Parallel Computer Centre

[Next] [Previous] [Top]

Introduction to FORTRAN 90

Student Notes

Rob Davies
Cardiff HPC Training and Education Centre

Alan Rea
Parallel Computer Centre, Belfast

Dimitris Tsaptsinos
SEL - HPC

[Next] [Previous] [Top]

[Next] [Previous] [Top]

1 Introduction

1.1 Programming in general

A program is the tool a user employs to exploit the power of the computer. A program is written in a language which is understood by the computer hardware. 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.

1.1.1 Available programming languages

There are hundreds of programming languages. A general txonomy of the available programming languages is given below.

- Machine codes use strings of 0s and 1s to express instructions and they dependent on the underlying hardware.
- Assembly languages are also dependent on hardware and utilise a symbolic form to express instructions.
- High level languages were developed to ease the programming effort and to provide hardware independence. Despite that they are multi-purpose languages they have different strengths. For example, Fortran is popular with the scientific and engineering community, Cobol is used for business applications and C for systems programming.
- Logic programming involves the construction of a database with facts and rules and the program examines the database to locate one or more rule that apply with a given input.
- Functional programming involves the construction of functions. A function is written using normal mathematical principles and the computer evaluates the function and prints the result(s).
- Simulation languages are used to model activities of discrete systems (traffic flow) and are used to

predict the behaviour (traffic jams) of the system by asking hypothetical questions (traffic density increase)

- String manipulation languages perform pattern matching where strings of characters are compared.

- Object-oriented languages such as Smalltalk provide programming environments by integrating the language with support tools. Languages such as C++ encourage the decomposition of a problem into smaller sub-problems by allowing encapsulation, polymorphism and inheritance.

- 4GLs remove the need for a user to write programs from scratch by using pre-programmed forms.

1.1.2 Choosing a programming language

The choice of what language to employ depends on two factors:

- Practical considerations - these include cost consideration (a company might have already invested a lot of money in a particular language by purchasing appropriate compilers/hardware. Existing Fortran programs easier to change to 90 rather than to C); application consideration (Fortran is the language for number crunching but would not use for database development); expertise consideration (cost for re-training staff in a new language).

- Language considerations: In general one desires a language with a notation that fits the problem, simple to write and learn, powerful operations etc. Fortran is superior to other languages for numerical computation, many diverse and reliable libraries of routines are available, an official standard exists which helps towards portability.

1.2 History

Fortran (mathematical FORmula TRANslation system) was originally developed in 1954 by IBM. Fortran was one the first to allow the programmer to use a higher level language rather than machine code (0s and 1s) or assembly language (using mnemonics). 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 (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 was released which defined a solid base for further development of the language.

In 1978 the second ANSI standard 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 was released in 1991, with a new revision expected within 10 years.

1.3 ANSI Standard

Fortran 90 is a superset of Fortran 77. New facilities for array type operations, new methods for specifying precision, free form, recursion, dynamic arrays etc. were introduced. Despite that the whole Fortran77 is included the new ANSI standard proposes that some of the Fortran77 features are obsolete and will be removed in the next version.

In theory a Fortran 77 program should compile successfully with a Fortran 90 compiler with minor changes. This is the last time a reference to Fortran 77 is made and it is recommended that programmers new to Fortran not to consult any Fortran 77 books.

The Fortran 90 version was augmented with a number of new features because previously modern developments were not accommodated. Developments such as the recent importance of dynamic data structures and the (re)introduction of parallel architecture.

Comparing with other languages, and only for number crunching, one can see that Fortran90 scores higher on numeric polymorphism, decimal precision selection, real Kind type etc. Only 90 has data parallel capabilities meaningful for numeric computation which are missing from other languages. Also 90's data abstraction is not as powerful as in C++ but it avoids the complexities of object-oriented programming.

1.4 The Program - Planning

Writing a program is not a floating task. Previous to code writing one has to go through certain stages:

- Analyse and specify requirements.

- Design the solution.

- Code the solution using the chosen programming language.

At the end of coding, verification, testing and maintenance are also required.

The stages are iterative and the sooner errors are found the better. These stages though are not discussed in this course but the interested reader can consult any software book for more information. Here, the concentration lies with coding with a brief introduction to design using algorithms.

1.5 The Program - Algorithms

The design phase is one of the most important and usually difficult stage. One tool used to design the program is the algorithm. With an algorithm the steps required to carry out a given task are clearly described. The algorithm will include instructions for how to:

- accept information

- display information

- transformations
- how to select decisions
- how to repeat sub-tasks
- when to terminate

Writing musical score is an algorithm where the notes express tasks. For programming though an algorithm is expressed using English-like instructions. An algorithm is independent of the programming language or the computer hardware to be used, but influenced. A programming language acts like a convenient medium for expressing algorithm and the computer is simply the medium of execution. The solution process expressed as an algorithm can not obviously be executed since computers do not handle well the idiosyncrasies of a natural language or subset of it (but moving towards it)

1.6 The Program - Example of an algorithm

Consider the following algorithm;

1. Get a number
2. Save the number
3. Get a new number
4. While there is a new number
5. If the new number is greater than that saved

   Save the new number

   end if
6. Get a new number

   end while
7. Print saved number

This is a proposed solution for finding and displaying the greatest number from a given list of numbers. The input terminates when the user enters the blank character.

Notice the English-like expressions. Obviously, one can use Read or Input instead of Get; or store instead of save; or '>' instead of greater than. Notice also the numbering system. This helps towards stepwise refinement. For example, if statement X needed more clarification then the new statements take the value X.1 to X.n. This makes referencing statements easier.

Notice the indentation and the end-if and end-while which make clear where a comparison / loop terminates.

1.7 The minimum program

Consider the following program

```
PROGRAM nothing

! does nothing

END PROGRAM nothing
```

This is probably the simplest Fortran90 program. It does nothing. The first statement simply tells the compiler that a program named nothing is to follow. The second statement is a comment (because of the exclamation mark) and it is ignored by the compiler. The third and last statement informs the compiler that the program terminates at that point. Notice that statements between PROGRAM and END are executed in the order that they are written (not strictly true but ok for the moment). Keywords such as PROGRAM and END are written in capitals just for illustration purposes. Small case or a mixture of upper and lower case is acceptable. So PROGRAM, Program, PROgrAM are all acceptable.

Consider the following (more complicated) program

```
PROGRAM hi

! display a message

WRITE(*,*) 'Hello World!'

END PROGRAM hi
```

The above program introduces the concept of displaying information to the screen (or other devices). Running this program the message Hello World (without the quotes) will appear on the screen. This is achieved by employing the keyword WRITE and typing the appropriate message between single quotes. One can extend the program to, say, display the name of the current user. Then using in a similar fashion another available keyword (READ) the user can enter his/her name and by changing the WRITE statement he/she can display the name.

1.8 Compilation

Once the program has been designed and entered into a file then the following steps are followed:

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

  ```
  f90 filename.f90
  ```

  its purpose is to translate the high-level statements into machine 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 (referencing a variable with no declaration). This step generates the object code version which is stored in a file of the same name but different extension.

- Linking step: This might be initiated by the compiler and its purpose is to insert code for a referenced operation from the library. This generates the executable code version which again is stored in a file with a different extension.
- Execution step: This is initiated by the programmer/user, by typing a.out, and its purpose is to run the program to get some answers. During execution the program might crash if it comes across an execution error (most common execution error is the attempt to divide by zero).

  Notice that logical errors (multiply rather than add) can not be checked by the compiler and it is the responsibility of the designer 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.

[Next] [Previous] [Top]

2 Variables and Statements

2.1 Data types

As humans we process various forms of information using senses such as sight, hearing, smell and touch. Much of the material produced by the academic community is processed visually, for example books, papers or notes. When processing numeric material we write the data as a stream of numeric characters, such as

```
365

96.4

3.14159
```

and then read the numbers as streams of characters. However in our minds we can think of each number as a numeric value rather than a series of characters. This is similar to the way a computer processes numeric data.

A computer programmer writing a program to manipulate numeric values uniquely
identifies each value by a name which refers to a discrete object remembered in the computer's memory. Thus the previous values may be identified by names such as:

```
daysinyear

temperature

pi
```

Note that it is good programming practice to use names that relate to the value that is being referred to.

There are two main forms of numeric data, namely INTEGER and REAL. Integers are essentially a restricted set of the mathematical whole numbers and as such may only have discrete values (i.e. no fractional part). The following are valid integer values:

```
-3124 -96 0 10 365
```

Real numbers may have a fractional part and have a greater range of possible values. For example:

```
10.3 -8.45 0.00002
```

There are two forms in which real values may be written. The examples above are in fixed point form but floating point form (similar to scientific notation) may be used, for example:

```
2.576x1032 1.3 x 10-22
```

may be written in Fortran as

```
2.576E32 1.3E-22
```

where the E stands for ‘exponent’ or ‘multiplied by 10 to the power of’.

The integer and real values listed above, when used in a computer program, are known as literal constants.

Why reals and integers? Integers are more accurate for discrete values and are processed faster but reals are necessary for many scientific calculations.

In addition to the basic real and integer numbers there are other types of number such as double precision (which have more significant figures than the default REAL type) and complex numbers (with a real and imaginary part).

As well as numbers, 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).

2.2 Naming Convention

In Fortran objects are referred to by name. The naming convention permits names of between 1 and 31 alphanumeric characters (letters, numerals and the underscore character) with the restriction that the first character must be a letter. There is no case sensitivity in Fortran, the lower and uppercase versions of a character are treated as equivalent.

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. The programmer should take great care when naming objects not to use any words which form part of the language. In the 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.

Table 1: Examples of Valid and Invalid Names

Valid Names	Valid (but do not use)	Invalid
x	REAL	ten.green.bottles
x1	INTEGER	1x
mass	DO	a thing
pressure	SUBROUTINE	two-times
day_of_week	PROGRAM	_time

2.3 Variables

Programs make use of objects whose value is allowed to change while it is running, these are known as variables. A variable must have an associated data type, such as REAL or INTEGER, and be identified at the start of the section of the program in which it is used (see later). This is referred to as declaring 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 described below.

```
REAL :: temperature=96.4

INTEGER :: daysinyear=365, monthsinyear=12, weeksinyear=52
```

The general form of a variable declaration is:

```
TYPE [,attr] :: variable list
```

Where attr are optional Fortran 90 ‘commands’ to further define the properties of variables. Attributes will be described throughout the course as they are introduced.

2.3.1 Implicit Declaration

Fortran 90 permits variables to be typed and declared implicitly, that is without using a variable declaration as given above. An implicit declaration is performed whenever a name appears which has not been explicitly declared and the program section does not contain the statement IMPLICIT NONE (see sample program). The implicit declaration facility is provided to comply with earlier definitions of the Fortran language and as this has been the cause of many programming problems this feature should be disabled using the IMPLICT NONE statement. Variables are typed 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.

2.3.2 Parameters

The term parameter in Fortran is slightly misleading, it refers to a value which will be constant, for example the programmer will want the value of pi to be unaltered during a program. Therefore pi may be defined as

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

The word REAL defines the type of pi and the word PARAMETER is an attribute of the REAL object which is known as pi and has the value 3.141592. Parameters may also be defined for other data types, for example:

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

The objects declared to be parameters may not be altered in the program.

2.4 Arithmetic Expressions

Variables, parameters and numeric constants may be combined using the following operators:

+ Addition

- Subtraction

* Multiplication

/ Division

** Exponentiation

For example

```
cost * number
cost * number + postage
10 + 3
4 * pi
1 / pressure
pi * radius * radius
```

The expressions formed with arithmetic operators may be used in a variety of ways, one of which, the assignment statement, is described in the next section.

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

```
pi*radius**2
```

might be interpreted as

```
(pi*radius)**2
```

The section of the expression which appears inside brackets is always evaluated first. In expressions which contain more than one operator the operations are carried out in an order which is determined by what are known as the "rules of precedence". The following table lists the priority or order of execution of the various operators.

Table 1: Operator Precedence

Precedence	Operator
1	()
2	**
3	*/
4	+ -

The operators are evaluated in order of ascending precedence, that is, brackets first, then ** followed by * / and finally + -. Operators of equal precedence are evaluated working from left to right across the expression.

2.5 Assignment Statement

The expressions formed using arithmetic operators may be used to assign a value to a variable using the assignment operator, thus

```
area = pi*radius*radius
```

The assignment statement has the general form:

```
variable = expression
```

2.6 Simple Input and Output

On most computer systems the user can tell the program what values to perform a calculation upon by typing these at a keyboard. This is known as input and the values are assigned to the correct variables using the READ statement. The user will also wish to know the results generated by the program and this will usually be displayed on a screen using the WRITE statement - this is known as output.

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

```
READ(5,*)radius
READ(*,*) radius
```

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

```
WRITE(6,*) area
WRITE(*,*) area
```

The characters "(5,*)" should appear after every READ and the characters "(6,*)" after every WRITE (note that "(*,*)" may appear with either the READ or WRITE statements). The significance of these will be explained in a later section.

Several variables (or expressions) may be specified on one READ or WRITE statement as follows:

```
READ(5,*) length, breadth
```

```
WRITE(6,*) temperature, pressure, mass

WRITE(*,*) pi*radius**2, 2.0
```

2.7 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 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 in the future especially as this is likely to be performed by a different programmer.

```
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
```

2.8 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

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
- there is an explanation of the program in the form of comment statements
- the variable declarations follow the program statement
- the variable declaration are grouped together and appear before statements such as assignments statements and input/output statements
- blank lines are used to emphasize the different sections of the program

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.

2.9 Derived Data Types

2.9.1 Definition and specification

In many algorithms there are data items which can be grouped together to form an aggregate structure. 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, thus 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
```

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. If a type, point, is defined

```
TYPE point

REAL :: x_coord, y_coord

ENDTYPE point
```

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

```
TYPE circle

TYPE (point) :: centre

INTEGER :: radius

REAL :: area

ENDTYPE circle
```

The general form of a derived type definition is

```
TYPE type name

component definition statement

component definition statement

.....

END TYPE [type name]
```

This is a simplified version of the complete specification of a derived type, other elements may be added to this definition later. Note that the typename is optional on the ENDTYPE statement but should always be included to improve program clarity.

The general form of the variable declaration statement may be modified to included the specification of a derived type

```
TYPE [(type name)] [,attr] :: variable list
```

2.9.2 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%centre%x_coord = 5.0

cir_a%centre%y_coord = 6.0
```

2.10 Exercises

1. Which of the following values represent integers and which represent real numbers?

   ```
   0 1 1.2E-10 -1 -1.0

   0.0 0.1 1024 64.0 -1.56E12
   ```

2. Which of the following are invalid names in Fortran and state 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 type of each result. Finally, insert brackets to clarify the meaning of these expressions according to the operator precedence table.

   ```
   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
   ```

4. Write definitions of derived types 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.

   (e) a type containing 3 reals and 2 integers.

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.)

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

[Next] [Previous] [Top]

[Next] [Previous] [Top]

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 characters and words 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 (6,*) 'Please type a value for the radius of a circle'

READ (5,*) radius

area = pi*radius*radius

WRITE (6,*) '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 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 (note the use of the keyword len)

```
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 leftmost 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 + - 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 rightmost characters are truncated and if too few characters are specified the rightmost 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 leftmost 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 strings. These intrinsic functions perform a number of commonly required character manipulations which programmers would otherwise have to write themselves.

Table 1: Intrinsic functions for character strings.

Function	Action
LEN(string)	returns the length of a character string
INDEX(sub,string)	finds the location of a substring in another string: returns 0 if not found.
CHAR(int)	converts an integer into a character
ICHAR(ch)	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('al', firstname) !pos=2
firstname = 'Fred'
pos = INDEX('al', firstname) !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 expressions, using the 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.

4. Write a program which would test the results of the expressions defined in the previous exercise.

5. 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.

6. Write a Fortran 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
```

[Next] [Previous] [Top]

[Next] [Previous] [Top]

4 Arrays

4.1 Terminology

4.1.1 Arrays and elements

Previous modules introduced simple data types, such as integer, real and character. In this module a structured data type called array is introduced.

An array is a collection of (scalar) data, all of the same type, whose individual elements are arranged in a regular pattern.

There are 3 possible types of arrays depending on the binding of an array to an amount of storage.

Static arrays: their size is fixed when the array is declared and can not be altered during execution. This is inflexible for certain circumstances (one has to re-entry the program, change the dimension(s) and re-compile) and wasteful in terms of storage space (since one might declare a very large array to avoid the previous problem)

Semi-dynamic arrays: the size of an array is determined after entering a subroutine and arrays can be created to match the exact size required but can only be used for a subroutine. In Fortran90 such arrays are called assumed-shape, and 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 slow run-time performance and lack of any bound 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 (one can easily operate on various array areas in different ways).

- Easier to understand the code (notational convenience).
- Inherent Data Parallelism (perform a similar computation on many data objects simultaneously).
- Optimization opportunities (for compiler designers).
- Reduction of program size.

This is an example of an array which contains integer numbers:

5	7	13	24	0	65	5	22

Assuming at the moment that the index (or 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

the first three elements are 5, 7, 13 with Indices of 1, 2 and 3 respectively and they form what is known as a section.

4.1.2 Array properties

The term Rank (or alternatively called dimension) refers to the number of subscripts needed to locate an element within an array. A scalar variable has a rank of zero.

Vector: An array with a rank of one is called a vector.

Matrix: An array with a rank of 2 or greater is called a matrix

Consider again the following array:

5	7	13	24	0	65	5	22

This array represents a vector since it is one-dimensional.

Consider the following array:

5	7	13	24
0	65	5	22

This array represents a matrix since it is two-dimensional.

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

The term Extent 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 term Shape is a vector 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 but more about this later. Both vector and matrix above have a size of 8.

The term Conformance refers to arrays that have the same shape. This is a condition for array to array operations. Obviously an operation between an array and a scalar satisfies the conformance criterion. In such a case the scalar operation is repeated for all the elements of the array, as shown later.

4.2 Specifications

To specify an array the following attributes of the array must be known:

The name given to the array (e.g. Student_mark). The name given to the array is up to 31 alphanumeric characters including underscore but the first character must be a letter.

The type of the elements (e.g. integer). All elements must be of the same type and the type can be integer, real, logical, character, or derived.

The dimensions of the array (e.g. 1 dimension). Up to 7 dimensions are allowed

The lower and upper bounds for each dimension (e.g 1 and 8). Declaring the lower bound is optional. If the lower bound is not specified Fortran90 assumes that the index begins with 1. Notice that the type of the bounds is always integer. The alternate and equivalent forms used to declare an array are as follows:

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

type [,attr] :: name (bound)
```

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

The following declarations are equivalent. Both declare an integer array a with 6 elements; an array b with 10 real elements and a logical 2-dimensional array named 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 the same type but different bounds need to be declared. The choice is influenced by the style followed by the programmer but certain circumstances might dictate the use of one form rather than another.

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

```
INTEGER, DIMENSION(8) :: x,y

REAL:: alpha(1:3), beta(4:9)

REAL, DIMENSION(0:5,12:45,6) :: data

CHARACTER(len=10) :: names(25)
```

The first example declares two arrays of the same dimension and type, therefore the dimension attribute form is employed. The second example declares two arrays of the same type but different dimension hence the array specification form is followed. For the third and fourth examples any of the two forms could have been used. The fourth example declares an array which has 25 elements with each element having a character string of size 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. Components are accessed as follows:

```
object(1)%position(1) !position 1 object 1

object(7)%position(2:) !positions 2 and 3 object 7

object(4)%position(:) !positions 1, 2 and 3 object 4

object(1:5)%position(1) !illegal object not array section.
```

Note that the array object cannot be used as an array section, although its components can (this is due to the unconventional storage requirements used by derived data types).

A third form is a mixture of the two above, as shown below:

```
type, DIMENSION(bound1) [,attr] :: aname, bname(bound2)
```

where aname takes the ‘default’ number of elements, but bname takes another explicitly defined value. This is still legal but avoid using it due to the confusion that might arise.

4.3 Array Sections

One is able to access individual elements or sections rather than the whole array. Individual elements and sections of an array are uniquely identified through subscripts, one per rank.

4.3.1 Individual elements

To access a single element of an array the name of the array followed by an integer value enclosed in parentheses is needed. The integer value is the index of the element. For multi-dimensional arrays a list of integer values is required separated by a comma.

array (index, [...])

a(5) refers to the fifth element of the array

b(4,2) refers to the element at the intersection of the 4th row and 2nd column.

For above examples assume that lower bound is 1 and use the following declarations:

```
REAL, DIMENSION(8) ::a

INTEGER, DIMENSION(5,4) ::b
```

4.3.2 Sections

To access a section of an array you need the name of the array followed by two integer values separated by a colon enclosed in parentheses. The integer values represent the indices of the section required. For multi-dimensional arrays use a list of integer values separated by a comma.

array ([lower]:[upper]:[step], [...]) where lower and upper default to the declared dimensions and step defaults to 1.

a(3:5) refers to elements 3, 4, 5 of the array

a(1:5:2) refers to elements 1, 3, 5 of the array

b(1:3, 2:4) refers to elements from rows 1 to 3 and columns 2 to 4.

Using colon: This is a facility that enables us to access whole or parts of columns or rows. For example, b(:4) refers to all elements of the fourth column.

Using subscripts: For example, alpha(i,j) refers to the element at the intersection of ith row and jth column. Subscripts i,j are defined previously within the program.

Using expressions: For example, alpha(2*k) refers to an element whose position is the result of the multiplication. The result of an expression must be an integer within the declared bounds.

Using stride: For example, beta(3,1:7:2) refers to elements 1,3,5,7 of the third row., beta(1,2:11:2) refers to elements 2,4,6,8,10 of the first row. This is a valid statement despite that the upper bound of the second dimension is 10.

4.4 Vector Subscripts

This is a clever way providing a shorthand facility for accessing particular elements of a large array. Vector subscripts are integer expressions of rank 1 and take the form (/list/). Consider the following example.

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

Here two arrays have been declared, a with size 9 and random with size 3. The third statement assigns the values of 6, 3, and 8 to the three elements of random. Whatever value existed beforehand now has been overwritten. Hence,

```
random(1)=6
random(2)=3
random(3)=8
```

The fourth statement uses random as an array of indices and assigns the value of 0.0 to the array elements of a. Expanding the left hand side we get

```
a(random)=a(random(1),random(2),random(3))=a(6,3,8)
```

Hence the third, sixth and eighth element of a are the ones being overwritten with a zero value.

The fifth statement demonstrates an alternative use of the vector subscript. Hence the 7th, 8th and 9th element of the array are assigned the value of 1.2

Care must be taken not to duplicate values in a vector subscript when used in the LHS of an expression as demonstrated with the illegal fourth statement below.

```
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
```

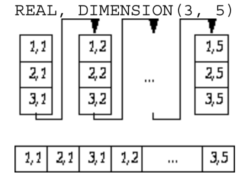
4.5 Array storage

The physical storage: How an array is stored in memory depends on the computer implementation.

The array element ordering: It is wrong to assume that two elements of an array are next to each other BUT conceptualise a linear sequence of the array elements with the first index changing first.

Consider the following example:

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



4.6 Array Assignment

4.6.1 Whole array assignment

This is to be used when the elements of an array need to be assigned with the same value (scalar) or by coping the values of another array. In the former the scalar is broadcasted to all the elements of the array. In the latter case the operands in the array expression must be conformable

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
```

The first assignment involves an array expression on the right hand side. Since a and b are conformable it is a valid statement. Each element of b takes the corresponding value of a multiplied by 2 and adding a 4 to the product.

The second assignment involves a scalar on the right hand side, hence there is automatic conformability. 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 are conformable then their product can be evaluated. The product refers to element by element multiplication. The result is another array which is conformable with c therefore each element of c takes the product of the corresponding elements in a and b.

The fourth assignment is illegal because the two arrays are not conformable.

4.6.2 Array section assignment

In case that sections of an array have to be assigned certain values conforming array sections may appear in the expressions.

Consider the following example:

```
REAL, DIMENSION(10) :: alpha, beta

REAL :: gamma(20)

alpha(1:5)=2.0

alpha(1:10:2)=beta(1:5)/6

alpha(2:10)=alpha(1:9)

gamma(11:20)=beta
```

The first assignment simply assigns the value of 2 to the first 5 elements of alpha, the rest of the elements remain intact.

The second assignment involves two conformable array sections, hence it is a valid statement. The following assignments are made:

```
alpha(1)=beta(1)/6

alpha(3)=beta(2)/6

alpha(5)=beta(3)/6

alpha(7)=beta(4)/6

alpha(9)=beta(5)/6
```

The third assignment shows a powerful operation using arrays where values are shifted automatically and without the need of DO loops. Therefore, the 9 elements of alpha starting from the second element take the value of the first 9 element of alpha, so at the end of the process the first two elements of alpha have the same value.

The last assignment demonstrates another important concept. Whereas beta and gamma are not conformable the section used by gamma satisfies the criterion so it is a valid statement.

4.6.3 Elemental intrinsic procedures

Elemental procedures are specified for scalar arguments, but may take conforming array arguments.

Consider the following example:

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

INTEGER ::length(5)

CHARACTER(len=7) :: c(5)

x=SQRT(num)

a=SQRT(a)

length=LEN( TRIM(c) )
```

The first assignment is between two scalars and assigns the square root of num to x.

The second assignment involves the same elemental intrinsic procedure but with an array argument. 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 command ignores the trailing blank.

4.7 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(arr==1))=0

a(1:count(arr==1))=0
```

The first statement initialises a to 1 2 1 1 3 values.

The second statement arr(1:count(arr==1))=0 will change 1,2,1,1,3 to 0,0,0,1,3 since the original array had 3 elements with the value of 1.

The third statement arr(1:count(arr==4))=0 will do nothing because it is a zero-sized array (lower bound is 1, higher bound is 0 since there are no elements with the value of 4). Allowing for zero-sized arrays means that if the original array is empty or contains no elements with the required value the statement becomes a do nothing statement.

4.8 Initialising arrays

4.8.1 Constructors

This is to be used for 1-dimensional arrays that need to be assigned with various values. 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

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

The constructor can be used during declaration as shown above or in a separate statement but only the latter form can be employed to initialise an array with constant values.

4.8.2 Reshape

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.

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(2,3) :: a

a=RESHAPE((/i,i=0,5/),(/3,2/))
```

The last statement will generate a rank 2 array with extents 3 and 2.

4.8.3 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 which is repeated 4 times. Do not confuse this with the multiplication operator.

The third and fourth statements use a list by section where the first row takes 0 0 and the second row takes the values of 1 1.

The last two statements use a list by implied DO loops 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.
- it can not be used for initialisation of arrays with constant values.
- may be used for other variables as well as arrays.

4.9 WHERE

To be used when the value of an element depends on the outcome of some condition. It takes a statement form or a construction form

The WHERE statement allows a single array assignment only if a logical condition is true. The syntax is as follows:

```
WHERE (condition) statement
```

Consider the following situation:

```
INTEGER :: a(2,3,4)

WHERE(a< 0) a=0

WHERE(a*3>10) a=999
```

The first WHERE statement means that all negative values of‘a are set to zero, the non-negative values of a remain intact.

The second WHERE statement means that elements of a are set to 999 if the product is greater than ten.

The WHERE construct allows array assignment(s) only if a logical condition is true, and alternative array assignement(s) if false. The syntax is as follows:

```
WHERE (condition)

block1

[ELSEWHERE

block2]

ENDWHERE
```

Examine the following section of a program.

```
INTEGER :: b(8,8)

WHERE (b<=0)

b=0

ELSEWHERE

b=1/b

ENDWHERE
```

So all negative valued elements of 'b' are set to zero and the rest take their reciprocal value.

4.10 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
- array reduction
- array inquiry
- array construction
- array reshape
- array manipulation
- array location

A sample will now be presented.

4.10.1 Example of reduction

```
ALL (condition, [DIM])
```

determines whether all elements along a given dimension (DIM) satisfy the condition. The outcome is either a scalar (if dimension part is missing) or an array (if dimension part is declared) of logical type.

```
LOGICAL :: test, test2(2), test3(3)

REAL, DIMENSION(3,2) :: a

a = (/5,9,6,10,8,12/)

...

test=All(a>5)

test2=All(a>5, DIM=1) !false, true, true

test3=All(a>5, DIM=2) !false, true
```

The first statement gives false since the first element is equal to 5 and not greater.

The second statement gives [false,true,true] since the first element of the first row is equal to 5 and not greater, whereas both elements on the remaining two rows are greater than 5.

The third statement gives [false,true] since first element of the first column is equal to 5 and not greater, whereas all 3 elements on the remaining column are greater than 5.

4.10.2 Example of inquiry

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

returns the extent of an array for the specified dimension (DIM). If the dimension part is missing it returns the total number of elements in the array.

```
REAL, DIMENSION(3,2) :: a

num=Size(a)

num=Size(a,DIM=1)

num=Size(a,DIM=2)
```

The first statement gives 6, the second gives 2, and the last gives 3.

4.10.3 Example of construction

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

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

4.10.4 Example of location

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

determines the location of the first encountered element of the given array which has the maximum value and satisfies the optional mask.

```
REAL :: a(5) a=(/2,8,5,3,4/)

num = MAXLOC( a )

num = MAXLOC( a, MASK=a<5 )

num = MAXLOC( a(2:4) )
```

The first MAXLOC statement returns 2 since this is the position of the highest number on the list.

The second MAXLOC statement returns 5 since this is the position of the highest number on the list when numbers greater than 5 are excluded.

The third MAXLOC statement returns the value 1 since this is the position of the highest valued element in the array section. Note that it is worth remembering that elements in array section statements are renumbered with one as the lower bound in each dimension.

4.11 Exercises

1. (a) A user enters the following elements to a (3x3) array: 1, 2, 5,8, 6, 7, 5, 0 , 0. What is the value of element(2,1); (3,2); (1,2); (2,3).

   (b) An array with rank 7 and extent of 5 in each dimension, how many elements does it have?

   (c) An array with rank 3 and extents of 10, 5 and 3, how many elements does it have?

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?

3. Declare an array for representing a noughts and crosses board (a board of 3x3 squares, indicating an empty square with false, otherwise with true)

4. Given the following declarations:

   ```
   REAL, DIMENSION(-1:5,3,8) :: alpha

   REAL, DIMENSION(-3:3,0:2,-7:0) :: beta
   ```

   Are the two arrays conformable?

5. Given the following array declaration

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

   which of the following references are legal?

   ```
   a(2,3), a(6,2), a(0,3), a(5,6), a(0,0)
   ```

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 beta with the following elements

   ```
   5 6

   4 2

   0 5
   ```

8. Declare and initialise the array gamma with the following element values: 2.1, 6.5, 4.3, 8.9, 12.5

9. Declare and initialise the 2-rank array delta which has the following elements

   ```
   0 0 0 1

   0 0 1 1

   0 1 1 1
   ```

10. Using a vector subscript declare an array zeta with 100 elements and place the value 8 to the 1st, 2nd, 10th, 34th, 99th and 100th element.

11. The following array declarations are given:

    ```
    REAL, DIMENSION(50) :: alpha

    REAL, DIMENSION(60) :: beta
    ```

    which of the following statements are valid?

    ```
    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
    ```

12. 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

13. 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/)
```

14. What are the values of the array delta which has been declared and initialised as follows:

```
REAL, DIMENSION(2,2) ::delta=Reshape((/((10*i+j,i=1,2),j=1,2)/), (/2,2/))
```

15. If the array beta has been declared as

```
INTEGER, DIMENSION(10) :: beta
```

what elements are referenced by each of the following statements?

```
beta(2:8:3)
```

```
beta(1:10)
```

```
beta(3:5)
```

```
beta(:9)
```

```
beta(:)
```

```
beta(::4)
```

```
beta(3:10:0)
```

16. If the array gamma has been declared as

```
REAL, DIMENSION(3,4) : gamma
```

what elements are referenced by each of the following statements?

```
gamma(2,:)
```

```
gamma(:,3)
```

```
gamma(2,3:4)
```

```
gamma(::2,:)
```

17. 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

```
MAXLOC(alpha)
```

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

18. Determine what the following array constructor does and then simplify the constructor: (/((A(i)+10.34,j=1,1000),i=1,1000) /)

19. Write a WHERE statement which only changes the sign of the elements of array alpha that are negative.

20. Write a WHERE statement which replicates every non-zero element of an array beta by its reciprocal and every zero element by 1.

[Next] [Previous] [Top]

---

5 Logical & comparison expressions

5.1 Relational operators

Recall that a logical variables denoted with the keyword LOGICAL, and it can take two logical values(.TRUE. or .FALSE.) which are used to record Boolean information about the variable.

Recall that declaring logical variables is in the following form

```
LOGICAL :: guess, date
```

and assigning a logical variable is in the following form

```
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. In this section the logical and comparison operators are introduced and how to perform comparisons is illustrated.

More Examples:

```
5 < 6 !True

5 > 6 !False

5 == 6 !False

5 /= 6 !True

5 <= 6 !True

5 >= 6 !False

age > 34 !a variable compared with a constant

age /= my_age !two variables are compared

45 == your_age !a variable can appear in any side

name= 'Smith' !characters are allowed

alpha(3) /= 33 !array elements are allowed

(age*3) /= your_age !expressions are allowed
```

5.2 Logical expressions

The .AND. logical operator is used to link expressions which evaluate to TRUE only if all given expressions are true, otherwise evaluates to FALSE. Consider the following example: (salary*0.4) .and. (age<45). There are two sub-expressions here. If both are true the expression evaluates to true, otherwise the value false is assigned.

The .OR. logical operator is used to link expressions which evaluate to TRUE only if any of the expressions is true, otherwise evaluates to FALSE. Consider the following example: (name='Dimitris') .or. (name='James') .or. (name='Jim'). Again if the user enters any of the names the expression is given the true value, otherwise the value false is assigned.

The .NOT. logical operator is used to inverts the logical value of an expression. For example true becomes false and vice versa. The form is: .not. (salary*0.4) where the statement enclosed by brackets is assigned a value which in turn is inverted.

The .EQV. logical operator is used to link expressions which evaluate to TRUE only if all expressions have the same logical value (can be true or false), otherwise evaluates to FALSE. For example: (5*3>12) .EQV. (6*2>8) evaluates to TRUE because both sub-expressions take the true value.

The .NEQV. logical operator is used to link expressions which evaluate to TRUE only if at least one of the expressions has a different logical value than the others, otherwise evaluates to FALSE. For example: (5*3>12) .NEQV. (6*2>13) evaluates to TRUE because the first sub-expression is true whereas the second is false.

Comparing real & integer converts the integer to its real equivalent Comparing real & real must be performed with caution because of rounding errors resulting from arithmetic operations. It is advisable to test their difference rather than their actual values. For instance, (a-b<0.005) is better than (a==b).

5.3 Character Comparisons

Certain rules have to be obeyed when comparing characters or character strings . (Is A greater than B ?) When one of the character strings has a shorter length, it is filled 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 terminates with the first encountered difference. Comparing character strings depends on the collating sequence of the machine used. The collating sequence must obey the following rules.

A < B < ... < Z

a < b < ... < z

0 < 1 < 2 ... < 9

digits before A or after Z; or before a or after z blank before letters or digits Rest of characters have no defined position, machine dependant Note that standard does not define if upper case characters come before or after lower case characters

The earliest 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:

Is 'Alexis' > than 'Alex'?

The right expression 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 answer is yes because the blank comes before letters! (the earlier a character comes in the collating sequence the smaller value it has)

5.4 Portability Issues

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

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

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

LLT(string1, string2) !less than
```

Because the collating sequence might differ from machine to machine one can use one of the above intrinsic functions either 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.

5.5 Exercises

1. Given that

   ```
   INTEGER :: age=34, old=92, young=16
   ```

   what is the value of the following expressions?

   ```
   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 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 .eqv. (4<=3)
   ```

3. Is this true?

   ```
   (a<b .and. x<y) .or. (a>=b .and. x>=y) = (a<b .eqv. x<y)
   ```

   Re-write 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 expression

   ```
   "Adam" > "Eve"

   "ADAM" > "Adam"

   "M1" < "M25"

   "version_1" > "version-2"

   " more" < "more"

   LGT("Adam","adam")

   LLT("Me","me")

   LLT("me","me?")
   ```

---

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 we make decisions based on certain circumstances. For instance after listening to the weather forecast one might take an umbrella. 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 arising circumstances. For instance, to take different actions based on experimental results.

6.1.1 Flow control

Selection and routing control through the appropriate path of the program is a very powerful and useful operation. Fortran90 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.

6.1.2 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 phone for a taxi and take an umbrella’. This time the ‘then’ part is required. Notice that an END IF (or ENDIF) part is required to indicate the end of the selection process. If it is raining the block of actions are executed and control passes to the next statement after END IF, otherwise the block of actions are skipped and control passes to the next statement after END IF.

A more complex situation is when one wants to perform alternative actions depending on the condition. For instance, both previous examples do not tell us what to do when it is not raining. The rules above can now be rephrased as: if it rains then phone taxi and take umbrella else walk.

Notice the use of the else part. The action-block parts may contain a single or more actions. The else part covers every other eventuality: sunshine, snowing etc. The passing of control follows the same rules as mentioned above.

There are situations though that alternative actions have to be taken depending on the value the condition takes. For instance, one might want to perform different action if it rains or snows or the sun is out. For example, if it is raining then phone taxi, take umbrella, else if it is snowing then stay at home, else if the sun is out then go to park, else walk. Notice the use of the ELSEIF part. The ELSE part acts as a default again in order to cover other eventualities. The same rules concerning passing of control apply.

The form can be used in a number of ways. For instance, multiple ELSEIFs can appear and/or the ELSE branch can be omitted and/or more IF constructs might follow ELSEIF or ELSE.

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,0.0 ) THEN

...

ELSE outer

inner: IF( y<0.0 ) THEN

...

ENDIF inner

ENDIF outer
```

6.1.3 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 type 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: any value from a minimum value upwards.
- :max any value from a maximum value downwards.
- min : :max any value between the two limits.

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)

WRITE(*,*) ‘Spring’

CASE(6,7)

WRITE(*,*) ‘Summer’

CASE(8:10)

WRITE(*,*) ‘Autumn’

CASE(11,1:3,12)

WRITE(*,*) ‘Winter’

CASE DEFAULT

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.

6.1.4 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 with the corresponding label. 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.

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 the repetition (or iteration) construct it is possible to restructure the program to repeat the same statement and convert the required number of characters.

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

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

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:

```
all: DO i=1,10

WRITE(6,*) i !write numbers 1 to 10

END DO all

even: DO j=10,2,-2

WRITE(6,*) j !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

END DO [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 the 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.

The CYCLE statement is transferring control back to the beginning of the loop to allow the next iteration of the loop to begin.

Confusion can arise from multiple and nested (i.e. one inside another) DO loops, EXIT and CYCLE statements hence naming loops is highly recommended. As an example consider the following program:

```
PROGRAM averscore

REAL :: mark, average

INTEGER:: stid, loop

mainloop: DO

WRITE(*,*) 'Please give student id'

READ(*,*) stid

IF (stid==0) EXIT mainloop

average=0

innerloop: DO loop=1,3

WRITE(*,*) 'Please enter mark'

READ(*,*) mark

IF (mark==0) CYCLE innerloop
```

```
negs: IF (mark<0) THEN
WRITE(*,*) 'Wrong mark. Start again'
CYCLE mainloop
END IF negs
average=(average+mark)
END DO innerloop
average=(average)/5
WRITE(*,*) 'Average of student',stid,' is = ',average
END DO mainloop
END PROGRAM averscore
```

This program calculates the average mark of student given a series of 3 marks. It terminates when the user enters zero as the student id. In the case of a negative mark being entered the user has to re-enter all marks of that particular student (not only the wrong one!). In case of a zero mark the program asks for the next mark and saves adding a zero to the average total.

Notice that the labelling of DO and IF statements make the program not only easier to read and understand but more importantly able to perform the desired actions. Using EXIT or CYCLE without labels it would had made it difficult to comprehend which loop is referred to. Consider the case when the statement Cycle MainLoop was stripped from its label. The program thinks we refer to the InnerLoop and for every negative number we enter we miss a valid mark. Entering the following marks: 2 2 2 2 -4 results in an average of 1.6. The last value (-4) forces the program to go to the closest DO loop and to increase the counter. The counter then becomes five and the loop exits. So the logic of the program has been altered. If labels are not used then Exit will transfer control to the first statement after the END DO associated with the closest to Exit DO.

Similarly, Cycle will transfer control to the closest DO loop.This is a possible execution of the program:

- Pass-1

  ```
  Please give student id 1 ! This is a valid id
  Please enter mark 40 ! valid
  Please enter mark 53 ! valid
  Please enter mark 65 ! valid
  Average mark of student 1 is = 0000
  ```

  Control returns to mainloop

- Pass-2

  ```
  Please give student id 2
  Please enter mark 45
  Please enter mark 45
  Please enter mark 0 !ignored and control to InnerLoop
  Average mark of student 2 is = 30.00000
  ```

  Control returns to mainloop

- Pass-3

  ```
  Please give student id 3
  Please enter mark 40
  Please enter mark -1 ! Control to MainLoop
  Wrong mark. Start again
  Please give student id 3
  Please enter mark 40
  Please enter mark 85
  Please enter mark 86
  Please enter mark 87
  Please enter mark 88
  Average mark of student 3 is = 52.60000
  ```

  Control returns to mainloop

- Pass-4

  ```
  Please give student id 0
  ```

  Control to 1st statement after END DO mainloop

6.3 Exercises

1. Predict the values loop takes and the value loop has after termination of each of the following DO constructs, predictions may be tested by writing a program which accepts the values used in the loop control clause 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

2. 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.

3. Write a program called ‘papersize’ to calculate and display the size of A0 to A6 papers in 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.

4. 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.

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

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

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

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

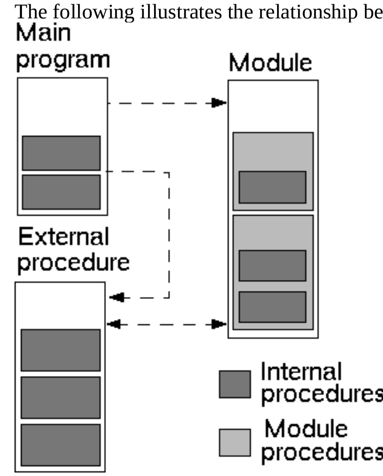
where T is the temperature in centigrade, and t is the time in seconds. Write a program which prints the temperature of such a reaction at 1 minute intervals, The initial temperature is supplied by the user and the above equations should be re-calculated once every second. The program should terminate 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 ! always assign a value to the function id.
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 statement. 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 subroutine 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.

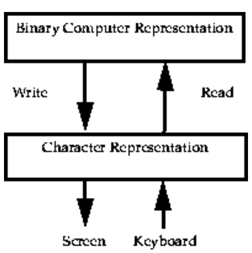
[Next] [Previous] [Top]

---

8 Interactive Input and Output

This module deals with the interaction between a user and the program via the standard input and output devices (keyboard and screen). Data represented by characters, which is a human readable form, are transferred to and from the program. During the transfer process the data are converted to or from the machine readable binary form. In particular the layout or formatting of the data will be considered. A subset of the formatting facilities is presented as the full set is complicated and a number of the features are rarely used.

The process of I/O can be summarised as:



The internal hexadecimal representation of a real number may be

```
BE1D7DBF
```

which corresponds to the real value

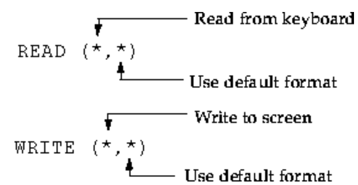
```
0.000450
```

which may be written as

```
-0.45E-03
```

This conversion of the internal representation to a user readable form is known as formatted I/O and choosing the exact form of the character string is referred to as formatting. The formatting of I/O is the underlying theme of much of this module.

Consider the I/O statements used in the previous modules:



8.1 FORMAT Statement

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 and may be used by a WRITE or READ statement within the same program unit by specifying the label number as the second parameter to either or by use of a keyword, for example:

```
READ (*,100) i, j

WRITE (*,100) i, j

READ (*,FMT=200) x, y

WRITE (*,200) x, y

.....

100 FORMAT (2I)

200 FORMAT (2F10.6)
```

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

```
label FORMAT (flist)
```

where flist is a list of edit descriptors which include

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

only the following will be covered

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

Many edit descriptors can be prefixed by a repeat count and suffixed with a fieldwidth. Thus in the two examples given above, 2I and 2F10.6, could be described as two integers and two floating-point real numbers with fieldwidths of 10.6 (a description follows).

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:

```
READ (*,'(2I)') I, J
```

```
WRITE (*,'(2F12.6)') X, Y
```

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. However, format descriptor lists may be used by more than one I/O statement and a labelled format statement reduces the risk of introducing inconsistencies between the multiple instances of the descriptor list.

8.2 Edit Descriptors

Edit descriptors specify exactly how values should be converted into a character string on an output device or internal file, or converted from a character string on an input device or internal file. The edit descriptors are defined in terms of the following key letters

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

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.

8.2.1 Integer

- I, Iw, Iw.m - integer data
- May be repeated i.e. aI, aIw, aIw.m
- 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
- For example: I I6 I10.3 5I 4I6.4
- I6.4 specifies a total of 6 characters including a sign with a minimum of 4 digits thus:

```
WRITE (6,'(I10.6)') 56
```

would ouput two spaces followed by 0056

8.2.2 Real - Fixed Point Form

- Fixed point notation for real numbers
- Possible forms: F, Fw, Fw.d aF, aFw, aFw.d
- If no decimal point is supplied d digits are read as the fractional part.
- For example: F10.5 F12.6 5F14.7
- F12.6 specifies a total of 12 characters including decimal point and where required a minus sign with 6 digits following the decimal point, thus:

```
WRITE(6,'(2F12.6)') 12.6, -131.4567891
```

would output ^^^12.600000 and -131.456789 where ^ represents a space

8.2.3 Real - Exponential Form

- Floating point notation for real data
- Possible forms: E, Ew, Ew.d, aE, aEw, aEw.d
- The total fieldwith w includes the signs and decimal point
- The E descriptor specifies a number in the following form S0.XXXESXX where S signifies a sign, X digits and the character E separates the mantissa from the exponent
- On output the exponent is adjusted to place the most significant digit to the right of the decimal point eg. 0.123E-2
- Two alternative forms 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

8.2.4 Character

- Possible forms: A, aA, Aw, aAw
- Use to read or write single characters or character strings
- If a fieldwith w is greater than the number of characters then the characters are right justified and space filled
- On input the character string does not need to be enclosed in quotes

8.2.5 Skip Character Positions

- aX - skip specified number of characters

- On input characters are ignored
- On output the required number of spaces is written

8.2.6 Logical

- L - logical data
- On input T, F, .TRUE., FALSE. are acceptable
- On output T or F will be written

8.2.7 Other Special Characters

- / specifies take a new line
- : terminate I/O if list exhausted
- ( ) group descriptors, normally for repetition e.g. 4(I5.5,F12.6)
- ‘ Output the character string specified

8.3 Input/Output Lists

Input/Output lists are used to specify the quantities to be read in or written out. On output expressions may be used but variables are only permitted for input. 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 (*,*) I(1), I(2), I(3)

READ (*,*) I

READ (*,*) I(1:3)
```

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

```
INTEGER :: I(10), K(3)

K = (/1,2,1/)

READ (*,*) I(K)
```

would be illegal as I(1) appears twice.

8.3.1 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 TRIANGLE

TYPE (POINT) A, B, C

END TYPE
```

the following two statements are equivalent

```
READ (*,*) P, T

READ (*,*) P%X, P%Y, T%A%X, T%A%Y, T%B%X, T%B%Y, T%C%X, T%C%Y
```

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

Any pointers in an I/O list must be associated with a target, the target is the data on which the I/O statement operates, for example:

```
REAL, POINTER :: PTRA, PTRB

REAL, TARGET :: X

X = 10.0

PTRA => X

WRITE (*,*) PTRA
```

would output the value 10.0, whereas

```
WRITE (*,*) PTRB
```

would generate an error as PTRB is not associated with a target.

8.3.2 Implied DO Loop

The Implied-DO-list, which is often used when performing I/O on an array, has the general form:

```
(do-object-list, do-var=expr,expr[,expr])
```

This syntax is similar to the simple indexed DO loop described earlier. Consider the following examples:

```
INTEGER :: J

REAL, DIMENSION(10) :: A

READ (*,*) (A(J),J=1,10)
```

```
WRITE (*,*) (A(J), J=10,1,-1)
```

The first statement would read 10 values in to each element of I. The second statement would write all 10 values of I 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)
```

Note:

- No do-var may be a do-var of any implied-do-list in which it is contained, nor be associated with such a do-var (eg. pointer association).
- In an input implied-do-list a variable which is an item in a do-object-list may not be a do-var of any implied-do-list in which it is contained.

8.4 Namelist

This is a facility for grouping variables for I/O. The rules governing the use of NAMELIST are fairly complex so, for the scope of this course, only explicitly declared variables will be used, pointers and allocatable arrays will not be covered. The use of NAMELIST for output only will be considered as this can be useful for program testing and debugging. It's use on input is slightly more complicated and is best considered only where necessary.

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

```
NAMELIST namelist-spec
```

where namelist-spec is

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

for example

```
NAMELIST /WEEK/ MON, TUES, WED,THURS, FRI
```

The list may extended within the same scoping unit by repeating the namelist-group-name on more than one statement, as follows:

```
NAMELIST /WEEK/ SAT, SUN
```

More than one group may be defined in one NAMELIST statement but this feature should not be used. Variables should be declared before appearing in a NAMELIST group. Variables with the PRIVATE and PUBLIC attributes should not appear in the same namelist-group. The namelist-group may be used in place of the format specifier in an I/O statement. Only the WRITE statement is considered here.

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

will produce

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

where

```
INTEGER :: SUN, MON, TUES, .....
SUN = 1
MON = 2
.....
```

Note the output record is an annotated list of the form:

```
& namelist-group-name namelist-variable=value {,namelist-variable=value} /
```

This record format must be used for input.

Arrays may also be specified, for example

```
INTEGER, DIMENSION(10) :: ITEMS
NAMELIST /GROUP/ ITEMS
ITEMS(1) = 1
WRITE (*, NML=GROUP)
```

would produce

```
& GROUP ITEMS(1)=1, ITEMS(2:10)=0 /
```

8.5 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:

```
READ(*,*,ADVANCE='YES') ...
WRITE (*,*, ADVANCE='NO') ....
```

There are two optional keywords, EOR and SIZE, which have the form:

```
EOR=eor-label
```

```
SIZE=integer-variable
```

The EOR keyword specifies a labelled statement to which control is passed if an error occurs (see ERR keyword later) or if the end of record is encountered. The SIZE keyword specifies an integer variable which is set to the number of characters read.

By default unfilled characters in an input record are padded out with blank characters but these characters are not included in the value assigned to SIZE. The PAD keyword to the OPEN statement (see later) may be used to override the default action.

Examples.

(i)

```
WRITE(*,*,ADVANCE='NO') 'Enter new value: '

READ(*,*) I
```

If the user enters the value 10 this would appear on the screen as

```
Enter new value: 10
```

(ii)

```
CHARACTER(LEN=32) :: filename

INTEGER :: length

bb: DO

WRITE (*,*, ADVANCE='NO') 'Filename? '

READ(*,*,ADVANCE='NO',EOR=20, SIZE=length) filename

OPEN(10, FILE=filename(1:length),..)

EXIT bb

20 WRITE(*,'(/A,I)') 'Error: maximum length exceeded.',length

END DO
```

8.6 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
```

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 the following input records

were used?

(a) bbb5.34bbbNOb.true.

(b) 5.34bbbbbbYbbFbbbbb

(b) b6bbbbbb3211bbbbbbT

(d) bbbbbbbbbbbbbbbbbbF

where b represents a space or blank character.

3. Write statements to output all 100 elements of a one dimensional array of real numbers with 10 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.

   Write a program which will initialise all the elements of a suitably declared array to 123.456789 and will test the output statements you have just written. What would happen if the initial value was -123456789.789?

4. A file contains records in two groups with each group headed by a specification of the format in which the following data is to be read. Each group of data is terminated by the special value 99.9. Write a program to read in all the data. The general form of the input is known i.e.. he first group will contain 3 reals and an integer and the second group one real and an integer. Example data:

```
'(3F6.4,I5)'

3.40 5.6 7.9 10
```

```
4.52 6.3 3.2 11

99.9 0 0 0

'(F6.1,I10)'

4.23 9

5.89 6

99.9 0
```

[Next] [Previous] [Top]

9 File-based Input and Output

In the previous modules all input and output was performed to the default devices namely the screen and the keyboard. 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.

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 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, NUNIT+I
- 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: UNIT=10 or UNIT=NUNIT. The unit number may also be specified as a positional argument as shown later.

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 a Unix 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 on most systems.

9.2 READ and WRITE Statements

9.2.1 READ Statement

There are two forms of the READ statement, which correspond to the PRINT and WRITE output statements covered later.

```
READ format-spec,I/O list !This form is not used in this course
```

or

```
READ (clist) [I/O 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]
```

For example:

```
READ *,I,J

READ *,LINE

READ 100, I

READ (*,*) A,B,C

READ (5,*) LINE

READ (5,100) X, Y, Z

READ (UNIT=10,FMT=100,ERR=10,IOSTAT=ios)
```

The unit number and format-specifier must be supplied and in the correct order but the other items are optional. In the last example, if an error occurrs, control passes to the statement labelled 10 and the variable specified as ios will return a positive, system dependent integer. The value 0 will be returned if the operation completes successfully.

9.2.2 WRITE Statement

There are two output statements: the PRINT and the WRITE statement. Only the WRITE statement is covered in this course as the PRINT statement is simply a limited form of the WRITE statement. The WRITE statement may be list-directed or format-directed and has the general form:

```
WRITE (clist) [I/O 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]
```

For example:

```
WRITE (*,*)

WRITE (6,*) I,J

WRITE (6,100) I

WRITE (6,*,ERR=10) LINE

WRITE (UNIT=file1,FMT=100,REC=recordnumber, ERR=10) newline
```

9.3 OPEN Statement

The OPEN statement is used to connect a unit number to a file specifying 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 (u, [olist] )
```

where

```
u is a valid unit number specifier (with or without the keyword)

olist is a list of keyword clauses:

keyword "=" value {"," keyword "=" value}
```

For example:

```
OPEN(10)

OPEN (UNIT=10)

OPEN (UNIT=IFILE)
```

The following keywords are specified in the Fortran 90 language standard:

`FILE=filename`

where filename is a valid filename for the particular system. Note that case sensitivity is system specific. e.g. FILE='output.test'

`STATUS=st`

where st may be 'OLD', 'NEW', 'REPLACE', 'SCRATCH' or 'UNKNOWN'. If 'OLD' is specified the file must exist; if 'NEW' the file must not exist; if 'REPLACE' and the file exists it will be deleted before a new file is created; and if 'SCRATCH' the file will be deleted when closed. In general use 'OLD' for input and 'NEW' for output.

`ERR=label`

GOTO label if an error occurs opening the file.

`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.

`FORM=fm`

where fm may be 'FORMATTED' or 'UNFORMATTED', the default is 'FORMATTED' for sequential files and 'UNFORMATTED' for direct access files.

`ACCESS=acc`

where acc may be 'SEQUENTIAL' or 'DIRECT'

`RECL=rl`

where rl is the maximum record length (positive integer) for a direct access file. For formatted files this is the number of characters and for unformatted it is usually the number of bytes or words (system dependent).

`BLANK=bl`

where bl is either 'NULL' or 'ZERO' and determines how blanks in a numeric field are interpreted.

`POSITION=pos`

where pos may be 'ASIS', 'REWIND' or 'APPEND' which are interpreted as positioning the file at the position it was previously accessed, positioning the file at the start; and positioning the file after the previously end of the file. Defaults to ASIS.

`PAD=pad`

where pad may be 'YES' or 'NO'. If 'YES' formatted input is padded out with blank characters; if 'NO' the length of the input record should not be exceeded.

`DELIM=del`

where del may be 'APOSTROPHE' or 'QUOTE' or 'NONE' indicating which character used when delimiting character expressions in list-directed or NAMELIST output. Defaults to 'NONE'.

`ACTION=act`

where act may be 'READ', 'WRITE' or 'READWRITE' specifying the permitted modes of operation on the file. Default is processor dependent.

For example:

```
OPEN (UNIT=10,FILE='fibonacci.out')

OPEN (UNIT=11,FILE='fibonacci.out',STATUS='NEW',ERR=10)

.......

10 CONTINUE

WRITE(6,*) 'Error opening file: fibonacci.out.'

OPEN (UNIT=12, FILE='student.records', STATUS='OLD', &

ACCESS='DIRECT',RECL=200, FORM='FORMATTED',&

ERR=20, IOSTAT=IOS)

........

20 CONTINUE

IF (ERR .GE. 0) THEN

WRITE (6,*) &
```

```
'Error opening file: student.records.'

WRITE (6,*) 'IOS = ',IOS

ENDIF

STOP
```

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 statements. Specifying some values alter the default values of others and some specifies are mutually exclusive, i.e. only one or the other but not both, for example RECL and ACCESS='SEQUENTIAL' may not be used together.

9.4 CLOSE statement

This statement permits the orderly disconnection of a file from a unit 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. The CLOSE statement has the general form

```
CLOSE ([UNIT=]u [,IOSTAT=ios] [,ERR=label] [,STATUS=st])
```

where st can be 'KEEP' or 'DELETE'. The value 'KEEP' cannot be applied to a file opened as 'SCRATCH'.

For example:

```
CLOSE (10)

CLOSE (UNIT=10, ERR=10)

CLOSE (UNIT=NUNIT, STATUS='DELETE',ERR=10)
```

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 combination of the (possible return values are given as a comment)

```
EXIST=lex ! true or false

OPENED=lod ! true or false

NUMBER=unum ! unit number

NAME=fnm ! filename

ACCESS=acc ! 'DIRECT' or 'SEQUENTIAL'

SEQUENTIAL=seq ! 'YES or 'NO'

DIRECT=dir ! 'YES' or 'NO'

FORMATTED=fmt ! 'YES' or 'NO'

UNFORMATTED=unfmt ! 'YES' or 'NO'

FORM=frm ! 'FORMATTED' or'UNFORMATTED'

NEXTREC=recn ! number of next record for direct access files only

RECL=recl ! record length for direct access files only
```

Note that recl, recn and unum and integer variables.

9.6 Direct Access Files

A direct access file is a random access table-like structure which may be inspected or amended. Such a file may be created and accessed using the RECL and ACCESS='DIRECT' keywords on the OPEN statement and REC on the relevant READ and WRITE statements as follows:

```
CHARACTER (LEN=200) :: STAFFRECORD

OPEN (UNIT=10, FILE='STAFF.RECORDS', &

RECL=200, ACCESS='DIRECT', ......)

READ(UNIT=10,REC=20) STAFFRECORD

WRITE(UNIT=10,REC=20) STAFFRECORD
```

The ERR and IOSTAT keywords should be used to handle possible error conditions such as reading beyond the end of file. One limitation on a direct access file is that all the records must be of fixed length. On some computer systems a direct access file may not be created by a program but must be created using system commands prior to program execution. Also on some systems such a file may not be extended by the program but must also have a known fixed number of records.

9.7 Exercises

1. Complete the following statement, which would open an unformatted direct access file with a record length of 100 bytes

   ```
   OPEN(UNIT=10,......)
   ```

2. Write a section of code which would open 10 files on the unit numbers from 20 to 29. The default values should be used for all keywords.

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 direct access file and if so what the record length is.

   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.

5. Write a Fortran program which will create a temporary direct access file, prompt for the name of an existing file and read that file sequentially writing each line to the next record of the temporary direct access file. The program should then repeatedly prompt the user for a number representing the number of a line in the input file and display that line on the screen. The program should halt when the number 0 is entered. The program should handle all possible error conditions such as the file does not exist or a line number out of range is specified and inform the user accordingly.

---

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.

Maintained by Alan Rea, email A.Rea@qub.ac.uk
Generated with CERN WebMaker

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
```

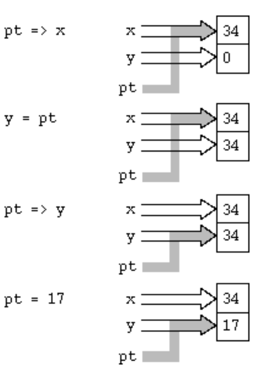
Where pointer is a pointer variable and target is any valid target. pointer may now be used as an alias to the data stored by target. The pointer assignment operator also allocates storage required by the pointer.

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
```

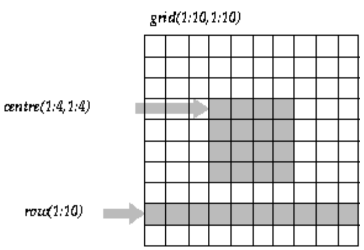
In the above example p2 points to the storage allocated to p1, however when that storage is deallocated p2 no longer has a valid target and its state becomes undefined. In this case dereferencing p2 would produce unpredictable results.

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,:)
```

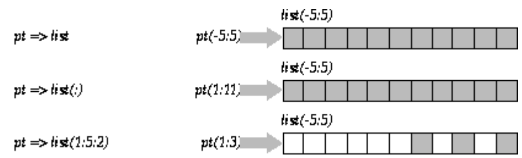


An array pointer can be associated with the whole array or just a section. The size and extent of an array pointer may change as required, just as with allocatable arrays. For example:

```
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:

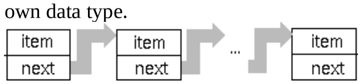
```
DO i=1,3

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.

Maintained by Alan Rea, email A.Rea@qub.ac.uk
Generated with CERN WebMaker

Appendix A: 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).

A.1 Argument presence enquiry

PRESENT(A) - true if A is present.

A.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.

A.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.

A.4 Character functions

ACHAR(I) - returns the Ith 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 Ith 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.

A.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.

A.6 Logical functions

LOGICAL(L [, KIND]) - convert between different logical kinds.

A.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.

A.8 Bit enquiry functions

BIT_SIZE(I) - returns the number of bits in the model.

A.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.

A.10 Transfer functions

TRANSFER(SOURCE, MOLD [, SIZE]) - converts SOURCE to the type of MOLD.

A.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.

A.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.

A.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.

A.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.

A.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.

A.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).

A.17 Pointer association status enquiry functions

ASSOCIATED(POINTER [, TARGET]) - returns .TRUE. if POINTER is associated.

A.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.

Appendix B: 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.

All documents are the responsibility of, and copyright, © their authors and do not represent the views of The Parallel Computer Centre, nor of The Queen's University of Belfast.
Maintained by Alan Rea, email A.Rea@qub.ac.uk
Generated with CERN WebMaker