Modules and Interfaces

Motivation

Passing arguments may not be the most efficient way to share a large number of things between a large number of procedures

- Just writing all of the argument lists and getting them in the proper order may be a significant chore (and may reduce efficiency)

Modules provide a way of sharing procedures as well as data

- Especially useful when building a package or library that may be accessible to many different programs

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
  Very helpful to 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

  For simplest use, keep them one-to-one

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

```
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
```

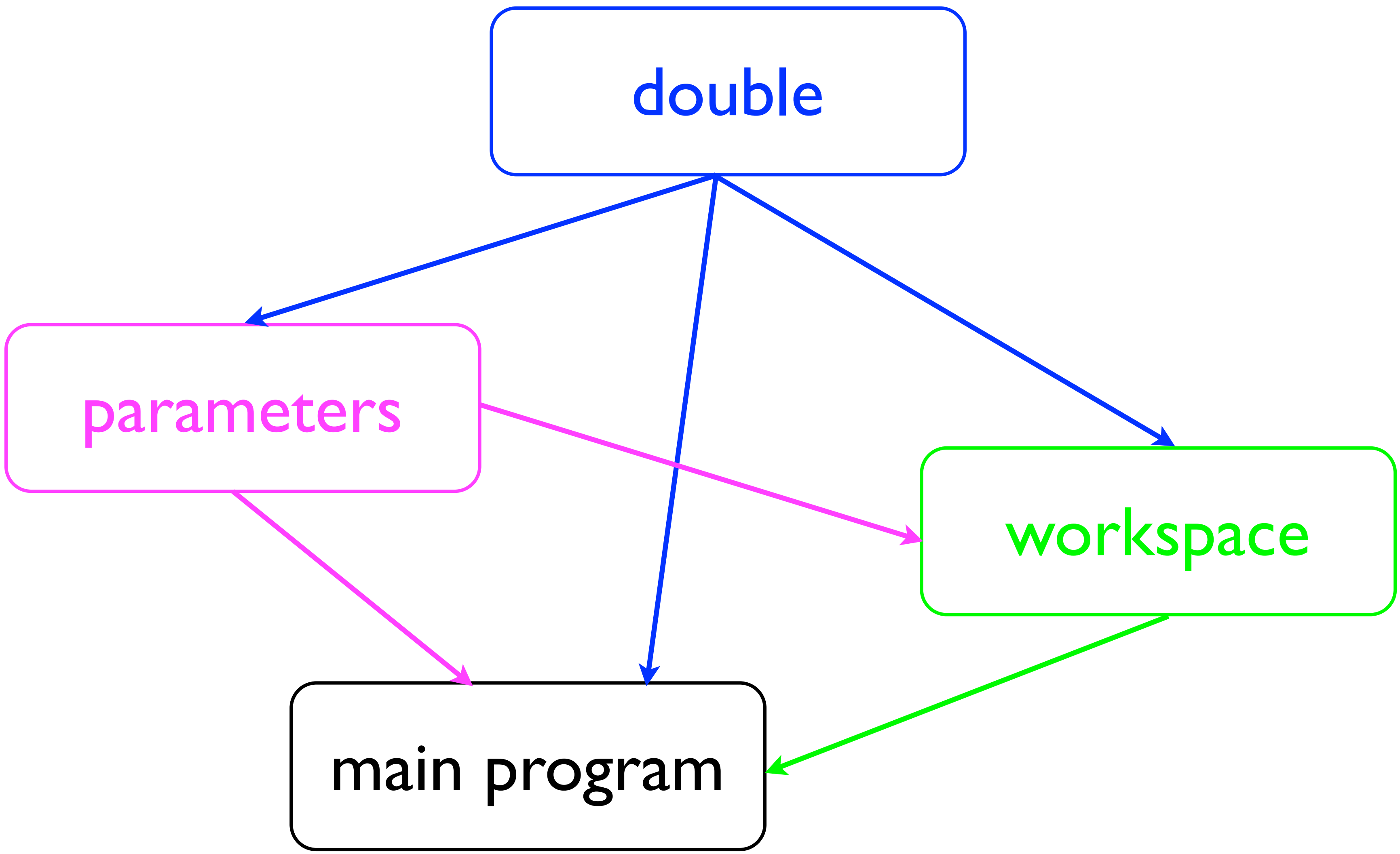
Example (cont.)

The main program might look like this

```
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



Shared Constants

We have already seen and used this:

```
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

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

Example (1)

```
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:

```
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

Example: goodpass2.f90

Example

```
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)
```

Procedures in Modules (1)

Including all procedures within modules works very well in almost all programs

These are very much like internal procedures

Everything accessible in the module can also be used in the procedure

Again, a local name takes precedence
But reusing the same name is very confusing

Procedures in Modules (2)

```
MODULE thing
    INTEGER, PARAMETER :: temp = 123
    CONTAINS
    SUBROUTINE pete ()
        INTEGER, PARAMETER :: temp = 456
        PRINT *, temp
    END SUBROUTINE pete
END MODULE thing
```

This will print 456, not 123

Avoid doing this as it's very confusing

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
```

Compiling Modules

Just as with external subroutines, you'll want to compile modules with the -c compiler switch

```
gfortran -c mymod.f90
```

This will create files mymod.mod and mymod.o
They contain the interface and the code

Using Compiled Modules

The program just needs the USE statement

Compile all of the modules in a dependency order
If A contains USE B, compile B first

Then add a *.o for every module when linking

```
gfortran -o main main.f90 mymod.o
gfortran -o main main.f90 mymod.o \
   mod_a.o mod_b.o mod_c.o
```

Interfaces in Modules

The module can define just the interface
The procedure code is supplied elsewhere
The interface block comes before CONTAINS

- Be absolutely sure they are consistent!

The interface and code are not checked

Examples: **goodpass3.f90, goodpass4.f90**

What Are Interfaces?

The FUNCTION or SUBROUTINE statement
And everything directly connected to that

Strictly, the argument names are not part of it
You are strongly advised to keep them the same

Local variables can be left out

Interface Blocks

These start with an INTERFACE statement
Include any number of procedure interfaces
End with an END INTERFACE statement

```
INTERFACE
    SUBROUTINE Fred (arg)
        REAL :: arg
    END SUBROUTINE FRED
    FUNCTION Joe ()
        LOGICAL :: Joe
   END FUNCTION Joe
END INTERFACE
```

Example

SUBROUTINE does_something(A)	YES
USE DOUBLE	YES
INTEGER :: j, n	NO
REAL(KIND=dp) :: A(:,:), X	YES for A
	NO for X
. . .	
END SUBROUTINE does_something	YES

Procedures as Arguments

With Fortran 90/95 it was essential to use an interface block for using procedure arguments

Fortran 2003/2008: not true anymore

Example: proc_as_arg

* I tried using an intrinsic function as an argument and it failed, but some compilers may support this

Another Interface Format

Enables the use of generic procedures

```
INTERFACE
   MODULE PROCEDURE proc_a, proc_b, …
END INTERFACE
```

Example: genericswap.f90

Interface Bodies and Names (1)

An interface body does NOT import names
The reason is that you can't undeclare names

For example, this does not work as expected:

```
USE double   ! This does not allow usage of dp
INTERFACE
    FUNCTION square (arg)
        REAL(KIND=dp) :: square, arg
    END FUNCTION square
END INTERFACE
```

Interface Bodies and Names (2)

So there is another statement to import names

```
USE double
INTERFACE
    FUNCTION square (arg)
        IMPORT :: dp                    ! This solves it
        REAL(KIND=dp) :: square, arg
    END FUNCTION square
END INTERFACE
```

It is available ONLY in interface bodies

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

Accessibility (4)

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
- Ideally, use both ONLY and PRIVATE
  Almost always use at least one of them
- Another case when it is almost essential:
  If you don't use IMPLICIT NONE religiously!

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!

Example (1)

```
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
  IMPLICIT NONE
  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

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

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

Renaming Inclusion (1)

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) :: pooh
END MODULE corner

PROGRAM house
    USE corner, sanders => pooh
    INTEGER, DIMENSION(20) :: pooh
    . . .
END PROGRAM house
```

pooh is accessible under the name sanders

The name pooh 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!
```

Protected Status (1)

NEW in Fortran 2003: PROTECTED attribute and statement

A module procedure can only modify a protected module entity (or its subobjects) if the same module defines both the procedure and the entity

Protected Status (2)

There are three possible access properties:

- public : outside code has read and write access
- private : outside code has NO access
- public, protected : outside code has read access

Example: **protected.f90**