THE UNIVERSITY
of MANCHESTER

Fortran 90

A Conversion Course for Fortran 77 Programmers

Speaker Notes

F Lin, S Ramsden

Manchester and North HPC T&EC

M A Pettipher, J M Brooke, G S Noland

Manchester Computing Centre, University of Manchester

MAN T&EC
The Manchester and North HPC Training and Education Centre

Edition 3.0 July 1995

Fortran 90

A Conversion Course for Fortran 77 Programmers

The course presents an overview of Fortran 90 and explains the features that are new to the language.

On line access with examples and exercises is an integral part of the course.

Prerequisites are: ability to program in Fortran or experienced programmer in other high level language.

Acknowledgements

These slides were developed using the Manchester Computing Centre Fortran 90 course as a basis, which was compiled by:

- M A Pettipher
- J M Brooke
- G S Noland

We are grateful to Dr. A C Marshall from the University of Liverpool and Michael Hennecke from the University of Karlsruhe for their very useful comments.

Thanks to system group at MCC, especially Miss L J Savage, for supporting the Fortran 90 compilers at MCC.

Contents

- Introduction
- Source, Types and Control Structures
- Procedures and Modules
- Array Processing
- Pointers
- New I/O Features
- Intrinsic Procedures
- Redundant Features
- Further Development

The contents of the Student Notes are kept in the same order of the OHP masters, which are cross referenced in the Speaker Notes.

Introduction

Introduction

Topics

- History
- Objectives
- Major new features
- Other new features
- Availability of compilers
- References
- Resource list
- Fortran 90 at MCC
- Coding convention

History

- 1957 IBM Fortran
- 1966 Fortran 66 standard
- 1978 Fortran 77 standard
- 1991 Fortran 90 standard

The development of Fortran dates back to the 1950's, the first Fortran system being released in 1957, for the IBM 704.

A standards committee was established in 1962, and the first Fortran standard was published in 1966.

In the 1970s a new standard was formulated to overcome the problems of Fortran 66 and incorporate several new features. In 1978, the new standard, Fortran 77 was published.

The standards preceding Fortran 90 attempted mainly to standardise existing extensions and practices.

Fortran 90, however, is much more an attempt to develop the language, introducing new features using experience from other languages.

See page 1 of the Student Notes.

Objectives

- Language evolution
  - Obsolescent features
- Standardise vendor extensions
  - Portability
- Modernise the language
  - Ease-of-use improvements through new features such as free source form and derived types
  - Space conservation of a program with dynamic memory allocation
  - Modularisation through defining collections called modules
  - Numerical portability through selected precision

The obsolescent features are considered to be redundant and are recommended for deletion in the next revision, which provides a means to remove features and for the language to evolve.

To modernise the language in response to the developments in language design which have been exploited in other languages.

See page 1 of the Student Notes.

Objectives

- Provide data parallel capability
  - Parallel array operations for better use of vector and parallel processors
- Compatibility with Fortran 77
  - Fortran 77 is a subset of Fortran 90
- Improve safety
  - Reduce risk of errors in standard code
- Standard conformance
  - Compiler must report non standard code and obsolescent features

The ability of parallel array operations is one of the most significant new features in Fortran 90.

The vast investment in Fortran 77 code is preserved by keeping compatible with Fortran 77.

The storage association in Fortran 77, such as COMMON/EQUIVALENCE, which is quite risky to use, can be obviated by the new features.

Major new features

- Array processing
- Dynamic memory allocation
- Modules
- Procedures:
  - Optional/Keyword Parameters
  - Internal Procedures
  - Recursive Procedures
- Pointers

These are the most important topics for numerical programming in the scientific and engineering field.

See page 2 of the Student Notes.

Other new features

- Source form
- Specifications/IMPLICIT NONE
- Parameterised data types (KIND)
- Derived types and operator overloading
- Control structures
- New intrinsic functions
- New I/O features

Notice that many new features are interdependent. For example, the value of array syntax would be greatly reduced without array intrinsic functions.

The new features allow the writing of more readable compact code, resulting in more understandable modular programs with increased functionality.

Numerical portability is provided through selected precision, programming errors are reduced by the use of explicit interfaces to sub-programs and memory space is conserved via dynamic memory allocation.

Additionally, data parallel capability is provided through the array processing features, which make Fortran 90 a more efficient language on the new generation of high performance computers.

See page 2 of the Student Notes.

Obsolescent features

May be removed at the next major revision:

- Arithmetic IF
- REAL and DOUBLE precision DO variables and control expressions
- Shared DO termination, and DO termination on a statement other than on a CONTINUE or an END DO statement
- ASSIGN and assigned GO TO statements
- Assigned FORMAT specifiers
- Branching to END IF from outside IF block
- Alternate RETURN
- PAUSE statement
- H edit descriptor

May be in 10 years? (some will be removed in Fortran 95).

Keep in mind that these obsolescent features should never be used in the new or revised programs.

Availability

Fortran 90 compilers

- **Cray CF90** - YMP (EL98), vectorising and autotasking, but limited messages and some intrinsics not tuned
- **DEC Fortran 90** - DEC OSF/1 AXP, including HPF extensions
- **EPC Fortran 90** - SPARC Solaris 1X and 2X, IBM RS/6000, Intel 3/486, Motorola 88000
- **IBM XLF V3** - RISC System 6000
- **Lahey LF90** - for DOS, Windows
- **Microway** - DOS, OS/2, Unix

Foreseen:

CRAY full compiler CF90 in a Sparc version in 94.

Fujitsu full compiler (Beta test as of August 94).

DEC Fortran 90 for OpenVMS AXP (Beta test last quarter 94).

Sun full compiler (unofficially end-94)

MicroSoft after next release of present PowerStationFortran.

KAP pre-processor (IBM F90 late 09/94, SPARC F90 beta as of 09/94, SGI F90 1st quarter 95).

Availability

Fortran 90 compilers

- **NA Software F90+** - 386/486, Sparc, T800, T9000
- **NAG f90** - Variety of Unix platforms, and VAX VMS. Uses C as intermediate language
- **Pacific Sierra VAST-90** - Uses F77 as intermediate language
- **Parasoft** - Uses F77 as intermediate language
- **Salford FTN90** - PC implementation of NAG f90, direct generation of object code

References

- Information Technology - Programming Languages - Fortran

  on-line or paper, the official standard, ISO/IEC 1539:1991 or ANSI x3.198-1992
- Fortran 90 Handbook
  Complete ANSI/ISO Reference

  J C Adams et. al., McGraw-Hill, 1992
- Programmer's Guide to Fortran 90

  2nd edition, W S Brainerd et. al., Unicomp, 1994
- Fortran 90

  M Counihan, Pitman, 1991
- Fortran 90 Programming

  T M R Ellis et. al., Wesley, 1994

Quite interesting comments for a lot of references can be found in Bo Einarsson, and Yurij Shokin "Fortran 90 for the Fortran 77 programmer", which is available on WWW, (see Slide 16: Resource list).

For Ellis's book, Kerrigan's book and Smith's book (see the next slide), the example code is available via FTP (the FTP address can be found in the book).

References

- Fortran 90 for Scientists and Engineers

  B D Hahn, Edward Arnold, 1994

- Migrating to Fortran 90

  J Kerrigan, O'Reilly and Associates, 1993

- Fortran 90 Explained

  M Metcalf & J Ried, Oxford University Press, 1992

- Programming in Fortran 90

  J S Morgan & J L Schonfelder, Alfred Waller Ltd., 1993

- Programming in Fortran 90

  I M Smith, Wiley

Resource list

The compiled resource list is available on WWW:

http://www.hpctec.mcc.ac.uk/hpctec/courses/Fortran90/resource.html

which has links to:

- Fortran market
- Fortran 90 FAQ
- Tutorials and articles available on WWW
- NAG Fortran 90 software repository
- Fortran 90/HPF email discussion group
- Fortran newsgroup in Netnews/Usenet

Fortran 90 FAQ contains a lot of useful information, which is highly recommended.

The London & South-East centre for High Performance Computing holds an extensible article archive on HPC. In Fortran 90 and HPF subject area, there are a lot tutorials and articles which are free.

NAG is supporting a web-based public Fortran 90 Software Repository. The service can be used for uploading Fortran 90 files which others can then examine, recover and use.

The general information and whole mail archive about comp-fortran-90 can be explored via the link.

The Fortran Market is one place to find all information, products, and services related to Fortran.

Fortran 90 at MCC

- Cray EL98

  Compile with f90

- NAG f90 on Cray CS6400

  Compile with f90

- Salford FTN90 on fileserver MCC.FS1

  Compile and run with ftn90/lgo

- IBM

  Compile with xlf90 (Note default code generated for ibmb, ibmc, ibmd, NOT ibma)

Coding convention

Recommended by NAG and others

- Put all Fortran 90 keywords and intrinsic function names in upper case, everything else in lower case
- Indent by 2 columns in the body of program units and INTERFACE blocks, DO-loops, IF-blocks, CASE-blocks etc.
- Always include the name of a program, a subroutine and a function on its END statement
- In USE statements, use the ONLY clause to document explicitly all entities which are actually accessed from that module
- In CALL statement and function references, always use argument keywords for optional arguments

See page 3 of the Student Notes.

Sources
Types
Control Structures

Sources, types and control structures

Topics

- Source form
- Specifications
- IMPLICIT NONE
- Kind values
- Derived types
- Control structures

Source form

Free source form

- Lines up to 132 characters
- Lowercase letters permitted
- Names up to 31 characters (including underscore)
- Semicolon to separate multiple statements on one line
- Comments may follow exclamation (!)
- Ampersand (&) is a continuation symbol
- Character set includes + < > ; ! ? % - “ &
- New relational operators: ‘<’, ‘<=’, ‘==’, ‘/=’, ‘>=’, ‘>’

Fortran 90 supports two forms of source code, the old Fortran 77 source code form now called fixed form, and the new free form.

For new or revised programs, the free source form is recommended.

See page 5 of the Student Notes.

Source form

Example

```
PROGRAM free_source_form
! Long names with underscores

    ! No special columns
    IMPLICIT NONE

    ! upper and lower case letters
    REAL :: tx, ty, tz          ! trailing comment

    ! Multiple statements per line
    tx = 1.0; ty = 2.0; tz = tx * ty;

    ! Continuation on line to be continued
    PRINT *, &
          tx, ty, tz

END PROGRAM free_source_form
```


Programs exist to convert from Fortran 77 fixed source form to Fortran 90 free source form, e.g., Metcalf’s Fixed to Free Form convertor or Translate Program in NAG-Ware f90 Tools, which can be retrieved from NAG Fortran 90 software repository.

Note the incompatibility between source forms: continuation lines and comments. So can not treat Fortran 77 code as Fortran 90 free source format unless the code has no comments nor continuations!

Specifications

type [[,_attribute_]... ::] entity list

- _type_ can be INTEGER, REAL, COMPLEX, LOGICAL or CHARACTER with optional kind value:
  - INTEGER [(KIND=] kind-value)]
  - CHARACTER ([actual parameter list])

    ([LEN=] len-value and/or [KIND=] kind-value)
  - TYPE (type name)
- _attribute_ can be

  PARAMETER, PUBLIC, PRIVATE,
  ALLOCATABLE, POINTER, TARGET,
  INTENT(inout), DIMENSION (extent-list),
  OPTIONAL, SAVE, EXTERNAL,
  INTRINSIC
- Can initialise variables in specifications

The initialisation of variables in specifications is an alternative to the DATA statement, but the DATA statement is still needed for hexadecimal and octal initialisation.

The double colon :: are required if attribute used or initialisation performed (always recommend to use).

Note attribute form of many existing statements in Fortran 77: PARAMETER, SAVE, EXTERNAL.

See page 6 of the Student Notes.

Specifications

Examples

```
! Integer variables:
INTEGER :: ia, ib

! Parameters:
INTEGER, PARAMETER :: n=100, m=1000

! Initialisation of variables:
REAL :: a = 2.61828, b = 3.14159

! Character variable of length 8:
CHARACTER (LEN = 8) :: ch

! Integer array with negative lower bound:
INTEGER, DIMENSION(-3:5, 7) :: ia

! Integer array using default dimension:
INTEGER,DIMENSION(-3:5, 7) :: ia, ib, ic(5, 5)
```

IMPLICIT NONE

- In Fortran 77, implicit typing permitted use of undeclared variables. This has been the cause of many programming errors.
- IMPLICIT NONE forces you to declare all variables.
- IMPLICIT NONE may be preceded in a program unit only by USE and FORMAT statements (see Order of statements).

For backward compatibility, the implicit typing of integers and reals by the first character is carried over.

The IMPLICIT NONE statement is recommended to be included in all program units. This switches off implicit typing and so all variables must be declared, and helps to catch errors at compile time when they are easier to correct.

See page 7 of the Student Notes.

Kind values

- 5 intrinsic types: REAL, INTEGER, COMPLEX, CHARACTER, LOGICAL
- Each type has an associated non negative integer value called the KIND type parameter
- Useful feature for writing portable code requiring specified precision
- A processor must support at least 2 kinds for REAL and COMPLEX, and 1 for INTEGER, LOGICAL and CHARACTER
- Many intrinsics for enquiring about and setting kind values

Simplest explanation is, for example, to replace REAL*4, and REAL*8 (or REAL and DOUBLE PRECISION).

This new concept requires “playing with”, as it is not obvious from any explanation exactly what is going on.

See page 7 of the Student Notes.

Kind values

REAL

```
REAL (KIND = wp) :: ra          ! or
REAL(wp) :: ra
```

- Declare a real variable, ra, whose precision is determined by the value of the kind parameter, wp
- Kind values are system dependent
- An 8 byte (64 bit) real variable usually has kind value 8 or 2
- A 4 byte (32 bit) real variable usually has kind value 4 or 1
- Literal constants set with kind value:

```
const = 1.0_wp
```


On IBM, CRAY, the kind values of 4 and 8 for 4 and 8 byte reals.

On PC (Salford FTN90), the kind values of 1 and 2 for 4 and 8 byte reals.

In Fortran 77, if you wish to set a double precision literal constant, you **must** give D0:

```
const = 1.0D0
```

See page 7 of the Student Notes.

Kind values

REAL

- Simplest example is to replace DOUBLE PRECISION:

```
INTEGER, PARAMETER :: idp = KIND(1.0D0)
REAL (KIND = idp) :: ra
```

ra is declared as 'double precision', but this is system dependent.

- To declare real in system independent way, specify kind value associated with precision and exponent range required:

```
INTEGER, PARAMETER :: &
    i10 = SELECTED_REAL_KIND(10, 200)
REAL (KIND = i10) :: a, b, c
```

a, b and c have at least 10 decimal digits of precision and the exponent range 200.

If you request range or precision that is not supported, i10 will be set to negative value, which will cause compile time error.

You will still get only single, double and quad precision - does not give you anything new; just a better way of expressing them.

The main purpose is to get away from single/double precision concept as this is system dependent. 64 bit or 8 byte terminology is system independent, but no way of using this in Fortran 90 - must use precision and range.

Kind values

INTEGER

- Integers usually have 16, 32 or 64 bit
- 16 bit integer normally permits -32768 < i < 32767
- Kind values for each supported type
- To declare integer in system independent way, specify kind value associated with range of integers required:

```
...
INTEGER, PARAMETER :: &
    i8 = SELECTED_INT_KIND(8)
INTEGER (KIND = i8) :: ia, ib, ic
...
```

ia, ib and ic can have values between -10^8 and $+10^8$ **at least (**if permitted by processor**)**.

If the range is too big or invalid, returns -1?

On IBM, CRAY, the kind values of 4 and 8 for 4 and 8 byte integers.

On PC (Salford FTN90), the kind values of 1 and 2 for 4 and 8 byte integers.

See page 9 of the Student Notes.

Kind values

Intrinsics

```
INTEGER, PARAMETER :: &
    i8 = SELECTED_INT_KIND(8)
INTEGER (KIND = i8) :: ia
PRINT *, HUGE(ia), KIND(ia)
```

This will print the largest integer available for this integer type (2147483674), and its kind value.

```
INTEGER, PARAMETER :: &
    i10 = SELECTED_REAL_KIND(10, 200)
REAL (KIND = i10) :: a
PRINT *, RANGE(a), PRECISION(a), KIND(a)
```

This will print the exponent range, the decimal digits of precision and the kind value of a.

As the kind value is system dependent, better to avoid specifying value directly - use intrinsics to request the required range for integers or precision and exponent range for reals.

Note that the result of HUGE (if available) is greater than 10^8 and the kind value is system dependent.

See page 9 of the Student Notes.

Derived types

- Defined by user
- Can include different intrinsic types and other derived types
- Components accessed using percent (%)
- Only assignment operator (=) is defined for derived types
- Can (re)define operators - see later operator overloading

Often also called structures.

Derived types enable programmers to express their problems in terms of data types exactly matched to their requirements.

Derived types have great importance in modules, program structures and pointers, which can be seen later in the course.

See page 10 of the Student Notes.

Derived types

Examples

- Define the form of derived type

```
TYPE vreg
    CHARACTER (LEN = 1) :: year
    INTEGER :: number
    CHARACTER (LEN = 3) :: place
END TYPE vreg
```

- Create the structures of that type

```
TYPE (vreg) :: mycar1, mycar2
```

- Assigned by a derived type constant

```
mycar1 = vreg('L', 240, 'VPX')
```

- Use % to select a component of that type

```
mycar2%year = 'R'
```


We have to put the word TYPE. The compiler knows whether we are defining a new type or are declaring a variable of that type because we put the type name in brackets for the declarations of the variables, but not for the type definition.

Derived types

Examples

- Arrays of derived types:

```
TYPE (vreg), DIMENSION (n) :: mycars
```

- Derived type including derived type:

```
TYPE household
    CHARACTER (LEN = 30) :: name
    CHARACTER (LEN = 50) :: address
    TYPE (vreg) :: car
END TYPE household

TYPE (household) :: myhouse

myhouse%car%year = 'R'
```


Control structures

- Three block constructs
  - IF
  - DO
  - CASE
- All can be nested
- All may have construct names to help readability or to increase flexibility

See page 13 of the Student Notes.

Control structure

IF

- General form

```
[name:]IF (logical expression) THEN
          block
       [ELSE IF (logical expression) THEN [name]
          block]...
       [ELSE [name]
          block]
       END IF [name]
```

- Example

```
selection:  IF (i < 0) THEN
               CALL negative
            ELSE IF (i == 0) THEN selection
               CALL zero
            ELSE selection
               CALL positive
            END IF selection
```


Notice there is one minor extension, which is that the IF construct may be named.

The ELSE and ELSE IF can have names only if IF and END IF do.

See page 13 of the Student Notes.

Control structures

DO loops

- General form

```
[name:]   DO [control clause]
             block
          END DO [name]
```

- *control clause* may be:
  - an iteration control clause

    count = initial, final [,inc]
  - a WHILE control clause

    WHILE (logical expression)
  - or nothing (no control clause at all)

The END DO statement should be used to terminate a DO loop. This makes programs much more readable than using a labelled CONTINUE statement, and, as it applies to one loop only, avoids the possible confusion caused by nested DO loops terminating on the same CONTINUE statement.

See page 13 of the Student Notes.

Control structures

DO loops

- Iteration control clause:

```
rows: DO i = 1, n
cols:      DO j = 1, m
               a(i, j) = i + j
           END DO cols
       END DO rows
```

- WHILE control clause:

```
true:  DO WHILE (i <= 100)
           ...
           body of loop
           ...
       END DO true
```


DO WHILE is ‘redundant’ because the same effect can be achieved using the DO loop with an EXIT statement which is described later.

Control structures

DO loops

- Use of EXIT and CYCLE:
  - exit from loop with EXIT
  - transfer to END DO with CYCLE
  - EXIT and CYCLE apply to inner loop by default but can refer to specific, named loop

```
outer: DO i = 1, n
middle:  DO j = 1, m
inner:       DO k = 1, l
                 IF (a(i,j,k) < 0.0) EXITouter ! leave loops
                 IF (j == 5) CYCLE middle    ! set j = 6
                 IF (i == 5) CYCLE      ! skip rest of inner
                 ...
             END DO inner
         END DO middle
     END DO outer
```


EXIT transfers control to the statement following the END DO, whereas CYCLE transfers control to a notional dummy statement immediately preceding the END DO.

Control structures

DO loops

- No control clause:

```
DO
    READ (*, *) x
    IF (x < 0) EXIT
    y = SQRT(x)
    ...
END DO
```

Notice that this form can have the same effect as a DO WHILE loop.

The EXIT statement allow us to simplify the DO statement even further to the 'do for-ever' loop.

Control structures

CASE construct

- Structured way of selecting different options, dependent on value of single expression
- Replacement for
  - computed GOTO
  - or IF ... THEN ... ELSE IF ... END IF constructs

See page 15 of the Student Notes.

Control structures

CASE construct

- General form:

```
[name:]  SELECT CASE (expression)
            [CASE (selector) [name]
               block]
               ...
         END SELECT [name]
```

- *expression* - character, logical or integer
- *selector* - DEFAULT, or one or more values of same type as expression:
  - single value
  - range of values separated by : (character or integer only), upper or lower value may be absent
  - list of values or ranges

The selectors must not overlap.

If a valid selector is found, the corresponding statements are executed and control then passes to the END SELECT.

If no valid selector is found, execution continues with the first statement after END SELECT.

The CASE DEFAULT clause is optional and covers all other possible values of the expression not included in the other selectors. It need not necessarily come at the end.

Control structures

CASE construct

- Example

```
colour: SELECT CASE (ch)
           CASE ('C', 'D', 'G':'M')
              colour = 'red'
           CASE ('X':)
              colour = 'green'
           CASE DEFAULT
              colour = 'blue'
        END SELECT colour
```


Note a colon may be used to specify a range, as in the example.

Procedures and Modules

Procedures and modules

Topics

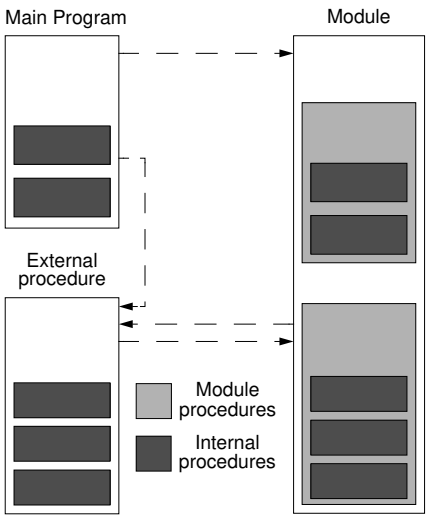
- Program units
- Procedures
- INTERFACE blocks
- Internal procedures
- Procedure arguments
- RESULT clause for functions
- Array-valued functions
- Recursive procedures

Procedures and modules

Topics

- Generic procedures
- Modules
- Overloading operators
- Defining operators
- Assignment overloading
- Scope
- Program structure

Program units



Thinking about program structure is not something Fortran 77 programmers had to do, because Fortran 77 has rather simple program units: main program, external procedure and block data (‘redundant’).

Fortran 90 additionally introduces modules, module procedures and internal procedures.

A module program unit exists to make some or all of the entities declared within it accessible to more than one program unit (global).

A subprogram which is a member of a collection in a module is called a module procedure.

A subprogram which is placed inside a module procedure, an external procedure, or a main program, is called an internal procedure.

See page 19 of the Student Notes.

Main program

- Form:

```
PROGRAM [name]
    [specification statements]
    [executable statements]
    ...
END [PROGRAM [name]]
```

- Example:

```
PROGRAM test
    ...
    ...
!   END
!   END PROGRAM
    END PROGRAM test
```


Note the new form of END statement.

The same syntax applies to other program units.

See page 20 of the Student Notes.

Procedures

Functions and subroutines

- Structurally, procedures may be:
  - External - self contained (not necessarily Fortran)
  - Internal - inside a program unit
  - Module - member of a module
- Fortran 77 has only external procedures

Procedures may be subroutines or functions. Self-contained sub-tasks should be written as procedures.

See page 20 of the Student Notes.

External procedures

Form

```
SUBROUTINE name ( dummy-argument-list )
    [specification-statements]
    [executable-statements]
    ...
END [SUBROUTINE [name]]
```

or

```
FUNCTION name ( dummy-argument-list )
    [specification-statements]
    [executable-statements]
    ...
END [FUNCTION [name]]
```


See page 20 of the Student Notes.

Internal procedures

- Each program unit can contain internal procedures
- Internal procedures are collected together at the end of a program unit and preceded by a CONTAINS statement
- Same form as external procedures except that the word SUBROUTINE/FUNCTION must be present on the END statement
- Variables defined in the program unit remain defined in the internal procedures, unless redefined there
- Nesting of internal procedures is not permitted

See page 21 of the Student Notes.

Internal procedures

Example

```
PROGRAM main

    IMPLICIT NONE
    REAL :: a, b, c
    REAL :: mainsum

    ...

    mainsum = add( )
    ...

CONTAINS

    FUNCTION add ( )
        REAL :: add      ! a,b,c defined in 'main'
        add = a + b + c
    END FUNCTION add
    ...

END PROGRAM main
```


Note brackets are still required for the function even if the parameter list is empty.

If the function used x, y and z instead of a, b, and c, would need to declare them and pass arguments.

If arguments passed, a, b, and c are dummy so must be declared.

INTERFACE blocks

Explicit vs implicit

- If an 'explicit' interface for a procedure is provided, compiler can check argument inconsistency
- Module and internal procedures have an 'explicit' interface by default
- External procedures have an 'implicit' interface by default
- INTERFACE block can be used to specify an 'explicit' interface for external procedures
- Always use INTERFACE block in the calling program unit for external procedures

In order to generate calls to subprograms correctly, the compiler needs to know certain things about the subprogram, including name, number and type of arguments. The explicit interface provides such information.

The explicit interface idea is similar as the function prototype in C programming language.

See page 22 of the Student Notes.

INTERFACE blocks

Form

- General form:

```
INTERFACE
    interface_body
    ...
END INTERFACE
```

where *interface_body* is an exact copy of the subprogram specification, its dummy argument specifications and its END statement

- Example

```
INTERFACE
    REAL FUNCTION func(x)
        REAL, INTENT(IN) :: x
    END FUNCTION func
END INTERFACE
```


Can not name an interface block except for the generic name (Generic procedures - see later).

The interface block must be placed in the calling program unit.

Use interface block instead of EXTERNAL statement.

Procedure arguments

INTENT

- Argument intent - can specify whether an argument is for:
  - input (IN),
  - output (OUT)
  - or both (INOUT)
- Examples

```
INTEGER, INTENT(IN) :: in_only

REAL, INTENT(OUT) :: out_only

INTEGER, INTENT(INOUT) :: both_in_out
```


If the intent is IN, the argument value may not be changed within the subprogram.

If the intent is OUT, the argument may only be used to return information from the procedure to the calling program.

If the intent is INOUT, then the argument may be used to transfer information in both directions between the procedure and calling program.

See page 22 of the Student Notes.

Procedure arguments

Keyword arguments

- Example

```
REAL FUNCTION area (start, finish, tol)
    IMPLICIT NONE
    REAL, INTENT(IN) :: start, finish, tol
    ...
END FUNCTION area
```

- Call with:

```
a = area(0.0, 100.0, 0.01)

b = area(start = 0.0, tol = 0.01, finish = 100.0)

c = area(0.0, finish = 100.0, tol = 0.01)
```

- Once a keyword is used, all the rest must use keywords

We are already familiar with keyword arguments in the input/output statements of Fortran 77:

```
READ(UNIT = 5, FMT = 101, END = 9000) X, Y, Z
```

The advantage of using keywords is that you don't need to remember the order of the parameters, but you do need to know the variable names used in the procedure (order independent).

All arguments can always be redefined with keywords.

See page 23 of the Student Notes.

Procedure arguments

Optional arguments

- Example

```
REAL FUNCTION area (start, finish, tol)
    IMPLICIT NONE
    REAL, INTENT(IN), OPTIONAL ::     &
       start, finish, tol
    ...
END FUNCTION area
```

- Call with:

```
a = area(0.0, 100.0, 0.01)

b = area(start=0.0, finish=100.0, tol=0.01)

c = area(0.0)

d = area(0.0, tol=0.01)
```


In some situations, not all the procedure's arguments need be present each time it is invoked. An argument which need not always be given can be specified as OPTIONAL in the type declaration statement.

If using non-keyword, can only miss out from the right hand side.

Best to use keywords.

See page 23 of the Student Notes.

Procedure arguments

PRESENT

- Example

```
REAL FUNCTION area (start, finish, tol)
    IMPLICIT NONE
    REAL, INTENT(IN), OPTIONAL ::     &
        start, finish, tol
    REAL :: ttol
    ...
    IF ( PRESENT(tol) ) THEN
        ttol = tol
    ELSE
        ttol = 0.01
    END IF
    ...
END FUNCTION area
```

- Use intrinsic function PRESENT to check for presence of argument

Note the use of local variable ttol. The local variable ttol may be redefined, while the argument tol is unchanged (tol can not be changed if it is INTENT(IN)).

See page 24 of the Student Notes.

Procedure arguments

Derived type

Procedure arguments can be of derived type if:

- the procedure is internal to the program unit in which the derived type is defined
- or the derived type is defined in a module which is accessible from the procedure

The derived type must be defined in only one place.

See page 24 of the Student Notes.

Procedure arguments

Procedures as arguments

- In Fortran 77, a procedure argument is declared as EXTERNAL
- In Fortran 90, the procedure that is passed as argument must be an external or a module procedure
- Internal procedures are not permitted
- For external procedure, you are recommended to supply an INTERFACE block in the calling program unit
- A module procedure has an explicit interface by default

For external procedures, not strictly necessary to have interface blocks, as Fortran 77 code which does not use them is valid, but strongly recommended.

See page 24 of the Student Notes.

Procedure arguments

Procedures as arguments

- Example

The calling program unit:

```
...
INTERFACE
   REAL FUNCTION func(x, y)
      REAL, INTENT(IN) :: x, y
   END FUNCTION func
END INTERFACE
...
CALL area(func, start, finish, tol)
```

The external function:

```
REAL FUNCTION func(x, y)
   IMPLICIT NONE
   REAL, INTENT(IN) :: x, y
   ...
END FUNCTION func
```

RESULT clause for functions

- Functions can have a result clause:

```
FUNCTION add (a, b, c) RESULT (sum_abc)
    IMPLICIT NONE
    REAL, INTENT(IN) :: a, b, c
    REAL :: sum_abc
    sum_abc = a + b + c
END FUNCTION add
```

- Result clause required for recursive functions which call themselves directly

The result name that will be used within the function must be specified in brackets after the keyword RESULT at the end of the function statement.

See page 25 of the Student Notes.

Array-valued functions

- In Fortran 90 functions may have an array-valued result:

```
FUNCTION add_vec (a, b, n)

    IMPLICIT NONE
    INTEGER, INTENT(IN) :: n
    REAL, DIMENSION (n), INTENT(IN) :: a, b
    REAL, DIMENSION (n) :: add_vec
    INTEGER :: i

    DO i = 1, n
       add_vec(i) = a(i) + b(i)
    END DO

END FUNCTION add_vec
```


It is a good practice to declare all arguments as INTENT(IN) where appropriate.

The type of an array-valued function ‘can not’ be specified in the initial FUNCTION statement, but in a type declaration in the body of the function, where the dimension of the array must also be specified.

See page 25 of the Student Notes.

Recursive procedures

- Procedures may be called recursively:
  - either A calls B calls A, or
  - A calls A directly (RESULT clause required).
- Must be defined as recursive procedure:

```
RECURSIVE FUNCTION fact(n) RESULT(res)
    IMPLICIT NONE
    INTEGER, INTENT(IN) :: n
    INTEGER :: res
    IF (n == 1) THEN
        res = 1
    ELSE
        res = n * fact(n - 1)
    END IF
END FUNCTION fact
```


Note use of test (n == 1) to stop recursion

See page 26 of the Student Notes..

Generic procedures

- Can define your own generic procedures
- Need distinct procedures for specific type of arguments and a 'generic interface':

```
INTERFACE generic_name
    specific_interface_body
    specific_interface_body
    ...
END INTERFACE
```

- Each distinct procedure can be invoked using the generic name, depending on the type of arguments
- Generic procedures with derived type of arguments require Modules - see later

Analogous to generic intrinsics, such as MAX instead of MAX0, AMAX1, etc.

A generic interface block has a generic name and includes specifications for each external procedure (or a list of module procedures - see later).

See page 27 of the Student Notes.

Generic procedures

Example

- Two distinct external subroutines for REAL and INTEGER types:

```
SUBROUTINE swapreal (a, b)
    IMPLICIT NONE
    REAL, INTENT(INOUT) :: a, b
    REAL :: temp
    temp = a; a = b; b = temp
END SUBROUTINE swapreal

SUBROUTINE swapint (a, b)
    IMPLICIT NONE
    INTEGER, INTENT(INOUT) :: a, b
    INTEGER :: temp
    temp = a; a = b; b = temp
END SUBROUTINE swapint
```


The more distinct subroutines can be defined for other types, e.g., LOGICAL or even user derived types.

Generic procedures

Example

- Generic interface:

```
INTERFACE swap          ! generic name
    SUBROUTINE swapreal (a, b)
        REAL, INTENT(INOUT) :: a, b
    END SUBROUTINE swapreal
    SUBROUTINE swapint (a, b)
        INTEGER, INTENT(INOUT) :: a, b
    END SUBROUTINE swapint
END INTERFACE
```

- Invoke with:

```
CALL swap(x, y)
```


All the procedures specified in a generic interface block must be unambiguously distinct, and as a consequence of this either all must be subroutines or all must be functions.

If x, y are reals, swapreal is actually invoked through the generic name swap.

If x, y are integers, swapint is actually invoked through the generic name swap.

Modules

- Very powerful facility with many applications in terms of program structure
- Method of sharing data and/or procedures to different program units
- Very useful role for definitions of types and associated operators
- Form:

```
MODULE module-name
    [specification-stmts]
    [executable-stmts]
[CONTAINS
    module procedures]
END [MODULE [module-name]]
```

- Accessed via the USE statement

The USE statement is non-executable, and must appear at the very beginning of a program unit before any other non-executable statements, and after the PROGRAM, or other program unit statement.

A program unit may invoke a number of different modules by having a series of USE statements.

Note that a module itself may USE another module, but a module cannot invoke itself either directly or indirectly.

See page 28 of the Student Notes.

Modules

Global data

- Global if used in main program or with SAVE attribute (substitute for COMMON)

```
MODULE globals
    REAL, SAVE :: a, b, c
    INTEGER, SAVE :: i, j, k
END MODULE globals
```

- Examples of the USE statements:

```
USE globals
! allows all variables in the module to be accessed

USE globals, ONLY : a, c
! allows only variables a and c to be accessed

USE globals, r => a , s => b
! allows a, b and c to be accessed with local
! variables r, s and c
```


The use of variables from a module could potentially cause problems if the same names have been used for different variables in different parts of a program. The USE statement with the => symbol can overcome this problem by allowing the specification of a different local name for data accessed from a module.

Note that r => a means “a goes to r”, not as one might intuitively think “r goes to a”.

The ONLY qualifier limits access to certain items within the module.

See page 29 of the Student Notes.

Modules

Module procedures

- Modules may contain procedures that can be accessed by other program units
- Same form as external procedure except:
  - Procedures must follow a CONTAINS statement
  - The END statement must have SUBROUTINE or FUNCTION specified.
- Particularly useful for a collection of derived types and associated operations

Note that, unlike external procedures, module procedures must be supplied in Fortran.

See page 30 of the Student Notes.

Modules

Module procedures

Example to 'add' variables with derived type:

```
MODULE point_module
    TYPE point
        REAL :: x, y
    END TYPE point
CONTAINS
    FUNCTION addpoints (p, q)
        TYPE (point), INTENT(IN) :: p, q
        TYPE (point) :: addpoints
        addpoints%x = p%x + q%x
        addpoints%y = p%y + q%y
    END FUNCTION addpoints
END MODULE point_module
```

A program unit would contain:

```
USE point_module
TYPE (point) :: px, py, pz
...
pz = addpoints(px, py)
```


Module procedures are invoked using the normal CALL statement or function reference, but can only be invoked by a program unit which has invoked, via the USE statement, the module containing the procedures.

A module procedure may call other module procedures in the same module.

The data declared in the module before the CONTAINS statement is directly accessible to all the module procedures.

However, any items declared within a module procedure are local and cannot be accessed outside that procedure.

Modules

Generic procedures

- Using modules allows arguments of derived type and hence generic procedures with derived types

See page 30 of the Student Notes.

Modules

Generic procedures

- Example with derived type:

```
MODULE genswap
   IMPLICIT NONE
   TYPE point
      REAL :: x, y
   END TYPE point
   INTERFACE swap       ! generic interface
      MODULE PROCEDURE swapreal, &
         swapint, swaplog, swappoint
   END INTERFACE
CONTAINS
   SUBROUTINE swappoint (a, b)
      TYPE (point), INTENT(INOUT) :: a, b
      TYPE (point) :: temp
      temp = a; a = b; b = temp
   END SUBROUTINE swappoint
   ...          ! swapint, swapreal, swaplog
                ! procedures are defined here
END MODULE genswap
```


Note use of the interface block with

MODULE PROCEDURES ...

The external procedure version of interface block can be given with specifications.

Modules

Public and private objects

- By default all objects in a module are available to a program unit which includes the USE statement
- Can restrict the use of certain objects to the guest program
- May wish to update module subroutines at any time, keeping the purpose and interface the same, but changing the meaning of specific variables
- Achieved via PRIVATE attribute or PRIVATE statement:

```
INTEGER, PRIVATE :: keep, out
```


See page 31 of the Student Notes.

Overloading operators

- Can extend the meaning of an intrinsic operator to apply to additional data types - operator overloading
- Need an INTERFACE block with the form

```
INTERFACE OPERATOR (intrinsic_operator)
    interface_body
END INTERFACE
```

- Example - define '+' for character variables to concatenate two strings ignoring any trailing blanks

See page 32 of the Student Notes.

Overloading operators

Example

```
MODULE operator_overloading
   IMPLICIT NONE
   ...
   INTERFACE OPERATOR (+)
      MODULE PROCEDURE concat
   END INTERFACE
   ...
CONTAINS
   FUNCTION concat(cha, chb)
      CHARACTER (LEN=*), INTENT(IN) ::&
         cha, chb
      CHARACTER (LEN=LEN_TRIM(cha) +&
         LEN_TRIM(chb)) :: concat
      concat = TRIM(cha) // TRIM(chb)
   END FUNCTION concat
   ...
END MODULE operator_overloading
```

Now the expression ‘cha + chb’ is meaningful

If the function held in an external procedure, would require actual specifications in the interface body:

```
INTERFACE OPERATOR (+)
   FUNCTION concat(cha, chb)
      CHARACTER (LEN=*), INTENT(IN) :: &
         cha, chb
      CHARACTER (LEN=LEN_TRIM(cha) + &
         LEN_TRIM(chb)) :: concat
   END FUNCTION concat
END INTERFACE
```

Note the form of interface block statement:

```
INTERFACE OPERATOR (+)
```

Defining operators

- Can define new operators - especially useful for user defined types
- Operator name must have a ‘.’ at the beginning and end
- Need to define the operation via a function which has one or two non-optional arguments with INTENT(IN)
- Example - find the straight line distance between two derived type ‘points’

```
PROGRAM main
   IMPLICIT NONE
   USE distance_module
   TYPE (point) :: p1, p2
   REAL :: distance
   ...
   distance = p1 .dist. p2
   ...
END PROGRAM main
```


In the preceding example, could have used ‘.plus.’ instead of ‘+’.

See page 32 of the Student Notes.

Defining operators

Example

```
MODULE distance_module
    IMPLICIT NONE
    TYPE point
        REAL :: x, y
    END TYPE point
    ...
    INTERFACE OPERATOR (.dist.)
        MODULE PROCEDURE calcdist
    END INTERFACE
    ...
CONTAINS
    ...
    REAL FUNCTION calcdist (px, py)
        TYPE (point), INTENT(IN) :: px, py
        calcdist = SQRT ((px%x-py%x)**2 &
            + (px%y-py%y)**2 )
    END FUNCTION calcdist
    ...
END MODULE distance_module
```


Note the form of interface statement:

```
INTERFACE OPERATOR (.dist.)
```

Assignment overloading

- When using derived data types, may need to extend the meaning of assignment (=) to new data types:

```
REAL :: ax
TYPE (point) :: px
...
ax = px       ! type point assigned to type real
...           ! not valid until defined
```

- Need to define the assignment via a subroutine with two non-optional arguments, the first having INTENT(OUT) or INTENT(INOUT), the second having INTENT(IN) and create an interface assignment block:

```
INTERFACE ASSIGNMENT (=)
    subroutine interface body
END INTERFACE
```


Might want ax to take the value of the larger of the x and y components of px.

See page 34 of the Student Notes.

Assignment overloading

Example

```
MODULE assignoverload_module
    IMPLICIT NONE
    ...
    TYPE point
        REAL :: x, y
    END TYPE point
    ...
    INTERFACE ASSIGNMENT (=)
        MODULE PROCEDURE assign_point
    END INTERFACE
    ...
CONTAINS
    SUBROUTINE assign_point(ax, px)
        REAL, INTENT(OUT) :: ax
        TYPE (point), INTENT(IN) :: px
        ax = MAX(px%x, px%y)
    END SUBROUTINE assign_point
    ...
END MODULE assignoverload_module
```


Note the form of interface statement:

```
INTERFACE OPERATOR (=)
```

Assignment overloading

Example

The program unit might include:

```
...
USE assignoverload_mod
...
REAL :: ax
TYPE (point) :: px
...
ax = px    ! Type point to type real now defined
...
```

Scope

Scoping unit

The scope of a named entity or label is the set of non-overlapping scoping units where that name or label may be used unambiguously.

A scoping unit is one of the following:

- a derived type definition,
- a procedure interface body, excluding any derived-type definitions and interface bodies contained within it, or
- a program unit or an internal procedure, excluding derived-type definitions, interface bodies, and subprograms contained within it.

Already referred to the aspects of scope in internal procedures and global data etc.

Scope formalises these ideas.

See page 35 of the Student Notes.

Scope

Labels and names

- The scope of a label is a main program or a procedure, excluding any internal procedures contained within it
- Entities declared in different scoping unit are always different
- Within a scoping unit, each named entity must have a distinct name, with the exception of generic names of procedures
- The names of program units are global, so each must distinct from the others and from any of the local entities of the program unit
- The scope of a name declared in a module extends to any program units that USE the module

See page 35 of the Student Notes.

Scope

Example

```
MODULE scope1                    ! scope 1
...                              ! scope 1
CONTAINS                         ! scope 1
   SUBROUTINE scope2             ! scope 2
      TYPE scope3                ! scope 3
         ...                     ! scope 3
      END TYPE                   ! scope 3
      INTERFACE                  ! scope 3
      ...                        ! scope 4
      END INTERFACE              ! scope 3
      REAL x, y                  ! scope 2
100   ...                        ! scope 2
   CONTAINS                      ! scope 2
         FUNCTION scope5(...)    ! scope 5
            REAL y               ! scope 5
            y = x + 1.0          ! scope 5
100         ...                  ! scope 5
         END FUNCTION scope5     ! scope 5
   END SUBROUTINE scope2         ! scope 2
END MODULE scope1                ! scope 1
```


The label 100 has different scope (one is in the scope 3 and the other in the scope 5).

The variable y declared in different scoping unit is different, even if it has the same name and properties.

The scope of the variable x extends from the scope 3 to the scope 5 (host association).

See page 36 of the Student Notes.

Program structure

Order of statements

<table>
<tr><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE,
or MODULE Statement</td></tr>
<tr><td colspan="3">USE Statements</td></tr>
<tr><td rowspan="4">FORMAT
Statements</td><td colspan="2">IMPLICIT NONE Statement</td></tr>
<tr><td>PARAMETER
Statements</td><td>IMPLICIT Statements</td></tr>
<tr><td>PARAMETER
and DATA
Statements</td><td>Derived-type Definitions,
Interface Blocks,
Type Declaration Statements,
and Specification Statements</td></tr>
<tr><td colspan="2">Executable Statements</td></tr>
<tr><td colspan="3">CONTAINS Statement</td></tr>
<tr><td colspan="3">Internal Subprograms
or Module Subprograms</td></tr>
<tr><td colspan="3">END Statement</td></tr>
</table>

Taken from Metcalf & Reid's book.

Note that IMPLICIT NONE may be preceded in a program unit only by USE and FORMAT statements.

See page 36 of the Student Notes.

Program structure

When to use INTERFACE blocks

When a module/external procedure is called:

- Which defines or overloads an operator, or the assignment
- Using a generic name

Additionally, when an external procedure:

- Is called with keyword/optional argument
- Is an array-valued/pointer function or a character function which is neither a constant nor assumed length
- Has a dummy argument which is an assumed-shape array, a pointer/target
- Is a dummy or actual argument (not mandatory but recommended)

Need to provide these interface blocks more frequently for Fortran 77 style coding with external procedures.

See page 37 of the Student Notes.

Program structure

Summary

Fortran 77 style:

- Main program with external procedures, possibly in a library.
- No explicit interfaces, so argument inconsistencies are not checked by compiler.

Simple Fortran 90:

- Main program with internal procedures.
- Interfaces are ‘explicit’, so argument inconsistencies are trapped by compiler.

See page 37 of the Student Notes.

Program structure

Summary

Fortran 90 with modules:

- Main program and module(s) containing interfaces and possibly specifications, and external procedures (possibly precompiled libraries).

  A Fortran 90 version of the Fortran 77 style - with interfaces to permit compiler checking of argument inconsistencies.

- Main program and module(s) containing specifications, interfaces and procedures.

  No external procedures.

  Expected for sophisticated Fortran 90 programs.

Main difference between these two is the use of external procedures in the former, in which there is no way of checking consistencies within external procedures.

For new and revised programs, always think in Fortran 90 way in terms of program structure.

Array Processing

Array processing

Topics

- Terminology
- Specifications
- Whole array operations
- WHERE statement and construct
- Elemental intrinsic procedures
- Array sections
- Element location versus subscript

Array processing

Topics

- Zero-sized arrays
- Array assignment
- Array constructors
- Allocatable arrays
- Automatic arrays
- Assumed shape arrays
- Array intrinsic procedures

Terminology

- Rank

  Number of dimensions
- Extent

  Number of elements in a dimension
- Shape

  Vector of extents
- Size

  Product of extents
- Conformance

  Same shape

Rank: Scalar has rank o, Vector has rank 1 and Matrix has rank 2.

Size: The total number of elements which make up the array, and may be zero.

Conformance: Two objects are conformable if they have the same shape or if one is a scalar as the scalar is broadcast to an array with the same shape.

See page 41 of the Student Notes.

Terminology

Example

```
...
REAL, DIMENSION :: a(-3:4, 7)
REAL, DIMENSION :: b(8, 2:8)
INTEGER :: c
REAL, DIMENSION :: d(8, 1:8)
...
```

- a has
  - rank 2
  - extents 8 and 7
  - shape (/ 8, 7 /)
  - size 56
- a is conformable with b & c, but not with d

Notice the use of array constructors to create the shape vectors - this will be explained later in “Array Constructors”.

Specifications

type [[,DIMENSION (*extent-list*)] [,*attribute*]... ::] *entity-list*

where:

- *type* - INTRINSIC or derived type
- DIMENSION - Optional, but required to define default dimensions
- (*extent-list*) - Gives array dimension:
  - Integer constant
  - integer expression using dummy arguments or constants.
  - : if array is allocatable or assumed shape.
- *attribute* - as given earlier
- *entity-list* - list of array names optionally with dimensions and initial values.

This is simply a special case of the form of declaration given in Slide 23: Specifications.

See page 42 of the Student Notes.

Specifications

Example

- Two dimensions:

```
REAL, DIMENSION(-3:4, 7) :: ra, rb
```

- Initialisation:

```
INTEGER, DIMENSION (3) :: ia = (/ 1, 2, 3 /), &
    ib = (/ (i, i = 1, 3) /)
```

- Automatic arrays:

```
LOGICAL, DIMENSION (SIZE(loga)) :: logb
```

where loga is a dummy array argument

- These are all explicit shape arrays

For detail, see later Slide 112: Array constructor and Slide 118: Automatic arrays.

Specifications

Example

- Allocatable arrays (deferred shape):

  REAL, DIMENSION (:, :), ALLOCATABLE :: a, b

  Dimensions are defined in subsequent ALLOCATE statement.

- Assumed shape arrays:

  REAL, DIMENSION (:, :, :) :: a, b

  Dimensions taken from actual arguments in calling routine.

For detail, see later Slide 116: Dynamic arrays and Slide 122: Assumed shape arrays.

Whole array operations

- Arrays for whole array operation must be conformable
- Evaluate element by element, i.e., expressions evaluated before assignment
- Scalars broadcast
- Functions may be array valued

In Fortran 77 it was not possible to work with whole arrays, instead each element of an array had to be operated on separately, often requiring the use of nested DO-loops. When dealing with large arrays, such operations could be very time consuming and furthermore the required code was very difficult to read and interpret.

An important new feature in Fortran 90 is the ability to perform whole array operations, enabling an array to be treated as a single object and removing the need for complicated, unreadable DO-loops. Furthermore, it provides data parallel capability for better use of vector and parallel processors.

See page 43 of the Student Notes.

Whole array operations

Example

- Fortran 77:

```
      REAL a(20), b(20), c(20)
      ...
      DO 10 i = 1, 20
         a(i) = 0.0
10    CONTINUE
      ...
      DO 20 i = 1, 20
         a(i) = a(i) / 3.1 + b(i) * SQRT(c(i))
      20 CONTINUE
```

- Fortran 90:

```
      REAL, DIMENSION (20) :: a, b, c
      ...
      a = 0.0
      ...
      a = a / 3.1 + b * SQRT(c)
```


Whole array operations

Example

- Fortran 77:

```
      REAL a(5, 5), b(5, 5), c(5, 5)
      ...
      DO 20 i = 1, 5
         DO 10 j = 1, 5
            c(j, i) = a(j, i) * b(j, i)
10       CONTINUE
20    CONTINUE
```

- Fortran 90:

```
      REAL, DIMENSION (5, 5) :: a, b, c
      ...
      c = a * b
```


Note that this is elemental multiplication, not matrix multiplication.

Whole array operations

Example

Find the maximum value less than 1000 in a three dimensional array:

- Fortran 77 requires triple DO loop and IF statements
- Fortran 90:

```
REAL, DIMENSION (10, 10, 10) :: a
amax = MAXVAL(a, MASK = (a < 1000))
```

Find average value > 3000 in an array:

- Fortran 77 requires DO loops and IF statements
- Fortran 90:

```
av = SUM(a, MASK = (a > 3000)) / &
    COUNT(MASK=(a> 3000))
```


Note that the power of whole array operations are increased in conjunction with array intrinsic functions.

Note the use of the optional MASK argument. MASK is a logical array expression. Only those elements of a that correspond to elements of MASK that have the value true take part in the function call. So in the first example amax is the value of the maximum element in a which is less than 1000.

Elemental intrinsic procedures

- Elemental procedures specified for scalar arguments
  - May also be applied to conforming array arguments
  - As if applied to each element separately.
- Example

```
! To find square root of all elements of array, a
a = SQRT(a)

! To find the string length excluding trailing blanks
! for all elements of a character array, ch
length = LEN_TRIM(ch)
```


See page 45 of the Student Notes.

WHERE statement

- Form:

  WHERE (*logical-array-expr*) *array-assignment*

- Assignment if logical condition is true

```
REAL DIMENSION (5, 5) :: ra, rb
...
WHERE (rb > 0.0) ra = ra / rb
```

- Note mask (rb > 0.0) must conform with LHS ra

This is useful to perform an array operation on only certain elements of an array.

See page 45 of the Student Notes.

WHERE construct

- Form:

```
WHERE (logical-array-expr)
    array-assignments
END WHERE

WHERE (logical-array-expr)
    array-assignments
ELSEWHERE
    array-assignments
END WHERE
```

- Example

```
REAL DIMENSION (5, 5) :: ra, rb
...
WHERE (rb > 0.0)
    ra = ra / rb
ELSEWHERE
    ra = 0.0
END WHERE
```


Note that WHERE construct (called block WHERE also) is inconsistent in style from IF construct.

ELSEWHERE in WHERE construct is equivalent to ELSE in IF construct.

No WHERE construct is equivalent to ELSE IF.

See page 45 of the Student Notes.

Array sections

A subarray, called a section, of an array may be referenced by specifying a range of subscripts, either:

- A simple subscript

  a (2, 3, 1) ! single array element

- A subscript triplet
  - [lower bound]:[upper bound] [:stride]
  - defaults to declared bounds and stride 1
- A vector subscript

An array section can be used in the same way as an array, but it is not possible to reference the individual elements belonging to the section directly.

Stride is the increment by which array elements are accessed.

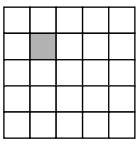
See page 46 of the Student Notes.

Array sections

Example

REAL, DIMENSION (5, 5) :: ra

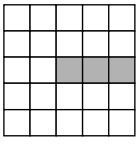
ra(2, 2) (=ra(2:2:1, 2:2:1))



Array Element

Shape (/ 1 /)

ra(3, 3:5)



Sub-row

Shape (/ 3 /)

ra(2, 2) is simple subscript.

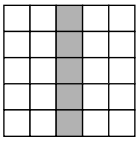
ra(2:2, 2:2, 1) is subscript triplet.

Array sections

Example

REAL, DIMENSION (5, 5) :: ra

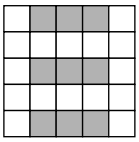
ra(:, 3)



Whole column

Shape (/ 5 /)

ra(1::2, 2:4)



Stride 2

Shape (/ 3, 3 /)

Vector subscripts

- 1D integer array used as array subscript

  (/ 3, 2, 12, 2, 1 /)

- Example

```
...
REAL, DIMENSION :: ra(6), rb(3)

INTEGER, DIMENSION (3) :: iv

iv = (/ 1, 3, 5 /)          ! rank 1 integer expression

ra = (/ 1.2, 3.4, 3.0, 11.2, 1.0, 3.7 /)

rb = ra(iv)                 ! iv is the vector subscript
    ! = (/ ra(1), ra(3), ra(5) /)
    ! = (/ 1.2, 3.0, 1.0 /)
...
```


Each element of this expression must be defined with values that lie within the parent array subscript bounds. The elements of a vector subscript may be in any order.

See page 47 of the Student Notes.

Vector subscripts

- Vector subscript can be on LHS of expression

```
iv = (/ 1, 3, 5 /)
ra(iv) = (/ 1.2, 3.4, 5.6 /)
    ! = ra( (/ 1, 3, 5 /) ) = (/ 1.2, 3.4, 5.6 /)
```

- Must not repeat values of elements on LHS (many to one)

```
iv = (/ 1, 3, 1 /)
ra(iv) = (/ 1.2, 3.4, 5.6 /)      ! not permitted
    ! = ra((/ 1, 3, 1 /)) = (/ 1.2, 3.4, 5.6 /)

iv = (/ 1, 3, 5 /)
ra(iv) = (/ 1.2, 3.4, 5.6 /)      ! permitted
```


Array assignment

- Operands must be conformable
- Example

```
REAL, DIMENSION (5, 5) :: ra, rb, rc
INTEGER :: id

...

ra = rb + rc * id            ! Shape(/ 5, 5 /)

ra(3:5, 3:4) = rb(1::2, 3:5:2) + rc(1:3, 1:2)
! Shape(/ 3, 2 /)

ra(:, 1) = rb(:, 1) + rb(:, 2) + rb(:, 3)
! Shape(/ 5 /)
```


Both whole arrays and array sections can be used as operands in array assignments provided that all the operands are conformable.

See page 48 of the Student Notes.

Beware of recursion

The following code:

```
DO i=2,n
    x(i) = x(i) + x(i-1)                (1)
END DO
```

is not the same as:

```
x(2:n) = x(2:n) + x(1:n-1)              (2)
```

- In (1):

  x(i) = x(i) + x(i-1) + x(i-2) + ... + x(1)

- Fortran 90 equivalent:

  x(2:n) = (/ (SUM(x(1:i)), i = 2, n) /)

Recursion is a problem for vectorisation and parallelisation. Perhaps this coding “problem” is good - it makes the programmer aware of recursion.

See page 48 of the Student Notes.

Element location versus subscript

MAXLOC/MINLOC returns the max/min element location instead of subscript:

```
REAL, DIMENSION (1:8) :: ra
REAL, DIMENSION (-3:4) :: rb
INTEGER, DIMENSION (1) :: locmax1, locmax2
REAL :: max1, max2
ra = (/ 1.2, 3.4, 5.4, 11.2, 1.0, 3.7, 1.0, 1.0/)
rb = ra

! To find the location of max value
locmax1 = MAXLOC(ra)            ! = (/ 4 /)
locmax2 = MAXLOC(rb)            ! = (/ 4 /)

! To find maximum value from the location
max1 = ra(locmax1(1))
! OK, because ra defined with 1 as lower bound

max2 = rb(LBOUND(rb) + locmax2(1) - 1)
! general form required when lower bound ≠ 1
```


See page 48 of the Student Notes.

Zero-sized arrays

- When lower bound exceeds upper bound, array has size zero
- Valid providing normal rules are followed
- In particular, must be conformable (if one is zero-sized, all must be)
- Useful at boundaries - code does not need to treat boundaries in a special way:

```
INTEGER, PARAMETER :: n = 10
REAL, DIMENSION (n, n) :: a
REAL, DIMENSION (n) :: b, x
...
DO i = 1, n
    x(i) = b(i) / a(i, i)
    b(i+1:n) = b(i+1:n) - a(i+1:n, i) * x(i)
    ! zero-sized when i = n
END DO
```


See page 49 of the Student Notes.

Array constructor

- Construction of rank 1 array:

```
REAL, DIMENSION (6) :: a, b(6,6)
a = (/ array-constructor-value-list /)
```

where *array-constructor-value-list* can be:

- (/ 1.2, 3.4, 3.0, 11.2, 1.0, 3.7 /)
- (/ b(i,2:4), b(1:5:2, i+3) /)
  ! = (/ b(i,2),b(i,3),b(i,4),b(1,i+3),b(3,i+3), b(5,i+3) /)
- (/ (1.0 / REAL(i), i = 1, 6) /)
  !=(/1.0/1.0,1.0/2.0,1.0/3.0,1.0/4.0,1.0/5.0,1.0/6.0/)
- (/ ((i + j, i = 1, 3), j = 1, 2) /)
  ! = (/ 2, 3, 4, 3, 4, 5 /)

An array constructor creates a rank-one array containing specified values. The values can be specified by listing them or by using an implied DO-loop, or a combination of both.

See page 49 of the Student Notes.

Array constructor

- Transfer to rank > 1 array via RESHAPE function:

  RESHAPE(SOURCE, SHAPE [,PAD] [,ORDER])

- Note that the size of SHAPE must be a constant
- Example:

```
REAL, DIMENSION (3, 2) :: ra
ra = RESHAPE( (/ ((i + j, i = 1, 3), j = 1, 2) /), &
    SHAPE = (/ 3, 2 /) )
```

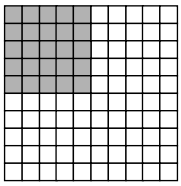
```
2   3
3   4       Shape(/ 3, 2 /)
4   5
```


The argument SOURCE can be an array of any sort (in this case a rank-one array), and the elements of source are rearranged to form an array RESHAPE of shape SHAPE. If SOURCE has more elements than RESHAPE, then the unwanted elements will be ignored. If RESHAPE has more elements than SOURCE, then the argument PAD must be present. The argument PAD must be an array of the same type as SOURCE, and the elements of PAD are used in array element order, using the array repeatedly if necessary, to fill the missing elements of RESHAPE. Finally, the optional argument ORDER allows the elements of RESHAPE to be placed in an alternative order to array element order. The array ORDER must be the same size and shape as SHAPE, and contains the dimensions of RESHAPE in the order that they should be run through.

Array arguments 1

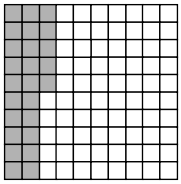
Fortran 77 (wrong)

```
PROGRAM array
PARAMETER (n = 10)
REAL a(n, n)
...
READ (*, *) n1
CALL sub(a, n1, res)
...
END
```



main program (n1 = 5)
25 elements wanted

```
SUBROUTINE sub(a, n1, res)
REAL a(n1, n1)
REAL work(10, 10)
...
res = a(...)
...
END
```



sub
25 elements passed

Consider this example in Fortran 77.

The intension is to define an array to have size big enough for all cases of interest and then to pass a subset of the array to a subprogram.

The shading elements in the main program is the subset wanted to pass.

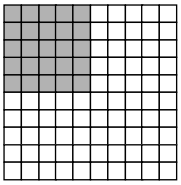
The shaded elements in the subprogram is the subset actually passed, which is wrong in this case because of the ordering way of array elements.

Note that the column ordering of array elements is assumed but may not be same as the memory storage ordering of array elements.

Array arguments 2

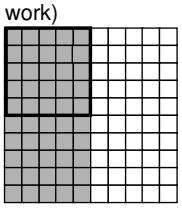
Fortran 77 (right)

```
PROGRAM array
PARAMETER (n = 10)
REAL a(n, n)
REAL work(n, n)
...
READ (*, *) n1
CALL sub(a, n, n1, res, work)
...
END
```



main program (n1 = 5)
25 elements wanted

```
SUBROUTINE sub(a, n, n1, res, work)
REAL a(n, n1)
REAL work(n1, n1)
...
res = a(...)
...
END
```



sub
50 elements passed

Answer in Fortran 77 is to define the array in the subprogram to have the same leading dimension as in the main program. So the array in the subprogram is bigger than wanted.

Work array is required only as local array, but for flexibility standard procedure is to pass it as an argument also.

Dynamic arrays

- Fortran 77
  - static (fixed) memory allocation
- Fortran 90
  - allocate and deallocate storage as required via **allocatable arrays**
  - allow local arrays in a procedure to have different size and shape every time the procedure is invoked via **automatic arrays**
  - reduce overall storage requirement
  - simplify subroutine arguments

Now let us look at the features of Fortran 90 which simplify the example just described.

Allocatable arrays

- A deferred shape array which is declared with the ALLOCATABLE attribute
  - ALLOCATE(allocate_object_list [, STAT= status])
  - DEALLOCATE(allocate_obj_list [, STAT= status])
  - When STAT= is present, status = 0 (success) or status > 0 (error). When STAT= is not present and an error occurs, the program excution aborts
- Example:

```
...
REAL, DIMENSION (:, :), ALLOCATABLE :: ra
INTEGER :: status
READ (*, *) nsize1, nsize2
ALLOCATE (ra(nsize1, nsize2), STAT = status)
IF (status > 0) STOP 'Fail to allocate meomry'
...
IF (ALLOCATED(ra)) DEALLOCATE (ra)
...
```


Note the form of declaration statement for ra (ALLOCATABLE attribute), which does not allocate any memory for the array and whose shape is not determined until space is created. Space is dynamically allocated later in the program, when the array is required, using the ALLOCATE statement.

Both ALLOCATE and DEALLOCATE statements can allocate/deallocate several arrays in one single statement.

It is possible to check whether or not an array is currently allocated using the intrinsic function ALLOCATED.

The DEALLOCATE statement can only be used on arrays which are currently allocated, and similarly, the ALLOCATE statement can only be used on arrays which are not currently allocated.

See page 50 of the Student Notes.

Automatic arrays

- An explicit shape array in a procedure, whose bounds are provided when the procedure is invoked via:
  - dummy arguments
  - variables defined by use or host associations
- Automatic array must not appear in SAVE or NAMELIST statement, nor be initialised in type declaration
- Be aware that Fortran 90 provides no mechanism for checking whether there is sufficient memory for automatic arrays. If there is not, the outcome is unpredictable - the program will probably abort

Automatic arrays are not dummy arguments, but their bounds may depend on dummy arguments, global variables defined in modules, or in case of internal procedures variables defined in its host program unit.

Automatic arrays are automatically created (allocated) upon entry to the procedure in which they are declared, and automatically deallocated upon exit from the procedure.

Thus the size of the automatic array can be different every time the procedure is invoked.

See page 52 of the Student Notes.

Automatic arrays

Example

- Example 1: Bounds of automatic arrays depend on dummy arguments

```
SUBROUTINE sub(n, a)
    IMPLICIT NONE
    INTEGER :: n
    REAL, DIMENSION(n, n) :: a
    REAL, DIMENSION (n, n) :: work1
    REAL, DIMENSION (SIZE(a, 1)) :: work2
    ...
END SUBROUTINE sub
```

- Example 2: Bounds of an automatic array is defined by the global variable in a module

The example 1 shows the bounds of automatic arrays depend on dummy arguments and the usage of SIZE.

Automatic arrays

Example

```
MODULE auto_mod
    IMPLICIT NONE
    INTEGER :: n
CONTAINS
    SUBROUTINE sub
        REAL, DIMENSION(n) :: w
        WRITE (*, *) 'Bounds and size of a: ', &
            LBOUND(w), UBOUND(w), SIZE(w)
    END SUBROUTINE sub
END MODULE auto_mod

PROGRAM auto_arrays
! automatic arrays using modules instead of
! procedure dummy arguments
    USE auto_mod
    IMPLICIT NONE
    n = 10
    CALL sub
END PROGRAM auto_arrays
```


The example 2 shows the use association and host association - where the bounds of an automatic array is defined by the global variable in a module.

Note that 'use association' is where variables declared in the main body of a module are made available to a program unit by a USE statement, and 'host association' is where variables declared in a program unit are made available to its contained internal procedures.

Array arguments 3

Fortran 90 dynamic arrays

```
PROGRAM array
   IMPLICIT NONE
   REAL, ALLOCATABLE, DIMENSION (:, :) :: a
   ...
   READ (*, *) n1
   ALLOCATE (a(n1, n1), STAT = alloc_stat)
   IF (alloc_stat /= 0) THEN
      ! ... error processing code ...
   END IF
   CALL sub(a, n1, res)
   DEALLOCATE(a)
   ...
CONTAINS
   SUBROUTINE sub(a, n1, res)
      REAL, DIMENSION (n1, n1) :: a
      REAL, DIMENSION(n1, n1) :: work
      ...
      res = a(...)
      ...
   END SUBROUTINE sub
END PROGRAM array
```


Same example as above.

Now the array is exactly the size we require in the main program, so we pass exactly what we want. The work array is an automatic array whose bounds depend on the dummy argument n1.

Note that there are three restrictions on the use of allocatable arrays:

1) Allocatable arrays can not be dummy arguments of a procedure and must, therefore, be allocated and deallocated in the same program unit.

2) The result of a function cannot be an allocatable array.

3) Allocatable arrays cannot be used in a derived type definition.

Assumed shape arrays

- Shape of actual and dummy arguments must agree
- Fortran 77: pass array dimensions as arguments
- Fortran 90: not necessary to pass array dimensions
  - Assumed shape array uses dimension of actual arguments
  - Can specify lower bound
  - Explicit interface required, so must provide an INTERFACE block if using an external procedure

In Fortran 77, the upper bound of the last dimension may be *, so for 1-dimension array with lower bound 1, do not need to pass the size (unless needed by subprogram).

See page 54 of the Student Notes.

Assumed shape arrays

Example

```
...                              ! calling program unit
INTERFACE
   SUBROUTINE sub (ra, rb, rc)
      REAL, DIMENSION (:, :) :: ra, rb
      REAL, DIMENSION (0:, 2:) :: rc
   END SUBROUTINE sub
END INTERFACE
REAL, DIMENSION (0:9,10) :: ra ! Shape (/ 10, 10 /)
...
CALL sub(ra, ra(0:4, 2:6), ra(0:4, 2:6))
...

SUBROUTINE sub(ra, rb, rc)       ! External
   REAL, DIMENSION (:, :) :: ra  ! Shape (/10, 10/)
   REAL, DIMENSION (:, :) :: rb  ! Shape (/ 5, 5 /)
   ! = REAL, DIMENSION (1:5, 1:5) :: rb
   REAL, DIMENSION (0:, 2:) :: rc ! Shape (/ 5, 5 /)
   ! = REAL, DIMENSION (0:4, 2:6) :: rc
   ...
END SUBROUTINE sub
```


Note that the lower bound of an assumed shape array is 1 in each dimension by default (e.g., rb) unless specified (e.g., rc).

Interface blocks required only if external procedures.

Array arguments 4

Fortran 90: allocatable, automatic, and assumed shape

```
PROGRAM array
   IMPLICIT NONE
   REAL, ALLOCATABLE, DIMENSION(:, :) :: a
   ...
   READ (*, *) n1
   ALLOCATE ( a(n1, n1) )          ! allocatable
   CALL sub(a, res)
   DEALLOCATE(a)
   ...
CONTAINS
   SUBROUTINE sub(a, res)
   REAL, DIMENSION (:, :) :: a     ! assumed shape
   REAL, DIMENSION (SIZE(a, 1),SIZE(a, 2)) :: work
                                   ! automatic
   ...
   res = a(...)
   ...
   END SUBROUTINE sub
END PROGRAM array
```


Use the same example as before.

Automatic arrays are useful for work arrays when assumed shape arrays are used:

Only two arguments now, rather than 4 in Fortran 77.

Pass exactly the data required.

Much better than Fortran 77.

Array intrinsic functions

List

- Reduction:

```
ALL(MASK[,DIM])
ANY(MASK[,DIM])
COUNT(MASK[,DIM])
MAXVAL(ARRAY[,DIM][,MASK])
MINVAL(ARRAY[,DIM][,MASK])
PRODUCT(ARRAY[,DIM][,MASK])
SUM(ARRAY[,DIM][,MASK])
```

- Inquiry:

```
ALLOCATED(ARRAY)
LBOUND(ARRAY[,DIM])
SHAPE(SOURCE)
SIZE(ARRAY[,DIM])
UBOUND(ARRAY[,DIM])
```


The value & power of array syntax would be greatly reduced without array intrinsic functions. See page 56 of the Student Notes.

ALL - True if all elements true

ANY - True if any element true

COUNT - Number of true elements

MAXVAL - Maximum element value

MINVAL - Minimum element value

PRODUCT - Product of array elements

SUM - Sum of array elements

ALLOCATED - True if array allocated

LBOUND - Lower bounds of array

SHAPE - Shape of array (or scalar)

SIZE - Size of array

UBOUND - Upper bounds of array

Array intrinsic functions

List

- Construction:

```
MERGE(TSOURCE,FSOURCE,MASK)
PACK(ARRAY,MASK[,VECTOR])
UNPACK(VECTOR,MASK,FIELD)
SPREAD(SOURCE,DIM,NCOPIES)
```

- Reshape:

```
RESHAPE(SOURCE,SHAPE[,PAD][,ORDER]
```

- Array Location:

```
MAXLOC(ARRAY[,MASK])
MINLOC(ARRAY[,MASK])
```


MERGE - Merge arrays subject to mask

PACK - Pack elements into vector subject to mask

UNPACK - Unpack elements of vector subject to mask

PACK/UNPACK ≡ GATHER/SCATTER on vector machines. Sparse matrix - pack & unpack can save space and enhance performance.

SPREAD - Construct an array by duplicating an array section

RESHAPE - Reshape array

MAXLOC - Location of maximum element

MINLOC - Location of minimum element

Array intrinsic functions

List

- Array manipulation:

  CSHIFT(ARRAY,SHIFT[,DIM])
  EOSHIFT(ARRAY,SHIFT[,BOUND-ARY][,DIM])
  TRANSPOSE(MATRIX)

- Vector and matrix arithmetic:

  DOT_PRODUCT(VECTOR_A,VECTOR_B)
  MATMUL(MATRIX_A, MATRIX_B)

CSHIFT - Perform circular shift

EOSHIFT - Perform end-off shift

TRANSPOSE - Transpose matrix

DOT_PRODUCT - Compute dot product

MATMUL - Matrix multiplication, also valid for logical

Array intrinsic functions

Example

Three students take four exams. The results are stored in an INTEGER array:

score(1:3,1:4) =	85	76	90	60
	71	45	50	80
	66	45	21	55

- Largest score:

  MAXVAL (score) ! = 90

- Largest score for each student:

  MAXVAL (score, DIM = 2)
  ! = (/ 90, 80, 66 /)

Finding the largest score for each student: for each of the 1st dimension (row ≡ students), finding maxval over the 2nd dimension (column ≡ exams).

See page 57 of the Student Notes.

Array intrinsic functions

Example

- Student with largest score:

```
MAXLOC (MAXVAL (score, DIM = 2))
! = MAXLOC((/ 90, 80, 66 /)) = (/ 1 /)
```

- Average score:

```
average = SUM (score) / SIZE (score)     ! = 62
! average is an INTEGER variable
```

- Number of scores above average:

```
above = score > average
! above(3, 4) is a LOGICAL array
!                    T    T    T    F
! above =            T    F    F    T
!                    T    F    F    F
n_gt_average = COUNT (above)     ! = 6
! n_gt_average is an INTEGER variable
```


For number of scores above average, could say:

n_gt_average = COUNT (score > average)

Array intrinsic functions

Example

- Pack all scores above the average:

```
...
INTEGER, ALLOCATABLE, DIMENSION (:) :: &
    score_gt_average
...
ALLOCATE (score_gt_average(n_gt_average))
scores_gt_average = PACK (score, above)
! = (/ 85, 71, 66, 76, 90, 80 /)
```

- Did any student always score above the average?

```
ANY (ALL (above, DIM = 2))     ! = .FALSE.
```

- Did all students score above the average on any of the tests?

```
ANY (ALL (above, DIM = 1))     ! = .TRUE.
```

Array example

Conjugate gradient algorithm

```
PROGRAM conjugate_gradients

    IMPLICIT NONE
    INTEGER :: iters, its, n
    LOGICAL :: converged
    REAL :: tol, up, alpha, beta
    REAL, ALLOCATABLE :: a(:,:), b(:), x(:), r(:), &
        u(:), p(:), xnew(:)

    READ (*,*) n, tol, its
    ALLOCATE ( a(n,n), b(n), x(n), r(n), u(n), &
        p(n), xnew(n) )

    OPEN (10, FILE='data')
    READ (10,*) a
    READ (10,*) b

    x = 1.0
    r = b - MATMUL(a,x)
    p = r
```


This example is taken from 'Programming in Fortran 90' Wiby ISBN 0471-94185-9, by Prof. I M Smith from Department of Engineering, University of Manchester.

See page 58 of the Student Notes.

Array example (contd)

```
    iters = 0

    DO
        iters = iters + 1
        u = MATMUL(a, p)
        up = DOT_PRODUCT(r, r)
        alpha = up / DOT_PRODUCT(p, u)
        xnew = x + p * alpha
        r = r - u * alpha
        beta = DOT_PRODUCT(r, r) / up
        p = r + p * beta
        converged = ( MAXVAL(ABS(xnew-x)) / &
            MAXVAL(ABS(x)) < tol )
        x = xnew
        IF (converged .OR. iters == its) EXIT
    END DO

    WRITE (*,*) iters
    WRITE (*,*) x

END PROGRAM conjugate_gradients
```

Pointers

Pointers

Topics

- What is a pointer
- Specifications
- Pointer assignment
- Pointer status
- Dynamic storage
- Pointer arguments
- Pointer functions
- Arrays of pointer
- Linked list

What is a pointer?

A pointer variable has the POINTER attribute and may point to:

- Another data object of the same type, which has the TARGET attribute, or
- An area of dynamically allocated memory

The use of pointers provides:

- A more flexible alternative to allocatable arrays
- The tool to create and manipulate linked lists and other dynamic data structures

The use of pointers provides a more flexible alternative to allocatable arrays and the tool to create and manipulate linked lists and other dynamic data structures.

The latter one opens the door to powerful recursive algorithms as well as the means to tailor the storage requirements exactly to the needs of the problem and the data.

See page 63 of the Student Notes.

Specifications

***type* [[,*attribute*]... ::] list of variables**

- Where *attribute* includes:
  - POINTER for a pointer variable, or
  - TARGET for a target variable
- The type, type parameters and rank of a pointer must be the same as the type and rank of any target to which it is pointing
- If a pointer is an array pointer, only the rank, not the shape, need be defined

```
REAL, DIMENSION(20), POINTER :: p   ! illegal

REAL, DIMENSION(:), POINTER :: p    ! legal
```


The type declaration statement for an array pointer specifies the type and the rank of arrays that it can point to.

Note that only the rank is required, not the extent or array bounds.

The dimension attribute of an array pointer cannot specify an explicit-shape or an assumed-shape array, but must take the form of a deferred-shape array, in a similar manner to that used for an allocatable array.

See page 63 of the Student Notes.

Pointer assignment

=>

```
REAL, POINTER :: p1, p2
REAL, TARGET :: t1 = 3.4, t2 = 4.5
p1 => t1
PRINT *, t1, p1              ! all 3.4
p2 => t2
PRINT *, t2, p2              ! all 4.5
```

```
p2 => p1
PRINT *, t1, p1, p2          ! all 3.4
```

```
t1 = 5.2
PRINT *, t1, p1, p2          ! all 5.2
```


A pointer can be set up as an alias of a target by a pointer assignment statement, which is executable and takes the form:

pointer => target

See page 64 of the Student Notes.

Pointer assignment

Vs ordinary assignment

```
REAL, POINTER :: p1, p2
REAL, TARGET :: t1 = 3.4, t2 = 4.5
p1 => t1
PRINT *, t1, p1              ! all 3.4
p2 => t2
PRINT *, t2, p2              ! all 4.5
```

```
p2 = p1                      ! Equivalent to t2 = t1
                             ! So p1 = p2 = t1 = t2 = 3.4
PRINT *, t1, t2, p1, p2 ! all 3.4
```

p2 = p1 is an ordinary assignment, contrast this with the preceding example.

See page 64 of the Student Notes.

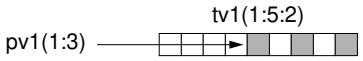
Pointer assignment

Array pointers

```
REAL, DIMENSION (:), POINTER :: pv1
REAL, DIMENSION (-3:5), TARGET :: tv1
pv1 => tv1                ! pv1 aliased to tv1
```

```
pv1 => tv1(:)             ! pv1 points to tv1 with
                          ! section subscript
```

```
pv1 => tv1(1:5:2)         ! pv1 points to a section of
                          ! tv1 with subscript triplet
```



The pointer pv1 may be associated at different times with arrays (array sections) having different extents, because it is only the rank, not the extent that matters.

If an array pointer is aliased with an array, its extents remain the same as its target array. So with pv1 => tv1, pv1 has the same lower and upper bounds as tv1, i.e., -3:5.

If an array pointer points to an array section, its lower bound in each dimension is always renumbered with 1. So with pv1 => tv1(:), where the array section subscript is used, the lower and upper bounds of pv1 are 1:9 instead of -3:5.

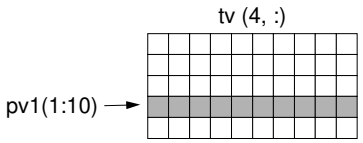
It is legitimate to associate an array pointer with an array section defined by a subscript triplet, but it is not permitted to associate one with an array section defined by a vector subscript.

See page 65 of the Student Notes.

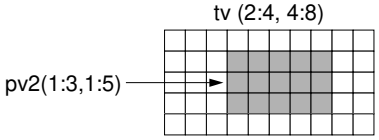
Pointer assignment

Array pointers

```
REAL, DIMENSION (:), POINTER :: pv1
REAL, DIMENSION (:, :), POINTER :: pv2
REAL, DIMENSION (5, 10), TARGET :: tv
pv1 => tv(4, :)      ! pv1 aliased to the 4th row of tv
```




```
pv2 => tv(2:4, 4:8)
```



The lower bound renumbering also happens when pv2 is aliased to the array sections of tv2.

Pointer status

- Undefined - As at start of program
- Null - Not the alias of any data object

  Must not reference undefined pointer, so set to null with NULLIFY statement
- Associated - The alias of a data object

  The status can be tested by ASSOCIATED intrinsic function, for example:

```
REAL, POINTER :: p          ! p undefined
REAL, TARGET :: t
PRINT *, ASSOCIATED (p)     ! not valid
NULLIFY (p)
PRINT *, ASSOCIATED (p)     ! .FALSE.
p => t
PRINT *, ASSOCIATED (p)     ! .TRUE.
```


A NULLIFY statement can have more than one pointer argument.

There is one restriction concerning the use of ASSOCIATED function, that is the pointer argument must not have an undefined pointer association status, otherwise the result is unpredictable and users can not rely on the result when this type of inquiry is made.

Therefore, it is recommended that a pointer should always be either associated with a target immediately after its declaration, or nullified by the NULLIFY statement to ensure its null status.

See page 66 of the Student Notes.

Dynamic storage

ALLOCATE and DEALLOCATE

- Can allocate storage for a pointer to create an un-named variable or array of specified size with implied target attribute:

```
REAL, POINTER :: p
REAL, DIMENSION (:, :), POINTER :: pv
INTEGER :: m, n
...
ALLOCATE (p, pv(m, n))
```

- Can release storage when no longer required:

```
DEALLOCATE (pv)     ! pv is in null status
```

- Before assignment like p = 3.4 is made, p must be associated with its target via ALLOCATE, or aliased with target via pointer assignment statement

When the area of memory allocated for a pointer is deallocated, the association status of that pointer becomes null.

See page 67 of the Student Notes.

Dynamic storage

Danger!

- Dangling pointer

```
REAL, POINTER :: p1, p2
ALLOCATE (p1)
p1 = 3.4
p2 => p1
...
DEALLOCATE (p1)
```

- Unreferenced storage

```
REAL, DIMENSION(:), POINTER :: p
...
ALLOCATE(p(1000))
...
NULLIFY(p)  ! nullify p without first deallocating it
```


The ability to create dynamic memory brings greater versatility and freedom to programming, but also brings its own problems if care is not taken.

Dangling pointer: p1 and p2 both are alias of the same dynamic variable. After the deallocation, p1 is disassociated and the dynamic variable to which it was pointing is destroyed. Since the dynamic variable that p2 was aliasing has disappeared, p2 becomes a dangling pointer and a reference to p2 will produce unpredictable results. So make sure that p2 is nullified after the deallocation.

Unreferenced storage: If p is nullified, set to point to somewhere else, or the subprogram is exited (p has no SAVE attribute), without first deallocating it, there is no way to refer to that storage block and so it can not be released. So make sure to deallocate p before modifying it.

Pointer arguments

- Pointers, whether allocated or not, are allowed to be procedure arguments (allocatable arrays can not be dummy arguments and only allocated allocatable arrays can be actual argument)
- If a procedure has a pointer or target dummy argument, the interface to the procedure must be explicit
- If a dummy argument is a pointer, then the actual argument must be a pointer with the same type, type parameter and rank
- A pointer dummy argument can not have the intent attribute
- If the actual argument is a pointer but the dummy argument is not, the dummy argument becomes associated with the target of the pointer

See page 68 of the Student Notes.

Pointer arguments

Example

```
...                              ! calling program unit
INTERFACE                        ! do not forget interface
   SUBROUTINE sub2(b)
      REAL, DIMENSION(:, :), POINTER :: b
   END SUBROUTINE sub2
END INTERFACE
REAL, DIMENSION(:, :), POINTER :: p
...
ALLOCATE (p(50, 50))
CALL sub1(p)                     ! both sub1 and sub2
CALL sub2(p)                     ! are external procedures
...

SUBROUTINE sub1(a)     ! a is not a pointer
   REAL, DIMENSION(:, :) :: a
   ...
END SUBROUTINE sub1

SUBROUTINE sub2(b)     ! b is a pointer
   REAL, DIMENSION(:, :), POINTER :: b
   ...
END SUBROUTINE sub2
```


Because sub2 has a pointer dummy argument, an interface block is required to provide an explicit interface in the calling program unit (not for sub1 since it has an assumed shape array as a dummy argument). An alternative approach would be using a module or internal procedure to provide an explicit interface by default.

Pointer functions

- A function result may also have the POINTER attribute
- Useful if the result size depends on calculations performed in the function
- The result can be used in an expression, but must be associated with a defined target
- The interface to a pointer function must be explicit

See page 69 of the Student Notes.

Pointer functions

Example

```
...
IMPLICIT NONE
INTEGER, DIMENSION(100) :: x
INTEGER, DIMENSION(:), POINTER :: p
...
p => gtzero(x)
...
CONTAINS
   ! function to get all values .gt. 0 from a
   FUNCTION gtzero(a)
      INTEGER, DIMENSION(:), POINTER :: gtzero
      INTEGER, DIMENSION(:) :: a
      INTEGER :: n
      ...      ! find the number of values .gt. 0, n
      ALLOCATE (gtzero(n))
      ...      ! put the found values into gtzero
   END FUNCTION gtzero
   ...
END
```


The pointer function gtzero has been put as an internal procedure, because the interface to a pointer function must be explicit.

If n equals 0, the complete code should be:

```
...
IF (n == 0)
   NULLIFY(gtzero)
ELSE
   ALLOCATE(gtzero(n))
ENDIF
...
```

See page 69 of the Student Notes.

Arrays of pointers

- An array of pointers can not be declared directly:

```
REAL, DIMENSION(20), POINTER :: p   ! illegal
```

- An array of pointers can be simulated by means of a derived type having a pointer component:

```
TYPE real_pointer
   REAL, DIMENSION(:), POINTER :: p
END TYPE real_pointer
TYPE(real_pointer), DIMENSION(100) :: a
INTEGER :: i
...
! possible to refer to the ith pointer by a(i)%p
DO i = 1, 100
   ALLOCATE (a(i)%p(i))
END DO
```


We have already stated that, because a pointer is an attribute and not a data type, an array of pointers can not be declared directly.

However, a pointer not only can point at an object of intrinsic type or derived type, but also can be a component of a derived type. Therefore, an array of pointers can be easily simulated by means of a derived type having a pointer component of the desired type, and then creating an array of that derived type.

The example shows each row of a lower triangular matrix may be represented by a dynamic array of increasing size.

Note that a(i)%p points to a dynamically allocated real array of size i and therefore this representation uses only half the storage of conventional two dimensional array.

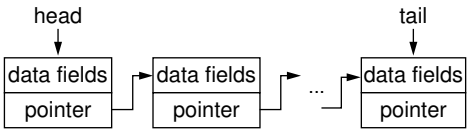
See page 70 of the Student Notes.

Linked list

- A pointer component of a derived type can point at an object of the same type; this enables a linked list to be created

```
TYPE node
    INTEGER :: value                  ! data field
    TYPE (node), POINTER :: next      ! pointer field
END TYPE node
```

- A linked list typically consists of objects of a derived type containing fields for the data plus a field that is a pointer to the next object of the same type in the list



One of the common and powerful applications of pointers is in setting up and manipulating a linked list.

The connected objects in a linked list can be called nodes.

Conventionally, the first node in the list is referred to as the head of the list, while the last node is called the tail.

Linked list

- In a linked list, the connected objects
  - Are not necessarily stored contiguously
  - Can be created dynamically at execution time
  - May be inserted at any position in the list
  - May be removed dynamically
- The size of a list may grow to an arbitrary size as a program is executing
- Trees or other dynamic data structures can be constructed in a similar way

Linked list

Example

```
...
TYPE node
    INTEGER :: value                  ! data field
    TYPE (node), POINTER :: next      ! pointer field
END TYPE node

INTEGER :: num
TYPE (node), POINTER :: list, current

NULLIFY(list)             ! initially nullify list (empty)

DO
    READ *, num           ! read num from keyboard
    IF (num == 0) EXIT    ! until 0 is entered
    ALLOCATE(current)     ! create new node
    current%value = num
    current%next => list  ! point to previous one
    list => current       ! update head of list
END DO
...
```


Firstly, we define the type of node which contains an integer value as a data field and a pointer component which can point to the next node.

Then two variables of this type are declared, list and current, where list will be used to point to the head of the list and current to a general node of the list.

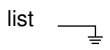
See page 71 of the Student Notes.

Linked list

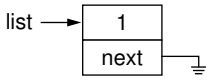
Example

If, for example, the values 1, 2, 3 are entered in that order, the list looks like (progressively):

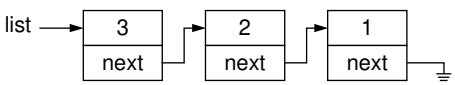
- After NULLIFY(list)



- After the first num is read



- After all 3 numbers are read



At first, the list is empty, so list should point to both the beginning and the end. This is effected by initially nullifying list, which is represented by the symbol for earthing an electrical conductor.

Now suppose the value 1 is read in (num contains the value), a list is initially set up with one node, containing the integer value 1. This is achieved by firstly allocating a dynamic storage for current, then giving it the value num, and finally setting list to point to the head of the list which is the newly allocated current node.

This process can be repeated as long as the value 0 is read in.

See page 71 of the Student Notes.

New I/O Features

New I/O features

Topics

- Non-advancing I/O
- INQUIRE by I/O list
- NAMELIST
- New edit descriptors
- New statement specifiers

Non-advancing I/O

- Each READ or WRITE normally involves full records
- Non-advancing I/O permits READ or WRITE without advancing position to new record
- ADVANCE='NO' specifier:

```
WRITE(*, '("Input size?")', ADVANCE='NO')
READ(*, '(I5)') n
```

- Non-advancing I/O not applicable with list directed I/O

On the READ or WRITE statement, the default normal (advancing) I/O occurs.

See page 75 of the Student Notes.

INQUIRE by I/O list

```
INQUIRE (IOLENGTH=length) output-list
```

- To determine the length of an unformatted output item list
- May be used as value of RECL specifier in subsequent OPEN statement
- Example:

```
INTEGER :: rec_len
...
INQUIRE (IOLENGTH = rec_len) name, title, &
    age, address, tel
...
OPEN (UNIT = 1, FILE = 'test', RECL = rec_len, &
    FORM = 'UNFORMATTED')
...
WRITE(1) name, title, age, address, tel
...
```


INQUIRE by FILE and UNIT have already existed.

RECL - length of record if access is direct.

See page 76 of the Student Notes.

NAMELIST

- Gather set of variables into group to simplify I/O
- General form of NAMELIST statement:

  NAMELIST /namelist-group-name/ variable-list
- Use namelist-group-name as format instead of io-list on READ and WRITE
- Input record has specific format:

  &namelist-group-name var2=x, var1=y, var3=z/
- Variables optional and order unimportant

The NAMELIST statement has been available as a suppliers extension to Fortran since the early days (it was available as an IBM extension to FORTRAN II in the early 60's!). It has now been included in the Fortran 90 language.

Don't use it unless you have to.

See page 76 of the Student Notes.

NAMELIST

Example

```
...
INTEGER :: size = 2
CHARACTER (LEN = 4) :: &
    colour(3) = (/ ' red', 'pink', 'blue' /)
NAMELIST /clothes/ size, colour
WRITE(*, NML = clothes)
...
```

outputs:

```
&CLOTHES SIZE = 2, COLOUR =  redpinkblue/
```

New edit descriptors

- EN - (Engineering) Same as E but exponent divisible by 3, value before decimal point between 1 and 1000
- ES - (Scientific) Same as E but value before decimal point is between 1 and 10
- B - Binary
- O - Octal
- Z - Hexadecimal
- G - Generalised edit descriptor now applicable for all intrinsic types

Edit descriptors specify exactly how values should be converted into a character string on an output device or internal file, or converted from a character string on an input device or internal file. These are new edit descriptors provided by Fortran 90.

See page 77 of the Student Notes.

New edit descriptors

Example

To compare the differences among E, EN, ES and G edit descriptors:

```
PROGRAM e_en_es_g_compare
   IMPLICIT NONE
   REAL, DIMENSION(4) :: &
      x=(/1.234, -0.5, 0.00678, 98765.4/)
   PRINT '(4E14.3/4EN14.3/4ES14.3/4G14.3)',&
      x, x, x, x
END PROGRAM e_en_es_g_compare
```

Output:

```
 0.123E+01    -0.500E+00  0.678E-02   0.988E+05
 1.234E+00  -500.000E-03  6.780E-03  98.765E+03
 1.234E+00    -5.000E-01  6.780E-03   9.877E+04
  1.23           -0.500    0.678E-02   0.988E+05
```


Taken from Ellis' book.

G edit descriptor is 'redundant' because using I, E, EN, ES, F, L, or A edit descriptors can provide some checking whether the data types are all corrected.

New statement specifiers

- INQUIRE

```
POSITION     =   'ASIS'  'REWIND'  'APPEND'
ACTION       =   'READ'  'WRITE'   'READWRITE'
DELIM        =   'APOSTROPHE' 'QUOTE' 'NONE'
PAD          =   'YES'   'NO'
READWRITE    = )
READ         = )  'YES'   'NO'     'UNKNOWN'
WRITE        = )
```

- OPEN

  POSITION, ACTION, DELIM, PAD are same as above
  STATUS = 'REPLACE'

- READ/WRITE

```
NML          =   namelist_name
ADVANCE      =   'YES'   'NO'
```

- READ

  EOR = label SIZE = character_count

DELIM: The character used to delimit character constants (default NONE). If the output is being sent to a file to be subsequently read by another program, using list-directed or namelist formatting, then it may be essential to delimit any character strings in this way. PAD: 'YES' (default) means the input record is padded with blanks. 'NO' means the input record must be long enough to accommodate the input list (except for non advancing I/O).

STATUS = 'REPLACE': If the file to open does not exist it is created, and if it does exist it is deleted and a new one is created.

EOR: Control passes to statement label when an end-of-record condition occurs. SIZE: The integer variable character_count is assigned the number of characters read-excluding any padding characters, on completion of the non-advancing read.

See page 77 of the Student Notes.

Intrinsic Procedures

Intrinsic procedures

Topics

- Rich set with four categories
- List of new intrinsic procedures

Fortran 90 offers over one hundred intrinsic procedures (104), all of which can be referenced using keyword arguments and many having optional arguments. Intrinsic functions that can only be used with one data type have now been superseded by generic versions.

Intrinsic procedures

Four categories

- Elemental procedures

  A set of functions and one sub-routine, specified for scalar arguments, but applicable for conforming array arguments,

- Inquiry functions

  Return properties of principal arguments that do not depend on their values

- Transformational functions

  usually have array arguments and an array result whose elements depend on many of the elements of the arguments

- Nonelemental subroutines

Elemental procedures: numeric - ABS, INT, etc. and mathematical - SIN, SQRT, etc.

Inquiry functions: KIND, PRESENT, ASSOCIATED, etc.

Transformational functions: array reduction functions ANY, ALL, MAXVAL, SUM etc.

Nonelemental subroutines: not mentioned before, about real-time clock and random numbers.

See page 81 of the Student Notes.

New intrinsic procedures

Elemental functions:

- Numeric
  - CEILING (A)
  - FLOOR (A)
  - MODULO (A, P)
- Character
  - ACHAR (I)
  - ADJUSTL (STRING)
  - ADJUSTR (STRING)
  - IACHAR (C)
  - INDEX (STRING, SUBSTRING [, BACK])
  - LEN_TRIM (STRING)
  - SCAN (STRING, SET [, BACK])
  - VERIFY (STRING, SET [, BACK])

For elemental functions, see page 81 to page 83 of the Student Notes.

New intrinsic procedures

- Bit manipulation
  - BTEST (I, POS)
  - IAND (I, J)
  - IBCLR (I, POS)
  - IBITS (I, POS, LEN)
  - IBSET (I, POS)
  - IEOR (I, J)
  - IOR (I, J)
  - ISHFT (I, SHIFT)
  - ISHFTC (I, SHIFT [, SIZE])
  - NOT (I)
- Kind
  - SELECTED_INT_KIND (R)
  - SELECTED_REAL_KIND (P,R)

New intrinsic procedures

- Floating point manipulation
  - EXPONENT (X)
  - FRACTION (X)
  - NEAREST (X, S)
  - RRSPACING (X)
  - SCALE (X, I)
  - SET_EXPONENT (X, I)
  - SPACING (X)
- Logical:
  - LOGICAL (L [,KIND])

Elemental subroutine:

- MVBITS (FROM, FROMPOS, LEN, TO, TOPOS)

New intrinsic procedures

Inquiry functions:

- PRESENT (A)
- ASSOCIATED (POINTER [,TARGET])
- KIND (X)
- BIT_SIZE (I)

- Numeric
  - DIGITS (X)
  - EPSILON (X)
  - HUGE (X)
  - MAXEXPONENT (X)
  - MINEXPONENT (X)
  - PRECISION (X)
  - RADIX (X)
  - TINY (X)

For inquiry functions, see page 83 of the Student Notes.

New intrinsic procedures

Transformational functions:

- REPEAT (STRING, NCOPIES)
- TRIM (STRING)
- TRANSFER (SOURCE, MOLD [, SIZE])

Non elemental intrinsic subroutines:

- DATE_AND_TIME ([DATE][,TIME][,ZONE][VALUES])
- SYSTEM_CLOCK ([COUNT][,COUNT_RATE][COUNT_MAX])
- RANDOM_NUMBER (HARVEST)
- RANDOM_SEED ([SIZE] [,PUT] [,GET])

Array intrinsic procedures:

- See section on array processing

For transformational functions, see page 84 of the Student Notes.

For non elemental intrinsic subroutines, see page 84 of the Student Notes.

Redundant Features

Redundant features

Topics

- Obsolescent features
- Other redundant features
- Redundant features are divided into 5 groups:
  - Source form
  - Data
  - Control
  - Procedures
  - Input/Output

Obsolescent features are those listed by the standard itself, which might be removed from the next major revision (.e.g, Fortran 95) and should never be used in new or revised programs.

Fortran 90 includes other redundant features which are not recommended to use in new or revised programs because there are newer features that are better designed and safer to use.

See page 87 of the Student Notes.

Redundant features

Source form

- Fixed source form

  Use new free source form instead

- INCLUDE line

  Use modules instead

See page 87 of the Student Notes.

Redundant features

Data

- DOUBLE PRECISION data type

  Superseded by the concept of parameterised data types for numerical **portability**

- Implicit typing and IMPLICIT statement

  Use IMPLICT NONE statement always

- Old form of type declaration statements e.g.: REAL list of real variables

  Use the new form with a double colon (::) between the type and the list of variables

- PARAMETER, DIMENSION and other attribute statements

  Use of the attribute forms, rather than the statement forms

See page 87 of the Student Notes.

Redundant features

Data

- DATA statement

  Not needed as variables may be initialised in a type statement except for octal, hexadecimal and array section initialisations

- Assumed-size arrays as dummy arrays

  Superseded by the assumed-shape arrays

- COMMON blocks and BLOCK DATA

  Use of modules obviates the need for them

- EQUIVALENCE statement

  The introduction of modules, dynamic storage allocation, pointers, and the intrinsic function TRANSFER makes its use unnecessary

- SEQUENCE attribute

  Should never be used

Redundant features

Control

- Obsolescent features
  - **Arithmetic IF statement**
  - **Shared DO termination, and DO termination on a statement other than on a CONTINUE or an END DO statement**
  - **REAL and DOUBLE precision DO variables and control expressions**
  - **ASSIGN and assigned GO TO statements**
  - **Branching to END IF from outside IF block**

  Can be replaced by IF statement, IF, DO and CASE control constructs, and EXIT and CYCLE statements
  - **Alternate RETURN**
  - **PAUSE statement**

  The above listed should never be used in new or revised programs.

See page 88 of the Student Notes.

Redundant features

Control

- Labels

  The use of labels are rarely necessary with the introduction of modern control constructs and the character string format specifications
- CONTINUE statement to end DO loop

  Can be replaced by using DO construct, EXIT and CYCLE statements
- GO TO and computed GO TO statements

  IF, DO and CASE constructs, EXIT and CYCLE statements should be used instead
- DO WHILE logical-expression statement

  Can replaced by DO loop construct and EXIT statement

Redundant features

Procedures

- Intrinsic functions with specific names for different data types

  Superseded by generic versions (note: a specific name is required only when an intrinsic function is being used as an actual argument)

- ENTRY statement

  Can be replaced by internal procedures and modules

- Statement functions

  Superseded by the concept of internal procedures

- External procedures and EXTERNAL statement

  Putting procedures either in a module or another program unit as internal procedures makes things much easier that it really is not necessary to use external procedures in any new or revised programs

External procedures and EXTERNAL statement may still need when linking to Fortran 77 libraries.

See page 88 of the Student Notes.

Redundant features

Input/Output

- Obsolescent features
  - **Assigned FORMAT specifiers,** should be replaced by the character string format specifications
  - **H edit descriptor**, should be replaced by the character constant edit descriptor A
- FORMAT statement

  The same effect can be achieved by assigning the format specification to a character string

- ERR=, END=, and EOR= specifiers

  Each of these conditions are recommended to be checked by using the IOSTAT= specifier

- NAMELIST input/output

  Is an extremely poorly designed feature and best not to use it unless absolutely necessary

See page 88 of the Student Notes.

Redundant features

Input/Output

- D edit descriptor

  Superseded by the E edit descriptor

- BN and BZ edit descriptors

  The BLANK specifier provides a better way to deal with the problem

- P edit descriptor

  Can lead to confusion and is best to avoid

- G edit descriptor

  Using I, E, EN, ES, F, L, or A edit descriptors can provide some checking whether the data types are all corrected

- X edit descriptor

  This position edit descriptor is the same as Turn

Further Development

Fortran 95

- Corrections to Fortran 90
- FORALL

```
FORALL (i = 1:n, j = 1:n, y(i, j) /= 0 .AND. i /= j) &
    x(i, j) = 1.0 / y(i, j)

FORALL (i = 1:n)
    a(i, i) = i
    b(i) = i*I
END FORALL
```

- PURE attribute - make PURE procedures safe for use in FORALL statement
- CPU time - CALL CPU_TIME(t1)
- Also allocatable dummy arguments and results

Work did not stop with the publication of the Fortran 90 standard.

A new release is scheduled for 1996 (called 95), mainly devoted to clarifications, corrections and interpretations. A number of obsolescent features will be deleted from the standard and additional features will be declared as obsolescent. Fortran 95 only has minor additions to Fortran 90.

A more important revision is scheduled for 2000.

Documents related to the work of X3J3 can be found via anonymous ftp on ftp.ncsa.uiuc.edu, directory x3j3.

See page 91 of the Student Notes.

Fortran 95

- Nested WHERE

```
WHERE (mask1)
    ...
    WHERE (mask2)
        ...
    ELSEWHERE
        ...
    ENDWHERE
ELSEWHERE
    ...
ENDWHERE
```

- Initial pointer or type default status

```
REAL, POINTER :: p(:) => NULL()

TYPE string
    CHARACTER, POINTER :: ch(:) => NULL()
ENDTYPE
```

Further development

With FORTRAN 90 barely out of cradle, why are there already efforts to enhance it?

- The limelight of HPC research has recently shifted away from vector computers and towards parallel and “massive” parallel computers
- Fortran 90 may be suitable for HPC on conventional and vector computers, but it does not include features to obtain the best performance on parallel systems.
- Demands for such features has led to the development of two de facto standards:
  - High Performance Fortran (HPF)
  - Message Passing Interface (MPI)

HPC: High Performance Computing

See page 92 of the Student Notes.

HPF

Goals

HPF provides a set of language extensions to Fortran 90 to support:

- Data parallel programming
- Top performance on MIMD and SIMD computers with non-uniform memory access
- Code tuning for various architectures
- Minimal deviation from other standards
- Define open interfaces to other languages

MIMD: Multiple Instruction Stream, Multiple Data Stream

SIMD: Single Instruction Stream, Multiple Data Stream

See page 92 of the Student Notes.

HPF

What is in it

- Compiler directives for data alignment and distribution
- Concurrent execution features using the FORALL statement
- A number of intrinsic functions to enquire about machine specific details
- A number of extrinsic functions which provide an escape mechanism from HPF
- A library of routines to support global operations

MPI

What

Message Passing Interface is a proposed standard for:

- Explicit message passing
- Application programs
- MIMD distributed memory concurrent computers
- Workstation networks

See page 92 of the Student Notes.

MPI

Why

- Portability and ease-of-use
- Time right for standard
- Library construction
- Prerequisite for development of concurrent software industry
- Provides hardware vendors with well-defined set of routines that they must implement efficiently

MAN T&EC

The Manchester and North HPC
Training and Education Centre

Fortran 90 (Edition 3.0)

Speaker Notes Comment Sheet

Please send comments you have on any of the Fortran 90 materials to:

Andrew Grant
Manchester Computing Centre
The University of Manchester
Manchester, M13 9PL

Alternatively, email hpc-info@mcc.ac.uk

Name: .....

Postal Address:.....

.....

Email Address:.....

Comments:

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....