ASC Student Supercomputer Challenge:

Introduction to Fortran

LI Huimin
Supercomputing Center,
University of Science and Technology of China
2022.01.20

Why Fortran

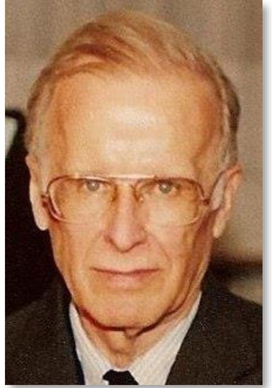
- The biggest feature: close to the natural description of mathematical formulas, has high execution efficiency in the computer
- Easy to learn, strict grammar
- Directly operate on matrices and complex numbers, which MatLab has inherited
- Widely used in the field of numerical computing, accumulated a large number of efficient and reliable source programs
- Widely used in parallel computing and high-performance computing
- Many specialized large-scale numerical computing computers optimized for Fortran
- The successive introduction of Fortran 90/95/2003 makes Fortran language possess some features of modern high-level programming languages
- The storage order of its matrix elements in the memory space adopts Column major, MatLab inherits this(most used C language adopts Row major)

$$A = \begin{bmatrix} a11 & a12 & a13 \\ a21 & a22 & a23 \end{bmatrix}$$

	Fortran	C/C++
Adress	Column-major order	Row-major order
0	a11	a11
1	a21	a12
2	a12	a13
3	a22	a21
4	a13	a22
5	a23	a23

History of Fortran

- The first widely used high-level programming language
- Name derived from Formula Translating System, Formula Translator, Formula Translation, or Formulaic Translation
- The Fortran language was developed to meet the needs of numerical computing.
- In December 1953, IBM engineer John Backus (J. Backus) deeply realized the difficulty of programming, and wrote a memorandum to Chairman Cuthbert Hurd, suggesting that the IBM 704 system Design a new computer language to improve development efficiency.
- John von Neumann(Dec. 28, 1903 – Feb. 8, 1957), a consultant at IBM at the time, strongly opposed it because he thought it was impractical and unnecessary.
- In 1957, IBM developed the first FORTRAN language, which ran on the IBM704 computer.
- The first FORTRAN program was tested at the Westinghouse Beddes nuclear power plant in Maryland. On the **afternoon of Friday**, April 20, 1957, an IBM software engineer decided to compile the first FORTRAN program in the power plant. When the code was entered, after compiling, the printer listed a message: "Error in the source program...behind the right parenthesis. No comma", which surprised the field personnel, and after correcting this error, the printer output the correct result.



John Warner Backus
(Dec. 3, 1924 – Mar. 17, 2007)

IBM 704 mainframe computer

- FORTRAN (1957)
- FORTRAN II (1958)
- FORTRAN III (1958)
- FORTRAN IV (1962)
- FORTRAN 66 (1966)
- FORTRAN 77 (1977)
- Fortran 90 (ISO/IEC 1539:1991)
  - “fairly” modern (structures, etc.)
  - Current “workhorse” Fortran
- Fortran 95 (1997)
- Fortran 2003(ISO/IEC 1539-1:2004)
  - Gradually being implemented by compiler companies
  - Object-oriented support
  - Interoperability with C is in the standard
- Fortran 2008(September 2010)
- Fortran 2018(28 November 2018)

Simple Porgrams

```
C------Average----------------------
      PROGRAM Example_1_1
      REAL a, b, av1, av2
      READ (*,*) a, b
      av1 = (a + b)/2
      av2 = sqrt(a*b)
      WRITE(*,*) av1, av2
      END
```

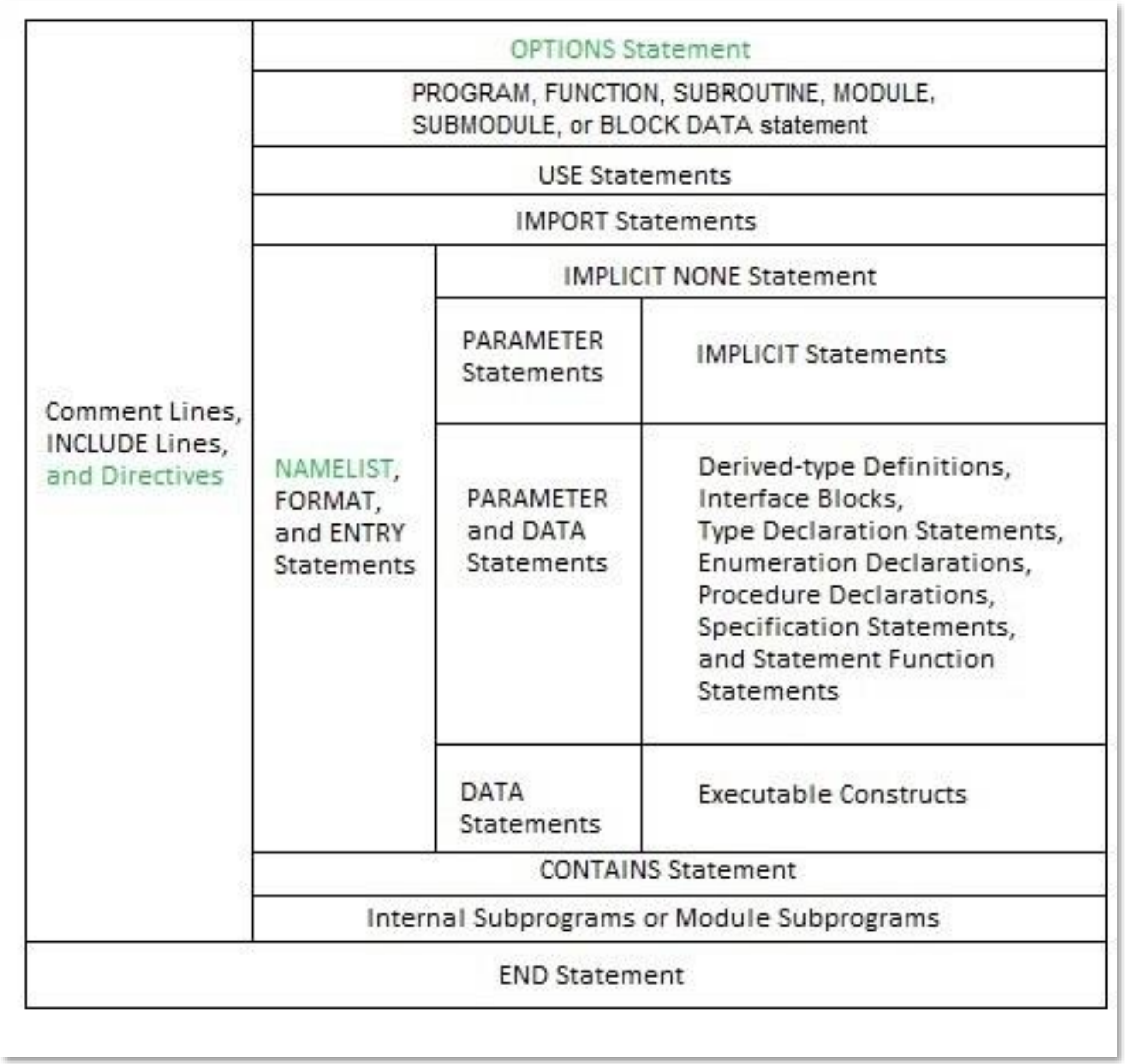
```
PROGRAM Example_1_1   ! Average
      REAL :: a, b, av1, av2
      READ *, a, b
      av1 = (a + b)/2; av2 = (a*b)**0.5
      PRINT *, av1, av2
      CALL ave(a,b,av1)
END
Subroutine AVE(x, y, sum)
      real x, y, sum
      sum=x+y
End subroutine
```

Program Units

A Fortran program consists of one or more program units, terminated by an END statement.

- main program: Only one
- external subprogram
  - a function or subroutine that is not contained within a main program, a module, a submodule, or another subprogram.
  - defines a procedure to be performed and can be invoked from other program units of the Fortran program
  - modules, submodules, and block data program units are not executable, so they are not considered to be procedures. (Modules and submodules can contain module procedures, though, which are executable.)
- modules and submodules
  - contain definitions that can be made accessible to other program units: data and type definitions, definitions of procedures (called module subprograms), procedure interfaces
  - module subprograms can be either functions or subroutines
  - can be invoked by other module subprograms in the module, or by other program units that access the module
- block data
  - specifies initial values for data objects in named common blocks
  - can be replaced by a module program unit

Statements



Scoping Unit	Restricted Statements
Main program	ENTRY, IMPORT, and RETURN statements
Module	ENTRY, FORMAT, IMPORT, OPTIONAL, and INTENT statements, statement functions, and executable statements
Submodule	ENTRY, FORMAT, IMPORT, OPTIONAL, and INTENT statements, statement functions, and executable statements
Block data program unit	CONTAINS, ENTRY, IMPORT, and FORMAT statements, interface blocks, statement functions, and executable statements
Internal subprogram	CONTAINS, IMPORT, and ENTRY statements
Interface body	CONTAINS, DATA, ENTRY, IMPORT[2], SAVE, and FORMAT statements, statement functions, and executable statements
BLOCK construct	CONTAINS, DATA, ENTRY, and IMPORT statements, statement functions, and these specification statements: COMMON, EQUIVALENCE, IMPLICIT, INTENT (or its equivalent attribute), NAMELIST, OPTIONAL (or its equivalent attribute), and VALUE (or its equivalent attribute)

Keywords

- A keyword can either be a part of the syntax of a statement (statement keyword), or it can be the name of a dummy argument (argument keyword). Examples of statement keywords are WRITE, INTEGER, DO, and OPEN
- In the intrinsic function UNPACK (vector, mask, field), for example, vector, mask, and field are argument keywords. They are dummy argument names, and any variable may be substituted in their place.
- Keywords are not reserved. For example, a program can have an array named IF, read, or Goto, even though this is not good programming practice. The only exception is the keyword PARAMETER. If you plan to use variable names beginning with PARAMETER in an assignment statement, you need to use compiler option altparam.
- Using keyword names for variables makes programs harder to read and understand. For readability, and to reduce the possibility of hard-to-find bugs, avoid using names that look like parts of Fortran statements. Rules that describe the context in which a keyword is recognized are discussed in topic Program Units and Procedures.
- Argument keywords are a feature of Standard Fortran that let you specify dummy argument names when calling intrinsic procedures, or anywhere an interface (either implicit or explicit) is defined. Using argument keywords can make a program more readable and easy to follow.

Names

- *Names* identify entities within a Fortran program unit (such as variables, function results, common blocks, named constants, procedures, program units, namelist groups, and dummy arguments). In FORTRAN 77, names were called "symbolic names".
- A name can contain letters, digits, underscores (_), and the dollar sign ($) special character. The first character must be a letter or a dollar sign.
  - Be careful when defining names that contain dollar signs. A dollar sign can be a symbol for command or symbol substitution in various shell and utility commands.
- In Fortran 2003, a name can contain up to 63 characters.
- The length of a module name (in MODULE and USE statements) may be restricted by your file system.
- In an executable program, the names of the following entities are global and must be unique in the entire program:
  - Program units
  - External procedures
  - Common blocks
  - Modules

Valid Names
NUMBER
FIND_IT
X

Invalid Names	
5Q	Begins with a numeral.
B.4	Contains a special character other than _ or $.
_WRONG	Begins with an underscore.

Character Sets

- Consists of follow:
  - All uppercase and lowercase letters (A - Z and a - z)
  - The numerals 0 - 9
  - The underscore (_)
  - The right special characters
- Other printable characters
  - include the tab character (09 hex), ASCII characters with codes in the range 20(hex) through 7E(hex), and characters in certain special character sets
  - that are not in the Standard Fortran character set can only appear in comments, character constants, Hollerith constants, character string edit descriptors, and input/output records
- Uppercase and lowercase letters:
  - case-insensitive: treated as equivalent when used to specify program behavior (except in character constants and Hollerith constants)
  - case-sensitive: treated as not equivalent when on the Unix/Linux/Mac OS(Windows/DOS is equivalent), the uppercase and lowercase of file or directory name: INCLUDE(‘a.f90’)

Character	Name	Character	Name
blank or <Tab>	Blank (space) or tab	;	Semicolon
=	Equal sign	!	Exclamation point
+	Plus sign	"	Quotation mark or quote
-	Minus sign	%	Percent sign
*	Asterisk	&	Ampersand
/	Slash	~	Tilde
\	Backslash	<	Less than
(	Left parenthesis	>	Greater than
)	Right parenthesis	?	Question mark
[	Left square bracket	'	Apostrophe
]	Right square bracket	`	Grave accent
{	Left curly bracket	^	Circumflex accent
}	Right curly bracket	|	Vertical line
,	Comma	$	Dollar sign (currency symbol)
.	Period or decimal point	#	Number sign
:	Colon	@	Commercial at

Source Forms

- source code can be in free, fixed, or tab form
  - Fixed or tab forms must not be mixed with free form in the same source program
  - different source forms can be used in different source programs.
- allow lowercase characters to be used as an alternative to uppercase characters
- several characters are indicators in source code (unless they appear within a comment or a Hollerith or character constant). The following are rules for indicators in all source forms:
  - Comment indicator
    - A comment indicator can precede the first statement of a program unit and appear anywhere within a program unit. If the comment indicator appears within a source line, the comment extends to the end of the line
    - An all blank line is also a comment line
    - Comments have no effect on the interpretation of the program unit
  - Statement separator
    - More than one statement (or partial statement) can appear on a single source line if a statement separator is placed between the statements. The statement separator is a semicolon character (;).
    - Consecutive semicolons (with or without intervening blanks) are considered to be one semicolon.
    - If a semicolon is the first character on a line, the last character on a line, or the last character before a comment, it is ignored.
  - Continuation indicator
    - A statement can be continued for more than one line by placing a continuation indicator on the line. Intel Fortran allows at least 511 continuation lines for a fixed or tab source program. Although Standard Fortran permits up to 256 continuation lines in free-form programs, Intel Fortran allows up to 511 continuation lines.
    - Comments can occur within a continued statement, but comment lines cannot be continued.

Indicators in Source Forms

Source Item	Indicator[1]	Source Form	Position
Comment	!	All forms	Anywhere in source code
Comment line	!	Free	At the beginning of the source line
	!, C, or *	Fixed	In column 1
		Tab	In column 1
Continuation line[2]	&	Free	At the end of the source line
	Any character except zero or blank	Fixed	In column 6
	Any digit except zero	Tab	After the first tab
Statement separator	;	All forms	Between statements on the same line
Statement label	1 to 5 decimal digits	Free	Before a statement
		Fixed	In columns 1 through 5
		Tab	Before the first tab
A debugging statement[3]	D	Fixed	In column 1
		Tab	In column 1

```
!Free Forms
READ*, A !,B
! READ*,C
WRITE(*,*) A, &
B
A=A+1; B=B+1
If(A==1) GOTO 100
A=A+C
100     A=A+B
```

```
CFixed Forms
C23456789
      READ*, A !,B
! READ*,C
C     WRITE(*,*) A
      WRITE(*,*) A
      A=A+1;B=B+1
100   A=A+B
D PRINT*,1
```

[1] If the character appears in a Hollerith or character constant, it is not an indicator and is ignored.
[2] For fixed or tab source form, at least 511 continuation lines are allowed. For free source form, at least 255 continuation lines are allowed.
[3] Fixed and tab forms only.

Free Source Form

- Statements are not limited to specific positions on a source line.
  - In Standard Fortran, a line can contain from 0 to 132 characters
  - Intel Fortran allows the line to be of any length
- Blank characters are significant in free source form
  - must not appear in lexical tokens, except within a character context. Ex.: there can be no blanks between the exponentiation operator **
  - can be used freely between lexical tokens to improve legibility
  - must be used to separate names, constants, or labels from adjacent keywords, names, constants, or labels. Ex.: consider the following statements:

```
INTEGER NUM
GO TO 40
20 DO K=1,8
```

The blanks are required after INTEGER, TO, 20, and DO.

- Some adjacent keywords must have one or more blank characters between them. Others do not require any, ex.: BLOCK DATA can also be spelled BLOCKDATA.

Optional Blanks	Required Blanks
BLOCK DATA	CASE DEFAULT
DOUBLE COMPLEX	DO WHILE
DOUBLE PRECISION	IMPLICIT *type-specifier*
ELSE IF	IMPLICIT NONE
ELSE WHERE	INTERFACE ASSIGNMENT
END ASSOCIATE	INTERFACE OPERATOR
END BLOCK	MODULE PROCEDURE
END BLOCK DATA	RECURSIVE FUNCTION
END DO	RECURSIVE SUBROUTINE
END ENUM	RECURSIVE *type-specifier* FUNCTION
END FILE	*type-specifier* FUNCTION
END FORALL	*type-specifier* RECURSIVE FUNCTION
END FUNCTION	
END IF	
END INTERFACE	
END MODULE	
END PROGRAM	
END SELECT	
END SUBMODULE	
END SUBROUTINE	
END TYPE	
END WHERE	
GO TO	
IN OUT	
SELECT CASE	
SELECT TYPE	

Free Source Form: Comment Indicator

the exclamation point character (!) indicates a comment if it is within a source line, or a comment line if it is the first character in a source line.

```
TCOSH(Y) = EXP(Y)   ! The initial statement line
! A continuation line
```

Free Source Form: Continuation Indicator

- The ampersand character (&) indicates a continuation line (unless it appears in a Hollerith or character constant, or within a comment). The continuation line is the first noncomment line following the ampersand. Standard Fortran permits up to 256 continuation lines in free-form programs, Intel Fortran allows up to 511 continuation lines.

```
TCOSH(Y) = EXP(Y) + & ! The initial statement line
   EXP(-Y) ! A continuation line
```

- If the first nonblank character on the next noncomment line is an ampersand, the statement continues at the character following the ampersand.

```
    TCOSH(Y) = EXP(Y) + &
     & EXP(-Y)
```

- If a lexical token must be continued, the first nonblank character on the next noncomment line must be an ampersand followed immediately by the rest of the token.

```
    TCOSH(Y) = EXP(Y) + EX&
     &P(-Y)
```

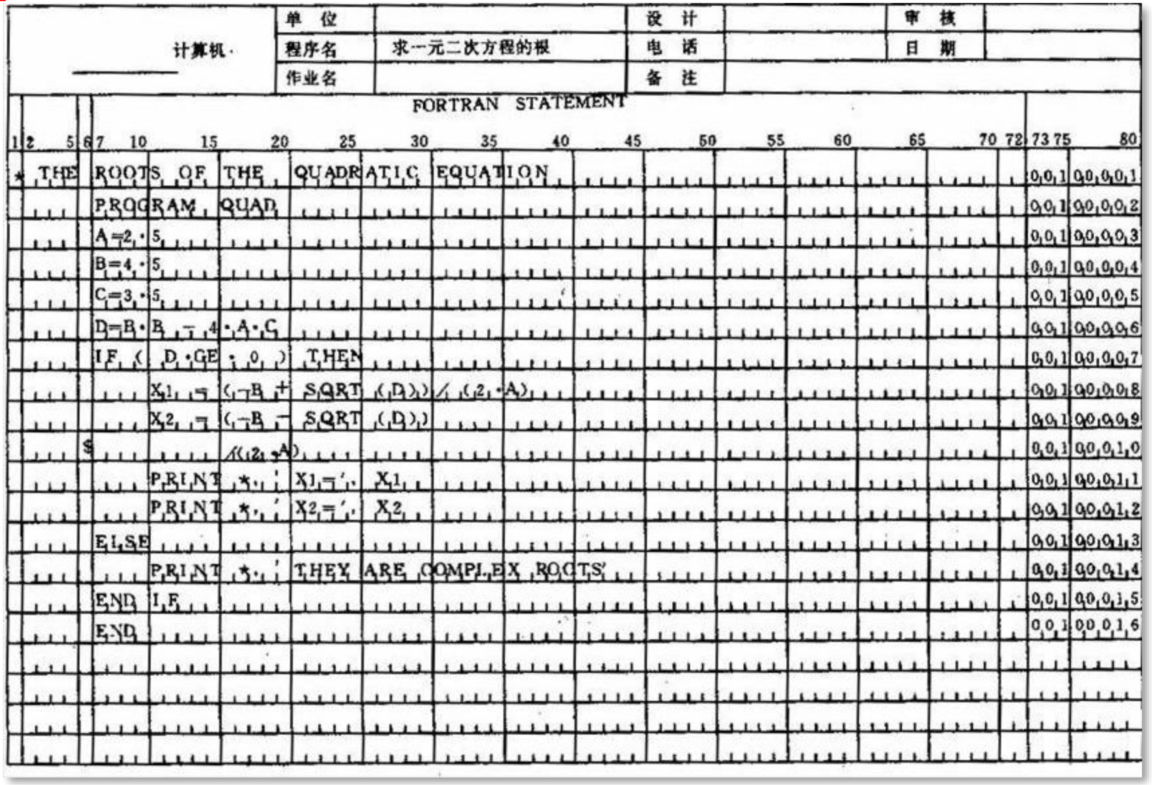
- If you continue a character constant, an ampersand must be the first non-blank character of the continued line; the statement continues with the next character following the ampersand.

```
    ADVERTISER = "Davis, O'Brien, Chalmers & Peter&
    &son“
    ARCHITECT = "O'Connor, Emerson, and Davis&
     & Associates"
```

- If the ampersand is omitted on the continued line, the statement continues with the first non-blank character in the continued line. So, in the preceding example, the whitespace before "Associates" would be ignored.
- The ampersand cannot be the only nonblank character in a line, or the only nonblank character before a comment; an ampersand in a comment is ignored.

Fixed and Tab Source Forms

- Fixed source form is identified as obsolescent.
- There are restrictions on where a statement can appear within a line.
- By default, a statement can extend to character position 72, any text following position 72 is ignored and no warning message is printed. You can specify compiler option extend-source to extend source lines to character position 132.
  - GNU Compiler(gfortran): -ffree-line-length-[n]
  - Intel Fortran Compiler(ifort): -extend-source [size], -noextend-source
- Except in a character context, blanks are not significant and can be used freely throughout the program for maximum legibility.
- Some Fortran compilers use blanks to pad short source lines out to 72 characters. If portability is a concern, you can use the concatenation operator to prevent source lines from being padded by other Fortran compilers or you can force short source lines to be padded by using compiler option pad-source.
  - GNU Compiler(gfortran): -fno-pad-source
  - Intel Fortran Compiler(ifort): -pad-source, -nopad-source



计算机· | 单 位 | 程序名 求一元二次方程的根 | 作业名 | 设 计 | 电 话 | 备 注 | 审 核 | 日 期

FORTRAN STATEMENT

```
* THE ROOTS OF THE QUADRATIC EQUATION                                  001 00001
      PROGRAM QUAD                                                     001 00002
      A=2.5                                                            001 00003
      B=4.5                                                            001 00004
      C=3.5                                                            001 00005
      D=B*B-4*A*C                                                      001 00006
      IF (D.GE.0) THEN                                                 001 00007
         X1=(-B+SQRT(D))/(2*A)                                         001 00008
         X2=(-B-SQRT(D))                                               001 00009
     $       /(2*A)                                                    001 00010
         PRINT *,'X1=',X1                                              001 00011
         PRINT *,'X2=',X2                                              001 00012
      ELSE                                                             001 00013
         PRINT *,'THEY ARE COMPLEX ROOTS'                              001 00014
      END IF                                                           001 00015
      END                                                              001 00016
```

Fixed and Tab Forms: Comment Indicator

- In fixed and tab source forms, the exclamation point character (!) indicates a comment if it is within a source line. It must not appear in column 6 of a fixed form line(that column is reserved for a continuation indicator).
- The letter C (or c), an asterisk (*), or an exclamation point (!) indicates a comment line when it appears in column 1 of a source line.

```
c23456789
 ! ‘This is a comment line’
     !  WRITE(*,*) !this is a WRITE statement
        WRITE(*,*) A
```

```
c23456789
c  This is a comment line
*       Write(*,*) !this is a coment line
        WRITE(*,*) A
```

Fixed and Tab Forms: Continuation Indicator

In fixed and tab source forms, a continuation line is indicated by one of the following:

- For fixed form: Any character (except a zero or blank) in column 6 of a source line
- For tab form: Any digit (except zero) after the first tab

The compiler considers the characters following the continuation indicator to be part of the previous line

- Although Standard Fortran permits up to 19 continuation lines in a fixed-form program
- Intel Fortran allows up to 511 continuation lines.

If a zero or blank is used as a continuation indicator, the compiler considers the line to be an initial line of a Fortran statement.

The statement label field of a continuation line must be blank (except in the case of a debugging statement).

When long character or Hollerith constants are continued across lines, portability problems can occur. Use the concatenation operator to avoid such problems. For example:

```
C23456789
      PRINT *, 'This is a very long character constant '//
     + 'which is safely continued across lines'
```

Use this same method when initializing data with long character or Hollerith constants. For example:

```
C23456789
      CHARACTER*(*) LONG_CONST
      PARAMETER (LONG_CONST = 'This is a very long '//
     + 'character constant which is safely continued  '//
     + 'across lines' )
      CHARACTER*100 LONG_VAL
      DATA LONG_VAL /LONG_CONST/
```

The Fortran Standard requires that, within a program unit, the END statement cannot be continued, and no other statement in the program unit can have an initial line that appears to be the program unit END statement.

Fixed and Tab Forms: Debugging Statement Indicator

- In fixed and tab source forms, the statement label field can contain a statement label, a comment indicator, or a debugging statement indicator.
- The letter D indicates a debugging statement when it appears in column 1 of a source line. The initial line of the debugging statement can contain a statement label in the remaining columns of the statement label field.
- If a debugging statement is continued onto more than one line, every continuation line must begin with a D and a continuation indicator.
- By default, the compiler treats debugging statements as comments. However, you can specify compiler option d-lines to force the compiler to treat debugging statements as source text to be compiled.
  - GNU Compiler(gfortran): -fd-lines-as-code, -fd-lines-as-comments
  - Intel Fortran Compiler(ifort): -d-lines, -nod-lines, -DD

Fixed-Format Lines

- a source line has columns divided into fields for:
  - statement labels
  - continuation indicators
  - statement text
  - sequence numbers
- Each column represents a single character

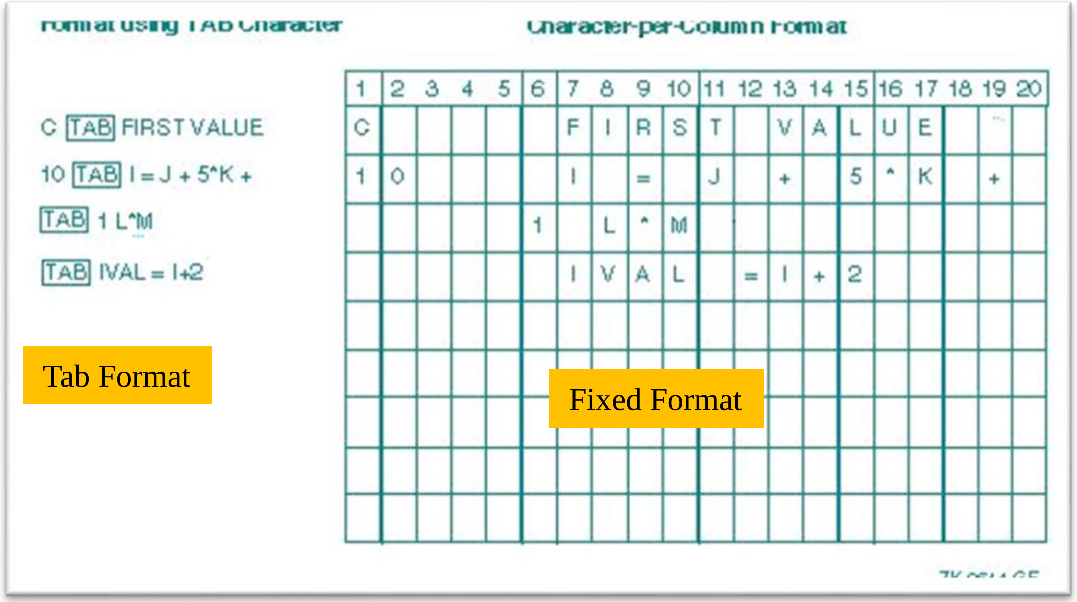
Field	Column
Statement label	1 through 5
Continuation indicator	6
Statement	7 through 72 (or 132 with compiler option extend-source)
Sequence number	73 through 80

```
C234567... ......737475...80
                  line 234
      A=1
```

- By default, a sequence number or other identifying information can appear in columns 73 through 80 of any fixed-format line in an Intel Fortran program. The compiler ignores the characters in this field.
- If extend the statement field to position 132, the sequence number field does not exist.
- If use the sequence number field, do not use tabs anywhere in the source line, or the compiler may interpret the sequence numbers as part of the statement field

Tab-Format Lines

- Can specify a statement label field, a continuation indicator field, and a statement field, but not a sequence number field.
- The following figure shows equivalent source lines coded with tab and fixed source form.



- The statement label field precedes the first tab character. The continuation indicator field and statement field follow the first tab character.
- The continuation indicator is any nonzero digit. The statement field can contain any Fortran statement. A Fortran statement cannot start with a digit.
- If a statement is continued, a continuation indicator must be the first character (following the first tab) on the continuation line.
- Many text editors and terminals advance the terminal print carriage to a predefined print position when you press the <Tab> key. However, the Intel Fortran compiler does not interpret the tab character in this way. It treats the tab character in a statement field the same way it treats a blank character. In the source listing that the compiler produces, the tab causes the character that follows to be printed at the next tab stop (usually located at columns 9, 17, 25, 33, and so on).

If you use the sequence number field, do not use tabs anywhere in the source line, or the compiler may interpret the sequence numbers as part of the statement field in your program.

Source Code Useable for All Source Forms

To write source code that is useable for all source forms (free, fixed, or tab), follow these rules:

Blanks	Treat as significant
Statement labels	Place in column positions 1 through 5 (or before the first tab character)
Statements	Start in column position 7 (or after the first tab character)
Comment indicator	Use only !. Place anywhere *except* in column position 6 (or immediately after the first tab character)
Continuation indicator	Use only &. Place in column position 73 of the initial line and each continuation line, and in column 6 of each continuation line (no tab character can precede the ampersand in column 6)

The following example is valid for all source forms:

```
Column:
12345678...                                                73
__________________________________________________________________________________
! Define the user function MY_SIN
      DOUBLE PRECISION FUNCTION MY_SIN(X)
        MY_SIN = X - X**3/FACTOR(3) + X**5/FACTOR(5)                    &
     &           - X**7/FACTOR(7)
      CONTAINS
        INTEGER FUNCTION FACTOR(N)
         FACTOR = 1
         DO 10 I = N, 1, -1
  10     FACTOR = FACTOR * I
        END FUNCTION FACTOR
      END FUNCTION MY_SIN
```

Intrinsic Data Types

- INTEGER

kind parameters are available:

  - INTEGER([KIND=]1) or INTEGER*1
  - INTEGER([KIND=]2) or INTEGER*2
  - INTEGER([KIND=]4) or INTEGER*4
  - INTEGER([KIND=]8) or INTEGER*8

- REAL

kind parameters are available:

  - REAL([KIND=]4) or REAL*4
  - REAL([KIND=]8) or REAL*8
  - REAL([KIND=]16) or REAL*16

- DOUBLE PRECISION

No kind parameter is permitted for data declared with type DOUBLE PRECISION. This data type is the same as REAL([KIND=]8).

•COMPLEX

kind parameters are available:

  - COMPLEX([KIND=]4) or COMPLEX*8
  - COMPLEX([KIND=]8) or COMPLEX*16
  - COMPLEX([KIND=]16) or COMPLEX*32

•DOUBLE COMPLEX

No kind parameter is permitted for data declared with type DOUBLE COMPLEX. This data type is the same as COMPLEX([KIND=]8).

•LOGICAL

kind parameters are available:

  - LOGICAL([KIND=]1) or LOGICAL*1
  - LOGICAL([KIND=]2) or LOGICAL*2
  - LOGICAL([KIND=]4) or LOGICAL*4
  - LOGICAL([KIND=]8) or LOGICAL*8

•CHARACTER

There is one kind parameter available for data of type character: CHARACTER([KIND=]1).

•BYTE

This is a 1-byte value; the data type is equivalent to INTEGER([KIND=]1).

```
INTEGER, PARAMETER:: MY_INT_KIND = SELECTED_INT_KIND(9)
...
INTEGER(MY_INT_KIND) :: J
...
```

Portable program

The intrinsic function KIND can be used to determine the kind type parameter of a representation method. For more portable programs, should not use the forms INTEGER([KIND=]n) or REAL([KIND=]n), should instead define a PARAMETER constant using the SELECTED_INT_KIND or SELECTED_REAL_KIND function, whichever is appropriate. For example, the following statements define a PARAMETER constant for an INTEGER kind that has 9 digits:

```
INTEGER, PARAMETER :: MY_INT_KIND = SELECTED_INT_KIND(9)
...
INTEGER(MY_INT_KIND) :: J
...
```

`result = SELECTED_INT_KIND (r)`

r: (Input) Must be scalar and of type integer.

- The result is a scalar of type default integer. The result has a value equal to the value of the kind parameter of the integer data type that represents all values *n* in the range of values *n* with $-10^r < n < 10^r$.
- If no such kind type parameter is available on the processor, the result is -1. If more than one kind type parameter meets the criteria, the value returned is the one with the smallest decimal exponent range.

```
i = SELECTED_INT_KIND(6)  ! returns 4
i = SELECTED_INT_KIND(8)  ! returns 4
i = SELECTED_INT_KIND(3)  ! returns 2
i = SELECTED_INT_KIND(10) ! returns 8
i = SELECTED_INT_KIND(20) ! returns -1 because 10**20 ! is bigger than 2**63
```

Integer Data Types

Integer data types can be specified as follows:

INTEGER

INTEGER([KIND=]*n*)

INTEGER**n*

n Is an initialization expression that evaluates to kind 1, 2, 4, or 8.

If a kind parameter is specified, the integer has the kind specified. If a kind parameter is not specified, integer constants are interpreted as follows:

- If the integer constant is within the default integer kind range, the kind is default integer.
- If the integer constant is outside the default integer kind range, the kind of the integer constant is the smallest integer kind that holds the constant.

Default integer is affected by compiler option integer-size, the INTEGER compiler directive, and the OPTIONS statement.

An entity-oriented example:

```
INTEGER, DIMENSION(:), POINTER :: days, hours
INTEGER(2), POINTER :: k, limit
INTEGER(1), DIMENSION(10) :: min
```

entity -> ::

An attribute-oriented example:

```
INTEGER days, hours
INTEGER(2) k, limit
INTEGER(1) min
DIMENSION days(:), hours(:), min (10)
POINTER days, hours, k, limit
```

An integer can be used in certain cases when a logical value is expected, such as in a logical expression evaluating a condition, as in the following:

```
INTEGER I, X
READ (*,*) I
IF (I) THEN
    X = 1
END IF
```

Integer Constants

An *integer constant* is a whole number with no decimal point. It can have a leading sign and is interpreted as a decimal number.

Integer constants take the following form:

[*s*]*n*[*n*...][_*k*]

s	Is a sign; required if negative (-), optional if positive (+).
n	Is a decimal digit (0 through 9). Any leading zeros are ignored.
k	Is the optional kind parameter: 1 for INTEGER(1), 2 for INTEGER(2), 4 for INTEGER(4), or 8 for INTEGER(8). It must be preceded by an underscore (_).

Valid Integer (base 10) Constants
0
-127
+32123
47_2

Invalid Integer (base 10) Constants	
9999999999999999999	Number too large.
3.14	Decimal point not allowed; this is a valid REAL constant.
32,767	Comma not allowed.
33_3	3 is not a valid kind type for integers.

Integer Constants

Integer constants are interpreted as decimal values (base 10) by default. To specify a constant that is not in base 10, use the following extension syntax:

[*s*] [[*base*] #] *nnn*...

s	Is an optional plus (+) or minus (-) sign.
base	Is any constant from 2 through 36. If *base* is omitted but # is specified, the integer is interpreted in base 16. If both *base* and # are omitted, the integer is interpreted in base 10. For bases 11 through 36, the letters A through Z represent numbers greater than 9. For example, for base 36, A represents 10, B represents 11, C represents 12, and so on, through Z, which represents 35. The case of the letters is not significant. The value of *nnn* cannot be bigger than 2**31-1. The value is extended with zeroes on the left or truncated on the left to make it the correct size. A minus sign for *s* negates the value.

The following seven integers are all assigned a value equal to 3,994,575 decimal:

```
I = 2#1111001111001111001111
m = 7#45644664
J = +8#17171717
K = #3CF3CF
n = +17#2DE110
L = 3994575
index = 36#2DM8F
```

The following seven integers are all assigned a value equal to -3,994,575 decimal:

```
I = -2#1111001111001111001111
m = -7#45644664
J = -8#17171717
K = -#3CF3CF
n = -17#2DE110
L = -3994575
index = -36#2DM8F
```

Real Data Types

Real data types can be specified as follows:

- REAL
- REAL([KIND=]*n*)
- REAL*n*
- DOUBLE PRECISION

n Is an initialization expression that evaluates to kind 4, 8, or 16.

- If a kind parameter is specified, the real constant has the kind specified. If a kind parameter is not specified, the kind is default real.
- Default real is affected by compiler options specifying real size and by the REAL directive.
- The default KIND for DOUBLE PRECISION is affected by compiler option double-size. If this compiler option is not specified, default DOUBLE PRECISION is REAL(8).
- No kind parameter is permitted for data declared with type DOUBLE PRECISION.
- The intrinsic inquiry function KIND returns the kind type parameter. The intrinsic inquiry function RANGE returns the decimal exponent range, and the intrinsic function PRECISION returns the decimal precision. You can use the intrinsic function SELECTED_REAL_KIND to find the kind values that provide a given precision and exponent range.

An entity-oriented example:

```
REAL (KIND = high), OPTIONAL :: testval
REAL, SAVE :: a(10), b(20,30)
```

An attribute-oriented example:

```
REAL (KIND = high) testval
REAL a(10), b(20,30)
OPTIONAL testval
SAVE a, b
```

General Rules for Real Constants

A *real constant* approximates the value of a mathematical real number. The value of the constant can be positive, zero, or negative.

The following is the general form of a real constant with no exponent part:

- [*s*]*n*[*n*...][_*k*]
- A real constant with an exponent part has one of the following forms:
- [*s*]*n*[*n*...]E[*s*]*nn*...[_*k*]
- [*s*]*n*[*n*...]D[*s*]*nn*...
- [*s*]*n*[*n*...]Q[*s*]*nn*...

s	Is a sign; required if negative (-), optional if positive (+).
n	Is a decimal digit (0 through 9). A decimal point must appear if the real constant has no exponent part.
k	Is the optional kind parameter: 4 for REAL(4), 8 for REAL(8), or 16 for REAL(16). It must be preceded by an underscore (_).

REAL(4) Constants

A single-precision REAL constant occupies four bytes of memory. The number of digits is unlimited, but typically only the leftmost 7 digits are significant.

Valid REAL(4) Constants
3.14159
3.14159_4
621712._4
-.00127
+5.0E3
2E-3_4

Invalid REAL(4) Constants	
1,234,567.	Commas not allowed.
325E-47	Too small for REAL; this is a valid DOUBLE PRECISION constant.
-47.E47	Too large for REAL; this is a valid DOUBLE PRECISION constant.
625._6	6 is not a valid kind for reals.
100	Decimal point is missing; this is a valid integer constant.
$25.00	Special character not allowed.

REAL(8) or DOUBLE PRECISION Constants

- A REAL(8) or DOUBLE PRECISION constant has more than twice the accuracy of a REAL(4) number, and greater range.
- A REAL(8) or DOUBLE PRECISION constant occupies eight bytes of memory. The number of digits that precede the exponent is unlimited, but typically only the leftmost 15 digits are significant.
- IEEE* binary64 format is used.
- Note that compiler option double-size can affect DOUBLE PRECISION data.
- The default KIND for DOUBLE PRECISION is affected by compiler option double-size.

Valid REAL(8) or DOUBLE PRECISION Constants
123456789D+5
123456789E+5_8
+2.7843D00
-.522D-12
2E200_8
2.3_8
3.4E7_8

Invalid REAL(8) or DOUBLE PRECISION Constants	
-.25D0_2	2 is not a valid kind for reals.
+2.7182812846182	No D exponent designator is present; this is a valid single-precision constant.
123456789.D400	Too large for any double-precision format.
123456789.D-400	Too small for any double-precision format.

REAL(16) Constants

- A REAL(16) constant has more than four times the accuracy of a REAL(4) number, and a greater range.
- A REAL(16) constant occupies 16 bytes of memory. The number of digits that precede the exponent is unlimited, but typically only the leftmost 33 digits are significant.
- IEEE* binary128 format is used.

Valid REAL(16) Constants
123456789Q4000
-1.23Q-400
+2.72Q0
1.88_16

Invalid REAL(16) Constants	
1.Q5000	Too large.
1.Q-5000	Too small.

Complex Data Types

Complex data types can be specified as follows:

- COMPLEX
- COMPLEX([KIND=]*n*)
- COMPLEX*s*
- DOUBLE COMPLEX

n	Is an initialization expression that evaluates to kind 4, 8, or 16.
s	Is 8, 16, or 32. COMPLEX(4) is specified as COMPLEX*8; COMPLEX(8) is specified as COMPLEX*16; COMPLEX(16) is specified as COMPLEX*32.

- If a kind parameter is specified, the complex constant has the kind specified. If no kind parameter is specified, the kind of both parts is default real, and the constant is of type default complex.
- Default real is affected by compiler option real-size and by the REAL directive.
- The default KIND for DOUBLE COMPLEX is affected by compiler option `double-size`. If the compiler option is not specified, default DOUBLE COMPLEX is COMPLEX(8).
- No kind parameter is permitted for data declared with type DOUBLE COMPLEX.
- A complex number of any kind is made up of a real part and an imaginary part. The REAL and AIMAG intrinsic functions return the real and imaginary parts of a complex number respectively. The CMPLX intrinsic constructs a complex number from two real numbers. The %re and %im complex part designators access the real and imaginary parts of a complex number respectively.

Complex Data Types: Examples

An entity-oriented example:

```
COMPLEX (4), DIMENSION (8) :: cz, cq
```

An attribute-oriented example:

```
COMPLEX(4) cz, cq
DIMENSION(8) cz, cq
```

The following shows an example of the parts of a complex number:

```
COMPLEX (4) :: ca = (1.0, 2.0)
REAL (4) :: ra = 3.0, rb = 4.0
PRINT *, REAL (ca), ca%RE  ! prints 1.0, 1.0
PRINT *, AIMAG (ca), ca%IM ! prints 2.0, 2.0
PRINT *, CMPLX (ra, rb)    ! prints (3.0, 4.0)
ca = CMPLX (ra, AIMAG (ca))
PRINT *, ca ! prints (3.0, 2.0)
ca%im = rb
PRINT *, ca ! prints (3.0, 4.0)
```

General Rules for Complex Constants

- A *complex constant* approximates the value of a mathematical complex number. The constant is a pair of real or integer values, separated by a comma, and enclosed in parentheses. The first constant represents the real part of that number; the second constant represents the imaginary part.
- The following is the general form of a complex constant: (*c*, *c*)

c Is as follows:

- For COMPLEX(4) constants, *c* is an integer or REAL(4) constant.
- For COMPLEX(8) constants, *c* is an integer, REAL(4) constant, or DOUBLE PRECISION (REAL(8)) constant. At least one of the pair must be DOUBLE PRECISION.
- For COMPLEX(16) constants, *c* is an integer, REAL(4) constant, REAL(8) constant, or REAL(16) constant. At least one of the pair must be a REAL(16) constant.

COMPLEX(4) Constants

- A COMPLEX(4) constant is a pair of integer or single-precision real constants that represent a complex number.
- A COMPLEX(4) constant occupies eight bytes of memory and is interpreted as a complex number.
- If the real and imaginary part of a complex literal constant are both real, the kind parameter value is that of the part with the greater decimal precision.
- The rules for REAL(4) constants apply to REAL(4) constants used in COMPLEX constants.
- The REAL(4) constants in a COMPLEX constant have IEEE* binary32 format.
- Note that compiler option *real-size* can affect REAL data.

Valid COMPLEX(4) Constants
(1.7039,-1.70391)
(44.36_4,-12.2E16_4)
(+12739E3,0.)
(1,2)

Invalid COMPLEX(4) Constants	
(1.23,)	Missing second integer or single-precision real constant.
(1.0, 2H12)	Hollerith constant not allowed.

COMPLEX(8) or DOUBLE COMPLEX Constants

- A COMPLEX(8) or DOUBLE COMPLEX constant is a pair of constants that represents a complex number. One of the pair must be a double-precision real constant, the other can be an integer, single-precision real, or double-precision real constant.
- A COMPLEX(8) or DOUBLE COMPLEX constant occupies 16 bytes of memory and is interpreted as a complex number.
- The rules for DOUBLE PRECISION (REAL(8)) constants also apply to the double precision portion of COMPLEX(8) or DOUBLE COMPLEX constants.
- The DOUBLE PRECISION constants in a COMPLEX(8) or DOUBLE COMPLEX constant have IEEE* binary64 format.
- The default KIND for DOUBLE COMPLEX is affected by compiler option `double-size`.
- Note that compiler option real-size can affect REAL data.

Valid COMPLEX(8) or DOUBLE COMPLEX Constants
(1.7039,-1.7039D0)
(547.3E0_8,-1.44_8)
(1.7039E0,-1.7039D0)
(+12739D3,0.D0)

Invalid COMPLEX(8) or DOUBLE COMPLEX Constants	
(1.23D0,)	Second constant missing.
(1D1,2H12)	Hollerith constants not allowed.
(1,1.2)	Neither constant is DOUBLE PRECISION; this is a valid single-precision constant.

COMPLEX(16) Constants

- A COMPLEX(16) constant is a pair of constants that represents a complex number. One of the pair must be a REAL(16) constant, the other can be an integer, single-precision real, double-precision real, or REAL(16) constant.
- A COMPLEX(16) constant occupies 32 bytes of memory and is interpreted as a complex number.
- The rules for REAL(16) constants apply to REAL(16) constants used in COMPLEX constants.
- The REAL(16) constants in a COMPLEX constant have IEEE* binary128 format.
- Note that compiler option real-size can affect REAL data.

Valid COMPLEX(16) Constants
(1.7039,-1.7039Q2)
(547.3E0_16,-1.44)
(+12739D3,0.Q0)

Invalid COMPLEX(16) Constants	
(1.23Q0,)	Second constant missing.
(1D1,2H12)	Hollerith constants not allowed.
(1.7039E0,-1.7039D0)	Neither constant is REAL(16); this is a valid double-precision constant.

Logical Data Types

Logical data types can be specified as follows:

- LOGICAL
- LOGICAL([KIND=]*n*)
- LOGICAL**n*

n Is an initialization expression that evaluates to kind 1, 2, 4, or 8.

An entity-oriented example:

```
LOGICAL, ALLOCATABLE :: flag1, flag2
LOGICAL (KIND = byte), SAVE :: doit, dont
```

An attribute-oriented example:

```
LOGICAL flag1, flag2
LOGICAL (KIND = byte) doit, don’t
ALLOCATABLE flag1, flag2
SAVE doit, dont
```

Logical Constants

A logical constant represents only the logical values true or false, and takes one of the following forms:

- .TRUE.[_*k*]
- .FALSE.[_*k*]

k Is the optional kind parameter: 1 for LOGICAL(1), 2 for LOGICAL(2), 4 for LOGICAL(4), or 8 for LOGICAL(8). It must be preceded by an underscore (_).

- The numeric value of .TRUE. and .FALSE. can be -1 and 0 or 1 and 0 depending on compiler option fpscomp [no]logicals.
- Logical data can take on integer data values.
- Logical data type ranges correspond to their comparable integer data type ranges. For example, the LOGICAL(2) range is the same as the INTEGER(2) range.

Character Data Type

The character data type can be specified as follows:

- CHARACTER
- CHARACTER([LEN=] *len*)
- CHARACTER(LEN= *len*, KIND= *n*)
- CHARACTER(len, [KIND=] *n*)
- CHARACTER(KIND= *n* [, LEN= *len*])
- CHARACTER* *len* [,]

n	Is an initialization expression that evaluates to kind 1.
len	Is a string length (not a kind).

Character Constants

A *character constant* is a character string enclosed in delimiters (apostrophes or quotation marks). It takes one of the following forms:

- [*k*_]'[*ch*...]' [*C*]
- [*k*_]"[*ch*...]" [*C*]

k	Is the optional kind parameter: 1 (the default). It must be followed by an underscore (_). Note that in character constants, the kind must precede the constant.
ch	Is an ASCII character.
C	Is a C string specifier. C strings can be used to define NUL-terminated strings.

- The value of a character constant is the string of characters between the delimiters. The value does not include the delimiters, but does include all blanks or tabs within the delimiters.
- If a character constant is delimited by apostrophes, use two consecutive apostrophes (' ') to place an apostrophe character in the character constant.
- If a character constant is delimited by quotation marks, use two consecutive quotation marks ("") to place a quotation mark character in the character constant.
- The length of the character constant is the number of characters between the delimiters, but two consecutive delimiters are counted as one character.
- The length of a character constant must be in the range of 0 to 7188. Each character occupies one byte of memory.
- If a character constant appears in a numeric context (such as an expression on the right side of an arithmetic assignment statement), it is considered a Hollerith constant.
- A zero-length character constant is represented by two consecutive apostrophes or quotation marks

Valid Character Constants
"WHAT KIND TYPE? "
'TODAY''S DATE IS: '
"The average is: "
''

Invalid Character Constants	
'HEADINGS	No trailing apostrophe.
'Map Number:"	Beginning delimiter does not match ending delimiter.

C Strings in Character Constants

- String values in the C language are terminated with null characters (CHAR(0)) and can contain nonprintable characters (such as backspace).
- Nonprintable characters are specified by escape sequences. An escape sequence is denoted by using the backslash (\) as an escape character, followed by a single character indicating the nonprintable character desired.
- This type of string is specified by using a standard string constant followed by the character C. The standard string constant is then interpreted as a C-language constant. Backslashes are treated as escapes, and a null character is automatically appended to the end of the string (even if the string already ends in a null character).
- The right table shows the escape sequences that are allowed in character constants.

Escape Sequence	Represents
\a or \A	A bell
\b or \B	A backspace
\f or \F	A formfeed
\n or \N	A new line
\r or \R	A carriage return
\t or \T	A horizontal tab
\v or \V	A vertical tab
\x*hh* or \X*hh*	A hexadecimal bit pattern
ooo	An octal bit pattern
\0	A null character
\\	A backslash

C Strings in Character Constants

- If a string contains an escape sequence that isn't in this table, the backslash is ignored.
- A C string must also be a valid Fortran string. If the string is delimited by apostrophes, apostrophes in the string itself must be represented by two consecutive apostrophes (`''`).
- For example, the escape sequence `\'string` causes a compiler error because Fortran interprets the apostrophe as the end of the string. The correct form is `\''string`.
- If the string is delimited by quotation marks, quotation marks in the string itself must be represented by two consecutive quotation marks ("").
- The sequences \ooo and \x*hh* allow any ASCII character to be given as a one- to three-digit octal or a one- to two-digit hexadecimal character code. Each octal digit must be in the range 0 to 7, and each hexadecimal digit must be in the range 0 to F. For example, the C strings `'\010'C` and `'\x08'C` both represent a backspace character followed by a null character.
- The C string `'\\abcd'C` is equivalent to the string `'\abcd'` with a null character appended. The string `''C` represents the ASCII null character.

Character Substrings

A *character substring* is a contiguous segment of a character string. It takes one of the following forms:

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

v	Is a character scalar constant, or the name of a character scalar variable or character structure component.
e1	Is a scalar integer (or other numeric) expression specifying the leftmost character position of the substring; the *starting* point.
e2	Is a scalar integer (or other numeric) expression specifying the rightmost character position of the substring; the *ending* point.
a	Is the name of a character array.
s	Is a subscript expression.

- Character positions within the parent character string are numbered from left to right, beginning at 1.
- If the value of the numeric expression *e1* or *e2* is not of type integer, it is converted to integer before use (any fractional parts are truncated).
- If *e1* is omitted, the default is 1. If *e2* is omitted, the default is *len*. For example, NAMES(1,3)(:7) specifies the substring starting with the first character position and ending with the seventh character position of the character array element NAMES(1,3).

```
CHARACTER*8 C, LABEL
LABEL = 'XVERSUSY'
C = LABEL(2:7)
```

LABEL(2:7) specifies the substring starting with the second character position and ending with the seventh character position of the character variable assigned to LABEL, so C has the value `'VERSUS'`.

POINTER - Fortran

Statement and Attribute: Specifies that an object or a procedure is a pointer (a dynamic variable). A pointer does not contain data, but *points* to a scalar or array variable where data is stored. A pointer has no initial storage set aside for it; memory storage is created for the pointer as a program runs.

The POINTER attribute can be specified in a type declaration statement or a POINTER statement, one of the following forms:

Type Declaration Statement:

type,[*att-ls*,] POINTER [, *att-ls*] :: *ptr*[(*d-spec*)][, *ptr*[(*d-spec*)]]...

Statement:

POINTER [::]*ptr*[(*d-spec*)][, *ptr*[(*d-spec*)]]...

type-spec	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
ptr	Is the name of the pointer. The pointer cannot be declared with the INTENT or PARAMETER attributes.
d-spec	(Optional) Is a deferred-shape specification (: [, :] ...). Each colon represents a dimension of the array.

Fortran pointers are *not* the same as integer pointers

- Fortran pointers: POINTER [::]ptr[(d-spec)]
- Integer pointers: POINTER (*pointer*, *pointee*)

```
REAL, POINTER :: arrow (:)
REAL, ALLOCATABLE, TARGET :: bullseye (:,:)
! The following statement associates the pointer with an unused
! block of memory.
ALLOCATE (arrow (1:8), STAT = ierr)
IF (ierr.eq.0) WRITE (*,'(/1x,a)') 'ARROW allocated'
arrow = 5.
WRITE (*,'(1x,8f8.0/)') arrow
ALLOCATE (bullseye (1:8,3), STAT = ierr)
IF (ierr.eq.0) WRITE (*,*) 'BULLSEYE allocated'
bullseye = 1.
bullseye (1:8:2,2) = 10.
WRITE (*,'(1x,8f8.0)') bullseye
! The following association breaks the association with the first
! target, which being unnamed and unassociated with other pointers,
! becomes lost. ARROW acquires a new shape.
arrow => bullseye (2:7,2)
WRITE (*,'(/1x,a)') 'ARROW is repointed & resized, all the 5s are lost'
WRITE (*,'(1x,8f8.0)') arrow
NULLIFY (arrow)
IF (.NOT.ASSOCIATED(arrow)) WRITE (*,'(/a/)') ' ARROW is not pointed'
DEALLOCATE (bullseye, STAT = ierr)
IF (ierr.eq.0) WRITE (*,*) 'Deallocation successful.'
END
```

POINTER - Integer

Establishes pairs of objects and pointers, in which each pointer contains the address of its paired object. This statement is different from the Fortran POINTER statement.

POINTER (*pointer*,*pointee*) [,(*pointer*,*pointee*)] . . .

pointer	Is a variable whose value is used as the address of the *pointee*.
pointee	Is a variable; it can be an array name or array specification. It can also be a procedure named in an EXTERNAL statement or in a specific (non-generic) procedure interface block.

Using %LOC

```
INTEGER I(10)
INTEGER I1(10) /10*10/
POINTER (P,I)
P = %LOC(I1)
I(2) = I(2) + 1
```

Using MALLOC

```
INTEGER I(10)
POINTER (P,I)
P = MALLOC(40)
I = 10
I(2) = I(2) + 1
```

Example

```
POINTER (p, k)
INTEGER j(2)
! This has the same effect as j(1) = 0, j(2) = 5
p = LOC(j)
k = 0
p = p + SIZEOF(k)
! 4 for 4-byte integer
k = 5
```

Derived Data Types

```
TYPE REPORT
        CHARACTER (LEN=20) REPORT_NAME
        INTEGER DAY
        CHARACTER (LEN=3) MONTH
        INTEGER :: YEAR = 1995 ! Only component with default
END TYPE REPORT ! initialization
```

```
TYPE (REPORT), PARAMETER :: NOV_REPORT = REPORT ("Sales", 15, "NOV", 1996)
```

```
TYPE MGR_REPORT
        TYPE (REPORT) :: STATUS = NOV_REPORT
        INTEGER NUM
END TYPE MGR_REPORT
TYPE (MGR_REPORT) STARTUP
```

Components are accessed using %:

```
STARTUP%NUM = 20
```

Binary, Octal, Hexadecimal, and Hollerith Constants

A *binary* constant:

- B'*d*[*d*...]'
- B"*d*[*d*...]"

d s a binary (base 2) digit (0 or 1).

You can specify up to 128 binary digits in a binary constant.

Valid Binary Constants
B'0101110'
B"1"

Invalid Binary Constants	
B'0112'	The character 2 is invalid.
B10011'	No apostrophe after the B.
"1000001"	No B before the first quotation mark.

An *octal* constant:

- O'*d*[*d*...]'
- O"*d*[*d*...]"

d is an octal (base 8) digit (0 through 7).

You can specify up to 128 bits (43 octal digits) in octal constants.

Valid Octal Constants
O'07737'
O"1"

Invalid Octal Constants	
O'7782'	The character 8 is invalid.
O7772'	No apostrophe after the O.
"0737"	No O before the first quotation mark.

A *hexadecimal* constant:

- Z'*d*[*d*...]'
- Z"*d*[*d*...]“

d s a hexadecimal (base 16) digit (0 through 9, or an uppercase or lowercase letter in the range of A to F).

You can specify up to 128 bits (32 hexadecimal digits) in hexadecimal constants

Valid Hexadecimal Constants
Z'AF9730'
Z"FFABC"
Z'84'

Invalid Hexadecimal Constants	
Z'999.'	Decimal not allowed.
ZF9"	No quotation mark after the Z.

Hollerith Constants

- A *Hollerith constant* is a string of printable ASCII characters preceded by the letter H. Before the H, there must be an unsigned, nonzero default integer constant stating the number of characters in the string (including blanks and tabs).
- Hollerith constants are strings of 1 to 2000 characters. They are stored as byte strings, one character per byte.

Valid Hollerith Constants
16HTODAY'S DATE IS:
1HB
4H ABC

Invalid Hollerith Constants	
3HABCD	Wrong number of characters.
0H	Hollerith constants must contain at least one character.

Enumerations and Enumerators

An enumeration defines the name of a group of related values and the name of each value within the group. It takes the following form:

ENUM, BIND(C)

ENUMERATOR [::] *c1* [= *expr*][, *c2* [= *expr*]]...

[ENUMERATOR [::] *c3* [= *expr*][, *c4* [= *expr*]]...]...

END ENUM

c1,*c2*,*c3*,*c4*	Is the name of the enumerator being defined.
expr	Is an optional scalar integer initialization expression specifying the value for the enumerator.

The order in which the enumerators appear in an enumerator definition is significant.
If you do not explicitly assign each enumerator a value by specifying an *expr*, the compiler assigns a value according to the following rules:
- If the enumerator is the first enumerator in the enumerator definition, the enumerator has the value 0.
- If the enumerator is not the first enumerator in the enumerator definition, its value is the result of adding one to the value of the immediately preceding enumerator in the enumerator definition.

```
ENUM, BIND(C)
      ENUMERATOR ORANGE
      ENUMERATOR :: RED = 5, BLUE = 7
      ENUMERATOR GREEN
END ENUM
```

be equivalent to the enumeration definition

```
INTEGER(SELECTED_INT_KIND(4)), PARAMETER :: ORANGE = 0, RED = 5, BLUE = 7, GREEN = 8
```

Implicit Typing Rules

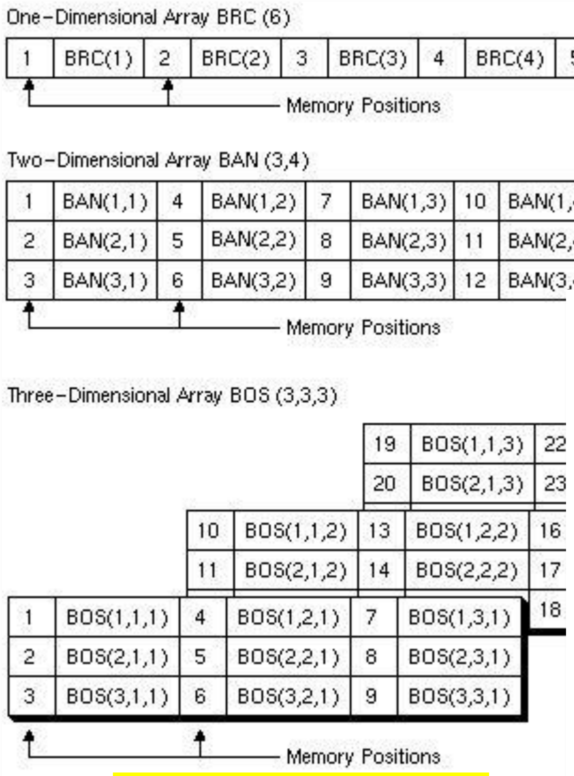
Real Variables	Integer Variables
ALPHA	JCOUNT
BETA	ITEM_1
TOTAL_NUM	NTOTAL

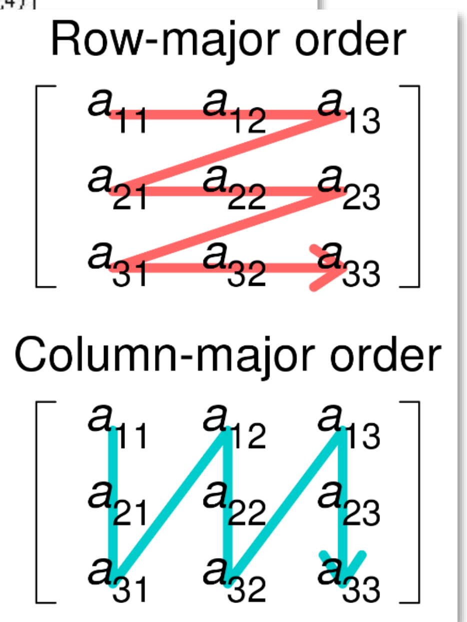
- I-N rules, by default:
  - all scalar variables with names beginning with I, J, K, L, M, or N are assumed to be default integer variables.
  - Scalar variables with names beginning with any other letter are assumed to be default real variables.
  - Names beginning with a dollar sign ($) are implicitly INTEGER.
- Can override the default data type implied in a name by specifying data type in either an IMPLICIT statement or a type declaration statement.
- Advice: IMPLICIT NONE
  - The IMPLICIT NONE statement disables all implicit typing defaults. When IMPLICIT NONE is used, all names in a program unit must be explicitly declared. An IMPLICIT NONE statement must precede any PARAMETER statements, and there must be no other IMPLICIT statements in the scoping unit.
  - To receive diagnostic messages when variables are used but not declared, you can specify compiler option warn declarations instead of using IMPLICIT NONE.

Array

The elements of an array are stored as a linear sequence of values. A multidimensional array is stored so that the leftmost subscripts vary most rapidly(Column-major order, contrary to C/C++ Row-major order).

```fortran
INTEGER, DIMENSION(2:11,3) :: L ! Specifies the type and
                                ! dimensions of array L
REAL A(10)
REAL, ALLOCATABLE :: E(:,:)
L = 10 ! The value 10 is assigned to all the
       ! elements in array L
WRITE *, L ! Prints all the elements in array L
A(1:5:2) = 3.0 ! Sets elements A(1), A(3), A(5) to 3.0
A(:5:2) = 3.0  ! Same as the previous statement
               ! because the lower bound defaults to 1
A(2::3) = 3.0  ! Sets elements A(2), A(5), A(8) to 3.0
               ! The upper bound defaults to 10
A(7:9) = 3.0   ! Sets elements A(7), A(8), A(9) to 3.0
               ! The stride defaults to 1
A(:) = 3.0     ! Same as A = 3.0; sets all elements of
               ! A to 3.0
...
IF (.NOT. ALLOCATED(E)) ALLOCATE(E(2:4,7))
...
DEALLOCATE (E) ! Deallocates array E

WHERE(A .NE. 0) C = B/A !only for none-zero elements of A
```





Vector Subscripts

```fortran
REAL A(3, 3), B(4)
INTEGER K(4)
! Vector K has repeated values, Syntax
! (/.../) denotes an array constructor
K = (/3, 1, 1, 2/)
! Sets all elements of A to 5.0
A = 5.0
B = A(3, K) !B is A(3,3) A(3,1) A(3,1) A(3,2)
```

Array Constructors

```
INTEGER C(4),C2(20)

TYPE EMPLOYEE
     INTEGER ID
     CHARACTER(LEN=30) NAME
END TYPE EMPLOYEE
TYPE(EMPLOYEE) CC_4T(4) !Derived Data Type Array.
CC_4T = (/EMPLOYEE(2732,"JONES"),EMPLOYEE(0217,"LEE"), &
EMPLOYEE(1889,"RYAN"), EMPLOYEE(4339,"EMERSON")/)

C = (/4,8,7,6/) ! A scalar expression
C = [4,8,7,6]
C = (/B(I, 1:5), B(I:J, 7:9)/) ! An array expression
C = (/(I, I=1, 4)/) ! An implied-DO loop
C2= (/4, A(1:5), (I, I=1, 4), 7/)
```

Array operator

Some intrinsic functions to perform some operations on entire arrays:

- +, -, *, /, SIN, COS, WHERE, ABS …
- SUM: not same as A+B
  - Returns the sum of all the elements in an entire array or in a specified dimension of an array.
- DOT_RRODUCT: not same as A*B
  - Performs dot-product multiplication of numeric or logical vectors (rank-one arrays).
- PRODUCT: not same as A*B
  - Returns the product of all the elements in an entire array or in a specified dimension of an array.
- MATMUL:
  - Performs matrix multiplication of numeric or logical matrices.
- MAXVAL:
  - Returns the maximum value of all elements in an array, a set of elements in an array, or elements in a specified dimension of an array.
- MAXLOC:
  - Returns the minimum value of all elements in an array, a set of elements in an array, or elements in a specified dimension of an array.
- MINVAL:
  - Returns the minimum value of all elements in an array, a set of elements in an array, or elements in a specified dimension of an array.
- MINLOC:
  - Returns the minimum value of all elements in an array, a set of elements in an array, or elements in a specified dimension of an array.
- TRANSPOSE:
  - Transposes an array of rank two.
- RESHAPE:
  - Constructs an array with a different shape from the argument array.

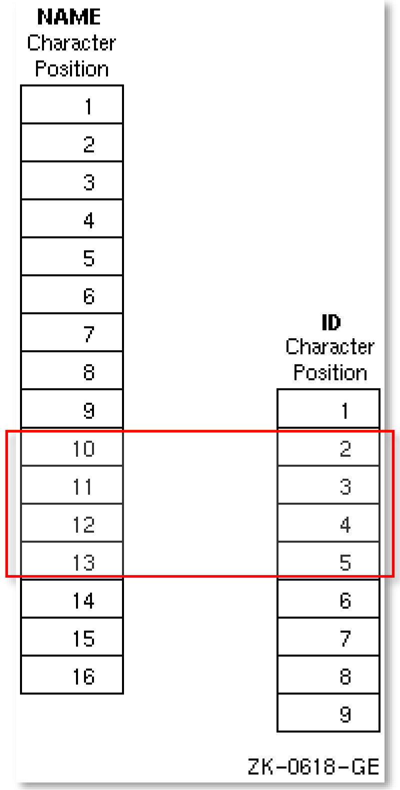
```
INTEGER array (2, 3)
INTEGER AR1(3), AR2(2)
array = RESHAPE((/1, 4, 2, 5, 3, 6/),(/2,3/))
! array is 1 2 3
!          4 5 6
AR1 = PRODUCT(array, DIM = 1)
! returns [ 4 10 18 ]
AR2 = PRODUCT(array, MASK =array .LT. 6, DIM = 2)
! returns [ 6 20 ]
AR2=SHAPE(array) ! Return AR2=(/2, 3/)
END
```

Making Arrays and Substrings Equivalent

```
DIMENSION TABLE (2,2), TRIPLE (2,2,2)
EQUIVALENCE(TABLE(2,2), TRIPLE(1,2,2))
```

```
CHARACTER NAME*16, ID*9
EQUIVALENCE(NAME(10:13), ID(2:5))
```

Array TRIPLE		Array TABLE	
Array Element	Element Number	Array Element	Element Number
TRIPLE(1,1,1)	1		
TRIPLE(2,1,1)	2		
TRIPLE(1,2,1)	3		
TRIPLE(2,2,1)	4	TABLE(1,1)	1
TRIPLE(1,1,2)	5	TABLE(2,1)	2
TRIPLE(2,1,2)	6	TABLE(1,2)	3
TRIPLE(1,2,2)	7	TABLE(2,2)	4
TRIPLE(2,2,2)	8		



Expressions and Assignment Statements

Operator	Function
**	Exponentiation, A^E is A**E
*	Multiplication
/	Division
+	Addition or unary plus (identity)
-	Subtraction or unary minus (negation)

Relational Expressions

Operator	Relationship
.LT. or <	Less than
.LE. or <=	Less than or equal to
.EQ. or ==	Equal to
.NE. or /=	Not equal to
.GT. or >	Greater than
.GE. or >=	Greater than or equal

Character Expressions

A character expression consists of a character operator (//) that concatenates two operands of type character.
Parentheses do not affect the evaluation of a character expression; the following character expressions are equivalent 'ABCDEF':

```
A=('ABC'//'DE')//'F'
A='ABC'//('DE'//'F')
A='AABC'//'DE'//'F'
```

Logical Expressions

Operator	Example	Meaning
.AND.	A .AND. B	Logical conjunction: the expression is true if both A and B are true.
.OR.	A .OR. B	Logical disjunction (inclusive OR): the expression is true if either A, B, or both, are true.
.NEQV.	A .NEQV. B	Logical inequivalence (exclusive OR): the expression is true if either A or B is true, but false if both are true.
.XOR.	A .XOR. B	Same as .NEQV.
.EQV.	A .EQV. B	Logical equivalence: the expression is true if both A and B are true, or both are false.

Summary of Operator Precedence

Category	Operator	Precedence
	Defined Unary Operators	Highest
Numeric	**	.
Numeric	* or /	.
Numeric	Unary + or -	.
Numeric	Binary + or -	.
Character	//	.
Relational	.EQ., .NE., .LT., .LE., .GT., .GE., = =, /=, <, <=, >, >=	.
Logical	.NOT.	.
Logical	.AND.	.
Logical	.OR.	.
Logical	.XOR., .EQV., .NEQV.	.
	Defined Binary Operators	Lowest

DO, DO WHILE and GO TO loop

DO loop repeats calculation over range of indices

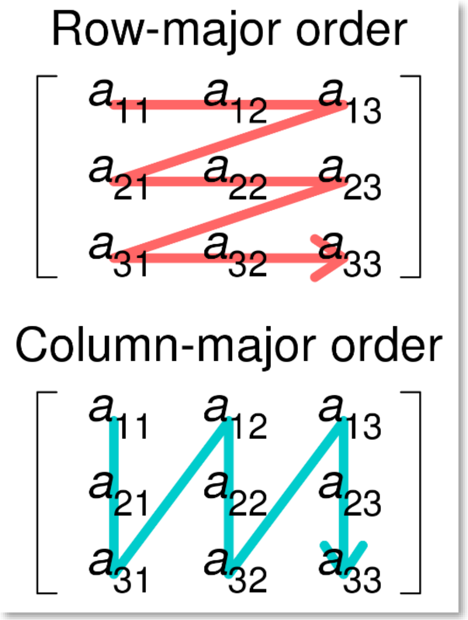
```
DO
 READ *, N
 IF (N == 0) STOP
 If(N==2) CYCLE  ! If true, the next statement is omitted
                 ! from the loop and the loop is tested again.
 CALL SUBN
END DO

DO 20 I = 1, N
        DO 20 J = 1 + I, N
20 RESULT(I,J) = 1.0 / REAL(I + J)

DO J = 1, 10, 2
  DO I = 1, 10, 2
     A(I,J) = B(I,J) +  C(I,J)
  ENDDO
ENDDO
! A=B+C !Matrix operation directly, same as the up 2 do loops

LOOP_1: DO I = 1, N
   A(I) = C * B(I)
END DO LOOP_1
DO I = 10, -10, -2
```

```
DO WHILE (LINE(I:I) .EQ. ' ')
  I = I + 1
END DO
```

```
100  I = I  + 1
…
IF ( I < 100 ) GOTO 100
```

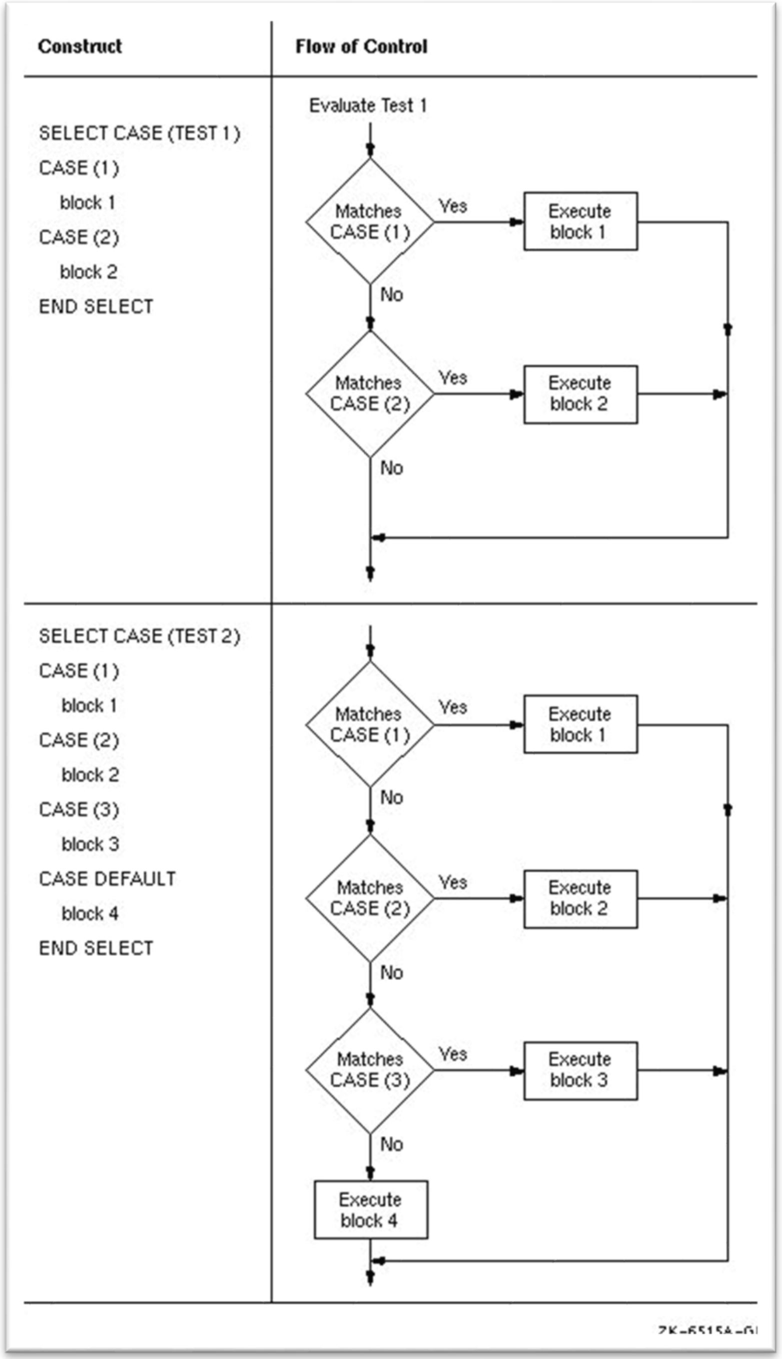
C/C++

Fortran
MATLAB
IDL



SELECT CASE

```
INTEGER FUNCTION STATUS_CODE (I)
INTEGER I
  CHECK_STATUS: SELECT CASE (I)
    CASE (:-1)
      STATUS_CODE = -1
    CASE (0)
      STATUS_CODE = 0
    CASE (1:)
      STATUS_CODE = 1
END SELECT CHECK_STATUS
END FUNCTION STATUS_CODE

SELECT CASE (J)
 CASE (1, 3:7, 9) ! Values: 1, 3, 4, 5, 6, 7, 9
    CALL SUB_A
 CASE DEFAULT
   CALL SUB_B
END SELECT
```



IF-THEN-ELSE

```
IF (X < 0.0) Y=1


IF ( X > 0.0  .AND.  Y > 0.0 ) THEN
   Z = 1.0/(X+Y)
ELSEIF ( X < 0.0  .AND.  Y < 0.0) THEN
   Z = -2.0/(X+Y)
  GO TO 200
ELSE
   PRINT*, 'ERROR CONDITION1'
ENDIF


200 PRINT*, 'ERROR CONDITION2'
```

Subprograms

- Calculations may be grouped into subroutines and functions to perform specific tasks such as:
  - read or write data
  - initialize data
  - solve a system of equations
- Function returns a single object (number, array, etc.), and usually does not alter the arguments
  - Fortran uses pass-by-reference; change of variables' values pass into subprogram will be changed after returning
    - Altering certain argument's value in a subprogram, considered a "side effect," is bad programming practice. Changing a pre-defined constant is an example. It may either cause a segmentation fault or worse, the variable got changed.
- Subroutine transfers calculated values (if any) through arguments

FUNCTION

- Definition starts with a return type
- End with "end function" analogous to "end program"
- Example: distance between two vectors

```
REAL FUNCTION FAHRENHEIT(C)
    REAL :: C
    FAHRENHEIT = (9.0/5.0)*C + 32.0  ! Convert Celsius to Fahrenheit
END FUNCTION FAHRENHEIT
```

- Use:

```
F = FAHRENHEIT(0.0)  ! 0 degree Celsius equals 32 degrees fahrenheit
```

  - Names of dummy arguments don't have to match actual names
  - Function name must be declared in calling routine

  REAL :: FAHRENHEIT

SUBROUTINE

- End with "END SUBROUTINE" analogous to "END PROGRAM"
- Distance subroutine

```
SUBROUTINE TEMP_CONVERSION(celsius, fahrenheit)
   REAL:: celsius, fahrenheit
   fahrenheit = (9.0/5.0)*celsius + 32.0
END SUBROUTINE TEMP_CONVERSION
```

- Use:

  CALL TEMP_CONVERSION(c, f)

  - As with function, names of dummy arguments don't have to match actual names

SAVE

Statement and Attribute: Causes the values and definition of objects to be retained after execution of a RETURN or END statement in a subprogram.
The SAVE attribute can be specified in a type declaration statement or a SAVE statement, and takes one of the following forms:
Type Declaration Statement:
`type,[att-ls,] SAVE [, att-ls] :: entity[, entity ] ...`
Statement:
`SAVE [[::]entity [, entity ] ...]`

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
entity	Is the name of an object, the name of a procedure pointer, or the name of a common block enclosed in slashes (/*common-block-name*/).

```
SUBROUTINE TEST()
 REAL, SAVE :: X, Y
 SAVE B, /BLOCK_B/, C, /BLOCK_D/, E
```

```
SUBROUTINE MySub
 COMMON /z/ da, in, a, idum(10)
 real(8) x,y ...
 SAVE x, y, /z/
 ! alternate declaration
 REAL(8), SAVE :: x, y
 SAVE /z/
```

BLOCK DATA

Identifies a block-data program unit, which provides initial values for variables in named common blocks.

```
BLOCK DATA [name]
     [specification-part]
END [BLOCK DATA[name]]
```

```
BLOCK DATA BLKDAT
 INTEGER S,X
 LOGICAL T,W
 DOUBLE PRECISION U
 DIMENSION R(3)
 COMMON /AREA1/R,S,U,T /AREA2/W,X,Y
 DATA R/1.0,2*2.0/, T/.FALSE./, U/0.214537D-7/, W/.TRUE./, Y/3.5/
END
```

COMMON

Defines one or more contiguous areas, or blocks, of physical storage (called common blocks) that can be accessed by any of the scoping units in an executable program(sharing data). COMMON statements also define the order in which variables and arrays are stored in each common block, which can prevent misaligned data items.

`COMMON [ /[cname]/] var-list[[,] /[cname]/ var-list]...`

cname	(Optional) Is the name of the common block. The name can be omitted for blank common (//).
var-list	Is a list of variable names, separated by commas. The variable must not be a dummy argument, allocatable array, automatic object, function, function result, a variable with the BIND attribute,or entry to a procedure. It must not have the PARAMETER attribute. If an object of derived type is specified, it must be a sequence type or a type with the BIND attribute.

```
SUBROUTINE unit1
 REAL(8) x(5)
 INTEGER J
 CHARACTER str*12
 TYPE(member) club(50)
 COMMON / blocka / x, j, str, club
 ...
END

SUBROUTINE unit2
 REAL(8) z(5)
 INTEGER m
 CHARACTER chr*12
 TYPE(member) myclub(50)
 COMMON / blocka / z, m, chr, myclub
 ...
END
```

MODULE

- Program units that group variables and subprograms
- Good for global variables (more power then COMMON)
- Checking of subprogram arguments
  - If type or number is wrong, linker will yell at you
- Can be convenient to package variables and/or subprograms of a given type
- In program unit that needs to access components of module USE *MODULE-NAME*
- USE statement must be *before* IMPLICIT NONE

```
MODULE MOD_A
 INTEGER :: B, C
 REAL E(25,5)
END MODULE MOD_A
...
SUBROUTINE SUB_Z
 USE MOD_A ! Makes scalar variables B and C, and array
 ...
 ! E available to this subroutine
 USE MOD_A ONLY: B, E !not use C
 IMPLICITE NONE
 INTEGER F
END SUBROUTINE SUB_Z
```

```
MODULE RESULTS
 ...
 CONTAINS
  FUNCTION MOD_RESULTS(X,Y)
  ! A module procedure
  ...
  END FUNCTION MOD_RESULTS
END MODULE RESULTS
```

```
MODULE ARRAY_CALCULATOR
 INTERFACE
  FUNCTION CALC_AVERAGE(D)
   REAL :: CALC_AVERAGE
   REAL, INTENT(IN) :: D(:)
  END FUNCTION
 END INTERFACE
END MODULE ARRAY_CALCULATOR
```

INTERFACE

On occasions, you may need to help the compiler a bit on what you are trying to do with *interface*

- For C programmers, this is roughly the same as prototype
- Previously, we learn to compute dot product with a function which returns a scalar. To compute cross product of two cartesian vectors, we need to warn the compiler ahead of time that the return value is not a scalar

```
!An interface to an external subroutine SUB1 with header:
!SUBROUTINE SUB1(I1,I2,R1,R2)
!INTEGER I1,I2 !REAL R1,R2
INTERFACE
 SUBROUTINE SUB1(int1,int2,real1,real2)
  INTEGER int1,int2
  REAL real1,real2
 END SUBROUTINE SUB1
END INTERFACE

INTEGER int
...
```

Internal Procedures: CONTAINS

Internal procedures are functions or subroutines that follow a CONTAINS statement in a program unit. The program unit in which the internal procedure appears is called its *host*.

Internal procedures can appear in the main program, in an external subprogram, or in a module subprogram.

An internal procedure takes the following form:

```
PROGRAM COLOR_GUIDE
 ...
 CONTAINS
  FUNCTION HUE(BLUE) ! An internal procedure
   ...
  END FUNCTION HUE
END PROGRAM
```

CONTAINS
internal-subprogram
[*internal-subprogram*] ...

internal-subprogram	Is a function or subroutine subprogram that defines the procedure. An internal subprogram must not contain any other internal subprograms.

Internal procedures are the same as external procedures, except for the following:

- Only the host program unit can use an internal procedure
- An internal procedure has access to host entities by host association; that is, names declared in the host program unit are useable within the internal procedure.
- An internal procedure must not contain an ENTRY statement.

```
program
 real a,b,c
 call find
 print *, c
 contains
  subroutine find ! Use of internal subroutine
    read *, a,b c = sqrt(a**2 + b**2)
  end subroutine find
end
```

INTRINSIC

Allows the specific name of an intrinsic procedure to be used as an actual argument.
The INTRINSIC attribute can be specified in a type declaration statement or an INTRINSIC statement, and takes one of the following forms:

Type Declaration Statement:

```
type,[att-ls,] INTRINSIC [, att-ls] :: in-pro[, in-pro]...
```

Statement:

```
INTRINSIC [::] in-pro[, in-pro] ...
```

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
in-pro	Is the name of an intrinsic procedure.

- In a type declaration statement, only *functions* can be declared INTRINSIC. However, you can use the INTRINSIC *statement* to declare subroutines, as well as functions, to be intrinsic.
- The name declared INTRINSIC is assumed to be the name of an intrinsic procedure. If a generic intrinsic function name is given the INTRINSIC attribute, the name retains its generic properties.
- Some specific intrinsic function names cannot be used as actual arguments.

```
INTRINSIC SIN, COS
 REAL X, Y, R
 ! SIN and COS are arguments to Calc2:
 R = Calc2 (SIN, COS)
```

EXTERNAL

- Allows an external procedure, a dummy procedure, a procedure pointer, or a block data subprogram to be used as an actual argument. (To specify intrinsic procedures as actual arguments, use the INTRINSIC attribute.)
- The EXTERNAL attribute can be specified in a type declaration statement or an EXTERNAL statement, and takes one of the following forms:

Type Declaration Statement:

type,[*att-ls*,] EXTERNAL [, *att-ls*] :: *ex-pro*[, *ex-pro*]...

Statement:

EXTERNAL [::]*ex-pro*[, *ex-pro*]...

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
ex-pro	Is the name of an external (user-supplied) procedure, a dummy procedure, a procedure pointer, or block data subprogram.

```
PROGRAM TEST
 ...
 INTEGER, EXTERNAL :: BETA
 LOGICAL, EXTERNAL :: COS
 ...
 CALL SUB(BETA) ! External function BETA is an actual argument
```

INCLUDE

Directs the compiler to stop reading statements from the current file and read statements in an included file or text module.

The INCLUDE line takes the following form:

`INCLUDE 'filename[/[NO]LIST]'`

filename	Is a character string specifying the name of the file to be included; it must not be a named constant. The form of the file name must be acceptable to the operating system, as described in your system documentation.
[NO]LIST	Specifies whether the incorporated code is to appear in the compilation source listing. In the listing, a number precedes each incorporated statement. The number indicates the "include" nesting depth of the code. The default is /NOLIST. /LIST and /NOLIST must be spelled completely. You can only use /[NO]LIST if you specify compiler option vms (which sets OpenVMS defaults).

Main Program File

```
PROGRAM
 INCLUDE 'COMMON.FOR'
 REAL, DIMENSION(M) :: Z
 CALL CUBE
 DO I = 1, M
  Z(I) = X(I) + SQRT(Y(I))
  ...
 END DO
END

SUBROUTINE CUBE
 INCLUDE 'COMMON.FOR'
 DO I=1,M
   X(I) = Y(I)**3
 END DO
 RETURN
END
```

COMMON.FOR File

```
INTEGER, PARAMETER :: M=100
REAL, DIMENSION(M) :: X, Y
COMMON X, Y
```

Subprogram Example

```
module Ray
    real(8), save:: V(3), X0(3)
    logical, save:: triio
endmodule Ray

Program main
Implicit none
…
Call  Triangle_Intersect(P, tmp, Flag)
End

function Cross_Product(a, b)
    real(8),dimension(3):: Cross_Product,a(3),b(3)
    Cross_Product=(/a(2)  *b(3) - a(3) * b(2),   a(3) * b(1) -   &
 a(1) * b(3),    a(1) * b(2) - a(2) * b(1)/)
endfunction
```

```
subroutine Triangle_Intersect(P, tmp, Flag)
use Ray
implicit none
integer Flag,  i0, ii(1)
real(8) P(3, 3), tmp,  N(3), d, Beta, Tp(3), Q(3), NV
interface
    function Cross_Product(a, b)
    real(8),dimension(3):: Cross_Product,a,b
    endfunction
endinterface
Flag=0; tmp=0.0d0
N=Cross_Product(P(2, :) - P(1, :), P(3, :) - P(1, :))
end
```

OPEN and CLOSE file

Connects an external file to a unit, creates a new file and connects it to a unit, creates a preconnected file, or changes certain properties of a connection.

```
OPEN ([UNIT=] io-unit[, FILE= name] [, ERR= label] [, IOMSG=msg-var] [, IOSTAT=i-var],slist)
```

```
OPEN (unit=10, FILE='test.dat', FORM='FORMATTED', STATUS='OLD')
OPEN (10, FILE='test.dat', FORM= 'UNFORMATTED', STATUS= 'UNKNOWN')
CLOSE (10)
```

FORM = ***m***

m	Is a scalar default character expression that evaluates to one of the following values:
'FORMATTED'	Indicates formatted data transfer
'UNFORMATTED'	Indicates unformatted data transfer
'BINARY'	Indicates binary data transfer

STATUS = sta

sta	Is a scalar default character expression that evaluates to one of the following values:
'OLD'	Indicates an existing file.
'NEW'	Indicates a new file; if the file already exists, an error occurs. Once the file is created, its status changes to 'OLD'.
'SCRATCH'	Indicates a new file that is unnamed (called a scratch file). When the file is closed or the program terminates, the scratch file is deleted.
'REPLACE'	Indicates the file replaces another. If the file to be replaced exists, it is deleted and a new file is created with the same name. If the file to be replaced does not exist, a new file is created and its status changes to 'OLD'.
'UNKNOWN'	Indicates the file may or may not exist. If the file does not exist, a new file is created and its status changes to 'OLD'.

READ data

Input data from screen or files.

```
DIMENSION ia(10,20)
! Read in the bounds for the array.
! Then read in the array in nested implied-DO lists
! with input format of 8 columns of width 5 each.
! 6 is the file number.
 OPEN(6, FILE='input.dat’ )
 READ (6, 990) il, jl, ((ia(i,j), j = 1, jl), i =1, il)
 990 FORMAT (2I5, /, (8I5))
 CLOSE(6) !CLOSE the file.

! Internal read gives a variable string-represented numbers
 CHARACTER*12 str
 str = '123456’
 READ (str,'(i6)') i

! List-directed read uses no specified format
 REAL x, y
 INTEGER i, j
 READ (*,*) x, y, i, j
```

PRINT, WRITE, TYPE data

```
CHARACTER*16 NAME, JOB
PRINT *, NAME, JOB !* mean default format
PRINT 400, NAME, JOB ! 400 mean use 400 label format
400 FORMAT ('NAME=', A, 'JOB=', A)
OPEN(6,FILE='output.dat', FORM='FORMATTED',STATUS='NEW’ )
WRITE (6, '(A11)') 'Abbottsford’ ! Output to file 6
WRITE (6, ‘(A, F6.2, I5, ES15.3)’) 'Answers are ', x, j, y !Format output
CLOSE(6)
```

```
! The following statements are equivalent:
PRINT '(A11)', 'Abbottsford’
WRITE (*, '(A11)') 'Abbottsford’ !* mean screen
TYPE '(A11)', 'Abbottsford'
```

Unformatted I/O

- Binary data take less disk space than ASCII (formatted) data
- Data can be written to file in binary representation
  - Not directly human-readable

```
OPEN (199, file = 'unf.dat' , FORM = 'UNFORMATTED' )
WRITE (199)  x(1:100000),  j1, j2
READ (199)    x(1:100000),  j1, j2
```

  - Note that there is no format specification
- Fortran unformatted slightly different format than C binary
  - Fortran unformatted contains record delimiters

Understanding File Extensions
(For Intel Fortran Complier)

The file extension determines whether a file gets passed to the compiler or to the linker. The following types of files are used with the compiler:

- Files passed to the compiler: `.f90`, `.for`, `.f`, `.fpp`, `.i`, `.i90`, `.ftn`
- Typical Fortran source files have a file extension of `.f90`, `.for`, and `.f`. When editing your source files, you need to choose the source form, either free-source form or fixed-source form (or a variant of fixed form called tab form). You can use a compiler option to specify the source form used by the source files or you can use specific file extensions when creating or renaming your files. For example, the compiler assumes that files with an extension of:
  - `.f90` or `.i90`: free-form source files
  - `.f`, `.for`, `.ftn`, or `.i`: fixed-form (or tab-form) files
- Files passed to the linker: `.a`, `.lib`, `.obj`, `.o`, `.exe`, `.res`, `.rbj`, `.def`, `.dll`

Input File Extensions

Filename	Interpretation	Action
file.a (Linux* and OS X*) file.lib (Windows*)	Object library	Passed to the linker.
file.f file.for file.ftn file.i	Fortran fixed-form source	Compiled by the Intel Fortran compiler.
file.fpp On Linux*, filenames with the following uppercase extensions: file.FPP file.F file.FOR file.FTN	Fortran fixed-form source	Automatically preprocessed by the Intel Fortran preprocessor fpp; then compiled by the Intel Fortran compiler.
file.f90 file.i90	Fortran free-form source	Compiled by the Intel Fortran compiler.
file.F90 (Linux* and OS X*)	Fortran free-form source	Automatically preprocessed by the Intel Fortran preprocessor fpp; then compiled by the Intel Fortran compiler.
file.s (Linux* and OS X*) file.asm (Windows*)	Assembly file	Passed to the assembler.
file.o (Linux* and OS X*) file.obj (Windows*)	Compiled object file	Passed to the linker.

Compilation

- A compiler is a program that reads source code and converts it to a form usable by the computer
- Internally, three steps are performed:
  - **preprocess** source code
  - **check** source code for syntax errors
  - **compiler** translates source code to assembly language
  - **assembler** translates assembly language to machine language
  - **linker** gathers machine-language modules and libraries
  - All these steps sometimes loosely referred to as “compiling”

Compiling Commands

-o outfilename

Enable debug mode: -g

- Intel compiler
  - Serial: ifort myprog.f90 -o myprog
  - OpenMP: ifort -qopenmp my-openmp-prog.f90 -o my-openmp-prog
- PGI compiler
  - Serial:
    - F90: pgif90 myprog.f90 -o myprog
    - F77: pgf77 myprog.f –o myprog
  - OpenMP: pgf90 -mp my-openmp-prog.f90 -o my-openmp-prog
- GNU compiler
  - Serial: gfortran myprog.f90 -o myprog
  - OpenMP: gfortran -fopenmp my-openmp-prog.f90 -o my-openmp-prog
- MPI
  - Intel MPI: mpiifort, mpif90, mpif77
  - HPC-X(Include Open MPI) and Open MPI: mpifort, mpif90, mpif77
  - MPICH: mpif90, mpif77

Understanding Errors During the Build Process

Intel compiler have the following format:

filename(linenum): severity #error number: message

Diagnostic	Meaning
filename	Indicates the name of the source file currently being processed.
linenum	Indicates the source line where the compiler detects the condition.
severity	Indicates the severity of the diagnostic message: *Warning*, *Error*, or *Fatal error*.
message	Describes the problem.

Error Message Example
echar.for(7): error #6321: An unterminated block exits.   DO I=1,5 -------^

The pointer (−−−^) indicates the place on the source program line where the error was found

GNU Fortran：

Filename:linenumber:position

```
NOlihm.f90:146.14:


  n2nd=0;  npr=0
              1
Error: Symbol 'npr' at (1) has no IMPLICIT type
```

Run-Time Message Display and Format

```
x = -1.0000000E+32 x*100.0 = -1.0000000E+34
forrtl: error (72): floating overflow
Image PC Routine Line Source ovf.exe 08049E4A MAIN__ 14  ovf.f90
ovf.exe 08049F08 Unknown Unknown Unknown
Ovf.exe 400B3507 Unknown Unknown Unknown
ovf.exe 08049C51 Unknown Unknown Unknown
Abort
```

Fortran run-time messages have the following format:

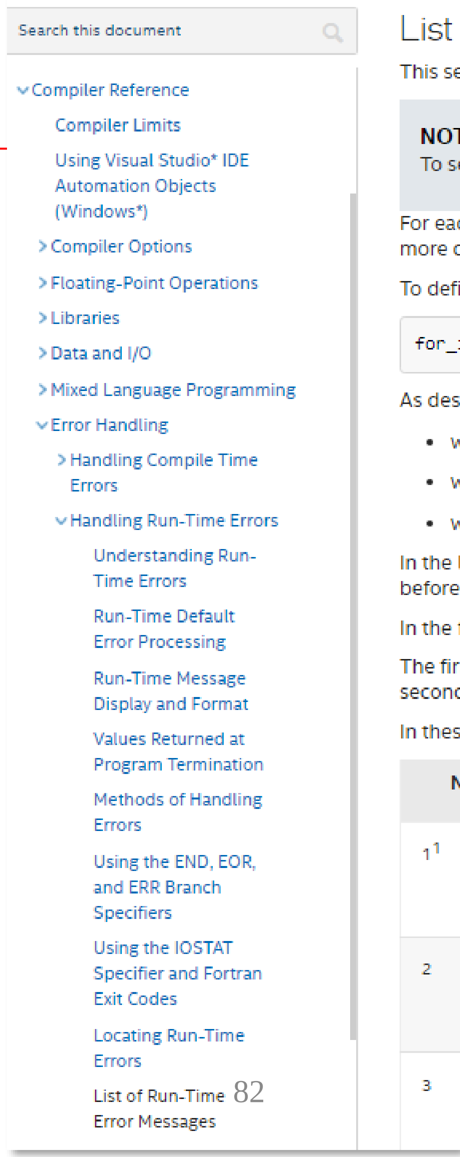
forrtl: *severity* (*number*): *message-text*

where:

- ***forrtl***: Identifies the source as the Intel® Fortran run-time system (Run-Time Library or RTL).
- ***severity***: The severity levels are: *severe* , *error*, *warning*, or *info*
- ***number***: This is the message number
- ***message-text***: Explains the event that caused the message.

Severity	Description
severe	Must be corrected. The program's execution is terminated when the error is encountered unless the program's I/O statements use the END, EOR, or ERR branch specifiers to transfer control, perhaps to a routine that uses the IOSTAT specifier.
error	Should be corrected. The program might continue execution, but the output from this execution might be incorrect.
warning	Should be investigated. The program continues execution, but output from this execution might be incorrect.
info	For informational purposes only; the program continues.

List of Run-Time Error Messages: Intel Fortran Compiler Classic and Intel Fortran Compiler Developer Guide and Reference
https://www.intel.com/content/www/us/en/develop/documentation/fortran-compiler-oneapi-dev-guide-and-reference/top/compiler-reference/error-handling/handling-run-time-errors/list-of-run-time-error-messages.html



Some document

Supercomputing Center of USTC: http://scc.ustc.edu.cn/

Thank you!

Welcome USTC

中国科学技术大学

University of Science and Technology of China