Karlsruher Institut für Technologie

Fortran 95/2003 Course

I/O instructions and Intrinsics – Robert Barthel
March 26, 2015

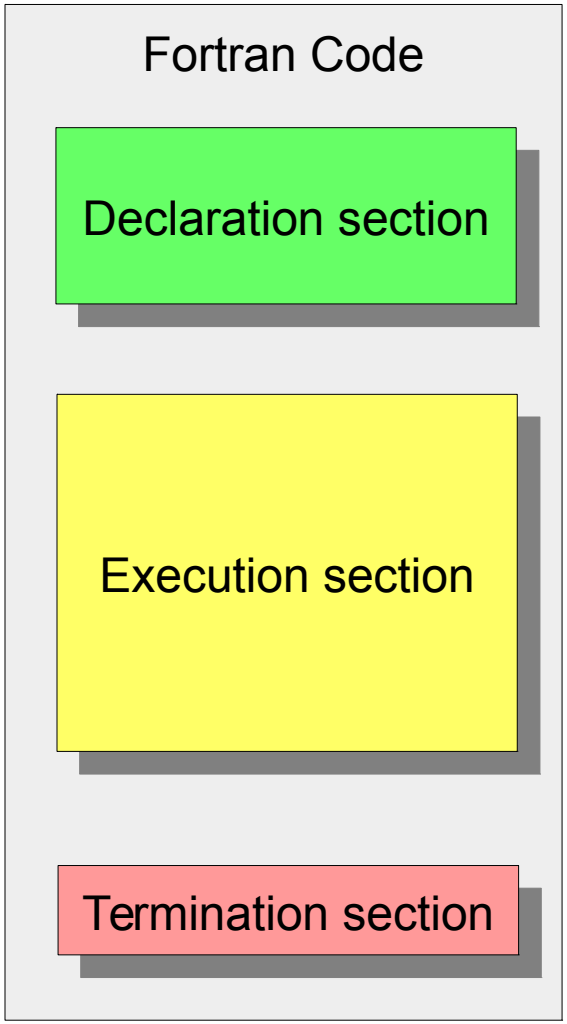
STEINBUCH CENTRE FOR COMPUTING - SCC

Structuring Fortran Code

- Declaration section:
  - Nonexecutable statements
- Execution section:
  - Statements describing actions

  → **I/O and Intrinsics**
- Termination section:
  - Statements stopping execution

  `STOP, END`



Repeat: Good Programming Practice

- Indenting your block structures
- Comment & label your block structures

```
! Comment about loop labelled outer
outer: DO i = 1, dimA
   statements
   ! Comment about loop labelled inner
   inner: DO j = i, dimB
      statements
      ! Comment about if branching
      getlog: IF (a(i,j) >= 0.) THEN
         b(i,j) = log(a(i,j))
         statements
      ENDIF getlog
      statements
   ENDDO inner
   statements
ENDDO outer
```

I/O instructions

Fortran 95 I/O

- Basic Fortran 95 I/O instructions/statements
  - READ
  - PRINT
  - WRITE
  - OPEN
  - CLOSE
  - INQUIRE
  - BACKSPACE
  - REWIND
  - ENDFILE

```
PRINT *, reading real
READ *, r
PRINT *, 'Real =', r
!! Store it to file my_reals.txt
!! name = my_reals.txt

INQUIRE(FILE=name, EXIST=l_ex)

if (.not.l_ex) stop 'file no existing'

OPEN(UNIT=12,FILE=name,STATUS=old)
   WRITE(UNIT=12, '(F10.2)') r
CLOSE(UNIT=12)
```

`READ` statement

- Data transfer input statement
- General form

```
READ (<io-control-list>) [<input-item-list>]
```

or

```
READ format [, <input-item-list>]
```

- Already used most simple statement

```
READ *, <variable>
```

`READ` statement: I/O control list

```
READ ([UNIT=]io-unit,          &
      [FMT=]format,            &
      [NML=]name,              &
      ADVANCE=char_expr,       &
      BLANK=char_expr,         &
      END=stmt_label,          &
      EOR=stmt_label,          &
      ERR=stmt_label,          &
      ID=integer_var,          &
      IOMSG=iomsg_var,         &
      IOSTAT=ios,              &
      NUM=integer_var,         &
      PAD=char_expr,           &
      POS=integer_expr,        &
      REC=integer_expr,        &
      SIZE=count) [<io-item-list>]
```

→ Descriptions on the the following slides

→ Do not use statement labels `END`, `EOR`, `ERR`

READ: UNIT clause

- [UNIT=] *io-unit*: external unit specifier

- *io-unit* is the device unit number from which input is taken
- *io-unit* may be
  - → non-negative integer value or expression (*0* ≤ *io-unit*)
  - → * for default input unit (STDIN)
  - → Integer variable

- A unit number identifies exactly one external file only
- *io-unit=0/5/6* is standard error/input/output

- UNIT= can be omitted if the unit clause is first in list
- A unit must be connected to a file before use

READ: FMT clause

- [FMT=]*format*: format specifier
  → Identifies the needed format
- *format* can be
  - integer as the label of a FORMAT statement
  - character variable with format specification
  - character constant with format specification
  - character expression that gives a format specification
  - * to indicate list-directed formatting

- Example

```
READ (12, '(a6,2i8)') record,i1,i2
READ (23, 2000) line1,r1,r2
2000 FORMAT (a80,2f8.2)
```

READ: NML clause

- [NML=] *name*: NAMELIST group name
- Specifies the name of a namelist group declared in a NAMELIST statement
- Syntax:

```
&name entity1=value1 [,entity2=value2] [,…] /
```

- Example

```
CHARACTER (LEN=20) :: sname,fname
INTEGER :: zipcode
NAMELIST /address/ sname,fname,zipcode
…
READ (*, nml=address)
WRITE (*,*) sname, fname, zipcode
```

Input

```
&address sname='Barthel', fname='Robert',zipcode=76131/
```

READ: ADVANCE/SIZE clause

- `ADVANCE=`*`char_expr`*
  - → advance specifier
  - → Possible values for *`char_expr`*: "YES" or "NO"
  - → Default value is YES
  - → For `FMT=*, ADVANCE="NO"` is not allowed

- `SIZE=`*`count`*
  - → Only for nonadvancing I/O:
  - → Number of characters read during nonadvancing I/O

READ: IOSTAT / IOMSG clause

- IOSTAT=*ios*:

  → status specifier (integer variable)

  ios=0: no error, no EOR (end of record), no EOF (end of file)

  ios>0: error occurred

  ios<0: EOR or EOF reached

- IOMSG=*iomsg-variable*:

  → status specifier (character var.)

  → since Fortran 2003

  for *ios>0* value *iomsg-variable* contains explanatory message

READ Examples: I/O-control-list (1)

Example 1

```
...
CHARACTER (LEN=4)  :: key
CHARACTER (LEN=80) :: io_msg
INTEGER            :: u_no=12, s, ios
!! Open file keys.dat with unit number 12
OPEN (UNIT=u_no, FILE='keys.dat')
   READ (u_no,'(A4)',ADVANCE='no',SIZE=s,IOSTAT=ios,IOMSG=io_msg) key
   IF (ios == 0) THEN
      WRITE(*,*) s, key
   ELSE
      !! End of file or end of record
      key(s+1:) = 'none'
      WRITE(*,'(2A)') 'I/O message: ', io_msg
   END IF
CLOSE (u_no)
...
```

`READ`: Depreciated for I/O-control-list

- `END=`*`stmt_label`*`:`
  → end-of-file specifier
  → Program continues at statement label if an endfile record encounters and no error occurs

- `ERR=`*`stmt_label`*`:`
  → error specifier
  → Program continues at statement label if an error occurs

- `EOR=`*`stmt_label`*`:`
  → end-of-record specifier
  → Program continues at statement label if an end-of-record condition occurs and no error occurs

`PRINT` statement

- Data transfer output statement (STDOUT)
  - → General form

```
PRINT format [, <output-item-list>]
```

  - → Already used most simple statement

```
PRINT *, <variable>
```

- Example:

```
INTEGER :: n=10; REAL :: a=14.236, b=77.77
PRINT    *, n,a,b
PRINT 1000, n,a,b
1000 FORMAT (i4,1x,f8.2,1x,e16.10)
```

- Output:

```
10 14.236 77.77
  10    14.24 0.7776999664E+02
```

`WRITE` statement

- Data transfer output statement

  → General form

  ```
  WRITE (<io-control-list>) [<output-item-list>]
  ```

- `PRINT format`

  is the same as

  `WRITE (*, format)`

- Most simple statement

  ```
  WRITE (*,*) <variable>
  ```

- If one writes to unit number `u` with no file associated to it, usually a file "`fort.<u>`" is created as result
- Prettify your output by using format edit descriptors

Format edit descriptors

`'string'`	Insert string
`A, Aw`	Read/Write next entity/next ***w*** characters
`Bw[.m]`	Read/Write binary values (***m***>0: leading zeros)
`Dw.d`	Double precision real number
`Ew.d[Ee]`	Read/Write next ***w*** characters as real number with ***d*** digits after the decimal place (if no decimal point in input); ***e*** exponent digits
`ENw.d[Ee]`	Real numbers in engineering notation (exponent divisible by 3)
`ESw.d[Ee]`	Real numbers in scientific notation (digit /= 0 before decimal point)

Format edit descriptors (2)

`Fw.d`	Read/Write *w* characters as real number with *d* digits after the decimal point
`Gw.d[Ee]`	Real numbers, extended range
`Iw[.m]`	Read/Write *w* characters as integer number (*m*>0: leading zeros)
`Lw`	Read/Write *w* characters as logical value
`Ow[.m]`	Read/Write *w* characters as octal value
`Zw[.m]`	Read/Write *w* characters as hexadecimal value
`nx`	Read: Ignore the next *n* characters Write: Print *n* spaces
`Tc`	Next character at position *c* (overwriting!)

Format edit descriptors (3)

Examples

```
INTEGER (KIND=4) :: i = 38
```

```
PRINT *, i
WRITE (*,*) i
WRITE (*, '(I6)') i
WRITE (*, '(I6.3)') i
WRITE (*, '(I0,A)') i, '##'
WRITE (*, '(I0.3,A)') i, '##'
```

Output

```
38
38
    38
   038
38##
038##
```

Format edit descriptors (4)

Examples (2)

```
REAL (KIND=8), PARAMETER :: one = 1.
REAL (KIND=8)            :: r = exp(one)
```

```
PRINT *, r
WRITE (*,*) r
WRITE (*, '(E15.5)') r
WRITE (*, '(E15.5E1)') r
WRITE (*, '(E15.5E3)') r
WRITE (*, '(D15.5)') r
WRITE (*, '(F15.5)') r
WRITE (*, '(F0.5,a)') r,'###'
WRITE (*, '(ES15.5)') r
WRITE (*, '(ES15.5)') 100.*r
```

Output

```
2.718281828459045
2.718281828459045
    0.27183E+01
     0.27183E+1
   0.27183E+001
    0.27183D+01
        2.71828
2.71828###
    2.71828E+00
    2.71828E+02
```

Format edit descriptors (5)

Examples (3)

```
WRITE (*, '(EN15.5)') r
WRITE (*, '(EN15.5)') 10.*r
WRITE (*, '(EN15.5)') 100.*r
WRITE (*, '(EN15.5)') 1000.*r
WRITE (*, '(G15.5)') 0.01*r
WRITE (*, '(G15.5)') 0.10*r
WRITE (*, '(G15.5)') 1.0*r
WRITE (*, '(G15.5)') 10.*r
WRITE (*, '(G15.5)') 100.*r
```

Output

```
  2.71828E+00
 27.18282E+00
271.82818E+00
  2.71828E+03
  0.27183E-01
  0.27183
   2.7183
   27.183
   271.83
```

Format edit descriptors (6)

- More complicated example

```
REAL :: a=2, b=3, c=4, vol
vol = a*b*c
WRITE (*, '(3(8X,A), T30, "|", A &
   / 1X, 44("-"), T30, "|" &
   / 3(1X,F8.2), 5X, E15.3, T30, "|")') &
   "a", "b", "c", " Volume", a, b, c, vol
```

- Output:

```
       a        b        c   | Volume
 ---------------------------|---------------
    2.00     3.00     4.00   |        0.240E+02
```

Formatted vs. unformatted I/O

- Formatted:
  → By default all files are text files, or formatted files
  - Human readable
  - Portable, i.e. machine independent

- Unformatted
  → Fortran can also use binary files, or unformatted files
  - Machine readable only
  - Much faster to use than formatted files
  - Suitable for large amount of data (reduces file size)
  - Internal format used for numbers: no number conversion, no rounding errors
  - Not necessarily portable (need to have the same internal data representation between machines that are writing and reading such files)

Formatted vs. unformatted I/O (2)

- Example of unformatted write

```
OPEN (10, FILE='foo.dat', FORM='unformatted')
WRITE (10) rval
WRITE (10) string
CLOSE (10)
```

- Notice: no format (`FMT=` specifier) in write:

```
WRITE (unit=u) …
```

- Reading is done similarly

```
OPEN (10, FILE='foo.dat', FORM='unformatted')
READ (10) rval
READ (10) string
CLOSE (10)
```

Internal I/O = Constructing character strings

- Sometimes it is handy to read/write data from/to Fortran character strings to filter out or to format data
- This can be accomplished by replacing the unit number in read/write statement with a character string
- No external files are used

- Example

```
CHARACTER (LEN=10) :: input = '12345.7890'
INTEGER :: n7890, myproc
CHARACTER (LEN=12) :: filename
…
READ (input, FMT='(6x,i4)') n7890
! n7890 = 7890
WRITE (filename, FMT='(a,i4.4,a)') 'sol_', myproc, '.dat'
! filename = 'sol_0001.dat'
```

`OPEN` statement

- Use to
  - connect an existing external file to a unit
  - create an external file that is preconnected
  - create an external file and connect it to a unit
  - change certain specifiers of a connection between an external file and a unit
- Syntax:

```
OPEN (<open-list>)
```

- `<open-list>`

  → must contain one unit specifier, can also contain one of each of the other valid specifiers

OPEN statement: open-list

```
OPEN ([UNIT=]io-unit,          &
      ACCESS=char_expr,        &
      ACTION=char_expr,        &
      ASYNCH=char_expr,        &
      BLANK=char_expr,         &
      DELIM=char_expr,         &
      ERR=stmt_label,          &
      FILE=char_expr,          &
      FORM=char_expr,          &
      IOMSG=iomsg_var,         &
      IOSTAT=ios,              &
      PAD=char_expr,           &
      POSITION=char_expr,      &
      RECL=integer_expr,       &
      STATUS=char_expr)
```

→ Descriptions on the the following slides

→ Do not use statement labels `ERR`

OPEN: UNIT/ACCESS clause

- [UNIT=] *io-unit*
  - → external unit specifier (*iounit* ≥ *0*)
  - → If the optional UNIT= omitted, *io-unit* must be first item in open-list

- ACCESS=*char_expr*
  - → specifies access method for the connection of the file
  - → Possible values for *char_expr*
    "SEQUENTIAL", "DIRECT" (or "STREAM" since Fortran 2003)
  - → Default: "SEQUENTIAL"
  - → If ACCESS="DIRECT", RECL= must be specified

OPEN: ACTION clause

- ACTION=*char_expr*
  → specifies the allowed I/O operations
  → Possible values of *char_expr*
  "READ", "WRITE" or "READWRITE"

- If READ is specified:
  → WRITE and ENDFILE statements cannot refer to this connection

- If WRITE is specified,
  → READ statements cannot refer to this connection

- If ACTION= specifier is omitted
  → default depends on the actual file permissions (STATUS specifier)

OPEN: FILE/FORM clause

- FILE=*char_expr*

  → specifies name of file to be connected to the specified unit

- Depreciated: ERR=*stmt_label* → use IOSTAT

  → error specifier

  → if error → does jump/goto statement label

- FORM=*char_expr*

  → connection specification for formatted or unformatted I/O

  → Possible value for *char_expr*:
  "FORMATTED" or "UNFORMATTED"

  → Default: ACCESS="SEQUENTIAL" => "FORMATTED"
  ACCESS="DIRECT" => "UNFORMATTED"

OPEN: IOSTAT / IOMSG clause

→ See for READ: IOSTAT / IOMSG clause

- IOSTAT=*ios*

  → status specifier (integer variable)

  ios=0: no error

  ios>0: error occurred

- IOMSG=*iomsg-var*:

  → since Fortran 2003

  → status specifier (character var.)

  ios=0: value unchanged

  else: variable is assigned with explanatory message

OPEN: further clauses

- POSITION=*char_expr*:
  → File position for a file connected for sequential or stream access
  → Possible values of *char_expr*:

REWIND:	positions the file at its initial point
APPEND:	positions the file before the endfile record
ASIS:	leaves the position unchanged (default)

- RECL=*integer_expr*:
  → the length of each record in a file being connected for direct access
    → mandadory if a file is connected for direct access
  → *integer_expr* must be positive

OPEN: further clauses (2)

- STATUS=*char_expr*:

  → specifies the status of the file when it is opened

  → Possible values for *char_expr*:

OLD:	Connect an existing file to a unit. If specified, file must exist
NEW:	Create a new file, connect it to a unit, change the status to OLD. If NEW is specified, the file must not exist.
SCRATCH:	Create and connect a new file that will be deleted when it is disconnected
REPLACE:	If not existing, file is created. If existing, file is deleted, creation of new file with the same name. Afterwards changes to OLD.
UNKNOWN:	If the file exists, it is connected as OLD. If the file does not exist, it is connected as NEW.
Default	→ UNKNOWN

Direct access files

- Access type by default sequential:
  → lines are read/written one after another,
- Direct access files
  → allows random order → practical, e.g. in database access
  → normally binary (unformatted), but can also be formatted
  → Length of a record is given by `RECL=` specifier in `OPEN`.
    → `RECL` = no. `CHARS` for formatted / no. bytes for unformatted files

- Example:

```
INTEGER, DIMENSION(10) :: r
INTEGER :: u = 20
OPEN (u, FILE='data.db', ACCESS='direct', RECL=40)
WRITE (u, REC=13) r
```

CLOSE statement

- Disconnects an external file from a unit
- Syntax

```
CLOSE (<close-list>)
```

→ close-list must contain one unit specifier, can also contain one of each of the other valid specifiers

- Valid specifiers of close-list:

```
CLOSE ([UNIT=]io-unit,        &
       ERR=stmt_label,        &
       IOMSG=iomsg_var,       &
       IOSTAT=ios,            &
       STATUS=char_expr)
```

CLOSE: UNIT / IOSTAT / IOMSG clause

- [UNIT=] *io-unit*:
  - → external unit specifier (*iounit* ≥ *0*)
  - → If optional UNIT= is omitted, *io-unit* must be first item in close-list

- IOSTAT=*ios*:
  - → status specifier (integer variable)
    ios=0: no error, *ios>0*: error occurred

- IOMSG=*iomsg-var*:
  - → status specifier (character var.)
    ios=0: imsg-var unchanged; else: explanatory message
  - → Since Fortran 2003

CLOSE: STATUS clause

- STATUS=*char_expr*:
  - → specifies the status of the file after it is closed
  - → Possible values for *char_expr*:
    "KEEP" or "DELETE"
  - → KEEP:
    Only and existing file will continue to exist after the CLOSE statement
  - → DELETE:
    File will not exist after the CLOSE statement

`INQUIRE` statement

- Obtains info about the properties of a named file or the connection to a particular unit

- Three forms of the INQUIRE statement
  - Inquire by file, which requires the `FILE=` specifier
  - Inquire by output list, which requires the `IOLENGTH=` specifier
  - Inquire by unit, which requires the `UNIT=` specifier
- Syntax:

```
INQUIRE (<inquiry-list>)
```

or

```
INQUIRE (IOLENGTH=iol) <output-item-list>
```

→ e.g. to check if a file exists before connecting to it or
to determine the length of an unformatted output item list

INQUIRE statement (2)

- Check if file exists before connecting it to a unit by an OPEN statement

```
INQUIRE (FILE=char_expr, EXIST=ex)
```

→ FILE= specifies the name of the file

→ EXIST= logical if a file exists: .true. = yes, .false. = no

- Example:

```
LOGICAL :: ex
INQUIRE (FILE="test", EXIST=ex)
IF (.not.ex) THEN
  PRINT *, "File <test> does not exist"; STOP
END IF
OPEN (UNIT=11, FILE="test", IOSTAT=ios)
```

INQUIRE statement (3)

- Determine the length of an unformatted output item list
- Syntax:

```
INQUIRE (IOLENGTH=iol) <output-item-list>
```

→ `iol` may be used as value for `RECL=` specifier in subsequent `OPEN` statement

- Example:

```
INTEGER :: rec_len, u_no=11
...
INQUIRE (IOLENGTH=rec_len) name, title, age, address, tel
OPEN (UNIT=u_no, FILE="test", RECL=rec_len, FORM="unformatted")
    WRITE (u_no) name, title, age, address, tel
CLOSE (u_no)
```

File positioning statements

- `BACKSPACE:`

  → file connected to specified unit → positioned before the preceding record

```
BACKSPACE([UNIT=]u [, IOSTAT=ios])
```

- `REWIND:` causes the specified file to be positioned at its initial point

```
REWIND([UNIT=]u [, IOSTAT=ios])
```

- `ENDFILE:`

  → writes an endfile record as the next record of the file

  → file is then positioned after the endfile record

  → reposition for further I/O only via `BACKSPACE` and `REWIND`

```
ENDFILE([UNIT=]u [, IOSTAT=ios])
```

Intrinsic procedures

Fortran intrinsic procedures

- Fortran defines a number of procedures, called intrinsic procedures, that are available to any program
- 4 classes of intrinsic procedures
  - Inquiry intrinsic functions
  - Elemental intrinsic procedures
  - Transformational intrinsic functions
  - Intrinsic subroutines

Fortran intrinsic procedures (2)

- In the following intrinsic function definitions, arguments are used according to their implicit types
  - `I` for `INTEGER`
  - `C` for `CHARACTER`
  - `X` for `INTEGER` or `REAL`
  - Other for `REAL`
- Used keywords
  - `KIND` describes the KIND number
  - `SET` a string that contains a set of characters
  - `BACK` a logical scalar
  - `MASK` a logical array
  - `DIM` a selected dimension of an argument (`INTEGER`)

Inquiry intrinsic functions

- Set of 19 functions
- Result depends on the properties of its principal arguments, not on the values of the arguments

```
ALLOCATED
ASSOCIATED
BIT_SIZE
DIGITS
EPSILON
HUGE
KIND
LBOUND
LEN
```

```
MAXEXPONENT
MINEXPONENT
PRECISION
PRESENT
RADIX
RANGE
SHAPE
SIZE
TINY
UBOUND
```

Inquiry intrinsic functions (2)

- `BIT_SIZE(I)`

  returns the number of bits in an integer type
- `DIGITS(X)`

  returns the number of significant digits in X
- `EPSILON(X)`

  returns the smallest difference representable by the type and KIND of X
- `HUGE(X)` / `TINY(R)`

  returns the largest/smallest positive number representable by the type and KIND of X/REAL

Inquiry intrinsic functions (3)

Example

```
INTEGER :: i
REAL :: r
REAL (KIND=8) :: d

PRINT *, BIT_SIZE(i)
PRINT *, DIGITS(i), DIGITS(d), DIGITS(r)
PRINT *, EPSILON(d), EPSILON(r)
PRINT *, HUGE(d), HUGE(r), HUGE(i)
PRINT *, TINY(d), TINY(r)
```

Output

```
          32
          31          53          24
  2.22044604925031E-016  1.1920929E-07
  1.79769313486231E+308  3.4028235E+38  2147483647
  2.22507385850720E-308  1.1754944E-38
```

Inquiry intrinsic functions (4)

- `KIND(X)`

  returns the value of the kind type parameter of X

- `LEN(STRING)`

  returns the length of a string

- `MAXEXPONENT(R)` / `MINEXPONENT(R)`

  returns the maximum/minimum exponent value in the model

Inquiry intrinsic functions (5)

Example

```
INTEGER :: i
REAL :: r
REAL (KIND=8) :: d
CHARACTER (LEN=32) :: s

PRINT *, KIND(i), KIND(r), KIND(d)
PRINT *, LEN(s)
PRINT *, MAXEXPONENT(r), MAXEXPONENT(d)
PRINT *, MINEXPONENT(r), MINEXPONENT(d)
```

Output

```
           4           4           8
          32
         128        1024
        -125       -1021
```

Inquiry intrinsic functions (6)

- `PRECISION(R)`

  returns the decimal precision in the model

- `PRESENT(A)`

  determine whether an optional argument is present

- `RADIX(X)`

  return the base of the model

Inquiry intrinsic functions (7)

Example

```
SUBROUTINE sub(x,y)
REAL :: x
REAL, OPTIONAL :: y

IF (PRESENT(y)) THEN
  ! use y like any other variable
  x = x+y
ELSE
  ! here we cannot use or define y
  x = x+100
END IF

RETURN
END SUBROUTINE sub
```

Inquiry intrinsic functions (8)

Example

```
INTEGER :: i
REAL :: r
REAL (KIND=8) :: d

PRINT *, PRECISION(r), PRECISION(d)
PRINT *, RADIX(r), RADIX(d), RADIX(i)
```

Output

```
           6          15
           2           2           2
```

Elemental intrinsic functions

- Set of 68 functions
- Specified for *scalar* arguments, but also accept *arrays*
- If all arguments are scalar, the result is scalar
- If any argument is an array,
- → all arguments must be arrays of the same shape or conformable with them.

  → The shape of the result is the shape of the argument with the greatest rank.

  → The elements of the result are the same as if the function was applied individually to the corresponding elements of each argument.

Elemental intrinsic functions (2)

```
ABS
ACHAR
ACOS
AJUSTL
AJUSTR
AIMAG
AINT
ANINT
ASIN
ATAN
ATAN2
BTEST
CEILING
CHAR
CMPLX
CONJG
COS
COSH
```

```
DBLE
DIM
DPROD
EXP
EXPONENT
FLOOR
FRACTION
IACHAR
IAND
IBCLR
IBITS
IBSET
ICHAR
IEOR
INDEX
INT
IOR
ISHFT
```

```
ISHFTC
LEN_TRIM
LGE
LGT
LLE
LLT
LOG
LOG10
LOGICAL
MAX
MERGE
MIN
MOD
MODULO
MVBITS
NEAREST
NINT
NOT
```

```
REAL
RRSPACING
RSHIFT
SCALE
SCAN
SET_EXPONENT
SIGN
SIN
SINH
SPACING
SQRT
TAN
TANH
VERIFY
```

Elemental intrinsic functions (3)

- Numeric
  - `CEILING(A)`: returns smallest integer ≥ A
  - `FLOOR(A)`: returns largest integer ≤ A
  - `MODULO(A, P)`: returns remainder of A/P
  - `ANINT(A)`: rounds to the nearest value of A
  - `DIM(X, Y)`: returns maximum of X-Y or 0
  - `NINT(A)`: returns the nearest integer to A
  - `REAL(A)`: converts to a real
  - `INT(A)`: converts to an integer
  - `SIGN(A, B)`: returns |A| if B≥0 and -|A| if B<0
  - `MAX(A1, A2 [, A3…])`: returns the maximum value
  - `MIN(A1, A2 [, A3…])`: returns the minimum value

Elemental intrinsic functions (4)

- Mathematical
  - `COS(R)` / `ACOS(R)` — cosine / arccosine
  - `SIN(R)` / `ASIN(R)` — sine / arcsine
  - `TAN(R)` / `ATAN(R)` — tangent / arctangent
  - `COSH(R)` — hyperbolic cosine
  - `SINH(R)` — hyperbolic sine
  - `TANH(R)` — hyperbolic tangent
  - `EXP(R)` — exponential function
  - `LOG(R)` — natural logarithm
  - `LOG10(R)` — base 10 logarithm
  - `SQRT(R)` — square root

Elemental intrinsic functions (5)

- Character
  - `ACHAR(I)`

    returns character corresponding to I^{th} character in ASCII
  - `IACHAR(C)`

    returns ASCII code corresponding to character C
  - `ADJUSTL(STRING)` / `ADJUSTR(STRING)`

    left / right adjust STRING by removing leading / trailing spaces, spaces are inserted at the end / start of the string
  - `LEN_TRIM(STRING)`

    returns length of STRING neglecting trailing spaces

Elemental intrinsic functions (6)

Example

```
INTEGER :: i=71
CHARACTER :: c="h"
CHARACTER (LEN=28) :: s="    I love Fortran          "

PRINT *, ACHAR(i), IACHAR(c)
PRINT *, ADJUSTL(s)
PRINT *, ADJUSTR(s)
PRINT *, LEN_TRIM(s)
```

Output

```
G           104
I love Fortran
                I love Fortran
           18
```

Elemental intrinsic functions (7)

- Character
  - `INDEX(STRING, SUBSTRING [, BACK])`

    returns starting position of SUBSTRING within STRING
  - `SCAN(STRING, SET [, BACK])`

    returns the position of the first occurance of any of the characters

    in SET within STRING
  - `VERIFY(STRING, SET [, BACK])`

    returns 0 if every character in STRING is also in SET;

    otherwise, returns the position of the first character in STRING that is not in SET

Elemental intrinsic functions (8)

Example

```
CHARACTER (LEN=40) :: &
  s="She sells sea shells by the sea shore."
CHARACTER (LEN=7)  :: t="fortran"

PRINT *, &
  INDEX(s,"sea"), INDEX(s,"sea",.true.), INDEX(s,"Sea")
PRINT *, SCAN(t,"ao"), SCAN(t,"ao",.true.), SCAN(t,"C")
PRINT *, &
  VERIFY(t,"ao"), VERIFY(t,"fon"), VERIFY(t,"fortran")
```

Output

```
          11          29           0
           2           6           0
           1           3           0
```

Elemental intrinsic functions (9)

- Bit manipulation
  - `BTEST(I, POS)`

    returns .TRUE. if bit POS of I is 1, otherwise .FALSE.
  - `IBCLR(I, POS)`

    returns I with bit POS set to 0
  - `IBSET(I, POS)`

    returns I with POS set to 1
  - `IBITS(I, POS, LEN)`

    returns a right-adjusted sequenced of bits from I of length LEN starting with bit POS; all other bits 0

Elemental intrinsic functions (10)

Example

```
INTEGER :: i=1024, a=Z'F', b=Z'5'

PRINT *, BTEST(i,6), BTEST(i,10)
PRINT *, IBITS(i,8,3), IBCLR(a,2), IBSET(b,5)
```

Output

```
 F T
           4           11           37
```

Elemental intrinsic functions (11)

- Bit manipulation
  - `IAND(I, J)` / `IOR(I, J)`

    returns the bitwise AND / OR of I and J
  - `IEOR(I, J)`

    returns the bitwise exclusive OR (XOR) of I and J
  - `INOT(I)`

    returns the bitwise complement (negation) of I
  - `ISHFT(I, SHIFT)`

    returns I shifted bitwise SHIFT bits to the left; vacated position filled with 0
  - `ISHFTC(I, SHIFT [, SIZE]))`

    returns I circular-shifted bitwise SHIFT bits to the left

Elemental intrinsic functions (12)

Example

```
INTEGER :: a=Z'F', b=Z'5'

PRINT *, IAND(a,b), IOR(a,b)
PRINT *, IEOR(a,b), NOT(a)
```

Output

```
           5          15
          10         -16
```

Elemental intrinsic functions (13)

- Floating point manipulation
  - `EXPONENT(X)`

    returns exponent of X in base 2 (=1+log_2(int(X)))
  - `FRACTION(X)`

    returns fractional part of X
  - `NEAREST(X, S)`

    returns nearest machine-representable number of X in direction S (S>0 forward, S<0 backward)
  - `RRSPACING(X)`

    returns reciprocal of relative spacing of numbers near X

Elemental intrinsic functions (14)

Example

```
REAL :: x=1E2, y=17.77E16

PRINT  *, EXPONENT(x), EXPONENT(y)
PRINT  *, FRACTION(x), FRACTION(y)
PRINT 10, NEAREST(x,1.0), NEAREST(x,-1.0)
PRINT  *, RRSPACING(x), RRSPACING(y)

10 FORMAT (2E18.10)
```

Output

```
           7           58
  0.7812500      0.6165207
  0.1000000076E+03  0.9999999237E+02
  1.3107200E+07  1.0343501E+07
```

Elemental intrinsic functions (15)

- Floating point manipulation
  - `SCALE(X, Y)`

    returns X*(2**Y)

  - `SET_EXPONENT(X, I)`

    returns a number whose fractional part is the same as X and whose exponent is I

  - `SPACING(X)`

    returns the absolute spacing of numbers near X

Elemental intrinsic functions (16)

Example

```
REAL :: x=1E2, y=17.77E16

PRINT *, SCALE(x,2), SCALE(y,-1)
PRINT *, SET_EXPONENT(x,1), SET_EXPONENT(y,4)
PRINT *, SPACING(x), SPACING(y)
```

Output

```
   400.0000       8.8849997E+16
   1.562500        9.864331
  7.6293945E-06   1.7179869E+10
```

Transformational intrinsic functions

- Set of 23 functions
- Generally accept array arguments and return array results dependent on values of elements in argument arrays

```
ALL
ANY
COUNT
CSHIFT
DOT_PRODUCT
EOSHIFT
MATMUL
MAXLOC
MAXVAL
MINLOC
MINVAL
PACK
```

```
PRODUCT
REPEAT
RESHAPE
SELECTED_INT_KIND
SELECTED_REAL_KIND
SPREAD
SUM
TRANSFER
TRANSPOSE
TRIM
UNPACK
```

Transformational intrinsic functions (2)

- Array reduction functions
  - `ALL(MASK [, DIM]) / ANY(MASK [, DIM])`

    returns .TRUE. If all / any elements of MASK are .TRUE.
  - `COUNT(MASK [, DIM])`

    returns the number of elements of MASK that are .TRUE.
  - `MAXVAL(ARRAY [, DIM] [, MASK]),`
    `MINVAL(ARRAY [, DIM] [, MASK])`

    returns the value of the maximum / minimum array element
  - `PRODUCT(ARRAY [, DIM] [, MASK])`

    returns the product of array elements
  - `SUM(ARRAY [, DIM] [, MASK])`

    returns the sum of array elements

Transformational intrinsic functions (3)

Example

```
INTEGER :: x(6) = &
  (/ 1, 666666, 22, 55555, 333, 4444 /)
LOGICAL :: l(6) = &
  (/ .true., .false., .true., .false., .true., .false. /)

PRINT *, MAXVAL(x), MAXLOC(x)
PRINT *, MAXVAL(x, MASK=x<100), MAXLOC(x, MASK=x<100)
PRINT *, MAXVAL(x, MASK=l), MAXLOC(x, MASK=l)
```

Output

```
      666666           2
          22           3
         333           5
```

Transformational intrinsic functions (4)

- Array constructor functions
  - `PACK(ARRAY, MASK [, VECTOR])`

    pack elements of ARRAY corresponding to true elements of MASK into a rank one result

  - `SPREAD(SOURCE, DIM, NCOPIES)`

    returns an array of rank which is one greater than rank of SOURCE containing NCOPIES of SOURCE

  - `UNPACK(VECTOR, MASK, FIELD)`

    unpack elements of rank one array VECTOR with at least the number of true elements of MASK

Transformational intrinsic functions (5)

- Example: `PACK`

```
! A is the array | 0 7 0 |
!                | 1 0 3 |
!                | 4 0 0 |

PRINT *, PACK(A, MASK=A .NE. 0, &
              VECTOR=(/ -1,-1,-1,-1,-1,-1 /))
```

Output

```
 1 4 7 3 -1 -1
```

Transformational intrinsic functions (6)

Example: UNPACK

```
! VECTOR is the array (/ 5, 6, 7, 8 /),
! MASK is | F T T |, FIELD is | -1 -4 -7 |
!        | T F F |           | -2 -5 -8 |
!        | F F T |           | -3 -6 -9 |

PRINT *, UNPACK(VECTOR, MASK, FIELD)
PRINT *, UNPACK(VECTOR, MASK, FIELD=0)
```

Output

```
 -1 5 -3 6 -5 -6 7 -8 8
 0 5 0 6 0 0 7 0 8
```

```
| -1  6  7 |
|  5 -5 -8 |
| -3 -6  8 |
```

```
|  0  6  7 |
|  5  0  0 |
|  0  0  8 |
```

Transformational intrinsic functions (7)

Example: SPREAD

```
! A is the array (/ -4, 6, 1 /)

PRINT *, SPREAD(A, DIM=1, NCOPIES=3)
PRINT *, SPREAD(A, DIM=2, NCOPIES=3)
PRINT *, SPREAD(A, DIM=2, NCOPIES=0)
```

Output

```
-4 -4 -4 6 6 6 1 1 1
-4 6 1 -4 6 1 -4 6 1
```

```
| -4  6  1 |
| -4  6  1 |
| -4  6  1 |
```

```
| -4 -4 -4 |
|  6  6  6 |
|  1  1  1 |
```

```
(/        /)
```

Transformational intrinsic functions (8)

- KIND
  - `SELECTED_INT_KIND(I)`

    returns a KIND value representing all INTEGERS < 10^I
  - `SELECTED_REAL_KIND(I1 [,I2])`

    returns a KIND value representing all REALs with at least I_1 significant digits and a decimal exponent range of at least I_2

Transformational intrinsic functions (9)

Example

```
INTEGER, PARAMETER :: &
  ki1 = SELECTED_INT_KIND(8),        &
  ki2 = SELECTED_INT_KIND(10),       &
  kr1 = SELECTED_REAL_KIND(15,300), &
  kr2 = SELECTED_REAL_KIND(25,300)

PRINT *, ki1, ki2
PRINT *, kr1, kr2
```

Output

```
           4           8
           8          16
```

Intrinsic procedures

- Set of 5 procedures
  - `DATE_AND_TIME([DATE] [,TIME] [,ZONE] [,VALUES])`
    returns the current date and time
  - `SYSTEM_CLOCK([COUNT] [,COUNT_RATE] [,COUNT_MAX])`
    returns data from the system real-clock
  - `RANDOM_NUMBER(HARVEST)`
    returns a (pseudo-)random number in `HARVEST`
  - `RANDOM_SEED([SIZE] [,PUT] [,GET])`
    restarts the pseudo-random number generator used by `RANDOM_NUMBER`, or returns parameters about the generator
  - `MVBITS(FROM,FROMPOS,LEN,TO,TOPOS)`
    copies a sequence of bits in `FROM`, starting at `FROMPOS` and `LEN`
    bits long, into `TO` beginning at `TOPOS`

Random number generator

```
PROGRAM random
  INTEGER, DIMENSION(8) :: time_info
  INTEGER :: msec,i,n
  REAL, DIMENSION(10) :: num

  CALL DATE_AND_TIME(VALUES=time_info)
  ! VALUES(7): seconds of the minute
  ! VALUES(8): milliseconds of the second
  msec = time_info(7)*1000+time_info(8)

  CALL RANDOM_SEED(SIZE=n)  ! sets n
  CALL RANDOM_SEED(PUT=(/ (msec+10*i,i=1,n) /))
  CALL RANDOM_NUMBER(num)
END PROGRAM random
```

Karlsruhe Institute of Technology

Thank you for your attention!