Introduction to Fortran 90

An introduction Course for

Novice Programmers

Student Notes

Rob Davies

Cardiff

Alan Rea

Belfast

Dimitris Tsaptsinos

SEL - HPC

Version 1.0

9 Introduction

9 Programming in general
9 History
10 ANSI Standard
10 Compilation
11 Coding conventions

13 Variables and Statements

13 Variables
14 Naming Convention
14 Specification or declaration
15 Parameters
15 Implicit Declaration
15 KIND type
16 Portability
17 Type conversion
17 Arithmetic expressions
18 Comments
18 Program Layout
19 Derived Data Types
19 Definition and specification
20 Accessing Components
21 Exercises

23 Character Processing

23 Character Type
23 Character Constants
24 Character Variables
24 Character manipulation
24 Concatenation
25 Substrings
25 Intrinsic Functions
26 Exercises

29 Logical & comparison expressions

29 Relational operators
30 Logical expressions
31 Character Comparisons
31 Portability Issues
32 Exercises

35 Arrays

35 Terminology
35 Arrays and elements

36 Array properties
36 Specifications
37 Array Sections
37 Individual elements
38 Sections
39 Vector Subscripts
39 Array storage
40 Array Assignment
40 Whole array assignment
40 Array section assignment
41 Renumbering
41 Elemental intrinsic procedures
41 Zero-sized arrays
42 Arrays and derived types
43 Initialising arrays
43 Constructors
43 Reshape
44 DATA statement
44 WHERE
45 Array intrinsic functions
45 Example of reduction
46 Example of inquiry
46 Example of construction
46 Example of location
47 Exercises

51 Control statements

51 Conditional statements
51 IF statement and construct
53 SELECT CASE construct
53 GOTO
54 Repetition
54 DO construct
55 Transferring Control
56 Nesting
56 Exercises

59 Program units

59 Program structure
60 The main program
60 Procedures
61 Actual and dummy arguments
62 Internal procedures
62 External procedures
63 Procedure variables

63 SAVE
63 Interface blocks
64 Procedures arguments
64 Assumed shape objects
65 The INTENT attribute
65 Keyword arguments
66 Optional arguments
66 Procedures as arguments
67 Recursion
67 Generic procedures
68 Modules
69 Global data
69 Module procedures
70 PUBLIC and PRIVATE
71 Generic procedures
71 Overloading operators
72 Defining operators
72 Assignment overloading
73 Scope
73 Scoping units
73 Labels and names
74 Exercises

77 Interactive Input and Output

78 Simple Input and Output
78 Default formatting
79 Formated I/O
79 Edit Descriptors
80 Integer
80 Real - Fixed Point Form
80 Real - Exponential Form
81 Character
81 Logical
82 Blank Spaces (Skip Character Positions)
82 Special Characters
82 Input/Output Lists
83 Derived DataTypes
83 Implied DO Loop
83 Namelist
84 Non-Advancing I/O
85 Exercises

87 File-based Input and Output

87 Unit Numbers
88 READ and WRITE Statements

88 READ Statement
89 WRITE Statement
89 OPEN Statement
90 CLOSE statement
90 INQUIRE statement
91 Exercises

93 Dynamic arrays
93 Allocatable arrays
93 Specification
93 Allocating and deallocating storage
94 Status of allocatable arrays
95 Memory leaks
96 Exercises

97 Pointer Variables
97 What are Pointers?
97 Pointers and targets
97 Specifications
98 Pointer assignment
99 Dereferencing
100 Pointer association status
100 Dynamic storage
101 Common errors
101 Array pointers
103 Derived data types
103 Linked lists
103 Pointer arguments
104 Pointer functions
106 Exercises

107 Intrinsic procedures
107 Argument presence enquiry
107 Numeric functions
108 Mathematical functions
108 Character functions
109 KIND functions
109 Logical functions
109 Numeric enquiry functions
109 Bit enquiry functions
109 Bit manipulation functions
110 Transfer functions

110 Floating point manipulation functions
110 Vector and matrix functions
110 Array reduction functions
111 Array enquiry functions
111 Array constructor functions
111 Array reshape and manipulation functions
111 Pointer association status enquiry functions
111 Intrinsic subroutines

113 Further reading

1 Introduction

This course is designed for beginning programmers who may have little or no experience of computer programming and who wish to take advantage of the new Fortran standard.

1.0.1 Programming in general

A program is the tool a user employs to exploit the power of the computer. It is written using the commands and syntax of a language which may be interpreted (via a compiler) by the computer hardware. This course outlines the commands and syntax of the Fortran 90 language.

A program consists of a sequence of steps which when executed result in a task being carried out. Execution means that the computer is able to interpret each step (instruction), interpretation refers to understanding what is required and instructing the hardware to carry it out. Each instruction might require a calculation to be performed, or a decision to be selected, or some information to be stored or retrieved. The nature of the instruction depends on what programming language is used. Each programming language has its own set of statements.

1.1 History

Fortran (mathematical FORmula TRANslation system) was originally developed in 1954 by IBM. Fortran was one of the first languages to allow the programmer to use higher level (i.e. architecture independent) statements rather than a particular machine's assembly language. This resulted in programs being easier to read, understand and debug and saved the programmer from having to work with the details of the underlying computer architecture.

In 1958 the second version was released with a number of additions (subroutines, functions, common blocks). A number of other companies then started developing their own versions of compilers (programs which translate the high level commands to machine code) to deal with the problem of portability and machine dependency.

In 1962 Fortran IV was released. This attempted to standardize the language in order to work independent of the computer (as long as the Fortran IV compiler was available!)

In 1966 the first ANSI (American National Standards Institute) standard (Fortran 66) was released which defined a solid base for further development of the language.

In 1978 the second ANSI standard (Fortran 77) was released which standardized extensions, allowed structured programming, and introduced new features for the IF construct and the character data type.

The third ANSI standard (Fortran 90) was released in 1991, with a new revision expected within 10 years.

1.2 ANSI Standard

Fortran 90 is a superset of Fortran 77, that is programs written in Fortran 77 may be compiled and run as Fortran 90 programs. However Fortran 90 is more than a new release of Fortran 77. The Fortran 90 standard introduces many new facilities for array type operations, new methods for specifying precision, free form, recursion, dynamic arrays etc. Although the whole of Fortran 77 is included in the Fortran 90 release, the new ANSI standard proposes that some of the Fortran 77 features are 'depreciated'. Depreciated features are likely to be classed as 'obsolete' in subsequent releases and removed from Fortran 90.

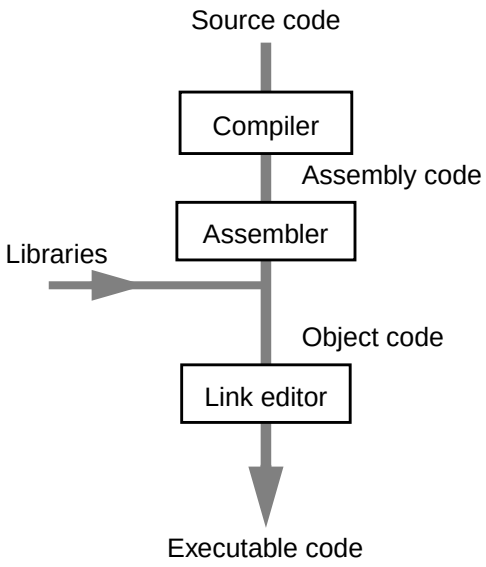
At present an ANSI standard Fortran 77 program should compile successfully with any Fortran 90 compiler without change. However the structure of a Fortran 90 program can be significantly different from that of its Fortran 77 equivalent. Programmers should beware of mixing the two styles, and of consulting Fortran 77 text books for advice. It is recommended that programmers new to Fortran not consult any Fortran 77 books.

A Fortran 90 compiler is required to report any non-conforming code (i.e. the use of statements or variables which are illegal under the rules set out by the ANSI standard). As well as reporting errors a Fortran 90 compiler is required to provide a reason for reporting the error. This should help programmers to write correct code.

As mentioned, Fortran 90 has been augmented with a number of new features to take advantage of modern computing needs and developments; developments such as the recent importance of dynamic data structures and the introduction of parallel architectures.

1.3 Compilation

Once the Fortran 90 program has been designed and entered as source code into a file (usually with the suffix `.f90`) then the following steps are possible:



- Compilation - This is initiated by the programmer, by typing:

  ```
  f90 filename.f90
  ```

  (or something similar) its purpose is to translate the high-level statements (source code) into intermediate assembly code, and from there to machine (object) code. The compiler checks the syntax of the statements against the

standard (`write` rather than `write` will give an error) and the semantics of the statements (misuse of a variable, etc.). This step generates the object code version which is stored in a file of the same name but different extension (usually `o` on UNIX systems).

- Linking - This might be initiated by the compiler, its purpose is to insert any code that is required from a library or other pre-compiled file. This generates the executable code version which again is stored in a file with a different extension (on a UNIX system the default name is `a.out`).
- Execution - This is initiated by the programmer / user, by typing the name of the executable file. Its purpose is to run the program to get some answers. During execution the program might crash if it comes across an execution error (the most common execution error is an attempt to divide by zero).

Note that logical errors (multiply rather than add) can not be checked by the compiler and it is the responsibility of the programmer to identify and eliminate such errors. One way to do so is by testing against data with known results but for more complex programs testing can not take into consideration all possible combinations of inputs therefore great care must be taken during the initial design. Identifying errors at the design phase is cheaper and easier.

1.4 Coding conventions

In these notes all examples of code are written in courier font, e.g.

```
PROGRAM hi
   ! display a message
   WRITE(*,*) 'Hello World!'
END PROGRAM hi
```

As an aid to following the code examples, the convention followed throughout these notes (recommended by NAG) is:

- All keywords and intrinsic procedure names (i.e. those commands and functions that are a part of the standard) are in upper case, everything else is in lower case.
- To help with the reading of code, the body 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 o their `END` statements.
- In `USE` statements, the `ONLY` clause is used to document explicitly all entities which are accessed from that module.
- In `CALL` statements and function references, argument keywords are always used for optional arguments.

2 Variables and Statements

2.1 Variables

It is usual for people to associate a name or phrase with a piece of information. For example, the phrase “today’s date” has an associated numeric value which varies day by day. This is similar to the concept of a program variable; a program variable is some name (chosen by the programmer) which uniquely identifies an object (piece of data) stored in memory.

For example, a program may identify the following values:

```
7
96.4
3.14159
```

by these variable names:

```
daysinweek
temperature
pi
```

It is common for the value of a variable to change while a program runs but it is not required (e.g. the value of `temperature` might well change but `pi` is unlikely to). Variable names are usually a word, phrase, acronym, etc. which is written as one word (see Naming Convention below). Note that it is good programming practice to use variable names that are reminiscent of the information being referred to.

It is important for a computer to identify the data type it has stored. There are several forms of numeric data, namely:

- *Integers*: may only have discrete, whole values (e.g. -3124, -960, 10, 365, etc.).
- *Reals*: may have a fractional part and have a greater range of possible values (e.g. 10.3, -8.45, 0.00002, etc.).
- *Complex* numbers: have both a real and imaginary part (e.g. 3-2i, -5+4i, etc.).

Integers are more accurate for discrete values and are processed fastest, but reals are necessary for many calculations. Complex numbers are necessary for some scientific applications.

As well as numerical data, Fortran programs often require other types of data. Single letters, words and phrases may be represented by the *character* data type, while the logical values ‘true’ and ‘false’ are represented by the *logical* data type (details later).

Finally, It is possible not to use a variable to represent data but to used the value explicitly. For example, instead of using `pi`, a programmer might choose to write 3.14159. Such values are known as literal constants.

2.1.1 Naming Convention

In a Fortran program, variable names must correspond to a naming convention. The naming convention permits names of between 1 and 31 alphanumeric characters (the 26 letters `a...z`, the 10 numerals `0...9` and _ the underscore character) with the restrictions that the first character must be a letter.

Note that there is no case sensitivity in Fortran, the lower and uppercase versions of a character are treated as equivalent, therefore `name`, `Name`, `NaMe` and `NAME` all refer to the same object.

Unlike some programming languages in which certain words are reserved and may only be used by the programmer in precisely defined contexts, Fortran has no reserved words. However the programmer should take care when naming variables and try not to use any words which form part of the language.

- Valid variable names: `x`, `x1`, `mass`, `cost`, `day_of_the_week`
- Valid variable names (but do not use!): `real`, `integer`, `do`, `subroutine`, `program`
- Invalid: `ten.green.bottles`, `1x`, `a thing`, `two-times`, `_time`

In these course notes, all words which have a defined meaning in the Fortran languages are given in uppercase and the user defined objects are given in lowercase.

2.2 Specification or declaration

All variable used in a program must have an associated data type, such as `REAL` `INTEGER` or `COMPLEX`, which is usually identified at the start of the program. This is referred to as declaring or specifying a variable, for example:

```
REAL :: temperature, pressure
INTEGER :: count, hours, minutes
```

declares five variables, two which have values that are real numbers and three that have integer values.

The variable declaration statement may also be used to assign an initial value to variables as they are declared. If an initial value is not assigned to a variable it should not be assumed to have any value until one is assigned using the assignment statement.

```
REAL :: temperature=96.4
INTEGER :: days=365, months=12, weeks=52
```

The general form of a variable declaration is:

```
type [,attributes...] :: variable list
```

Where `type` may be one of the following, intrinsic data types:

```
INTEGER
REAL
COMPLEX
CHARACTER
LOGICAL
```

and `attribute...` is an optional list of ‘keywords’, each separated by a comma, used to further define the properties of variables:

```
ALLOCATABLE     INTENT(...)  PARAMETER  PUBLIC
DIMENSION(...)  INTRINSIC    POINTER    SAVE
EXTERNAL        OPTIONAL     PRIVATE    TARGET
```

CHARACTER and LOGICAL data types are discussed in separate sections, while the attributes will be described as required throughout these notes.

2.2.1 Parameters

The term parameter in Fortran is slightly misleading, it refers to a value which will not change during a program's execution. For example the programmer is likely to want the value of pi to be unaltered by a program. Therefore pi may be defined:

```
REAL, PARAMETER :: pi=3.141592
```

REAL specifies the data type while the PARAMETER attribute further defines the variable pi. All parameters must be given a value in their declaration statement (in this case 3.141592). Parameters may also be defined for other data types, for example:

```
INTEGER, PARAMETER :: maxvalue=1024
INTEGER, PARAMETER :: repeatcount=1000
```

It is an error to try to redefine the value of a parameters while a program executes.

2.2.2 Implicit Declaration

Fortran 90 permits real and integer variables to be typed and declared implicitly, that is used in a program without using a declaration statement. The implicit declaration facility is provided to comply with earlier definitions of the Fortran language and can cause programming problems unless handled carefully.

It is possible, and advisable, to disable this feature by including the statement:

```
IMPLICIT NONE
```

at the start of each program. This forces a programmer to declare all variables that are used, and means that some potential errors may be identified during compilation.

If implicit typing is permitted then variables are have a data type according to the initial letter of their name: those beginning with I, J, K, L, M and N being integers; and those beginning A to H and O to Z being reals.

2.3 KIND type

Each data type has one or more values of a KIND type parameter associated with it. Data types with different KIND type values use a different number of bytes to store information. This means that numeric data types with different KIND type parameters have a different range of possible values and/or different levels of numerical accuracy.

For example, the NAG compiler (used in the development of this course) has three values of the KIND type parameter for the INTEGER type (KIND=1, 2 or 3); 3 is the default. Variables are declared with the desired precision by using the KIND attribute:

```
type(KIND = kind_type_value) [, attributes...] :: variable list
```

For Example:

```
INTEGER :: a                  !default KIND=3
INTEGER(KIND=3) :: b          !default
INTEGER(KIND=1) :: c          !limited precision -127 <= c <= 127
INTEGER(2) :: d               !KIND= is optional
INTEGER :: e=1_2              !e=1 and is of kind type 2
```

Both INTEGER, and LOGICAL data types have several possible KIND type values, each of which uses less memory than the default (which in the case of INTEGER types leads to smaller possible range of values - so beware!). These alternative KIND values are usually used only when data storage is at a premium. It is unusual for the CHARACTER data type to have more than one KIND value.

The REAL (and COMPLEX) data type has two KIND values. The default (KIND=1) has a lower level of precision than (KIND=2). (The two values of KIND are analogous to Fortran 77's single and double precision variables.) It is common place to use a REAL of KIND=2 to store higher levels of precision and / or when a large range of values are anticipated, i.e.:

```
REAL :: a                    !default KIND=1
REAL(KIND=2) :: b, c         !larger range and/or precision
COMPLEX(KIND=2) :: d         !larger range and/or precision
```

The exact level of precision and possible range of values can be checked by using *intrinsic functions* (these are self contained, often used blocks of code included in the language to help the programmer). The intrinsic functions RANGE(), HUGE(), PRECISION(), etc. all give information about the limits of variables of the different KIND types. Below are some examples from the NAG compiler:

```
INTEGER :: a            !default KIND=3
INTEGER(KIND=2) :: b
REAL :: c               !default KIND=1
REAL(KIND=2) :: d       !larger range and/or precision

HUGE( b )            !largest number = 32767
HUGE( c )            !largest number = 3.4028235E+38
HUGE( d )            !largest number = 1.7976931348623157*10**308

RANGE( a )           !largest exponent = 9
RANGE( d )           !largest exponent = 307

PRECISION( c )       !precision (in digits) = 6
PRECISION( d )       !precision (in digits) = 15
```

2.3.1 Portability

The number and value(s) of all KIND parameters depend on the compiler being used. You should be aware that explicitly using a value (like (KIND=3)) in a program may not work (or worse give unexpected errors) when using different compilers. For example some compilers use KIND=1, 2 and 3 for the INTEGER data type while others use KIND=1, 2 and 4.

One way around having to edit programs for different KIND values is to use the SELECTED_REAL_KIND() and SELECTED_INTEGER_KIND() intrinsic functions. For integers SELECTED_INTEGER_KIND() is supplied with the maximum exponent of the data to be stored (i.e 10^r where r is the range), the function returns the associated KIND type parameter for that compiler. For reals SELECTED_REAL_KIND() is supplied with both the range and the decimal precision and again returns the associated KIND value.

```
INTEGER, PARAMETER :: k2 = SELECTED_REAL_KIND(10,200)
REAL(KIND=k) :: a            !range= 10**200, 10 decimal places
INTEGER, PARAMETER :: k5 = SELECTED_INTEGER_KIND(5)
INTEGER(KIND=k5) :: b                    !range= 10**5
```

If the compiler cannot support the requested range and / or level of precision the SELECTED_type_KIND() functions return a value of -1, and the program will not compile.

2.3.2 Type conversion

When assigning one variable type to another (or variables of the same type but with different KIND types), care must be taken to ensure data remains consistent. When assigning data to different types (e.g. assigning reals to integers) or when assigning data to different kind types (e.g. assigning a 4 byte INTEGER to a single byte INTEGER), there is the potential for information to be lost. There are a number of intrinsic functions which handle the conversion of data in a reliable and consistent fashion. For example,

```
INTEGER :: total=13, num=5, share
share = total/num                    !total/num=2.6, share=2
share = INT( total/num )             !total/num=2.6, share=2
```

The result of total/num is a real number, therefore this is an example of a REAL being assigned to an INTEGER. Note the value assigned to share will be truncated (not rounded to the nearest!). The intrinsic function INT( ) converts its argument (i.e. the result of total/num) into an integer, and is assumed to be present whenever a numeric non-integer is assigned to a variable of INTEGER type.

Other data types have similar type conversion functions; REAL( ) converts the argument to type REAL, CMPLX( ) converts its argument to type COMPLEX (often truncating the imaginary part). It is possible to convert data from CHARACTER to INTEGER (and vice versa) using the intrinsic functions like IACHAR( ) and ACHAR( ), see later.

To allow the conversion between different KIND types (as well as different data types) each of the conversion function may be supplied with a KIND type value which is to be the KIND type of the converted data. For example:

```
INTEGER, PARAMETER :: k2=SELECTED_INT_KIND(2)
INTEGER :: long                                !default kind=3
INTEGER(KIND=K2) :: short
REAL(KIND=2) :: large
long = 99
short = INT( long, KIND=k2 )    !convert 99 to INTEGER(KIND=k2)
large = REAL( short, KIND=2 )   !convert 99 to REAL(KIND=2)
```

Beware! When converting data from one type to another the variable receiving the data must be capable of storing the value (range and/or precision). If not error can occur:

```
INTEGER(KIND=1) :: short        !-127 <= short <= 127
INTEGER :: long=130             !-32767 <= long <= 32767
short = long                    !error
short = INT( long, KIND=1 )     !still an error
```

2.4 Arithmetic expressions

Numerical variables, parameters and literal constants may be combined using the operators + (addition), - (subtraction), * (multiplication), / (division) and ** (exponentiation), and assigned using the assignment operator =. For example:

```
estimate_cost = cost * number
actual_cost = cost * number + postage
sum = 10 + 3
circumference = 2 * pi * radius
```

The arithmetic expression may also include brackets which should be used to clarify the required sequence of operations in an expression. For example:

```
y = 1+x/2
```

might be interpreted as either ‘add 1 to x and divide the result by 2’ or ‘add one to half of x’. The use of brackets can make this clear:

```
y = 1+(x/2)
y = (1+x)/2
```

Any expression which appears inside brackets is always evaluated first. In expressions which contain more than one operator the operations are carried out (working from the left to right in the expression) in an order which is determined by the *operator precedence*. Operators are evaluated in the following order:

- Bracketed expressions, `( . . . )`.
- Exponentiation, `**`.
- Multiplication and/or division, * or /.
- Addition and/or subtraction, + or -.

Operators of equal precedence are evaluated working from left to right across the expression, e.g.

```
area = pi*radius**2          !pi*radius*radius
area_not = (pi*radius)**2    !pi*radius * pi*radius
```

2.5 Comments

All programs should have a textual commentary explaining the structure and meaning of each section of the program. All characters appearing on a line to the right of the ! character are ignored by the compiler and do not form any part of the executable program. The text appearing after a ! character is referred to as a comment and this feature should be used to explain to the reader of a program what the program is trying to achieve. This is particularly important if the program will have to be altered sometime in the future.

```
area = pi*radius*radius         !Calculate the area of circle
```

Comments are also used to inhibit the action of statements that are used to output intermediate values when testing a program but which may be required again. The following statement is said to be ‘commented out’ and is not executed.

```
! WRITE (6,*) temp, radius*radius
```

2.6 Program Layout

A sample program:

```
PROGRAM circle_area
   IMPLICIT NONE
   !reads a value representing the radius of a circle,
   !then calculates and writes out the area of the circle.

   REAL :: radius, area
   REAL, PARAMETER :: pi=3.141592

   READ (5,*) radius
   area = pi*radius*radius      !calculate area
   WRITE (6,*) area

END PROGRAM circle_area
```

There are a number of points to note in this program:

- The program starts with a program statement in which the program is given a name, i.e. `circle_area`. The program is terminated with an `END PROGRAM` statement (which is also named). All statements in the program are indented to improve readability.
- There is an explanation of the program, both at the start and in the main body of code, in the form of comment statements. Again this improves readability.
- The `IMPLICIT NONE` statement comes first in the program followed by variable declarations. The variable declarations are grouped together and appear before all *executable* statements such as assignments statements and input/output statements.
- Blank lines are used to emphasize the different sections of the program, again for readability.

In general programs should be laid out with each statement on one line. However, there is an upper limit of 132 characters per line, (depending on the editor used it is often more convenient to keep to a maximum of 80 characters per line) a statement which exceeds the line limit may be continued on the next line by placing an ampersand & at the end of the line to be continued. The line should not be broken at an arbitrary point but at a sensible place.

```
WRITE (6,*)temp_value, pi*radius*radius, &
           length, breadth
```

More than one statement may be placed on one line using a semicolon(;) as a statement separator.

```
length=10.0; breadth=20.0; area=length*breadth
```

This is not recommended as it can lead to programs which are difficult to read - a statement may be overlooked.

2.7 Derived Data Types

2.7.1 Definition and specification

In many algorithms there are data objects which can be grouped together to form an aggregate structure. This might be for readability, convenience or sound programming reasons. A circle, for example may have the following properties:

```
radius
area
```

A programmer may define special data types, known as *derived* data types, to create aggregate structures. A circle could be modelled as follows:

```
TYPE circle
   INTEGER :: radius
   REAL :: area
ENDTYPE circle
```

This would create a template which could be used to declare variables of this type

```
TYPE(circle) :: cir_a, cir_b
```

A derived data type may be constructed from any number or combination of the intrinsic data types (or from other already defined derived data types). The general form of the `TYPE` statement is:

```
TYPE :: name
   component definition statements
   ...
END TYPE [name]

TYPE(name) [,attribute] :: variable list
```

Note that the type name is optional on the ENDTYPE statement but should always be included to improve program clarity.

Just like the intrinsic data types, the components of a derived data type may be given an initial value. For example:

```
TYPE (circle) :: cir=circle(2,12.57)
```

The derived type is so named because it is derived from the intrinsic types, such as REAL and INTEGER. However derived types may be used in the definition of other derived types. For example, if a type, point, is defined by:

```
TYPE point
   REAL :: x, y
ENDTYPE point
```

then the previously defined type, circle, could be modified to include a spacial position:

```
TYPE circle
   TYPE (point) :: centre
   INTEGER :: radius
   REAL :: area
ENDTYPE circle
```

Including one statement inside another block of statements is called *nesting*.

2.7.2 Accessing Components

The elements of a derived type may be accessed by using the variable name and the element name separated by the % character, as follows:

```
cir_a%radius = 10.0
cir_a%area = pi * cir_a%radius**2
```

If a derived type has an element which is a derived type then a component may be accessed as follows:

```
cir_a%position%x = 5.0
cir_a%position%y = 6.0
```

Components of a derived type may appear in any expressions and statements that their data type allow. It is also possible to assign one instance of a derived type to another instance of the same derived type. The = operator is the only operator that may be applied to a derived type variable, all other operations require the programmer to explicitly access component by component.

```
cir_a%radius = cir_b%radius
cir_a%area = cir_b%area
cir_a%position%x = cir_b%position%x
cir_a%position%y = cir_b%position%y
cir_a = cir_b                      !shorthand for all the above

cir_a = cir_b * 2                  !illegal
```

2.8 Exercises

1. Write a program which declares variables to hold the following data:

(a) an integer set to zero.

(b) an integer set to minus one.

(c) 64.0

(d) -1.56×10^{12} (this should be written as `-1.56E12`)

Check the program by writing out variables to the screen.

2. Which of the following are invalid names in Fortran 90 and why?

```
abignumber    thedate    A_HUGE_NUMBER
Time.minutes  10times    Program
1066          X          HELP!
f[t]          no way     another-number
```

3. Given the following variable declarations and assignments evaluate the subsequent expressions and state the value and type of each result. Check your results by writing a program to write out the results of the expressions. Finally, insert brackets to clarify the meaning of these expressions according to operator precedence.

```
REAL :: x=10.0 y=0.01, z=0.5
INTEGER :: i=10, j=25, k=3

i + j + k * i
z * x / 10 + k
z * k + z * j + z * i
i * y - k / x + j
x / i / z
```

4. Write definitions of derived types, together with initial values, which represent the following:

(a) a point with `x`, `y` and `z` coordinates.

(b) a time in hours, minutes and seconds.

(c) a date in day, month and year.

(d) a time comprised of the two types above.

5. Write a program which will read in two real numbers representing the length and breadth of a rectangle, and will print out the area calculated as length times breadth. (Use a derived type to represent the rectangle and its area.)

6. Write a program which will read in five integers and will output the sum and average of the numbers.

Note: the values held by a program variable can be read from and written to the screen using the `READ()` and `WRITE()` statements (which are explained later in the course), i.e.

```
READ(*,*) variable1 [, variable2]
WRITE(*,*) variable1 [, variable2]
```

3 Character Processing

3.1 Character Type

In the previous chapter the intrinsic numeric types REAL and INTEGER were introduced, a third intrinsic type CHARACTER is presented in this section. This type is used when the data which is being manipulated is in the form of single characters and strings (words or sentences) rather than numbers. Character handling is very important in numeric applications as the input or output of undocumented numbers is not very user friendly.

In Fortran characters may be treated individually or as contiguous strings. Strings have a specific length and individual characters within the string are referred to by position, the left most character at position 1, etc. As with numeric data the programmer may specify literal constants of intrinsic type character as described below.

3.2 Character Constants

The example below is taken from a program which calculates the area of a circle, the program reads in a value for the radius and writes out the area of the circle. Without prompts the user‘s view of such a program is very bleak, that is there is no indication of what the input is for or when it should be supplied nor is there an explanation of the output. By including some character constants (or literals) in the output the user’s view of the program can be greatly enhanced, for example

```
WRITE (*,*) ‘Please type a value for the radius of a circle’
READ (*,*) radius
area = pi*radius*radius
WRITE (*,*) ‘The area of a circle of radius ‘, radius, &
          ‘ is ‘, area
```

The characters which appear between pairs of apostrophes are character constants and will appear on screen as

```
Please type a value for the radius of a circle
12.0
The area of a circle of radius 12.0 is 452.38925
```

The double quote character may also be used to define character literals. If a string of characters is to contain one of the delimiting characters (apostrophes or double quotes) then the other may be used. However if the string is to contain both delimiting characters or a programmer wishes to always define strings using the same character then the delimiter may appear in a string as two adjacent apostrophes or double quotes. These are then treated as a single character.

```
“This string contains an apostrophe ‘.”
‘This string contains a double quote “.‘
“This string contains an apostrophe ‘ and a double quote ““.”
```

This would appear in output as

```
This string contains an apostrophe ‘.
This string contains a double quote “.
This string contains an apostrophe ‘ and a double quote “.
```

3.3 Character Variables

The declaration of character variables is similar to that for REAL and INTEGER variables. the following statement declares two character variables each of which can contain a single character

```
CHARACTER :: yesorno, sex
```

A value may be assigned to a character variable in the form of a literal constant thus

```
yesorno = ‘N’
sex = ‘F’
```

However character variables are more frequently used to store multiple characters known as strings. For example to store a person’s name the following declarations and assignments may be made

```
CHARACTER(LEN=12) :: surname, firstname
CHARACTER(LEN=6) :: initials, title
title = ‘Prof.‘
initials = ‘fjs‘
firstname = ‘Fred‘
surname = ‘Bloggs‘
```

Notice that all the strings were defined as being long enough to contain the literal constants assigned. Variables which have unused characters are space filled at the end. If the variable is not large enough to contain the characters assigned to it then the left-most are used and the excess truncated, for example

```
title = ‘Professor‘
```

would be equivalent to

```
title = ‘Profes‘
```

The general form of a character declaration is:

```
CHARACTER [(LEN= )] [,attributes] :: name
```

3.4 Character manipulation

3.4.1 Concatenation

The arithmetic operators such as + and - should not be used with character variables. The only operator for character variables is the concatenation operator / /. This may be used to join two strings as follows

```
CHARACTER (len=24) :: name
CHARACTER (len=6) :: surname
surname = ‘Bloggs’
name = ‘Prof ‘//‘ Fred ‘//surname
```

As with character literals if the expression using the / / operator exceeds the length of the variable the right-most characters are truncated and if too few characters are specified the right-most characters are filled with spaces.

3.4.2 Substrings

As the name suggests substrings are sections of larger character strings. The characters in a string may be referred to by position within the string starting from character 1 the left-most character.

```
CHARACTER (LEN=7) :: lang
lang = ‘Fortran’
WRITE (6,*) lang(1:1), lang(2:2), lang(3:4), lang(5:7)
```

would produce the following output

```
Fortran
```

The substring is specified as (start-position : end-position). If the value for start-position is omitted 1 is assumed and if the value for end-position is omitted the value of the maximum length of the string is assumed thus, lang(:3) is equivalent to lang(1:3) and lang(5:) is equivalent to lang(5:7).

The start-position and end-position values must be integers or expressions yielding integer values. The start-position must always be greater than or equal to 1 and the end-position less than or equal to the string length. If the start-position is greater than the maximum length or the end-position then a string of zero length is the result.

3.4.3 Intrinsic Functions

Functions will be dealt with in more depth later in the course, however it is convenient to introduce some functions at this early stage. An intrinsic function performs an action which is defined by the language standard and the functions tabulated below relate to character string. These intrinsic functions perform a number of commonly required character manipulations which programmers would otherwise have to write themselves:

- `LEN(string)` returns the length of a character string
- `INDEX(string,sustring)` finds the location of a substring in another string, returns 0 if not found.
- `CHAR(int)` converts an integer into a character
- `ICHAR(c)` converts a character into an integer
- `TRIM(string)` returns the string with the trailing blanks removed.

The conversion between characters and integers is based on the fact that the available characters form a sequence and the integer values represent the position within a sequence. As there are several possible character sequences and these are machine dependent the precise integer values are not discussed here. However, it is possible to state that regardless of the actual sequence the following are possible:

```
INTEGER :: i
CHARACTER :: ch
   ...
   i=ICHAR(CHAR(i))
   ch=CHAR(ICHAR(ch))
```

Below is an example of how intrinsic functions may be used:

```
CHARACTER(len=12) :: surname, firstname
INTEGER :: length, pos
   ...
   length = LEN(surname)           !len=12
   firstname = ‘Walter‘
```

```
pos = INDEX(firstname, 'al')    !pos=2
firstname = 'Fred'
pos = INDEX(firstname, 'al')    !pos=0
length = LEN(TRIM(firstname))   !len=4
```

3.5 Exercises

1. Given the following variable declaration and initialization:

```
CHARACTER(len=5) :: vowels='aeiou'
```

what are the substrings specified below?

(a) vowels(1:1)

(b) vowels(:2)

(c) vowels(4:)

(d) vowels(2:4)

2. Given the following variable declaration and initialization:

```
CHARACTER(len=27) :: title='An Introduction to Fortran.'
```

define substrings which would specify the character literals below?

(a) `to`

(b) `Intro`

(c) `Fortran.`

3. Using the variable title defined above write a program using intrinsic functions, which would:

(a) find the location of the string `duct`

(b) find the length of the string

(c) extract and concatenate substrings to produce the string `Fortran, An Introduction to.`

In all cases, output the results.

4. Design a derived data type which contains the following details relating to yourself: surname, forename, intials, title and address. The address should be a further derived type containing house number, street, town/city and country.

5. Write a program which will request input corresponding to your name and address as defined in the text and which will output your name and address in two forms as follows:

```
Mr. Joseph Bloggs,
12, Oil Drum Lane,
Anytown,
United Kingbom

JF Bloggs, 12 Oil Drum Lane, Anytown
```

4 Logical & comparison expressions

4.1 Relational operators

A logical variable, denoted with the keyword LOGICAL to define its type, can take one of two logical values (.TRUE. or .FALSE.) which are used to record Boolean information about the variable. Recall that declaring logical variables takes the following form:

```
LOGICAL [, attribute] :: variable
```

Logical variable may be assigned a value either explicitly or via a logical expression, for example:

```
LOGICAL :: guess, date
LOGICAL, PARAMETER :: no = .false.
INTEGER :: today_date
...
guess = .true.
date = (today_date==5)
```

if today_date has previously been assigned a value and that value is 5 then date holds .TRUE., otherwise .FALSE.. The relational operator == may be read as 'equal to', thus today_date==5 is read as 'is today_date equal to 5?'. Below are a list of the relational operators together with their meaning:

<	less than,
<=	less than or equal to,
>	greater than,
>=	greater than or equal to,
==	equal to,
/=	not equal to.

Below are some examples of how the relational operators can be used:

```
LOGICAL :: test
INTEGER :: age, my_age
CHARACTER(LEN=5) :: name
...
test = 5 < 6       !True
test = 5 > 6       !False
test = 5 == 6      !False
test = 5 /= 6      !True
test = 5 <= 6      !True
test = 5 >= 6      !False
...
test = age > 34            !a variable compared with a constant
test = age /= my_age       !two variables are compared
```

```
test = 45 == my_age        !a variable can appear in any side
test = name == 'Smith'     !characters are allowed
test = (age*3) /= my_age   !expressions are allowed
```

4.2 Logical expressions

Expressions containing logical variables and / or relational operators may be combined into logical expressions using the following operators:

`.AND.`	logical intersection,
`.OR.`	logical union,
`.NOT.`	logical negation,
`.EQV.`	logical equivalence,
`.NEQV.`	logical non-equivalence,

The logical intersection operator `.AND.`, requires two expressions or variables and gives a `.TRUE.` result only if both expressions are true, otherwise evaluates to `.FALSE.`. Consider the following example:

```
LOGICAL :: test, employed=.true.
INTEGER :: age=50
...
test = employed .AND. (age<45)  !test=.false.
```

There are two sub-expressions here, one `.TRUE.` the other `.FALSE.` hence the result is `.FALSE.`.

The logical union operator `.OR.` requires two expressions or variables and gives the value `.TRUE.` if either one of the expressions is true, and `.FALSE.` otherwise. Consider the following example:

```
LOGICAL :: test
CHARACTER(LEN=10) :: name = ‘James’
...
test = (name='Dimitris') .OR. (name='James') !test=.true.
```

The logical negation operator .NOT. is used to invert the logical value of an expression, i.e. `.TRUE.` becomes `.FALSE.` and vice versa. For example:

```
INTEGER :: big=100, small=2
LOGICAL :: test
...
test = .NOT. (big>small)      !test=.false.
test = .NOT. test             !test=.true.
```

where the statement enclosed by the (optional) brackets is assigned a value which in turn is inverted.

The logical equivalence operator .EQV. is used to check if all variables or expressions have the same logical value (can be either `.TRUE.` or `.FALSE.`). If both values are the same the whole expression evaluates to `.TRUE.`, otherwise it evaluates to `.FALSE.`. For example:

```
LOGICAL :: test
...
test = (5*3>12) .EQV. (6*2>8)   !test=.true.
test = (5*3<12) .EQV. (6*2<8)   !test=.true.
```

both statements evaluate to .TRUE. because the sub-expressions in each statement take the same logical values.

The logical non-equivalence operator .NEQV. is used to evaluate expressions to .TRUE. only if one of the expressions has a different logical value to the other(s), otherwise evaluates to .FALSE.. For example:

```
LOGICAL :: test
...
test = (5*3>12) .NEQV. (6*2>13)    !test=.true.
test = (5*3>12) .NEQV. (6*2<13)    !test=.false.
```

the first expression has one true and one false component and therefore evaluates to .TRUE., the second expression has two true components and therefore evaluates to .FALSE..

When comparing REAL with INTEGER values the compiler will convert the integer to type REAL. Comparing REAL with REAL values must be performed with caution; rounding errors in the precision of a REAL variable may mean that two REAL numbers should never be equated. It is advisable to test their difference rather than their actual values, i.e.

```
REAL :: a=100.0, b
LOGICAL :: equal
b = 2*50.0
equal = (a==b)          !potential error
equal = (a-b)<0.0005    !good programming practice
```

4.3 Character Comparisons

Certain rules have to be obeyed when comparing characters or character strings, (Is A greater than B?) Most importantly when one of the character strings has a shorter length, it is padded with blanks (right side). The comparison is character by character

The comparison starts from the left side The comparison terminates either when a difference has been found or the end of the string has been reached. If no difference is found the character strings are the same, otherwise the comparison terminates with the first encountered difference. Comparing character strings depends on the *collating sequence* of the machine used. The ASCII collating sequence has the following rules:

blank 0 < 1 < 2 ... < 9 A < B < ... < Z a < b < ... < z

that is blank before digits before a to z before A to Z. The rest of characters have no defined position and are machine dependant. The ASCII character set is the most commonly used collating sequence. Note that the Fortran standard does not define if upper case characters come before or after lower case characters.

The earlier a character comes in the collating sequence the smaller value it has. Hence, a blank is always smaller than a digit or a letter. An example:

```
'Alexis' > 'Alex'
```

The right string is shorter, hence 'Alex' becomes 'Alex ' The first 4 letters are the same - no difference has been found so search continues character i is greater than ‘blank’ - comparison terminates and the result is .TRUE. because the blank comes before the letter i in the collating sequence!

4.3.1 Portability Issues

The collating sequence is machine dependable. Intrinsic functions for string comparison are available which are based on the universal ASCII collating sequence:

```
LGT(string1, string2)      !greater  than or equal to
LGE(string1, string2)      !greater than
```

```
LLE(string1, string2)      !less than or equal to
LLT(string1, string2)      !less than
```

Because the collating sequence might differ from machine to machine the above intrinsic functions (based on the ASCII collating sequence) should be used to compare strings. More intrinsic functions are available. For example intrinsic functions that identify the position of a character in a sequence in the ASCII or machine collating sequence. Some of them are presented through the exercise sections.

4.4 Exercises

1. Given the values below, what is the value of each of the following expressions? Write a program to test your answers.

```
INTEGER :: age=34, old=92, young=16
age /= old
age >= young
age = 62
(age==56 .AND. old/=92)
(age==56 .OR. old/=92)
(age==56 .OR. (.NOT.(old/=92)))
.NOT. (age==56 .OR. old/=92)
```

2. What are the logical values of the following expressions?

```
15 > 23
(12+3) <= 15
(2>1) .AND. (3<4)
((3>2) .AND. (1+2)<3) .OR. (4<=3)
((3>2) .AND. (1+2)<3) .EQU. (4<=3)
```

3. Simplify the following expressions using different logical operators:

```
.NOT. (a<b .AND. b<c)
.NOT. (a<b .EQV. x<y)
```

4. Determine the logical value of each of the following expressions. Write a program to test your answers.

```
CHARACTER(LEN=4) :: name = 'Adam'
name > 'Eve'
"ADAM" > name
'M1' < 'M25'
'version_1' > 'version-2'
' more' < 'more'
LGT("Adam","adam")
LLT("Me",'me')
LLT('me',"me?"
```

5 Arrays

5.1 Terminology

5.1.1 Arrays and elements

Previous chapters introduced simple data types, such as `INTEGER`, `REAL` and `CHARACTER`, and derived data types. In this chapter a means of organising and structuring data called an *array* is introduced. An array is a collection of data, all of the same type, whose individual elements are arranged in a regular pattern.

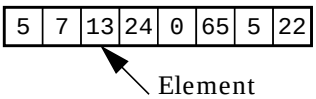
There are 3 possible types of arrays (based on how the array is to be stored in memory):

- Static arrays - are the most common type and have their size fixed when the array is declared. The number of elements in the array cannot be altered during the program's execution. This can be inflexible in certain circumstances (to change the array sizes parts of the program must be edited and the whole recompiled) and can be wasteful in terms of storage space (since the largest possible array sizes might be declare and may be only partly used).
- Semi-dynamic arrays - have their size determined on entering a sub-program. Arrays can be created to match the exact size required but can only be used within that particular sub-program. In Fortran 90 such arrays are either *assumed shape*, or *automatic* arrays.
- Dynamic arrays - the size and therefore the amount of storage used by a dynamic array can be altered during execution. This is very flexible but may slow runtime performance and lack any bounds checking during compilation. In Fortran90 such arrays are called *allocatable* arrays.

The reasons for using an array are:

- Easier to declare (one variable name instead of tens or even thousands).
- Easier to operate upon (because of whole array operations the code is closer to underlying mathematical form).
- Flexible accessing (it is easy operate on various sections (or parts) of an array in different ways).
- Easier to understand the code (notational convenience).
- Inherent data parallelism (perform a similar computation on many data objects simultaneously - if the program is running on a parallel machine).
- Optimization opportunities (for compiler designers).

Below is an example of an array containing eight integer elements:



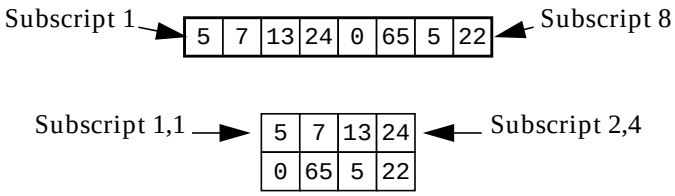
Each element has a *subscript* (or index) to identify its place in the array. Assuming that the subscript starts at 1 then:

- the first element of the array is 5 with an index of 1
- the second element of the array is 7 with an index of 2
- the last element of the array is 22 with an index of 8

5.1.2 Array properties

The *rank* (or *dimension*) of an array refers to the number of subscripts needed to locate an element within that array. A scalar variable has a rank of zero (i.e. needs no subscripts because it only has one; an array with a rank of one is called a vector; an array with a rank of 2 is called a matrix.

Consider again the following arrays:



The upper array is a vector since it is one-dimensional while the lower array is a matrix since it is two-dimensional. At most an array may have seven dimensions.

The term *bounds* refers to the lowest and highest subscript in each dimension. The vector above has a lower bound of 1 and a higher bound of 8, whereas the matrix has bounds of 1 and 2 in the first dimension and 1 and 4 in the second dimension.

The *extent* of an array refers to the number of elements in a dimension. Hence the above vector has an extent of 8, whereas the above matrix has an extent of 2 and 4 in each dimension.

The *shape* of an array is a vector (i.e. an array!) containing the extents of an array. Hence the above vector has a shape of (8) whereas the matrix has a shape of (2,4).

The term *size* refers to the total number of elements of an array, which simply is the product of extents. The size of an array may be zero (see later). Both the vector and matrix above have a size of 8.

Arrays that have the same shape are said to *conform*. This is the condition for whole array or array section operations. The above examples do not conform with one another. All arrays conform with scalar values.

5.2 Specifications

Like other variables arrays are specified with a specific data type (`INTEGER`, `REAL`, derived type, etc.). For static arrays, the rank (up to a maximum of 7) and the bounds (upper and lower) in each dimension are declared. Declaring the lower bound is optional. If the lower bound is not specified Fortran90 assumes that the index begins with 1.

Alternate and equivalent forms used to declare an array are as follows:

```
type, DIMENSION(bound) [, attribute] :: name
type [, attribute] :: name (bound)
```

where `[, attribute]` allows for the declaration of other type attributes, if required.

The following declarations are equivalent. Both declare an integer array a with 6 elements; a real array b with 10 elements and a 2-dimensional logical array yes_no.

```
INTEGER, DIMENSION(6) :: a
REAL, DIMENSION(0:9) :: b
LOGICAL, DIMENSION(2,2) :: yes_no

INTEGER :: a(6)
REAL :: b(0:9)
LOGICAL :: yes_no(2,2)
```

Use the DIMENSION attribute form when several arrays of the same bounds and type need to be declared. Use second form when several arrays of different types and/or bounds need to be declared. A third form is a mixture of the two above, as shown below:

```
type, DIMENSION(bound1) [, attribute] :: a, b(bound2)
```

where a takes the 'default' bounds bound1, but b takes another explicitly defined value bound2.

A mixture of the three forms are allowed in the same program. Some further examples are shown below:

```
INTEGER, DIMENSION(8) :: x, y, z(16)
REAL :: alpha(1:3), beta(4:9)
REAL, DIMENSION(0:5,12:45,6) :: data
CHARACTER(len=10) :: names(25)
```

The first statement declares x and y to have 8 elements each and z to have 16 elements. The second statement declares two arrays of the same type but with different bounds. The third statement declares a rank 3 array. The last statement declares an array which has 25 elements, each element being a character string of length 10.

It is possible to include arrays as components of a derived data type and to declare arrays of derived data types, for example:

```
TYPE(point)
   REAL :: position(3)
TYPE(point)
TYPE(point) :: object(10)
```

The type point is comprised of 3 real numbers, while the array object consists of 10 items of data, each consisting of 3 real numbers.

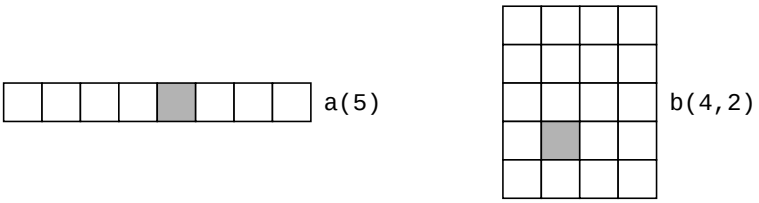
5.3 Array Sections

5.3.1 Individual elements

Individual elements and sections of an array are uniquely identified through subscripts, one per rank separated by commas. This subscript is an integer value (or an expression whose result is an integer value)

```
REAL, DIMENSION(8) :: a
INTEGER, DIMENSION(5,4) :: b
...
a(5)       !fifth element
b(4,2)     !element at intersection of the 4th row and 2nd column.
```

Subscripts (e.g. `(i,j)`) refers to the element at the intersection of row `i` and column `j`, where `i` and `j` have integer values between the upper and lower bounds for their respective dimensions.



Using expressions (e.g. `(2*k)`) refers to an element whose subscript is the result of the multiplication. The result of an expression must be an integer within the declared bounds. It is an error to refer to a subscript beyond (either above or below) the range of an array's bounds.

5.3.2 Sections

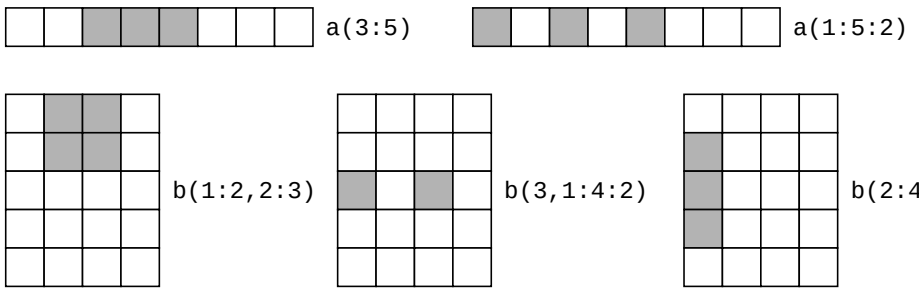
As well as identifying individual elements it is possible to reference several elements (called a *section*) with the same statement. Accessing a section of an array requires the upper and lower bounds of the section to be specified together with a stride (for each dimension). This notation is called a subscript triplet:

```
array ([lower]:[upper][:stride], ...)
```

where `lower` and `upper` default to the declared extents if missing, and `stride` defaults to 1.

```
REAL, DIMENSION(8) :: a
INTEGER, DIMENSION(5,4) :: b
INTEGER :: i=3
...
a(3:5)       !elements 3, 4, 5
a(1:5:2)     !elements 1, 3, 5
b(1:2,2:i)   !elements (1,2) (2,2) (1,3) and (2,3)
b(i,1:4:2)   !elements 1 and 3 of the third row
b(2:4,1)     !elements 2, 3 and 4 of the first column
```

The bounds in a subscript triplet can take any (integer) values. Using Subscript triplets is a compact and convenient way of referencing arbitrary sections of an array.

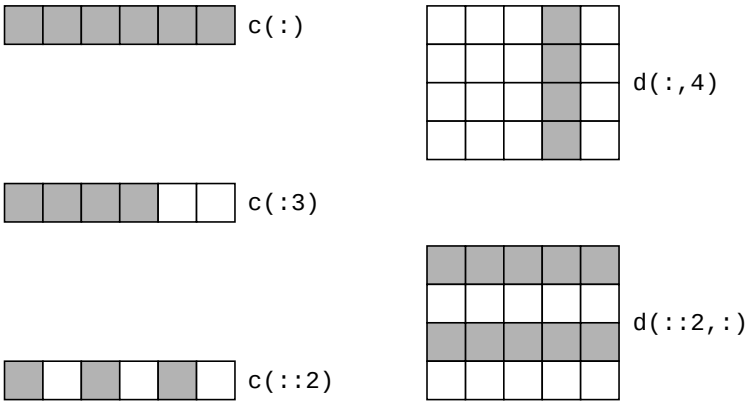


Some more complex array section are given below. Note how the upper and lower subscripts may be omitted:

```
REAL, DIMENSION(0:5) :: c
INTEGER, DIMENSION(4:5) :: d

c(:)            !whole array
c(:3)           !elements 0,1,2,3
```

```
c(::2)          !elements 0,2 and 4
d(:,4)          !all elements of the fourth column.
d(::2,:)        !all elements of every other row
```



5.4 Vector Subscripts

Vector subscripts are integer arrays of rank 1 and take the form:

```
(/ list /)
```

where `list` is a list of subscripts, in any order, to be referenced. Consider the following example:

```
REAL, DIMENSION(9) :: a
INTEGER, DIMENSION(3) :: random
   random=(/6,3,8/)              !set values for random
   a(random)=0.0                 !a(6)=a(3)=a(8)=0.0
   a((/7,8,9/))=1.2              !a(7)=a(8)=a(9)=1.2
```

For the vector subscript to be valid, list may not contain a value outside the bounds of an array in which it is used to identify elements

Care must be taken not to duplicate values in a vector subscript when used in the LHS of an expression. This would result in several values being written to the same array element:

```
REAL, DIMENSION(5) :: a
INTEGER, DIMENSION(3) :: list
list=(/2,3,2/)
a(list)=(/1.1, 1.2, 1.3/)     !illegal - element 2 set twice
a((/0,2,4/)) = 0.0            !illegal - subscript out of bounds
```

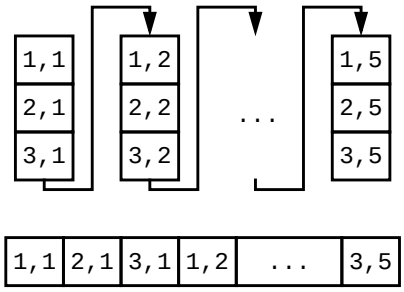
5.5 Array storage

The physical storage: How an array is stored in memory depends on the computer implementation. This is usually of little interest to a programmer.

The array element ordering: It is wrong to assume that two elements of an array are next to each other (in memory) just because their indices differ by a single value. However it is possible to imagine multi-dimensional arrays stored as a linear sequence of the array elements by counting through the ranks, lowest rank changing first. In this way matrices may be thought of as being stored column-wise. It is important to keep this in mind when manipulating and initialising multi-dimensional arrays.

Consider the following example:

```
REAL, DIMENSION(3, 5) :: a
```



The array a is stored in memory as a linear sequence, as shown.

5.6 Array Assignment

5.6.1 Whole array assignment

Whole array operations are used when the elements of an array need to be assigned with the same value (i.e. a scalar) or when coping the values of one array to another. In the former the scalar is broadcasted to all the elements of the array. In the latter array elements are equated, one with another. In all cases the operands in the array expression must conform. Consider the following example:

```
REAL, DIMENSION(100) :: a, b, c
REAL : d(10,10) = 0.0
b = 2*a+4
a = 2.0
c = b*a
c = d                    !illegal - arrays do not conform
```

The first assignment involves an array expression on the right hand side. Since a and b conform it is a valid statement. Each element of b takes the corresponding value of a multiplied by 2, plus 4.

The second assignment involves a scalar on the right hand side, (recall that scalars conform with arrays of all shapes. Each element of a takes the value of 2.

The third assignment involves an array product on the right hand side. Since a and b conform then their product can be evaluated, the product is an element by element multiplication (not a matrix multiplication!). The result is another array which conforms with c, therefore each element of c is the product of the corresponding elements in a and b.

5.6.2 Array section assignment

Just as whole arrays may appear in expressions, so to can array sections. Again all array sections in an expression must conform to one another. Consider the following example:

```
REAL, DIMENSION(10) :: alpha, beta
REAL :: gamma(20)
...
alpha(1:5) = 2.0             !first 5 elements all 2.0
alpha(6:) = 0.0              !last 5 elements all 0.0
...
beta(1:10:2) = alpha(1:5)/6  !conforming array sections
alpha(2:10) = alpha(1:9)
gamma(11:20) = beta
```

The last three statements all have conforming array sections of various sizes (5, 9 an 10 element respectively). The first of these sets every other element of beta to the first five elements of alpha (`beta(1)=alpha(1)`, `beta(3)=alpha(2)`, etc.). The second shows a powerful operation using arrays where values are shifted automatically and without the need of `DO` loops. Elements 2,3,...10 of `alpha` take the value of elements 1,2,...9, thereby shifting values along in the array. The last assignment demonstrates another important concept. Whereas the arrays `beta` and `gamma` do not conform, the section of `gamma` used in the expression does conform with `beta`, hence the expression is a valid statement.

5.6.3 Renumbering

It is important to remember that the elements in an array section always have a lower bound of 1 in each dimension. Regardless of the subscript of the element(s) in the original array, elements of a section are renumbered so that indices are consecutive (in each dimension) beginning at 1. Renumbered is automatic.

```
INTEGER :: nums(10), i
nums = (/ 1,3,5,7,9,2,4,6,8,0 /)
...
i = MAXLOC( nums )         !i=5, element 9 is maximum
i = MAXLOC( nums(1:5) )    !i=5, last element in section =9
i = MAXLOC( nums(3:8) )    !i=3, third element in section =9
i = MAXLOC( nums(::2) )    !i=3, third element in section =9
```

In the above example, the element with the value 9 is always the maximum element in the array or section. However its index changes due to a renumbering of the section

5.6.4 Elemental intrinsic procedures

Elemental procedures are specified for scalar arguments, but may take array arguments. Consider the following example:

```
REAL :: num, root
REAL, DIMENSION(3,3) :: a
INTEGER ::length(5)
CHARACTER(LEN=7) :: c(5)
...
root = SQRT(num)
a = SQRT(a)
length = LEN( TRIM(c) )    !length=6
```

The function `SQRT()` returns the square root of its argument. If the argument is a scalar variable, a single value is returned. If the argument is an array, an array of square roots is returned. Hence, every element of `a` is substituted by the square root of the existing value.

The third assignment finds the string length for each element of `c` and rejects any trailing blanks. Hence, if `c(1)` is '`Alexis  `' the `TRIM()` function returns '`Alexis`' (i.e. minus the trailing blank), the length of which is 6.

Many of the intrinsic functions (including the mathematical functions) may take arrays, array sections or simple scalar variables as arguments.

5.7 Zero-sized arrays

Fortran 90 allows arrays to have zero size. This occurs when the lower bound is greater than the upper bound. A zero-sized array is useful because it has no element values, holds no data, but is valid and always defined. Zero-sized arrays allow the

handling of certain situations without the need of extra code. As an example consider the following situation:

```
INTEGER :: a(5)=(/1,2,1,1,3/)
...
a(1:COUNT(a==1))=0    !a=(/ 0,0,0,1,3 /)
a(1:COUNT(a==4))=1    !a unchanged
```

The first statement initialises `a` using a constructor (see below). The function `COUNT()` returns the number of elements which have a value equal to 1 and 4 respectively. The first statement sets `a(1:3)` to zero, but the second does nothing (because no element of `a` has the value 4, and the array section `a(1:0)` is of zero-size).

Allowing for zero-sized arrays means that if an array or section has no elements the statement becomes a ‘do nothing’ statement and the programmer is saved from having to deal with the problem.

5.8 Arrays and derived types

As well as specifying arrays of intrinsic data types, it is also possible to include array specifications as part of a derived data type. This makes it easier to group together several (or many) instances of data. Recall the definition of the derived data type `circle`:

```
TYPE(circle)
   INTEGER :: radius
   REAL :: area
END TYPE circle
```

Previously, a second derived type called `position` was used to store the real coordinates of the circles centre; `position` had two real numbers as components. It is possible to replace the use of the derived type position by a real array:

```
TYPE(circle)
   REAL, DIMENSION(2) :: pos
   INTEGER :: radius
   REAL :: area
END TYPE circle
```

Here `pos(1)` holds say an x coordinate while `pos(2)` holds a y coordinate. Arrays can be used when the number of components becomes to large (or just inconvenient). Array components are referenced in much the same way as other components:

```
TYPE(circle) :: first
...
first%pos(1)          !element 1 of pos
first%pos(1:)         !whole array (section)
```

Just as several (or many) instances of an intrinsic data type may be grouped together as a single array, so it is possible to group together instances of derived data types. For example:

```
TYPE(circle), DIMENSION(100) :: all_circles
...
all_circles(1)%radius         !radius of circle 1
all_circles(51:100)%radius    !radius of last half of circles
all_circles(1:100:2)%area     !area of every other circle
all_circles(:)%pos(1)         !x coords of every circle
all_circles%pos               !all coords of all circles
all_circles(10)%pos(:)        !both coords of circle 10
```

An array of a derived data type and/or an array component of a derived type have the same requirements (i.e. conformance in expressions, etc.) and restrictions as other arrays in Fortran 90. For example:

```
TYPE(circle), DIMENSION(100) :: cir
...
cir(1:10) = cir(91:100)      !sections of derived type conform
cir(i)%pos = cir(i-1)%pos(:) !arrays of reals conform
cir%pos(1:2) = cir(1:2)%pos  !error, cir=cir(1:2) non-conforming
```

Care must be taken to ensure that any labelling of array sections is applied to the correct part of an expression.

5.9 Initialising arrays

5.9.1 Constructors

Constructors are used to initialise 1-dimensional arrays which require fixed values at the start of a program. A constructor is a list enclosed in parentheses and back-slash. The general form is:

```
array = (/ list /)
```

where `list` can be one of the following:

- a list of values of the appropriate type:

  ```
  INTEGER :: a(6)=(/1,2,3,6,7,8/)
  ```

- variable expression(s)

  ```
  REAL :: b(2)=(/SIN(x),COS(x)/)
  ```

- array expression(s)

  ```
  INTEGER :: c(5)=(/0,a(1:3),4/)
  ```

- implied DO loops (see later)

  ```
  REAL :: d(100)=(/REAL(i),i=1,100/)
  ```

The constructor can be used during declaration as shown above or in a separate statement. Arrays with a rank of two or more should not be initialise with a simple constructor, but instead should use a combination of constructor(s) and the RESHAPE( ) function (see below).

5.9.2 Reshape

The RESHAPE( ) function is to be used for the initialisation or assignment of multi-dimensional arrays, i.e., arrays with rank greater than 1. It can be used on a declaration statement or in a separate statement. The format is:

```
RESHAPE (list, shape [,PAD] [,ORDER])
```

where `list` is a 1-dimensional array or constructor containing the data, and `shape` a 1-dimensional array or vector subscript containing the new shape of the data. PAD is an array containing data to be used to pad out the data in `list` to the required shape. ORDER can be used to change the order in which data is reshaped.

The size of the array determines the dimension of the new array. The elements determine the extent of each dimension. Consider the following example:

```
INTEGER, DIMENSION(3,2) :: a
a=RESHAPE((/0,1,2,3,4,5/),(/3,2/))   !put values into a
```

RESHAPE( ) will generate a rank 2 array with extents 3 and 2 from the list of values in the constructor. Since this array conforms with the array a, whole array assignment is used to give each element of a the required value. Unless the ORDER argument is used values from the constructor will be returned in array element order, i.e. a(1,1)=0, a(2,1)=1, a(3,1)=2, a(1,2)=3, etc...

5.9.3 DATA statement

Use the DATA when other methods are tedious and / or impossible. For example for more than one array initialisation or for array section initialisation.

The format is:

```
DATA variable / list / ...
```

For example see following:

```
INTEGER :: a(4), b(2,2), c(10)
DATA a /4,3,2,1/
DATA a /4*0/
DATA b(1,:) /0,0/ DATA b(2,:)/1,1/
DATA (c(i),i=1,10,2/5*1/ DATA (c(i),i=2,10,2)/5*2/
```

The first DATA statement uses a list by value where the value for each array element is explicitly declared. The second DATA statement uses a list by whole array where 4 is the size of the array and 0 is the required value. Do not confuse with the multiplication operator. The third and fourth statements use a list by section where the first row takes values 0 and 0 and the second row takes the values of 1 and 1.

The last two statements use a list by implied DO loops (see later) where the odd indexed elements are assigned the value 1 and the even indexed elements take the value of 2.

Remember that:

- a DATA statement can split in more than one line but each line must have a DATA keyword.
- may be used for other variables as well as arrays.

5.10 WHERE

A WHERE statement is used to control which elements of an array are used in an expression, depending on the outcome of some logical condition. It takes a statement or a construct form. The WHERE statement allows the expression on an element by element basis only if a logical condition is true. The syntax is as follows:

```
WHERE (condition) expression
```

Consider the following situation:

```
INTEGER :: a(2,3,4)
WHERE( a<0 ) a = 0
WHERE( a**2>10 ) a = 999
WHERE( a/=0 ) a = 1/a
```

The first WHERE statement results in all negative values of a being set to zero, the non-negative values remain intact. The second WHERE statement results in elements of data being set to 999 if their square is greater than ten. The third statement calculates

the reciprocal of each element of data, except those with a value of zero (hence avoiding the error 'divide by zero').

The WHERE construct allows array assignment(s) (again on an element by element basis) only if a logical condition is true, but may provide alternative array assignment(s) if false. The syntax is as follows:

```
WHERE (condition)
   block1
[ELSEWHERE
   block2]
ENDWHERE
```

Look at the following section of code.

```
INTEGER :: btest(8,8)
...
WHERE ( btest<=0 )
   btest = 0
ELSEWHERE
   btest = 1/btest
ENDWHERE
```

All negative valued elements of btest are set to zero and the rest take their reciprocal value. A WHERE statement or construct is one way of avoiding 'divide by zero' errors at run-time.

5.11 Array intrinsic functions

Several intrinsic procedures are available in Fortran90. Their role is to save time and effort when programming. They can be divided into 7 sections for:

- Vector and matrix multiplication.
- Reduction.
- Inquiry.
- Construction.
- Reshape.
- Manipulation.
- Location.

5.11.1 Example of reduction

The intrinsic function ALL() has the form:

```
ALL( condition )
```

and determines whether all elements in an array satisfy the condition. The result is the logical value .TRUE. if all elements satisfy the condition and .FALSE. otherwise.

```
LOGICAL :: test, test2, test3
REAL, DIMENSION(3,2) :: a
a = RESHAPE( (/5,9,6,10,8,12/), (/3,2/) )
...
test = ALL( a>5 )               !false
test2 = ALL( a<20 )             !true
test3 = ALL( a>=5 .AND. test2 ) !true
```

The first statement returns `.false.` since the first element is equal to 5 and not greater. The second statement returns `.true.` since all elements have a value less than 20. The third statement returns `.true.`since all element have a value 5 or greater and the value of `test2` is `.true.`.

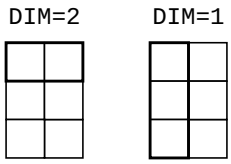
5.11.2 Example of inquiry

The intrinsic function `SIZE()` has the form:

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

and returns the extent of an array for the specified dimension (specified by the argument `DIM`). If the dimension is missing `SIZE()` returns the total number of elements in the array.

```
REAL, DIMENSION(3,2) :: a
num=Size(a)                    !num=6
num=Size(a,DIM=1)              !num=3
num=Size(a,DIM=2)              !num=2
```



The value given to the `DIM` argument specifies the dimension, `DIM=1` returns the number of rows, `DIM=2` the number of columns, etc.

5.11.3 Example of construction

The intrinsic function `SPREAD()` has the form:

```
SPREAD(array, DIM, NCOPIES)
```

and replicates the given array by adding a dimension, where `DIM` stands for dimension and `NCOPIES` for number of copies.

```
REAL, DIMENSION(3) :: a=(/2,3,4/)
REAL, DIMENSION(3,3) :: b,c
b=SPREAD(a, DIM=1, NCOPIES=3)
c=SPREAD(a, DIM=2, NCOPIES=3)
```

The first `SPREAD` statement replicates array `a` three times on the `row` dimension. The second `SPREAD` statement replicates array `a` three times on the `column` dimension.

b

2	3	4
2	3	4
2	3	4

c

2	2	2
3	3	3
4	4	4

5.11.4 Example of location

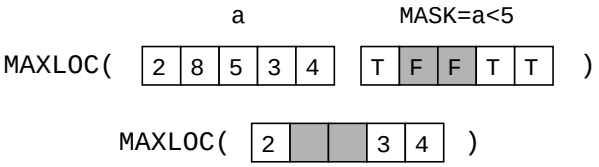
The intrinsic function `MAXLOC()` has the form:

```
MAXLOC(array, [mask])
```

determines the location of the element which has the largest value in an array and satisfies the optional *mask*. A mask is a logical array (or condition involving an array)

which conforms to `array`. The only elements of `array` which take part in the search for the maximum valued elements are those which correspond to `.TRUE.` elements in the mask.

```
REAL :: a(5)=(/2,8,5,3,4/)
num = MAXLOC( a )               !num=2
num = MAXLOC( a(2:4) )          !num=1, note renumbering
num = MAXLOC( a, MASK=a<5 )     !num=5
```



The first statement returns 2 since this is the position of the highest number on the list. The second `MAXLOC()` statement returns the value 1 since this is the position of the highest valued element in the array section. Remembering that elements in array section statements are renumbered with one as the lower bound in each dimension. The third `MAXLOC()` statement returns 5 since this is the position of the highest number on the list when numbers greater than 5 are excluded by the mask.

5.12 Exercises

1. Consider the following array:

   ```
   INTEGER, DIMENSION(3,3) :: a
   a = RESHAPE( (/ 1, 2, 5, 8, 6, 7, 5, 0, 0 /), (/3,3/) )
   ```

   What is the value of element a(2,1); a(3,2); a(1,2); a(2,3). Write a program to display all required values.
2. Given the following declarations:

   ```
   REAL, DIMENSION(1:10,1:20) :: a
   REAL, DIMENSION(10,-5:10) :: b
   REAL, DIMENSION(0:5,1:3,6:9) :: c
   REAL, DIMENSION(1:10,2:15) :: d
   ```

   What is the rank, size, bounds, and extents of `a`, `b`, `c` and `d`? Write a program which uses the intrinsic functions `SIZE()`, `LBOUND()`, `UBOUND()` and `SHAPE()` to display the required information.
3. Declare an array for representing a chessboard (a board of 8x8), indicating a white square with `.false.`, and a black square with `.true.`.
4. Given the following declarations:

   ```
   REAL, DIMENSION(-1:5,3,8) :: alpha=1.0
   REAL, DIMENSION(-3:3,0:2,-7:0) :: beta=0.0
   ```

   Are the two arrays conformable? Write a program including the statement `b=a` to confirm your answer.
5. Given the array declaration below which of the following references are valid? Write a program to view to output of the valid references.

   ```
   REAL :: a(0:5,3)=1.0

   a(2,3)     a(6,2)     a(0,3)     a(5,6)     a(0,0)
   a(2:,:4)   a(0,3:1)   a(0,1:3:-1)a(::2, 1) a(:,0:5:6)
   ```
6. What is the array element order of the following array?
   ```
   INTEGER, DIMENSION(-1:1,2,0:1) :: alpha
   ```
7. Declare and initialise the array (using `RESHAPE()`) `beta` with the following elements

```
5  6  1
4   2  2
0   5  3
```

8. Using vector subscripts declare a rank 1 array `zeta` with 30 elements and place the following values in the array:

a) 1.0 to the 1st and 2nd elements.

b) 2.0 to the 10th, 12th, 14th and 16th elements.

c) 3.0 to 24th, 25th, 27th and 22th element.

9. For the following array declarations, which of the following statements are valid (i.e. for which of the following are the array expressions conforming?) Test your answer by writing a program.

```
REAL, DIMENSION(50) :: alpha
REAL, DIMENSION(60) :: beta
alpha = beta
alpha(3:32) = beta(1:60:2)
alpha(10:50) = beta
alpha(10:49) = beta(20:59)
alpha = beta(10:59)
alpha(1:50:2) = beta
beta = alpha
beta(1:50) = alpha
```

10. Initialise an array of rank one and extend 10 with the values 1 to 10 using

a) a constructor with the list of values

b) a constructor with the DO Loop

11. An array of rank one and extent 50 has been declared and needs to be initialised with the values of -1 (first element), 1 (last element) and 0 (rest of elements). Which of the following constructor structures are valid (if any)?

```
alpha = (/-1,(0,i=2,49),1/)
alpha = (/-1,(0,i=1,48),1/)
alpha = (/-1,(0,i=37,84),1/)
alpha = (/-1,48*0,1/)
```

12. If alpha has been declared and initialised as follows

```
INTEGER, DIMENSION(-5:0) :: alpha=(/2,18,5,32,40,0/)
```

What is the result of:

a) `MAXLOC(alpha)`

b) `MAXLOC(alpha,MASK=alpha/=40)`

13. Determine what the following array constructor does and then simplify the constructor:

```
REAL, DIMENSION(1000,1000) :: data
data = (/((data(i,j)+10.34,j=1,1000),i=1,1000) / )
```

14. Write a `WHERE` statement (or construct) which:

a) only changes the sign of the elements of array that are negative.

b) replicates every non-zero element of an array beta by its reciprocal and every zero element by 1.

15. The number of permutations of *n* objects, taken *r* at a time is:

$$P_{n(r)} = \frac{n!}{\langle n-r \rangle !}$$

Write a program which sets up a rank one array to hold the values 1,2,3,...,10. Using the intrinsic function PRODUCT() (which returns the product of all array elements passed to it) and various array sections, calculate:

a) The number of permutations n=5 people may pose for a photo standing in

r=1 rows.

b) the number of permutations n=8 students may sit in a row of r=4 front row desks

6 Control statements

Fortran 90 has three main types of control construct:

- IF
- CASE
- DO

Each construct may be *nested* one within another, and may be named in order to improve readability of a program.

6.1 Conditional statements

In everyday life decisions are based on circumstances. For example, the decision to take an umbrella depends on whether it is raining or not. Similarly, a program must be able to select an appropriate action according to circumstances. For instance, to take different actions based on experimental results.

Selection and routing control through the appropriate path of the program is a very powerful and useful operation. Fortran 90 provides two mechanisms which enable the programmer to select alternative action(s) depending on the outcome of a (logical) condition.

- The IF statement and construct.
- The select case construct, CASE.

6.1.1 IF statement and construct

The simplest form of the `IF` statement is a single action based on a single condition:

```
IF( expression ) statement
```

Only if `expression` (a logical variable or expression) has the value .`TRUE`. is `statement` executed. For example:

```
IF( x<0.0 ) x=0.0
```

Here, if `x` is less than zero then it is given a new value, otherwise x retains it's previous value. The `IF` statement is analogous to phrases like 'if it is raining, take an umbrella'.

The structure of an `IF` construct depends on the number of conditions to be checked, and has the following general form:

```
[name:]IF (expression1) THEN
          block1
       ELSEIF (expression2) THEN [name]
          block2
       ...
       [ELSE [name]
```

```
      block]
   ENDIF [name]
```

Where expression# is a logical variable or expression.

The construct is used when a number of statements depend on the same condition. For example, 'if it rains then take a coat and take an umbrella'. This time a THEN part is required. Notice that an END IF (or ENDIF) statement is required to indicate the end of a conditional block of statements.

```
LOGICAL :: rain
INTEGER :: numb=0, ncoat
...
IF ( rain ) THEN
   ncoat = 1
   numb = numb+1
ENDIF
```

If rain is .TRUE. the block of statements are executed and control passes to the next statement after ENDIF, otherwise the block of statements is skipped and control passes directly to the next statement after ENDIF.

More complex situation can occur when performing alternative actions depending on a single condition. For instance, the previous examples does not make a distinction between levels of rainfall. The example above can be rephrased as 'if there is light rain then take a coat otherwise (else) take a coat and an umbrella'.

```
REAL :: inches_of_rain
INTEGER :: numb=0, ncoat
...
IF( inches_of_rain>0.05 ) THEN     !heavy rain
   ncoat = 1
   numb = numb+1
ELSE                               !light rain
   ncoat = 1
ENDIF
```

Notice the use of the ELSE part separating different options and that each block may contain one or more statements. The second block of statements acts as a set of 'default' statements for when the condition is not satisfied. The passing of control follows the same rules as mentioned above.

There are situations when alternative actions depend on several conditions. For example, a discount applied to a purchase may vary depending on the number of items purchased, the larger the purchase the larger the discount; i.e.

```
REAL :: cost, discount
INTEGER :: n                          !number of items
   ...
   IF ( n>10 ) THEN                   !25% discount on 11 or more
      discount = 0.25
   ELSEIF ( n>5 .AND. n<=10 ) THEN    !15% discount on 6-10 items
      discount = 0.15
   ELSEIF ( n>1 .AND. n<=5 ) THEN     !15% discount on 2-5 items
      discount = 0.1
   ELSE                               !no dicount for 1 item
      discount = 0.0
   ENDIF
   cost = cost-(cost*discount)
   WRITE(*,*) 'Invoice for ', cost
```

Notice the use of the ELSEIF to add further conditions to the block (other discount bands in this case). The ELSE statement acts as a default in order to cover other eventualities. Again, the same rules concerning passing of control apply.

IF constructs can be labelled. Naming constructs can be useful when one is nested inside another, this kind of labelling makes a program easier to understand, for example:

```
outer:  IF( x==0 ) THEN
           ...
        ELSE outer
inner:     IF( y==0.0 ) THEN
              ...
           ENDIF inner
        ENDIF outer
```

6.1.2 SELECT CASE construct

The SELECT CASE construct provides an alternative to a series of repeated IF ... THEN ... ELSE IF statements. The general form is:

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

The result of expression may be of a CHARACTER, LOGICAL or INTEGER; value must be of the same type as the result of expression and can be any combination of:

- A single integer, character, or logical depending on type.
- min : max any value between the two limits.
- min: any value from a minimum value upwards.
- :max any value up to a maximum value.

CASE DEFAULT is optional and covers all other possible values of the expression not already covered by other CASE statements. For example:

```
INTEGER :: month

season:SELECT CASE( month )
        CASE(4,5)                  !months 4 and 5
           WRITE(*,*) 'Spring'
        CASE(6,7)                  !months 6 and 7
           WRITE(*,*) 'Summer'
        CASE(8:10)                 !months 8,9 and 10
           WRITE(*,*) 'Autumn'
        CASE(11,1:3,12)            !months 1,2,3,11,12
           WRITE(*,*) 'Winter'
        CASE DEFAULT               !integer not in range 1-12
           WRITE(*,*) 'not a month'
        END SELCET season
```

The above example prints a season associated with a given month. If the value of the integer month is not in the range 1-12 the default case applies and the error message 'not a month' is printed, otherwise one of the CASE statements applies. Notice that there is no preferred order of values in a CASE statement.

6.1.3 GOTO

The GOTO statement can be used to transfer control to another statement, it has the form:

```
GOTO label
```

The GOTO statement simply transfers control to the statement, skipping any statements in between. For example:

```
   ...
   IF( x<10 ) GOTO 10
   ...
10 STOP
```

The GOTO statement should be avoided where ever possible, programs containing such statements are notoriously hard to follow and maintain. The STOP statement terminates a program.

6.2 Repetition

An important feature of any programming language is the ability to repeat a block of statements. For example, converting a character from upper to lower case (or visa versa) can be done in a single executable statement. In order to convert several characters (in say a word or sentence) one has to either repeat the statement or re-execute the program. Using a loop construct it is possible to restructure the program to repeat the same statement as many times as required.

6.2.1 DO construct

In Fortran 90 it is the DO loop (or construct) which enables the programmer to repeat a a block of statements. The DO construct has the general form:

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

The DO construct may take two forms:

- A count controlled DO loop.
- A 'forever' DO loop.

A count controlled loop uses a control clause to repeat a block of statements a predefined number of times:

```
[name:] DO count = start, stop [,step]
           block
        END DO [name]
```

The control clause is made up of the following:

- count is an integer variable and is used as the 'control'.
- start is an integer value (or expression) indicating the initial value of count.
- stop is an integer value (or expression) indicating the final value of count.
- step is an integer value (or expression) indicating the increment value of count. The step is optional and has a default value of 1 if omitted.

On entering the loop count will take the value start, the second time round (after executing the statements in block) count will have the value start+step (or start+1 if step is missing) and so on until the last iteration when it will take the

value stop (or an integer value no greater than stop). The number of times the statements will be executed can be calculated from:

$$iterations = (stop + step - start)/(step)$$

It is possible for stop to be less than start and for step to be negative. In this case the loop counts backwards (note this is in contrast to array sections which have zero size if the upper bound is ever below the lower bound!) If stop is smaller than start and step is positive then count will take the value zero and the statement(s) will not be executed at all. The value of count is not allowed to change within the loop. For example:

```
INTEGER :: i, j, k
all:  DO i=1,10
         WRITE(*,*) i       !write numbers 1 to 10
      ENDDO all

nil:  DO j=10,1
         WRITE(*,*) j       !write nothing
      ENDDO nil

even: DO k=10,2,-2
         WRITE(*,*) k       !write even numbers 10,8,6,4,2
      END DO even
```

In the absence of a control clause the block of statements is repeated indefinitely.

```
[name:] DO
           block
        ENDDO [name]
```

The block of statements will be repeated forever, or at least until somebody stops the program. In order to terminate this type of loop the programmer must explicitly transfer control to a statement outside the loop.

6.2.2 Transferring Control

The EXIT statement is a useful facility for transferring control outside a DO loop before the END DO is reached or the final iteration is completed. After an EXIT statement has been executed control is passed to the first statement after the loop. For example:

```
INTEGER :: value=0, total=0
   ...
   sum: DO
      READ(*,*) value            !read in a number
      IF (value==0) EXIT sum     !if nothing to add, exit loop
      total = total + value      !calculate running total
   END DO sum
```

The CYCLE statement transfers control back to the beginning of the loop to allow the next iteration of the loop to begin (thereby skipping the rest of the current iteration). For example:

```
INTEGER :: int
   ...
   name: DO
      READ(*,*) int           !read in a number
      IF (int<0) CYCLE name   !if negative, read in another
      ...
   ENDDO name
```

The name of the loop can be omitted from an EXIT or CYCLE statement. However confusion can arise from multiple and nested (i.e. one inside another) DO loops, EXIT and CYCLE statements, therefore naming loops is highly recommended.

Where loops are nested, unnamed EXIT and CYCLE statements refer to the inner most loop in which they sit. It is possible to pass control from any inner loop to any outer loop by specifying the loop name. As an example consider the following:

```
outer: DO i=1,10
   inner1: DO
      IF( x<0 ) EXIT             !exit loop inner1
      IF( x==0 ) EXIT outer      !exit loop outer
      ...
   ENDDO inner1

   inner2: DO
      IF( x<0 ) CYCLE            !cycle loop inner2
      IF( x==0 ) CYCLE inner1    !illegal
      ...
   ENDDO inner2
   ...
ENDDO outer
```

6.3 Nesting

Placing one block construct inside another (IF-THEN statements inside DO loops, etc.) is possible, but the inner block(s) must by completely within the outer block(s). It is illegal to overlap nested statements. For example:

```
main: IF( sum>100 ) THEN
   inner: DO i=1,n
      ...
ENDIF main
   ENDDO inner     !illegal, inner must be within main
```

This is generally true of all constructs, i.e. DO loops, IF-THEN-ELSEIF and CASE constructs. The maximum level of nesting (constructs inside constructs inside constructs...) is compiler dependant. In the case of the NAG compiler this level is 20 for DO loops and 30 for CASE constructs.

6.4 Exercises

1. Write a program which reads in a single character and returns a character according to the following conditions:

- if an upper case letter is entered it returns the lower case equivalent.
- if a lower case letter is entered it returns the upper case equivalent.
- if a non-letter is entered it does absolutely nothing.

Hint: In the ANSI collating sequence upper and lower case letters differ by 32; So to convert from upper to lower use the character-to-integer (IACHAR( )) and integer-to-character (ACHAR( )) ANSI based function as follows:

```
CHARACTER(LEN=1) :: charin, charout
charout = ACHAR( IAHAR(charin)+32 )     !upper to lower
charout = ACHAR( IAHAR(charin)-32 )     !lower to upper
```

2. A company pays its employees weekly according to the following rules:

- No person works for more than 60 hours.

- Overtime is paid for more than 40 hours.
- Overtime rate is a time and a half of the basic rate.
- Basic rate can not exceed £15.

Write a program which reads the employee's number of hours worked. If the hours worked exceed 60, print an appropriate error message. Otherwise print the expected payment.

3. Given the power (watts) and the voltage (volts) the current (amps) drawn in a circuit can be found from: Current=(Power)/(Voltage).

Write a program that calculates the current given the power of a device (remember that the voltage is 240v in the UK) and displays the most suitable cable for use. Consider 3 suitable cables (up to 5 amps, up to 13 amps, and up to 30 amps). In case a suitable cable can not be found the program should print an appropriate message.

4. Predict the values `loop` takes and the value `loop` has after termination of each of the following `DO` constructs. Your predictions may be tested by writing a program which reads in the values used in the loop control clause (start, stop and step) as input.

```
(a) DO loop=5, 3, 1
(b) DO loop=-6, 0
(c) DO loop=-6, 0, -1
(d) DO loop=-6, 0, 1
(e) DO loop=6, 0, 1
(f) DO loop=6, 0, -1
(g) DO loop=-10, -5, -3
(h) DO loop=-10, -5, 3
```

5. Write a program which prints a multiplication table (i.e. 1n=?, 2n=?,... 12n=?). Allow the user to determine which table (value of n) they require.

6. Write a program to calculate and display the size of A0 to A6 papers in both mm and inches. Use following formula:

$$Height(cm) = 2^{((1/4)-(n/2))}$$

$$Width(cm) = 2^{(-(1/4)-(n/2))}$$

Where n is the size of the paper 0 to 6, and one inch=2.54cm.

7. Write a program to produce the Fibonacci sequence. This sequence starts with two integers, 1 and 1. The next number in the sequence is found by adding the previous two numbers; for example, the 4th number in the series is the sum of the 2nd and the 3rd and so on. Terminate when the nth value is greater than 100.

8. The increase in temperature *dT* of a chemical reaction can be calculated using:

$$dT = 1 - \exp(-kt)$$

$$k = \exp(-q)$$

$$q = 2000/(T + 273.16)$$

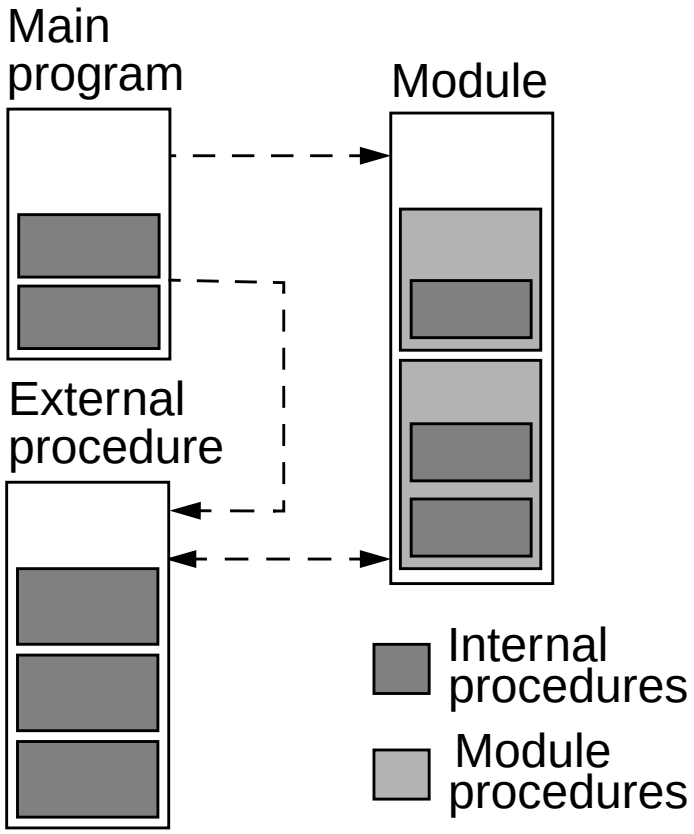
where *T* is the temperature in centigrade, and *t* is the time in seconds. Write a program which prints the temperature of such a reaction at 1 minute intervals, The initial temperature is supplied by the user and the above equations should be re-calculated once every second. The program should terminate when the temperature reaches twice the initial temperature.

7 Program units

7.1 Program structure

A single Fortran 90 program can be made up of a number of distinct program units, namely procedures (internal, external and module) and modules. An executable program consists of one main program, and any number (including zero) of other program units. It is important to realise that the internal details of each program unit is separate from other units. The only link between units is the interface, where one unit invokes another by name. The key to writing programs through program units is to ensure that the procedure interfaces are consistent.

The following illustrates the relationship between the different types of program units:



Dividing a program into units has several advantages:

- Program units can be written and tested independently.
- A program unit that has a well defined task is easier to understand and maintain.
- Once developed and tested modules and external procedures can be re-used in

other programs (allowing the programmer to build up personal libraries).

- Some compilers can better optimise code when in modular form.

7.2 The main program

All programs have one (and only one) main program. A program always begins executing from the first statement in the main program unit, and proceeds from there. The general form of the main program unit is:

```
PROGRAM [name]
   [specification statements]
   [executable statements]
   ...
[CONTAINS
   internal procedures]
END [PROGRAM [name]]
```

The PROGRAM statement marks the beginning of the main program unit while the END PROGRAM statement not only marks the end of the unit but also the end of the program as a whole. The name of the program is optional but advisable. The CONTAINS statement serves to identify any procedures that are internal to the main program unit. (Internal procedures are dealt with later on in this chapter.) When all executable statements are complete, control is passed over any internal procedures to the END statement.

A program can be stopped at any point during its execution, and from any program unit, through the STOP statement:

```
STOP [label]
```

where label is an optional character string (enclosed in quotes) which may be used to inform the user why and at what point the program has stopped.

7.3 Procedures

Procedures are a type of program unit, and may be either subroutines or functions. Procedures are used to group together statements that perform a self-contained, well defined task. Both subroutines and functions have the following general form:

```
procedure name [(argument list)]
   [specification statements]
   [executable statements]
   ...
[CONTAINS
   internal procedures]
END procedure [name]
```

where procedure may be either SUBROUTINE or FUNCTION.

There are several different types of procedure:

- Internal - inside another program unit.
- External - self contained (possibly in languages other than Fortran 90).
- Module - contained within a module.

To use a procedure (regardless of type) requires a referencing statement. Subroutines are invoked by the CALL statement while functions are referenced by name:

```
   CALL name [( argument list )]
```

```
result = name [( argument list )]
```

In both cases control is passed to the procedure from the referencing statement, and is returned to the same statement when the procedure exits. The argument list are zero or more variables or expressions, the values of which are used by the procedure.

7.3.1 Actual and dummy arguments

Procedures are used to perform well defined tasks using the data available to them. The most common way to make data available to a procedure is by passing it in an argument list when the procedure is referenced.

An argument list is simply a number of variables or expressions (or even procedure names - see later). The argument(s) in a referencing statement are called actual arguments, while those in the corresponding procedure statement are call dummy arguments. Actual and dummy argument are associated by their position in a list, i.e the first actual argument corresponds to the first dummy argument, the second actual argument with the second dummy argument, etc. The data type, rank, etc. of actual and dummy arguments must correspond exactly.

When a procedure is referenced data is copied from actual to dummy argument(s), and is copied back from dummy to actual argument(s) on return. By altering the value of a dummy argument, a procedure can change the value of an actual argument.

- A subroutine is used to change the value of one or more of its arguments; for example:

```
REAL, DIMENSION(10) :: a, c
...
CALL swap( a,c )

SUBROUTINE swap( a,b )
   REAL, DIMENSION(10) :: a, b, temp
   temp = a
   a = b
   b = temp
END SUBROUTINE swap
```

  The subroutine `swap` exchanges the contents of two real arrays.

- A function is used to generate a single result based on its arguments, for example:

```
REAL :: y,x,c
...
y = line( 3.4,x,c )

FUNCTION line( m,x,const )
   REAL :: line
   REAL :: m, x, const
   line = m*x + const
END FUNCTION line
```

  The function `line` calculates the value of `y` from the equation of a straight line. The name of the function, `line`, is treated exactly like a variable, it must be declared with the same data type as `y` and is used to store the value of the function result.

Note that in both examples, the name of a dummy argument may be the same as or different from the name of the actual argument.

7.3.2 Internal procedures

Program units (the main program, external procedures and modules) may contain internal procedures. They are gathered together at the end of a program unit after the CONTAINS statement. A unit 'hosts' any procedures that are contained within it. Internal procedures may not themselves contain other internal procedures and thus cannot include the CONTAINS statement.

Internal procedures may only be referenced by their host and other procedures internal to the same host, although internal procedures may invoke other (external and module) procedures.

For example:

```
PROGRAM outer
   REAL :: a, b, c
   ...
   CALL inner( a )
   ...
CONTAINS

   SUBROUTINE inner( a )         !only available to outer
      REAL :: a                  !passed by argument
      REAL :: b=1.0              !redefined
      c = a + b                  !c host association
   END SUBROUTINE inner

END PROGRAM outer
```

The program outer contains the internal subroutine inner. Note that variables defined in the host unit remain defined in the internal procedure, unless explicitly redefined there. In the example, although a, b and c are all defined in outer:

- The value of a is passed by argument to a redefined variable (dummy argument) also called a. Even though they hold the same value, the variables a are different objects.
- Like a, the variable b is redefined in the subroutine and so is a different object to b in the host program. The value of b is not passed by argument or by host association.
- c is a single object, common to both outer and inner through host association.

In order to prevent redefining a variable by mistake, it is good practice to declare all variables used in a procedure.

7.3.3 External procedures

External procedures are self contained program units (subroutines or functions) that may contain (i.e. host) internal procedures. For example:

```
PROGRAM first
   REAL :: x
   x = second()
   ...
END PROGRAM first

FUNCTION second()     !external
   REAL :: second
      ...             !no host association
END FUNCTION second
```

External procedures have no host program unit, and cannot therefore share data through host association. Passing data by argument is the most common way of sharing data with an external procedure. External procedures may be referenced by all other types of procedure.

7.4 Procedure variables

Any variables declared in a procedure (what ever its type) are referred to as local to that procedure, i.e. generally they cannot be used outside of the procedure in which they are declared. Dummy variables are always local to a procedure.

Variables declared inside a procedure usually only exist while the procedure in question is executing:

- Whenever a procedure is referenced, variables declared in the procedure are ‘created’ and allocated the required storage from memory.
- Whenever a procedure exits, by default variables declared in the procedure are ‘destroyed’ and any storage they may have used is recovered.

This ‘creation’ and ‘destruction’ of procedures variables means that by default, no variable declared inside a procedure retains is value from one call to the next. This default can be overcome to allow local variables to retain their values from call to call.

7.4.1 SAVE

The SAVE attribute forces the program to retain the value of a procedure variable from one call to the next. Any variable that is given an initial value in its declaration statement has the SAVE attribute by default. For example:

```
FUNCTION func1( a_new )
   REAL :: func1
   REAL :: a_new
   REAL, SAVE :: a_old             !saved
   INTEGER :: counter=0            !saved
      ...
      a_old = a_new
      counter = counter+1
END FUNCTION func1
```

The first time the function `func1` is referenced, `a_old` has an undefined value while `counter` is set to zero. These values are reset by the function and saved so that in any subsequent calls `a_old` has the value of the previous argument and `counter` is the number of times `func1` has previously been referenced.

Note: it is not possible to save dummy arguments or function results!

7.5 Interface blocks

Interfaces occur where ever one program unit references another. To work properly a program must ensure that the actual arguments in a reference to a procedure are consistent with the dummy arguments expected by that procedure. Interfaces are checked by the compiler during the compilation phase of a program and may be:

- explicit - as with references to internal and module procedures, where the compiler can see the details of the call and procedure statements.
- implicit - as with references to external procedures, here the compiler assumes the details of the call and procedure statements correspond.

Where ever possible interfaces should be made explicit. This can be done through the interface block:

```
INTERFACE
   interface statements
END INTERFACE
```

The interface block for a procedure is included at the start of the referencing program unit. The interface statements consist of a copy of the SUBROUTINE (or FUNCTION) statement, all declaration statements for dummy arguments and the END SUNROUTINE (or FUNCTION) statement. For example:

```
PROGRAM count
   INTERFACE
      SUBROUTINE ties(score, nties)
         REAL :: score(50)
         INTEGER :: nties
      END SUBROUTINE ties
   END INTERFACE
   REAL, DIMENSION(50):: data
   ...
   CALL ties(data, n)
   ...
END PROGRAM count

   SUBROUTINE ties(score, nties)
      REAL :: score(50)
      INTEGER :: nties
      ...
   END SUBROUTINE ties
```

The interface block in the program `count` provides an explicit interface to the subroutine `ties`. If the `count` were to reference other external procedures, their interface statements could be placed in the same interface block.

7.6 Procedures arguments

7.6.1 Assumed shape objects

One of the most powerful aspects of using a procedure to perform a task is that once written and tested the procedure may be used and reused as required (even in other programs).

Since it is often the case that a program may wish to pass different sized arrays or character strings to the same procedure, Fortran 90 allows dummy arguments to have a variable sizes. Such objects are call assumed shape objects. For example:

```
SUBROUTINE sub2(data1, data3, str)
REAL, DIMENSION(:) :: data1
INTEGER, DIMENSION(:,:,:) :: data3
CHARACTER(len=*) :: str
...
```

The dummy arguments `data1` and `data3` are both arrays which have been declared with a rank but no size, the colon ':' is used instead of a specific size in each dimension. Similarly `str` has no explicit length, it adopts the length of the actual argument string. When the subroutine `sub2` is called, all three dummy arguments assume the size of their corresponding actual arguments; all three dummy arguments are assumed shape objects.

7.6.2 The INTENT attribute

It is possible, and good programming practice, to specify how a dummy argument will be used in a procedure using the INTENT attribute:

- `INTENT(IN)` - means that the dummy argument is expected to have a value when the procedure is referenced, but that this value is not updated by the procedure.
- `INTENT(OUT)` - means that the dummy argument has no value when the procedure is referenced, but that it will given one before the procedure finishes.
- `INTENT(INOUT)` - means that the dummy argument has an initial value that will be updated by the procedure.

For example:

```
SUBROUTINE invert(a, inverse, count)
   REAL, INTENT(IN) :: a
   REAL, INTENT(OUT) :: inverse
   INTEGER, INTENT(INOUT) :: count
      inverse = 1/a
      count = count+1
END SUBROUTINE invert
```

The subroutine invert has three dummy arguments. `a` is used in the procedure but is not updated by it and therefore has `INTENT(IN)`. `inverse` is calculated in the subroutine and so has `INTENT(OUT)`. `count` (the number of times the subroutine has been called) is incremented by the procedure and so requires the `INTENT(INOUT)` attribute.

7.6.3 Keyword arguments

Instead of associating actual argument with dummy arguments by position only, it is possible to associate with a dummy argument by name. This can help avoid confusion when referencing a procedure and is often used when calling some of Fortran 90's intrinsic procedures. For example:

```
SUBROUTINE sub2(a, b, stat)
   INTEGER, INTENT(IN) :: a, b
   INTEGER, INTENT(INOUT):: stat
   ...
END SOBROUTINE sub2
```

could be referenced using the statements:

```
INTEGER :: x=0
...
CALL sub2( a=1, b=2, stat=x )
CALL sub2( 1, stat=x, b=2)
CALL sub2( 1, 2, stat=x )
```

The dummy variable names act as keywords in the call statement. Using keywords, the order of arguments in a call statement can be altered, however keywords must come after all arguments associated by position:

```
CALL sub2( 1, b=2, 0 )     !illegal
CALL sub2( 1, stat=x, 2)   !illegal
```

When using keyword arguments the interface between referencing program unit and procedure must be explicit. Note also that arguments with the `INOUT` attribute must be assigned a variable and not just a value, `stat=0` would be illegal.

7.6.4 Optional arguments

Occasionally, not all arguments are required every time a procedure is used. Therefore some arguments may be specified as optional, using the OPTIONAL attribute:

```
SUBROUTINE sub1(a, b, c, d)
   INTEGER, INTENT(INOUT):: a, b
   REAL, INTENT(IN), OPTIONAL :: c, d
   ...
END SUBROUTINE sub1
```

Here a and b are always required when calling sub1. The arguments c and d are optional and so sub1 may be referenced by:

```
CALL sub1( a, b )
CALL sub1( a, b, c, d )
CALL sub1( a, b, c )
```

Note that the order in which arguments appear is important (unless keyword arguments are used) so that it is not possible to call sub1 with argument d but no argument c. For example:

```
CALL sub1( a, b, d )       !illegal
```

Optional arguments must come after all arguments associated by position in a referencing statement and require an explicit interface.

It is possible to test whether or not an optional argument is present when a procedure is referenced using the logical intrinsic function PRESENT. For example:

```
REAL :: inverse_c

IF( PRESENT(c) ) THEN
   inverse_c = 0.0
ELSE
   inverse_c = 1/c
ENDIF
```

If the optional argument is present then PRESENT returns a value .TRUE. In the above example this is used to prevent a run-time error (dividing by zero will cause a program to ‘crash’).

7.6.5 Procedures as arguments

It is possible to use a procedure as an actual argument in a call another procedure. Frequently it is the result of a function which is used as an actual argument to another procedure. For example:

```
PROGRAM test
   INTERFACE
      REAL FUNCTION func( x )
         REAL, INTENT(IN) ::x
      END FUNCTION func
   END INTERFACE
   ...
   CALL sub1( a, b, func(2) )
   ...
END PROGRAM test

REAL FUNCTION func( x )!external
   REAL, INTENT(IN) :: x
   func = 1/x
```

```
END FUNCTION func
```

When the call to sub1 is made the three arguments will be a, b and the result of func, in this case the return value is 1/2. The procedure that is used as an argument will always execute before the procedure in whose referencing statement it appears begins. Using a procedure as an argument requires an explicit interface.

Note that the specification statement for the function func identifies the result as being of type REAL, this is an alternative to declaring the function name as a variable, i.e.

```
REAL FUNCTION func( x )
   REAL, INTENT(IN) :: x
   func = 1/x
END FUNCTION func
```

and

```
FUNCTION func( x )
   REAL :: func
   REAL, INTENT(IN) :: x
   func = 1/x
END FUNCTION func
```

are equivalent.

7.7 Recursion

It is possible for a procedure to reference itself. Such procedures are called recursive procedures and must be defined as such using the RECURSIVE attribute. Also for functions the function name is not available for use as a variable, so a RESULT clause must be used to specify the name of the variable holding the function result, for example:

```
RECURSIVE FUNCTION factorial( n ) RESULT(res)
   INTEGER, INTENT(IN) :: n
   INTEGER :: res
      IF( n==1 ) THEN
         res = 1
      ELSE
         res = n*factorial( n-1 )
      END IF
END FUNCTION factorial
```

Recursion may be one of two types:

- Indirect recursion - A calls B calls A...
- Direct recursion - A calls A calls A...

both of which require the RECURSIVE attribute for the procedure A.

Recursive procedures require careful handling. It is important to ensure that the procedure does not invoke itself continually. For example, the recursive procedure factorial above uses an IF construct to either call itself (again) or return a fixed result. Therefore there is a limit to the number of times the procedure will be invoked.

7.8 Generic procedures

It is often the case that the task performed by a procedure on one data type can be applied equally to other data types. For example the procedure needed to sort an array of real numbers into ascending order is almost identical to that required to sort

an array of integers. The difference between the two arrays is likely to be the data type of the dummy arguments.

For convenience, Fortran 90 allows two or more procedures to be referenced by the same, generic name. Exactly which procedure is invoked will depend on the data type (or rank) of the actual argument(s) in the referencing statement. This is illustrated by some of the intrinsic functions, for example:

The SQRT() intrinsic function (returns the square root of its argument) can be given a real, double precision or complex number as an argument:

- if the actual argument is a real number, a function called SQRT is invoked.
- if the actual argument is a double precision number, a function called DSQRT is invoked.
- if the actual argument is a complex number, a function called CSQRT is invoked.

A generic interface is required in order to declared a common name and to identify which procedures can be referred to by the name. For example:

```
INTERFACE swap

   SUBROUTINE iswap( a, b )
      INTEGER, INTENT(INOUT) :: a, b
   END SUBROUTINE iswap

   SUBROUTINE rswap( a, b )
      REAL, INTENT(INOUT) :: a, b
   END SUBROUTINE rswap

END INTERFACE
```

The interface specifies two subroutines iswap and rswap which can be called using the generic name swap. If the arguments to swap are both real numbers then rswap is invoked, if the arguments are both integers iswap is invoked.

While a generic interface can group together any procedures performing any task(s) it is good programming practice to only group together procedures that perform the same operation on a different arguments.

7.9 Modules

Modules are a type of program unit new to the Fortran standard. They are designed to hold definitions, data and procedures which are to be made available to other program units. A program may use any number of modules, with the restriction that each must be named separately.

The general form of a module follows that of other program units:

```
MODULE name
   [definitions]
   ...
[CONTAINS
   module procedures]
END [MODULE [name]]
```

In order to make use of any definitions, data or procedures found in a module, a program unit must contain the statement:

```
USE name
```

at its start.

7.9.1 Global data

So far variables declared in one program unit have not been available outside of that unit (recall that host association only allows procedures within the same program unit to 'share' variables).

Using modules it is possible to place declarations for all global variables within a module and then USE that module. For example:

```
MODULE global
   REAL, DIMENSION(100) :: a, b, c
   INTEGER :: list(100)
   LOGICAL :: test
END MODULE global
```

All variables in the module global can be accessed by a program unit through the statement:

```
USE global
```

The USE statement must appear at the start of a program unit, immediately after the PROGRAM or other program unit statement. Any number of modules may be used by a program unit, and modules may even use other modules. However modules cannot USE themselves either directly (module A uses A) or indirectly (module A uses module B which uses module A).

It is possible to limit the variables a program unit may access. This can act as a 'safety feature', ensuring a program unit does not accidentally change the value of a variable in a module. To limit the variables a program unit may reference requires the ONLY qualifier, for example:

```
USE global, ONLY: a, c
```

This ensures that a program unit can only reference the variables `a` and `c` from the module `global`. It is good programming practice to USE ... ONLY those variables which a program unit requires.

A potential problem with using global variables are name clashes, i.e. the same name being used for different variables in different parts of the program. The USE statement can overcome this by allowing a global variable to be referenced by a local name, for example:

```
USE global, state=>test
```

Here the variable `state` is the local name for the variable `test`. The => symbol associates a different name with the global variable.

7.9.2 Module procedures

Just as variables declared in a module are global, so procedures contained within a module become global, i.e. can be referenced from any program unit with the appropriate USE statement. Procedures contained within a module are called module procedures.

Module procedures have the same form as external procedures, that is they may contain internal procedures. However unlike external procedures there is no need to provide an interface in the referencing program unit for module procedures, the interface to module procedures is implicit.

Module procedures are invoked as normal (i.e. through the CALL statement or function reference) but only by those program units that have the appropriate USE state-

ment. A module procedure may call other module procedures within the same module or in other modules (through a USE statement). A module procedure also has access to the variables declared in a module through 'host association'. Note that just as with other program units, variables declared within a module procedure are local to that procedure and cannot be directly referenced elsewhere.

One of the main uses for a module is to group together data and any associated procedures. This is particularly useful when derived data types and associated procedures are involved. For example:

```
MODULE cartesian
   TYPE point
      REAL :: x, y
   END TYPE point
CONTAINS
   SUBROUTINE swap( p1, p2 )
   TYPE(point), INTENT(INOUT):: p1
   TYPE(point), INTENT(INOUT):: p2
   TYPE(point) :: tmp
      tmp = p1
      p1 = p2
      p2 = tmp
   END SUBROUTINE swap
END MODULE cartesian
```

The module `carteasian` contains a declaration for a data type called `point`. `cartesian` also contains a module subroutine which swaps the values of its point data type arguments. Any other program unit could declare variables of type `point` and use the subroutine swap via the USE statement, for example:

```
PROGRAM graph
   USE cartesian
   TYPE(point) :: first, last
   ...
   CALL swap( first, last)
   ...
END PROGRAM graph
```

7.9.3 PUBLIC and PRIVATE

By default all entities in a module are accessible to program units with the correct USE statement. However sometimes it may be desirable to restrict access to the variables, declaration statements or procedures in a module. This is done using a combination of PUBLIC and / or PRIVATE statements (or attributes).

The PRIVATE statement / attribute prevents access to module entities from any program unit, PUBLIC is the opposite. Both may and be used in a number of ways:

- As a statement PUBLIC or PRIVATE can set the default for the module, or can be applied to a list of variables or module procedure names.
- As an attribute PUBLIC or PRIVATE can control access to the variables in a declaration list.

```
MODULE one
   PRIVATE              !set the default for module
   REAL, PUBLIC :: a
   REAL :: b
   PUBLIC :: init_a
CONTAINS
   SUBROUTINE init_a()  !public
   ...
```

```
   SUBROUTINE init_b()  !private
   ...
END MODULE one
```

7.9.4 Generic procedures

It is possible to reference module procedures through a generic name. If this is the case then a generic interface must be supplied. The form of the interface block is as follows:

```
INTERFACE generic_name
   MODULE PROCEDURE name_list
END INTERFACE
```

where `name_list` are the procedures to be referenced via `generic_name`, for example a module containing generic subroutines to swap the values of two arrays including arrays of derived data types would look like:

```
MODULE cartesian
   TYPE point
      REAL :: x, y
   END TYPE point

   INTERFACE swap
      MODULE PROCEDURE pointswap, iswap, rswap
   END INTERFACE
CONTAINS
   SUBROUTINE pointswap( a, b )
      TYPE(point) :: a, b
      ...
   END SUBROUTINE pointswap

   !subroutines iswap and rswap

END MODULE cartesian
```

7.10 Overloading operators

Referencing one of several procedures through a generic interface is known as overloading; it is the generic name that is overloaded. Exactly which procedure is invoked depends on the arguments passed in the invoking statement. In a similar way to the overloading of procedure names, the existing operators (+, -, *, etc.) may be overloaded. This is usually done to define the effects of certain operators on derived data types.

Operator overloading is best defined in a module and requires an interface block of the form:

```
INTERFACE OPERATOR( operator )
   interface_code
END INTERFACE
```

where `operator` is the operator to be overloaded and the `interface_code` is a function with one or two `INTENT(IN)` arguments. For example:

```
MODULE strings
   INTERFACE OPERATOR ( / )
      MODULE PROCEDURE num
   END INTERFACE
CONTAINS
```

```
   INTEGER FUNCTION num( s, c )
   CHARACTER(len=*), INTENT(IN) :: s
   CHARACTER, INTENT(IN) :: c
      num = 0
      DO i=1,LEN( s )
         IF( s(i:i)==c ) num=num+1
      END DO
   END FUNCTION num
END MODULE strings
```

Usually, the / operator is not defined for characters or strings but the module strings contains an interface and defining function to allow a string to be divide by a character. The result of the operation is the number of times the character appears in the string:

```
USE strings
   ...
   i = ‘hello world’/’l’   !i=3
   i = ‘hello world’/’o’   !i=2
   i = ‘hello world’/’z’   !i=0
```

7.11 Defining operators

As well as overloading existing operators, it is possible to define new operators. This is particularly useful when manipulating derived data types. Any new operator(s) have the form `.name.` and their effect is defined by a function. Just as with overloaded operators, the defining function requires an `INTERFACE OPERATOR` block and one or two non-optional `INTENT(IN)` arguments, for example:

```
MODULE cartesian
   TYPE point
      REAL :: x, y
   END TYPE point
   INTEFACE OPERATOR ( .DIST. )
      MODULE PROCEDURE dist
   END INTERFACE
CONTAINS
   REAL FUNCTION dist( a, b )
   TYPE(point) INTENT(IN) :: a, b
      dist = SQRT( (a%x-b%x)**2 + (a%y-b%y)**2 )
   END FUNCTION dist
END MODULE cartesian
```

The operator `.DIST.` is used to find the distance between two points. The operator is only defined for the data type `point`, using it on any other data type is illegal. Just as with overloaded operators, the interface and defining function are held in a module. It makes sense to keep the derived data type and associated operator(s) together.

Any program unit may make use of the data type `point` and the operator `.DIST.` by using the module `cartesian`, for example:

```
USE cartesian
TYPE(point) :: a, b
REAL :: distance
   ...
   distance = a .DIST. b
```

7.12 Assignment overloading

It is possible to overload the meaning of the assignment operator (=) for derived data types. This again requires an interface, this time to a defining subroutine. The subrou-

tine must have two, non-optional arguments, the first must have INTENT(INOUT) or INTENT(OUT); the second must have INTENT(IN). For example:

```
MODULE cartesian
   TYPE point
      REAL :: x, y
   END TYPE point
   INTEFACE ASSIGNMENT( = )
      MODULE PROCEDURE max_point
   END INTERFACE
CONTAINS
   SUBROUTINE max_point( a, pt )
   REAL, INTENT(OUT) :: a
   TYPE(point), INTENT(IN) :: pt
      a = MAX( pt%x, pt%y )
   END SUBROUTINE max_point
END MODULE cartesian
```

Using the module cartesian allows a program unit to assign a type point to a type real. The real variable will have the largest value of the components of the point variable. For example:

```
USE cartesian
TYPE(point) :: a = point(1.7, 4.2)
REAL :: coord
   ...
   coord = a            !coord = 4.2
```

7.13 Scope

7.13.1 Scoping units

The scope of a named entity (variable or procedure) is that part of a program within which the name or label is unique. A scoping unit is one of the following:

- A derived data type definition.
- An interface block, excluding any derived data type definitions and interface blocks within it.
- A program unit or internal procedure, excluding any derived data type definitions and interfaces.

All variables, data types, labels, procedure names, etc. within the same scoping unit must have a different names. Entities with the same name, which are in different scoping units, are always separate from one another.

7.13.2 Labels and names

All programs and procedures have their own labels (e.g. see FORMAT statements later). Therefore it is possible for the same label to appear in different program units or internal procedures without ambiguity. The scope of a label is the main program or a procedure, excluding any internal procedures.

The scope of a name (for say a variable) declared in a program unit is valid from the start of the unit through to the unit's END statement. The scope of a name declared in the main program or in an external procedure extends to all internal procedures unless redefined by the internal procedure. The scope of a name declared in an internal procedure is only the internal procedure itself - not other internal procedures.

The scope of a name declared in a module extends to all program units that use that module, except where an internal procedure re-declares the name.

The names of program units are global and must therefore be unique. The name of a program unit must also be different from all entities local to that unit. The name of an internal procedure extends throughout the containing program unit. Therefore all internal procedures within the same program unit must have different names.

The following shows an 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 scope3    !scope 3
      INTERFACE          !scope 3
         ...             !scope 4
      END INTERFACE      !scope 3
      REAL :: a, b       !scope 3
10    ...                !scope 3
   CONTAINS              !scope 2
      FUNCTION scope5()  !scope 5
         REAL :: b       !scope 5
         b = a+1         !scope 5
10       ...             !scope 5
      END FUNCTION       !scope 5
   END SUBROUTINE        !scope 2
END MODULE               !scope 1
```

7.14 Exercises

1. Write a program with a single function to convert temperatures from Fahrenheit to Centigrade. In the body of the main program read in the temperature to be converted, and output the result. The actual calculation is to be done in a function.

a) Write an internal function which requires no actual arguments, but which uses host association to access the value to be converted. The result of the function is the converted temperature.

b) Write an external function which requires the temperature to be converted to be passed as a single argument. Again the function result is the converted temperature. Do not forget to include an interface block in the main program.

Use the following formula to convert from Fahrenheit to Centigrade:

$$Centigrade = (Fahrenheit - 32) \times (5/9)$$

2. Write a program with a single subroutine to sort a list of integer numbers into order. In the main program read a list of random integers (about 5) into an array, call the subroutine to perform the sort, and output the array.

a) Write an internal subroutine which requires no actual arguments, but which uses host association to access the array to be sorted.

b) Write an external subroutine which requires that the array to be sorted be passed as an argument. The external subroutine will require an interface block.

Use the following selection sort algorithm to sort the values in an array a:

```
INTEGER :: a(5), tmp
INTEGER :: j, last, swap_index(1)
```

```
last = SIZE( a )
DO j=1, last-1
   swap_index = MINLOC( a(j:last) )
   tmp = a( j )
   a( j ) = a( (j-1)+swap_index(1) )
   a( (j-1)+swap_index(1) ) = tmp
END DO
```

The selection sort algorithm passes once through the array to be sorted, stopping at each element in turn. At each element the remainder of the array is checked to find the element with the minimum value, this is then swapped with the current array element.

3. Write a program which declares three rank one, real arrays each with 5 elements and that uses array constructors to set a random value for each element (say between 1 and 20) for each array. Write an internal subroutine which finds the maximum value in an array (use the MAX and MAXVAL intrinsic function) and reports and SAVEs that value. Call the subroutine once for each array, the final call should report the maximum value from all arrays.

4. Change the subroutine in written in **3** to accept arrays of any size (if you have not already done so). Test the new subroutine by calling it with three arrays, each of different size.

5. Write a program which declares an rank 1, integer array and use a constructor to set values for each element in the range -10 to 10. The program will pass the array as an argument to an external subroutine, along with two optional arguments `top` and `tail`.

The subroutine is to replace any values in the array greater than `top` with the value of `top`; similarly the subroutine replaces any values lower than `tail` with `tail`. The values of `top` and `tail` are read in by the main program. If either top or tail is absent on call then no respective action using the value is taken. (Remember it is good programming practice to refer to all optional arguments by keyword.)

6. Write a module to contain the definition for a derived data type point, which consists of two real numbers representing the x an y coordinates of that point. Along with this declaration, include a global parameter representing the origin at (0.0,0.0).

The module should also contain a function to calculate the distance between two arbitrary points (this is done earlier in the notes, as an operator). Write a program to read in an x and y coordinate and calculate its distance from the origin.

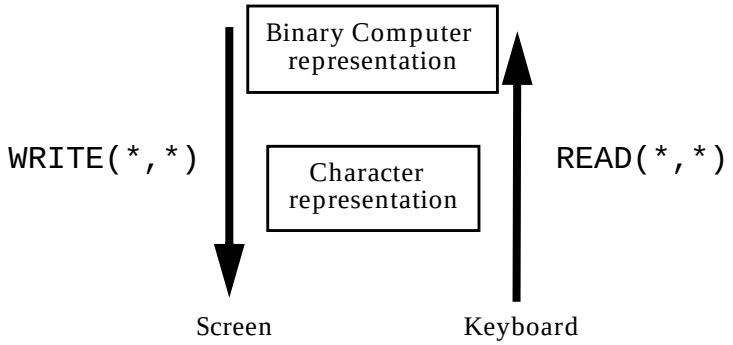
7. Using the selection sort algorithm in question 2 write a module containing two subroutines, one which sorts real arrays the other which sorts integer arrays (both of rank one). The module should provide a generic interface to both subroutines. Check the module and the generic interface by writing a program that uses the module.

8 Interactive Input and Output

This chapter deals with the interaction between a user and the program via the standard input and output devices, namely the keyboard and screen. Data can be stored and represented in several different ways; programs store data in binary form (called *unformatted* data) while programmers and program users perfer to work with characters (or *formatted* data). Almost all interactive input and output (I/O) uses characters and hence formatted data.

When data is read into a program, the characters are converted to the machine's binary form. Similarly, data stored in a binary form is converted when written to the screen. The layout or *formatting* of data can be specified by a programmer or take the default format used in Fortran 90. A subset of the formatting facilities is presented later, the full set is rarely used.

The process of I/O can be summarised as:



The internal hexadecimal representation of a real number may be

```
BE1D7DBF
```

which is difficult to understand (and hence of limited use when written to screen) but corresponds to the real value 0.00045. This may be formatted and written as any or all of:

```
0.45E-03
4.5E-04
0.000450
```

where E## stands for exponent and is equivilent to $x10^{\#\#}$.

This conversion of the internal representation to a user readable form is known as formatted I/O and choosing the exact form of the characters is referred to as formatting.

8.1 Simple Input and Output

A user may assign values to variables using the READ statement. A user will also wish to know the results generated by the program, these will usually be displayed on a screen using the WRITE statement.

To read in a value to say, a variable called radius, the following statement would be suitable:

```
READ(*,*) radius
```

and the value of the variable area would be displayed on the screen by:

```
WRITE(*,*) area
```

The general form of the READ and WRITE statements are:

```
READ( [UNIT=]unit, [FMT=]format ) variable list
WRITE( [UNIT=]unit, [FMT=]format ) variable list
```

unit is an integer associated with the screen or a file (see later) and format describes how the data should look. When reading from the keyboard unit can be either 5 or *; when writing to the screen unit can be either 6 or *.

```
READ(5,*) length, breadth
WRITE(UNIT=6,*) temperature, pressure, mass
WRITE(*,*) pi*radius**2, 2.0
```

Several variables (or expressions) may be specified on one READ or WRITE statement.

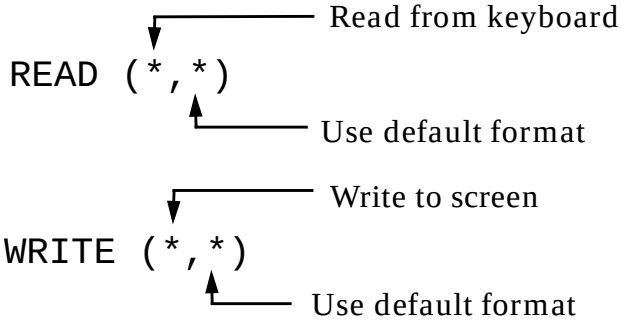
```
READ(5,*) length, breadth
WRITE(6,*) temperature, pressure, mass
WRITE(*,*) pi*radius**2, 2.0
```

8.1.1 Default formatting

When reading and writing to and from screen a Fortran program automatically converts data to the required form; characters for the screen, binary for machine use.

```
INTEGER :: i, j
REAL :: data(3)
...
READ(*,*) i
WRITE(*,*) i, j, data
```

The *s allow a program to use I/O defaults. The first * represents a location (e.g. 'read from the standard input') while the second * represents the default format of the variables which changes from data type to data type. This form of outputting data is quick, simple and convinient.



8.2 Formated I/O

The FORMAT statement may be used to read or write data in a form other than the default format. A FORMAT statement is a labelled statement referenced by a WRITE or READ statement within the same program unit by specifying the label number, for example:

```
      READ(*,100) i, j
      WRITE(*,100) i, j
      READ(*,FMT=200) x, y
      WRITE(*,200) x, y
      ...
100   FORMAT (2I8)              !2I8 is an edit descriptor
200   FORMAT (2F10.6)           !2F10.6 is an edit descriptor
```

Formatting is sometimes known as I/O editing. The I/O is controlled using edit descriptors (explianed later). The general form of a FORMAT statement is:

```
label FORMAT (flist)
```

where label is an identifying number (unique to that part of the program) and flist is a list of edit descriptors which include one or more of:

```
I, F, E, ES, EN, D, G, L, A, H, T, TL, TR,
X, P, BN, BZ, SP, SS, S, /, :, ', and ,(comma)
```

In these notes only the following will be covered

```
I, F, E, ES, EN, A, X, /, :, ', and ,(comma)
```

since many of the edit descriptors cover advanced features, such as output of binary number, etc. and are of limited general use.

The labelled FORMAT statement may be replaced by specifying the format descriptor list as a character string directly in the WRITE or READ statement, as follows:

```
INTEGER :: i, j
REAL :: x, y, z
...
READ (*,'(2I8)') i, j
WRITE (*,'(3F12.6)') x, y, z
```

This has the advantage of improved clarity, i.e. the reader does not have to look at two statements which may not be consecutive in the source listing to determine the effect of the I/O statement.

8.3 Edit Descriptors

Edit descriptors specify exactly how data should be converted into a character string for an output device or internal file, or converted from a character string on an input device or internal file. In the descriptions below, the key letters have the following meaning:

- a -repeat count.
- w -width of field - total number of characters.
- m -minimum number of digits.
- d -digits to right of decimal point.
- e -number of digits in exponent.

Many edit descriptors can be prefixed by a repeat count and suffixed with a field-width, i.e. the total number of digets. Thus in the two examples given above, `2I` and `3F10.6` could be described as two integers and three floating-point real numbers. The fieldwidths of the of the numbers are the default fro the integers and 10 for the reals (a description follows).

In general, if w is larger than the number of digets required to represent the number leading spaces are added. If w is too small to represent the number then on output w asterisks are printed and on input the leftmost w digits are read (truncating the number so beware!).

The I/O statement will use as many of the edit descriptors as it requires to process all the items in the I/O list. Processing will terminate at the next edit descriptor which requires a value from the I/O list.

8.3.1 Integer

The edit descriptor `I` is used to control the format of integers and can have the form `Iw` or `Iw.m`. Several integers may be read/written in the same format by including a repeat count, i.e. `aIw` or `aIw.m`. For example:

```
INTEGER :: itest=1234567         !number to write
...
WRITE(*,*) itest                 ! 1234567
WRITE(*,'(I6)') itest            !******
WRITE(*,'(I10)') itest           !   1234567
WRITE(*,'(I10.9)') itest         ! 001234567
WRITE(*,'(2I7)') itest, 7654321 !12345677654321
WRITE(*,'(2I8)') itest, 7654321 ! 1234567 7654321
```

`I10.9` specifies a total of 10 characters (including a minus sign if appropriate) with a minimum of 9 digits hence the output appears with additional, leading zeros.

8.3.2 Real - Fixed Point Form

The edit descriptor `F` is used to control the format of real (and complex) numbers where a fixed decimal point notation is required. It has the form `Fw.d`. Several real numbers may be read/written in the same format by including a repeat count, i.e. `aFw.d`. For example:

```
REAL :: itest=123.4567           !number to write
...
WRITE(*,*) itest                    ! 1.2345670E+02 -not F format
WRITE(*,'(F8.0)') itest             !    123.
WRITE(*,'(F10.4)') itest            !  123.4567
WRITE(*,'(F10.5)') itest            ! 123.45670
WRITE(*,'(F10.9)') itest            !**********
WRITE(*,'(2F8.4)') itest, 7654321  !123.4567765.4321
WRITE(*,'(2F10.4)') itest, 7654321 !  123.4567  765.4321
```

It is important to remember that the decimal point is counted in the the width `w` of the output. In the above example, although there are 7 numerals to write to the screen the field width must be 8 (or larger) to cater for the decimal point.

8.3.3 Real - Exponential Form

The edit descriptor `E` is used to control the format of real (and complex) numbers where a floating decimal point notation is required. It has the form `Ew.d` or `Ew.dEe` where `e` is the number of digets in the exponent, i.e. a number $x10^{e}$. The exponent is useful for displaying numbers with values below 0.001 or above 1000. As before, sev-

eral real numbers may be read / written in the same format by including a repeat count, i.e. aEw.d. If w is too large to represent the number leading spaces are added before the digets. For example:

```
REAL :: itest=123.45*1000000    !number to write times 1 million
...
WRITE(*,*) itest                   !   1.2345670E+02
WRITE(*,'(E10.4)') itest           !0.1234E+09
WRITE(*,'(E10.5)') itest           !.12345E+09
WRITE(*,'(E10.4E3)') itest         !.1234E+009
WRITE(*,'(E10.9)') itest           !**********
WRITE(*,'(2E12.4)') itest, 7654321 ! 0.12345E+09 0.76543E+04
WRITE(*,'(2E10.4)') itest, 7654321 !0.1234E+090.7654E+04
```

Two alternative forms to the E descriptor are available:

- EN - Engineering - the exponent is always divisible by 3 and the value before the decimal point lies in the range 1..1000
- ES - Scientific - the value before the decimal point always lies in the range 1..10

Both are used in the same way as the E descriptor, for example:

```
REAL :: itest=123.45*100     !number to write times 100
...
WRITE(*,*) itest             !   1.2345000E+04
WRITE(*,'(EN13.6)')          !12.345000E+03
WRITE(*,'(ES13.6)')          ! 1.234500E+04
```

8.3.4 Character

The A edit descriptor is used to control the format of characters and strings. It has the form A or Aw. The A descriptor will writes as many characters as required while Aw writes a string of width w. If w is bigger than the character string leading spaces are added before the string's characters. For example:

```
CHARACTER(LEN=8) :: long='Bookshop'
CHARACTER(LEN=1) :: short='B'
...
WRITE(*,*) long              ! Bookshop
WRITE(*,'(A)') long          !Bookshop
WRITE(*,'(A8)') long         !Bookshop
WRITE(*,'(A5)') long         !Books
WRITE(*,'(A10)') long        !  Bookshop
WRITE(*,'(A)') short         !B
WRITE(*,'(2A) short, long    !BBookshop
WRITE(*,'(2A3) short, long   !  BBoo
```

When using the A descriptor in formatted READ( ) statement (i.e. input) the character string does not need to be enclosed in quotes.

8.3.5 Logical

Logical data is formatted using the L descriptor and has the form Lw or aLw for repeated counts. Usuall only two forms of the L descriptor are used, L for the single charater 'T' or 'F' format and L7 which allows '.TRUE.' and '.FALSE'. to be input .

```
LOGICAL :: ltest=.FALSE.
WRITE(*,*) ltest                        ! F
WRITE(*,'(2L1)') ltest, .NOT.ltest      !FT
WRITE(*,'(L7)') ltest                   !       F
```

8.3.6 Blank Spaces (Skip Character Positions)

The descriptor X is used to introduce spaces between output values to improve readability; it has the form aX. Additional spaces are only meaningful for output (i.e. WRITE( ) statements), they are ignored in formatted READ( ) statements. For Example:

```
INTEGER :: n=1234                    !number to write
...
WRITE(*,'(I4, 2X, I4)') i, i-1       !1234  1233
WRITE(*,'(I4, 4X, I4)') i, i-1       !1234    1233
```

8.3.7 Special Characters

There are a number of other characters which control the format of data; most are used in WRITE( ) statements only.

- ' ' to output the character string specified.
- / specifies take a new line.
- **( )** to group descriptors, normally for repetition.
- : terminate I/O if list exhausted.

For example:

```
INTEGER :: value = 100
INTEGER :: a=101, b=201
...
WRITE(*,'( 'The value is', 2X, I3, ' units.')') value
WRITE(*,'( 'a =', 1X, I3, /, 'b = ', 1X, I3)')
WRITE(*,'( 'a and b =', 2(1X, I3) )') a, b
```

writes the following lines to the screen:

```
The value is  100 units.
a = 101
b =  201
a and b = 101 201
```

Notice that spaces may be specified in the output line by either the X edit descriptor or by spaces in a character string, as in 'b = 201'.

8.4 Input/Output Lists

When more than one variable is written or read in the same WRITE( ) or READ( ) statement, it is refered to as an *I/O list*. For output variables and/or expressions may be used but for input only variables are permitted. Implied-DO loops (see below) may be used for either input or output.

An array may be specified as either to be processed in its entirety, or element by element, or by subrange; for example:

```
INTEGER, DIMENSION(10) :: a
READ (*,*) a(1), a(2), a(3)          !read values into 3 elements
READ (*,*) a                         !read in 10 values
READ (*,*) a(5:8)                    !read values into 4 elements
```

Array elements may only appear once in an I/O list, for example:

```
INTEGER :: b(10), c(3)
```

```
c= (/1,2,1/)
READ (*,*) b(c)      !illegal
```

would be illegal as `b(1)` appears twice.

8.4.1 Derived DataTypes

I/O is performed on derived data types as if the components were specified in order. Thus for `p` and `t` of type `POINT` and `TRIANGLE` respectively, where

```
TYPE point
   REAL :: x, y
END TYPE
TYPE (point) :: pt

TYPE triangle
   TYPE (point) :: a, b, c
END TYPE
TYPE (triangle) :: tri
```

the following two statement pairs are equivalent:

```
READ (*,*) pt
READ (*,*) pt%x, pt%y
...
READ (*,*) tri
READ (*,*) tri%a%x, tri%a%y, tri%b%x, tri%b%y, tri%c%x, tri%c%y
```

An object of a derived data type which contains a pointer (see later) may not appear in an I/O list. This restriction prevents problems occurring with recursive data types.

8.4.2 Implied DO Loop

The Implied-DO-list is like a shorthand version of the DO loop construct. The Implied-DO-list is often used when performing I/O on an array, has the general form:

```
(object, do_var=start, stop [,step])
```

where `do_var` is an interger (and cannot be a pointer). Consider the following examples:

```
INTEGER :: j
REAL, DIMENSION(5) :: a
READ (*,*) ( a(j), j=1,5)       !a(1), a(2), a(3), a(4), a(5)
WRITE (*,*) ( a(j), j=5,1,-1)   !a(5), a(4), a(3), a(2), a(1)
```

The first statement would read 5 values in to each element of `a`, in assending order. The second statement would write all 5 values of `a` in reverse order.

The implied-do-list may also be nested

```
INTEGER :: I, J
REAL, DIMENSION(10,10) :: B
WRITE (*,*) ((B(I,J),I=1,10), J=1,10)
```

This kind of output is an alernative to using an array section.

8.5 Namelist

A namelist is a facility for grouping variables for I/O. A `NAMELIST` is most useful for output as this can be useful for program testing and debugging. It's use on input is slightly more complicated and is best considered elsewhere.

The NAMELIST statement is used to define a group of variables as follows:

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

for example:

```
INTEGER :: sat=7, sun=1, mon=2, tues=3, wed=4, thur=5, fri=6
...
NAMELIST / week / mon, tues, wed, thur, fri   !list
NAMELIST / week / sat, sun                    !may be extended
```

Variables must be declared before appearing in a NAMELIST group (and must not be a mixture of PRIVATE and PUBLIC variables). The keyword NML= may be used in place of the format specifier in an I/O statement. For example:

```
WRITE (*,NML=week)
```

will output the following line:

```
&WEEK SUN=1, MON=2, TUES=3, .../
```

Note the output is an annotated list of the form:

```
& group-name variable1=value {, variable2=value} /
```

This record format (including & and / characters) must be used for input.

Arrays may also be specified, for example

```
INTEGER, DIMENSION(3) :: items
NAMELIST / group / items
ITEMS(1) = 1
WRITE (*, NML=group)
```

would produce

```
&GROUP ITEMS(1)=1 ITEMS(2)=0 ITEMS(3)=0 /
```

8.6 Non-Advancing I/O

The normal action of an I/O statement is to advance to the next record on completion. Thus on input if a record is only partially read the rest of the input record is discarded. On output a WRITE() statement will complete with the cursor positioned at the start of a new line.

Non-advancing I/O permits records to be read in sections (for example a long record of unknown length) or to create a neat user-interface where a prompt for input and the user's response appear on the same line.

There is a complex set of rules covering the use of non-advancing I/O and its various associated keywords. This section only deals with the screen management aspects of this topic.

The ADVANCE keyword is used in write or read statements as follows:

```
INTEGER :: i, j
...
WRITE(*,*,ADVANCE='NO') 'Enter i value: '
READ(*,*) i
WRITE(*,*) 'Enter j value: '              !'ADVANCE='YES'
READ(*,*) j
```

If the user enters the values 10 and 20 this would appear on the screen as

```
Enter i value: 10
Enter j value:
20
```

The non-advancing I/O looks neat compared to the (default) advancing I/O which is spread over two lines.

8.7 Exercises

1. What values would be read into the variables in the READ( ) statement in the following:

```
REAL :: a, b, c
REAL, DIMENSION (1:5) :: array
INTEGER :: i, j, k
READ(*,*) a, b, c
READ(*,*) i, j, k, array
```

given the following input records:

```
1.5 3.4 5.6 3 6 65
2*0 45
3*23.7 0  0
```

Check your answer by write a program which write the values of the variables to the screen.

2. Given the statements:

```
REAL :: a
CHARACTER(LEN=2) :: string
LOGICAL :: ok
READ (*,'(F10.3,A2,L10)') a, string, ok
```

what would be read into a, string and ok if each of the following lines were typed as input records (where b represents a space or blank character)?

```
(a) bbb5.34bbbNOb.true.
(b) 5.34bbbbbbYbbFbbbbb
(b) b6bbbbbb3211bbbbbbT
(d) bbbbbbbbbbbbbbbbbbF
```

Check your answer by write a program which write the values of the variables to the screen.

3. Write statements to output all 16 elements of a one dimensional array of real numbers with 4 numbers per line each in a total fieldwidth of 12 and having two spaces between each number. The array should be output in fixed point notation with 4 characters following the decimal point and then in floating point notation with three significant digits.
Hint: An implied DO loop is useful when grouping array elements on the same line.

4. Write a program which will output the following table to screen:

```
Position  Name    Score
----------------------
 1         tom     93.6
 2         dick    87.0
```

```
 3          harry    50.9
```

When initialising arrays to hold the table's values note that the header might be stored as a string while the underline is simply a repeated character!. The first column should be an interger, the second a string, the third a real number.

9 File-based Input and Output

In the previous modules all input and output was performed from and to the default devices, namely the keyboard and screen. In many circumstances this is not the most appropriate action, i.e. temporary storage of large amounts of intermediate results; large amounts of input or output; output from one program used as the input of another; a set of input data which is used many times, etc.

A mechanism is required which permits a programmer to direct input to be performed on data from a source other than the keyboard (during execution time) and to store output in a more 'permanent' and capacious form. This is generally achieved by utilizing the computer's filestore which is a managed collection of files. A file such as the source program or a set of I/O data is normally formatted, which means it consists of an ordered set of character strings separated by an end of record marker. A formatted file may be viewed using an editor or printed on a printer. An unformatted file (see later) has no discernable structure and should be regarded as single stream of bytes of raw data. An unformatted file is normally only viewed using a suitable user written program.

9.1 Unit Numbers

Fortran I/O statements access files via a unique numeric code or *unit number*. Each unit number is an integer which specifies a data channel which may be connected to a particular file or device. The program may set up a connection specifically, or use the defaults, and may at any time break and redefine the connection. These numbers must lie in the range 1...99.

Unit numbers may be specified as:

- an integer constant e.g. 10
- an integer expression e.g. `nunit` or `nunit+1`
- an asterisk * denoting the default unit.
- the name of an internal file.

A statement such as a `READ()`, `WRITE()` or `OPEN()` is directed to use a particular unit by specifying the `UNIT` keyword as follows:

```
INTEGER :: nunit=10
...
READ(UNIT=10,*)
WRITE(UNIT=nunit,*)
```

The unit number may also be specified as a positional argument as shown later.

Certain unit numbers are reserved to for I/O with the keyboard and screen. The unit number 5 refers to the keyboard (so you should never write to it!) while 6 refers to the screen (so you should never read from it!). The following READ() statements are all equivalent, as are the `WRITE()` statements:

```
READ(*,*) data          !recall * refers to default settings
READ(5,*) data
READ(unit=5,*) data
...
WRITE(*,*) data         !recall * refers to default settings
WRITE(6,*) data
WRITE(UNIT=6,*) data
```

Some computer systems have a naming convention which will "map" unit numbers to default file names, for example when using unit number 10 on a VAX/VMS system this will map to a file called FOR010.DAT and on some UNIX systems to a file called fort.10.

Also some computer systems provide a form of external variable which may be defined prior to execution and the contents of the variable used as a filename. Again on a VAX/VMS system accessing unit 10 will cause an external variable FOR010 to be checked for a filename.

System specific information such as this is provided in the language reference manual for that system.

9.2 READ and WRITE Statements

9.2.1 READ Statement

As has been seen before, the `READ()` statement has the form:

```
READ(clist) list
```

where `clist` is defined as:

```
[UNIT=] unit-number,
[FMT=] format-spec
[,REC= record-number]
[,IOSTAT=ios]
[,ADVANCE=adv]
[,SIZE=integer-variable]
[,EOR=label]
[,END=label]
[,ERR=label]
```

Note that a `unit-number` and `format-spec` are required in that order (though the keywords are not), the rest are optional. Most of the keywords are for advanced file manipulation, and as such will not be discussed here.

A useful argument to detect errors in I/O, particularly to files, is `IOSTAT`. If an error occurs during I/O, the variable specified by `IOSTAT` will return a positive, system dependent integer. The value 0 will be returned if the operation completes successfully. For example:

```
READ (*,*) a,b,c              !read from keyboard, default format
READ (10,FMT=*) line          !read from unit 10, default format
READ (UNIT=5,*) x,y,z         !read from keyboard
READ (UNIT=10,*,IOSTAT=ios)   !ios=0 if all goes ok.
```

Problems with I/O can be detected while a program runs as follows:

```
INTEGER :: fileno=50, ios=0, data
...
READ(UNIT=fileno,*,IOSTAT=ios) data       !read value from file
IF (ios /= 0) THEN
   READ(*,*) 'ERROR in reading from file'  !error message
```

```
   STOP                                        !terminate program
ENDIF
```

9.2.2 WRITE Statement

The WRITE() statement has a general form similar to the READ() statement:

```
WRITE(clist) list
```

where clist is defined as:

```
[UNIT=] unit-number,
[FMT=] format-spec
[,REC= record-number]
[,IOSTAT=ios]
[,ADVANCE=adv]
[,SIZE=integer-variable]
[,EOR=label]
[,ERR=label]
```

Again note that a unit-number and format-spec are required in that order and that other arguments are optional. IOSTAT remains one of the most useful arguments and works the same for WRITE() as for READ() statements. For example:

```
INTEGER :: n=10, ios=0
...
WRITE (*,*) a,b,c          !write to screen, default format
WRITE (UNIT=6,*) i,j       !write to screen, default format
WRITE (10,FMT=*) I         !write to unit 10, default format
WRITE (UNIT=n,FMT=*,IOSTAT=ios) data
```

9.3 OPEN Statement

The OPEN() statement is used to connect a unit number to a file, and to specify certain properties for that file which differ from the defaults. It can be used to create or connect to an existing file. In addition to the standard form described some compliers may provide a number of non-standard additional keywords.

Common programming practice places all OPEN statements in a subroutine which is called in the initialization phase of the main program. OPEN statements invariably contain system specific file names and non-standard features thus, should the program be required to run on more than one computer system, the OPEN statements may be easily located.

The OPEN() statement has the general form:

```
OPEN(unit_no, [olist] )
```

where unit_no is a valid unit number specifier (with or without the keyword) and olist is a list of keyword clauses (explained below) For example, the following OPEN() statement all open a file associated with the unit number 10:

```
INTEGER :: ifile=10
...
OPEN(10)
OPEN(UNIT=10)
OPEN(UNIT=ifile)
```

The following keywords are some of those specified in the Fortran 90 language standard and may be used to specify the nature of the file opened:

- FILE=filename; where filename is a valid string for the particular system. Note that case sensitivity is system specific. e.g. FILE='output.test'
- STATUS=st; where st may be one of 'OLD', 'NEW', 'REPLACE', 'SCRATCH' or 'UNKNOWN'. 'OLD' specifies a file that must already exist; 'NEW' creates a new file; 'REPLACE' deletes an existing file before a new file (with the same name) is created; 'SCRATCH' creates a temporary file which exist only while the program is running and is lost there after. In general use 'OLD' for input and 'NEW' for output.
- ERR=label; is similar to a GOTO statement which works only if an error occurs opening the file. If possible use IOSTAT instead.
- IOSTAT=ios; where ios is an integer variable which is set to zero if the statement is executed successfully or to an implementation dependent constant otherwise.
- ACTION=act; where act may be 'READ', 'WRITE' or 'READWRITE' specifying the permitted modes of operation on the file. The default is processor dependent.

Some common file opening statements:

```
OPEN (UNIT=10,FILE='fibonacci.out')
OPEN (UNIT=11,FILE='fibonacci.out',STATUS='NEW',IOSTAT=ios)
IF( ios/=0) THEN
   WRITE(6,*) 'Error opening file: fibonacci.out.'
   STOP
ENDIF
OPEN (UNIT=12, FILE='student.records', STATUS='OLD', &
      FORM='FORMATTED', IOSTAT=ios)
```

If you are in any doubt about the default values for any of the fields of the OPEN() statement, especially as some are machine dependent, specify the required values. The combinations of possible error conditions, mean that careful thought should be given to the specification of OPEN() statements and the associated error handling.

Specifying some argument values alter the default values of others while some combinations of argument values are mutually exclusive, such problems are beyond these notes.

9.4 CLOSE statement

This statement permits the orderly disconnection of a file from a unit and is done either at the completion of the program or so that a connection may be made to a different file or to alter a property of the file. In its simplest form the CLOSE() statement requires a unit number (of a file already open), but often IOSTAT is used:

```
CLOSE ([UNIT=]unit-number [,IOSTAT=ios])
```

For example:

```
CLOSE (10)
CLOSE (UNIT=10)
CLOSE (UNIT=nunit, IOSTAT=ios)
```

9.5 INQUIRE statement

This statement may be used to check the status of a file or the connection to a file. It causes values to be assigned to the variables specified in the inquiry-list which indicate the status of the file with respect to the specified keywords. The INQUIRE() statement has the general form:

```
INQUIRE (inquiry-list)
```

where `inquiry-list` may be either:

```
FILE=fname
```

or

```
UNIT=unum
```

plus any the following (other keywords / arguments exist but are for more advanced file I / O):

```
[, EXIST=lex]                    !true or false
[, OPENED=lod]                   !true or false
[, NUMBER=unum]                  !unit number
[, NAME=fnm]                     !filename
[, FORMATTED=fmt]                !'YES' or 'NO'
[, UNFORMATTED=unfmt]            !'YES' or 'NO'
[, FORM=frm]                     !'FORMATTED' or'UNFORMATTED'
```

'`EXIST`' determines whether a file of a given name exists; '`OPENED`' determines whether or not it has been opened by the current program; '`NUMBER`' determines the unit number associated with a file; '`NAME`' determines the name associated with a unit number; '`FORMATTED`', '`UNFORMATTED`' or '`FORM`' determine the format of a particular file. The values returned by the `INQUIRE()` statement's arguments are also listed above.

9.6 Exercises

1. Complete the following statement, which would open an unformatted file called 'result.dat' that does not exist.

   ```
   OPEN(UNIT=10,..)
   ```

2. Write a section of code which would open 5 files on the unit numbers from 20 to 25. The default values should be used for all keywords. Your code should include a means of detecting errors.
3. Write sections of code to perform the following:
   (a) test for the existence of a file called `TEMP.DAT`
   (b) test if a file has been opened on unit 10.
   (c) test to see if the file opened on unit 15 is a formatted or unformatted file.
   The program fragments should output the results in a suitable form.
4. Write a Fortran program which will prompt the user for a file name, open that file and then read the file line by line outputting each line to the screen prefixed with a line number. Use the file which contains the source of the program as a test file.

10 Dynamic arrays

So far all variables that have been used have been static variables, that is they have had a fix memory requirement, which is specified when the variable is declared. Static arrays in particular are declared with a specified shape and extent which cannot change while a program is running. This means that when a program has to deal with a variable amount of data, either:

- an array is dimensioned to the largest possible size that will be required, or
- an array is given a new extent, and the program re-complied every time it is run.

In contrast dynamic (or allocatable) arrays are not declared with a shape and initially have no associated storage, but may be allocated storage while a program executes. This is a very powerful feature which allows programs to use exactly the memory they require and only for the time they require it.

10.1 Allocatable arrays

10.1.1 Specification

Allocatable arrays are declared in much the same way as static arrays. General form:

```
type, ALLOCATABLE [,attribute] :: name
```

They must include the ALLOCATABLE attribute and the rank of the array, but cannot specify the extend in any dimension or the shape in general. Instead a colon (:) is used for each dimension. For example:

```
INTEGER, DIMENSION(:), ALLOCATABLE :: a    !rank 1
INTEGER, ALLOCATABLE :: b(:,:)             !rank 2
REAL, DIMENSION(:), ALLOCATABLE :: c       !rank 1
```

On declaration, allocatable arrays have no associated storage and cannot be referenced until storage has been explicitly allocated.

10.1.2 Allocating and deallocating storage

The ALLOCATE statement associates storage with an allocatable array:

```
ALLOCATE( name(bounds) [,STAT] )
```

- if successful name has the requested bounds (if present STAT=0).
- if unsuccessful program execution stops (or will continue with STAT>0 if present).

it is possible to allocate more than one array with the same ALLOCATE statement, each with different bounds, shape or rank. If no lower bound is specified then the default is 1. Only allocatable arrays with no associated storage may be the subject of an ALLOCATE statement, for example

```
n=10
ALLOCATE( a(100) )
ALLOCATE( b(n,n), c(-10:89) ).
```

The storage used by an allocatable array may be released at any time using the DEALLOCATE statement:

```
DEALLOCATE( name [,STAT] )
```

- If successful arrayname no longer has any associated storage (if present STAT=0)
- If unsuccessful execution stops (or will continue with STAT>0 if present).

The DEALLOCATE statement does not require the array shape. It is possible to deallocate more than one array with the same DEALLOCATE statement, each array can have different bounds, shape or rank. Only allocatable arrays with associated storage may be the subject of a DEALLOCATE statement.

The following statements deallocate the storage from the previous example:

```
DEALLOCATE ( a, b )
DEALLOCATE ( c, STAT=test )
IF (test .NE. 0) THEN
   STOP 'deallocation error'
ENDIF
```

It is good programming practice to deallocate any storage that has been reserved through the ALLOCATE statement. **Beware**, any data stored in a deallocated array is lost permanently!

10.1.3 Status of allocatable arrays

Allocatable arrays may be in either one of two states:

- 'allocated' - while an array has associated storage.
- 'not currently allocated' - while an array has no associated storage.

The status of an array may be tested using the logical intrinsic function ALLOCATED:

```
AllOCATED( name )
```

which returns the value:

- .TRUE. if name has associated storage, or
- .FALSE. otherwise.

For example:

```
IF( ALLOCATED(x) ) DEALLOCATE( x )
```

or:

```
IF( .NOT. ALLOCATED( x ) ) ALLOCATE( x(1:10) )
```

On declaration an allocatable array's status is 'not currently allocated' and will become 'allocated' only after a successful ALLOCATE statement. As the program continues and the storage used by a particular array is deallocated, so the status of the array returns to 'not currently allocated'. It is possible to repeat this cycle of allocating and deallocating storage to an array (possibly with different sizes and extents each time) any number of times in the same program.

10.2 Memory leaks

Normally, it is the program that takes responsibility for allocating and deallocating storage to (static) variables, however when using dynamic arrays this responsibility falls to the programmer.

Statements like ALLOCATE and DEALLOCATE are very powerful. Storage allocated through the ALLOCATE statement may only be recovered by:

- a corresponding DEALLOCATE statement, or
- the program terminating.

Storage allocated to local variables (in say a subroutine or function) must be deallocated before the exiting the procedure. When leaving a procedure all local variable are deleted from memory and the program releases any associated storage for use elsewhere, however any storage allocated through the ALLOCATE statement will remain 'in use' even though it has no associated variable name!. Storage allocated, but no longer accessible, cannot be released or used elsewhere in the program and is said to be in an 'undefined' state This reduction in the total storage available to the program called is a 'memory leak'.

```
SUBROUTINE swap(a, b)
REAL, DIMENSION(:) :: a, b
REAL, ALLOCATABLE :: work(:)
   ALLOCATE( work(SIZE(a)) )
   work = a
   a = b
   b = work
   DEALLOCATE( work )         !necessary
END SUBROUTINE swap
```

The automatic arrays a and b are static variables, the program allocates the required storage when swap is called, and deallocates the storage on exiting the procedure. The storage allocated to the allocatable array work must be explicitly deallocated to prevent a memory leak.

Memory leaks are cumulative, repeated use of a procedure which contains a memory leak will increase the size of the allocated, but unusable, memory. Memory leaks can be difficult errors to detect but may be avoided by remembering to allocate and deallocate storage in the same procedure.

10.3 Exercises

1. Write a declaration statement for each of the following allocatable arrays:

(a) Rank 1 integer array.

(b) A real array of rank 4.

(c) Two integer arrays one of rank 2 the other of rank 3.

(d) A rank one real array with lower and upper bound of -*n* and *n* respectively.

2. Write allocation statements for the arrays declared in question **1**, so that

(a) The array in **1** (a) has 2000 elements

(b) The array in **1** (b) has 16 elements in total.

(c) In **1** (c) the rank two array has 10 by 10 elements, each index starting at element 0; and the rank three array has 5 by 5 by 10 elements, each index starting at element -5.

(d) The array in **1** (d) is allocated as required.

3. Write deallocation statement(s) for the arrays allocated in **2**.

4. Write a program to calculate the mean and the variance of a variable amount of data. The number of values to be read into a real, dynamic array `x` is `n`. The program should use a subroutine to calculate the mean and variance of the data held in `x`. The mean and variance are given by:

$$mean = \left(\sum_{i=1}^{n} x(i) \right) / n$$

$$variance = \left(\sum_{i=1}^{n} (x(i) - mean)^2 \right) / (n-1)$$

5. Write a module called `tmp_space` to handle the allocation and deallocation of an allocatable work array called `tmp`. The module should contain two subroutines, the first (`make_tmp`) to deal with allocation, the second (`unmake_tmp`) to deal with deallocation. These subroutines should check the status of `tmp` and report any error encountered. Write a program that tests this module.

The idea behind such a module is that once developed it may be used in other programs which require a temporary work array.

11 Pointer Variables

11.1 What are Pointers?

A *pointer variable*, or simply a *pointer*, is a new type of variable which may reference the data stored by other variables (called *targets*) or areas of dynamically allocated memory.

Pointers are a new feature to the Fortran standard and bring Fortran 90 into line with languages like C. The use of pointers can provide:

- A flexible alternative to allocatable arrays.
- The tools to create and manipulate dynamic data structures (such as linked lists).

Pointers are an advanced feature of any language. Their use allows programmers to implement powerful algorithms and tailor the storage requirements exactly to the size of the problem in hand.

11.1.1 Pointers and targets

Pointers are best thought of as variables which are dynamically associated with (or aliased to) some target data. Pointers are said to ‘point to’ their targets and valid targets include:

- Variables of the same data type as the pointer and explicitly declared with the TARGET attribute.
- Other pointers of the same data type.
- Dynamic memory allocated to the pointer.

Pointers may take advantage of dynamic storage but do not require the ALLOCATABLE attribute. The ability to allocate and deallocate storage is an inherent property of pointer variables.

11.2 Specifications

The general form for pointer and target declaration statements are:

```
type, POINTER [,attr] :: variable list
type, TARGET [,attr] :: variable list
```

Where:

- `type` is the type of data object which may be pointed to and may be a derived data type as well as intrinsic types.
- `attribute` is a list of other attributes of the pointer.

A pointer must have the same data type and rank as its target. For array pointers the declaration statement must specify the rank but not the shape (i.e. the bounds or extend of the array). In this respect array pointers are similar to allocatable arrays.

For example, the following three pairs of statements, all declare pointers and one or more variables which may be targets:

```
REAL, POINTER :: pt1
REAL, TARGET :: a, b, c, d, e

INTEGER, TARGET :: a(3), b(6), c(9)
INTEGER, DIMENSION(:), POINTER:: pt2

INTEGER, POINTER :: pt3(:,:)
INTEGER, TARGET :: b(:,:)
```

Note that the following is an examples of an illegal pointer declaration:

```
REAL, POINTER, DIMENSION(10) :: pt           !illegal
```

The `POINTER` attribute is incompatible with the `ALLOCATABLE`, `EXTERNAL`, `INTENT`, `INTRINSIC`, `PARAMETER` and `TARGET` attributes. The `TARGET` attribute is incompatible with the `EXTERNAL`, `INTRINSIC`, `PARAMETER` and `POINTER` attributes.

11.3 Pointer assignment

There are two operators which may act on pointers:

- The pointer assignment operator (=>)
- The assignment operator (=)

To associate a pointer with a target use the pointer assignment operator (=>):

```
pointer => target
```

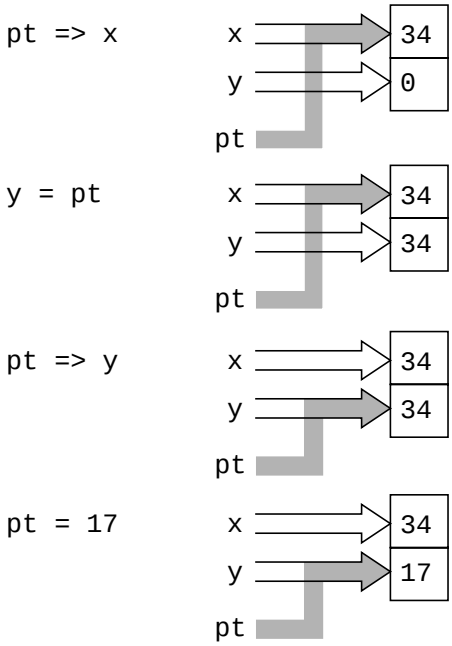
Where `pointer` is a pointer variable and `target` is any valid target. `pointer` may now be used as an alias to the data stored by `target`. The pointer assignment operator also allocates storage required by the pointer.

To change the value of a pointer's target (just like changing the value of a variable) use the usual assignment operator (=). This is just as it would be for other variable assignment with a pointer used as an alias to another variable.

The following are examples of pointer assignment:

```
INTEGER, POINTER :: pt
INTEGER, TARGET :: x=34, y=0
...
pt => x    ! pt points to x
y = pt     ! y equals x
pt => y    ! pt points to y
pt = 17    ! y equals 17
```

The declaration statements specify a three variables, `pt` is an integer pointer, while `x` and `y` are possible pointer targets. The first executable statement associates a target with `pt`. The second executable statement changes the value of y to be the same as `pt`'s target, this would only be allowed when `pt` has an associated target. The third executable statement re-assigns the pointer to another target. Finally, the fourth executable statement assigns a new value, 17, to `pt`'s target (not `pt` itself!). The effect of the above statements is illustrated below.



It is possible to assign a target to a pointer by using another pointer. For example:

```
REAL, POINTER :: pt1, pt2
...
pt2 => pt1      !legal only if pt1 has an associated target
```

Although this may appear to be a pointer pointing to another pointer, `pt2` does not point to `pt1` itself but to `pt1`'s target. It is wrong to think of 'chains of pointers', one pointing to another. Instead all pointers become associated with the same target.

Beware, of using the following statements, they are both illegal:

```
pt1 => 17          !constant expression is not valid target
pt2 => pt1 + 3     !arithmetic expression is not valid target
```

11.3.1 Dereferencing

Where a pointer appears as an alias to a variable it is automatically dereferenced; that is the value of the target is used rather than the pointer itself. For a pointer to be dereferenced in this way requires that it be associated with a target.

Pointer are automatically dereferenced when they appear:

- As part of an expression.
- In I/O statements.

For example:

```
pt => a
b = pt                !b equals a, pt is dereferenced
IF( pt<0 ) pt=0       !pt dereferenced twice

WRITE(6,*) pt         !pt's target is written
READ(5,*) pt          !value stored by pt's target
```

11.4 Pointer association status

Pointers may be in one of three possible states:

- Associated - when pointing to a valid target.
- Disassociated - the result of a NULLIFY statement.
- Undefined - the initial state on declaration.

A pointer may become disassociated through the NULLIFY statement:

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

A pointer that has been nullified may be thought of as pointing ‘at nothing’.

The status of a pointer may be found using the intrinsic function:

```
ASSOCIATED ( list of pointers [,TARGET] )
```

The value returned by ASSOCIATED is either .TRUE. or .FALSE. When TARGET is absent, ASSOCIATED returns a value .TRUE. if the pointer is associated with a target and .FALSE. if the pointer has been nullified. When TARGET is present ASSOCIATED reports on whether the pointer points to the target in question. ASSOCIATED returns a value .TRUE. if the pointer is associated with TARGET and .FALSE. if the pointer points to another target or has been nullified.

It is an error to test the status of an undefined pointer, therefore it is good practice to nullify all pointers that are not immediately associated with a target after declaration.

The following example shows the use of the ASSOCIATED function and the NULLIFY statement:

```
REAL, POINTER :: pt1, pt2              !undefined status
REAL, TARGET :; t1, t2
LOGICAL :: test
pt1 => t1                              !pt1 associated
pt2 => t2                              !pt2 associated
test = ASSOCIATED( pt1 )               ! .T.
test = ASSOCIATED( pt2 )               ! .T.
...
NULLIFY( pt1 )                         !pt1 disassociated
test = ASSOCIATED( pt1 )               ! .F.
test = ASSOCIATED( pt1, pt2 )          ! .F.
test = ASSOCIATED( pt2, TARGET=t2)     ! .T.
test = ASSOCIATED( pt2, TARGET=t1)     ! .F.
NULLIFY( pt1, pt2)                     !disassociated
```

The initial undefined status of the pointers is changed to associated by pointer assignment, there-after the ASSOCIATED function returns a value of .TRUE. for both pointers. Pointer pt1 is then nullified and its status tested again, note that more than one pointer status may be tested at once. The association status of pt2 with respect to a target is also tested. Finally both pointers are nullified in the same (last) statement.

11.5 Dynamic storage

As well as pointing to existing variables which have the TARGET attribute, pointers may be associated with blocks of dynamic memory. This memory is allocated through the ALLOCATE statement which creates an un-named variable or array of the specified size, and with the data type, rank, etc. of the pointer:

```
REAL, POINTER :: p, pa(:)
INTEGER :: n=100
```

```
...
ALLOCATE( p, pa(n) )
...
DEALLOCATE( p, pa )
```

In the above example p points to an area of dynamic memory and can hold a single, real number and pa points to a block of dynamic memory large enough to store 100 real numbers. When the memory is no longer required it may be deallocated using the DEALLOCATE statement. In this respect pointers behave very much like allocatable arrays.

11.5.1 Common errors

Allocating storage to pointers can provide a great degree of flexibility when programming, however care must be taken to avoid certain programming errors:

- Memory leaks can arise from allocating dynamic storage to the pointer and then re-assigning the pointer to another target:

  ```
  INTEGER, POINTER :: pt(:)
  ...
  ALLOCATE( pt(25) )
  NULLIFY( pt )              !wrong
  ```

  Since the pointer is the only way to reference the allocated storage (i.e. the allocated storage has no associated variable name other than the pointer) reassigning the pointer means the allocated storage can no longer be released. Therefore all allocated storage should be deallocated before modifying the pointer to it.

- It is possible to assign a pointer to a target, but then remove the target (by deallocating it or exiting a procedure to which it is local), in that case the pointer may be left ‘dangling’:

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

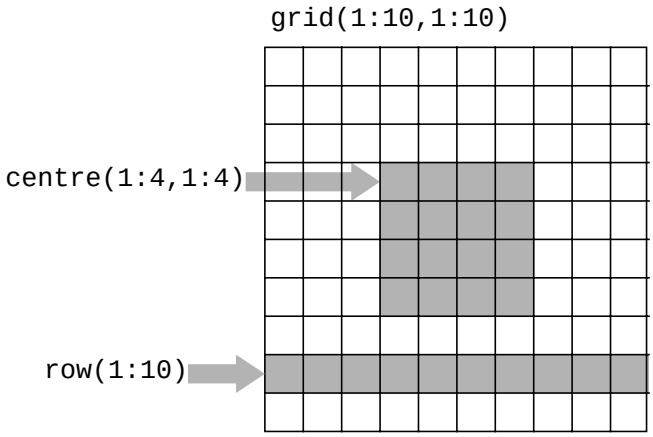
  In the above example p2 points to the storage allocated to p1, however when that storage is deallocated p2 no longer has a valid target and its state becomes undefined. In this case dereferencing p2 would produce unpredictable results.

  Programming errors like the above can be avoided by making sure that all pointers to a defunked target are nullified.

11.6 Array pointers

Pointers may act as dynamic aliases to arrays and array sections, such pointers are called array pointers. Array pointers can be useful when a particular section is referenced frequently and can save copying data. For example:

```
REAL, TARGET :: grid(10,10)
REAL, POINTER :: centre(:,:), row(:)
...
centre => grid(4:7,4:7)
row => grid(9,:)
```

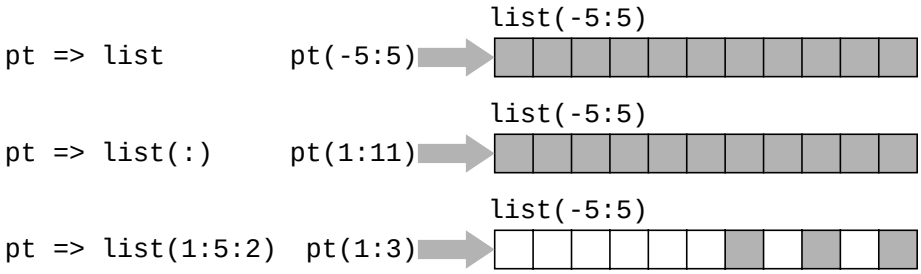


An array pointer can be associated with the whole array or just a section. The size and extent of an array pointer may change as required, just as with allocatable arrays. For example:

```
centre => grid(5:5,5:6)      !inner 4 elements of old centre
```

Note, an array pointer need not be deallocated before its extent or bounds are redefined.

```
INTEGER, TARGET :: list(-5:5)
INTEGER, POINTER :: pt(:)
INTEGER, DIMENSION(3) :: v = (/-1,4,-2/)
...
pt => list                !note bounds of pt
pt => list(:)             !note bounds of pt
pt => list(1:5:2)
pt => list( v )           !illegal
```



The extent (or bounds) of an array section are determined by the type of assignment used to assign the pointer. When an array pointer is aliased with an array the array pointer takes its extent form the target array; as with `pt => list` above, both have bounds -5:5. If the array pointer is aliased to an array section (even if the section covers the whole array) its lower bound in each dimension is 1; as with `pt => list(:)` above, `pt`'s extent is 1:11 while `list`'s extent is -5:5. So `pt(1)` is aliased to `list(-5)`, `pt(2)` to `list(-4)`, etc.

It is possible to associate an array pointer with an array section defined by a subscript triplet. It is not possible to associate one with an array section defined with a vector subscript, `v` above. The pointer assignment `pt => list(1:5:2)` is legal with `pt(1)` aliased to `list(1)`, `pt(2)` aliased to `list(3)` and `pt(3)` aliased to `list(5)`.

11.7 Derived data types

Pointers may be a component of a derived data type. They can take the place of allocatables arrays within a derived data type, or act as pointers to other objects, including other derived data types:

The dynamic nature of pointer arrays can provide varying amounts of storage for a derived data type:

```
TYPE data
   REAL, POINTER :: a(:)
END TYPE data
TYPE( data ) :: event(3)

DO i=1,3
   READ(5,*) n                          !n varies in loop
   ALLOCATE( event(i)%a(n) )
   READ(5,*) event(i)%a
END DO
```

The number of values differs for each event, the size of the array pointer depends on the input value `n`. When the data is no longer required the pointer arrays should be deallocated:

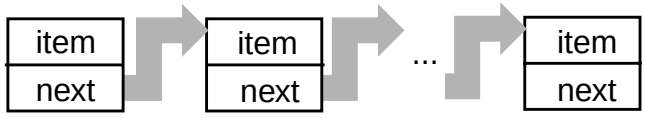
```
DO i=1,3
   DEALLOCATE( event(i)%a )
END DO
```

11.7.1 Linked lists

Pointers may point to other members of the same data type, and in this way create 'linked lists'. For example consider the following data type:

```
TYPE node
   REAL :: item
   TYPE( node ), POINTER :: next
END TYPE node
```

The derived type `node` contains a single object `item` (the data in the list) and a pointer `next` to another instance of `node`. Note the recursion-like property in the declaration allowing the pointer to reference its own data type.



Linked lists are a very powerful programming concept, their dynamic nature means that they may grow or shrink as required. Care must be taken to ensure pointers are set up and maintained correctly, the last pointer in the list is usually nullified. Details of how to implement, use and manipulate a linked list can be found in some of the reading material associated with these notes.

11.8 Pointer arguments

Just like other data types, pointers may be passed as arguments to procedures. There are however a few points to remember when using pointers as actual or dummy arguments:

- As with other variables, actual and dummy arguments must have the same data

type and rank. dummy arguments that are pointer may not have the INTENT attribute, since it would be unclear whether the intent would refer to the pointer itself or the associated target.

- Pointer arguments to external procedures require INTERFACE blocks.

When both the actual and dummy arguments are pointers, the target (if there is one) and association status is passed on call and again on return. It is important to ensure that a target remains valid when returning from a procedure (i.e. the target is not a local procedure variable), otherwise the pointer is left 'dangling'.

When the actual argument is a pointer and the corresponding dummy argument is not, the pointer is dereferenced and it is the target that is copied to the dummy argument. On return the target takes the value of the dummy argument. This requires the actual argument to be associated with a target when the procedure is referenced.

For example:

```
PROGRAM prog
   INTERFACE                      !needed for external subroutine
      SUBROTINE suba( a )
         REAL, POINTER :: a(:)
      END SUBROUTINE suba
   END INTERFACE
   REAL, POINTER :: pt(:)
   REAL, TARGET :: data(100)
   ...
   pt => data
   CALL suba( pt )
   CALL subb( pt )
   ...
CONTAINS
   SUBROUTINE subb( b )           !internal
      REAL, DIMENSION(:) :: b     !assumed shape of 100
      ...
   END SUBROUTINE subb
END PROGRAM prog

SUBROUTINE suba( a )              !external subroutine
   REAL, POINTER :: a(:)          !a points to data
   ...
END SUBROUTINE suba
```

It is not possible for a non-pointer actual argument to correspond with a pointer dummy argument.

11.9 Pointer functions

Functions may return pointers as their result. This is most useful where the size of the result depends on the function's calculation. Note that:

- The result must have the POINTER attribute.
- The returning function must have a valid target or have been nullified.
- Pointer results from external procedures require INTERFACE blocks.

For example:

```
INTERFACE
   FUNCTION max_row ( a )
      REAl, TARGET :: a(:,:)
      REAL, POINTER :: max_row(:)
   END FUNCTION max_row
```

```
END INTERFACE
REAL, TARGET :: a(3,3)
REAL, POINTER :: p(:)
...
p => max_row ( a )
...

FUNCTION max_row ( a )                !external
   REAL, TARGET :: a(:,:)
   REAL, POINTER :: max_row(:)        !function result
   INTEGER :: location(2)
      location = MAXLOC( a )          !row and column of max value
      max_row  => a(location(1),:)    !pointer to max row
END FUNCTION max_row
```

Here the external function `max_row` returns the row of a matrix containing the largest value. The pointer result is only allowed to point to the dummy argument a because it is declared as a target, (otherwise it would have been a local array and left the pointer dangling on return). Notice the function result is used on the right hand side of a pointer assignment statement. A pointer result may be used as part of an expression in which case it must be associated with a target.

11.10 Exercises

1. Write a declaration statement for each of the following pointers and their targets:

(a) A pointer to a single element of an array of 20 integers.

(b) A pointer to a character string of length 10.

(c) An array pointer to a row of a 10 by 20 element real array.

(d) A derived data type holding a real number three pointers to neighbouring nodes, left, right and up (this kind of derived data structure may be used to represent a binary tree).

2. For the pointer and target in the following declarations write an expression to associate the pointer with:

(a) The first row of the target.

(b) A loop which associates the pointer with each column of the target in turn.

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

3. Write a program containing an integer pointer and two targets. Nullify and report the initial status of the pointer (using the ASSOCIATED intrinsic function). Then associate the pointer with each of the targets in turn and output their values to the screen. Finally ensure the pointer ends with the status 'not currently associated'.

4. Write a program containing a derived data type. The data type represents different experiments and should hold the number of reading taken in an experiment (an integer) and values for each of the readings (real array pointer).

Read in the number and values for a set of experimental readings, say 4, and output their mean. Deallocate all pointers before the program finishes.

5. Write an internal function that takes a single rank one, integer array as an argument and returns an array pointer to all elements with non-zero values as a result. The function will need to count the number of zero's in the array (use the COUNT intrinsic), allocate the required storage and copy each non-zero value into that storage. Write a program to test the function.

12 Intrinsic procedures

Fortran 90 offers many intrinsic function and subroutines, the following lists provide a quick reference to their format and use.

In the following intrinsic function definitions arguments are usually named according to their types (`I` for integer `C` for character, etc.), including those detained below. Optional arguments are shown in square brackets [ ], and keywords for the argument names are those given.

`KIND` - describes the `KIND` number.

`SET` - a string containing a set of characters.

`BACK` - a logical used to determine the direction a string is to be searched.

`MASK` - a logical array used to identfy those element which are to take part in the desired operation.

`DIM` - a selected dimension of an argument (an integer).

12.1 Argument presence enquiry

`PRESENT( A )` - true if `A` is present.

12.2 Numeric functions

`ABS( A )` - return the absolute value of `A`.

`AIMAG( Z )` - return the imaginary part of complex number `Z`.

`AINT( A [, KIND] )` - returns a value `A` truncated to a whole number.

`ANINT( A [, KIND] )` - returns a value rounded to the nearest value of `A`.

`CEILING( A )` - returns the lowest integer greater than or equal to `A`.

`CMPLX( X [, Y][, KIND] )` - converts `A` to a complex number.

`CONJG( Z )` - returns the conjugate of a complex number.

`DBLE( A )` - converts A to a double precision real.

`DIM( X, Y )` - returns the maximum of `X-Y` or 0.

`DPROD( X, Y )` - returns a double precision product.

`FLOOR( A )` - returns the largest integer less than or equal to `A`.

`INT( A [, KIND] )` - converts to an integer.

`MAX( A1, A2 [, A3...] )` - returns the maximum value.

`MIN( A1, A2 [, A3...] )` - returns the minimum value.

MOD(A, P) - returns remainder modulo P i.e. A-INT(A/P)*P.

MODULO(A, P) - A modulo P.

NINT(A [, KIND]) - returns the nearest integer to A.

REAL(A [, KIND]) - converts to a real.

SIGN(A, B) - returns the absolute value of A times the sign of B.

12.3 Mathematical functions

ACOS(X) - arccosine.

ASIN(X) - arcsine.

ATAN(X) - arctan.

ATAN2(X, Y) - arctan.

COS(X) - cosine.

COSH(X) - hyperbolic cosine.

EXP(X) - exponential.

LOG(X) - natural logarithm.

LOG10(X) - base 10 logarithm.

SIN(X) - sine.

SINH(X) - hyperbolic sine.

SQRT(X) - square root.

TAN(X) - tan.

TANH(X) - hyperbolic tan.

12.4 Character functions

ACHAR(I) - returns the I^{th} character in the ASCII collating sequence.

ADJUSTL(STRING) - adjusts string left by removing any leading blanks and inserting trailing blanks.

ADJUSTR(STRING) - adjusts string right by removing trailing blanks and inserting leading blanks.

CHAR(I [, KIND]) - returns the I^{th} character in the machine specific collating sequence.

IACHAR(C) - returns the position of the character in the ASCII collating sequence.

ICHAR(C) - returns the position of the character in the machine specific collating sequence.

INDEX(STRING, SUBSTRING [, BACK]) - returns the leftmost (rightmost if BACK is .TRUE.) starting position of SUBSTRING within STRING.

LEN(STRING) - returns the length of a string.

LEN_TRIM(STRING) - returns the length of a string without trailing blanks.

LGE(STRING_A, STRING_B) - lexically greater than or equal to.

LGT(STRIN_A1, STRING_B) - lexically greater than.

LLE(STRING_A, STRING_B) - lexically less than or equal to.

LLT(STRING_A, STRING_B) - lexically less than.

REPEAT(STRING, NCOPIES) - repeats concatenation.

SCAN(STRING, SET [, BACK]) - returns the index of the leftmost (rightmost if BACK is .TRUE.) character of STRING that belong to SET, or 0 if none belong.

TRIM(STRING) - removes training spaces from a string.

VERIFY(STRING, SET [, BACK]) - returns zero if all characters in STRING belong to SET or the index of the leftmost (rightmost if BACK is .TRUE.) that does not.

12.5 KIND functions

KIND(X) - returns the kind type parameter value.

SELECTED_INT_KIND(R) - kind of type parameter for specified exponent range.

SELECTED_REAL_KIND([P] [,R]) - kind of type parameter for specified precision and exponent range.

12.6 Logical functions

LOGICAL(L [, KIND]) - convert between different logical kinds.

12.7 Numeric enquiry functions

DIGITS(X) - returns the number of significant digits in the model.

EPSILON(X) - returns the smallest value such that REAL(1.0, KIND(X)) + EPSILON(X) is not equal to REAL(1.0, KIND(X)).

HUGE(X) - returns the largest number in the model.

MAXEXPONENT(X) - returns the maximum exponent value in the model.

MINEXPONENT(X) - returns the minimum exponent value in the model.

PRECISION(X) - returns the decimal precision.

RADIX(X) - returns the base of the model.

RANGE(X) - returns the decimal exponent range.

TINY(X) - returns the smallest positive number in the model.

12.8 Bit enquiry functions

BIT_SIZE(I) - returns the number of bits in the model.

12.9 Bit manipulation functions

BTEST(I, POS) - is .TRUE. if bit POS of integer I has a value 1.

IAND(I, J) - logical .AND. on the bits of integers I and J.

IBCLR(I, POS) - clears bit POS of interger I to 0.

IBITS(I, POS, LEN) - extracts a sequence of bits length LEN from integer I starting at POS

IBSET(I, POS) - sets bit POS of integer I to 1.

IEOR(I, J) - performas an exclusive .OR. on the bits of integers I and J.

IOR(I, J) - performes an inclusive .OR. on the bits of integers I and J.

ISHIFT(I, SHIFT) - logical shift of the bits.

ISHIFTC(I, SHIFT [, SIZE]) - logical circular shift on a set of bits on the right.

NOT(I) - logical complement on the bits.

12.10 Transfer functions

TRANSFER(SOURCE, MOLD [, SIZE]) - converts SOURCE to the type of MOLD.

12.11 Floating point manipulation functions

EXPONENT(X) - returns the exponent part of X.

FRACTION(X) - returns the fractional part of X.

NEAREST(X, S) - returns the nearest different machine specific number in the direction given by the sign of S.

RRSPACING(X) - returns the reciprocal of the relative spacing of model numbers near X.

SCALE(X) - multiple X by its base to power I.

SET_EXPONENT(X, I) - sets the expontnt part of X to be I.

SPACING(X) - returns the absolute spacing of model numbers near X.

12.12 Vector and matrix functions

DOT_PRODUCT(VECTOR_A, VECTOR_B) - returns the dot product of two vectors (rank one arrays).

MATMUL(MATRIX_A, MATRIX_B) - returns the product of two matricies.

12.13 Array reduction functions

ALL(MASK [, DIM]) - returns .TRUE. if all elements of MASK are .TRUE.

ANY(MASK [, DIM]) - returns .TRUE. if any elements of MASK are .TRUE.

COUNT(MASK [, DIM]) - returns the number of elements of MASK that are .TRUE.

MAXVAL(ARRAY [, DIM] [,MASK]) - returns the value of the maximum array element.

MINVAL(ARRAY [, DIM] [,MASK]) - returns the value of the minimum array element.

PRODUCT(ARRAY [, DIM] [, MASK]) - returns the product of array elements

SUM(ARRAY [, DIM] [, MASK]) - returns the sum of array elements.

12.14 Array enquiry functions

ALLOCATED(ARRAY) - returns .TRUE. if ARRAY is allocated.

LBOUND(ARRAY [, DIM]) - returns the lower bounds of the array.

SHAPE(SOURCE) - returns the array (or scalar) shape.

SIZE(ARRAY [, DIM]) - returns the total number of elements in an array.

UBOUND(ARRAY [, DIM]) - returns the upper bounds of the array.

12.15 Array constructor functions

MERGE(TSOURCE, FSOURCE, MASK) - returns value(s) of TSOURCE when MASK is .TRUE. and FSOURCE otherwise.

PACK(ARRAY, MASK [, VECTOR]) - pack elements of ARRAY corresponding to true elements of MASK into a rank one result

SPREAD(SOURCE, DIM, NCOPIES) - returns an array of rank one greater than SOURCE containing NCOPIES of SOURCE.

UNPACK(VECTOR, MASK, FIELD) - unpack elements of VECTOR corresponding to true elements of MASK.

12.16 Array reshape and manipulation functions

CSHIFT(ARRAY, SHIFT [, DIM]) - performs a circular shift.

EOSHIFT(ARRAY, SHIFT [, BOUNDARY] [, DIM]) - performs an end-off shift.

MAXLOC(ARRAY [, MASK]) - returns the location of the maximum element.

MINLOC(ARRAY [, MASK]) - returns the location of the minimum element.

RESHAPE(SOURCE, SHAPE [, PAD] [, ORDER]) - rehapes SOURCE to shape SHAPE

TRANSPOSE(MATRIX) - transpose a matrix (rank two array).

12.17 Pointer association status enquiry functions

ASSOCIATED(POINTER [, TARGET]) - returns .TRUE. if POINTER is associated.

12.18 Intrinsic subroutines

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

MVBITS(FROM, FROMPOS, LEN, TO TOPOS) - copy bits.

RANDOM_NUMBER(HARVEST) - random number in the range 0-1 (inclusive).

RANDOM_SEED([SIZE] [, PUT] [, GET]) - initialise or reset the random number generator.

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

13 Further reading

Fortran 90 handbook - J.C. Adams et. al., McGraw-Hill, 1992.

Programmer's Guide to Fortran 90 - W.S. Brainerd et. al., Unicomp, 1994.

Fortran 90 - M. Counihan, Pitman, 1991.

Fortran 90 programming - T.M.R. Ellis et. al., Wesley, 1994.

Fortran 90 for Scientists and Engineers - B.D. Hahn, Edward Arnold, 1994.

Fortran 90 Explained - M. Metcalf and J. Ried, Oxford University Press, 1992.

Programming in Fortran 90 - J.S. Morgan and J.L. Schonfelder, Alfred Walker Ltd, 1993.

Programming in Fortran 90 - I.M. Smith, Wiley.