Kind and Precision
(a.k.a. 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 90/95/2003/2008 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.

KIND Values (1)

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
```

Sample program: mykinds.f90

SELECTED_REAL_KIND (1)

You can request a minimum precision and range as well as a specific radix (*radix is new in Fortran2008)

SELECTED_REAL_KIND(Prec, Range, Radix)

This gives at least Prec decimal places and range of 10^{-Range} to 10^{Range}

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

SELECTED_REAL_KIND (2)

Return codes:

- -1 = does not support P value, but r and radix okay
- -2 = does not support R value, but p and radix okay
- -3 = if radix but not P and R reqs are fulfillable
- -4 = if radix and either P and R reqs are fulfillable
- -5 = if there is no real time with the given radix

Using KIND (1)

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

```
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
```

Using KIND (2)

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
```

Sample module: shr_kind_mod.F90

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

```
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,
3.1415926535893D0
```

Sample program: setkinds.f90

Intrinsic Procedures

- Almost all intrinsics "just work" (i.e., are generic)

  REAL, INT, NINT, MAX, MIN, ABS etc.

- Avoid specific (old) names for intrinsics

  AMAX0, DMIN1, DSQRT, FLOAT, IFIX, etc.

- Don't use the INTRINSIC statement
- Don't pass intrinsic functions as arguments

Type Conversion (1)

This is the main "gotcha" - you should use:

```
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!

```
x = <integer expression>
x = SQRT(<integer expression>+0.0_DP)
```

Type Conversion (2)

There is a legacy intrinsic function
If you are using explicit DOUBLE PRECISION

x = DBLE(<integer expression>)

All other "gotchas" are for COMPLEX

Warning

You will often see code like:

```
REAL*8 X, Y, Z
INTEGER*8 M,N
```

A Fortran IV feature, not a standard one

‘8’ is NOT always the size in bytes

I strongly recommend converting to KIND

INTEGER KIND

You can choose different sizes of integer

```
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)

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

Derived Types

What Are Derived Types?

As we discussed back in "Data Types and Basic Calculation", there are five intrinsic data types available in Fortran. A derived type is a special form of data type that can encapsulate other built-in types as well as other derived types.

C++, Python, etc. are very similar (structures)

Simple Derived Types

```
TYPE  Wheel
    INTEGER :: spokes
    REAL :: diameter, width
    CHARACTER(LEN=15) :: material
END TYPE  Wheel
```

That defines a derived type Wheel
Using derived types needs a special syntax

```
TYPE(Wheel) :: w1
print *, w1%spokes
```

Usage

1. Declare the type

   TYPE <derived type name>
   declarations
   END TYPE <derived type name>

2. Create an instance of the type

   TYPE(<derived type name>) :: <varname>

More Complicated Ones

You can include almost anything in there

```
TYPE Bicycle
    CHARACTER(LEN=80) :: description(100)
    TYPE(Wheel) :: front, back
    REAL, ALLOCATABLE, DIMENSION(:) :: times
    INTEGER, DIMENSION(100) :: codes
END TYPE Bicycle
```

And so on...

Sample program: bike.f90

Fortran 90/95 Restriction

Fortran 90/95 was much more restrictive
You couldn't have ALLOCATABLE arrays
Had to use POINTER instead

Fortran 2003 removed that restriction
Most compilers already include this feature

Be sure to check your own compiler

Component Selection

The selector "%" is used for this
Followed by a component of the derived type

It delivers whatever type that field is
You can then subscript or select it

```
TYPE(Bicycle) :: mine

mine%times(52:53) = (/ 123.4, 98.7 /)
PRINT *, mine%front%spokes
```

Selecting from Arrays

You can select from arrays and array sections
It produces an array of that component alone

```
TYPE Rabbit
    CHARACTER(LEN=16) :: variety
    REAL :: weight, length
    INTEGER :: age
END TYPE Rabbit

TYPE(Rabbit), DIMENSION(100) :: exhibits
REAL, DIMENSION(50) :: fattest

fattest = exhibits(51:)%weight
```

Assignment (1)

You can assign complete derived types
That copies the values element-by-element

```
TYPE(Bicycle) :: mine, yours

yours = mine
mine%front = yours%back
```

Assignment is the only intrinsic operation

You can redefine that or define other operations
But they are some of the topics that I am omitting

Assignment (2)

Each derived type is unique

You cannot assign between different ones

```
TYPE :: Fred
    REAL :: x
END TYPE Fred
TYPE :: Joe
    REAL :: x
END TYPE Joe
TYPE(Fred) :: a
TYPE(Joe) :: b
a = b    ! This is erroneous
```

Constructors

A constructor creates a derived type value

```
TYPE Circle
    REAL :: X, Y, radius
    LOGICAL :: filled
END TYPE Circle

TYPE(Circle) :: a
a = Circle(1.23, 4.56, 2.0, .False.)
```

Fortran 2003 allows keywords for components

```
a = Circle(X=1.23, Y=4.56, radius=2.0, filled=.False.)
```

Default Initialization

You can specify default initial values

```
TYPE Circle
    REAL :: X = 0.0, Y = 0.0, radius = 1.0
    LOGICAL :: filled = .False.
END TYPE Circle

TYPE(Circle) :: a, b, c
a = Circle(1.23, 4.56, 2.0, .True.)
```

This becomes much more useful in with keywords

```
a = Circle(X=1.23, Y=4.56)
```

I/O on Derived Types

Can do normal I/O with the ultimate components
A derived type is flattened much like an array
(recursively if it includes embedded derived types)

```
TYPE(Circle) :: a, b, c
a = Circle(1.23, 4.56, 2.0, .True.)
PRINT *, a ;  PRINT *, b ;  PRINT *, c
```

```
 1.230000  4.5599999  2.0000000  T
0.0000000E+00  0.0000000E+00  1.0000000  F
0.0000000E+00  0.0000000E+00  1.0000000  F
```

Private Derived Types

When you define them in modules

A derived type can be wholly private
  i.e., accessible only to module procedures

Or its components can be hidden
  i.e., it's visible as an opaque type

Wholly Private Types

```
MODULE Marsupial
   TYPE, PRIVATE :: Wombat
      REAL :: width, length
   END TYPE Wombat
   REAL, PRIVATE :: koala
   CONTAINS
    . . .
END MODULE Marsupial
```

Wombat is not exported from Marsupial
No more than the variable Koala is

Hidden Components (1)

Hidden components allow opaque types
The module procedures use them normally

- Users of the module can't look inside them

They can assign them like variables
They can pass them as arguments
Or call the module procedures to work on them

An important software engineering technique
Usually called data encapsulation

Hidden Components (2)

```
MODULE Marsupial
   TYPE ::Wombat
      PRIVATE
      REAL :: width, length
   END TYPE Wombat
   CONTAINS
    . . .
END MODULE Marsupial
```

Wombat IS exported from Marsupial
But its components (width, length) are not

Trees

Example: Type A contains an array of type B
Objects of type B contain arrays of type C

```
TYPE Leaf
    CHARACTER(LEN=20) :: name
    REAL(KIND=dp), DIMENSION(3) :: data
END TYPE Leaf
TYPE Branch
    TYPE(Leaf), ALLOCATABLE :: leaves(:)
END TYPE Branch
TYPE Trunk
    TYPE(Branch), ALLOCATABLE :: branches(:)
END TYPE Trunk
```

Going Beyond the Basics

Fortran 2003 greatly extended/expanded derived types

- full object orientation
- type bound procedures
- polymorphism (abstract types)
- and LOTS more

It's enough for a separate course
Beyond what this audience really needs

Extending a Derived Type

Inheritance: allowing "child" types derive from extensible parent types

TYPE, EXTENDS(parent) :: child

Here the child inherits all the members and functionality from the parent type.

Example: test_employee.f90

Recursive Types

Pointers allow that to be done a little more flexibly
You don't need a separate type for each level

People often use more complicated structures
You build those using derived types
  e.g., linked lists (also called chains)

Both very commonly used for sparse matrices
And algorithms like Dirichlet tesselation

We shall return to this when we cover pointers