Chapter 2

Constants and Data Structures

This chapter contains the following subsections:

- "Constants"
- "Character Substrings"
- "Records"
- "Arrays"

This chapter discusses the various types of Fortran constants and provides examples of each. It also explains a few of the ways data can be structured, including character substrings, records, and arrays.

Constants

A *constant* is a data value that cannot change during the execution of a program. It can be of the following types:

- arithmetic
- logical
- character
- Hollerith
- bit

The form in which a constant is written specifies both its value and its data type. A symbolic name can be assigned for a constant using the **PARAMETER** statement. Blank characters occurring within a constant are ignored by the processor unless the blanks are part of a character constant.

The sections that follow describe the various types of constants in detail.

Arithmetic Constants

The Fortran compiler supports the following types of arithmetic constants:

- integer
- real
- double−precision
- complex

An arithmetic constant can be signed or unsigned. A signed constant has a leading plus or minus sign to denote a positive or negative number. A constant that can be either signed or unsigned is called an optionally signed constant. Only arithmetic constants can be optionally signed.

Note: The value zero is neither positive nor negative; a signed zero has the same value as an unsigned zero.

Integer Constants

An integer constant is a whole number with no decimal points; it can have a positive, negative, or zero value. Hexadecimal and octal integer constants are extensions to the standard integer constant.

Format for Integer Constants

The format for an integer constant is

sww

where

s	is the sign of the number: - for negative, + (optional) for positive.
ww	is a whole number.

In Fortran, integer constants must comply with the following rules:

- It must be a whole number, that is, without a fractional part.
- If negative, the special character minus (-) must be the leading character. The plus sign (+) in front of positive integers is optional.
- It must not contain embedded commas.

Examples of valid integer constants are

```
0 +0   +176   -1352   06310   35
```

Examples of invalid integer constants are

`2.03`	Decimal point not allowed. This is a *real constant* (described later in this chapter).
`7,909`	Embedded commas not allowed.

Hexadecimal Integer Constants

Use hexadecimal integer constants for a base 16 radix. Specify a dollar sign ($) as the first character, followed by any digit (0 through 9) or the letters A through F (either uppercase or lowercase). The following are valid examples of hexadecimal integer constants:

```
$0123456789
$ABCDEF
$A2B2C3D4
```

You can use hexadecimal integer constants wherever integer constants are allowed. Note that in mixed−mode expressions, the compiler converts these constants from type integer to the dominant type of expression in which they appear.

Octal Integer Constants

Use octal integer constants for a base 8 radix. The type of an octal integer constant is **INTEGER**, in contrast to the octal constant described in "Bit Constants". This constant is supported to provide compatibility with PDP-11 Fortran.

The format of an octal constant is as follows:

o"*string*"

where *string* is one or more digits in the range of 0 through 7.

Real Constants

A real constant is a number containing a decimal point, exponent, or both; it can have a positive, negative, or zero value.

A real constant can have the following forms:

sww.ff	Basic real constant
*sww.ff***E***see*	Basic real constant followed by a real exponent
*sww***E***see*	Integer constant followed by a real exponent

where

s	is the sign of the number: - for negative, + (optional) for positive.
ww	is a string of digits denoting the whole number part, if any.
.	is a decimal point.
ff	is a string of digits denoting the fractional part, if any.
E*see*	denotes a real exponent, where *see* is an optionally signed integer.

A basic real constant is written as an optional sign followed by a string of decimal digits containing an optional decimal point. There must be at least one digit.

A real exponent is a power of ten.

The value of a real constant is either the basic real constant or, for the forms *sww.ff***E***see* and *sww***E***see*, the product of the basic real constant or integer constant and the power of ten indicated by the exponent following the
letter **E**.

All three forms can contain more digits than the precision used by the processor to approximate the value of the real constant. See the *Fortran 77 Programmer's Guide* for information on the magnitude and precision of a real number.

Table 2–1illustrates real constants written in common and scientific notation with their corresponding **E** format.

Table 2–1 Notation Forms for Real Constants

Common Notation	Scientific Notation	Real Exponent Form
5.0	0.5*10	.5E1 or 0.5E1

364.5	3.465*102	.3645E3
49,300	4.93*104	.493E5
-27,100	-2.71*104	-.271E5
-.0018	-1.8*10-3	-.18E-2

The following real constants are equivalent:

```
5E4    5.E4    .5E5    5.0E+4    +5E04    50000.
```

Table 2−2lists examples of invalid real constants and the reasons they are invalid.

Table 2−2 Invalid Real Constants

Invalid Constant	Reason Invalid
-18.3E	No exponent following the E
E-5	Exponent part alone
6.01E2.5	Exponent part must be an integer
3.5E4E2	Only one exponent part allowed per constant
19,850	Embedded commas not allowed

Double−Precision Constants

A double−precision constant is similar to a real constant except that it can retain more digits of the precision than a real constant. (The size and value ranges of double−precision constants are given in the *Fortran 77 Programmer's Guide.*)

A double−precision constant assumes a positive, negative, or zero value in one of the following forms:

*sww***D***see* An integer constant followed by a double−precision exponent

*sww.ff***D***see* A basic real constant followed by a double−precision exponent

where

s is an optional sign.

ww is a string of digits denoting the whole number part, if any.

ff is a string of digits denoting the fractional part, if any.

D*see* denotes a double−precision exponent where*see* is an optionally signed exponent.

The value of a double−precision constant is the product of the basic real constant part or integer constant part and the power of ten indicated by the integer following the letter **D** in the exponent part. Both forms can contain more digits than those used by the processor to approximate the value of the real constant. Refer to the *Fortran 77 Programmer's Guide* for information on the magnitude and precision of a double−precision constant.

Valid forms of double−precision constants are

```
1.23456D3
8.9743D0
```

```
-4.D-10
16.8D-6
```

For example, the following forms of the numeric value 500 are equivalent:

```
5D2  +5D02  5.D2  5.D+02  5D0002
```

Table 2–3lists examples of invalid double–precision constants and the reasons they are invalid.

Table 2–3 Invalid Double–Precision Constants

Invalid Constant	**Reason Invalid**
2.395D	No exponent following the D
-9.8736	Missing D exponent designator
1,010,203D0	Embedded commas not allowed

Complex Constants

A complex constant is a processor approximation of the value of a complex number. It is represented as an ordered pair of real data values. The first value represents the real part of the complex number, and the second represents the imaginary part. Each part has the same precision and range of allowed values as real data.

A complex constant has the form (*m, n*) where *m* and *n* each have the form of a *real constant*, representing the complex value $m + n\mathbf{i}$, where **i** is the square root of -1. The form *m* denotes the real part; *n* denotes the imaginary part. Both *m* and *n* can be positive, negative, or zero. Refer to Table 2–4for examples of valid forms of complex data.

Table 2–4 Valid Forms of Complex Data

Valid Complex Constant	**Equivalent Mathematical Expression**
(3.5, -5)	3.5 -5i
(0, -1)	- i
(0.0, 12)	0 + 12i or 12i
(2E3, 0)	2000 + 0i or 2000

Table 2–5provides examples of invalid constants and lists the reasons they are invalid.

Table 2–5 Invalid Forms of Complex Data

Invalid Constant	**Reason Invalid**
(1,)	No imaginary part
(1, 2.2, 3)	More than two parts
(10, 52.D5)	Double–precision constants not allowed for either part
(1.15, 4E)	Imaginary part has invalid form

Logical Constants

Logical constants represent only the values true or false, represented by one of the forms inTable 2–6

Table 2–6 Logical Constant Values

Form	**Value**

.TRUE.	True
.FALSE.	False

Character Constants

A character constant is a string of one or more characters that can be represented by the processor. Each character in the string is numbered consecutively from left to right beginning with 1.

Note: The quotation mark (") is an extension to Fortran 77.

If the delimiter is ", then a quotation mark within the character string is represented by two consecutive quotation marks with no intervening blanks.

If the delimiter is ’, then an apostrophe within the character string is represented by two consecutive apostrophes with no intervening blanks.

Blanks within the string of characters are significant.

The length of a character constant is the number of characters, including blanks, between the delimiters. The delimiters are not counted, and each pair of apostrophes or quotation marks between the delimiters counts as a single character.

A character constant is normally associated with the **CHARACTER** data type. The Fortran 77 standard is extended (except as noted below) to allow character constants to appear in the same context as a numeric constant. A character constant in the context of a numeric constant is treated the same as a Hollerith constant.

Note: Character constants cannot be used as actual arguments to numeric typed dummy arguments.

Table 2−7provides examples of valid character constants and shows how they are stored.

Table 2−7 Valid Character Constants

Constant	Stored as
’DON’’T’	DON’T
"I’M HERE!"	I’M HERE!
’STRING’	STRING
’LMN""OP’	LMN""OP

Table 2−8lists examples of invalid character constants and the reasons they are invalid.

Table 2−8 Invalid Character Constants

Invalid Constant	Reason Invalid
’ISN.T	Terminating delimiter missing
.YES’	Mismatched delimiters
CENTS	Not enclosed in delimiters
’’	Zero length not allowed
""	Zero length not allowed

Hollerith Constants

Use Hollerith constants to manipulate packed character strings in the context of integer data types. A Hollerith constant consists of a character count followed by the letter **H** (either uppercase or lowercase) and a string of characters as specified in the character count and has the following format:

*n***H***xxx...x*

where *n* is a nonzero, unsigned integer constant and where the *x*'s represent a string of exactly *n* contiguous characters. The blank character is significant in a Hollerith constant.

Examples of valid Hollerith constants are

```
3H A
10H'VALUE = '
8H MANUAL
```

Table 2–9provides some examples of invalid Hollerith constants and the reasons they are invalid.

Table 2–9 Invalid Hollerith Constants

Invalid Constant	Reason Invalid
2H YZ	Blanks are significant; should be 3H YZ
-4HBEST	Negative length not allowed
0H	Zero length not allowed

The following rules apply to Hollerith constants:

- Hollerith constants are stored as byte strings; each byte is the ASCII representation of one character.
- Hollerith constants have no type; they assume a numeric data type and size depending on the context in which they are used.
- When used with a a binary operator, octal and hexadecimal constants assume the data type of the other operand. For example,

  ```
  INTEGER*2 HILO
  HILO = 'FF'X
  ```

- The constant is assumed to be of type **INTEGER*2** and two bytes long.
- In other cases, when used in statements that require a specific data type, the constant is assumed to be the required type and length.
- A length of four bytes is assumed for hexadecimal and octal constants used as arguments; no data type is assumed.
- In other cases, the constant is assumed to be of type **INTEGER*4**.
- When a Hollerith constant is used in an actual parameter list of a subprogram reference, the formal parameter declaration within that subprogram must specify a numeric type, not a character type.
- A variable can be defined with a Hollerith value through a **DATA** statement, an assignment statement, or a **READ** statement.
- The number of characters (*n*) in the Hollerith constant must be less than or equal to *g*, the maximum

number of characters that can be stored in a variable of the given type, where *g* is the size of the variable expressed in bytes. If $n < g$, the Hollerith constant is stored and extended on the right with *(g -n)* blank characters. (Refer to the *Fortran 77 Programmer's Guide* for the sizes of arithmetic and logical data types.)

Bit Constants

You can use bit constants anywhere numeric constants are allowed. Table 2−10shows the allowable bit constants and their format.

Table 2−10 Valid Substring Examples

Format	Meaning	Valid *substring* Characters	Maximum
b' *string*' or '*string*'b[a]	Binary	0, 1	64
O' *string*' or '*string*'oa	Octal	0 - 7	22
x' *string*' or '*string*'xa	Hexadecimal	0 - 9; a - f	16
z' *string*' or '*string*'za	Hexadecimal	0 - 9; a - f	16

[a] b, o, x, and z can be lower− or uppercase (B, O, X, Z)

The following are examples of bit constants used in a **DATA** statement.

```
integer a(4)
data a/b'1010',o'12',z'a',x'b'/
```

The above statement initializes the first elements of a four−element array to binary, the second element to an octal value, and the last two elements to hexadecimal values.

The following rules apply to bit constants:

- Bit constants have no type; they assume a numeric data type and size depending on the context in which they are used.
- When used with a binary operator, octal and hexadecimal constants assume the data type of the other operand. For example,

  ```
  INTEGER*2 HILO
  HILO = 'FF'X
  ```

  The constant is assumed to be of the **INTEGER*2** type and two bytes long.
- In other cases, when used in statements that require a specific data type, the constant is assumed to be the required type and length.
- A length of four bytes is assumed for hexadecimal and octal constants used as arguments; no data type is assumed.
- In other cases, the constant is assumed to be of the **INTEGER*4** data type.
- A hexadecimal or octal constant can specify up to 16 bytes of data.
- Constants are padded with zeros to the left when the assumed length of the constant is more than the digits specified by the constant. Constants are truncated to the left when the assumed length is less

than that of the digits specified.

Character Substrings

A character substring is a contiguous sequence of *characters* that is part of a character data item. A character substring cannot be empty; that is, it must contain at least one byte of storage. Each character is individually defined or undefined at any given time during the execution of a program.

Substring Names

A substring name defines the corresponding substring and allows it to be referenced in a character expression. A substring name has one of the following forms:

v([*e1*]:[*e2*])

a(*s*[,*s*]...) ([*e1*]:[*e2*])

where

v	is a character variable name.
a	is a character array name.
e1 and *e2*	are integer expressions, called substring expressions.You can specify a non–integer character for *e1* and *e2*. If specified, each non–integer character is converted to an integer before use; fractional portions remaining after conversion are truncated.
s	is a subscript expression.

The value *e1* specifies the left most character position of the substring relative to the beginning of the variable or array element from which it was abstracted, while *e2* is the right most position. Positions are numbered left to right beginning with 1. For example, the following denotes characters in positions three through five of the character variable **EX**:

```
EX(3:5)
```

The following specifies characters in positions one through five of the character array element **NAME(2,4)**:

```
NAME(2,4)(1:5)
```

A character substring has the length *e2* - *e1* + 1.

Substring Values e1, e2

The value of the numeric expressions *e1* and *e2* in a substring name must fall within the range

1 \xb2 *e1* \xb2 *e2* \xb2 *len*

where *len* is the length of the character variable or array element. A value of one is implied if *e1* is omitted. A value of *len* is taken if *e2* is omitted. When both *e1* and *e2* are not specified, the form *v*(:) is equivalent to *v* and the form *a*(*s* [,*s*]...)(:) is equivalent to *a*(*s* [,*s*]...).

The specification for *e1* and *e2* can be any numeric integer expression, including array element references and function references. Consider the character variable

```
XCHAR = 'QRSTUVWXYZ'
```

Table 2−11lists examples of valid substrings taken from this variable.

Table 2−11 Valid Substring Examples

Expression	Substring Value	Substring Length
EX1 = XCHAR (3:8)	STUVWX	6
EX2 = XCHAR (:8)	QRSTUVWX	8
EX3 = XCHAR (5:)	UVWXYZ	6

Other examples are

`BQ(10)(2:IX)` Specifies characters in positions 2 through integer **IX** of character array **BQ(10)**. The value of **IX** must be \xb3 2 and \xb2 the length of an element of **BQ**.

`BLT(:)` Equivalent to the variable **BLT**.

Records

The record−handling extension enables you to declare and operate on multifield records in Fortran programs. Avoid confusing the term *record* as it is used here with the term *record* that describes input and output data records.

Overview of Records and Structures

A *record* is a composite or aggregate entity containing one or more record elements or fields. Each element of a record is usually named. References to a record element consist of the name of the record and the name of the desired element. Records allow you to organize heterogeneous data elements within one structure and to operate on them either individually or collectively. Because they can be composed of heterogeneous data elements, records are not typed like arrays are.

Define the form of a record with a group of statements called a *structure definitionblock*. Establish a structure declaration in memory by specifying the name of the structure in a **RECORD** statement. A structure declaration block can include one or more of the following items:

- Typed data declarations (variables or arrays)
- Substructure declarations
- Mapped field declarations
- Unnamed fields

The following sections describe these items. Refer to the **RECORD** and **STRUCTURE** declarations block sections in Chapter 4, "Specification Statements," for details on specifying a structure in a source program.

Typed Data Declarations (Variables or Arrays)

Typed data declarations in structure declarations have the form of normal Fortran typed data declarations. You can freely intermix different types of data items within a structure declaration.

Substructure Declarations

Establish substructures within a structure by using either a nested structure declaration or a **RECORD** statement.

Mapped Field Declarations

Mapped field declarations are made up of one or more typed data declarations, substructure declarations (structure declarations and **RECORD** statements), or other mapped field declarations. A block of statements, called a union declaration, defines mapped field declarations. Unlike typed data declarations, all mapped field declarations that are made within a single union declaration share a common location within the containing structure.

Unnamed Fields

Declare unnamed fields in a structure by specifying the pseudo–name **%FILL** in place of an actual field name. **%FILL** generates empty space in a record for purposes such as alignment.

Record and Field References

The generic term *scalar reference* refers to all references that resolve to single typed data items. A scalar field reference of an aggregate falls into this category. The generic term *aggregate reference* is used to refer to all references that resolve to references of structured data items defined by a **RECORD** statement.

Scalar field references can appear wherever normal variables or array elements can appear, with the exception of **COMMON**, **SAVE**, **NAMELIST**, and **EQUIVALENCE** statements. Aggregate references can only appear in aggregate assignment statements, in unformatted I/O statements, and as parameters to subprograms.

Aggregate Assignment Statement

Aggregates can be assigned as whole entities. This special form of the assignment statement is indicated by an aggregate reference on the left–hand side of an assignment statement and requires an identical aggregate to appear on the right–hand side of the assignment.

Arrays

An array is a non–empty sequence of data of the same type occupying consecutive bytes in storage. A member of this sequence of data is referred to as an array element.

Each array has the following characteristics:

- array name
- data type

- array elements
- array declarator specifying:
  - number of dimensions
  - size and bounds of each dimension

Define an array using a **DIMENSION**, **COMMON**, or type statement (described in Chapter 4, "Specification Statements"); it can have a maximum of seven dimensions.

Note: For information on array handling when interacting with programs written in another language, see the *Fortran 77 Programmer's Guide.*

Array Names and Types

An array name is the symbolic name given to the array and must conform to the rules given in Chapter 1, "Fortran Elements and Concepts," for symbolic names. When referencing the array as a whole, specify only the array name. An array name is local to a program unit.

An array element is specified by the array name and a subscript. The form of an array element name is

a (*s* [,*s*]...)

where

a	is an array name.
(*s* [,*s*]...)	is a subscript.
s	is a subscript expression.

For example, **DATE(1,5)** accesses the element in the first row, fifth column, of the **DATE** array.

The number of subscript expressions must be equal to the number of dimensions in the array declarator for the array name.

An array element can be any of the valid Fortran data types. All array elements must be the same data type. Specify the data type explicitly using a type statement or implicitly using the first character of the array name. Refer to Chapter 1, "Fortran Elements and Concepts," for details about data types.

Reference a different array element by changing the subscript value of the array element name.

Array Declarators

An array declarator specifies a symbolic name for the array, the number of dimensions in the array, and the size and bounds of each dimension. Only one array declarator for an array name is allowed in a program unit. The array declarator can appear in a **DIMENSION** statement, a type statement, or a **COMMON** statement but not in more than one of these.

An array declarator has the form

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

where

a is a symbolic name of the array.

d is a dimension declarator of the following form:

[*d1*:] *d2*

where:

d1 is a lower−dimension bound that must be a numeric expression.

d2 is an upper−dimension bound that must be a numeric expression or an asterisk (*). Specify an asterisk only if *d2* is part of the last dimension declarator (see below).

If *d1* or *d2* is not of type integer, it is converted to integer values; any fractional part is truncated.

An array declarator can have a dummy argument as an array name and, therefore, be a dummy array declarator. An array declarator can be one of three types: a constant array declarator, an adjustable array declarator, or an assumed−size array declarator.

Each of the dimension bounds in a *constant array declarator* is a numeric constant expression. An *adjustable array declarator* is a dummy array declarator that contains one or more dimension bounds that are integer expressions but not constant integer expressions. An *assumed−size array declarator* is a dummy array declarator that has integer expressions for all dimension bounds, except that the upper dimension bound, *d2*, of the last dimension is an asterisk (*).

A dimension−bound expression cannot contain a function or array element name reference.

Value of Dimension Bounds

The lower−dimension bound, *d1*, and the upper−dimension bound, *d2*, can have positive, negative, or zero values. The value of the upper−dimension bound, *d2*, must be greater than or equal to that of the lower−dimension bound, *d1*.

If a lower−dimension bound is not specified, its value is assumed to be one (1). The upper−dimension bound of an asterisk (*) is always greater than or equal to the lower dimension bound.

The size of a dimension that does not have an asterisk (*) as its upper bound has the value ($d1 - d2$) +1.

The size of a dimension that has an asterisk (*) as its upper bound is not specified.

Array Size

The size of an array is exactly equal to the number of elements contained by the array. Therefore, the size of an array equals the product of the dimensions of the array. For constant and adjustable arrays, the size is straightforward. For assumed−size dummy arrays, however, the size depends on the actual argument corresponding to the dummy array. There are three cases:

- If the actual argument is a non−character array name, the size of the assumed−size array equals the size of the actual argument array.
- If the actual argument is a non−character array element name with a subscript value of j in an array of size x, the size of the assumed−size array equals $x - j + 1$.

- If the actual argument is either a character array name, a character array element name, or a character array element substring name, the array begins at character storage unit *t* of an array containing a total of *c* character storage units; the size of the assumed−size array equals:

  INT((c - t + 1)/*ln*)

  where *ln* is the length of an element of the dummy array.

Note: Given an assumed−size dummy array with *n* dimensions, the product of the sizes of the first n - 1 dimensions must not be greater than the size of the array (the size of the array is determined as described above).

Storage and Element Ordering

Storage for an array is allocated in the program unit in which it is declared, except in subprograms where the array name is specified as a dummy argument. The former declaration is called an actual array declaration. The declaration of an array in a subprogram where the array name is a dummy argument is called a *dummy array declaration.*

The elements of an array are ordered in sequence and stored in column order. This means that the left most subscript varies first, as compared to row order, in which the right most subscript varies first. The first element of the array has a *subscript value* of one; the second element has a *subscript value* of two; and so on. The last element has a *subscript value* equal to the size of the array.

Consider the following statement that declares an array with an **INTEGER** type statement:

```
INTEGER t(2,3)
```

Figure 2−1shows the ordering of elements of this array.

t(1,1)	t(2,1)	t(1,2)	t(2,2)	t(1,3)	t(2,3)

Figure 2−1 Order of Array Elements

Subscripts

The *subscript* describes the position of the element in an array and allows that array element to be defined or referenced. The form of a subscript is

(*s* [,*s*]...)

where *s* is a *subscript expression.* The term *subscript* includes the parentheses that delimit the list of subscript expressions.

A *subscript expression* must be a numeric expression and can contain array element references and function references. However, it cannot contain any function references that affect other subscript expressions in the same subscript.

A non−integer character can be specified for subscript expression. If specified, the non−integer character is converted to an integer before use; fractional portions remaining after conversion are truncated.

If a subscript expression is not of type integer, it is converted to integer values; any fractional part is truncated.

Because an array is stored as a sequence in memory, the values of the subscript expressions must be combined into a single value that is used as the offset into the sequence in memory. That single value is called the *subscript value.*

The subscript value determines which element of the array is accessed. The subscript value is calculated from the values of all the subscript expressions and the declared dimensions of the array (see Table 2−12).

Table 2−12 Determining Subscript Values

n	Dimension Declarator	Subscript	Subscript Value
1	(j1:k1)	(s1)	1 + (s1 - j1)
2	(j1:k1, j2:k2)	(s1, s2)	1 + (s1 - j1) + (s2 - j2)*d1
3	(j1:k1, j2:k2, j3:k3)	(s1, s2, s3)	1 + (s1-j1) + (s2-j2) * d1 + (s3-j3) * d2 * d1
n	(j1:k1, ...jn:kn)	(s1, ...sn)	1 + (s1 - j1) + (s2 - j2)*d1 + (s3-j3)*d1*d2 + ... + (sn - jn) * dn-1*dn-2*d1

The subscript value and the subscript expression value are not necessarily the same, even for a one−dimensional array. For example,

```
DIMENSION X(10,10),Y(-1:8)
Y(2) = X(1,2)
```

Y(2) identifies the fourth element of array **Y**, the subscript is **(2)** with a subscript value of four, and the subscript expression is **2** with a value of two. **X(1,2)** identifies the eleventh element of **X**, the subscript is **(1,2)** with a subscript value of eleven, and the subscript expressions are **1** and **2** with the values of one and two, respectively.