

# Fortran 90, 95 and 2003

---

# Some conventions in this presentation

---

- *red italics* for code and code fragments
- underlining or **blue roman** for emphasis
- *blue italics* for code comments
- *black italics* for quotations
- **green** for emphasis or C/C99/C++ conventions
- All of these conventions may be violated in some places (like with this bullet item)

# Ancient History

---

- Old professors tend to think [Fortran77](#) when the F-word comes up
- Biggest syntactic change came with [Fortran90](#), which some compilers had (partly) implemented starting 1988
- [Fortran95](#) was a minor revision - details given later
- [Fortran 2000 = F2k = Fortran 2003](#) was a major release
  - not all compilers support the full specification (2008)
  - major improvement: “FORTRAN” can officially be called “Fortran”
- [Fortran 2008](#) draft standard is now being circulated (2008)
- I use any of [F90](#), [F95](#), [F2k](#) when describing some feature, usually based on the first version that supported the feature

# Post-Fortran77 Versions

---

- Generally speaking, anyone in scientific computing should be able to at least **read** a Fortran code
- Applies to chemists, physicists, computer scientists, even mathematicians in scientific computing
- Fortran is still favorite language in scientific computing
  - Includes both **legacy** codes and **currently written** codes
  - Major principle: add all the goodies most people want to the language, provided it does not interfere with the **ability of a compiler to generate efficient machine code**
- Reason for restrictions on pointers, user derived types, etc.
- **Multi-language systems** are routine in scientific computing: Fortran for numerics, C for string-handling, Java for user interfaces, Perl and Python for scripting

# Fortran-C English Translations

---

- Nomenclature is tricky for folks unaccustomed to Fortran:
- **derived type** = **user-defined** type [struct]
- **module** = **class** [not exactly, but close enough]
- **scalar** = **basic entity** [real, int, complex, arrays, ... ]
- **functional** language = **procedural** language
- **extending** = **subclassing** [more sensible]
- **intrinsic** function = **built-in** function: “*those available for use without any declaration or definition*”. Also sometimes just called “intrinsic”
- **rank**, **length**, **size** refer to characteristics of an array

# Fortran-C English Translations

---

- dummy argument = prototype argument
- procedure = function + subprograms
  - ★ denotes both Fortran functions and subroutines
  - ★ functions/subroutines differ in invocation syntax:
    - *call subroutineA(arg1, arg2, returnarg)*, versus
    - *returnarg = functionA(arg1, arg2)*
  - A subroutine is like a function of type void, e.g.
    - *void subroutineA(arg1, arg2, returnarg)*

# Fortran-C English Translations

---

- `formatted` file = `text` file
- `unformatted` file = `binary` file
- Variable types translation:
  - `character` = `string` [beware of this one]
  - `real` = `float`
  - `double precision` = `double`
  - `logical variable` = `boolean`
- “real” can refer to both `4-byte single precision` and `8-byte double precision` types
- Sometimes (nonstandard but widely used) type declaration for a real is `real*4` or `real*8`

# Fortran 90+ Look and Feel

---

- F77 statements are **restricted to columns 7-71 only**; F90+ is “**free form**”
- Lines ended by carriage return; use **&** at end of line to have a statement stretch over more than one line.
- Comments start with **!**, and end when the line does
- Complex numbers are built-in basic types
  - *complex :: A(n,n,10)*
- Relational operators can be acronyms (old style) or symbols (new style): (*.eq.* or **==**), (*.le.* or **<=**), (*.ne.* or **/=**)
- Boolean operators include *.not.*, *.and.*, *.or.*, ...

# Fortran 90: Look and Feel

---

- Useful clarity feature: loops, if clauses, formats, any statement can be labeled with a name:

*sinaxpy: do k = 1, n*

*x(k) = y(k) + a\*sin(x(k))*

*end do sinaxpy*

- Compare to using matching braces as in C, Scheme, Java
- Name *sinaxpy* need not have been given on the end statement above, and could have just used *end do*
- Not **required**, but **use labels**, especially for multipage/lengthy constructs

# Fortran 90: General

---

- **Array** handling: multidimensional arrays are directly supported constructs and handled with great efficiency
- The **rank** of an array is its dimensionality
  - rank-1 **array** is not the same as a rank-1 **matrix**
  - *integer A(n,n,10)* is a rank-3 array; up to rank-7 allowed
  - draft F2008 standard allows rank-10
- **Complex numbers** are built-in (and usually are better performant than current C99/C++ libraries)
  - At last check (admittedly years ago), no C++ or Java library correctly handled the full IEEE 754 standard for complex arithmetic
  - *complex A(lda, n)* is a rank-2 array of complex numbers

# Fortran 90: Example 1

---

! Name of program need not be "main"

*program whatever*

! name is declared a character variable of length 20

*character, dimension(20) :: name*

! can use semicolons to stack commands

*name(1) = 'A '; name(2) = 'b'; name(3) = 'e'*

! I/O using defaults

*print \*, "Hello ", name*

*end program whatever*

## Fortran 90: Example 2

---

*program circle*

```
real :: radius = 4.2 , x, y  
integer, parameter :: points = 100  
do k = 1, points  
    theta = 360.0 *k / points  
    x = radius*cos(theta)  
    y = radius*sin(theta)  
    write(*,*) x, y  
end do
```

*end program circle*

Something is missing here in the standard idioms of C/C++/Java. What is it?

## Fortran 90: Example 2

---

*program circle*

*real :: radius = 4.2 , x, y*

*integer, parameter :: points = 100*

*do k = 1, points*

*theta = 360.0 \*k / points*

*x = radius\*cos(theta)*

*y = radius\*sin(theta)*

*write(\*,\*) x, y*

*end do*

*...*

*end program circle*

**Implicit typing** rule:  
variable names  
starting with letters  
i-n are assumed to be  
integer, real otherwise

## Fortran 90: Example 2

---

```
program circle  
implicit none  
integer :: k  
real :: radius = 4.2 , x, y, theta  
integer, parameter :: points = 100  
do k = 1, points  
    theta = 360.0 *k / points  
    x = radius*cos(theta)  
    y = radius*sin(theta)  
    write(*,*) x, y  
end do  
end program circle
```

Use *implicit none* for newly written code, but many existing codes still rely on implicit typing

# Control constructs

---

- *do ... end do*. Can *exit* or *cycle* in loop
- *where ... end* (example later in arrays)
- Case construct:

```
integer :: grade ...
```

```
select case (grade)
```

```
case(0:90)
```

```
letter_grade = 'F'
```

```
case(91:100)
```

```
letter_grade = 'A'
```

```
casedefault
```

```
print *, 'Invalid grade, re-enter it'
```

```
end case
```

# Fortran 90: Example 3

---

- List-directed I/O

*read(\*,\*) int1, int2, real3, real4, complex\_a*

- User can separate data entries with commas, spaces or returns in the input file (or other characters if you specify them).
- The types of variables in the list “direct” the formatting while reading the input data.
- Can pass in status variable to detect EOF.
- File must first be 'opened' using an *open* statement

# Fortran 90: Example 3

---

- List-directed output

*write(\*,\*) int1, int2, real3, real4, 3.1415*

- Automatically formats output according to type of the argument (including constants)
- Newline assumed after each unformatted read/write
- Can specify the format

*write(\*, intreal\_format) int1, real2*

*intreal\_format: &*

*format('step # ', i10, ' value ', e21.17)*

# Fortran 90: Example 3

---

- Formats more commonly use a numeric line tag

```
write(*, 200) int1, real2
```

```
200 format('step # ', i8, ' value ', f10.8)
```

- Can put format string directly in the *write* statement

```
write(*, fmt="(a,i8,a,f10.8)") &  
'step #', int1, 'value ', real2
```

# Fortran 90: Example 3

---

- File I/O is similar

```
open(unit=9, iostat=ios, file = 'results')
```

```
write(9, 100) int1, real2
```

```
100 format('step ', i10, 'val ', e21.17)
```

- Don't like writing to "unit 9"? Name it:

```
res_file = 9
```

```
open(unit=res_file, iostat=ios, file = 'results')
```

```
write(res_file,100) int1, real2
```

- Fortran 2008 has a better mechanism for unit numbers

# Array Syntax

---

- Can replace array operations such as

*do k = 1, n*

$$y(k) = y(k) + a*x(k)$$

*end do*

with the one-liner

$$y(1:n) = y(1:n) + a*x(1:n)$$

or, if  $x$  and  $y$  are exactly of length  $n$ ,  $y = y + a*x$

- Same holds if  $x$ ,  $y$  are rank- $k$  arrays; rank-2 example:

$$y(1:m, 1:n) = y(1:m, 1:n) + a*x(1:m, 1:n) \text{ or}$$

$$y = y + a*x$$

# Arrays and intrinsic functions

---

- All Fortran *intrinsic functions* can be used with arrays instead of just scalars:

*do k = 1, n*

*x(k) = y(k) + a\*sin(x(k))*

*end do*

becomes (if x,y are of conformant length)

*x = y + a\*sin(x)*

- Fortran2k adds the ability to do this with (some of) functions the user writes

# Fortran 90 Array Intrinsic

---

`all(mask, dim)` : true if all values are true

`cshift(array, shift, dim)` : circular shift

`dot_product(A, B)` : dot product of two rank-one arrays

`matmul(A, B)` :  $A*B$ , if conformant and at least one is rank-2

`maxval(array, dim, mask)` : maximum value of array

`merge(tsource, fsource, mask)` : combining two arrays using a mask

mask is a true/false Boolean array (type *logical* in Fortran)

common mask: `where(A > 0)`

# Fortran 90 Array Intrinsic

---

maxloc(array, mask) : *location* of element with maximum value

spread(source, dim, ncopies) : replicate an array by adding a  
dimension

sum(array, dim, mask) : sum array elements

transpose(matrix) : transpose array of rank two

# Fortran 90 Precision Control

---

- Precision control of reals: If at least 10 decimal digits will be needed and exponents from  $10^{-50}$  to  $10^{+50}$ , you can declare

*real(kind=selected\_real\_kind(10,50)) :: x, y*

- Or you can match the precision with a given constant:

*integer, parameter :: dbk = kind(1.0d0)*

*real(kind=dbk) :: x, y*

- Notice the letter *d* used for the exponent instead of the more common *e*; *d* forces it to be a double precision value of 1.0
- Recommended approach: put all your precision kinds into a *module*, then *use* that module.

# Fortran 90: Precision Control

---

- BEWARE: although a *kind* is an integer, the mapping between integers and the actual kinds is compiler implementor dependent. So avoid  
*real(kind=2) :: x, y*
- SGI uses the integer 1 for single precision and 2 for double. Intel uses 4 for single precision and 8 for double.
- A common portability problem when changing platforms

# Bramley's Kinds

---

*module kinds*

*implicit none*

*public*

*integer, parameter :: real4 = selected\_real\_kind(6, 37)*

*integer, parameter :: real8 = selected\_real\_kind(13, 300)*

**! Names that C programmers feel more comfortable with**

*integer, parameter :: float\_type = real4*

*integer, parameter :: double\_type = real8*

...

# Bramley's Kinds

---

...

*integer, parameter:: integer4 = selected\_int\_kind(9)*

*integer, parameter:: integer8 = selected\_int\_kind(18)*

**! Names that C programmers feel more comfortable with**

*integer, parameter:: int\_type = selected\_int\_kind(18)*

*integer, parameter:: longint\_type =  
selected\_int\_kind(18)*

*end module kinds*

# Fortran 90

---

- Array declarations allow arbitrary upper/lower bounds  
*real(real8), dimension(3,4) :: A, B* ! real8 defined earlier  
*real, dimension(-2:5,3:4) :: C*
- Declares A and B to be 3x4 matrices of datatype *real8*
- Declares C as a 8x2 array of default real type, with row indices ranging from -2 to 5 and column indices from 3 to 4
- Default indexing is from 1, not from 0
- Array bounds are *inclusive* of each end - unlike most other languages which exclude the upper bound
- What about the declaration *real, dimension(5:-2) :: D* ?

# Fortran 90

---

- Array sections: these work just like in Matlab, but with the **stride last**. So if

*real(kind=real8), dimension(3,5) :: A*

then section  $A(:,3:2,1::2) = A(1:3:2,1:5:2)$  refers to the matrix

$[A(1,1) \ A(1,3) \ A(1,5)]$

$[A(3,1) \ A(3,3) \ A(3,5)]$

Also like Matlab, you can use vector subscripts. So if

*integer, dimension(2) :: R = (/1,3/)* ! initialization of array

*integer, dimension(3) :: C = (/1,3,5/)*

then the array section  $A(R,C)$  is the same as  $A(:,3:2,1::2)$

- Can then pass  $A(R,C)$  as an argument to a procedure

# Fortran 90

---

- Array operations may use a **mask operation**, keyword *where*:

*where (x > 0) y = log(x)*

The general form of the *where* has an else clause:

*where (x > 0)*

*y = log(x)*

*elsewhere*

*y = -inf*

*end where*

# Fortran 90

---

- What if you call a function with argument  $x$  in one of the clauses?
  - All statements within a *where* must be array assignments

# Fortran 90: Array Classes

---

- Five classes of arrays are possible:
  - 1) explicit shape (also available in F77)
  - 2) assumed-shape
  - 3) automatic
  - 4) assumed size (also available in F77)
  - 5) deferred-shape (allocatable arrays)

# Explicit Shape Arrays

---

- Explicit shape have their dims passed as arguments to a subroutine (or passed in via modules or host info access)

```
subroutine explicit(A,B,m,n)  
  integer, intent(in) :: m,n  
  real, dimension(0:m+1, 0:n+1), intent(inout) :: A  
  real, dimension(m,n,3), intent(out) :: B  
  ...  
end subroutine explicit
```

- The **bounds** of the actual and dummy args need not match  
 Caller could have had *real, dimension(m+2,n+2) :: A*
- The **extents** (length of each dimension) should match

# Assumed Shape Arrays

---

- Assumed shape have their extents associated with actual arguments when the call is made

```
subroutine assumed(A,B)  
  real, dimension(0:, 0:), intent(inout) :: A  
  real, dimension(:, :, :), intent(out) :: B  
  n = ubound(B, dim=2) - 1  
end subroutine assumed
```

- Actual arguments must have same type and have same rank as their dummy arguments
- Can retrieve *lbound*, *ubound*, and *size* of each dimension.

# Automatic Arrays

---

- Automatic arrays are explicit shape arrays which are not dummy args and have at least one nonconstant index bound

```
subroutine automatic(B, n)
```

```
integer :: n
```

```
real, dimension(0:n+1, 0:n+1) :: A
```

```
real, dimension(size(B,1)) :: C
```

```
real, dimension(1000) :: D ! Not an auto array
```

```
end subroutine automatic
```

- Space is allocated dynamically on entry, deallocated on exit
- Can be passed to other routines in between

# Assumed Size Arrays

---

- Assumed size are the worst of all. All extents except last must be given explicitly; last is given as (\*) or (*lower\_bound* : \*)
- **Size** of actual argument and dummy argument must match
- **Shapes** of actual and dummy arguments need not match.
- So can declare *A(10,10,10)* and pass to a procedure which declares the dummy argument as *B(5, \*)*.
- Deprecated in F90; not sure about status in F2k but likely it will remain.

# Allocatable Arrays

---

program arrayexample

integer :: i, j, n, m ! newer style declaration of integers

real, dimension(:,:), allocatable :: A, B, C

real, dimension(:) , allocatable :: E

write(\*, "('Enter the size of A(n,m) :')", advance='no')

read(\*,\*) n, m

allocate (A(n, m), B(m, n), C(1:n,1:n) )

allocate ( E(size(C)) ) ! Creates rank-1 array, length n\*n

call define\_AB( A, B, n, m ) ! user-defined setup of A,B

C = matmul( A, B ) ! intrinsic function matrix multiply

print \*, 'C = ' ! Output of C with some default formatting

deallocate(A) ! deallocation order not important

...

# More Recent Fortran

---

- Modules (included blocks)
- User-defined types (like C structs)
- Pointers (nothing like C pointers)
- Optional arguments
- Interface blocks
- Other Fortran 2000 features

# Modules and Derived Types

---

- Example of a rational arithmetic module. Store a rational numbers as two integers, the numerator and denominator
- Should allow interoperating with standard integers
- Want to use the more natural notation  $rat2 = int * rat$  instead of a procedure call like  
$$rat2 = int\_rat\_multiply(int, rat)$$
- Syntax note: Fortran uses  $\%$  to reference a field within a user-defined type:  $mytype\%numerator$

# Modules and Derived Types

```
module rational_arithmetic
```

```
  type ratnum
```

```
    integer :: num, den
```

```
  end type ratnum
```

```
  interface operator(*)
```

```
    module procedure rat_rat, int_rat,  
    rat_int
```

```
  end interface
```

```
  private :: rat_rat, int_rat, rat_int
```

```
  contains
```

```
    type(ratnum) function rat_rat(lf, rt)
```

```
      type(ratnum), intent(in) :: lf,rt
```

```
      rat_rat%num = lf%num * rt%num
```

```
      rat_rat%den = lf%den * rt%den
```

```
    end function rat_rat
```

```
  type(ratnum) function int_rat(lf, rt)
```

```
    type(ratnum), intent(in) :: rt
```

```
    integer, intent(in) :: lf
```

```
    ...
```

```
  end function int_rat
```

```
end module rational_arithmetic
```

```
program main
```

```
  use rational_arithmetic
```

```
  integer :: i = 32
```

```
  type(ratnum) :: a, b, c
```

```
  a = ratnum(1,16); b = 2*a; c = b*3
```

```
  b = a*i*b*c
```

```
end program main
```

# Modules and Derived Types

---

- A derived type can have
  - a further derived type
  - the type currently being declared (usual linked list data structure mechanism)
- *sequence* attribute can force storage locations
- User defined types cannot contain *allocatable* entities. Restriction is not in F2k, but check your compiler to see if it supports this or not.
- Not all vendors support full F2k standard

# Pointers

---

- Not pointers in the C sense of the word, more like aliases
- Restrictions on pointers allows compiler to generate efficient array code
- *ptr => y*      ! pointer assignment (aliasing)
- *ptr = x*      ! value assignment; sets  $y = x$
- *nullify(ptr)*      ! disassociate ptr and y
- Thing pointed at must have the *target* attribute  
*real, dimension(1:n, 1:n), target :: x*

# Pointers

---

- *real, dimension(:, :), pointer :: pa, pb*
  - *pa* is a pointer for a 2D array of reals
  - can associate memory with a pointer  
*allocate(pb(0:n, 0:2\*n\*n), stat=ierr)*
  - cannot have *pa* point to a rank-1 array, or a real scalar
- *pa => A(-k:k, -j:j)*
  - *pa* is now an alias for (part of) A.
- Pointers are **automatically dereferenced** when used:  
*pa(15:25, 5:15) = pa(10:20, 0:10) + 1.0*
- What about the overlapping entries in the last statement?

# Optional Arguments

---

```
subroutine optargs(scale, x)  
  real, intent(in) :: x  
  real, intent(in), optional :: scale  
  real :: actual_scale  
  actual_scale = 1.0  
  if (present(scale)) actual_scale = scale  
  ...  
end subroutine optargs
```

# Interface Blocks

---

*interface*

*subroutine assumed(A, B)*

*real, dimension(-1:, -1:), intent(inout) :: A*

*real, dimension(:, :, :), intent(out) :: B*

*end subroutine assumed*

*end interface*

- Mandatory for external non-module procedures with
  - *optional* or *keyword* arguments
  - *pointer* or *target* arguments
  - assumed shape or sliced array arguments
  - array or pointer valued procedures
- Always use; compiler can check both caller and callee

# Function Overloading

---

- User defined overload set

```
interface zero  
  module procedure zero_int  
  module procedure zero_real  
end interface zero
```

- Generic name *zero* is associated with specific names *zero\_int* and *zero\_real*
- The one called is based on **argument signature**, like C++
- **Signature** is the sequence of types in the argument list

# Fortran95 and Fortran 2000

---

- All Fortran compilers (except gnu) support Fortran 95 standard – minor additions over Fortran 90
- No compilers yet (2008) fully implement F2k
- Extensions are in the really advanced category
  - Unlikely to be adopted by many apps users for a few years, if ever
  - Attempts to be more object-oriented

# Fortran 95: Minor Enhancements over F90

- *forall* statement and construct
- *pure* and *elemental* procedures
  - *pure* means no side effects; prefix for function
  - *elemental* is a *pure* function with only scalar dummy arguments. When called with actual arguments being conformant arrays, it operates componentwise.
- structure and pointer default initialization

# Fortran 2000

- Exception handling
- Standard C interoperability (main intended for OS calls)
- Allows derived data types to have allocatable components and parameterized types (*kinds*), a limited form of C++'s templating
- Asynchronous and derived type I/O
- Object oriented features (as long as they don't interfere with performance capabilities)

# More Fortran 2000 Additions

- Parameterized derived types: can use *kinds* and have allocatable entities

```
type matrix(precision, m, n)
```

```
integer, kind    :: precision
```

```
integer, nonkind :: m, n
```

```
real(precision) :: A(m, n)
```

```
end type
```

```
type(matrix(kind(0.0d0), 20, 20) :: G
```

```
type(matrix(kind(0.0), :, :), allocatable :: H
```

# More Fortran 2000 Additions

- Procedure pointer w/ same interface as procedure *Bessel*:

*procedure (Bessel), pointer : p => null( )*

- Abstract interfaces to allow ptr w/o procedure predefined

*abstract interface*

*real function Whatever(x,y)*

*real, intent(in) :: x, y*

*end function*

*end interface*

# More Fortran 2000 Additions

- **Type-bound** procedures
  - analogous to class methods in C++ and Java
  - reference like a type's data fields, using % operator

# Fortran 2000 I/O Additions

- **asynchronous transfer** (transfer = read or write)
- stream access
- user specified **transfer ops for derived types**
- user control of rounding during format conversions
- named constants for preconnected units (stdin, stderr, stdout)
- **flush** statement
- regularization of keywords
- access to error messages

# Fortran 2000 Odds and Ends

- Recursive functions allowed. Specify via the *recursive* attribute when declared
- Default: pass arguments by reference. Most compilers allow
  - *%REF( )* arguments (passes in character arrays *without* the implicit appending of the character array's size)
  - *%VAL( )* arguments (pass by value)

# Legacy C-Fortran Interoperability

---

# Naming Conventions

---

- Fortran compiler appends an **underscore** to the names of procedures (functions or subroutines). Except for IBM.
- If calling Fortran **foo(...)** from C, use name **foo\_( ... )** in C
- If calling a C function from Fortran, the C function must have an underscore after its name.
  - C function defines **bar\_(...)**
  - Fortran calls **bar(...)**
- Behaviour now can be controlled via most compilers with some options; use man pages to search for 'underscore'
- Beware of Fortran's **case insensitivity**
  - Some compilers automatically **upper case** all letters in a procedure name
  - Some **lower case** them

# Naming Conventions

---

- Can use C preprocessor (cpp) to control some weirdness and architecture dependencies

```
#ifdef REMOVE_
```

```
#define readmt_ readmt
```

```
#endif
```

and in the makefile insert the line

```
REMOVE_ = true
```

if the machine/compiler does not use the underscore convention.

# Legacy C-Fortran interoperability

---

- To have the `cpp` run automatically on Fortran files, some platforms require a `.F` suffix instead of `.f`, `.f90`, ...
  - Can handle by creating a soft link with same name but different suffix for all Fortran files
  - Can handle by explicitly running `.f` files through `cpp` in your makefile
- Weakness: Fortran does not specify a preprocessor like CPP. Some systems have FPP (Fortran pre-processor), but usually not needed:
  - `include` is a Fortran statement, handles `.h` files
  - `use module_name` handles other cases
  - `allocate( )` removes need for compile-time constants

# Legacy C-Fortran interoperability

---

- Standard datatypes map well
  - C `int` = Fortran integer
  - C `double` = Fortran `real(kind(1.0d0))`
  - C `char` = Fortran *single* character
- Other datatypes do not map portably
  - C `complex`  $\neq$  Fortran complex (same for STL of C++)
  - C `string`  $\neq$  Fortran string (where “string” means something like declaration *`character(len=12) :: fstr`*)

# C-Fortran Strings

---

- C appends a `\0` (null character) to every string
- Fortran keeps a **hidden descriptor** containing the string's declared length
- When calling a procedure with a character variable, Fortran passes along the length as a hidden argument
- Solutions:
  - Explicitly append `\0` to any string passed to a C function
  - when calling Fortran, explicitly pass the string's length as an argument
  - **don't pass strings between languages, just chars**
- Similar problems with Java and C++ string interoperability.

# Array sections

---

- In Fortran: *call foo\_(A(1:100:2), n)* . can be handled by a compiler vendor at run-time system by
  - create a **copy** of 50 contiguous items, then copy it back to the correct locations in A on return
  - pass along a **hidden array descriptor**
  - sometimes one, sometimes another (based on size)
- **Don't pass array sections between languages**
- What if the Fortran code is a legacy one?
- What if it is a licensed package to which you cannot get access to the source code to modify?

# Other considerations

---

- C function *float foo(x)* may return a double instead
- Default Fortran passes arguments by reference. Default C passes by value. **So always pass addresses**
- Arrays indexing: C starts from 0, Fortran starts from wherever you tell it to start (default: 1)
- C has **row-major** storage (last index varies fastest)
- Fortran has **col-major** storage (first index varies fastest)
- Fortran logical types *.true.* and *.false.* may not port to C++ booleans
- Fortran uses *iargc* and *getarg* for number of args, and args.

# Compiling Mixed C-Fortran Code

---

- Fortran automatically links in many libraries that C doesn't
- If you have a Fortran `main` program, use the Fortran linker
  - keep in mind: a Fortran program need not be named *main*
- If you have a C/C++ *main* function, need to add other libraries like `-lftn` or `-lF90`, `-lg2c`, etc. Check the man pages
- C++ name mangling: from C++, declare your Fortran procedures as *extern "C" { ... }* just as you would for a C function
- A `blas.h` header file usually declares those externs in C++

# Language Standard C-Fortran Interop

- Interoperability is via `ISO_C_BINDING` module

*C\_INT = int*

*C\_SIZE\_T = size\_t*

*C\_CHAR = char (but not strings)*

*C\_PTR = void \* (interoperates w/ any C ptr)*

- Use the function `C_LOC(x)` to find addresses
  - *x* can be a procedure, pointer to procedure, a variable w/ interoperable type, scalar with *target* attribute and already allocated

# F2k and Mixed C-Fortran Code

- Derived types must be explicitly “bindable”

*type, bind(C) :: stuff*

*....*

*end type stuff*

- All subcomponents of the type must be interoperable, not a pointer, and not allocatable
- Equivalent to a corresponding C struct
- Not valid: C structs with union, bit fields, flexible array components

# F2k and Mixed C-Fortran Code

- Fortran arrays interoperable with C arrays, but with reversal of subscript order
  - *real(kind(0.0d0)) :: A(7, 4:9)*
  - *double A[6][7]*
- functions can specify pass by value for a dummy argument
  - integer, value :: x*
- This allows a compiler do a one-way copy, and the user to avoid side-effects
  - note that *intent(in)* also prevents side-effects

# F2k and Mixed C-Fortran Code

- Make Fortran functions available by name to a C process using `bind(C)`:

*function, bind(C) :: bar(x,y)*

- Can give a different C name at same time:

*function, bind(C, name='foo') :: bar(x,y)*

- The name is available in C without underscore convention, as *bar* (first case) or *foo* (second case)

# Current and Future Efforts

---

- CCA project has [Babel](#), which takes a generic interface definition and then creates bindings for multiple languages. Currently has F90, F77, C, C++, Java, Python.
- Other automatic "binding" generator: [SWIG](#)
- Microsoft's [Common Language Infrastructure](#) potentially will allow languages to be compiled into a shared, single intermediate language which is then further compiled into machine code
- In scientific computing Fortran still dominates - not just as legacy code, but as most lines of code being written now.