Fortran Seminar Series

Spring 2017
Overview

• Presented by Mark Branson, Don Dazlich and Ross Heikes

• Presentation materials and sample codes available at the web site:

  kiwi.atmos.colostate.edu/fortran

• Kelley keeps this updated on a weekly basis
Intended Audience

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

This course is intended for all three groups!
Why Fortran?

• Almost **every major model** in atmospheric and oceanic science is still written in Fortran (CAM or CESM, RAMS, WRF, ECMWF’s suite of models, NWP, etc.)

• Fortran has a reputation for being hopelessly out of date (mainly due to Fortran 77?)

• No courses offered except in Meteorology departments
Speed Test

Solve 2D Laplace equation with Jacobi interactive solver

\[ U_{xx} + U_{yy} = 0 \]

using a fourth-order compact finite difference scheme

\[
U_{ij} = \frac{4(U_{i-1,j} + U_{i,j-1} + U_{i+1,j} + U_{i,j+1}) +
U_{i-1,j-1} + U_{i+1,j-1} + U_{i+1,j+1} + U_{i-1,j+1})}{20}
\]
### Dang It’s Fast!

Results with different software (Execution time in seconds)

<table>
<thead>
<tr>
<th>Compiler/Package</th>
<th>n=50</th>
<th>n=100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>46.15</td>
<td>751.78</td>
</tr>
<tr>
<td>NumPy</td>
<td>0.61</td>
<td>6.39</td>
</tr>
<tr>
<td>Matlab</td>
<td>0.64</td>
<td>6.53</td>
</tr>
<tr>
<td>Java</td>
<td>0.12</td>
<td>2.2</td>
</tr>
<tr>
<td>gfortran</td>
<td>0.24</td>
<td>3.25</td>
</tr>
<tr>
<td>ifort</td>
<td>0.052</td>
<td>0.66</td>
</tr>
</tbody>
</table>
Courses

• CSU Atmos used to have a programming course (Fortran/UNIX, IDL and Matlab)
  http://www.atmos.colostate.edu/programming/

• Iowa State has a Fortran/Python course
  http://www.meteor.iastate.edu/classes/mt227/

• Univ of Miami Scientific Programming course
  http://www.rsmas.miami.edu/personal/miskandarani/Courses/MSC321/
Proposed Syllabus

1. Beginnings
2. Data Types and Basic Calculation
3. Control Constructs
4. Array Concepts
5. Subroutines and Functions
6. Modules
7. Parameterized Data Types
Proposed Syllabus (cont.)

8. Input and Output (Don)
9. Derived Types
10. Computer Arithmetic (Don)
11. Make and Makefiles
12. Introduction to Parallel Programming (Ross)
Beginnings

- Fortran does not have a command-line interpreter like **IDL**, **Matlab** or **Python**.
- You need an **editor** to write the code in and a Unix or Linux **shell window** to compile and execute it.
- Each individual will need to determine the **compiler** that's available on their system.
Classes of Language

Interpreted

- Shell
- Perl
- Python

Compiled

- Java
- C,C++,Fortran

Fortran is the best choice for pure number crunching!
FORmula TRANslator invented 1954-8 by John Backus and his team and IBM

general purpose programming language mainly intended for mathematical computations in engineering

first-ever high-level programming language using the first compiler ever developed
History (2)

FORTRAN 66 (ISO Standard 1972)

FORTRAN 77 (1980) \{ HUGE TRANSITION! \}

Fortran 90 (1991)

Fortran 95 (1996)


Fortran 2008 (2010)

Fortran 2015 (ongoing)
Disclaimer

This course will cover modern, free-format Fortran only!

• Don’t want to teach newcomers “old” fortran.

• At the same time almost all of you already have or will encounter your fair share of legacy Fortran codes.

• Almost all old Fortran remains legal.
Hardware and Software

A system is built from **hardware** and **software**

The **hardware** is the physical medium

- CPU, memory, keyboard, display

The **software** is a set of computer programs

- operating system, compilers, editors
- Fortran programs
Programs

Fortran 90 is a high-level language

Uses English-like words and math expressions

\[ Y = X + 3 \]

PRINT *, Y

Compilers translate into machine instructions

A linker then creates an executable program

The operating system runs the executable
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 known as 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 only.

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 in 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. Specification part
   
   Declare types and sizes of data

3. - 6. Execution part
   
   All of the “action” statements

7. End of program (or procedure)
   
   END PROGRAM example1
Program and File Names

- The program and file names are NOT related. PROGRAM QES can be in the file QuadSolver.f90

Some implementations like the same names, sometimes converted to lower- or upper-case.

The compiler documentation should tell you!
The Specification Part

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)

The output data are ‘Time... =’ and temp

They can be any expression not just a variable
The Execution Part

Write prompt to display

PRINT *, ‘Type the hours, …’

Read the time in hours, minutes and seconds

READ *, hours, mins, secs

Convert the time into seconds

\[
\text{temp} = 60 \times (\text{hours} \times 60 + \text{mins}) + \text{sec}
\]

Write out the number of seconds

PRINT *, ‘Time in seconds = ‘, temp
Compiling and Executing

Compile your program into an executable:

```
f90 [-o exename] program_name.f90
```

where

```
f90 = name of your compiler (f90, ifort, gfortran, g90, etc.)
```

If you do not specify an executable most systems will use `a.out` by default.
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 can deliver a value
There are four main steps:

1. Specify the problem
2. Analyze and subdivide into tasks
3. Write the Fortran 90 code
4. Compile and run (testing phase)

Each step may require several iterations. You may need to restart from an earlier step. The testing phase is very important.
Errors

• ALWAYS keep in mind the **golden rule:**

  **Computers ONLY do what you tell them to do.**

  • If something is wrong, it’s probably your own fault. I’m sorry, but it is.

  • Corollary: Sometimes you don’t know that you told the computer to do it wrong, OR somebody else did the telling.
Errors

- If the **syntax** is incorrect, the compiler says so
  
  ```
  INTEGER :: mins, secs
  ```

- If the action is **invalid**, things are messier
  
  ```
  X / Y when Y is zero
  ```

  Error message at run-time **OR**
  
  Program may crash or hang or produce nonsense values
Fortran Language Rules

• This course is modern, free-format source only

• Almost all old Fortran remains legal BUT you should avoid using it as modern Fortran is better
Important Warning

- Fortran **syntax** (the arrangement of words and phrases) is verbose and horrible. It can fairly be described as a historical mess.
- Fortran **semantics** (the mean of words, phrases, or text) are fairly clean and consistent.
- Verbosity causes problems for examples. Many use poor style to be readable, lack error checking.

- **DO WHAT I SAY NOT WHAT I DO**
Correctness

Humans understand language quite well even when it is not strictly correct.

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 behavior”).
Examples of Errors

Consider \((N\times M/1024+5)\)

If you mistype the ‘0’ as a ‘)’: \((N\times M/1)24+5\)

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 ‘-’: \((N\times M/1-24+5)\)

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

_special_ = + - * / ( ) , . ’ : ! ” % & ; < > ? $ 

slash (/) is also used for divide
hyphen (-) is also used for minus
asterisk (*) is also used for multiply
apostrophe (‘) is also used for single quote
period (.) is also used for decimal point

The others are described when we use them.
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:
  ENDDDO is the same as END DO
  But EN  DDO is treated as EN and DDO
Source Form (2)

- Do not run keywords and names together
  PROGRAMMyPROG - illegal
  PROGRAM MyPROG - 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 as well as lines consisting only of comments
Lines and Comments

A line is a sequence of up to 132 characters.

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

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.

1.0e6 is clearer than 1.e6 or .1e7
Use of Comments

• Appropriate commenting is very important.
• Document assumptions that may break later.
• Also helps to remind you to not make the same mistake twice!
• Good commenting can slow coding by 25% BUT it really speeds up initial debugging!
• Overall in research it repays itself 3:1. Can be 10:1 for production codes.
Use of Case

• It doesn’t matter which case convention you use BUT do try to be moderately consistent.

• Very important for clarity and editing/searching.

• One possible convention:
  • UPPER case for keywords
  • Lower case for names
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)

Semi-colons can be used to put multiple statements on the same line:

\[
a = 3 ; \quad b = 4 ; \quad c = 5
\]

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

Avoid mixing continuation with that and comments. It is legal but makes code VERY hard to read.

\[
a = b + c ; \quad d = e + f + & \quad g + h \\
a = b + c + & ! \text{ More coming...}
\]
Breaking Character Strings

Continuation lines can start with an & Preceding spaces and the & are suppressed.

The following works and allows indentation:

```plaintext
PRINT *, 'Assume that this string &
is far too long and complicated &
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 letters, digits and underscores.
• Names must start with a letter.
• Upper and lower case are equivalent.

DEPTH, Depth and depth are all the same.
• The following are valid fortran names:

A, AA, aaa, Tax, INCOME, Num1, Num2, NUM333, N12MO5, atmospheric_pressure, Line_Color, 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:  contains illegal characters ‘.’ and ‘:’
Data Types and Basic Calculation
Intrinsic Data Types

Fortran supports five intrinsic data types:

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

2. **REAL** for approximate, **fractional numbers**
   e.g., 1.1, 3.0, 23.565, 3.1415, \( \exp(1) \), etc.

3. **COMPLEX** for complex, **fractional numbers**
   e.g., \((1.1, -23.565)\), etc.
4. **LOGICAL** for truth values (boolean)  
   These may only have values of true or false  
   e.g., `.TRUE.`, `.FALSE.`

5. **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 special character type (unlike C).
Integers (1)

- Integers are restricted to lie in a finite range.
  Typically $\pm 2\,147\,483\,647$ ($-2^{31}$ 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 $\pm 32\,768$ ($-2^{15}$ to $2^{15} - 1$)

- More on 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**

Example: Calculations involving **counts** (or money)
Reals

- **Reals** are used for continuously varying values.
- **Reals** are stored as floating-point values. They also have a finite range and precision.

**THEY ARE INEXACT**

- It is essential to use floating-point appropriately.
Floating Point Basics

• One key fundamental: Floating point on computers is usually base-2 whereas the external representation is base-10.

• Most floating point numbers can be represented as $1.fffffff \times 2^n$ where
  • 1 is the integer bit
  • the fs are the fractional bits
  • n is the exponent

• Base-2 arithmetic is so much faster than base-10 on digital computers.
Floating Point Standard

• The Institute of Electrical and Electronics Engineers (IEEE) has produced a standard for floating point arithmetic. IEEE 754-1985.

• This defines 32-bit and 64-bit floating point representations.

• 32-bit: $10^{-38}$ to $10^{+38}$ and 6-7 decimal places

• 64-bit: $10^{-308}$ to $10^{+308}$ and 15-16 decimal places
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 by E or D and an exponent
  1.0D6, 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)

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**
Names starting with **A-H** and **O-Z** are **REAL**
YOU SHOULD ALWAYS USE IMPLICIT NONE
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:

xyMax = 2 * xyMin
mySum = mySum + Term1 + Term2 + (Eps * Err)
There are five built-in numeric operations:

- addition
- subtraction
- multiplication
- division
- exponentiation

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 + 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
- Exponentiation binds from right to left

The precedence from highest to lowest is:

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

Parentheses are used to control it. Use them whenever the order matters or it is clearer.
Examples

\[ X + Y \times Z \quad \text{is equivalent to} \quad X + (Y \times Z) \]
\[ X + Y / 7.0 \quad \text{is equivalent to} \quad X + (Y / 7.0) \]
\[ A - B + C \quad \text{is equivalent to} \quad (A - B) + C \]
\[ A + B \times C \quad \text{is equivalent to} \quad A + (B \times C) \]
\[ -A \times 2 \quad \text{is equivalent to} \quad -(A \times 2) \]
\[ A - (((B + C))) \quad \text{is equivalent to} \quad A - (B + C) \]

You can force any order you like:

\[ (X + Y) \times Z \]

Adds \( X \) to \( Y \) and then multiplies by \( Z \)
Warning

\[ X + Y + Z \] may be evaluated as any of
\[ X + (Y + Z) \text{ or } (X + Y) + Z \text{ or } Y + (X + Z) \text{ or } \ldots \]

Fortran defines \textbf{what} an expression means
It does not define \textbf{how} it is calculated

The are all \textbf{mathematically} equivalent
But may sometimes give slightly different results
Integer Expressions

Expressions involving integer constants and variables

These are evaluated in integer arithmetic. Division always truncates toward zero.

\[
\text{INTEGER :: } K, L, N \\
N = K + L/2 \\
\text{If } K = 4 \text{ and } L = 5 \text{ then } N = 6
\]

\[
(-7)/3 \text{ and } 7/(-3) \text{ are both } -2
\]
Mixed Expressions

In the CPU calculations must be performed between objects of the same **type**, so if an expression mixes type some objects must change type.

**Default types** have an implied ordering:

1. INTEGER (**lowest**)
2. REAL
3. COMPLEX (**highest**)

The result of an expression is always of the **highest** type. e.g., INTEGER * REAL gives a REAL

Be careful with this as it can be deceptive!
Conversions

There are several ways to force conversion

- **Intrinsic functions** INT, REAL and COMPLEX
  
  \[
  X = X + \text{REAL}(K)/2 \\
  N = 100 \times \text{INT}(X/1.25) + 25
  \]

- Use the appropriate constants. (You can even add zero or multiply by one

  \[
  X = X + K/2.0 \\
  X = X + (K+0.0)/2
  \]

The second method isn’t very nice but works well enough. (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 toward zero

Similar remarks apply to COMPLEX

The RHS is evaluated independently of the LHS

Example: mixedassigned.f90
Intrinsic Functions

Built-in functions that are always available
- No declaration is needed -- just use them!

Examples:

\[
\begin{align*}
Y &= \text{SQRT}(X) \\
\pi &= 4.0 \times \text{ATAN}(1.0) \\
Z &= \text{EXP}(3.0 \times Y) \\
X &= \text{REAL}(N) \\
N &= \text{INT}(X) \\
Y &= \text{SQRT}(-2.0 \times \text{LOG}(X))
\end{align*}
\]
Intrinsic Numeric Functions

REAL(n) ! Converts its argument 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 of x
LOG(x) ! The natural logarithm of x
LOG10(x) ! The base 10 logarithm of x

SIN(x) ! The sine of x (x in radians)
COS(x) ! The cosine of x (x in radians)
TAN(x) ! The tangent of x (x in radians)
ASIN(x) ! The arc sine of x (x in radians)
ACOS(x) ! The arc cosine of x (x in radians)
ATAN(x) ! The arc tangent of x (x in radians)
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.
ELSE IF (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 to
`>` (or `.GT.`) greater than
`>=` (or `.GE.`) greater than or equal to
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.` can be used instead of `==`

`.NEQV.` can 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 behavior is up to the compiler
One of the reasons that it is so optimizable
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:

In ‘Time^^=^^13:15’, with ‘^’ being a space. The character string is of length 14, Character 1 is T, 8 is a space, 10 is I, etc.

Example program: charstrings.f90
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
address is of length 32
Another Form

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

While this form is historical it is more compact

Don’t mix the forms -- that is an abomination
CHARACTER(LEN=10) :: dept, faculty, addr*32

For some obscure reasons using LEN= is cleaner
It avoids some arcane syntactic “gotchas”
Character Assignment

CHARACTER(LEN=6) :: firstname, lastname
firstname = 'Mark' ; lastname = 'Branson'

firstname is padded with spaces ('Mark^^')
lastname is truncated to fit ('Branso')

Unfortunately you won't get told
But at least it won't overwrite something else
Values may be joined using the // operator

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

Sets `identity` to ‘THOMAS’
But `Z` is set to ‘TH’ – why?

// does not remove trailing spaces
It used 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*15, lastname*7, title*3
  name = ‘Mr. Joe Johnson’
  title = name(1:3)
  lastname = name(9:)

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 10th character

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

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 = ‘ Smith ‘
newname = TRIM(ADJUSTL(name))

newname would contain ‘Smith’

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 “fred” < “Fred” or not

Fortran is not a locale-based language
It specifies only the following

‘A’ < ‘B’ < ‘C’ < ... < ‘Y’ < ‘Z’
‘a’ < ‘b’ < ‘c’ < ... < ‘y’ < ‘z’
‘0’ < ‘1’ < ‘2’ < ... < ‘8’ < ‘9’
‘ ’ is less than all of ‘A’, ‘a’ and ‘0’

A shorter operand is extended with blanks (‘ ‘)
It avoids some arcane syntactic “gotchas”
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 equations much clearer
Much clearer which constant is being used

They make it easier to modify the program later
INTEGER, PARAMETER :: MAX_DIMENSION = 10000
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 = 3.1415, &
    pi_by_4 = pi/4, two_pi = 2*pi

CHARACTER(LEN=*), PARAMETER :: &
    all_names = ‘Bob, Jennifer, Karen’, &
    karen = all_names(16:20)

Generally anything reasonable is allowed
It must be determinable at compile time
Initialization

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

**Initialization** is much 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.
```
Control Constructs
Control Constructs

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

The main ones are:
- Conditionals (IF etc.)
- Loop (DO etc.)
- Switches (SELECT/CASE etc.)

Loops are by far the most complicated.
The oldest and the simplest is the single statement IF

\[ \text{IF (logical expression) simple statement} \]

If the logical expression is \texttt{.True.} then the simple statement is executed.

If the logical expression is \texttt{.False.} then the whole statement has no effect.
Single Statement IF (2)

Some examples:

IF (X < A) X = A

IF (INT(a*b-c) <= 47) mytest = .true.

IF (MOD(Cnt,10) == 0) WRITE(*,*) CNT

Unsuitable for anything complicated.

Only action statements (assignment, input/output) can be used. Nothing complicated like another IF statement or anything containing blocks.
A block IF statement is much more flexible.

Here is the most traditional form of it:

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

If the expr is `.TRUE.` then the first block is executed.
If not, the second block is executed.

`ENDIF` or `END IF` can be used.
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)
ENDIF
Omitting the ELSE

The **ELSE** and its block can also be omitted.

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

IF (name(1:4) == "Miss" .OR. &
    name(1:4) == "Mrs.") THEN
  name(1:3) = "Ms."
  name(4:) = name(5:)
ENDIF
```
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 ! This third
    X = B + (C-B)*(X-2.0)
ELSE ! This is used if none succeed
    X = C
ENDIF
Including ELSE IF Blocks

(2)

• You can have as many ELSE IFs as you wish
• There is only one ENDIF for the whole block
• All ELSE IFs must come before any ELSE
• They are checked in order and the first success is taken
• You can omit the ELSE in these constructs
• ELSE IF can also be spelled ELSEIF
Named IF Statements (1)

The IF can be preceded by <name>:
And the END IF followed by <name>:
And any ELSE IF / THEN and ELSE may be

```
myifblock: IF (X < 0.0) THEN
    X = A
ELSE IF (X < 2.0) THEN myifblock
    X = A + (B-A)*(X-1.0)
ELSE    myifblock
    X = C
ENDIF myifblock
```
Named IF Statements (2)

The **IF construct name** must match and be distinct
Can be a great help for checking and clarity
You should name at least all long **IFs**

If you don’t nest **IFs** that much this style is fine:

```plaintext
myifblock: IF (X < 0.0) THEN
    X = A
ELSE IF (X < 2.0) THEN
    X = A + (B-A)*(X-1.0)
ELSE
    X = C
ENDIF myifblock
```
Block Contents

• Almost any executable statements are okay
  Both kinds of IF, complete loops, etc.
  You may never notice the few restrictions

• This applies to all of the block statements
  IF, DO, SELECT, etc.

• 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
        ...
    ENDIF
ELSE IF (state == 2) THEN phasetest
    IF (phase > pi) PRINT *, 'A bit odd here'
ELSE phasetest
    IF (phase < pi) THEN
        ...
    ENDIF
ENDIF

ENDIF
An alternative to the IF block for selective execution is the SELECT CASE statement. Can be used if the selection criteria are based on simple values in INTEGER, LOGICAL and CHARACTER.

It provides a streamlined syntax for an important special case of a multiway selection.
The **basic format** is:

```
SELECT CASE ( <selector> )
    CASE (label-list-1)
        statements-1
    CASE (label-list-2)
        statements-2
    CASE (label-list-n)
        statements-n
    CASE DEFAULT
        statements-default
END SELECT
```
The label-list can take one of many forms:

- **val** → a specific value
- **val1, val2, val3** → a specific set of values
- **val1:val2** → values between **val1** and **val2** inclusive
- **val1:** → values larger than or equal to **val1**
- **:** **val2** → values less than or equal to **val2**

**val**, **val1**, and **val2** must be **constants** or **parameters**!

Example: **select_example.f90**
Some important notes:

- The values in the label-lists should be unique. Otherwise you will get a compilation error.

- **CASE DEFAULT** should be used if possible as it guarantees that a match will be found even if it is an error condition.

- Technically the **CASE DEFAULT** can be placed anywhere within the **SELECT CASE** statement but the preferred position is at the bottom.
DO Construct

The loop construct in Fortran is known as the do loop. The basic syntax is:

```
[ loop name ] DO [ loop control ]
block of statements
END DO [ loop name ]
```

- loop name and loop control are optional
- With no loop control it loops indefinitely
- **END DO** or **ENDDO** can be used.
Indexed DO Loop (1)

This is the most common form.

```plaintext
DO <control-var> = <initial>, <final> [,<step>]  
block of statements
END DO
```

- `<control var>` is an integer variable.
- `<initial>`, `<final>` and `<step>` are integer expressions.
- If `<step>` is omitted its default value is 1.
- `<step>` cannot be zero.
Indexed DO Loop (2)

If `<step>` is positive:

- `<control-var>` receives the value of `<initial>`.
- If the value of `<control-var>` is less than or equal to `<final>`, the block of statements contained within the loop are executed.
- Then the value of `<control-var>` is iterated by `<step>` and compared to `<final>`.
- When the value of `<control-var>` exceeds the value of `<final>` execution moves below the END DO.
Indexed DO Loop (3)

If `<step>` is negative:

- `<control-var>` receives the value of `<initial>`.
- If the value of `<control-var>` is greater than or equal to `<final>`, the block of statements contained within the loop are executed.
- Then the value of `<control-var>` is iterated by `<step>` and compared to `<final>`.
- When the value of `<control-var>` is less than the value of `<final>` execution moves below the END DO.
Indexed DO Loop (4)

Important notes:

• `<step>` cannot be zero.

• Before the loop starts the values of `<initial>`, `<final>` and `<step>` are evaluated exactly once. i.e., these values are never re-evaluated as the loop executes.

• Never attempt to change the values of `<control-var>`, `<initial>`, `<final>` or `<step>`.

• Don’t use real variables for the loop expressions.

• Examples: `simpleloop.f90`
Non-Indexed DO Loop

We can omit the loop control but then we need a way to exit the loop.

- The `EXIT` statement brings the flow of control to the statement following the `END DO`.
- The `CYCLE` statement starts the next iteration.
- Examples: `exitloop.f90`
WHILE Loop

The **WHILE loop control** has the following form:

```
DO WHILE ( <logical expression> )
.
END DO
```

- The **logical expression** is reevaluated for each cycle.
- The loop exits as soon as it becomes `.FALSE.`.
- It’s actually a redundant feature as the same thing can be accomplished with an **EXIT** statement.
- Examples: `whileloop.f90`
CONTINUE Statement

CONTINUE is a statement that does nothing
Used to be fairly common particularly before END DO came along but now it is rare.

It’s mainly a placeholder for labels
This is purely to make the code clearer

It can be used anywhere a statement can.
RETURN and STOP

RETURN causes a procedure to halt execution with control given back to the calling program.

STOP halts execution cleanly. Typically used with an IF statement to stop the program if some error condition is encountered.
Array Concepts
Array Declarations (1)

Fortran 90 uses the **DIMENSION** attribute to declare arrays. The most common examples are:

```
INTEGER, DIMENSION(30) :: days_in_month
CHARACTER(LEN=10), DIMENSION(250) :: names
REAL, DIMENSION(350,350) :: box_locations
```

In Fortran the **starting index** defaults to a value of 1 (not 0 as is common in many other languages - C/C++/Python)
Array Declarations (2)

BUT you can specify a lower bound different than 1. It will just default to 1 if you omit it.

The syntax is `<lower bound>:<upper bound>` where the bound values are INTEGERs.

INTEGER, DIMENSION(0:99) :: arr1, arr2, arr3
CHARACTER(LEN=10), DIMENSION(1:250) :: names
REAL, DIMENSION(-10:10,-10:10) :: pos1, pos2
REAL, DIMENSION(0:5,1:7,2:9,1:4,-5:-2) :: pos1, pos2
REAL :: A(0:99), B(3,6:9,5)

- The **rank** of an array is the number of dimensions. 
  The **maximum** number of dimensions is **7**!
  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 **bounds 1:3, 6:9** and **1:5**

- The **extent** of an array dimension is range of its index.
REAL :: A(0:99), B(3,6:9:5)

- The **size** of an array is the total number of elements.
  
  A has size **100** and B has size **60**

- The **shape** of an array is its **rank** and **extents**.
  
  A has shape **(100)** and B has shape **(3,4,5)**

Arrays are **conformable** if they share the same **shape**. The **bounds** do not have to be the same.
Array References

In general, there are three different ways to reference arrays:

• **individual** array elements [arr1(5), myintval(-10)]
• **entire** array [arr1 or arr1(:)]
• **array section** [arr1(5:24), arr1(-10:-7)]
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

Set the even elements to the odd indices and vice versa

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

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 $O(N \log(N))$.

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
Using Arrays as Objects

Set all the **elements** of an array to a single value

```plaintext
INTEGER, DIMENSION(1:50) :: series
series = 0
```

You can use entire arrays as simple variables provided they are **conformable**

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

The **RHS** and any **LHS** indices are evaluated, and then the **RHS** is assigned to the **LHS**.
Array sections create an aliased subarray. It is a simple variable with a value:

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

Even this is legal but it forces a **copy**:

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

A(1:6, 1:8)

A(1:3, 1:4)

A(2:5, 7)
Short Form

Existing array bounds may be omitted
Especially useful for multidimensional arrays

If we have \texttt{REAL, DIMENSION(1:6, 1:8) :: A}

\texttt{A(3:, :4)} is the same as \texttt{A(3:6, 1:4)}
\texttt{A, A(:,:) and A(1:6, 1:8)}

\texttt{A(6, :) is the 6th row as a 1-D vector}
\texttt{A(:, 3) is the 3rd column as a 1-D vector}
\texttt{A(6:6, :) is the 6th row as a 1x8 matrix}
\texttt{A(:, 3:3) is the 3rd columns as a 6x1 matrix}
Conformability of Sections

The **conformability** rule applies to sections, too.

```plaintext
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)
```
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

```fortran
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

**A(1:6, 1:8)**

- **A(:3, 1:5:2)**
- **A(2:6:2, 7)**
Subscripts and sections must be within the array bounds. The following are invalid (undefined behavior):

- 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,:)

Most compilers will NOT check for this automatically! Errors will lead to overwriting, etc. and CHAOS.
Most built-in operators/functions are elemental. They act element-by-element on arrays.

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

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, the compute for that dimension only. And the result is an integer scalar. Otherwise, the result 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 the compute for that dimension only

TRANSPOSE(x) means $X_{ij} \Rightarrow X_{ji}$

It must have two dimensions but need not be square

DOT_PRODUCT(x,y) means $\sum_i X_i \cdot Y_i \Rightarrow Z$

Two vectors, both of same length and type
Array Intrinsic Functions (4)

MATMUL(x,y) means \( \sum_k X_{ik} \cdot Y_{kj} \Rightarrow Z_{ij} \)

2nd dimension of \( X \) must match the 1st of \( Y \)
The matrices need not be the same shape
Either \( X \) or \( Y \) may be a vector

Many more for array reshaping and array masking
Array Element Order (1)

This is also called the “storage order”

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

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

The elements of A are stored in this 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, IDL, 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**

• Optimization (caching and locality)
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:

READ *, oxo(1,1), oxo(2,1), oxo(3,1), & oxo(1,2), oxo(2,2), oxo(3,2)
Array sections can also be used

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

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

This last statement equivalent to:

```fortran
READ *, oxo(1,3), oxo(2,3), oxo(3,3), &
        oxo(3,1), oxo(2,1), oxo(1,1)
```
Array Constructors (1)

Commonly used for assigning array values
An array constructor will create a temporary array

\[
\text{INTEGER, DIMENSION}(6) :: \text{marks} \\
\text{marks} = (\text{/} 10, 25, 32, 54, 56, 60 \text{/})
\]

Constructs an array with the elements
10, 25, 32, 54, 56, 60
And then copies that array into marks

Fortran 2003 addition: Also can use square brackets

\[
\text{marks} = [10, 25, 32, 54, 56, 60]
\]
Array Constructors (2)

Variable expressions are okay in constructors

\[ \text{marks} = (\ x, \ 2.0*y, \ \sin(t*w/3.0), \ ... \ /) \]

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

All expressions must be the same type
This can be relaxed in Fortran 2003
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 5:

\[
\text{marks} = (\langle 0.0, (k/10.0, k=2, n), 1.0 \rangle)
\]

This is equivalent to:

\[
\text{marks} = (\langle 0.0, 0.2, 0.3, 0.4, 0.5, 1.0 \rangle)
\]
Array constructors can be very useful for this. All elements must be initialization expressions, i.e., ones that can be evaluated at compile time.

For rank one arrays just use a constructor:

```plaintext
REAL, PARAMETER :: a(3) = (/ 1.23, 4.56, 7.89 /)
REAL :: b(3) = (/ 1.23, 4.56, 7.89 /

b = exp(b)
```
Other types can be initialized in the same way:

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

Initialization expressions are allowed:

```
INTEGER, PARAMETER :: N = 3, M = 6, P = 12
INTEGER :: arr(3) = (/(N, (M/N), (P/N)/))
```
What about this?

```fortran
REAL :: arr(3) = (/ 1.0, exp(1.0), exp(2.0) /)
```

Fortran 90 does **NOT** allow this but Fortran 2003 does

Not just **intrinsic functions** but all sorts of things
Multiple Dimensions

Constructors cannot be nested - e.g., NOT:

```
REAL, DIMENSION(3,4) :: xvals = &
(/ (/ 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

Use the RESHAPE intrinsic function to construct higher rank arrays. We’ll cover this later if time permits.
Allocatable Arrays (1)

Arrays can be declared with an **unknown shape**
Use the **ALLOCATABLE** attribute in the type declaration

```
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))
```

You can also allocate multiple arrays in a single
ALLOCATE statement
Allocatable Arrays (2)

Failures 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
...
ENDIF
```

Arrays can be deallocated using

```
DEALLOCATE(counts)
```
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 /= 0.0) THEN
   PRINT *, ‘Error in allocate statement’
ENDIF
WHERE Construct (1)

Used for **masked array assignment**

Example: Set all negative elements of an array to zero

```
REAL, DIMENSION(20,30) :: array

DO j = 1,30
  DO k = 1,20
    IF (array(i,j) < 0.0) array(k,j) = 0.0
  ENDDO
ENDDO
```

But the WHERE statement is much more convenient

```
WHERE (array < 0.0) array = 0.0
```
WHERE Construct (2)

It has a statement construct form, too
Example: Set all negative elements of an array to zero

```
WHERE (array < 0.0)
array = 0.0
ELSE WHERE
array = 0.01 * array
ENDWHERE
```

Masking expressions are LOGICAL arrays
You can use an actual array there, if you want
Masks and assignments need the same shape
Subroutines and Functions
Subdividing the Problem

• Most problems are thousands of lines of code. Few people can grasp all of the details.

• Good design principle: Exhibit the overall structure in the main program and put the details into subroutines and functions.

• You often use similar code in several places.

• You often want to test only 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

function:
- Purpose is to return a single result
- Invoked by inserting the function name
- It is called only when its result is needed

subroutine:
- May or may not return result(s)
- Invoked with the CALL statement

Example: sort3.f90, sort3a.f90, sort3b.f90
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 declaration of its dummy arguments
• And anything that use those (see later)

SUBROUTINE Sortit(array)
INTEGER :: [temp, ] array(:) [, J, K]
Structure and Syntax

Subroutine syntax:

```
SUBROUTINE subroutine-name(arg1, arg2,...,argn)
IMPLICIT NONE
[specification part]
[execution part]
END SUBROUTINE subroutine-name
```

If the subroutine does not require any arguments, the `(arg1, arg2,...,argn)` can be omitted.

Similar syntax is used for functions.
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

In general, dummy and actual argument lists must match:

• The number of arguments must be the same
• Each argument must match in type and rank

These can be relaxed in some cases.

Most of the complexities involve array arguments.
Often the required result is a single value (or array) In that case it makes more sense to write a function

Function syntax:

\[
\text{type FUNCTION funct-name} (\text{arg1}, \ldots, \text{argn}) [\text{result return-value-name}]
\]

\[
\text{IMPLICIT NONE}
\]

\[
[\text{specification part}]
\]

\[
[\text{execution part}]
\]

\[
\text{END FUNCTION funct-name}
\]
Functions (2)

• If a result variable is not specifically defined then the result is returned through the function name.

• The result variable must be declared in the function’s specification area.

• You can optionally specify the type of the function:

  REAL FUNCTION VARIANCE(array)

  • If this is done, no local declaration is needed.

• Example: variance.f90, series.f90
Usage

How do we incorporate subroutines and functions into our code?

1. Attach them to a main program as internal procedures using the CONTAINS statement

2. Include them in a MODULE (also with CONTAINS)

Legacy Fortran had to use external procedures. I will show you why these are a BAD IDEA
Internal Procedures (1)

For relatively small programs you can include procedures in the main program using CONTAINS

- You can include any number of procedures
- Visible to the outer program only
- These internal subprograms may not contain their own internal subprograms
Internal Procedures (2)

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

- All of the local declarations
- 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

Examples: `checkarg_int.f90`, `checkarg_ext.f90`
Internal Procedures (3)

A local name takes precedence

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

This will print 4.56, not 1.23
Avoid doing this as it’s very confusing
Module Procedures

You can also place procedures in a module using a `CONTAINS` statement

• Module `internal subprograms` may contain their own internal subprograms

• Module name need not be the same as the file name but for large programs that is highly recommended

• Include the module with the `USE` statement

Example: `checkarg_mod.f90`, etc.
You can make arguments **read-only**

SUBROUTINE Summarize(array, size)
IMPLICIT NONE
INTEGER, INTENT(IN) :: size
REAL, DIMENSION(size) :: array

Will prevent you from writing to a variable by accident
Or calling another procedure that does that
May also help the compiler to optimize

**Strongly** recommended for **read-only** arguments
Intent (2)

You can also make arguments write-only
Less useful but still worthwhile

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

As useful for optimization as INTENT(IN)
Intent (3)

The default is effectively \texttt{INTENT(INOUT)}
Specifying it can be useful as it can catch certain errors

\begin{verbatim}
SUBROUTINE Munge(value)
  REAL, INTENT(INOUT) :: value
  value = 100.0 * value
END SUBROUTINE Munge

CALL Munge(1.23)
\end{verbatim}

This would be okay:
\begin{verbatim}
x = 1.23
CALL Munge(x)
\end{verbatim}
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
Keyword Arguments

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

Keywords are NOT names in the calling procedure. They are only used to map dummy arguments.

Example: series2.f90
Optional Arguments

Use **OPTIONAL** for setting **defaults** only
Check for existence using **PRESENT** function
Use **only** local copies thereafter
That way all variables will be well-defined when used

Example: **series3.f90**
The best way to declare array arguments
Simply specify all bounds with 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

  ...

  REAL, DIMENSION(1000) :: one
  REAL, DIMENSION(100,100) :: two

  CALL peculiar(one, two)
  CALL peculiar(one(101:160), two(21:,26:75))

In the second call vector will be dimensioned (1:60) and matrix will be dimensioned (1:80, 1:50)
Assumed Shape Arrays (2)

Array query functions were described earlier:
SIZE, SHAPE, LBOUND, UBOUND

Gives the ability to write completely generic procedures:

```fortran
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
```
Setting Lower Bounds

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

SUBROUTINE peculiar(vector, matrix,n)
  REAL, DIMENSION(2*n+1:) :: vector
  REAL, DIMENSION(0:,0:) :: matrix
Local arrays with **bounds** specified at **run-time** are called automatic arrays.

Bounds may be taken from an **argument**, or a **constant** or **variable** in a **module**.

```fortran
SUBROUTINE aardvark (arrsize)
    USE sizemod    ! this defines the var “worksize”
    INTEGER, INTENT(IN) :: arrsize
    REAL, DIMENSION(1:worksize) :: array_1
    REAL, DIMENSION(1:arrsize*(arrsize+1)) :: array_2
```
Another very common use is a “shadow” array i.e., one that is the same shape as an argument

```fortran
SUBROUTINE swap_arrays (A, B)
   REAL, DIMENSION(:) :: A, B
   REAL, DIMENSION(SIZE(A)) :: temp
   temp = A ; A = B ; B = temp
END SUBROUTINE swap_arrays
```
Automatic Arrays (3)

Multi-dimensional example of the same concept:

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

Automatic arrays are very flexible.
Explicit Shape Array Args

(1)

We cover these because of their importance. They were the only mechanism available in Fortran 77. 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.
You can use constants

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

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

   CALL expl_shape(table, workspace)
```
Explicit Shape Array Args

It is common to pass the bounds as arguments.

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

You can use expressions but it’s not generally recommended.
```
WARNING

Argument overlap will NOT be detected
Not even if you turn on array-bounds checking
This is a common cause of obscure errors

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

Example: overlap.f90
Character Arguments

Few scientists do anything fancy with these

People often use a constant length
You can specify this as a digit string
OR define it as a PARAMETER
That is best done in a module

Or define it as an assumed length argument
The *dummy* should match the *actual argument*.
You are likely to get confused if it doesn’t.

```fortran
SUBROUTINE sorter (list)
  CHARACTER(LEN=8), DIMENSION(:) :: list
  ...
  CHARACTER(LEN=8) :: data(1000)
  ...
  CALL sorter(data)
```
MODULE Constants
   INTEGER, PARAMETER :: charlen=8
END MODULE Constants

SUBROUTINE sorter (list)
   USE Constants
   CHARACTER(LEN=charlen), DIMENSION(:) :: list

   USE Constants
   CHARACTER(LEN=charlen) :: data(1000)
   CALL sorter(data)
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

FUNCTION is_palindrome(word)
LOGICAL :: is_palindrome
CHARACTER(LEN=*) :: word
Static Data

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

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

The best style avoids this use.

Example: localsave.f90
Modules, Make and Interfaces
Module Summary

• Similar to same term used in other languages. As usual, modules fulfill multiple purposes

• For shared declarations (i.e., “headers”)

• Defining global data (old COMMON)

• Defining procedure interfaces

• Semantic extension (described later)

And more...
Use of Modules

- Think of a module as a high-level interface. It collects <whatevers> into a coherent unit.
- Design your modules carefully. As the ultimate top-level program structure, perhaps only a few, perhaps dozens.
- Good place for high-level comments. Please document purpose and interfaces.
Module Structure

**MODULE** `module-name`

- Static data definitions (often exported)
- **CONTAINS**
  - Procedure definitions and interfaces

**END MODULE** `module-name`

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

For simplest use, keep them 1 to 1
Modules should also use this important specification

```fortran
MODULE double
  IMPLICIT NONE
  INTEGER, PARAMETER :: DP = KIND(0.0D0)
END MODULE double

MODULE parameters
  USE double
  IMPLICIT NONE
  REAL(KIND=DP), PARAMETER :: one = 1.0_DP
END MODULE parameters
```
Module Interactions

Modules can **USE** other modules

Dependency graph shows **visibility/usage**

**Modules** may not depend on themselves

i.e., the standard does not permit the recursive or circular use of modules

```
MODULE A
    USE B
END MODULE A

MODULE B
    USE A
    USE A
END MODULE B
```
MODULE double
  INTEGER, PARAMETER :: DP = KIND(0.0D0)
END MODULE double

MODULE parameters
  USE double
  REAL(KIND=DP), PARAMETER :: one = 1.0_DP
  INTEGER, PARAMETER :: nx = 10, ny = 25
END MODULE parameters

MODULE workspace
  USE double
  USE parameters
  REAL(KIND=DP), DIMENSION(nx,ny) :: now, then
END MODULE workspace
The main program might look like this

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

Could omit the USE double and USE parameters as they would be inherited through USE workspace
Module Dependencies

- double
- parameters
- workspace
- main program
Shared Constants

We have already seen and used this:

```fortran
MODULE double
    INTEGER, PARAMETER :: DP = KIND(0.0D0)
END MODULE double
```

You can do a great deal of this sort of thing

Greatly improves *clarity* and *maintainability*

The larger the program, the more it helps

Example from the CAM:  `shr_const_mod.F90`
Derived Type Definitions

We shall cover these later:

```
MODULE Bicycle
  REAL, PARAMETER :: pi = 3.141592
  TYPE Wheel
    INTEGER :: spokes
    REAL :: diameter, width
    CHARACTER(LEN=15) :: material
  END TYPE Wheel
END MODULE Bicycle
```

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

Variables in modules define global data. These can be fixed-size or allocatable arrays.

- You need to specify the SAVE attribute.
  Set automatically for initialized variables.
  But it is good practice to do it explicitly.

A simple SAVE statement saves everything.
- This isn’t always the best thing to do.
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:

```fortran
MODULE state_variables
  IMPLICIT NONE
  SAVE
  INTEGER, PARAMETER :: nx=100, ny=100
  REAL, DIMENSION(NX,NY) :: &
    current, increment, values
  REAL :: time = 0.0
END MODULE state_variables
```
Example (3)

The arrays sizes do not have to be fixed:

```fortran
MODULE state_variables
  REAL, DIMENSION(:,,:), ALLOCATABLE, SAVE :: &
  current, increment, values
END MODULE state_variables

USE state_variables
IMPLICIT NONE
INTEGER :: NX, NY
READ *, NX, NY
ALLOCATE(current(NX,NY), increment(NX,NY), &
  values(NX,NY))
```
Explicit Interfaces

Procedures now need explicit interfaces e.g., for assumed shape arrays, keywords

- Modules are the primary way of doing this
We will come to the secondary way later

Simplest to include the procedures in modules
The procedure code goes after CONTAINS
This is what we discussed earlier
MODULE mymod
CONTAINS
  FUNCTION Variance (Array)
    REAL :: Variance, X
    REAL, INTENT(IN), DIMENSION(:) :: Array
    X = SUM(Array)/SIZE(Array)
    Variance = SUM(((Array-X)**2)/SIZE(Array))
  END FUNCTION Variance
END MODULE mymod

PROGRAM main
  USE mymod
  PRINT *, 'Variance = ', Variance(array)
Procedures in Modules (1)

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

• There really isn’t much more to it

It doesn’t handle very large modules well
Try to avoid designing these if possible
These are very much like **internal procedures**
Works very well in almost all programs

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
END MODULE thing

CONTAINS

SUBROUTINE pete ()
  INTEGER, PARAMETER :: temp = 456
  PRINT *, temp
END SUBROUTINE pete

END MODULE thing

This will print 456, not 123
Avoid doing this as it’s very confusing
Interfaces in Modules

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

• Be absolutely sure they are consistent!
The interface and code are not checked

Example: Cholesky decomposition
What Are Interfaces?

The \texttt{FUNCTION} or \texttt{SUBROUTINE} statement
And everything \texttt{directly connected} to that

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

Local variables can be left out
SUBROUTINE cholesky(A)  
USE DOUBLE  
INTEGER :: j, n  
REAL(KIND=dp) :: A(:,,:), X  

...  

END SUBROUTINE cholesky
Interfaces in Procedures

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

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

- Essential for using certain specific features
e.g., keyword arguments within a module

Example: proc_as_arg

Generic procedure example: genericswap.f90
Accessibility (1)

Can separate exported from hidden definitions

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

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

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

They are just another attribute of declarations

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

PUBLIC/PRIVATE statement sets the default
The default default is PUBLIC

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

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

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

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

You can include only some names in USE

USE bigmodule, ONLY : errors, invert

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

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

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

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

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

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

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

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

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

• The resulting chaos is almost unfindable
From bitter experience in many years of Fortran!
Example (1)

MODULE settings
  INTEGER, PARAMETER :: DP = KIND(0.0D0)
  REAL(KIND=DP) :: Z = 1.0_DP
END MODULE settings

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

PROGRAM main
  IMPLICIT NONE
  USE workspace
  Z = 123
  ...
END PROGRAM main

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

The following are better and best

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

MODULE workspace
  USE settings, ONLY : DP
  PRIVATE :: DP
  REAL(KIND=DP), DIMENSION(1000) :: scratch
END MODULE workspace
Renaming Inclusion (1)

You can rename a name when you include it

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

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

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

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

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

pooh is accessible under the name sanders
The name pooh is the local array
Why Is This Lethal?

MODULE one
  REAL :: X
END MODULE one

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

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

Kind and Precision
(a.k.a. Parameterized Data Types)
**Background**

- **Fortran 77** had a problem with numeric portability. A default **REAL** might support numbers up to $10^{68}$ on one machine and up to $10^{136}$ on another.

- **Fortran 90/95/2003** includes a **KIND** parameter which provides a way to parameterize the selection of different possible machine representations for each of the intrinsic data types (**INTEGER**, **REAL**, **COMPLEX**, **LOGICAL** and **CHARACTER**).

- **Main usage**: Provide a mechanism for making the selection of numeric **precision** and **range** portable.
The intrinsic inquiry function \texttt{KIND} will return the \texttt{kind value} of a given variable. The return value is a scalar.

Although it is common for the return value to be the same as the \texttt{number of bytes} stored in a variable of that kind, it is \texttt{NOT REQUIRED} by the Fortran standard.
KIND Values (2)

On a lot of systems:

```f90
REAL(KIND=4) :: xs    ! 4-byte IEEE float
REAL(KIND=8) :: xd    ! 8-byte IEEE float
REAL(KIND=16) :: xq   ! 16-byte IEEE float
```

But on some systems.compilers:

```f90
REAL(KIND=1) :: xs    ! 4-byte IEEE float
REAL(KIND=2) :: xd    ! 8-byte IEEE float
REAL(KIND=3) :: xq    ! 16-byte IEEE float
```

Quick sample program:  `mykinds.f90`
You can request a minimum precision and range

\[ \text{SELECTED\_REAL\_KIND}(P, R) \]

This gives at least \( P \) decimal places and range of \( 10^{-R} \) to \( 10^{R} \)

e.g., \[ \text{SELECTED\_REAL\_KIND}(12) \] will give at least 12 decimal places

Return codes:
-1 = does not support \( P \) value
-2 = does not support \( R \) value
-3 = neither is supported
For large programs it is extremely handy to put this into a module:

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

Then, 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, 1.23E12_DP, 3.141592653589793_DP
Using KIND (3)

Note that the above makes it trivial to change all variables and constants in a large program. All you need to do is change the module

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

requires **IEEE 754 double** or better

Or even: `SELECTED_REAL_KIND(25, 1000)`
DOUBLE PRECISION

This was the second “kind” of real type in Fortran 77.

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

\[
\text{REAL (KIND=KIND(0.0D0)} :: a, b, c \\
\text{DOUBLE PRECISION, DIMENSION(10)} :: x, y, z
\]

Constants use D for the exponent

\[
0.0D0, 7.0D0, 0.25D0, 1.23D12, \\
3.141592653589793D0
\]

Quick sample program: mykinds1.f90
Type Conversion (1)

This is the main “gotcha” - you should use:

```plaintext
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!

```plaintext
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

\[ x = \text{DBLE}(<\text{integer expression}>) \]

All other “gotchas” are for COMPLEX
You can choose different sizes of integer

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

bignum can hold values up to $10^{12}$

Few users will need this - mainly for OpenMP

Some compilers may allocate smaller integers e.g., by using `SELECTED_INT_KIND(4)`
It can be used to select the encoding.
It is mainly a Fortran 2003 feature.
Can select default, ASCII, or ISO 10646.
ISO 10646 is effectively Unicode.
Not covered in this course.
Notes

• The Fortran standard requires that each compiler support at least two real kinds which must have different precisions. The default real kind is the lower precision of these.

• There are two ways to specify a double precision real:

1. With a REAL specifier using the KIND parameter corresponding to double precision (portable)

2. Using a DOUBLE PRECISION specifier (not portable)
Related Inquiry Functions

KIND(x) returns the kind value of x
PRECISION(x) returns the decimal precision of x
RANGE(x) returns the decimal exponent range of x
TINY(x) returns the smallest non-zero number of x
HUGE(x) returns the largest non-infinite number of x
DIGITS(x) returns the number of significant digits in the internal model representation of x
RADIX(x) returns the base of the model representing x
MINEXPONENT(x) returns the minimum exponent of the model representing x
MAXEXPONENT(x) returns the maximum exponent of the model representing x
Derived Types
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
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 90/95 Restriction

Fortran 90/95 was much more restrictive
You couldn’t have **ALLOCATABLE** arrays
Had to use **pointers** instead

Fortran 2003 removed that restriction
Most compilers already include this feature

Be sure to check your own compiler
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.

```fortran
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.

```fortran
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
```
You can assign complete derived types
That copies the values 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 that I am omitting
Each derived type is unique.
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

```fortran
TYPE Circle
    REAL :: X, Y, radius
    LOGICAL :: filled
END TYPE Circle

TYPE(Circle) :: a
a = Circle(1.23, 4.56, 2.0, .False.)

Fortran 2003 allows keywords for components

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

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 in Fortran 2003

```
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 embedded derived types)

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

```
1.230000  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 :: width, 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)

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
Hidden Components (2)

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

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

Example: 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 tessellation.

We shall return to this when we cover pointers.
SUBROUTINE make_vmm(xyz, nat, imm, nmm, &
                      vmm, ielem, fudge_a, fudge_b, fvdws)
  INTEGER, INTENT(IN) :: nat, nmm
  INTEGER, INTENT(IN) :: imm(5,nmm), ielem(nat)
  REAL, INTENT(IN) :: xyz(3,nat), fudge_a, fudge_b, &
                      fvdws(6)
  REAL, INTENT(OUT) :: vmm(nmm)
END SUBROUTINE make_vmm
MODULE Delocal

INTEGER, PARAMETER :: MaxCoords = 1000
INTEGER, PARAMETER :: MaxAtoms = 100

TYPE MolecularMechanicsCoords
  INTEGER :: nmm
  INTEGER :: imm(5, MaxCoords)
  REAL :: vmm(MaxCoords)
END TYPE MolecularMechanicsCoords

TYPE MMFactors
  REAL :: fudge_a, fudge_b
  REAL :: fvdws(6)
END TYPE MMFactors
TYPE Geometry
  INTEGER :: nat
  INTEGER :: ielem(MaxAtoms)
  REAL :: xyz(3,MaxAtoms)
END TYPE Geometry
END MODULE Delocal

SUBROUTINE make_vmm(mmCoords, geom, factors)
  USE Delocal
  TYPE(MolecularMechanicsCoords) :: mmCoords
  TYPE(Geometry) :: geom
  TYPE(MMFactors) :: factors
END SUBROUTINE make_vmm
SUBROUTINE make_vmm(xyz, nat, imm, nmm, &
vmm, ielem, fudge_a, fudge_b, fudge_c, fvdws)
INTEGER, INTENT(IN) :: nat, nmm
INTEGER, INTENT(IN) :: imm(5,nmm), ielem(nat)
REAL, INTENT(IN) :: xyz(3,nat), fudge_a, fudge_b, &
fudge_c, fvdws(7)
REAL, INTENT(OUT) :: vmm(nmm)
END SUBROUTINE make_vmm

REAL :: fudge_c, fvdws(7)
CALL make_vmm(xyz, nat, imm, nmm, &
vmm, ielem, fudge_a, fudge_b, fudge_c, fvdws)
MODULE Delocal

... TYPE MMFactors
    REAL :: fudge_a, fudge_b, fudge_c
    REAL :: fvdws(7)
END TYPE MMFactors

...

CALL make_vmm(mmCoords, geom, factors)
Make and Makefiles
This course will give a brief overview of how to use make with Fortran.

Will cover the basics only!

Then look at how modules complicate the use of make.
What is Make?

Make is a tool which controls the generation of executables from a program’s source files.

It gets its knowledge of how to build your program from a file called the makefile.

The compilation procedure is much faster:

• The compilation is done with a single command.
• Only files that have been modified are recompiled.
• Allows managing large programs with lots of dependencies.
Makefile Basics (1)

A rule in the makefile tells Make how to execute a series of commands in order to build a target file from source files. It also specifies a list of dependencies of the target file. Here is what a simple rule looks like:

```make
  target: dependencies ... (also called prerequisites)
  <tab> commands
```

The <tab> is absolutely necessary!
Make uses **timestamps** to locate the files that have been modified since the last time make was executed.

By default when you type **make** it looks for the file **makefile** or **Makefile**. You can designate a specific name with **make -f <thismakefile>**.

Can also use **macros** to give names to variables within the makefile. **NOTE** these are **case-sensitive**!

If no specific target is given in the make command then **Make** starts with the **first** target listed in the makefile.

Let’s start with a very simple example (**abc** program).
Makefile Basics (3)

Comments are delimited by the `#` symbol.

A backslash `\` can be used as a continuation character.

Common extra tidbit: Create a “phony target” called `clean` which can be run to do a fresh recompile of all source code.
Makefile Automatic Variables

These can only be values in the *recipe*. They cannot be used in the *target list* of a rule

$<$ The name of the first prerequisite

$^$ The names of the all prerequisites

$@$ The file name of the target of the rule

And there are even more available
Compiling Modules

When modules are compiled both a .o and .mod file are created.

A .mod file is like a compiled header. This is what the compiler searches for when it sees a USE statement.

The dependencies can start to get cumbersome and complicated when many modules are USED and inherited.

Make has no method for determining these for you.

Take a look at example2.
Helpful Tools

**mkDepends** - generate a list of dependencies

**mkSrcfiles** - generate a list of all source files

Versions of these perl scripts are used in atmospheric models like **SAM** and **CAM**

**mkdep** - requires both GNU make and Python

**fmfmk.pl** - generate a makefile

**foraytool** - made especially for compiling large Fortran codes