Chapter Sixteen

HANDLING DATA IN FORTRAN

Introduction

There are two fundamental types of numbers in mathematics and programming: those that are whole numbers and those that are not. In Fortran, these are known as integers and real numbers respectively, and the difference between them is important. A third fundamental data type allows character information to be stored and manipulated.

This lesson discusses these three basic data types and the ways that they can be used in calculations and expressions. This chapter also discusses the use of arrays, both linear arrays (one dimension) and matrices (two dimension).

The implementation of user-defined structures by Compaq Alpha Fortran, as an extension to the FORTRAN-77 standard makes creation of data records that match SuRPAS database record layouts very easy. The last topic of this chapter duscusses the structure, mapped unions and records, and how they are used in SuRPAS.

Variables and Constants

Overview

Constants and variables provide the Fortran programmer with the tools to manage and manipulate data within a program or program unit. In general, all data is stored as variable in Fortran with the exception of data defined by the PARAMETER statement.

This section discusses:

- A comparison of constants and variables
- Integer variables
- Real variables
- Character string variables
- Logical variables

Comparison of Variables and Constants

Variables and constants are used in all programming languages to assign meaningful names to information. Variables are named memory locations containing information that can change as the program executes. Constants, as the name implies, are named memory locations containing information that will not change during program execution.

Fortran Constants

A Fortran constant is a data object that is defined before a program begins running, and does not change its value during the program's execution. When the Fortran compiler resolves a constant, usually through the use of a PARAMETER statement, it places its value in a known memory location. It references that memory location whenever the program uses the constant.

Fortran Variables

A Fortran variable is a data object that can change value during the execution of a program. (The value of a Fortran variable may or may not be initialized before the program is executed.) When the Fortran compiler resolves a variable, it reserves a known memory location and then references that memory location whenever the program uses the variable.

Each Fortran variable in a program unit must have a unique name. Fortran names may be up to 31 characters long and may contain any combination of alphabetic characters, numbers, and the underscore character (_). (Compaq Alpha Fortran also supports the dollar sign but, in the SuRPAS environment it is only used when defining PARAMETER constants.) The first character of a variable name must always be alphabetic.

The following are examples of legal Fortran variable names:

- JulianTime
- long_distance
- T100xr
- The_April_Secret_Number

The following are examples of Fortran variable names that are not legal:

- 3_Months (begins with a number)
- This_Variable_Name_Is_Too_Long_To_Be_Compiled (too long)
- FAL% (illegal special character)

Assigning Values to Variables

Fortran variables receive their value through assignment. A variable is assigned the result of an expression or calculation and that result is placed in the memory location of the variable for later reference by the executing program. Each variable is associated with data of a specific type. The type of the variable is determined in one of three ways:

- Default Implicit Type Specification
- Declared Implicit Type Specification
- Explicit Type Declaration

The Default Implicit Type

In the absence of any declaration to the contrary, the data type is determined according to the rules of ***default implicit type*** specification, depending on the first letter of the variable name. If the initial letter of the variable name begins with an I, J, K, L, M, or N the variable is of type integer. If the variable name begins with any other letter, it is of type real.

The Declared Implicit Type and the IMPLICIT Statement

The default implicit type rules may be superseded by a ***declared implicit type*** specification expressed by the IMPLICIT statement. For example, the following declaration changes the implicit type rules so that variables whose names begin with P, Q, or R are now also of type integer, while variables beginning with the letter "M" are now of type real:

```
IMPLICIT INTEGER*4 (P-R), REAL*8 (M)
```

The default rules for variable names beginning with other letters remain in effect.

SuRPAS coding standards do not support the usage of the IMPLICIT statement. All variables and constants must be declared using explicit type declarations, as discussed in the next section.

Explicit Type Declarations

An ***explicit type declaration*** may also be used to specify the type of individual integer, real, and character variables. Such declarations supersede the default or declared implicit type rules, but have no effect on variable names not listed. The following are examples of explicit type declarations:

```
• INTEGER*4    Big, Wide, Hardy, X15, W
• REAL*4       Large, Prime, J22, Q
• CHARACTER*16 Report_Name, Last_Name, First_Name
```

This list can be expanded to support all data type declarations as we will see in the sections that follow. Every data type declaration issued consists of the data type followed by a list of variables to be specified, with each variable in the list separated by a comma (as seen in the above examples). Data type declarations can be issued numerous times in a program unit for readability and clarity purposes. There is no order or sequence required for issuing explicit data statements. *SuRPAS coding standards require a separate line for each variable or constant declared using explicit type declarations (with a comment included). The above examples do not follow the SuRPAS standard and should be coded as follows:*

```
• INTEGER*4    big            ! the large size
  INTEGER*4    wide           ! the expanded width
  INTEGER*4    hardy          ! voluminous container
  INTEGER*4    x15            ! result multiplied
                              ! by 15
  INTEGER*4    w              ! temperature calc.

• REAL*4   large               ! big expressed as a
                               ! floating-point num
  REAL*4   prime               ! a prime number
  REAL*4   j22                 ! version 22 of the
                               ! joint calculation
  REAL*4   q                   ! the quotient

• CHARACTER*16  report_name    ! report name string
  CHARACTER*16  last_name      ! last name of client
  CHARACTER*16  first_name     ! 1st name of  client
```

Integers

An ***integer*** is any number that does not contain a decimal point; it can not represent a number with fractional parts. No commas may be embedded within an integer, and a positive integer may be written with or without a plus sign. The following are examples of valid integers:

- 0
- -199
- 123456789
- +32

The following integers are not valid in Fortran:

- 1,000,000 (Embedded commas are illegal.)
- -100. (It is not a valid integer constant if it has a decimal point.)

An ***integer variable*** is a uniquely named variable containing a value of the data type integer.

Variables of the integer data type are usually stored in a single longword (32-bits) of Alpha memory (a range between -2×10^9 and $+2 \times 10^9$). On an OpenVMS system an integer can be stored in any of the following:

- an 8-bit byte (`INTEGER*1`)
- a 16-bit word (`INTEGER*2`)
- a 32-bit longword (`INTEGER*4`)
- a 64-bit quadword (`INTEGER*8`)

If the size is not specified, the integer will be stored in the smallest size memory location that will fit the value.

At PFPC, size is always specified when declaring variable types (integer, real, or character).

Integers are expressed in decimal values (base 10) by default. Compaq Alpha Fortran supports numbers expressed in values up to base 36, using the 10 numbers and 26 alphabetic letters (followed by the number sign) to indicate the base of the integer value being expressed. To specify a constant that is not in base 10, you must use this notation. A few examples of the decimal number 350 in other bases follow:

- F#15d (hexadecimal)
- 2#101011110 (binary)
- 8#536 (octal)

Real Numbers

The real data type consists of numbers stored in real or floating-point format. Unlike integers, the real data type can represent numbers with fractional components.

The ***real number*** is a number written with a decimal point. If the value is positive, it may be written with or without a plus sign. No commas may be embedded within a real number.

Real numbers may be written with or without an exponent. If you are using an exponent, be aware of the size of the real number. Compaq Alpha Fortran supports 32-bit, 64-bit, and 128-bit real numbers. The exponent of a 32-bit real number begins with the letter *E*, and is followed by a positive or negative number that corresponds to the power of 10 used when the number is written in scientific notation. (Double precision or 64-bit real numbers begin their exponents with the letter *D*.) If the exponent is positive, the plus sign may be omitted. The mantissa of the number should contain a decimal point. The following are examples of valid REAL*4 (32-bit) numbers:

- 36.
- -454.3
- 1.14159
- 0.245E+2
- +1.01E-3
- 0.0D0 (double precision - REAL*8 numbers must have exponent notation)

The following are examples of invalid REAL*4 numbers:

- 10,256. (commas not allowed)
- $12.95 (special characters not allowed)
- 348 (no decimal point - this is a valid integer)
- -38.E47 (too large for REAL*4; -38.D47 is a valid DOUBLE PRECISION number)

A ***real variable*** is a variable containing a value of the data type real.

Variables of the real data type are usually stored in a single longword (32-bits) of Alpha memory (approximately between -10^{38} and $+10^{38}$, to 7 or 8 significant digits). On an OpenVMS system a real number can be stored in:

- a 32-bit longword (REAL*4)
- a 64-bit quadword (REAL*8) - also called DOUBLE PRECISION
- a 128-bit quad-precision word (REAL*16)

If the size is not specified, the number will be stored in the smallest size memory location that will fit its value. ***Unspecified size is not supported in SuRPAS code.***

Character Strings

A ***character string*** is a sequence of characters enclosed between apostrophes (single quotes). The enclosing apostrophes are not included in the character information represented by the string, but rather denote the beginning and end of the character set. There must be at least one character between the pair of single quotes, and blanks or spaces are as significant as other characters. Both Fortran and non-Fortran characters are permitted in character strings.

When an apostrophe is required within the character string, that single quote is replaced by a *pair* of single quotes (not a double-quote). For example, the string containing the word HAVEN'T would be written as **'HAVEN''T'**.

Numbers can also be placed in character strings. The numbers are interpreted by Fortran as a series of characters, not with any numeric value. As an example, the character string '512' represents a sequence of the three characters 5,1, and 2.

The length of a character string is the number of individual characters between the enclosing apostrophes, except when a pair of single quotes is included to represent a single quote. In this case the pair of quotes is counted as one character. Remember that blanks and spaces are counted along with special characters and non-Fortran characters.

The following are examples of valid character strings:

- '123'
- 'HELLO THERE'
- 'DIDN''T'
- 'Did you see Mary''s book?'

Variables of the character data type must be explicitly or implicitly declared because there is no default implicit type specification for types other than integer or real. A character type declaration must include a length specification (unless the variable is to have a default length of one). In its simplest form, the length specification consists of an asterisk followed by an integer constant.

Other Types of Variables

Logicals

The logical data type contains one of only two possible values: true or false. A ***logical*** can have one of the following values: `.TRUE.` or `.FALSE.`. (The periods are required on either side of the values to distinguish them from variable names.) A ***logical variable*** is a variable containing a value of the logical data type. Logical variables must be declared using the explicit type **`LOGICAL`**. Although logical constants (as created by the PARAMETER statement) are rarely used, logical variables and expressions are commonly used to control program execution.

Expressions

Overview

Calculating simple and complex operations and assigning the results to a variable in a given memory location is fundamental to solving problems in a computer environment. The calculations may be arithmetic or logical, the result of high-level mathematical functions, or comparisons of character strings. Fortran provides an evaluation process to achieve these results using a defined set of operators and rules. The results must still be available for future use in the program.

This section discusses:

- The assignment statement
- Arithmetic expressions and operators
- Logical expressions and operators
- Character expressions and operators

The Assignment Statement

The assignment statement is one of two basic kinds of Fortran statements that can be used to store a new value in a memory location. (The other is the READ statement.) An assignment statement consists of two parts, separated by an equal sign. A variable appears on the left side of the equal sign and an expression is on the right.

The syntax for a standard assignment statement is:

```
Variable = Expression
```

In Fortran, a ***variable*** is associated with a particular memory location. In this case the program executes the expression in the assignment statement and stores the result of the expression in the variable memory location. Don't be confused by the equal sign in the assignment statement; it is acting as a direction signal (the variable is the result of the expression) rather than indicating equality. For this reason we often refer to this equal sign as the ***assignment operator***.

The ***expression*** may be very complicated or it may simply be a constant. The following are some examples of simple expressions in an assignment statement:

- A = 3.145
- X = 4.5
- PI = 3.14159
- Blossoms = 5
- Miles = 34
- N1 = 428
- Large = 348209

In the examples above there are two kinds of constants: integer constants and real constants. In Fortran, real constants are written with a decimal point and integer constants are written without one. The expression to the right of the equal sign can be any valid combination of constants, variables, parentheses, and arithmetic or logical operators.

Arithmetic Operators

The standard Arithmetic operators included in Fortran are:

+	**Addition**
-	**Subtraction**
*	**Multiplication**
/	**Division**
**	**Exponentiation**

Note that the symbols for multiplication (*), division (/), and exponentiation (**) are not the ones used in ordinary mathematical expressions. These symbols were chosen because they were available in the 1950's era computer character sets and were different than the characters being used in variable names.

The five arithmetic operators listed above are binary operators. They occur between and apply to two variables or constants. In addition, the plus sign (+) and minus sign (-) can occur as unary operators. They can apply to a single variable or constant to indicate that it is a positive or negative number.

Precedence of Operations

The rules for evaluating a Fortran expression follow three rules of evaluation. These rules define the order or precedence of operations.

1. Quantities within parentheses are computed first, before the parenthesized subexpression is combined with other parts of the expression. If there are nested sets of parentheses, the innermost set is evaluated first.

2. Arithmetic operations within any expression or parenthesized subexpression are performed in the following order of precedence:
   First - exponentiation
   Second - multiplication and division
   Third - addition, subtraction, and negation

3. When an expression involves two or more operations that are on the same level in the precedence list, their position within the expression determines the order in which these operations are performed. With the exception of exponentiation, they are performed in order from left to right. In the case where two exponentiation activities occur in order, they are evaluated from right to left, so that A ** B ** C is interpreted as $A^{(B^C)}$

Rules For Using Arithmetic Operators

The following rules apply when using Fortran arithmetic operators:

- No two operators may occur side by side. In a case where a binary operator is acting on a unary operator, the two operators must be separated by a set of parentheses. In the case of **A * -B**, where **B** is negative, the **-B** must be surrounded by a set of parentheses. In Fortran this would be written as **A * (-B)**. Similarly, if **C** is raised to the power of **-5**, it is written as **C ** (-5)** in Fortran.

- Implied multiplication is illegal in Fortran. The arithmetic expression **X(Y + Z)** means that **Y** and **Z** should be added and the result should be multiplied by **X.** In Fortran the implied multiplication must be written explicitly as **X * (Y + Z)**.

- Parentheses may be used to group terms whenever desired. Expressions inside parentheses are evaluated before the expressions outside the parentheses.

  For example, the expression **2 ** ((8 + 2) / 5)** is evaluated as follows (note that each step is underlined and then replaced by its result):

  - 2 ** (***(8 + 2)*** / 5)
    10 = (8 + 2)

  - 2**(***10 / 5***)
    2 = (10 / 5)

  - ***2**(2)***
    4 = 2 ** 2

Logical Expressions and Operators

Like arithmetic calculations, logical operations are performed with an assignment statement that has the form:

```
Logical_variable_name = Logical expression
```

The expression to the right of the equal sign can be any combination of valid logical constants, logical variables, and logical operators. A ***logical operator*** is an operator on numeric, character, or logical data that yields a logical result. The two basic types of logical operators are ***relational operators*** and ***combinational operators***.

Relational Operators

Relational logical operators are operators with two numeric or character operands that yield a logical result. Since the result depends on the relationship between the two values being compared, these operators are called relational. The following illustrates the general form of a relational logical expression:

```
operand-a operator operand-b
```

In this relational logical expression, `operand-a` and `operand-b` are arithmetic expressions, variable constants, or character strings and the `operator` is one of the relational logical operators listed in Table 16-2 below.

Operation	Meaning
.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greater than or equal to
.LT.	Less than
.LE.	Less than or equal to

Table 16-2 - Relational Logical Operators

If the relationship between operand-a and operand-b is true, then the operation returns a value of .TRUE.; otherwise the operation returns a value of .FALSE.. Some examples of relational operations and their results can be found in the list below:

Operation	Result
7 .LT. 8	.TRUE.
7 .LE. 8	.TRUE.
7 .EQ. 8	.FALSE.
7 .GT. 8	.FALSE.
8 .LE. 8	.TRUE.
'A' .LT. 'B'	.TRUE.

Fortran uses the ASCII character set as its basis for the logical evaluation of characters. Each ASCII character is assigned a numeric value. Alphabetic characters are assigned sequential values beginning with upper case A-Z, followed by lower case a-z. For this reason, 'A' is evaluated as less than 'B'. Additionally, 'B' would be evaluated as less than 'a'.

Hierarchy of Logical Operations

When determining the hierarchy of operations, relational operators are evaluated after all arithmetic operators have been evaluated. Therefore, the following two expressions have equivalent results (both result in .TRUE.):

```
 7 + 3  .LT.  2 + 11
(7 + 3) .LT. (2 + 11)
```

If the comparison is between real and integer values, the integer value is converted to a real value before the comparison operation is performed. Comparison operations between numerical data and character data are illegal and will cause a compile-time error.

`4 .EQ. 4.` Result is .TRUE. (Integer is converted to real and the comparison is made.)

`4 .LE. 'A'` Illegal operation - produces a compile-time error

Combinational Operators

Combinational logical operators are operators with one or two operands that yield a logical result. There are four binary operators in this category and one unary operator. They are defined in Table 16-3 below.

Operator	Function	Definition
Binary		
I_1 .AND. I_2	Logical AND	Result is TRUE if both I_1 and I_2 are TRUE
I_1 .OR. I_2	**Logical OR**	**Result is TRUE if either or both of I_1 and I_2 is/are TRUE**
I_1 .EQV. I_2	Logical equivalence	Result is TRUE if I_1 is the same as I_2 (either both are TRUE or both FALSE)
I_1 .NEQV. I_2	**Logical nonequivalence**	**Result is TRUE if one of I_1 and I_2 is TRUE and the other is FALSE**
Unary		
.NOT. I_1	Logical NOT	Result is TRUE if I_1 is FALSE, and FALSE if I_1 is TRUE

Table 16-3 - Combinational Logical Operators

Periods are a part of the operator and must always be present. If the relationship between I_1 and I_2 expressed by the operator is true, then the operator returns a value of .TRUE.; otherwise, the operation returns a value of .FALSE..

In the hierarchy of operations, combinational logical operators are evaluated after all arithmetic operations and all relational operators have been evaluated. As with arithmetic operators, parentheses can be used to change the default order of evaluation.

Character Expressions and Operators

Character manipulation is performed in much the same manner as arithmetic and logical operations, using an assignment statement with the following form:

```
character_variable_name = character expression
```

The character assignment statement calculates the value of the character expression to the right of the equal sign and assigns that value to the variable named to the left of the equal sign.

The expression to the right of the equal sign can be any combination of valid character constants, character variables, and character operators. A character operator is an operator on character data that yields a character result. The two basic types of character operators are ***substring specification*** and ***concatenation***.

A character expression may be assigned to a character variable with an assignment statement. If the character expression is *shorter* than the length of the character variable to which it is assigned, then the rest of the variable is padded with blanks. If the character expression is *longer* than the length of the character variable, then the excess portion of the character variable is discarded.

The Substring Specification

It may be useful to extract one or more characters from a string in a program. Fortran contains a substring specification that selects a subrange of characters in a variable and treats it as if it were an independent character variable. If the variable `vstring` is a six-character variable containing the string `'market'`, the substring `vstring (2:4)` would be a three-character variable containing the string `'ark'`. The substring `vstring (2:4)` refers to the same memory locations as characters 2 through 4 of `vstring`; therefore, if the contents of `vstring (2:4)` is changed, the characters in the middle of `vstring` will also be changed.

In the following code snippet, three substrings are extracted from two variables and then printed.

```
CHARACTER*20  Name
CHARACTER*20  Title
Name = 'Caroline Hobson'
Title = 'THE QUICK BROWN FOX'
PRINT *, Name(10:15), Title(1:3), Title(17:19)
```

Code Example 16-1 - Substring Code Snippet

The output produced from this snippet is: **Hobson THE FOX**

The Concatenation Operator

Fortran provides the ability to combine two or more strings or substrings into a single large string. This operation is known as ***concatenation***. The concatenation operator in Fortran is a double slash with no space between the slashes (//). The same code snippet that was used to demonstrate the substring can be easily adapted to demonstrate the concatenation operator.

```
CHARACTER*20  Name
CHARACTER*20  Title
CHARACTER*20  Result
Name = 'Caroline Hobson'
Title = 'THE QUICK BROWN FOX'
Result = Name (10:16) // Title (1:3) // Title (16:19)
PRINT *, Result
```

Code Example 16-2 - Concatenation Code Snippet

The output produced from this snippet is: **Hobson THE FOX**. Note that in this snippet, the spaces between the concatenated words are forced by the range of characters selected in each substring.

There is a restriction in Fortran that prohibits using the same character positions in the expressions on the right side that are included in the string named on the left side of an assignment statement.

The following substring and concatenation assignments are valid in Fortran:

```
Axe (1:3) = Axe (17:19)
Yoyo = Name (1:5) // ' ' // Axe (8:12)
Ox = Name ( :7)
```

These substring and concatenation assignments are invalid:

```
Axe (1:3) = Axe (2:4)
Ox = 'ZZ' // Ox (3:14)
Ark = Ark (1:8) // Axe
Ox = Ox ( :7) // ' ' // Ox (9: )
```

Arrays and Array Elements

Overview

An ***array*** is a group of variables or constants, all of the same type, that are referenced by a single symbolic name. The values in the group occupy consecutive memory locations. An individual value within the array is called an ***array element***; it is identified by the name of the array together with a ***subscript*** pointing to a particular location within it.

Arrays can be powerful tools. They permit us to apply the same algorithm over and over again to many different data items with a simple DO loop. We can manipulate and perform calculations with individual elements of arrays one by one, with whole arrays at once, or with various subsets of arrays. In this section, we will learn how to use individual array elements, whole arrays, and array subsets in Fortran statements.

This section discusses:

- Declaring array structures and array elements
- Array data types
- Linear Arrays
- Multi-dimensional Arrays

Declaring Lists or Linear Arrays in Fortran

Solving problems through the use of a computer often requires building and manipulating lists of data. When declaring a variable that represents a series of data elements or an array of elements, it is necessary to "tell" the computer that the symbolic name represents a group (rather than only one element) in order for the data to be accessed. When this list is sequential we refer to it as a ***linear array***. Linear arrays can be visualized as a series of values laid out in a line or column.

Using Data Declarations to Declare Arrays

In Fortran, declaring a variable as a list of elements can take on a number of forms. Using the standard Fortran data type declaration, a programmer can declare the type and number of elements of a variable as a list of elements. This is done by following the symbolic name declaration with the number of elements surrounded in parentheses. The example data declaration statements that follow declare a REAL array named **degrees** with 360 elements and two INTEGER arrays - one named **radians** with 120 elements and a second named **arcs** with 15 elements.

```
REAL*4    Degrees  (360)
INTEGER*4 Radians  (120)
INTEGER*4 Arcs  (15)
```

Note that declarations for two or more arrays can be combined into one declaration. A specific element of the array can be referenced by attaching an element number to the array name. The element number is also referred to as the array element's ***subscript***.

The subscript is enclosed in parentheses and is a part of the array name. It can be an integer or integer expression, the value of which is *not less than one and not greater than the length defined in the array declaration*. The following are examples of valid array element names:

```
Arc (3)
Radian (K)
Thrift (MMkt + 2)
Market (Count - 1)
Degrees (2 * Dia)
A (NINT (SQRT (0,5 * X) + A(K) ) + 7)
```

Note that a real expression can be included within a subscript if explicit type conversion is applied. Use of the rounding function NINT is preferable to INT for this conversion because small errors in the representation of the real expression can cause its value to be just less than the expected integer.

Using Array Elements

Each element of an array is a variable just like any other variable, and an array element may be used in any place where an ordinary variable of the same type may be used. Array elements may be included in arithmetic and logical expressions, and the results of an expression may be assigned to an array element. As an example, if arrays `index` and `temp` are declared as:

```
INTEGER*4 Index (10)
REAL*4    Temp (3)
```

Then the following Fortran statements are valid:

```
Index (1) = 1
Temp (3)  = REAL (Index (1)) / 4.
```

Array elements can be initialized by one of three techniques (which will be discussed in later chapters):

- Assignment statements (usually in a DO loop)
- Type declaration statements (at compile time)
- READ statements (interactive or from a file)

Always initialize the elements of an array before using them!

Two-Dimensional Arrays

The linear arrays that have been discussed so far describe data that is a function of one independant variable, such as a series of temperature measurements or daily mutual fund closing values. Some types of data are functions of more than one independent variable. For example, we may wish to record the opening, noontime, and closing values for five different mutual funds. In this example, our fifteen recorded values could logically be grouped into five rows of three measurements with a separate row for each mutual fund. Fortran can easily support this sorting of data with a two-dimensional array (also called a ***matrix***).

Building Matrices Using Two-Dimensional Arrays

Declaring a two-dimensional array is not very different from declaring a linear array. When explicitly declaring the array variable, two subscripts are supplied and separated by commas. For our 5 row by 3 column mutual fund example, we might issue an explicit declaration that looks like the one below.

```
REAL*4 MFund (5,3)
```

The first subscript represents the five mutual funds (rows) and the second subscript, the three sampling times (columns).

	AM	NOON	PM
Mutual Fund #1	**1,1**	**1,2**	**1,3**
Mutual Fund #2	**2,1**	**2,2**	**2,3**
Mutual Fund #3	**3,1**	**3,2**	**3,3**
Mutual Fund #4	**4,1**	**4,2**	**4,3**
Mutual Fund #5	**5,1**	**5,2**	**5,3**

Storing Array Elements in Memory

Fortran stores linear array elements in consecutive memory locations. It stores two-dimensional array elements in ***column major order*** (the first column's elements, then the second column's elements, etc.) until all columns have been allocated. In the case of our 5 row by 3 column array, the morning values for each mutual fund are stored, followed by the noontime values, and then finally the afternoon values.

Multi-dimensional Arrays

Fortran supports more complex arrays with up to seven different subscripts. These larger arrays are declared, initialized, and used in the same manner as the two-dimensional arrays described in the previous section.

Storing Multi-Dimensional Array Elements in Memory

Multi-dimensional array elements are stored in memory in a manner that is an extension of the column order used in two-dimensional arrays. Memory allocation for a three-dimensional array will store the entire range of elements of the first subscript and then the entire range of elements for the second subscript before storing the full range of elements for the third subscript. So if we added a third subscript for our mutual fund array by storing five days of information instead of a single day, the data declaration might be issued as follows.

```
REAL*4 MFund (5,3,5)
```

The data storage order would be:

```
MFUND  (1,1,1)
MFUND  (2,1,1)
MFUND  (3,1,1)
MFUND  (4,1,1)
MFUND  (5,1,1)
MFUND  (1,2,1)
MFUND  (2,2,1)
MFUND  (3,2,1)
MFUND  (4,2,1)
MFUND  (5,2,1)
MFUND  (1,3,1)
MFUND  (2,3,1)
MFUND  (3,3,1)
MFUND  (4,3,1)
MFUND  (5,3,1)
MFUND  (1,1,2)
MFUND  (2,1,2)
    •
    •
    •
MFUND  (2,3,5)
MFUND  (3,3,5)
MFUND  (4,3,5)
MFUND  (5,3,5)
```

DATA and PARAMETER Declarations

Overview

One difficulty with constants is that they are not always constant. It is often necessary to change some constants that are incorporated in the program text. Searching for all the appearances of a certain constant among the statements of a long program can be a laborious task.

Fortran provides two ***declaration*** mechanisms for incorporating data values known prior to execution. Either of these can be used to establish a value that is used instead of a numerical constant in the program text. Specifying the value just once in a declaration that appears ahead of the program body makes it easier to locate and change at a later time.

This section discusses:

- The DATA declaration
- Implied DO loops in DATA declarations
- The PARAMETER declaration

Initializing Data Values

Although DATA statements and PARAMETER statements can be used in similar situations, these two mechanisms operate quite differently. The DATA statement is specificly useful for establishing the initial value of a variable that may change during program execution, while the PARAMETER statement specifies the value of a constant that was declared in a previous data declaration. Either mechanism may be preferable (depending on other circumstances) for setting a value that will be referenced in an executable statement but will not be changed during execution of the program.

The DATA Statement

The DATA statement specifies data values that are to be assigned initially to a variable and stored in the associated memory locations *prior* to execution of the program. The value initially assigned to the variable can change during the execution of the program. The general syntax is:

```
DATA list-of-items / list-of-constants /
```

The ***list-of-items*** is similar to an input or output list, and the ***list-of-constants*** provides the corresponding values in sequential order. These may be numeric constants, logical constants, character constants, or a mix. In this code snippet a number of different data type variables are initialized using the DATA statement.

```
INTEGER*4      Mt
REAL*4         R, S(4), T(4,4)
LOGICAL        Sw
CHARACTER*5    Title
DATA      R, S(4), T(1,3), Mt/-9.25, 4.9E3, 5.9, 14/
DATA      Sw, Title           /.TRUE., 'Hello'/
```

Code Example 16-3 - DATA Declaration Code Snippet

These DATA statements are equivalent to the execution of the following assignment statements prior to running the executable code of the program.

```
Mt = 14
R = -9.25
S (4) = 4.9E3
T (1,3) = 5.9
Sw = .TRUE.
Title = 'Hello'
```

Note the one-to-one correspondence between the number of items in the ***list-of-items*** and the number of data constants in the ***list-of-constants***.

Implied DO Loops in DATA Declarations

An ***implied DO loop*** may be used in a DATA declaration. In this DATA statement the *list-of-items* includes a single implied DO loop as a part of the variable item.

```
DATA (A (I), I = 1,5)   /5.1, 6.2, 7.3, 8.4, 9.5/
```

This DO loop designates all of the elements A_1 through A_5. The loop parameters (initial value, limit value, and increment) must be integer constants or constant expressions. The implied DO loop `(A(I), I = J,K)` would be invalid even if values for **`J`** and **`K`** were specified in preceding DATA statements.

Implied DO loops can be nested. The parameters of the outermost implied DO loop must be integer constants or constant expressions. However, those of the inner loop may also involve integer variables that appear as parameters in the outer list of the same nest. In the next DATA statement the limit of the inner loop is **`I`**, which is the index variable of the outer loop.

```
DATA ( (K(I,J),J = 1,I), I = 1,8) /36 * 1/
```

What portion of the array **`K`** has been initialized with 1's?

The PARAMETER Declaration

A PARAMETER declaration creates a named constant, a symbolic name (having the same appearance as a variable name) that can be used in a program in the same way as a constant. The use of named constants in executable statements is an alternative to using a variable whose value is set initially in a DATA statement.

Declarations do not permit the inclusion of variables whose values are established during execution, or even just prior to execution in a DATA statement. A named constant, however, may be declared in a PARAMETER statement and then used in any declaration (except a FORMAT declaration) that follows it in the program code. In this code snippet, the REAL variable `matrix` is declared as a 100 by 100 rectangular array, and the REAL variable `width` as a linear array of 100 cells.

```
PARAMETER (N = 100)
REAL*4    Matrix (N,N)
REAL*4    Width (N)
```

To change the size of **`Matrix`** to a 50 by 50 array, while decreasing **`Width`** to 50 cells as well, it is only necessary to change the value of **`N`** (specified in the PARAMETER statement) to 50.

A PARAMETER declaration consists of the word PARAMETER followed by a parenthesized list of pseudo-assignments. Each of these consists of a name (which is declared to refer to a named constant), followed by an equal sign and an expression. The expression is composed of constants including previously declared named constants. (It must not contain variables, array element references, or function references.) The pseudo-assignment may involve mixed types in the same manner as an actual assignment statement; the type associated with the named constant is determined (implicitly or explicitly) by the declared name.

In the following code snippet the PARAMETER statement establishes the named character constant **able** with the value **cane**, as well as setting up some values for use in the DATA declaration that follows it.

```
DIMENSION cable  (10)
PARAMETER (able = 'cane', J = 4, M = 7, NJM = 4)
DATA      (cable (I), I = J, M)   /NJM * '    '/
```

BUILDING USER-DEFINED STRUCTURES

Introduction

Fortran supports the capability for programmers to create their own data types to supplement the intrinsic types provided within the Fortran language. Because these new data types must be derived from the intrinsic data types and/or previously defined new data types they are called ***derived data types***.

This chapter addresses the procedure for creating derived data types.

Objectives

To build a user-defined structure, a programmer should be able to:

- Define a derived data type structure
- Declare user records

User-defined Structures

We have studied Fortran's intrinsic data types: integer, real, character, and logical. In addition to these data types, the Fortran language permits us to create our own data types to add new features to the language or to help us solve specific classes of problems. A user-defined data type may have any number and combination of components, but each component must be either an intrinsic data type or a user-defined data type that was previously defined. User-defined data types are called ***derived data types*** because ultimately, they must be derived from intrinsic data types.

A derived data type is a convenient way to group together all the information about a particular item. Like an array, a single derived data type can have many components. Unlike an array, the components of a derived data type may have different types. One component may be an integer while the next component is a real, the next a character string, and so forth. Furthermore, each component is known by a name instead of a number.

A derived data type is defined by a sequence of type declaration statements beginning with a STRUCTURE statement and ending with an END STRUCTURE statement. Between these two statements are the definitions of the components in the derived data type. The syntax of a derived data type structure is:

```
STRUCTURE [/structure_name/] [field_namelist]
    field_declaration
    [field_declaration]
    ...
END STRUCTURE
```

The following snippet is an example of a STRUCTURE to support a name and address record:

```
STRUCTURE  /shareholder_account/
         CHARACTER*12     First_Name
         CHARACTER*1      MI
         CHARACTER*16     Last_Name
         INTEGER*4        Age
         CHARACTER*16     City
         CHARACTER*2      State
         CHARACTER*9      Zip_Code
END  STRUCTURE
```

Example 16-4 - DATA Declaration Code Snippet

The Structure Name

The ***structure_name*** is the name used to identify a structure. A structure name is enclosed by slashes. If the slashes are present, a name must be specified between them. Subsequent RECORD statements use the structure name to refer to the structure. A structure name must be unique among structure names.

Structure declarations can be nested. A structure name is required for the derived data type declaration at the outermost level of nesting, and optional for structure declarations nested in it. If you intend to reference a nested structure in a RECORD statement, it must have a name.

Structure, field, and record names are all local to the defining program unit. When records are passed as arguments, the fields in the defining structure within the calling and called subprograms must match in type, size, and order.

Field Namelist

The ***field_namelist*** is a list of fields having the structure of the associated structure declarations. A field namelist is allowed only in nested structure declarations.

Field Declarations

The ***field_declarations*** make up the declaration body of the derived data type structure. A derived data type structure can have as many field declarations as desired. A field declaration consists of any combination of the following:

- **Type declarations** - intrinsic Fortran data type declarations.
- **Substructure declarations** - a field within a derived data type structure can be a substructure composed of fields, other structures or a combination of both.
- **Union declarations** - a union declaration is composed of one or more mapped field declarations. The mapped fields logically share a common location within a derived data type structure.
- **PARAMETER statements** - PARAMETER statements can appear in a derived data type structure declaration, but cannot be given a data type within the declaration block.

Rules and Behavior

Unlike type declarations, derived data type structures do not create variables. Structured variables or records are created when you use the RECORD statement containing the name of a previously declared derived data type structure. The RECORD statement can be considered a kind of type declaration statement. The difference is that multiple items, not single items, are being defined.

Within a structure declaration, the ordering of both the statements and the field names within the statements is important because this ordering determines the order of the fields in records.

In a derived data type structure declaration, each field offset is the sum of the lengths of the previous fields, so the length of the structure is the sum of the lengths of the fields. The structure is packed; you must explicitly provide any alignment that is needed by including unnamed fields of the appropriate length. Declaring an unnamed field with type and %FILL achieves this goal.

Substructure Declarations

A field within a derived data type structure can itself be a structure composed of other fields, other structures, or both. You can declare a substructure in two ways:

- By nesting structure declarations within other structure or union declarations. one or more field names must be defined in the STRUCTURE statement for the substructure because all fields in a structure must be named. In this case, the substructure is being used as a field within a structure or union.

- By using a RECORD statement that specifies another previously defined record structure, thereby including it in the structure being defined.

Union Declarations

A union declaration is a multi-statement declaration defining a data area that can be shared intermittently during program execution by one or more fields or groups of fields. A union declaration must be within a derived data type structure declaration.

A union declaration is initiated by a UNION statement and terminates by an END UNION statement. Enclosed within these statements are two or more map declarations, initiated and terminated by MAP and END MAP statements. Each unique field or group of fields is defined by a separate map declaration. The syntax of a union declaration is:

```
UNION
     map_declaration
     map_declaration
     ...
END UNION
```

The syntax of the map declaration is:

```
MAP
     field_declaration
     field_declaration
     ...
END MAP
```

The snippet of code below demonstrates the use of both the MAP declaration and the UNION statement:

```
UNION

  MAP
      CHARACTER*15      Iob      ! Entire structure
  END MAP
  MAP
     CHARACTER*1       Type     ! 'N'=NRA; 'P'=Pens
      CHARACTER*6      Rate     ! Rate pct, fmt 99.999
      CHARACTER*8      Date     ! Rate eff. date, ccyymmdd
  END MAP

END UNION
```

Example 16-5 - UNION and MAP Declaration Code Snippet

The RECORD Statement

The RECORD statement creates a record for which the form was specified in a previously declared structure declaration. The effect of a RECORD statement is comparable to that of an intrinsic type declaration (e.g. INTEGER*4, REAL*4, CHARACTER). The difference is that instead of declaring scalar data items, composite or aggregate data items are declared. The syntax of the RECORD statement is:

```
RECORD /structure_name/ record_namelist
    [,/structure_name/ record_namelist]...
```

The ***structure_name*** is the name of a previously declared structure. The ***record_namelist*** is a list of one or more record names (variable names or array names), separated by commas. All of the records named in this list have the same structure and are allocated separately in memory.

You can use record names in COMMON statements, but not in DATA or EQUIVALENCE statements. Records initially have undefined values unless you define their values in the structure declarations.

The following example is a snippet of code from **NRAPENRATE.PAR**. It can be found in **FAL$SUB:GETCTYRAT.FOR**. This snippet illustrates the way the Fortran structure is used in most PFPC code. As you examine the code you will see the use of the UNION and MAP statements to overlay a character string on top of multiple fields of the structure.

```
INTEGER  *  4   NRAPEN_IOBLEN
PARAMETER     (NRAPEN_IOBLEN  =  512)

STRUCTURE     /  NPRINF  /
   UNION

     MAP
          CHARACTER*15      IOB      !  Entire structure
      END  MAP
     MAP
         CHARACTER*1        TYPE     !  'N'=NRA;  'P'=Pens
          CHARACTER*6       RATE     !  Rate  pct,  fmt  99.999
          CHARACTER*8       DATE     !  Rate  eff.  date,  ccyymmdd
      END  MAP

   END  UNION
END  STRUCTURE                          ! of /NPRINF/

STRUCTURE     /  NRAPENRATE  /
   UNION

     MAP
           CHARACTER*(NRAPEN_IOBLEN)    IOB
      END  MAP
     MAP
           CHARACTER*12       KEY         !'NRAPENRATE'//cc(001:012)
                                            !  (cc=country code)
          CHARACTER*42       NAME          !  Country  name(013:054)
           RECORD  /NPRINF/  NPRINF_R(30)  !Typ/dat/dat(055:504)
          CHARACTER*8        FILLER01     !  Filler         (505:512)
      END  MAP

   END  UNION
END  STRUCTURE

RECORD    /  NRAPENRATE  /  NRAPENRATE_R
```

Code Example 16-6 - NRAPENRATE.PARAMS - using the UNION and MAP statements

Predefined Arithmetic Fortran Functions

Overview

In mathematics, a function is an expression that accepts one or more input values and calculates a single result from them. Scientific and technical calculations usually require functions that are more complex than simple addition, subtraction, and multiplication. Some of these functions are very common and are used in many different technical disciplines. Examples of these are trigonometric functions, logarithms, and square roots.

Fortran has mechanisms to support all functions no matter how common or rare. Many of the most common ones are built into the Fortran language. They are called ***intrinsic*** or ***predefined*** functions. The less common functions are left to the programmer to supply when needed. Fortran has a rich set of functions built into the language. The functions discussed on the following pages represent a small sampling of those available to the programmer. In addition, there is a large library of SuRPAS functions available.

This section discusses:

- Type conversion functions
- Rounding and truncation functions
- Real and integer mathematical functions

Type Conversion, Rounding, and Truncation Functions

The following functions provide type conversion, rounding, and truncation support within the Fortran language.

The REAL Function

The REAL function accepts an integer or real argument as input and converts the argument to a REAL type variable as output. The syntax of this function is:

real-variable = REAL (*integer-or-real-argument*)

The NINT Function

The NINT function accepts a real input argument and rounds it to a whole number integer as output. The syntax of this function is:

integer-variable = NINT (*real-argument*)

The ANINT Function

The ANINT function accepts a real input argument and rounds it to a whole number of type real as output. The syntax of this function is:

real-variable = ANINT (*real-argument*)

The INT Function

The INT function accepts a real or integer input argument and rounds it to a whole number integer as output. In this case the value is the nearest whole number that is not greater than the absolute value of the input argument (may be smaller). The syntax of this function is:

integer-variable = INT (*int-or-real-argument*)

The AINT Function

The AINT function accepts a real input argument and rounds it to a whole number of type real as output. In this case the value is the nearest whole number that is not greater than the absolute value of the input argument (may be smaller). The syntax of this function is:

real-variable = AINT (*real-argument*)

Other Real and Integer Functions

Fortran supports a number of functions that accept either real or integer arguments with the result being of the same type as the input argument. If there are two or more arguments, they must all be of the same type. The following are some of the more common functions.

The ABS Function

The ABS function accepts either a real or integer argument and supplies the absolute value of that argument (either positive or zero) as the result. The syntax of this function is:

absolute-variable = ABS (*int-or-real-argument*)

The MOD Function

The MOD function provides the remainder that results from integer division of the first argument by the second argument. (The second argument can not be zero.) The syntax of this function is:

remainder-variable = MOD (*argument1, argument2*)

The SIGN Function

The SIGN function provides a result whose magnitude is determined from the first argument and whose sign is obtained from the second argument. (The sign of the first argument is ignored.) The syntax of this function is:

sign-variable = SIGN (*argument1, argument2*)

The DIM Function

The DIM function subtracts the second argument from the first argument. If the difference is positive, it is the result of the function; if the difference is negative, the value of the function is zero. The syntax of this function is:

diff-variable = DIM (*argument1, argument2*)

The MAX Function

The MAX function evaluates the arguments (two or more) provided as input and sets the result equal to the largest input argument. The syntax of this function is:

```
large-variable = MAX (argument1, argument2,...)
```

The MIN Function

The MIN function evaluates the arguments (two or more) provided as input and sets the result equal to the smallest input argument. The syntax of this function is:

```
small-variable = MIN (argument1, argument2,...)
```

The SQRT Function

The SQRT function calculates the square root of a real input argument and provides a real result. (The input argument must be greater than or equal to zero.) The syntax of this function is:

```
real-variable = SQRT (real-input-argument)
```

Summary

Concepts

Variables and Constants

- A Fortran constant is a data object that is defined before a program begins execution and does not change its value during the execution of the program.
- A Fortran variable is a data object that can change value during the execution of a program. Each Fortran variable must have a unique name that is a maximum of 31 characters long.
- Fortran constants and variables receive their value through assignment or declaration. Each is associated with data of a specific type.
- Type declarations are made either implicitly or explicitly using the IMPLICIT statement or one of the explicit data type statements.

Expressions

- The assignment statement is one of two basic kinds of Fortran statements used to store a new value in a memory location. It consists of two parts separated by an assignment operator (equal sign). A variable is on the left side of the equal sign and an expression is on the right.
- The standard arithmetic operators included in Fortran are:
  - \+ Addition
  - \- Subtraction
  - * Multiplication
  - / Division
  - ** Exponentiation
- There are three rules for evaluating a Fortran expression:
  1. Quantities within parentheses are evaluated first. If there are nested parentheses, the innermost set is evaluated first.
  2. Arithmetic operations are evaluated in the following order:
     First: exponentiation
     Second: multiplication and division
     Third: addition and subtraction
  3. Expressions with two or more operations at the same level are evaluated from left to right.

Expressions - continued

- Logical operations yield a logical result (TRUE or FALSE). The logical operators are:

.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greater than or equal to
.LT.	Less than
.LE.	Less than or equal to
.AND.	TRUE if both are TRUE
.OR.	TRUE if either are TRUE
.EQV.	TRUE if both are the same (TRUE or FALSE)
.NEQV.	TRUE if one is TRUE and the other is FALSE
.NOT.	TRUE if operand is FALSE; FALSE if operand is TRUE

- In character operations, substrings, and strings can be combined using the concatenation operator (double slash).

Arrays and Array Elements

- An array is a group of variables or constants, all of the same type, that are referenced by a single symbolic name. The values in the group occupy consecutive memory locations.

- An individual value within the array is called an array element and is identified by the name of the array together with a subscript number pointing to a particular location within it.

- When a list of array elements is sequential we refer to it as a ***linear array***. Linear arrays can be visualized as a series of values laid out in a line or column.

- Two-dimensional arrays (arrays with two independent variables) can easily be supported by Fortran through the use of two subscripts attached to the declared variable. (This is also called a ***matrix.***)

DATA and PARAMETER Statements

- A DATA declaration specifies data values that are to be assigned initially and stored in the memory locations of a variable prior to execution of a program.

- An implied DO loop may be used within a DATA statement to populate numerous elements of an array.

- A PARAMETER declaration creates a named constant that can be used in a program in the same way as a constant.

Commands

Variables and Constants

IMPLICIT
Default implicit type declarations

INTEGER*2, INTEGER*4, INTEGER *8
Explicit declaration of integer type variables

REAL*4
Explicit declaration of real type variables

REAL*8, DOUBLE PRECISION
Explicit declaration of double precision variables

CHARACTER*_n_
Explicit declaration of character variables

LOGICAL
Explicit declaration of logical/boolean variables

Array and Array Elements

DIMENSION _variable-list_
Provides size information to a previously declared data variable

DATA and PARAMETER Declarations

DATA _list-of-items_ / _list-of-contents_ /
Specifies data values that are to be assigned to variables

PARAMETER (_x_=_n_)
Creates a named constant that can be used in a program

User-defined Structures

STRUCTURE [/_structure_name_/] [_field_namelist_]
_field_declaration_
[_field_declaration_]
...
END STRUCTURE
Specifies the format andlayout of a user-defined structure

RECORD /_structure_name_/ _record_namelist_
[,/_structure_name_/ _record_namelist_]...
Creates a user-defined record with a layout as previously defined by the mentioned structure

User-defined Structures (continued)

UNION
map_declaration
map_declaration
...
END UNION
Creates a user-defined UNION
MAP
field_declaration
field_declaration
...
END MAP
Creates a user-defined MAP