THE UNIVERSITY of MANCHESTER

# Fortran 90

## *A Conversion Course for Fortran 77 Programmers*

## Student Notes

**S Ramsden, F Lin**

*Manchester and North HPC T&EC*

**M A Pettipher, G S Noland, J M Brooke**

*Manchester Computing Centre, University of Manchester*

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

*Edition 3.0 July 1995*

# Acknowledgements

These student notes were developed using the Manchester Computing Centre Fortran 90 course, which was compiled by J M Brooke, G S Noland, and M A Pettipher as a basis.

Useful comments were also provided by the following: T L Freeman, J Gajjar and A J Grant (The University of Manchester), and A Marshall and J S Morgan (The University of Liverpool).

Michael Hennecke, University of Kahlsruhe, Germany, provided comprehensive comments which were incorporated in edition 3.0 of the materials.

Some material was provided, with permission, by Walt Brainerd, copyright Unicomp, Inc.

# Table of Contents

# 1 Introduction

This course covers the transition from the programming language Fortran 77 to the more modern Fortran 90, and is aimed at Fortran 77 programmers who require an understanding of the principles and new features of Fortran 90. The course may also be suitable for programmers familiar with languages such as C or Pascal, but not for complete beginners in programming.

## 1.1 History

The programming language Fortran was originally designed for the solution of problems involving numerical computation. The development of Fortran dates back to the 1950s, the first Fortran system being released in 1957, for the IBM 704.

In the early 1960s, as other manufacturers released Fortran compilers for their own computer systems, the need to control and standardise Fortran became apparent. A standards committee was established in 1962, and the first Fortran standard was published in 1966.

Unfortunately, the 1966 Standard did not give a clear, precise definition of Fortran. 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.

The next Fortran revision is expected within the next 10 years.

## 1.2 Objectives

The objectives of the new Fortran 90 standard were:

- to modernise the language in response to the developments in language design which have been exploited in other languages.
- to standardise vendor extensions such that an efficient portable language is provided.
- to improve the safety of programming in the language and to tighten the conformance requirement, such that the risk of error in standard code is reduced.
- to keep compatible with Fortran 77 by adopting a language evolution method such that the vast investment in Fortran 77 code is preserved.

## 1.3 Language Evolution

Fortran 90 is a superset of Fortran 77, and so all standard Fortran 77 programs should run. To prevent the language growing progressively larger, however, as new revisions

are produced, the standards committee has adopted a policy of removing obsolete features.

This procedure adopted involves the inclusion of two lists with each new standard. One list contains the deleted features, and the other contains the obsolescent features. The obsolescent list consists of features which are considered to be redundant and may be deleted in the next revision. A feature must appear on the obsolescent list before it can be deleted, thus providing a period of notice of at least one revision cycle.

Fortran 90 contains no deleted features, but contains the following obsolescent features which may be removed at the next 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 `GOTO` statements
- Assigned `FORMAT` specifiers
- Branching to `END IF` from outside `IF` block
- Alternate `RETURN`
- `PAUSE` statement
- `H` edit descriptor

# 1.4 New Features

The following major new features are included in Fortran 90:

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

Other new features include:

- Free format source code
- Specifications/ `IMPLICIT NONE`
- Parameterised data types
- Derived types
- Operator overloading
- `CASE` statement
- `EXIT` and `CYCLE`
- Many new intrinsic functions
- New I/O features

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 is conserved by dynamic memory allocation. Additionally, data parallel capability is provided through the array processing features, which makes Fortran 90 a more efficient language on the new generation of high performance computers.

# 1.5 Organisation

These student notes are arranged in the following chapters:

1. Introduction.
2. Sources, Types and Control Structures.
3. Procedures and Modules.
4. Array Processing.
5. Pointer Variables.
6. Input/Output.
7. Intrinsic Procedures.
8. Redundant Features.
9. Further Development.

Where appropriate, exercises are included at the end of each chapter. Source code of example programs and solutions to exercises are all available on line. Program names appearing in parenthesis are solutions to exercises.

Fortran 90 references and further sources of information are provided in the Resource List supplied with the course material. Additionally, the compiled resource list is available on the World Wide Web via the following URL:

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

# 1.6 Coding Convention

In these student notes code is in this font, for example:

```
! this is code
```

The coding convention followed throughout the student notes is:

- All keywords and intrinsic function names are in capitals; everything else is in lower case.
- The bodies of program units are indented by two columns, as are `INTERFACE` blocks, `DO`-loops, `IF`-blocks, `CASE`-blocks, etc.
- The name of a program, subroutine, or function is always included on its `END` statement.
- In `USE` statements, the `ONLY` clause is used to document explicitly all entities which are actually accessed from that module.
- In `CALL` statements and function references, argument keywords are always used for optional arguments.

# 2 Sources, Types and Control Structures

## 2.1 Source Form

Fortran 90 supports two forms of source code; the old Fortran 77 source code form (now called *fixed form*), and the new free form. Using free source form, columns are no longer reserved and so Fortran statements can now appear anywhere on a source line. The source line may contain up to 132 characters.

The character set now includes both upper and lower case letters and the underscore. A good convention is that the words which are not in your control are in upper case and names which you invent yourselves, such as variable names, are in lower case.

Identifier names can consist of between 1 and 31 alphanumeric characters (including the underscore), the only restriction being that the first must be a letter . Remember that the use of sensible names for variables helps readability.

Fortran 90 introduces new symbols, including the exclamation mark, the ampersand, and the semicolon, and the alternative form of relational operators. These are discussed in the following paragraphs.

The exclamation mark introduces a comment. A comment can start anywhere on a source line and thus can be placed alongside the relevant code. The rest of the line after the ! is ignored by the compiler.

```
REAL :: length1 ! Length at start in mm (room temperature)
REAL :: length2 ! Length at end in mm (after cooling)
```

The ampersand character, &, means 'continued on the next line'. Usually you will arrange the line break to be in a sensible place (like between two terms of a complicated expression), and then all that is needed is the & at the end of all lines except the last. If you split a string, though, you also need an ampersand at the *start* of the continuation line.

```
loggamma = f + (y-0.5)*log(y) - y + 0.91893853320 + &
          (((-0.00059523810*z + 0.00079365079)*z - &
          0.00277777778)*z + 0.08333333333)/y

WRITE(*,'UNIVERSITY OF MANCHESTER DEPARTMENT&
      & OF THEORETICAL STUDIES')
```

The semicolon is used as a statement separator, allowing multiple statements to appear on one line. The use of multiple-statement lines can, however, produce unreadable code, and should therefore be used only for simple cases, for example:

```
a = 2; b = 7; c = 3
```

Alternative forms of the relational operators are now provided:

```
.LT.  or  <
.LE.  or  <=
.EQ.  or  ==
.NE.  or  /=
.GT.  or  >
.GE.  or  >=
```

# 2.2 Program and Subprogram Names

All programs and subprogram have names. A name can consist of up to 31 characters (letters, digits, or underscore), starting with a letter.

Using square brackets to signify optional items, the syntax of the `PROGRAM` and `END` statements in Fortran 90 is of the form:

```
PROGRAM test
  ...
  ...
END [PROGRAM [test]]
```

where `test` is the name of the program. The `END` statement may optionally be any of:

```
END
END PROGRAM
END PROGRAM test
END PROGRAM TEST
```

If the program name is present then the word `PROGRAM` must also be present, and the name must match that in the `PROGRAM` statement (but case is not significant).

The same syntax applies for other program elements, such as `FUNCTION` or `MODULE.`

# 2.3 Specifications

Fortran 90 allows an extended form of declaration, in which all the attributes of a particular entity may be declared together.

The general form of the declaration statement is:

```
type [ [, attribute ] ... :: ] entity list
```

where *`type`* represents one of the following:

```
INTEGER [([KIND=]kind-value)]
REAL [([KIND=]kind-value)]
COMPLEX [([KIND=]kind-value)]
CHARACTER [(actual-parameter-list)]
LOGICAL [([KIND=]kind-value)]
TYPE (type-name)
```

and *`attribute`* is one of the following:

```
PARAMETER
PUBLIC
PRIVATE
POINTER
TARGET
ALLOCATABLE
```

```
DIMENSION(extent-list)
INTENT(inout)
OPTIONAL
SAVE
EXTERNAL
INTRINSIC
```

For example, it is now possible to initialize variables when they are declared, so there is no need for a separate DATA statement:

```
REAL :: a=2.61828, b=3.14159
! two real variables declared and assigned initial values

INTEGER, PARAMETER :: n = 100, m = 1000
! two integer constants declared and assigned values

CHARACTER (LEN=8) :: ch
! character string of length 8 declared

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

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

# 2.4 Strong Typing

For backward compatibility, the implicit typing of integers and reals by the first character is carried over, but the IMPLICIT statement has been extended to include the parameter NONE. It is recommended that the statement

```
IMPLICIT NONE
```

be included in all program units. This switches off implicit typing and so all variables must be declared. This helps to catch errors at compile time when they are easier to correct. The IMPLICIT NONE statement may be preceded within a program unit only by USE and FORMAT statements.

# 2.5 The Concept of KIND

In Fortran 90 each of the five intrinsic types REAL, INTEGER, COMPLEX, CHARACTER and LOGICAL, has an associated non negative integer value called the kind type parameter. A processor must support at least two kinds for REAL and COMPLEX, and one for INTEGER, CHARACTER and LOGICAL.

KIND values are system dependent. However, there are intrinsics provided for enquiring about and setting KIND values, and these allow the writing of portable code using specified precision.

## 2.5.1 Real Values

The kind type parameter associated with REAL variables specifies minimum precision and exponent range requirements. If the kind type parameter is not specified explicitly then default real is assumed. The assumption of default kind type parameter in the absence of explicit specification is usual for all intrinsic types. The kind value assigned to default real is, of course, processor-dependent.

A kind value is specified explicitly by including the value in brackets in the type declaration statement. For example:

```
REAL(KIND=2) :: a
! a is declared of kind type 2

REAL(KIND=4) :: b
! b is declared of kind type 4
```

The KIND= is optional and so the above declarations could also be given as:

```
REAL(2) :: a
REAL(4) :: b
```

The intrinsic function KIND, which takes one argument of any intrinsic type, returns the kind value of the argument. For example:

```
REAL(KIND=2) :: x     !x declared of kind type 2
REAL :: y             !y declared of default type
INTEGER :: i,j

i = KIND(x)           !i=2
j = KIND(y)           !j set to kind value of default real
                      !j is system dependent
```

The intrinsic function SELECTED_REAL_KIND has two optional integer arguments p and r (optional arguments will be discussed in more detail later in the course). The variable p specifies the minimum precision (number of decimal digits) required and r specifies the minimum exponent range required.

The function SELECTED_REAL_KIND(p,r) returns the kind value that meets, or minimally exceeds, the requirements specified by p and r. If more than one kind type satisfies the requirements, the value returned is the one with the smallest decimal precision. If the precision is not available the value -1 is returned, if the range is not available -2 is returned, and if neither is available -3 is returned. The use of kind type together with this function can provide complete portability.

The simplest example of KIND is to replace DOUBLE PRECISION:

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

Here, the intrinsic function KIND returns the kind value of DOUBLE PRECISION and assigns the value to idp. The variable ra is declared as double precision by specifying KIND=idp. Note that in this case the kind value is system dependent.

In order to declare a real in a system independent way, a kind value associated with a required precision and exponent range must be specified. To do this, the function SELECTED_REAL_KIND should be used. For example:

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

The real variables a, b and c are declared to have *at least* 10 decimal digits of precision and exponent range of *at least* $10^{-200}$ to $10^{+200}$, if permitted by the processor.

Constants can also be specified to be of a particular kind. The kind type parameter is explicitly specified by following the constant's value by an underscore and the kind parameter. If no kind type is specified then the type default is assumed. For example:

```
REAL, PARAMETER :: d = 5.78_2   !d is real of kind type 2
REAL, PARAMETER :: e = 6.44_wp  !e is real of kind type wp
REAL, PARAMETER :: f = 2.7      !f is default real
                                !(system dependent)
```

## 2.5.2 Integer Values

The intrinsic function `SELECTED_INT_KIND` is used in a similar manner to `SELECTED_REAL_KIND`. The function has one integer argument, `r`, which specifies the integer range required. Thus, `SELECTED_INT_KIND(r)` returns the kind value that can represent, *at least*, all the integer values in the range $-10^r$ to $10^r$. If more than one kind type satisfies the r equirement, the value returned is the one with the smallest exponent range. If this range is not available, then the function returns the value -1.

The following example shows the declaration of an integer in a system independent way, specifying a kind value associated with a required range:

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

The integer variables `ia`, `ib` and `ic` can have values between $-10^8$ to $10^8$ *at least*, if permitted by the processor.

Integer constants can also be specified to be of a particular kind in the same way as real constants. For example:

```
INTEGER, PARAMETER :: short = SELECTED_INT_KIND(2)
! the kind type short is defined
INTEGER, PARAMETER :: linefeed = 10_short
INTEGER, PARAMETER :: escape = 27_short
! constants linefeed and escape of kind type short
```

## 2.5.3 Intrinsics

The intrinsic function `KIND`, discussed in section 2.5.1, “Real Values”, can take an argument of any intrinsic type. For example:

```
KIND(0)         ! returns the default integer kind
                ! (processor dependent)
KIND(0.0)       ! returns the default real kind
                ! (processor dependent)
KIND(.FALSE.)   ! returns the default logical kind
                ! (processor dependent)
KIND('A')       ! gives the default character kind (always 1)
```

Further intrinsic functions can be seen in the following examples:

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

This prints the largest integer available for this integer type, and its kind value.

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

This prints the exponent range, the decimal digits of precision, and the kind value of `a`.

### 2.5.4 Complex

Complex data types are built from two reals and so, by specifying the components as reals with the appropriate kind we could have the equivalent of `DOUBLE PRECISION COMPLEX`:

```
INTEGER, PARAMETER :: idp = KIND(1.0D)
COMPLEX(KIND=idp) :: firstroot, secondroot
```

### 2.5.5 Logical

There may be more than one logical kind. For example, on the Salford compiler there are two: the default kind is one 'word' long and has kind value 2, but kind value 1 specifies compression to one byte.

### 2.5.6 Character

Only one kind is generally available, which maps to the standard ASCII set, but the language now allows for other kinds to be provided to cater for foreign language characters.

```
CHARACTER (LEN=5) :: 'aeiou'
CHARACTER (LEN=5, KIND=1) :: 'aeiou'
```

For character constants, the kind value *precedes* the constant (separated by an underscore):

```
CHARACTER, LEN=5, PARAMETER :: vowels = 1_'aeiou'
```

## 2.6 Derived Types

One of the major advances of Fortran 90 over previous versions is the ability to define your own types. These are called *derived types*, but are often also called *structures*.

Let us define a new type, `point`, which will be constructed from three values representing the x, y, and z values in Cartesian space.

```
TYPE point
  REAL :: x, y, z
END TYPE point
```

We can now declare new variables to be of type `point` as follows:

```
TYPE (point) :: centre, apex
```

Here we have declared two variables, `apex`, and `centre` to be of type `point`. Notice that the syntax of Fortran 90 doesn't allow us to say simply:

```
point :: centre, apex ! Illegal
```

You 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 `point` in brackets for the declarations of the variables, but not for the type definition.

Each of the components of the variable `apex` may be referenced individually by means of the *component selector* character, %.

```
apex%x = 0.0
apex%y = 1.0
apex%z = 0.0
```

The value `apex%y` is a real quantity and the assignment operator (=) is defined for real quantities. For derived types the assignment is implicitly defined to be on a component by component basis, which will usually be what is wanted, so we can say, for example:

```
centre = apex
```

No other operators are defined for our new type by default, however , and we might not want assignment to do a straight copy (for example, one component might be a date field and we might want to update it, or check it). This problem is overcome by *overloading* the assignment operator. This and the associated problem of defining what interpretation should be given to other operations on variables of our new type will be dealt with later in the course.

We can use our new type as a primitive in constructing further more complicated types:

```
TYPE block
  TYPE (point) :: bottomleftnear, toprightfar
END TYPE block
```

To refer to the x component of the bottom left corner of a variable `firstbrick` (say) of type `block`, we would need two % signs:

```
xoffset = firstbrick%bottomleftnear%x
```

## 2.6.1 Arrays of a Derived Type

We can declare an array of a derived type:

```
INTEGER, PARAMETER :: male = 1, female = 0
INTEGER, PARAMETER :: nbefore = 53, nafter = 37

TYPE person
  INTEGER :: ident
  INTEGER :: sex
  REAL :: salary
END TYPE person

TYPE (person), DIMENSION (nbefore) :: group1
TYPE (person), DIMENSION (nafter) :: group2
```

Here we have declared two arrays, `group1`, and `group2` of type `person`. If we now say

```
group1%sex = female
```

we will set the sex of *all* the members of our first group to female.

## 2.6.2 Constants of Derived Types

We can define a constant of a derived type:

```
TYPE (point) :: origin = point(0.0, 0.0, 0.0)
TYPE (person) :: boss = person(1, male, 100000.0)
```

The order of the components must follow the order in the definition. The constants, such as `point(0.0,0.0,0.0)` may appear anywhere that a variable of the appropriate type may appear.

## 2.6.3 Derived Type Examples

Define the form of the derived type:

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

Declare structures of type `vreg`:

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

Assign a constant value to `mycar1`:

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

Use % to assign a component of `mycar2`:

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

Define an array of a derived type:

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

Define a derived type including another derived type:

```
TYPE household
  CHARACTER (LEN=1) :: name
  CHARACTER (LEN=50) :: address
  TYPE(vreg) :: car
END TYPE household
```

Declare a structure of type `household`:

```
TYPE(household) :: myhouse
```

Use % to refer to `year` component:

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

# 2.7 Control Statements

Fortran 90 contains three block control constructs:

- `IF`
- `DO`
- `CASE`

All three constructs may be nested, and additionally may be named in order to help readability and increase flexibility.

## 2.7.1 IF Statements

The general form of the IF construct is:

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

Notice there is one minor extension, which is that the `IF` construct may be named. The `ELSE` or `ELSE IF` parts may optionally be named, but, if either is, then the `IF` and `END IF` statements must also be named (with the same name).

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

For long or nested code this can improve readability.

## 2.7.2 DO Loop

The general form of the `DO` loop is:

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

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.

Old style code:

```
      DO 10 I = 1,N
      DO 10 J = 1,M
10      A(I,J) = I + J
```

Fortran 90 code:

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

Notice that there is no need for the statement label at all.

The `DO` and `END DO` may be named:

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

One point to note is that the loop variable must be an integer and it must be a simple variable, not an array element.

The `DO` loop has three possible control clauses:

- an iteration control clause (as in example above).
- a `WHILE` control clause (described below).
- an empty control clause(section 2.7.4, “EXIT and CYCLE”)

## 2.7.3 DO WHILE

A `DO` construct may be headed with a `DO WHILE` statement:

```
DO WHILE (logical expression)

  body of loop

END DO
```

The body of the loop will contain some means of escape, usually by modifying some variable involved in the test in the `DO WHILE` line.

```
DO WHILE (diff > tol)
    .
    .
    .
  diff = ABS(old - new)
    .
    .
    .
END DO
```

Note that the same effect can be achieved using the `DO` loop with an `EXIT` statement which is described below.

## 2.7.4 EXIT and CYCLE

The `EXIT` statement permits a quick and easy exit from a loop before the `END DO` is reached. (It is similar to the `break` statement in C.)

The CYCLE statement is used to skip the rest of the loop and start again at the top with the test-for-completion and the next increment value (rather like the continue statement in C).

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

These two statements allow us to simplify the DO statement even further to the 'do forever' loop:

```
DO
    .
    .
    .
  IF ( ... ) EXIT
    .
    .
    .
END DO
```

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

By default the CYCLE statement applies to the inner loop if the loops are nested, but, as the DO loop may be named, the CYCLE statement may cycle more than one level. Similarly, the EXIT statement can specify the name of the loop from which the exit should be taken, if loops are nested, the default being the innermost loop.

```
outer:DO i = 1,n
middle: DO j = 1,m
inner:    DO k = 1,l
            .
            .
            .
          IF (a(i,j,k)<0) EXIT outer  ! Leave loops
          IF (j==5) CYCLE middle      ! Omit j==5 and set j=6
          IF (i==5) CYCLE             ! Skip rest of inner loop,
            .                         ! and go to next iteration
            .                         ! of inner loop
            .
            END DO inner
          END DO middle
        END DO outer
```

## 2.7.5 CASE Construct

Repeated IF ... THEN ... ELSE constructs can be replaced by a CASE construct, as can the 'computed GOTO'. The general form of the CASE construct is:

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

The expression can be of type INTEGER,LOGICAL, or CHARACTER, and 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.

```
SELECT CASE (day) ! sunday = 0, monday = 1, etc
  CASE (0)
    extrashift = .TRUE.
    CALL weekend
  CASE (6)
    extrashift = .FALSE.
    CALL weekend
  CASE DEFAULT
    extrashift = .FALSE.
    CALL weekday
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.

A colon may be used to specify a range, as in:

```
CASE ('a':'h','o':'z')
```

which will test for letters in the ranges `a` to `h` and `o` to `z`.

## 2.7.6 GOTO

The `GOTO` statement is still available, but, it is usually better to use `IF, DO,` and `CASE` constructs, and `EXIT` and `CYCLE` statements instead.

# 2.8 Exercises

**Derived Types:**

1. Run the program `vehicle.f90`. What difference do you notice in the output of the two `WRITE` statements?

2. Run the program `circle1.f90`. Create a new derived type for a rectangle and assign and write out the corners of the rectangle. (`rectdef.f90`)

3. Create a file `circle.dat` which contains the components of the centre and radius of a circle so that it can be read by program `circle2.f90`. Run the program.

4. Alter program `circle4.f90` so that it prints a circle centred at the origin (0,0) with radius 4.0.

5. Define a derived type that could be used to store a date of birth in the following type of format:

   15 May 1990

   Write a program to test your derived type in a similar manner to the above examples. (`birth1.f90`)

6. Modify the derived type in exercise 5 to include a component for a name. (`birth2.f90`).

**Control Structure:**

7. Write a program containing a `DO` construct which reads numbers from the data file `square.dat`, skips negative numbers, adds the square root of positive numbers, and concludes if the present number is zero (use `EXIT` and `CYCLE`). (`sq_sum.f90`)

8. Write a program that reads in a month number (between 1 and 12) and a year number. Use the `CASE` construct to assign the number of days for that month, taking leap years into account. (`no_days.f90`)

9. Write a program that reads in a character string. Use the `CASE` construct in converting upper case characters to lower case and vice versa, and write out the new string. (Use `IACHAR("a") - IACHAR("A")` to determine the difference in the position in the collation sequence between lower and upper case characters.) (`convert.f90`)

**Kind Values:**

10. Run the program `kind_int.f90`. Notice how this program uses `SELECTED_INT_KIND` to find the kind values for integer variables on this system. Modify this program to find the kind values for real variables. (`kind_rl.f90`)

11. Run the program `mc_int.f90`. Notice how this program uses the kind values of integer variables found in exercise 1, and the numeric intrinsic functions to find some of the machine constants for this system. Modify this program by using the kind values of real variables found in exercise 1 and the numeric intrinsic functions (`PRECISION`, `HUGE`, `TINY` and `RANGE`) to find some of the machine constants for this system. (`mc_real.f90`)

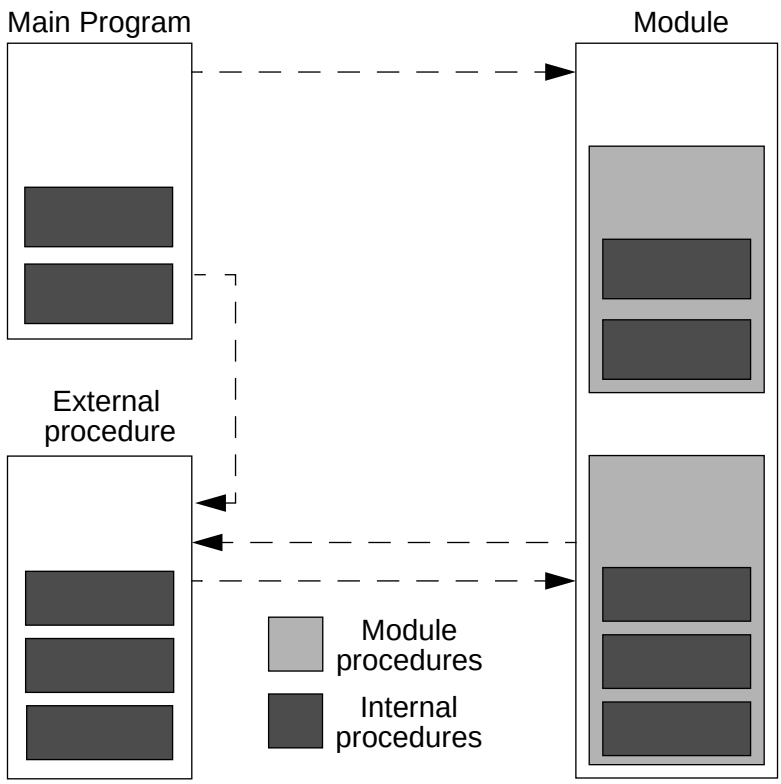
# 3 Procedures and Modules

## 3.1 Program Units

Fortran 90 consists of the main program unit and external procedures as in Fortran 77, and additionally introduces internal procedures and modules and module procedures. A program must contain exactly one main program unit and any number of other program units (modules or external procedures).

A module exists to make some or all of the entities declared within it accessible to more than one program unit. A subprogram which is contained within 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.

The following diagram illustrates the nesting of subprograms in program units:



The form of the program units and procedures is summarised below.

Main program:

```
[PROGRAM program_name]
  [specification-statements]
  [executable-statements]
[CONTAINS
  internal procedures]
END [PROGRAM [program_name]]
```

Module:

```
MODULE module_name
  [specification-statements]
  [executable-statements]
[CONTAINS
  module procedures]
END [MODULE [module_name]]
```

External procedures:

```
[RECURSIVE] SUBROUITNE subroutine_name(dummy-argument-list)
  [specification-statements]
  [executable-statements]
[CONTAINS
  internal procedures]
END [SUBROUTINE [subroutine-name]]
```

or

```
[type] [RECURSIVE] FUNCTION function_name &
(dummy-argument-list) [RESULT(result_name)]
  [specification-statements]
  [executable-statements]
[CONTAINS
  internal procedures]
END [FUNCTION [function-name]]
```

Module procedures have exactly the same form as external procedures except that the word `SUBROUTINE` or `FUNCTION` *must* be present on the `END` statement.

Internal procedures also *must* have the word `SUBROUTINE` or `FUNCTION` present on the `END` statement:

```
[RECURSIVE] SUBROUTINE subroutine_name(dummy-argument-list)
  [specification-statements]
  [executable-statements]
END SUBROUTINE [subroutine_name]

[type] [RECURSIVE] FUNCTION function_name &
(dummy-argument-list) [RESULT (result_name)]
  [specification-statements]
  [executable-statements]
END FUNCTION [function_name]
```

# 3.2 Procedures

Procedures may be *subroutines* or *functions*. Self-contained sub-tasks should be written as procedures. A function returns a single value and does not usually alter the values

of its arguments, whereas a subroutine can perform a more complicated task and return several results through its arguments.

Fortran 77 contained only external procedures, whereas in Fortran 90, structurally, procedures may be:

- Internal - inside a program unit.
- External - self contained (and not necessarily written in Fortran).
- Module - contained within a module.

An interface block is used to define the procedure argument details, and must always be used for external procedures.

## 3.2.1 Internal Procedures

Program units can contain internal procedures, which may NOT, however, contain further internal procedures. That is, nesting of internal procedures is not permitted.

The internal procedures are collected together at the end of the program unit and are preceded by a `CONTAINS` statement. For example,

```
PROGRAM main

  IMPLICIT NONE
  REAL :: a,b,c
    .
    .
    .
  mainsum=add()
    .
    .
    .
CONTAINS

  FUNCTION add()
    IMPLICIT NONE
    REAL :: add   !a,b,c,defined in 'main'
    add=a+b+c
  END FUNCTION add

END PROGRAM main
```

Variables defined in the program unit, remain defined in the internal procedure, unless redefined there. It is good practice to declare all variables used in subprograms in order to avoid the use of global variables in the wrong context.

`IMPLICIT NONE` in a program unit is also in effect in all internal procedures it `CONTAINS`. However, it is recommended that `IMPLICT NONE` is also included in all internal procedures for both clarity and avoidance of errors.

```
SUBROUTINE arithmetic(n,x,y,z)

  IMPLICIT NONE
  INTEGER :: n
  REAL,DIMENSION(100) :: x,y,z
     .
     .
     .
CONTAINS
```

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

END SUBROUTINE arithmetic
```

## 3.2.2 Interface Blocks

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. In the case of intrinsic subprograms, internal subprograms and modules, this information is always known by the compiler and is said to be explicit.

However, when the compiler calls an external subprogram, this information is not available and is said to be implicit. The Fortran 90 interface block provides a means of making this information available. The general form of the interface block is:

```
INTERFACE
  interface body
END INTERFACE
```

Note that, unlike other program unit `END` statements, the `END INTERFACE` statement *cannot* be named.

The *`interface body`* consists of the `FUNCTION` (or `SUBROUTINE`) statement, argument type declaration statements, and the `END FUNCTION` (or `END SUBROUTINE`) statement. In other words it is an exact copy of the subprogram without its executable statements or internal subprograms. For example,

```
INTERFACE
  REAL FUNCTION func(x)
    REAL,INTENT(IN) :: x    !INTENT is described in the next
  END FUNCTION func         !section
END INTERFACE
```

The interface block must be placed in the calling program unit. Note that an interface block can contain interfaces to more than one procedure.

## 3.2.3 INTENT

It is possible to specify whether a procedure argument is intended to be used for input, output, or both, using the `INTENT` attribute. For example,

```
INTEGER, INTENT(IN) :: x
REAL,INTENT(OUT) :: y
REAL, INTENT(INOUT) :: Z
```

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.

An Example

```
SUBROUTINE swapreal(a,b)
  IMPLICIT NONE
  REAL,INTENT(INOUT) :: a,b
```

```
  REAL :: temp
    temp = a
    a = b
    b = temp
END SUBROUTINE swapreal
```

This is used by:

```
CALL swapreal(x,y)
```

## 3.2.4 Keyword Arguments

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
```

When a procedure has several arguments, keywords are an excellent way of avoiding confusion between arguments. 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.

For example, we could have the following internal function:

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

which could be called by:

```
a=area(0.0,100.0,0.00001)

b=area(start=0.0,tol=0.00001,finish=100.0)

c=area(0.0,tol=0.00001,finish=100.0)
```

where `a`, `b` and `c` are variables declared as `REAL`. All arguments prior to the first keyword must match — once a keyword is used all the rest must use keywords. Hence it is not possible to say:

```
c=area(0.0,tol=0.00001,100.0) !not allowed
```

Note that an interface is not required in the above example as `area` is an internal function, and similarly one would not be required for a module subprogram with keyword arguments. This is because both have explicit interfaces. In the case of an external procedure with argument procedures, an interface *must* be provided.

## 3.2.5 Optional Arguments

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 is known as an 'optional' argument. An argument can be given this attribute by specifying it as `OPTIONAL` in the type declaration statement. For example,

```
REAL FUNCTION area(start,finish,tol)
```

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

This could be called by:

```
a=area(0.0,100.0,0.010)

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

c=area(0.0)

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

where `a`, `b`, `c` and `d` are variables declared as `REAL`. The intrinsic logical function `PRESENT` is used to check for the presence of an optional argument. For example, in the function example above it may be necessary to both check for the presence of the variable `tol`, and set a default if `tol` is absent. This is achieved as follows:

```
REAL :: ttol
IF (PRESENT(tol)) THEN
  ttol = tol
ELSE
  ttol = 0.01
END IF
```

The local variable `ttol` is used here as this may be redefined, wher eas the argument `tol` cannot be changed (as it is `INTENT(IN)`)

As in the case of keyword arguments, if the procedure is *external* and has any optional arguments, an interface *must* be supplied. Thus, if the function in the example above was external, the following interface block would need to be provided:

```
INTERFACE
  REAL FUNCTION area(start,finish,tol)
    REAL,INTENT(IN),OPTIONAL :: start, finish, tol
  END FUNCTION area
END INTERFACE
```

## 3.2.6 Derived Types as Procedure Arguments

Procedure arguments can be of derived type if the derived type is defined in only one place. This can be achieved in two ways:

1. the procedure is internal to the program unit in which the derived type is defined
2. the derived type is defined in a module which is accessible fr om the procedure.

## 3.2.7 Procedures as Arguments

Prior to Fortran 90, we would declare a procedure argument as `EXTERNAL`. In Fortran 90 the procedure that is passed as an argument must either be an external procedure or a module procedure. Internal procedures are not permitted.

If the argument procedure is an *external* procedure, you are recommended to supply an interface block in the calling program unit. For example, consider the external function `func`:

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

Suppose the subroutine `area` passes `func` as an argument, then the calling program unit would contain

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

## 3.2.8 RESULT Clause for Functions

Functions can have a `RESULT` variable. 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. For example,

```
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
```

Directly recursive functions, section 3.2.10, “Recursion”, must have a `RESULT` variable.

## 3.2.9 Array-valued Functions

A function’s result does not have to be scalar, it may alternatively be an array. Such a function is known as an array-valued function. The type of an array-valued function is not 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.

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

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

Note that if the array-valued function is external, an interface must be provided in the calling program.

```
INTERFACE
```

```
  FUNCTION add_vec (a,b,n)
    REAL, DIMENSION (n) :: add_vec
    INTEGER, INTENT(IN) :: n
    REAL, DIMENSION (n), INTENT(IN) :: a, b
  END FUNCTION add_vec
END INTERFACE
```

## 3.2.10 Recursion

It is possible for a procedure to invoke itself, either directly (i.e. the function name occurs on the right-hand side of a statement in the body of the function definition) or indirectly. This is known as recursion. For example,

- A calls B calls A (i.e. indirect), or
- A calls A directly.

This can be made possible by including the keyword `RECURSIVE` before the procedure's name in the first line of the pr ocedure. This applies to both subroutines and functions. A direct recursive function must also have a `RESULT` variable. This is necessary as the function name is already used within the body of the function as a result variable, and hence using it as a recursive reference to itself may cause ambiguities in some cases. Thus a `RESULT` variable is used, with a name different to the function itself, and then within the function, any reference to the actual function name is interpreted as a recursive call to the function.

The classic textbook example of a recursive function, is the factorial calculation:

```
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)
  ENDIF
END FUNCTION fact
```

An important application of recursive procedures is where we have a variable number of `DO` loops:

```
DO
  DO
    DO
      .
      .
      .
    END DO
  END DO
END DO
```

For example, suppose we want to write a program called ANOVA to analyse a general factorial design. At the time of writing the program we don't know how many factors there are. Even Fortran 90 doesn't allow us to declare arrays with a variable number of dimensions, and so it is usual for this problem to use a one-dimensional array and calculate the offset in the program. To calculate this offset we still seem to need a number of `DO` loops equal to the number of factors in the model.

Consider the sub-problem of reading in the initial data. (For reasons specific to the problem, the array needs to be of length

$$\prod_{i=1}^{factors} (level_i + 1)$$

where each factor will be represented at a specific number of levels.)

Fortran 90 allows us to code this as follows:

```
SUBROUTINE anova(factors,level,x, ... )
  INTEGER,INTENT(IN) :: factors
  INTEGER,DIMENSION(:),INTENT(IN) :: level
  REAL,DIMENSION(:),INTENT(OUT) :: x
    .
    .
    .
  INTEGER :: i,k,n,element
  INTEGER,DIMENSION(factors) :: c,istep
  n = factors + 1
  DO i=1,factors
    IF (i == 1) THEN
      istep(i) = 1
    ELSE
      istep(i) = istep(i-1) * (level(i-1) + 1)
    END IF
  END DO

  CALL data
    .
    .
    .
CONTAINS

  RECURSIVE SUBROUTINE data
    INTEGER :: cn
    n = n-1
    IF (n == 0) THEN
      element = 1
      DO k=1,factors
        element = element + (c(factors + 1 - k) - 1) * istep(k)
        read *,x(element)
      END DO
    ELSE
      DO cn=1,level(factors+1-n)
        c(n) = cn     ! do-variable must be a simple variable
        CALL data
      END DO
    END IF
    n = n + 1
  END SUBROUTINE data

END SUBROUTINE anova
```

## 3.2.11 Generic Procedures

A powerful new feature of Fortran 90 is the ability to define your own generic procedures so that a single procedure name may be used within a program, and the action taken when this name is used is dependent on the type of its arguments. This is also known as polymorphic typing. A generic procedure is defined using an interface

block and a generic name is used for all the procedures defined within that interface block. Thus the general form is:

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

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

For example, suppose we want a subroutine to swap two numbers whether they are both real or both integer. This would require two external subroutines:

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

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

This could be invoked with `CALL swap(x,y)`, provided there is an interface block:

```
INTERFACE swap
  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
```

# 3.3 Modules

A major new Fortran 90 feature is a new type of program unit called the module. The module is very powerful in communicating data between subprograms and in organising the overall architecture of a large program.

The module is important for both sharing data and sharing procedures (known as module procedures) between program units. Modules also provide a means of global access to entities such as derived type definitions and associated operators. Additionally, using the `PRIVATE` attribute, it is possible to limit access to entities in a module. A program may include several different modules, but they must all have a different names.

The form of a module is:

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

## 3.3.1 Global Data

In Fortran, variables are usually local entities. Using modules, it is possible for the same sets of data to be accessible by a number of different program units. For example, suppose we want to have access the integers `i, j, k` and the reals `a, b, c` in different procedures. Simply place the appropriate declaration statements in a module as follows:

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

Note the use of the `SAVE` attribute. This allows modules to be used to provide global data. This simple use of the module is a substitute for the `COMMON` block used previously in Fortran 77.

The data is made accessible in other program units by supplying the `USE` statement, i.e.

```
USE globals
```

The `USE` statement is non-executable, and must appear at the very beginning of a program unit before any other non-executables, 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.

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 can overcome this problem by allowing the specification of a different local name for data accessed from a module. For example,

```
USE globals, r=>a, s=>b
```

Here, `r` and `s` are used to refer to the module data items `a` and `b`, and so `a` and `b` can be used for something completely different within the program unit. The `=>` symbols link the local name with the module name.

There is also a form of the `USE` statement which limits access to certain items within the module. This requires the qualifier `ONLY` followed by a colon and an *only-list*. For example, only variables `a` and `c` can be accessed via the statement:

```
USE globals, ONLY : a,c
```

These two facilities can also be combined:

```
USE globals, ONLY : r=>a
```

A program unit may have more than one USE statement referring to the same module. However, note that a USE statement with ONLY does not cancel out a less restrictive USE statement.

## 3.3.2 Module Procedures

Procedures which are specified within modules are known as module procedures. These can be either subroutines or functions, and have the same form as external procedures except that they must be preceded by the CONTAINS statement, and the END statement must have a SUBROUTINE or FUNCTION specified. Note that, unlike external procedures, module procedures must be supplied in Fortran. There can be several module procedures contained in one module.

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 which contains 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.

Module procedures can be useful for several reasons. For example, a module which defines the structure of a particular set of data could also include special procedures needed to operate on the data, or a module could be used to hold a library of related procedures.

For example. a module can be used 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
```

The main program would contain:

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

## 3.3.3 Generic procedures

Modules allow arguments of derived type and hence generic procedures with derived types. Thus it is possible to extend the generic procedure swap introduced in section 3.2.11, "Generic Procedures" to swap two variables of derived type point.

```
MODULE genswap
  IMPLICIT NONE

  TYPE point
    REAL :: x, y
  END TYPE point

  INTERFACE swap
    MODULE PROCEDURE swapreal, swapint, swaplog, swappoint
  END INTERFACE

CONTAINS

  SUBROUTINE swappoint (a,b)
    IMPLICIT NONE
    TYPE (point), INTENT(INOUT) :: a, b
    TYPE (point) :: temp
    temp = a
    a = b
    b = temp
  END SUBROUTINE swappoint

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

  !similar subroutines for swapint and swaplog
  ...
END MODULE genswap
```

### 3.3.4 Private and Public Attributes

By default, all entities in a module are available to a program unit which includes the `USE` statement. Sometimes it is sensible to forbid the use of certain entities to the host program to force usage of the module routines rather than allow the user to take his own short-cuts, or to allow flexibility for internal change without the users needing to be informed or the documentation changed.

This is done by using the `PRIVATE` statement:

```
PRIVATE :: sub1, sub2
```

or, the `PRIVATE` attribute:

```
INTEGER,PRIVATE,SAVE :: currentrow,currentcol
```

## 3.4 Overloading

Fortran 90 allows operator and assignment overloading, and in these cases an interface block is required. Modules are often used to provide global access to assignment and operator overloading.

## 3.4.1 Overloading Intrinsic Operators

It is possible to extend the meaning of an intrinsic operator to apply to additional data types. This requires an interface block with the form:

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

For example, the '+' operator could be extended for character variables in order to concatenate two strings ignoring any trailing blanks, and this could be put in a module:

```
MODULE operator_overloading
  IMPLICIT NONE
  ...
  INTERFACE OPERATOR (+)
    MODULE PROCEDURE concat
  END INTERFACE
  ...
CONTAINS
  FUNCTION concat(cha,chb)
    IMPLICIT NONE
    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 in any program unit which '`USES`' this module.

Notice in this example the interface block. The procedure defining the operator is in a module and it is not necessary to have explicit interfaces for module procedures within the same module. An interface block is required, in this case, which provides a generic name or operator for a set of procedures and should be of the form:

```
INTERFACE ...
  MODULE PROCEDURE list
END INTERFACE
```

where *`list`* is a list of the names of the module procedures concerned.

## 3.4.2 Defining Operators

It is possible to define new operators. and this is particularly useful when using defined types. Such an operator must have a '.' at the beginning and end of the operator name. For example, in the preceding example `.plus.` could have been defined instead of using '+'. The operation needs to be defined via a function, which has one or two non-optional arguments with `INTENT(IN)`.

The following example shows the definition of an operator `.dist.` which calculates the straight line distance between two derived type 'points'. The operator has been defined within a module and so can be used by several pr ogram units.

```
MODULE distance_mod
  IMPLICIT NONE
  ...
  TYPE point
```

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

The calling program will include:

```
USE distance_mod
TYPE(point) :: px,py
  ...
distance = px .dist. py
```

The power of modules can be seen in the following example, as a way to define a derived type and all the associated operators:

```
MODULE moneytype
  IMPLICIT NONE

  TYPE money
    INTEGER :: pounds, pence
  END TYPE money

  INTERFACE OPERATOR (+)
    MODULE PROCEDURE addmoney
  END INTERFACE

  INTERFACE OPERATOR (-)
    MODULE PROCEDURE negatemoney, subtractmoney
  END INTERFACE

CONTAINS

  FUNCTION addmoney(a,b)
    IMPLICIT NONE
    TYPE (money) :: addmoney
    TYPE (money), INTENT(IN) :: a,b
    INTEGER :: carry, temppence
    temppence = a%pence + b%pence
    carry = 0
      IF (temppence>100) THEN
        temppence = temppence - 100
        carry = 1
      END IF
    addmoney%pounds = a%pounds + b%pounds + carry
    addmoney%pence = temppence
  END FUNCTION addmoney

  FUNCTION negatemoney(a)
```

```
    IMPLICIT NONE
    TYPE (money) :: negatemoney
    TYPE (money), INTENT(IN) :: a
    negatemoney%pounds = -a%pounds
    negatemoney%pence = -a%pence
  END FUNCTION negatemoney

  FUNCTION subtractmoney(a,b)
    IMPLICIT NONE
    TYPE (money) :: subtractmoney
    TYPE (money), INTENT(IN) :: a,b
    INTEGER :: temppound, temppence, carry
    temppence = a%pence - b%pence
    temppound = a%pounds - b%pounds

! IF construct to incorporate any carry required from subtraction
      IF ((temppence<0).AND.(temppound>0)) THEN
        temppence = 100 + temppence
        temppound = temppound - 1
      ELSE IF ((temppence>0).AND.(temppound<0)) THEN
        temppence = temppence - 100
        temppound = temppound + 1
      END IF

    subtractmoney%pence = temppence
    subtractmoney%pounds = temppound
  END FUNCTION subtractmoney

END MODULE moneytype
```

## 3.4.3 Assignment Overloading

It may be necessary to extend the meaning of assignment (=) when using derived types.

For example, suppose the variables `ax` and `px` are declared as follows:

```
REAL :: ax
TYPE (point) :: px
```

and within the program the following assignment is required

```
  ax = px
```

i.e type point is assigned to type real. Such an assignment is not valid until it has been defined.

Continuing with this example, suppose we require that `ax` takes the larger of the `x` and `y` components of `px`. This assignment needs to be defined via a subr outine with two non-optional arguments, the first having `INTENT(OUT)` or `INTENT(INOUT)`, the second having `INTENT(IN)` and an interface assignment block must be created.

The interface block required for assignment overloading is of the form

```
INTERFACE ASSIGNMENT (=)
  subroutine_interface_body
END INTERFACE
```

The assignment definition could be placed in a module, as follows

```
MODULE assignoverload_mod
```

```
  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_mod
```

The main program needs to invoke this module, with the `USE` statement, and the assignment type point to type real is now defined and can be used as required:

```
USE assignoverload_mod
REAL :: ax
TYPE (point) :: px
...
ax = px
```

# 3.5 Scope

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.

## 3.5.1 Labels

Every subprogram, internal or external, has its own independent set of labels. Thus the same label can be used in a main program and its internal subprograms without ambiguity. Therefore, the scope of a label is a main program or a procedure, excluding any internal procedures contained within it.

## 3.5.2 Names

The scope of a name declared in a program unit extends from the program unit's header to its `END` statement. The scope of a name declared in a main program or external subprogram extends to all the subprograms it contains, unless the name is redeclared in the subprogram.

The scope of a name declared in an internal subprogram is only the subprogram itself, not other internal subprograms. The scope of the name of an internal subprogram, and of its number and type of arguments, extends throughout the containing program unit, and therefore all other internal subprograms.

The scope of a name declared in a module extends to all program units which USE that module, unless the named entity has the PRIVATE attribute, or is renamed in the host program unit, or the USE statement has an ONLY qualifier and that named entity is not in the only-list. The scope of a name declared in a module extends to any internal subprograms, excluding those in which the name is redeclared.

Consider the *scoping unit* defined above:

- Entities declared in different scoping unit are always different, even if they have the same names and properties.
- 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 be distinct from the others and from any of the local entities of the program unit.
- The scope of the name of an internal procedure extends throughout the containing program unit only.
- The scope of a name declared in an internal procedure is that internal procedure.

Names are said to be accessible either by '*host association*' or '*use association*':

- Host association - The scope of a name declared in a program unit extends from the program unit's header to its END statement.
- Use association - The scope of a name declared in a module, which does not have the PRIVATE attribute, extends to any program units that USE the module.

Note that both associations do not extend to any external procedures that may be invoked, and do not include any internal procedures in which the name is redeclared

### 3.5.3 Example of Scoping Units

```
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
100   ...                    ! scope 4
    END INTERFACE            ! scope 3
    REAL x, y                ! scope 2
    ...                      ! 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
```

## 3.6 Program Structure

### 3.6.1 Order of Statements

Within this chapter, several new statements have been introduced. The following table summarises the order of statements in program units.

*Table 1: Order of Statements*

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

## 3.6.2 Interface Blocks

In this chapter an interface block has been required in several situations. In summary:

- An interface block is needed when a module or external procedure is called:
  - which defines or overloads an operator, or overloads assignment.
  - uses a generic name.
- An interface block is needed when an external procedure:
  - is called with a keyword and/or optional argument.
  - is an array-valued or pointer function, or a character function which is neither a constant nor assumed length.
  - has a dummy argument which is an assumed size array, a pointer or a target.
  - is a dummy or actual argument (in this case an interface block is recommended, not mandatory).

## 3.6.3 Summary

Using Fortran 77 it was only possible to use a main program unit calling external procedures, and the compiler had no means of checking for argument inconsistencies between the procedures. In simple terms, Fortran 90 provides internal procedures with an explicit interface allowing the compiler to check for any argument inconsistencies.

Structured Fortran 90 programs will consist of a main program and modules containing specifications, interfaces and procedures - external procedures no longer being required. The introduction of many new features such as derived types, overloading, internal subprograms and modules make possible the creation of sophisticated Fortran 90 code.

# 3.7 Exercises

1. Write a program that calls a function to sum all of the integers between min and max. Set min and max to be optional keyword arguments which default to 1 and 10 respectively. (`opt_par.f90`)

2. Look at program `err_main.f90` and `err_sub.f90`. Compile and run. What is wrong? Rewrite it in a better way in Fortran 90. (`err_sol.f90`)

3. Write a recursive function to calculate the nth value of the Fibonacci sequence. Notice that fib(1)=1, fib(2)=1, fib(i)=fib(i-1)+fib(i-2) i.e. 1, 1, 2, 3, 5, 8, 13, ... (`fibon.f90`)

4. Write a program which defines a generic function to return the maximum absolute value of two variables, for real, integer and complex variable types. (`maxabs.f90`)

5. Write a module which defines kind values for single and double precision real variables, and a main program which uses this module and can be changed from single to double precision by changing a single value. (`prec.f90, prec_mod.f90`)

6. Look at the program `generic.f90`. Modify this program to include a function for swapping two variables of type (`point`) by using a module, where '`point`' is defined with two real variables. (`gen_mod.f90, swap_mod.f90`)

7. Look at the program `money.f90`. From these code fragments, construct a module that allows you to run the program `mon_main.f90`. (`mon_main.f90, mon_mod.f90`)

8. Write a module which defines a vector type with x and y components and the associated operators '+' and '-' overloading, and a main program which uses this module to apply all associated operators overloading to the variables of derived type vector. (`vec_main.f90, vec_mod.f90`)

# 4 Array Processing

A major new feature of Fortran 90 are the array processing capabilities. It is possible to work directly with a whole array or an array section without explicit `DO`-loops. Intrinsic functions can now act elementally on arrays, and functions can be array-valued. Also available are the possibilities of allocatable arrays, assumed shape arrays, and dynamic arrays. These and other new features will be described in this chapter, but first of all it is necessary to introduce some terminology.

## 4.1 Terminology and Specifications

Fortran permits an array to have up to seven subscripts, each of which relates to one dimension of the array. The dimensions of an array may be specified using either a dimension attribute or an array specification. By default the array indices start at 1, but a different range of values may be specified by providing a lower bound and an upper bound. For example,

```
REAL, DIMENSION(50) :: w
REAL, DIMENSION(5:54) :: x
REAL y(50)
REAL z(11:60)
```

Here, w, x, y and z are all arrays containing 50 elements.

The *rank* of an array is the number of dimensions. Thus, a scalar has rank 0, a vector has rank 1 and a matrix has rank 2.

The *extent* refers to a particular dimension, and is the number of elements in that dimension.

The *shape* of an array is a vector consisting of the extent of each dimension.

The *size* of an array is the total number of elements which make up the array. This may be zero.

Two arrays are said to be *conformable* if they have the same shape. All arrays are conformable with a scalar, as the scalar is broadcast to an array with the same shape.

Take, for example the following arrays:

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

The array a has

- rank 2
- extents 8 and 7
- shape (/8,7/)

- size 56

Also, a is conformable with b and c, as b has shape (/8,7/) and c is scalar. However, a is not conformable with d, as d has shape (/8,9/). Notice the use of array constructors to create the shape vectors - this will be explained later in section 4.10, "Array Constructors".

The general form of an array specification is as follows:

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

This is simply a special case of the form of declaration given in section 2.3, "Specifications".

Here, *type* can be any intrinsic type or a derived type (so long as the derived type definition is accessible to the program unit declaring the array). DIMENSION is optional and defines default dimensions in the *extent-list*, these can alternatively by defined in the *entity list*.

The *extent-list* gives the array dimensions as:

- integer constants
- integer expressions using dummy arguments or constants
- ':' to show the array is allocatable or assumed shape

As before, *attribute* can be any one of the following

```
PARAMETER
PUBLIC
PRIVATE
POINTER
TARGET
ALLOCATABLE
DIMENSION(extent-list)
INTENT(inout)
OPTIONAL
SAVE
EXTERNAL
INTRINSIC
```

Finally, the *entity list* is a list of array names with optional dimensions and initial values.

The following examples show the form of the declaration of several kinds of arrays, some of which are new to Fortran 90 and will be met later in this chapter:

1. Initialisation of one-dimensional arrays containing 3 elements:

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

2. Declaration of automatic array logb. Here loga is a dummy array argument, and SIZE is an intrinsic function which returns a scalar default integer corresponding to the size of the array loga:

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

3. Declaration of 2D dynamic (allocatable) arrays a and b. The shape would be defined in a subsequent ALLOCATE statement:

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

4. Declaration of 3D assumed shape arrays a and b. The shape would be taken from the actual calling routine:

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

# 4.2 Whole Array Operations

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.

In order for whole array operations to be performed, the arrays concerned *must* be conformable. Remember, that for two arrays to be conformable they must have the same shape, and any array is conformable with a scalar. Operations between two conformable arrays are carried out on an element by element basis, and all intrinsic operators are defined between two such arrays.

For example, if a and b are both 2x3 arrays

$$a = \begin{bmatrix} 3 & 4 & 8 \\ 5 & 6 & 6 \end{bmatrix}, \quad b = \begin{bmatrix} 5 & 2 & 1 \\ 3 & 3 & 1 \end{bmatrix}$$

the result of addition is

$$a + b = \begin{bmatrix} 8 & 6 & 9 \\ 8 & 9 & 7 \end{bmatrix}$$

and of multiplication is

$$a \times b = \begin{bmatrix} 15 & 8 & 8 \\ 15 & 18 & 6 \end{bmatrix}$$

If one of the operands is a scalar, then the scalar is broadcast into an array which is conformable with the other operand. Thus, the result of adding 5 to b is

$$b + 5 = \begin{bmatrix} 5 & 2 & 1 \\ 3 & 3 & 1 \end{bmatrix} + \begin{bmatrix} 5 & 5 & 5 \\ 5 & 5 & 5 \end{bmatrix} = \begin{bmatrix} 10 & 7 & 6 \\ 8 & 8 & 6 \end{bmatrix}$$

Such broadcasting of scalars is useful when initialising arrays and scaling arrays.

An important concept regarding array-valued assignment is that the right hand side evaluation is computed before any assignment takes place. This is of relevance when an array appears in both the left and right hand side of an assignment. If this were not the case, then elements in the right hand side array may be affected before the operation was complete.

The advantage of whole array processing can best be seen by comparing examples of Fortran 77 and Fortran 90 code:

1. Consider three one-dimensional arrays all of the same length. Assign all the elements of a to be zero, then perform the assignment `a(i) = a(i)/3.1 + b(i)*SQRT(c(i))` for all i.

   **Fortran 77 Solution**

```
REAL a(20), b(20), c(20)
```

```
      ...
      DO 10 i=1,20
         a(i)=0
10    CONTINUE

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

**Fortran 90 Solution**

```
REAL, DIMENSION(20) :: a, b, c
...
a=0
...
a=a/3.1+b*SQRT(c)
```

Note, the intrinsic function `SQRT` operates on each element of the array `c`.

**2.** Consider three two-dimensional arrays of the same shape. Multiply two of the arrays element by element and assign the result to the third array.

**Fortran 77 Solution**

```
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 Solution**

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

**3.** Consider a three-dimensional array. Find the maximum value less than 1000 in this array.
In Fortran 77 this requires triple `DO` loop and `IF` statements, whereas the Fortran 90 code is:

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

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 this example `amax` is the value of the maximum element in `a` which is less than 1000.

**4.** Find the average value greater than 3000 in an array.
In Fortran 77 this requires DO loops and IF statements, whereas Fortran 90 code is:

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

Note in the last two examples the use of the following array intrinsic functions:

`MAXVAL` - returns the maximum array element value.
`SUM` - returns the sum of the array elements.
`COUNT` - returns the number of true array elements.

# 4.3 Elemental Intrinsic Procedures

Fortran 90 also allows whole array elemental intrinsic procedures. That is, arrays may be used as arguments to intrinsic procedures in the same way that scalars are. The intrinsic procedure will be applied to each element in the array separately, but again arrays must be conformable.

The following are examples of elemental intrinsic procedures:

1. Find the square roots of all elements of an array, a. (Note that the SQRT function has already been seen in an example in section 4.2, "Whole Array Operations".)

```
b=SQRT(a)
```

2. Find the string length excluding trailing blanks for all elements of a character array ch.

```
length=LEN_TRIM(ch)
```

# 4.4 WHERE Statement

The `WHERE` statement can be used to perform assignment only if a logical condition is true and this is useful to perform an array operation on only certain elements of an array.

A simple example is to avoid division by zero:

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

The general form is

```
WHERE(logical-array-expression) array-variable=array-expression
```

The *`logical-array-expression`* is evaluated, and all those elements of *`array-expression`* which have value true are evaluated and assigned to *`array-variable`*. The elements which have value false remain unchanged. Note that the *`logical-array-expression`* must have the same shape as the array variable.

It is also possible for one logical array expression to determine a number of array assignments. The form of this `WHERE` construct is:

```
WHERE (logical-array-expression)
  array-assignment-statements
END WHERE
```

or

```
WHERE (logical-array-expression)
  array-assignment-statements
ELSEWHERE
```

```
  array-assignment-statements
END WHERE
```

In the latter form, the assignments after the `ELSEWHERE` statement are performed on those elements that have the value false for the logical array expression.

For example, the `WHERE` construct can be used to divide every element of the array `ra` by the corresponding element of the array `rb` avoiding division by zero, and assigning zero to those values of `ra` corresponding to zero values of `rb`.

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

# 4.5 Array Sections

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

Array sections can be extracted using either:

- A simple subscript.
- A subscript triplet.
- A vector subscript.

## 4.5.1 Simple Subscripts

A simple subscript extracts a single array element. Consider a 5x5 array, then `ra(2,2)` is a simple subscript:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & X & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} = \text{ra(2,2)}$$

## 4.5.2 Subscript Triplets

The form of a subscript triplet is:

```
[lower bound]:[upper bound][:stride]
```

If either the lower bound or upper bound is omitted, then the bound of the array from which the array section is extracted is assumed, and if stride is omitted the default stride=1 is used.

The following examples show various array sections of an array using subscript triplets. The elements marked with x denote the array section. Let the defined array from which the array section is extracted be a 5x5 array.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & X & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$ = `ra(2:2,2:2);` Array element, shape (/1/)

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & X & X & X \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$ = `ra(3,3:5);` Sub-row, shape(/3/)

$$\begin{bmatrix} 0 & 0 & X & 0 & 0 \\ 0 & 0 & X & 0 & 0 \\ 0 & 0 & X & 0 & 0 \\ 0 & 0 & X & 0 & 0 \\ 0 & 0 & X & 0 & 0 \end{bmatrix}$$ = `ra(:,3);` Whole column, shape(/5/)

$$\begin{bmatrix} 0 & X & X & X & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & X & X & X & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & X & X & X & 0 \end{bmatrix}$$ = `ra(1::2,2:4);` Stride 2 in rows, shape(/3,3/)

## 4.5.3 Vector Subscripts

A vector subscript is an integer expression of rank 1. Each element of this expression must be defined with values that lie within the par ent array subscript bounds. The elements of a vector subscript may be in any order.

An example of an integer expression of rank 1 is:

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

An example showing the use of a vector subscript `iv` is:

```
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 /)
```

Note that the vector subscript can be on the left hand side of an expression:

```
iv = (/1, 3, 5/)      ! vector subscript
ra(iv) = (/1.2, 3.4, 5.6/)
! = ra((/1, 3, 5/)) = (/1.2, 3.4, 5.6/)
! = ra(1:5:2) = (/1.2, 3.4, 5.6/)
```

It is also possible to use the same subscript more than once and hence using a vector subscript an array section that is bigger than the parent array can be constructed. Such a section is called a many-one array section. A many-one section cannot appear on the left hand side of an assignment statement or as an input item in a `READ` statement, as such uses would be ambiguous.

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

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

# 4.6 Array Assignment

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

```
REAL, DIMENSION(5,5) :: ra,rb,rc
INTEGER :: id
  .
  .
  .
! Shape(/5,5/) and scalar
ra = rb + rc*id

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

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

# 4.7 Recursion

It is important to be aware of how to achieve recursion in Fortran 90:

For example, the code:

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

is not the same as:

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

In the first case, the assignment is:

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

whereas in the second case the assignment is:

```
x(i) = x(i) + x(i-1)
```

In order to achieve the recursive effect of the `DO`-loop, in Fortran 90 it would be appropriate to use the intrinsic function `SUM`. This function returns the sum of all the elements of its array argument. Thus the equivalent assignment is:

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

# 4.8 Element Location Versus Subscript

The two array location intrinsics `MAXLOC` and `MINLOC` return the location of the maximum and minimum element of the array argument respectively. When arrays have

been defined with lower bounds not equal to 1, it is important to be aware that these intrinsics return the element location and *not* the element subscript. This can be seen in the following example:

```
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 location of max value

locmax1 = MAXLOC(ra)      ! = (/ 4 /)
locmax2 = MAXLOC(rb)      ! = (/ 4 /)

! To find maximum value from location

max1 = ra(locmax(1))
! OK because ra defined with 1 as lower bound

max2 = rb(LBOUND(rb) + locmax2(1) - 1)
! general form required with lower bound not equal to 1
```

# 4.9 Zero Sized Arrays

If the lower bound of an array dimension is greater than the upper bound, then the array has zero size. Zero sized arrays follow the normal array rules, and in particular zero sized arrays must be conformable to be used as operands.

Zero sized arrays are useful for boundary operations. For example,

```
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
```

# 4.10 Array Constructors

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. The general form is

```
(/ array-constructor-value-list /)
```

For example,

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

where, for example, *array-constructer-value-list* can be any of:

```
(/(i,i=1,6)/)
! = (/1,2,3,4,5,6/)

(/7,(i,i=1,4),9/)
! = (/7,1,2,3,4,9/)

(/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)/)
! = (/((1+j,2+j,3+j),j=1,2)/)
! = (/2,3,4,3,4,5/)

(/a(i,2:4),a(1:5:2,i+3)/)
! = (/a(i,2),a(i,3),a(i,4),a(1,i+3),a(3,i+3),a(5,i+3)/)
```

It is possible to transfer a rank-one array of values to an array of a different shape using the RESHAPE function. The RESHAPE function has the form

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

where 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.

A simple example is:

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

which creates ra= $\begin{bmatrix} 2 & 3 \\ 3 & 4 \\ 4 & 5 \end{bmatrix}$

If the argument ORDER is included as follows

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

then the result would be ra= $\begin{bmatrix} 2 & 3 \\ 4 & 3 \\ 4 & 5 \end{bmatrix}$

# 4.11 Allocatable Arrays

A major new feature of Fortran 90 is the ability to declare dynamic variables, in particular dynamic arrays. Fortran 90 provides allocatable and automatic arrays, both of which are dynamic. Using allocatable arrays, which are discussed in this section, it is possible to allocate and deallocate storage as required. Automatic arrays allow local arrays in a procedure to have a different size and shape every time the procedure is invoked. These are explained in more detail in section 4.12, "Automatic Arrays"

Allocatable arrays allow large chunks of memory to be used only when required and then be released. This produces a much more efficient use of memory than Fortran 77, which offered only static (fixed) memory allocation.

An allocatable array is declared in a type declaration statement with the attribute ALLOCATABLE. The rank of the array must also be specified in the declaration statement and this can be done by including the appropriate number of colons in the DIMENSION attribute. For example, a two dimensional array could be declared as:

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

This form of declaration statement does not allocate any memory space to the array. Space is dynamically allocated later in the program, when the array is required, using the ALLOCATE statement. The ALLOCATE specifies the bounds of the array and, as with any array allocation, the lower bound defaults to one if only the upper bound is specified. For example, the array declared above could be allocated with lower bound zero:

```
ALLOCATE (a(0:n,m))
```

The bounds may also be integer expressions, for example:

```
ALLOCATE (a(0:n+1,m))
```

The space allocated to the array with the ALLOCATE statement can later be released with the DEALLOCATE statement. The DEALLOCATE statement requires only the name of the array concerned and not the shape. For example:

```
DEALLOCATE (a)
```

Both the ALLOCATE and DEALLOCATE statement have an optional specifier STAT. The general form of the statements is:

```
ALLOCATE(allocate-object-list [,STAT=checkstat])
DEALLOCATE(allocate-object-list [,STAT=checkstat])
```

where *checkstat* is a scalar integer variable. If STAT= is present, *checkstat* is given the value zero if ALLOCATION/DEALLOCATION was successful, or a positive value if there was an error. If STAT= is not present and ALLOCATION/DEALLOCATION was unsuccessful, then program execution aborts.

Allocatable arrays make possible the frequent requirement to declare an array having a variable number of elements. For example, it may be necessary to read variables, say nsize1 and nsize2, and then declare an array to have nsize1 x nsize2 elements:

```
INTEGER n
REAL, DIMENSION(:,:), ALLOCATABLE :: ra
INTEGER :: checkstat
  ...
READ(*,*) nsize1,nsize2
ALLLOCATE (ra(nsize1,nsize2), STAT = checkstat)
IF (checkstat > 0) THEN
! ... error processing code ...
END IF
  ...
DEALLOCATE (ra)
```

Note that both ALLOCATE and DEALLOCATE statements can allocate/deallocate several arrays in one single statement.

An allocatable array is said to have an allocation status. When an array has been defined but not allocated the status is said to be *unallocated* or *not currently allocated*. When an array appears in an ALLOCATE statement then the array is said to be *allocated*, and once the array has been deallocated it is said to be *not currently 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. Thus, ALLOCATE can only be used on a previously allocated array if it has been deallocated first.

It is possible to check whether or not an array is currently allocated using the intrinsic function ALLOCATED. This is a logical function with one argument, which must be the name of an allocatable array. Using this function, statements like the following are possible:

```
IF (ALLOCATED(a)) DEALLOCATE(a)

or

IF (.NOT.ALLOCATED(a)) ALLOCATE(a(5,20))
```

An allocatable array has a third allocation status, *undefined*. An array is said to be undefined if it is allocated within a pr ocedure and the program returns to the calling program without deallocating it. Once an array is undefined, it can no longer be used. Hence it is good programming practice to deallocate all arrays that have been allocated. There are, however, two other ways around this problem. Firstly, an allocatable array can be declared with the SAVE attribute:

```
REAL, DIMENSION(:), ALLOCATABLE, SAVE :: a
```

This permits the allocatable array to remain allocated upon exit from the procedure and preserves the current values of the array elements. Secondly, the allocatable arrays could be put into modules, and in this case the arrays are preserved as long as the executing program unit uses the modules. The array can also be ALLOCATED and DEALLOCATED by any program unit using the module which the array was declared in.

Finally, there are three restrictions on the use of allocatable arrays:

- Allocatable arrays cannot be dummy arguments of a procedure and must, therefore, be allocated and deallocated in the same program unit
- The result of a function cannot be an allocatable array
- Allocatable arrays cannot be used in a derived type definition

# 4.12 Automatic Arrays

Automatic arrays are explicit-shape arrays within a procedure, which are not dummy arguments. Some, or all, of the bounds of automatic arrays are provided when the procedure is invoked. The bounds can depend on dummy arguments, or on variables defined by use or host association. Note that 'use association' is wher e 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.

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 in different procedure calls.

Note that Fortran 90 provides *no* mechanism for checking that there is sufficient memory for automatic arrays. If there is not, the program execution aborts.

The intrinsic function SIZE is often used when declaring automatic arrays. SIZE has the form:

```
SIZE(ARRAY [,DIM])
```

This returns the extent of ARRAY along dimension DIM, or returns the size of ARRAY if DIM is absent.

Note that an automatic array must not appear in a SAVE or NAMELIST statement, nor be initialised in a type declaration.

The following example shows the automatic arrays, work1 and work2 which take their size from the dummy arguments n and a:

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

The next example shows automatic array bounds dependent on a global variable defined in a module. Both use association and host association ar e shown:

```
MODULE auto_mod
  IMPLICIT NONE
  INTEGER :: n=1        ! set default n=1
CONTAINS
  SUBROUTINE sub
    IMPLICIT NONE
    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
  INTEGER :: n
  n = 10
  CALL sub
END PROGRAM auto_arrays
```

In the example below the power of dynamic arrays can be seen when passing only part of an array to a subroutine. Suppose the main program declares a nxn array, but the subroutine requires a n1xn1 section of this array a. In order to achieve this in Fortran 77, both parameters n and n1 must be passed as subroutine arguments:

```
PROGRAM array
  INTEGER n,n1
  PARAMETER (n=10)
  REAL a(n,n),work(n,n)
  REAL res
  ...
  READ(*,*) n1
  if (n1 .LE. n) then
    CALL sub(a,n,n1,res,work)
  else
c       ... error processing code ...
  end if
  ...
END PROGRAM array


SUBROUTINE sub(a,n,n1,res,work)
```

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

Note the use of a work array, which is passed as an argument, in the above example. The use of temporary work arrays is frequently necessary, particularly in numerical analysis. In Fortran 77, this presented serious problems for providers of subroutine libraries, who had to resort to requiring the calling sequence to include the work arrays along with the genuine parameters. The parameter list was further lengthened by the need to pass information about the dimensions of an array. Using dynamic arrays in Fortran 90, this can be achieved with much simplified argument passing:

```
PROGRAM array
  IMPLICIT NONE
  REAL, ALLOCATABLE, DIMENSION(:,:) :: a
  REAL :: res
  INTEGER :: n1,alloc_stat
  ...
  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,STAT=alloc_stat)
  IF (alloc_stat /= 0) THEN
    ! ... error processing code ...
  END IF
  ...
CONTAINS

  SUBROUTINE sub(a,n1,res)
    IMPLICIT NONE
    INTEGER, INTENT(IN) :: n1
    REAL, INTENT(INOUT) :: res
    REAL, DIMENSION(n1,n1), INTENT(IN) :: a
    REAL, DIMENSION(n1,n1):: work
    ...
    res=a(...)
    ...
  END SUBROUTINE sub
END PROGRAM array
```

Notice that using an allocatable array `a`, the array is exactly the size we require in the main program and so we can pass this easily to the subroutine. The work array, `work`, is an automatic array whose bounds depend on the dummy argument `n1`.

# 4.13 Assumed Shape Arrays

An assumed shape array is an array whose shape is not known, but which takes on whatever shape is imposed by the actual argument. When declaring an assumed shape array, each dimension is specified as:

```
[lower_bound]:
```

where the lower bound defaults to 1 if omitted.

Assumed shape arrays make possible the passing of arrays between program units without having to pass the dimensions as arguments. However, if an external procedure has an assumed shape array as a dummy argument, then an interface block must be provided in the calling program unit.

For example, consider the following external subprogram with assumed shape arrays `ra`, `rb` and `rc` (note that the shapes given are relevant only to this example):

```
SUBROUTINE sub(ra,rb,rc)
  IMPLICIT NONE
  REAL, DIMENSION(:,:), INTENT(IN) :: ra    ! Shape (10, 10)
  REAL, DIMENSION(:,:), INTENT(IN) :: rb    ! Shape (5, 5)
  ! = REAL, DIMENSION(1:5,1:5) :: rb
  REAL, DIMENSION(0:,2:), INTENT(OUT) :: rc ! Shape (5, 5)
  ! = REAL, DIMENSION (0:4,2:6) :: rc
  .
  .
  .
END SUBROUTINE sub
```

The calling program might include:

```
REAL, DIMENSION (0:9,10) :: ra        ! Shape (10, 10)

INTERFACE
    SUBROUTINE sub(ra,rb,rc)
      REAL, DIMENSION(:,:), INTENT(IN) :: ra,rb
      REAL, DIMENSION(0:,2:), INTENT(OUT) :: rc
    END SUBROUTINE sub
END INTERFACE
  .
  .
  .
CALL SUB (ra,ra(0:4,2:6),ra(0:4,2:6))
```

The following example uses allocatable, automatic and assumed shape arrays, and shows another method of coding the final example in section 4.12, "Automatic Arrays":

```
PROGRAM array
  IMPLICIT NONE
  REAL, ALLOCATABLE, DIMENSION(:,:) :: a
  REAL :: res
  INTEGER :: n1

  INTERFACE
    SUBROUTINE sub(a,res)
      REAL, DIMENSION(:, :), INTENT(IN) :: a
      REAL, DIMENSION(SIZE(a, 1),SIZE(a, 2)) :: work
    END SUBROUTINE sub
  END INTERFACE
  ...
  READ (*, *) n1
  ALLOCATE (a(n1, n1))          ! allocatable array
  CALL sub(a,res)
  ...
CONTAINS

  SUBROUTINE sub(a,res)
    IMPLICIT NONE
```

```
      REAL, INTENT(OUT) :: res
      REAL, DIMENSION(:, :), INTENT(IN) :: a! assumed shape array
      REAL, DIMENSION (SIZE(a, 1),SIZE(a, 2)) :: work
                                              ! automatic array
      ...
      res = a(...)
      ...
    END SUBROUTINE sub
  END PROGRAM array
```

# 4.14 Array Intrinsics

**Reduction**

`ALL(MASK[,DIM])`
True if all elements true

`ANY(MASK[,DIM])`
True if any element true

`COUNT(MASK[,DIM])`
Number of true elements

`MAXVAL(ARRAY[,DIM][,MASK])`
Maximum element value

`MINVAL(ARRAY[,DIM][,MASK])`
Minimum element value

`PRODUCT(ARRAY[,DIM][,MASK])`
Product of array elements

`SUM(ARRAY[,DIM][,MASK])`
Sum of array elements

**Inquiry**

`ALLOCATED(ARRAY)`
True if array allocated

`LBOUND(ARRAY[,DIM])`
Lower bounds of array

`SHAPE(SOURCE)`
Shape of array (or scalar)

`SIZE(ARRAY[,DIM])`
Size of array

`UBOUND(ARRAY[,DIM])`
Upper bounds of array

**Construction**

`MERGE(TSOURCE,FSOURCE,MASK)`
Merge arrays subject to mask

`PACK(ARRAY,MASK[,VECTOR])`
Pack elements into vector subject to mask

`SPREAD(SOURCE,DIM,NCOPIES)`
Construct an array by duplicating an array section

`UNPACK(VECTOR,MASK,FIELD)`
Unpack elements of vector subject to mask

**Reshape**

`RESHAPE(SOURCE,SHAPE[,PAD][,ORDER])`
Reshape array

**Array Location**

`MAXLOC(ARRAY[,MASK])`
Location of maximum element

`MINLOC(ARRAY[,MASK])`
Location of minimum element

**Array manipulation**

`CSHIFT(ARRAY,SHIFT[,DIM])`
Perform circular shift

`EOSHIFT(ARRAY,SHIFT[,BOUNDARY][,DIM])`
Perform end-off shift

`TRANSPOSE(MATRIX)`
Transpose matrix

**Vector and matrix arithmetic**

`DOT_PRODUCT(VECTOR_A,VECTOR_B)`
Compute dot product

`MATMUL(MATRIX_A,MATRIX_B)`
Matrix multiplication

The following example shows the use of several intrinsic functions:

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

```
                    85 76 90 60
score(1:3,1:4) =    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 /)
```

- 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
```

- 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.
```

# 4.15 Array Example

The following example shows the use of arrays in the 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

  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
```

# 4.16 Exercises

1. Run the program `matrix.f90` which declares a 2-dimensional integer array, with extents (n,n), where n is set to 9 in a `PARAMETER` statement.

   This program uses `DO` loops to assign elements of the array to have values r c, where r is the row number and c is the column number, e.g., a(3,2) = 32, a(5,7) = 57. It writes the resulting array to the file `matrix.dat` for later use.

   ```
   11 12 13 14 15 16 17 18 19
   21 22 23 24 25 26 27 28 29
   31 32 33 34 35 36 37 38 39
   41 42 43 44 45 46 47 48 49
   51 52 53 54 55 56 57 58 59
   61 62 63 64 65 66 67 68 69
   71 72 73 74 75 76 77 78 79
   81 82 83 84 85 86 87 88 89
   91 92 93 94 95 96 97 98 99
   ```

2. From the array constructed in exercise 1, use array sections to write out:
   (a) the first row
   (b) the fifth column
   (c) every second element of each row and column, columnwise

   ```
   11 31 51 71 91
   13 33 53 73 93
   15 35 55 75 95
   17 37 57 77 97
   19 39 59 79 99
   ```

   (d) every second element of each row and column, rowwise

   ```
   11 13 15 17 19
   31 33 35 37 39
   51 53 55 57 59
   71 73 75 77 79
   91 93 95 97 99
   ```

   (e) the 3 non-overlapping 3x3 sub-matrices in columns 4 to 6 (`section.f90`)

   ```
   14 15 16      44 45 46      74 75 76
   24 25 26      54 55 56      84 85 86
   34 35 36      64 65 66      94 95 96
   ```

3. Write a program which generates an 8x8 chequerboard, with 'B' and 'W' in alternate positions. Assume the first position is 'B'. ( `board.f90`)

4. From the array constructed in exercise 1, use the `WHERE` construct to create an array containing all of the odd values and 0 elsewhere (use elemental function, `MOD`). (`where.f90`)

5. Declare a vector subscript, iv, with extent 5. From the array constructed in exercise 1 create a 9x5 array containing only the odd values. (`vec_subs.f90`)

6. Generate the array constructed in exercise 1 using a single array constructor. (`reshape.f90`)

7. Look at the Fortran 77 code `sum2.f90`. Rewrite it using Fortran 90 with allocatable and assumed-shape arrays. (`sum4.f90`)

   Is there any instrinsic function which can simplify the same job? (`sum5.f90`)

8. Create an integer array whose size is allocated dynamically (read size from terminal). Assign odd and even values to the array (same as `matrix.f90`). Pass the array to a subroutine which uses an assumed shape argument and returns all odd values of the array and 0 elsewhere.
   (`odd_val.f90`)

9. Run the program `spread1.f90`. Modify it to create an real array with element values 1.0/REAL(i+j+1), where i is the row number and j is the column number. (`spread2.f90`)

   Can you find another way using Fortran 90 array?

10. Look at the program `m_basis.f90`. Modify it to select all values greater than 3000 and find the number of them, the maximum, the minimum and the average. (`munro.f90`)

# 5 Pointer Variables

## 5.1 What is a Pointer

A *pointer variable*, or simply a *pointer*, has the `POINTER` attribute, and may point to (be an alias of) another data object of the same type, which has the `TARGET` attribute, or an area of dynamically allocated memory.

The introduction of pointer variables brings Fortran 90 into the league of languages like Pascal and C. But they are quite different from, for example, pointers in C. In Fortran 90, a pointer variable does not contain any data itself and should not be thought of as an address. Instead, it should be thought of as a variable associated dynamically with or aliased to another data object where the data is actually stored - the target.

The use of pointers provides several benefits, of which the two most important are:

- The ability to provide a more flexible alternative to allocatable arrays.
- 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.

## 5.2 Specifications

The general forms for a pointer type and a target type declaration statements are

```
type [[,attribute]...] POINTER :: list of pointer variables
type [[,attribute]...] TARGET :: list of target variables
```

where

- the *type* specifier specifies what type of data object can be pointed to, which includes intrinsic types as well as derived types,
- the *attribute* list gives the other attributes (if any) of the data type.

A pointer variable must have the same type, type parameter and rank as its target variable. 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.

Thus, the statement

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

declares a pointer, `p`, which can point to any rank one, default-real array.

But, the statement

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

is an illegal statement, which is not allowed.

# 5.3 Pointer Assignments

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
```

where *pointer* is a variable with the pointer attribute and *target* is a variable which has either the target attribute or the pointer attribute.

Once a pointer is set up as an alias of a target, its use in a situation where a *value* is expected (for example, as one of the operands of an operator) is as if it were the associated target, i.e., the object being pointed to.

The following code and figure illustrate some pointer assignment statements and their effects:

```
REAL, POINTER :: p1, p2
REAL, TARGET :: t1 = 3.4, t2 = 4.5
p1 => t1              ! p1 points to t1
p2 => t2              ! p2 points to t2
p2 => p1              ! p2 points to the target of p1
```

The first line here declares two variables `p1` and `p2` to be pointers to areas of memory able to store real variables. The second line declares `t1` and `t2` to be real variables and specifies that they might be targets of pointers.

The next two pointer assignment statements make `p1` points to `t1` and `p2` point to `t2`, which results the following situation:


After the last pointer assignment statement is executed, the target of `p2` is changed to that of `p1`, so that `p1` and `p2` are now both alias of `t1` but leaves the value `t2` unchanged:


Note that the statement

```
p2 => p1 + 4.3            ! illegal
```

is illegal because we cannot associate a pointer with an arithmetic expression.

## 5.3.1 Pointer Versus Ordinary Assignments

Contrast this with the following code (only the last line is different):

```
REAL, POINTER :: p1, p2
REAL, TARGET :: t1 = 3.4, t2 = 4.5
p1 => t1              ! p1 points to t1
p2 => t2              ! p2 points to t2
p2 = p1               ! ordinary assignment, equivalent to t2 = t1
```

After the last ordinary assignment (versus pointer assignment) statement is executed, the situation is as follows:


Note that this assignment has exactly the same effect as

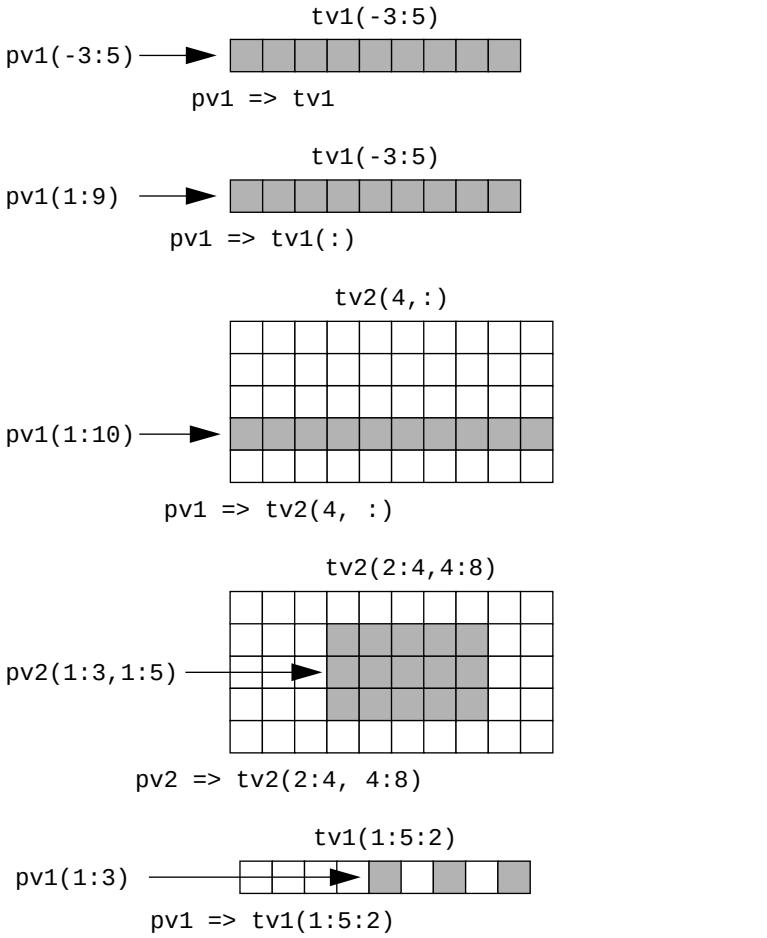
```
t2 = t1
```

since `p1` is an alias of `t1` and `p2` is an alias of `t2`.

## 5.3.2 Array Pointers

The target of a pointer can also be an array. Such a pointer can be referred to as an array pointer. The following example and figure show the use of array pointers:

```
REAL, DIMENSION (:), POINTER :: pv1
REAL, DIMENSION (:, :), POINTER :: pv2
REAL, DIMENSION (-3:5), TARGET :: tv1
REAL, DIMENSION (5, 10), TARGET :: tv2
INTEGER, DIMENSION(3) :: v = (/ 4, 1, -3 /)
pv1 => tv1           ! pv1 aliased to tv1
pv1 => tv1(:)        ! pv1 points to tv1 with section subscript
pv1 => tv2(4, :)     ! pv1 points to the 4th row of tv2
pv2 => tv2(2:4, 4:8) ! pv2 points to a section of tv2
pv1 => tv1(1:5:2)    ! pv1 points to a section of tv1
pv1 => tv1(v)        ! invalid
```



There are several important points to observe:

- The pointer `pv1` is associated at different times with arrays (array sections) having different extents. This is allowed because it is only the rank that matters; the extent of array does not matter.
- If an array pointer is aliased with an array, its extents remains 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`; thus `pv1(1)` is interpreted as `tv1(-3)`, `pv1(2)` is `tv1(-2)`, and so on. This renumbering also happens when `tv2` is aliased to the array section `tv2(2:4, 4:8)`.
- 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. So the pointer assignment `pv1 => tv1(1:5:2)` is valid with `pv1(1)` aliased to `tv1(1)`, `pv1(2)` to `tv1(3)`, and `pv1(3)` to `tv1(5)`, but the last pointer assignment is invalid.

# 5.4 Pointer Association Status

Every pointer has one of the following three association states:

1. *Undefined* - when it is initially specified in a type declaration statement.
2. *Null* (*disassociated*) - when it is nullified by a NULLIFY statement.
3. *Associated* - when it points to a target.

A pointer may be explicitly disassociated from its target and set to point at 'nothing' by executing a `NULLIFY` statement, whose general form is

```
NULLIFY(list of pointers)
```

The intrinsic function `ASSOCIATED` can be used to test the association status of a pointer with one argument or with two:

```
ASSOCIATED(p, [,t])
```

When `t` is absent, it returns the logical value `.TRUE.` if the pointer `p` is currently associated with a target and `.FALSE.` otherwise. If `t` is present and is a target variable, it returns `.TRUE.` if the pointer `p` is associated with `t` and `.FALSE.` otherwise. The second argument `t` may itself be a pointer, in which case it returns `.TRUE.` if both pointers are associated to the same target or disassociated and `.FALSE`. otherwise.

There is one restriction concerning the use of this function, that is the pointer argument must not have an undefined pointer association status. Ther efore, 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.

The following code shows the status of pointers at different stages:

```
REAL, POINTER :: p, q           ! undefined association status
REAL, TARGET :: t = 3.4
p => t                          ! p points to t1
q => t                          ! q also points to t1
PRINT *, "After p => t, ASSOCIATED(p) = ", ASSOCIATED(p)     ! .T.
PRINT *, "ASSOCIATED(p, q) = ", ASSOCIATED(p, q)             ! .T.
NULLIFY(p)
PRINT *, "After NULLIFY(p), ASSOCIATED(p) = ", ASSOCIATED(p) ! .F.
PRINT *, "ASSOCIATED(p, q) = ", ASSOCIATED(p, q)             ! .F.
```

```
...
p => t                          ! p points to t2
NULLIFY(p, q)
```

Note that the disassociation of p did not affect q even though they were both pointing at the same object. After being nullified, p can be associated again either with the same or different object later. The last line just illustrates that a NULLIFY statement can have more than one pointer argument.

# 5.5 Dynamic Storage

Besides pointing to existing variables with a TARGET attribute, a pointer may be associated with a dynamically allocated area of memory via the ALLOCATE statement. The ALLOCATE statement creates an un-named variable or array of the specified size, having the correct type, type parameters and rank, and with an implied target attribute:

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

In this example, the pointer p is set to point to a dynamically allocated area of memory able to store a real variable, and the pointer pv to a dynamically allocated real array of size m by n.

The area of memory which was created by a pointer allocate statement can be released when no longer required by means of the DEALLOCATE statement:

```
DEALLOCATE(pv)
```

Here when the area of memory allocated for pv is deallocated, the association status of pv becomes null.

The general forms of the ALLOCATE and DEALLOCATE statements are

```
ALLOCATE(pointer[(dimension specification)]... [,STAT = status])
DEALLOCATE(pointer... [,STAT = status]
```

where *pointer* is a pointer variable, *dimension specification* is the specification of the extents for each dimension if the pointer variable has both the dimension and pointer attributes (array pointer), and *status* is an integer variable which will be assigned the value zero after a successful allocation/deallocation, or a positive value after an unsuccessful allocation/deallocation. Note that both statements can allocate/deallocate memory for more than one pointer.

The ability to create dynamic memory brings greater versatility and freedom to programming, but also brings its own problems if care is not taken. In particular there are two potential dangers which need to be avoided.

The first is the *dangling pointer*. Consider the following

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

The pointers p1 and p2 both are alias of the same dynamic variable. After the execution of the DEALLOCATE statement, it is clear that 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. In this case, the solution is to make sure that p2 is nullified immediately after the deallocation.

The second is that of *unreferenced storage*. Consider the following

```
...
REAL, DIMENSION(:), POINTER :: p
ALLOCATE ( p(1000) )
...
```

If p is nullified or set to point to somewher e else, or the subprogram is exited (note that p has no SAVE attribute), without first deallocating it, ther e is no way to refer to that block of memory and so it can not be released. The solution is to deallocate a dynamic object before modifying a pointer to it.

# 5.6 Pointer Arguments

Pointers, whether allocated or not, are allowed to be procedure arguments, but only as long as the following conditions are adhered to:

- 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, 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.

Consider the following program extract:

```
...          ! program unit which calls sub1 and sub2
INTERFACE    ! interface block for sub2
   SUBROUTINE sub2(b)
      REAL, DIMENSION(:, :), POINTER :: b
   END SUBROUTINE sub2
END INTERFACE
REAL, DIMENSION(:, :), POINTER :: p
...
ALLOCATE (p(50, 50))
CALL sub1(p)
CALL sub2(p)
...

SUBROUTINE sub1(a)! a is not a pointer but an assumed shape array
   REAL, DIMENSION(:, :) :: a
   ...
END SUBROUTINE sub1

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

The important points here are:

- Both sub1 and sub2 are external procedures. 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.

- The calling program unit sets the pointer `p` as an alias to a dynamically allocated real array of size `50` by `50`, and then calls sub2. This associates the dummy pointer b with the actual pointer argument p. When sub2 deallocates b, this also deallocates the actual argument p in the calling program unit and sets the association status of p to null.

In contrast, allocatable arrays can not be used as dummy arguments, and must, therefore, be allocated and deallocated in the same program unit. Only allocated allocatable arrays can be passed as actual arguments, but not unallocated allocatable arrays.

# 5.7 Pointer Functions

A function result may also have the pointer attribute, which is useful if the result size depends on calculations performed in the function. For example

```
...
INTEGER, DIMENSION(100) :: x
INTEGER, DIMENSION(:), POINTER :: p
...
p => gtzero(x)
...
CONTAINS
   FUNCTION gtzero(a)! function to get all values .gt. 0 from a
      INTEGER, DIMENSION(:), POINTER :: gtzero
      INTEGER, DIMENSION(:) :: a
      INTEGER :: n
      ...  ! find the number of values .gt. 0, n
      IF (n == 0)
         NULLIFY(gtzero)
      ELSE
         ALLOCATE (gtzero(n))
      ENDIF
      ...  ! put the found values into gtzero
   END FUNCTION gtzero
...
```

There are two points which need to be mentioned in the above example:

- The pointer function `gtzero` has been put as an internal procedure, because the interface to a pointer function must be explicit.
- The pointer function result can be used as an expression (but must be associated with a defined tar get firstly) in a pointer assignment statement. As a r esult, the pointer p points to a dynamically allocated integer array, of the correct size, containing all positive values of the array `x`.

# 5.8 Arrays of Pointers

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.

Suppose an array of pointers to reals is required. A derived type `real_pointer` can be defined, whose only component is a pointer to reals:

```
TYPE real_pointer
   REAL, DIMENSION(:), POINTER :: p
END TYPE real_pointer
```

Then an array of variables of this type can be defined:

```
TYPE(real_pointer), DIMENSION(100) :: a
```

It is now possible to refer to the *i*th pointer by writing `a(i)%p`.

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

```
INTEGER, PARAMETER :: n=10
TYPE(real_pointer), DIMENSION(n) :: a
INTEGER :: i

DO i = 1, n
   ALLOCATE (a(i)%p(i)) ! refer to the ith pointer by a(i)%p
END DO
```

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.

# 5.9 Linked List

One of the common and powerful applications of pointers is in setting up and manipulating linked list. In a linked list, the connected objects (such an object can be called a node):

- are not necessarily stored contiguously,
- can be created dynamically (i.e., at execution time),
- may be inserted at any position in the list,
- may be removed dynamically.

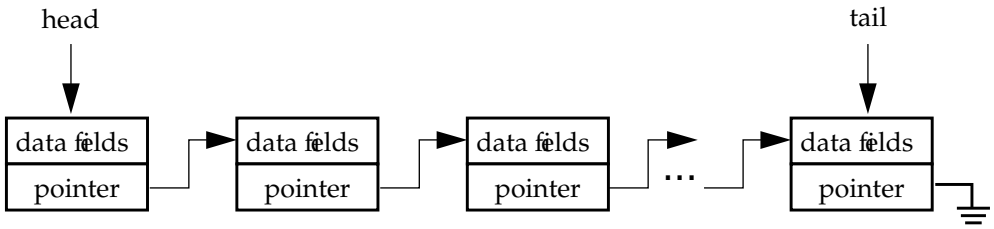
Therefore, the size of a list may grow to an arbitrary size as a program is executing.

In this section we will give a simple example to explain how to build a linked list. Note that trees or other dynamic data structures can be constructed in a similar way to linked lists.

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
```

As shown above, 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. It is convenient to represent a linked list in diagrammatic form, as shown in following figure:




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

Consider the following example:

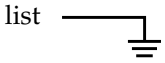
```
PROGRAM simple_linked_list
   IMPLICIT NONE
   TYPE node
      INTEGER :: value             ! data field
      TYPE (node), POINTER :: next  ! pointer field
   END TYPE node
   INTEGER :: num, status
   TYPE (node), POINTER :: list, current
   ! build up the list
   NULLIFY(list)              ! initially nullify list (empty)
   DO
      READ *, num             ! read num from keyboard
      IF (num == 0) EXIT      ! until 0 is entered
      ALLOCATE(current, STAT = status)! create new node
      IF (status > 0) STOP 'Fail to allocate a new node'
      current%value = num     ! giving the value
      current%next => list    ! point to previous one
      list => current         ! update head of list
   END DO
   ! traverse the list and print the values
   current => list            ! make current as alias of list
   DO
      IF (.NOT. ASSOCIATED(current)) EXIT! exit if null pointer
      PRINT *, current%value  ! print the value
      current => current%next ! make current alias of next node
   END DO
END PROGRAM simple_linked_list
```

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.

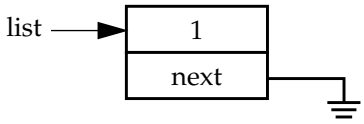
Then two variables of this type are declared, `list` and `current`, where `list` will be used to point to the head of the list and `current` to a general node of the list.

The procedure of building up this linked list is illustrated progressively as follows:

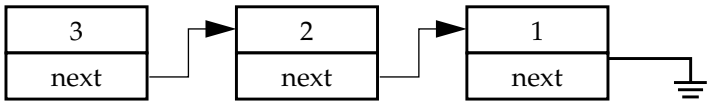
- At first, the list is empty , so `list` should point to both the beginning and the end. This is effected by initially nullifying `list`, which is represented by the symbol for earthing an electrical conductor:




- 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, and letting `current%next` point to null.




- This process can be repeated as long as the value 0 is read in. If, for example, the values 1, 2, 3 are entered in that order, the linked list looks like:




Having built up the linked list, the next thing is to traverse it and print all the values:

- We start by making current an alias of list, which points to the head of the list.
- Then we print the value of that node (current%value).
- After printing, current is made to point to the next node.
- Each time, the association status of the current node is tested to see if it is null. If null, the tail of the list has been reached and the list traversing is finished.

Note that it is important to never loose the head of the list, as this would cause unreferenced storage.

One of advantages of using linked list is that its storage can be released when no longer needed. This can be easily done by traversing the list and deallocating all the nodes in a similar way as the above traversing and printing all the values:

```
! traverse the list and deallocate each node
current => list           ! make current point to head of list
DO
   IF (.NOT. ASSOCIATED(current)) EXIT! exit if null pointer
   list => current%next    ! make list point to next node of head
   DEALLOCATE(current)     ! deallocate current head node
   current => list         ! make current point to new head
END DO
```

Note that the linked list built up stores the reading values in reverse order. If the order of reading values are to be preserved in the linked list, more housekeeping work is required, and this is left as an exercise.

# 5.10 Exercises

1. Write a program, in which you:

   a) Define two pointers pv1 and pv2, where pv1 can point to a one dimensional real array and pv2 can point to a two dimensional real array.

   b) Define two target real arrays, tv1 and tv2, where tv1 is one dimensional with bounds -3:5 and tv2 is two dimensional with bounds 1:5, 1:10.

   c) Set up the array pointer pv1 to point to tv1 such that pv1 has the lower bound -3 (Write out the lower bound of pv1 for confirmation).

   d) Set up the array pointer pv1 to point to tv1 such that pv1 has the lower bound 1 (Write out the lower bound of pv1 for confirmation).

   e) Can you set pv1 to point to tv1 such that pv1 has the lower bound -2?

   f) Use pointers (pv1 or pv2) to write out the 4th row of tv2, the section tv2(2:4, 4:8) and the section tv1(1:5:2).

   (`p_array.f90`)

2. Look at the program `status.f90`, write down what you think will be printed. Then run the program to compare.

3. Write a program which uses an array of pointers (simulated by means of a derived type having a pointer component of the desired type) to set up a lower-triangular matrix. (`p_matrix.f90`)

4. Run the program `simple.f90`, notice that the linked list stores the typed-in numbers in reverse order. Modify this program such that the linked list preserves the order of typed-in numbers. (`linklist.f90`)

5. Run the program `polyline.f90`, which uses the polyline module (`poly_mod.f90`), notice that the linked list stores the points read in reverse order. When prompted for a y value enter the y value of the point that you want to delete from the list. Compare the two versions of the list that have been printed. Has your point been deleted?

   Modify this program such that the linked list preserves the order of points read. (`poly2.f90`)

# 6 Input/Output

The only major new input/output features in Fortran 90 are NAMELIST, non-advancing I/O and some new edit descriptors.

## 6.1 Non-advancing I/O

In Fortran 77, every READ or WRITE statement involved complete records. There are occasions where it would be convenient to read/write only part of a record, and read/write the rest later. In Fortran 90, this facility is provided by non-advancing I/O.

Non-advancing I/O obviates the need for records to be read as a whole and for the record length to be known beforehand. It is specified with

```
ADVANCE='NO'
```

on the READ or WRITE statement and inhibits the automatic advance to the next record on completion of the statement. If

```
ADVANCE='YES'
```

is specified, or the specifier is absent, then the default normal (advancing) I/O occurs.

It is possible to specify advancing and non-advancing I/O on the same record or file. A common use of this is to write a prompt to the screen, specifying non-advancing I/O, and then read the next character position on the screen. For example:

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

It is often useful to determine how many characters have been read on a non-advancing input. This can be achieved using the SIZE specifier , which has the general form

```
SIZE=character_count
```

The integer variable character_count is assigned the number of characters read, excluding any padding characters, on completion of the non-advancing read.

If a non-advancing input reads beyond the end of a record, this can be detected using the IOSTAT specifier which has the form

```
IOSTAT=io_status
```

On completion of a READ statement io_status is assigned a value which indicates whether an end-of-record or end-of-file condition has occurred. For example, the NAG compiler returns -1 in the IOSTAT specifier when end-of-file is encountered, and -2 for end-of-record.

# 6.2 INQUIRE by I/O List

This is used to determine the length of an unformatted output item list. The form

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

The length may be used as a value of the RECL specifier in subsequent OPEN statements. For 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
...
```

# 6.3 NAMELIST

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. However, NAMELIST is a poorly designed feature and should be avoided whenever possible.

NAMELIST is a facility whereby a set of variables can be gathered together into a named group in order to simplify I/O. The NAMELIST statement is a specification statement and must, therefore, appear before any executable code in the defining program unit. The general form of the NAMELIST statement is:

```
NAMELIST/namelist-group-name/variable-list
```

Note that a variable in a NAMELIST group may not be an array dummy argument with non-constant bounds, a variable with assumed character length, an automatic object, an allocatable array, a pointer, or a variable which at any depth of component selection is a pointer.

In READ or WRITE statements, the *namelist-group-name* may be specified with the NML specifier , or may replace the format specifier . There is no need for input/output lists.

An I/O record for a namelist group has a specific format:

```
&namelist-group-name var1=x, var2=y, var3=z
```

It is possible to omit items when inputting data, and such items remain unchanged. Also, items do not have to be input in the order specified in the NAMELIST statement.

## 6.3.1 Example

This example shows the namelist group named clothes:

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

The output would be:

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

# 6.4 New Edit Descriptors

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. Fortran 90 provides the following 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

## 6.4.1 Example

This example illustrates 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
```

The output would be

```
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

1.234       -0.500         0.678E-02    0.988E+05
```

# 6.5 New Statement Specifiers

The `INQUIRE`, `OPEN`, `READ` and `WRITE` statements are not new, but a few new specifiers have been added.

**INQUIRE**

`POSITION` `= ASIS, REWIND, APPEND or UNDEFINED`

The initial file position as specified by the corresponding OPEN statement.

`ACTION` `= READ, WRITE, READWRITE or UNDEFINED`

`DELIM` = `APOSTROPHE, QUOTE, NONE or UNDEFINED`

The character used to delimit character constants with list-directed or `NAMELIST` I/O. The default is `'NONE'`.

`PAD` = `YES or NO`

`'YES'` means the input record is (to be regarded as) padded with blanks. `'NO'` means the input record must be long enough to accommodate the input list (except for `ADVANCE='NO'`). The default is `PAD='YES'`.

`READWRITE` = `YES, NO or UNKNOWN`

Indicates whether `READWRITE` is allowed, not allowed or undetermined for a file

`READ` = `YES, NO or UNKNOWN`

Indicates whether `READ` is allowed, not allowed or undetermined for a file

`WRITE` = `YES, NO or UNKNOWN`

Indicates whether `WRITE` is allowed, not allowed or undetermined for a file

**OPEN**

The specifiers `POSITION`, `ACTION`, `DELIM` and `PAD` have the same values and meanings as for `INQUIRE`. One additional value has been provided for the `STATUS` specifier:

`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.

**READ/WRITE**

`NML` = `namelist_name`

This would be used in place of the `FMT` specifier when using a `NAMELIST` group.

`ADVANCE` = `YES, or NO`

**READ**

`EOR` = `label`

Control passes to statement label when an end-of-record condition occurs.

`SIZE` = `character count`

# 6.6 Exercises

Look at the programs `non_adv.f90`, `inquire.f90`, `namelist.f90`, `edit1.f90`, `edit2.f90`, and `io_spec.f90`, and run them. Notice how these programs use new I/O facilities.

# 7 Intrinsic Procedures

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

The intrinsic procedures fall into four distinct categories:

- Elemental procedures

  These are specified for scalar arguments, but are also applicable to conforming array arguments applying the procedure element by element.

- Inquiry functions

  Inquiry functions return properties of principal arguments that do not depend upon their values.

- Transformational functions

  These functions usually have array arguments and an array result whose elements depend on many of the elements in the array arguments.

- Nonelemental Subroutines

The new intrinsic features provided by Fortran are described briefly in this chapter, divided into the four categories given above.

## 7.1 Elemental Procedures

### 7.1.1 Elemental Functions

**Numeric**

`CEILING(A)`
Smallest integer not less than `A`.

`FLOOR(A)`
Largest integer not exceeding `A`.

`MODULO(A,P)`
A modulo `P` for `A` and `P` both real or both integer.

**Character**

`ACHAR(I)`
Character in position `I` of ASCII collating sequence.

`ADJUSTL(STRING)`
Adjust left, change leading blanks into trailing blanks.

`ADJUSTR(STRING)`
Adjust right, change trailing blanks into leading blanks.

`IACHAR(C)`
Position of character `C` in ASCII collating sequence.

`INDEX(STRING,SUBSTRING[,BACK])`
Starting position of `SUBSTRING` within `STRING`. If more than one `SUBSTRING` than position of first (or last if `BACK` true) is returned.

`LEN_TRIM(STRING)`
Length of `STRING` without trailing blanks.

`SCAN(STRING,SET[,BACK])`
Index of left-most (right-most if `BACK` true) character of string that belongs to `SET`; zero if none belong.

`VERIFY(STRING,SET[,BACK])`
The position of left-most (or right-most if `BACK` true) character of `STRING` that is not in `SET`. Zero if each character of `STRING` appears in `SET`.

**Bit Manipulation**

`BTEST(I,POS)`
True if bit `POS` of integer `I` has value 1.

`IAND(I,J)`
Logical `AND` on all corresponding bits of `I` and `J`.

`IBCLR(I,POS)`
Bit `POS` of `I` cleared to zero.

`IBITS(I,POS,LEN)`
Extract sequence of `LEN` bits of `I` starting from bit `POS`.

`IBSET(I,POS)`
Bit `POS` of `I` set to 1.

`IEOR(I,J)`
Logical exclusive `OR` on all corresponding bits of `I` and `J`.

`IOR(I,J)`
Logical inclusive `OR` on all corresponding bits of `I` and `J`.

`ISHFT(I,SHIFT)`
Value of `I` with bits shifted `SHIFT` places to left (right if negative) and zeros shifted in from other end.

`ISHFTC(I,SHIFT[,SIZE])`
Value of `I` with circular shift of `SIZE` right-most bits `SHIFT` places to the left (right if negative).

`NOT(I)`
Logical complement of all bits of `I`.

**Kind**

`SELECTED_INT_KIND(R)`
Kind of type parameter for specified exponent range. -1 returned if no such kind is available.

`SELECTED_REAL_KIND(P,R)`
Kind of type parameter for specified precision and exponent range. -1 returned is precision is unavailable, -2 if range is unavailable and -3 if neither are available.

**Floating Point Manipulation**

`EXPONENT(X)`
Exponent part of the model for `X`.

`FRACTION(X)`
Fractional part of the model for `X`.

`NEAREST(X,S)`
Nearest different machine number in the direction given by the sign of `S`

`RRSPACING(X)`
Reciprocal of relative spacing of the model numbers near `X`.

`SCALE(X I)`
$X2^I$ (real)

`SET_EXPONENT(X,I)`
Real whose sign and fractional part are those of `X`, and whose exponent part is `I`.

`SPACING(X)`
Absolute spacing of model numbers near `X`.

**Logical**

`LOGICAL(L[,KIND])`
Converts between kinds of logical numbers.

### 7.1.2 Elemental Subroutine

`CALL MVBITS(FROM,FROMPOS,LEN,TO,TOPOS)`
Copy `LEN` bits of `FROM` starting at position `FROMPOS` to `TO`, starting at position `TOPOS`.

## 7.2 Inquiry Functions

`ASSOCIATED(POINTER[,TARGET])`
True if pointer associated with a target. If target present, then true only if associated with specified target.

`BIT_SIZE(I)`
Maximum number of bits that may be held in an integer.

`KIND(X)`
Kind type parameter for `X`.

`PRESENT(A)`
True if optional argument `A` is present.

**Numeric**

`DIGITS(X)`
Number of significant digits in the model for `X`.

`EPSILON(X)`
Number that is almost negligible compared with 1 in the model for numbers like `X`.

`HUGE(X)`
Largest number in the model for numbers like `X`.

`MAXEXPONENT(X)`
Maximum exponent in the model for numbers like `X`.

`MINEXPONENT(X)`
Minimum exponent in the model for numbers like `X`.

`PRECISION(X)`
Decimal precision in the model for `X`.

`RADIX(X)`
Base of the model for numbers like `X`.

`RANGE(X)`
Decimal exponent range in the model that includes integer, real and complex `X`.

`TINY(X)`
Smallest positive number in the model for numbers like `X`.

# 7.3 Transformational Functions

`REPEAT(STRING,NCOPIES)`
Concatenates `NCOPIES` of `STRING`.

`TRANSFER(SOURCE,MOLD[,SIZE])`
Same physical representation as `SOURCE`, but of type `MOLD`.

`TRIM(STRING)`
Removes trailing blanks from `STRING`.

# 7.4 Non Elemental Intrinsic Subroutines:

`CALL DATE_AND_TIME([DATE][,TIME][,ZONE][VALUES])`
Real-time clock reading date and time.

`RANDOM_NUMBER(HARVEST)`
Random numbers in range $0 \le X < 1$.

`RANDOM_SEED([SIZE][,PUT][,GET])`
Initialize or restart random number generator.

`SYSTEM_CLOCK([COUNT][,COUNT_RATE][COUNT_MAX])`
Integer data from real-time clock.

# 7.5 Array Intrinsic Procedures

**Reduction**

`ALL(MASK[,DIM])`
True if all elements true

`ANY(MASK[,DIM])`
True if any element true

`COUNT(MASK[,DIM])`
Number of true elements

`MAXVAL(ARRAY[,DIM][,MASK])`
Maximum element value

`MINVAL(ARRAY[,DIM][,MASK])`
Minimum element value

`PRODUCT(ARRAY[,DIM][,MASK])`
Product of array elements

`SUM(ARRAY[,DIM][,MASK])`
Sum of array element

**Inquiry**

`ALLOCATED(ARRAY)`
: True if array allocated

`LBOUND(ARRAY[,DIM])`
: Lower bounds of array

`SHAPE(SOURCE)`
: Shape of array (or scalar)

`SIZE(ARRAY[,DIM])`
: Size of array

`UBOUND(ARRAY[,DIM])`
: Upper bounds of array

**Construction**

`MERGE(TSOURCE,FSOURCE,MASK)`
: Merge arrays subject to mask

`PACK(ARRAY,MASK[,VECTOR])`
: Pack elements into vector subject to mask

`SPREAD(SOURCE,DIM,NCOPIES)`
: Construct an array by duplicating an array section

`UNPACK(VECTOR,MASK,FIELD)`
: Unpack elements of vector subject to mask

**Reshape**

`RESHAPE(SOURCE,SHAPE[,PAD][,ORDER])`
: Reshape array

**Array Location**

`MAXLOC(ARRAY[,MASK])`
: Location of maximum element

`MINLOC(ARRAY[,MASK])`
: Location of minimum element

**Array manipulation**

`CSHIFT(ARRAY,SHIFT[,DIM])`
: Perform circular shift

`EOSHIFT(ARRAY,SHIFT[,BOUNDARY][,DIM])`
: Perform end-off shift

`TRANSPOSE(MATRIX)`
: Transpose matrix

**Vector and matrix arithmetic**

`DOT_PRODUCT(VECTOR_A,VECTOR_B)`
: Compute dot product

`MATMUL(MATRIX_A,MATRIX_B)`
: Matrix multiplication

# 7.6 Exercises

1. Look at the programs `char_int.f90, model.f90, mod_int.f90` and `convert.f90` and run them.

   Notice how these programs use intrinsic functions for certain purposes.

# 8 Redundant Features

The method of removing redundant features of Fortran, as adopted by the standards committee, was described in section 1.3, "Language Evolution". The obsolescent list includes features which might be removed in the next revision and should not be used in new or revised programs. Additionally, Fortran 90 includes other redundant features which are safer to use then those in the obsolescent list, but will probably be included in the obsolescent list in the next version, and therefore are recommended not to be used in new or revised programs.

The redundant features of Fortran 90 fall into five groups: Source form, Data, Control, Procedures and Input/Output.

## 8.1 Source Form

The fixed source form based on the layout of a punched card has now been replaced by the free source form and this should be used for all new programs. It is possible to make simple modifications to fixed source form in order to produce code which is both legal fixed and free source code.

The use of modules is recommended rather than the `INCLUDE` line. The `INCLUDE` line contained a character literal constant indicating what text should be inserted at a specified point or where text to be inserted should be obtained from.

## 8.2 Data

Fortran 77 provided two forms of real variables and constants, `REAL` and `DOUBLE PRECISION`. These have been superseded by the concept of parameterised data types which provide numerical portability, and hence `DOUBLE PRECISION` should no longer be used in new programs.

The dangerous concept of implicit typing and the `IMPLICIT` statement should not be used. The `IMPLICIT NONE` statement should be included at the beginning of every program unit to ensure explicit declaration of all variables.

The new form of declaring variables with a double colon (::) between the type and the list of variables is recommended. Additionally, the use of attribute forms of `PARAMETER`, `DIMENSION`, etc., in the type declaration, rather than the statement forms is recommended.

The `DATA` statement is no longer generally needed as variables may now be initialised in a type statement. Exceptions to this are octal, hexadecimal and array section initialisations.

The only form of adjustable size array in Fortran 77 was the assumed-size array. In Fortran 90 this has been superseded by the assumed-shape array, and thus the assumed-size array should no longer be used in new programs.

`COMMON` blocks and `BLOCK DATA` should no longer be used as the use of modules obviates the need for them. Similarly the `EQUIVALENCE` statement has become unnecessary

due to the introduction of modules, dynamic storage allocation, pointers, and the intrinsic function TRANSFER.

It is recommended that the SEQUENCE attribute is never used.

# 8.3 Control

The obsolescent features listed in Fortran 90 are:

- 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.
- Alternate RETURN.
- PAUSE statement.

These should never be used in new or revised programs. Except for alternate RETURN and PAUSE, these can be replaced by the IF statement, DO and CASE control constructs, and EXIT and CYCLE statements

With the introduction of modern control constructs and the character string format specifications, the need for labels is redundant.

The DO construct and EXIT and CYCLE statements replace the use of the CONTINUE statement to end a DO loop.

GO TO and computed GO TO statements should be avoided, using IF, DO and CASE constructs, and EXIT and CYCLE statements instead.

The DO WHILE statement was introduced in Fortran 90. This functionality can equally be provided using the DO loop construct and EXIT statement, and this form is recommended.

# 8.4 Procedures

Intrinsic functions using specific names for different data types have been superseded by generic versions. Note that the specific names are required when an intrinsic function is being used as an actual argument.

The ENTRY statement allows a procedure to have more than one entry point. The introduction of modules, where each entry point becomes a module procedure, has made the ENTRY statement unnecessary.

The statement function provided a means of defining and using a one-line function. This has been superseded by the concept of internal procedures.

The use of module procedures and internal procedures means that it is not necessary to use external procedures. Thus, external procedures and the EXTERNAL statement are effectively redundant.

# 8.5 Input/Output

The obsolescent features listed in Fortran 90 are:

- Assigned Format specifiers (replaced by the character string format specifications).

- H edit descriptor (replaced by the character constant edit descriptor A).

Assigned format specifiers should be replaced by character string format specifications.

The END, EOR and ERR specifiers are used when exceptional conditions occur in input and output. It is recommended that the IOSTAT specifier is used instead of these.

Namelist input/output is a poorly designed feature and it is recommended that namelist should not be used unless absolutely necessary.

Finally, it is recommended that six edit descriptors should not be used, namely, D, BN, BZ, P, G and X.

The D edit descriptor has been superseded by the E edit descriptor, and the BN and BZ edit descriptors have both been replaced by the BLANK specifier.

The P edit descriptor allows numeric data to be scaled on either input or output, however this can lead to unnecessary confusion and is therefore best avoided.

The G edit descriptor is a generalized edit descriptor which can be used to input or output values of any intrinsic type. However, the use of I, E, EN, F, L or A edit descriptors is preferable as these provide some check that the data types are correct.

The X edit descriptor has the same effect as the TR edit descriptor, and the latter is recommended.

# 9 Further Development

## 9.1 Fortran 95

Fortran 95 will be a fairly small update to Fortran 90, consisting mainly of clarifications and corrections to Fortran 90. The next major changes are expected in Fortran 2000.

Fortran 95 will, however, provide some new features including:

- `FORALL` statement and construct

  This allows for more flexible array assignments. For example:

```
FORALL (i=1:n) a(i,i)=i

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
```

- `PURE` attribute
  Allowing `PURE` procedures safe for use in `FORALL` statements.
- CPU time intrinsic inquiry function

```
CALL CPU_TIME(t1)
```

- Allocatable dummy arguments and results
- Nested `WHERE`

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

- Object Initialisation
  Initial pointer or type default status.

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

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

# 9.2 Parallel Computers

It is important, nowadays, that a new programming language standard should permit efficient compilation and execution of code on super computers as well as conventional computers. Fortran 90 is said to be efficient on conventional computers and on vector processors, but less efficient on parallel computers. However , the limelight of supercomputing research has recently shifted away from vector computers and towards parallel and "massively" parallel computers. This inter est in parallel computers has lead to the development of two de facto standards:

- High Performance Fortran (HPF).
- Message Passing Interface (MPI).

## 9.2.1 High Performance Fortran

The goal of High Performance Fortran is to provide 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 turning for various architectures.
- Minimal deviation from other standards.
- Define open interfaces to other languages.
- Encourage input from the high performance computing community.

Fortran 90 supports data parallel programming through the array operations and intrinsics. HPF extends this support with:

- Compiler directives for data alignment and distribution.
- Concurrent execution features using the `FORALL` statement.
- The `INDEPENDENT` directive which allows the programmer to provide the compiler with information about the behaviour of a `DO` loop or `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.

## 9.2.2 Message Passing Interface

MPI is a proposed standard Message Passing Interface for:

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

Such a standard is required for several reasons:

- 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.

MPI contains:

- Point-to-point message passing.
  - Blocking and non-blocking sending and receiving in 3 modes: ready, standard and synchronous.
  - Generalising the description of buffers, the type and the process identifier, heterogeneity.
- Collective communication routines.
  - Data movement (one-all and all-all versions of the broadcast, scatter, and gather routines).
  - Global computation (reduce and scan routines).
- Support for process groups and communication contexts.
  - Communicators combine context and group for message security and thread safety.
- Support for application topologies (grids and graphs).

# Appendix A: References

Adams, J. C. et. al. (1992) *Fortran 90 Handbook*. McGraw-Hill. ISBN 0-07-000406-4

Brainerd, W. S. et. al., (1994) *Programmer's Guide to Fortran 90*. 2nd edition, Unicomp. ISBN 0-07-000248-7

Buckley, A. G. (1993) *Conversion to Fortran 90: A Case Study.* Via Web.

Counihan, M. (1991) *Fortran 90*. Pitman. ISBN 0-273-03073-6

Dodson, Z. (1994) *A Fortran 90 Tutorial*. ViaWeb.

Einarsson, B. and Shokin, Y. (1993) *Fortran 90 for the Fortran 77 Programmer.* Via Web.

Ellis, T. M. R. et. al., *Fortran 90 Programming*. Wesley. ISBN 0-201-54446-6

Hahn, B. D. (1994) *Fortran 90 for Scientists and Engineers*. Edward Arnold. ISBN: 0-340-60034-9

Kerrigan, J. (1993) *Migrating to Fortran 90*. O'Reilly and Associates. ISBN 1-56592-049-X

Metcalf, M. (1990) *Fortran 90 Tutorial*. Via Web.

Metcalf, M. & Reid, J. (1992) *Fortran 90 Explained*. Oxford University Press. ISBN: 0-19-853772-7

Morgan, J. S. & Schonfelder, J. L. (1993) *Programming in Fortran 90*. Alfred Waller Ltd. ISBN 1-872474-06-3

Sawyer, M. *A Summary of Fortran 90.* Via Web.

Smith, I M. *Programming in Fortran 90*. Wiley. 0471-94185-9

For articles obtained via World Wide Web, see URL http://www.lpac.ac.uk/SEL-HPC/Articles/GeneratedHtml/hpc.f90.html