THE UNIVERSITY of LIVERPOOL

An Introduction to Fortran 90

(1 Day Seminar)

—

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

with acknowledgements to Steve Morgan and Lawrie Schonfelder.

Lecture 1: Overview of Fortran 90

Fortran Evolution

History:

- FORmula TRANslation.
- first compiler: 1957.
- first official standard 1972: 'Fortran 66'.
- 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.

Design Goals

A compromise between:

- Fortran 77 as a subset;
- efficiency;
- portability;
- regularity;
- ease of use;

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 (`KIND`s);
5. derived data types and operators;
6. recursion;
7. `MODULE`s

- □ procedure interfaces;
- □ enhanced control structures;
- □ user defined generic procedures;
- □ enhanced I/O.

Source Form

Free source form:

- 132 characters per line;
- extended character set;
- '!' comment initiator;
- '&' line continuation character;
- ';' statement separator;
- significant blanks.

New Style Declarations and Attributing

Can state `IMPLICIT NONE` meaning that variables must be declared.

Syntax

$<$ *type* $>$ [,$<$ *attribute*-list $>$] [`::`]`&`
$<$ *variable*-list $>$ [=$<$ *value* $>$]

The are no new data types. (If $<$ *attribute*-list $>$ or =$<$ *value* $>$ are present then so must be `::`.)

The following are all valid declarations,

```
SUBROUTINE Sub(x,i,j)
 IMPLICIT NONE
 REAL, INTENT(IN) :: x
 LOGICAL, POINTER :: ptr
 REAL, DIMENSION(10,10) :: y, z(10)
 CHARACTER(LEN=*), PARAMETER :: 'Maud''dib'
 INTEGER, TARGET :: k = 4
```

The `DIMENSION` attribute declares a 10×10 array, this can be overridden as with `z`.

New Control Constructs

- `IF` construct names for clarity (new relational and logical operators too),

```
zob: IF (A > 0) THEN
      ...
     ELSEIF (A == -1) THEN zob
      ...
     ELSE zob
 chum: IF (c == 0 .EQV. B >= 0) THEN
        ...
       ENDIF chum
      ...
     ENDIF zob
```

- `SELECT CASE` for integer and character expressions,

```
SELECT CASE (case_expr)
 CASE(1,3,5)
  ...
 CASE(2,4,6)
  ...
 CASE(7:10)
  ...
 CASE(11:)
  ...
 CASE DEFAULT
  ...
END SELECT
```

New Control Constructs

- `DO` names, `END DO` terminators, `EXIT` and `CYCLE`,

```
outa: DO i = 1,n
 inna: DO j = 1,m
       ...
      IF (X == 0) EXIT
       ...
      IF (X < 0) EXIT outa
       ...
      IF (X > 10) CYCLE inna
       ...
      IF (X > 100) CYCLE outa
       ...
     END DO inna
    END DO outa
```

- `DO WHILE` but this superseded by `EXIT` clause.

New Procedure Features

- internal procedures,

```
SUBROUTINE Subby(a,b,c)
 IMPLICIT NONE
  ...
  CALL Inty(a,c)
   ...
CONTAINS
 SUBROUTINE Inty(x,y)
  ...
 END SUBROUTINE Inty
END SUBROUTINE Subby
```

- INTENT attribute specify how variables are to be used,

```
INTEGER FUNCTION Schmunction(a,b,rc)
 IMPLICIT NONE ! New too
 REAL, INTENT(IN) :: a
 REAL, INTENT(INOUT) :: b
 INTEGER, INTENT(OUT) :: rc
  ...
END FUNCTION Schmunction ! New END
```

New Procedure Features

- `OPTIONAL` and keyword arguments,

```
SUBROUTINE Schmubroutine(scale,x,y)
 IMPLICIT NONE ! Use it
 REAL, INTENT(IN) :: x,y ! New format
 REAL, INTENT(IN), OPTIONAL :: scale
 REAL :: actual_scale
  actual_scale = 1.0
  IF (PRESENT(scale)) actual_scale = scale
  CALL Plot_line(x,y,actual_scale)
END SUBROUTINE Schmubroutine ! Neater
```

called as

```
CALL Schmubroutine(x=1.0,y=2.0)
CALL Schmubroutine(10.0,1.0,2.0)
```

- Explicit recursion is permitted,

```
RECURSIVE SUBROUTINE Factorial(N, Result)
 IMPLICIT NONE
 INTEGER, INTENT(IN)    :: N
 INTEGER, INTENT(INOUT) :: Result
  IF (N > 0) THEN
   CALL Factorial(N-1,Result)
   Result = Result * N
  ELSE
   Result = 1
  END IF
END SUBROUTINE Factorial
```

`EXTERNAL` Procedure Interfaces

- `INTERFACE` blocks,

```
INTERFACE
 SUBROUTINE Schmubroutine(scale,x,y)
  REAL, INTENT(IN) :: x, y
  REAL, INTENT(IN), OPTIONAL :: scale
 END SUBROUTINE Schmubroutine
END INTERFACE
```

  these are mandatory for `EXTERNAL` procedures with,

  - optional and keyword arguments;
  - pointer and target arguments;
  - new style array arguments;
  - array or pointer valued procedures.

New Array Facilities

- arrays as objects,

```
REAL, DIMENSION(10,10) :: A, B
REAL, ALLOCATABLE(:,:) :: C
REAL :: x = 1.0 ! new
 A = 10.0 ! scalar conformance
 B = A    ! shape conformance
```

- elemental operations,

```
B = x*A + B*B
```

- sectioning,

```
PRINT*, A(2:4,2:6:2)
B(:,10:1:-1) = A(:,:)
```

- array valued intrinsics,

```
B = SIN(A)
B(:,4) = ABS(A(:,5))
```

- masked assignment,

```
WHERE (A > 0.0) B = B/A
```

Program Packaging — Modules

- the `MODULE` program unit may contain
  - definitions of user types,
  - declarations of constants,
  - declaration of variables (possibly with initialisation),
  - accessibility statements,
  - definition of procedures,
  - definition of interfaces for external procedures,
  - declarations of generic procedure names and operator symbols,

  the above provides basis of object oriented technology.

- the `USE` statement,
  - names the particular `MODULE`,
  - imports the public objects,

- provides global storage without `COMMON`,

Stack Example

```
MODULE stack
 IMPLICIT NONE
 PRIVATE
 INTEGER, PARAMETER :: stack_size = 100
 INTEGER, SAVE :: store(stack_size), pos = 0
 PUBLIC push, pop
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 MODULE stack
```

Rational Arithmetic Example

```
MODULE RATIONAL_ARITHMETIC
 TYPE RATNUM
  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
 CONTAINS
  TYPE(RATNUM) FUNCTION rat_rat(l,r)
   TYPE(RATNUM), INTENT(IN) :: l,r
   rat_rat%num = l%num * r%num
   rat_rat%den = l%den * r%den
  END FUNCTION rat_rat
  TYPE(RATNUM) FUNCTION int_rat(l,r)
   INTEGER, INTENT(IN) :: l
   TYPE(RATNUM), INTENT(IN) :: r
    ...
   END FUNCTION int_rat
   FUNCTION rat_int(l,r)
    ...
   END FUNCTION rat_int
 END MODULE RATIONAL_ARITHMETIC
 PROGRAM Main;
  USE RATIONAL_ARITHMETIC
  INTEGER :: i = 32
  TYPE(RATNUM) :: a,b,c
   a = RATNUM(1,16); b = 2*a; c = 3*b
   b = a*i*b*c; PRINT*, b
 END PROGRAM Main
```

User Defined Entities

- Define Type

```
TYPE person
 CHARACTER(LEN=20) :: name
 INTEGER :: age
 REAL  ::  height
END TYPE person
TYPE couple
 TYPE(person) :: he, she
END TYPE couple
```

- Declare structure

```
TYPE(person) :: him, her
TYPE(couple) :: joneses
```

- Component selection

```
him%age, her%name, joneses%he%height
```

- Structure constructor

```
him = person('Jones', 45, 5.8)
them = couple(person(...),person(...))
```

Operators and Generics

- Overloaded operators and assignment

```
INTERFACE OPERATOR (+)
 ... ! what + means in this context
END INTERFACE ! OPERATOR (+)
INTERFACE ASSIGNMENT (=)
 ... ! what = means in this context
END INTERFACE ! ASSIGNMENT (=)
 ...
joneses = him+her
```

- Defined operators

```
INTERFACE OPERATOR (.YOUNGER.)
 ... ! what .YOUNGER. means
END INTERFACE ! OPERATOR (.YOUNGER.)
 ...
IF (him.YOUNGER.her) ...
```

- Generic interfaces (intrinsic and user defined),

```
INTERFACE LLT
 ... ! what LLT means in this context
END INTERFACE ! LLT
INTERFACE My_Generic
 ... ! what My_Generic means in this context
END INTERFACE ! My_Generic
 ...
IF (LLT(him,her)) ...
```

Pointers

- Objects declared with the `POINTER` attribute

```
REAL, DIMENSION(:,:), POINTER :: pra, prb
```

  `pra` is a descriptor for a 2D array of reals,

- objects to be referenced must have `TARGET` attribute,

```
REAL, DIMENSION(-10:10,-10:10), TARGET :: a
```

- a pointer is associated with memory by allocation,

```
ALLOCATE(prb(0:n,0:2*n*n),STAT=ierr)
```

- pointer assignment,

```
pra => a(-k:k,-j:j)
```

  `{\tt pra} is now an alias for part of {\tt a}.`

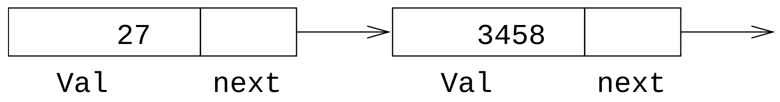
- pointers are automatically dereferenced, in expressions they reference the value(s) stored in the current target,

```
pra(15:25,5:15) = pra(10:20,0:10) + 1.0
```

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 subtlely different from normal,

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

is equivalent to:

```
A%val = B%val
A%next => B%next
```

Parameterised Data Types

- Intrinsic types can be parameterised to select accuracy and range of the representation,
- for example,

  ```
  INTEGER(KIND=2) :: i
  INTEGER(KIND=k) :: j
  REAL(KIND=l) :: x
  ```

  where `k` and `m` are default integer constant expressions and are called kind values,
- can have constants

  ```
  24_2, 207_k, 1.08_l
  ```

- `SELECTED_INT_KIND`, `SELECTED_REAL_KIND` can be parameterised and return kind value of appropriate representation. This gives portable data types.

```
INTEGER, PARAMETER :: k = SELECTED_INT_KIND(2)
INTEGER, PARAMETER :: l = SELECTED_REAL_KIND(10,68)
```

- a generic intrinsic function `KIND(object)` returns the kind value of the object representation:
  - `KIND(0.0)` is kind value of default `REAL`.
  - `KIND(0_k)` is `k`.

New I/O Features

- normal Fortran I/O always advances to the next record for any `READ` or `WRITE` statement,

- Fortran 90 supports non-advancing form of I/O added,

```
WRITE(...,ADVANCE='NO',...) a
```

appends output characters to the current record and

```
READ(...,ADVANCE='NO',...) a
```

reads from the next available character in a file

```
READ(...,ADVANCE='NO',EOR=99,SIZE=nch) a
```

detects end of record and `nch` will contain the number of characters actually read.

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;
- `ASSIGN`ed `GOTO` statements;
- `ASSIGN`ed `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;

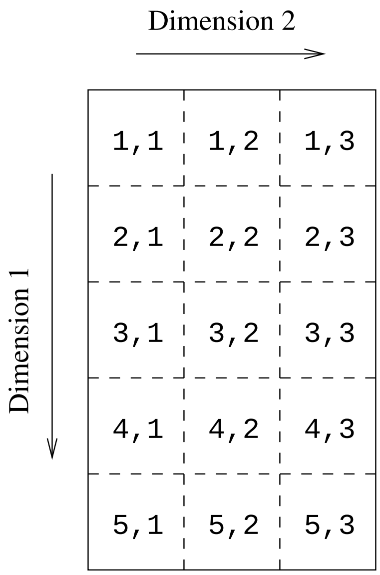
Lecture 2:
Arrays

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:

And a 5 $\times$ 3 array as:



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

Array Terminology

Examples of declarations:

```
REAL, DIMENSION(15)      :: X
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,

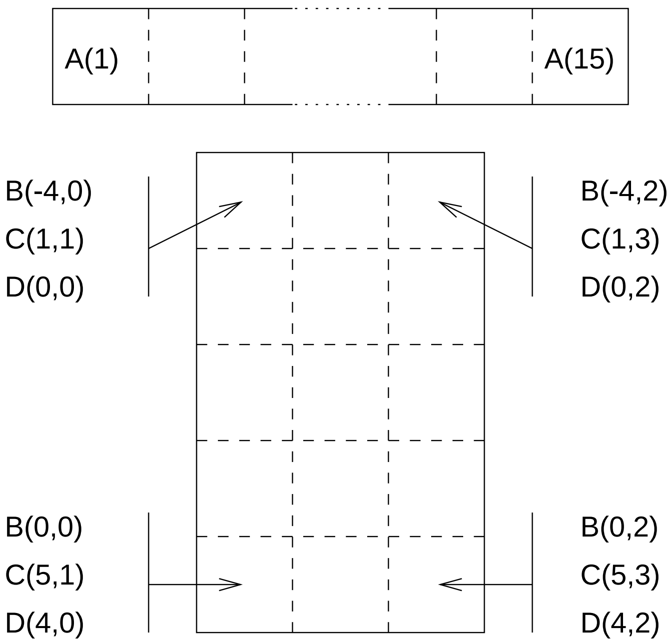
```
REAL, DIMENSION(100)       :: R
REAL, DIMENSION(1:10,1:10) :: S
REAL                       :: T(10,10)
REAL, DIMENSION(-10:-1)    :: X
INTEGER, PARAMETER            :: lda = 5
REAL, DIMENSION(0:lda-1)      :: Y
REAL, DIMENSION(1+lda*lda,10) :: Z
```

- 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)` 7^{th} element of `A`, or `C(3,2)`, 3 elements down, 2 across.



Array Conformance

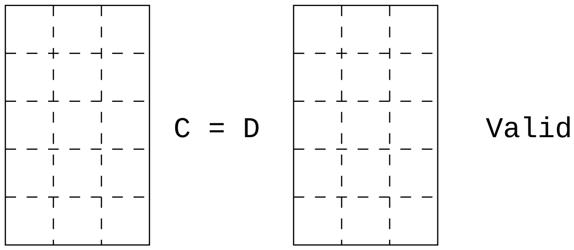
Arrays or sub-arrays must conform with all other objects in an expression:

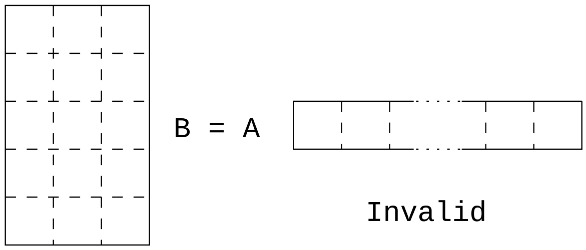
- a scalar conforms to an array of any shape with the same value for every element:

```
C = 1.0   ! is valid
```

- two array references must conform in their shape.

  Using the declarations from before:





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

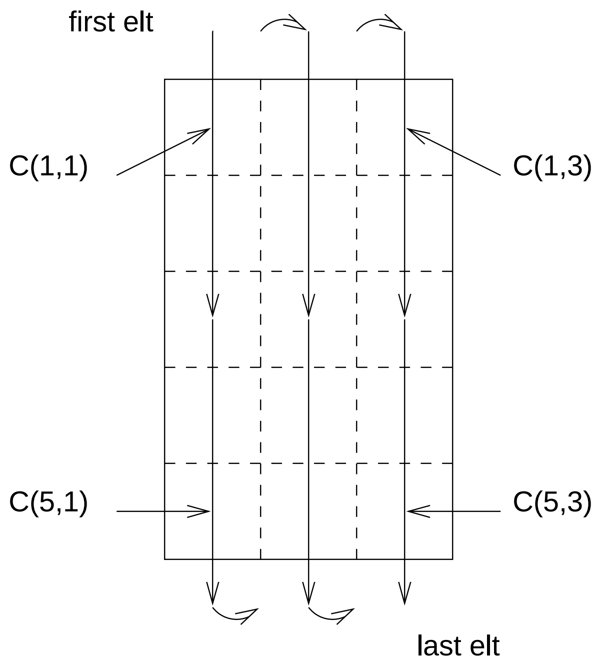
Array Element Ordering

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),..,C(5,1),C(1,2),C(2,2),..,C(5,3)
```

Array Syntax

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`.

Whole Array Expressions

Arrays can be treated like a single variable in that:

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

```
B = C * D - B**2
```

  this is equivalent to concurrent execution of:

```
B(-4,0) = C(1,1)*D(0,0)-B(-4,0)**2 ! in ||
B(-3,0) = C(2,1)*D(1,0)-B(-3,0)**2 ! in ||
 ...
B(-4,1) = C(1,2)*D(0,1)-B(-4,1)**2 ! in ||
 ...
B(0,2)  = C(5,3)*D(4,2)-B(0,2)**2  ! in ||
```

- elemental intrinsic functions can be used,

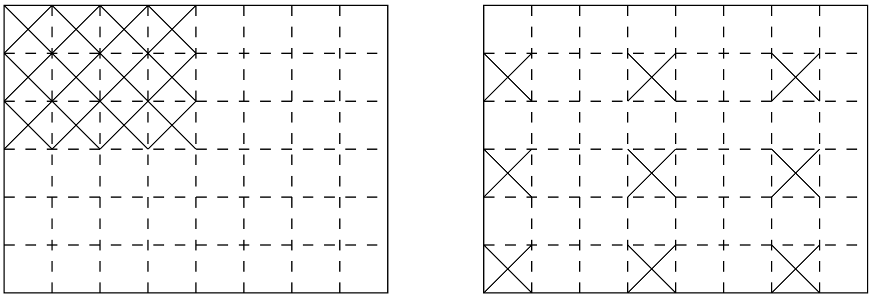
```
B = SIN(C)+COS(D)
```

  the function is applied element by element.

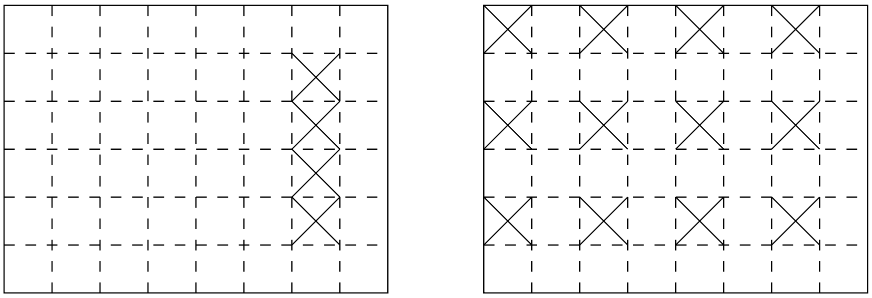
Array Sections — Visualisation

Given,

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



P(1:3,1:4) P(2:6:2,1:7:3)



P(2:5,7) P(2:5,7:7) P(1:6:2,1:8:2)

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 Inquiry Intrinsics

These are often useful in procedures, consider the declaration:

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

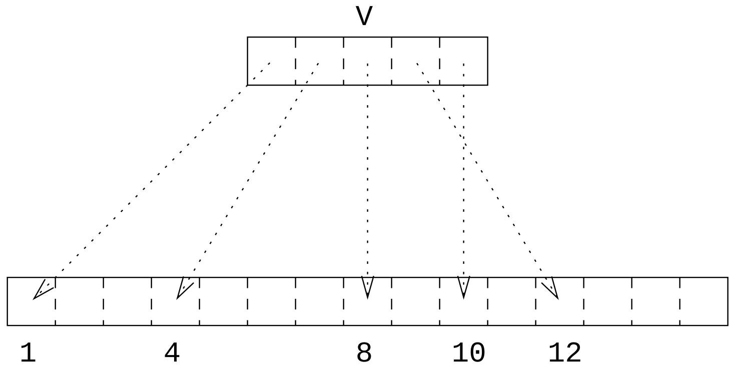
- `LBOUND(SOURCE[,DIM])` — lower bounds of an array (or bound in an optionally specified dimension).
  - `LBOUND(A)` is `(/-10,1,14/)` (array);
  - `LBOUND(A,1)` is -10 (scalar).
- `UBOUND(SOURCE[,DIM])` — upper bounds of an array (or bound in an optionally specified dimension).
- `SHAPE(SOURCE)` — shape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

Vector-valued Subscripts

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

```
INTEGER, DIMENSION(5) :: V = (/1,4,8,12,10/)
INTEGER, DIMENSION(3) :: W = (/1,2,2/)
```

- `A(V)` is `A(1)`, `A(4)`, `A(8)`, `A(12)`, and `A(10)`.



- the following are valid assignments:

```
A(V) = 3.5
C(1:3,1) = A(W)
```

- it would be invalid to assign values to `A(W)` as `A(2)` is referred to twice.

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

```
A(1) = SUM(C(V,W))
```

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:

```
RESHAPE(SOURCE, SHAPE)
```

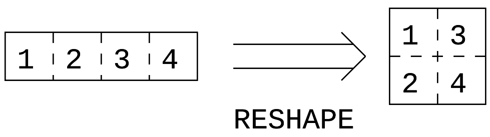
For example,

```
A = RESHAPE((/1,2,3,4/),(/2,2/))
```

`A` is filled in array element order and looks like:

```
1  3
2  4
```

Visualisation,



Allocatable Arrays

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

- Declaration:

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

  Note `ALLOCATABLE` attribute and fixed rank.

- Allocation:

  ```
  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:

```
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 `DEALLOCATE`d 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

$$\mathtt{B} = \begin{pmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \end{pmatrix}$$

and,

$$\mathtt{I} = \begin{pmatrix} \boxed{2} & 0 \\ 0 & \boxed{2} \end{pmatrix}$$

then

$$\mathtt{A} = \begin{pmatrix} \boxed{0.5} & \cdot \\ \cdot & \boxed{2.0} \end{pmatrix}$$

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:

```
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.

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.

Assumed-shape Arrays

Should declare dummy arrays as assumed-shape arrays:

```
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.

Automatic Arrays

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,

```
PROGRAM Main
 IMPLICIT NONE
  INTEGER :: IX, IY
 .....
  CALL une_bus_riot(IX,2,3)
  CALL une_bus_riot(IY,7,2)
CONTAINS
 SUBROUTINE une_bus_riot(A,M,N)
  INTEGER, INTENT(IN) :: M, N
  INTEGER, INTENT(INOUT) :: A(:,:)
  REAL :: A1(M,N)                  ! auto
  REAL :: A2(SIZE(A,1),SIZE(A,2)) ! auto
   ...
 END SUBROUTINE
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.

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,

```
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.

```
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)*B(1) + A(2)*B(2) + ...
  ```

  For LOGICAL arrays, the corresponding operation is a logical .AND..

  ```
  DP = LA(1) .AND. LB(1) .OR. &
       LA(2) .AND. LB(2) .OR. ...
  ```

- 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..

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,

`MAXLOC(X) = (/6/)`

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

A 2D example. If

$$\mathtt{Array} = \begin{pmatrix} 0 & -1 & 1 & 6 & -4 \\ 1 & -2 & 5 & 4 & -3 \\ 3 & 8 & 3 & -7 & 0 \end{pmatrix}$$

then

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

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

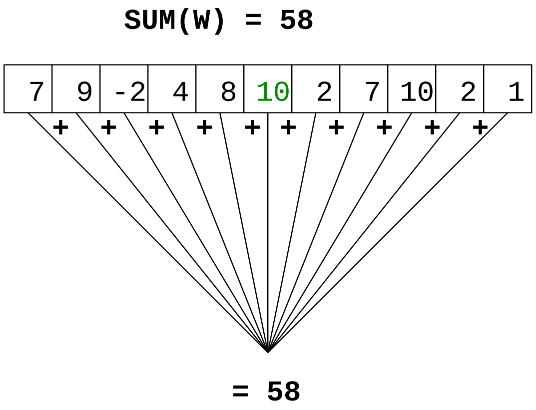
- `MAXLOC(MAXLOC(Array,Array.LE.7))` is `(/2/)` (array valued).

Array Reduction Intrinsics

- `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:



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 3:
Modules

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;

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 — Global Data

Fortran 90 implements a new mechanism to implement global data:

- declare the required objects within a module;
- give them the `SAVE` attribute;
- `USE` the module when global data is needed.

For example, to declare `pi` as a global constant

```
MODULE  Pye
 REAL, SAVE :: pi = 3.142
END MODULE Pye

PROGRAM Area
 USE Pye
 IMPLICIT NONE
 REAL :: r
  READ*, r
  PRINT*, "Area= ",pi*r*r
END PROGRAM Area
```

`MODULE`s should be placed *before* the program.

Module Global Data Example

For example, the following defines a very simple 100 element integer stack

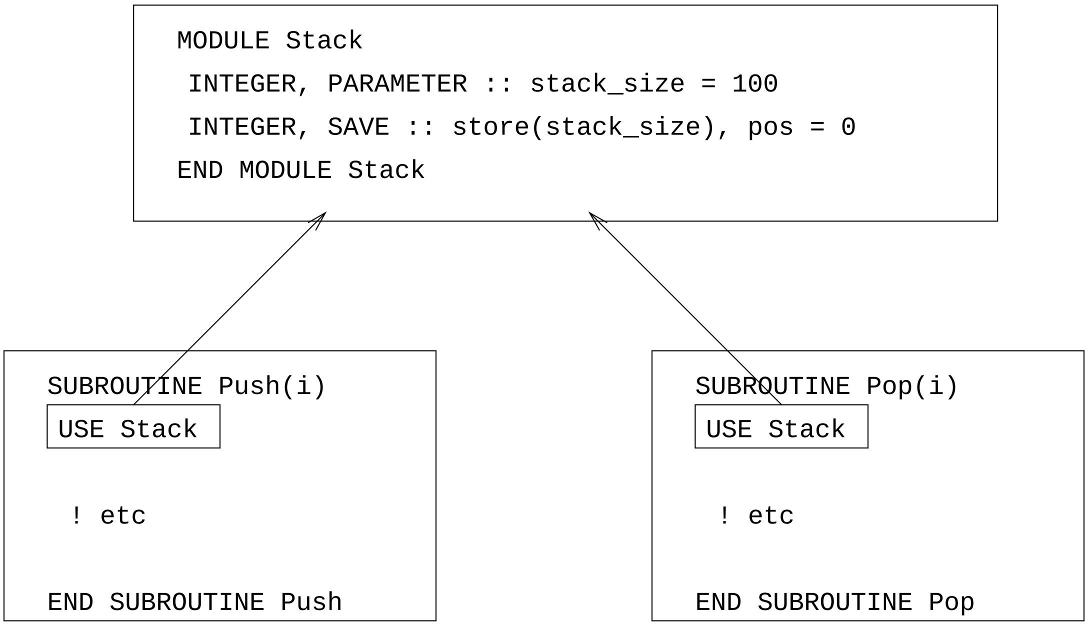
```
MODULE stack
 INTEGER, PARAMETER :: stack_size = 100
 INTEGER, SAVE :: store(stack_size), pos=0
END MODULE stack
```

and two access functions,

```
SUBROUTINE push(i)
 USE stack
 IMPLICIT NONE
  ...
END SUBROUTINE push
SUBROUTINE pop(i)
 USE stack
 IMPLICIT NONE
  ...
END SUBROUTINE pop
```

A main program can now call `push` and `pop` which simulate a 100 element `INTEGER` stack — this is much neater than using `COMMON` block.

Visualisation of Global Storage



Both procedures access the same (global) data in the `MODULE`.

Modules — Procedure Encapsulation

Module procedures are specified after the CONTAINS separator,

```
MODULE related_procedures
 IMPLICIT NONE
 ! INTERFACEs of MODULE PROCEDURES do
 ! not need to be specified they are
 ! 'already present'
CONTAINS
 SUBROUTINE sub1(A,B,C)
  ! Can see Sub2's INTERFACE
  ...
 END SUBROUTINE sub1
 SUBROUTINE sub2(time,dist)
  ! Can see Sub1's INTERFACE
  ...
 END SUBROUTINE sub2
END MODULE related_procedures
```

The main program attaches the procedures by *use-association*

```
PROGRAM use_of_module
 USE related_procedures ! includes INTERFACES
 CALL sub1((/1.0,3.14,0.57/),2,'Yobot')
 CALL sub2(t,d)
END PROGRAM use_of_module
```

`sub1` can call `sub2` or vice versa.

Encapsulation - Stack example

We can also encapsulate the `stack` program,

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

Any program unit that includes the line:

```
USE stack
CALL push(2); CALL push(6); ..
CALL pop(i); ....
```

can access `pop` and `push` therefore use the 100 element global integer stack.

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 Type Constructors

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:

```
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)
  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 ABS(X) comprises the specific functions:

- ☐ CABS — called when X is COMPLEX,
- ☐ ABS — called when X is REAL,
- ☐ IABS — called when X is INTEGER,

These specific functions are called the *overload set*.

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

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

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

Generic Interfaces - Example

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 - Commentry

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.

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,

```
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.

Overloading Operators

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 PROCEDURE`s (`FUNCTION`s) 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 = ...
  FUNCTION int_rat(l,r)      ! int * rat
   INTEGER, INTENT(IN)      :: l
   TYPE(RATNUM), INTENT(IN) :: r
   ...
  FUNCTION rat_int(l,r)      ! rat * int
   TYPE(RATNUM), INTENT(IN) :: l
   INTEGER, INTENT(IN)      :: r
   ...
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,

$.<\textit{name}>.$

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.)
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 `INTENT(OUT)`;

- the second is the expression whose value is converted and must have `INTENT(IN)`.

Overloading the assignment operator differs from other operators:

- assignment overload sets do **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 `SUBROUTINE`s 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`.

Restricting Visibility

- Objects in a `MODULE` can be given visibility attributes:

```
PRIVATE :: rat_ass_int, real_ass_rat
PRIVATE :: rat_int, int_rat, rat_rat
PUBLIC :: OPRATOR(*)
PUBLIC :: ASSIGNMENT(=)
```

  only allows access to symbolic versions of multiply and assignment (* and =).

- This allows the internal structure of a module to be changed without modifying the users program.

- default visibility is `PUBLIC`, this can be reversed by a `PRIVATE` statement.

- individual declarations can also be attributed,

```
INTEGER, PRIVATE :: Intern
```

Derived Types with Private Components

The type `RATNUM` is declared with `PRIVATE` internal structure,

```
    TYPE RATNUM
     PRIVATE
     INTEGER :: num, den
    END TYPE RATNUM
```

The user is unable to access specific components,

```
    TYPE (RATNUM) :: splodge
     splodge = RATNUM(2,3) ! invalid
     splodge%num = 2       ! invalid
     splodge%den = 3       ! invalid
     splodge = set_up_RATNUM(2,3) ! OK
 ! set_up_RATNUM must be module procedure
     CALL Print_out_RATNUM(splodge)
 ! Print_out_RATNUM must be module procedure
```

this allows the internal representation of the type to be changed:

```
    TYPE RATNUM
     PRIVATE
     REAL :: numb
    END TYPE RATNUM
```

Accessibility Example

We can update our stack example,

```
MODULE stack
 IMPLICIT NONE
 PRIVATE
 INTEGER, PARAMETER :: stack_size = 100
 INTEGER, SAVE :: store(stack_size), pos = 0
 PUBLIC push, pop
CONTAINS
 SUBROUTINE push(i)
  INTEGER, INTENT(IN) :: i
   ... ! as before
 END SUBROUTINE push
 SUBROUTINE pop(i)
  INTEGER, INTENT(OUT) :: i
   ... ! as before
 END SUBROUTINE pop
END MODULE stack
```

User cannot now alter the value of `store` or `pos`.

Another Accessibility Example

The visibility specifiers can be applied to all objects including type definitions, procedures and operators:

For example,

```
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.

The USE Renames Facility

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

Syntax:

```
USE < module-name > &
        [,< new-name > => < use-name >...]
```

module entities can be renamed,

```
USE Stack, IntegerPop => 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.

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 4: Miscellaneous Features

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,

```
INTEGER(KIND=1) :: ik1
REAL(4) :: rk4
```

- 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,

```
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_long
```

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

- There are other pitfalls too, the constant

```
1000_short
```

  may not be valid as `KIND = 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,

```
print*, INT(1.0,KIND=3), NINT(1.0,KIND=3)
x = x + REAL(j,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

  ```
  INTEGER(short) :: members, attendees
  INTEGER(long)  :: salaries, costs
  ```

  the expression:

  - `members + attendees` is of kind `short`,
  - `salaries - costs` is of kind `long`,
  - `members * costs` is also of kind `long`.

- Care must be taken to ensure the LHS is able to hold numbers returned by the RHS.

Kinds and Procedure Arguments

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

```
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,

```
 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.

Logical KIND Selection

- There is no SELECTED_LOGICAL_KIND intrinsic, however, the KIND intrinsic can be used as normal.

  For example,

  ```
  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.

Mathematical Intrinsic Functions

Summary,

`ACOS(x)`	arccosine
`ASIN(x)`	arcsine
`ATAN(x)`	arctangent
`ATAN2(y,x)`	arctangent of complex number (x, y)
`COS(x)`	cosine where x is in radians
`COSH(x)`	hyperbolic cosine where x is in radians
`EXP(x)`	e raised to the power x
`LOG(x)`	natural logarithm of x
`LOG10(x)`	logarithm base 10 of x
`SIN(x)`	sine where x is in radians
`SINH(x)`	hyperbolic sine where x is in radians
`SQRT(x)`	the square root of x
`TAN(x)`	tangent where x is in radians
`TANH(x)`	tangent where x is in radians

Numeric Intrinsic Functions

Summary,

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

Character Intrinsic Functions

Summary,

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

Bit Manipulation Intrinsic Functions

Summary,

`BTEST(i,pos)`	bit testing
`IAND(i,j)`	AND
`IBCLR(i,pos)`	clear bit
`IBITS(i,pos,len)`	bit extraction
`IBSET(i,pos)`	set bit
`IEOR(i,j)`	exclusive OR
`IOR(i,j)`	inclusive OR
`ISHFT(i,shft)`	logical shift
`ISHFTC(i,shft)`	circular shift
`NOT(i)`	complement
`MVBITS(ifr,ifrpos, len,ito,itopos)`	move bits (`SUBROUTINE`)

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

s-1		*3*	*2*	*1*	*0*	
0	..	0	0	0	0	value = 0

s-1		*3*	*2*	*1*	*0*	
0	..	0	1	0	1	value = 5

s-1		*3*	*2*	*1*	*0*	
0		0	0	1	1	value = 3

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.

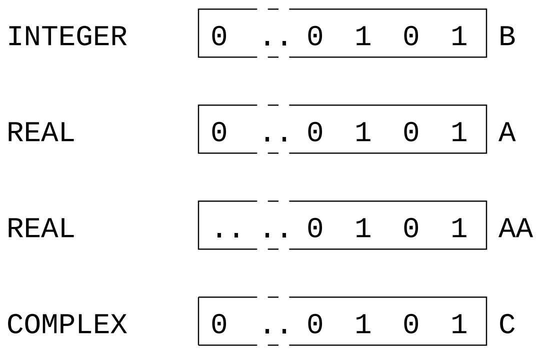
`TRANSFER` Intrinsic

`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) /))
 ...
```



Fortran 95

Fortran 95 will be the new Fortran Standard.

- FORALL statement and construct

```
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

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:

```
!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$ DISTRIBUTE T(CYCLIC(2),BLOCK(3)) ONTO P
```

Data Alignment