

Introduction to FORTRAN

- History and purpose of FORTRAN
- FORTRAN essentials
 - ◆ Program structure
 - ◆ Data types and specification statements
 - ◆ Essential program control
 - ◆ FORTRAN I/O
 - ◆ subfunctions and subroutines
- Pitfalls and common coding problems
- Sample problems

By: Manoj Malik

FORTRAN History

- One of the oldest computer languages
 - ◆ created by John Backus and released in 1957
 - ◆ designed for scientific and engineering computations
- Version history
 - ◆ FORTRAN 1957
 - ◆ FORTRAN II
 - ◆ FORTRAN IV
 - ◆ FORTRAN 66 (released as ANSI standard in 1966)
 - ◆ FORTRAN 77 (ANSI standard in 1977)
 - ◆ FORTRAN 90 (ANSI standard in 1990)
 - ◆ FORTRAN 95 (ANSI standard version)
 - ◆ FORTRAN 2003 (ANSI standard version)
- Many different “dialects” produced by computer vendors (one of most popular is Digital VAX Fortran)
- Large majority of existing engineering software is coded in FORTRAN (various versions)

Why FORTRAN

- FORTRAN was created to write programs to solve scientific and engineering problems
- Introduced integer and floating point variables
- Introduced array data types for math computations
- Introduced subroutines and subfunctions
- Compilers can produce highly optimized code (fast)
- Lots of available numerical-math libraries
- Problems
 - ◆ encouraged liberal use of GO TO statements
 - ◆ resulted in hard to decipher and maintain (“spaghetti”) code
 - ◆ limited ability to handle nonnumeric data
 - ◆ no recursive capability (not completely true)

FORTRAN Today

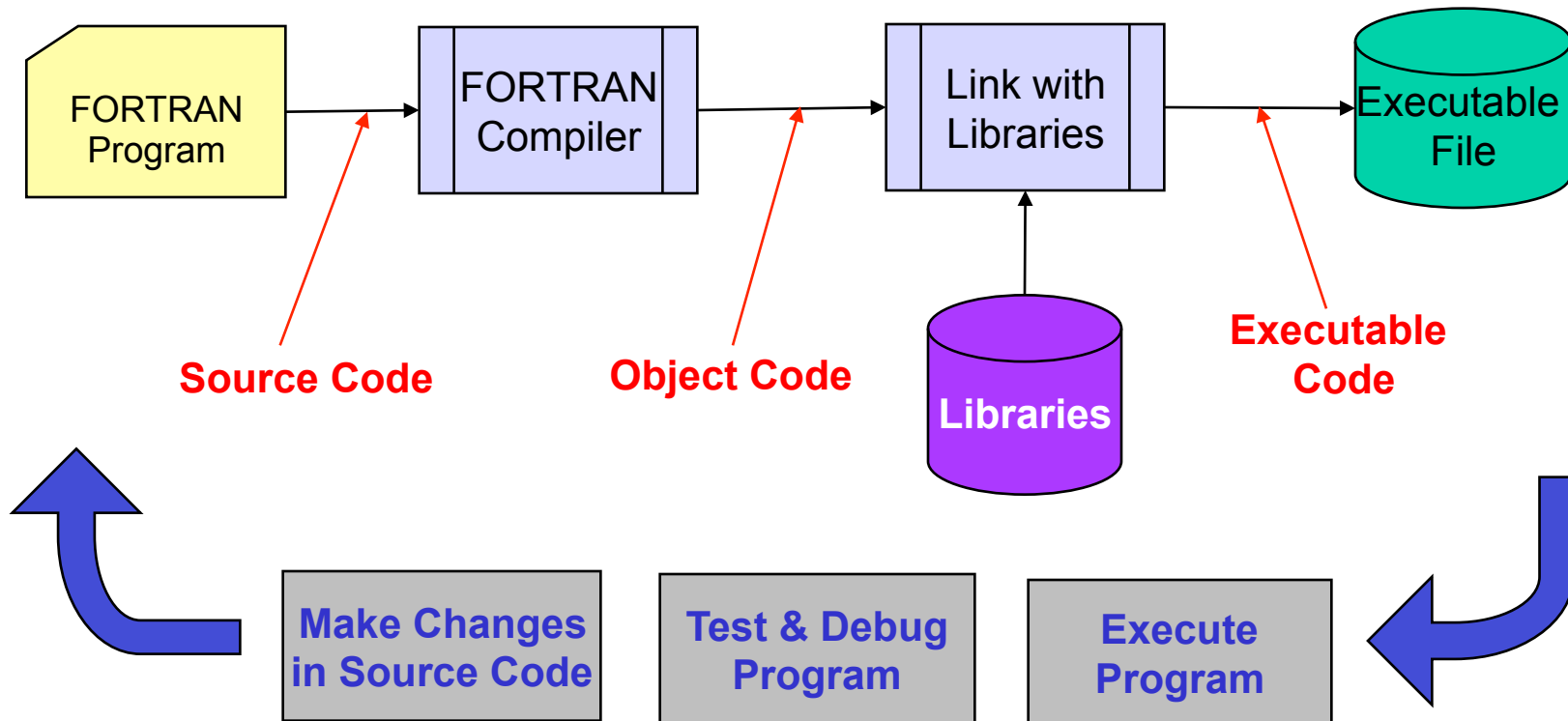
- FORTRAN 77 is “standard” but FORTRAN 90/95 has introduced contemporary programming constructs
- There are proprietary compilers
 - ◆ Compaq/HP Visual Fortran; Absoft Fortran; Lahey Fortran
- There is a free compiler in Unix-Linux systems
 - ◆ f77, g77
 - ◆ g95, gfortran
- Available scientific libraries
 - ◆ LINPACK: early effort to develop linear algebra library
 - ◆ EISPACK: similar to Linpack
 - ◆ IMSL: commercial library (\$'s)
 - ◆ NAG: commercial library (\$'s)

Class Objectives

- Not nearly enough time to teach all the details of FORTRAN (which has evolved into a VERY complex language with many “dialects” ...)
- We’ ll try to highlight some of the most important features:
 - ◆ that are confusing or often lead to problems,
 - ◆ that appear in older programs written in FORTRAN 77 (or IV)
 - ◆ that are quite different from contemporary languages
 - ◆ For example:
 - I/O instructions
 - variable declarations
 - subprograms: functions and subroutines
- We’ ll look at some code fragments, and
- You’ ll program a simple example problem

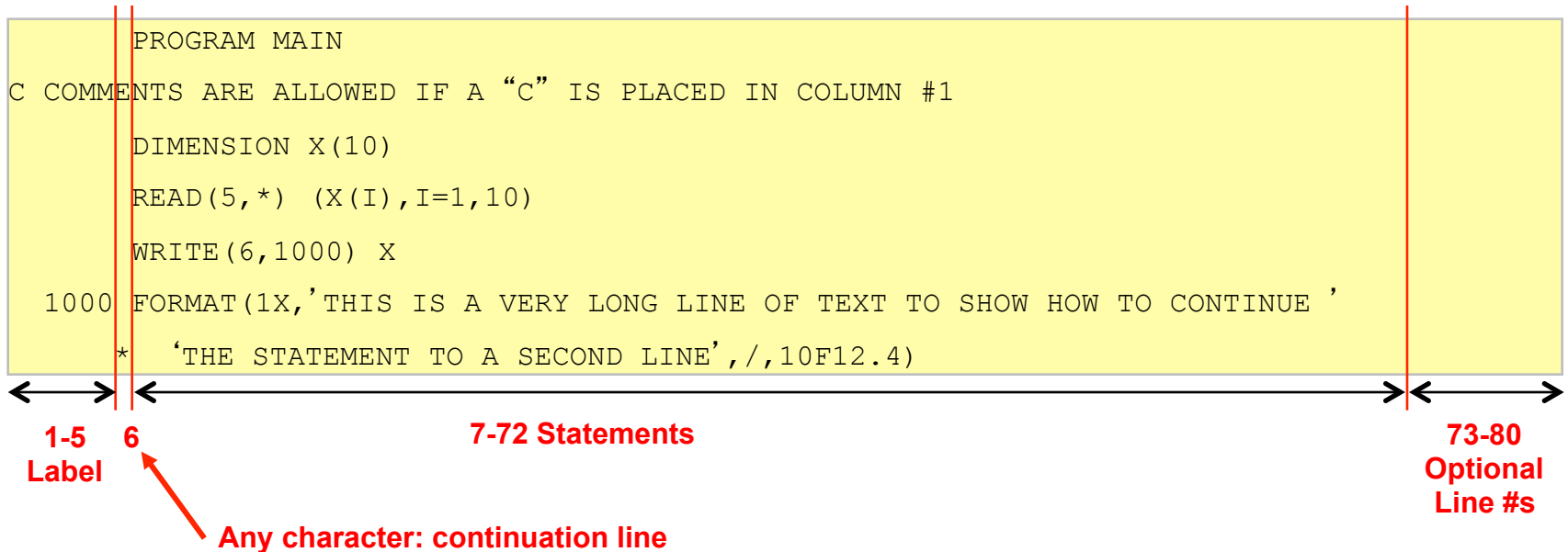
How to Build a FORTRAN Program

- FORTRAN is a compiled language (like C) so the source code (what you write) must be converted into machine code before it can be executed (e.g. Make command)



Statement Format

- FORTRAN 77 requires a fixed format for programs



- FORTRAN 90/95 relaxes these requirements:
 - ◆ allows free field input
 - ◆ comments following statements (! delimiter)
 - ◆ long variable names (31 characters)

Program Organization

- Most FORTRAN programs consist of a main program and one or more subprograms (subroutines, functions)
- There is a fixed order:

```
Heading  
Declarations  
Variable initializations  
Program code  
Format statements  
  
Subprogram definitions  
(functions & subroutines)
```


Data Type Declarations

- Basic data types are:
 - ◆ **INTEGER** – integer numbers (+/-)
 - ◆ **REAL** – floating point numbers
 - ◆ **DOUBLE PRECISION** – extended precision floating point
 - ◆ **CHARACTER*n** – string with up to n characters
 - ◆ **LOGICAL** – takes on values **.TRUE.** or **.FALSE.**
 - ◆ **COMPLEX** – complex number
- Integer and Reals can specify number of bytes to use
 - ◆ Default is: **INTEGER*4** and **REAL*4**
 - ◆ **DOUBLE PRECISION** is same as **REAL*8**
- Arrays of any type must be declared:
 - ◆ **DIMENSION A(3,5)** – declares a 3 x 5 array (implicitly REAL)
 - ◆ **CHARACTER*30 NAME(50)** – directly declares a character array with 30 character strings in each element
- FORTRAN 90/95 allows user defined types

Implicit vs Explicit Declarations

- By default, an implicit type is assumed depending on the first letter of the variable name:
 - ◆ A-H, O-Z define **REAL** variables
 - ◆ I-N define **INTEGER** variable
- Can use the **IMPLICIT** statement:
 - ◆ **IMPLICIT REAL (A-Z)** makes all variables **REAL** if not declared
 - ◆ **IMPLICIT CHARACTER*2 (W)** makes variables starting with W be 2-character strings
 - ◆ **IMPLICIT DOUBLE PRECISION (D)** makes variables starting with D be double precision
- **Good habit:** force **explicit** type declarations
 - ◆ **IMPLICIT NONE**
 - ◆ User must explicitly declare all variable types

Other Declarations

- Define constants to be used in program:
 - ◆ `PARAMETER (PI=3.1415927, NAME='BURDELL')`
 - ◆ `PARAMETER (PIO2=PI/2, FLAG=.TRUE.)`
 - ◆ these cannot be changed in assignments
 - ◆ can use parameters to define other parameters
- Pass a function or subroutine name as an argument:
 - ◆ `INTRINSIC SIN` – the `SIN` function will be passed as an argument to a subprogram (subroutine or function)
 - ◆ `EXTERNAL MYFUNC` – the `MYFUNC` function defined in a `FUNCTION` subprogram will be passed as an argument to another subprogram

Initializing Variables

- The **DATA** statement can be used to initialize a variable:
 - ◆ **DIMENSION A(10,10)** – dimension a REAL array
 - ◆ **DATA A/100*1.0/** - initializes all values to 1.0
 - ◆ **DATA A(1,1),A(10,1),A(5,5) /2*4.0,-3.0/** - initialize by element
 - ◆ **DATA ((A(I,J),I=1,5,2),J=1,5) /15*2.0/** - initialize with implied-do list
 - ◆ **DATA FLAG /.TRUE./** - initialize a LOGICAL
 - ◆ **DATA NAME /30*' */** - initialize a CHARACTER string
- Cannot initialize:
 - ◆ dummy argument in a function or subroutine definition
 - ◆ function, function result
 - ◆ variables in COMMON blocks (more details later...)
- **DATA** statements can appear within the program code

FORTRAN Assignment Statements

- Assignment statement:

`<label> <variable> = <expression>`

- ◆ `<label>` - statement label number (1 to 99999)
- ◆ `<variable>` - FORTRAN variable (max 6 characters, alphanumeric only for standard FTN-77)

- Expression:

- ◆ Numeric expressions: `VAR = 3.5 * COS (THETA)`
- ◆ Character expressions: `DAY (1 : 3) = 'TUE'`
- ◆ Relational expressions: `FLAG = ANS .GT. 0`
- ◆ Logical expressions: `FLAG = F1 .OR. F2`

Numeric Expressions

- Very similar to other languages

- ◆ Arithmetic operators:
- ◆ Precedence: ** (high) → - (low)

Operator	Function
**	exponentiation
*	multiplication
/	division
+	addition
-	subtraction

- ◆ Casting: numeric expressions are up-cast to the highest data type in the expression according to the precedence:
(low) logical – integer – real – complex (high) and smaller byte size (low) to larger byte size (high)

- Example

$3.42 + (A1+C0) / \text{SIN}(A) - R^{**3}$

Character Expressions

- Only built-in operator is Concatenation
 - ◆ defined by `//` - `'ILL' // '-' // 'ADVISED'`
- Character arrays are most commonly encountered...
 - ◆ treated like any array (indexed using `:` notation)
 - ◆ fixed length (usually padded with blanks)
 - ◆ Example:

CODE

```
CHARACTER FAMILY*16  
FAMILY = 'GEORGE P. BURDELL'  
PRINT*, FAMILY (:6)  
PRINT*, FAMILY (8:9)  
PRINT*, FAMILY (11:)  
PRINT*, FAMILY (:6) // FAMILY (10:)
```

OUTPUT

```
GEORGE  
P.  
BURDELL  
GEORGE BURDELL
```

Hollerith Constants

- This is a relic of early FORTRAN that did not have the CHARACTER type..
- Used to store ASCII characters in numeric variables using one byte per character
- Examples: **2HQW**, **4H1234**, **10HHELLOWORLD**
- Binary, octal, hexadecimal and hollerith constants have no intrinsic data type and assume a numeric type depending on their use

```
INTEGER*4 IWORD, KWORD
INTEGER*2 CODE
REAL*8 TEST
CODE = 2HXZ
IWORD = 4HABCD
KWORD = O'4761' (octal)
TEST = Z'3AF2' (hexidecimal)
```

- This can be VERY confusing; consult FORTRAN manual for target compiler! (avoid whenever possible)

Relational Expressions

- Two expressions whose values are compared to determine whether the relation is true or false
 - ◆ may be numeric (common) or non-numeric
 - ◆ Relational operators:

Operator	Relationship
<code>.LT.</code> or <code><</code>	less than
<code>.LE.</code> or <code><=</code>	less than or equal to
<code>.EQ.</code> or <code>==</code>	equal to
<code>.NE.</code> or <code>/=</code>	not equal to
<code>.GT.</code> or <code>></code>	greater than
<code>.GE.</code> or <code>>=</code>	greater than or equal to

- Character strings can be compared
 - ◆ done character by character
 - ◆ shorter string is padded with blanks for comparison

Logical Expressions

- Consists of one or more logical operators and logical, numeric or relational operands
 - ◆ values are `.TRUE.` or `.FALSE.`
 - ◆ Operators:

Operator	Example	Meaning
<code>.AND.</code>	<code>A .AND. B</code>	logical AND
<code>.OR.</code>	<code>A .OR. B</code>	logical OR
<code>.NEQV.</code>	<code>A .NEQV. B</code>	logical inequivalence
<code>.XOR.</code>	<code>A .XOR. B</code>	exclusive OR (same as <code>.NEQV.</code>)
<code>.EQV.</code>	<code>A .EQV. B</code>	logical equivalence
<code>.NOT.</code>	<code>.NOT. A</code>	logical negation

- Need to consider overall operator precedence (next slide)
- Remark: can combine logical and integer data with logical operators but this is tricky (avoid!)

Operator Precedence

- Can be tricky; use () when in doubt...

Category	Operator	Precedence
numeric	**	highest
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.	lowest

Arrays in FORTRAN

- Arrays can be multi-dimensional (up to 7) and are indexed using ():
 - ◆ `TEST (3)`
 - ◆ `FORCE (4 , 2)`
- Indices are normally defined as 1...N
- Can specify index range in declaration
 - ◆ `REAL L (2:11 , 5)` – L is dimensioned with rows numbered 2-11 and columns numbered 1-5
 - ◆ `INTEGER K (0:11)` – K is dimensioned from 0-11 (12 elements)
- Arrays are stored in column order (1st column, 2nd column, etc) so accessing by incrementing row index first usually is fastest.
- Whole array reference:
 - ◆ `K=-8` - assigns 8 to all elements in K (not in 77)

Execution Control in FORTRAN

- Branching statements (**GO TO** and variations)
- IF constructs (**IF**, **IF-ELSE**, etc)
- **CASE (90+)**
- Looping (**DO**, **DO WHILE** constructs)
- **CONTINUE**
- **PAUSE**
- **STOP**
- **CALL**
- **RETURN**
- **END**

NOTE:

We will try to present the FORTRAN 77 versions and then include some of the common variations that may be encountered in older versions.

Unconditional GO TO

- This is the only GOTO in FORTRAN 77
 - ◆ Syntax: **GO TO label**
 - ◆ Unconditional transfer to labeled statement

```
10  -code-  
    GO TO 30  
    -code that is bypassed-  
30  -code that is target of GOTO-  
    -more code-  
    GO TO 10
```

- Flowchart:



- Problem: leads to confusing “spaghetti code”

Other GO TO Statements

- Computed GO TO

- ◆ Syntax: **GO TO (list_of_labels) [,] expression**
- ◆ selects from list of labels based on ordinal value of expression
- ◆ Ex: **GO TO (10, 20, 30, 50) KEY+1**

- Assigned GO TO

- ◆ Syntax: **ASSIGN label TO intvar**
GO TO intvar [[,] (list_of_valid_labels)]
- ◆ Ex: **ASSIGN 100 TO L2**
- code -
GO TO L2, (10, 50, 100, 200)

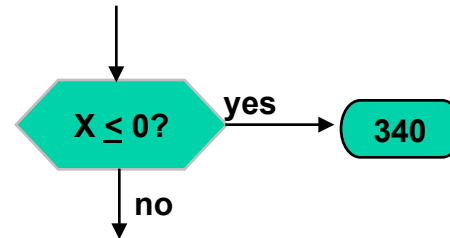
NOTE:

In syntax, [] means items enclosed are optional

IF Statements

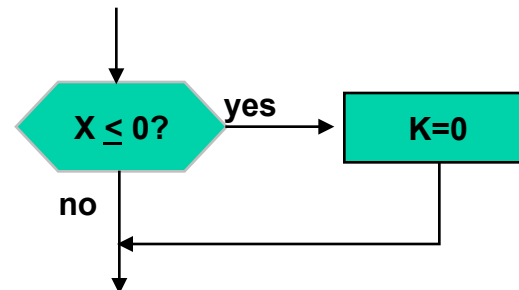
● Basic version #1

- ◆ Syntax: **IF (logical_expression) GO TO label**
- ◆ If logical expression is true, execute GO TO, otherwise continue with next statement
- ◆ Ex: **IF (X.LE.0) GO TO 340**
- ◆ Flowchart:



● Basic version #2

- ◆ Syntax: **IF (logical_expression) statement**
- ◆ If logical expression is true, execute statement and continue, otherwise, skip statement and continue
- ◆ Ex: **IF (K.LE.0) K=0**
- ◆ Flowchart



IF THEN ELSE Statement

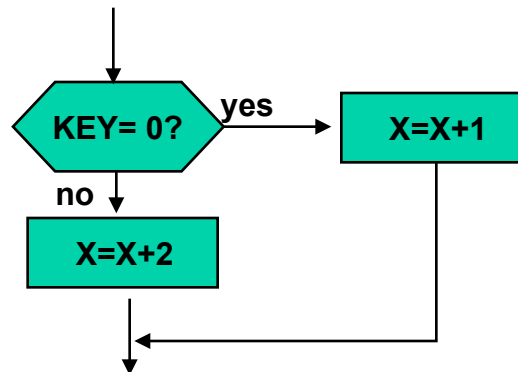
- Basic version:

- ◆ Syntax: **IF (logical_expression) THEN**
statement1(s)
ELSE
statement2(s)
ENDIF

- ◆ If logical expression is true, execute statement1(s), otherwise execute statements2(s), then continue after ENDIF.

- ◆ Ex: **IF (KEY.EQ.0) THEN**
X=X+1
ELSE
X=X+2
ENDIF

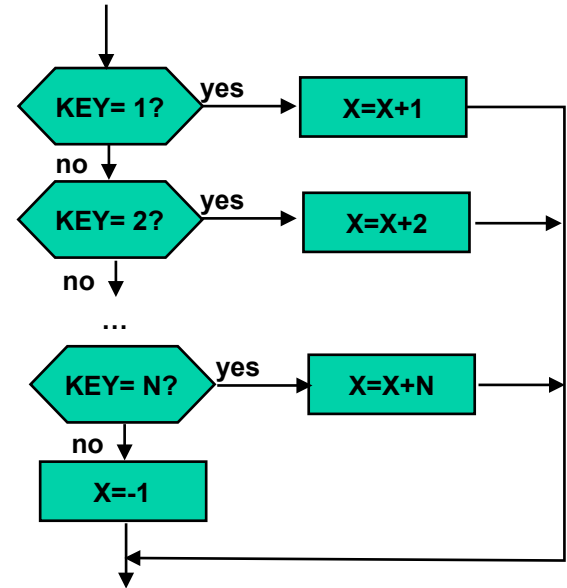
- ◆ Flowchart:



IF ELSE IF Statement

- Basic version:

- ◆ Syntax: **IF (logical_expr1) THEN**
statement1(s)
ELSE IF (logical_expr2) THEN
statement2(s)
ELSE
statement3(s)
ENDIF



- ◆ If logical expr1 is true, execute statement1(s), if logical expr2 is true, execute statement2(s), otherwise execute statements3(s).
- ◆ Ex:

```
10 IF (KSTAT.EQ.1) THEN
    CLASS='FRESHMAN'
ELSE IF (KSTAT.EQ.2) THEN
    CLASS='SOPHOMORE'
ELSE IF (KSTAT.EQ.3) THEN
    CLASS='JUNIOR'
ELSE IF (KSTAT.EQ.4) THEN
    CLASS='SENIOR'
ELSE
    CLASS='UNKNOWN'
ENDIF
```

Notes on IF Statements

- Avoid using **IF** with **GO TO** which leads to complex code
- Statement blocks in **IF THEN** and **IF ELSE IF** statements can contain almost any other executable FORTRAN statements, including other **IF**'s and loop statements.
- CANNOT transfer control into an **IF** statement block from outside (e.g., using a **GO TO**)
- CAN transfer control out of an **IF** statement block (e.g., using an **IF () GO TO N** statement)
- Use indenting to make code readable.

Old IF Statements

- Arithmetic IF statement (3-branch)
 - ◆ Syntax: **IF (num_expr) label1, label2, label3**
 - ◆ If num expr is <0 then go to label1, if =0 then label2, and if >0 then go to label3
 - ◆ Ex: **IF (THETA) 10, 20, 100**
- Arithmetic IF statement (2-branch)
 - ◆ Syntax: **IF (num _ expr) label1, label2**
 - ◆ If num expr is <0 then go to label1, if >=0 then go to label2
 - ◆ Ex: **IF (ALPHA-BETA) 120, 16**
- Notes:
 - ◆ Avoid whenever possible!
 - ◆ Leads to very confusing and hard to understand code..

Spaghetti Code

- Use of GO TO and arithmetic IF's leads to bad code that is very hard to maintain
- Here is the equivalent of an IF-THEN-ELSE statement:

```
10 IF (KEY.LT.0) GO TO 20
   TEST=TEST-1
   THETA=ATAN(X,Y)
   GO TO 30
20 TEST=TEST+1
   THETA=ATAN(-X,Y)
30 CONTINUE
```

- Now try to figure out what a complex IF ELSE IF statement would look like coded with this kind of simple IF. . .

Loop Statements

- DO loop: structure that executes a specified number of times
- Nonblock DO

- ◆ Syntax: **DO label , loop_control**
do_block
label terminating_statement

- ◆ Execute do_block including terminating statement, a number of times determined by loop-control

- ◆ Ex:

```
DO 100 K=2,10,2
PRINT*,A(K)
100 CONTINUE
```

Spaghetti Code Version

```
K=2
10 PRINT*,A(K)
K=K+2
IF (K.LE.11 GO TO 10
20 CONTINUE
```

- ◆ Loop_control can include variables and a third parameter to specify increments, including negative values.
- ◆ Loop always executes ONCE before testing for end condition

Loop Statements – cont' d

- WHILE DO statement

- ◆ Syntax: **WHILE (logical_expr) DO
statement(s)
ENDWHILE**

- ◆ Executes statement(s) as long as logical_expr is true, and exits when it is false. Note: must preset logical_expr to true to get loop to start and must at some point set it false in statements or loop will execute indefinitely.

- ◆ Ex:

```
READ* ,R  
WHILE (R.GE.0) DO  
  VOL=2*PI*R**2*CLEN  
  READ* ,R  
ENDWHILE
```

- ◆ Use when cannot determine number of loops in advance.
- ◆ CANNOT transfer into WHILE loop.
- ◆ CAN transfer out of WHILE loop.

New Loop Statements

- Block DO

- ◆ Syntax: **DO loop_control**
do_block
END DO

- ◆ Execute do_block including terminating statement, a number of times determined by loop-control

- ◆ Ex:

```
DO K=2,10,2  
  PRINT*,A(K)  
END DO
```

- ◆ Loop_control can include a third parameter to specify increments, including negative values.

- ◆ Loop always executes ONCE before testing for end condition

- ◆ If loop_control is omitted, loop will execute indefinitely or until some statement in do-block transfers out.

New Loop Statements – cont' d

- General DO

- ◆ Syntax: **DO**
 statement_sequence1
 IF (logical_expr) EXIT
 statement_sequence2
 END DO

- ◆ Execute do_block including terminating statement and loop back continually (without the IF this is basically an “infinite loop”)

- ◆ Must include some means (i.e., IF) to exit loop

- ◆ Ex:

```
DO
  READ*,R
  IF (R.LT.0) EXIT
  VOL=2*PI*R**2*CLEN
  PRINT*,R
END DO
```

- ◆ Loop always starts ONCE before testing for exit condition

- ◆ If EXIT is omitted, loop will execute indefinitely or until some statement in do-block transfers out.

New Loop Statements - cont' d

● DO WHILE

◆ Syntax: **DO [label][,] WHILE (logical_expr)**
do_block
[label] END DO

◆ Execute do_block while logical_expr is true, exit when false

◆ Ex:

```
READ*,R
DO WHILE (R.GE.0)
  VOL=2*PI*R**2*CLEN
  READ*,R
END DO
```

```
READ*,R
DO 10 WHILE (R.GE.0)
  VOL=2*PI*R**2*CLEN
  READ*,R
10 CONTINUE
```

◆ Loop will not execute at all if logical_expr is not true at start

Comments on Loop Statements

- In old versions:
 - ◆ to transfer out (exit loop), use a **GO TO**
 - ◆ to skip to next loop, use **GO TO** terminating statement (this is a good reason to always make this a **CONTINUE** statement)
- In NEW versions:
 - ◆ to transfer out (exit loop), use **EXIT** statement and control is transferred to statement following loop end. This means you cannot transfer out of multiple nested loops with a single **EXIT** statement (use **GO TO** if needed). This is much like a **BREAK** statement in other languages.
 - ◆ to skip to next loop cycle, use **CYCLE** statement in loop.

Input and Output Statements

- FORTRAN has always included a comprehensive set of I/O instructions.
 - ◆ Can be used with standard input and output devices such as keyboards, terminal screens, printers, etc.
 - ◆ Can be used to read and write files managed by the host OS.
- Basic instructions:
 - ◆ **READ** – reads input from a standard input device or a specified device or file.
 - ◆ **WRITE** – writes data to a standard output device (screen) or to a specified device or file.
 - ◆ **FORMAT** – defines the input or output format.
- Advanced instructions
 - ◆ Used to manipulate files maintained by the OS file manager.
 - ◆ Often dependent on features in a particular OS or computer.

READ Statement

- Format controlled READ:

- ◆ Syntax: **READ(dev_no, format_label) variable_list**
- ◆ Read a record from dev_no using format_label and assign results to variables in variable_list
- ◆ Ex:

```
READ (5,1000) A,B,C
      1000 FORMAT (3F12.4)
```
- ◆ Device numbers 1-7 are defined as standard I/O devices and 1 is the keyboard, but 5 is also commonly taken as the keyboard (used to be card reader)
- ◆ Each READ reads one or more lines of data and any remaining data in a line that is read is dropped if not translated to one of the variables in the variable_list.
- ◆ Variable_list can include implied DO such as:

```
READ (5,1000) (A(I), I=1,10)
```

READ Statement – cont' d

- List-directed READ

- ◆ Syntax: **READ*, variable_list**
- ◆ Read enough variables from the standard input device (usually a keyboard) to satisfy variable_list
 - input items can be integer, real or character.
 - characters must be enclosed in ' '.
 - input items are separated by commas.
 - input items must agree in type with variables in variable_list.
 - as many records (lines) will be read as needed to fill variable_list and any not used in the current line are dropped.
 - each READ processes a new record (line).
- ◆ Ex: **READ* ,A ,B ,K** – read line and look for floating point values for A and B and an integer for K.

- Some compilers support:

- ◆ Syntax: **READ(dev_num, *) variable_list**
- ◆ Behaves just like above.

WRITE Statement

- Format controlled WRITE

- ◆ Syntax: **WRITE(dev_no, format_label) variable_list**
- ◆ Write variables in *variable_list* to output *dev_no* using format specified in format statement with *format_label*
- ◆ Ex:

```
WRITE (6,1000) A,B,KEY
1000 FORMAT (F12.4,E14.5,I6)
```

Output:

```
|-----+-----o-----+-----o-----+-----o-----+-----|
      1234.5678   -0.12345E+02      12
```

- ◆ Device number 6 is commonly the printer but can also be the screen (standard screen is 2)
- ◆ Each **WRITE** produces one or more output lines as needed to write out *variable_list* using format statement.
- ◆ Variable_list can include implied DO such as:

```
WRITE (6,2000) (A(I), I=1,10)
```

WRITE Statement – cont' d

- List directed WRITE

- ◆ Syntax: **PRINT***, **variable_list**
- ◆ Write variables in `variable_list` to standard output device using format appropriate to variable type. Variables are separated by either spaces or commas, depending on system used.
- ◆ Ex: `PRINT* , 'X= ' , X , ' Y= ' , Y , ' N= ' , N`

Output:

```
x= 4.56, y= 15.62, N= 4
```


Error Control

- It is possible to handle error conditions, such as encountering an end-of-file, during **READ** statements.
- Extended **READ** statement
 - ◆ Syntax: **READ(dev_num, format_label, END=label) list** or **READ(*,*,END=label) list**
 - ◆ If an EOF is encountered by **READ**, transfer control to the statement label specified by **END=**.
 - ◆ Ex 1:

```
READ (5, 500, END=300) X, Y, Z
```
 - ◆ Ex 2:

```
READ (*, *, END=300) X, Y, Z
```
- Can also specify, **ERR=label**, to transfer control to label in the event of a **READ** error of some kind.

FORMAT Statement

- Very powerful and versatile but can be quite tedious to master and may vary between dialects
- Designed for use with line printers (not screens)
- Only infrequently used for input unless data format is clearly defined and consistently applied
- General:
 - ◆ Syntax: **label_no FORMAT(format-specifiers)**
 - ◆ Specifies format to be used in READ or WRITE statement that references this label_no.
 - ◆ format_specifiers are quite extensive and complex to master.
 - ◆ each format specifier is separated by a comma.

Format Specifiers

- X format code
 - ◆ Syntax: **nX**
 - ◆ Specifies n spaces to be included at this point
- I format code
 - ◆ Syntax: **lw**
 - ◆ Specifies format for an integer using a field width of w spaces. If integer value exceeds this space, output will consist of ****
- F format code
 - ◆ Syntax: **Fw.d**
 - ◆ Specifies format for a REAL number using a field width of w spaces and printing d digits to the right of the decimal point.
- A format code
 - ◆ Syntax: **A** or **Aw**
 - ◆ Specifies format for a CHARACTER using a field width equal to the number of characters, or using exactly w spaces (padded with blanks to the right if characters are less than w).

Format Specifiers – cont' d

- T format code
 - ◆ Syntax: **Tn**
 - ◆ Skip (tab) to column number n
- Literal format code
 - ◆ Syntax: **'quoted_string'**
 - ◆ Print the quoted string in the output (not used in input)
- L format code
 - ◆ Syntax: **Lw**
 - ◆ Print value of logical variable as L or F, right-justified in field of width, w.

Format Specifiers – cont' d

- E format code

- ◆ Syntax: Ew.d
- ◆ Print value of REAL variable using “scientific notation” with a mantissa of d digits and a total field width of w.

- ◆ Ex:

E14.5 produces for the REAL value -1.23456789e+4:

```
|-----+-----o-----+-----o-----+-----o-----+-----|  
-0.12345E+05
```

- ◆ You must leave room for sign, leading 0, decimal point, E, sign, and 2 digits for exponent (typically at least 7 spaces)
- ◆ If specified width is too small, mantissa precision, d, will be reduced unless $d < 1$ in which case ******* will be output.
- ◆ Using nP prefix will shift mantissa digit right by n and reduce exponent by -n. Ex; **1PE14.5** above yields:

```
|-----+-----o-----+-----o-----+-----o-----+-----|  
-1.23456E+04
```

Format Specifiers – cont' d

- G format code

- ◆ Syntax: Gw.d
- ◆ Print value of REAL variable using Fw.d format unless value is too large or too small, in which case use Ew.d format.
- ◆ Ex:

G14.5 produces for the REAL value $-1.23456789e+4$:

```
|-----+-----o-----+-----+-----o-----+-----|  
-12345.67890
```

- ◆ When the number gets too big (or too small) for F, it is switched to an E format. Ex: the value $-1.23456789e-18$ becomes:

```
|-----+-----o-----+-----+-----o-----+-----|  
-0.1234567E-19
```

- ◆ Note: the usefulness is more apparent when smaller field widths (w values) are specified for more compact output.

Other FORMAT Features

- Forward slash, /
 - ◆ Used to cause a new line to be started
 - ◆ Does not need to be separated by commas
- Repeat factor
 - ◆ Format specifiers may be repeated by prepending a number to specify the repeat factor
 - ◆ Ex: **4F12.5** – same as **F12.5, F12.5, F12.5, F12.5**
- Carriage control
 - ◆ Line printers interpret the first character of each line as a carriage control command and it is not printed.
 - 1 means start new page,
 - _(blank) means begin a new line,
 - + means over print current line
 - ◆ Common use: **1000 FORMAT(1X, 4F12.4)**

Other FORMAT Features – cont' d

- When the end of the `format_specifiers` in a `FORMAT` statement are reached before all of the variables in the `variable_list` have been output, the `format_specifiers` are re-scanned starting at the first left parenthesis, (`.`
- Many other format specifiers are available but are not included in these notes. These include formats for Binary, Octal and Hexidecimal data, formats for Double Precision numbers (replace `E` with `D`), and formats for Complex numbers.
- When formatted `READ` is used, any decimal point in data will override format specifier. If no decimal is supplied, format specifier will determine where decimal should be (even though it is not in input data)

```
      |-----+-----o-----+-----o-----+-----o-----+-----|
Data: 123456 1.23456

      READ(5,1000) A,B
1000  FORMAT(2F8.2)

Result: A=1234.56, B=1.23456
```


NAMELIST

- It is possible to pre-define the structure of input and output data using **NAMELIST** in order to make it easier to process with **READ** and **WRITE** statements.
 - ◆ Use **NAMELIST** to define the data structure
 - ◆ Use **READ** or **WRITE** with reference to **NAMELIST** to handle the data in the specified format
- This is not part of standard **FORTRAN 77**... but it is included in **FORTRAN 90**.
- It is being included in these notes because it has been widely used in **ASDL** for a number of years.

NAMELIST Statement

- Used to define the structure of the I/O data
 - ◆ Syntax: **NAMELIST /group/ var_list [, [/group/var_list]]**
 - ◆ Associates a group name with a comma separated list of variables for I/O purposes. Variables can appear in more than one group.
 - ◆ Ex:

```
NAMELIST /INPUT/LENGTH,WIDTH,THICK,DENSITY,  
*/OUTPUT/AREA,DENSITY,WEIGHT
```

This defines INPUT to include 4 variables and OUTPUT to include 3 variables. One (density) is repeated.
- The **READ** or **WRITE** statement simply refers to the **NAMELIST** statement rather than a list of variables.
 - ◆ Syntax: **READ(dev_no,NML=group)**
WRITE(dev_no,NML=group)
 - ◆ Ex:

```
READ (5 ,NML=INPUT)
```

NAMELIST Data Structure

- On input, the **NAMELIST** data for the previous slide must be structured as follows:

```
&INPUT  
  THICK=0.245,  
  LENGTH=12.34,  
  WIDTH=2.34,  
  DENSITY=0.0034  
/
```

- And on executing the **READ (5, NML=INPUT)**, the following values are assigned:
 - ◆ **THICK=0.245, LENGTH=12.34, WIDTH=2.34, DENSITY=0.0034**
 - ◆ It is not necessary to provide values for all variables in a **NAMELIST** group; values not provided result in no changes.
 - ◆ For arrays, assignment can be partial and can start at any index and can skip values by including ,, in input.

Input NAMELIST Examples

- Parts or all of the data can be assigned
- Multiple READ's can be used with successive NAMELIST data

```
NAMELIST /TEST/TITLE,FLAG,A
DIMENSION A(10)
LOGICAL FLAG
CHARACTER*10 TITLE
...
READ(5,NML=TEST)
...
READ(5,NML=TEST)
```

```
&TEST
  TITLE='TEST567890',
  FLAG=.TRUE.,
  A=1.2,3.3,8*0.0
/
Results in:
TITLE='TEST567890'
FLAG=.TRUE.
A=1.2,3.3,rest=0
```

```
&TEST
  TITLE(9:10)='77',
  A(5)=5*10.0
/
Results in:
TITLE='TEST567877'
FLAG=unchanged
A(5)..A(10)=10.0
```

Output NAMELIST Examples

- Output behavior is similar to input:

```
CHARACTER*8 NAME(2)
REAL PITCH,ROLL,YAW,POSITION(3)
INTEGER ITER
LOGICAL DIAG
NAMELIST /PARAM/NAME,PITCH,ROLL,YAW,POSITION,DIAG,ITER
DATA NAME/2*' '/,POSITION/3*0.0/
...
READ(5,NML=TEST)
...
WRITE(6,NML=TEST)
```

```
&PARAM
NAME(2)(4:8)='FIVE',
PITCH=5.0,YAW=0.0,ROLL=-5.0,
DIAG=.TRUE.,ITER=10
/
```

```
&PARAM
NAME= ' ', 'FIVE',
PITCH= 5.0,
ROLL = -5.0,
YAW = 0.0,
POSITION= 3*0.00000e+00,
DIAG = T,
ITER = 10
/
```

Functions and Subroutines

- **Functions & Subroutines** (*procedures* in other languages) are subprograms that allow modular coding
 - ◆ **Function**: returns a single explicit function value for given function arguments
 - ◆ **Subroutine**: any values returned must be returned through the arguments (no explicit subroutine value is returned)
 - ◆ Functions and Subroutines are not recursive in FORTRAN 77
- In FORTRAN, subprograms use a separate namespace for each subprogram so that variables are local to the subprogram.
 - ◆ variables are passed to subprogram through argument list and returned in function value or through arguments
 - ◆ Variables stored in **COMMON** may be shared between namespaces (e.g., between calling program and subprogram)

FUNCTION Statement

- Defines start of Function subprogram
 - ◆ Serves as a prototype for function call (defines structure)
 - ◆ Subprogram must include at least one **RETURN** (can have more) and be terminated by an **END** statement
- FUNCTION structure:
 - ◆ Syntax: **[type] FUNCTION fname(p₁,p₂, ... p_N)**
 - ◆ Defines function name, *fname*, and argument list, p_1, p_2, \dots, p_N , and optionally, the function type if not defined implicitly.
 - ◆ Ex:

```
REAL FUNCTION AVG3 (A, B, C)
AVG3= (A+B+C) /3
RETURN
END
```

```
Use :
AV=WEIGHT*AVG3 (A1 , F2 , B2)
```

- ◆ Note: function type is implicitly defined as REAL

Statement Function

- FORTRAN provides a “shortcut” method to define simple, single expression functions without having to create a separate subprogram...
- Statement Function:
 - ◆ Syntax: **function_name(p₁,p₂,...p_N) = expression**
 - ◆ This definition can be continued to additional lines but must be a single statement (no IF' s, DO' s, etc) and it must appear before any other executable code but after all type declarations.

◆ Ex:

```
PROGRAM MAIN
REAL A,B,C
FUNC (X) =A*X**2-B*X+C
...program...
ANS=FUNC (4.2) +1.2
...
END
```

- ◆ Note: argument is treated as a dummy variable and may be replaced by other variables or literals when used in program; other variables in function are in program scope.

SUBROUTINE Statement

- Defines start of Subroutine subprogram
 - ◆ Serves as a prototype for subroutine call (defines structure)
 - ◆ Subprogram must include at least one **RETURN** (can have more) and be terminated by an **END** statement
- SUBROUTINE structure:
 - ◆ Syntax: **SUBROUTINE sname(p₁,p₂, ... p_N)**
 - ◆ Defines subroutine name, *sname*, and argument list, $p_1, p_2, \dots p_N$.
 - ◆ Ex:

```
SUBROUTINE AVG3S (A, B, C, AVERAGE)
AVERAGE= (A+B+C) /3
RETURN
END
```

```
Use :
CALL AVG3S (A1, F2, B2, AVR)
RESULT=WEIGHT*AVR
```

- ◆ Subroutine is invoked using the **CALL** statement.
- ◆ Note: any returned values must be returned through argument list.

Placement of Subprograms

- Subprograms are placed immediately following main program **END** statement.

```
PROGRAM MAIN
...program body...
END

REAL FUNCTION AVG3(A,B,C)
...function body...
END

SUBROUTINE AVG3S(A,B,C,AV)
...subroutine body...
END
```

- Subprograms can be written and compiled separately but must then be made available to link-loader in order to be linked into executable program. In not, an “undefined externals” error will be generated.

Arguments

- Arguments in subprogram are “dummy” arguments used in place of the real arguments used in each particular subprogram invocation. They are used in subprogram to define the computations.
- Actual subprogram arguments are passed by reference (address) if given as symbolic; they are passed by value if given as literal.
 - ◆ If passed by reference, the subprogram can then alter the actual argument value since it can access it by reference (address).
 - ◆ Arguments passed by value cannot be modified.

```
CALL AVG3S (A1 , 3 . 4 , C1 , QAV)
```

OK: 2nd argument is passed by value; QAV contains result.

```
CALL AVG3S (A , C , B , 4 . 1)
```

NO: no return value is available since 4.1 is a value and not a reference to a variable!

Arguments – cont' d

- Dummy arguments appearing in a Subprogram declaration cannot be an individual array element reference, e.g., A(2), or a literal, for obvious reasons!
- Arguments used in invocation (by calling program) may be *variables, subscripted variables, array names, literals, expressions, or function names.*
- Using symbolic arguments (variables or array names) is the only way to return a value (result) from a **SUBROUTINE**.
- It is considered BAD coding practice, but **FUNCTIONs** can return values by changing the value of arguments. This type of use should be strictly avoided!

FUNCTION versus Array

- How does FORTRAN distinguish between a **FUNCTION** and an array having the same name?
 - ◆ **REMAINDER (4 , 3)** could be a 2D array or it could be a reference to a function that returns the remainder of 4/3
 - ◆ If the name, including arguments, matches an array declaration, then it is taken to be an array.
 - ◆ Otherwise, it is assumed to be a **FUNCTION**
- Be careful about implicit versus explicit Type declarations with **FUNCTIONS**...

```
PROGRAM MAIN
INTEGER REMAINDER
...
KR=REMAINDER (4 , 3)
...
END

INTEGER FUNCTION REMAINDER (INUM , IDEN)
...
END
```

Arrays with Subprograms

- Arrays present special problems in subprograms...
 - ◆ Must pass by reference to subprogram since there is no way to list array values explicitly as literals.
 - ◆ How do you tell subprogram how large the array is? (Answer varies with FORTRAN version and vendor (dialect)...)
- When an array element, e.g., $A(1)$, is used in a subprogram invocation (in calling program), it is passed as a reference (address), just like a simple variable.
- When an array is used by name in a subprogram invocation (in calling program), it is passed as a reference to the entire array. In this case the array must be appropriately dimensioned in the subroutine (and this can be tricky...).

Arrays with Subprograms – cont' d

- Explicit array declaration in Subprogram
 - ◆ If you know the dimension and it does not change for any invocation of the subprogram, then declare it explicitly:

```
REAL FUNCTION AAVG (ARRAY)
DIMENSION ARRAY (10)
SUM=0.0
DO 100 I=1,10
    SUM=SUM+ARRAY (I)
100 CONTINUE
AAVG=SUM/10
RETURN
END
```

- ◆ Beware: calling this function with a scalar will cause problems! (solution: always test argument type if possible)
- ◆ Note: this is really a badly designed function because it assumes that the array dimension is 10. A better design would pass the array dimension as an argument.

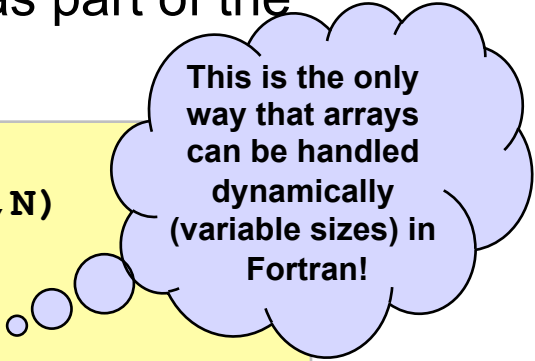
Arrays with Subprograms – cont' d

- Variable array dimensioning in Subprogram

- ◆ If the dimensions of arrays passed to the subprogram can vary between calls, the dimension can be passed as part of the argument list.

- ◆ Ex:

```
SUBROUTINE AADD (A, B, SUM, M, N)
DIMENSION A (M, N) , B (M, N) , SUM (M, N)
DO 200 I=1, M
    DO 100 J=1, N
        SUM (I, J) =A (I, J) +B (I, J)
100    CONTINUE
200    CONTINUE
RETURN
END
```



This is the only way that arrays can be handled dynamically (variable sizes) in Fortran!

- ◆ Different **FORTRAN 77** dialects offer variations. **FORTRAN 90/95** defines above example as an “*explicit-shape, adjustable array*” and also define “*assumed-shape*”, “*assumed-size*” and “*deferred-shape*” arrays! You will need to check documentation for your particular dialect...

COMMON Statement

- The **COMMON** statement allows variables to have a more extensive scope than otherwise.
 - ◆ A variable declared in a Main Program can be made accessible to subprograms (without appearing in argument lists of a calling statement)
 - ◆ This can be selective (don't have to share all everywhere)
 - ◆ Placement: among type declarations, after **IMPLICIT** or **EXPLICIT**, before **DATA** statements
 - ◆ Can group into labeled **COMMONs**
- Must use the **BLOCK DATA** subprogram to initialize variables declared in a **COMMON** statement

COMMON Statement – cont' d

- The **COMMON** statement (also called **blank COMMON**)
 - ◆ Syntax: **COMMON variable_list**
 - ◆ Declares that the variables in the variable_list are stored in a special area where they can be shared with other subprograms. Each subprogram must include a **COMMON** statement in order to access these shared (common) variables. The variable_list must agree strictly in type of variable but different names can be used in each subprogram (can be VERY confusing).

◆ Ex:

```
PROGRAM MAIN
COMMON D (3) , KEY (4 , 4)
D (1) = 2 . 2
D (2) = -1 . 3
D (3) = 5 . 6
RESULT = FUNC (-4 . 3)
END
```

COMMON is declared larger in main program

```
REAL FUNCTION FUNC (X)
COMMON C (3)
FUNC = C (1) * X ** 2 + C (2) * X + C (3)
RETURN
END
```

Use different name in function and don't declare all of **COMMON**

COMMON Statement – cont' d

- Can declare array dimensions in COMMON or not...

- ◆ These are all acceptable:

```
COMMON X(100)
```

```
REAL X  
COMMON X(100)
```

```
REAL X(100)  
COMMON X
```

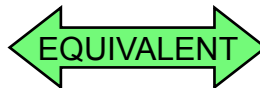
- ◆ But this is not:



```
REAL X(100)  
COMMON X(100)
```

- Can combine:

```
COMMON A,B  
COMMON C(10,3)  
COMMON KPY
```



```
COMMON A,B,C(10,3),KPY
```

- Cannot initialize with DATA statement:



```
COMMON X(100)  
DATA X/100*1.0/
```

Labeled COMMON Statement

- When the defined **COMMON** block is large, a single subprogram may not need to refer to all variables.
- Solution: **labeled COMMON**:
 - ◆ Syntax: **COMMON [/block_name1/] var_list /block_name2/ var_list /block_name3/var_list/ ...**
 - ◆ Defines one or more labeled **COMMON** blocks containing specified lists of variables. If first block name is omitted, this defines “blank” common. For some dialects, **COMMON** blocks must have same length in all subprograms and if character arrays appear in a block, no other types can appear in that block.

◆ Ex:

```
PROGRAM MAIN
COMMON D (3) /PARAMS /A, B, C (4)
COMMON /STATE /X (100), Y (100), Z (100)
...code...
END
```

D is in **blank common**

Multiple **COMMON**'s are treated as a single long statement, and variables are defined in order

BLOCK DATA Subprogram

- **BLOCK DATA** is a subprogram that simply assigns values to arrays and variables in a **COMMON** block.
 - ◆ Syntax: **BLOCK DATA name**
specifications
END [BLOCK DATA [name]]
 - ◆ This is placed with subprograms and is used to initialize variables in **COMMON** and labeled **COMMON**.
 - ◆ Ex:

```
PROGRAM MAIN
COMMON D(3)/PARAMS/A,B,C(4)
COMMON /STATE/X(100),Y(100),Z(100)
...code...
END

BLOCK DATA
DIMENSION X(100),Y(100),Z(100)
COMMON D(3)/PARAMS/A,B,C(4)/STATE/X,Y,Z
DATA X,Y,X/300*0.0/,C/4*1.0/,D/3*0.0/,A/22/
END
```

Slightly different
declaration than above

A More Complicated Example

- Finite element structural analysis programs:
 - ◆ must assemble an N by N global stiffness matrix, K, from individual element stiffness matrices, where N is the total number of unconstrained DOF' s.
 - ◆ must also generate a force vector, LOAD, with N components for each loading case.
 - ◆ must construct a solution for the displacements at each DOF that is defined by: $DISP = K^{-1} * LOAD$.
- Program considerations:
 - ◆ global stiffness is defined in labeled COMMON
 - ◆ load vector is defined in labeled COMMON
 - ◆ subroutine to compute stiffness inverse must access COMMON
 - ◆ matrices must be defined for largest problem since FTN77 does not support dynamic memory allocation

A More Complicated Example – cont' d

● Skeleton of a program...

```
PROGRAM MAIN
REAL KGLO, FORC, KEL
COMMON /STIF/KGLO(100,100)/LOAD/FORC(100)/DEF/D(100)
C ...read in data and initialize problem...
DO 100 IELEM=1, NELEMS
C ...assemble global stiffness matrix...
  CALL KELEM(IELEM, KEL)
  CALL ASMBK(IELEM, KEL)
100 CONTINUE
DO 200 ILOAD=1, NLOADS
C ...assemble load vector...
  CALL LODVEC(ILOAD, LOAD)
200 CONTINUE
  CALL CONSTR(KDOFS)
  CALL SOLVE(NDOFS)
C ...print out results etc. ...
END
```

partial list of declarations

Calculate stiffness matrix, KEL, for a single element

Add KEL to global stiffness matrix, KGLO

Construct FORC from individual loads defined in LOAD array

Must constrain problem at specified DOF's (or no solution possible)

Compute solution for displacements

A More Complicated Example – cont' d

- Example code for SOLVE

```
SUBROUTINE SOLVE(NDOFS)
DIMENSION CGLO(100,100)
COMMON /STIF/KGLO(100,100)/LOAD/FORC(100)/DEF/D(100)
CALL MATINV(KGLO,CGLO,NDOFS)
CALL MMULT(CGLO,FORC,D,NDOFS)
RETURN
END
```

```
      SUBROUTINE MATINV(A,B,N)
      DIMENSION A(N,N),B(N,N)
      C ...compute inverse of A and return as B...
      RETURN
      END
```

```
      SUBROUTINE MMULT(A,B,C,N)
      DIMENSION A(N,N),B(N),C(N)
      C ...compute A*B=C ...
      RETURN
      END
```

- Avoids having to include all arrays in all calls to subroutines that process some or all of data.

Additional Subprogram Details

- Multiple entries into a subprogram
 - ◆ Syntax: **ENTRY name(p1,p2,...pN)**
 - ◆ Provides an alternate entry point into a subprogram with an alternate set of arguments. When included in a FUNCTION, this name will return a value.
 - ◆ Ex:

```
PROGRAM MAIN
REAL X(2,3),RS(2),CS(3)
CALL MAT(2,3)
...define X values...
CALL RMAT(X,2,3,RS,CS)
...code...
CALL CMAT(X,2,3,CS)
...code...
```

```
SUBROUTINE MAT(A,M,N)
REAL A(M,N),RSUM(M),CSUM(N)
DO 10 I=1,M
  DO 5 J=1,N
    5 A(I,J)=0.0
  10 CONTINUE
  RETURN
  ENTRY RMAT(A,M,N,RSUM,CSUM)
  DO 14 I=1,M
    RSUM(I)=0.0
    DO 12 J=1,N
      12 RSUM(I)=RSUM(I)+A(I,J)
    14 CONTINUE
  ENTRY CMAT(A,M,N,CSUM)
  DO 18 J=1,N
    CSUM(J)=0.0
    DO 16 I=1,M
      16 CSUM(J)=CSUM(I)+A(I,J)
    18 CONTINUE
  RETURN
END
```

Return from MAT()

Return from RMAT() and CMAT()

NOTE:

Lack of RETURN before CMAT means call to RMAT will calc both sums.

Additional Subprogram Details – cont' d

- Multiple **RETURNS** to calling program from subprogram
 - ◆ Syntax: **CALL subname(p₁,p₂,&s₁,p₃,&s₂,...)** where **SUBROUTINE subname(a₁*,a₂*,...)**
 - ◆ Allows a subroutine to return to multiple locations in calling program where entry points are specified by labels s₁... Note that subroutine definition includes * for return points in argument list.
 - ◆ Example: it is somewhat difficult to come up with meaningful examples of what is basically poor program design today... One possible example is to return to a different location if an error condition is encountered.

```
PROGRAM MAIN
...define X values...
CALL ROOT(GUESS,VALUE,&10)
...code...
STOP
10 PRINT*, 'NO ROOT FOUND NEAR GUESS.'
...code...
END
```

```
SUBROUTINE ROOT(GUESS,VALUE,*)
...code...
IF (ITER.GT.MAX) RETURN 1
RETURN
END
```

Passing Function as Arguments

- Often it is useful to be able to pass the name of a function to a subroutine. For example:
 - ◆ a subroutine, **NEWTON** () that computes a root of $f(x)$ using Newton's Method will need to be able to evaluate the function, $f(x)$, and its derivative, $df(x)/dx$, for arbitrary values of x .
 - ◆ How can we tell the subroutine how to compute these two functions?
- We could simply pass the name of a **FUNCTION** subprogram as an argument to the subroutine
 - ◆ How can this be distinguished from another variable name?
 - ◆ We need to “tag” the function name somehow
- Solution: the **EXTERNAL** or **INTRINSIC** statements

Passing Function as Arguments – cont' d

● EXTERNAL statement

- ◆ Syntax: **EXTERNAL list_of_names**
- ◆ Define names in *list_of_names* of user-written subprograms that are to be passed as arguments to a function or subroutine
- ◆ Ex:

```
PROGRAM MAIN
EXTERNAL FUNC,DFUNC
...code...
CALL NEWTON(GUESS,ROOT,FUNC,DFUNC,&30)
PRINT*,'ROOT IS:',ROOT
...
30 PRINT*,'NO ROOT FOUND.')
END

REAL FUNCTION FUNC(X)
FUNC=X**5-5.3*SIN(3.2*X)
RETURN
END

REAL FUNCTION DFUNC(X)
DFUNC=5*X**4-5.3*3.2*COS(3.2*X)
RETURN
END
```

```
SUBROUTINE NEWTON(G,R,F,DF,*)
C ...Newton's Method...
R=G-F(G)/DF(G)
RETURN
C ...didn't converge...
RETURN 1
END
```

Passing Function as Arguments – cont' d

● INTRINSIC statement

- ◆ Syntax: **INTRINSIC list_of_names**
- ◆ Define names in *list_of_names* of built-in functions that are to be passed as arguments to another function or subroutine
- ◆ Ex:

```
PROGRAM MAIN
INTRINSIC SIN
EXTERNAL DFSIN
...code...
CALL NEWTON(GUESS,ROOT,SIN,DFSIN,&30)
PRINT*,'ROOT IS:',ROOT
...
30 PRINT*,'NO ROOT FOUND.')
END

REAL FUNCTION DFSIN(X)
DFUNC=COS(X)
RETURN
END
```

```
SUBROUTINE NEWTON(G,R,F,DF,*)
C ...Newton's Method...
R=G-F(G)/DF(G)
RETURN
C ...didn't converge...
RETURN 1
END
```

Comments on Subprograms

- The present presentation is essentially **FORTRAN 77** with a few minor extensions for popular dialects
- **FORTRAN 90** considerably extends the language and introduces a number of significant changes
 - ◆ Major changes are in area of new type declarations and in how code can be modularized
 - ◆ many changes are to make **FORTRAN** more “contemporary”
 - ◆ You will need to consult a **FORTRAN 90** reference manual for more details.
- However, the most readily available compilers (e.g., gnu g77) support only a limited set of extensions to **FORTRAN 77**.

File-Directed Input and Output

- Much of early **FORTRAN** was devoted to reading input data from Cards and writing to a line printer, and what we have seen so far is quite adequate.
- Today, most I/O is to and from a file.
 - ◆ Requires more extensive I/O capabilities.
 - ◆ This was not standardized until **FORTRAN 77** but each manufacturer often created a specific “dialect.”
 - ◆ It is included in **FORTRAN 90** which we will discuss.
- Important concepts:
 - ◆ **OPEN**, **CLOSE** and position commands manipulate a file,
 - ◆ Once opened, file is referred to by an assigned device number,
 - ◆ Files can have variable length records (sequential access), or they can be fixed length (direct access) which is faster,
 - ◆ Can use unformatted **READ** & **WRITE** if no human readable data are involved (much faster access, smaller files).

Sequential versus Direct Access

- When each record can be a different length, individual records cannot easily be accessed randomly:
 - ◆ it is necessary to read sequentially through the file,
 - ◆ the file can be rewound to beginning or backspaced to previous record,
 - ◆ generally a slow process.
- If each record is a fixed length, it is possible to easily position to individual records because the offset from the start can quickly be computed:
 - ◆ can use a seek operation to go to a specified record,
 - ◆ provides the fastest access.
- Requires special care to handle EOF on input or output.

OPEN Statement

- **OPEN** is used to make file available to **READ** & **WRITE**
 - ◆ Syntax: **OPEN ([UNIT=]io_unit [,FILE=name] [,ERR=label] [,IOSTAT=i_var], slist)**
 - ◆ Named **FILE** will be opened and associated with given **UNIT**, transfer to **ERR** label if error, also **IOSTAT=0** if no error or positive error number if error; **slist** is list of **specifier= 'value'** pairs as defined by **OPEN** command.
 - ◆ Ex:

```
OPEN (12, FILE='D:\AE\test.dat', ERR=1000, IOSTAT=IER)
```

Opens file D:\AE\test.dat for sequential read&write (default) and specifies device number 12 for access.
 - ◆ Ex:

```
OPEN (14, FILE='D:\test1.dat', ERR=1000, IOSTAT=IER,  
*ACCESS='SEQUENTIAL', ACTION='WRITE' )
```

Opens file D:\test1.dat for sequential, write-only mode to device 14.
 - ◆ Default format is formatted. To use for unformatted READ or WRITE, include: **FORM= 'UNFORMATTED'**

Writing an Output File

- Commands have detailed parameters defined in reference manuals:
 - ◆ **OPEN** file for output (typically using sequential access)
 - ◆ **WRITE** each record (presence of “/” in **FORMAT** will cause another record to be started).
 - ◆ **CLOSE** file (automatically leaves EOF at end of data)
 - ◆ Need to check for error conditions on each operation.
 - ◆ When writing direct access file, it is necessary to specify the record to be written.

Reading an Input File

- Commands have detailed parameters defined in reference manuals:
 - ◆ **OPEN** file for input (typically using sequential access),
 - ◆ **READ** each record (use formatted or list-directed **READ**),
 - ◆ Can position using **BACKSPACE** or **REWIND** if needed,
 - ◆ **CLOSE** file (**OPEN** with same `io_unit` closes previous),
 - ◆ Need to check for error conditions on each operation.
 - ◆ When reading direct access file, it is necessary to specify the record to be read.

Positioning and Closing

- To BACKSPACE a sequential access file:
 - ◆ Syntax: `BACKSPACE io_unit` or
`BACKSPACE ([unit=]io_unit [,ERR=label] [,IOSTAT=i_var])`
 - ◆ Ex: `BACKSPACE 14`
- To REWIND a file (position at start):
 - ◆ Syntax: `REWIND io_unit` or
`REWIND ([unit=]io_unit [,ERR=label] [,IOSTAT=i_var])`
 - ◆ Ex: `REWIND (12, ERR=2000)`
- To CLOSE a file:
 - ◆ Syntax: `CLOSE ([unit=]io_unit [,STATUS=p] [,ERR=label]`
`* [,IOSTAT=i_var])`
 - ◆ Ex: `CLOSE (14, STATUS='DELETE')` (deletes file, default=save)

Unformatted vs Formatted I/O

● Unformatted **READ** or **WRITE**

- ◆ Syntax: **READ**(dev_no[,IOSTAT=i_var][,ERR=label]) var_list
WRITE(dev_no[,IOSTAT=i_var][,ERR=label]) var_list
- ◆ Simply leaving out the **FORMAT** label creates a form in which the internal binary data is read or written (fastest I/O).
- ◆ Control arguments are optional.
- ◆ Ex: **READ (12) A, B, C** or **WRITE (14) DATA**

● Formatted **READ** or **WRITE**

- ◆ Syntax: **READ**(dev_no,format_label [,IOSTAT=i_var] [,ERR=label]) variable_list
WRITE(dev_no,format_label [,IOSTAT=i_var] [,ERR=label]) variable_list
- ◆ Can use list-directed **READ** by using * in place of format_label.
- ◆ Ex: **WRITE (12, 1000, ERR=2200) DATA**
READ (10, *, ERR=2100) RAWDAT

Direct Access I/O

- Direct Access **READ** or **WRITE**
 - ◆ Must specify the actual record number to seek and read.
 - ◆ Syntax: **READ**(dev_no,format_label,REC=rec [,IOSTAT=i_var] [,ERR=label]) variable_list
WRITE(dev_no,format_label,REC=rec [,IOSTAT=i_var] [,ERR=label]) variable_list
 - ◆ It is only necessary to add the REC=rec specifier to set the record number to be read or written. User must compute positions.
 - ◆ Ex: **READ (12 , * , REC=KR , ERR=2200) DATA**

Some Other Interesting Stmts

● EQUIVALENCE statement

- ◆ Syntax: **EQUIVALENCE (list_of_variables) [,...]**
- ◆ Used to make two or more variables share the same storage in memory. This used to be an important way to conserve memory without having to use the same variable names everywhere. It can also be used to access an array element using a scalar variable name (or to represent a subarray with another name).

◆ Ex:

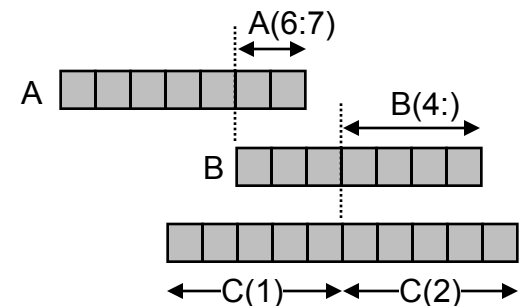
```
PROGRAM MAIN
DIMENSION A(5), B(5), C(10,10), D(10)
EQUIVALENCE (A(1), B(1)), (A(5), ALAST)
EQUIVALENCE (C(1,1), D(1))
...
```

A and B are same

A(5) can be referred to as ALAST

D refers to first column of C (because arrays are stored columnwise)

```
PROGRAM MAIN
CHARACTER A*7, B*7, C(2)*5
EQUIVALENCE (A(6:7), B), (B(4:), C(2))
...
```



Reading & Writing to/from Internal Storage

- Older code may include statements that transfer data between variables or arrays and internal (main memory) storage. This is a fast but temporary storage mechanism that was popular before the widespread appearance of disks.
- One method is to use the **ENCODE** & **DECODE** pairs
 - ◆ **DECODE** – translates data from character to internal form,
 - ◆ **ENCODE** – translates data from internal to character form.
- Another method that is in some **FORTRAN 77** dialects and is in **FORTRAN 90** is to use **Internal READ/WRITE** statements.

ENCODE/DECODE Statements

● DECODE statement

- ◆ Syntax: **DECODE (c,f,b [,IOSTAT=i_var] [,ERR=label]) [var_list]**
- ◆ Translate data from character into internal form, where:
 - **c** = scalar integer expr defining number of characters to be translated into internal form,
 - **f** = format label (error if more than one record is specified),
 - **b** = scalar or array to be decoded (if array, it is processed by columns)
 - **var_list** = variables used to store internal form
 - **IOSTAT, ERR** = as before for other file I/O statements

◆ Ex:

The diagram illustrates the DECODE statement and its results. It features a central yellow box containing the following code:

```
INTERNAL STORAGE    CHARACTERS
INTEGER*4 K(3)
CHARACTER*12 A,B
DATA A/'ABCDEFGHIJKL' /
...
DECODE(12,100,A) K
100 FORMAT(3I4)
```

Callouts point to specific parts of the code:

- "Internal storage" points to the `INTEGER*4 K(3)` declaration.
- "Characters" points to the `CHARACTER*12 A,B` declaration.
- "How to fit characters into INTEGER*4" points to the `DECODE(12,100,A) K` statement.

Results:

```
Results:
K(1)=ABCD
K(2)=EFGH
K(3)=IJKL
```

ENCODE/DECODE Statements – cont' d

● ENCODE statement

- ◆ Syntax: **ENCODE (c,f,b [,IOSTAT=i_var] [,ERR=label]) [var_list]**
- ◆ Translate data from internal (binary) form into character, where:
 - **c** = scalar integer expr defining number of characters to be translated into character form,
 - **f** = format label (error if more than one record is specified),
 - **b** = scalar or array where encoded characters are put (if array, it is processed by columns)
 - **var_list** = variables to be translated (encoded)
 - **IOSTAT, ERR** = as before for other file I/O statements

◆ Ex:

```
INTERNAL STORAGE          ENCODED CHARACTERS (RETURNED)
```

```
INTEGER*4 K(3)
CHARACTER*12 A,B
DATA A/'ABCDEFGHIJKL' /
...
DECODE(12,100,A) K
100 FORMAT(3I4)
ENCODE(12,100,B) K(3),K(2),K(1)
```

Values:
K(1)=ABCD
K(2)=EFGH
K(3)=IJKL

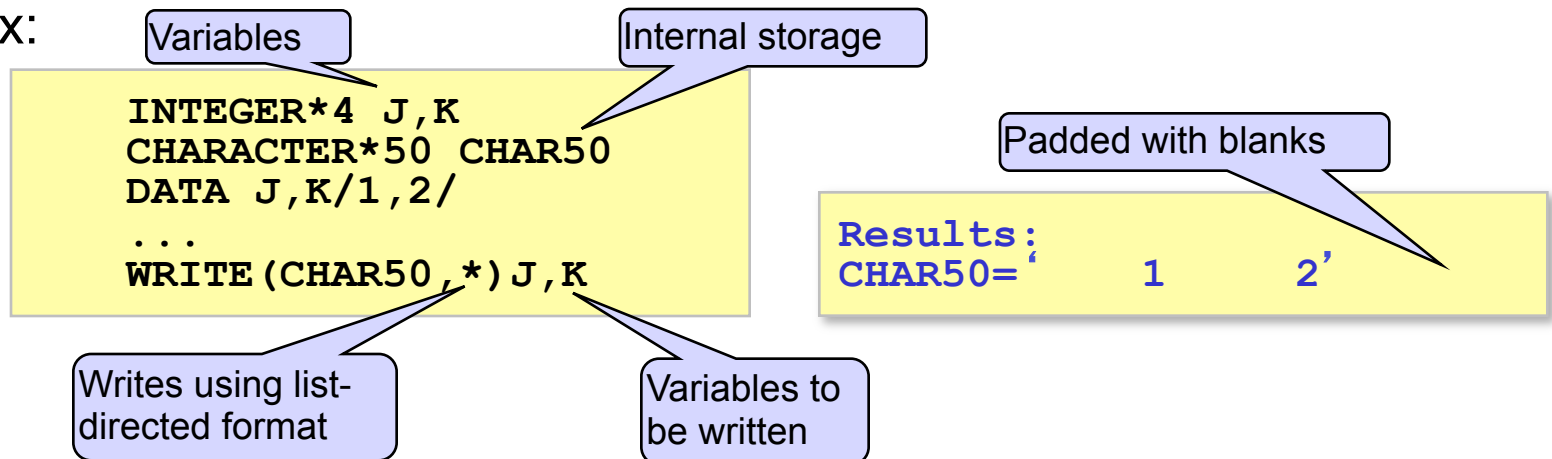
How to fit characters into INTEGER*4

Results:
B= 'IJKLEFGHABCD'

Internal WRITE Statement

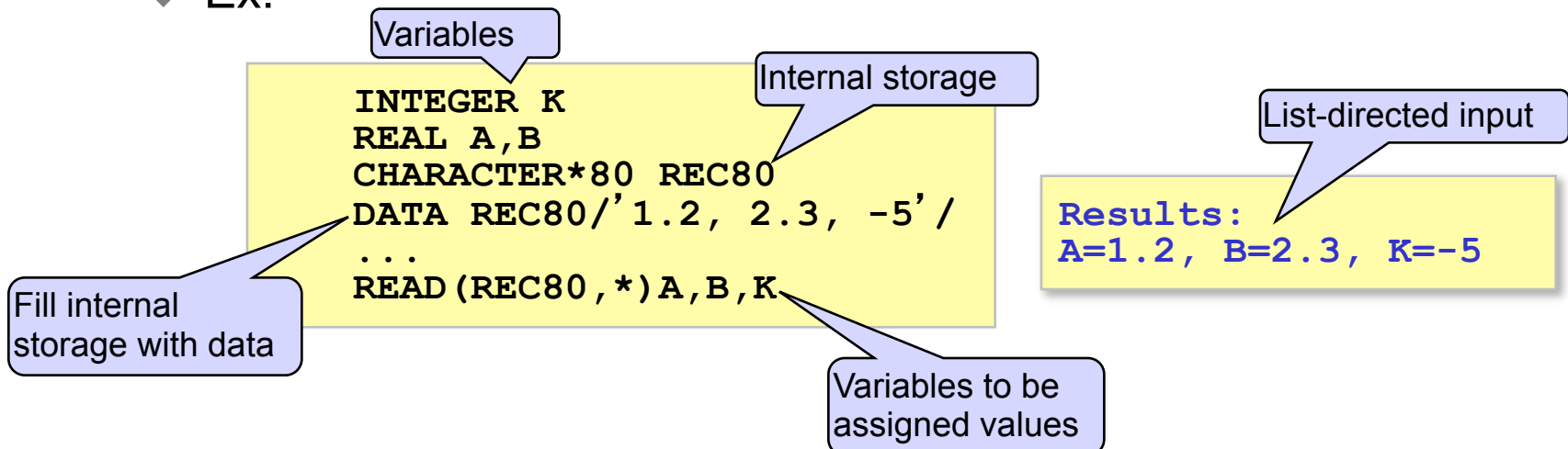
- Internal **WRITE** does same as **ENCODE**
 - ◆ Syntax: **WRITE (dev_no, format_label [,IOSTAT=i_var] [,ERR=label]) [var_list]**
 - ◆ Write variables in *var_list* to internal storage defined by character variable used as *dev_no* where:
 - **dev_no** = default character variable (not an array),
 - **format_label** = points to **FORMAT** statement or * for list-directed,
 - **var_list** = list of variables to be written to internal storage.

◆ Ex:



Internal READ Statement

- Internal **READ** does same as **DECODE**
 - ◆ Syntax: **READ (dev_no, format_label [,IOSTAT=i_var] [,ERR=label] [END=label]) [var_list]**
 - ◆ Read variables from internal storage specified by character variable used as *dev_no* and output to *var_list* where:
 - **dev_no** = default character variable (not an array),
 - **format_label** = points to **FORMAT** statement or * for list-directed,
 - **var_list** = list of variables to be written from internal storage.
 - ◆ Ex:



Conclusions

- FORTRAN in all its standard versions and vendor-specific dialects is a rich but confusing language.
- FORTRAN is still ideally suited for numerical computations in engineering and science
 - ◆ most new language features have been added in FORTRAN 95
 - ◆ “High Performance FORTRAN” includes capabilities designed for parallel processing.
- You have seen most of FORTRAN 77 but only a small part of FORTRAN 90/95.
 - ◆ Many new FORMAT and I/O statements and options
 - ◆ Several new control statements
 - ◆ New derived variable types (like structures)
 - ◆ Recursive functions
 - ◆ Pointers and dynamic variables
 - ◆ Modules