Kind and Precision
(aka Parameterized Data Types)
Background

• Fortran 77 had a problem with numeric portability. A default `REAL` might support numbers up to $10^{68}$ on one machine and up to $10^{136}$ on another.

• Fortran 90/95/2003 includes a `KIND` parameter which provides a way to parameterize the selection of different possible machine representations for each of the intrinsic data types (`INTEGER`, `REAL`, `COMPLEX`, `LOGICAL` and `CHARACTER`)

• Main usage: Provide a mechanism for making the selection of numeric precision and range portable.
The intrinsic inquiry function **KIND** will return the *kind value* of a given variable. The return value is a scalar.

Although it is common for the return value to be the same as the *number of bytes* stored in a variable of that kind, it is **NOT REQUIRED** by the Fortran standard.
KIND Values (2)

On a lot of systems:

REAL(KIND=4) :: xs    ! 4-byte IEEE float
REAL(KIND=8) :: xd    ! 8-byte IEEE float
REAL(KIND=16) :: xq    ! 16-byte IEEE float

But on some systems/compilers:

REAL(KIND=1) :: xs    ! 4-byte IEEE float
REAL(KIND=2) :: xd    ! 8-byte IEEE float
REAL(KIND=3) :: xq    ! 16-byte IEEE float

Quick sample program: mykinds.f90
SELECTED_REAL_KIND

You can request a minimum precision and range

SELECTED_REAL_KIND(P, R)

This gives at least P decimal places and range of $10^{-R}$ to $10^{R}$

e.g., SELECTED_REAL_KIND(12) will give at least 12 decimal places

Return codes:
-1 = does not support P value
-2 = does not support R value
-3 = neither is supported
Using KIND (1)

For large programs it is extremely handy to put this into a module:

```fortran
MODULE double
    INTEGER, PARAMETER :: DP = &
    SELECTED_REAL_KIND(12)
END MODULE double

Then, immediately after every procedure statement (i.e., PROGRAM, SUBROUTINE or FUNCTION):

USE double
IMPLICIT NONE
```
Declaring variables, etc. is easy

REAL (KIND=DP) :: a, b, c
REAL (KIND=DP), DIMENSION(10) :: x, y, z

Using constants is more tedious but easy

0.0_DP, 7.0_DP, 0.25_DP, 1.23E12_DP, 3.141592653589793_DP
Using KIND (3)

Note that the above makes it trivial to change all variables and constants in a large program. All you need to do is change the module

```plaintext
MODULE double
  INTEGER, PARAMETER :: DP = &
  SELECTED_REAL_KIND(15, 300)
END MODULE double

requires IEEE 754 double or better

Or even: SELECTED_REAL_KIND(25, 1000)
```
DOUBLE PRECISION

This was the second “kind” of real type in Fortran 77.

You can still use it just like REAL in declarations
Using KIND is more modern and compact

REAL (KIND=KIND(0.0D0) :: a, b, c
DOUBLE PRECISION, DIMENSION(10) :: x, y, z

Constants use D for the exponent

0.0D0, 7.0D0, 0.25D0, 1.23D12,
3.141592653589793D0

Quick sample program: mykinds1.f90
Type Conversion (1)

This is the main “gotcha” - you should use:

```plaintext
REAL (KIND=DP) :: x
x = REAL(<integer expression>, KIND=DP)
```

Omitting the KIND=DP may lose precision with no warning from the compiler

**Automatic** conversion is actually safer!

```plaintext
x = <integer expression>
x = SQRT(<integer expression>+0.0_DP)
```
There is a legacy intrinsic function
If you are using explicit DOUBLE PRECISION

\[ x = \text{DBLE}(<\text{integer expression}>) \]

All other “gotchas” are for COMPLEX
You can choose different sizes of integer

```fortran
INTEGER, PARAMETER :: big = &
   SELECTED_INT_KIND(12)
INTEGER (KIND=big) :: bignum
```

bignum can hold values up to $10^{12}$
Few users will need this - mainly for OpenMP

Some compilers may allocate smaller integers
e.g., by using `SELECTED_INT_KIND(4)`
CHARACTER KIND

It can be used to select the encoding.
It is mainly a Fortran 2003 feature.

Can select default, ASCII, or ISO 10646.
ISO 10646 is effectively Unicode.

Not covered in this course.
Notes

• The Fortran standard requires that each compiler support at least two real kinds which must have different precisions. The default real kind is the lower precision of these.

• There are two ways to specify a double precision real:

  1. With a REAL specifier using the KIND parameter corresponding to double precision (portable)

  2. Using a DOUBLE PRECISION specifier (not portable)

• Cool program: kindfinder.f90
Related Inquiry Functions

KIND(x) returns the kind value of x
PRECISION(x) returns the decimal precision of x
RANGE(x) returns the decimal exponent range of x
TINY(x) returns the smallest non-zero number of x
HUGE(x) returns the largest non-infinite number of x
DIGITS(x) returns the number of significant digits in the internal model representation of x
RADIX(x) returns the base of the model representing x
MINEXPONENT(x) returns the minimum exponent of the model representing x
MAXEXPONENT(x) returns the maximum exponent of the model representing x
Modules, Make and Interfaces
Module Summary

• Similar to same term used in other languages. As usual, modules fulfill multiple purposes
• For shared declarations (i.e., “headers”)
• Defining global data (old COMMON)
• Defining procedure interfaces
• Semantic extension (described later)

And more...
Use of Modules

- Think of a module as a high-level interface. It collects <whatevers> into a coherent unit.

- Design your modules carefully. As the ultimate top-level program structure. Perhaps only a few, perhaps dozens.

- Good place for high-level comments. Please document purpose and interfaces.
Module Structure

**MODULE** module-name

Static data definitions (often exported)

**CONTAINS**

Procedure definitions and interfaces

**END MODULE** module-name

Files may contain several **modules**

Modules may be split across several **files**

For simplest use, keep them 1 to 1
IMPLICIT NONE

Modules should also use this important specification

MODULE double
    IMPLICIT NONE
    INTEGER, PARAMETER :: DP = KIND(0.0D0)
END MODULE double

MODULE parameters
    USE double
    IMPLICIT NONE
    REAL(KIND=DP), PARAMETER :: one = 1.0_DP
END MODULE parameters
Module Interactions

Modules can **USE** other modules

Dependency graph shows visibility/usage

Modules may not depend on themselves
i.e., the standard does not permit the recursive or
circular use of modules

```plaintext
MODULE A
  USE B
END MODULE A

MODULE B
  USE A
END MODULE B
```
MODULE double
   INTEGER, PARAMETER :: DP = KIND(0.0D0)
END MODULE double

MODULE parameters
   USE double
   REAL(KIND=DP), PARAMETER :: one = 1.0_DP
   INTEGER, PARAMETER :: nx = 10, ny = 25
END MODULE parameters

MODULE workspace
   USE double
   USE parameters
   REAL(KIND=DP), DIMENSION(nx,ny) :: now, then
END MODULE workspace
The main program might look like this

```fortran
PROGRAM main
    USE double
    USE parameters
    USE workspace
    ...
END PROGRAM main
```

Could omit the `USE double` and `USE parameters` as they would be inherited through `USE workspace`
Module Dependencies

- double
- parameters
- workspace
- main program
Shared Constants

We have already seen and used this:

```fortran
MODULE double
  INTEGER, PARAMETER :: DP = KIND(0.0D0)
END MODULE double
```

You can do a great deal of this sort of thing

Greatly improves **clarity** and **maintainability**

The larger the program, the more it helps

Example from the CAM: `shr_const_mod.F90`
Derived Type Definitions

We shall cover these later:

MODULE Bicycle
    REAL, PARAMETER :: pi = 3.141592
    TYPE Wheel
        INTEGER :: spokes
        REAL :: diameter, width
        CHARACTER(LEN=15) :: material
    END TYPE Wheel
END MODULE Bicycle

USE Bicycle
TYPE(Wheel) :: w1
Global Data

Variables in modules define global data. These can be fixed-size or allocatable arrays.

- You need to specify the SAVE attribute. Set automatically for initialized variables. But it is good practice to do it explicitly.

A simple SAVE statement saves everything.
- This isn’t always the best thing to do.
MODULE state_variables
    INTEGER, PARAMETER :: nx=100, ny=100
    REAL, DIMENSION(NX,NY), SAVE :: &
        current, increment, values
    REAL, SAVE :: time = 0.0
END MODULE state_variables

USE state_variables
IMPLICIT NONE
DO
    current = current + increment
    CALL next_step(current, values)
END DO
Example (2)

This is equivalent to the previous example:

```Fortran
MODULE state_variables
  IMPLICIT NONE
  SAVE
  INTEGER, PARAMETER :: nx=100, ny=100
  REAL, DIMENSION(NX,NY) :: &
    current, increment, values
  REAL :: time = 0.0
END MODULE state_variables
```
Example (3)

The arrays sizes do not have to be fixed:

```
MODULE state_variables
    REAL, DIMENSION(:,,:), ALLOCATABLE, SAVE :: &
    current, increment, values
END MODULE state_variables

USE state_variables
IMPLICIT NONE
INTEGER :: NX, NY
READ *, NX, NY
ALLOCATE(current(NX,NY), increment(NX,NY), &
        values(NX,NY))
```
Explicit Interfaces

Procedures now need explicit interfaces
e.g., for assumed shape arrays, keywords

• Modules are the primary way of doing this
We will come to the secondary way later

Simplest to include the procedures in modules
The procedure code goes after CONTAINS
This is what we discussed earlier
MODULE mymod
CONTAINS
  FUNCTION Variance (Array)
    REAL :: Variance, X
    REAL, INTENT(IN), DIMENSION(:) :: Array
    X = SUM(Array)/SIZE(Array)
    Variance = SUM((Array-X)**2)/SIZE(Array)
  END FUNCTION Variance
END MODULE mymod

PROGRAM main
  USE mymod
  PRINT *, 'Variance = ', Variance(array)