Chapter 4

Specification Statements

This chapter contains the following subsections:

- "AUTOMATIC, STATIC"
- "BLOCK DATA"
- "COMMON"
- "DATA"
- "Data Type Statements"
- "DIMENSION"
- "EQUIVALENCE"
- "EXTERNAL"
- "IMPLICIT"
- "INTRINSIC"
- "NAMELIST"
- "PARAMETER"
- "POINTER"
- "PROGRAM"
- "RECORD"
- "SAVE"
- "STRUCTURE / UNION"
- "VOLATILE"

Specification statements are non−executable Fortran statements that provide the processor with information about the nature of specific data and the allocation of storage space for this data.

The specification statements are summarized below.

AUTOMATIC, **STATIC**
Controls the allocation of storage to variables and the initial value of variables within called subprograms.

BLOCK DATA First statement in a block data subprogram used to assign initial values to variables and array elements in named common blocks.

COMMON Declares variables and arrays to be put in a storage area that is accessible to multiple program units, thus allowing program units to share data without using arguments.

DATA Supplies initial values of variables, array elements, arrays, or substrings.

Data type	Explicitly defines the type of a constant, variable, array, external function, statement function, or dummy procedure name. Also, can specify dimensions of arrays and the length of the character data.
DIMENSION	Specifies the symbolic names and dimension specifications of arrays.
EQUIVALENCE	Specifies the sharing of storage units by two or more entities in a program unit, thus associating those entities.
EXTERNAL	Identifies external or dummy procedure.
IMPLICIT	Changes or defines default implicit type of names.
INTRINSIC	Identifies intrinsic function or system subroutine.
NAMELIST	Associates a group of variables or array names with a unique group name.
PARAMETER	Gives a constant a symbolic name.
POINTER	Establishes pairs of variables and pointers.
PROGRAM	Defines a symbolic name for the main program.
RECORD	Creates a record in the format specified by a previously declared **STRUCTURE** statement.
SAVE	Retains the values of variables and arrays after execution of a **RETURN** or **END** statement in a subprogram.
STRUCTURE	Defines a record structure that can be referenced by one or more **RECORD** statement.
VOLATILE	Prevents the compiler from optimizing specified variables, arrays, and common blocks of data.

Detailed descriptions of these statements follow in alphabetical order.

AUTOMATIC, STATIC

STATIC and **AUTOMATIC** statements control, within a called subprogram, the allocation of storage to variables and the initial value of variables.

Syntax

`{STATIC | AUTOMATIC}` *v* `[,`*v*`]` …

where *v* is the name of a previously declared variable, array, array declarator, symbolic constant, function, or dummy procedure.

Method of Operation

Table 4−1summarizes the differences between *static* and *automatic* variables on entry and exit from a subprogram.

Table 4–1 Static and Automatic Variables

	AUTOMATIC	STATIC
Entry	Variables are unassigned. They do not reflect any changes caused by the previous execution of the subprogram.	Values of the variables in the subprogram are unchanged since the last execution of the subprogram.
Exit	The storage area associated with the variable is deleted.	The current value of the variable is retained in the static storage area.

AUTOMATIC variables have two advantages:

- The program executes more efficiently by taking less space and reducing execution time.
- They permit recursion; a subprogram can call itself either directly or indirectly, and the expected values are available on either a subsequent call or a return to the subprogram.

Rules for Use

- By default, unless you specify the **-static** command line option (described in the *f77*(1) manual page and Chapter 1 of the *Fortran 77 Programmer's Guide*), all variables are **AUTOMATIC** except
  - initialized variables
  - common blocks
  - variables used in **EQUIVALENCE** statements
- Override the command line option in effect for specific variables by specifying as applicable the **AUTOMATIC** or **STATIC** keywords in the variable type statements, as well as in the **IMPLICIT** statement.
- Any variable in **EQUIVALENCE**, **DATA**, or **SAVE** statements is static regardless of any previous **AUTOMATIC** specification.

Example

```
REAL   length, anet, total(50)
STATIC length, anet, total
COMPLEX i, B(20), J(2,3,5)
STATIC i
IMPLICIT INTEGER(f,m-p)
IMPLICIT STATIC (f,m-p)
```

BLOCK DATA

BLOCK DATA is the first statement in a block data subprogram. It assigns initial values to variables and array elements in named common blocks.

Syntax

```
BLOCK DATA [sub]
```

where *sub* is the symbolic name of the block data subprogram in which the **BLOCK DATA** statement appears.

Method of Operation

A block data subprogram is a non−executable program unit with a**DATA** statement as its first statement, followed by a body of specification statements and terminated by an **END** statement. The types of specification statements include **COMMON**, **DATA**, **DIMENSION**, **EQUIVALENCE**, **IMPLICIT**, **PARAMETER**, **SAVE**, **STRUCTURE** declarations, and type statements. A block data subprogram can also contain comment lines.

Only entities in named common blocks or entities associated with an entity in a common block can be initially defined in a block data subprogram.

Rules for Use

- The optional name *sub* is a global name and must be unique. Thus, **BLOCK DATA** subprograms cannot have the same external name.
- An executable program can contain more than one block data subprogram but cannot contain more than one unnamed block data subprogram.
- A single block data subprogram can initialize the entities of more than one named common block.

COMMON

The **COMMON** statement declares variables and arrays so that they are put in a storage area that is accessible to multiple program units, thus allowing program units to share data without using arguments.

Syntax

```
COMMON [/[cb]/] nlist [[,]/[cb]/ nlist]...
```

where

cb	is a common block name.
nlist	is a list of variable names, array names, array declarators, or records.

Method of Operation

A storage sequence, composed of a series of storage units that are shared between program units, is referred to as *common storage*. For each common block, a common block storage sequence is formed consisting of the storage sequences of all entities in the list of variables and arrays for that common block. The order of the storage sequence is the same as its order of appearance in the list. In each**COMMON** statement, the entities specified in the common block list *nlist* following a block name *cb* are declared to be in common block *cb*.

In an executable program, all common blocks with the same name have the same first storage unit. This establishes the association of data values between program units.

The storage sequence formed above is extended to include all storage units of any storage sequence associated with it by equivalence association.

Fortran has the following types of common storage:

- *Blank* common storage does not have an identifying name and can be accessed by all program units in which it is declared. One blank common area exists for the complete executable program.
- *Named* common storage has an identifying name and is accessible by all program units in which common storage with the same name is declared.

You can initially define entities in a named common block by using the **DATA** initialization statement in a **BLOCK DATA** subprogram. However, you *cannot* use the **DATA** statement to initialize entities in blank common block.

The number of storage units needed to store a common block is referred to as its size. This number includes any extensions of the sequence resulting from equivalence association. The size of a named common block must be the same in *all* program units in which it is declared. The size of blank common block need *not* be the same size in all program units.

Rules for Use

- A variable name, array name, array declarator, or record can appear only once in all common block lists within a program unit.
- Specify a blank common block by omitting the common block name *cb* for each list. Thus, omitting the first common block name places entities appearing in the first *nlist* in a blank common block.
- Omitting the first *cb* makes the first two slashes optional. Two slashes without a block name between them declare the entities in the following list to be in a blank common block.
- Any common block name *cb* or an omitted *cb* for a blank common block can occur more than once in one or more **COMMON** statements in a program unit. The list following each appearance of the same common block name is treated as a continuation of the list for that common block name.
- As an extension to the standard, a named common block can be declared as having different sizes in different program units. If the common block is not initially defined with a **DATA** statement, its size will be that of the longest common block declared. However, if it is defined in one of the program units with **DATA** statements, then its size is the size of the defined common block. In other words, to work correctly, the named common block must be declared with the longest size when it is defined, even though it can be declared with shorter sizes somewhere else. Defining a common block multiple times produces incorrect results.
- The compiler aligns entities in a common block on 32−bit boundaries. You can change this alignment using the compiler switches **-align8** and **-align16**. However, these changes can degrade

performance. See the *Fortran 77 Programmer's Guide* for more information.

Restrictions

- Names of dummy arguments of an external procedure in a subprogram must not appear in a common block list.
- A variable name that is also a function name must not appear in the list.

Examples

The following equivalent statements define a blank common block. (Note that these two **COMMON** statements cannot appear in the same program unit).

```
COMMON //F,X,B(5)
COMMON F,X,B(5)
```

This declaration

```
COMMON /LABEL/NAME,AGE,DRUG,DOSE//Y(33),
 Z,/RECORD/,DOC, 4 TIME(5), TYPE(8)
```

makes the following **COMMON** storage assignments:

- **NAME**, **AGE**, **DRUG**, and **DOSE** are placed in common block **LABEL**.
- **Y** and **Z** are placed in a blank common block.
- **DOC**, **TIME**, and **TYPE** are placed in a common block **RECORD**.

The following program contains two **COMMON** statements: one in the calling program and one in the subroutine. Both define the same four entities in the **COMMON** even though each common statement uses a unique set of names. The calling program can access **COMMON** storage through the entities **TOT**, **A**, **K**, and **XMEAN**. Subroutine **ADD** has access to the same common storage through the use of the entities **PLUS**, **SUM**, **M**, and **AVG**.

```
c  THIS PROGRAM READS VALUES AND PRINTS THEM
c  SUM AND AVERAGE
   COMMON TOT, A(20), K, XMEAN
   READ (5,10) K, ( A(I), I = 1, K)
   CALL ADD
   WRITE (6,20) TOT, XMEAN
10 FORMAT (I5/F(10.0))
20 FORMAT (5X,5HSUM =,2X,F10.4/5X,
  +       12HMEAN VALUE =,2X,F10.4)
   STOP
c
c  THIS SUBROUTINE CALCULATES THE SUM AND AVERAGE
c
```

```
   COMMON PLUS, SUM(20), M, AVG
   PLUS = SUM (1)
   DO 5 I = 2, M
5  PLUS = SUM (I) + PLUS
   AVG = PLUS / FLOAT (M)
   END
```

DATA

The **DATA** statement supplies initial values of variables, array elements, arrays, or substrings.

Syntax

`DATA` *nlist/clist/* `[[ , ]` *nlist/clist/*`]` …

where

nlist is a list of variable names, array names, array element names, substring names or implied-**DO** lists (described later in this chapter) separated by commas.

clist *clist* is composed of one or more elements, separated by commas, of either of the following forms:*cr*c* where *c* is a constant or the symbolic name of a constant. *r* is a nonzero, unsigned integer constant or the symbolic name of a positive integer constant. The second form implies *r* successive appearances of the constant c.

Method of Operation

In data initialization, the first value in *clist* is assigned to the first entity in *nlist*, the second value in *clist* to the second entity in *nlist*, and so on. There is a one−to−one correspondence between the items specified by *nlist* and the constants supplied in *clist.* Hence, each *nlist* and its corresponding *clist* must contain the same number of items and must agree in data type. If necessary, the *clist* constant is converted to the type or length of the *nlist* entity exactly as for assignment statements.

If the length of the character entity in *nlist* is greater than the length of its corresponding character constant in *clist*, then blank characters are added to the right of the character constant. But if the length of the character entity in *nlist* is less than that of its corresponding constant in *clist*, the extra right most characters in the constant are ignored; only the left most characters are stored. Each character constant initializes only one variable, array element, or substring.

As an enhancement to Fortran 77, you can define an arithmetic or logical entity initially using a Hollerith constant for *c* in a *clist*, using the form

n**H***x1 x2 x3 … xn*

where

n is the number of characters *xn.*

xi is the actual characters of the entity.

The value of *n* must be > *g*, where *g* is the number of character storage units for the corresponding entity.

If $n < g$, the entity is initially defined with the n Hollerith characters extended on the right with $g - n$ blank characters. The compiler generates a warning message for data initializations of this type.

Rules for Use

- Each *nlist* and its corresponding *clist* must have the same number of items and must correspond in type when either is **LOGICAL** or **CHARACTER**. If either is of arithmetic type, then the other must be of arithmetic type.
- If an unsubscripted array name is specified in *nlist*, the corresponding *clist* must contain one constant for each element of the array.
- If two entities are associated in common storage, only one can be initialized in a **DATA** statement.
- Each subscript expression in *nlist* must be an integer constant expression, except for implied-**DO** variables.
- Each substring expression in *nlist* must be an integer constant expression.
- A numeric value can be used to initialize a character variable or element. The length of that character variable or array element must be one, and the value of the numeric initializer must be in the range 0 through 255.
- An untyped hexadecimal, octal, or binary constant can be used to initialize a variable or array element. If the number of bits defined by the constant is less than the storage allocation for that variable or array element, then leading zeros are assumed. If the number of bits exceed the number of bits of storage available for the variable or array element, then the leading bits of the constant are truncated accordingly.
- A Hollerith constant can be used to initialize a numeric variable or array element. The rules for Hollerith assignment apply.

Restrictions

- The list *nlist* cannot contain names of dummy arguments, functions, and entities in blank common, or those associated with entities in blank common.
- Do not initialize a variable, array element, or substring more than once in an executable program. If you do, the subsequent initializations will override the previous ones.
- If a common block is initialized by a **DATA** statement in a program unit, it cannot be initialized in other program units.

Example

Given the following declarations,

```
REAL          A (4), b
LOGICAL       T
COMPLEX       C
```

```
      INTEGER       P, K(3), R
      CHARACTER*5   TEST(4)
      PARAMETER     (P=3)
      DATA   A,B/0.,12,5.12E5,0.,6/, T/.TRUE./,
 +                  C/(7.2, 1.234)/,K/P*0/,
 +                  TEST/3*'MAYBE','DONE?'/
```

the **DATA** statement above defines the variables declared immediately preceding it as follows:

```
A(1) = .0E+00    A(2) = .12E+02
A(3) = .512E+06  A(4) = .0E+00
B = 6
T = .TRUE.
C = (.72E+01, .1234+01)
K(1) = 0   K(2) = 0 K(3) = 0
TEST(1) = 'MAYBE' TEST(2) = 'MAYBE'
TEST(3) = 'MAYBE' TEST(4) = 'DONE?'
```

The following statements are examples of implied-**DO** statements using **DATA** statements:

```
DATA LIMIT /1000/, (A(I), I= 1,25)/25*0/
DATA ((A(I,J), J = 1,5), I = 1,10)/50*1.1/
DATA (X(I,I), I = 1,100) /100 * 1.1/
DATA ((A(I,J), J = 1,I), I =1,3)/11,21,22,31,32,33/
```

Data Type Statements

The data type statement explicitly defines the type of a constant, variable, array, external function, statement function, or dummy procedure name. It can also specify dimensions of arrays and the length of character data. The two kinds of data type statements are numeric and character.

Numeric Data Types

Use numeric data types to

- override implicit typing
- explicitly define the type of a constant, variable, array, external function, statement function, or dummy procedure name
- specify dimensions of arrays

Syntax

type v [**len*] [/*clist*/] [, *v*[**len*]/*clist*/]]

where

type is one of the keywords listed in Table 4–2

v	is a variable name, array name, array declarator, symbolic name of a constant, function name, or dummy procedure name.
len	is one of the acceptable lengths for the data type being declared; *len* is one of the following: an unsigned, nonzero integer constant; a positive−value integer constant expression enclosed in parentheses; or an asterisk enclosed in parentheses (*). If the type being declared is an array, *len* follows immediately after the array name.
clist	is a list of values bounded by slashes; the value becomes the initial value of the type being declared.

Table 4−2 Keywords for Type Statements

INTEGER	COMPLEX
INTEGER*1	DOUBLE COMPLEX
BYTE	COMPLEX*8
INTEGER*2	COMPLEX*16
INTEGER*4	
LOGICAL	REAL
LOGICAL*1	REAL*4
LOGICAL*2	REAL*8
LOGICAL*4	REAL*16
DOUBLE PRECISION	

Note: When the complier encounters a **REAL*16** declaration, it issues a warning message. **REAL*16** items are allocated 16 bytes of storage per element, but only the first 8 bytes of each element are used; those 8 bytes are interpreted according to the format for **REAL*8** floating numbers.

The following pairs of keywords are synonymous:

- **BYTE** and **INTEGER*1**
- **REAL** and **REAL*4**
- **DOUBLE PRECISION** and **REAL*8**
- **COMPLEX** and **COMPLEX*8**
- **DOUBLE COMPLEX** and **COMPLEX*16**
- **LOGICAL** and **LOGICAL*4**

See Chapter 2 of the *Fortran 77 Programmer's Guide* for information on the alignment, size, and value ranges of these data types.

Method of Operation

The symbolic name of an entity in a type statement establishes the data type for that name for all its subsequent appearances in the program unit in which it is declared.

The *type* specifies the data type of the corresponding entities. That is, the **INTEGER** statement explicitly

The *type* specifies the data type of the corresponding entities. That is, the **INTEGER** statement explicitly declares entities of type integer and overrides implicit typing of the listed names. The **REAL** statement specifies real entities, the **COMPLEX** statement specifies complex entities, and so on.

Rules for Use

- Type statements are optional and must appear in the beginning of a program unit. However, type statements can be preceded by an **IMPLICIT** statement.
- Symbolic names, including those declared in type statements, have the scope of the program unit in which they are included.
- A program unit can contain type statements that begin with identical keywords.
- Do not explicitly specify the type of a symbolic name more than once within a program unit.
- Do not use the name of a main program, subroutine, or block data subprogram in a type statement.
- The compiler provides a **DOUBLE COMPLEX** version of the functions in Table 4–3

Table 4–3 Double Complex Functions

Name	Purpose
dcmplx	Explicit type conversion
dconjg	Complex conjugate
dimag	Imaginary part of complex argument
zabs	Complex absolute value

- The **-i2** compiler option (see the *f77*(1) man page or Chapter 2 of the *Fortran 77 Programmer's Guide*) causes the following to take place:
  - converts integer constants whose values are within the range allowed for the **INTEGER*2** data types to **INTEGER*2**
  - converts the data type of variable returned by a function to **INTEGER*2**, where possible
  - ensures that variables of type **LOGICAL** occupy the same amount of storage as **INTEGER*2** variables

Examples

```
REAL length, anet, TOTAL(50)
INTEGER hour, sum(5:15), first, uvr(4,8,3)
LOGICAL bx(1:15,10), flag, stat
COMPLEX I, B(20), J(2,3,5)
```

The code above declares that

- **length** and **anet** are names of type real. The specification of **anet** confirms implicit typing using the first letter of the name and could have been omitted in the **REAL** statement.
- **TOTAL** is a real array.
- **hour** and **first** are integer names. **uvr** and **sum** are integer arrays and illustrate the use of the type

hour and **first** are integer names. **uvr** and **sum** are integer arrays and illustrate the use of the type statement to specify the dimensions of an array. Note that when an array is dimensioned in a type statement, a separate **DIMENSION** statement to declare the array is not permitted.

- **flag** and **stat** are logical variables; **bx** is a logical array.
- **I** is a complex variable; **B** and **J** are complex arrays.

Character Data Types

Character data type statements declare the symbolic name of a constant, variable, array, external function, statement function, or dummy procedure name and specify the length of the character data.

Syntax

`CHARACTER [*`*len*` [,]] `*nam*` [,`*nam*`] . . .`

where

len — is a length specification that gives the length, in number of characters, of a character variable, character array element, character constant, or character function. *len* is one of the following:

- an unsigned, nonzero integer constant
- a positive–value integer constant expression enclosed in parentheses
- an asterisk enclosed in parentheses (*)

nam — is one of the following: *v* [**len*] where *v* is a variable name, symbolic name of a constant, function name, or dummy procedure name *a* [(*d*)] [**len*] where *a*(*d*) is an array declarator

Rules for Use

- The length specification *len* that follows the keyword **CHARACTER** denotes the length of each entity in the statement that does not have its own length specification.
- A length specification immediately following an entity applies only to that entity. The length specified when an array is declared applies to each array element.
- If no length specification is given, a length of one is assumed.
- The length specifier of (*) can be used only for names of external functions, dummy arguments of an external procedure, and character constants.
  - For a character constant, the (*) denotes that the length of the constant is determined by the length of the character expression given in the **PARAMETER** statement.
  - For a dummy argument of an external procedure, the (*) denotes that the length of the dummy argument is the length of the actual argument when the procedure is invoked. If the associated actual argument is an array name, the length of the dummy argument is the length of an element of the actual array.

- For an external function name, the (*) denotes that the length of the function result value and the local variable with the same name as the function entry name is the length that is specified in the program unit in which it is referenced. Note that the function name must be the name of an entry to the function subprogram containing this **TYPE** statement.

- If an actual *len* is declared for an external function in the referencing program unit and in the function definition, *len* must agree with the length specified in the subprogram that specifies the function. If not, then the function definition must use the asterisk (*) as covered previously, but the actual *len* in the referencing unit must not be (*).
- The length specified for a character statement function or statement function dummy argument of type character must be an integer constant expression.

Example

```
CHARACTER name*40, gender*1, pay(12)*10
```

The above declaration defines

- **name** as a character variable with a length of 40
- **gender** as a character variable with a length of one
- **pay** as a character array with 12 elements, each of which is 10 characters in length

DIMENSION

The **DIMENSION** statement specifies the symbolic names and dimension specifications of arrays.

Syntax

DIMENSION *a*(*d*) [,*a*(*d*)] ...

where *a*(*d*) is an array declarator.

To be compatible with PDP−11 Fortran, the **VIRTUAL** statement is synonymous with the **DIMENSION** statement and carries the identical meaning.

Method of Operation

A symbolic name *x* appears in a **DIMENSION** statement causing an array *x* to be declared in that program unit.

Rules for Use

- The dimension specification of an array can appear only once in a program unit.
- The name of an array declared in a **DIMENSION** statement can appear in a type statement or a **COMMON** statement without dimensioning information.

Examples

The following **DIMENSION** statement declares **z** as an array of 25 elements, **a** as an array of 36 elements (6 x 6), and **ams** as an array of 50 elements (2 x 5 x 5).

```
DIMENSION z(25), a(6,6), ams(2,5,5)
```

EQUIVALENCE

The **EQUIVALENCE** statement allows two or more entities in a program unit to share storage units, thus associating those entities. This statement allows the same information to be referenced by different names in the same program unit.

Syntax

`EQUIVALENCE (`*nlist*`) [,(`*nlist*`)] ...`

where *nlist* is a list of variable names, array element names, array names, and character substring names.

Method of Operation

The storage sequences of the entities in the list must have the same first storage unit. This requirement associates the entities in the list or other elements as well. The **EQUIVALENCE** statement only associates storage units and does not cause type conversion or imply mathematical equivalence. Thus, if a variable and an array are equivalenced, the variable does not assume array properties and vice versa.

Character entities can be associated by equivalence only with other character entities. Specify the character entities, character variables, character array names, character array element names, or character substring names. Association is made between the first storage units occupied by the entities appearing in the equivalence list of an **EQUIVALENCE** statement. This statement can associate entities of other character elements as well. The lengths of the equivalenced character entities are not required to be equal.

Variables and arrays can be associated with entities in common storage to lengthen the common block. However, association through the use of the **EQUIVALENCE** statement must not cause common storage to be lengthened by adding storage units before the first storage unit in the common block.

Rules for Use

- Each subscript expression or substring expression in an equivalence list must be an integer constant expression.
- If an array element name is specified in an **EQUIVALENCE** statement, the number of subscript expressions must be the same as the number of dimensions declared for that array.
- An array name without a subscript is treated as an array element name that identifies the first element of the array.
- Multidimensional array elements can be referred to in an **EQUIVALENCE** statement with only one subscript. The compiler considers the array to be one–dimensional according to the array element ordering of Fortran. Consider the following example:

```
DIMENSION a(2,3), b(4:5,2:4)
```

The following shows a valid **EQUIVALENCE** statement using the arrays **a** and **b**:

```
EQUIVALENCE (a(1,1), b(4,2))
```

The following example achieves the same effect:

```
EQUIVALENCE (a(1), b(4))
```

The lower−bound values in the array declaration are always assumed for missing subscripts (in the above example, 1 through 3 for array **a** and 2 through 4 for array **b**).

Restrictions

- Names of dummy arguments of an external procedure in a subprogram cannot appear in an equivalence list.
- A variable name that is also a function name cannot appear in the list.
- A storage unit can appear in no more than one **EQUIVALENCE** storage sequence.
- An **EQUIVALENCE** statement cannot specify non−consecutive storage positions for consecutive storage units.
- An **EQUIVALENCE** statement cannot associate a storage unit in one common block with any storage unit in a *different* common block.

Example 1

The two statements below are represented in storage as shown in Figure 4−1

```
DIMENSION M(3,2),P(6)
EQUIVALENCE (M(2,1),P(1))
```

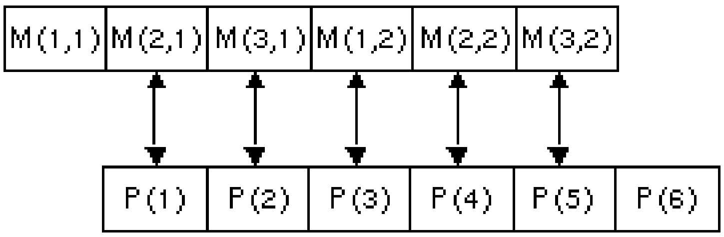

Figure 4−1 Storage Representation of an EQUIVALENCE Statement

Example 2

The two statements below cause the logical representation in storage shown in Figure 4−2

```
CHARACTER ABT*6, BYT(2)*4, CDT*3
EQUIVALENCE (ABT, BYT(1)),(CDT, BYT(2))
```

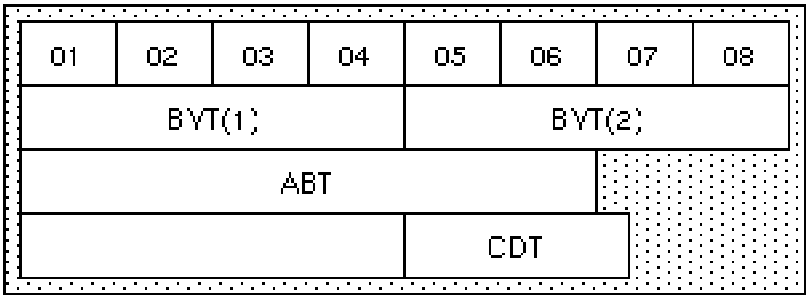

Figure 4–2 Logical Representation of an EQUIVALENCE Statement

Example 3

The following statements are invalid because they specify non–consecutive storage positions for consecutive storage units.

```
REAL A(2)
DOUBLE PRECISION S(2)
EQUIVALENCE (A(1), S(1)), (A(2), S(2))
```

Note that a double–precision variable occupies two consecutive numeric storage units in a storage sequence.

EXTERNAL

The **EXTERNAL** statement specifies a symbolic name to represent an external procedure or a dummy procedure. The symbolic name can then be used as an actual argument in a program unit.

Syntax

`EXTERNAL` *proc* `[,`*proc*`] ...`

where *proc* is the name of an external procedure or dummy procedure.

Rules for Use

- An external procedure name or a dummy procedure name must appear in an **EXTERNAL** statement in the program unit if the name is to be used as an actual argument in that program unit.
- If an intrinsic function name appears in an **EXTERNAL** statement, indicating the existence of an external procedure having that name, the intrinsic function is not available for use in the same program unit in which the **EXTERNAL** statement appears.
- A symbolic name can appear only once in all the **EXTERNAL** statements of a program unit.

- A **NOF77** qualifier in an **OPTIONS** statement or the **-nof77** command line option permits the following:
  - Intrinsic function names can appear in the list of subprograms in an **EXTERNAL** statement.
  - An asterisk (*) can precede program names listed in an **EXTERNAL** statement. This indicates that a user–supplied function has the same name as a Fortran intrinsic function and that the user–supplied function is to be invoked.

Restriction

Do not specify a statement function name in an **EXTERNAL** statement.

Example

Consider the following statements:

```
EXTERNAL G
CALL SUB1 (X,Y,G)
```

and the corresponding subprogram:

```
SUBROUTINE SUB1 (RES, ARG, F)
RES = F(ARG)
END
```

The dummy argument **F** in subroutine **SUB1** is the name of another subprogram; in this case, the external function **G**.

IMPLICIT

The **IMPLICIT** statement changes or defines default–implicit types of names. This section explains the three syntactic forms of the **IMPLICIT** statement.

Syntax 1

`IMPLICIT` *typ* `(`*a*`[,`*a*`]...) [,`*typ*`(`*a*`[,`*a*`]...)]...`

where

typ	is a valid data type.
a	is either a single *alphabetic* character or a range of letters in alphabetical order. A range of letters is specified as *l*1 - *l*2, where *l*1 and *l*2 are the first and last letters of the range, respectively.

An **IMPLICIT** statement specifies a type for all variables, arrays, external functions, and statement functions for which no type is explicitly specified by a type statement. If a name has not appeared in a type statement, then its type is implicitly determined by the first character of its name. The **IMPLICIT** statement establishes which data type (and length) will be used for the indicated characters.

By default, names beginning with the alphabetic characters A through H or O through Z are implicitly typed **REAL**; names beginning with I, J, K, L, M, or N are implicitly typed **INTEGER**. Use the **IMPLICIT** statement to change the type associated with any individual letter or range of letters.

An **IMPLICIT** statement applies only to the program unit that contains it and is overridden by a type statement or a **FUNCTION** statement in the same subprogram.

Syntax 2

```
IMPLICIT {AUTOMATIC | STATIC} (a[,a]...)
  [,typ (a[,a]...)]
```

An **AUTOMATIC** or **STATIC** keyword in an **IMPLICIT** statement causes all associated variables to be assigned automatic or static storage characteristics. See the description of the **AUTOMATIC** and **STATIC** statements earlier in this chapter for information on their function. An example using these keywords is also given.

Syntax 3

```
IMPLICIT {UNDEFINED | NONE}
```

Note: UNDEFINED and **NONE** are synonymous and, therefore, perform the same function.

When a type is not declared explicitly for a variable, the implicit data typing rules cause a default type of **INTEGER** to apply if the first letter of the variable is i, j, k, l, m, or n or **REAL** if the first letter is any other alphabetic character.

Use the **IMPLICIT UNDEFINED** statement, **IMPLICIT NONE** statement, or the **-u** command line option to turn off the implicit data typing.

Using Syntax 3 of the **IMPLICIT** statement within a program allows you to override the default assignments given to individual characters; the **-u** command line option (see Chapter 1 of the *Fortran 77 Programmer's Guide*) overrides the default assignments for all alphabetic characters.

The following declaration

```
IMPLICIT UNDEFINED
```

turns off the implicit data typing rules for all variables. The example has the same effect as specifying the **-u** command line option.

Rules for Use

The following rules are for all three syntactic forms of the **IMPLICIT** statement.

- **IMPLICIT** statements must precede all other specification statements except **PARAMETER** statements.
- Multiple **IMPLICIT** statements are allowed in a program unit.
- **IMPLICIT** statements cannot be used to change the type of a letter more than once inside a program unit. Because letters can be part of a range of letters as well as stand alone, ranges of letters cannot overlap.

- Lowercase and uppercase alphabetic characters are *not* distinguished. Implicit type is established for *both* the lower– and uppercase alphabetic characters or range of alphabetic characters regardless of the case of *l*1 and *l*2.
- The **-u** command line option turns off all default data typing and any data typing explicitly specified by an **IMPLICIT** statement.

Examples

Consider the following example:

```
IMPLICIT NONE
IMPLICIT INTEGER (F,M−P)
IMPLICIT STATIC (F,M−P)
IMPLICIT REAL (B,D)
INTEGER bin, dale
```

The previous statements declare that

- All variables with names beginning with the letters F(f), M(m), N(n), O(o), or P(p) are of type **INTEGER** and are assigned the **STATIC** attribute.
- All variables with names beginning with the letter B(b) or D(d) are of type **REAL**, except for variables **bin** and **dale**, which are explicitly defined as type **INTEGER**.

The following four **IMPLICIT** statements are equivalent:

```
IMPLICIT CHARACTER (g − k)
IMPLICIT CHARACTER (g − K)
IMPLICIT CHARACTER (G − k)
IMPLICIT CHARACTER (G − K)
```

INTRINSIC

INTRINSIC statements associate symbolic names with intrinsic functions and system subroutines. The name of an intrinsic function can be used as an actual argument.

Syntax

`INTRINSIC` *func* `[,`*func*`] ...`

where *func* is a name of intrinsic functions.

Rules for Use

- The name of every intrinsic function or system subroutine used as an actual argument must appear in an **INTRINSIC** statement in that program unit.
- A symbolic name can appear only once in all of the **INTRINSIC** statements of a program unit.

Restrictions

- The same name cannot appear in both an **INTRINSIC** and an **EXTERNAL** statement in the same program unit.
- The same name can appear only once in all the **INTRINSIC** statements of a program unit.
- The names of intrinsic functions that perform type conversion, test lexical relationship, or choose smallest/largest value cannot be passed as actual arguments. These functions include the conversion, maximum−value, and minimum−value functions listed in Appendix A, "Intrinsic Functions."

Examples

Consider the following statements:

```
INTRINSIC ABS
CALL ORD (ABS, ASQ, BSQ)
```

and the corresponding subprogram:

```
SUBROUTINE ORD(FN,A,B)
A = FN (B)
RETURN
END
```

In the above example, the **INTRINSIC** statement allows the name of the intrinsic function **ABS** (for obtaining the absolute value) to be passed to subprogram **ORD**.

NAMELIST

The **NAMELIST** statement associates a group of variables or array names with a unique group−name in a namelist−directed I/O statement.

Syntax

`NAMELIST /`*group−name*`/`*namelist*`[,] /`*group−name*`/ `*namelist*`...`

where *group−name* is the name to be associated with the variables or array names defined in *namelist*. Each item in *namelist* must be separated by a comma.

Rules for Use

- The items in *namelist* are read or written in the order they are specified in the list.
- The items can be of any data type, which can be specified either explicitly or implicitly.
- The following items are not permitted in *namelist*:
  - dummy arguments
  - array elements

– character substrings

– records

– record fields

See also the description of the **READ** and **WRITE** statements in Chapter 8, "Input/Output Statements," for more information on namelist–directed I/O.

Examples

In the following statement, **input**, when specified to a namelist–directed I/O statement, refers to **item** and **quantity**; likewise, **output** refers to **item** and **total**:

```
NAMELIST /input/ item, quantity /output/ item, total
```

PARAMETER

The **PARAMETER** statement assigns a symbolic name to a constant.

Syntax

Format 1

PARAMETER (*p*=*e* [,*p*=*e*] ...)

Format 2

PARAMETER *p*=*e* [,*p*=*e*] ...

where

p	is a symbolic name.
e	is a constant, constant expression, or the symbolic name of a constant.

Method of Operation

The value of the constant expression e is given the symbolic name *p*. The statement defines *p* as the symbolic name of the constant. The value of the constant is the value of the expression *e* after conversion to the type name *p*. The conversion, if any, follows the rules for assignment statements.

Format 1, which has bounding parentheses, causes the symbolic name to be typed either of the following ways:

- According to a previous explicit type statement.
- If no explicit type statement exists, the name is typed according to its initial letter and the implicit rules in effect. See the description of the **IMPLICIT** statement in "IMPLICIT" for details.

Format 2, which has no bounding parentheses, causes the symbolic name to be typed by the form of the actual constant that it represents. The initial letter of the name and the implicit rules do not affect the data type.

A symbolic name in a **PARAMETER** statement has the scope of the program unit in which it was declared.

Rules for Use

- If *p* is of type integer, real, double precision, or complex, *e* must be an arithmetic constant expression.
- If *p* is of type character or logical, *e* must be a character constant expression or a logical constant expression, respectively.
- If a named constant is used in the constant expression *e*, it must be previously defined in the same **PARAMETER** or a preceding **PARAMETER** statement in the same program unit.
- A symbolic name of a constant must be defined only once in a **PARAMETER** statement within a program unit.
- The data type of a named constant must be specified by a type statement or **IMPLICIT** statement before its first appearance in a **PARAMETER** statement if a default implied type is not to be assumed for that symbolic name.
- Character symbolic named constants must be specified as type character in a **CHARACTER** statement, or the first letter of the name must appear in an **IMPLICIT** statement with the type **CHARACTER**. Specification must be made before the definition of the name in the **PARAMETER** statement.
- Once a symbolic name is defined, it can be used as a primary in any subsequent expressions or **DATA** statements in that program unit.
- The functions **IAND**, **IOR**, **NOT**, **IEOR**, **ISHFT**, **LGE**, **LGT**, **LLE**, and **LLT** with constant operands can be specified in a logical expression.
- The function **CHAR** with a constant operand can be specified in a character expression.
- The functions **MIN**, **MAX**, **ABS**, **MOD**, **ICHAR**, **NINT**, **DIM**, **DPROD**, **CMPLX**, **CONJG**, and **IMAG** with constant operands can be specified in arithmetic expressions.
- Symbolic names cannot specify the character count for Hollerith constants.
- Symbolic constants can appear in a **FORMAT** statement only within the context of a general expression bounded by angle brackets (<>).
- Symbolic constants cannot appear as part of another constant except when forming the real or imaginary part of a complex constant.

Restrictions

A constant and a symbolic name for a constant are generally not interchangeable. For example, a symbolic name of an integer constant cannot be used as a length specification in a **CHARACTER** type statement without enclosing parentheses. For instance, **CHARACTER*(I)** is valid, but **CHARACTER*I** is not.

However, a symbolic name of a constant can be used to form part of another constant, such as a complex constant, by using an intrinsic function as shown below:

```
complex c
real r
parameter (r = 2.0)
parameter (c = cmplx(1.0,r))
```

Examples

The following statements declare that 1 is converted to 1E0, making **X** the name of a **REAL** constant:

```
REAL X
PARAMETER (X = 1)
```

The following example converts 3.14 to 3, making **I** the name of an **INTEGER** constant:

```
INTEGER I
PARAMETER (I = 3.14)
```

The following example assigns the constant value of .087769 to **interest_rate**:

```
REAL*4 interest_rate
PARAMETER (interest_rate = .087769)
```

The same result could be achieved using Format 2 as follows:

```
PARAMETER interest_rate = .087769
```

POINTER

The **POINTER** statement establishes pairs of variables and pointers where each pointer contains the address of its paired variable.

Syntax

POINTER (*p1*,*v1*) [,(*p2*,*v2*) ...]

where

v1 and *v2*	are pointer−based variables.
p1 and *p2*	are the corresponding pointers. The pointer integers are automatically typed that way by the compiler. The pointer−based variables can be of any type, including structures. Even if there is a size specification in the type statement, no storage is allocated when such a pointer−based variable is defined.

Rules for Use

- Once you have defined a variable as based on a pointer, you must assign an address to that pointer. Reference the pointer−based variable with standard Fortran, and the compiler does the referencing by the pointer. (Whenever your program references a pointer–based variable, that variable's address is taken from the associated pointer.) Provide an address of a variable of the appropriate type and size.

- You must provide a memory area of the right size, and assign the address to a pointer, usually with the normal assignment statement or data statement, because no storage is allocated when a pointer–based variable is defined.

Restrictions

- A pointer–based variable cannot be used as a dummy argument or in**COMMON**, **EQUIVALENCE**, **DATA**, or **NAMELIST** statements.
- A pointer–based variable cannot itself be a pointer.
- The dimension expressions for pointer–based variables must be constant expressions in main programs. In subroutines and functions, the same rules apply for pointer–based variables as for dummy arguments. The expression can contain dummy arguments and variables in **COMMON** statements. Any variable in the expressions must be defined with an integer value at the time the subroutine or function is called.

Example

The following program uses a **POINTER** statement:

```
pointer (ptr,v), (ptr2, v2)
character a*12, v*12, z*1, v2*12
data a/'abcdefghijkl'/
ptr = loc (a)
ptr = ptr +4
ptr2 = malloc (12)
v2 = a
z = v (1:1)
print *, z
z = v2(5:5)
print *, z
call free (ptr2)
end
```

PROGRAM

The **PROGRAM** statement defines a symbolic name for the main program.

Syntax

PROGRAM *pgm*

where *pgm* is a symbolic name of the main program, which cannot be the name of an external procedure, block data subprogram, or common block or a local name in the same program unit.

Rules for Use

- The **PROGRAM** statement is optional. However, it must be the first statement in the main program when used.
- The symbolic name must be unique for that executable program. It must not be the name of any entity within the main program or any subprogram, entry, or common block.

RECORD

The **RECORD** statement creates a record in the format specified by a previously declared **STRUCTURE** statement. The effect of a **RECORD** statement is comparable to that of an ordinary type declaration.

Syntax

RECORD /*structure−name*/ *record−name* [, *record−name*]
  [, *record−name*] ... [/*structure−name*/
  record−name [, *record−name*] [, *record−name*] ...] ...

where

structure−name is the name of a previously declared structure (see the description of the **STRUCTURE** statement in "STRUCTURE / UNION").

record−name is a variable, an array, or an array declarator.

Method of Operation

The *record−name* can be used in **COMMON** and **DIMENSION** statements but not in **DATA**, **EQUIVALENCE**, **NAMELIST**, or **SAVE** statements. Records created by the **RECORD** statement are initially undefined unless the values are defined in the related structure declaration.

Examples

In the following statements, the record **latest** has the format specified by the structure **weather**; **past** is an array of 1,000 records, each record having the format of the structure **weather**.

```
STRUCTURE /weather/
    integer              month, day, year
    character*40   clouds
    real                 rainfall
end structure
record /weather/ latest, past (1000)
```

Individual items in the structure can be referenced using *record−name* and the name of the structure item. For example

```
past(n).rainfall = latest.rainfall
```

where **n** represents a number from 1 to 1,000 specifying the target array element. See the description of the **STRUCTURE** statement in this chapter for an example of how to declare a structure format.

SAVE

The **SAVE** statement retains the values of variables and arrays after execution of a **RETURN** or **END** statement in a subprogram. Therefore, those entities remain defined for subsequent invocations of the subprogram.

Syntax

```
SAVE [a[,a]...]
```

where *a* is one of the following:

- A variable or array name
- A common block name, preceded and followed by slashes

Method of Operation

The **SAVE** statement prevents named variables, arrays, and common blocks from becoming undefined after the execution of a **RETURN** or **END** statement in a subprogram. Normally, all variables and arrays become undefined on exit from a subprogram, except when they are

- specified by a **SAVE** statement
- defined in a **DATA** statement
- used in an **EQUIVALENCE** statement
- contained in a blank common
- contained in a named common that is declared in the subprogram and in a calling program unit in **SAVE** statements

All variables and arrays declared in the main program maintain their definition status throughout the execution of the program. If a local variable or array is not in a common block and is specified in a **SAVE** statement, it has the same value when the next reference is made to the subprogram.

All common blocks are treated as if they had been named in a **SAVE** statement. All data in any common block is retained on exit from a subprogram.

Note: Default **SAVE** status for common blocks is an enhancement to Fortran 77. In Fortran 77, a common block named without a corresponding **SAVE** statement causes the variables and arrays in the named common block to lose their definition status on exit from the subprogram.

Rules for Use

- A **SAVE** statement without a list is treated as though all allowable entities from that program unit were specified on the list.

- The main program can contain a **SAVE** statement, but it has no effect.
- A given symbolic name can appear in only one **SAVE** statement in a program unit.

Restrictions

Procedure names and dummy arguments cannot appear in a **SAVE** statement. The names of individual entries in a common block are not permitted in a **SAVE** statement.

Examples

The following statements are examples of **SAVE** statements:

```
SAVE L, V
SAVE /DBASE/
```

STRUCTURE / UNION

The **STRUCTURE** statement defines a record structure that can be referenced by one or more **RECORD** statement.

Syntax (General)

STRUCTURE [/*structure−name*/] [*field−names*]
    [*field−definition*]
    [*field−definition*] ...
END STRUCTURE

where

structure−name identifies the structure in a subsequent **RECORD** statement. Substructures can be established within a structure by means of either a nested **STRUCTURE** declaration or a **RECORD** statement.

field−names (for substructure declarations only) one or more names having the structure of the substructure being defined.

field−definition can be one or more of the following:

- Typed data declarations, which can optionally include one or more data initialization values.
- Substructure declarations (defined by either **RECORD** statements or subsequent **STRUCTURE** statements).
- **UNION** declarations, which are mapped fields defined by a block of statements. The syntax of a **UNION** declaration is described below.
- **PARAMETER** statements, which do not affect the form of the structure.

UNION Declaration Syntax

A **UNION** declaration is enclosed between **UNION** and **END UNION** statements, which contain two more map declarations. Each map declaration is enclosed between **MAP** and **END MAP** statements.

```
UNION
   MAP
           [field−definition] [field−definition] ...
   END MAP
   MAP
           [field−definition] [field−definition] ...
   END MAP
   [MAP
           [field−definition] [field−definition] ...
   END MAP] ...
 END UNION
```

Method of Operation

- Typed data declarations (variables or arrays) in structure declarations have the form of normal Fortran typed data declarations. Data items with different types can be freely intermixed within a structure declaration.
- Unnamed fields can be declared in a structure by specifying the pseudo name **%FILL** in place of an actual field name. You can use this mechanism to generate empty space in a record for purposes such as alignment.
- All mapped field declarations that are made within a **UNION** declaration share a common location within the containing structure. When initializing the fields within a **UNION**, the final initialization value assigned overlays any value previously assigned to a field definition that shares that field.

Examples (General)

```
structure /weather/
    integer              month, day, year
    character*20   clouds
    real                   rainfall
end structure
record /weather/ latest
```

In the preceding example, the **STRUCTURE** statement produces the storage mapping, shown in Figure 4−3, for the *latest* specification in the **RECORD** statement.

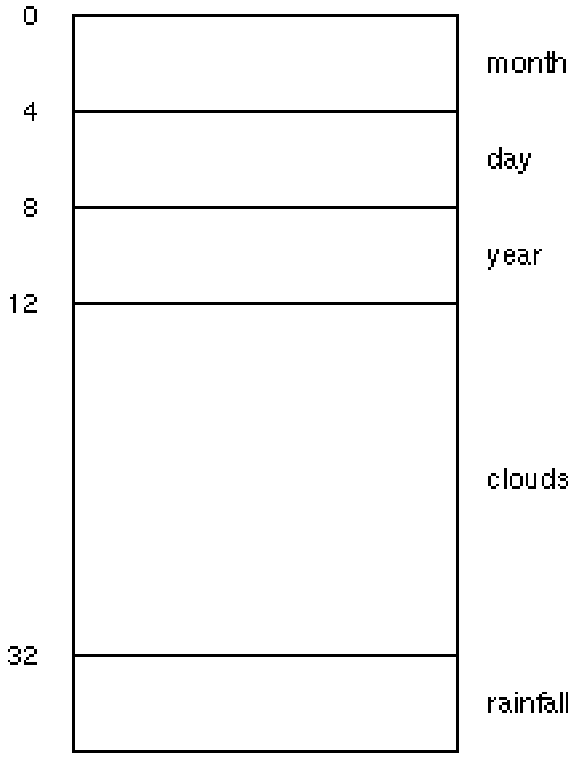

Figure 4–3 Logical Representation of a STRUCTURE Statement

The following gives an example of initializing the fields within a structure definition block:

```
   program weather
   structure /weather/
          integer*1     month /08/, day /10/, year /89/
          character*20  clouds /' overcast'/
          real   rainfall /3.12/
   end structure
   record /weather/ latest
   print *, latest.month, latest.day, latest.year,
 +  latest.clouds, latest.rainfall
```

The above example prints the following:

```
8 10 89 overcast  3.120000
```

Examples (UNION)

```
   program writedate
   structure /start/
          union
                  map
```

```
                    character*2 month
                    character*2 day
                    character*2 year
                end map
                map
                    character*6 date
                end map
            end union
        end structure
        record /start/ sdate
        sdate.month =       '08'
        sdate.day =   '10'
        sdate.year =  '89'
        write (*, 10) sdate.date
10      format (a)
        stop
        end
```

In the above example, text is written to the standard I/O device as follows:

```
081089
```

VOLATILE

The **VOLATILE** statement prevents the compiler from optimizing specified variables, arrays, and common blocks of data.

Syntax

`VOLATILE` *volatile−items*

where *volatile−items* is one or more names of variables, common blocks, or arrays, each separated by a comma

For more information on optimization, refer to the *IRIX System Programming Guide* and the *f77(1)* manual page.