Introduction to Modern Fortran

Modules, Make and Interfaces

Nick Maclaren

Computing Service

nmm1@cam.ac.uk, ext. 34761

September 2011

Module Summary

- Similar to same term in other languages

As usual, modules fulfil 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
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 <name>
    Static (often exported) data definitions
CONTAINS
    Procedure definitions and interfaces
END MODULE <name>

Files may contain several modules
Modules may be split across many files

- For simplest use, keep them $1 \equiv 1$

IMPLICIT NONE

Add MODULE to the places where you use this

```
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
```

Reminder

I do not always do it, because of space

Module Interactions

Modules can USE other modules
Dependency graph shows visibility/usage

- Modules may not depend on themselves

Languages that allow that are very confusing

Can do anything you are likely to get to work

- If you need to do more, ask for advice

Example (1)

```
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 = 20
END MODULE parameters

MODULE workspace
      USE double ;    USE parameters
   REAL(KIND=DP), DIMENSION(NX, NY) :: now, then
END MODULE workspace
```

Example (2)

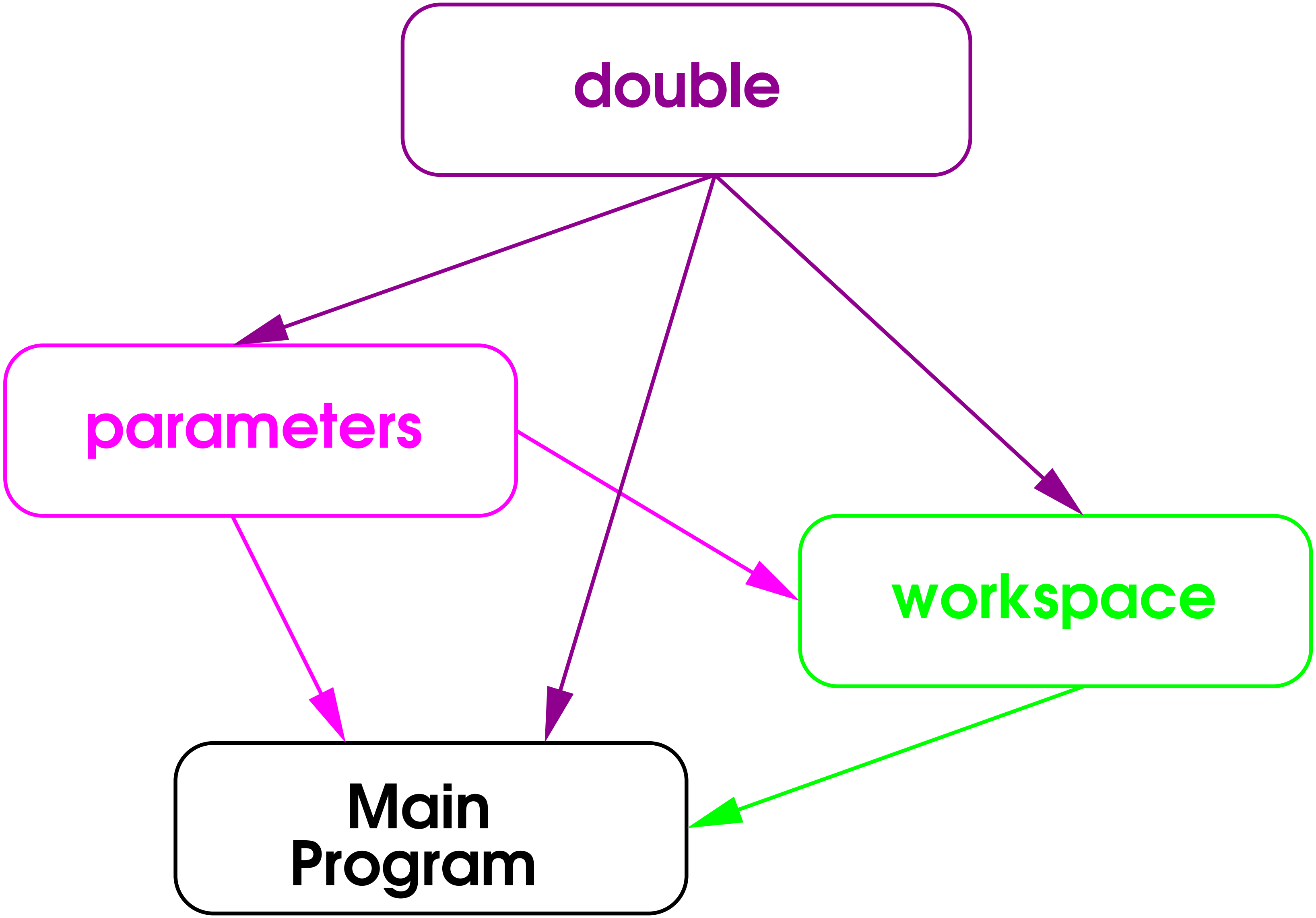
The main program might use them like this

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

- Could omit the USE double and USE parameters

They would be inherited through USE workspace

Module Dependencies



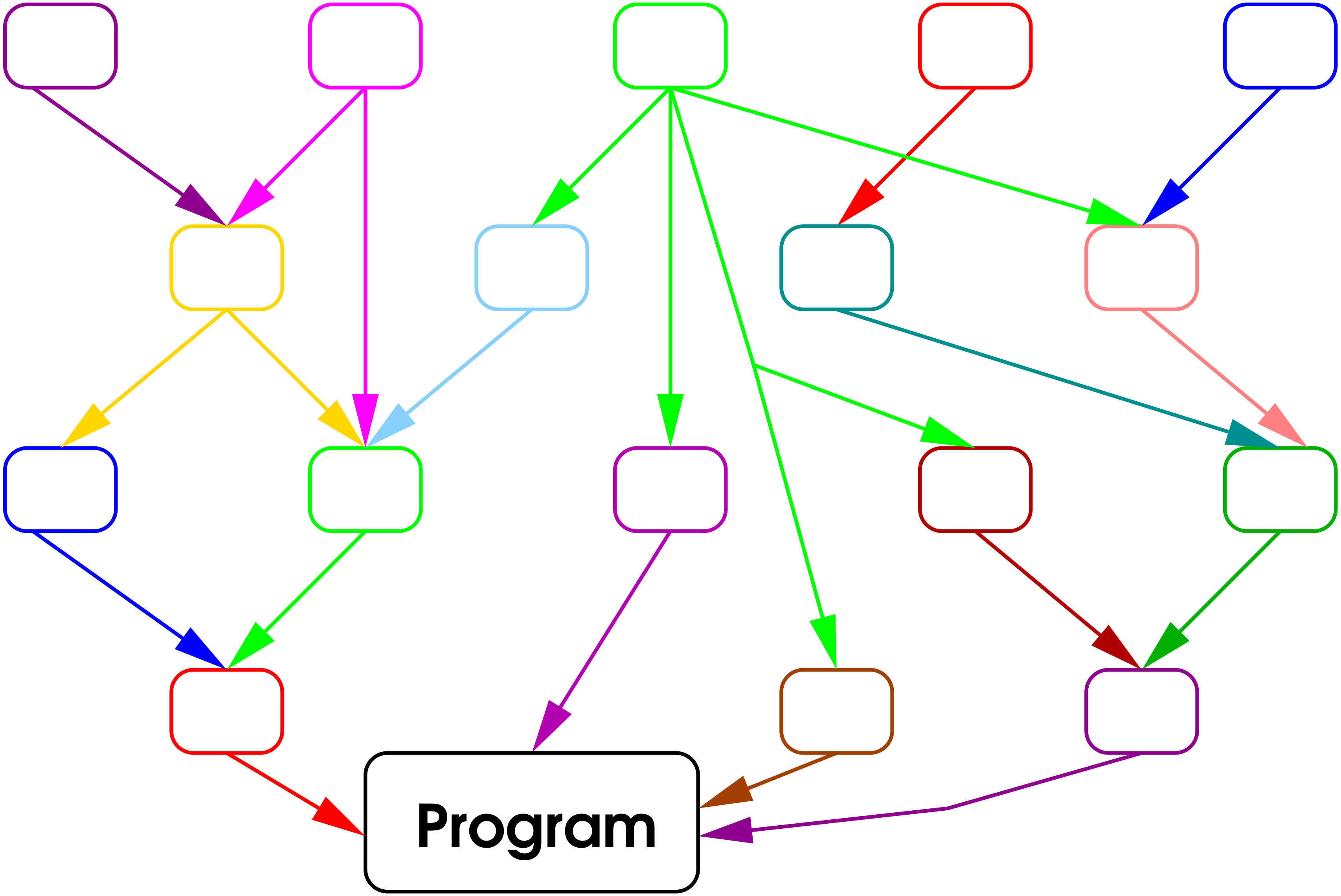
double

parameters

workspace

Main
Program

Module Dependencies



Makefile Warnings

This does NOT teach how to use make
It teaches just the Fortran–specific aspects

See Building, installing and running software
If you haven't been to it, DO SO before starting!

The defaults for $(FC) and $(FFLAGS) are broken
Hopelessly outdated, and no longer work

That applies to both POSIX and GNU make!

- You must set them yourself

Or you can use other names, if you prefer

Makefile Basics (1)

Use make in exactly the same way as for C

- Must set $(FC) and $(FFLAGS) or whatever
- Modules create both .mod and .o files
- Do not need to set LDFLAGS = –lm

Will give a very simple example:
The module file utils.f90 creates a module UTILS
And that is used by a program file trivial.f90

Makefile Basics (2)

```
    FC = nagfor
    FFLAGS = –C=all

    LDFLAGS =

    all: trivial

    utils.mod utils.o: utils.f90
<tab> $(FC) $(FFLAGS) –c –o utils.o utils.f90

    trivial: utils.mod utils.o trivial.f90
<tab> $(FC) $(FFLAGS) $(LDFLAGS) –o trivial trivial.f90 utils.o
```

Compiling Modules (1)

This is a FAQ – Frequently Asked Question
The problem is the answer isn’t simple

- That is why I give some of the advice that I do

The following advice will not always work
OK for most compilers, but not necessarily all

- This is only the Fortran module information

Compiling Modules (2)

The module name need not be the file name
Doing that is strongly recommended, though

- You can include any number of whatevers

You now compile it, but don’t link it

```
nagfor –C=all –c mymod.f90
```

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

We now need to describe the process in more detail

What Compilers Do (1)

A file frederick.f90 contains modules fred and alf
You compile this with:

```
nagfor –C=all –c frederick.f90
```

It will create files frederick.o, fred.mod and alf.mod

- frederick.o contains the compiled code

Link this into into the executable, in the usual way:

```
nagfor –C=all program program.f90 frederick.o
```

What Compilers Do (2)

- fred.mod and alf.mod contain the interfaces

Think of them as being a sort of compiled header

- You don’t do anything with these, explicitly

The compiler will do find them and use them

A file program.f90 contains USE fred and USE alf

- The compiler will search for fred.mod and alf.mod

Searched for using the same paths as headers
To add another search path, use –I<directory>

- Be warned – compilers vary – see their docs

Makefile Rules (1)

You need to set up rules to compile the modules
And to add dependencies to ensure they are rebuilt

- Dependencies are exactly like headers

The object file has a dependency on the module

A lot of people forget about headers in makefiles

- Doing that with modules is disastrous

Gets the compiled code out of step with the interface
E.g. gets the new fred.o and the old fred.mod

Makefile Rules (2)

A file program.f90 contains USE fred and USE alf
Modules fred and alf are in files fred.f90 and alf.f90
This is how you set up the dependency and rules:

```
program:  program.o  fred.o  alf.o
<tab>  $(FC)  $(FFLAGS)  $(LDFLAGS)  –o  program

program.o:  program.f90  fred.mod  alf.mod
<tab>  $(FC)  $(FFLAGS)  –c  program.f90

fred.mod  fred.o:  fred.f90
<tab>  $(FC)  $(FFLAGS)  –c  fred.f90

alf.mod  alf.o:  alf.f90
<tab>  $(FC)  $(FFLAGS)  –c  fred.f90
```

Makefile Rules (3)

Say frederick.f90 contains modules fred and alf
and includes the statement USE double
A file program.f90 contains USE frederick

```
program: program.o frederick.o double.o
<tab> $(FC) $(FFLAGS) $(LDFLAGS) –o program

program.o: program.f90 fred.mod alf.mod
<tab> $(FC) $(FFLAGS) –c program.f90

double.mod double.o: double.f90
<tab> $(FC) $(FFLAGS) –c double.f90

fred.mod alf.mod frederick.o: frederick.f90 double.mod
<tab> $(FC) $(FFLAGS) –c double.f90
```

Doing Better (1)

Can clean up the Makefile somewhat, fairly easily

E.g. use the $@, $< and $* macros
But take care, as things are a little tricky

- Problem is one module file produces two results

And headers are not compiled, but modules are

It's still a bit tedious with a lot of modules

Doing Better (2)

You can do a good deal better, but it's advanced use
Beyond Building, installing and running software

Need either inference rules or pattern rules
Worse, POSIX and GNU are wildly different

It can be done, and it's not even very difficult

- But it is very system–dependent!

Build Warnings (1)

The following names are global identifiers

  All module names

  All external procedure names

    I.e. not in a module or internal

- They must all be distinct

And remember their case is not significant

- Avoid using any built–in procedure names

That works, but it is too easy to make errors

Build Warnings (2)

Avoid file names like fred.f90 AND
external names like FRED
Unless FRED is inside fred.f90

- It also helps a lot when hunting for FRED

This has nothing at all to do with Fortran
It is something that implementations get wrong
Especially the fancier sort of debuggers

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 that sort of thing

- Greatly improves clarity and maintainability

The larger the program, the more it helps

Example

```
MODULE hotchpotch
    INTEGER, PARAMETER :: DP = KIND(0.0D0)
    REAL(KIND=DP), PARAMETER :: &
        pi = 3.141592653589793_DP, &
        e = 2.718281828459045_DP
    CHARACTER(LEN=*), PARAMETER :: &
        messages(3) = &
            (\ "Hello", "Goodbye", "Oh, no!" \)
    INTEGER, PARAMETER :: stdin = 5, stdout = 6
    REAL(KIND=DP), PARAMETER, &
        DIMENSION(0:100, –1:25, 1:4) :: table = &
        RESHAPE( (/ . . . /), (/ 101, 27, 4 /) )
END MODULE hotchpotch
```

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 initialised variables
But it is good practice to do it explicitly

A simple SAVE statement saves everything

- That 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 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))
```

Use of SAVE

If a variable is set in one procedure
and then it is used in another

- You must specify the SAVE attribute

- If not, very strange things may happen

If will usually ‘‘work’’, under most compilers
A new version will appear, and then it won’t

- Applies if the association is via the module

Not when it is passed as an argument

Example (1)

```
MODULE status
    REAL, DIMENSION :: state
END MODULE status

SUBROUTINE joe
    USE status
    state = 0.0
END SUBROUTINE joe

SUBROUTINE alf (arg)
    REAL :: arg
    arg = 0.0
END SUBROUTINE alf
```

Example (2)

```
SUBROUTINE fred
    USE status

    CALL joe
    PRINT *, state    ! this is UNDEFINED

    CALL alf(state)
    PRINT *, state    ! this is defined to be 0.0

END SUBROUTINE fred
```

Shared Workspace

Shared scratch space can be useful for HPC
It can avoid excessive memory fragmentation

You can omit SAVE for simple scratch space
This can be significantly more efficient

- Design your data use carefully

Separate global scratch space from storage
And use them consistently and correctly

- This is good practice in any case

Explicit Interfaces

Procedures now need explicit interfaces
E.g. for assumed shape or keywords
Without them, must use Fortran 77 interfaces

- Modules are the primary way of doing this

We will come to the secondary one later

Simplest to include the procedures in modules
The procedure code goes after CONTAINS
This is what we described earlier

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 in 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 those, if possible

It also doesn’t handle procedure arguments

Procedures in Modules (2)

They are very 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
```

Will print 456, not 123

Avoid doing this – it’s very confusing

Interfaces in Modules

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

- You had better get them consistent!

The interface and code are not checked

- Extract interfaces from procedure code

NAGWare and f2f90 can do it automatically

Cholesky Decomposition

```
SUBROUTINE CHOLESKY(A)
    USE double    ! note that this has been added
    INTEGER :: J, N
    REAL(KIND=dp) :: A(:, :), X
    N = UBOUND(A, 1)
    DO J = 1, N
        X = SQRT(A(J, J) – &
        DOT_PRODUCT(A(J, :J–1), A(J, :J–1)))
        A(J,J) = X
        IF (J < N) &
            A(J+1:, J) = (A(J+1:, J) – &
            MATMUL(A(J+1:, :J–1), A(J, :J–1))) / X
    END DO
END SUBROUTINE CHOLESKY
```

The Interface Module

```
MODULE MYLAPACK
    INTERFACE
        SUBROUTINE CHOLESKY(A)
            USE double    ! part of the interface
            IMPLICIT NONE
            REAL(KIND=dp) :: A(:, :)
        END SUBROUTINE CHOLESKY
    END INTERFACE
! This is where CONTAINS would go if needed
END MODULE MYLAPACK
```

The Main Program

```
PROGRAM MAIN
    USE double
    USE MYLAPACK
    REAL(KIND=dp) :: A(5,5) = 0.0_dp, Z(5)
    DO N = 1,10
        CALL RANDOM_NUMBER(Z)
        DO I = 1,5 ;   A(:,I) = A(:,I)+Z*Z(I) ;   END DO
    END DO
    CALL CHOLESKY(A)
    DO I = 1,5 ;   A(:I–1,I) = 0.0 ;   END DO
    WRITE (*, ’(5(1X,5F10.6/))’) A
END PROGRAM MAIN
```

What Are Interfaces?

The FUNCTION or SUBROUTINE statement
And everything directly connected to that
USE double needed in argument declaration

Strictly, the argument names are not part of it
You are strongly advised to keep them the same
Which keywords if the interface and code differ?

Actually, it's the ones in the interface

Example

```
SUBROUTINE CHOLESKY(A)    ! this is part of it
    USE errors    ! this ISN'T part of it
    USE double    ! this is, because of A
    IMPLICIT NONE    ! this ISN'T part of it
    INTEGER :: J, N    ! this ISN'T part of it
    REAL(KIND=dp) :: A(:, :), X    ! A is but not X
    . . .
END SUBROUTINE CHOLESKY
```

Interfaces In Procedures

Can use an interface block as a declaration
Provides an explicit interface for a procedure

Can be used for ordinary procedure calls
But using modules is almost always better

- It is essential for procedure arguments

Can't put a dummy argument name in a module!

Example (1)

Assume this is in module application

```
FUNCTION apply (arr, func)
    REAL :: apply, arr(:)
    INTERFACE
        FUNCTION func (val)
            REAL :: func, val
        END FUNCTION
    END INTERFACE
    apply = 0.0
    DO I = 1,UBOUND(arr, 1)
        apply = apply + func(val = arr(i))
    END DO
END FUNCTION apply
```

Example (2)

And these are in module functions

```
FUNCTION square (arg)
    REAL :: square, arg
    square = arg**2
END FUNCTION square

FUNCTION cube (arg)
    REAL :: cube, arg
    cube = arg**3
END FUNCTION cube
```

Example (3)

```
PROGRAM main
    USE application
    USE functions
  REAL, DIMENSION(5) :: A = (/ 1.0, 2.0, 3.0, 4.0, 5.0 /)
    PRINT *, apply(A,square)
    PRINT *, apply(A,cube)
END PROGRAM main
```

Will produce something like:

```
  55.0000000
   2.2500000E+02
```

Accessibility (1)

Can separate exported from hidden definitions

Fairly easy to use in simple cases

- Worth considering when designing modules

PRIVATE names accessible only in 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 USE

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
However 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 it is strongly recommended

When using USE in 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

Is if you don’t use IMPLICIT NONE religiously

Partial Inclusion (3)

If you don't restrict exporting and importing:

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 Fortran and C!

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 OK
- 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 is always 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
END PROGRAM three
```