的值传递属性(ByVal)没有标示出来。Visual Basic 和 Fortran 一样，缺省的参数传递方式为引用传递(ByRef)。当使用值传递时，必须在参数前添加 ByVal 传递属性，但 ArrRowSum 方法中的 Row 和 Col 参数没有给出 ByVal 属性。Visual Basic 调用 COM 组件的代码为：

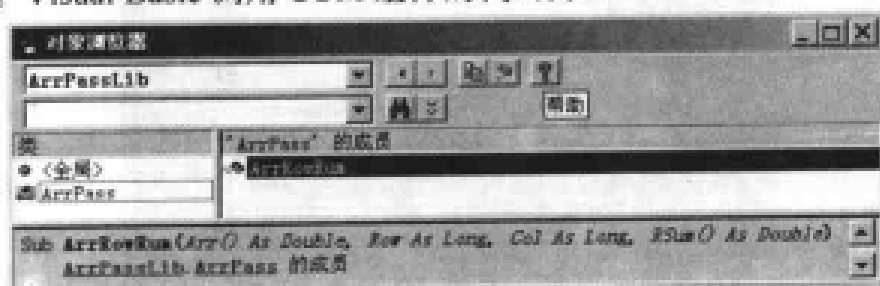


图 12-10 对象浏览器中的接口及方法

Option Explicit

Private ISum As New ArrPassLib.ArrPass

Private Sub Form\_Load()

Dim Arr(5) As Double

Dim S(1) As Double

Arr(0) = 1: Arr(1) = 4: Arr(2) = 2

Arr(3) = 5: Arr(4) = 3: Arr(5) = 6

ISum.ArrRowSum Arr, 2, 3, S

Debug.Print Str(S(0)) + vbLf + Str(S(1))

End Sub

使用 New 关键字声明 Visual Basic 类对象的同时, 创建该实例对象, 之后, 就可直接调用对象方法了。

值得注意的是: 程序中的数组虚参为动态数组, 实参为固定数组。

### 三、Delphi

Delphi 中使用 COM 组件也需要类型库的支持。打开“Project/Import Type Library...”对话框(见图 12-11), 在列表中选取组件类型库 ArrPass 1.0 Type Library

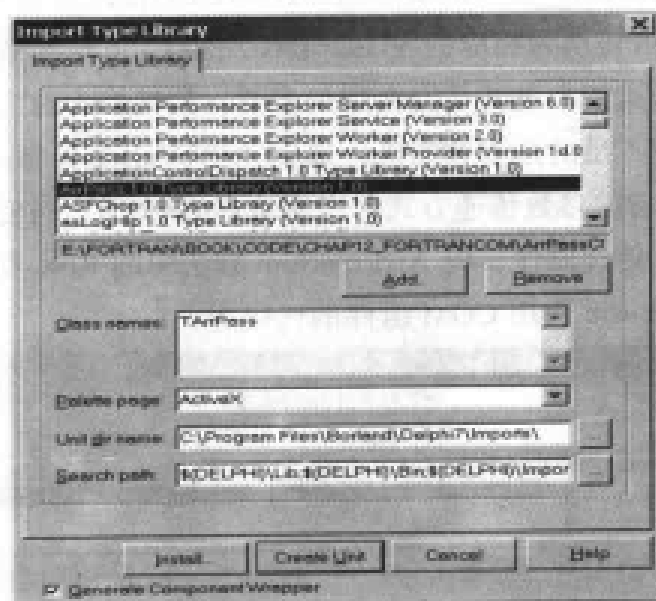


图 12-11 引入组件类型库

(Version 1.0), 保留缺省类名 TArrPass, 选取 “Generate Component Wrapper”, 点击 “Create Unit” 按钮, 以产生 Delphi 的组件包装类, 并生成相应的类型库单元 (ArrPassLib\_TLB)。

调用 COM 组件的程序单元为:

---

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, ArrPassLib_TLB, ActiveX;

type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);
var
  IArr: IArrPass;
  a, b: array of Double;
  ix: array of Integer;
  pa, pb: PSafeArray;
```

```
sabound:array[0..1]of SafeArrayBound;
i,j:Integer;
begin
  SetLength(a,6);
  a[0]:=1.0; a[1]:=4.0; a[2]:=2.0;
  a[3]:=5.0; a[4]:=3.0; a[5]:=6.0;

  SetLength(b,2);
  SetLength(ix,6);
  ix[0]:=1; ix[1]:=2; ix[2]:=3;
  ix[3]:=4; ix[4]:=5; ix[5]:=6;

  sabound[0].lLbound := 1;
  sabound[0].cElements := 6;
  sabound[1].lLbound := 1;
  sabound[1].cElements := 2;

  pa := SafeArrayCreate(VT_R8, 1, sabound[0]);
  pb := SafeArrayCreate(VT_R8, 1, sabound[1]);

  for i := 0 to 5 do
    SafeArrayPutElement(pa, ix[i], a[i]);

  IArr:=CoArrPass.Create;
  IArr.ArrRowRum(pa,2,3,pb);

  for j :=0 to 1 do
    SafeArrayGetElement(pb, ix[j], b[j]);

  ShowMessage(FloatToStr(b[0])+' '+ FloatToStr(b[1]));
end;

end.
```

---

程序中,引用了类型库单元(ArrPassLib\_TLB)和 ActiveX 单元。类型库单元包含了组件类、接口与方法的定义,以及组件包装类、创建组件对象的简便方法等,使得程序中可以采取以下简单方式创建 COM 接口:

```
IArr:=CoArrPass.Create
```

ActiveX 单元则包含了 COM API 函数声明。针对 COM 要求的安全数组, Visual C++ 直接调用相应的 COM API 函数来创建、设置和访问; Delphi 则通过其封装函数间接调用 COM API 函数。两者操作的方式都是相同的。

### 小 结

- 在 Visual Fortran 6.6 环境下,利用其 COM 组件向导,可以方便地定义组件类、接口、方法等,将 Fortran 计算程序转换成 COM 组件,从而为 Fortran 计算程序在其他应用程序中的集成提供了先进的方法。
- 建造 Fortran COM 组件,通常是将 Fortran 计算例程转换为组件接口的方法。组件接口应采用双接口,方法参数的数据类型应选择与自动化兼容的数据类型,以使建造的组件能被各种客户程序所访问。
- 定义接口方法时,使用 Intent 规定参数(及函数结果)的传递属性。三个值 In、InOut 和 Out,分别对应值传递、引用传递和函数结果。数组参数采取引用传递,并规定为假定形状数组,接口描述库(IDL)则将其转换为安全数组。
- Visual C++ 可直接利用组件接口定义文件(Server\_i.c 和 Server.h),调用 COM API 函数(CoCreateInstance)创建组件对象。安全数组的操作,也是调用 COM API 函数完成的。
- Visual Basic 对 COM 组件的访问比较简便:引用组件类型库,声明并创建组件对象,然后调用对象方法。值得一提的是: Visual Basic 在读取组件类型库信息时,没有将方法参数的值传递属性(ByVal)显式标示出来。
- Delphi 对 COM 组件的访问需要引入组件类型库,创建组件包装类,利用包装类直接创建组件对象。对安全数组的操作,类似于 Visual C++,只不过是间接调用 COM API 函数实现的。

## 参 考 文 献

- [1] Brian Hahn. FORTRAN 90 for Scientists and Engineers. 1994
- [2] Compaq Computer Corporation. Compaq Fortran Language Reference Manual. 1999
- [3] John R. Cary and Svetlana G. Shasharina. Comparison of C++ and Fortran 90 for Object Oriented Scientific Programming. Computer Physics Communications, Nov. 1996
- [4] Compaq Computer Corporation. Compaq Visual Fortran Version 6.6 online documentation. 2001
- [5] Microsoft Corporation. MSDN Library Visual Studio 6. 1998
- [6] Borland Software Corporation. Borland Delphi Enterprise Version 7.0 online documentation. 2002
- [7] 周振红, 杨国录, 周洞汝, 等. FORTRAN 与 VISUAL BASIC 混合编程的研究. 武汉水利电力大学学报, 1999,32(2)
- [8] 周振红, 颜国红, 吴虹娟. Fortran 与 Visual C++混合编程研究. 武汉大学学报(工学版), 2001,34(2)
- [9] 周振红, 周洞汝, 杨国录. 基于 COM 的软件组件. 计算机应用, 2001,21(3)
- [10] 周振红, 李端有, 谈戈. Fortran 与 Delphi 混合编程. 计算机应用, 2001,21(4)
- [11] 周振红, 任慧, 杜丽平. Fortran DLL 组件集成到 .NET 平台. 武汉大学学报(工学版), 2005,38(4)
- [12] 周振红, 任慧, 杜丽平. Fortran COM 组件集成到 .NET 平台. 武汉大学学报(工学版), 2005,38(5)