Fortran95 and Fortran2003

Tips and Techniques to Help Build Robust, Maintainable Code

1. Introduction

Paul van Delst

Betty Petersen Memorial Library
Technical Seminar Series
Introduction

• The goal of these Fortran seminars is quite simple:
  – Describe Fortran95 features.
  – Introduce Fortran2003 syntax and features.
  – Discuss how can we use these features in our software development to create high quality, robust, maintainable (extensible?) code?

• What is your experience with f95/2003?

• What do you want to know about f95/2003?
Overview

• What we’ll be covering this time
  – Kind types
  – Attributes
  – Derived types
  – Array syntax

• Most of these topics are fundamental, so you may already know about them and use them.

• The subject matter isn’t covered linearly, particularly with respect to Fortran2003 features and more esoteric Fortran95 usage.

• Please ask questions for clarification.
**Kind types**

- New intrinsics to allow user to select the “kind” of integer or real variable.
- `SELECTED_INT_KIND(I)`
  - Return the kind value of the *smallest* integer type that can represent all values $-10^I$ to $10^I$
- `SELECTED_REAL_KIND(P, R)`
  - Return the kind value of a real data type with decimal precision of *at least* $P$ digits and exponent range greater than *at least* $R$.
- Note the “smallest” and “at least” in the definitions. Supported kind types still depend on hardware and software.
Kind types - Integer

- Some examples:

<table>
<thead>
<tr>
<th>Integer</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECTED_INT_KIND(1)</td>
<td>1</td>
</tr>
<tr>
<td>SELECTED_INT_KIND(2)</td>
<td>1</td>
</tr>
<tr>
<td>SELECTED_INT_KIND(3)</td>
<td>2</td>
</tr>
<tr>
<td>SELECTED_INT_KIND(4)</td>
<td>2</td>
</tr>
<tr>
<td>SELECTED_INT_KIND(5)</td>
<td>4</td>
</tr>
<tr>
<td>SELECTED_INT_KIND(6)</td>
<td>4</td>
</tr>
<tr>
<td>SELECTED_INT_KIND(7)</td>
<td>4</td>
</tr>
<tr>
<td>SELECTED_INT_KIND(8)</td>
<td>4</td>
</tr>
<tr>
<td>SELECTED_INT_KIND(9)</td>
<td>4</td>
</tr>
<tr>
<td>SELECTED_INT_KIND(10)</td>
<td>8</td>
</tr>
</tbody>
</table>

These specify the size of the integer type.

These are the resultant integer \textit{KIND TYPES}.

NOTE: These values are \textit{not} portable. Different compilers may use different values.

Note the repeated kind type values for different sizes. Not all integer representations are available. E.g. \(\pm10^5\) to \(\pm10^9\) values are possible with a single kind type ("long integer").
Kind types - Real

• Some examples:

<table>
<thead>
<tr>
<th>Real</th>
<th>KIND TYPES</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECTED_REAL_KIND(1) = 4</td>
<td></td>
</tr>
<tr>
<td>SELECTED_REAL_KIND(2) = 4</td>
<td></td>
</tr>
<tr>
<td>SELECTED_REAL_KIND(3) = 4</td>
<td></td>
</tr>
<tr>
<td>SELECTED_REAL_KIND(4) = 4</td>
<td></td>
</tr>
<tr>
<td>SELECTED_REAL_KIND(5) = 4</td>
<td></td>
</tr>
<tr>
<td>SELECTED_REAL_KIND(6) = 4</td>
<td></td>
</tr>
<tr>
<td>SELECTED_REAL_KIND(7) = 8</td>
<td></td>
</tr>
<tr>
<td>SELECTED_REAL_KIND(8) = 8</td>
<td></td>
</tr>
<tr>
<td>SELECTED_REAL_KIND(9) = 8</td>
<td></td>
</tr>
<tr>
<td>SELECTED_REAL_KIND(10) = 8</td>
<td></td>
</tr>
</tbody>
</table>

These specify the decimal precision of the real type.

These are the resultant real KIND TYPES.

Again, these values are not portable. Different compilers may use different values.

Note the repeated kind type values for different precisions. As with integer types, not all real representations are available. Remember the “at least” part of the definition.
Kind types - Portability issues

- You’re probably familiar with the f77 extension: `INTEGER*4` or `REAL*8` where the “4” and “8” indicate byte size.
- These are *not* synonymous with the f95 syntax: `INTEGER(4)`, or `REAL(8)` where the “4” and “8” are kind types. Some compilers use
  `INTEGER(3)`, or `REAL(2)` to represent the same types.
- Never use literal constants for kind types.
- Always use parameterised kind types derived from the `SELECTED_*_KIND()` intrinsics.

Do not do this!
Kind types - the Module

MODULE Type_Kinds
  ! No implicit typing
  IMPLICIT NONE
  ! Explicit visibility declaration
  PRIVATE
  PUBLIC :: Byte, Short, Long
  PUBLIC :: Single, Double
  PUBLIC :: ik, rk

  ! Integer kinds
  INTEGER, PARAMETER :: Byte  = SELECTED_INT_KIND(1) ! Byte
  INTEGER, PARAMETER :: Short = SELECTED_INT_KIND(4) ! Short
  INTEGER, PARAMETER :: Long  = SELECTED_INT_KIND(8) ! Long

  ! Floating point kinds
  INTEGER, PARAMETER :: Single = SELECTED_REAL_KIND(6) ! Single
  INTEGER, PARAMETER :: Double = SELECTED_REAL_KIND(15) ! Double

  ! Generic kinds
  INTEGER, PARAMETER :: ik = Long   ! Generic integer kind
  INTEGER, PARAMETER :: rk = Double ! Generic real kind
END MODULE Type_Kinds

Use of generic kind types makes precision changes easier.

Always do this!
Self-documentation!
**Kind types – Using the Module**

PROGRAM Display_Type_Kinds

! Use the module
USE Type_Kinds

! No implicit typing
IMPLICIT NONE

! Declare variables
INTEGER(Byte) :: bi
INTEGER(Short) :: si
INTEGER(Long) :: li
REAL(Single) :: sr
REAL(Double) :: dr

WRITE(*,'("Huge(bi)="\,i10,5x,\,i10")') HUGE(bi), 10(Byte)**1 - 1
WRITE(*,'("Huge(si)="\,i10,5x,\,i10")') HUGE(si), 10(Short)**4 - 1
WRITE(*,'("Huge(li)="\,i10,5x,\,i10")') HUGE(li), 10(Long)**8 - 1
WRITE(*,'("Prec,Range(sr)="\,i5,1x,\,i5")') PRECISION(sr), RANGE(sr)
WRITE(*,'("Prec,Range(dr)="\,i5,1x,\,i5")') PRECISION(dr), RANGE(dr)

END PROGRAM Display_Type_Kinds

**How to use modules in your code.**

**How to declare variables of different kinds.**

**New intrinsics to inquire about a datatype:**

HUGE(), PRECISION(), RANGE().
**Kind types – USEing the Module**

```fortran
PROGRAM Display_Type_Kinds
  ! Use the module
  USE Type_Kinds
  ! No implicit typing
  IMPLICIT NONE
  ! Declare variables
  INTEGER(Byte) :: bi
  INTEGER(Short) :: si
  INTEGER(Long) :: li
  REAL(Single) :: sr
  REAL(Double) :: dr

  WRITE(*,'("Huge(bi)=",i10,5x,i10)') HUGE(bi),10_Byte**1 - 1
  WRITE(*,'("Huge(si)=",i10,5x,i10)') HUGE(si),10_Short**4 - 1
  WRITE(*,'("Huge(li)=",i10,5x,i10)') HUGE(li),10_Long**8 - 1
  WRITE(*,'("Prec,Range(sr)=",i5,1x,i5)') PRECISION(sr), RANGE(sr)
  WRITE(*,'("Prec,Range(dr)=",i5,1x,i5)') PRECISION(dr), RANGE(dr)
END PROGRAM Display_Type_Kinds
```

This is how you type literal constants.

<table>
<thead>
<tr>
<th>Type</th>
<th>Range</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hi(bi)</td>
<td>127</td>
<td>9</td>
</tr>
<tr>
<td>Hi(si)</td>
<td>32767</td>
<td>99999</td>
</tr>
<tr>
<td>Hi(li)</td>
<td>2147483647</td>
<td>999999999</td>
</tr>
<tr>
<td>Sr</td>
<td>6</td>
<td>37</td>
</tr>
<tr>
<td>Dr</td>
<td>15</td>
<td>307</td>
</tr>
</tbody>
</table>

Fortran95/2003 Seminar Series. 1: Introduction

December, 2008
Kind types - Literal constants

- Always use the kind type when defining and assigning real literal constants. If we have a double precision kind type “dp”,

```fortran
REAL(dp), PARAMETER :: MW_DRYAIR = 28.9648_dp
```

```fortran
REAL(dp) :: g_per_g, g_per_kg
```

```fortran
g_per_g = 1.0e-03_dp * g_per_kg
```

- The “_dp” suffix on the literal constants ensures the constant has the same precision as its data type. Doing something like,

```fortran
    g_per_kg = 8.754
```

assigns a *single* precision constant to a *double* precision variable. To be consistent, do

```fortran
    g_per_kg = 8.754_dp
```
Kind types - Literal constants(2)

PROGRAM Display_Literals
    USE Type_Kinds
    IMPLICIT NONE
    REAL(Double) :: var1, var2, var3
    var1 = 10.0_Double
    var2 =  3.1415927        * var1
    var3 =  3.1415927_Double * var1
    WRITE(*,'("var2 = ",f19.15)') var2
    WRITE(*,'("var3 = ",f19.15)') var3
END PROGRAM Display_Literals

$:/~Fortran/Type_Kinds $ gfortran -c Type_Kinds.f90
$:/~Fortran/Type_Kinds $ gfortran Display_Literals.f90
$:/~Fortran/Type_Kinds $ a.out

var2 =  31.415927410125732
var3 =  31.415927000000000

The precision of the result is dependent on the precision of the literal, regardless of all the type declarations.

The difference is one literal is typed, the other is not
Attributes

- Attributes are properties that can be specified for an object (variable, argument, function, etc)
- There are 12 attributes in Fortran95. Fortran2003 adds 5 more.
- Some attributes in Fortran95 are carried over from FORTRAN77 statements
  - DIMENSION, EXTERNAL, INTRINSIC, PARAMETER, and SAVE
- We’ll look at some “old” attributes but using new syntax, and then cover new attributes.
Attributes - “Old”

- **PARAMETER.** FORTRAN77 syntax:
  
  ```fortran
  INTEGER N
  PARAMETER (N=10)
  ```

  Fortran95 syntax:
  
  ```fortran
  INTEGER, PARAMETER :: N=10
  ```

- **SAVE.** Same syntax change as for parameters:
  
  ```fortran
  INTEGER, SAVE :: my_int
  ```

  **Important:** _initialisation implies save!_

  ```fortran
  INTEGER :: my_int = 0
  ```

  This declaration is equivalent to:
  
  ```fortran
  INTEGER, SAVE :: my_int = 0
  ```

  Double colon required for inline attribute specification

  Required for initialisation
Attributes - Allocatable(1)

- **ALLOCATABLE.** Fortran finally has dynamic storage allocation! Type declaration example,

  ```fortran
  REAL, ALLOCATABLE :: x (: , :)
  ```

  Allocation via the `ALLOCATE` statement...

  ```fortran
  ALLOCATE (x (N , N), STAT= alloc_stat)
  ```

  Allocation error if ≠ 0. Please use.

  …and deallocation via the obvious,

  ```fortran
  DEALLOCATE (x, STAT= alloc_stat)
  ```

  Fortran95 standard stipulates that local allocatable arrays are deallocated upon exit.

  Fortran90 does not!
Attributes - Allocatable(2)

• An allocatable array has both a *definition* status and an *allocation* status.
  – Definition status is the same as for other objects and depends on whether the object has had its value established. In general, objects start their lives in an undefined state.
  – Allocation status lets you know whether an allocatable array is or is not allocated. This can be determined with a new intrinsic function like so,

\[
\text{IF (ALLOCATED(x)) THEN...}
\]

The allocation status of an allocatable array is *never* undefined.

Remember this!
Attributes - Target

• **TARGET**. Indicates the variable (or argument) can be pointed to by a Fortran pointer. Type declaration example,
  
  ```fortran
  REAL, TARGET :: x(N,N), y(N)
  ```

• **Why require this?**
  – To provide compilers with information that can be utilised to produce efficient code.
  – Also makes clear the intent of the programmer that the variables/arguments in question can be aliased.

• **You cannot access a variable/argument via a pointer unless it has the TARGET attribute.**
Attributes - Pointer(1)

• **POINTER.** Now easier to shoot oneself in the foot in Fortran too! (Just kidding).

• Fundamental differences between Fortran pointers and those of other languages.
  – There is no mechanism for indicating that a pointer to one data type is to be treated as if it were a pointer to different data type.
  – No generic pointer type.
  – No pointer arithmetic.
Attributes - Pointer(2)

- Type declaration example,
  
  ```fortran
  REAL, POINTER :: ptr(:) => NULL()
  
  This is very important. You should always do this!
  
  Fortran90 does not have this capability!
  
  - `ptr` can *only* point to rank-1 real arrays, e.g.
    
    ```fortran
    REAL, TARGET :: x(N,N), y(N)
    ptr=>x(:,1)
    ptr=>y
    
    - Can also allocate new space directly, e.g.
      
      ```fortran
      ALLOCATE(ptr(N), STAT=alloc_stat)
      ```
Attributes - Pointer(3)

- Similar to allocatables, a pointer has both a definition status and an association status.
  - Association status lets you know whether a pointer is or is not associated with a target. This can be determined with a new intrinsic function like so,
    
    \[
    \text{IF (ASSOCIATED(ptr)) THEN...}
    \]

- Unless you initialise the pointer, the association status is undefined.

- In this case, it is illegal to query the association status of the pointer.
Attributes - Pointer(4)

• **Important**: a pointer with no initialisation,
  \[ \text{REAL, POINTER :: ptr(:)} = \text{NULL()} \]
  means its association status is *undefined*.

  ![Always do this!]

• In this case, testing its association status is *illegal*,
  \[ \text{IF (ASSOCIATED(ptr)) THEN...} \]

• You should either initialise in the declaration as shown before, or immediately after by using
  \[ \text{NULLIFY(ptr)} \quad \text{[Fortran90/95/2003]} \]
  or
  \[ \text{ptr=}\text{NULL()} \quad \text{[Fortran95/2003 only]} \]
Attributes - Intent

- **INTENT**. Allows programmer to explicitly state whether subprogram arguments are input, output, or both. Good self-documentation.

- Subprogram example,

```
SUBROUTINE mysub(a, b, x, y)
    REAL, INTENT(IN)    :: a, b
    REAL, INTENT(IN OUT) :: x
    REAL, INTENT(OUT)    :: y

    -- You cannot change the value of the "a" and "b" dummy arguments in mysub().
    -- Any value the actual argument "y" had prior to calling mysub() is lost - assignment is expected.
```
Attributes - Optional(1)

• **OPTIONAL.** Allows programmer to specify dummy arguments as optional.

• Subprogram example,

```
SUBROUTINE mysub(a,b,x,c)
  REAL,          INTENT(IN)  :: a,b
  REAL,          INTENT(OUT) :: x
  REAL, OPTIONAL, INTENT(IN) :: c

  IF (PRESENT(c)) THEN...
```

– You can test if the actual argument was passed using a new intrinsic,

```
IF (PRESENT(c)) THEN...
```
Attributes - Optional(2)

- There is no mechanism for specifying a default value for an omitted argument. Use a local variable,

```fortran
SUBROUTINE mysub(...other arguments..., c)
    REAL, OPTIONAL, INTENT(IN) :: c
    REAL :: c_local
    ...
    IF (PRESENT(c)) THEN
        c_local = c
    ELSE
        c_local = default value
    END IF
```
Attributes - Optional(3)

- Recommended to use keyword arguments in calls to distinguish mandatory and optional arguments,
  
  ```fortran
  CALL mysub(a, b, x, c=c)
  ```

  If the convention is adopted, then the calling statement indicates which arguments are optional.

- Explicit interface is required.
  - Use of optional attribute for dummy argument.
  - Use of keyword form in passing actual arguments

We’ll cover interfaces later on.
Attributes - Accessibility

- These attributes control accessibility of entities (type definitions, variables, procedures, parameters) and are inherently related to the use of modules so I’ll only mention them briefly - we’ll look more closely at them when we cover modules.

- **PUBLIC**. The default. Declares module entities are accessible outside the module via use association.

- **PRIVATE**. Prevents access of entities outside a module by use association.
Attributes - Fortran2003

- **ASYNCHRONOUS**. Specifies a variable may be used in asynchronous I/O. Used to facilitate compiler optimisation.
- **BIND(C)**. Pertains to interoperability with the C language.
- **PROTECTED**. Allows a module variable to be visible (i.e. not private), but not definable outside the module.
- **VALUE**. Specifies a form of argument association for dummy arguments.
- **VOLATILE**. Specifies a variable may be used or modified by means not specified in the program. E.g. if some other program modifies shared memory.
Derived Types

- More commonly known as structures. Also referred to as user defined types.
- Defined within a `TYPE...END TYPE` construct.
- May contain any combination of intrinsic or derived type components.
- Limitations on component attributes:
  - Only `POINTER` or `DIMENSION` attributes allowed.
  - Fortran2003 (or TR15881 compliant) compiler allows the `ALLOCATABLE` attribute.
- Fortran2003 also greatly increases the utility of derived types for OOP. We’ll cover those changes in a later seminar.
Derived Types - Simple Example

Give your type a name. Note: my personal style is to append "_type" so I know it's a derived type name.

```fortran
TYPE :: whereami_type
  REAL(rk) :: lat
  REAL(rk) :: lon
  REAL(rk) :: alt
  TYPE(date_type) :: date
  CHARACTER(80) :: location
END TYPE whereami_type
```

The components

Other derived types can be components

Naming the END TYPE is not mandatory, but recommended (e.g. if you have scripts for creating documentation)
Derived Types - Assignment(1)

- Can assign to a derived type two ways:
  - Regular assignment
    ```fortran
    TYPE(whereami_type) :: x
    x%lat = 38.89_rk
    x%lon = 77.02_rk
    x%alt = 6.1_rk
    x%date%year = 2008
    x%date%month = 12
    x%date%day = 4
    x%location = "Washington, DC"
    ```

  The component selector is "%". Many other languages use ".". Why not Fortran? Think about .AND., .OR., .NOT., etc.

Structure components are referenced recursively.
Derived Types - Assignment(2)

- Structure constructors

```fortran
x = whereami_type(38.89_rk, &
    77.02_rk, &
    6.1_rk, &
    date_type(2008,12,4), &
    "Washington, DC")
```

The derived type name

The sequence of component values. Must agree in number and order.

An embedded constructor.
Derived Types - Parameters

- A structure constructor is how you specify a derived type parameter,

```fortran
TYPE(whereami_type), PARAMETER :: x = &
  whereami_type(38.89_rk,&
    77.02_rk,&
    6.1_rk,&
    date_type(2008,12,4),&
    "Washington, DC")
```
Derived Types - Default Init(1)

- Default initialisation of our simple structure,

```
TYPE :: whereami_type
  REAL(rk) :: lat = -999.0_rk
  REAL(rk) :: lon = -999.0_rk
  REAL(rk) :: alt = -999.0_rk
  CHARACTER(80) :: location = ""
  TYPE(date_type) :: date = date_type(0,0,0)
END TYPE whereami_type
```

Note that this initialisation overrides any default initialisation in the date_type definition. The “highest level” initialisation wins.
Derived Types - Default Init(2)

• **Warning #1.** Keep dummy argument INTENT in mind.
  - Let’s say we had a subroutine interface like so
  
  ```fortran
  SUBROUTINE get_map(x)
    TYPE(whereami_type), INTENT(OUT) :: x
  
  - Everytime `get_map()` is called, `x` is reinitialised to its default value because of the `INTENT(OUT)` attribute of the dummy argument.
  
  - For arrays, or large structures, this can be expensive. Not dangerous, just expensive.
Derived Types - Default Init(3)

- **Warning #2.** Keep dummy argument INTENT in mind. (Sound familiar?)
  - Let’s say we have the derived type
    ```fortran
    TYPE :: ptr_type
        REAL(rk), POINTER :: p(:) => NULL()
    END TYPE ptr_type
    ```
  - And let’s allocate an instance,
    ```fortran
    TYPE(ptr_type) :: ptr
    ALLOCATE(ptr%p(100000))
    ```
  - And let’s use it in this subroutine,
    ```fortran
    SUBROUTINE futz_ptr(ptr)
        TYPE(ptr_type), INTENT(OUT) :: ptr
        ptr = p(:) => NULL()
    END SUBROUTINE futz_ptr
    ```
  - What will happen? *Memory leak!*

`ptr` is reinitialised, so our allocated memory is no longer reachable.
Derived Types - Example 2

TYPE :: CRTM_Atmosphere_type
  ! Dimension values
  INTEGER :: n_Layers = 0  ! K dimension
  INTEGER :: n_Absorbers = 0  ! J dimension
  INTEGER :: n_Clouds = 0  ! Nc dimension
  INTEGER :: n_Aerosols = 0  ! Na dimension
  ! Number of added layers
  INTEGER :: n_Added_Layers = 0
  ! Climatology model associated with the profile
  INTEGER :: Climatology = INVALID_MODEL
  ! Absorber ID and units
  INTEGER, POINTER :: Absorber_ID(:) => NULL() ! J
  INTEGER, POINTER :: Absorber_Units(:) => NULL() ! J
  ! Profile LEVEL and LAYER quantities
  REAL(fp), POINTER :: Level_Pressure(:) => NULL()  ! 0:K
  REAL(fp), POINTER :: Pressure(:)       => NULL()  ! K
  REAL(fp), POINTER :: Temperature(:)    => NULL()  ! K
  REAL(fp), POINTER :: Absorber(:,:)     => NULL()  ! K x J
  ! Clouds associated with each profile
  TYPE(CRTM_Cloud_type),   POINTER :: Cloud(:)   => NULL()  ! Nc
  ! Aerosols associated with each profile
  TYPE(CRTM_Aerosol_type), POINTER :: Aerosol(:) => NULL()  ! Na
END TYPE CRTM_Atmosphere_type

TYPE :: CRTM_Cloud_type
  ! Dimension values
  INTEGER :: n_Layers = 0  ! K dimension
  INTEGER :: n_Added_Layers = 0
  ! Cloud type
  INTEGER :: Type = NO_CLOUD
  ! Cloud state variables
  REAL(fp), POINTER :: Effective_Radius(:)   => NULL() ! K
  REAL(fp), POINTER :: Effective_Variance(:) => NULL() ! K
  REAL(fp), POINTER :: Water_Content(:)      => NULL() ! K
END TYPE CRTM_Cloud_type

TYPE :: CRTM_Aerosol_type
  ! Dimensions
  INTEGER :: n_Layers = 0  ! K dimension
  INTEGER :: n_Added_Layers = 0
  ! Aerosol type
  INTEGER :: Type = NO_AEROSOL
  ! Aerosol state variables
  REAL(fp), POINTER :: Effective_Radius(:) => NULL() ! K
  REAL(fp), POINTER :: Concentration(:)    => NULL() ! K
END TYPE CRTM_Aerosol_type
Derived Types - Example 3

TYPE :: CRTM_CSVariables_type
PRIVATE
! The interpolation data
TYPE(CSinterp_type) :: csi(MAX_N_LAYERS, MAX_N_CLOUDS)
! The interpolation result
REAL(fp), DIMENSION(MAX_N_LAYERS, MAX_N_CLOUDS) :: ke
REAL(fp), DIMENSION(MAX_N_LAYERS, MAX_N_CLOUDS) :: w
REAL(fp), DIMENSION(MAX_N_LAYERS, MAX_N_CLOUDS) :: g
REAL(fp), DIMENSION(0:MAX_N_LEGENDRE_TERMS, MAX_N_PHASE_ELEMENTS, MAX_N_LAYERS, MAX_N_CLOUDS) :: pcoeff
! The accumulated volume scattering coefficient
REAL(fp), DIMENSION(MAX_N_LAYERS) :: Total_bs
END TYPE CRTM_CSVariables_type

TYPE :: CSinterp_type
! The interpolating polynomials
TYPE(LPoly_type) :: wlp, xlp, ylp
! The LUT interpolation indices
INTEGER :: i1, i2, j1, j2, k1, k2
! The LUT interpolation boundary check
LOGICAL :: f_outbound, r_outbound, t_outbound
! The interpolation input
REAL(fp) :: f_int, r_int, t_int
! The data to be interpolated
REAL(fp) :: f(NPTS), r(NPTS), t(NPTS)
END TYPE CSinterp_type

TYPE :: LPoly_type
INTEGER :: Order=ORDER
INTEGER :: nPts =NPOLY_PTS
! Left and right side polynomials
REAL(fp) :: lp_left(NPOLY_PTS) = ZERO
REAL(fp) :: lp_right(NPOLY_PTS) = ZERO
! Left and right side weighting factors
REAL(fp) :: w_left = ZERO
REAL(fp) :: w_right = ZERO
! Polynomial numerator differences
REAL(fp) :: dxi_left(NPOLY_PTS) = ZERO
REAL(fp) :: dxi_right(NPOLY_PTS) = ZERO
! Polynomial denominator differences
REAL(fp) :: dx_left(NPOLY_PTS) = ZERO
REAL(fp) :: dx_right(NPOLY_PTS) = ZERO
END TYPE LPoly_type
Array Syntax (1)

- Array syntax has been available since Fortran90. Many powerful, and convenient, features.

- Array constructors.
  
  \[
  \text{REAL :: } x(4) = (/ 3.7, 4.6, 7.2, 6.9 /) \quad [f90/95/2003] \\
  \text{INTEGER :: } j(N) = [(i*2, i=1,N)] \quad [f2003 \text{ only}]
  \]

- Subscript triplets specify array sections.
  - Backwards access, \( j(N:1:-1) \)
  - Every other element, \( j(1:N:2) \)
  - Elements are optional, \( j(4:N-3), j(4,:), j(:N-1) \)

- Vector Subscripts.
  If \( k = (/ 7, 4, 10 /) \), then \( j(k) \equiv (/ 14, 8, 20 /) \)
Array Syntax (2)

• This is a pet peeve of mine. Given an array,
  \[
  \text{REAL :: } x(N)
  \]
that will be assigned a value, some people recommend,
  \[
  x(:)=1.0
  \]
rather than
  \[
  x=1.0
  \]
with the idea that the “(:)” indicates to readers that “x” is an array.
• But, \textit{x (:)} is an array section, an expression, derived from the actual array, \textit{x}.
• 99.99% of the time it probably doesn’t matter, but just remember they are not the same thing.
Summary

• What we’ve looked at
  – Kind types and how to use them.
  – Attributes in type definitions.
  – Derived type definitions.
  – Quick look at array syntax.

• What to do with this information?
  – Apply it to new code (if it ain’t broke…), or
  – Refactor old code (let me tell you about unit testing…)

• Was the information useful? What would you like to see in any future Fortran seminars? Volunteers?
  – Let me know: paul.vandelst@noaa.gov
Next Time

- Concept of scope
- Interfaces
  - Difference between explicit and implicit interfaces
- Modules
  - Create your own generic procedures
  - Create your own operators
- Block constructs
- Input and output processing
Where to Get More Information

- EMC Fortran Forum. Post questions, and/or discoveries. It’s not too helpful for non-NCEP folks without VPN so maybe setup similar on library website?

Forum for language issues.

Forum for compiler and compilation issues.
Where to Get More Information

• Many online guides.
  – Google is your friend.

• comp.lang.fortran newsgroup.
  – Many members of the current and past standards committee, and some compiler vendors, frequent it.

• Books!
  – The ones below I use almost daily.