Accessibility (1)

Can separate exported from hidden definitions

Fairly easy to use in simple cases
- Worth considering when designing modules

PRIVATE names are accessible only within the module (i.e., in module procedures after CONTAINS)

PUBLIC names are accessible by USE
This is commonly called exporting them
Accessibility (2)

They are just another attribute of declarations

MODULE fred
   REAL, PRIVATE :: array(100)
   REAL, PUBLIC :: total
   INTEGER, PRIVATE :: error_count
   CHARACTER(LEN=50), PUBLIC :: excuse
CONTAINS
   ...
END MODULE fred
ACCESSIBILITY (3)

PUBLIC/Private statement sets the default. The default default is PUBLIC.

Module fred
PRIVATE
REAL :: array(100)
REAL, PUBLIC :: total
CONTAINS
...
END Module fred

Only TOTAL is accessible by a USE statement.
You can specify names in the statement
Especially useful for included names

MODULE workspace
  USE double
  PRIVATE :: dp
  REAL(KIND=dp), DIMENSION(1000) :: scratch
END MODULE workspace

DP is no longer exported via workspace
Partial Inclusion (1)

You can include only some names in USE

USE bigmodule, ONLY : errors, invert

Makes only errors and invert visible regardless of how many names bigmodule exports

Using ONLY is good practice
Makes it easier to keep track of uses

Can find out what is used where with grep
Partial Inclusion (2)

• One case when ONLY is strongly recommended:
  When using USE within modules

• All included names are exported
  Unless you explicitly mark them PRIVATE
  Perhaps only a few, perhaps dozens

• Ideally, use both ONLY and PRIVATE
  Almost always use at least one of them

• Another case when it it almost essential:
  If you don’t use IMPLICIT NONE liberally!
Partial Inclusion (3)

If you don’t restrict exporting and importing then a typing error could trash a module variable.

Or forget that you had already used the name in another file far, far away...

• The resulting chaos is almost unfindable

From bitter experience in many years of Fortran!
MODULE settings
   INTEGER, PARAMETER :: DP = KIND(0.0D0)
   REAL(KIND=DP) :: Z = 1.0_DP
END MODULE settings

MODULE workspace
   USE settings
   REAL(KIND=DP), DIMENSION(1000) :: scratch
END MODULE workspace
Example (2)

PROGRAM main
  USE workspace
  Z = 123
  ...
END PROGRAM main

- DP is inherited, which is okay
- Did you mean to update Z in settings?
- No problem if workspace had used ONLY : DP
Example (3)

The following are **better** and **best**

```fortran
MODULE workspace
  USE settings, ONLY : DP
  PRIVATE :: DP
  REAL(KIND=DP), DIMENSION(1000) :: scratch
END MODULE workspace
```

```fortran
MODULE workspace
  USE settings, ONLY : DP
  REAL(KIND=DP), DIMENSION(1000) :: scratch
END MODULE workspace
```
You can rename a name when you include it

**WARNING**: this is footgun territory
i.e., point gun at foot, pull trigger

This technique is sometimes incredibly useful
• But it is also incredibly dangerous

Use it only when you really need to
And even then as little as possible
Renaming Inclusion (2)

MODULE corner
   REAL, DIMENSION(100) :: temp
END MODULE corner

PROGRAM house
   USE corner, dum_array => temp
   INTEGER, DIMENSION(20) :: temp
   ...
END PROGRAM house

temp is accessible under the name dum_array
The name temp is the local array
Why Is This Lethal?

MODULE one
   REAL :: X
END MODULE one

MODULE two
   USE one, Y => X
   REAL :: Z
END MODULE two

PROGRAM three
   USE one
   USE two
   !-- Both X and Y refer to the same variable!
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.
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

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:

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

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

```plaintext
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
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

```fortran
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

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

Constants use D for the exponent

```plaintext
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:

\[
\begin{align*}
\text{REAL (KIND=DP) :: } x \\
 x &= \text{REAL(<integer expression>, KIND=DP)}
\end{align*}
\]

Omitting the \texttt{KIND=DP} may lose precision with \texttt{no warning} from the compiler

\textbf{Automatic} conversion is actually safer!

\[
\begin{align*}
x &= <\text{integer expression}> \\
x &= \text{SQRT(<integer expression>+0.0_DP)}
\end{align*}
\]
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