THE UNIVERSITY

of LIVERPOOL

Fortran 90 Programming

(5 Day Course)

Dr. A C Marshall (funded by JISC/NTI)

with acknowledgements to Steve Morgan and Lawrie Schonfelder.
Lecture 1:
Introduction
Course Philosophy

The course:

- assumes a familiarity with a high level language;
- stresses modern scientific programming syntax, for example, array language, modules, defined types, recursion and overloaded operators;
- gives many examples;
**Fortran Evolution**

**History:**

- FORmula TRANslation.
- first compiler: 1957.
- updated in 1980 to Fortran 77.
- updated further in 1991 to Fortran 90.
- next upgrade due in 1996 - remove obsolescent features, correct mistakes and add limited basket of new facilities such as ELEMENTAL and PURE user-defined procedures and the FORALL statement.
- Fortran is now an ISO/IEC and ANSI standard.
**Drawbacks of Fortran 77**

Fortran 77 was limited in the following areas,

1. awkward ‘punched card’ or ‘fixed form’ source format;

2. inability to represent intrinsically parallel operations;

3. lack of dynamic storage;

4. non-portability;

5. no user-defined data types;

6. lack of explicit recursion;

7. reliance on unsafe storage and sequence association features.
Fortran 90 New features

Fortran 90 supports,

1. free source form;

2. array syntax and many more (array) intrinsics;

3. dynamic storage and pointers;

4. portable data types (KINDS);

5. derived data types and operators;

6. recursion;

7. MODULES
   - procedure interfaces;
   - enhanced control structures;
   - user defined generic procedures;
   - enhanced I/O.
Advantages of Additions

Fortran 90 is:

☐ more natural;

☐ greater flexibility;

☐ enhanced safety;

☐ parallel execution;

☐ separate compilation;

☐ greater portability;

but is

☐ larger;

☐ more complex;
Language Obsolescence

Fortran 90 has a number of features marked as obsolescent, this means,

- they are already redundant in Fortran 77;
- better methods of programming already existed in the Fortran 77 standard;
- programmers should stop using them;
- the standards committee’s intention is that many of these features will be removed from the next revision of the language, Fortran 95;
Obsolescent Features

The following features are labelled as obsolescent and will be removed from the next revision of Fortran, Fortran 95,

- the arithmetic IF statement;
- ASSIGN statement;
- ASSIGNED GOTO statements;
- ASSIGNED FORMAT statements;
- Hollerith format strings;
- the PAUSE statement;
- REAL and DOUBLE PRECISION DO-loop control expressions and index variables;
- shared DO-loop termination;
- alternate RETURN;
- branching to an ENDIF from outside the IF block;
Undesirable Features

- fixed source form layout - use free form;
- implicit declaration of variables - use IMPLICIT NONE;
- COMMON blocks - use MODULE;
- assumed size arrays - use assumed shape;
- EQUIVALENCE statements;
- ENTRY statements;
- the computed GOTO statement - use IF statement;
Object Oriented Facilities

Fortran 90 has some Object Oriented facilities such as:

- *data abstraction* — user-defined types;
- *data hiding* — PRIVATE and PUBLIC attributes;
- *encapsulation* — Modules and data hiding facilities;
- *inheritance* and *extensibility* — super-types, operator overloading and generic procedures;
- *polymorphism* — user can program his / her own polymorphism by generic overloading;
- *reusability* — Modules;
Lecture 2: Elements of Fortran 90
Example Fortran 90 program:

```
MODULE Triangle_Operations
  IMPLICIT NONE
  CONTAINS
  FUNCTION Area(x,y,z)
    REAL :: Area ! function type
    REAL, INTENT(IN) :: x, y, z
    REAL :: theta, height
    theta=ACOS((x**2+y**2-z**2)/(2.0*x*y))
    height=x*SIN(theta); Area=0.5*y*height
  END FUNCTION Area
END MODULE Triangle_Operations

PROGRAM Triangle
  USE Triangle_Operations
  IMPLICIT NONE
  REAL :: a, b, c, Area
  PRINT*, 'Welcome, please enter the\n        &lengths of the 3 sides.'
  READ*, a, b, c
  PRINT*, 'Triangle’s area: ',Area(a,b,c)
END PROGRAM Triangle
```
**Coding Style**

It is recommended that the following coding convention is adopted:

- *always* use IMPLICIT NONE.

- Fortran 90 keywords, intrinsic functions and user defined entities should be in upper case,

- other user entities should be in lower case but may start with a capital letter.

- indentation should be 1 or 2 spaces and should be applied to the bodies of program units, control blocks, INTERFACE blocks, etc.

- the names of program units are always included in their END statements,

- argument keywords are always used for optional arguments,

---

Please note: In order that a program fits onto a slide these rules are sometimes relaxed here.
Free source form:

- 132 characters per line;
- ‘!’ comment initiator;
- ‘&’ line continuation character;
- ‘;’ statement separator;
- significant blanks.

Example,

```
PRINT*, "This line is continued &
 &On the next line"; END ! of program
```
The following are valid in a Fortran 90 program:

- **alphanumeric:**
  - a–z, A–Z, 0–9, and _ (the underscore)

- **symbolic:**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>space</td>
<td></td>
<td>=</td>
<td>equal</td>
</tr>
<tr>
<td>+</td>
<td>plus</td>
<td>-</td>
<td>minus</td>
</tr>
<tr>
<td>*</td>
<td>asterisk</td>
<td>/</td>
<td>slash</td>
</tr>
<tr>
<td>(</td>
<td>left paren</td>
<td>)</td>
<td>right paren</td>
</tr>
<tr>
<td>,</td>
<td>comma</td>
<td>.</td>
<td>period</td>
</tr>
<tr>
<td>'</td>
<td>single quote</td>
<td>&quot;</td>
<td>double quote</td>
</tr>
<tr>
<td>:</td>
<td>colon</td>
<td>;</td>
<td>semicolon</td>
</tr>
<tr>
<td>!</td>
<td>shriek</td>
<td>&amp;</td>
<td>ampersand</td>
</tr>
<tr>
<td>%</td>
<td>percent</td>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>$</td>
<td>dollar</td>
</tr>
<tr>
<td>?</td>
<td>question mark</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Significance of Blanks

In free form source code blanks must not appear:

- within keywords

  INTEGER :: wizzy  ! is a valid keyword
  INTEGER :: wizzy  ! is not

- within names

  REAL :: running_total  ! is a valid name
  REAL :: running total  ! is not

Blanks must appear:

- between two separate keywords

- between keywords and names not otherwise separated by punctuation or other special characters.

  INTEGER FUNCTION fit(i)  ! is valid
  INTEGERFUNCTION fit(i)  ! is not
  INTEGER FUNCTION fit(i)  ! is not

Blanks are optional between some keywords mainly ‘END <construct>’ and a few others; if in doubt add a blank (it looks better too).
Comments

It is good practice to use lots of comments, for example,

```fortran
PROGRAM S addo
!
! Program to evaluate marriage potential
!
LOGICAL :: TrainSpotter  ! Do we spot trains?
LOGICAL :: SmellySocks   ! Have we smelly socks?
INTEGER :: i, j          ! Loop variables
```

- everything after the ! is a comment;

- the ! in a character context does **not** begin a comment, for example,

```fortran
PRINT*, "No chance of ever marrying!!!"
```
Names

In Fortran 90 variable names (and procedure names etc.)

- must start with a letter

```
REAL :: a1 ! valid name
REAL :: 1a ! not valid name
```

- may use only letters, digits and the underscore

```
CHARACTER :: atoz ! valid name
CHARACTER :: a-z ! not valid name
CHARACTER :: a_z ! OK
```

- underscore should be used to separate words in long names

```
CHARACTER(LEN=8) :: user_name ! valid name
CHARACTER(LEN=8) :: username ! different name
```

- may not be longer than 31 characters
The following table details the prescribed ordering:

<table>
<thead>
<tr>
<th>PROGRAM, FUNCTION, SUBROUTINE, MODULE or BLOCK DATA statement</th>
<th>USE statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORMAT and ENTRY statements</td>
<td>IMPLICIT NONE</td>
</tr>
<tr>
<td>PARAMETER statement</td>
<td>IMPLICIT statements</td>
</tr>
<tr>
<td>PARAMETER and DATA statements</td>
<td>Derived-Type Definition, Interface blocks, Type declaration statements, Statement function statements and specification statements</td>
</tr>
<tr>
<td>DATA statements</td>
<td>Executable constructs</td>
</tr>
<tr>
<td>CONTAINS statement</td>
<td></td>
</tr>
<tr>
<td>Internal or module procedures</td>
<td></td>
</tr>
<tr>
<td>END statement</td>
<td></td>
</tr>
</tbody>
</table>
Intrinsic Types

Fortran 90 has three broad classes of object type,

- character;
- boolean;
- numeric.

these give rise to six simple intrinsic types, known as default types,

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER</td>
<td>sex</td>
<td>letter</td>
</tr>
<tr>
<td>CHARACTER(LEN=12)</td>
<td>name</td>
<td>string</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>wed</td>
<td>married?</td>
</tr>
<tr>
<td>REAL</td>
<td>height</td>
<td></td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>pi</td>
<td>3.14...</td>
</tr>
<tr>
<td>INTEGER</td>
<td>age</td>
<td>whole No.</td>
</tr>
<tr>
<td>COMPLEX</td>
<td>val</td>
<td>x + iy</td>
</tr>
</tbody>
</table>
**Literal Constants**

A literal constant is an entity with a fixed value:

12345 ! INTEGER
1.0 ! REAL
-6.6E-06 ! REAL: -6.6*10**(-6)
.FALSE. ! LOGICAL
.TRUE. ! LOGICAL
"Mau’dib" ! CHARACTER
'Mau’dib’ ! CHARACTER

Note,

- there are only two LOGICAL values;
- REALS contain a decimal point, INTEGERS do not,
- REALS have an exponential form
- character literals delimited by " and ’;
- two occurrences of the delimiter inside a string produce one occurrence on output;
- there is only a finite range of values that numeric literals can take.
Implicit Typing

Undeclared variables have an implicit type,

- if first letter is I, J, K, L, M or N then type is INTEGER;
- any other letter then type is REALS.

Implicit typing is potentially very dangerous and should always be turned off by adding:

```plaintext
IMPLICIT NONE
```
as the first line after any USE statements.

Consider,

```plaintext
DO 30 I = 1.1000
...            
30 CONTINUE
```
in fixed format with implicit typing this declares a REAL variable DO30I and sets it to 1.1000 instead of performing a loop 1000 times!
Numeric and Logical Declarations

With IMPLICIT NONE variables must be declared. A simplified syntax follows,

\[ <\text{type}> [,<\text{attribute-list}>] :: <\text{variable-list}> \& [ =<\text{value}> ] \]

The following are all valid declarations,

\begin{align*}
\text{REAL} &:: x \\
\text{INTEGER} &:: i, j \\
\text{LOGICAL, POINTER} &:: \text{ptr} \\
\text{REAL, DIMENSION(10,10)} &:: y, z \\
\text{INTEGER} &:: k = 4
\end{align*}

The DIMENSION attribute declares an array \((10 \times 10)\).
Character Declarions

Character variables are declared in a similar way to numeric types. CHARACTER variables can

- refer to one character;

- refer to a string of characters which is achieved by adding a length specifier to the object declaration.

The following are all valid declarations,

```plaintext
CHARACTER(LEN=10) :: name
CHARACTER :: sex
CHARACTER(LEN=32) :: str
CHARACTER(LEN=10), DIMENSION(10,10) :: Harray
CHARACTER(LEN=32), POINTER :: Pstr
```
Constants (Parameters)

Symbolic constants, oddly known as parameters in Fortran, can easily be set up either in an attributed declaration or parameter statement,

\[
\text{REAL, PARAMETER :: pi = 3.14159} \\
\text{CHARACTER(LEN=*)}, \text{PARAMETER :: &} \\
\quad \text{son = 'bart', dad = "Homer"}
\]

CHARACTER constants can assume their length from the associated literal (LEN=*).

Parameters should be used:

- if it is known that a variable will only take one value;
- for legibility where a ‘magic value’ occurs in a program such as \(\pi\);
- for maintainability when a ‘constant’ value could feasibly be changed in the future.
Initialisation

Variables can be given initial values:

- can use *initialisation expressions*,
- may only contain *PARAMETERs* or literals.

```plaintext
REAL         :: x, y = 1.0D5
INTEGER      :: i = 5, j = 100
CHARACTER(LEN=5) :: light = 'Amber'
CHARACTER(LEN=9) :: gumboot = 'Wellie'
LOGICAL      :: on = .TRUE., off = .FALSE.
REAL, PARAMETER :: pi = 3.141592
REAL, PARAMETER :: radius = 3.5
REAL         :: circum = 2 * pi * radius
```

gumboot will be padded, to the right, with blanks.

In general, intrinsic functions *cannot* be used in initialisation expressions, the following can be: REPEAT, RE-SHAPE, SELECTED_INT_KIND, SELECTED_REAL_KIND, TRANSFER, TRIM, LBOUND, UBOUND, SHAPE, SIZE, KIND, LEN, BIT_SIZE and numeric inquiry intrinsics, for, example, HUGE, TINY, EPSILON.
Expressions

Each of the three broad type classes has its own set of intrinsic (in-built) operators, for example, +, // and .AND.,

The following are valid expressions,

- NumBabiesBorn+1 — numeric valued
- "Ward "/Ward — character valued
- TimeSinceLastBirth .GT. MaxTimeTwixtBirths — logical valued

Expressions can be used in many contexts and can be of any intrinsic type.
Assignment

Assignment is defined between all expressions of the same type:

Examples,

\[ a = b \]
\[ c = \text{SIN}(.7) \times 12.7 \quad ! \text{SIN in radians} \]
\[ \text{name} = \text{initials} // \text{surname} \]
\[ \text{bool} = (a.EQ.b.OR.c.NE.d) \]

The LHS is an object and the RHS is an expression.
Intrinsic Numeric Operations

The following operators are valid for numeric expressions,

- ** exponentiation, dyadic operator, for example, 10**2, (evaluated right to left);
- * and / multiply and divide, dyadic operators, for example, 10*7/4;
- + and – plus and minus or add and subtract, monadic and dyadic operators, for example, 10+7–4 and –3;

Can be applied to literals, constants, scalar and array objects. The only restriction is that the RHS of ** must be scalar.

Example,

\[
\begin{align*}
a &= b - c \\
f &= -3*6/5
\end{align*}
\]
Relational Operators

The following *relational operators* deliver a **LOGICAL** result when combined with numeric operands,

\[
\begin{array}{|c|l|}
\hline
\text{.GT. } & \text{greater than} \\
\text{.GE. } & \text{greater than or equal to} \\
\text{.LE. } & \text{less than or equal to} \\
\text{.LT. } & \text{less than} \\
\text{.NE. } & \text{not equal to} \\
\text{.EQ. } & \text{equal to} \\
\hline
\end{array}
\]

For example,

```plaintext
bool = i .GT. j
boule = i > j
IF (i .EQ. j) c = D
IF (i == j)  c = D
```

When using real-valued expressions (which are approximate) .EQ. and .NE. have no real meaning.

```plaintext
REAL :: Tol = 0.0001
IF (ABS(a-b) .LT. Tol) same = .TRUE.
```
Intrinsic Logical Operations

A LOGICAL or boolean expression returns a .TRUE. / .FALSE. result. The following are valid with LOGICAL operands,

- .NOT. — .TRUE. if operand is .FALSE..
- .AND. — .TRUE. if both operands are .TRUE.;
- .OR. — .TRUE. if at least one operand is .TRUE.;
- .EQV. — .TRUE. if both operands are the same;
- .NEQV. — .TRUE. if both operands are different.

For example, if T is .TRUE. and F is .FALSE.

- .NOT. T is .FALSE., .NOT. F is .TRUE..
- T .AND. F is .FALSE., T .AND. T is .TRUE..
- T .OR. F is .TRUE., F .OR. F is .FALSE..
- T .EQV. F is .FALSE., F .EQV. F is .TRUE..
- T .NEQV. F is .TRUE., F .NEQV. F is .FALSE.
Intrinsic Character Operations

Consider,

```
CHARACTER(LEN=*) , PARAMETER :: str1 = "abcdef"
CHARACTER(LEN=*) , PARAMETER :: str2 = "xyz"
```

substrings can be taken,

- `str1` is 'abcdef'
- `str1(1:1)` is 'a' (**not** `str1(1)` — illegal)
- `str1(2:4)` is 'bcd'

The concatenation operator, `//`, is used to join two strings.

```
PRINT* , str1//str2
PRINT* , str1(4:5)//str2(1:2)
```

would produce

```
abcdefxyz
dxy
```
## Operator Precedence

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>user-defined monadic</td>
<td>Highest</td>
<td>.INVERSE.A</td>
</tr>
<tr>
<td>**</td>
<td>.</td>
<td>10**4</td>
</tr>
<tr>
<td>* or /</td>
<td>.</td>
<td>89*55</td>
</tr>
<tr>
<td>monadic + or -</td>
<td>.</td>
<td>-4</td>
</tr>
<tr>
<td>dyadic + or -</td>
<td>.</td>
<td>5+4</td>
</tr>
<tr>
<td>//</td>
<td>.</td>
<td>str1//str2</td>
</tr>
<tr>
<td>.GT., &gt;, .LE., &lt;=, etc</td>
<td>.</td>
<td>A &gt; B</td>
</tr>
<tr>
<td>.NOT.</td>
<td>.</td>
<td>.NOT.Bool</td>
</tr>
<tr>
<td>.AND.</td>
<td>.</td>
<td>A.AND.B</td>
</tr>
<tr>
<td>.OR.</td>
<td>.</td>
<td>A.OR.B</td>
</tr>
<tr>
<td>.EQV. or .NEQV.</td>
<td>.</td>
<td>A.EQV.B</td>
</tr>
<tr>
<td>user-defined dyadic</td>
<td>Lowest</td>
<td>X.DOT.Y</td>
</tr>
</tbody>
</table>

**Note:**

- in an expression with no parentheses, the highest precedence operator is combined with its operands first;

- in contexts of equal precedence left to right evaluation is performed except for **.
The following expression,
\[ x = a + b/5.0 - c**d + 1*e \]
is equivalent to
\[ x = a + b/5.0 - (c**d) + 1*e \]
as ** is highest precedence. This is equivalent to
\[ x = a + (b/5.0) - (c**d) + (1*e) \]
as / and * are next highest. The remaining operators' precedences are equal, so we evaluate from left to right.
**Precision Errors**

Since real numbers are stored approximately:

- every operation yields a slight loss of accuracy,
- after many operations such 'round-off' errors build up,
- catastrophic accuracy loss can arise when values that are almost equal are subtracted, leading digits are cancelled and a rounding errors become visible.

Consider,

\[
x = 0.123456; \quad y = 0.123446 \\
\text{PRINT*}, \quad "x = ",x," \quad y = ",y \\
\text{PRINT*}, \quad "x-y = ",x-y," \quad \text{should be 0.100d-4}''
\]

May produce:

\[
x = 0.123457 \quad y = 0.123445 \\
x-y = 0.130d-4 \quad \text{should be 0.100d-4}
\]

which is 30% in error.

A whole branch of Numerical Analysis is dedicated to minimising this class of errors in algorithms. Be careful!
Lecture 3:
Control Constructs,
Intrinsics and Basic
I/O
Control Flow

Control constructs allow the normal sequential order of execution to be changed.

Fortran 90 supports:

- conditional execution statements and constructs, 
  (IF ... and IF ... THEN ... ELSE ... END IF);

- loops, (DO ... END DO);

- multi-way choice construct, (SELECT CASE);
**IF Statement**

Example,

```plaintext
IF (bool_val) A = 3
```

The basic syntax is,

```plaintext
IF(< logical-expression >)< exec-stmt >
```

If `<logical-expression>` evaluates to `.TRUE.` then execute `<exec-stmt>` otherwise do not.

For example,

```plaintext
IF (x .GT. y) Maxi = x
```

means ‘if x is greater than y then set Maxi to be equal to the value of x’.

More examples,

```plaintext
IF (a*b+c <= 47) Boolie = .TRUE.
IF (i .NE. 0 .AND. j .NE. 0) k = 1/(i*j)
IF (i /= 0 .AND. j /= 0) k = 1/(i*j) ! same
```
Consider the IF statement

\[
\text{IF (} I > 17 \text{) Print*, } "I > 17"
\]

this maps onto the following control flow structure,
IF ... THEN ... ELSE Construct

The block-IF is a more flexible version of the single line IF. A simple example,

    IF (i .EQ. 0) THEN
        PRINT*, "I is Zero"
    ELSE
        PRINT*, "I is NOT Zero"
    ENDF

note the how indentation helps.

Can also have one or more ELSEIF branches:

    IF (i .EQ. 0) THEN
        PRINT*, "I is Zero"
    ELSEIF (i .GT. 0) THEN
        PRINT*, "I is greater than Zero"
    ELSE
        PRINT*, "I must be less than Zero"
    ENDF

Both ELSE and ELSEIF are optional.
Visualisation of the IF ... THEN Construct

Consider the IF ... THEN construct

```
IF (I > 17) THEN
    PRINT*, "I > 17"
END IF
```

this maps onto the following control flow structure,
Visualisation of the **IF** ... **THEN** ... **ELSE** **Construct**

Consider the **IF** ... **THEN** ... **ELSE** construct

```plaintext
IF (I > 17) THEN
    Print*, "I > 17"
ELSE
    Print*, "I <= 17"
END IF
```

this maps onto the following control flow structure,
The IF construct has the following syntax,

\[
\text{IF}(<\text{logical-expression}>)\text{THEN} \\
\quad <\text{then-block}> \\
[ \text{ELSEIF}(<\text{logical-expression}>)\text{THEN} \\
\quad <\text{elseif-block}> \\
... ] \\
[ \text{ELSE} \\
\quad <\text{else-block}> ] \\
\text{END IF}
\]

The first branch to have a true \(<\text{logical-expression}\>\) is the one that is executed. If none are found then the \(<\text{else-block}\>\), if present, is executed.

For example,

\[
\text{IF} (x \ .\text{GT.} \ 3) \text{THEN} \\
\quad \text{CALL SUB1} \\
\text{ELSEIF} (x \ .\text{EQ.} \ 3) \text{THEN} \\
\quad A = B*C-D \\
\text{ELSEIF} (x \ .\text{EQ.} \ 2) \text{THEN} \\
\quad A = B*B \\
\text{ELSE} \\
\quad \text{IF} (y \ .\text{NE.} \ 0) A=B \\
\text{ENDIF}
\]

IF blocks may also be nested.
Visualisation of IF ... THEN .. ELSEIF Construct

Consider the IF ... THEN ... ELSEIF construct

\[
\text{IF (I > 17) THEN} \\
\quad \text{Print*, "I > 17"} \\
\text{ELSEIF (I == 17) } \\
\quad \text{Print*, "I == 17"} \\
\text{ELSE } \\
\quad \text{Print*, "I < 17"} \\
\text{ENDIF}
\]

this maps onto the following control flow structure,
Nested and Named IF Constructs

All control constructs can be both named and nested.

```plaintext
outa: IF (a .NE. 0) THEN
    PRINT*, "a /= 0"
    IF (c .NE. 0) THEN
        PRINT*, "a /= 0 AND c /= 0"
    ELSE
        PRINT*, "a /= 0 BUT c == 0"
    ENDIF
ELSEIF (a .GT. 0) THEN outa
    PRINT*, "a > 0"
ELSE outa
    PRINT*, "a must be < 0"
ENDIF outa
```

The names may only be used once per program unit and are only intended to make the code clearer.
Conditional Exit Loops

Can set up a DO loop which is terminated by simply jumping out of it. Consider,

\[
i = 0
\]
\[
DO
\]
\[
i = i + 1
\]
\[
IF \ (i \ GT \ 100) \ EXIT
\]
\[
PRINT*, "I is", i
\]
\[
END \ DO
\]
\[
! \ if \ i>100 \ control \ jumps \ here
\]
\[
PRINT*, "Loop finished. I now equals", i
\]

this will generate

\[
I \ is \ 1
\]
\[
I \ is \ 2
\]
\[
I \ is \ 3
\]
\[
....
\]
\[
I \ is \ 100
\]
\[
Loop \ finished. I \ now \ equals \ 101
\]

The EXIT statement tells control to jump out of the current DO loop.
Conditional Cycle Loops

Can set up a DO loop which, on some iterations, only executes a subset of its statements. Consider,

\[
i = 0 \\
DO \\
i = i + 1 \\
IF (i \geq 50 \text{ AND } i \leq 59) \text{ CYCLE} \\
IF (i > 100) \text{ EXIT} \\
PRINT*, "I is", i \\
END DO \\
PRINT*, "Loop finished. I now equals", i
\]

this will generate

I is 1
I is 2
....
I is 49
I is 60
....
I is 100
Loop finished. I now equals 101

CYCLE forces control to the innermost active DO statement and the loop begins a new iteration.
Named and Nested Loops

Loops can be given names and an EXIT or CYCLE statement can be made to refer to a particular loop.

```
0| outa: DO
1| inna: DO
2| ...
3| IF (a.GT.b) EXIT outa ! jump to line 9
4| IF (a.EQ.b) CYCLE outa ! jump to line 0
5| IF (c.GT.d) EXIT inna ! jump to line 8
6| IF (c.EQ.a) CYCLE ! jump to line 1
7| END DO inna
8| END DO outa
9| ...
```

The (optional) name following the EXIT or CYCLE highlights which loop the statement refers to.

Loop names can only be used once per program unit.
DO ... WHILE Loops

If a condition is to be tested at the top of a loop a DO ... WHILE loop could be used,

    DO WHILE (a .EQ. b)
        ...
    END DO

The loop only executes if the logical expression evaluates to .TRUE.. Clearly, here, the values of a or b must be modified within the loop otherwise it will never terminate.

The above loop is functionally equivalent to,

    DO; IF (a .NE. b) EXIT
        ...
    END DO
Indexed DO Loops

Loops can be written which cycle a fixed number of times. For example,

```
DO i1 = 1, 100, 1
    ... ! i is 1,2,3,...,100
    ... ! 100 iterations
END DO
```

The formal syntax is as follows,

```
DO < DO-var >=< expr1 >,< expr2 > [ ,< expr3 > ]
    < exec-stmts >
END DO
```

The number of iterations, which is evaluated before execution of the loop begins, is calculated as

```
MAX(INT((< expr2 >-< expr1 >+< expr3 >)/< expr3 >),0)
```

If this is zero or negative then the loop is not executed.

If `< expr3 >` is absent it is assumed to be equal to 1.
Examples of Loop Counts

A few examples of different loops,

1. upper bound not exact,

   loopy: DO i = 1, 30, 2
         ... ! i is 1, 3, 5, 7, ..., 29
         ... ! 15 iterations
         END DO loopy

2. negative stride,

   DO j = 30, 1, -2
       ... ! j is 30, 28, 26, ..., 2
       ... ! 15 iterations
       END DO

3. a zero-trip loop,

   DO k = 30, 1, 2
       ... ! 0 iterations
       ... ! loop skipped
       END DO

4. missing stride — assume it is 1,

   DO l = 1, 30
       ... ! i = 1, 2, 3, ..., 30
       ... ! 30 iterations
       END DO
**Scope of DO Variables**

1. I is recalculated at the top of the loop and then compared with `<expr2>`,

2. if the loop has finished, execution jumps to the statement after the corresponding *END DO*,

3. I retains the value that it had just been assigned.

For example,

    DO i = 4, 45, 17
        PRINT*, "I in loop = ", i
    END DO
    PRINT*, "I after loop = ", i

will produce

    I in loop =  4
    I in loop = 21
    I in loop = 38
    I after loop = 55

An index variable may not have its value changed in a loop.
SELECT CASE Construct I

Simple example

SELECT CASE (i)
    CASE (3, 5, 7)
        PRINT*,'i is prime'
    CASE (10:)
        PRINT*,'i is > 10'
    CASE DEFAULT
        PRINT*,'i is not prime and is < 10'
END SELECT

An IF..ENDIF construct could have been used but a
SELECT CASE is neater and more efficient. Another exam-
ple,

SELECT CASE (num)
    CASE (6, 9, 99, 66)
        IF(num==6.OR. . . .OR.num==66) THEN
            PRINT*,'Woof woof'
    CASE (10:65, 67:98)
        ELSEIF((num >= 10 .AND. num <= 65) .OR. . . .
            PRINT*,'Bow wow'
    CASE DEFAULT
        ELSE
            PRINT*,'Meeeoow'
END SELECT

! ENDIF
Consider the SELECT CASE construct

SELECT CASE (I)
    CASE(1); Print*, "I==1"
    CASE(2:9); Print*, "I>=2 and I<=9"
    CASE(10); Print*, "I>=10"
    CASE DEFAULT; Print*, "I<=0"
END SELECT CASE

this maps onto the following control flow structure,
SELECT CASE Construct II

This is useful if one of several paths must be chosen based on the value of a single expression.

The syntax is as follows,

\[
[<\text{name}>:]\text{SELECT CASE}(<\text{case-expr}>) \\
[\text{CASE}(<\text{case-selector}>)[<\text{name}>] \\
<\text{exec-stmts}> ]... \\
[\text{CASE DEFAULT}[<\text{name}>] \\
<\text{exec-stmts}> ]
\]

END SELECT[<\text{name}>]

Note,

- the \(<\text{case-expr}>\) must be scalar and \text{INTEGER}, \text{LOGICAL} or \text{CHARACTER} valued;

- the \(<\text{case-selector}>\) is a parenthesised single value or range, for example, (.TRUE.), (1) or (99:101);

- there can only be one \text{CASE DEFAULT} branch;

- control cannot jump into a \text{CASE} construct.
Mixed Type Numeric 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. DOUBLE PRECISION
4. COMPLEX — highest

The result of an expression is always of the highest type, for example,

- INTEGER * REAL gives a REAL, (3*2.0 is 6.0)
- REAL * INTEGER gives a REAL, (3.0*2 is 6.0)
- DOUBLE PRECISION * REAL gives DOUBLE PRECISION,
- COMPLEX * < anytype > gives COMPLEX,
- DOUBLE PRECISION * REAL * INTEGER gives DOUBLE PRECISION.

The actual operator is unimportant.
Mixed Type Assignment

Problems often occur with mixed-type arithmetic; the rules for type conversion are given below.

- **INTEGER = REAL (or DOUBLE PRECISION)**
  The RHS is evaluated, truncated (all the decimal places lopped off) then assigned to the LHS.

- **REAL (or DOUBLE PRECISION) = INTEGER**
  The RHS is promoted to be REAL and stored (approximately) in the LHS.

For example,

```plaintext
REAL :: a = 1.1, b = 0.1
INTEGER :: i, j, k
i = 3.9    ! i will be 3
j = -0.9   ! j will be 0
k = a - b  ! k will be 1 or 0
```

Note: as a and b stored approximately, the value of k is uncertain.
Integer Division

Confusion often arises about integer division; in short, division of two integers produces an integer result by truncation (towards zero). Consider,

\[
\begin{align*}
\text{REAL} &:: a, b, c, d, e \\
a & = 1999/1000 \quad \text{! LHS is 1} \\
b & = -1999/1000 \quad \text{! LHS is -1} \\
c & = (1999+1)/1000 \quad \text{! LHS is 2} \\
d & = 1999.0/1000 \quad \text{! LHS is 1.999} \\
e & = 1999/1000.0 \quad \text{! LHS is 1.999}
\end{align*}
\]

- a is (about) 1.000
- b is (about) -1.000
- c is (about) 2.000
- d is (about) 1.999
- e is (about) 1.999

Great care must be taken when using mixed type arithmetic.
Intrinsic Procedures

Fortran 90 has 113 in-built or intrinsic procedures to perform common tasks efficiently, they belong to a number of classes:

☐ elemental such as:
  ◊ mathematical, for example, SIN or LOG.
  ◊ numeric, for example, SUM or CEILING;
  ◊ character, for example, INDEX and TRIM;
  ◊ bit, for example, IAND and IOR;

☐ inquiry, for example, ALLOCATED and SIZE;

☐ transformational, for example, REAL and TRANSPOSE;

☐ miscellaneous (non-elemental SUBROUTINES), for example, SYSTEM_CLOCK and DATE_AND_TIME.

Note all intrinsics which take REAL valued arguments also accept DOUBLE PRECISION arguments.
Type Conversion Functions

It is easy to transform the type of an entity,

- \text{REAL}(i) \text{ converts } i \text{ to a real approximation,}
- \text{INT}(x) \text{ truncates } x \text{ to the integer equivalent,}
- \text{DBLE}(a) \text{ converts } a \text{ to DOUBLE PRECISION,}
- \text{IACHAR}(c) \text{ returns the position of CHARACTER } c \text{ in the ASCII collating sequence,}
- \text{ACHAR}(i) \text{ returns the } i^{th} \text{ character in the ASCII collating sequence.}

All above are intrinsic functions. For example,

\begin{verbatim}
PRINT*, REAL(1), INT(1.7), INT(-0.9999)
PRINT*, IACHAR('C'), ACHAR(67)
\end{verbatim}

are equal to

\begin{verbatim}
1.000000 1 0
67 C
\end{verbatim}
### Mathematical Intrinsic Functions

Summary,

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACOS(x)</td>
<td>arccosine</td>
</tr>
<tr>
<td>ASIN(x)</td>
<td>arcsine</td>
</tr>
<tr>
<td>ATAN(x)</td>
<td>arctangent</td>
</tr>
<tr>
<td>ATAN2(y, x)</td>
<td>arctangent of complex number (x, y)</td>
</tr>
<tr>
<td>COS(x)</td>
<td>cosine where x is in radians</td>
</tr>
<tr>
<td>COSH(x)</td>
<td>hyperbolic cosine where x is in radians</td>
</tr>
<tr>
<td>EXP(x)</td>
<td>e raised to the power x</td>
</tr>
<tr>
<td>LOG(x)</td>
<td>natural logarithm of x</td>
</tr>
<tr>
<td>LOG10(x)</td>
<td>logarithm base 10 of x</td>
</tr>
<tr>
<td>SIN(x)</td>
<td>sine where x is in radians</td>
</tr>
<tr>
<td>SINH(x)</td>
<td>hyperbolic sine where x is in radians</td>
</tr>
<tr>
<td>SQRT(x)</td>
<td>the square root of x</td>
</tr>
<tr>
<td>TAN(x)</td>
<td>tangent where x is in radians</td>
</tr>
<tr>
<td>TANH(x)</td>
<td>tangent where x is in radians</td>
</tr>
</tbody>
</table>
# Numeric Intrinsic Functions

Summary,

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS(a)</td>
<td>absolute value</td>
</tr>
<tr>
<td>AINT(a)</td>
<td>truncates a to whole REAL number</td>
</tr>
<tr>
<td>ANINT(a)</td>
<td>nearest whole REAL number</td>
</tr>
<tr>
<td>CEILING(a)</td>
<td>smallest INTEGER greater than or equal to REAL number</td>
</tr>
<tr>
<td>CMPLX(x,y)</td>
<td>convert to COMPLEX</td>
</tr>
<tr>
<td>DBLE(x)</td>
<td>convert to DOUBLE PRECISION</td>
</tr>
<tr>
<td>DIM(x,y)</td>
<td>positive difference</td>
</tr>
<tr>
<td>FLOOR(a)</td>
<td>biggest INTEGER less than or equal to real number</td>
</tr>
<tr>
<td>INT(a)</td>
<td>truncates a into an INTEGER</td>
</tr>
<tr>
<td>MAX(a1,a2,a3,...)</td>
<td>the maximum value of the arguments</td>
</tr>
<tr>
<td>MIN(a1,a2,a3,...)</td>
<td>the minimum value of the arguments</td>
</tr>
<tr>
<td>MOD(a,p)</td>
<td>remainder function</td>
</tr>
<tr>
<td>MODULO(a,p)</td>
<td>modulo function</td>
</tr>
<tr>
<td>NINT(x)</td>
<td>nearest INTEGER to a REAL number</td>
</tr>
<tr>
<td>REAL(a)</td>
<td>converts to the equivalent REAL value</td>
</tr>
<tr>
<td>SIGN(a,b)</td>
<td>transfer of sign — ABS(a)*(b/ABS(b))</td>
</tr>
</tbody>
</table>
## Character Intrinsic Functions

Summary,

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACHAR(i)</td>
<td>(i^{th}) character in ASCII collating sequence</td>
</tr>
<tr>
<td>ADJUSTL(str)</td>
<td>adjust left</td>
</tr>
<tr>
<td>ADJUSTR(str)</td>
<td>adjust right</td>
</tr>
<tr>
<td>CHAR(i)</td>
<td>(i^{th}) character in processor collating sequence</td>
</tr>
<tr>
<td>IACHAR(ch)</td>
<td>position of character in ASCII collating sequence</td>
</tr>
<tr>
<td>ICHAR(ch)</td>
<td>position of character in processor collating sequence</td>
</tr>
<tr>
<td>INDEX(str, substr)</td>
<td>starting position of substring</td>
</tr>
<tr>
<td>LEN(str)</td>
<td>Length of string</td>
</tr>
<tr>
<td>LEN_TRIM(str)</td>
<td>Length of string without trailing blanks</td>
</tr>
<tr>
<td>LGE(str1, str2)</td>
<td>lexically .GE.</td>
</tr>
<tr>
<td>LGT(str1, str2)</td>
<td>lexically .GT.</td>
</tr>
<tr>
<td>LLE(str1, str2)</td>
<td>lexically .LE.</td>
</tr>
<tr>
<td>LLT(str1, str2)</td>
<td>lexically .LT.</td>
</tr>
<tr>
<td>REPEAT(str, i)</td>
<td>repeat (i) times</td>
</tr>
<tr>
<td>SCAN(str, set)</td>
<td>scan a string for characters in a set</td>
</tr>
<tr>
<td>TRIM(str)</td>
<td>remove trailing blanks</td>
</tr>
<tr>
<td>VERIFY(str, set)</td>
<td>verify the set of characters in a string</td>
</tr>
</tbody>
</table>
PRINT Statement

This is the simplest form of directing unformatted data to the standard output channel, for example,

PROGRAM Owt
  IMPLICIT NONE
  CHARACTER(LEN=*) , PARAMETER :: &
      long_name = "Llanfair...gogogoch"
  REAL :: x, y, z
  LOGICAL :: lacigel
  x = 1; y = 2; z = 3
  lacigel = (y .eq. x)
  PRINT*, long_name
  PRINT*, "Spock says ""illogical&
       &Captain"
  PRINT*, "X = ",x," Y = ",y," Z = ",z
  PRINT*, "Logical val: ",lacigel
END PROGRAM Owt

produces on the screen,

Llanfair...gogogoch
Spock says "illogical Captain"
X = 1.000  Y = 2.000  Z = 3.000
Logical val:  F
**PRINT Statement**

Note,

- each PRINT statement begins a new line;

- the PRINT statement can transfer any object of intrinsic type to the standard output;

- strings may be delimited by the double or single quote symbols, " and ";

- two occurrences of the string delimiter inside a string produce one occurrence on output;
READ Statement

READ accepts unformatted data from the standard input channel, for example, if the type declarations are the same as for the PRINT example,

    READ*, long_name
    READ*, x, y, z
    READ*, lacigol

accepts

    Llanphairphwyll...gogogoch
    0.4 5. 1.0e12
    T

Note,

- each READ statement reads from a newline;

- the READ statement can transfer any object of intrinsic type from the standard input;
Lecture 4:
Arrays
Arrays

Arrays (or matrices) hold a collection of different values at the same time. Individual elements are accessed by subscripting the array.

A 15 element array can be visualised as:

```
1 2 3
```

And a 5 × 3 array as:

```
1,1 1,2 1,3
2,1 2,2 2,3
3,1 3,2 3,3
4,1 4,2 4,3
5,1 5,2 5,3
```

Every array has a type and each element holds a value of that type.
Array Terminology

Examples of declarations:

\[
\text{REAL, DIMENSION(15)} :: X \\
\text{REAL, DIMENSION(1:5,1:3)} :: Y, Z
\]

The above are explicit-shape arrays.

Terminology:

- **rank** — number of dimensions.
  Rank of \( X \) is 1; rank of \( Y \) and \( Z \) is 2.

- **bounds** — upper and lower limits of indices.
  Bounds of \( X \) are 1 and 15; Bound of \( Y \) and \( Z \) are 1 and 5 and 1 and 3.

- **extent** — number of elements in dimension;
  Extent of \( X \) is 15; extents of \( Y \) and \( Z \) are 5 and 3.

- **size** — total number of elements.
  Size of \( X, Y \) and \( Z \) is 15.

- **shape** — rank and extents;
  Shape of \( X \) is 15; shape of \( Y \) and \( Z \) is 5,3.

- **conformable** — same shape.
  \( Y \) and \( Z \) are conformable.
Declarations

Literals and constants can be used in array declarations,

```
REAL, DIMENSION(100) :: R
REAL, DIMENSION(1:10,1:10) :: S
REAL :: T(10,10)
REAL, DIMENSION(-10:-1) :: X
INTEGER, PARAMETER :: lda = 5
REAL, DIMENSION(0:lda-1) :: Y
REAL, DIMENSION(1+lda*lda,10) :: Z
```

- default lower bound is 1,
- bounds can begin and end anywhere,
- arrays can be zero-sized (if lda = 0),
Visualisation of Arrays

REAL, DIMENSION(15) :: A
REAL, DIMENSION(-4:0,0:2) :: B
REAL, DIMENSION(5,3) :: C
REAL, DIMENSION(0:4,0:2) :: D

Individual array elements are denoted by subscripting the array name by an INTEGER, for example, \( A(7) \) 7th element of \( A \), or \( C(3,2) \), 3 elements down, 2 across.
Array Conformance

Arrays or sub-arrays must conform with all other objects in an expression:

- a scalar conforms to an array of any shape with the same value for every element:
  \[ C = 1.0 \quad \text{is valid} \]

- two array references must conform in their shape.
  Using the declarations from before:

\[ \begin{aligned}
\text{C = D} & \quad \text{Valid} \\
\text{B = A} & \quad \text{Invalid}
\end{aligned} \]

A and B have the same size but have different shapes so cannot be directly equated.
Array Element Ordering

Organisation in memory:

- Fortran 90 does not specify anything about how arrays should be located in memory. **It has no storage association.**

- Fortran 90 does define an array element ordering for certain situations which is of column major form,

The array is conceptually ordered as:

\[ C(1,1), C(2,1), \ldots, C(5,1), C(1,2), C(2,2), \ldots, C(5,3) \]
Can reference:

- whole arrays
  - $A = 0.0$
    - sets whole array $A$ to zero.
  - $B = C + D$
    - adds $C$ and $D$ then assigns result to $B$.

- elements
  - $A(1) = 0.0$
    - sets one element to zero,
  - $B(0,0) = A(3) + C(5,1)$
    - sets an element of $B$ to the sum of two other elements.

- array sections
  - $A(2:4) = 0.0$
    - sets $A(2)$, $A(3)$ and $A(4)$ to zero,
  - $B(-1:0,1:2) = C(1:2,2:3) + 1.0$
    - adds one to the subsection of $C$ and assigns to the subsection of $B$. 

71
Whole Array Expressions

Arrays can be treated like a single variable in that:

- can use intrinsic operators between conformable arrays (or sections),

\[ B = C \times D - B^{**2} \]

this is equivalent to concurrent execution of:

\[ B(-4,0) = C(1,1) \times D(0,0) - B(-4,0)^{**2} \text{ ! in } || \]
\[ B(-3,0) = C(2,1) \times D(1,0) - B(-3,0)^{**2} \text{ ! in } || \]
\[ \ldots \]
\[ B(-4,1) = C(1,2) \times D(0,1) - B(-4,1)^{**2} \text{ ! in } || \]
\[ \ldots \]
\[ B(0,2) = C(5,3) \times D(4,2) - B(0,2)^{**2} \text{ ! in } || \]

- elemental intrinsic functions can be used,

\[ B = \text{SIN}(C) + \text{COS}(D) \]

the function is applied element by element.
Array Sections — Visualisation

Given,

REAL, DIMENSION(1:6,1:8) :: P

Consider the following assignments,

- \( P(1:3,1:4) = P(1:6:2,1:8:2) \) and \( P(1:3,1:4) = 1.0 \) are valid.

- \( P(2:8:2,1:7:3) = P(1:3,1:4) \) and \( P(2:6:2,1:7:3) = P(2:5,7) \) are not.

- \( P(2:5,7) \) is a 1D section (scalar in dimension 2) whereas \( P(2:5,7:7) \) is a 2D section.
**Array Sections**

**subscript-triplets** specify sub-arrays. The general form is:

```
[< bound1 >]:[< bound2 >][:< stride >]
```

The section starts at `< bound1 >` and ends at or before `< bound2 >`. `< stride >` is the increment by which the locations are selected.

`< bound1 >`, `< bound2 >` and `< stride >` must all be scalar integer expressions. Thus

```
A(:)        ! the whole array
A(3:9)      ! A(m) to A(n) in steps of 1
A(3:9:1)    ! as above
A(m:n)      ! A(m) to A(n)
A(m:n:k)    ! A(m) to A(n) in steps of k
A(8:3:-1)   ! A(8) to A(3) in steps of -1
A(8:3)      ! A(8) to A(3) step 1 => Zero size
A(m:)       ! from A(m) to default UPB
A(:n)       ! from default LWB to A(n)
A(::2)      ! from default LWB to UPB step 2
A(m:m)      ! 1 element section
A(m)        ! scalar element - not a section
```

are all valid sections.
Array I/O

The conceptual ordering of array elements is useful for defining the order in which array elements are output. If A is a 2D array then:

\[
\text{PRINT*}, A
\]

would produce output in the order:

\[
A(1,1), A(2,1), A(3,1), \ldots, A(1,2), A(2,2), \ldots
\]

\[
\text{READ*}, A
\]

would assign to the elements in the above order.

This order could be changed by using intrinsic functions such as RESHAPE, TRANSPOSE or CSHIFT.
Array I/O Example

Consider the matrix A:

\[
\begin{array}{ccc}
1 & 4 & 7 \\
2 & 5 & 8 \\
3 & 6 & 9 \\
\end{array}
\]

The following PRINT statements produce on the screen,

```
... 
PRINT*, 'Array element =', a(3,2) 
PRINT*, 'Array section =', a(:,1) 
PRINT*, 'Sub-array =', a(:,2:2) 
PRINT*, 'Whole Array =', a 
PRINT*, 'Array Transp'’d =', TRANSPOSE(a) 
END PROGRAM Owit
```

Array element = 6 
Array section = 1 2 3 
Sub-array = 1 2 4 5 
Whole Array = 1 2 3 4 5 6 7 8 9 
Array Transposed = 1 4 7 2 5 8 3 6 9
Array Inquiry Intrinsics

These are often useful in procedures, consider the declaration:

```
REAL, DIMENSION(-10:10,23,14:28) :: A
```

- **LBOUND(SOURCE[,DIM])** — lower bounds of an array (or bound in an optionally specified dimension).
  - LBOUND(A) is (/ -10,1,14/) (array);
  - LBOUND(A,1) is -10 (scalar).

- **UBOUND(SOURCE[,DIM])** — upper bounds of an array (or bound in an optionally specified dimension).

- **SHAPE(SOURCE)** — shape of an array,
  - SHAPE(A) is (/21,23,15/) (array);
  - SHAPE(/4/) is (/1/) (array).

- **SIZE(SOURCE[,DIM])** — total number of array elements (in an optionally specified dimension),
  - SIZE(A,1) is 21;
  - SIZE(A) is 7245.

- **ALLOCATED(SOURCE)** — array allocation status;
Array Constructors

Used to give arrays or sections of arrays specific values. For example,

```
IMPLICIT NONE
INTEGER         :: i
INTEGER, DIMENSION(10) :: ints
CHARACTER(len=5), DIMENSION(3) :: colours
REAL, DIMENSION(4) :: heights
```

```
heights = (/5.10, 5.6, 4.0, 3.6/)
colours = (/’RED’, ’GREEN’, ’BLUE’/)
! note padding so strings are 5 chars
ints    = (/ 100, (i, i=1,8), 100 /)
```

- constructors and array sections must conform.
- must be 1D.
- for higher rank arrays use RESHAPE intrinsic.
- (i, i=1,8) is an implied DO and is 1,2,..,8, it is possible to specify a stride.
**The RESHAPE Intrinsic Function**

**RESHAPE** is a general intrinsic function which delivers an array of a specific shape:

\[
\text{RESHAPE(SOURCE, SHAPE)}
\]

For example,

\[
A = \text{RESHAPE}((/1,2,3,4/),(/2,2/))
\]

A is filled in array element order and looks like:

\[
\begin{array}{cc}
1 & 3 \\
2 & 4
\end{array}
\]

Visualisation,

\[
\begin{array}{c}
1 & 2 & 3 & 4 \\
\end{array}
\rightarrow
\begin{array}{c}
1 & 3 \\
2 & 4
\end{array}
\]

RESHAPE
Arrays can be initialised

    INTEGER, DIMENSION(4) :: solution = (/2,3,4,5/)
    CHARACTER(LEN=*) , DIMENSION(3) :: &
      lights = (/’RED ’,’BLUE ’,’GREEN’/) 

In the second statement all strings must be same length.

Named array constants may also be created:

    INTEGER, DIMENSION(3), PARAMETER :: &
      Unit_vec = (/1,1,1/)  
    REAL, DIMENSION(3,3), PARAMETER :: &
      unit_matrix = &
        RESHAPE((/1,0,0,0,1,0,0,0,1/),(/3,3/))
**Allocatable Arrays**

Fortran 90 allows arrays to be created on-the-fly; these are known as *deferred-shape* arrays:

- **Declaration:**

  ```fortran
  INTEGER, DIMENSION(:,), ALLOCATABLE :: ages  ! 1D
  REAL, DIMENSION(:,,:), ALLOCATABLE :: speed  ! 2D
  ```

  Note ALLOCATABLE attribute and fixed rank.

- **Allocation:**

  ```fortran
  READ*, isize
  ALLOCATE(ages(isize), STAT=ierr)
  IF (ierr /= 0) PRINT*, "ages : Allocation failed"

  ALLOCATE(speed(0:isize-1,10),STAT=ierr)
  IF (ierr /= 0) PRINT*, "speed : Allocation failed"
  ```

  - The optional STAT= field reports on the success of the storage request. If the INTEGER variable ierr is zero the request was successful otherwise it failed.
Deallocating Arrays

Heap storage can be reclaimed using the DEALLOCATE statement:

\[
\text{IF (ALLOCATED(ages)) DEALLOCATE(ages,STAT= ierr)}
\]

- it is an error to deallocate an array without the ALLOCATE attribute or one that has not been previously allocated space,

- there is an intrinsic function, ALLOCATED, which returns a scalar LOGICAL values reporting on the status of an array,

- the STAT= field is optional but its use is recommended,

- if a procedure containing an allocatable array which does not have the SAVE attribute is exited without the array being DEALLOCATED then this storage becomes inaccessible.
**Masked Array Assignment — Where Statement**

This is achieved using *WHERE*:

WHERE (I .NE. 0) A = B/I

the LHS of the assignment must be array valued and the mask, (the logical expression,) and the RHS of the assignment must all conform;

For example, if

\[
B = \begin{pmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \end{pmatrix}
\]

and,

\[
I = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}
\]

then

\[
A = \begin{pmatrix} 0.5 & \cdot \\ \cdot & 2.0 \end{pmatrix}
\]

Only the indicated elements, corresponding to the non-zero elements of I, have been assigned to.
Where Construct

- There is a block form of masked assignment:

```plaintext
WHERE(A > 0.0)
  B = LOG(A)
  C = SQRT(A)
ELSEWHERE
  B = 0.0 ! C is NOT changed
ENDWHERE
```

- The mask must conform to the RHS of each assignment; A, B, and C must conform;

- `WHERE ... END WHERE` is *not* a control construct and cannot currently be nested;

- The execution sequence is as follows: evaluate the mask, execute the `WHERE` block (in full) then execute the `ELSEWHERE` block;

- The separate assignment statements are executed sequentially but the individual elemental assignments within each statement are (conceptually) executed in parallel.
**Vector-valued Subscripts**

A 1D array can be used to subscript an array in a dimension. Consider:

\[
\text{INTEGER, DIMENSION}(5) :: V = (/1,4,8,12,10/)
\]
\[
\text{INTEGER, DIMENSION}(3) :: W = (/1,2,2/)
\]

- \( \text{A}(V) \) is \( \text{A}(1), \text{A}(4), \text{A}(8), \text{A}(12), \text{and} \ \text{A}(10). \)

- the following are valid assignments:

\[
\text{A}(V) = 3.5 \\
\text{C}(1:3,1) = \text{A}(W)
\]

- it would be invalid to assign values to \( \text{A}(W) \) as \( \text{A}(2) \) is referred to twice.

- only 1D vector subscripts are allowed, for example,

\[
\text{A}(1) = \text{SUM(C(V,W))}
\]
Random Number Intrinsic

- `RANDOM_NUMBER(HARVEST)` will return a scalar (or array of) pseudorandom number(s) in the range $0 \leq x < 1$.

  For example,

  ```fortran
  REAL :: HARVEST
  REAL, DIMENSION(10,10) :: HARVEYS
  CALL RANDOM_NUMBER(HARVEST)
  CALL RANDOM_NUMBER(HARVEYS)
  ```

- `RANDOM_SEED([SIZE=< int >])` finds the size of the seed.

- `RANDOM_SEED([PUT=< array >])` seeds the random number generator.

  ```fortran
  CALL RANDOM_SEED(SIZE=isze)
  CALL RANDOM_SEED(PUT=IArr(1:isze))
  ```
Vector and Matrix Multiply Intrinsics

There are two types of intrinsic matrix multiplication:

- **DOT_PRODUCT(VEC1, VEC2)** — inner (dot) product of two rank 1 arrays.
  
  For example,
  
  \[
  DP = DOT_PRODUCT(A,B)
  \]
  
  is equivalent to:
  
  \[
  DP = A(1)\times B(1) + A(2)\times B(2) + \ldots
  \]
  
  For LOGICAL arrays, the corresponding operation is a logical .AND..

- **MATMUL(MAT1, MAT2)** — ‘traditional’ matrix-matrix multiplication:
  
  - if MAT1 has shape \((n, m)\) and MAT2 shape \((m, k)\) then the result has shape \((n, k)\);
  
  - if MAT1 has shape \((m)\) and MAT2 shape \((m, k)\) then the result has shape \((k)\);
  
  - if MAT1 has shape \((n, m)\) and MAT2 shape \((m)\) then the result has shape \((n)\);
  
  For LOGICAL arrays, the corresponding operation is a logical .AND..
**Maximum and Minimum Intrinsics**

There are two intrinsics in this class:

- **MAX(Source1,Source2[,...])** — maximum values over all source objects

- **MIN(Source1,Source2[,...])** — minimum values over all source objects

Scan from left to right, choose **first** occurrence if there are duplicates

```
MAX(X)
```

```
7 9 -2 4 8 10 2 7 10 2 1
```

- **MAX(1,2,3)** is 3
- **MIN((/-1,2/),(-3,4/))** is (/-3,2/)
- **MAX((/-1,2/),(-3,4/))** is (/-1,4/)

88
Array Location Intrinsics

There are two intrinsics in this class:

- **MINLOC**(*SOURCE[,MASK]*) — Location of a minimum value in an array under an optional mask.

- **MAXLOC**(*SOURCE[,MASK]*) — Location of a maximum value in an array under an optional mask.

A 1D example,

\[
\text{MAXLOC}(X) = (/6/)
\]

\[
\begin{array}{cccccccc}
7 & 9 & -2 & 4 & 8 & 10 & 2 & 7 & 10 & 2 & 1
\end{array}
\]

A 2D example. If

\[
\text{Array} = \begin{pmatrix}
0 & -1 & 1 & 6 & -4 \\
1 & -2 & 5 & 4 & -3 \\
3 & 8 & 3 & -7 & 0
\end{pmatrix}
\]

then

- **MINLOC**(*Array*) is (/3,4/)

- **MAXLOC**(*Array,Array.LE.7*) is (/1,4/)

- **MAXLOC**(*MAXLOC(*Array,Array.LE.7*)) is (/2/) (array valued).
**Array Reduction Intrinsic**

- `PRODUCT(SOURCE[,DIM][,MASK])` — product of array elements (in an optionally specified dimension under an optional mask);
- `SUM(SOURCE[,DIM][,MASK])` — sum of array elements (in an optionally specified dimension under an optional mask).

The following 1D example demonstrates how the 11 values are reduced to just one by the `SUM` reduction:

\[
\text{SUM}(W) = 58
\]

Consider this 2D example, if

\[
A = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}
\]

- `PRODUCT(A)` is 720
- `PRODUCT(A,DIM=1)` is `(2, 12, 30)`
- `PRODUCT(A,DIM=2)` is `(15, 48)`
Array Reduction Intrinsics (Cont’d)

These functions operate on arrays and produce a result with less dimensions that the source object:

- **ALL(MASK[,DIM])**— .TRUE. if all values are .TRUE., (in an optionally specified dimension);

- **ANY(MASK[,DIM])**— .TRUE. if any values are .TRUE., (in an optionally specified dimension);

- **COUNT(MASK[,DIM])**— number of .TRUE. elements in an array, (in an optionally specified dimension);

- **MAXVAL(SOURCE[,DIM][,MASK])**— maximum Value in an array (in an optionally specified dimension under an optional mask);

- **MINVAL(SOURCE[,DIM][,MASK])**— minimum value in an array (in an optionally specified dimension under an optional mask);

If DIM is absent or the source array is of rank 1 then the result is scalar, otherwise the result is of rank \( n - 1 \).
Lecture 5:
Procedures
Program Units

Fortran 90 has two main program units

- **main PROGRAM**, 
  the place where execution begins and where control should eventually return before the program terminates. May contain procedures.

- **MODULE**, 
  a program unit which can contain procedures and declarations. It is intended to be attached to any other program unit where the entities defined within it become accessible.

There are two types of procedures:

- **SUBROUTINE**, 
  a parameterised named sequence of code which performs a specific task and can be invoked from within other program units.

- **FUNCTION**, 
  as a SUBROUTINE but returns a result in the function name (of any specified type and kind).
Main Program Syntax

PROGRAM Main
! ...
CONTAINS ! Internal Procs

SUBROUTINE Sub1(..)
! Executable stmts
END SUBROUTINE Sub1
! etc.
FUNCTION Funkyn(...) 
! Executable stmts
END FUNCTION Funkyn

END PROGRAM Main

[ PROGRAM [ <main program name>] ]
<declaration of local objects>
...
<executable stmts>
...
[ CONTAINS
  <internal procedure definitions> ]
END [ PROGRAM [ <main program name> ] ]
Program Example

PROGRAM Main
  IMPLICIT NONE
  REAL :: x
  READ*, x
  PRINT*, FLOOR(x) ! Intrinsic
  PRINT*, Negative(x)
CONTAINS
  REAL FUNCTION Negative(a)
  REAL, INTENT(IN) :: a
  Negative = -a
  END FUNCTION Negative
END PROGRAM Main
Introduction to Procedures

The first question should be: “Do we really need to write a procedure?”

Functionality often exists,

- intrinsics, Fortran 90 has 113,

- libraries, for example, NAG f190 Numerical Library has 300+, BLAS, IMSL, LaPACK, Uniras

- modules, number growing, many free! See WWW.

Library routines are usually very fast, sometimes even faster than Intrinsics!
Subroutines

Consider the following example,

```
PROGRAM Thingy
   IMPLICIT NONE
   ..... 
   CALL OutputFigures(NumberSet)
   ..... 
   CONTAINS
   SUBROUTINE OutputFigures(Numbers)
      REAL, DIMENSION(:), INTENT(IN) :: Numbers
      PRINT*, "Here are the figures", Numbers
   END SUBROUTINE OutputFigures
END PROGRAM Thingy
```

Internal subroutines lie between CONTAINS and END PROGRAM statements and have the following syntax

```
SUBROUTINE < procname >[ (< dummy args >) ]
   < declaration of dummy args >
   < declaration of local objects >
   ...
   < executable stmts >
END [ SUBROUTINE [< procname > ] ]
```

Note that, in the example, the IMPLICIT NONE statement applies to the whole program including the SUBROUTINE.
Functions

Consider the following example,

```fortran
PROGRAM Thingy
  IMPLICIT NONE
  ....
  PRINT*, F(a,b)
  ....
  CONTAINS
    REAL FUNCTION F(x,y)
    REAL, INTENT(IN) :: x,y
    F = SQRT(x*x + y*y)
    END FUNCTION F
END PROGRAM Thingy
```

Functions also lie between CONTAINS and END PROGRAM statements. They have the following syntax:

```fortran
[< prefix >] FUNCTION < procname > ( [< dummyargs >])
  < declaration of dummy args >
  < declaration of local objects >
  ...
  < executable stmts, assignment of result >
END FUNCTION [ < procname > ]
```

It is also possible to declare the function type in the declarations area instead of in the header.
Argument Association

Recall, on the SUBROUTINE slide we had an invocation:

CALL OutputFigures(NumberSet)

and a declaration,

SUBROUTINE OutputFigures(Numbers)

NumberSet is an actual argument and is argument associated with the dummy argument Numbers.

For the above call, in OutputFigures, the name Numbers is an alias for NumberSet. Likewise, consider,

PRINT*, F(a,b)

and

REAL FUNCTION F(x,y)

The actual arguments a and b are associated with the dummy arguments x and y.

If the value of a dummy argument changes then so does the value of the actual argument.
Local Objects

In the following procedure

```
SUBROUTINE Madras(i,j)
    INTEGER, INTENT(IN) :: i, j
    REAL :: a
    REAL, DIMENSION(i,j):: x
```

a, and x are know as local objects. They:

- are created each time a procedure is invoked,
- are destroyed when the procedure completes,
- do not retain their values between calls,
- do not exist in the programs memory between calls.

x will probably have a different size and shape on each call.

The space usually comes from the programs stack.
Argument Intent

Hints to the compiler can be given as to whether a dummy argument will:

- only be referenced — INTENT(IN);
- be assigned to before use — INTENT(OUT);
- be referenced and assigned to — INTENT(INOUT);

```fortran
SUBROUTINE example(arg1,arg2,arg3)
    REAL, INTENT(IN) :: arg1
    INTEGER, INTENT(OUT) :: arg2
    CHARACTER, INTENT(INOUT) :: arg3
    REAL :: r
    r = arg1*ICHAR(arg3)
    arg2 = ANINT(r)
    arg3 = CHAR(MOD(127,arg2))
END SUBROUTINE example
```

The use of INTENT attributes is recommended as it:

- allows good compilers to check for coding errors,
- facilitates efficient compilation and optimisation.

Note: if an actual argument is ever a literal, then the corresponding dummy must be INTENT(IN).
Scoping Rules

Fortran 90 is *not* a traditional block-structured language:

- the *scope* of an entity is the range of program unit where it is visible and accessible;

- internal procedures can inherit entities by *host association*.

- objects declared in modules can be made visible by *use-association* (the USE statement) — useful for global data;
Consider,

```fortran
PROGRAM CalculatePay
  IMPLICIT NONE
  REAL :: Pay, Tax, Delta
  INTEGER :: NumberCalcsDone = 0
  Pay = ...;  Tax = ... ; Delta = ...
  CALL PrintPay(Pay,Tax)
  Tax = NewTax(Tax,Delta)
  ....
END CONTAINS
SUBROUTINE PrintPay(Pay,Tax)
  REAL, INTENT(IN) :: Pay, Tax
  REAL :: TaxPaid
  TaxPaid = Pay * Tax
  PRINT*, TaxPaid
  NumberCalcsDone = NumberCalcsDone + 1
END SUBROUTINE PrintPay
SUBROUTINE NewTax(Tax,Delta)
  REAL, INTENT(IN) :: Tax, Delta
  NewTax = Tax + Delta*Tax
  NumberCalcsDone = NumberCalcsDone + 1
END FUNCTION NewTax
END PROGRAM CalculatePay
```

Here, NumberCalcsDone is a global variable. It is available in all procedures by host association.
Scope of Names

Consider the following example,

PROGRAM Proggie
  IMPLICIT NONE
  REAL :: A, B, C
  CALL sub(A)
  CONTAINS
    SUBROUTINE Sub(D)
      REAL :: D ! D is dummy (alias for A)
      REAL :: C ! local C (diff from Proggie’s C)
      C = A**3 ! A cannot be changed
      D = D**3 + C ! D can be changed
      B = C ! B from Proggie gets new value
    END SUBROUTINE Sub
  END PROGRAM Proggie

In Sub, as A is argument associated it may not be have its value changed but may be referenced.

C in Sub is totally separate from C in Proggie, changing its value in Sub does not alter the value of C in Proggie.
SAVE Attribute and the SAVE Statement

SAVE attribute can be:

- applied to a specified variable. NumInvocations is initialised on first call and retains its new value between calls,

```
SUBROUTINE Barmy(arg1,arg2)
   INTEGER, SAVE :: NumInvocations = 0
   NumInvocations = NumInvocations + 1
```

- applied to the whole procedure, and applies to all local objects.

```
SUBROUTINE Mad(arg1,arg2)
   REAL :: saved
   SAVE
   REAL :: saved_an_all
```

Variables with the SAVE attribute are static objects.

Clearly, SAVE has no meaning in the main program.
Keyword Arguments

Can supply dummy arguments in any order using keyword arguments, for example, given

```
SUBROUTINE axis(x0,y0,l,min,max,i)
    REAL, INTENT(IN) :: x0,y0,l,min,max
    INTEGER, INTENT(IN) :: i
    ... ...
END SUBROUTINE axis
```

can invoke procedure:

- using positional argument invocation:

```
CALL AXIS(0.0,0.0,0,100.0,0.1,1.0,10)
```

- using keyword arguments:

```
CALL AXIS(0.0,0.0,Max=1.0,Min=0.1, &
          L=100.0,I=10)
```

The names are from the dummy arguments.
Keyword Arguments

Keyword arguments:

- allow arguments to be specified in any order.

- makes it easy to add an extra argument — no need to modify any calls.

- helps improve the readability of the program.

- are used when a procedure has optional arguments.

Note: once a keyword is used all subsequent arguments must be keyword arguments.
Optional Arguments

Optional arguments:

- allow defaults to be used for missing arguments;
- make some procedures easier to use.

Also:

- once an argument has been omitted all subsequent arguments must be keyword arguments,
- the `PRESENT` intrinsic can be used to check for missing arguments.
Optional Arguments Example

For example, consider the internal procedure,

```
SUBROUTINE SEE(a,b)
    REAL, INTENT(IN), OPTIONAL :: a
    INTEGER, INTENT(IN), OPTIONAL :: b
    REAL :: ay; INTEGER :: bee
    ay = 1.0; bee = 1 ! defaults
    IF(PRESENT(a)) ay = a
    IF(PRESENT(b)) bee = b
    ...
```

both a and b have the optional attribute so SEE can be called in the following ways

```
CALL SEE()
CALL SEE(1.0,1); CALL SEE(b=1,a=1.0) ! same
CALL SEE(1.0); CALL SEE(a=1.0) ! same
CALL SEE(b=1)
```
Lecture 6:
More Procedures
**Dummy Array Arguments**

There are two main types of dummy array argument:

- **explicit-shape** — all bounds specified;

  ```
  REAL, DIMENSION(8,8), INTENT(IN) :: expl_shape
  ```

  The actual argument that becomes associated with an explicit-shape dummy must conform in size and shape.

- **assumed-shape** — no bounds specified, all inherited from the actual argument;

  ```
  REAL, DIMENSION(:,,:), INTENT(IN) :: ass_shape
  ```

  An explicit interface *must* be provided.

- dummy arguments cannot be (unallocated) ALLOCATABLE arrays.
Explicit-shape Arrays

A dummy argument that is an explicit-shape array must conform in size and shape to the associated actual argument.

PROGRAM Main
IMPLICIT NONE
INTEGER, DIMENSION(8,8) :: A1
INTEGER, DIMENSION(64) :: A2
INTEGER, DIMENSION(16,32) :: A3

... 
CALL subby(A1) ! OK
CALL subby(A2) ! ERROR
CALL subby(A3(:,2,:,4)) ! OK
CALL subby(RESHAPE(A2,(/8,8/))) ! OK

... 
CONTAINS
SUBROUTINE subby(expl_shape)
  INTEGER, DIMENSION(8,8) :: expl_shape
...
END SUBROUTINE subby
END PROGRAM Main
**Assumed-shape Arrays**

Should declare dummy arrays as assumed-shape arrays:

```fortran
PROGRAM Main
IMPLICIT NONE
   REAL, DIMENSION(40)   :: X
   REAL, DIMENSION(40,40) :: Y
   ...
   CALL gimlet(X,Y)
   CALL gimlet(X(1:39:2),Y(2:4,4:4))
   CALL gimlet(X(1:39:2),Y(2:4,4)) ! invalid
CONTAINS
   SUBROUTINE gimlet(a,b)
      REAL, INTENT(IN)   :: a(:), b(:,:)
   ...
   END SUBROUTINE gimlet
END PROGRAM
```

Note:

- the actual arguments cannot be a vector subscripted array,
- the actual argument cannot be an assumed-size array.
- in the procedure, bounds begin at 1.
Other arrays can depend on dummy arguments, these are called *automatic* arrays and:

- their size is determined by dummy arguments,
- they cannot have the SAVE attribute (or be initialised);

Consider,

```plaintext
PROGRAM Main
    IMPLICIT NONE
    INTEGER :: IX, IY
    ....
    CALL une_bus_riot(IX,2,3)
    CALL une_bus_riot(IY,7,2)
END PROGRAM
```

The SIZE intrinsic or dummy arguments can be used to declare automatic arrays. A1 and A2 may have different sizes for different calls.
SAVE Attribute and Arrays

Consider,

SUBROUTINE sub1(dim)
  INTEGER, INTENT(IN) :: dim
  REAL, ALLOCATABLE, DIMENSION(:,,:), SAVE :: X
  REAL, DIMENSION(dim) :: Y
...
  IF (.NOT.ALLOCATED(X)) ALLOCATE(X(20,20))

As X has the SAVE attribute it will retain its allocation status between calls otherwise it would disappear.

As Y depends on a dummy argument it cannot be given SAVE attribute.
Explicit Length Character Dummy Arguments

An explicit-length dummy CHARACTER argument must match the actual in kind and rank (like explicit length arrays), however, it can be shorter than the actual.

```fortran
PROGRAM Mian
  IMPLICIT NONE
  CHARACTER(LEN=10), DIMENSION(10) :: wurd
  ...
  CALL char_example(wurd(3))
  CALL char_example(wurd(6:))
CONTAINS
  SUBROUTINE char_example(wird,werds)
    CHARACTER(LEN=10), :: wird
    CHARACTER(LEN=10), :: werds(:)
    ...
  END SUBROUTINE char_example
END PROGRAM Mian
```
Assumed Length Character Dummy Arguments

- CHARACTER dummies can inherit their length specifier (LEN=) from the actual argument;

- the kind and rank must still match;

- such an argument is an assumed-length character dummy;

PROGRAM Main
IMPLICIT NONE
...
CHARACTER(LEN=10) :: vari1
CHARACTER(LEN=20) :: vari2
CALL char_lady vari1
CALL char_lady vari2
CONTAINS
SUBROUTINE char_lady (word)
CHARACTER(LEN=*) , INTENT(IN) :: word
...
PRINT*, "Length of arg is", LEN(word)
END SUBROUTINE char_lady
END PROGRAM Main

will produce,

Length of arg is 10
Length of arg is 20
Array-valued Functions

Functions can return arrays, for example,

```
PROGRAM Maian
  IMPLICIT NONE
  INTEGER, PARAMETER :: m = 6
  INTEGER, DIMENSION(M,M) :: im1, im2
  ...
  IM2 = funnie(IM1,1) ! invoke
CONTAINS
  FUNCTION funnie(ima,scal)
    INTEGER, INTENT(IN) :: ima(:,:)
    INTEGER, INTENT(IN) :: scal
    INTEGER, DIMENSION(SIZE(ima,1),SIZE(ima,2)) &
      :: funnie
    funnie(:, :) = ima(:, :) * scal
  END FUNCTION funnie
END PROGRAM
```

Note how the `DIMENSION` attribute cannot appear in the function header.
Character-valued Functions

Can define a function that returns a CHARACTER string of a given length:

FUNCTION reverse(word)
    CHARACTER(LEN=*) , INTENT(IN) :: word
    CHARACTER(LEN=LEN(word)) :: reverse
    INTEGER :: lw
    lw = LEN(word)
    ! reverse characters
    DO I = 1,lw
      reverse(lw-I+1:lw-I+1) = word(I:I)
    END DO
END FUNCTION reverse

Here the length of the function result is determined by the dummy argument.
Side Effect Functions

If, in

```
rezzy = funky1(a,b,c) + funky2(a,b,c)
```

`funky1` and `funky2` modify the value of, `a` then the order of execution could be important. Consider:

```
INTEGER FUNCTION funky1(a,b,c)
...
a = a*a
funky1 = a/b
```

and

```
INTEGER FUNCTION funky2(a,b,c)
...
a = a*2
funky2 = a/c
```

With `a=4`, `b=2` and `c=4` the following happens:

- If `funky1` executed first then `rezzy=8+8=16`
- If `funky2` executed first then `rezzy=2+32=34`

A properly constructed function should not change its arguments or any global entities.
Recursive Procedures

In Fortran 90 recursion is supported as a feature.

- recursive procedures call themselves (either directly or indirectly),

- recursion is a neat technique

- recursion may incur certain efficiency overheads,

- recursive procedures must be explicitly declared

- recursive functions declarations must contain a RESULT keyword, and one type declaration refers to both the function name and the result variable.
Recursive Function Example

The following example calculates the factorial of a number and uses \( n! = n(n - 1)! \)

```fortran
PROGRAM Mayne
IMPLICIT NONE
PRINT*, fact(12) ! etc
CONTAINS
RECURSIVE FUNCTION fact(N) RESULT(N_Fact)
  INTEGER, INTENT(IN) :: N
  INTEGER :: N_Fact ! also defines type of fact
  IF (N > 0) THEN
    N_Fact = N * fact(N-1)
  ELSE
    N_Fact = 1
  END IF
END function FACT
END PROGRAM Mayne
```

To calculate 4!,

1. 4! is \( 4 \times 3! \), so calculate 3! then multiply by 4,

2. 3! is \( 3 \times 2! \), need to calculate 2!,

3. 2! is \( 2 \times 1! \), 1! is \( 1 \times 0! \) and 0! = 1

4. can now work back up the calculation and fill in the missing values.
The important thing to remember is that one may only add and remove items from the top of the stack.
The top of the stack is marked by an arrow, data is added to the right of this arrow and removed from the left.
For example, the following defines a very simple 100 element integer stack and two access functions,

```
PROGRAM stack
  INTEGER, PARAMETER :: stack_size = 100
  INTEGER, SAVE :: store(stack_size), pos=0
  ....
CONTAINS
SUBROUTINE push(i)
  INTEGER, INTENT(IN) :: i
  IF (pos < stack_size) THEN
    pos = pos + 1; store(pos) = i
  ELSE
    STOP 'Stack Full error'
  END IF
END SUBROUTINE push
SUBROUTINE pop(i)
  INTEGER, INTENT(OUT) :: i
  IF (pos > 0) THEN
    i = store(pos); pos = pos - 1
  ELSE
    STOP 'Stack Empty error'
  END IF
END SUBROUTINE pop
END PROGRAM stack
```

The main program can now call push and pop which simulate a 100 element INTEGER stack.
Reusability - Modules

A stack is a useful data structure. To allow it to be used elsewhere convert to a MODULE. This is called encapsulation:

```fortran
MODULE stack
  IMPLICIT NONE
  INTEGER, PARAMETER :: stack_size = 100
  INTEGER, SAVE :: store(stack_size), pos = 0
CONTAINS
  SUBROUTINE push(i)
    ....
  END SUBROUTINE push
  SUBROUTINE pop(i)
    ....
  END SUBROUTINE pop
END MODULE stack
```

Stack can now be accessed by other programs. The USE statement attaches it to a program:

```fortran
PROGRAM StackUser
  USE stack
  IMPLICIT NONE
  ....
  CALL Push(14); CALL Push(21);
  CALL Pop(i); CALL Pop(j)
  ....
END PROGRAM StackUser
```

It is as if the code had been included in StackUser.
Reusability - Modules II

Points raised on the previous slide:

- within a module, functions and subroutines are called *module procedures*,

- module procedures may contain internal procedures (like PROGRAMS),

- module objects which retain their values should be given the *SAVE* attribute,

- modules can also be *USED* by procedures and other modules,

- modules can be compiled separately. They should be compiled *before* the program unit that uses them.
Restricting Visibility

In stack example, the main program has access to store and pos and so could modify the values. This is dangerous.

Can prevent this by assigning visibility attributes:

PRIVATE :: pos, store, stack_size  ! hidden
PUBLIC :: pop, push  ! not hidden

store, stack_size and pos are hidden, pop and push are not.

This makes the module much safer and allows internals of module to be changed without modifying the users program: only allow the user to access what he needs.

Alternatively, use statements or attributes:

PUBLIC  ! set default visibility
INTEGER, PRIVATE, SAVE :: store(stack_size), pos
INTEGER, PRIVATE, PARAMETER :: stack_size = 100

so, in the main PROGRAM:

CALL Push(21); CALL Pop(i)  ! OK
pos = 22; store(pos) = 99  ! Forbidden
The USE Renames Facility

The USE statement names a module whose public definitions are to be made accessible.

Syntax:

\[
\text{USE } \langle \text{module-name} \rangle \ & \\
[,, \langle \text{new-name} \rangle \Rightarrow \langle \text{use-name} \rangle \ldots]
\]

module entities can be renamed,

\[
\text{USE Stack, IntegerPop } \Rightarrow \text{Pop}
\]

The module object Pop is renamed to IntegerPop when used locally.
USE ONLY Statement

Another way to avoid name clashes is to only use those objects which are necessary. It has the following form:

USE < module-name > [ ONLY:< only-list >...]

The < only-list > can also contain renames (=>).

For example,

    USE Stack, ONLY:pos, &
    IntegerPop => Pop

Only pos and Pop are made accessible. Pop is renamed to IntegerPop.

The ONLY statement gives the compiler the option of including only those entities specifically named.
Module - General Form

```
MODULE Nodule
    ! TYPE Definitions
    ! Global data
    ! ..
    ! etc ..
CONTAINS
    SUBROUTINE Sub(..)
        ! Executable stmts
    CONTAINS
        SUBROUTINE Int1(..)
            END SUBROUTINE Int1
            ! etc.
            SUBROUTINE Intn(..)
        END SUBROUTINE Int2n
    END SUBROUTINE Sub
        ! etc.
    FUNCTION Funky(..)
        ! Executable stmts
    CONTAINS
        ! etc
    END FUNCTION Funky
END MODULE Nodule
```

```
MODULE < module name >
    < declarations and specifications statements >
    [ CONTAINS
        < definitions of module procedures > ]
END [ MODULE [ < module name > ] ]
```
Modules — An Overview

The MODULE program unit provides the following facilities:

- global object declaration;
- procedure declaration (includes operator definition);
- semantic extension;
- ability to control accessibility of above to different programs and program units;
- ability to package together whole sets of facilities;
Lecture 7:
Pointers and Derived Types
**Pointers and Targets**

It is often useful to have variables where the space referenced by the variable can be changed as well as the values stored in that space.

- The pointer often uses less space than the target.

- A reference to the pointer will in general be a reference to the target, (pointers are automatically dereferenced).
**Terminology**

A pointer has 3 possible states:

- if a pointer has a particular target then the pointer is said to be *associated* with that target,

- a pointer can be made to have no target — the pointer is *disassociated*,

- the initial status of a pointer is *undefined*.

Visualisation,

- **Associated**: Pointer → Target
- **Disassociated**: Pointer → Null
- **Undefined**: Pointer → ???

Use `ASSOCIATED` intrinsic to get the (association) status of a pointer.
A POINTER:

- is a variable with the POINTER attribute;
- has static type, kind and rank determined by its declaration, for example,

\[
\text{REAL, POINTER :: Ptor} \\
\text{REAL, DIMENSION(:,:), POINTER :: Ptoa}
\]

- Ptor is a pointer to a scalar real target,
- Ptoa is a pointer to a 2-D array of reals.

So,

- the declaration fixes the type, kind and rank of the target;
- pointers to arrays are declared with deferred-shape array specifications;
- the rank of a target is fixed but the shape may vary.
Target Declaration

Targets of a pointer must have the TARGET attribute.

\[
\text{REAL, TARGET :: } x, y \\
\text{REAL, DIMENSION(5,3), TARGET :: } a, b \\
\text{REAL, DIMENSION(3,5), TARGET :: } c
\]

With these declarations (and those from the previous slide):

\[\square \text{ x or y may become associated with } \text{Ptor;}\]

\[\square \text{ a, b or c may become associated with } \text{Ptoa.}\]
Pointer Manipulation

The following operators manipulate pointers:

- \( \Rightarrow \), *pointer assignment* — alias a pointer with a given target;

- \( = \), *‘normal’ assignment* — assign a value to the space pointed at by the pointer.

Pointer assignment makes the pointer and the variable reference the same space while the normal assignment alters the value contained in that space.
Consider,

\[ x = 3.14159 \]
\[ \text{Ptor} \rightarrow y \]
\[ \text{Ptor} = x \]

- \( x \) and \( \text{Ptor} \) have the same value.
- \( \text{Ptor} \) is an alias for \( y \) so the last statement sets \( y = 3.14159 \).
- if the value of \( x \) is subsequently changed, the value of \( \text{Ptor} \) and \( y \) do not.

Coding,

\[ \text{Ptor} = 5.0 \]

sets \( y \) to 5.0.
Association with Arrays

An array pointer may be associated a regular section of a target of the correct rank (and type and kind). It cannot, however, be associated with a vector subscripted array section.

Assuming the same declarations as the previous slide:

- these are valid (2D sections):
  
Ptoa => a(3:5, ::2)
Ptoa => a(1:1,2:2)

- these are invalid (wrong rank):
  
Ptoa => a(1:1,2)
Ptoa => a(1,2)
Ptoa => a(1,2:2)

- this is invalid (vector subscript):
  
v = (/2,3,1,2/)
Ptoa => a(v,v)
Visualisation of Pointers to Sections

\[\text{Ptoa} \Rightarrow a(3::2, ::2)\]

Here the top left subsection element can be referred to through TWO names

- A(3,1)
- Ptoa(1,1)
Dynamic Targets

Targets can also be created dynamically by allocation. `ALLOCATE` can make space become the target of a pointer.

ALLOCATE(Ptor,STAT= ierr)
ALLOCATE(Ptoa(n*n,2*k-1),STAT=ierr)

- the first statement allocates a single real as the target of Ptor.
- the second allocates a rank 2 real array as the target of Ptoa.

It is not an error to allocate an array pointer that is already associated.
Automatic Attributing

Pointer variables implicitly have the TARGET attribute.

Ptoa => A(3::2,1::2)
Ptor => Ptoa(2,1)

Element (2,1) of Ptoa is associated with Ptor.

Here the bottom left subsection element can be referred to through THREE names:

□ Ptor

□ A(5,1)

□ Ptoa(2,1)

When a new association for a pointer is established any previous association with a target is broken.
**Association Status**

The status of a defined pointer may be tested by an intrinsic function:

\[ \text{ASSOCIATED}(\text{Ptoa}) \]

If \text{Ptoa} is defined and associated then this will return \text{.TRUE.}; if it is defined and disassociated it will return \text{.FALSE.}. If it is undefined the result is also undefined.

The target of a defined pointer may also be tested:

\[ \text{ASSOCIATED}(\text{Ptoa, arr}) \]

If \text{Ptoa} is defined and currently associated with the specific target, \text{arr}, then the function will return \text{.TRUE.}, otherwise if it will return \text{.FALSE.}. 

141
**Pointer Disassociation**

Pointers can be disassociated with their targets by:

□ nullification

    NULLIFY(Ptor)
    ◦ breaks the connection of its pointer argument with its target (if any),
    ◦ disassociates the pointer.

Note: it is good practice to nullify all pointers before use.

□ deallocation

    DEALLOCATE(Ptoa, STAT= ierr)
    ◦ breaks the connection between the pointer and its target,
    ◦ deallocates the target.
Pointers to Arrays vs. Allocatable Arrays

There are a number of subtle differences between ALLOCATABLE and POINTER arrays:

- ALLOCATABLE arrays are more efficient,

- TARGET arrays may be referred to by a number of different names,

- unallocated ALLOCATABLE arrays cannot be actual arguments but unassociated (unallocated) POINTER arrays can be.

In summary,

- ALLOCATABLE arrays are more efficient,

- POINTERS are more flexible.
Practical Example

Pointers can be of great use in iterative problems. Iterative methods:

- make guess at required solution;
- use guess as input to an equation to produce better approximation;
- use new approximation to obtain better approximation;
- repeat until degree of accuracy is obtained;

```fortran
REAL, DIMENSION(100,100), TARGET :: app1, app2
REAL, DIMENSION(:,,:), POINTER :: prev_app, &
  next_app, swap
prev_app => app1
next_app => app2
prev_app = initial_app(.....)
DO
  next_app = iteration_function_of(prev_app)
  IF(ABS(MAXVAL(next_app-prev_app))<0.0001)EXIT
  swap => prev_app
  prev_app => next_app
  next_app => swap
END DO
```

Using pointers here avoids having to copy the large matrices.
Pointers and Procedures

A actual pointer argument can be interpreted in two ways:

- immediately dereference and pass the target — the corresponding dummy argument does not have the POINTER attribute;
- pass the pointer so it can be manipulated as a pointer in the procedure — the corresponding dummy argument must have the POINTER attribute.

For example,

```
PROGRAM Supping
   IMPLICIT NONE
   INTEGER, POINTER :: Pint1, Pint2
   ...
   CALL Beer(Pint1,Pint2)
   ...
   CONTAINS
   SUBROUTINE Beer(arg1,arg2)
      INTEGER, POINTER :: arg1
      INTEGER, INTENT(IN) :: arg2
      ...
   END SUBROUTINE Beer
END PROGRAM Supping
```

Note: a POINTER dummy argument cannot have the INTENT attribute.

An explicit interface is required if a pointer is used as an actual argument to a procedure.


**Pointer Valued Functions**

A function may be pointer valued,

```
PROGRAM main
    IMPLICIT NONE
    INTEGER, TARGET :: a, b
    INTEGER, POINTER :: largest
    a = ...; b = ...
    largest => ptr()
    print*, largest
    CONTAINS
        FUNCTION ptr()
            INTEGER, POINTER :: ptr
            IF (a .GT. b) THEN
                ptr => a
            ELSE
                ptr => b
            END IF
        END FUNCTION ptr
    END PROGRAM main
```
Pointer I/O

Pointers which appear in input/output lists:

□ are always dereferenced;

□ must not be disassociated as dereferencing would make no sense;

```fortran
PROGRAM Main
IMPLICIT NONE
REAL, DIMENSION(3), TARGET :: arr=/1.,2.,3./
REAL, DIMENSION(:), POINTER :: p, q
p => arr
PRINT*, p
ALLOCATE(q(5))
READ*, q
PRINT*, q
DEALLOCATE(q)
PRINT*, q ! invalid, no valid target
END PROGRAM Main
```
Derived Types

It is often advantageous to express some objects in terms of aggregate structures, for example: coordinates, \((x, y, z)\).

Fortran 90 allows compound entities or *derived types* to be defined:

```
TYPE COORDS_3D
  REAL :: x, y, z
END TYPE COORDS_3D
TYPE(COORDS_3D) :: pt1, pt2
```

Derived types definitions should be placed in a `MODULE`. 
Supertypes

Previously defined types can be used as components of other derived types,

    TYPE SPHERE
    TYPE (COORDS_3D) :: centre
    REAL :: radius
    END TYPE SPHERE

Objects of type SPHERE can be declared:

    TYPE (SPHERE) :: bubble, ball
Derived Type Assignment

Values can be assigned to derived types in two ways:

- component by component;
- as an object.

An individual component may be selected, using the % operator:

\[
\begin{align*}
\text{pt1\%x} &= 1.0 \\
\text{bubble\%radius} &= 3.0 \\
\text{bubble\%centre\%x} &= 1.0
\end{align*}
\]

The whole object may be selected and assigned to using a constructor:

\[
\begin{align*}
\text{pt1} &= \text{COORDS\_3D(1.,2.,3.)} \\
\text{bubble\%centre} &= \text{COORDS\_3D(1.,2.,3.)} \\
\text{bubble} &= \text{SPHERE(bubble\%centre,10.)} \\
\text{bubble} &= \text{SPHERE(COORDS\_3D(1.,2.,3.),10.)}
\end{align*}
\]

The derived type component of SPHERE must also be assigned to using a constructor.

Assignment between two objects of the same derived type is intrinsically defined,

\[
\text{ball} = \text{bubble}
\]
Arrays and Derived Types

It is possible to define derived type objects which contain non-ALLOCATABLE arrays and arrays of derived type objects.

Consider,

    TYPE FLOBBLE
      CHARACTER(LEN=6) :: nom
      INTEGER, DIMENSION(10,10) :: harry
    END TYPE FLOBBLE
    ...
    TYPE (FLOBBLE) :: bill
    TYPE (FLOBBLE), DIMENSION(10) :: ben

We can refer to an element or subsection of the array component:

    bill%harry(7,7)
    bill%harry(:,::2)
    ben(1)%harry(7,7)
    ben(9)%harry(:,:)
    ben(:)%harry(7,7)

but not a collection of elements or subsections of the array component:

    ben(9:9)%harry(:,:)) ! invalid
    ben(:)%harry(:,::2) ! invalid
Derived Type I/O

Derived type objects which do not contain pointers (or private) components may be input or output using ‘normal’ methods:

PRINT*, bubble

is exactly equivalent to

PRINT*, bubble%centre%x, bubble%centre%y, &
  bubble%centre%z, bubble%radius

Derived types are handled on a component by component basis.
Derived Types and Procedures

Derived types definitions should be packaged in a MODULE.

```
MODULE VecDef
  TYPE vec
    REAL :: r
    REAL :: theta
  END TYPE vec
END MODULE VecDef
```

to make the type definitions visible, the module must be used:

```
PROGRAM Up
  USE VecDef
  IMPLICIT NONE
  TYPE(vec) :: north
  CALL subby(north)
  ...
END CONTAINS

SUBROUTINE subby(arg)
  TYPE(vec), INTENT(IN) :: arg
  ...
END SUBROUTINE subby
END PROGRAM Up
```

Type definitions can only become accessible by *host* or *use* association.
Derived Type Valued Functions

Functions can return results of an arbitrary defined type

```fortran
FUNCTION Poo(kanga, roo)
  USE VecDef
  TYPE (vec) :: Poo
  TYPE (vec), INTENT(IN) :: kanga, roo
  Poo = ...
END FUNCTION Poo
```

Recall that the definitions of `VecDef` must be made available by use or host association.
Lecture 8:
Modules and
Object-based
Programming
POINTER Components of Derived Types

- ALLOCATABLE arrays cannot be used as components in a derived type,

- POINTERS can be,

Dynamically sized structures can be created and manipulated, for example,

```fortran
TYPE VSTRING
    CHARACTER, DIMENSION(:), POINTER :: chars
END TYPE VSTRING
```

this has a component which is a pointer to a 1-D array of characters.

```fortran
TYPE(VSTRING) :: Pvs1
...
ALLOCATE(Pvs1%chars(5))
Pvs1%chars = (/'H','e','l','l','o'/)
```

```
Pvs1 -> H e l l l o
```
Pointers and Recursive Data Structures

- Derived types which include pointer components provide support for recursive data structures such as linked lists.

```
TYPE CELL
    INTEGER :: val
    TYPE (CELL), POINTER :: next
END TYPE CELL
```

- Assignment between structures containing pointer components is subtly different from normal,

```
TYPE (CELL) :: A
TYPE (CELL), TARGET :: B
A = B
```

is equivalent to:

```
A%val = B%val
A%next => B%next
```
Practical Example of Linked Lists

The following fragment would create a linked list of cells starting at head and terminating with a cell whose next pointer is null (disassociated).

PROGRAM Thingy
  IMPLICIT NONE
  TYPE (CELL), TARGET :: head
  TYPE (CELL), POINTER :: curr, temp
  INTEGER :: k

  head%val = 0 ! listhead = default
  NULLIFY(head%next) ! un-define
  curr => head ! curr head of list
  DO
    READ*, k ! get value of k
    ALLOCATE(temp) ! create new cell
    temp%val = k ! assign k to new cell
    NULLIFY(temp%next) ! set disassociated
    curr%next => temp ! attach new cell to ! end of list
    curr => temp ! curr points to new ! end of list
  END DO
END PROGRAM Thingy
Example (Continued)

The statements,

```plaintext
head%val = 0; NULLIFY(head%next); curr => head
```
give,

```
curr

0 --> Null
head
```

```plaintext
ALLOCATE(temp); temp%val = k; NULLIFY(temp%next)
```
give,

```
k --> Null
   temp
```

```plaintext
curr%next => temp; curr => temp
```
give,

```
curr

0 --> k --> Null
head
```

The final list structure is,

```
curr

0 --> ____ --> k --> Null
head
```

158
Example (Continued)

A “walk-through’ the previous linked list could be written as follows:

```
curr => head
DO
   PRINT*, curr%val
   IF(.NOT.ASSOCIATED(curr%next)) EXIT
   curr => curr%next
ENDDO
```

All sorts of multiply linked lists can be created and manipulated in analogous ways.
Arrays of Pointers

It is possible to create what are in effect arrays of pointers:

```
TYPE iPTR
  INTEGER, POINTER :: compon
END TYPE iPTR
TYPE(iPTR), DIMENSION(100) :: ints
```

Visualisation,

```
                            +---->
                           /     |
                          /      |
                         /       |
                        /        |
                       /         |
                      /          |
                     /           |
                    /             |
                   /               |
```
ints

Cannot refer to a whole array of pointers,

```
ints(10)%compon ! valid
ints(:)%compon   ! not valid
```

If desired ints could have been ALLOCATABLE.
Modules — Object Based Programming

We can write a module that allows a derived type to behave in the same way as an intrinsic type. The module can contain:

- the type definitions,
- constructors,
- overloaded intrinsics,
- overload set of operators,
- other related procedures

An example of such a module is the varying string module which is to be an ancillary standard.
Derived types have in-built constructors, however, it is better to write a specific routine instead.

Purpose written constructors can support default values and will not change if the internal structure of the type is modified. It is also possible to hide the internal details of the type:

```fortran
MODULE ThreeDee
  IMPLICIT NONE
  TYPE Coords_3D
    PRIVATE
    REAL :: x, y, z
  END TYPE Coords_3D
CONTAINS
  TYPE(Coords_3D) FUNCTION Init_Coords_3D(x,y,z)
    REAL, INTENT(IN), OPTIONAL :: x,y,z
    ! Set Defaults
    Init_Coords_3D = Coords_3D(0.0,0.0,0.0,0.0)
    IF (PRESENT(x)) Init_Coords_3D%x = x
    IF (PRESENT(y)) Init_Coords_3D%y = y
    IF (PRESENT(z)) Init_Coords_3D%z = z
  END FUNCTION Init_Coords_3D
END MODULE ThreeDee
```

If an argument is not supplied then the corresponding component of Coords_3D is set to zero.
**Generic Interfaces**

Most intrinsics are generic in that their type is determined by their argument(s). For example, the generic function \( \text{ABS}(x) \) comprises the specific functions:

- **CABS** — called when \( x \) is \text{COMPLEX},
- **ABS** — called when \( x \) is \text{REAL},
- **IABS** — called when \( x \) is \text{INTEGER},

These specific functions are called the **overload set**.

A user may define his own overload set in an **INTERFACE** block:

```
INTERFACE CLEAR
  MODULE PROCEDURE clear_int
  MODULE PROCEDURE clear_real
END INTERFACE ! CLEAR
```

The **generic name**, `CLEAR`, is associated with **specific names** `clear_int` and `clear_real` (the overload set).
The full module would be

MODULE Schmodule
  IMPLICIT NONE
  INTERFACE CLEAR
    MODULE PROCEDURE clear_int
    MODULE PROCEDURE clear_real
  END INTERFACE CLEAR
  CONTAINS
  SUBROUTINE clear_int(a)
    INTEGER, DIMENSION(:), INTENT(INOUT) :: a
    ... ! code to do clearing
  END SUBROUTINE clear_int
  SUBROUTINE clear_real(a)
    REAL, DIMENSION(:), INTENT(INOUT) :: a
    ... ! code to do clearing
  END SUBROUTINE clear_real
END MODULE Schmodule

PROGRAM Main
  IMPLICIT NONE
  USE Schmodule
  REAL :: prices(100)
  INTEGER :: counts(50)
  CALL CLEAR(prices) ! generic call
  CALL CLEAR(counts) ! generic call
END PROGRAM Main

The first procedure invocation would be resolved with clear_real and the second with clear_int.
Generic Interfaces - Commentary

In order for the compiler to be able to resolve the reference, both module procedures must be unique:

- the specific procedure to be used is determined by the number, type, kind or rank of the non-optional arguments,

- the overload set of procedures must be unambiguous with respect to their dummy arguments,

- default intrinsic types should not be used in generic interfaces, use parameterised types.

Basically, by examining the argument(s), the compiler calculates which specific procedure to invoke.
Derived Type Output Procedures

Derived types with PRIVATE components need special procedures for I/O:

```
MODULE ThreeDee
IMPLICIT NONE
TYPE Coords_3D
  PRIVATE
  REAL :: x, y, z
END TYPE Coords_3D
INTERFACE Print
  MODULE PROCEDURE Print_Coords_3D
END INTERFACE ! Print
CONTAINS
  SUBROUTINE Print_Coords_3D(Coord)
    TYPE(Coords_3D), INTENT(IN) :: Coord
    PRINT*, Coord%x, Coord%y, Coord%z
  END SUBROUTINE Print_Coords_3D(Coord)
END MODULE ThreeDee
```

Define an INTERFACE called Print and overload to accept arguments of type Coords_3D
Overloading Intrinsic Procedures

When a new type is added, it is a simple process to add a new overload to any relevant intrinsic procedures.

The following extends the LEN_TRIM intrinsic to return the number of letters in the owners name for objects of type HOUSE,

```fortran
MODULE new_house_defs
  IMPLICIT NONE
  TYPE HOUSE
    CHARACTER(LEN=16) :: owner
    INTEGER :: residents
    REAL :: value
  END TYPE HOUSE
  INTERFACE LEN_TRIM
    MODULE PROCEDURE owner_len_trim
  END INTERFACE
  CONTAINS
    FUNCTION owner_len_trim(ho)
      TYPE(HOUSE), INTENT(IN) :: ho
      INTEGER :: owner_len_trim
      owner_len_trim = LEN_TRIM(ho%owner)
    END FUNCTION owner_len_trim
    .... ! other encapsulated stuff
  END MODULE new_house_defs
```

The user defined procedures are added to the existing generic overload set.
Intrinsic operators, such as -, = and *, can be overloaded to apply to all types in a program:

- specify the generic operator symbol in an INTERFACE OPERATOR statement,

- specify the overload set in a generic interface,

- declare the MODULE PROCEDURES (FUNCTIONS) which define how the operations are implemented.

These functions must have one or two non-optional arguments with INTENT(IN) which correspond to monadic and dyadic operators.

Overloads are resolved as normal.
Operator Overloading Example

The '*' operator can be extended to apply to the rational number data type as follows:

MODULE rational_arithmetic
  TYPE RATNUM
    INTEGER :: num, den
  END TYPE RATNUM
  INTERFACE OPERATOR (*)
    MODULE PROCEDURE rat_rat,int_rat,rat_int
  END INTERFACE
  CONTAINS
    FUNCTION rat_rat(l,r)  ! rat * rat
      TYPE(RATNUM), INTENT(IN) :: l,r
      ...
      rat_rat = ...
    END FUNCTION rat_rat
    FUNCTION int_rat(l,r)  ! int * rat
      INTEGER, INTENT(IN) :: l
      TYPE(RATNUM), INTENT(IN) :: r
      ...
      int_rat = ...
    END FUNCTION int_rat
    FUNCTION rat_int(l,r)  ! rat * int
      TYPE(RATNUM), INTENT(IN) :: l
      INTEGER, INTENT(IN) :: r
      ...
      rat_int = ...
    END FUNCTION rat_int
  END MODULE rational_arithmetic

The three new procedures are added to the operator overload set allowing them to be used as operators in a normal arithmetic expressions.
Example (Cont’d)

With,

USE rational_arithmetic
TYPE (RATNUM) :: ra, rb, rc

we could write,

rc = rat_rat(int_rat(2,ra),rb)

but better:

rc = 2*ra*rb

And even better still add visibility attributes to force user into good coding:

MODULE rational_arithmetic
  TYPE RATNUM
    PRIVATE
      INTEGER :: num, den
  END TYPE RATNUM
INTERFACE OPERATOR (*)
  MODULE PROCEDURE rat_rat,int_rat,rat_int
END INTERFACE
PRIVATE :: rat_rat,int_rat,rat_int
....
Defining New Operators

can define new monadic and dyadic operators. They have the form,

\. <name>.

Note:

- monadic operators have precedence over dyadic.

- names must be 31 letters (no numbers or underscore) or less.

- basic rules same as for overloading procedures.
Defined Operator Example

For example, consider the following definition of the .TWIDDLE. operator in both monadic and dyadic forms,

```
MODULE twiddle_op
  INTERFACE OPERATOR (.TWIDDLE.)
    MODULE PROCEDURE itwiddle, iitwiddle
  END INTERFACE ! (.TWIDDLE.)
END CONTAINS
FUNCTION itwiddle(i)
  INTEGER itwiddle
  INTEGER, INTENT(IN) :: i
  itwiddle = -i*i
END FUNCTION
FUNCTION iitwiddle(i,j)
  INTEGER iitwiddle
  INTEGER, INTENT(IN) :: i,j
  iitwiddle = -i*j
END FUNCTION
END MODULE
```

The following

```
PROGRAM main
  USE twiddle_op
  print*, 2.TWIDDLE.5, .TWIDDLE.8, &
          .TWIDDLE.(2.TWIDDLE.5), &
          .TWIDDLE.2.TWIDDLE.5
END PROGRAM
```

produces

```
-10 -64 -100 20
```
Precedence

- user defined monadic operators are most tightly binding.
- user defined dyadic operators are least tightly binding.

For example,

```
.TWIDDLE.e**j/a.TWIDDLE.b+c.AND.d
```

is equivalent to

```
(((.TWIDDLE.e)**j)/a).TWIDDLE.((b+c).AND.d)
```
User-defined Assignment

Assignment between two different user-defined types must be explicitly programmed; a SUBROUTINE with two arguments specifies what to do,

- the first argument is the result variable and must have \texttt{INTENT(OUT)};

- the second is the expression whose value is converted and must have \texttt{INTENT(IN)}.

Overloading the assignment operator differs from other operators:

- assignment overload sets do \textbf{not} have to produce an unambiguous set of overloads;

- later overloads override earlier ones if there is an ambiguity;
Defined Assignment Example

Should put in a module,

```
INTERFACE ASSIGNMENT(=)
  MODULE PROCEDURE rat_ass_int, real_ass_rat
END INTERFACE
PRIVATE :: rat_ass_int, real_ass_rat
```
specify SUBROUTINES in the CONTAINS block:

```
SUBROUTINE rat_ass_int(var, exp)
  TYPE (RATNUM), INTENT(OUT) :: var
  INTEGER, INTENT(IN) :: exp
  var%num = exp
  var%den = 1
END SUBROUTINE rat_ass_int

SUBROUTINE real_ass_rat(var, exp)
  REAL, INTENT(OUT) :: var
  TYPE (RATNUM), INTENT(IN) :: exp
  var = REAL(exp%num) / REAL(exp%den)
END SUBROUTINE real_ass_rat
```

Wherever the module is used the following is valid:

```
ra = 50
x = rb*rc
```

for real x.
Semantic Extension Example

Collect stuff from previous slides:

```plaintext
MODULE rational_arithmetic
  IMPLICIT NONE
  PUBLIC :: OPERATOR (*)
  PUBLIC :: ASSIGNMENT (=)
  TYPE RATNUM
  PRIVATE
    INTEGER :: num, den
  END TYPE RATNUM
  TYPE, PRIVATE :: INTERNAL
    INTEGER :: lhs, rhs
  END TYPE INTERNAL
  INTERFACE OPERATOR (*)
    MODULE PROCEDURE rat_rat, int_rat, rat_int
  END INTERFACE ! OPERATOR (*)
  PRIVATE rat_rat, int_rat, rat_int
  ... ! and so on

The type INTERNAL is only accessible from within the module.

Should also add:

- constructors for RATNUM: init_RATNUM
- output procedure for RATNUM: Print
- overloaded intrinsics: REAL, CEILING, FLOOR, etc
```
**Semantic Extension Modules**

The real power of the `MODULE / USE` facilities appears when coupled with derived types and operator and procedure overloading to provide *semantic extensions* to the language.

Semantic extension modules require:

- a mechanism for defining new types;
- a method for defining operations on those types;
- a method of overloading the operations so user can use them in a natural way;
- a way of encapsulating all these features in such a way that the user can access them as a combined set;
- details of underlying data representation in the implementation of the associated operations to be kept hidden (desirable).

This is an Object Oriented approach.
Lecture 9:
Parametrised Intrinsic Types Plus
Complex Data

This data type has the same precision as default REAL.

☐ COMPLEX object declaration:

    COMPLEX :: z, j, za(1:100)

☐ symbolic constants are expressed in the same way:

    COMPLEX, PARAMETER :: i = (0.0,1.0)

☐ complex constants are represented as co-ordinate pairs: (1.0,1.0) or (3.141,1.0E-9).

☐ complex values can be constructed using the CMPLX function,

    z = CMPLX(x,y)

☐ complex expressions can be used in the same way as other types

    REAL :: x; COMPLEX :: a, b, c,
    ...
    a = x*((b+c)*CMPLX(0.1,-0.1))
    b = 1

The real value x will be promoted to the complex value CMPLX(x,0). b will be set to CMPLX(1.0,0).
## Complex Intrinsic Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIMAG(z)</td>
<td>Imaginary part of a complex number</td>
</tr>
<tr>
<td>REAL(z)</td>
<td>Real part of a complex number</td>
</tr>
<tr>
<td>CONJG(z)</td>
<td>Conjugate of a complex number</td>
</tr>
<tr>
<td>ABS(z)</td>
<td>Absolute value of a complex number</td>
</tr>
<tr>
<td>SIN(z) etc</td>
<td>All appropriate mathematical functions</td>
</tr>
</tbody>
</table>
Parameterised Data Types

- Fortran 77 had a problem with numeric portability, the precision (and exponent range) between processors could differ,

- Fortran 90 implements a portable precision selecting mechanism,

- intrinsic types can be parameterised by a kind value (an integer). For example,

\[
\begin{align*}
\text{INTEGER(KIND=1)} &:: \text{ik1} \\
\text{REAL(4)} &:: \text{rk4}
\end{align*}
\]

- the kind parameters correspond to differing precisions supported by the compiler (details in the compiler manual).

- objects of different kinds can be mixed in arithmetic expressions but procedure arguments must match in type and kind.
Integer Data Type by Kind

- selecting kind, by an explicit integer is still **not** portable,

- must use the SELECTED_INT_KIND intrinsic function. For example, SELECTED_INT_KIND(2) returns a kind number capable of expressing numbers in the range, \((-10^2, 10^2)\).

- here the argument specifies the minimum decimal exponent range for the desired model. For example,

```fortran
INTEGER :: short, medium, long, vlong
PARAMETER (short = SELECTED_INT_KIND(2), &
medium= SELECTED_INT_KIND(4), &
long = SELECTED_INT_KIND(10),&
 vlong = SELECTED_INT_KIND(100))

INTEGER(short) :: a,b,c
INTEGER(medium) :: d,e,f
INTEGER(long) :: g,h,i
```
Constants of Selected Integer Kind

□ Constants of a selected kind are denoted by appending underscore followed by the kind number or an integer constant name (better):

\[ 100\_2, 1238\_4, 54321\_\text{long} \]

□ Be **very careful** not to type a minus sign ‘-’ instead of an underscore ‘_’!

□ There are other pitfalls too, the constant

\[ 1000\_\text{short} \]

may not be valid as \text{KIND} = \text{short} may not be able to represent numbers greater than 100. Be very careful.
Real KIND Selection

Similar principle to INTEGER:

- SELECTED_REAL_KIND(8,9) will support numbers with a precision of 8 digits and decimal exponent range from (−9, 9). For example,

  INTEGER, PARAMETER ::
  r1 = SELECTED_REAL_KIND(5,20), &
  r2 = SELECTED_REAL_KIND(10,40)
  REAL(KIND=r1) :: x, y, z
  REAL(r2), PARAMETER :: diff = 100.0_r2

- COMPLEX variables are specified in the same way,

  COMPLEX(KIND=r1) :: cinema
  COMPLEX(r2) :: inferiority = &
                 (100.0_r2,99.0_r2)

  Both parts of the complex number have the same numeric range.
Kind Functions

- It is often useful to be able to interrogate an object to see what kind parameter it has.

- `KIND` returns the integer which corresponds to the kind of the argument.

- For example, `KIND(a)` will return the integer parameter which corresponds to the kind of `a`. `KIND(20)` returns the kind value of the default integer type.

- The intrinsic type conversion functions have an optional argument to specify the kind of the result, for example,

\[
\text{print*}, \ \text{INT}(1.0, \text{KIND}=3), \ \text{NINT}(1.0, \text{KIND}=3) \\
x = x + \text{REAL}(j, \text{KIND}(x))
\]
Mixed Kind Expression Evaluation

Mixed kind expressions:

- If all operands of an expression have the same type and kind, then the result also has this type and kind.

- If the kinds are different, then operands with lower range are promoted before operations are performed. For example, if

\[
\begin{align*}
\text{INTEGER(short)} & \text{:: members, attendees} \\
\text{INTEGER(long)} & \text{:: salaries, costs}
\end{align*}
\]

the expression:

- \text{members + attendees} is of kind \text{short},
- \text{salaries - costs} is of kind \text{long},
- \text{members * costs} is also of kind \text{long}.

- Care must be taken to ensure the LHS is able to hold numbers returned by the RHS.
There is no SELECTED_LOGICAL_KIND intrinsic, however, the KIND intrinsic can be used as normal. For example,

```fortran
LOGICAL(KIND=4) :: yorn = .TRUE._4
LOGICAL(KIND=1), DIMENSION(10) :: mask
IF (yorn .EQ. LOGICAL(mask(1),KIND(yorn)))
```

- KIND=1 may only use one byte of store per variable,

```
LOGICAL(KIND=1)  1 byte
LOGICAL(KIND=4)  4 bytes
```

- Must refer to the compiler manual.
Character KIND Selection

- Every compiler must support at least one character set which must include all the Fortran characters. A compiler may also support other character sets:

  INTEGER, PARAMETER :: greek = 1  
  CHARACTER(KIND=greek) :: zeus, athena  
  CHARACTER(KIND=2,LEN=25) :: mohammed

- Normal operations apply individually but characters of different kinds cannot be mixed. For example,

  print*, zeus//athena ! OK  
  print*, mohammed//athena ! illegal  
  print*, CHAR(ICHAR(zeus),greek)

  Note CHAR gives the character in the given position in the collating sequence.

- Literals can also be specified:

  greek_"\alpha\delta\alpha\mu"

  Notice how the kind is specified first.
Kinds and Procedure Arguments

Dummy and actual arguments must match exactly in kind, type and rank, consider,

```fortran
SUBROUTINE subbie(a,b,c)
  USE kind_defs
  REAL(r2), INTENT(IN) :: a, c
  REAL(r1), INTENT(OUT) :: b
  ...
```

an invocation of `subbie` must have matching arguments, for example,

```fortran
USE kind_defs
REAL(r1) :: arg2
REAL(r2) :: arg3
...
CALL subbie(1.0_r2, arg2, arg3)
```

Using 1.0 instead of 1.0_r2 will not be correct on every compiler.

This is very important with generics.
### Bit Manipulation Intrinsic Functions

**Summary,**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTEST(i,pos)</td>
<td>bit testing</td>
</tr>
<tr>
<td>IAND(i,j)</td>
<td>AND</td>
</tr>
<tr>
<td>IBCLR(i,pos)</td>
<td>clear bit</td>
</tr>
<tr>
<td>IBITS(i,pos,len)</td>
<td>bit extraction</td>
</tr>
<tr>
<td>IBSET(i,pos)</td>
<td>set bit</td>
</tr>
<tr>
<td>IEOR(i,j)</td>
<td>exclusive OR</td>
</tr>
<tr>
<td>IOR(i,j)</td>
<td>inclusive OR</td>
</tr>
<tr>
<td>ISHFT(i,shft)</td>
<td>logical shift</td>
</tr>
<tr>
<td>ISHFTC(i,shft)</td>
<td>circular shift</td>
</tr>
<tr>
<td>NOT(i)</td>
<td>complement</td>
</tr>
<tr>
<td>MVBITS(ifr,ifrpos,len,it0,it0pos)</td>
<td>move bits (SUBROUTINE)</td>
</tr>
</tbody>
</table>

Variables used as bit arguments must be INTEGER valued. The model for bit representation is that of an unsigned integer, for example,

\[
\begin{array}{cccc}
\text{s-t} & 3 & 2 & 1 & 0 \\
0 .. & 0 & 0 & 0 & 0 \quad \text{value} = 0
\end{array}
\]

\[
\begin{array}{cccc}
\text{s-t} & 3 & 2 & 1 & 0 \\
0 .. & 0 & 1 & 1 & 1 \quad \text{value} = 5
\end{array}
\]

\[
\begin{array}{cccc}
\text{s-t} & 3 & 2 & 1 & 0 \\
0 .. & 0 & 0 & 1 & 1 \quad \text{value} = 3
\end{array}
\]

The number of bits in a single variable depends on the compiler.
Array Construction Intrinsics

There are four intrinsics in this class:

- MERGE(TSOURCE,FSOURCE,MASK) — merge two arrays under a mask,

- SPREAD(SOURCE,DIM,NCOPIES) — replicates an array by adding NCOPIES of a dimension,

- PACK(SOURCE,MASK[,VECTOR]) — pack array into a one-dimensional array under a mask.

- UNPACK(VECTOR,MASK,FIELD) — unpack a vector into an array under a mask.
MERGE(TSOURCE, FSOURCE, MASK) — merge two arrays under mask control. TSOURCE, FSOURCE and MASK must all conform and the result is TSOURCE where MASK is .TRUE. and FSOURCE where it is .FALSE.

If,

\[
\text{MASK} = \begin{pmatrix}
T & T & F \\
F & F & T
\end{pmatrix}
\]

and

\[
\text{TSOURCE} = \begin{pmatrix}
1 & 5 & 9 \\
3 & 7 & 11
\end{pmatrix}
\]

and

\[
\text{FSOURCE} = \begin{pmatrix}
0 & 4 & 8 \\
2 & 6 & 10
\end{pmatrix}
\]

we find

\[
\text{MERGE(TSOURCE, FSOURCE, MASK)} = \begin{pmatrix}
\boxed{1} & \boxed{5} & 8 \\
\boxed{2} & 6 & \boxed{11}
\end{pmatrix}
\]
**SPREAD Intrinsic**

SPREAD(SOURCE, DIM, NCOPIES) — replicates an array by adding NCOPIES of in the direction of a stated dimension.

If A is (/5, 7/), then

\[
\text{SPREAD}(A, 2, 4) = \begin{pmatrix}
5 & 5 & 5 & 5 \\
7 & 7 & 7 & 7
\end{pmatrix}
\]

and

\[
\text{SPREAD}(A, 1, 4) = \begin{pmatrix}
5 & 7 \\
5 & 7 \\
5 & 7 \\
5 & 7
\end{pmatrix}
\]
**PACK Intrinsic**

PACK(SOURCE,MASK[,VECTOR])—pack a arbitrary shaped array into a one-dimensional array under a mask. VECTOR, if present, must be 1-D and must be of same type and kind as SOURCE.

If

$$\text{MASK} = \begin{pmatrix} T & T & F \\ F & F & T \end{pmatrix}$$

and

$$\text{A} = \begin{pmatrix} 1 & 5 & 9 \\ 3 & 7 & 11 \end{pmatrix}$$

then

- PACK(A,MASK) is (/1, 5, 11/);
- PACK(A,MASK,(/3,4,5,6/)) is (/1, 5, 11, 6/).
- PACK(A,.TRUE.,(/1,2,3,4,5,6,7,8,9/)) is (/1,3,5,7,9,11,7,8,9/).
**UNPACK Intrinsic**

UNPACK(VECTOR,MASK,FIELD)—unpack a vector into an array under a mask. FIELD, must conform to MASK and and must be of same type and kind as VECTOR. The result is the same shape as MASK.

If

\[
\text{FIELD} = \begin{pmatrix} 9 & 5 & 1 \\ 7 & 7 & 3 \end{pmatrix}
\]

and

\[
\text{MASK} = \begin{pmatrix} T & T & F \\ F & F & T \end{pmatrix}
\]

then

\[
\text{UNPACK}((/6,5,4/),\text{MASK},\text{FIELD}) = \begin{pmatrix} 6 & 5 & 1 \\ 7 & 7 & 4 \end{pmatrix}
\]

and

\[
\text{UNPACK}((/3,2,1/),\text{.NOT.MASK},\text{FIELD}) = \begin{pmatrix} 9 & 5 & 1 \\ 3 & 2 & 3 \end{pmatrix}
\]
TRANSFER Intrinsic

TRANSFER converts (not coerces) physical representation between data types; it is a retyping facility. Syntax:

TRANSFER(SOURCE,MOLD)

- SOURCE is the object to be retyped,
- MOLD is an object of the target type.

REAL, DIMENSION(10) :: A, AA
INTEGER, DIMENSION(20) :: B
COMPLEX, DIMENSION(5) :: C

... 
A = TRANSFER(B, (/ 0.0 /))
AA = TRANSFER(B, 0.0)
C = TRANSFER(B, (/ (0.0,0.0) /))
...

<table>
<thead>
<tr>
<th>INTEGER</th>
<th>0 .. 0 1 0 1</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>REAL</td>
<td>0 .. 0 1 0 1</td>
<td>A</td>
</tr>
<tr>
<td>REAL</td>
<td>.. .. 0 1 0 1</td>
<td>AA</td>
</tr>
<tr>
<td>COMPLEX</td>
<td>0 .. 0 1 0 1</td>
<td>C</td>
</tr>
</tbody>
</table>
Lecture 10:
Odds and Ends
Fortran 90 has a wealth of I/O, too much to cover here.

Fortran 90 allows a number of different streams (files) to be connected to a program for both reading and writing.

In Fortran 90 a file is connected to a *logical unit* denoted by a number. Each unit can have many properties, for example,

- file — the name of the file connected to the unit,
- action — read, write, read and write, etc.
- status — old, new, replace, etc.
- access method — sequential, direct,
**OPEN Statement**

The syntax is,

```
OPEN([UNIT=]< integer>, &
    FILE=< filename >, ERR=< label >, &
    STATUS=< status >, ACCESS=< method >, &
    ACTION=< mode >, RECL=< int-expr >)
```

where,

- `< filename >` is a string,
- `< status >` is 'OLD', 'NEW', 'REPLACE', 'SCRATCH' or 'UNKNOWN',
- `< method >` is 'DIRECT' or 'SEQUENTIAL'
- `< mode >` is 'READ', 'WRITE' or 'READWRITE'
- if the file is direct access the RECL must be specified,

For example,

```
OPEN(17,FILE='output.dat',ERR=10, &
    STATUS='REPLACE', &
    ACCESS='SEQUENTIAL', ACTION='WRITE')
OPEN(14,FILE='input.dat',ERR=10, &
    STATUS='OLD', RECL=iexp, &
    ACCESS='DIRECT',ACTION='READ')
```
**READ Statement**

Not all the specifiers can be used at the same time,

```
READ([UNIT=]<unit>, [FMT=]<format>, IOSTAT= &
   <int-variable>, ERR=<label>, END=<label>, &
   EOR=<label>, ADVANCE=<advance-mode>, &
   REC=<int-expr>, SIZE=<num-chars>) <output-list>
```

where

- `<unit>` is a number or *.
- `<format>` is a string or a FORMAT statement label.
- `<int-variable>` holds a return code, zero means no error.
- `<label>` is an executable statement label.
- `<advance-mode>` is ’YES’ or ’NO’.
- `<num-chars>` is the number of characters read.
- `<int-expr>` is a record number (for direct access).

For example

```
READ(14,FMT=’(3(F10.7,1x))’,REC=iexp) a,b,c
READ(*,’(A)’,ADVANCE=’NO’,EOR=12,SIZE=nch) str
```
**WRITE Statement**

Can interchange READ and WRITE in the following. Not all the specifiers can be used at the same time,

```
WRITE([UNIT=]<unit>, [FMT=]<format>, &
    IOSTAT=<int-variable>, ERR=<label>, &
    ADVANCE=<advance-mode>, &
    REC=<int-expr>) <output-list>
```

where

- `<unit>` is a number or `*`,
- `<format>` is a string or a FORMAT statement label,
- `<int-variable>` holds a return code, zero means no error,
- `<label>` is an executable statement label,
- `<advance-mode>` is 'YES' or 'NO',
- `<int-expr>` is a record number (for direct access).

For example

```
WRITE(17,FMT='(I4)',IOSTAT=istat,ERR=10) ival
WRITE(*,'(A)',ADVANCE='NO') 'Yello'
```
**FORMAT Statement / FMT= Specifier**

FMT= can specify either a line number of a FORMAT statement, an actual format string or a *.

Fortran 90 has a rich formatting syntax, only the highlights are given here.

An example,

```fortran
WRITE(17,FMT=&
   '(2X,2I4,1X,''name ''A7)')i,j,str
READ(14,*) x,y
WRITE(*,FMT=10) a,b
10 FORMAT('vals',2(F15.6,2X))
```

The data is formatted using edit descriptors.

The following is written,

```
  11-195 name Philip
  vals    -1.051330   333356.000033
```
Edit Descriptors

Summary,

\begin{align*}
  &IW & w \text{ chars of integer data}, \\
  &FW.d & w \text{ chars of real data (d dec. pl.)}, \\
  &EW.d & w \text{ chars of real data (d dec. pl.)}, \\
  &LW & w \text{ chars of logical data}, \\
  &A[w] & w \text{ chars of CHARACTER data}, \\
  &nX & \text{ skip } n \text{ chars (n spaces)}, \\
\end{align*}

For example,

\begin{verbatim}
WRITE(*,FMT='(2X,2(I4,1X),''name ''&
         A4,F13.5,1X,E13.5)') &
77778,3,'abcdefghi',14.45,14.5666666
\end{verbatim}

gives

\begin{verbatim}
****  3 name abcd 14.45000 0.14567E+02
\end{verbatim}

Here 77778 will not fit into 4 spaces
Other I/O Statements

- **CLOSE** — unattaches the unit number specified in the statement.

- **REWIND** — puts the file pointer back to the start of the file.

- **BACKSPACE** — the file pointer is moved back one record.

- **ENDFILE** — forces an end-of-file to be written into the file.

The above statements have other specifiers such as **IOSTAT**, 

For example,

REWIND (UNIT=14)  
BACKSPACE (UNIT=17)  
ENDFILE (17)  
CLOSE (17, IOSTAT=ival)
External Procedures

Fortran 90 allows a class of procedure that is not contained within a PROGRAM or a MODULE — an EXTERNAL procedure.

This is the old Fortran 77-style of programming and is more clumsy than the Fortran 90 way.

Differences:

- they may be compiled separately,

- may need an explicit INTERFACE to be supplied to the calling program,

- can be used as arguments (in addition to intrinsics),

- should contain the IMPLICIT NONE specifier.
Subroutine Syntax

Syntax of a (non-recursive) subroutine declaration:

```
SUBROUTINE Ext_1(...)  
! ...  
CONTAINS ! Internal Procs  
   SUBROUTINE Int_1(...)  
      ! Executable stmts  
      END SUBROUTINE Int_1  
      ! etc.  
      FUNCTION Int_n(...)  
      ! Executable stmts  
      END FUNCTION Int_n  
END SUBROUTINE Ext_1  

SUBROUTINE Ext_2(...)  
      ! etc  
END SUBROUTINE Ext_2  
```

```
SUBROUTINE <procname>[ (<dummy args>)]  
   <declaration of dummy args>  
   <declaration of local objects>  
   ...  
   <executable stmts>  
   [ CONTAINS  
      <internal procedure definitions> ]  
END [ SUBROUTINE [<procname>] ]  
```
External Subroutine Example

An external procedure may invoke a further external procedure,

```
SUBROUTINE sub1(a,b,c)
    IMPLICIT NONE
    EXTERNAL sum_sq  ! Should declare or use an INTERFACE
    REAL :: a, b, c, s
    ...
    CALL sum_sq(a,b,c,s)
    ...
END SUBROUTINE sub1
```
calls,

```
SUBROUTINE sum_sq(aa,bb,cc,ss)
    IMPLICIT NONE
    REAL, INTENT(IN) :: aa, bb, cc
    REAL, INTENT(OUT) :: ss
    ss = aa*aa + bb*bb + cc*cc
END SUBROUTINE sum_sq
```
Function Syntax

Syntax of a (non-recursive) function:

\[
<prefix> \ \text{FUNCTION} \ \langle \text{procname} \rangle (\ [\langle \text{dummy args} \rangle]\)
\]
\[
\langle \text{declaration of dummy args} \rangle
\]
\[
\langle \text{declaration of local objects} \rangle
\]
\[
\ldots
\]
\[
\langle \text{executable stmts, assignment of result} \rangle
\]
\[
[ \ \text{CONTAINS} \]
\]
\[
\langle \text{internal procedure definitions} \rangle
\]
END [ FUNCTION [ \langle \text{procname} \rangle ] ]

Here, \langle prefix \rangle, specifies the result type. or,

\[
\text{FUNCTION} \ \langle \text{procname} \rangle (\ [\langle \text{dummy args} \rangle]\)
\]
\[
\langle \text{declaration of dummy args} \rangle
\]
\[
\langle \text{declaration of result type} \rangle
\]
\[
\langle \text{declaration of local objects} \rangle
\]
\[
\ldots
\]
\[
\langle \text{executable stmts, assignment of result} \rangle
\]
\[
[ \ \text{CONTAINS} \]
\]
\[
\langle \text{internal procedure definitions} \rangle
\]
END [ FUNCTION [ \langle \text{procname} \rangle ] ]

Here, \langle procname \rangle must be declared.
External Function Example

口 A function is invoked by its appearance in an expression at the place where its result value is needed,

\[ \text{total} = \text{total} + \text{largest}(a,b,c) \]

口 external functions should be declared as EXTERNAL or else the INTERFACE should be given,

```
INTEGER, EXTERNAL :: largest
```

口 The function is defined as follows,

```
INTEGER FUNCTION largest(i,j,k)
  IMPLICIT NONE
  INTEGER :: i, j, k
  largest = i
  IF (j .GT. largest) largest = j
  IF (k .GT. largest) largest = k
END FUNCTION largest
```

or equivalently as,

```
FUNCTION largest(i,j,k)
  IMPLICIT NONE
  INTEGER :: i, j, k
  INTEGER :: largest
  ...
END FUNCTION largest
```
Procedure Interfaces

For EXTERNAL procedures it is possible to provide an explicit interface for a procedure. Consider:

```fortran
SUBROUTINE expsum( n, k, x, sum ) ! in interface
  USE KIND_VALS:ONLY long
  IMPLICIT NONE
  INTEGER, INTENT(IN) :: n ! in interface
  REAL(long), INTENT(IN) :: k, x ! in interface
  REAL(long), INTENT(OUT) :: sum ! in interface
  REAL(long), SAVE :: cool_time
  sum = 0.0
  DO i = 1, n
    sum = sum + exp(-i*k*x)
  END DO
END SUBROUTINE expsum ! in interface
```

The explicit INTERFACE for this routine is given by the statements which appear in the declarations part of any program unit that calls expsum:

```fortran
INTERFACE ! for EXTERNAL procedures
  SUBROUTINE expsum( n, k, x, sum )
    USE KIND_VALS:ONLY long
    INTEGER, INTENT(IN) :: n
    REAL(long), INTENT(IN) :: k, x
    REAL(long), INTENT(OUT) :: sum
  END SUBROUTINE expsum
END INTERFACE
```

Interfaces replace any EXTERNAL statements and are not needed for internal (or module) procedures.

208
What Appears in an Interface?

An interface only contains:

- the SUBROUTINE or FUNCTION header,
- (if not included in the header) the FUNCTION type,
- declarations of the dummy arguments (including attributes),
- the END SUBROUTINE or END FUNCTION statement

Interfaces are only ever needed for EXTERNAL procedures and remove the need for any other form of declaration of that procedure.
Interface Example

The following program includes an explicit interface,

```
PROGRAM interface_example
  IMPLICIT NONE

  INTERFACE
    SUBROUTINE expsum(N,K,X,sum)
      INTEGER, INTENT(IN) :: N
      REAL, INTENT(IN) :: K,X
      REAL, INTENT(OUT) :: sum
    END SUBROUTINE expsum
  END INTERFACE

  REAL :: sum
  ...
  CALL expsum(10,0.5,0.1,sum)
  ...
END PROGRAM interface_example
```

Explicit interfaces permit separate compilation, optimisation and type checking.
Required Interfaces

Explicit interfaces are mandatory if an EXTERNAL procedure has:

- dummy arguments that are assumed-shape arrays, pointers or targets;
- OPTIONAL arguments;
- an array valued or pointer result (functions);
- a result that has an inherited LEN=* length specifier (character functions);

and when the reference:

- has a keyword argument;
- is a defined assignment;
- is a call to the generic name;
- is a call to a defined operator (functions).
**Procedure Arguments**

If `EXTERNAL` procedures are to be used as arguments they must be declared at the call site as:

- **INTRINSIC** — for in-built external procedures.

  ```
  INTRINSIC MVBITS
  REAL, INTRINSIC :: ASIN
  ```

- **EXTERNAL** — for external or dummy procedures.

  ```
  EXTERNAL My_Subby
  INTEGER, EXTERNAL :: My_Funky
  ```

  If an intrinsic procedure name is used in an `EXTERNAL` statement then only the external procedure of that name is visible in that scope; the intrinsic becomes unavailable.

In both cases the `specific`, not generic, procedure name must be used as an actual argument.

Internal procedures are **forbidden** to appear as arguments.
Example of Procedure Arguments

The following example demonstrates the use of procedures as arguments:

```fortran
PROGRAM main
    IMPLICIT NONE
    INTRINSIC ASIN
    REAL, EXTERNAL :: my_sin
    EXTERNAL diffo1
    CALL subby(ASIN,my_sin,diffo1,SIN(0.5))
END PROGRAM

SUBROUTINE subby(fun1,fun2,sub1,x)
    IMPLICIT NONE
    REAL, INTENT(IN) :: x
    REAL, EXTERNAL :: fun2, fun1
    EXTERNAL sub1
    PRINT*, fun1(x), fun2(x)
    CALL sub1(fun2(x),fun1,x)
END SUBROUTINE subby

SUBROUTINE diffo1(y,f,x)
    IMPLICIT NONE
    REAL, INTENT(IN) :: x,y
    REAL, EXTERNAL :: f
    print*, "Diffo1 = ",y-f(x)
END SUBROUTINE diffo1

REAL FUNCTION my_sin(x)
    ... 
END FUNCTION my_sin
```

213
The **DATA Statement**

The **DATA** statement is good for initialising odd shaped sections of arrays.

Syntax,

```
DATA < var1-list >/ < data1-list >/, &
    ...< varn-list >/ < datan-list >/ 
```

The number of constants in each `< data-list >` must be equal to the number of variables / array elements in the corresponding `< var-list >`.

Any object initialised by the **DATA** statement has the **SAVE** attribute
**DATA Statement Example**

As an example, consider initialising a $1000 \times 1000$ array with all the edge values equal to 1 and with the rest of the array zero. This is not as simple to do with an initialisation statement as it is with a DATA statement:

```plaintext
REAL :: matrix(100,100)
DATA matrix(1, 1:100) /100*1.0 /*top row
DATA matrix(100, 1:100) /100*1.0 /*bot row
DATA matrix(2:99, 1) /98*1.0 /*left col
DATA matrix(2:99, 100) /98*1.0 /*right col
DATA matrix(2:99, 2:99) /9604*0.0/*interior
```

The expression 100*1.0 means "100 occurrences of 1.0", the * is the repetition specifier. In this context it cannot be confused with the multiplication operator because such operators are not allowed in a DATA statement.
Data Statement — Implied DO Loop

In a DATA statement the < var-list > may be specified by means of an implied-DO.

Initialising a matrix to have a given constant value, say, 5.0 on the diagonal and zero everywhere else is simple to do using this method.

The object section is specified by a tight loop which is more expressive than array syntax would allow:

```fortran
REAL :: diag(100,100)
DATA (diag(i,i), i=1,100) / 100*5.0 /
   ! sets diagonal elements
DATA ((diag(i,j),diag(j,i),
     j=i+1,100),i=1,100)/ 9900*0.0 /
   ! sets the upper and lower triangles
```
**GOTO Statement**

The GOTO statement:

- is a powerful but undisciplined branching statement;
- can be used to create jumps to almost anywhere in a program unit;
- can be dangerous;
- can lead to unstructured code (logical spaghetti).
- can be very useful!

The basic syntax is

```
GOTO <numeric-label>
```

The label must exist, be in the same scoping unit as the statement and be executable.
**GOTO Statement Example**

Consider the following example of an atrocious use of the GOTO statement,

```
GOTO 10    ! jump forward
23 CONTINUE
    i = i - 1
    IF (i .eq. 0) GOTO 99
10 PRINT*, "Line 10"
69 j = j - 1 ! loop
    ... 
    IF (j .ne. 0) GOTO 69
    GOTO 23    ! jump back
099 CONTINUE
```

The code fragment demonstrates forward and backward transfer of control and the simulation of a loop.

The best use of GOTO statements is in jumping out of a heavily nested structure when an unexpected event occurs such as a potential divide by zero.
RETURN and STOP

Can be used to program exceptions in procedures.

For example,

```
SUBROUTINE sub(ierr)
  INTEGER, INTENT(OUT) :: ierr
  ...
  ALLOCATE(A(100),STAT(ierr))
  IF ( ierr > 0 ) THEN
    PRINT*, 'memory fault'
    RETURN
  END IF
  ...
END SUBROUTINE
```

STOP could be used instead of RETURN:

```
STOP 'stopped in sub'
```

the string is optional or can be a literal integer constant of up to 5 digits. It is output upon execution of STOP at which time the program finishes.
Fortran 95 will be the new Fortran Standard.

- FORALL statement and construct
  
  ```fortran
  FORALL(i=1:n:2,j=1:m:2)
  A(i,j) = i*j
  END FORALL
  ```

- nested WHERE constructs,

- **ELEMENTAL** and **PURE** procedures,

- user-defined functions in initialisation expressions,

- automatic deallocation of arrays,

- improved object initialisation,

- remove conflicts with IEC 559 (**IEEE 754/854**) (floating point arithmetic),

- deleted features, for example, **PAUSE**, assigned **GOTO**, **CH** edit descriptor,

- more obsolescent features, for example, fixed source form, assumed sized arrays, **CHARACTER*< len >** declarations, statement functions,

- language tidy-ups and ambiguities (mistakes),
High Performance Fortran (or HPF) is an ad-hoc standard based on Fortran 90. It contains

- Fortran 90,
- syntax extensions, `FORALL`, new intrinsics, `PURE` and `ELEMENTAL` procedures,
- discussion regarding storage and sequence association,
- compiler directives:

```fortran
!HPF$ PROCESSORS P(5,7)
!HPF$ TEMPLATE T(20,20)
   INTEGER, DIMENSION(6,10) :: A
!HPF$ ALIGN A(J,K) WITH T(J*3,K*2)
!HPF$ DISTRIBUT T(CYCLIC(2),BLOCK(3)) ONTO P
```
The End