Character Arguments

Few scientists do anything fancy with these

People often use a constant length
You can specify this as a digit string

Or define it using PARAMETER
That is best done in a module

Or define it as an assumed length argument

Explicit Length Character (1)

The dummy should match the actual argument

You are likely to get confused if it doesn't

```
SUBROUTINE sorter (list)
  CHARACTER(LEN=8), DIMENSION(:) :: list
      ...
END SUBROUTINE sorter

CHARACTER(LEN=8) :: data(1000)
      ...
CALL sorter(data)
```

Explicit Length Character (2)

```
MODULE Constants
  INTEGER, PARAMETER :: charlen=8
END MODULE Constants

SUBROUTINE sorter (list)
  USE Constants
  CHARACTER(LEN=charlen), DIMENSION(:) :: list
==========================================
USE Constants
CHARACTER(LEN=charlen) :: data(1000)
CALL sorter(data)
```

Assumed Length Character

A CHARACTER length can be assumed
The length is taken from the actual argument

You use an asterisk (*) for the length
It acts very like an assumed shape array

Note that it is a property of the type
It is independent of any array dimensions

Example (1)

```
FUNCTION is_palindrome(word)
   LOGICAL :: is_palindrome
   CHARACTER(LEN=*), INTENT(IN) :: word
    INTEGER :: n,i
    is_palindrome = .false.
    n = len(word)
    do i = 1,(n-1)/2
      if (word(i:i) /= word(n+1-i:n+1-i)) then
        RETURN
      endif
     enddo
     is_palindrome = .true.
 END FUNCTION is_palindrome
```

Example (2)

Such arguments do not have to be read-only

```
SUBROUTINE reverse_word(word)
   CHARACTER(LEN=*), INTENT(INOUT) :: word
   CHARACTER(LEN=1) :: c
   N = LEN(word)
   DO i = 1,(n-1)/2
      c = word(i:i)
      word(i:i) = word(n+1-i:n+1-i)
      word(n+1-i:n+1-i) = c
   ENDDO
END SUBROUTINE reverse_word
```

Static Data

Sometimes you need to store values locally
Use a value in the next call of the procedure

You can do this with the SAVE attribute
Initialized variables get this automatically

The best style avoids this use.

Warning for C/C++ Users

Initialization in a declaration without SAVE initializes **only once!**

It does **NOT** reinitialize each time it is called

Do it with an explicit assignment statement

Example: localsave.f90, test_saves.f90

Modules 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

```
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
```

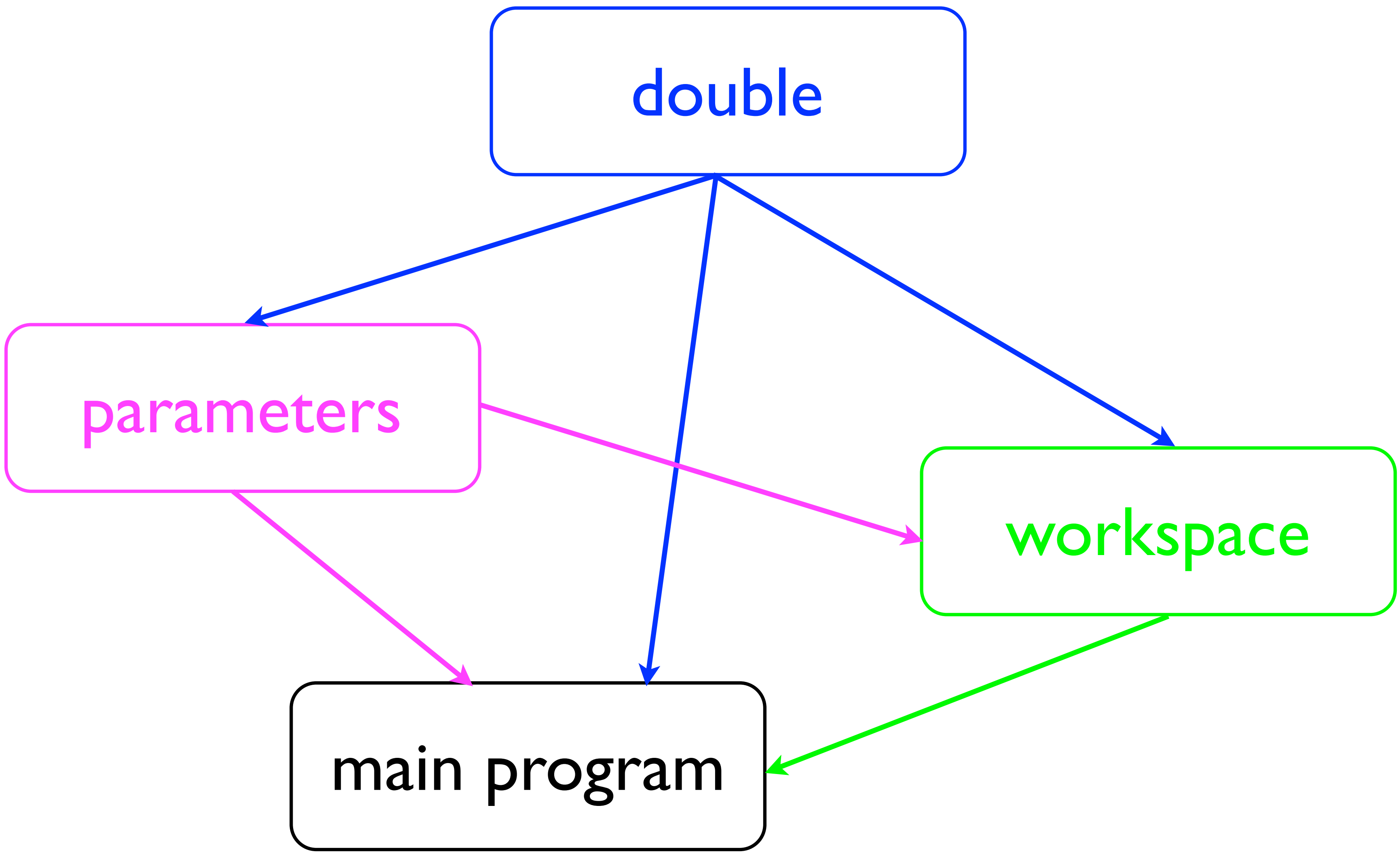
Example (cont.)

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)

That is including all procedures within modules
Works very well in almost all programs

- There really isn't much more to it

It doesn't handle very large modules well
Try to avoid designing these if possible

Procedures in Modules (2)

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 (3)

```
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

Example 1: goodpass3.F90

Example 2: Cholesky decomposition

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