Introduction to structured programming with Fortran

CISM/CÉCI Training Sessions 25/10/2018

Pierre-Yves Barriat

Fortran : shall we start ?

- You know already one computer langage ?
- You understand the very basic concepts :
  - What is a variable, an assignment, function call, etc.?
  - Why do I have to compile my code?
  - What is an executable?
- You (may) already know some Fortran ?
- You are curious about what comes next ?
- How to proceed from old Fortran, to much more modern languages like Fortran 90/2003?

Fortran : why ?

Because of the execution **speed** of a program

Fortran is a simple langage and it is (kind-of) easy to learn

We want to get our science done! Not learn languages!

How easy/difficult is it really to learn Fortran ?

The concept is easy:

variables, operators, controls, loops, subroutines/functions

Invest some time now, gain big later!

History

FORmula TRANslation invented 1954–8

by John Backus and his team at IBM

- FORTRAN 66 (ISO Standard 1972)
- FORTRAN 77 (1980)
- Fortran 90 (1991)
- Fortran 95 (1996)
- Fortran 2003 (2004)
- Fortran 2008 (2010)
- Fortran 2015 (ongoing)

Starting with Fortran77

- Old Fortran (Fortran77) provides only the absolute minimum!
- Basic features : data containers (integer, float, ...), arrays, basic operators, loops, I/O, subroutines and functions
- But this language has flaws:

  Fortran77: no dynamic memory allocation, old & obsolete constructs, “spaghetti” code, etc.
- Is that enough to write code?

Fortran 77 – Fortran >90

If Fortran77 is so simple,
Why is it then so difficult to write good code?

Is simple really better?

Using a language allows us to express our thoughts (on a computer)

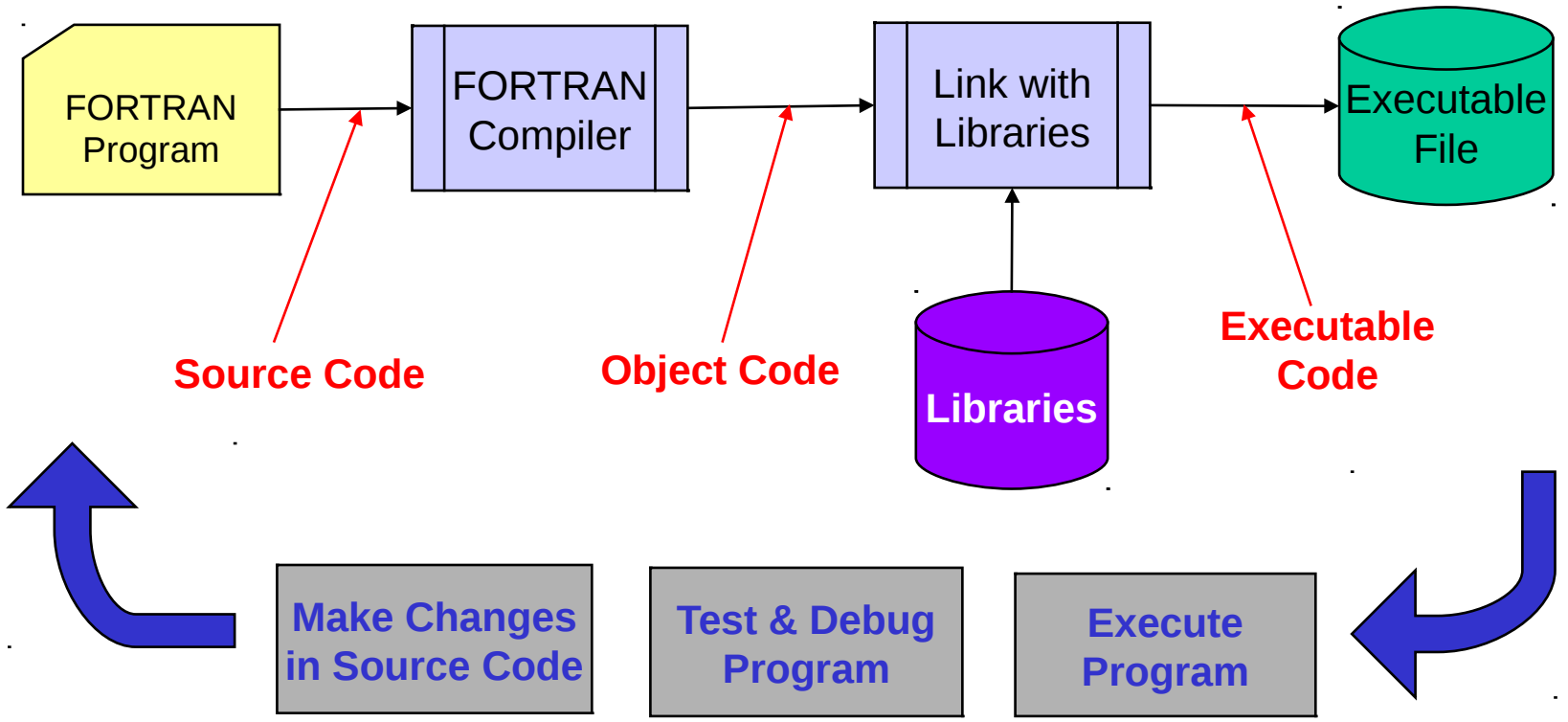
A more sophisticated language allows for more complex thoughts

More language elements to get organized

⇒ Fortran 90/95/2003 (recursive, OOP, etc)

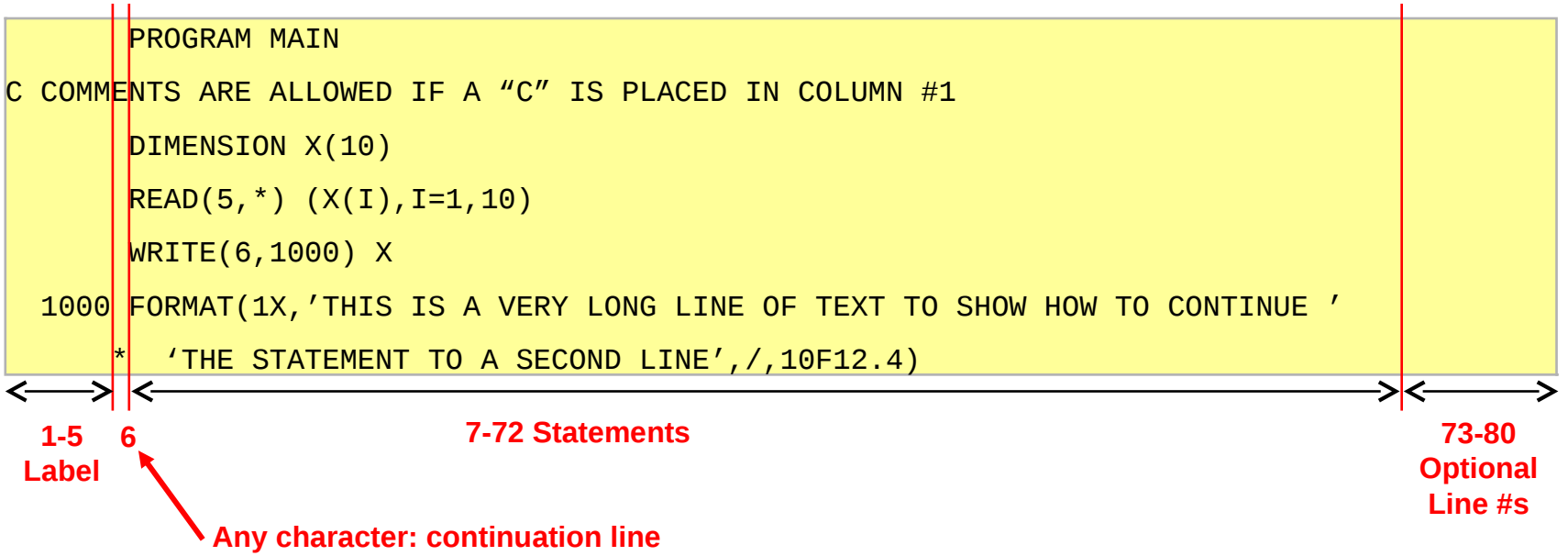
How to Build a FORTRAN Program

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.`**
- 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

Assignment Statements

- **Old** 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

  `arith.f`

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
```

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
`.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 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
.AND.	A .AND. B	logical AND
.OR.	A .OR. B	logical OR
.NEQV.	A .NEQV. B	logical inequivalence
.XOR.	A .XOR. B	exclusive OR (same as .NEQV.)
.EQV.	A .EQV. B	logical equivalence
.NOT.	.NOT. A	logical negation

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

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

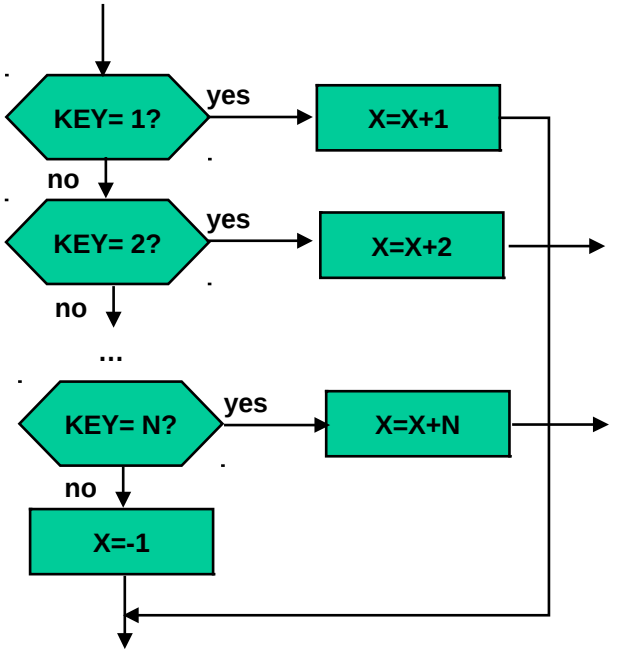
```
10   -code-
     GO TO 30
     -code that is bypassed-
30   -code that is target of GOTO-
     -more code-
     GO TO 10
```

- Flowchart:

- Problem: leads to confusing “spaghetti code”

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 statemens3(s).
  - Ex:

```
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
```

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

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

- 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 named loops if needed - myloop : do i=1,n). This is much like a BREAK statement in other languages.
  - to skip to next loop cycle, use **`CYCLE`** statement in loop.

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).

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)`

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.
- 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
/
```

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. It's also a variable → must be declared !
  - 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(p_1,p_2, … p_N)**
  - Defines function name, *fname*, and argument list, p_1,p_2, … 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

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(p_1,p_2, … p_N)**
  - Defines subroutine name, *sname*, and argument list, p_1,p_2, … 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.

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)

CALL AVG3S(A,C,B,4.1)
```

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

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…).

COMMON MODULE 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
- With Fortran 90, it's better to use the MODULE subprogram instead of the COMMON statement

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:

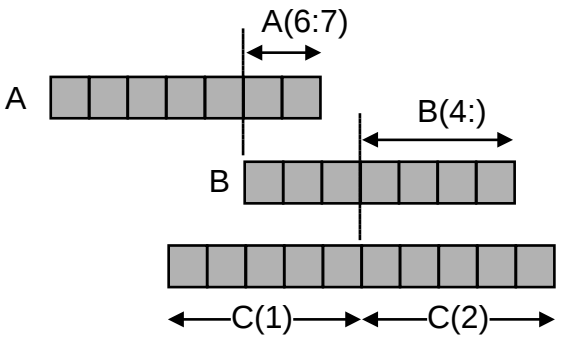
```
PROGRAM MAIN
DIMENSION A(5),B(5),C(10,10),D(10)
EQUIVALENCE (A(1),B(1)),(A(5),ALAST)
EQUIVALENCE (C(1,1),D(1))
...
```

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))
...
```

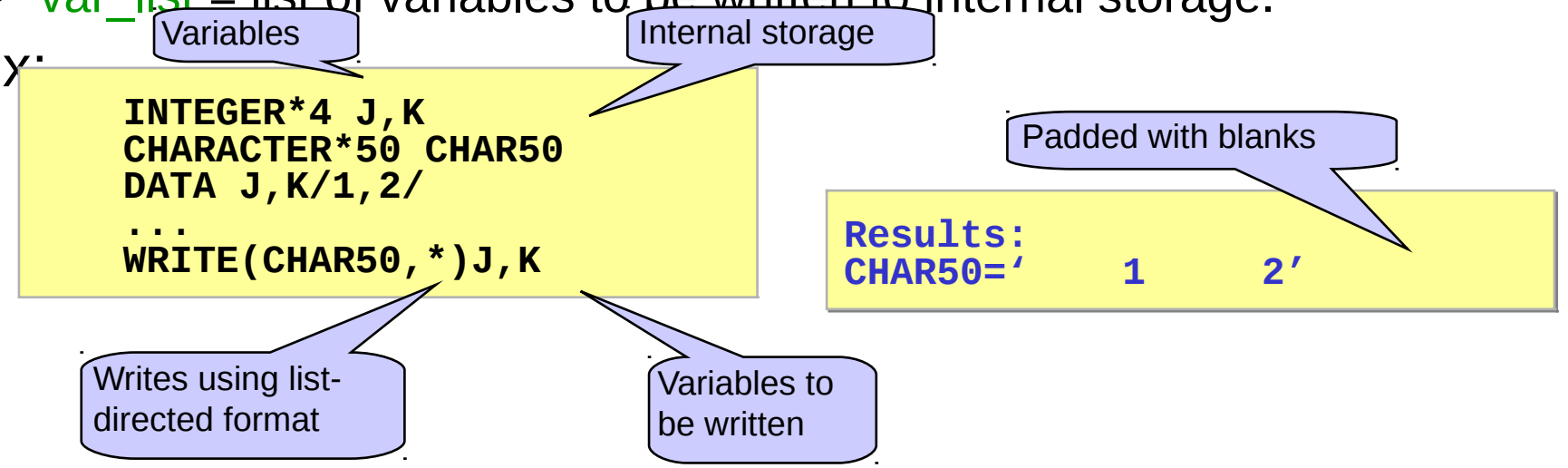


Reading & Writing to/from Internal Storage

- 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.

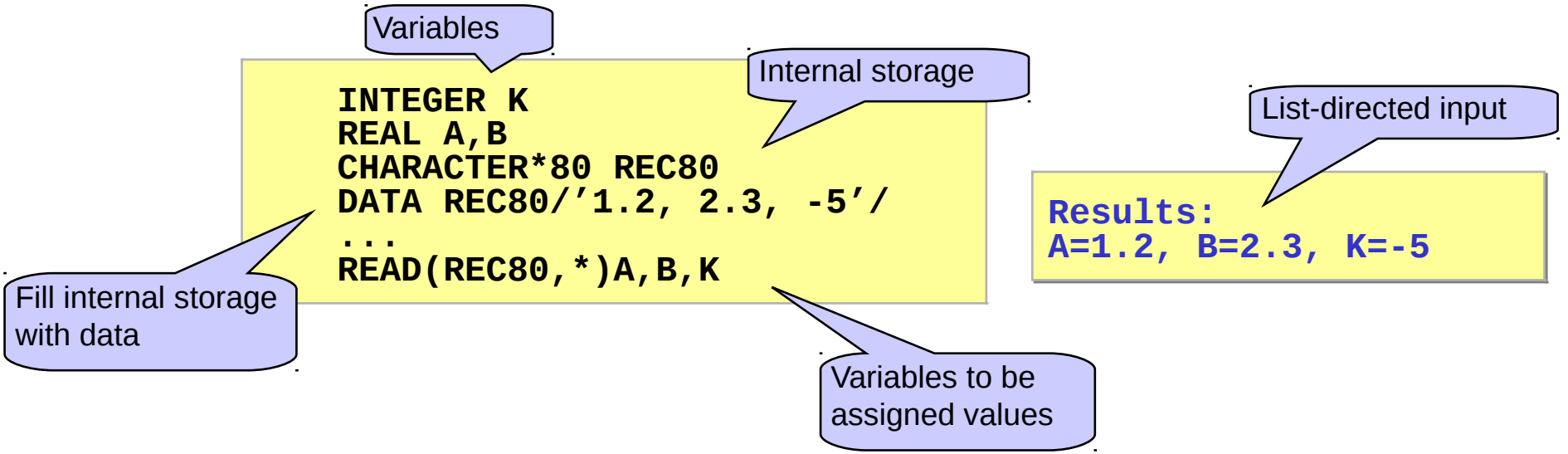
Internal WRITE Statement

- 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
  - etc