Scientific Programming With Modern Fortran

M. D. Jones, Ph.D.

Center for Computational Research
University at Buffalo
State University of New York

High Performance Computing I, 2014

Part I

Fortran Basic Operations

History of FORTRAN

One of the earliest of the high-level programming languages, FORTRAN (short for **For**mula **Tran**slation) was developed by John Backus at IBM in the 1950s (circa 1953 for the 704, first released in 1957):

- FORTRAN 66, ANSI standard (X3.9-1966) based on FORTRAN IV
- FORTRAN 77, X3.9-1978, improved I/O
- Fortran 90, ISO/IEC 1539:1991(E), FORTRAN becomes Fortran
- Fortran 95, ISO/IEC 1539-1:1997, minor revisions
- Fortran 2003, ISO/IEC 1539:2004, command line arguments, more intrinsics, IEEE exception handling, C interoperability
- Fortran 2008, revision to Fortran 2003, to include `BIT` type and `Co-array` parallel processing (starting to become better supported)

Why Is This Language Still Used?

So why is Fortran still in use?

- **Efficiency** has always been a priority, in the sense that compilers should produce efficient code. Fortran data are generally not allowed to be aliased (e.g. pointers) making life easier on the compiler
- Designed from the outset as the tool for numerically intensive programming in science and engineering
- Backward compatibility - most modern compilers can still compile old (as in decades!) code

Come On, Fortran is 50 Years Old!

Indeed, Fortran celebrated its 50th birthday in 2007. How many other high-level languages can you say that about? Think of it this way - it is arguably the most efficient, compact, portable high-level language available. And you will likely still be able to use programs written today until you retire (unmodified, at that) ...

Some Fortran Compilers

Fortran compilers are easy to find, some are freely available (including compiler source code):

- `gfortran`, part of the GNU toolchain (gcc/g++), supplanted `g77` as of version 4.0
- `g95`, another freebie
- `open64`, open sourced by SGI (derived from their MIPSPro compiler suite), further developed by Intel and the Chinese Academy of Sciences (Gnu Public License), intended mainly for compiler research (includes C/C++) on the IA64 platform
- `ifort`, Intel's commercial compiler
- `pgf95`, PGI's commercial compiler
- Other commercial compilers: IBM, Qlogic/Pathscale, Lahey/Fujitsu, Sun, NAGWare, ...

The Rest of This Talk

In this presentation I will focus almost entirely on Fortran >=90/95 (a.k.a *Modern Fortran*) features, with some side notes on `FORTRAN 77` (mostly for contrast). This talk is not intended to be exhaustive in its coverage of Fortran syntax, but should be enough to yield a "rough and ready" knowledge for HPC.

Source Code Format

While `FORTRAN 77` used by default a **fixed** format for its source code, the default with modern Fortran is now a **free** format, in which lines can be up to 132 columns long, with a maximum of 39 **continuation** lines, indicated by the & character.

Source Format Summary

```
132   characters per line
!     initiates comment
&     continuation character
;     statement separator (multiple per line)
```

```
print*,''You need two continuation &
        &characters when splitting a token &
        & or string across lines''
```


Statement Labels

Statement labels are still in fashion, and consist of 1-5 digits (leading zeros are neglected), e.g.

```
101  WRITE(6,*) 'Enter runid:(<=9chars) '
     READ(*,2) runid
2    format(a9)
```

Most commonly used for `FORMAT` statements.

Names

In Fortran, names must:

- Start with a letter (a-z)
- Contain only letters, digits, and underscore
- Must not be longer than 31 characters

More Resources

Good reference material on all things Fortran:

- M. Metcalf and J. Reid, *Fortran 90/95 Explained, 2nd Ed.*, (Oxford, Great Britain, 2000)
  Fortran 95/2003 Explained, 3rd Ed., (Oxford, Great Britain, 2004), with M. Cohen
  Modern Fortran Explained, (Oxford, Great Britain, 2011), with M. Cohen
- Metcalf's online tutorial:

  http://wwwasdoc.web.cern.ch/wwwasdoc/WWW/f90/f90.html

- Alan Miller's Fortran Resources:

  http://jblevins.org/mirror/amiller/

Implicit Types

Unfortunately FORTRAN historically allowed **implicit types**, which by default were:

FORTRAN 77

implicit real (a-h,o-z)
implicit integer (i-n)

Thanks to backward compatibility, this feature is still present:

Fortran

implicit type (letter-list) [,type (letter-list) ...]

Implicit None

I would recommend starting every declaration block with the following:

Fortran

implicit none

which will turn all implicit typing off, and save a lot of potential errors (misspelled variable names just being one of them). Use implicit types at your own risk!

Basic Declarations

Declarative Syntax

<type> [,<attribute-list>] :: <variable-list> & [=<value>]

The attribute is usually one of PARAMETER, DIMENSION, or POINTER.

Fortran Intrinsic Types

We have five different intrinsic types, integer, real, logical, complex, and character. The syntax looks like:

```
integer :: i
real :: x
complex :: z
logical :: beauty
character :: c
```

These are all of default **kind** ... (Note that integers can also be represented in binary (base 2), octal (base 8), and hex (base 16))

Kind

There is an intrinsic function `KIND` that returns an integer and can be used to query (or set) the “kind type” of a variable

Fortran

KIND(x) returns default integer whose value is the kind type parameter value of x

Example: Declarative Use of KIND

These are two of the very handy “transformational” functions:

SELECTED_REAL_KIND/SELECTED_INT_KIND

SELECTED_REAL_KIND([p][,r])

- p :: integer, decimal precision (intrinsic PRECISION)
- r :: integer, decimal exponent range (RANGE)
  (at least one of p or r must be present)

SELECTED_INT_KIND(r)

- r :: integer, decimal exponent range

the return values of both functions are default integers whose values can be used in the `KIND` intrinsic function. -1 is returned if the desired precision is unavailable (-2 for the range in `SELECTED_REAL_KIND`, -3 if both).

Best illustrated by example - this is a chunk of code that use in all of my Fortran codes:

```
integer,parameter :: si=KIND(1),sp=KIND(1.0),sc=KIND((1.0,1.0)),  &
        di=SELECTED_INT_KIND(2*RANGE(1_si)), &
        dp=SELECTED_REAL_KIND(2*PRECISION(1.0_sp)), &
        dc=SELECTED_REAL_KIND(2*PRECISION(1.0_sc))
```

Note the use of underscores in literal constants to indicate kind type, which is quite generally applicable:

```
real(kind=dp) :: a,b,c
complex(kind=dc) :: zi=(0.0_dp,1.0_dp)
```


PARAMETER Attribute

Instead of the old `FORTRAN 77 PARAMETER` statement, one can use the parameter attribute:

```
REAL, PARAMETER :: E = 2.71828, PI = 3.141592
```

Intrinsic Type Arrays

Arrays can be indicated by the old-type `DIMENSION` attribute, or simply in the declaration itself:

```
real,dimension(3) :: a
real :: b(3)
real :: c(-1:1)
```

are all arrays of **rank** 1, **shape** 3.

Language Elements Intrinsic Types

Note that arrays can be **sectioned**:

```
a(i:j)        ! rank 1 array, size j-i+1
b(k,1:n)      ! rank 1 array, size n
c(1:m,1:n,k)  ! rank 2 array, extent m by n
```

and **vector subscripts** can be used:

```
x(ivector)    ! ivector is an integer array
```

which makes for a very flexible (essentially arbitrary) indexing scheme.

Language Elements Intrinsic Types

Initializing Arrays

There is some special syntax for initialization arrays:

```
a(1:4) = 0.0
a(1:4) = (/1.1, 1.2, 1.3, 1.4/)
a(1:4) = (/ i, i=1,7,2 /)          ! implied do loop with stride 2
```

Note that the `RESHAPE` function can also be used to initialize an array of rank greater than 1.

Language Elements Intrinsic Types

Character Arrays

Better known as **strings**, Fortran is not the best for string handling (it is intended for number crunching, after all), but modern Fortran is considerably improved in this regard:

```
character,dimension(120) :: line1
character(len=120) :: line2
character(len=120),dimension(80) :: page
```

Substrings can be handled in several ways:

```
character(len=1200) :: line
character(len=12) :: words,term*8   ! term has length 8, alternate length spec
line(:i)                            ! same as line(1:i)
line(i:)                            ! same as line(i:120)
line(:)                             ! same as line(1:120)
```

and we will come back later to some of the intrinsic string handling functions and subroutines.

Derived Types

It is often advantageous to define your own data types:

```
TYPE Coords_3D
    real :: r,theta,phi    ! spherical coordinates
END TYPE Coords_3D

TYPE(Coords_3D) :: point1, point2
```

You can reference the internal components of a structure:

```
x1 = point1%r * SIN(point1%theta) * COS(point1%phi)
y1 = point1%r * SIN(point1%theta) * SIN(point1%phi)
z1 = point1%r * COS(point1%theta)
x2 = point2%r * SIN(point2%theta) * COS(point2%phi)
y2 = point2%r * SIN(point2%theta) * SIN(point2%phi)
z2 = point2%r * COS(point2%theta)
```

N.B. Derived type components can not be `ALLOCATABLE` (but they can be `POINTERS`).

Assigning Derived Types

Derived types can be assigned either by component or using a **constructor**:

```
point1%r = 1.0
point1%theta = 0.0
point1%phi = 0.0
point1 = Coords_3D(1.0,0.0,0.0)
point2 = point1
```

Note that two variables of the same type can be handled very simply. Derived types should always be placed in a **MODULE**.

Pointers

Fortran finally has the capability (often best left unused) of using pointers. Along with `ALLOCATABLE` and automatic arrays, pointers make up the 3 types of dynamic data in Fortran 90/95. Pointers can be members of derived types, most useful for implementation of linked lists:

```
real,pointer,dimension(:,:) :: a
real,pointer :: x,y
!
!
NULLIFY(x,y,a)               ! Point to nothing
x => null()                  ! Fortran 95 only
ALLOCATE(x,y,a(100:100))     ! Fresh storage allocation
```

We will talk a bit more about pointers later when dealing with other aspects of dynamic memory.

`DATA` Statement

The `DATA` statement can be used to initialize values:

```
DATA object-list /value-list/ [[,] object-list /value-list/] ...
```

Values initialized in `DATA` statements automatically have the `SAVE` attribute.

```
real :: a,b,c
integer :: i,j,k
DATA a,b,c/1.0,2.0,3.0/,i,j,k/3*0/
real :: diag(100,100)
! sets diagonal elements by implied do
DATA (diag(i,i), i=1,100) / 100*5.0 /
! sets the upper and lower triangles to 0, diagonal to 1
DATA ((matA(i,j),matA(j,i),j=i+1,100),i=1,100)/ 9900*0.0 /
DATA (matA(i,i),i=1,100)/ 100*1.0 /
```

Scalar Expressions

Scalar expressions use operators $**, *, /, +, -$. The following table summarizes the result `KIND` for all but exponentiation ($**$)

a	b	used(a)	used(b)	result
I	I	a	b	I
I	R	`real(a,kind(b))`	b	R
I	C	`cmplx(a,0,kind(b))`	b	C
R	I	a	`real(b,kind(a))`	R
R	R	a	b	R
R	C	`cmplx(a,0,kind(b))`	b	C
C	I	a	`cmplx(b,0,kind(a))`	C
C	R	a	`cmplx(b,0,kind(a))`	C
C	C	a	b	C

(I = integer, R = real, C = complex)

and for exponentiation ($**$):

a	b	used(a)	used(b)	result
I	I	a	b	I
I	R	`real(a,kind(b))`	b	R
I	C	`cmplx(a,0,kind(b))`	b	C
R	I	a	b	R
R	R	a	b	R
R	C	`cmplx(a,0,kind(b))`	b	C
C	I	a	b	C
C	R	a	`cmplx(b,0,kind(a))`	C
C	C	a	b	C

(I = integer, R = real, C = complex)

Relational Operators

Fortran supports the old-style `FORTRAN 77` relational operators as well as more modern syntax (which most compilers were supporting by extension anyway):

`77`	Modern
.lt.	<
.le.	<=
.eq.	==
.ne.	/=
.gt.	>
.ge.	>=

Logical Operators

The logical operators are simply:

.not.	logical negation
.and.	logical intersection
.or.	logical union
.eqv.	logical equivalence
.neqv.	logical non-equivalence

Character Operations

Substrings are easy using the same syntax as for arrays:

```
character(len=*),parameter :: alphabet1='abcdefghijklm' &
                              alphabet2='nopqrstuvwxyz'
character(len=*),parameter :: numerals='0123456789'

print*, ``First four letters: ``,alphabet1(1:4)
print*, ``alphanumeric: ``,alphabet1//alphabet2//numerals
```

where we also made use of the "//" concatenation operator.

Array Assignment

Arrays can be assigned using expressions that involve arrays of the same shape, or by scalar (in which case the scalar is applied to all elements of the array), e.g.

```
real :: a(20,20),b(10)

a(1,11:20) = b
a = a + 1.0
```

This flexible syntax makes for much more compact code.

IF Construct

```
[name:] if (expr) then
           block
        [else if (expr) then [name]
           block] ...
        [else [name]
           block]
        end if [name]
```

Note that the optional name is only for program clarity (especially for nested loops), and needs to be unique. Also note the single statement variation:

```
if (expr) single-statement
```


DO Loops

Old style loop syntax:

```
[name:] do var=expr1,expr2[,expr3]
           block
        end do [name]
```

where the loop is executed MAX(0,(expr2-expr1+expr3)/expr3) times, and var is a named scalar integer variable, and expr1, expr2, and expr3 are scalar integer expressions (expr3 must be nonzero if used).

Infinite DO Loop

One important variation on the DO construct is the endless do loop, which has a simple means to exit:

```
do
  i = i+1
  if (i >= enough) exit
end do
```

Note that the `exit` statment is also handy for early termination in bounded loops. The `cycle` statement is similar, but just drops execution to the next iteration.

DO−WHILE Loop?

There is a DO−WHILE construct, but with the `exit` option for early loop termination, it is not really worth talking about ...

```
DO WHILE (a /= b)
  ...
END DO
```

is exactly the same as

```
DO
   if (a == b) exit
   ...
END DO
```


The GOTO Statement

Ah, the bane of FORTRAN 77 programmers everywhere - the infamous GOTO statement. Still supported in Fortran 90/95, be wary of using the GOTO. If you have to use it, make it as clear as you possibly can.

```
 101 WRITE(6,*) 'Enter runid:(<=9chars) '
     READ(*,'(a9)') runid
     i=INDEX(runid,' ')-1
     INQUIRE(file=runid(1:i)//".in",exist=ifex)
     if(.not.ifex) goto 101
     OPEN(unit=iunit,file=runid(1:i)//'.in',status='old', &
    &      form='formatted')
```


Using Named and Nested Loops

Here is an example of when using named loops comes in handy:

```
outa: DO
  inna: DO
    ...
    IF (a.GT.b) EXIT outa  ! jump to line 10
    IF (a.EQ.b) CYCLE outa ! jump to line 1
    IF (c.GT.d) EXIT inna  ! jump to line 9
    IF (c.EQ.a) CYCLE inna ! jump to line 2
  END DO inna
END DO outa
```

The CASE Construct

Somewhat like the C `switch` statement ...

```
[name:] select case (expr)
        [case selector [name]
           block] ...
        end select [name]
```

The `CASE` construct can be used as an efficient substitute for a more elaborate `IF ... THEN ... ELSEIF ... ELSE ... END IF` construct.

An example of using `CASE` when parsing an input file:

```
      select case (cterms(1))
      case ("MODEL")
         select case (cterms(2))
           case ("XBMC","xbmc")
              MCflg=.true.
           case ("NRL","nrl")
           case ("NN2","nn2")
           case default
         end select
      ...
```


Part II

Fortran Essentials

Modules

Modules provide a way to package together commonly used code (similar to the old `common` blocks in `FORTRAN 77`) and have distinct advantages:

- Can be used to hide internal data and routines through `PRIVATE` and `PUBLIC` declarations
- Can contain common subroutines and functions with explicit interfaces which can be changed without affecting calling code
- Modules can (and often should) be compiled separately, **before** the program units that use them

Module Syntax

```
module module-name
  [specification statements]
 [contains
   module-subprograms]
end [module [module-name]]
```


and a very simple example:

```
MODULE TBconst
   implicit none
   integer,parameter :: sp=KIND(1.0),                          &
       & dp=SELECTED_REAL_KIND(2*PRECISION(1.0_sp)),     &
       & sc=KIND((1.0,1.0)),                             &
       & dc=SELECTED_REAL_KIND(2*PRECISION(1.0_sc))
  real(kind=dp),parameter :: pi=3.14159265358979323846264 3_dp
END MODULE TBconst
```


Using Modules

Modules encapsulate code that can be made accessible to other program units through the `USE` statement:

```
MODULE TBatoms
  USE TBconst
...
```

Modules are free to load other modules, but not themselves.

Module Data Visibility

You can allow or prevent access to the internal workings of a module using the `PRIVATE` and `PUBLIC` attributes:

```
    PRIVATE :: pos, store, stack_size  ! hidden
    PUBLIC  :: pop, push               ! not hidden
```

or

```
    PUBLIC                    ! set default visibility
    INTEGER, PRIVATE, SAVE :: store(stack_size), pos
    INTEGER, PRIVATE, PARAMETER :: stack_size = 100
```

Renaming/`USE ONLY`

You can rename a module entity in a local context:

```
USE TBconst, local_dp => dp   ! dp becomes local_dp in current scope
```

or you can restrict access by

```
USE TBconst, ONLY:dp,dc       ! only load dp and dc from module TBconst
```


Module Summary

```
MODULE modname
 ... type defs
 ... Global data
 ...
 CONTAINS
  SUBROUTINE sub_1
   ...
  CONTAINS
    SUBROUTINE internal_1
     ...
    END SUBROUTINE internal_1
    SUBROUTINE internal_2
     ...
    END SUBROUTINE internal_2
  END SUBROUTINE sub_1
  ...
  FUNCTION fun_1
   ...
  CONTAINS
   ...
  END FUNCTION fun_1
END MODULE modname
```


Subroutine Syntax

The syntax for a subroutine call is given by

SUBROUTINE

[recursive] subroutine *subroutine-name*[([dummy arguments])]

Argument Intent

You can provide information to the compiler (always a good idea!) about the dummy arguments to a routine by:

```
REAL, INTENT(IN) :: arg1       ! passing in value
REAL, INTENT(OUT) :: arg2      ! returning value only
REAL, INTENT(INOUT) :: arg3    ! both apply
```

You should always make use of the `INTENT` attribute - it allows the compiler to do extensive error checking and optimization (remember, the more that your compilers knows about your code, the better it will be able to perform).

SAVE Attribute

The SAVE attribute can be applied to a specified entity, or all of the local entities in a procedure:

```
SUBROUTINE sub_1(arg1,arg2)
   integer,save :: number_calls = 0
   ...
   number_calls = number_calls +1
```

```
SUBROUTINE sub_1(arg1,arg2)
 ...
 SAVE
 ...
```

SAVE acts to preserve values between calls to the subroutine/function.

Function Call Syntax

The syntax for a function call is given by

FUNCTION

type [recursive] function *function-name*[([dummy arguments])] & [result(result-name)]

Explicit Interfaces

External subprograms have an implicit interface by default (even if one uses the external statement to indicate that a subunit is outside the current code, the arguments and their types remain unspecified), and an INTERFACE block is necessary to specify an explicit interface of an external subprogram; as mentioned above, this allows type-checking of actual and formal arguments in a reference to a subprogram

Explicit Interface Example

```
INTERFACE
   SUBROUTINE resid(m, n, x, fvec, iflag)
     USE TBatoms
     USE TBconst
     USE TBfitdata
     USE TBflags
     USE TBmat
     USE TBopt
     USE TBparams

     implicit none
     integer, intent(in) :: m,n
     integer, intent(in out) :: iflag
     real(kind=dp), intent(in) :: x(n)
     real(kind=dp), intent(in out) :: fvec(m)
   END SUBROUTINE resid
END INTERFACE
```

Note that the syntax of the interface-body is just an exact copy of the subprogram's header, argument specifications, function result, and END statement.

INTERFACE Properties

- Can not use both EXTERNAL and INTERFACE
- Explicit interfaces required for POINTER or TARGET dummy arguments in a procedure, or pointer-valued function result
- Explicit interfaces also required for OPTIONAL, KEYWORD, and procedural arguments
- Even when not required, explicit interfaces are a good idea (usually placed inside a MODULE)

Optional & Keyword Arguments

Dummy arguments can be optional - using OPTIONAL:

```
SUBROUTINE optargs(a,b)
  REAL, INTENT(IN), OPTIONAL    :: a
  INTEGER, INTENT(IN), OPTIONAL :: b
  REAL :: ay; INTEGER :: by

  ay = 1.0; bee = 1 ! defaults
  IF(PRESENT(a)) ay  = a
  IF(PRESENT(b)) bee = b
     ...
```

```
CALL optargs()
CALL optargs(1.0,1); CALL optargs(b=1,a=1.0)  ! same, using keywords
CALL optargs(1.0);   CALL optargs(a=1.0)      ! keywords handier still for long lists
CALL optargs(b=1)
```

Note that optional and keyword arguments need explicit interfaces, and should come after positional arguments.

Procedures as Arguments

When using a procedure as an argument, an explicit interface is required (as it is for POINTER, optional, and keyword arguments):

```
REAL FUNCTION minimum(a, b, func)
! returns the minimum value of the function func(x)
! in the interval (a,b)
   REAL, INTENT(in) :: a, b
   INTERFACE
      REAL FUNCTION func(x)
         REAL, INTENT(IN) :: x
      END FUNCTION func
   END INTERFACE
   REAL f,x
   :
   f = func(x)   ! invocation of the user function.
   :
END FUNCTION minimum
```


Recursion

Recursion is supported in Fortran >=90/95; we can illustrate it best by example:

```
RECURSIVE FUNCTION factorial(n) RESULT(res)
   INTEGER res, n
   IF(n.EQ.1) THEN
      res = 1
   ELSE
      res = n*factorial(n-1)
   END IF
END FUNCTION factorial
```

this would be an example of **direct** recursion ...

and an example of **indirect** recursion:

```
volume = integrate(func, bounds)
:
RECURSIVE FUNCTION integrate(f, bounds)
! Integrate f(x) from bounds(1) to bounds(2)
  REAL integrate
  INTERFACE
     FUNCTION f(x)
        REAL f, x
     END FUNCTION f
  END INTERFACE
  REAL, DIMENSION(2), INTENT(IN) :: bounds
  :
END FUNCTION integrate
:
FUNCTION func(x)
  USE MODfunc        ! module MODfunc contains function f
  REAL func, x
     xval = x
     func = integrate(f, bounds)
END FUNCTION func
:
```

Generally indirect recursion is of the form 'A calls B calls A ...' - in this case `integrate` calls `func` which calls `integrate` ...

Fortran I/O

Too large a topic to cover in its entirety here - we will focus on the basics required to familiarize you with basic Fortran I/O functionality.

OPEN

```
OPEN ([UNIT=integer,] FILE=filename, [ERR=label,] &
      [STATUS=status,] [ACCESS=method,] [ACTION=mode,] &
      [RECL=int-expr)
```

- `filename` is a string
- `status` is 'OLD', 'NEW', 'REPLACE', 'SCRATCH' or 'UNKNOWN'
- `method` is 'DIRECT' or 'SEQUENTIAL'
- `mode` is 'READ', 'WRITE' or 'READWRITE'
- RECL (record length) needs to be specified for DIRECT access files

OPEN Examples

Note that UNIT 1-7 are typically reserved (6, or $*$, is almost always the standard output, for example), and each file stream needs a unique number.

```
OPEN(17,FILE='output.dat',ERR=10, STATUS='REPLACE', &
     ACCESS='SEQUENTIAL',ACTION='WRITE')
:
OPEN(14,FILE='input.dat',ERR=10, STATUS='OLD', RECL=iexp, &
     ACCESS='DIRECT',ACTION='READ')
:
write(unit=iounit,fmt="(a)",iostat=badunit) title
if (badunit > 0) call error_handler(ERR_IO)
```

INQUIRE

A very handy statement for querying the status of a file by unit number or filename:

```
INQUIRE ([UNIT=]unit | FILE=filename, ilist)
```

and there are many possible entries in `ilist`, of which the most handy are:

- `IOSTAT=ios` as in the `OPEN` syntax
- `EXIST=log_exist` returns a logical on the existence of the file
- `OPENED=log_opened` returns a logical on whether the file is open

INQUIRE Example

```
!
! Open input file - fail gracefully if not found.
!
  INQUIRE(file=runid(1:len_runid)//'.in',exist=exist_in)
  if (.not.exist_in) then
     write(*,*) '<readin> Unable to open input file: ', &
                 runid(1:len_runid)//'.in'
     stop
  endif
  OPEN(file=runid(1:len_runid)//'.in',status='old',unit=inunit)
```


Other I/O Statements

- **CLOSE** unattach specified unit number
- **REWIND** place file pointer back to start of file
- **BACKSPACE** place file pointer back one record
- **ENDFILE** force writing end-of-file

```
REWIND(11)
BACKSPACE(UNIT=27)
ENDFILE(19)
CLOSE(13,IOSTAT=io_value)
```


READ

```
READ ([UNIT=unit,] [FMT=format,] [IOSTAT=int-variable,] &
      [ERR=label,] [END=label,] [EOR=label,] &
      [ADVANCE=adv-mode,] [REC=int-expr,]  &
      [SIZE=num-chars]) <output-list>
```

where the non-obvious entries are:

- `unit` is an integer (some lower values are reserved) or $*$ for standard input
- `format` is a string of `FORMAT` statement label number
- `label` is a statement label
- `adv-mode` is `'YES'` or `'NO'`
- `IOSTAT` returns zero for no error

READ Example

```fortran
READ(14,FMT='(3(F10.7,1x))',REC=iexp) a,b,c
READ(14,'(3(F10.7,1x))',REC=iexp)     a,b,c    ! same as above
READ(*,'(A)',ADVANCE='NO',EOR=12,SIZE=nch) str
```


WRITE

```fortran
WRITE ([UNIT=unit,] [FMT=format,] [IOSTAT=int-variable,] &
       [ERR=label,] [ADVANCE=adv-mode,] &
       [REC=int-expr) <output-list>
```

where the entries are as in the `READ` case.

FORMAT

Fortran does have quite an elaborate formatting system. Here are the highlights.

Iw	w chars of integer data
Fw.d	w chars of real data, d decimal places
Ew.d	w chars of real data, d decimal places
Lw	w chars of logical data
Aw	w chars of CHARACTER data
n**X**	skip n spaces

Note that:

- The **E** descriptor is just the **F** with scientific notation

FORMAT Example

```fortran
WRITE(*,FMT='(2X,2(I4,1X),''name '',A4,F13.5,1X,E13.5)') &
     77778,3,'abcdefghi',14.45,14.5666666
```

```
 ****    3 name abcd      14.45000    0.14567E+02
```

Note that are quite a few other format descriptors, much less commonly used.

Unformatted I/O

Unformatted I/O is simpler (no `FORMAT` statements) and involves less overhead, less chance of roundoff error), but is inherently non-portable, since it relies on the detailed numerical representation. A file must be either entirely formatted or unformatted, not a mix of the two.

```
READ(14) A
WRITE(15,IOSTAT=ios,ERR=2001) B
```

Note that unformatted i/o is generally quite a lot faster than formatted - so unless you are concerned with moving your files from one platform to another, you will be much better off using unformatted i/o.

Assumed-shape Arrays

Arrays passed as dummy arguments should generally be what are called **assumed-shape** arrays, meaning that the dimensions are left to the actual (calling arguments):

```
SUBROUTINE stubby(a,b)
  implicit none
  real,intent(in) :: a(:),b(:,:)
  :
```

- Note that the default bounds (1) apply
- Actual arguments can not be vector subscripted or themselves assumed-shape

Automatic Arrays

Local arrays whose extent is determined by dummy arguments are called **automatic objects**. Example:

```
SUBROUTINE stubby1(b,m,n)
  integer, intent(in) :: m,n
  real, intent(inout) :: b(:,:)    ! assumed
  REAL :: b1(m,n)                  ! automatic
  REAL :: b2(SIZE(b,1),SIZE(b,2)) ! automatic
```

Note that both assumed-shape arrays and automatic objects are likely to be placed on the **stack** in terms of memory storage.

Allocatable Arrays

Finally, dynamic data storage elements for Fortran! An array that is **not** a dummy argument or function can be given the `ALLOCATABLE` attribute:

```
real, allocatable :: a(:,:)
:
:
ALLOCATE(a(ntypes,0:ntypes+2)) ! ntypes is an integer
:
! lots of work
:
DEALLOCATE(a)
```

Note that originally `ALLOCATABLE` arrays could not be part of a derived type (have to use a `POINTER` to get the same functionality) - that oversight was fixed in Fortran 2003.

ALLOCATE & DEALLOCATE

The syntax for `ALLOCATE` and `DEALLOCATE` are given by:

```
ALLOCATE( list [,stat=istat] )
DEALLOCATE( list [,stat=istat] )
```

The optional `stat=` specifier can be used to test the success of the (de)allocation through the scalar integer `istat`. As usual, zero for success. Leaving out "`stat=`" should result in a termination if the (de)allocation was unsuccessful.

Pointer Flexibility

Note that you can not associate pointers with just any variable (as in C), instead the variables must be declared using the `target` attribute:

```
real, target :: x, y(100), z(4,4)
integer, target :: m, n(10), k(10,10)

real, pointer :: ptr1, ptr2, ptr_y(:), ptr_z1(:), ptr_z2(:,:)

ptr1 => x              ! simple pointer assignment
alpha = exp(ptr1)      ! pointer shares memory location with x,
                       ! but used like any value
nullify(ptr1)          ! frees up pointer

if (associated(ptr1)) then
   print*, 'ptr1 is associated'
   if (associated(ptr1, target=x)) then
      print*, 'ptr1 is associated with "x"'
   endif
endif
```


More Fun With Pointers

array-associated pointers have considerable flexibility:

```
ptr_y => y              ! can use ptr_y(i) just as y(i)
ptr_y => y(11:20)       ! ptr_y(1) is now y(11) ...
ptr_y => z(2,1:4)       ! loads row 2 of z intro ptr_y(:)
ptr_z2 => z(2:4,2:4)    ! ptr_z2 is 3x3 submatrix of z
ALLOCATE(ptr_z1(16))    ! direct allocation
```

Note that pointers can also be used as components of derived types, making for very flexible data structures.

Pointer Considerations

Some things to think about when using pointers in Fortran:

- Can create complex (and difficult to maintain) code
- Easy to create bugs that only arise at run-time
- Inhibit compiler optimization (difficult to predict data patters and disjoint memory structures)

Elemental Operations

We have already seen **elemental** operations in which conformable operands can be used with intrinsic operators, e.g.

```
real :: a(100,100)
...
a = SQRT( a )
```

will apply the square root operator individually to all of the elements of `a`. Not only intrinsics can be elemental, and you can also use the `elemental` declaration in user-defined functions (Requires Fortran >=95) as well.

Array-valued Functions

Functions can return arrays - just be careful that you ensure that the interface is an explicit one. An example:

```
PROGRAM arrfunc
  implicit none
  integer,parameter :: ndim=36
  integer,dimension(ndim,ndim) :: m1, m2
  :
  m2 = funky(m1,4)
  :
  CONTAINS
  FUNCTION funky(m,scal)
    integer, intent(in) :: ima(:,:)
    integer, intent(in) :: scal
    integer :: funky(SIZE(m,1),SIZE(m,2))
    funky(:,:) = m(:,:)*scal
  END FUNCTION funky
END PROGRAM arrfunc
```


`WHERE` Statement

Useful for performing array operations only on certain elements of an array, but preserving the compact syntax:

```
WHERE (logical-array-expr)
  array-assignments
END WHERE
```

```
WHERE (pressure <= 1.0)
  pressure = pressure + increment
ELSEWHERE
  pressure = pressure + check_pressure
END WHERE
```

in the example all arrays are of the same shape, and the assignment is said to be *masked* using the comparison (which is done element by element). **Fortran >=95** also provides for masks in additional `ELSEWHERE` statements.

`FORALL` Construct (Fortran 95)

Fortran 95 introduced the concept of a `FORALL` statement which is basically an array assignment with some explicit indexing:

```
FORALL(i = 1:M) grid(i,i) = diag(i)
FORALL(i = 1:M, j = 1:N) grid(i,j) = u(i)*v(j)
FORALL(i = 1:M, j = 1:N, eps(i.j) /= 0.0) grid(i,j) = 1.0/eps(i,j)
```

Implied is that the assignment is trivially data-parallel, i.e. it can be carried out in any order, and therefore can be more efficient than a more traditional loop. Construct form:

```
FORALL(i = 2:N-1, j = 2:N-1)
  U(i,j) = U(i,j-1) + U(i,j+1) + U(i-1,j) + U(i+1,j)
  V(i,j) = U(i,j)
END FORALL
```

where the results are executed in any order, held in temporary storage (to avoid indeterminate results), and then updated in arbitrary order.

Complete `FORALL` construct syntax:

```
[name:] FORALL(index=lower:upper[:stride] [,index2=lower2:upper2[:stride2]] ... &
                 [,scalar-logical-expr])
   [body]
END FORALL [name]
```

The *body* of a `FORALL` construct can be quite general (containing statements, additional `FORALL` or `WHERE` statements/constructs, etc.), but must not branch (e.g. `goto`) out of the construct. Any included subprograms must be **pure**, in the sense of inducing no undesired side-effects in the sense of inducing an order dependence that would impede parallel processing.

`PURE` Procedures (Fortran >=95)

Programmer can assert that a function or subroutine is `PURE`: by adding the `PURE` keyword to the function/subroutine statement:

- a pure function does not alter its dummy arguments (must be `INTENT(IN)`)
- `INTENT` of dummy arguments must be declared (`IN` for functions)
- does not alter variables accessed by host or use association
- contains no local variables with `SAVE` attribute
- contains no operations on external file
- contains no `STOP` statements
- any internal procedures must also be pure

all intrinsic functions are pure.

Array Intrinsics

Fortran is designed around the notion of data manipulation, so it is not a great surprise that it has a number of built-in functions for array manipulation, some of which we have already seen (elemental operations, masking).

Array Functions

Set of operations that involve common extractions from arrays:

ALL(MASK [,dim])	all relations in mask are true [along dimension dim]
ANY(MASK [,dim])	if any elements of mask are true [along dimension dim]
COUNT(MASK [,dim])	number of elements of mask that are true
MAXLOC(ARRAY [,mask])	location of element with maximum value
MINLOC(ARRAY [,dim[,mask]])	location of element with minimum value
MAXVAL(ARRAY [,dim[,mask]])	maximum value [of true elements in mask, along dim]
MINVAL(ARRAY [,dim[,mask]])	minimum value [of true elements in mask, along dim]
PRODUCT(ARRAY [,dim[,mask]])	products [of true elements in mask, along dim] of values
SUM (ARRAY [,dim[,mask]])	sum [of true elements in mask, along dim] of values

Example of Array Extraction Intrinsics

```fortran
integer :: i,j,max_element(2),max_element_2(2)
real :: Amax,Amax_2,Amat(100,100)

Amat = RESHAPE( (/ ((100*(i-1)+j,j=1,100),i=1,100) /), (/ 100, 100 /), ORDER=(/2,1/) )

max_element = MAXLOC( Amat )           ! finds the element of Amax with
                                       ! max value [ A(100,100) =
Amax = MAXVAL( Amat )                  !                     10000 ]

max_element_2 = MAXLOC( Amat, Amat<66) ! finds element in Amax
                                       ! that's < 66 [ A(1,65) =
Amax_2 = MAXVAL( Amat, Amat<66 )       !                    65 ]
```


Array Inquiry Functions

Very useful array inquiry functions:

ALLOCATED(ARRAY)	logical if A has been allocated
LBOUND(ARRAY [,dim])	lower bound for dimension dim of A (integer vector if no dim)
SHAPE(ARRAY)	returns integer vector of shape of A
SIZE(ARRAY [,dim])	size of dimension dim of A (else all of A)
UBOUND(ARRAY [,dim])	upper bound for dimension dim of A (integer vector if no dim)

Array Reshaping

`RESHAPE` is a general intrinsic function which delivers an array of a specific shape:

```fortran
RESHAPE(source, shape [,pad][,order])
```

returns an array whose shape is given by the constant rank-1 integer array (nonnegative elements) `shape` derived from the array `source`. If `order` is absent, elements of `pad` are used to fill out remaining elements in the result (whose size may then exceed that of `source`. `order` can be used to pad the result in non-element order.

```fortran
real :: A2(2,2)
A2 = RESHAPE( (/1,2,3,4/),(/2,2/) )       ! create a 2x2 matrix from 1x4 vector
print*, 'A2: 1,1 1,2 = ',A2(1,1),A2(1,2)
print*, 'A2: 2,1 2,2 = ',A2(2,1),A2(2,2)
```

produces (recall that Fortran is column-ordered):

```
 A2: 1,1 1,2 =     1.000000        3.000000
 A2: 2,1 2,2 =     2.000000        4.000000
```


Vector/Matrix Intrinsics

Fortran 90 has several intrinsics for vector dot products matrix multiplication and transposition:

DOT_PRODUCT(vector_1,vector_2)	dot product of two rank-1 equal length vectors
MATMUL(matrix_1,matrix_2)	matrix multiplication
TRANSPOSE(matrix)	transposition of any rank-2 array

Intrinsics Categories

There are roughly 75 new intrinsic routines (versus `FORTRAN 77`), but they roughly fall into 4 categories:

1. Elemental procedures
2. Inquiry functions
3. Transformational functions
4. Nonelemental subroutines

I am going to group them a bit differently, and only cover the more common ones. Consult a good reference[1] for a thorough list.

[1] Metcalf, Reid, and Cohen*Modern Fortran Explained*, (Oxford, Great Britain, 2011).

Mathematical Intrinsics

Far too many to enumerate here - you can find a handy reference for the full set in other references. Note that almost all support a generic interface supporting available `KIND` types, and that most are elemental.

`SQRT, EXP, LOG, LOG10, SIN, COS, TAN, ASIN, ACOS, ATAN, SINH, COSH, TANH`

Numerical Intrinsics

The list of numerical intrinsics (with syntax and usage):

INT(a[,KIND])	convert to integer, type KIND
REAL(a[,KIND])	convert to real, type KIND
CMPLX(x[,y][,KIND])	convert x or (x,y) to complex, type KIND
AINT(a[,KIND])	truncate real to lowest whole number, type KIND
ANINT(a[,KIND])	returns nearest whole number real, type KIND
NINT(a[,KIND])	integer (type KIND) value nearest a
ABS(a)	absolute value of a, same KIND as a
MOD(a,p)	remainder of a modulo p, a-int(a/p)*p (has sign of a)
MODULO(a,p)	a-floor(a/p)*p (has sign of p)
FLOOR(a[,kind])	greatest integer less than or equal to a, type KIND
CEILING(a[,KIND])	least integer greater than or equal to a, type KIND
SIGN(a,b)	absolute value of a times sign of b
DIM(x,y)	max(x-y,0.0)
MAX(a1,a2[,a3,...])	maximum of two or more numbers
MIN(a1,a2[,a3,...])	minimum of two or more numbers
AIMAG(z)	imaginary part of complex number z, type real, KIND(z)
CONJG(z)	conjugate of complex number z

String Functions

Yes, Fortran does have string handling capability! And in fact, it is much improved. The following table gives a brief synopsis:

ACHAR(I)	ASCII character of number I
ADJUSTL(STRING)	Adjusts to the left
ADJUSTR(STRING)	Adjusts to the right
CHAR(I, kind)	Returns character of number I
IACHAR(C)	ASCII number of char C
ICHAR(C)	Number of char C
INDEX(STRING, SUBSTRING, back)	Starting pos of substring in string
LEN(STRING)	Length of STRING
LEN_TRIM(STRING)	Length of string without trailing blanks
REPEAT(STRING, NCOPIES)	String concatnation
SCAN(STRING, SET, back)	Position of 1st occurrence of any char in SET in STRING
TRIM(STRING)	Returns string without trailing blanks
VERIFY(STRING, SET, back)	Position of 1st char in STRING not in SET

The following functions can be used for ASCII lexical string comparisons:

```
LGE(STRING_A, STRING_B)
LGT(STRING_A, STRING_B)
LLE(STRING_A, STRING_B)
LLT(STRING_A, STRING_B)
```

Note that if the strings are of differing length, the shorter will be padded with blanks for comparative purposes. All return default logical results.

Bitwise Intrinsics

Modern Fortran added support for quite a few bitwise operations:

BIT_SIZE(I)	number of bits in a word
BTEST(I, POS)	.true. if POS number of I is 1
IAND(I, J)	logical addition of bit chars in I and J
IBCLR(I, POS)	puts a zero in the bit in POS
IBITS(I, POS, LEN)	uses LEN bits of word I beginning at POS, additional bits are set to zero. POS + LEN <= BIT_SIZE(I)
IBSET(I, POS)	puts the bit in position POS to 1
IEOR(I, J)	performs logical exclusive OR
IOR(I, J)	performs logical OR
ISHIFT(I, SHIFT)	performs logical shift (to the right if the number of steps SHIFT < 0 and to the left if SHIFT > 0). Positions that are vacated are set to zero.
ISHIFTC(I, SHIFT, size)	performs logical shift a number of steps circularly to the right if SHIFT < 0, circularly to the left if SHIFT > 0. If SIZE is given, it is required that 0 < SIZE <= BIT_SIZE(I)
NOT(I)	logical complement

Random Numbers

Modern Fortran also has a built-in pseudorandom generator:

```
CALL RANDOM_NUMBER(harvest)
CALL RANDOM_SEED([size] | [put] | [get])
```

`harvest` can be an array, and the range of the random numbers are the interval $[0,1)$. `size` is intent OUT and returns the size of the integer seed array, which can be input (`put`) or returned (`get`).

Real-time Clock

Modern Fortran provides a pair of routines to access the real-time clock:

DATE_AND_TIME

DATE_AND_TIME([date] [,time] [,zone] [,values])

- **date** character string in form ccyymmdd
- **time** character string in form hhmmss.sss
- **zone** character string in form Shhmm, difference between local and UTC
- **values** integer vector of size at least 8 with year, month, day, difference from UTC in minutes, hour, minutes, seconds, milliseconds

SYSTEM_CLOCK

SYSTEM_CLOCK([count][,count_rate][,count_max])

- **count** processor-dependent value of processor clock (`-huge(0)` if no clock)
- **count_rate** clock counts per second (0 if no clock)
- **count_max** maximum for count (0 if no clock)

CPU Time

Fortran >=95 only:

CPU_TIME

CPU_TIME(time)

- **time** real assigned to processor-dependent time in seconds (negative value if no clock)

```fortran
real :: t1,t2
:
CALL CPU_TIME(t1) ! Fortran>=95 only
:
:
CALL CPU_TIME(t2)
print*, 'Time spent in code: ',t2-t1,' seconds'
```

In my experience the `CPU_TIME` intrinsic is not very precise, however, and depends rather strongly on the compiler ...

STOPWATCH

`STOPWATCH` is not part of standard Fortran, but is a nice little package written by William Mitchell at NIST:

`http://math.nist.gov/StopWatch`

which supports a more full featured set of timing routines (including wall time, cpu time, and system time).

Array Functions

The array intrinsic functions will be discussed in a special section devoted to arrays ...

Part III

Fortran Advanced Operations

Numerical Inquiry Functions

There are a bunch of numerical inquiry functions in Fortran 90/95. First, the integer representation model is given by

$$i = s\sum_{k=0}^{q-1} d_k r^k,$$

where

i	integer value
s	sign $(+,-)$
r	radix $(r > 1)$
q	number of digits $(q > 1)$
d_k	k-th digit, $(0 \leq d_k < r)$

The floating-point model is given by

$$x = sb^e\sum_{k=1}^{p} f_k b^{-k},$$

where

x	real value
s	sign $(+,-)$
b	base $(b > 1)$
e	exponent $(q > 1)$
p	number mantissa digits $(p > 1)$
f_k	k-th digit, $(0 \leq f_k < b)$

Inquiry Functions

The inquiry functions are given in the following table:

digits(x)	value of (q, p) for (integer,real)
epsilon(x)	b^{1-p}
huge(x)	largest value in model
minexponent(x)	minimum e
maxexponent(x)	maximum e
precision(x)	decimal precision
radix(x)	base b of integers
range(x)	decimal exponent range
tiny(x)	smallest positive value (real)

Note that all of these functions are generic, and can be used with any supported `KIND`.

Numeric Manipulation Functions

Using the same representational model as the inquiry functions. Designed to return values related to components of real type.

exponent(x)	value of e in real model
fraction(x)	fractional part in real model
nearest(x,s)	value nearest x in direction of sign of s
rrspacing(x)	reciprocal of relative spacing, $\lvert xb^{-e}\rvert b^{p}$
scale(x,i)	xb^{i}
set_exponent(x,i)	xb^{i-e}
spacing(x)	b^{e-p} if $x/=0$ and in range, else `TINY`

IEEE Arithmetic in Fortran

The Fortran 2003 (and later) standard contains facilities for IEEE exception handling. The IEEE[1]/ ISO[2] standard for floating-point arithmetic greatly helped in developing portable numeric code. The goal is to allow for a portable way to test and set the five floating-point exception flags in the IEEE standard.

[1] IEEE 754-1985, Standard for floating-point arithmetic

[2] IEC 559:1989, Binary floating-point arithmetic for microprocessor systems

IEEE Exceptions

IEEE exceptions:

Overflow	the result of an operation exceeds the data format
Division by Zero	finite numerator, zero denominator (result is $\pm\infty$)
Invalid	operation invalid (e.g. $\infty \times 0$) - result is `NaN`
Underflow	result of operation too small for data representation
Inexact	result of operation can not be represented without rounding

IEEE Intrinsics

Three intrinsic modules are provided:

- `IEEE_EXCEPTIONS`
- `IEEE_ARITHMETIC`, itself loads `IEEE_EXCEPTIONS`
- `IEEE_FEATURES`

Inability to load one of these modules will indicate a non-compliant compiler. For detailed usage, see, for example, Metcalf & Reid. At this point there are relatively few compilers that have explicit support for these modules.

Modules for OO Programming

Note that Fortran modules can be used as objected oriented programming (OOP) constructs:

- Creation of derived types that behave just like intrinsic types
- Intrinic types and operators can be overloaded
- Data can be hidden (encapsulated)
- In such a way one can create *semantic extensions*

Generic Interfaces

User-supplied functions, like most of the intrinsics, can have **generic** interfaces. For example, consider the intrinsic `ABS(x)` - behind the scenes, the function actually called depends on the `KIND` of the argument:

`CABS` for `x` complex
`ABS` for `x` real
`IABS` for `x` integer

Generic Interface Procedure Example

You can also make use of generic interfaces, of course:

```
MODULE useful_1
 IMPLICIT NONE
 INTERFACE exemplify
  MODULE PROCEDURE exemplify_int      ! Fortran 95 allows separated
  MODULE PROCEDURE exemplify_real     !  module procedure statements
  MODULE PROCEDURE exemplify_complex
 END INTERFACE exemplify  ! only Fortran >=95 allows specification at END
CONTAINS
 SUBROUTINE exemplify_int(x)
  integer, dimension(:), intent(inout) :: x
  :
 END SUBROUTINE exemplify_real(x)
 SUBROUTINE exemplify_real(x)
  real, dimension(:), intent(inout) :: x
  :
 END SUBROUTINE exemplify_complex(x)
 SUBROUTINE exemplify_complex(x)
  complex, dimension(:), intent(inout) :: x
  :
 END SUBROUTINE exemplify_complex(x)
END MODULE useful_1
```


Generic Interface Example (cont'd)

```
PROGRAM Main
 IMPLICIT NONE
 USE useful_1
 real :: rx(1000)
 integer :: ix(1000)

 :
 CALL exemplify(rx) ! generic call
 CALL exemplify(ix) ! generic call
 :
END PROGRAM Main
```

The rule being that the argument list makes the actual choice of routine unambiguous.

Operator Overloading

Operators can be overloaded using the `INTERFACE OPERATOR` statement:

- Specify the `MODULE PROCEDURE/` to deal with the implementation
- Useful (if not required) for derived types
- Operator name/symbox can be any of the intrinsics or any sequence of 31 characters or less in length enclosed in periods (other than `.true.` and `.false`)
- Be wary when overloading intrinsic operators - binary operators can not be made unary, etc.
- Can not redefine intrinsically defined operations (must remain unambiguous)

Operator Overloading Example

```
MODULE useful_2
 IMPLICIT NONE
 TYPE frac ! type for integer fractions
   integer :: numerator,denominator
 END TYPE frac
 INTERFACE OPERATOR (*)
  MODULE PROCEDURE frac_int, int_frac, frac_frac
 END INTERFACE
CONTAINS
 FUNCTION frac_int(left,right)
  type(frac), intent(in) :: left
  integer,intent(in) :: right
  :
 END FUNCTION frac_int
 FUNCTION int_frac(left,right)
  integer,intent(in) :: left
  type(frac), intent(in) :: right
  :
 END FUNCTION int_frac
 FUNCTION frac_frac(left,right)
  type(frac), intent(in) :: left
  type(frac), intent(in) :: right
  :
 END FUNCTION frac_frac
END MODULE useful_2
```

Note that operator precedence remains unaffected, however.

User Defined Operators

and in a similar way, you can define your own operators:

```
MODULE useful_2
 IMPLICIT NONE
 INTERFACE OPERATOR (.converged.) ! new op
  MODULE PROCEDURE testconv
 END INTERFACE
 :
 :
END MODULE useful_2
```


User Defined Assignment

The assignment operator (=) is a little special - overloading it requires an `INTERFACE ASSIGNMENT` with a `SUBROUTINE`:

- The first argument has `INTENT(OUT)` and represents the left-hand side of the assignment
- The second argument has `INTENT(IN)` and represents the right-hand side
- The subroutine must be “pure,” i.e. not alter global data, or produce any output

User Defined Assignment Example

```
MODULE useful_3
 IMPLICIT NONE
 INTERFACE ASSIGNMENT (=)
   MODULE PROCEDURE frac_eq
 END INTERFACE
 PRIVATE frac_eq
CONTAINS
 SUBROUTINE frac_eq(lhs,rhs)
  type(frac),intent(out) :: lhs
  type(frac),intent(in) :: rhs
  : ! body has to have an assignment to lhs
 END SUBROUTINE frac_eq
 :
END MODULE useful_3
```


Coarray Fortran Basics

Coarray Fortran (CAF, requires Fortran >=2008) follows the PGAS (partitioned global address space) model. It is composed of simple extensions to “regular” Fortran, similar to UPC (Unified Parallel C) and Titanium (Java).

- Designed to follow SPMD (Single Program Multiple Data)
- Fixed number of processes/threads/images
- Explicit data decomposition and synchronization, data and computing are local
- One-sided communciations through co-dimensions
- Current CAF design is very “minimalist,” introduced as few extensions to Fortran as possible

CAF Execution Model

CAF can be used in shared and distributed memory systems (as long as the runtime system supports it),

- **image** refers to each copy of the program in CAF, with $1 \leq$ `this_image()` $\leq$ `num_images()`
- Co-array syntax uses usual () to refer to local data, [] to refer to remote data
- An image moves remote data to local data through co-array syntax
- Programmer handles synchronization

CAF Coarrays

Data stored in other CAF images are referenced through cosubscripts (enclosed in square brackets), mapped to an image index (one to `num_images()`).

- Images have their own data, accessed in the standard Fortran way
- Data with codimensions (square brackets) has corank (at decalaration) and cobounds/coextent (at declaration or allocation)
- Coarray data has the same size and shape on each image

```
  real, dimension(100), codimension[*] :: a,b   ! array coarrays
  real, codimension[*] :: c                     ! scalar coarray

  a(:) = b(:)[i]                                ! coarray b in image i copied to a
                                                !  on executing image
```

`sync all`

Each CAF image executes on its own without regard to others until image control statements are reached, the simplest of which is `sync all`, which forces a barrier synchronization.

```
real :: a[*]     ! scalar coarray

sync all
if (this_image() == 1) then
   read(*,*) a    ! from stdin
   do image=2, num_images()
      a[image] = a
   end do
end if
sync all
! Broadcasts a to all images
```

The above code snippet forms a broadcast, the first `sync all` ensures that image 1 does not interfere with any other image's use of `a`, the second that no image uses the old value of `a` before its update by image one.

More Synchronization

- `sync images (image set)`
  Synchronizes calling image with all others in the image set - `sync all` is equivalent to `sync images (*)`
- `lock/unlock`
  Requires scalar lock variable of type `lock_type` defined by intrinsic module `iso_fortran_env`
- `critical/end critical`
  restricts execution to one image at a time
- `sync memory`
  Gives the ability to define boundaries on image between segments (statements between two image control statements), allows user-defined ordering (very flexible, but also very difficult to debug)

All synchronization statements have optional `stat=` and `errmsg=` arguments (as with `allocate/deallocate`).

Coarray Fortran Simple Example

Once again, our favorite "Hello, world" example:

```
program hello_caf
  !
  ! Simple Hello World for CAF
  !
  use ifport  ! intel specific
  implicit none
  integer :: istat,len
  character(len=max_hostnam_length+1) :: myhostname

  ! intel specific call for host name
  istat = hostnam(myhostname)
  len = len_trim(myhostname)

  write(*,'(a26,i4,a4,i4,a17,a)') "Hello, world, I am image ",this_image(), &
       & " of ",num_images()," total images on ",myhostname(1:len)
end program hello_caf
```


Note that Intel's **ifort** has CAF support, and an extra environment variable for controlling the number of images, and it is pretty simple to run in *shared* mode (i.e., on a single node):

```
[rush:~/d_fortran/d_caf]$ ifort -o hello_caf -coarray hello_caf2.f90
[rush:~/d_fortran/d_caf]$ export FOR_COARRAY_NUM_IMAGES=8
[rush:~/d_fortran/d_caf]$ ./hello_caf2
 Hello, world, I am image    1 of    8 total images on k07n14
 Hello, world, I am image    2 of    8 total images on k07n14
 Hello, world, I am image    3 of    8 total images on k07n14
 Hello, world, I am image    4 of    8 total images on k07n14
 Hello, world, I am image    5 of    8 total images on k07n14
 Hello, world, I am image    6 of    8 total images on k07n14
 Hello, world, I am image    7 of    8 total images on k07n14
 Hello, world, I am image    8 of    8 total images on k07n14
```

Note that running CAF in shared mode does not require jumping through any extra hoops (not so in the distributed case).

```
#!/bin/bash
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=8
#SBATCH --constraint=CPU-L5520|CPU-L5630
#SBATCH --partition=debug
#SBATCH --time=00:10:00
#SBATCH --mail-type=END
#SBATCH --mail-user=jonesm@buffalo.edu
#SBATCH --output=slurmQ.out
#SBATCH --job-name=caf-test
#
#export | grep SLURM
module load intel-mpi intel
module list
ifort -o hello_caf.dist -coarray=distributed hello_caf.f90
EXEC=./hello_caf.dist
export MY_NODEFILE=tmp.$$
srun -l hostname -s  | sort -n | awk '{print $2}' | uniq > $MY_NODEFILE
NNODES=`cat $MY_NODEFILE | wc -l`
export FOR_COARRAY_CONFIG_FILE=caf.$$
NODES=`cat $MY_NODEFILE | awk '{printf "%s ", $1}'`
for node in $NODES; do
  echo "-n $SLURM_NTASKS_PER_NODE -host $node $EXEC " >> $FOR_COARRAY_CONFIG_FILE
done
#export I_MPI_DEBUG=4 # enable to see detailed placement info
mpdboot -n $NNODES -f $MY_NODEFILE -v
mpdtrace
$EXEC
[ -e $MY_NODEFILE ] && \rm $MY_NODEFILE
[ -e $FOR_COARRAY_CONFIG_FILE ] && \rm $FOR_COARRAY_CONFIG_FILE
```

Note the use of `mpds` and the MPI subsystem for distributing tasks, although the MPI task launcher is not itself being used.


```
[rush:~/d_fortran/d_caf]$ cat slurmQ.out
Currently Loaded Modulefiles:
  1) null              3) use.own           5) intel-mpi/5.0.1
  2) modules           4) intel/15.0
running mpdallexit on d16n32
LAUNCHED mpd on d16n32  via
RUNNING: mpd on d16n32
LAUNCHED mpd on d16n34  via  d16n32
RUNNING: mpd on d16n34
d16n32
d16n34
 Hello, world, I am image    3 of   16 total images on d16n32
 Hello, world, I am image    4 of   16 total images on d16n32
 Hello, world, I am image    9 of   16 total images on d16n34
 Hello, world, I am image    5 of   16 total images on d16n32
 Hello, world, I am image   10 of   16 total images on d16n34
 Hello, world, I am image    6 of   16 total images on d16n32
 Hello, world, I am image   11 of   16 total images on d16n34
 Hello, world, I am image    7 of   16 total images on d16n32
 Hello, world, I am image   12 of   16 total images on d16n34
 Hello, world, I am image    8 of   16 total images on d16n32
 Hello, world, I am image   13 of   16 total images on d16n34
 Hello, world, I am image    1 of   16 total images on d16n32
 Hello, world, I am image   14 of   16 total images on d16n34
 Hello, world, I am image    2 of   16 total images on d16n32
 Hello, world, I am image   15 of   16 total images on d16n34
 Hello, world, I am image   16 of   16 total images on d16n34
```


Example 1, Build/Use Strings and Internal Files

It is fairly common to want to convert to a string, in `C/C++` you can **cast** or use `sprintf`, and in Fortran you can use an **internal file** to write to a string.

In the following example we use an internal file (basically a string) to convert an integer to a string, and then build a file name incorporating that string (suppose, for example, that you wanted to write out a separate file for each rank in an MPI application).


```fortran
program testchar
  implicit none
  ! convert integer to string for use in file name
  character(len=*),parameter :: alphabet1='abcdefghijklm', &
                               alphabet2='nopqrstuvwxyz'
  character(len=*),parameter :: numerals='0123456789'
  character(len=12) :: cstring
  integer :: testint

  print*, 'First four letters: ',alphabet1(1:4)
  print*, 'alphanumeric: ',alphabet1//alphabet2//numerals

  ! test convert int to string
  testint=123
  write(cstring,'(i12)') testint
  print*," test string = ",TRIM(ADJUSTL(cstring)),"."
  print*," filename test = ","outfile_"//TRIM(ADJUSTL(cstring))//".dat"
end program testchar
```

```
[rush:~/d_fortran]$ ifort -o testchar testchar.f90
[rush:~/d_fortran]$ ./testchar
 First four letters: abcd
 alphanumeric: abcdefghijklmnopqrstuvwxyz0123456789
  test string = 123.
  filename test = outfile_123.dat
```


Example 2, Command Line Arguments

New to Fortran >=2003, command line arguments are now supported. The following example illustrates the use of `get_command`, `command_argument_count`, and `get_command_argument` to process command line arguments to a Fortran program.


```fortran
! cmdline.f90 -- simple command-line argument parsing example
! test fortran2003 support for get_command_argument, get_command,
!  and command_argument_count
program cmdline
  implicit none

  character(len=255) :: cmd
  character(len=*), parameter :: version = '1.0'
  character(len=32) :: arg,date*8,time*10,zone*5
  logical :: do_time = .false.
  integer :: i

  call get_command(cmd)
  write(*,*) 'Entire command line:'
  write (*,*) trim(cmd)
  do i = 1, command_argument_count()
     call get_command_argument(i, arg)

     select case (arg)
     case ('-v', '--version')
        print '(2a)', 'cmdline version ', version
        stop
     case ('-h', '--help')
        call print_help()
        stop
     case ('-t', '--time')
        do_time = .true.
     case default
        print '(a,a,/)', 'Unrecognized command-line option: ', arg
        call print_help()
        stop
     end select
  end do
```



```fortran
  ! Print the date and, optionally, the time
  call date_and_time(DATE=date, TIME=time, ZONE=zone)
  write (*, '(a,"-",a,"-",a)', advance='no') date(1:4), date(5:6), date(7:8)
  if (do_time) then
     write (*, '(x,a,":",a,x,a)') time(1:2), time(3:4), zone
  else
     write (*, '(a)') ''
  end if

contains

  subroutine print_help()
    print '(a)', 'usage: cmdline [OPTIONS]'
    print '(a)', ''
    print '(a)', 'Without further options, cmdline prints the date and exits.'
    print '(a)', ''
    print '(a)', 'cmdline options:'
    print '(a)', ''
    print '(a)', '  -v, --version     print version information and exit'
    print '(a)', '  -h, --help        print usage information and exit'
    print '(a)', '  -t, --time        print time'
  end subroutine print_help

end program cmdline
```

```
[rush:~/d_fortran]$ ifort -o cmdline cmdline.f90
[rush:~/d_fortran]$ ./cmdline
 Entire command line:
 ./cmdline
2014-09-29
[rush:~/d_fortran]$ ./cmdline -v
 Entire command line:
 ./cmdline -v
cmdline version 1.0
[rush:~/d_fortran]$ ./cmdline --help
 Entire command line:
 ./cmdline --help
usage: cmdline [OPTIONS]

Without further options, cmdline prints the date and exits.

cmdline options:

  -v, --version     print version information and exit
  -h, --help        print usage information and exit
  -t, --time        print time
[rush:~/d_fortran]$ ./cmdline --time
 Entire command line:
 ./cmdline --time
2014-09-29 14:13 -0400
```