Introduction to Modern Fortran

See next foil for copyright information

Nick Maclaren

nmm1@cam.ac.uk

March 2014

Acknowledgement

Derived from the course written and owned by

Dr. Steve Morgan
Computing Services Department
The University of Liverpool

Important!

There is a lot of material in the course
And there is even more in extra slides ...

Some people will already know some Fortran
Some will be programmers in other languages
Some people will be complete newcomers

The course is intended for all of those people

- Please tell me if I am going too fast

Not afterwards, but as soon as you have trouble

Beyond the Course (1)

http://people.ds.cam.ac.uk/nmm1/...

.../Fortran/

.../OldFortran/

.../Arithmetic/ etc.

Programming in Fortran 90/95

by Steve Morgan and Lawrie Schonfelder

(Fortran Market, PDF, $15)

http://www.fortran.com/

Also Fortran 90 version of that

Beyond the Course (2)

Fortran 95/2003 Explained
by Michael Metcalf, John Reid and Malcolm Cohen

Also Fortran 90 version of that

Fortran 90 Programming
by Miles Ellis, Ivor Phillips and Thomas Lahey

Beyond the Course (3)

SC22WG5 (ISO Fortran standard committee)

    http://www.nag.co.uk/sc22wg5/

http://www.fortran.com/fortran/

    ⇒ 'Information', 'Standards Documents'

Miscellaneous information and useful guidance

    http://www.star.le.ac.uk/~cgp/fortran.html

Liverpool Course

    http://www.liv.ac.uk/HPC/...

        .../HTMLFrontPageF90.html

Beyond the Course (4)

A real, live (well coded) Fortran 95 application

http://www.wannier.org

Most of the others I have seen are not public

Please tell me of any you find that are

Important!

There is a lot of material in the course
And there is even more in extra slides ...

This has been stripped down to the bare minimum
Some loose ends will remain, unfortunately
You will need to skip a few of the practicals

- Please tell me if I am going too fast

Not afterwards, but as soon as you have trouble

Practicals

These will be delayed until after second lecture
Then there will be two practicals to do

One is using the compiler and diagnostics
Just to see what happens in various cases

The other is questions about the basic rules

Full instructions will be given then
Including your identifiers and passwords

History

FORmula TRANslation invented 1954–8
by John Backus and his team at IBM

FORTRAN 66 (ISO Standard 1972)
FORTRAN 77 (1980)
Fortran 90 (1991)
Fortran 95 (1996)
Fortran 2003 (2004)
Fortran 2008 (2011)

The "Old Fortran" slides have more detail

Hardware and Software

A system is built from hardware and software

The hardware is the physical medium, e.g.
- CPU, memory, keyboard, display
- disks, ethernet interfaces etc.

The software is a set of computer programs, e.g.
- operating system, compilers, editors
- Fortran 90 programs

Programs

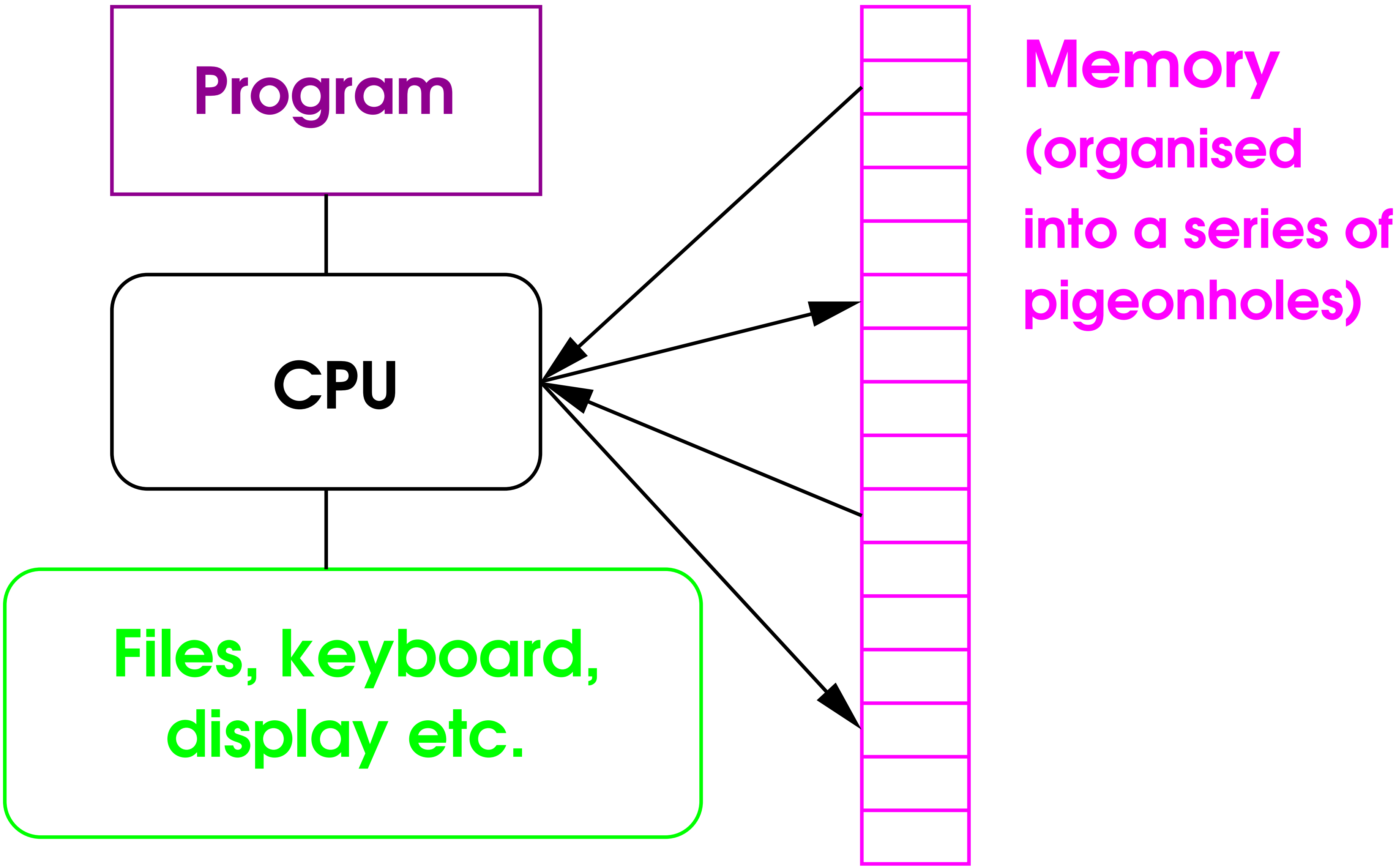
Fortran 90 is a high–level language
Sometimes called ‘‘third–generation’’ or 3GL

Uses English–like words and math–like expressions

```
Y = X + 3
PRINT *, Y
```

Compilers translate into machine instructions
A linker then creates an executable program
The operating system runs the executable

Fortran Programming Model



Algorithms and Models

An algorithm is a set of instructions
They are executed in a defined order
Doing that carries out a specific task

The above is procedural programming
Fortran 90 is a procedural language

Object–orientation is still procedural
Fortran 90 has object–oriented facilities

An Example of a Problem

Write a program to convert a time in hours, minutes and seconds to one in seconds

Algorithm:

1. Multiply the hours by 60
2. Add the minutes to the result
3. Multiply the result by 60
4. Add the seconds to the result

Logical Structure

1. Start of program
2. Reserve memory for data
3. Write prompt to display
4. Read the time in hours, minutes and seconds
5. Convert the time into seconds
6. Write out the number of seconds
7. End of program

The Program

```
PROGRAM example1
! Comments start with an exclamation mark
    IMPLICIT NONE
    INTEGER :: hours, mins, secs, temp
    PRINT *, 'Type the hours, minutes and seconds'
    READ *, hours, mins, secs
    temp = 60* ( hours*60 + mins ) + secs
    PRINT *, 'Time in seconds =', temp
END PROGRAM example1
```

High Level Structure

1. Start of program (or procedure)
   PROGRAM example1
2. Followed by the specification part
   declare types and sizes of data

3–6. Followed by the execution part
   all of the ‘action’ statements

7. End of program (or procedure)
   END PROGRAM example1

Comments do nothing and can occur anywhere

   ! Comments start with an exclamation mark

Program and File Names

- The program and file names are not related

PROGRAM QES can be in file QuadSolver.f90
Similarly for most other Fortran components

Some implementations like the same names
Sometimes converted to lower– or upper–case

The compiler documentation should tell you
It is sometimes in the system documentation
Please ask for help, but it is outside this course

The Specification Part

2. Reserve memory for data

    INTEGER :: hours, mins, secs, temp

INTEGER is the type of the variables

hours, mins, secs are used to hold input
The values read in are called the input data
temp is called a workspace variable

    also called a temporary variable etc.

The output data are 'Time . . . =' and temp
They can be any expression, not just a variable

The Execution Part

3. Write prompt to display
   ```
   PRINT *, 'Type the hours, ...'
   ```
4. Read the time in hours, minutes and seconds
   ```
   READ *, hours, mins, secs
   ```
5. Convert the time into seconds
   ```
   temp = 60*( hours*60 + mins) + secs
   ```
6. Write out the number of seconds
   ```
   PRINT *, 'Time in seconds =', temp
   ```

Assignment and Expressions

temp = 60*(hours*60 + mins) + secs

The RHS is a pseudo−mathematical expression
It calculates the value to be stored

- Expressions are very like A−level formulae

Fortran is FORmula TRANslation – remember?
We will come to the detailed rules later

- temp = stores the value in the variable

A variable is a memory cell in Fortran's model

Really Basic I/O

READ *, <variable list> reads from stdin
PRINT *, <expression list> writes to stdout

Both do input/output as human–readable text
Each I/O statement reads/writes on a new line

A list is items separated by commas (',')
Variables are anything that can store values
Expressions are anything that deliver a value

Everything else will be explained later

Repeated Instructions

The previous program handled only one value
A more flexible one would be:

1. Start of program
2. Reserve memory for data
3. Repeat this until end of file
    3.1 Read the value of seconds
    3.2 Convert to minutes and seconds
    3.3 Write out the result
4. End of Program

Sequences and Conditionals

Simple algorithms are just sequences
A simple algorithm for charging could be:

1. Calculate the bill
2. Print the invoice

Whereas it probably should have been:

1. Calculate the bill
2. If the bill exceeds minimum
    2.1 Then print the invoice
3. Otherwise
    3.1 Add bill to customer's account

Summary

There are three basic control structures:

- A simple sequence
- A conditional choice of sequences
- A repeated sequence

All algorithms can be expressed using these
In practice, other structures are convenient

Almost always need to split into simpler tasks
Even Fortran II had subroutines and functions!
Doing that is an important language–independent skill

Developing a Computer Program

There are four main steps:

1. Specify the problem
2. Analyse and subdivide into tasks
3. Write the Fortran 90 code
4. Compile and run (i.e. test)

Each step may require several iterations
You may need to restart from an earlier step

- The testing phase is very important

Errors

- If the syntax is incorrect, the compiler says so

For example: INTEGER :: ,mins, secs

- If the action is invalid, things are messier

For example: X/Y when Y is zero

/ represents division, because of the lack of ÷

You may get an error message at run–time

The program may crash, just stop or hang

It may produce nonsense values or go haywire

Introduction to Modern Fortran

Fortran Language Rules

Nick Maclaren

nmm1@cam.ac.uk

March 2014

Coverage

This course is modern, free–format source only

[If you don't understand this, don't worry]

The same applies to features covered later

Almost all old Fortran remains legal

Avoid using it, as modern Fortran is better

This mentions old Fortran only in passing

See the OldFortran course for those aspects

It describes fixed–format and conversion

Or ask questions or for help on such things, too

Important Warning

Fortran's syntax is verbose and horrible
It can fairly be described as a historical mess
Its semantics are fairly clean and consistent

Its verbosity causes problems for examples
Many of them use poor style, to be readable
And they mostly omit essential error checking

- Do what I say, don't do what I do

Sorry about that . . .

Correctness

Humans understad linguage quite well even when it isnt stroctly correc

Computers (i.e. compilers) are not so forgiving

- Programs must follow the rules to the letter

- Fortran compilers will flag all syntax errors

Good compilers will detect more than is required

But your error may just change the meaning
Or do something invalid (“undefined behaviour”)

Examples of Errors

Consider (N*M/1024+5)

If you mistype the '0' as a ')': (N*M/1)24+5)

You will get an error message when compiling
It may be confusing, but will point out a problem

If you mistype the '0' as a '–': (N*M/1–24+5)

You will simply evaluate a different formula
And get wrong answers with no error message

And if you mistype '*' as '8'?

Character Set

Letters (A to Z and a to z) and digits (0 to 9)
Letters are matched ignoring their case

And the following special characters
_ = + − * / ( ) , . ’ : ! " % & ; < > ? $
Plus space (i.e. a blank), but not tab
The end–of–line indicator is not a character

Any character allowed in comments and strings

- Case is significant in strings, and only there

Special Characters

_ = + − * / ( ) , . ’ : ! " % & ; < > ? $

slash (/) is also used for divide
hyphen (–) is also used for minus
asterisk (*) is also used for multiply

apostrophe (’) is used for single quote
period (.) is also used for decimal point

The others are described when we use them

Layout

- Do not use tab, form–feed etc. in your source

Use no positioning except space and line breaks

Compilers do bizarre things with anything else
Will work with some compilers but not others
And can produce some very strange output

Even in C, using them is a recipe for confusion
The really masochistic should ask me offline

Source Form (1)

Spaces are not allowed in keywords or names
INTEGER is not the same as INT EGER

HOURS is the same as hoURs or hours
But not HO URS – that means HO and URS

- Some keywords can have two forms

E.g. ENDDO is the same as END DO
But EN DDO is treated as EN and DDO

⇒ END DO etc. is the direction Fortran is going

Source Form (2)

- Do not run keywords and names together
  INTEGERI,J,K – illegal
  INTEGER I,J,K – allowed

- You can use spaces liberally for clarity
  INTEGER I , J , K
Exactly where you use them is a matter of taste

- Blank lines can be used in the same way
Or lines consisting only of comments

Double Colons

For descriptive names use underscore
largest_of, maximum_value or P12_56

- Best to use a double colon in declarations

Separates type specification from names

```
INTEGER :: I, J, K
```

This form is essential where attributes are used

```
INTEGER, INTENT(IN) :: I, J, K
```

Lines and Comments

A line is a sequence of up to 132 characters

A comment is from ! to the end of line
The whole of a comment is totally ignored

```
A = A+1     ! These characters are ignored
! That applies to !, & and ; too
```

Blank lines are completely ignored

```
!
! Including ones that are just comments
!
```

Use of Layout

Well laid–out programs are much more readable
You are less likely to make trivial mistakes
And much more likely to spot them!

This also applies to low–level formats, too
E.g. 1.0e6 is clearer than 1.e6 or .1e7

- None of this is Fortran–specific

Use of Comments

Appropriate commenting is very important
This course does not cover that topic
And, often, comments are omitted for brevity

“How to Help Programs Debug Themselves”
Gives guidelines on how best to use comments

- This isn’t Fortran–specific, either

Use of Case

- Now, this IS Fortran–specific!

It doesn't matter what case convention you use

- But DO be moderately† consistent!

Very important for clarity and editing/searching

For example:
UPPER case for keywords, lower for names
You may prefer Capitalised names

† *A foolish consistency is the hobgoblin of little minds*

Statements and Continuation

- A program is a sequence of statements
Used to build high–level constructs
Statements are made up out of lines

- Statements are continued by appending &

```
A = B + C + D + E + &
      F + G + H
```

Is equivalent to

```
A = B + C + D + E + F + G + H
```

Other Rules (1)

Statements can start at any position

- Use indentation to clarify your code

```
IF (a > 1.0) THEN
     b = 3.0
ELSE
     b = 2.0
END IF
```

- A number starting a statement is a label

```
10 A = B + C
```

The use of labels is described later

Other Rules (2)

You can put multiple statements on a line

```
a = 3 ;   b = 4 ;    c = 5
```

Overusing that can make a program unreadable
But it can clarify your code in some cases

Avoid mixing continuation with that or comments
It works, but can make code very hard to read

```
a = b + c ; d = e + f + &
    g + h
a = b + c + & ! More coming ...
    d = e + f + g + h
```

Breaking Character Strings

- Continuation lines can start with an &

Preceding spaces and the & are suppressed

The following works and allows indentation:

```
PRINT 'Assume that this string &
          &is far too long and complic&
          &ated to fit on a single line'
```

The initial & avoids including excess spaces
And avoids problems if the text starts with !

This may also be used to continue any line

Names

Up to 31 (now 63) letters, digits and underscores

- Names must start with a letter

Upper and lower case are equivalent
DEPTH, Depth and depth are the same name

The following are valid Fortran names

A, AA, aaa, Tax, INCOME, Num1, NUM2, NUM333, N12MO5, atmospheric_pressure, Line_Colour, R2D2, A_21_173_5a

Invalid Names

The following are invalid names

1A	does not begin with a letter
_B	does not begin with a letter
Depth$0	contains an illegal character '$'
A–3	would be interpreted as subtract 3 from A
B.5:	illegal characters '.' and ':'

A_name_made_up_of_more_than_31_letters

too long, 38 characters

Compiling and Testing

We shall use the gfortran under Linux
PWF/MCS/DS Windows does not have a Fortran
Using any Fortran compiler is much the same

Please ask about anything you don't understand
Feel free to bring problems with other Fortrans
Feel free to use gdb if you know it

Solutions to exercises available from
http://people.ds.cam.ac.uk/nmm1/Fortran/Answers

Instructions

If running Microsoft Windows, CTRL–ALT–DEL
Select Restart and then Linux
Log into Linux and start a shell and an editor
Create programs called prog.f90, fred.f90 etc.

- Run by typing commands like
  ```
  nagfor -C=all -o fred fred.f90
  ./fred
  ```

- Analyse what went wrong
- Fix bugs and retry

Instructions

- Run by typing commands like

```
gfortran -g -O3 -Wall -Wextra -o fred fred.f90
./fred
```

- Analyse what went wrong
- Fix bugs and retry

Introduction to Modern Fortran

Data Types and Basic Calculation

Nick Maclaren

nmm1@cam.ac.uk

March 2014

Data Types (1)

- INTEGER for exact whole numbers
  e.g. 1, 100, 534, −18, −654321 etc.

In maths, an approximation to the ring $\mathbb{Z}$

- REAL for approximate, fractional numbers
  e.g. 1.1, 3.0, 23.565, π, exp(1), etc.

In maths, an approximation to the field $\mathbb{R}$

- COMPLEX for complex, fractional numbers
  e.g. (1.1, −23.565), etc.

In maths, an approximation to the field $\mathbb{C}$

Data Types (2)

- LOGICAL for truth values

These may have only values true or false

e.g. .TRUE. , .FALSE.

These are often called boolean values

- CHARACTER for strings of characters

e.g. ‘?’, ‘Albert Einstein’, ‘X + Y = ’, etc.

The string length is part of the type in Fortran

There is no separate character type, unlike C

There is more on this later

Integers (1)

- Integers are restricted to lie in a finite range

Typically $\pm$2147483647 (-2^{31} to 2^{31}–1)
Sometimes $\pm 9.23 \times 10^{17}$ (-2^{63} to 2^{63}–1)

A compiler may allow you to select the range
Often including $\pm$32768 (-2^{15} to 2^{15}–1)

Older and future systems may have other ranges
There is more on the arithmetic and errors later

Integers (2)

Fortran uses integers for:

- Loop counts and loop limits
- An index into an array or a position in a list
- An index of a character in a string
- As error codes, type categories etc.

Also use them for purely integral values
E.g. calculations involving counts (or money)
They can even be used for bit masks (see later)

Integer Constants

Usually, an optional sign and one or more digits

e.g. 0, 123, −45, +67, 00012345

E.g. '60' in minutes = minutes + 60*hours

Also allowed in binary, octal and hexadecimal

e.g. B'011001', O'35201', Z'a12bd'

- As with names, the case is irrelevant

There is a little more, which is covered later

Reals

- Reals are held as floating–point values

These also have a finite range and precision

It is essential to use floating–point appropriately

- Much of the Web is misleading about this

This course will mention only the bare minimum
See "How Computers Handle Numbers"
There is simplified version of that later on

Reals are used for continuously varying values
Essentially just as you were taught at A–level

IEEE 754

You can assume a variant of IEEE 754
You should almost always use IEEE 754 64–bit
There is information on how to select it later

IEEE 754 32–, 64– and 128–bit formats are:
$\mathbf{10^{-38}}$ to $\mathbf{10^{+38}}$ and 6–7 decimal places
$\mathbf{10^{-308}}$ to $\mathbf{10^{+308}}$ and 15–16 decimal places
$\mathbf{10^{-4932}}$ to $\mathbf{10^{+4932}}$ and 33–34 decimal places

Older and future systems may be different

Real Constants

- Real constants must contain a decimal point or an exponent

They can have an optional sign, just like integers

The basic fixed–point form is anything like:

    123.456, −123.0, +0.0123, 123., .0123
    0012.3, 0.0, 000., .000

- Optionally followed E or D and an exponent

    1.0E6, 123.0D−3, .0123e+5, 123.d+06, .0e0

1E6 and 1D6 are also valid Fortran real constants

Complex Numbers

This course will generally ignore them
If you don't know what they are, don't worry

These are (real, imaginary) pairs of REALs
I.e. Cartesian notation, as at A−level

Constants are pairs of reals in parentheses
E.g. (1.23, −4.56) or (−1.0e−3, 0.987)

Declaring Numeric Variables

Variables hold values of different types

```
INTEGER :: count, income, mark
REAL :: width, depth, height
```

You can get all undeclared variables diagnosed
Add the statement IMPLICIT NONE at the start
of every program, subroutine, function etc.

If not, variables are declared implicitly by use
Names starting with I–N are INTEGER
Ones with A–H and O–Z are REAL

Implicit Declaration

- This is a common source of errors

```
REAL :: metres, inches
inch3s = meters*30.48
```

The effects can be even worse for function calls
Ask offline if you want to know the details

- Also the default REAL type is a disaster

Too inaccurate for practical use (see later)

- You should always use IMPLICIT NONE

Important Warning!

- I shall NOT do that myself

I warned you about this in the previous lecture
The problem is fitting all the text onto a slide
I shall often rely on implicit typing :–(

- Do what I say, don't do what I do

If I omit it in example files, it is a BUG

Assignment Statements

The general form is

<variable> = <expression>

This is actually very powerful (see later)

This first evaluates the expression on the RHS

It then stores the result in the variable on the LHS

It replaces whatever value was there before

For example:

```
Max = 2 * Min
Sum = Sum + Term1 + Term2 + (Eps * Err)
```

Arithmetic Operators

There are five built–in numeric operations

+	addition
–	subtraction
*	multiplication
/	division
**	exponentiation (i.e. raise to the power of)

- Exponents can be any arithmetic type:
  INTEGER, REAL or COMPLEX

Generally, it is best to use them in that order

Examples

Some examples of arithmetic expressions are

```
A*B–C
A + C1 – D2
X + Y/7.0
2**K
A**B + C
A*B–C
(A + C1) – D2
A + (C1 – D2)
P**3/((X+Y*Z)/7.0–52.0)
```

Operator Precedence

Fortran uses normal mathematical conventions

- Operators bind according to precedence
- And then generally, from left to right

The precedence from highest to lowest is

**	exponentiation
* /	multiplication and division
+ −	addition and subtraction

- Parentheses ('(' and ')') are used to control it

Use them whenever the order matters or it is clearer

Examples

X + Y * Z is equivalent to X + (Y * Z)

X + Y / 7.0 is equivalent to X + (Y / 7.0)

A – B + C is equivalent to (A – B) + C

A + B ** C is equivalent to A + (B ** C)

– A ** 2 is equivalent to – (A ** 2)

A – (((B + C))) is equivalent to A – (B + C)

- You can force any order you like

  (X + Y) * Z

Adds X to Y and then multiplies by Z

Warning

X + Y + Z may be evaluated as any of
X + (Y + Z) or (X + Y) + Z or Y+ (X + Z) or . . .

Fortran defines what an expression means
It does not define how it is calculated

They are all mathematically equivalent
But may sometimes give slightly different results

See “How Computers Handle Numbers” for more

Precedence Problems

Mathematical conventions vary in some aspects

A / B * C – is it A / (B * C) or (A / B) * C?

A ** B ** C – is it A ** (B ** C) or (A ** B) ** C?

Fortran specifies that:

A / B * C is equivalent to (A / B) * C

A ** B ** C is equivalent to A ** (B ** C)

- Yes, ** binds from right to left!

Parenthesis Problems

Always use parentheses in ambiguous cases
If only to imply "Yes, I really meant that"

And to help readers used to different rules
Programming languages vary in what they do

Be careful of over–doing it – what does this do?

```
(((A+(P*R+B)/2+B**3)/(4/Y)*C+D))+E)
```

- Several, simpler statements is better style

Integer Expressions

I.e. ones of integer constants and variables

```
INTEGER :: K, L, M
N = K*(L+2)/M**3-N
```

These are evaluated in integer arithmetic

- Division always truncates towards zero

If K = 4 and L = 5, then K+L/2 is 6
(−7)/3 and 7/(−3) are both −2

Mixed Expressions

INTEGER and REAL is evaluated as REAL
Either and COMPLEX goes to COMPLEX

Be careful with this, as it can be deceptive

```
INTEGER :: K = 5
REAL :: X = 1.3
X = X+K/2
```

That will add 2.0 to X, not 2.5
K/2 is still an INTEGER expression

Conversions

There are several ways to force conversion

- Intrinsic functions INT, REAL and COMPLEX

```
X = X+REAL(K)/2
N = 100*INT(X/1.25)+25
```

You can use appropriate constants
You can even add zero or multiply by one

```
X = X+K/2.0
X = X+(K+0.0)/2
```

The last isn't very nice, but works well enough
And see later about KIND and precision

Mixed-type Assignment

<real variable> = <integer expression>

- The RHS is converted to REAL

Just as in a mixed–type expression

<integer variable> = <real expression>

- The RHS is truncated to INTEGER

It is always truncated towards zero

Similar remarks apply to COMPLEX

- The imaginary part is discarded, quietly

The RHS is evaluated independently of the LHS

Examples

```
INTEGER :: K = 9, L = 5, M = 3, N
REAL :: X, Y, Z
X = K ; Y = L ; Z = M
N = (K/L)*M
N = (X/Y)*Z
```

N will be 3 and 5 in the two cases

$(-7)/3 = 7/(-3) = -2$ and $7/3 = (-7)/(-3) = 2$

Numeric Errors

See “How Computers Handle Numbers”
This a a very minimal summary

- Overflowing the range is a serious error
As is dividing by zero (e.g. 123/0 or 0.0/0.0)
Fortran does not define what those cases do

- Each numeric type may behave differently
Even different compiler options will, too
- And do not assume results are predictable

Examples

Assume the INTEGER range is ± 2147483647
And the REAL range is $\pm 10^{38}$

- Do you know what this is defined to do?

```
INTEGER :: K = 1000000
REAL :: X = 1.0e20
PRINT *, (K*K)/K, (X*X)/X
```

- The answer is "anything" – and it means it

Compilers optimise on the basis of no errors
Numeric errors can cause logical errors

Numeric Non-Errors (1)

- Conversion to a lesser type loses information

You will get no warning of this, unfortunately

REAL ⇒ INTEGER truncates towards zero
COMPLEX ⇒ REAL drops the imaginary part
COMPLEX ⇒ INTEGER does both of them

That also applies when dropping in precision
E.g. assigning a 64–bit real to a 32–bit one

Numeric Non-Errors (2)

Fortran does NOT specify the following
But it is true on all systems you will use

Results too small to represent are not errors

- They will be replaced by zero if necessary

- Inexact results round to the nearest value

That also applies when dropping in precision

Intrinsic Functions

Built–in functions that are always available

- No declaration is needed – just use them

For example:

```
Y = SQRT(X)
PI = 4.0 * ATAN(1.0)
Z = EXP(3.0*Y)
X = REAL(N)
N = INT(X)
Y = SQRT(-2.0*LOG(X))
```

Intrinsic Numeric Functions

```
REAL(n)       ! Converts its argument n to REAL
INT(x)        ! Truncates x to INTEGER (to zero)
AINT(x)       ! The result remains REAL
NINT(x)       ! Converts x to the nearest INTEGER
ANINT(x)      ! The result remains REAL
ABS(x)        ! The absolute value of its argument
! Can be used for INTEGER, REAL or COMPLEX
MAX(x,y,...)  ! The maximum of its arguments
MIN(x,y,...)  ! The minimum of its arguments
MOD(x,y)      ! Returns x modulo y
```

And there are more – some are mentioned later

Intrinsic Mathematical Functions

```
SQRT(x)    ! The square root of x
EXP(x)     ! e raised to the power x
LOG(x)     ! The natural logarithm of x
LOG10(x)   ! The base 10 logarithm of x

SIN(x)     ! The sine of x, where x is in radians
COS(x)     ! The cosine of x, where x is in radians
TAN(x)     ! The tangent of x, where x is in radians
ASIN(x)    ! The arc sine of x in radians
ACOS(x)    ! The arc cosine of x in radians
ATAN(x)    ! The arc tangent of x in radians
```

And there are more – see the references

Bit Masks

As in C etc., integers are used for these
Use is by weirdly–named functions (historical)

Bit indices start at zero
Bit K has value 2^K (little–endian)
As usual, stick to non–negative integers

- A little tedious, but very easy to use

Bit Intrinsics

```
BIT_SIZE(x)         ! The number of bits in x
BTEST(x, n)         ! Test bit n of x
IBSET(x, n)         ! Set bit n of x
IBCLR(x, n)         ! Clear bit n of x
IBITS(x, m, n)      ! Extract n bits
NOT(x)              ! NOT x
IAND(x, y)          ! x AND y
IOR(x, y)           ! x OR y
IEOR(x, y)          ! x (exclusive or) y
ISHFT(x, n)         ! Logical shift
ISHFTC(x, n, [k])   ! Circular shift
```

Logical Type

These can take only two values: true or false

.TRUE. and .FALSE.

- Their type is LOGICAL (not BOOL)

```
LOGICAL :: red, amber, green

IF (red) THEN
    PRINT *, 'Stop'
    red = .False. ; amber = .True. ; green = .False.
ELSIF (red .AND. amber) THEN
. . .
```

Relational Operators

- Relations create LOGICAL values

These can be used on any other built–in type

== (or .EQ.) equal to

/= (or .NE.) not equal to

These can be used only on INTEGER and REAL

< (or .LT.) less than

<= (or .LE.) less than or equal

> (or .GT.) greater than

>= (or .GE.) greater than or equal

See "How Computers Handle Numbers" for more

Logical Expressions

Can be as complicated as you like

Start with .TRUE., .FALSE. and relations
Can use parentheses as for numeric ones

.NOT., .AND. and .OR.
.EQV. must be used instead of ==
.NEQV. must be used instead of /=

- Fortran is not like C–derived languages

LOGICAL is not a sort of INTEGER

Short Circuiting

```
LOGICAL :: flag
flag = ( Fred( ) > 1.23 .AND. Joe( ) > 4.56 )
```

Fred and Joe may be called in either order
If Fred returns 1.1, then Joe may not be called
If Joe returns 3.9, then Fred may not be called

Fortran expressions define the answer only
The behaviour is up to the compiler
One of the reasons that it is so optimisable

Character Type

Used when strings of characters are required

Names, descriptions, headings, etc.

- Fortran's basic type is a fixed–length string

Unlike almost all more recent languages

- Character constants are quoted strings

```
PRINT *, 'This is a title'
PRINT *, "And so is this"
```

The characters between quotes are the value

Character Data

- The case of letters is significant in them
Multiple spaces are not equivalent to one space
Any representable character may be used

The only Fortran syntax where the above is so
Remember the line joining method?

In ’Time^^=^^13:15’, with ‘^’ being a space
The character string is of length 15
Character 1 is T, 8 is a space, 10 is 1 etc.

Character Constants

```
"This has UPPER, lower and MiXed cases"
'This has a double quote (") character'
"Apostrophe (') is used for single quote"
"Quotes ("") are escaped by doubling"
'Apostrophes ('') are escaped by doubling'
'ASCII `, |, ~, ^, @, # and \ are allowed here'

"Implementations may do non-standard things"
'Backslash (\) MAY need to be doubled'
"Avoid newlines, tabs etc. for your own sanity"
```

Character Variables

```
CHARACTER :: answer, marital_status
CHARACTER(LEN=10) :: name, dept, faculty
CHARACTER(LEN=32) :: address
```

answer and marital_status are each of length 1
They hold precisely one character each
answer might be blank, or hold 'Y' or 'N'

name, dept and faculty are of length 10
And address is of length 32

Another Form

```
CHARACTER :: answer*1,          &
   marital_status*1, name*10,         &
   dept*10, faculty*10, address*32
```

While this form is historical, it is more compact

- Don't mix the forms – this is an abomination

```
CHARACTER(LEN=10) :: dept, faculty, addr*32
```

- For obscure reasons, using LEN= is cleaner

It avoids some arcane syntactic "gotchas"

Character Assignment

```
CHARACTER(LEN=6) :: forename, surname
forename = 'Nick'
surname = 'Maclaren'
```

forename is padded with spaces ('Nick^^')
surname is truncated to fit ('Maclar')

- Unfortunately, you won't get told

But at least it won't overwrite something else

Character Concatenation

Values may be joined using the // operator

```
CHARACTER(LEN=6) :: identity, A, B, Z
identity = 'TH' // 'OMAS'
A = 'TH' ; B = 'OMAS'
Z = A // B
PRINT *, Z
```

Sets identity to 'THOMAS'
But Z looks as if it is still 'TH' – why?

// does not remove trailing spaces
It uses the whole length of its inputs

Substrings

If Name has length 9 and holds 'Marmaduke'
Name(1:1) would refer to 'M'
Name(2:4) would refer to 'arm'
Name(6:) would refer to 'duke' – note the form!

We could therefore write statements such as

```
CHARACTER :: name*20, surname*18, title*4
name = 'Dame Edna Everage'
title = name(1:4)
surname = name(11:)
```

Example

This is not an example of good style!

```
PROGRAM message
  IMPLICIT NONE
  CHARACTER :: mess*72, date*14, name*40
  mess = 'Program run on'
  mess(30:) = 'by'
  READ *, date, name
  mess(16:29) = date
  mess(33:) = name
  PRINT *, mess
END PROGRAM message
```

Warning – a “Gotcha”

CHARACTER substrings look like array sections
But there is no equivalent of array indexing

```
CHARACTER :: name*20, temp*1
temp = name(10)
```

- name(10) is an implicit function call

Use name(10:10) to get the tenth character

CHARACTER variables come in various lengths
name is not made up of 20 variables of length 1

Character Intrinsics

```
LEN(c)                   ! The STORAGE length of c
TRIM(c)                  ! c without trailing blanks
ADJUSTL(C)               ! With leading blanks removed
INDEX(str,sub)           ! Position of sub in str
SCAN(str,set)            ! Position of any character in set
REPEAT(str,num)          ! num copies of str, joined
```

And there are more – see the references

Examples

```
name = '   Bloggs      '
newname = TRIM(ADJUSTL(name))
```

newname would contain 'Bloggs'

```
CHARACTER(LEN=6) :: A, B, Z
A = 'TH' ; B = 'OMAS'
Z = TRIM(A) // B
```

Now Z gets set to 'THOMAS' correctly!

Collation Sequence

This controls whether "ffred" < "Fred" or not

- Fortran is not a locale–based language

It specifies only the following

```
'A' < 'B' < ... < 'Y' < 'Z'     | These ranges
'a' < 'b' < ... < 'y' < 'z'     | will not
'0' < '1' < ... < '8' < '9'     | overlap
' ' is less than all of 'A', 'a' and '0'
```

A shorter operand is extended with blanks (' ')

Working with CHARACTER

This is one of the things that has been omitted
It's there in the notes, if you are interested

Can assign, concatenate and compare them
Can extract substrings and do lots more

But, for the academy, you don't need to do that

- Skip the practicals that need those facilities

Named Constants (1)

- These have the PARAMETER attribute

```
REAL, PARAMETER :: pi = 3.14159
INTEGER, PARAMETER :: maxlen = 100
```

They can be used anywhere a constant can be

```
    CHARACTER(LEN=maxlen) :: string
    circum = pi * diam
    IF (nchars < maxlen) THEN
        ...
```

Named Constants (2)

Why are these important?

They reduce mistyping errors in long numbers
Is 3.14159265358979323846D0 correct?

They can make formulae etc. much clearer
Much clearer which constant is being used

They make it easier to modify the program later
`INTEGER, PARAMETER :: MAX_DIMENSION = 10000`

Named Character Constants

```
CHARACTER(LEN=*), PARAMETER ::          &
        author = 'Dickens', title = 'A Tale of Two Cities'
```

LEN=* takes the length from the data

It is permitted to define the length of a constant
The data will be padded or truncated if needed

- But the above form is generally the best

Named Constants (3)

- Expressions are allowed in constant values

```
REAL, PARAMETER :: pi = 3.14135,    &
      pi_by_4 = pi/4, two_pi = 2*pi,    &
      e = exp(1.0)

CHARACTER(LEN=*), PARAMETER ::    &
      all_names = 'Pip, Squeak, Wilfred',    &
      squeak = all_names(6:11)
```

Generally, anything reasonable is allowed

- It must be determinable at compile time

Initialisation

- Variables start with undefined values
They often vary from run to run, too

- Initialisation is very like defining constants
Without the PARAMETER attribute

```
INTEGER :: count = 0, I = 5, J = 100
REAL :: inc = 1.0E5, max = 10.0E5, min = −10.0E5
CHARACTER(LEN=10) :: light = 'Amber'
LOGICAL  :: red = .TRUE., blue = .FALSE.,    &
    green = .FALSE.
```

Information for Practicals

A program has the following basic structure:

```
PROGRAM name
Declarations
Other statements
END PROGRAM name
```

Read and write data from the terminal using:

```
READ *, variable [ , variable ]...
PRINT *, expression [ , expression ]...
```

Introduction to Modern Fortran

Control Constructs

Nick Maclaren

nmm1@cam.ac.uk

March 2014

Control Constructs

These change the sequential execution order
We cover the main constructs in some detail
We shall cover procedure call later

The main ones are:

- Conditionals (IF etc.)
- Loops (DO etc.)
- Switches (SELECT/CASE etc.)
- Branches (GOTO etc.)

Loops are by far the most complicated

Single Statement IF

Oldest and simplest is the single statement IF

IF (logical expression) simple statement

If the LHS is .True., the RHS is executed

If not, the whole statement has no effect

```
IF (MOD(count,1000) == 0)    &
        PRINT *, 'Reached', count
IF (X < A) X = A
```

Unsuitable for anything complicated

- Only action statements can be on the RHS

No IFs or statements containing blocks

Block IF Statement

A block IF statement is more flexible
The following is the most traditional form of it

```
IF (logical expression) THEN
    then block of statements
ELSE
    else block of statements
END IF
```

If the expr. is .True., the first block is executed
If not, the second one is executed

END IF can be spelled ENDIF

Example

```
LOGICAL :: flip

IF (flip .AND. X /= 0.0) THEN
    PRINT *, 'Using the inverted form'
    X = 1.0/A
    Y = EXP(-A)
ELSE
    X = A
    Y = EXP(A)
END IF
```

Omitting the ELSE

The ELSE and its block can be omitted

```
IF (X > Maximum) THEN
    X = Maximum
END IF

IF (name(1:4) == "Miss" .OR.        &
          name(1:4) == "Mrs.") THEN
    name(1:3) = "Ms."
    name(4:) = name(5:)
END IF
```

Including ELSE IF Blocks (1)

ELSE IF functions much like ELSE and IF

```
IF (X < 0.0) THEN          ! This is tried first
    X = A
ELSE IF (X < 2.0) THEN     ! This second
    X = A + (B-A)*(X-1.0)
ELSE IF (X < 3.0) THEN     ! And this third
    X = B + (C-B)*(X-2.0)
ELSE      ! This is used if none succeed
    X = C
END IF
```

Including ELSE IF Blocks (2)

You can have as many ELSE IFs as you like
There is only one END IF for the whole block

All ELSE IFs must come before any ELSE
Checked in order, and the first success is taken

You can omit the ELSE in such constructs

ELSE   IF can be spelled ELSE IF

Named IF Statements (1)

The IF can be preceded by <name> :
And the END IF followed by <name> – note!
And any ELSE IF/THENand ELSE may be

```
gnole : IF (X < 0.0) THEN
        X = A
ELSE IF (X < 2.0) THEN gnole
        X = A + (B−A)*(X−1.0)
ELSE gnole
        X = C
END IF gnole
```

Named IF Statements (2)

The IF construct name must match and be distinct
A great help for checking and clarity

- You should name at least all long IFs

If you don't nest IFs much, this style is fine

```
gnole : IF (X < 0.0) THEN
        X = A
ELSE IF (X < 2.0) THEN
        X = A + (B−A)*(X−1.0)
ELSE
        X = C
END IF gnole
```

Block Contents

- Almost any executable statements are OK

Both kinds of IF, complete loops etc.
You may never notice the few restrictions

That applies to all of the block statements
IF, DO, SELECT etc.
And all of the blocks within an IF statement

- Avoid deep levels and very long blocks

Purely because they will confuse human readers

Example

```
phasetest: IF (state == 1) THEN
      IF (phase < pi_by_2) THEN
            . . .
      ELSE
            . . .
      END IF
ELSE IF (state == 2) THEN phasetest
      IF (phase > pi) PRINT *, 'A bit odd here'
ELSE phasetest
      IF (phase < pi) THEN
            . . .
      END IF
END IF phasetest
```

Basic Loops (1)

- A single loop construct, with variations

The basic syntax is:

```
[ loop name : ] DO [ [ , ] loop control ]
        block
END DO [ loop name ]
```

loop name and loop control are optional
With no loop control, it loops indefinitely

END DO can be spelled ENDDO
The comma after DO is entirely a matter of taste

Basic Loops (2)

```
DO          ! Implement the Unix 'yes' command
    PRINT *, 'y'
END DO

yes: DO
    PRINT *, 'y'
END DO yes
```

The loop name must match and be distinct

- You should name at least all long loops

A great help for checking and clarity

Other of it uses are described later

Indexed Loop Control

The loop control has the following form

<integer variable> = <LWB> , <UPB>

The bounds can be any integer expressions

The variable starts at the lower bound
A: If it exceeds the upper bound, the loop exits
The loop body is executed †
The variable is incremented by one
The loop starts again from A

† See later about EXIT and CYCLE

Examples

```
DO I = 1 , 3
    PRINT *, 7*I-3
END DO
```

Prints 3 lines containing 4, 11 and 18

```
DO I = 3 , 1
    PRINT *, 7*I-3
END DO
```

Does nothing

Using an increment

The general form is

<var> = <start> , <finish> , <step>

<var> is set to <start>, as before
<var> is incremented by <step>, not one
Until it exceeds <finish> (if <step> is positive)
Or is smaller than <finish> (if <step> is negative)

- The direction depends on the sign of <step>
The loop is invalid if <step> is zero, of course

Examples

```
DO I = 1 , 20 , 7
    PRINT *, I
END DO
```

Prints 3 lines containing 1, 8 and 15

```
DO I = 20 , 1 , 7
    PRINT *, I
END DO
```

Does nothing

Examples

```
DO I = 20 , 1 , -7
    PRINT *, I
END DO
```

Prints 3 lines containing 20, 13 and 6

```
DO I = 1 , 20 , -7
    PRINT *, I
END DO
```

Does nothing

Mainly for C Programmers

The control expressions are calculated on entry

- Changing their variables has no effect

- It is illegal to assign to the loop variable

```
DO index = i*j, n**21, k
    n = 0; k = -1        ! Does not affect the loop
    index = index+1      ! Is forbidden
END DO
```

Loop Control Statements

EXIT leaves the innermost loop
CYCLE skips to the next iteration
EXIT/CYCLE name is for the loop named name
These are usually used in single–statement IFs

```
DO
    x = read_number()
    IF (x < 0.0) EXIT
    count = count+1; total = total+x
    IF (x == 0.0) CYCLE
    . . .
END DO
```

Example

```
INTEGER :: state(right), table(left , right)
FirstMatch = 0
outer: DO i = 1 , right
      IF (state(right) /= OK) CYCLE
      DO j = 1 , left
            IF (found(table(j,i)) THEN
                  FirstMatch = i
                  EXIT outer
            END IF
      END DO
END DO outer
```

Warning

What is the control variable's value after loop exit?

- It is undefined after normal exit

Web pages and ignoramuses often say otherwise

It IS defined if you leave by EXIT
Generally, it is better not to rely on that fact

WHILE Loop Control

The loop control has the following form

```
WHILE ( <logical expression> )
```

The expression is reevaluated for each cycle
The loop exits as soon as it becomes .FALSE.
The following are equivalent:

```
DO WHILE ( <logical expression> )

DO
    IF (.NOT. ( <logical expression> )) EXIT
```

CONTINUE

CONTINUE is a statement that does nothing
It used to be fairly common, but is now rare

Its main use is in blocks that do nothing
Empty blocks aren't allowed in Fortran

Otherwise mainly a placeholder for labels
This is purely to make the code clearer

But it can be used anywhere a statement can

RETURN and STOP

RETURN returns from a procedure

- It does not return a result

How to do that is covered under procedures

STOP halts the program cleanly

- Do not spread it throughout your code

Call a procedure to tidy up and finish off

Multi-way IFs

```
IF (expr == val1) THEN
    x = 1.23
ELSE IF (expr >= val2 .AND. expr <= val3) THEN
    CONTINUE
ELSE IF (expr == val4) THEN
    x = x + 4.56
ELSE
    x = 7.89 - x
END IF
```

Very commonly, expr is always the same
And all of the vals are constant expressions
Then there is another way of coding it

SELECT CASE (1)

```
PRINT *, 'Happy Birthday'
SELECT CASE (age)
CASE(18)
      PRINT *, 'You can now vote'
CASE(40)
      PRINT *, 'And life begins again'
CASE(60)
      PRINT *, 'And free prescriptions'
CASE(100)
      PRINT *, 'And greetings from the Queen'
CASE DEFAULT
      PRINT *, 'It''s just another birthday'
END SELECT
```

SELECT CASE (2)

- The CASE clauses are statements

To put on one line, use 'CASE(18) ; <statement>'

The values must be constant expressions

INTEGER, CHARACTER or LOGICAL

You can specify ranges for the first two

```
CASE (−42:42)        ! −42 to 42 inclusive
CASE (42:)           ! 42 or above
CASE (:42)           ! Up to and including 42
```

Be careful with CHARACTER ranges

SELECT CASE (3)

SELECT CASE can be spelled SELECTCASE
END SELECT can be spelled ENDSELECT
- CASE DEFAULT but NOT CASEDEFAULT

SELECT and CASE can be named, like IF

- It is an error for the ranges to overlap

It is not an error for ranges to be empty
Empty ranges don't overlap with anything
It is not an error for the default to be unreachable

Labels and GOTO

Warning: this area gets seriously religious!

Most executable statements can be labelled
GOTO <label> branches directly to the label

In old Fortran, you needed to use a lot of these

- Now, you should almost never use them

If you think you need to, consider redesigning

- Named loops, EXIT and CYCLE are better

Remaining uses of GOTO

- Useful for branching to clean–up code

E.g. diagnostics, undoing partial updates etc.
This is by FAR the main remaining use

Fortran does not have any cleaner mechanisms
E.g. it has no exception handling constructs

- They make a few esoteric algorithms clearer

E.g. certain finite–state machine models
I have seen such code 3–4 times in 40+ years

Clean-up Code (1)

```
SUBROUTINE Fred
DO . . .
      CALL SUBR (arg1 , arg2 , . . . , argn , ifail)
      IF (ifail /= 0) GOTO 999
END DO
. . . lots more similar code . . .
RETURN

999 SELECT CASE (ifail)
CASE(1) ! Code for ifail = 1
      . . .
CASE(2) ! Code for ifail = 2
      . . .
END SUBROUTINE Fred
```

Clean-up Code (2)

Many people regard this as better style:

```
SUBROUTINE Fred
DO . . .
    CALL SUBR (arg1 , arg2 , . . . , argn , ifail)
    IF (ifail /= 0) GOTO 999
END DO

999 CONTINUE
SELECT CASE (ifail)
CASE(1) ! Code for ifail = 1
    . . .
END SUBROUTINE Fred
```

Other Mechanisms

Switches, branches and labels are omitted
They're there in the notes, if you are interested

- You very rarely need to use them, anyway

Introduction to Modern Fortran

Array Concepts

Nick Maclaren

nmm1@cam.ac.uk

March 2014

Array Declarations

Fortran is the array–handling language
Applications like Matlab descend from it

You can do almost everything you want to

- Provided that your arrays are rectangular

Irregular arrays are possible via pointers

- Start by using the simplest features only

When you need more, check what Fortran has

We will cover the basics and a bit more

Array Declarations

Attributes qualify the type in declarations
Immediately following, separated by a comma

The DIMENSION attribute declares arrays
It has the form DIMENSION(<dimensions>)
Each <dimension> is <lwb>:<upb>

For example:

```
INTEGER, DIMENSION(0:99) :: table
REAL, DIMENSION(−10:10, −10:10) :: values
```

Examples of Declarations

Some examples of array declarations:

```
INTEGER, DIMENSION(0:99) :: arr1, arr2, arr3
INTEGER, DIMENSION(1:12) :: days_in_month
CHARACTER(LEN=10), DIMENSION(1:250) :: names
CHARACTER(LEN=3), DIMENSION(1:12) :: months
REAL, DIMENSION(1:350) :: box_locations
REAL, DIMENSION(–10:10, –10:10) :: pos1, pos2
REAL, DIMENSION(0:5, 1:7, 2:9, 1:4, –5:–2) :: bizarre
```

Lower Bounds of One

Lower bounds of one (1:) can be omitted

```
INTEGER, DIMENSION(12) :: days_in_month
CHARACTER(LEN=10), DIMENSION(250) :: names
CHARACTER(LEN=3), DIMENSION(12) :: months
REAL, DIMENSION(350) :: box_locations
REAL, DIMENSION(0:5, 7, 2:9, 4, -5:-2) :: bizarre
```

It is entirely a matter of taste whether you do

- C/C++/Python users note ONE not ZERO

Alternative Form

The same base type but different bounds

```
INTEGER :: arr1(0:99), arr2(0:99), arr3(0:99), &
        days_in_month(1:12)
REAL :: box_locations(1:350), &
        pos1(–10:10, –10:10), pos2(–10:10, –10:10), &
        bizarre(0:5, 1:7, 2:9, 1:4, –5:–2)
```

But this is thoroughly confusing:

```
INTEGER, DIMENSION(0:99) :: arr1, arr2, arr3, &
        days_in_month(1:12), extra_array, &
        days_in_leap_year(1:12)
```

Terminology (1)

```
REAL :: A(0:99), B(3, 6:9, 5)
```

The rank is the number of dimensions
A has rank 1 and B has rank 3

The bounds are the upper and lower limits
A has bounds 0:99 and B has 1:3, 6:9 and 1:5

A dimension’s extent is the UPB−LWB+1
A has extent 100 and B has extents 3, 4 and 5

Terminology (2)

```
REAL :: A(0:99), B(3, 6:9, 5)
```

The size is the total number of elements

A has size 100 and B has size 60

The shape is its rank and extents

A has shape (100) and B has shape (3,4,5)

Arrays are conformable if they share a shape

- The bounds do not have to be the same

Array Element References

An array index can be any integer expression
E.g. months(J), selects the Jth month

```
INTEGER, DIMENSION(-50:50) :: mark
DO I = -50, 50
    mark(I) = 2*I
END DO
```

Sets mark to −100, −98, ..., 98, 100

Index Expressions

```
INTEGER, DIMENSION(1:80) :: series
DO K = 1, 40
    series(2*K) = 2*K-1
    series(2*K-1) = 2*K
END DO
```

Sets the even elements to the odd indices
And vice versa

You can go completely overboard, too

```
series(int(1.0+80.0*cos(123.456))) = 42
```

Example of Arrays – Sorting

Sort a list of numbers into ascending order
The top–level algorithm is:

1. Read the numbers and store them in an array.
2. Sort them into ascending order of magnitude.
3. Print them out in sorted order.

Selection Sort

This is NOT how to write a general sort
It takes O(N^2) time – compared to O($Nlog(N)$)

For each location J from 1 to N–1
    For each location K from J+1 to N
        If the value at J exceeds that at K
            Then swap them
    End of loop
End of loop

Selection Sort (1)

```
PROGRAM sort10
      INTEGER, DIMENSION(1:10) :: nums
      INTEGER :: temp, J, K
! --- Read in the data
      PRINT *, 'Type ten integers each on a new line'
      DO J = 1, 10
            READ *, nums(J)
      END DO
! --- Sort the numbers into ascending order of magnitude
      .   .   .
! --- Write out the sorted list
      DO J = 1, 10
            PRINT *, 'Rank ', J, ' Value is ', nums(J)
      END DO
END PROGRAM sort10
```

Selection Sort (2)

```
! --- Sort the numbers into ascending order of magnitude
L1:    DO J = 1, 9
L2:          DO K = J+1, 10
                  IF(nums(J) > nums(K)) THEN
                        temp = nums(K)
                        nums(K) = nums(J)
                        nums(J) = temp
                  END IF
            END DO L2
      END DO L1
```

Valid Array Bounds

The bounds can be any constant expressions
There are two ways to use run–time bounds

- ALLOCATABLE arrays – see later
- When allocating them in procedures

We will discuss the following under procedures

```
SUBROUTINE workspace (size)
INTEGER :: size
REAL, DIMENSION(1:size*(size+1)) :: array
    . . .
```

Using Arrays as Objects (1)

Arrays can be handled as compound objects
Sections allow access as groups of elements
There are a large number of intrinsic procedures

Simple use handles all elements ‘‘in parallel’’

- Scalar values are expanded as needed

Set all elements of an array to a single value

```
INTEGER, DIMENSION(1:50) :: mark
mark = 0
```

Using Arrays as Objects (2)

You can use whole arrays as simple variables
Provided that they are all conformable

```
REAL, DIMENSION(1:200) :: arr1, arr2
. . .
arr1 = arr2+1.23*exp(arr1/4.56)
```

- I really do mean "as simple variables"

The RHS and any LHS indices are evaluated
And then the RHS is assigned to the LHS

Array Sections

Array sections create an aliased subarray
It is a simple variable with a value

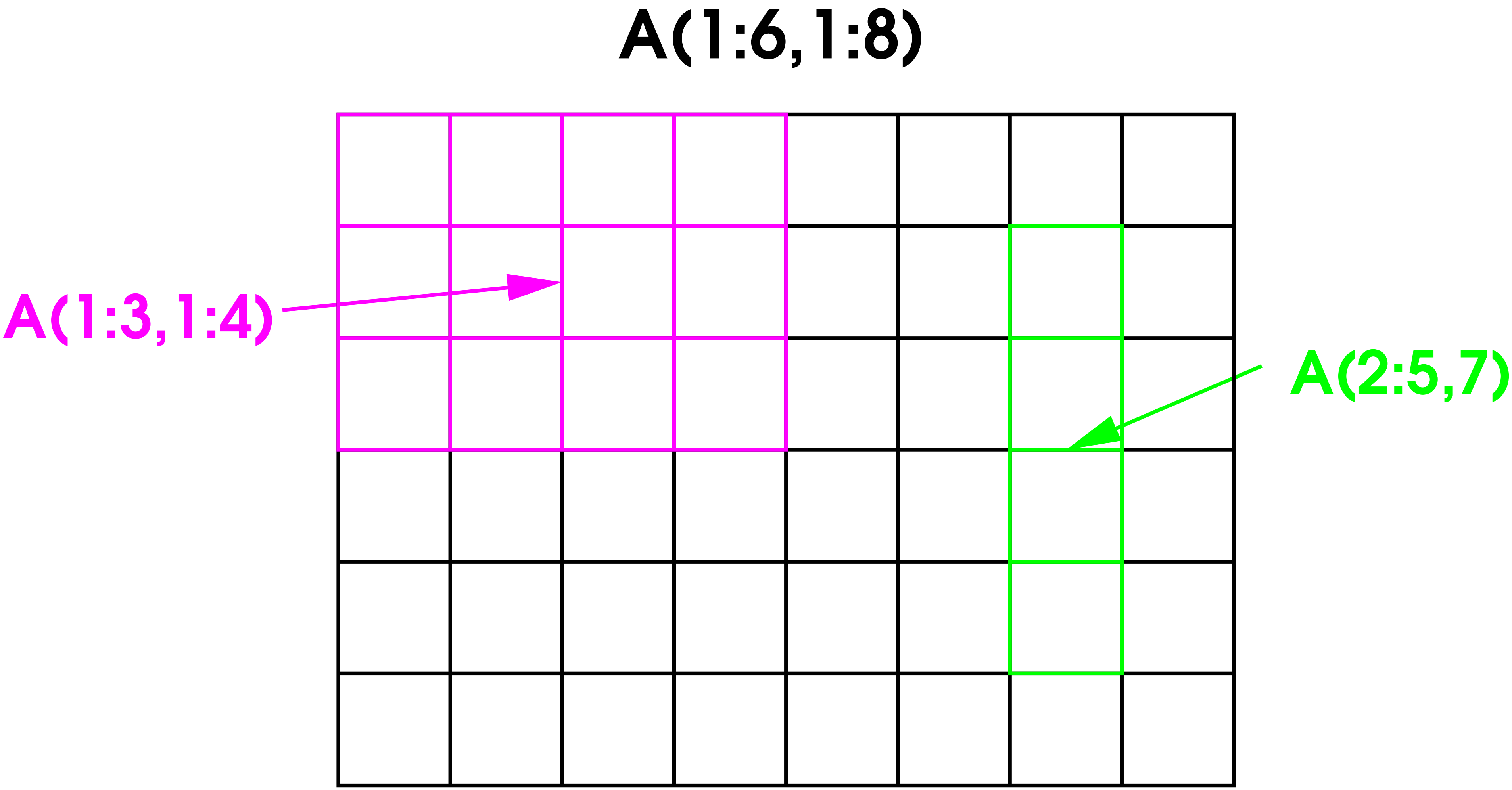
```
INTEGER :: arr1(1:100), arr2(1:50), arr3(1:100)

arr1(1:63) = 5 ;    arr1(64:100) = 7
arr2 = arr1(1:50)+arr3(51:100)
```

- Even this is legal, but forces a copy

```
arr1(26:75) = arr1(1:50)+arr1(51:100)
```

Array Sections



Short Form

Existing array bounds may be omitted
Especially useful for multidimensional arrays

If we have `REAL, DIMENSION(1:6, 1:8) :: A`
`A(3:, :4)` is the same as `A(3:6, 1:4)`
`A`, `A(:, :)` and `A(1:6, 1:8)` are all the same

`A(6, :)` is the 6th row as a 1–D vector
`A(:, 3)` is the 3rd column as a 1–D vector
`A(6:6, :)` is the 6th row as a 1×8 matrix
`A(:, 3:3)` is the 3rd column as a 6×1 matrix

Conformability of Sections

The conformability rule applies to sections, too

```
REAL :: A(1:6, 1:8), B(0:3, −5:5), C(0:10)

A(2:5, 1:7) = B(:, −3:3)    ! both have shape (4, 7)
A(4, 2:5) = B(:, 0) + C(7:)    ! all have shape (4)
C(:) = B(2, :)    ! both have shape (11)
```

But these would be illegal

```
A(1:5, 1:7) = B(:, −3:3)    ! shapes (5,7) and (4,7)
A(1:1, 1:3) = B(1, 1:3)    ! shapes (1,3) and (3)
```

Sections with Strides

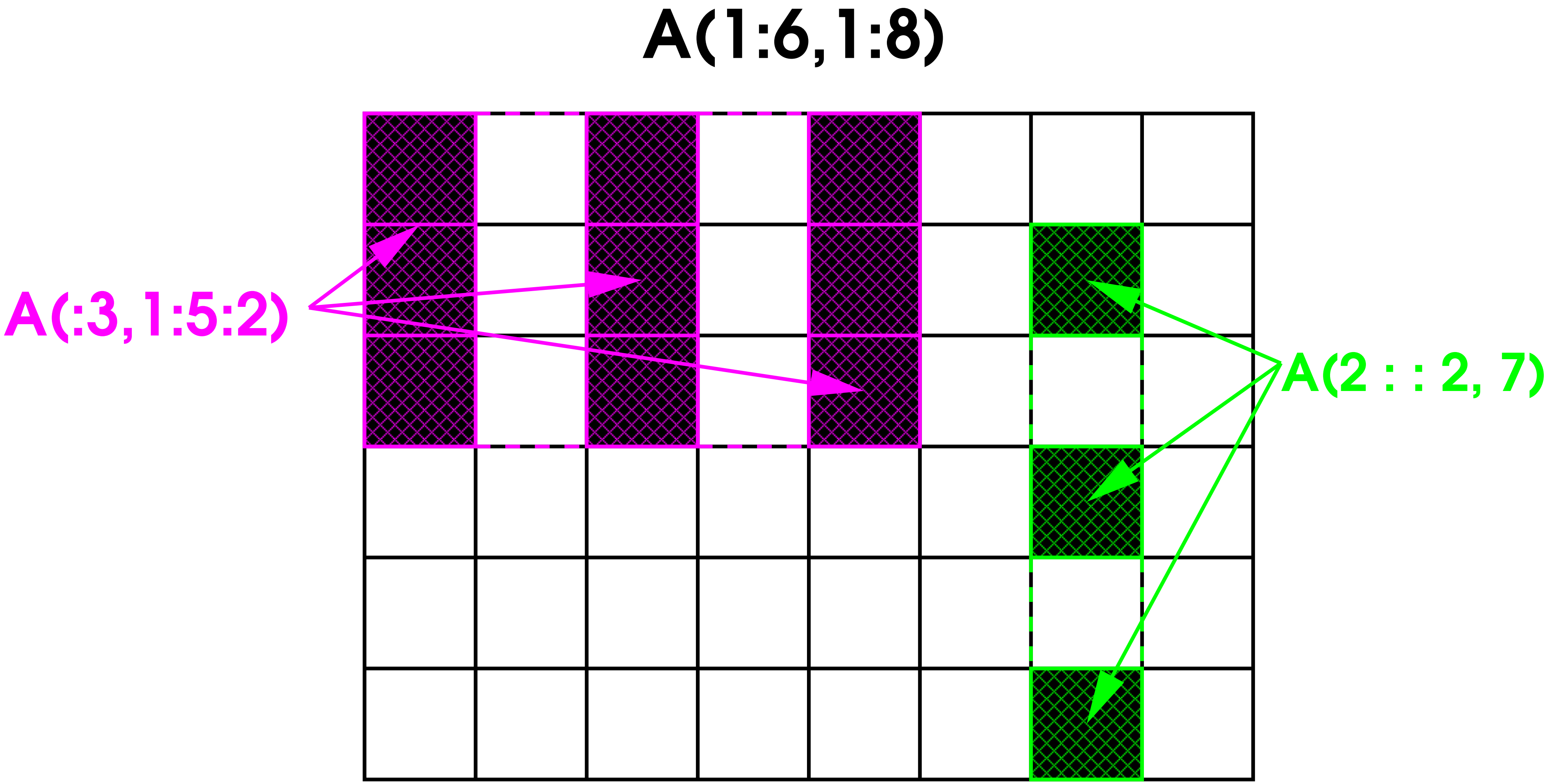
Array sections need not be contiguous
Any uniform progression is allowed

This is exactly like a more compact DO–loop
Negative strides are allowed, too

```
INTEGER :: arr1(1:100), arr2(1:50), arr3(1:50)
arr1(1:100:2) = arr2    ! Sets every odd element
arr1(100:1:-2) = arr3   ! Even elements, reversed

arr1 = arr1(100:1:-1)   ! Reverses the order of arr1
```

Strided Sections



Array Bounds

Subscripts/sections must be within bounds
The following are invalid (undefined behaviour)

```
REAL :: A(1:6, 1:8), B(0:3, -5:5), C(0:10)
A(2:5, 1:7) = B(:, -6:3)
A(7, 2:5) = B(:, 0)
C(:11) = B(2, :)
```

NAG will usually check; most others won't
Errors lead to overwriting etc. and CHAOS
Even NAG may not check all old-style Fortran

Elemental Operations

We have seen operations and intrinsic functions
Most built–in operators/functions are elemental
They act element–by–element on arrays

```
REAL, DIMENSION(1:200) :: arr1, arr2, arr3
arr1 = arr2+1.23*exp(arr3/4.56)
```

Comparisons and logical operations, too

```
REAL, DIMENSION(1:200) :: arr1, arr2, arr3
LOGICAL, DIMENSION(1:200) :: flags
flags = (arr1 > exp(arr2) .OR. arr3 < 0.0)
```

Array Intrinsic Functions (1)

There are over 20 useful intrinsic procedures
They can save a lot of coding and debugging

```
SIZE(x [, n])       ! The size of x (an integer scalar)
SHAPE(x)            ! The shape of x (an integer vector)

LBOUND(x [, n])     ! The lower bound of x
UBOUND(x [, n])     ! The upper bound of x
```

If n is present, down that dimension only
And the result is is an integer scalar
Otherwise the result is is an integer vector

Array Intrinsic Functions (2)

```
MINVAL(x)            ! The minimum of all elements of x
MAXVAL(x)            ! The maximum of all elements of x
```

These return a scalar of the same type as x

```
MINLOC(x)            ! The indices of the minimum
MAXLOC(x)            ! The indices of the maximum
```

These return an integer vector, just like SHAPE

Array Intrinsic Functions (3)

```
SUM(x [, n])              ! The sum of all elements of x
PRODUCT(x [, n])          ! The product of all elements of x
```

If n is present, down that dimension only

```
TRANSPOSE(x)              ! The transposition of
DOT_PRODUCT(x, y)         ! The dot product of x and y
MATMUL(x, y)              ! 1– and 2–D matrix multiplication
```

Reminder

TRANSPOSE(X) means $X_{ij} \Rightarrow X_{ji}$
It must have two dimensions, but needn't be square

DOT_PRODUCT(X, Y) means $\sum_i X_i \, . \, Y_i \Rightarrow Z$
Two vectors, both of the same length and type

MATMUL(X, Y) means $\sum_k X_{ik} \, . \, Y_{kj} \Rightarrow Z_{ij}$
Second dimension of X must match the first of Y
The matrices need not be the same shape

Either of X or Y may be a vector in MATMUL

Array Intrinsic Functions (4)

These also have some features not mentioned
There are more (especially for reshaping)
There are ones for array masking (see later)

Look at the references for the details

Warning

It’s not specified how results are calculated
All of the following can be different:

- Calling the intrinsic function
- The obvious code on array elements
- The numerically best way to do it
- The fastest way to do it

All of them are calculate the same formula
But the result may be slightly different

- If this starts to matter, consult an expert

Array Element Order (1)

This is also called “storage order”

Traditional term is “column–major order”
But Fortran arrays are not laid out in columns!
Much clearer: “first index varies fastest”

```
REAL :: A(1:3, 1:4)
```

The elements of A are stored in the order

A(1,1), A(2,1), A(3,1), A(1,2), A(2,2), A(3,2), A(1,3),
A(2,3), A(3,3), A(1,4), A(2,4), A(3,4)

Array Element Order (2)

Opposite to C, Matlab, Mathematica etc.

You don’t often need to know the storage order
Three important cases where you do:

- I/O of arrays, especially unformatted
- Array constructors and array constants
- Optimisation (caching and locality)

There are more cases in old–style Fortran
Avoid that, and you need not learn them

Simple I/O of Arrays (1)

Arrays and sections can be included in I/O
These are expanded in array element order

```
REAL, DIMENSION(3, 2) :: oxo
READ *, oxo
```

This is exactly equivalent to:

```
REAL, DIMENSION(3, 2) :: oxo
READ *, oxo(1, 1), oxo(2, 1), oxo(3, 1), &
    oxo(1, 2), oxo(2, 2), oxo(3, 2)
```

Simple I/O of Arrays (2)

Array sections can also be used

```
REAL, DIMENSION(100) :: nums
READ *, nums(30:50)

REAL, DIMENSION(3, 3) :: oxo
READ *, oxo(:, 3), oxo(3:1:-1,1)
```

The last statement is equivalent to

```
READ *, oxo(1, 3), oxo(2, 3), oxo(3, 3),    &
     oxo(3, 1), oxo(2, 1), oxo(1, 1)
```

Array Constructors (1)

An array constructor creates a temporary array

- Commonly used for assigning array values

```
INTEGER :: marks(1:6)
marks = (/ 10, 25, 32, 54, 54, 60 /)
```

Constructs an array with elements

10, 25, 32, 54, 54, 60

And then copies that array into marks

A good compiler will optimise that!

Array Constructors (2)

- Variable expressions are OK in constructors

  (/ x, 2.0*y, SIN(t*w/3.0),... etc. /)

They can be used anywhere an array can be
Except where you might assign to them!

- All expressions must be the same type

This has been relaxed in Fortran 2003

Array Constructors (3)

Arrays can be used in the value list
They are flattened into array element order

Implied DO-loops (as in I/O) allow sequences

If n has the value 7

```
(/ 0.0, (k/10.0, k = 2, n), 1.0 /)
```

Is equivalent to:

```
(/ 0.0, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 1.0 /)
```

Constants and Initialisation (1)

Array constructors are very useful for this
All elements must be constant expressions
I.e. ones that can be evaluated at compile time

For rank one arrays, just use a constructor

```
REAL, PARAMETER :: a(1:3) = (/ 1.23, 4.56, 7.89 /)
REAL, PARAMETER  :: b(3) = exp( (/ 1.2, 3.4, 5.6 /) )
```

But NOT:

```
REAL, PARAMETER :: arr(1:3) =    &
        myfunc ( (/ 1.2, 3.4, 5.6 /) )
```

Constants and Initialisation (2)

Other types can be initialised in the same way

```
CHARACTER(LEN=4), DIMENSION(1:5) :: names = &
      (/ 'Fred', 'Joe', 'Bill', 'Bert', 'Alf' /)
```

Constant expressions are allowed

```
INTEGER, PARAMETER :: N = 3, M = 6, P = 12
INTEGER :: arr(1:3) = (/ N, (M/N), (P/N) /)
REAL :: arr(1:3) = (/ 1.0, exp(1.0), exp(2.0) /)
```

But NOT:

```
REAL :: arr(1:3) = (/ 1.0, myfunc(1.0), myfunc(2.0) /)
```

Multiple Dimensions

Constructors cannot be nested – e.g. NOT:

```
REAL, DIMENSION(3, 4) :: array = &
      (/ (/ 1.1, 2.1, 3.1 /), (/ 1.2, 2.2, 3.2 /), &
            (/ 1.3, 2.3, 3.3 /), (/ 1.4, 2.4, 3.4 /) /)
```

They construct only rank one arrays

- Construct higher ranks using RESHAPE

This is covered in the extra slides on arrays

Allocatable Arrays (1)

Arrays can be declared with an unknown shape
Attempting to use them in that state will fail

```
INTEGER, DIMENSION(:), ALLOCATABLE :: counts
REAL, DIMENSION(:, :, :), ALLOCATABLE :: values
```

They become defined when space is allocated

```
ALLOCATE (counts(1:1000000))
ALLOCATE (value(0:N, –5:5, M:2*N+1))
```

Allocatable Arrays (2)

Failure will terminate the program
You can trap most allocation failures

```
INTEGER :: istat
ALLOCATE (arr(0:100, −5:5, 7:14), STAT=istat)
IF (istat /= 0) THEN
    . . .
END IF
```

Arrays can be deallocated using

```
DEALLOCATE (nums)
```

There are more features in Fortran 2003

Example

```
    INTEGER, DIMENSION(:), ALLOCATABLE :: counts
    INTEGER :: size, code
! --- Ask the user how many counts he has
    PRINT *, 'Type in the number of counts'
    READ *, size
! --- Allocate memory for the array
    ALLOCATE (counts(1:size), STAT=code)
    IF (code /= 0) THEN
        . . .
    END IF
```

Allocation and Fortran 95

Fortran 95 constrained ALLOCATABLE objects
Cannot be arguments, results or in derived types
I.e. local to procedures or in modules only

Fortran 2003 allows them almost everywhere
Almost all compilers already include those features
You may come across POINTER in old code
It can usually be replace by ALLOCATABLE

Ask if you hit problems and want to check

Testing Allocation

Can test if an ALLOCATABLE object is allocated

The ALLOCATED function returns LOGICAL:

```
INTEGER, DIMENSION(:), ALLOCATABLE :: counts
. . .
IF (ALLOCATED(counts)) THEN
    . . .
```

Generally, that is needed for advanced use only

Allocatable CHARACTER

Remember CHARACTER is really a string
Not an array of single characters, but a bit like one

You can use a colon (:) for a length
Provided that the variable is allocatable

This makes a copy of the text on an input line
It is also a Fortran 2003 feature

```
CHARACTER(LEN=100) :: line
CHARACTER(LEN=:), ALLOCATABLE :: message
ALLOCATE (message, SOURCE=TRIM(line))
```

Reminder

The above is all many programmers need
There is a lot more, but skip it for now

At this point, let's see a real example
Cholesky decomposition following LAPACK
With all error checking omitted, for clarity

It isn't pretty, but it is like the mathematics

- And that really helps to reduce errors

E.g. coding up a published algorithm

Cholesky Decomposition

To solve $A = LL^T$, in tensor notation:

$$L_{jj} = \sqrt{A_{jj} - \sum_{k=1}^{j-1} L_{jk}^2}$$

$$\forall_{i>j}, \; L_{ij} = (A_{ij} - \sum_{k=1}^{j-1} L_{ik} L_{jk}) / L_{jj}$$

Most of the Web uses i and j the other way round

Cholesky Decomposition

```
SUBROUTINE CHOLESKY ( A )
  IMPLICIT NONE
  INTEGER :: J, N
  REAL :: A (:, :)
  N = UBOUND (A, 1)
  DO J = 1, N
    A(J, J) = SQRT ( A(J, J) – &
      DOT_PRODUCT ( A(J, :J–1), A(J, :J–1) ) )
    IF (J < N) &
      A(J+1:, J) = ( A(J+1:, J) – &
        MATMUL ( A(J+1:, :J–1), A(J, :J–1) ) ) / A(J, J)
  END DO
END SUBROUTINE CHOLESKY
```

Other Important Features

These have been omitted for simplicity
There are extra slides giving an overview

- Constructing higher rank array constants
- Using integer vectors as indices
- Masked assignment and WHERE
- Memory locality and performance
- Avoiding unnecessary array copying

Introduction to Modern Fortran

Procedures

Nick Maclaren

nmm1@cam.ac.uk

March 2014

Sub-Dividing The Problem

- Most programs are thousands of lines

Few people can grasp all the details

- You often use similar code in several places

- You often want to test parts of the code

- Designs often break up naturally into steps

Hence, all sane programmers use procedures

What Fortran Provides

There must be a single main program
There are subroutines and functions
All are collectively called procedures

A subroutine is some out–of–line code
There are very few restrictions on what it can do
It is always called exactly where it is coded

A function's purpose is to return a result
There are some restrictions on what it can do
It is called only when its result is needed

Example – Cholesky (1)

We saw this when considering arrays
It is a very typical, simple subroutine

```
SUBROUTINE CHOLESKY (A)
    IMPLICIT NONE
    INTEGER :: J, N
    REAL :: A(:, :), X
    N = UBOUND(A, 1)
    DO J = 1, N
        . . .
    END DO
END SUBROUTINE CHOLESKY
```

Example – Cholesky (2)

And this is how it is called

```
PROGRAM MAIN
    IMPLICIT NONE
    REAL, DIMENSION(5, 5) :: A = 0.0
    REAL, DIMENSION(5) :: Z
        . . .
    CALL CHOLESKY (A)
        . . .
END PROGRAM MAIN
```

We shall see how to declare it later

Example – Variance

```
FUNCTION Variance (Array)
    IMPLICIT NONE
    REAL :: Variance, X
    REAL, INTENT(IN), DIMENSION(:) :: Array
    X = SUM(Array)/SIZE(Array)
    Variance = SUM((Array-X)**2)/SIZE(Array)
END FUNCTION Variance

    REAL, DIMENSION(1000) :: data
        . . .
    Z = Variance(data)
```

We shall see how to declare it later

Example – Sorting (1)

This was the harness of the selection sort
Replace the actual sorting code by a call

```
PROGRAM sort10
    IMPLICIT NONE
    INTEGER, DIMENSION(1:10) :: nums
    . . .
! --- Sort the numbers into ascending order of magnitude
    CALL SORTIT (nums)
! --- Write out the sorted list
    . . .
END PROGRAM sort10
```

Example – Sorting (2)

```
SUBROUTINE SORTIT (array)
    IMPLICIT NONE
    INTEGER :: temp, array(:), J, K
L1:   DO J = 1, UBOUND(array,1)-1
L2:       DO K = J+1, UBOUND(array,1)
             IF(array(J) > array(K)) THEN
                temp = array(K)
                array(K) = array(J)
                array(J) = temp
             END IF
          END DO L2
      END DO L1
END SUBROUTINE SORTIT
```

CALL Statement

The CALL statement evaluates its arguments
The following is an over–simplified description

- Variables and array sections define memory
- Expressions are stored in a hidden variable

It then transfers control to the subroutine
Passing the locations of the actual arguments

Upon return, the next statement is executed

SUBROUTINE Statement

Declares the procedure and its arguments
These are called dummy arguments in Fortran

The subroutine's interface is defined by:

- The SUBROUTINE statement itself
- The declarations of its dummy arguments
- And anything that those use (see later)

```
SUBROUTINE SORTIT (array)
INTEGER :: [ temp, ] array(:) [ , J, K ]
```

Subroutines With No Arguments

You aren't required to have any arguments
You can omit the parentheses if you prefer
Preferably either do or don't, but you can mix uses

```
SUBROUTINE Joe ( )

SUBROUTINE Joe


CALL Joe ( )

CALL Joe
```

Statement Order

A SUBROUTINE statement starts a subroutine
Any USE statements must come next
Then IMPLICIT NONE
Then the rest of the declarations
Then the executable statements
It ends at an END SUBROUTINE statement

PROGRAM and FUNCTION are similar

There are other rules, but you may ignore them

Dummy Arguments

- Their names exist only in the procedure

They are declared much like local variables

Any actual argument names are irrelevant

Or any other names outside the procedure

- The dummy arguments are associated with the actual arguments

Think of association as a bit like aliasing

Argument Matching

Dummy and actual argument lists must match
The number of arguments must be the same
Each argument must match in type and rank

That can be relaxed in several ways
See under advanced use of procedures

We shall come back to array arguments shortly
Most of the complexities involve them
This is for compatibility with old standards

Functions

Often the required result is a single value
It is easier to write a FUNCTION procedure

E.g. to find the largest of three values:

- Find the largest of the first and second
- Find the largest of that and the third

Yes, I know that the MAX function does this!

The function name defines a local variable

- Its value on return is the result returned

The RETURN statement does not take a value

Example (1)

```
FUNCTION largest_of (first, second, third)
    IMPLICIT NONE
    INTEGER :: largest_of
    INTEGER :: first, second, third
    IF (first > second) THEN
        largest_of = first
    ELSE
        largest_of = second
    END IF
    IF (third > largest_of) largest_of = third
END FUNCTION largest_of
```

Example (2)

```
INTEGER :: trial1, trial2 ,trial3, total, count
total = 0 ;    count = 0
DO
    PRINT *, 'Type three trial values:'
    READ *, trial1, trial2, trial3
    IF (MIN(trial1, trial2, trial3) < 0) EXIT
        count = count + 1
        total = total + &
          largest_of(trial1, trial2, trial3)
END DO
PRINT *, 'Number of trial sets = ', count, &
    ' Total of best of 3 = ',total
```

Functions With No Arguments

You aren't required to have any arguments
You must not omit the parentheses

```
FUNCTION Fred ( )
  INTEGER :: Fred

X = 1.23 * Fred ( )
CALL Alf ( Fred ( ) )
```

In the following, Fred is a procedure argument

```
CALL Alf ( Fred )
```

Internal Procedures (1)

Procedures can contain internal procedures
These can be SUBROUTINEs and FUNCTIONs
The statement order is as follows:

PROGRAM, SUBROUTINE or FUNCTION
    All of the code of the actual procedure
CONTAINS
    Any number of internal procedures
END PROGRAM, SUBROUTINE or FUNCTION

- Internal procedures may not themselves contain internal procedures

Internal Procedures (2)

- Warning: that order takes some getting used to

The procedure can use the internal procedures
And one of them can call any other

Most useful for small, private auxiliary ones
You can include any number of internal procedures

- They are visible only in the outer procedure

Won't clash with the same name elsewhere

Internal Procedures (3)

```
PROGRAM main
    REAL, DIMENSION(10) :: vector
    PRINT *, 'Type 10 values'
    READ *, vector
    PRINT *, 'Variance = ', Variance(vector)
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 PROGRAM main
```

Name Inheritance (1)

Everything accessible in the enclosing procedure
can also be used in the internal procedure

This includes all of the local declarations
And anything imported by USE (covered later)

Internal procedures need only a few arguments
Just the things that vary between calls
Everything else can be used directly

Name Inheritance (2)

A local name takes precedence

```
PROGRAM main
    REAL :: temp = 1.23
    CALL pete (4.56)
CONTAINS
    SUBROUTINE pete (temp)
        PRINT *, temp
    END SUBROUTINE pete
END PROGRAM main
```

Will print 4.56, not 1.23
Avoid doing this – it's very confusing

Using Procedures

Use this technique for solving test problems

- It is one of the best techniques for real code

There is another, equally good one, under modules

And there are yet others that you may need to use

INTENT (1)

You can make arguments read–only

```
SUBROUTINE Summarise (array, size)
    INTEGER, INTENT(IN) :: size
    REAL, DIMENSION(size) :: array
```

That will prevent you writing to it by accident
Or calling another procedure that does that
It may also help the compiler to optimise

- Strongly recommended for read–only args

INTENT (2)

You can also make them write–only
Less useful, but still very worthwhile

```
SUBROUTINE Init (array, value)
    IMPLICIT NONE
    REAL, DIMENSION(:), INTENT(OUT) :: array
    REAL, INTENT(IN) :: value
    array = value
END SUBROUTINE Init
```

As useful for optimisation as INTENT(IN)

INTENT (3)

The default is effectively INTENT(INOUT)

- But specifying INTENT(INOUT) is useful

It will trap the following nasty error

```
SUBROUTINE Munge (value)
    REAL, INTENT(INOUT) :: value
    value = 100.0*value
    PRINT *, value
END SUBROUTINE Munge

CALL Munge(1.23)
```

Example

```
SUBROUTINE expsum(n, k, x, sum)
    IMPLICIT NONE
    INTEGER, INTENT(IN) :: n
    REAL, INTENT(IN) :: k, x
    REAL, INTENT(OUT) :: sum
    INTEGER :: i
    sum = 0.0
    DO i = 1, n
        sum = sum + exp(-i*k*x)
    END DO
END SUBROUTINE expsum
```

Aliasing

Two arguments may overlap only if read–only
Also applies to arguments and global data

- If either is updated, weird things happen

Fortran doesn't have any way to trap that
Nor do any other current languages – sorry

Use of INTENT(IN) will stop it in many cases

- Be careful when using array arguments

Including using array elements as arguments

PURE Functions

You can declare a function to be PURE

All data arguments must specify INTENT(IN)
It must not modify any global data
It must not do I/O (except with internal files)
It must call only PURE procedures
Some restrictions on more advanced features

Generally overkill – but good practice
Most built–in procedures are PURE

Example

This is the cleanest way to define a function

```
PURE FUNCTION Variance (Array)
    IMPLICIT NONE
    REAL :: Variance, X
    REAL, INTENT(IN), DIMENSION(:) :: Array
    X = SUM(Array)/SIZE(Array)
    Variance = SUM((Array-X)**2)/SIZE(Array)
END FUNCTION Variance
```

Most safety, and best possible optimisation

ELEMENTAL Functions

Functions can be declared as ELEMENTAL
Like PURE, but arguments must be scalar

You can use them on arrays and in WHERE
They apply to each element, like built–in SIN

```
ELEMENTAL FUNCTION Scale (arg1, arg2)
    REAL, INTENT(IN) :: arg1, arg2
    Scale = arg1/sqrt(arg1**2+arg2**2)
END FUNCTION Scale

REAL, DIMENSION(100) :: arr1, arr2, array
array = Scale(arr1, arr2)
```

Keyword Arguments (1)

```
SUBROUTINE AXIS (X0, Y0, Length, Min, Max, Intervals)
     REAL, INTENT(IN)  :: X0, Y0, Length, Min, Max
     INTEGER, INTENT(IN) :: Intervals
 END SUBROUTINE AXIS

 CALL AXIS(0.0, 0.0, 100.0, 0.1, 1.0, 10)
```

- Error prone to write and unclear to read

And it can be a lot worse than that!

Keyword Arguments (2)

Dummy arg. names can be used as keywords
You don't have to remember their order

```
SUBROUTINE AXIS (X0, Y0, Length, Min, Max, Intervals)
        . . .

 CALL AXIS(Intervals=10, Length=100.0, &
      Min=0.1, Max=1.0, X0=0.0, Y0=0.0)
```

- The argument order now doesn't matter

The keywords identify the dummy arguments

Keyword Arguments (3)

Keywords arguments can follow positional
The following is allowed

```
SUBROUTINE AXIS (X0, Y0, Length, Min, Max, Intervals)
     . . .

CALL AXIS(0.0, 0.0, Intervals=10, Length=100.0, &
     Min=0.1, Max=1.0)
```

- Remember that the best code is the clearest

Use whichever convention feels most natural

Keyword Reminder

Keywords are not names in the calling procedure
They are used only to map to dummy arguments
The following works, but is somewhat confusing

```
SUBROUTINE Nuts (X, Y, Z)
    REAL, DIMENSION(:) :: X
    INTEGER :: Y, Z
END SUBROUTINE Nuts

INTEGER :: X
REAL, DIMENSION(100) :: Y, Z
CALL Nuts (Y=X, Z=1, X=Y)
```

Hiatus

That is most of the basics of procedures
Except for arrays and CHARACTER

Now might be a good time to do some examples
The first few questions cover the material so far

Assumed Shape Arrays (1)

- The best way to declare array arguments
You must declare procedures as above

- Specify all bounds as simply a colon (':')
The rank must match the actual argument
The lower bounds default to one (1)
The upper bounds are taken from the extents

```
REAL, DIMENSION(:) :: vector
REAL, DIMENSION(:, :) :: matrix
REAL, DIMENSION(:, :, :) :: tensor
```

Example

```
SUBROUTINE Peculiar (vector, matrix)
    REAL, DIMENSION(:), INTENT(INOUT) :: vector
     REAL, DIMENSION(:, :), INTENT(IN) :: matrix
     . . .
END SUBROUTINE Peculiar

REAL, DIMENSION(20:1000), :: one
REAL, DIMENSION(−5:100, −5:100) :: two
CALL Peculiar (one(101:160), two(21:, 26:75) )
```

vector will be DIMENSION(1:60)
matrix will be DIMENSION(1:80, 1:50)

Assumed Shape Arrays (2)

Query functions were described earlier

SIZE, SHAPE, LBOUND and UBOUND

So you can write completely generic procedures

```
SUBROUTINE Init (matrix, scale)
    REAL, DIMENSION(:, :), INTENT(OUT) :: matrix
    INTEGER, INTENT(IN) :: scale
    DO N = 1, UBOUND(matrix,2)
        DO M = 1, UBOUND(matrix,1)
            matrix(M, N) = scale*M + N
        END DO
    END DO
END SUBROUTINE Init
```

Cholesky Decomposition

```
SUBROUTINE CHOLESKY(A)
    IMPLICIT NONE
    INTEGER :: J, N
    REAL, INTENT(INOUT) :: A(:, :), X
    N = UBOUND(A, 1)
    IF (N < 1 .OR. UBOUND(A, 2) /= N)
        CALL Error("Invalid array passed to CHOLESKY")
    DO J = 1, N
            . . .
  END DO
END SUBROUTINE CHOLESKY
```

Now I have added appropriate checking

Setting Lower Bounds

Even when using assumed shape arrays
you can set any lower bounds you want

- You do that in the called procedure

```
SUBROUTINE Orrible (vector, matrix, n)
    REAL, DIMENSION(2*n+1:) :: vector
    REAL, DIMENSION(0:, 0:) :: matrix
    . . .
END SUBROUTINE Orrible
```

Warning

Argument overlap will not be detected
Not even for assumed shape arrays

- A common cause of obscure errors

No other language does much better

Explicit Array Bounds

In procedures, they are more flexible
Any reasonable integer expression is allowed

Essentially, you can use any ordinary formula
Using only constants and integer variables
Few programmers will ever hit the restrictions

The most common use is for workspace
But it applies to all array declarations

Automatic Arrays (1)

Local arrays with run–time bounds are called
automatic arrays

Bounds may be taken from an argument
Or a constant or variable in a module

```
SUBROUTINE aardvark (size)
USE sizemod    ! This defines worksize
INTEGER, INTENT(IN) :: size

REAL, DIMENSION(1:worksize) :: array_1
REAL, DIMENSION(1:size*(size+1)) :: array_2
```

Automatic Arrays (2)

Another very common use is a ‘shadow’ array
i.e. one the same shape as an argument

```
SUBROUTINE pard (matrix)
REAL, DIMENSION(:, :) :: matrix

REAL, DIMENSION(UBOUND(matrix, 1), &
        UBOUND(matrix, 2)) :: &
    matrix_2, matrix_3
```

And so on – automatic arrays are very flexible

Explicit Shape Array Args (1)

We cover these because of their importance
They were the only mechanism in Fortran 77
- But, generally, they should be avoided

In this form, all bounds are explicit
They are declared just like automatic arrays
The dummy should match the actual argument
Making an error will usually cause chaos

- Only the very simplest uses are covered

There are more details in the extra slides

Explicit Shape Array Args (2)

You can use constants

```
SUBROUTINE Orace (matrix, array)
    INTEGER, PARAMETER :: M = 5, N = 10
    REAL, DIMENSION(1:M, 1:N) :: matrix
    REAL, DIMENSION(1000) :: array
    . . .
END SUBROUTINE Orace

INTEGER, PARAMETER :: M = 5, N = 10
REAL, DIMENSION(1:M, 1:N) :: table
REAL, DIMENSION(1000) :: workspace
CALL Orace(table, workspace)
```

Explicit Shape Array Args (3)

It is common to pass the bounds as arguments

```
SUBROUTINE Weeble (matrix, m, n)
    INTEGER, INTENT(IN) :: m, n
    REAL, DIMENSION(1:m, 1:n) :: matrix
    . . .
END SUBROUTINE Weeble
```

You can use expressions, of course

- But it is not really recommended

Purely on the grounds of human confusion

Explicit Shape Array Args (4)

You can define the bounds in a module
Either as a constant or in a variable

```
SUBROUTINE Wobble (matrix)
    USE sizemod    ! This defines m and n
    REAL, DIMENSION(1:m, 1:n) :: matrix
    . . .
END SUBROUTINE Weeble
```

- The same remarks about expressions apply

Assumed Size Array Args

The last upper bound can be *

I.e. unknown, but assumed to be large enough

```
SUBROUTINE Weeble (matrix, n)
    REAL, DIMENSION(n, *) :: matrix
    . . .
END SUBROUTINE Weeble
```

- You will see this, but generally avoid it

It makes it very hard to locate bounds errors

It also implies several restrictions

Warnings

The size of the dummy array must not exceed
the size of the actual array argument

- Compilers will rarely detect this error

There are also some performance problems when
passing assumed shape and array sections
to explicit shape or assumed size dummies

That is in the advanced slides on procedures
Sorry – but it's complicated to explain

Example (1)

We have a subroutine with an interface like:

```
SUBROUTINE Normalise (array, size)
INTEGER, INTENT(IN) :: size
REAL, DIMENSION(size) :: array
```

The following calls are correct:

```
REAL, DIMENSION(1:10) :: data

CALL Normalise (data, 10)
CALL Normalise (data(2:5), SIZE(data(2:5)))
CALL Normalise (data, 7)
```

Example (2)

```
SUBROUTINE Normalise (array, size)
INTEGER, INTENT(IN) :: size
REAL, DIMENSION(size) :: array
```

The following calls are not correct:

```
INTEGER, DIMENSION(1:10) :: indices
REAL :: var, data(10)

CALL Normalise (indices, 10)    ! wrong base type
CALL Normalise (var, 1)    ! not an array
CALL Normalise (data, 10.0)    ! wrong type
CALL Normalise (data, 20)    ! dummy array too big
```

Character Arguments

Few scientists do anything very fancy with these
See the advanced foils for anything like that

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
    . . .
END SUBROUTINE sorter

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)
 comp: DO I = 1, (N-1)/2
        IF (word(I:I) /= word(N+1-I:N+1-I)) THEN
            RETURN
        END IF
    END DO comp
    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
    END DO
END SUBROUTINE reverse_word
```

Character Workspace

The rules are very similar to those for arrays
The length can be an almost arbitrary expression
But it usually just shadows an argument

```
SUBROUTINE sort_words (words)
    CHARACTER(LEN=*) :: words(:)
    CHARACTER(LEN=LEN(words)) :: temp
    . . .
END SUBROUTINE sort_words
```

Character Valued Functions

Functions can return CHARACTER values
Fixed–length ones are the simplest

```
FUNCTION truth (value)
    IMPLICIT NONE
    CHARACTER(LEN=8) :: truth
    LOGICAL, INTENT(IN) :: value
    IF (value) THEN
        truth = '.True.'
    ELSE
        truth = '.False.'
    END IF
END FUNCTION truth
```

Example

```
SUBROUTINE diagnose (message, value)
    CHARACTER(LEN=*), INTENT(IN) :: message
    REAL :: value
    PRINT *, message, value
END SUBROUTINE diagnose


CALL diagnose("Horrible failure",determinant)
```

Static Data

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

- You do this with the SAVE attribute

Initialised variables get that automatically
It is good practice to specify it anyway

The best style avoids most such use
It can cause trouble with parallel programming
But it works, and lots of programs rely on it

Example

This is a futile example, but shows the feature

```
SUBROUTINE Factorial (result)
    IMPLICIT NONE
    REAL, INTENT(OUT) :: result
    REAL, SAVE :: mult = 1.0, value = 1.0
    mult = mult+1.0
    value = value*mult
    result = value
END SUBROUTINE Factorial
```

Warning

Omitting SAVE will usually appear to work
But even a new compiler version may break it
As will increasing the level of optimisation

- Decide which variables need it during design

- Always use SAVE if you want it
And preferably never when you don't!

- Never assume it without specifying it

Warning for C/C++ Users

Initialisation without SAVE initialises once
It does NOT reinitialise each time it is called

- It can't be done using Fortran initialisation

Do it using an explicit assignment statement

Delayed Until Modules

Sometimes you need to share global data
It's trivial, and can be done very cleanly

Procedures can be passed as arguments
This is a very important facility for some people
For historical reasons, this is a bit messy

- However, internal procedures can't be

They can be in Fortran 2008 – i.e. shortly

We will cover both of these under modules
It just happens to be simplest that way!

Other Features

There is a lot that we haven't covered
We will return to some of it later

- The above covers the absolute basics
Plus some other features you need to know

- Be a bit cautious when using other features
Some have been omitted because of "gotchas"

- And I have over–simplified a few areas

Extra Slides

Topics in the advanced slides on procedures

- Argument association and updating
- The semantics of function calls
- Optional arguments
- Array– and character–valued functions
- Mixing explicit and assumed shape arrays
- Array arguments and sequence association
- Miscellaneous other points

Omissions

Rather a lot has been omitted here, unfortunately
It's there in the notes, if you are interested

If you think that Fortran can't do it, look deeper
Sorry about that, but this had to be simplified

Introduction to Modern Fortran

KIND, Precision and COMPLEX

Nick Maclaren

nmm1@cam.ac.uk

March 2014

The Basic Problem

REAL must be same size as INTEGER
This is for historical reasons – ask if you care

32 bits allows integers of up to 2147483647
Usually plenty for individual array indices

But floating–point precision is only 6 digits
And its range is only $10^{-38} - 10^{+38}$

Index values are not exact in floating–point
And there are many, serious numerical problems

Example

```
REAL, DIMENSION(20000000) :: A
REAL :: X
X = SIZE(A)-1
PRINT *, X
```

Prints 20000000.0 – which is not right
That code needs only 80 MB to go wrong

See “How Computers Handle Numbers”
Mainly on the numerical aspects

Ordinary REAL Constants

These will often do what you expect

- But they will very often lose precision

```
0.0, 7.0, 0.25, 1.23, 1.23E12,
0.1, 1.0E−1, 3.141592653589793
```

Only the first three will do what you expect

- In old Fortran constructs, can cause chaos

E.g. as arguments to external libraries

KIND Values

You can get the KIND of any expression

KIND(var) is the KIND value of var
KIND(0.0) is the KIND value of REAL
KIND(0.0D0) is that of DOUBLE PRECISION
This is described in a moment

Implementation–dependent integer values
selecting the type (e.g. a specific REAL)

- Don't use integer constants directly

SELECTED_REAL_KIND

You can request a minimum precision and range
Both are specified in decimal

SELECTED_REAL_KIND (Prec [, Range])

This gives at least Prec decimal places
and range $10^{-Range} - 10^{+Range}$

E.g. SELECTED_REAL_KIND(12)
at least 12 decimal places

Warning: Time Warp

Unfortunately, we need to define a module
We shall cover those quite a lot later

The one we shall define is trivial
Just use it, and don't worry about the details
Everything you need to know will be explained

Just compile it, but don't link it, using –c

    nagfor –C=all –c double.f90

Using KIND (1)

You should write and compile a module

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

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, 1.23_DP, 1.23E12_DP,
0.1_DP, 1.0E–1_DP, 3.141592653589793_DP
```

That's really all you need to know . . .

Using KIND (3)

Note that the above makes it trivial to change
ALL you need is to change the module

```
MODULE double
    INTEGER, PARAMETER :: DP = &
        SELECTED_REAL_KIND(15, 300)
END MODULE double
```

(15, 300) requires IEEE 754 double or better

Or even: SELECTED_REAL_KIND(25, 1000)

DOUBLE PRECISION (1)

- The best way to control precision

Most flexible, portable and future–proof
Advisable if you may want to use HECToR

All older (Fortran 77) code will do it differently
And quite a lot of programmers still do
The old method is fairly reliable, today

- You need to know about this, but avoid it

DOUBLE PRECISION (2)

DOUBLE PRECISION takes the space of 2 REALs
⇒ It need not be any more accurate, though

- Almost always, REAL is 32–bit IEEE 754

And DOUBLE PRECISION is 64–bit IEEE 754
Precision is 15 digits, range is $10^{-300} - 10^{+300}$

Main exception is Cray vector supercomputers
And when using compiler options to change precision

DOUBLE PRECISION (3)

You can use it just like REAL in declarations
Using KIND is more modern and compact

```
REAL(KIND=KIND(0.0D0)) :: a, b, c
```

Constants use D for the exponent – 1.23D12 or 0.0D0

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

```
0.0D0, 7.0D0, 0.25D0, 1.23D0, 1.23D12,
0.1D0, 1.0D–1, 3.14159265358979D0
```

Intrinsic Procedures

Almost all intrinsics ‘just work’ (i.e. are generic)
IMPLICIT NONE removes most common traps

- Avoid specific (old) names for procedures
AMAX0, DMIN1, DSQRT, FLOAT, IFIX etc.

- DPROD is also not generic – use a library

- 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
```

- Most of the Web and many books are wrong

A Fortran IV feature, NOT a standard one
'8' is NOT always the size in bytes

- I strongly recommend converting to KIND

Old Fortran Libraries

Be very careful with external libraries

- Make sure argument types are right

Automatic conversion does not happen
Not will you get a diagnostic (in general)

Any procedure with no explicit interface
I did say that using old Fortran was more painful

INTEGER KIND

You can choose different sizes of integer

```
INTEGER, PARAMETER :: big = &
        SELECTED_INT_KIND(12)
INTEGER(KIND=big) :: bignum
```

bignum can hold values of up to at least 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
Useful for handling non–ASCII character sets

It is not covered in this course
Very few scientists want or use it

Complex Arithmetic

Fortran is the answer – what was the question?

Has always been supported, and well integrated

COMPLEX is a (real, imaginary) pair of REAL
It uses the same KIND as underlying reals

```
COMPLEX(KIND=DP) :: c
c = (1.23_DP,4.56_DP)
```

Full range of operations, intrinsic functions etc.

Example

```
COMPLEX(KIND=DP) :: c, d, e, f

c = (1.23_DP,4.56_DP)*CONJG(d)+SIN(f*g)
e = EXP(d+c/f)*ABS(LOG(e))
```

The functions are the complex forms
E.g. ABS is $\sqrt{re^2 + im^2}$
CONJG is complex conjugate, of course

Using COMPLEX really IS that simple!

Worst "Gotcha"

- Must specify KIND in conversion function

```
c = CMPLX(<X−expr>, KIND=DP)
c = CMPLX(<X−expr>, <Y−expr>, KIND=DP)
```

This will not work – KIND is default REAL
Usually with no warning from the compiler

```
c = CMPLX(0.1_DP,0.2_DP)
```

Conversion to REAL

```
REAL(KIND=DP) :: x
COMPLEX(KIND=DP) :: c
  . . . lots of statements . . .
x = x+c
c = 2.0_DP*x
```

Loses the imaginary part, without warning
Almost all modern languages do the same

A Warning for Old Code

```
C = DCMPLX(0.1_DP, 0.1_DP)
```

That is often seen in Fortran IV legacy code
It doesn't work in standard (modern) Fortran

- It will be caught by IMPLICIT NONE

Complex I/O

The form of I/O we have used is list–directed
COMPLEX does what you would expect

```
COMPLEX(KIND=DP) :: c = (1.23_DP,4.56_DP)
WRITE (*, *) C
```

Prints "(1.23,4.56)"
And similarly for input

There is some more on COMPLEX I/O later

Exceptions

Complex exceptions are mathematically hard

- Overflow often does what you won't expect

Fortran, unfortunately, is no exception to this

See "How Computers Handle Numbers"

- Don't cause them in the first place

- Use the techniques described to detect them

Introduction to Modern Fortran

Modules and Interfaces

Nick Maclaren

nmm1@cam.ac.uk

March 2014

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 Interactions

Modules can USE other modules
Dependency graph shows visibility / usage

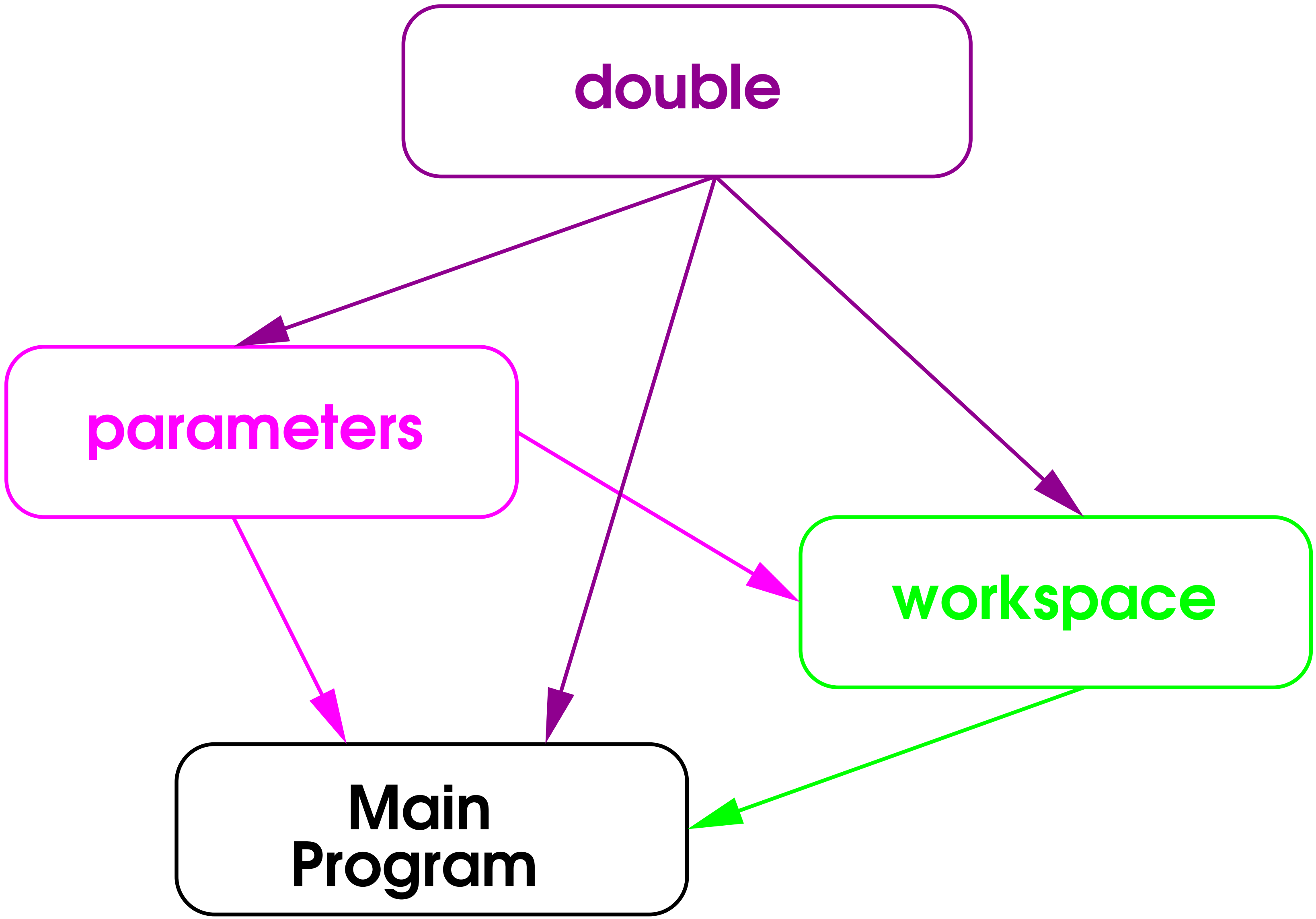
- Modules may not depend on themselves

Languages that allow that are very confusing

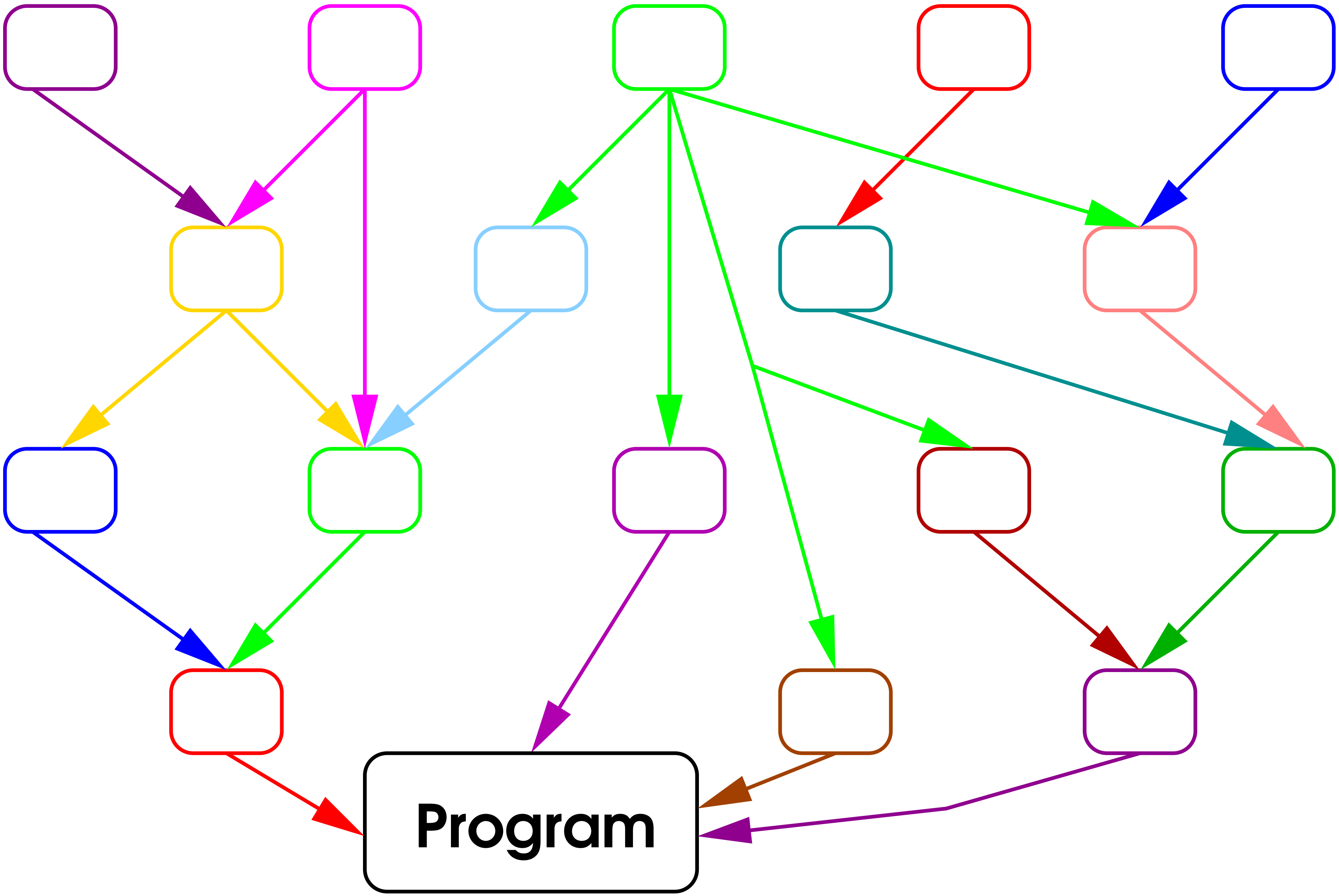
Can do anything you are likely to get to work

- If you need to do more, ask for advice

Module Dependencies



Module Dependencies



Module Structure

```
MODULE <name>
      Static (often exported) data definitions
CONTAINS
      Procedure definitions (i.e. their code)
END MODULE <name>
```

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

- For simplest use, keep them 1≡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

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

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
```

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 :: 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

Module Procedures (1)

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)
```

Module Procedures (2)

- Modules can contain any number of procedures
- You can use any number of modules

```
PROGRAM main
    USE mymod
    REAL, DIMENSION(10) :: array
    PRINT *, 'Type 10 values'
    READ *, array
    PRINT *, 'Variance = ', Variance(array)
END PROGRAM main
```

Using Procedures

Internal procedures or module procedures?
Use either technique for solving test problems

- They are the best techniques for real code
Simplest, and give full access to functionality
We will cover some other ones later

- Note that, if a procedure is in a module
it may still have internal procedures

Example

```
MODULE mymod
CONTAINS
    SUBROUTINE Sorter (array, opts)
        . . .
    CONTAINS
        FUNCTION Compare (value1, value2, flags)
            . . .
        END FUNCTION Compare
        SUBROUTINE Swap (loc1, loc2)
            . . .
        END FUNCTION Swap
    END SUBROUTINE Sorter
END MODULE mymod
```

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
Unfortunately, doing that has had to be omitted

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

Derived Type Definitions

We shall cover these later:

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

USE Bicycle
TYPE(Wheel) :: w1
```

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

Will describe the process in more detail later

Using Compiled Modules

All the program needs is the USE statements

- 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

```
nagfor -C=all -o main main.f90 mymod.o

nagfor -C=all -o main main.f90  \
     mod_a.o mod_b.o mod_c.o
```

Take a Breather

That is most of the basics of modules
Except for interfaces and access control

The first question covers the material so far

The remainder is important and useful
But it is unfortunately rather more complicated

What Are Interfaces?

The FUNCTION or SUBROUTINE statement
And everything directly connected to that
USE if needed for argument declarations

- And don't forget a function result 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

Interface Blocks

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

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

INTERFACE
    SUBROUTINE CHOLESKY (A)
        USE double
        REAL(KIND=dp) :: A(:, :)
    END SUBROUTINE CHOLESKY
END INTERFACE
```

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!

More on this in the Make and Linking lecture

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
```

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 doesn't help
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 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
```

Interfaces and Access Control

These are things that have been omitted
They're there in the notes, if you are interested

They are extremely important for large programs
But time is too tight to teach them now

- Do only the first practical and skip the rest

Introduction to Modern Fortran

Derived Types

Nick Maclaren

nmm1@cam.ac.uk

March 2014

Summary

There is one important new feature to cover

It is not complicated, as we shall do it

- But we won't cover it in great depth

Doing it fully would be a course in itself
The same applies in other languages, too

What Are Derived Types?

As usual, a hybrid of two, unrelated concepts
C++, Python etc. are very similar

- One is structures – i.e. composite objects
Arbitrary types, statically indexed by name

- The other is user–defined types
Often called semantic extension
This is where object orientation comes in

- This course will describe only the former

Why Am I Wimping Out?

Fortran 2003 has really changed this

- full object orientation
- semantic extension
- polymorphism (abstract types)
- and lots more

The course was already getting too big
And, yes, I was getting sick of writing it!

This area justifies a separate course
About one day or two afternoons, not three days
Please ask if you would like it written

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
```

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 ...

Fortran 95 Restriction

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

Fortran 2003 removed that restriction
You may come across POINTER in old code
It can usually be replace by ALLOCATABLE

Ask if you hit problems and want to check

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 value 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 I am omitting

Assignment (2)

Each derived type is a separate type
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.)
```

Or use keywords for components (Fortran 2003)

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

Default Initialisation

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 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 derived types]

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

  1.2300000   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

Both useful, even without semantic extension

Wholly Private Types

```
MODULE Marsupial
    TYPE, PRIVATE :: Wombat
        REAL :: weight, 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)

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

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

Hidden Components (2)

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

Trees

E.g. 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
```

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

Opaque Types etc.

This is another using aspect that has been omitted
It’s there in the notes, if you are interested

- Skip the practical that needs that facility

Introduction to Modern Fortran

I/O and Files

Nick Maclaren

nmm1@cam.ac.uk

March 2014

I/O Generally

Most descriptions of I/O are only half–truths
Those work most of the time – until they blow up
Most modern language standards are like that

Fortran is rather better, but there are downsides
Complexity and restrictions being two of them

- Fortran is much easier to use than it seems
- This is about what you can rely on in practice

We will start with the basic principles

Some ‘Recent’ History

Fortran I/O (1950s) predates even mainframes
OPEN and filenames was a CC† of c. 1975

Unix/C spread through CS depts 1975–1985
ISO C’s I/O model was a CC† of 1985–1988
Modern languages use the C/POSIX I/O model
Even Microsoft systems are like Unix here

- The I/O models have little in common

† CC = committee compromise

Important Warning

It is often better than C/C++ and often worse
But it is very different at all levels

- It is critical not to think in C–like terms

Trivial C/C++ tasks may be infeasible in Fortran

As always, use the simplest code that works
Few people have much trouble if they do that

- Ask for help with any problems here

Fortran's Sequential I/O Model

A Unix file is a sequence of characters (bytes)
A Fortran file is a sequence of records (lines)

For simple, text use, these are almost equivalent

In both Fortran and C/Unix:

- Keep text records short (say, $<$ 250 chars)
- Use only printing characters and space
- Terminate all lines with a plain newline
- Trailing spaces can appear and disappear

What We Have Used So Far

To remind you what you have been doing so far:

PRINT *, can be written WRITE (*,*)

READ *, can be written READ (*,*)

READ/WRITE (*,*) is shorthand for
    READ/WRITE (UNIT=*, FMT=*)

READ *, ... and PRINT *, ... are legacies
Their syntax is historical and exceptional

Record-based I/O

- Each READ and WRITE uses 1+ records

Any unread characters are skipped for READ
WRITE ends by writing an end–of–line indicator

- Think in terms of units of whole lines

A WRITE builds one or more whole lines
A READ consumes one or more whole lines

Fortran 2003 relaxes this, to some extent

Fortran's I/O Primitives

All Fortran I/O is done with special statements
Any I/O procedure is a compiler extension

Except as above, all of these have the syntax:

<statement> (<control list>) <transfer list>

The <transfer list> is only for READ and WRITE

The <control list> items have the syntax:

<specifier>=<value>

Translation

An I/O statement is rather like a command

A <control list> is rather like a set of options
Though not all of the specifiers are optional

The <transfer list> is a list of variables to read
or a list of expressions to write

We now need to describe them in more detail

Specifier Values (1)

All specifier values can be expressions
If they return values, they must be variables

- Except for * in UNIT=* or FMT=*

Even lunatic code like this is permitted

```
INTEGER, DIMENSION(20) :: N
CHARACTER(LEN=50) :: C

WRITE (UNIT = (123*K)/56+2, FMT = C(3:7)//')',    &
      IOSTAT=N(J**5-15))
```

Specifier Values (2)

The examples will usually use explicit constants

```
OPEN (23, FILE='trace.out', RECL=250)
```

- But you are advised to parameterise units
And anything else that is system dependent
Or you might need to change later

```
INTEGER, PARAMETER :: tracing = 23, tracelen = 250
CHARACTER(LEN=*), PARAMETER ::    &
    tracefile = 'trace.out'

OPEN (tracing, FILE=tracefile, RECL=tracelen)
```

Basics of READ and WRITE

READ/WRITE (<control list>) <transfer list>

Control items have form <specifier> = <value>
UNIT is the only compulsory control item
The UNIT= can be omitted if the unit comes first

The unit is an integer identifying the connection
It can be an expression, and its value is used

UNIT=* is an exceptional syntax
It usually means stdin and stdout

Transfer Lists

A list is a comma–separated sequence of items
The list may be empty (READ and WRITE)

- A basic output item is an expression
- A basic input item is a variable

Arrays and array expressions are allowed

- They are expanded in array element order

Fancy expressions will often cause a copy
Array sections should not cause a copy

Example

```
REAL :: X(10)
READ *, X(:7)
PRINT *, X(4:9:3)*1000.0
```

```
1.23 2.34 3.45 4.56 5.67 6.78 7.89
```

Produces a result like:

```
   4.5600000E+03   7.8900000E+03
```

Empty Transfer Lists

These are allowed, defined and meaningful

READ (*, *) skips the next line

WRITE (*, *) prints a blank line

WRITE (*, FORMAT) prints any text in FORMAT

That may print several lines

A Useful Trick

A useful and fairly common construction

```
INTEGER :: NA
REAL, DIMENSION(1:100) :: A
READ *, NA, A(1:NA)
```

Fortran evaluates a transfer list as it executes it

- Be warned: easy to exceed array bounds

At least, you should check the length afterwards
Safer to put on separate lines and check first

Implied DO-loops

There is an alternative to array expressions
Equivalent, but older and often more convenient

Items may be (<list> , <indexed loop control>)
This repeats in the loop order (just like DO)

((A(I,J) , J = 1,3) , B(I), I = 6,2,−2)

A(6,1), A(6,2), A(6,3), B(6), A(4,1), A(4,2),
A(4,3), B(4), A(2,1), A(2,2), A(2,3), B(2)

Programming Notes

You can do I/O of arrays in three ways:

- You can write a DO–loop around the I/O
- Array expressions for selecting and ordering
- You can use implied DO–loops

Use whichever is most convenient and clearest
There are no problems with combining them
More examples of their use will be shown later

There isn't a general ranking of efficiency

The UNIT Specifier

- A unit is an integer value
Except for UNIT=*, described above
It identifies the connection to a file

- UNIT= can be omitted if the unit is first

A unit must be connected to a file before use
Generally use values in the range 10–99

- That's all you need to know for now

The FMT Specifier

This sets the type of I/O and must match the file

- FMT= can be omitted if the format is second and the first item is the unit

- FMT=* indicates list–directed I/O
- FMT=<format> indicates formatted I/O

These can be interleaved on formatted files

- No FMT specifier indicates unformatted I/O

Example

These are formatted I/O statements

```
WRITE (UNIT = *, FMT = '(2F5.2)') c
READ (99, '(F5.0)') x
WRITE (*, FMT = myformat) p, q, r
```

These are list–directed I/O statements

```
WRITE (UNIT = *, FMT = *) c
READ (99, *) x
```

These are unformatted I/O statements

```
WRITE (UNIT = 64) c
READ (99) x
```

List-Directed Output (1)

What you have been doing with ‘PRINT *,’

The transfer list is split into basic elements
Each element is then formatted appropriately
It is separated by spaces and/or a comma

- Except for adjacent CHARACTER items

Write spaces explicitly if you want them

The format and layout are compiler–dependent

Example

```
REAL :: z(3) = (/4.56, 4.56, 4.56/)
CHARACTER(LEN=1) :: c = 'a'
PRINT *, 1.23, 'Oh dear', z, c, '"', c, ' ', c, c
```

Produces (under one compiler):

```
1.2300000 Oh dear 4.5599999 4.5599999
4.5599999 a"a aa
```

List-Directed Output (2)

You can cause character strings to be quoted
Very useful if writing data for reinput

```
WRITE (11, *, DELIM='quote') 'Kilroy was here'

"Kilroy was here"
```

Also DELIM='apostrophe' and DELIM='none'
They can also be specified in OPEN
Apply to all WRITEs with no DELIM

List-Directed Input (1)

What you have been doing with ‘READ *,’

This does the reverse of ‘PRINT *,’
The closest Fortran comes to free–format input

- It automatically checks the data type

- OK for lists of numbers and similar

Not much good for genuinely free–format

List-Directed Input (2)

Strings may be quoted, or not
Using either quote (") or apostrophe (')

- Quote all strings containing the following:
  ,   /   "   '   *   space   end–of–line

For the reasons why, read the specification
List–directed input is actually quite powerful
But very unlike all other modern languages

Example

```
REAL :: a, b, c
CHARACTER(LEN=8) :: p, q
READ *, a, p, b, q, c
PRINT *, a, p, b, q, c
```

```
123e–2 abcdefghijkl –003 "P""Q'R" 4.56
```

Produces (under one compiler):

```
1.2300000 abcdefgh –3.0000000 P"Q'R
4.5599999
```

Free-Format

Free–format I/O is not traditional in Fortran

Formatted output is far more flexible
Fortran 2003 adds some free–format support

Free–format input can be very tricky in Fortran
But it isn't hard to read lists of numbers

There is some more on this in extra slides

Unformatted I/O is Simple

Very few users have any trouble with it

- It is NOT like C binary I/O
- It is unlike anything in C

Most problems come from "thinking in C"

Unformatted I/O (1)

- It is what you use for saving data in files

E.g. writing your own checkpoint/restart
Or transferring bulk data between programs

- No formatting/decoding makes it a lot faster

100+ times less CPU time has been observed

- Assume same hardware and same system

If not, see other courses and ask for help

Unformatted I/O (2)

Just reads and writes data as stored in memory

- You must read back into the same types

- Each transfer uses exactly one record

With extra control data for record boundaries
You don't need to know what it looks like

- Specify FORM='unformatted' in OPEN

stdin, stdout and terminals are not suitable

That's ALL that you absolutely need to know!

Example

```
INTEGER, DIMENSION(1000) :: index
REAL, DIMENSION(1000000) :: array

OPEN (31, FILE='fred', FORM='unformatted')

DO k = 1,...
    WRITE (31) k, m, n, index(:m), array(:n)
END DO
```

In another run of the program, or after rewinding:

```
DO k = 1,...
    READ (31) junk, m, n, index(:m), array(:n)
END DO
```

Programming Notes

- Make each record (i.e. transfer) quite large

But don't go over 2 GB per record

- I/O with whole arrays is generally fastest

```
INTEGER :: N(1000000)
READ (29) N
```

Array sections should be comparably fast

- Remember about checking for copying

- Implied DO-loops should be avoided

At least for large loop counts

Formatted I/O

READ or WRITE with an explicit format
A format is just a character string
It can be specified in any one of three ways:

- A CHARACTER expression
- A CHARACTER array
  Concatenated in array element order
- The label of a FORMAT statement
  Old–fashioned, and best avoided

Formats (1)

A format is items inside parentheses
Blanks are ignored, except in strings

‘ (i3,f 5 . 2) ’ ≡ ‘(i3,f5.2)’

We will see why this is so useful later

Almost any item may have a repeat count

‘(3 i3, 2 f5.2)’ ≡ ‘(i3, i3, i3, f5.2, f5.2)’

Formats (2)

A group of items is itself an item
Groups are enclosed in parentheses

E.g. '(3 (2 i3, f5.2))' expands into:
'(i3, i3, f5.2, i3, i3, f5.2, i3, i3, f5.2)'

Often used with arrays and implied DO–loops

Nesting them deeply can be confusing

Example

```
REAL, DIMENSION(2, 3) :: coords
INTEGER, DIMENSION(3) :: index

WRITE (29, '( 3 ( i3, 2 f5.2 ) )')    &
      ( index(i), coords(:, i), i = 1,3)
```

This is how to use a CHARACTER constant:

```
CHARACTER(LEN=*), PARAMETER ::    &
      format = '( 3 ( i3, 2 f5.2 ) )'

WRITE (29, format) ( index(i), coords(:, i), i = 1,3)
```

Transfer Lists And Formats

Logically, both are expanded into flat lists
I.e. sequences of basic items and descriptors

The transfer list is the primary one
Basic items are taken from it one by one
Each then matches the next edit descriptor

The item and descriptor must be compatible
E.g. REAL vars must match REAL descs

Input Versus Output

We shall mainly describe formatted output
This is rather simpler and more general

Unless mentioned, all descriptions apply to input
It's actually much easier to use than output
But it is rather oriented to form–filling

More on flexible and free–format input later

Integer Descriptors

In (i.e. letter i) displays in decimal
Right–justified in a field of width n
In.m displays at least m digits

WRITE (*, '(I7)') 123 $\Rightarrow$ '   123'
WRITE (*, '(I7.5)') 123 $\Rightarrow$ ' 00123'

You can replace the I by B, O and Z
For binary, octal and hexadecimal

Example

```
WRITE (*, '( I7, I7 )') 123, -123
WRITE (*, '( I7.5, I7.5 )') 123, -123

    123   -123
  00123 -00123

WRITE (*, '( B10, B15.10 )') 123, 123
WRITE (*, '( O7, O7.5 )') 123, 123
WRITE (*, '( Z7, Z7.5 )') 123, 123

   1111011     0001111011
    173   00173
     7B   0007B
```

Values Too Large

This is field overflow on output
The whole field is replaced by asterisks

Putting 1234 into i4 gives 1234
Putting 12345 into i4 gives ****
Putting –123 into i4 gives –123
Putting –1234 into i4 gives ****

This applies to all numeric descriptors
Both REAL and INTEGER

Fixed-Format REAL

Fn.m displays to m decimal places
Right–justified in a field of width n

```
WRITE (*, '( F9.3 )') 1.23        ⇒ '    1.230'
WRITE (*, '( F9.5 )') 0.123e–4 ⇒ '  0.00001'
```

You may assume correct rounding
Not required, but traditional in Fortran

- Compilers may round exact halves differently

Widths of Zero

For output a width of zero may be used
But only for formats I, B, O, Z and F
It prints the value without any leading spaces

```
write (*, '("/",i0,"/",f0.3)') 12345, 987.654321
```

Prints

```
/12345/987.65
```

Exponential Format (1)

There are four descriptors: E, ES, EN and D
With the forms En.m, ESn.m, ENn.m and Dn.m

All of them use m digits after the decimal point
Right–justified in a field of width n

D is historical – you should avoid it
Largely equivalent to E, but displays D

For now, just use ESn.m – more on this later

Exponential Format (2)

The details are complicated and messy
You don't usually need to know them in detail
Here are the two basic rules for safety

- In En.m and ESn.m, make $n \geq m+7$

That's a good rule for other languages, too

- Very large or small exponents display oddly

I.e. exponents outside the range –99 to +99
Reread using Fortran formatted input only

Numeric Input

F, E, ES, EN and D are similar

- You should use only Fn.0 (e.g. F8.0)

For extremely complicated reasons

- Any reasonable format of value is accepted

There are more details given later

CHARACTER Descriptor

An displays in a field with width n
Plain A uses the width of the CHARACTER item

On output, if the field is too small:
    The leftmost characters are used
Otherwise:
    The text is right–justified

On input, if the variable is too small:
    The rightmost characters are used
Otherwise:
    The text is left–justified

Output Example

```
WRITE (*,'(a3)') 'a'
WRITE (*,'(a3)') 'abcdefgh'
```

Will display:

```
  a
abc
```

Input Example

```
CHARACTER(LEN=3) :: a

READ (*,'(a8)') a ;    WRITE (*,'(a)') a
READ (*,'(a1)') a ;    WRITE (*,'(a)') a
```

With input:

```
abcdefgh
a
```

Will display:

```
fgh
a
```

LOGICAL Descriptor

Ln displays either T or F
Right–justified in a field of width n

On input, the following is done

- Any leading spaces are ignored
- An optional decimal point is ignored
- The next char. must be T (or t) or F (or f)
- Any remaining characters are ignored

E.g. '.true.' and '.false.' are acceptable

The G Descriptor

The G stands for generalized
It has the forms Gn or Gn.m
It behaves according to the item type

INTEGER behaves like In
CHARACTER behaves like An
LOGICAL behaves like Ln
REAL behaves like Fn.m or En.m
depending on the size of the value

The rules for REAL are fairly sensible

Other Types of Descriptor

All of the above are data edit descriptors
Each of them matches an item in the transfer list
As mentioned, they must match its type

There are some other types of descriptor
These do not match a transfer list item
They are executed, and the next item is matched

Text Literal Descriptor

A string literal stands for itself, as text
It is displayed just as it is, for output
It is not allowed in a FORMAT for input

Using both quotes and apostrophes helps
The following are all equivalent

```
WRITE (29, '( "Hello" )')
WRITE (29, "( 'Hello' )")
WRITE (29, '( ''Hello'' )')
WRITE (29, "( ""Hello"" )")
```

Spacing Descriptor

X displays a single blank (i.e. a space)
It has no width, but may be repeated

On input, it skips over exactly one character

```
READ (*, '(i1, 3x, i1)') m, n
WRITE (*, '(i1, x, i1, 4x, a)') m, n, '!'
```

```
7PQR9
```

Produces '7 9   !'

Newline Descriptor (1)

/ displays a single newline (in effect)
It has no width, but may be repeated

It can be used as a separator (like a comma)
Only if it has no repeat count, of course

```
WRITE (*, '(i1/i1, 2/, a)') 7, 9, '!'

7
9

!
```

Newline Descriptor (2)

On input, it skips the rest of the current line

```
READ (*, '(i1/i1, 2/, i1)') l, m, n
WRITE (*, '(i1, 1x, i1, 1x, i1)') l, m, n

1 1 1 1
2 2 2 2
3 3 3 3
4 4 4 4
```

Produces "1 2 4"

Item-Free FORMATs

You can print multi–line text on its own

```
WRITE (*, '("Hello" / "Goodbye")')

Hello
Goodbye
```

And skip as many lines as you like

```
READ (*, '(////)')
```

Generalising That

That is a special case of a general rule

FORMATs are interpreted as far as possible

```
WRITE (*, '(I5, " cubits", F5.2)') 123

  123 cubits
```

This reads 42 and skips the following three lines

```
READ (*, '(I3///)') n

 42
```

Complex Numbers

For list–directed I/O, these are basic types
E.g. read and displayed like “(1.23,4.56)”

For formatted and unformatted I/O
COMPLEX numbers are treated as two REALs
Like an extra dimension of extent two

```
COMPLEX :: c = (1.23, 4.56)
WRITE (*, '(2F5.2,3X,2F5.2)') c, 2.0*c
```

```
 1.23  4.56     2.46  9.12
```

Exceptions and IOSTAT (1)

By default, I/O exceptions halt the program
These include an unexpected end–of–file

You trap by providing the IOSTAT specifier

```
INTEGER :: ioerr

OPEN (1, FILE='fred', IOSTAT=ioerr)

WRITE (1, IOSTAT=ioerr) array

CLOSE (1, IOSTAT=ioerr)
```

Exceptions and IOSTAT (2)

IOSTAT specifies an integer variable

Zero means success, or no detected error

Positive means some sort of I/O error
An implementation should describe the codes

Negative means end–of–file (but see later)
Fortran 2003 provides its value – see next lecture

What Is Trapped? (1)

The following are NOT errors
Fortran defines all of this behaviour

- Formatted READ beyond end–of–record

Padded with spaces to match the format

Fortran 2003 allows a little control of that

- Writing a value too large for a numeric field

The whole field is filled with asterisks (*****)

What Is Trapped? (2)

The following are NOT errors

- Writing too long a CHARACTER string

The leftmost characters are used

- Reading too much CHARACTER data

The rightmost characters are used

What Is Trapped? (3)

The following is what you can usually rely on

- End–of–file

- Unformatted READ beyond end–of–record
In most compilers, IOSTAT will be negative

- Most format errors (syntactically bad values)
E.g. 12t8 being read as an integer

That is roughly the same as C and C++

What Is Trapped? (4)

The following are sometimes trapped
The same applies to most other languages

- Numeric overflow (integer or floating–point)

Floating–point overflow may just deliver infinity
Integer overflow may wrap modulo 2^N
Or there may be even less helpful effects

- 'Real' (hardware or system) I/O errors

E.g. no space on writing, file server crashing
Anything may happen, and chaos is normal

2 GB Warning

I said "chaos is normal" and meant it
Be careful when using files of more than 2 GB

Most filesystems nowadays will support such files
But not all of the interfaces to them do
Things like pipes and sockets are different again

- Has nothing to do with the Fortran language

Different compilers may use different interfaces
And there may be options you have to specify

OPEN

Files are connected to units using OPEN

```
OPEN (UNIT=11, FILE='fred', IOSTAT=ioerr)
```

That will open a sequential, formatted file
You can then use it for either input or output

You can do better, using optional specifiers
Other types of file always need one or more

Choice of Unit Number

Unit numbers are non–negative integer values
The valid range is system–dependent
You can usually assume that 1–99 are safe

Some may be in use (e.g. for stdin and stdout)
They are often (not always) 5 and 6

It is simplest to use unit numbers 10–99
Most codes just do that, and have little trouble

- Better ways of doing it covered in next lecture

ACCESS and FORM Specifiers

These specify the type of I/O and file

'sequential' (default) or 'direct'
'formatted' (default) or 'unformatted'

```
OPEN (UNIT=11, FILE='fred', ACCESS='direct',    &
    FORM='unformatted', RECL=500, IOSTAT=ioerr)
```

That will open a direct–access, unformatted file
with a record length of 500
You can then use it for either input or output

Scratch Files

```
OPEN (UNIT=11, STATUS='scratch',    &
    FORM='unformatted', IOSTAT=ioerr)
```

That will open a scratch (temporary) file
It will be deleted when it is closed

It will be sequential and unformatted
That is the most common type of scratch file
But all other types and specifiers are allowed

- Except for the FILE specifier

The ACTION Specifier

- This isn't needed, but is strongly advised

It helps to protect against mistakes
It enables the reading of read–only files

```
OPEN (UNIT=11, FILE='fred', ACTION='read',    &
      IOSTAT=ioerr)
```

Also 'write', useful for pure output files

The default, 'readwrite', allows both

Example (1)

Opening a text file for reading data from

```
OPEN (UNIT=11, FILE='fred', ACTION='read',    &
      IOSTAT=ioerr)
```

Opening a text file for writing data or results to

```
OPEN (UNIT=22, FILE='fred', ACTION='write',    &
      IOSTAT=ioerr)
```

```
OPEN (UNIT=33, FILE='fred', ACTION='write',    &
      RECL=80, DELIM='quote', IOSTAT=ioerr)
```

Example (2)

Opening an unformatted file for reading from

```
OPEN (UNIT=11, FILE='fred', ACTION='read',    &
      FORM='unformatted', IOSTAT=ioerr)
```

Opening an unformatted file for writing to

```
OPEN (UNIT=22, FILE='fred', ACTION='write',    &
      FORM='unformatted', IOSTAT=ioerr)
```

Example (3)

Opening an unformatted workspace file
It is your choice whether it is temporary

```
OPEN (UNIT=22, STATUS='scratch',     &
      FORM='unformatted', IOSTAT=ioerr)

OPEN (UNIT=11, FILE='/tmp/fred',     &
      FORM='unformatted', IOSTAT=ioerr)
```

See extra slides for direct–access examples

Omitted For Sanity

These are in the extra slides

Techniques for reading free–format data
Some more detail on formatted I/O
Internal files and dynamic formats
More on OPEN, CLOSE, positioning etc.
Direct–access I/O

There are extra, extra slides on some details

Introduction to Modern Fortran

Data Pointers

Nick Maclaren

nmm1@cam.ac.uk

March 2014

Data Pointers

- Fortran pointers are unlike C/C++ ones
Not like Lisp or Python ones, either

- Errors with using pointers are rarely obvious
This statement applies to almost all languages

- Fortran uses a semi–safe pointer model
Translation: your footgun has a trigger guard

Use pointers only when you need to

Pointer and Allocatable

Pointers are a sort of changeable allocation
In that use, they almost always point to arrays
For example, needed for non–rectangular arrays

Always try to use allocatable arrays first
Only if they really aren't adequate, use pointers

ALLOCATABLE was restricted in Fortran 95
Fortran 2003 removed almost all restrictions
You may come across POINTER in old code
It can usually be replaced by ALLOCATABLE

Pointer-Based Algorithms

Some genuinely pointer–based algorithms
Fortran is not really ideal for such uses

- But don't assume anything else is any better!

There are NO safe pointer–based languages
Theoretically, one could be designed, but ...

In Fortran, see if you can use integer indices
That has software engineering advantages, too
If you can't, you may have to use pointers

Pointer Concepts

Pointer variables point to target variables

In almost all uses, pointers are transparent

- You access the target variables they point to

Dereferencing the pointer is automatic

- Special syntax for meaning the pointer value

The POINTER attribute indicates a pointer

The TARGET attribute indicates a target

No variable can have both attributes

Example

```
PROGRAM fred
    REAL, TARGET :: popinjay = 0.0
    REAL, POINTER :: arrow
    arrow => popinjay
    ! arrow now points to popinjay
    arrow = 1.23
    PRINT *, popinjay
    popinjay = 4.56
    PRINT *, arrow
END PROGRAM fred
```

```
 1.2300000
 4.5599999
```

Pointers and Target Arrays

```
REAL, DIMENSION(20), TARGET :: array
REAL, DIMENSION(:), POINTER :: index
```

Pointer arrays must be declared without bounds
They will take their bounds from their targets

- Pointer arrays have just a rank

Which must match their targets, of course

Very like allocatable arrays

Use of Targets

Treat targets just like ordinary variables

The ONLY difference is an extra attribute
Allows them on the RHS of pointer assignment

Valid targets in a pointer assignment?
If OK for INTENT(INOUT) actual argument
Variables, array elements, array sections etc.

```
REAL, DIMENSION(20, 20), TARGET :: array
REAL, DIMENSION(:, :), POINTER :: index
index => array(3:7:2, 8:2:−1)
```

Initialising Pointers

Pointer variables are initially undefined

- Not initialising them is a Bad Idea

- You can use the special syntax => null()

To initialise them to disassociated (*sic*)

```
REAL, POINTER :: index => null()
```

- Or you can point them at a target, ASAP

Note that null() is a disassociated target

Pointer Assignment

You use the special assignment operator =>
Note that using = assigns to the target

```
PROGRAM fred
    REAL, TARGET :: popinjay
    REAL, POINTER :: arrow
    arrow => popinjay        ! POINTER assignment
    ! arrow now points to popinjay
    arrow = 1.23             ! TARGET assignment
    PRINT *, popinjay
    popinjay = 4.56          ! TARGET assignment
    PRINT *, arrow
    arrow => null()          ! POINTER assignment
END PROGRAM fred
```

Pointer Expressions

Also pointer expressions on the RHS of =>
Currently, only the results of function calls

```
FUNCTION select (switch, left, right)
    REAL, POINTER :: select, left, right
    LOGICAL switch
    IF (switch) THEN
        select => left
    ELSE
        select => right
    END IF
END FUNCTION select

new_arrow => select(A > B, old_arrow, null())
```

ALLOCATE

You can use this just as for allocatable arrays
This creates some space and sets up array

```
REAL, DIMENSION(:, :), POINTER :: array
ALLOCATE(array(3:7:2, 8:2:−1), STAT=n)
```

If you can, stick to using ALLOCATABLE

Do you get the idea I don't like pointers much?
At the end, I mention why you may need them

DEALLOCATE

- Only on pointers set up by ALLOCATE

DEALLOCATE(array, STAT=n)

array now becomes disassociated
Other pointers to its target become undefined

- Don't DEALLOCATE undefined pointers
That is undefined behaviour

Previous Pointer Values

New pointer value overwrites the previous one
Applies to both assignment and ALLOCATE
Well, it is a sort of assignment . . .

- Does not affect other pointers to the target

But DEALLOCATE makes other pointers undefined
Also happens if the target goes out of scope

- That causes the dangling pointer problem

And assignment can break the last link

- Memory leaks and (rarely) worse problems

ASSOCIATED

- Can test if pointers are associated

```
IF (ASSOCIATED(array)) . . .
IF (ASSOCIATED(array, target)) . . .
```

Works if array is associated or disassociated
Latter tests if array is associated with target

- Don't use it on undefined pointers

That is undefined behaviour

A Nasty “Gotcha”

Fortran 95 forbids POINTER and INTENT

- Fortran 2003 applies INTENT to the link

```
subroutine joe (arg)
    real, target :: junk
    real, pointer, intent(in) :: arg
    allocate(arg)       ! this is ILLEGAL
    arg => junk         ! this is ILLEGAL
    arg = 4.56          ! but this is LEGAL :-(
end subroutine joe
```

Irregular Arrays

- Fortran does not support them

This is how you do the task, if you need to

```
TYPE Cell
    REAL, DIMENSION(:), ALLOCATABLE :: column
END TYPE Cell

TYPE(Cell), DIMENSION(:), ALLOCATABLE :: matrix
```

matrix can be a non−rectangular matrix
Note that pointers are not needed in this case

Example

```
TYPE Cell
    REAL, DIMENSION(:), ALLOCATABLE :: column
END TYPE Cell

TYPE(Cell), DIMENSION(:), ALLOCATABLE :: matrix

INTEGER, DIMENSION(100) :: rows
READ *, N, (rows(K), K = 1,N)
ALLOCATE(matrix(1:N))
DO K = 1,N
    ALLOCATE(matrix(K)%column(1:rows(K)))
END DO
```

Arrays of Pointers

- Fortran does not support them

This is how you do the task, if you need to

```
TYPE Cell
    REAL, DIMENSION(:), POINTER :: column
END TYPE Cell

TYPE(Cell), DIMENSION(100) :: matrix
```

Remember Trees?

This was the example we used in derived types

```
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
```

Recursive Types

We can do this more easily using recursive types

```
TYPE :: Node
    TYPE(Node), POINTER :: subnodes(:)
    CHARACTER(LEN=20) :: name
    REAL(KIND=dp), DIMENSION(3) :: data
END TYPE Node
```

Recursive components must be pointers
Fortran 2008 will allow allocatable
Obviously a type cannot include itself directly

More Complicated Structures

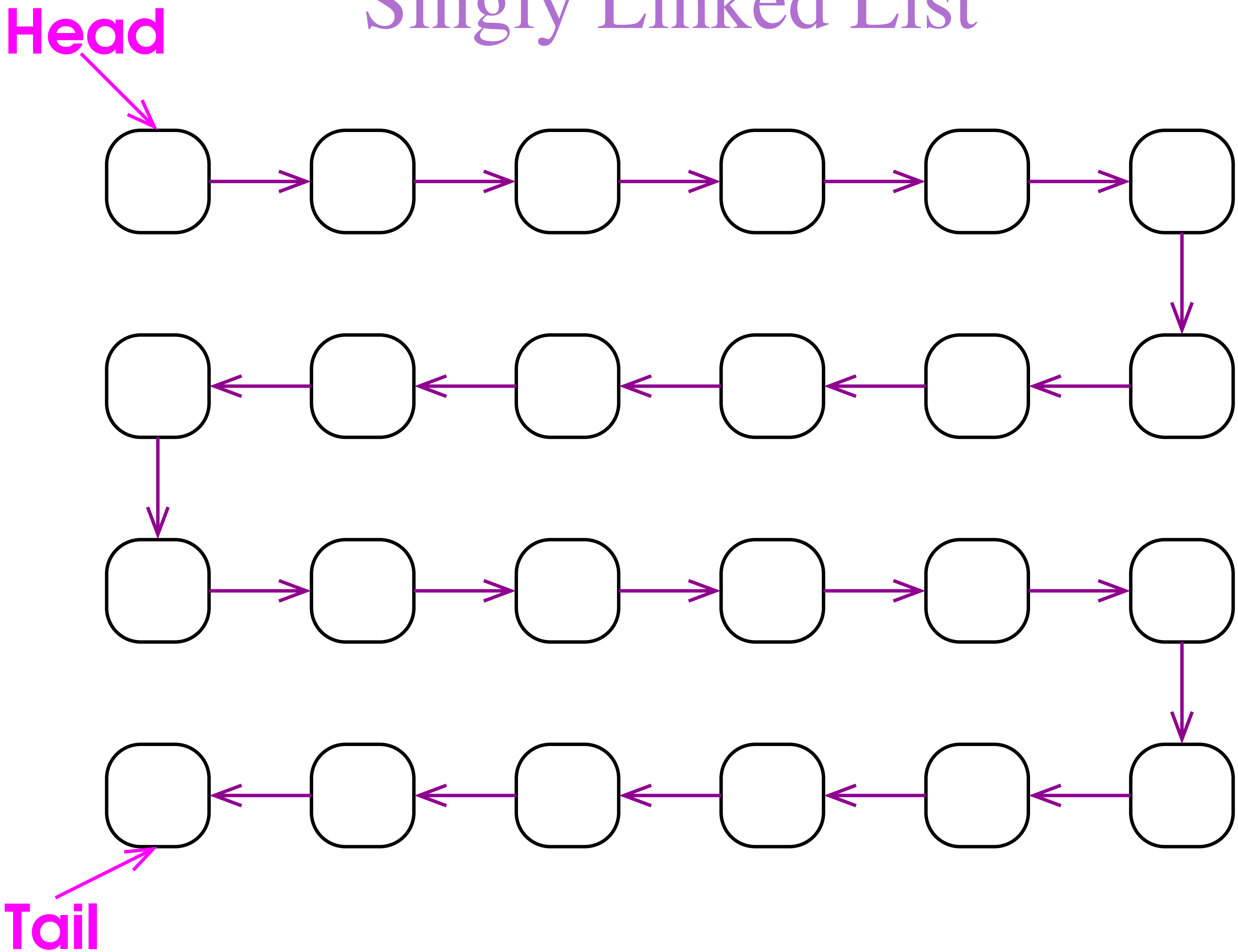
In mathematics, a graph is a set of linked nodes
Common forms include linked lists, trees etc.

A tree is just a hierarchy of objects
We have already covered these, in principle

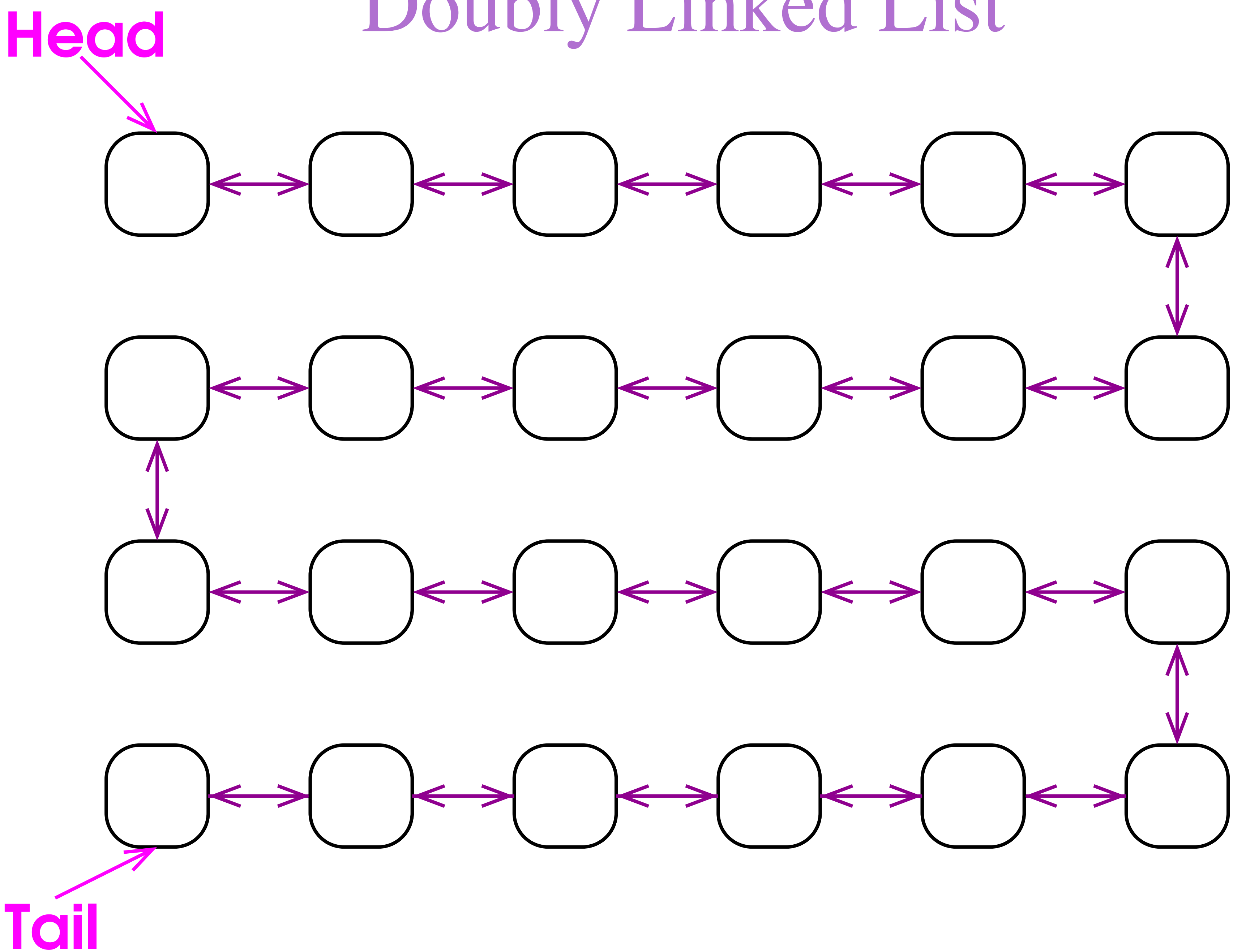
Linked lists (also called chains) are common
And there are lots of more complicated structures

Those are very painful to handle in old Fortran
So most Fortran programmers tend to avoid them
But they aren't difficult in modern Fortran

Singly Linked List



Doubly Linked List



Linked Lists

You can handle linked lists in a similar way
And any other graph−theoretic data structure, too

```
TYPE Cell
    CHARACTER(LEN=20) :: node_name
    REAL :: node_weight
    TYPE(Cell), POINTER :: next, last, &
        first_child, last_child
END TYPE Cell
```

Working with such data structures is non−trivial
Whether in Fortran or any other language

Graph Structures

Using pointers in Fortran is somewhat tedious
But it is as easy as in C++ and a little safer

Graph structures are in computer science
linked lists are probably the only easy case
Plenty of books on them, for example:

Cormen, T.H. et al. Introduction to Algorithms
Knuth,D.E. The Art Of Computer Programming
Also Sedgewick, Ralston, Aho et al. etc.

Procedure Pointers

Fortran 2003 allows them, as well as data pointers

Don't go there

This has absolutely nothing to do with Fortran
They are a nightmare in all languages, including C++
They are almost impossible to use safely
A fundamental problem in any scoped language

- Very rarely need them in clean code, anyway

Passing procedures as arguments is usually enough
Or one procedure calling a fixed set of others

Introduction to Modern Fortran

Advanced Array Concepts

Nick Maclaren

nmm1@cam.ac.uk

March 2014

Summary

This will describe some advanced array features
Use them only when you need their facilities

It will also cover some aspects of array use
Important for correctness and performance

There is a lot more on both

- Please ask if you need any help

Testing Allocation

Can test if an ALLOCATABLE object is allocated

```
INTEGER, DIMENSION(:), ALLOCATABLE :: counts
. . .
IF (ALLOCATED(counts)) THEN
    . . .
```

Warning: rules of (de)allocation are non–trivial
Can happen automatically under some circumstances

Generally, restructure your code to not need it

Higher Rank Constructors

Constructors create only rank one arrays
We shall now see how to construct higher ranks

It is done by constructing a rank one array
And then mapped using the RESHAPE function

This is very easy, but looks a bit messy

The RESHAPE Intrinsic (1)

This allows arbitrary restructuring of arrays
The following is only its very simplest use

```
RESHAPE (source, shape)
```

source provides the data in array element order
shape specifies the shape of array to deliver

The RESHAPE Intrinsic (2)

```
REAL, DIMENSION(3, 4) :: array

array = RESHAPE( (/ 1.1, 2.1, 3.1, 1.2, 2.2, &
      3.2, 1.3, 2.3, 3.3, 1.4, 2.4, 3.4 /),  (/ 3, 4 /) )
```

Is functionally equivalent to:

```
DO m = 1, 3
      DO n = 1, 4
            array(m, n) = m+0.1*n
      END DO
END DO
```

The RESHAPE Intrinsic (3)

It can be used in constant expressions

```
REAL, DIMENSION(3, 4) :: array = &
    RESHAPE( (/ 1.1, 2.1, 3.1, 1.2, 2.2, &
        3.2, 1.3, 2.3, 3.3, 1.4, 2.4, 3.4 /), (/ 3, 4 /) )
```

It also allows arbitrary reordering
And padding with copies of an array

See the references for more details

Example

Create the zero vector, and the three unit vectors

```
REAL, DIMENSION(1:3), PARAMETER :: &
     vec_0 = (/ 0.0, 0.0, 0.0 /), &
     vec_i = (/ 1.0, 0.0, 0.0 /), &
     vec_j = (/ 0.0, 1.0, 0.0 /), &
     vec_k = (/ 0.0, 0.0, 1.0 /)
```

Create the identity matrix

```
REAL, DIMENSION(1:3, 1:3), PARAMETER :: &
   identity = RESHAPE( (/ vec_i, vec_j, vec_k /), (/ 3, 3 /) )
```

RESHAPE More Generally

It isn't restricted to multi–dim. constants
You can use it for fancy array restructuring

- Study the specification before doing that

Restructuring arrays is dangerous territory

- And there are several other such intrinsics

I.e. ones with important uses but no simple uses

Vector Indexing (1)

Vectors may be used as indices

```
INTEGER, DIMENSION(1:5) :: &
        j = (/ 3, 1, 5, 2, 4 /), k = (/ 2, 3, 2, 1, 3 /)
REAL, DIMENSION(1:5) :: x, &
        y = (/ 1.2, 2.3, 3.4, 4.5, 5.6 /)
x(j) = y(k)
PRINT *, y(k)
PRINT *, x
```

```
2.3000000  3.4000001  2.3000000  1.2000000  3.4000001
3.4000001  1.2000000  2.3000000  3.4000001  2.3000000
```

Vector Indexing (2)

Using vector indices is a bit like sections
There are important differences – be careful

You can them for reading arrays quite safely
Elements must be distinct if updating

- NOT recommended for use in arguments

If used in arguments, those must not be updated
And it forces the compiler to copy the array

Masked Assignment (1)

Set all negative values in an array A to zero

```
REAL, DIMENSION(20, 30) :: array

DO j = 1,30
    DO k = 1,20
        IF (array(k,j) < 0.0) array(k,j) = 0.0
    END DO
END DO
```

But the WHERE statement is more convenient

```
WHERE (array < 0.0) array = 0.0
```

Masked Assignment (2)

It has a statement construct form, too

```
WHERE (array < 0.0)
    array = 0.0
ELSE WHERE
    array = 0.01*array
END WHERE
```

Masking expressions are LOGICAL arrays
You can use an actual array there, if you want
Masks and assignments need the same shape

Masked Assignment (3)

Fortran 2003 extends it considerably

Don't use LHS arrays in non–elemental functions
The following is asking for trouble:

```
WHERE (arr1 < arr2)
    arr1 = 1.0
ELSE WHERE
    arr2 = sum(arr1)
END WHERE
```

- Don't bother with the FORALL statement

Memory Efficiency (1)

Local arrays can be implemented in many ways
Only a few Ada compilers handle them properly

You can exhaust your program's stack with them
Too big, or too many due to deep recursion

- It will usually cause a truly horrible crash

Allocatable arrays always go on the 'heap'
Automatic arrays often go on the 'heap'
That is less efficient, but is handled much better

- Making all big arrays allocatable isn't stupid

Memory Efficiency (2)

As always, every solution has its own problems
Lots of allocation and deallocation isn't ideal

- Each (de)allocation costs some CPU time
Not generally a problem for Fortran programs

- Poor compilers may have memory leaks
Most Fortran compilers don't have them badly

Both because of the language's restrictions

Memory Efficiency (3)

- The big problem is memory fragmentation
Describing how and why is beyond this course
Luckily, in AD 2007, there is a simple solution

- Best one is to use 64–bit addressing
Gets rid of the worst of the problems, painlessly
I do that, even on systems with 2 GB of memory

- Please ask if you want to know more

Order of Evaluation (1)

Array assignments etc. are like implicit loops
But, except in I/O, no order of evaluation implied
Also the behaviour is different when modifying

- Each pass of a loop is executed in order
- Array assignments do it all ‘‘in parallel’’

- You should avoid code where it matters

The compiler may have to copy the array
It risks confusion when tuning your code

Order of Evaluation (2)

```
INTEGER, DIMENSION(5) :: array = (/ 1, 2, 3, 4, 5 /)
array(2:5) = array(1:4)
PRINT *, array

array = (/ 1, 2, 3, 4, 5 /)
DO k = 1,4
        array(k+1) = array(k)
END DO
PRINT *, array

1    1    2    3    4
1    1    1    1    1
```

Performance (1)

- Efficient use of arrays is critical

This course has NOT taught any of that
It covers quite enough without adding it!

- Generally, follow this procedure:

Start by writing clean and clear code
Get it working, and test it fairly thoroughly
If too slow, use a profiler to see where
And only then tune only those aspects

Performance (2)

You get most gain by using faster methods
Followed by the following aspects:

- Improve the layout and access patterns

This is locality (improved cache usage etc.)

- Avoid unnecessary array copying

Compilers often have to do that for some codes
Some compilers copy when they don't need to

- Improve the actual CPU efficiency

This is getting into advanced tuning

Memory Locality (1)

Things used together should be stored together
Remember that "first index varies fastest"

```
REAL, DIMENSION(3000, 5000) :: array
DO n = 1, 5000
    DO m = 1, 3000
        array(m, n) = m+0.1*n
    END DO
END DO
```

- Note that the first index varies fastest

Memory Locality (2)

Sections and masking can cause trouble

```
REAL, DIMENSION(1000, 1000) :: array
CALL FRED( array(123, :) )
```

The elements of the vector are a long way apart
A problem if FRED accesses it a lot

- Consider making a temporary copy of it

Access Patterns

- Sequential access is generally efficient
Avoid non–sequential access whereever possible

- This can be much slower than sequential

```
REAL, DIMENSION(1000) :: arr1, arr2
INTEGER, DIMENSION(1000) :: random
arr1(random) = arr2(random)
```

Unnecessary Copying (1)

It is hard to describe when this may occur
It helps if you can mentally compile the code

- Avoiding using the LHS array on the RHS
Except when the uses are purely elemental

- Generally, sections do not need a copy
Unlike arguments with vector indexed arrays

- Compilers often do unnecessary copying
In a very bad case, even for CALL Fred(data(:))

Example

```
INTEGER :: arr1(1:50), arr2(1:100), arr3(1:100)
REAL, DIMENSION(20, 20) :: mat1, mat2, mat3
```

These shouldn't require a copy

```
arr1 = arr1+arr2(1:50)+arr3(arr2(51:100))
mat1 = MATMUL(mat2, mat3)
```

But these almost certainly will

```
arr1 = arr1(::-1)+arr2(1:50)
mat1 = MATMUL(mat1, mat2)
```

Unnecessary Copying (2)

And, while this shouldn't, ...

```
mat1 = mat1 + MATMUL(mat2, mat3)
```

There is more on this under procedures

- Generally, don't worry unless you have to
If your program runs fast enough, who cares?

- If not, time and profile it first
Ask for advice if you have problems

High-Performance Problems

There are some other problems some people hit
Too complicated to even describe here

- Ignore them until you have problems

Then ask for help with tackling them

Buzzwords and phrases include:

TLB thrashing
Cache conflicts
False sharing
Memory banking

Reminder

- You don't have to remember all of this
- Start by using the simplest features only
- Use the fancy ones only when you need them

If you know they exist, you can look them up

Introduction to Modern Fortran

Advanced Use Of Procedures

Nick Maclaren

nmm1@cam.ac.uk

March 2014

Summary

We have omitted some important concepts
They are complicated and confusing

There are a lot of features we have omitted
Mostly because they are hard to use correctly
And sometimes because they are inefficient

This lecture covers some of the most important

- Refer to this when you need to

ALLOCATABLE and POINTER

You can pass ALLOCATABLE and POINTER arrays
In the usual case, the procedure has neither
The dummy argument is associated with the data

- You can't reallocate or redirect in the procedure

To do that, declare the dummy argument as
ALLOCATABLE or POINTER, as appropriate

Warning for INTENT(OUT) and ALLOCATABLE:
These are deallocated on entry, even if not used

Association (1)

Fortran uses argument association in calls
Dummy arguments refer to the actual ones

- You don't need to know exactly how it is done

It may be aliasing or copy–in/copy–out

Expressions are stored in a hidden variable
The dummy argument is associated with that

- It obviously must not be updated in any way

Using INTENT is strongly recommended

Association (2)

```
REAL, DIMENSION(1:10, 1:20, 1:3) :: data
CALL Fred (data(:, 5:15, 2), 1.23*xyz )

SUBROUTINE Fred (array, value)
REAL, DIMENSION(:, :) :: array
REAL, INTENT(IN) :: value
```

array in fred refers to data(:, 5:15, 2)
value refers to a location containing 1.23*xyz

Updating Arguments (1)

A dummy argument must not be updated if:

- The actual argument is an expression
- It overlaps another argument in any way

```
REAL, DIMENSION(1:20, 1:3) :: data
CALL Fred (data(5:15, 2), data(17:, 2))

SUBROUTINE Fred (arr1, arr2)
REAL, DIMENSION(:) :: arr1, arr2
arr1 = 1.23 ;    arr2 = 4.56
```

- The above works as you expect

Updating Arguments (2)

```
REAL, DIMENSION(1:20, 1:3) :: data
CALL Fred (data(5:15, 2), data(1:10, 2))

SUBROUTINE Fred (arr1, arr2)
REAL, DIMENSION(:) :: arr1, arr2
arr2(1, 1) = 4.56
```

- The above is not allowed

Because arr1 and arr2 overlap

Even though arr2(1, 1) is not part of arr1

Updating Arguments (3)

```
REAL :: X
CALL Fred (X + 0.0)

SUBROUTINE Fred (Y)
Y = 4.56
```

- The above is not allowed – obviously

- That also applies to array expressions
Vector indexing behaves like an expression

Warning for C/C++ People

```
REAL, DIMENSION(1:20) :: data
CALL Fred (data(2), data)

SUBROUTINE Fred (var, array)
REAL :: var
REAL, DIMENSION(:) :: array
array = 4.56
```

- The above is not allowed, either

Even array elements are associated

Using Functions

Functions are called just like built–in ones
They may be optimised in similar ways

```
REAL :: scale, data(1000)
    . . .
READ *, scale    ! assume that this reads 0.0
Z = Variance(data)/(scale+Variance(data)}
```

Variance may be called 0, 1 or 2 times

Impure Functions

Pure functions have defined behaviour

- Whether they are declared PURE or not

Impure functions occasionally misbehave
Generally, because they are over–optimised

There are rules for safety in practice
But they are too complicated for this course

- Ask if you need help with this

FUNCTION Result Variable

The function name defines the result variable
You can change this if you prefer

```
FUNCTION Variance_of_an_array (Array) RESULT(var)
    REAL :: var
    REAL, INTENT(IN), DIMENSION(:) :: Array
    var = SUM(Array)/SIZE(Array)
    var = SUM((Array-var)**2)/SIZE(Array)
END FUNCTION Variance_of_an_array

    REAL, DIMENSION(1000) :: data
        . . .
    Z = Variance_of_an_array(data)
```

PURE Subroutines

You can declare a subroutine to be PURE

Like functions, but with one fewer restriction
INTENT(OUT) and INTENT(INOUT) are allowed

```
PURE SUBROUTINE Init (array, value)
    REAL, DIMENSION(:), INTENT(OUT) :: array
    REAL, INTENT(IN) :: value
    array = value
END SUBROUTINE Init
```

They can be declared as ELEMENTAL, too

Recursion

Fortran 90 allowed this for the first time

Recursive procedures must be declared as such

- If you don't, recursion may cause chaos

```
RECURSIVE SUBROUTINE Chop (array, value)
    . . .
```

- Avoid it unless you actually need it

- Check all procedures in the recursive loop

OPTIONAL Arguments

- Use OPTIONAL for setting defaults only
On entry, check and copy ALL args
Use ONLY local copies thereafter
Now, all variables are well defined when used

- Can do the converse for optional results
Just before returning, check and copy back

- Beyond this should be done only by experts

OPTIONAL Example (1)

```
FUNCTION fred (alf, bert)
REAL :: fred, alf, mybert
REAL, OPTIONAL, INTENT(IN) :: bert
IF (PRESENT(bert)) THEN
    mybert = bert
ELSE
    mybert = 0.0
END IF
```

Now use mybert in rest of procedure

OPTIONAL Example (2)

```
SUBROUTINE fred (alf, bert)
REAL :: alf
REAL, OPTIONAL, INTENT(OUT) :: bert
...
IF (PRESENT(bert)) bert = ...
END SUBROUTINE fred
```

Fortran 2003

Adds potentially useful VALUE attribute
See OldFortran course for information
Seriously. It's also useful for conversion

And the PROCEDURE declaration statement
A cleaner and more modern form of EXTERNAL
Its usage is not what you would expect, though

And probably more ...

Arrays and CHARACTER

We have over–simplified these so far
No problem, if you use only recommended style

- You need to know more if you go beyond that

- We start by describing what you can do
Including some warnings about efficient use

And then continue with how it actually works

Array Valued Functions

Arrays are first–class objects in Fortran
Functions can return array results

- In practice, doing so always needs a copy

However, don't worry too much about this

Declare the function just as for an argument
The constraints on the shape are similar

- If it is too slow, ask for advice

Example

This is a bit futile, but shows what can be done

```
FUNCTION operate (mat1, mat2, mat3)
    IMPLICIT NONE
    REAL, DIMENSION(:, :), INTENT(IN) :: &
        mat1, mat2, mat3
    REAL, DIMENSION(UBOUND(mat1, 1), &
        UBOUND(mat2, 2)) :: operate
! Checking omitted, again
    operate = MATMUL(mat1, mat2) + mat3
END FUNCTION operate
```

Array Functions and Copying

The result need not be copied on return
The interface provides enough information
In practice, don't bet on it ...

Array functions can also fragment memory
Ask if you want to know how and why

- Generally a problem only for HPC

I.e. when either time or memory are bottlenecks

What Can Be Done

- Just use array functions regardless

If you don't have a problem, why worry?

- Time and profile your program

Tune only code that is a bottleneck

- Rewrite array functions as subroutines

I.e. turn the result into an argument

- Use ALLOCATABLE results (sic)

- Ask for further advice with tuning

CHARACTER And Copying

In this respect, CHARACTER $\equiv$ array
Most remarks about arrays apply, unchanged

- But it is only rarely important

Fortran is rarely used for heavy character work
It works fairly well, but it isn't ideally suited
Most people find it very tedious for that

- If you need to, ask for advice

Character Valued Functions (1)

Earlier, we considered just one form
Almost anything more needs a copy
Some compilers will copy even those

- Often, the cost of that does not matter

You are not restricted to just that form
Declare the function just as for an argument
The constraints on the shape are similar

- If it is too slow, ask for advice

Character Valued Functions (2)

The result length can be taken from an argument

```
FUNCTION reverse_word (word)
    IMPLICIT NONE
    CHARACTER(LEN=*), INTENT(IN) :: word
    CHARACTER(LEN=LEN(word)) :: reverse_word
    INTEGER :: I, N
    N = LEN(word)
    DO I = 1, N
        reverse_word(I:I) = word(N+1-I:N+1-I)
    END DO
END FUNCTION reverse_word
```

Character Valued Functions (3)

This is a bit futile, but shows what can be done
The result length is a non–trivial expression

```
FUNCTION interleave (text1, count, text2)
    IMPLICIT NONE
    CHARACTER(LEN=*), INTENT(IN) :: text1, text2
    INTEGER, INTENT(IN) :: count
    CHARACTER(LEN=LEN(text1)+count+ &
        LEN(text2)) :: interleave
    interleave = text1 // REPEAT(' ', count) // text2
END FUNCTION interleave
```

Explicit/Assumed Size/Shape (1)

- The good news is that everything works

Can mix assumed and explicit *ad lib.*

There are some potential performance problems

- Passing assumed to explicit forces a copy

- It can be a problem calling some libraries

Especially ones written in old Fortran

- Write clean code, and see if it is fast enough

If you find that it isn't, ask for advice

Explicit/Assumed Size/Shape (2)

This code is not a problem:

```
SUBROUTINE Weeble (matrix)
    REAL, DIMENSION(:, :) :: matrix
END SUBROUTINE Weeble

SUBROUTINE Burble (space, M, N)
    REAL, DIMENSION(M, N) :: space
    CALL Weeble(space)
END SUBROUTINE Burble

REAL, DIMENSION(100,200) :: work
CALL Burble(work, 100, 200)
```

Explicit/Assumed Size/Shape (3)

Nor even something as extreme as this:

```
SUBROUTINE Weeble (matrix)
    REAL, DIMENSION(:, :) :: matrix
END SUBROUTINE Weeble

SUBROUTINE Burble (space, N, J1, K1, J2, K2)
    REAL, DIMENSION(N, *) :: space
    CALL Weeble(space(J1:K1, J2:K2))
END SUBROUTINE Burble

REAL, DIMENSION(100, 200) :: work
CALL Burble(work, 100, 20, 80, 30, 70)
```

Explicit/Assumed Size/Shape (4)

But this code forces a copy:

```
SUBROUTINE Bubble (matrix, M, N)
    REAL, DIMENSION(M, N) :: matrix
END SUBROUTINE Bubble

SUBROUTINE Womble (space)
    REAL, DIMENSION(:, :) :: space
    CALL Bubble(space, UBOUND(space, 1), &
        UBOUND(space, 2))
END SUBROUTINE Womble

REAL, DIMENSION(100,200) :: work
CALL Womble(work)
```

Example – Calling LAPACK

LAPACK is written in Fortran 77
It cannot handle assumed shape arrays
So here is how to call SPOTRF (Cholesky)

```
SUBROUTINE Chol (matrix, info)
    REAL, DIMENSION(:, :), INTENT(INOUT) :: matrix
        INTEGER, INTENT(INOUT) :: info
        CALL SPOTRF('L', UBOUND(matrix, 1), &
                matrix, UBOUND(matrix, 1), info)
END SUBROUTINE Chol
```

matrix will be copied on call and return

Sequence Association (1)

Have covered assumed shape and char. length
And explicit shape and char. length
but only when the dummy and actual match

- That constraint is not required (nor checked)

You need to know an extra concept to go further
That is called sequence association

- You are recommended to go cautiously here

Don't do it until you are confident with Fortran

Sequence Association (2)

Explicit shape and assumed size arrays only
If the dummy and actual bounds do not match

Argument is flattened in array element order
And is given a shape by the dummy bounds
Exactly the way the RESHAPE intrinsic works

There are important uses of this technique

- Or you can shoot yourself in the foot

Example

```
SUBROUTINE operate_1 (vector, N)
    REAL, DIMENSION(N) :: vector
    . . .
SUBROUTINE operate_2 (matrix, M, N)
    REAL, DIMENSION(M, N) :: matrix
    . . .

REAL, DIMENSION(1000000) :: workspace
. . .
IF (cols = 0) THEN
  CALL operate_1(workspace, rows)
ELSE
  CALL operate_2(workspace, rows, cols)
END IF
```

Sequence Association (3)

The same holds for explicit length CHARACTER
Everything is concatenated and then reshaped

Character lengths are like an extra dimension
Naturally, it varies faster than the first index

One restriction needed to make this work
Assumed shape arrays of CHARACTER
need assumed length or matching lengths

Example

```
SUBROUTINE operate (fields, N)
    CHARACTER(LEN=8), DIMENSION(10, N) :: fields
END SUBROUTINE operate

CHARACTER(LEN=80), DIMENSION(1000) :: lines
. . .
! Read in N lines
CALL operate(lines, N)
```

Implicit Interfaces (1)

Calling an undeclared procedure is allowed
The actual arguments define the interface

- I strongly recommend not doing this

Mistyped array names often show up as link errors

```
REAL, DIMENSION(1000) :: lines
. . .
lines(5) = lones(7)
```

Undefined symbol lones_ in file test.o

Implicit Interfaces (2)

Only Fortran 77 interface features can be used
The args and result must be exactly right
Must declare the result type of functions

```
REAL, DIMENSION(KIND=dp) :: DDOT
. . .
X = DDOT(array)
```

- This is commonly done for external libraries

I.e. ones that are written in Fortran 77, C etc.

- Interface modules are a better way

EXTERNAL

This declares an external procedure name

It's essential only when passing as argument
I.e. if the procedure name is used but not called

- I recommend it for all undeclared procedures

More as a form of documentation than anything else

- But explicit interfaces are always better

Example

Here is the LAPACK example again

```
SUBROUTINE Chol (matrix, info)
  REAL, DIMENSION(:, :), INTENT(INOUT) :: matrix
    INTEGER, INTENT(INOUT) :: info
    EXTERNAL :: SPOTRF
    CALL SPOTRF('L', UBOUND(matrix, 1), &
        matrix, UBOUND(matrix, 1), info)
END SUBROUTINE Chol
```

Introduction to Modern Fortran

Advanced I/O and Files

Nick Maclaren

nmm1@cam.ac.uk

March 2014

Summary

This will describe some advanced I/O features
Some are useful but only in Fortran 2003
Some are esoteric or tricky to use

- The points here are quite important

Excluded only on the grounds of time

There is a lot more in this area

- Please ask if you need any help

Partial Records in Sequential I/O

Reading only part of a record is supported
Any unread data in the record are skipped
The next READ uses the next record

Fortran 90 allows you to change that

- But ONLY for formatted, external I/O

Specify ADVANCE='no' in the READ or WRITE
This is called non-advancing I/O

Non-Advancing Output

You can build up a record in sections

```
WRITE (*, '(a)', ADVANCE='no') 'value = '
IF (value < 0.0) THEN
   WRITE (*, '("None")') value
ELSE
   WRITE (*, '(F5.2)') value
END IF
```

This is, regrettably, the only portable use

Use for Prompting

```
WRITE (*, '(a)', ADVANCE='no') 'Type a number: '
READ (*, *) value
```

That will usually work, but may not

The text may not be written out immediately
Even using FLUSH may not force that

Too many prompts may exceed the record length

Non-Advancing Input

You can decode a record in sections
Just like for output, if you know the format

Reading unknown length records is possible
Here are two recipes that are safe and reliable

Unfortunately, Fortran 90 and Fortran 2003 differ

Recipe (1) - Fortran 2003

```
USE, INTRINSIC :: ISO_FORTRAN_ENV
CHARACTER, DIMENSION(4096) :: buffer
INTEGER :: status, count
READ (1, '(4096a)', ADVANCE='no', SIZE=count, &
        IOSTAT=status) buffer
```

If IOSTAT is IOSTAT_EOR, the record is short

If IOSTAT is IOSTAT_END, we are at end–of–file

SIZE returns the number of characters read

Recipe (2) - Fortran 90

```
CHARACTER, DIMENSION(4096) :: buffer
INTEGER :: count
READ (1, '(4096a)', ADVANCE='no', SIZE=count, &
      EOR=10, EOF=20) buffer
```

The EOR branch is taken if the record is short
The following happens whether or not it is

SIZE returns the number of characters read

General Free-Format Input

- Can read in whole lines, as described above
And then decode using CHARACTER operations
You can also use internal files for conversion

- Can use some other language for conversion
I use Python, but Perl is fine, too
Use it to convert to a Fortran–friendly format

- You can call C to do the conversion
That isn't always as easy as people think it is

List-Directed I/O (1)

This course has massively over–simplified
All you need to know for simple test programs
It is used mainly for diagnostics etc.

Here are a few of its extra features

Separation is by comma, spaces or both
That is why comma needs to be quoted
Theoretically, that can happen on output, too

List-Directed I/O (2)

You may use repeat counts on values
100*1.23 is a hundred repetitions of 1.23
That is why asterisk needs to be quoted
Theoretically, that can happen on output, too

There may be null values in input
“1.23 , , 4.56” is 1.23 , null value, 1.234.56
“100*   ” is a hundred null values
Null values suppress update of the variable

List-Directed I/O (3)

As described, slashes (/) terminates the call
That is why slash needs to be quoted

Before using it in complicated, important code:

- Read the specification, to avoid “gotchas”
- Work out exactly what you want to do with it

Formatted Input for REALs

m in Fn.m etc. is an implied decimal point
It is used only if you don't provide one
The k in En.mEk is completely ignored

And there are more historical oddities
Here is an extended set of rules

- Use a precision of zero (e.g. F8.0)
- Always include a decimal point in the number
- Don't use the P or BZ descriptors for input
- Don't set BLANK='zero' in OPEN or READ

The Sordid Details

If you want to know, read the actual standard
You won't believe me if I tell you!

And don't trust any books on this matter
They all over–simplify it like crazy

In any case, I doubt that any of you care
Follow the above rules and you don't need to

Choice of Unit Number

Preconnected units are open at program start
Includes at least ones referred to by UNIT=*

- OPEN on them will close the old connection

Can check for an open unit using INQUIRE

Fortran 2003 has a way of getting their numbers
Has names in the ISO_FORTRAN_ENV module

Critical only for significant, portable programs

INQUIRE By File (1)

You can check if a file exists or is open

```
LOGICAL :: here
INQUIRE (FILE='name', EXIST=here)
INQUIRE (FILE='name', OPENED=here)
```

- These answers may not mean what you expect

E.g. a new, output file may be open but not exist

- Name matching may be textual or by identity

Watch out when using ln or ln –s

INQUIRE By File (2)

Can query SIZE, READ, READWRITE, WRITE
Don't bet on it – not all compilers support them sanely
Some others, too, but not under Unix–like systems

Most other queries are handled like inquire by unit
Subject to matching the file name correctly
If not connected always return UNKNOWN
Not exactly the most useful behaviour!

However, at least they DO say UNKNOWN
And don't simply return plausible nonsense

INQUIRE By Unit (1)

Inquire by unit most usefully does two things:
Checks if the unit is currently connected
Returns the record length of an open file

```
LOGICAL :: connected
INQUIRE (UNIT=number, OPENED=connected)

INTEGER :: length
INQUIRE (UNIT=number, RECL=length)
```

You can ask about both together, of course

INQUIRE By Unit (2)

There are other potentially useful specifiers
Not all of them make much sense under POSIX

You can get all of the specifiers used for OPEN
Could be useful when writing generic libraries

SIZE gives the size of the file, probably in bytes
This is only in Fortran 2003, and unreliable
Again, nothing to do with Fortran, as such

See the references for details on them

Unformatted I/O

Using pipes or sockets is unreliable
The reasons are complicated and historical

So is unformatted I/O of derived types
The same applies in C++, for very similar reasons

- Ask for advice if you need to do these

Namelist

Namelist is a historical oddity, new in Fortran 90
This sounds impossible, but I assure you is true

- Not recommended, but not deprecated, either

STREAM Files

Fortran 2003 has introduced STREAM files
These are for interchange with C–like files
They provide all portable features of C

- They allow positioning, like C text files

I advise avoiding the POS= specifier
It's full of gotchas in both C and Fortran

I/O of Derived Types

The DT descriptor has been mentioned

- Unfortunately, it's often not implemented

You can do almost anything you need to
But this course cannot cover everything

Asynchronous I/O

Mainframes proved that it is the right approach
Fortran 2003 introduced it

- For complicated reasons, you should avoid it

- This has nothing to do with Fortran

Don't use POSIX asynchronous I/O, either
And probably not Microsoft's . . .

BACKSPACE

Don't go there

It was provided for magnetic tape file support
In those days, could often read backwards, too

It's almost always a performance disaster, at best
And it very often doesn't actually work reliably

- Again, that is NOT specific to Fortran

It applies to using seek in C/C++, too
Never reposition on sequential files
Rewinding to the beginning is usually OK

Oddities of Connection

- Try to avoid these, as they are confusing
You will see them in some of the references

Files can be connected but not exist
Ones newly created by OPEN may be like that

Units can be connected when the program starts
Ask me if you want to know why and how

OPEN can be used on an existing connection
It modifies the connection properties

Other Topics

There are a lot more optional features
You must read Fortran's specifications for them

Fortran 2003 adds many slightly useful features
Most compilers don't support many of them yet
The above has described the most useful ones

And a few features should be avoided entirely

For more on this, look at the OldFortran course

Last Reminder

Be careful when using Fortran I/O features
They don't always do what you expect

It is much cleaner than C/POSIX, but . . .

Fortran's model is very unlike C/POSIX's
Fortran's terminology can be very odd

The underlying C/POSIX can show through
In addition to Fortran's own oddities