The Kostelich Quickstart Fortran Guide

Eric J. Kostelich

ARIZONA STATE UNIVERSITY
SCHOOL OF **MATHEMATICAL & STATISTICAL SCIENCES**

Sept. 19, 2018

Quick history of Formula Translation

- Developed at IBM by a team led by John Backus
- Released 1957; first commercially successful high-level language
- Surprisingly efficient: program speed was within a factor of 2 of hand-coded assembly
- Even when computers cost millions of dollars, software development costs were greater than the machines'
- Half of all software was written in Fortran by 1960 (Alex Aiken)

It's Fortran, not FORTRAN

- First programming language to have an official standard (ANSI, 1966)
- 1977 revision added character variables, if/then/else
- 1990 revision added user-derived types, array notation, modules
- 1995 revision is the most widely supported one; updates Fortran '90
- 2003 standard adds OO support; full compilers by IBM and Cray (Intel's is close)
- 2008 standard adds coarrays and explicit parallelism

Many good sources of compilers for PCs/Linux

- gfortran from the GNU project at gcc.gnu.org/wiki/GFortran
- Intel Academic Developer Program (student licenses from $0 to $129 for C, C++, Fortran, Math Kernel Library, and more)
- Portland Group (pgfortran)
- g95 by ASU alum Andy Vaught at g95.org
- Also IBM (xlf), Absoft, Lahey, NAG (Numerical Algorithms Group), Sun/Oracle

The compilation process

- The compiler translates source code into “object code”
- Object code is machine instructions with unresolved addresses
- The linker combines object code files and libraries and resolves most addresses to create an executable file
- The loader finalizes all addresses, obtains stack and heap space from the kernel, then launches the program
- A command like `ifort` (Intel) or `gfortran` (GNU) runs the compiler and/or linker as needed
- The shell calls `execve`, which invokes the loader

Outline of the process

- Create a file, say `hello.f90`, using a text editor

```
program hi              optional statement
write(6,*) 'Hello world'    either single or double quotes
end program hi          or simply end
```

- Compile with `ifort –c hello.f90`
- Link with `ifort hello.o –o hello`
- Run from the shell as `hello` or `./hello`, depending on `PATH`
- Multiple files can be linked as

```
ifort file1.o file2.o file3.o –o myprog
```

Basic facts about Fortran syntax

- Every Fortran statement must either start with a keyword or be an assignment statement
- Case insensitive (`sin` and `SIN` are synonymous)
- There are no reserved words (e.g., `sum` is a built-in function but also can be used as an ordinary variable)
- Two formats: fixed form and free form
- Spaces are not significant to the compiler in fixed-form source, except in character strings
- Many compilers assume that `file.f90` is free form and `file.f` is fixed form

Fundamental components of a Fortran program

- Exactly one main program (starts with a `program` statement)
- Optional: one or more `module`s containing data and/or subprograms
- Fortran includes many built-in functions that require no declaration

Free-form source (recommended for new code)

- Statements may begin anywhere on the line
- Maximum of 132 characters per line
- Tab characters aren't allowed (but many compilers accept them)
- `!` outside a character string starts a comment that extends to the end of the line
- Use `&` for continuation:

```
sum = a + b + c + d + e &
      + f + g

msg = "Now is the time for all good people &
      &to come to the aid of their country"
```

Fixed-form syntax (from punch card days)

- All statements must be located between columns 7 and 72—no tabs allowed!
- Continuation characters must appear in column 6
- Statement labels must appear in columns 1–5
- Blanks are not significant except in quoted strings

`x=3.14 159 653 593`

`callsub(x)` *must be* `call sub(x)` *in free form*

`write(6,*)`

`1 x` *any nonzero character in column 6*

`S T O P`

`E N D`

The implicit typing rule

- Variables do not have to be declared before they are used
- By default, variables starting with the letters `I` through `N` are `integer`. All others are `real`
- Beware: If implicit typing is turned on, then the compiler allocates storage for any variable whenever it is mentioned
- Example: `TOTAL = T0TAL + 1`

The implicit typing rule, 2

- `implicit none` turns off implicit typing and requires all variables to be declared.
- Strongly recommended!
- `implicit none` follows after any `use` statements and applies to the rest of the module or subroutine in which it appears

Recommended program structure

`program myprog`

`use mod1` *module #1 (optional)*

`use mod2` *module #2 (optional)*

`implicit none` *turns off implicit typing*

variable declarations

executable statements

`stop` *optional*

`end program myprog` *or simply* `end`

- Recommendation: Put `module mymod` and only `module mymod` in a source file named `mymod.f90`

Modules

- Similar to Ada packages
- Contain data and/or subroutines to manipulate the data
- The compiler checks calls to module routines for agreement in argument type, kind, and rank (TKR agreement)
- Allows for data encapsulation and hiding
- Use modules to group related functions and data

Example: a module containing only code

```
module rootfinders
implicit none
contains
subroutine newton(f, df, x0, tol, maxit, ierr)
...
end subroutine newton
subroutine zeroin(f, x0, tol, maxit, ierr)
...
end subroutine zeroin
end module rootfinders
```

Example of a module containing data and code

```
module simulation
use rootfinders  ! zeroin is more reliable
implicit none
real, parameter:: PI = 3.14159265
real, save, private:: param = 0.0
contains
subroutine initialize(filename, x0)
...     compute or read param from file
end subroutine initialize
subroutine integrate(x0, solution)
...     use param as needed here
end subroutine integrate; end module simulation
```

Example, 3

```
program mysim
use simulation
implicit none
character(80):: infile, arg
real:: x, sol
call get_command_argument(1, infile)
call get_command_argument(2, arg)
read(arg,*) x        convert string to number
call initialize(infile, x)
call integrate(x, sol)
write(6,*) sol
end program mysim
```

Example, 4

Assuming the modules are put into source files with the same name, the simulation is compiled and linked as:

`ifort –c rootfinders.f90` *creates* *rootfinders.o*

`ifort –c simulation.f90` *creates* *simulation.o*

`ifort –c mysim.f90` *creates* *mysim.o*

`ifort mysim.o simulation.o rootfinders.o –o mysim` *all on one line*

- Modules must be compiled before they are used
- Beware circular dependencies: If `a` uses `b` and `b` uses `a`, then you can't use either!

Precedence rules follow mathematics

1. Parentheses, left to right
2. Exponentiation `**`, right to left
3. Multiplication and division, left to right
4. Addition and substraction, left to right

`x**3**2`	$x^{3^2} = x^9$
`3*x**2`	$3x^2 = 3 \times (x^2)$
`2*x+1`	$2x + 1 = (2x) + 1$
`2*(x+1)`	$2(x + 1) = 2x + 2$

Type, kind, rank

- Every Fortran variable has a type, and kind, and a rank
- The built-in types are `logical`, `integer`, `real`, `complex`, and `character`
- The `kind` denotes the precision of `real` and `integer` variables
- The `rank` denotes the number of subscripts
- Dummy arguments must agree with actual arguments in type, kind, and rank (TKR matching)
- The compiler automatically checks for TKR mismatches when subprograms are `contain`ed in modules

Data types and kinds—Real and complex

- `real` and `complex` are single precision (32 bits) by default
- Fortran supports double precision using `kind`
- Example: double precision (~ 15 decimal digits) `integer,`
  ```
  parameter:: DOUBLE = selected_real_kind(15)
  real(DOUBLE), parameter:: &
        PI = 3.1415926535897932_DOUBLE
  ```
- Example: single precision (~ 7 decimal digits) `integer,`
  ```
  parameter:: SINGLE = selected_real_kind(7)
  ```
- Be careful with manifest constants: `0` is an integer and `3.14` is single precision

Other constructs

- In literals, `e` indicates single precision and `d` double
- Example: 1.23×10^4 is `1.23e4` (single) and `1.23d4` (double)—alternatively, `1.23e+04 or 1.23d+04`
- If there is no exponent, then the quantity is single precision
- A better way to manage all this:

```
integer, parameter:: SINGLE=kind(1.0)
integer, parameter:: DOUBLE=kind(1.0d0)
integer, parameter:: WP=DOUBLE      Working Precision
```

- Then `1.23e4_WP` has type `WP`

Data types and kinds—Integers

- Integers are always signed
- The default kind of an integer is 32 bits on all modern systems
- `integer, parameter:: MYINT=selected_int_kind(n)` returns the kind parameter of integer types that can represent values to at least $|10^n|$
- Example: 64-bit integers (note: $2^{63} \approx 9 \times 10^{18}$)

```
integer, parameter:: LONGINT = selected_int_kind(18)
integer(LONGINT):: key
key = 12345678901234_LONGINT * 131
```

Rank and shape

```
real:: s, v(10), a(3,3), c(0:2,–1:2,5)
```

- `s` (scalar) has rank 0
- `v` (vector) has rank 1 and shape 10
- `a` (array) has rank 2 and shape $(3,3)$
- `c` has rank 3 and shape $(3,4,5)$
- Each has the default kind (single precision)
- Subscripts start from 1 unless otherwise specified

Dummy argument declarations

```
subroutine sub(x,y,z)
real, intent(in):: x
real, intent(out):: y
real, intent(inout):: z     or in out
integer k     local variable
```

- `x` is not altered by the execution of `sub`
- `y` is assigned a value by the execution of `sub`
- The value of `z` is used on input and may be altered on output
- Recommendation: always specify an `intent` for dummy arguments and declare them before local variables

Dummy array arguments, 1

```
program p
use mymod
implicit none
real:: a(20,10)
call sub(a,20,10)
...
```

```
module mymod
implicit none
contains
subroutine sub(arr,m,n)
integer, intent(in):: m,n
real, intent(out):: arr(m,n)
```

- Advantage: Works whether or not `sub` appears in a `module`
- Advantage: The intended array dimensions are obvious
- Disadvantage: The programmer must assure that `m` and `n` match the dimensions of the actual `a`

Dummy array arguments, 2

```
program p
use mymod
implicit none
real:: a(20,10)
call sub(a)
...
```

```
module mymod
implicit none
contains
subroutine sub(arr)
real, intent(out):: arr(:,:)
integer:: m,n     local variables
m=size(arr,1); n=size(arr,2)
```

- Advantage: `arr` is always passed with the right dimensions
- Disadvantage: The code will crash if the `sub` does not appear in a `module`, and the intended dimensions can be buried in executable code

Character strings

- `character(3):: answer` declares a string of length 3
- Strings know their length

  `answer='yes'` *OK*

  `answer='no'` *OK; one blank is silently added*

  `answer='no way'` *truncated to* `'no '`

  `if(answer(1:1) == 'y') then ...` *first character only*

- Strings are scalars, even though substrings look like arrays
- `answer(2:)` is all but the first character
- `answer(:2)` and `answer(1:2)` are the first two characters

Character strings, 2

- Declare dummy argument strings as follows:

```
subroutine sub(ans)
character(*), intent(in):: ans
```

- Comparisons pad the shorter string with blanks

`if(ans .eq. 'no') then` *OK; compared with* `'no '`

- `len(ans)` returns the length (3 if `answer` is passed)
- `trim(ans)` trims off any trailing blanks
- `len_trim(ans)` returns `len(trim(ans))`
- Beware: `character:: ans(3)` is an array of 3 characters

Example: a numbered sequence of file names

```
character(3):: anum      assumes n < 1000
character(80):: basename, filename
integer:: k, n

. . .

basename=’mydata’      we want mydata1.dat, mydata2.dat,...
do k=1, n
  write(anum,’(i0)’) k
  filename=trim(basename)//trim(anum)//’.dat’
  open(unit=4, file=filename, position=’rewind’)
  write(4,∗) data
  close(4)
enddo
```

Character strings, 3

- Use * for the length in a subroutine argument:

```
subroutine sub(filename)
character(*), intent(in):: filename
```

- `len(filename)` equals the length of the actual string that is passed
- Two equivalent ways to handle quote marks:

```
errmsg = "Can’t open "//filename
errmsg = ’Can’’t open ’//filename
```

- Beware! `filename=filename//’.dat’` does nothing

Scope

```
module constants
real, parameter:: &
   PI=3.1415926535
end module constants
module mod
use constants
contains
   subroutine sub
   real y, z; . . .
   y = f(z)      f requires no declaration
end subroutine sub
   real function f(x)
   real, intent(in):: x
      f = PI*x
   end function f(x)
end module mod
```

- `sub` and `f` may appear in any order within `mod`
- `use mod` imports the interfaces for `sub` and `f`

Scope, 2

```
module constants
real, parameter:: &
 PI=3.1415926535
end module constants
module mod
use constants
contains
subroutine sub
real y, z; . . .
y = f(z)      f requires no declaration
end subroutine sub
real function f(x)
real, intent(in):: x
f = PI * x
end function f(x)
end module mod
```

- `y` and `z` are defined only in `sub` and `x` only in `f`
- `PI` is visible to both `sub` and `f`

Control structures: `if`

- Single-statement version: `if(n > 0) sum=sum/n`
- Block construct:

```
if($condition_1$) then
$body_1$

else if($condition_2$) then      optional
$body_2$

⋮      as many else if's as you like

else      optional

. . .

endif
```

Control structures: case

```
integer n
...
select case(n)
case(1)
  write(6,*) 'one'        no fall-through, unlike C/C++
case(2:10)
  write(6,*) n
case(:0)
  write(6,*) 'nonpositive'
case default        i.e., n > 10
  write(6,*) 'out of range'
endselect        or end select
```

Comparison and logical operators

- Equivalent pairs:

  `(>, .gt.)`, `(>=, .ge.)`, `(<, .lt.)`, `(<=, .le.)`,
  `(==, .eq.)`, `(/=, .ne.)`

- Order of precedence: `.not.`, `.and.`, `.or.`
- `x.gt.0.and.y.gt.0.or.x.lt.0.and.y.lt.0` is equivalent to
  `(x > 0 .and. y > 0) .or. (x < 0 .and. y < 0)`

Fortran has no “short circuit” evaluation

Consider the expression `f(x) > 0 .or. g(y) < 0`

- In C/C++, f is evaluated first and g is not called if $f(x) > 0$
- In Fortran, either f or g may be evaluated first
- If you require f before g, then rewrite as

```
if(f(x) > 0) then
    if(g(y) < 0) then
    . . .
    endif
endif
```

Beware!

- `if(n > 0 .and. sum/n < 0)` is OK in C/C++
- Optimization flags can change the order of evaluation in Fortran!
- Rewrite as

```
if(n > 0) then
    if(sum/n < 0) then
    . . .
    endif
endif
```

Control structures: `do`

- Counted `do`:

```
do j=1, n
    loop body executes n times
enddo
```

all quantities must be integer — *or end do*

- The loop body is skipped if $n < 1$
- `do j=n, 1, –1` counts backwards (MATLAB: `j=n:–1:1`)
- `do j=1, n, 2` iterates over the odds
- It's illegal to attempt to change `j` inside the loop
- The loop count is fixed at the start (even if the loop body changes n)

Old-fashioned `do` constructs

- Prior to Fortran 90, `do` required a label:

```
   do 20 k=1, n
   do 10 j=1, m
10 a(j,k) = 0.0
20 continue
```

- Multiple loops can end on the same label

```
   do 10 k=1, n
   do 10 j=1, m
10 a(j,k) = 0.0
```

- Recommendation: use `enddo` for new code

Other do constructs in Fortran 90

- The infinite loop:

```
do
    . . .
    if(condition) exit      as necessary
enddo
```

- while loop:

```
do while(condition)
    . . .
   The loop body is skipped if condition is false at the start
enddo
```

Other control mechanisms

- `return` (in subroutines)
- `stop` (recommended in the main program only)
- Error/exception handling:

```
     if(disaster) goto 911      or go to
    ...
911 continue
    abort or recover
```

Advice and caveats

- Unless there's a good reason otherwise, use the most common or natural idiom for common tasks in the language
- In Fortran, index arrays from 1 to n
- In C/C++, index from 0 to $n - 1$
- Exceptions in Fortran include "ghost points" in finite-difference methods, where the actual grid points are 1 to n but arrays might be subscripted from 0 to $n + 1$ to simplify the algorithm

Advice and caveats, 2

- Don’t hardwire `kind` values! They are not standardized
- Example: Avoid `real(8):: x`
- Not all compilers use 8 for double precision
- Don’t rely on compiler flags to switch from single to double precision! Some compilers promote the types of constants (like 0.1) to double with such flags and others don’t
- Use `kind` and be explicit (e.g., `0.1_WP`)

Advice and caveats, 3

- Type conversions in assignments are silent and may lose precision:

`dy = sx`	*OK if* `dy` *is double and* `sx` *is single*
`sx = dy`	*loses precision*
`k1 = f(x)`	*disastrous with implicit typing*

- Use `sqrt` instead of exponentiation:

`y = sqrt(x)`	*good*
`y = x**0.5`	*less accurate and slower*
`y = x**1/2`	*disastrous*

- `1/2` is 0 but `1.0/2.0` is 0.5

Advice and caveats, 4

- Use `int` and `real` for explicit conversion
- `k = int(x)` *clearly intentional*
- `x = real(k)` *clearly intentional*
- `sp = real(dp, kind(sp))` *intentional double-to-single conversion*
- `x = real(k, kind(x))` *preserves precision if* `k` *is large and* `x` *is double*
- Special case: `x = 0` or `x = 0.0` are both OK

Advice and caveats, 4

- Be careful when passing literals! Given

```
subroutine sub(x,y)
use precision
real(DOUBLE), intent(in):: x,y
```

- This call is erroneous:

```
program p
use precision
real(DOUBLE):: x
. . .
call sub(x, 0.0)
```

error—`0.0` *is single*

Advice and caveats, 5

- Consider parameters for common cases:

```
real(DOUBLE), parameter:: ZERO=0.0
. . .
call sub(x, ZERO)
```

A portable and consistent method for precision control

```
module precision      Working Precision
integer, parameter:: WP=selected_real_kind(7)    single precision
end module precision
. . .
program myprog
use precision
real(WP), parameter::PI=3.1415926535897932_WP
```

- For double, say `WP = selected_real_kind(15)` and recompile

Be explicit about static initializations

- This doesn't do what you think it does:

```
subroutine sub(arg)
real, intent(inout):: arg
real:: x = 0.0
some calculation with x
```

- Instead you must say:

```
subroutine sub(arg)
real, intent(inout):: arg
real:: x
x = 0.0
...
```

Be explicit about static initializations, 2

- Correct usage—use `save` explicitly:

```
subroutine sub
logical, save:: firstcall = .true.
if(firstcall) then
```

perform some initialization

```
firstcall = .false.

endif
```

- Also OK:

```
subroutine randgen(number)
integer, save:: seed = 1311131
```