Introduction to Computer Programming using Fortran 95

Workbook

Edition 2
Spetember 2008

Introduction to Computer Programming using Fortran 95

Edition 2, September 2008

Document Number: 3570-2008

Acknowledgement

DR. A C MARSHALL from the University of Liverpool (funded by JISC/NTI) first presented this material. He acknowledged Steve Morgan and Lawrie Schonfelder.

Helen Talbot and Neil Hamilton-Smith took the overheads from that course and worked on them to produce this text: later Neil Hamilton-Smith revised it.

Contents

1. Fundamentals of Computer Programming

Telling a Computer What To Do

To get a computer to perform a specific task it must be given a sequence of unambiguous instructions or a program.
An everyday example is instructions on how to assemble a bedside cabinet. The instructions must be followed precisely and in the correct order:

- insert the spigot into hole `A';
- apply glue along the edge of side panel;
- press together side and top panels;
- attach toggle pin `B' to grommet `C';
- ... and so on.

The cabinet would turn out wonky if the instructions were not followed to the letter!

Programming Languages

Programming languages must be:

- totally unambiguous (unlike natural languages, for example, English);
- simple to use.

All programming languages have a very precise syntax (or grammar). This ensures that all syntactically correct programs have a single meaning.

High-level programming languages include Fortran 90, Fortran 95, C and Java. On the other hand assembler code is a Low-Level Language. Generally:

- a program is a series of instructions to the CPU of the computer;
- all programs could be written in assembler code but this is a slow, complex and error-prone process;
- high-level languages are more expressive, more secure and quicker to use;
- a high-level program is compiled (translated) into assembler code by a compiler.

Fortran Evolution

Fortran stands for FORmula TRANslation. The first compiler appeared in 1957 and the first official standard in 1972 which was given the name of `Fortran 66'. This was updated in 1980 to Fortran 77, updated in 1991 to Fortran 90, updated in 1997 to Fortran 95, and further updated in 2004 to Fortran 2003. At each update some obsolescent features were removed, some mistakes corrected and a limited number of new facilities were added. Fortran is now an ISO/IEC and ANSI standard.

Character Set

The following are valid in a Fortran 95 program:

- alphanumeric: a-z, A-Z, 0-9, and _ (the underscore);
- symbolic:

Symbol	Description	Symbol	Description
	blank	=	equals sign
+	plus sign	-	minus sign
*	asterisk	/	slash
(	left parenthesis	)	right parenthesis
,	comma	.	decimal point
'	apostophe	"	quotation mark
:	colon	;	semicolon
!	exclamation mark	&	ampersand
<	less than	>	greater than
%	percent	$	currency symbol
?	question mark		

How Does Computer Memory Work?

- Each memory location will contain some sort of value;
- The value stored in a location can be read, or the location can be written to;
- Fortran 95 allows (English) names to be given to memory locations.

Numeric Storage

In general, there are two types of numbers used in Fortran 95 programs, INTEGERs (whole numbers) and REALs (floating point numbers).

- INTEGERs are stored exactly, often in the range [-32768, 32767].
- REALs are stored approximately.
  Their form is a mantissa and an exponent. For example 6.6356×10^{23}
  The exponent can take only a finite range of values, typically [-307, 308].

You can get numeric exceptions:

overflow -- exponent is too big,
underflow -- exponent is too small.

In Fortran 95 you can decide what numeric range is to be supported.

CHARACTERs are stored differently.

Intrinsic Types

Fortran 95 has two broad classes of object type:

- numeric;
- non-numeric

which give rise to six simple intrinsic types, known as default types. These are demonstrated by the following code:

```
INTEGER           :: age             ! whole number
REAL              :: height          ! decimal number
COMPLEX           :: val             ! x + iy
CHARACTER         :: sex             ! letter
CHARACTER(LEN=12) :: name            ! string
LOGICAL           :: wed             ! truth value
```

Literal Constants

A literal constant is an entity with a fixed value. For example:

```
0             12345              ! INTEGER
-1.0          6.6E-06            ! REAL
(1.0,3.14)    (2.7,1.4)          ! COMPLEX
"Isn't"       'Isn''t'           ! CHARACTER
.TRUE.        .FALSE.            ! LOGICAL
```

Note:

- REALs contain a decimal point, INTEGERs do not;
- REALs can have an exponential form;
- there is only a finite range of values that numeric literals can take;
- character literals are delimited by " or ';
- two occurrences of the delimiter inside a string produce one occurrence on output;
- there are only two LOGICAL values.

Names

In Fortran 95 names for variables and procedures etc.:

- must be unique within the program;
- must start with a letter;
- may use only letters, digits and the underscore;
- may use the underscore to separate words in long names;
- may not be longer than 31 characters.

```
REAL              :: a1            ! valid name
REAL              :: 1a            ! not valid name
CHARACTER         :: atoz          ! valid name
CHARACTER         :: a_z           ! valid name
CHARACTER         :: a-z           ! not valid name
CHARACTER(LEN=8)  :: user_name     ! valid name
CHARACTER(LEN=8)  :: username      ! different name
```

Significance of Blanks

In free form source code blanks must not appear:

- within keywords;
- within names.

```
INTEGER   :: wizzy               ! is a valid keyword
INT EGER  :: wizzy               ! is not
REAL      :: running_total       ! is a valid name
REAL      :: running total       ! is not
```

Blanks must appear:

- between two separate keywords;
- between keywords and names not otherwise separated by punctuation or other special characters.

```
INTEGER FUNCTION fit(i)          ! is valid
INTEGERFUNCTION fit(i)           ! is not
INTEGER FUNCTIONfit(i)           ! is not
```

Blanks are optional between some keywords mainly `END < construct >' and a few others; if in doubt add a blank (it looks better too).

Implicit Typing

Any undeclared variable has an implicit type:

- if the first letter of its name is `I`, `J`, `K`, `L`, `M` or `N` then the type is `INTEGER`;
- if it is any other letter then the type is `REAL`.

Implicit typing is potentially very dangerous and should always be turned off by adding:

```
IMPLICIT NONE
```

at the start of the declaration of variables. Consider:

```
DOI = 1.1000
...
END DO
```

With implicit typing this declares a `REAL` variable `DOI` and sets it to `1.1000` (and leaves an unattached `END DO`) instead of performing a loop 1000 times!

Numeric and Logical Type Declarations

With `IMPLICIT NONE` variables must be declared. A simplified syntax follows:

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

Optional components are shown in `[square brackets]`
The following are all valid declarations:

```
INTEGER                  :: i, j
REAL                     :: x
REAL, DIMENSION(10,10)   :: y, z
INTEGER                  :: k = 4
LOGICAL                  :: flag
```

The `DIMENSION` attribute declares an array of 10 rows by 10 columns.

Character Declarations

Character variables are declared in a similar way to numeric types. `CHARACTER` variables can:

- refer to one character;
- refer to a string of characters which is achieved by adding a length specifier to the object declaration.

The following are all valid declarations:

```
CHARACTER                               :: sex
CHARACTER(LEN=10)                       :: name
CHARACTER(LEN=32)                       :: str
CHARACTER(LEN=10), DIMENSION(10,10)  :: Harray
```

Initialisation

Declaring a variable does not automatically assign a value, say zero, to this variable: until a value has been assigned to it a variable is known as an unassigned variable. Variables can be given initial values, which can use initialisation expressions and literals. Consider these examples:

```
INTEGER            :: i = 5, j = 100
REAL               :: x, y = 1.0E5
CHARACTER(LEN=5)   :: light = 'Amber'
CHARACTER(LEN=9)   :: gumboot = 'Wellie'
LOGICAL            :: on = .TRUE., off = .FALSE.
```

`gumboot` will be padded, to the right, with blanks. In general, intrinsic functions cannot be used in initialisation expressions. The following can be: `RESHAPE`, `SELECTED_INT_KIND`, `SELECTED_REAL_KIND`, `KIND`.

Constants (Parameters)

Symbolic constants, known as parameters in Fortran, can easily be set up in a declaration statement containing the `PARAMETER` attribute:

```
REAL, PARAMETER       :: pi = 3.141592
REAL, PARAMETER       :: radius = 3.5
REAL                  :: circum = 2.0 * pi * radius
CHARACTER(LEN=*), PARAMETER :: &
   son = 'bart', dad = "Homer"
```

`CHARACTER` constants can assume their length from the associated literal `(LEN=*)` only if the attribute `PARAMETER` is present.
Parameters should be used:

- if it is known that a variable will only take one value;
- for legibility where a value such as π occurs in a program;
- for maintainability when a constant value could feasibly be changed in the future.

Comments

It is good practice to include many comments, for example:

```
PROGRAM Saddo
!
! Program to evaluate marriage potential
!
LOGICAL   :: TrainSpotter     ! Do we spot trains?
LOGICAL   :: SmellySocks      ! Have we smelly socks?
INTEGER   :: i, j             ! Loop variables
```

- everything after each ! is a comment;
- the ! in a character context does not begin a comment, for example:

```
prospects = "No chance of ever marrying!!!"
```

Expressions

Each of the three broad type classes has its own set of intrinsic (in-built) operators, for example, `+`, `//` and `.AND.`. The following are all valid expressions:

```
NumBabiesBorn + 1   ! numeric valued: addition
"Ward "//Ward       ! character valued: concatenation
NewRIE .AND. Bus38  ! logical: intersection
```

Expressions can be used in many contexts and can be of any intrinsic type.

Assignment

Assignment is defined between all expressions of the same type.

Examples:

```
a = b - c
c = SIN(.7)*12.7         ! SIN argument in radians
name = initials//surname
```

The LHS is an object and the RHS is an expression.

Intrinsic Numeric Operations

The following operators are valid for numeric expressions:

** exponentiation is a dyadic operator, for example, 10**2, (evaluated right to left);
* and / multiply (there is no implied multiplication) and divide are dyadic operators, for example, 10*7/4;
+ and - plus and minus or add and subtract are monadic and dyadic operators, for example, -3 and 10+7-4;
They can be applied to literals, constants, scalar and array objects. The only restriction is that the RHS of ** must be scalar. As an example consider:

```
a = b - c
f = -3*6/2
```

Relational Operators

The following relational operators deliver a LOGICAL result when combined with numeric operands:

`.GT.`	`>`	greater than
`.GE.`	`>=`	greater than or equal to
`.LE.`	`<=`	less than or equal to
`.LT.`	`<`	less than
`.NE.`	`/=`	not equal to
`.EQ.`	`==`	equal to

For example:

```
bool = i > j
```

If either or both expressions being compared are complex then only the operators `==` and `/=` are available.

Intrinsic Logical Operations

A `LOGICAL` expression returns a `.TRUE.` or `.FALSE.` result. The following are valid with `LOGICAL` operands:

```
.NOT.          --  .TRUE. if operand is .FALSE.;
.AND.          --  .TRUE. if both operands are .TRUE.;
.OR.           --  .TRUE. if at least one operand is .TRUE.;
.EQV.          --  .TRUE. if both operands are the same;
.NEQV.         --  .TRUE. if both operands are different.
```

For example, if `T` is `.TRUE.` and `F` is `.FALSE.`

```
  .NOT. T   is .FALSE.          .NOT. F   is .TRUE.
T .AND. F   is .FALSE.        T .AND. T   is .TRUE.
T  .OR. F   is .TRUE.         F  .OR. F   is .FALSE.
T .EQV. F   is .FALSE.        F .EQV. F   is .TRUE.
T .NEQV. F  is .TRUE.         F .NEQV. F  is .FALSE.
```

Intrinsic Character Operations

Consider:

```
CHARACTER(LEN=*), PARAMETER    :: str1 = "abcdef"
CHARACTER(LEN=*), PARAMETER    :: str2 = "xyz"
CHARACTER(LEN=9)               :: str3, str4
```

Substrings can be taken. As an example consider:

```
str1        is "abcdef"
str1(1:1)   is "a"    (not str1(1) which is illegal)
str1(2:4)   is "bcd"
```

The concatenation operator, //, is used to join two strings or substrings:

```
str3 = str1//str2
str4 = str1(4:5)//str2(1:2)
```

would produce

```
abcdefxyz        stored in str3
dexy             stored in str4
```

Operator Precedence

Operator	Precedence	Example
user-defined monadic	Highest	`.INVERSE. A`
`**`	.	`10 ** 4`
`* or /`	.	`89 * 55`
`monadic + or -`	.	`- 4`
`dyadic + or -`	.	`5 + 4`
`//`	.	`str1 // str2`
`>, <=, etc`	.	`A > B`
`.NOT.`	.	`.NOT. Bool`
`.AND.`	.	`A .AND. B`
`.OR.`	.	`A .OR. B`
`.EQV. or .NEQV.`	.	`A .EQV. B`
user - defined dyadic	Lowest	`X .DOT. Y`

Note:

- in an expression with no parentheses, the highest precedence operator is combined with its operands first;
- in contexts of equal precedence left to right evaluation is performed except for **.

Consider an example of precedence, using the following expression:

```
x = a+b/5.0-c**d+1*e
```

Because ** is highest precedence, / and * are next highest, this is equivalent to:

```
x = a+(b/5.0)-(c**d)+(1*e)
```

The remaining operators' precedences are equal, so we evaluate from left to right.

Mixed Type Numeric Expressions

In the CPU, calculations must be performed between objects of the same type. So if an expression mixes type some objects must change type. The default types have an implied ordering:

1. COMPLEX -- highest
2. REAL
3. INTEGER -- lowest

The result of an expression is always of the higher type, for example:

```
INTEGER * REAL                    gives REAL , (3*2.0 is 6.0)
REAL * INTEGER                    gives REAL , (3.0*2 is 6.0)
COMPLEX * < anytype >             gives COMPLEX
```

The actual operator is unimportant.

Mixed Type Assignment

Problems can occur with mixed-type arithmetic. The rules for type conversion are given below:

- `INTEGER` = `REAL`
  The RHS is evaluated, truncated (all the decimal places removed) then assigned to the LHS.
- `REAL` = `INTEGER`
- The RHS is evaluated, promoted to be `REAL` (approximately) and then assigned to the LHS.

For example:

```
REAL :: a = 1.1, b = 0.1
INTEGER :: i, j, k
i = 3.9              ! i will be 3
j = -0.9             ! j will be 0
k = a - b            ! k will be 1
```

Note: although `a` and `b` are stored approximately, the value of `k` is always `1`.

Integer Division

Division of two integers produces an integer result by truncation (towards zero). Consider:

```
REAL :: a, b, c, d, e
a = 1999/1000              ! LHS a is (about) 1.000
b = -1999/1000             ! LHS b is (about) -1.000
c = (1999+1)/1000          ! LHS c is (about) 2.000
d = 1999.0/1000            ! LHS d is (about) 1.999
e =1999/1000.0             ! LHS e is (about) 1.999
```

Great care must be taken when using mixed type arithmetic.

Formatting input and output

The coding used internally by the computer to store values is of no concern to us: a means of converting these coded values into characters which can be read on a screen or typed in from a keyboard is provided by formatting. A format specification is a list of one or more edit descriptors enclosed in round brackets. Each edit descriptor gives the type of data expected (integer, real, character or logical) and the field width (counted in number of characters, non-blank or otherwise) of this data value and how the data item is represented within its field. Edit descriptors can be:

Edit Descriptor	**Value type**	**Format-spec. example**	**Value example**
`wX`	Space	`2X`	
`Iw`	Integer	`I5`	`1` or `-5600`
`Fw.d`	Floating point	`F7.2`	`1.00` or `-273.18`
`Ew.d`	Exponential	`E9.2`	`0.10E+01` or `-0.27E+03`
`Lw`	Logical	`L1`	`T`
`An`	Alphanumeric	`A11`	`'one billion'`
`Gw.d`	General	`G11.3`	`3.14`

The field width is given by a number which immediately follows the letter, unless the letter is `X` in which case the number precedes the letter.

A blank space is simplest of the edit descriptors to specify, consisting of the letter `X`. For example, `X` means ignore the next character position in the current input line, or leave a gap 1 character wide in the current output line. Multiple spaces are indicated by preceding the `X` by an integer count value, so `2X` means skip two positions in the input line or leave two spaces in the output line.

The edit descriptor for characters is almost as simple, consisting of the letter `A` followed by an unsigned integer, for example `A9`. In this case, if the character value were `'positions'` there would be no trouble as the length of the character string equals the width specified. If the value were `'characters'` then only the first 9 symbols would be read in or written out, *ie* `'character'`. If instead the value were `'places'` then the behaviour at input and output is significantly different. On input the 6 symbols would be read in and would be followed by 3 blanks: on output the 3 blanks would precede the 6 symbols.

The edit descriptors for numeric items have to allow for the number to be signed. If a number is being output and the field width given is too small then this field is filled with asterisks.

For integer values, the edit descriptor has the form `I` followed by an unsigned integer. On output, the integer is adjusted to the right-hand side of its field.

For real values there are two possible forms of edit descriptors.
One form is `Fw.d` where `w` is the field width and `d` is the number of digits appearing after the decimal point. The decimal point counts as one position in the field. If there

is a decimal point in a number being read in, then only the `w` and not both `w` and `d` takes effect.

The other form is `Ew.d` where `w` and `d` are similar to those for the `F` edit descriptor. For input the two forms are identical. For output, the value is scaled so that its absolute value is less than 1 and this value will be followed by an exponent in a field of width 4. After allowing for a sign, the decimal point and the exponent, there can be at most `w - 6` digits in the number which is written out.

Complex numbers need edit descriptors for a pair of real numbers: these descriptors need not be identical.

Logical values use an edit descriptor of the form `Lw`. Only if `w` is at least 7 can the values appear as `.true.` or `.false.` – otherwise they would be output as `T` or `F` in the right-most position of the field.

Any of the edit descriptors in a format specification may be preceded by an integer which is the repeat count for that descriptor. For example:

`'(I5,I5,F9.4,F9.4,F9.4)'` can be rewritten as `'(2I5,3F9.4)'`

If there are repeated sequences of edit descriptors then a repeat count can be applied to a single sequence. For example:

`'(2X,A5,F4.1,2X,A5,F4.1)'` can be rewritten as `'(2(2X,A5,F4.1))'`

If a format specification (without components in parentheses) is used with an input or output list that contains more elements than the total number of edit descriptors, applying any repeat counts, then a new record will be taken and the format specification will be repeated. On input new records will be read until the list is satisfied: this means that for any record which contains more data than is specified by the format specification the surplus data are ignored.

WRITE Statement

A simple form of the `WRITE` statement which allows you to output to the default output device using a default format, is:

```
Write(*,*)<list>
```

This form is handy for diagnostic output when testing a program.
A general form of the `WRITE` statement which allows you to output to any device using a specified format, is of the form:

```
Write(unit=u,fmt=<format_specification>)<list>
```

The unit number allows you to write to any device such as a file or the screen (6 specifies the screen). The format specification is a character string, starting with `(` and ending with `)`, defining how your data is to be laid out. `<list>` is a comma separated list of items. Consider this example code:

```
PROGRAM Owt
     IMPLICIT NONE
     CHARACTER(LEN=31) :: &
          format_spec="(a4,f4.1,2(2x,a5,f4.1))"
     CHARACTER(LEN=26) :: &
          long_name = "Llanfair...gogogoch"
     REAL :: x=1., y=2., z=3., tol=0.001
     LOGICAL :: lacigol
     lacigol = (abs(y - x) < tol)
     WRITE(unit=6,fmt="(a19)")  long_name
     WRITE(unit=6,fmt="(a30)")  "Spock says ""illogical &
          &Captain"" "
     WRITE(unit=6,fmt=format_spec)  "X = ", x, &
     " Y = ", y, " Z = ", z
     WRITE(unit=6,fmt="(a13,l1)")  "Logical val: ", &
          lacigol
END PROGRAM Owt
```

It produces the following result on the screen:

```
Llanfair...gogogoch
Spock says "illogical Captain"
X =  1.0   Y =  2.0   Z =  3.0
Logical val: F
```

Note:

- each `WRITE` statement begins output on a new line;
- the `WRITE` statement can transfer any object of intrinsic type to the standard output;
- strings may be delimited by the double or single quote symbols, " or ';
- two occurrences of the delimiter inside a string produce one occurrence on output.

READ Statement

A simple form of the `READ` statement which allows you to input from the default input device using a default format, is:

```
Read(*,*)list
```

A general form of the `READ` statement which allows you to input from any device using a specified format, is of the form:

```
Read(unit=u,fmt=<format_specification>)<list>
```

The unit number allows you to read from any device such as a file or the keyboard (5 specifies the keyboard), the format specification is a character string defining how your data is expected to be laid out, and `list` is a comma separated list of variables into which values will be read. For example, if the type declarations are the same as for the `WRITE` example, the statements:

```
READ(*,*)  long_name
READ(*,*)  x, y, z
READ(*,*)  lacigol
```

would accept the following input:

```
Llanphairphwyll___gogogoch
0.4  5.  1.0e12
T
```

Note that each READ statement reads from a new line and the READ statement can transfer any object of intrinsic type from the standard input. Data values on a line are separated by spaces.

Prompting for Input

Suppose a program asks the user for some value, say the temperature in degrees Fahrenheit. If the relevant output and input statements are of the form:

```
Write(unit=6,fmt='(a)',advance='no') &
     'Please type in the temp in F: '
Read(unit=5,fmt=*) Deg_F
```

then the screen dialogue could be the single line:

```
Please type in the temp in F:  32
```

instead of:

```
Please type in the temp in F:
32
```

Reading and writing to a file

In order to read from or write to a file the file concerned has to be specified. To do this use an OPEN statement such as:

```
Open (unit=u, file=<file_name>)
```

where u is the unit number in the READ or WRITE statement and <file_name> is the file name which is to be associated with the unit. Consider the following piece of code:

```
Integer   :: I=5
Real      :: x=5.3, y=2.45
Open (unit=10,file="result")
Write (unit=10,fmt="(i4,f4.1,f5.2)") I,x,y
```

This will result in the following output being written to the file called result

```
5 5.3 2.45
```

Note in this case the format specification has not been assigned to a character variable but has been given as a character literal constant. Either way of specifying the format is acceptable.

Intrinsic Procedures

Fortran 95 has 121 in-built or intrinsic procedures to perform common tasks efficiently. They belong to a number of classes:

- elemental such as:
  - mathematical, for example, `SIN` or `LOG`;
  - numeric, for example, `MAX` or `CEILING`;
  - character, for example, `INDEX` or `ADJUSTL`;
- bit, for example, `IAND` or `IOR`;
- inquiry, for example, `ALLOCATED` or `SIZE`;
- transformational, for example, `RESHAPE` or `SUM`;
- miscellaneous (non-elemental `SUBROUTINE`s), for example, `SYSTEM_CLOCK` and `DATE_AND_TIME`.

Note, all intrinsics which take `REAL` valued arguments also accept all `KIND` of `REAL` arguments.

Type Conversion Functions

It is easy to transform the type of an entity:

- `REAL(i)` converts `INTEGER i` to a real approximation;
- `INT(x)` truncates `REAL x` to the integer equivalent;
- `IACHAR(c)` returns the position of `CHARACTER c` in the ASCII collating sequence;
- `ACHAR(i)` returns the i^{th} character in the ASCII collating sequence.

All the above are intrinsic functions and this piece of code demonstrates them:

```
Write(unit=6,fmt="(F6.2,1X,I2,1X,I2)") &
     REAL(1), INT(1.7), INT(-0.9999)
Write(unit=6,fmt="(I3,1X,A1)") &
     IACHAR('C'), ACHAR(67)
```

will produce the output

```
 1.00  1  0
67 C
```

Mathematical Intrinsic Function Summary

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

Numeric Intrinsic Function Summary

ABS(a)	absolute value
AIMAG(z)	imaginary part of complex value z
AINT(a)	truncates a to whole REAL number
ANINT(a)	nearest whole REAL number
CEILING(a)	smallest INTEGER greater than or equal to REAL number
CMPLX(x,y)	convert to COMPLEX
CONJG(z)	conjugate of complex value z
DIM(x,y)	positive difference
FLOOR(a)	biggest INTEGER less than or equal to REAL number
INT(a)	truncates a into an INTEGER
MAX(a1,a2,a3,...)	the maximum value of the arguments
MIN(a1,a2,a3,...)	the minimum value of the arguments
MOD(a,p)	remainder function
MODULO(a,p)	modulo function
NINT(x)	nearest INTEGER to a REAL number
REAL(a)	converts to the equivalent REAL value
SIGN(a,b)	absolute value of a times the sign of b

Character Intrinsic Function Summary

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

How to Write a Computer Program

There are 4 main steps:

1. specify the problem;
2. analyse and break down into a series of steps towards solution;
3. write the Fortran 95 code;
4. compile and run (i.e., test the program).

It may be necessary to iterate between steps 3 and 4 in order to remove any mistakes. The testing step is very important. For example, consider a program to convert a temperature from Fahrenheit to Celsius scale.

To convert from ^{o}F (Fahrenheit) to ^{o}C (Celsius) we can use the following formula:

$$c = 5 \times (f - 32)/9$$

To convert from ^{o}C to ^{o}K (Kelvin) we add 273.

The algorithm consists of:

1. `READ` a value of temperature on the Fahrenheit scale;
2. calculate the corresponding temperature on the Celsius scale;
3. `WRITE` the value just found;
4. calculate the corresponding temperature in degrees Kelvin;
5. `WRITE` this value.

To program this problem one might use the following code in a file called `TempFtoC.f95`:

```
PROGRAM Temp_Conversion
!  Convert a temperature value from Fahrenheit to Celsius
IMPLICIT NONE

REAL :: Deg_F, Deg_C, Deg_K ! 3 real type variables
!  Obtain a temperature value
   WRITE(unit=6,fmt="(A28)",advance="no") &
     "Please type in the temp in F: "
   READ*, Deg_F
!  Convert from Fahrenheit to Celsius
   Deg_C = 5.0*(Deg_F-32.0)/9.0
!  Output this new value
   WRITE(unit=6,fmt="(A17,F6.1,A2)") &
     "This is equal to ", Deg_C, " C"
!  Convert to Kelvin and output
   Deg_K = Deg_C + 273.0
   WRITE(unit=6,fmt="(A4,F6.1,A2)") "and ", Deg_K, " K"

END PROGRAM Temp_Conversion
```

The form of the program source is essentially free with:

- up to 132 characters per line;
- significant blanks;
- `!' comment initiator;
- `&' line continuation character;
- `;' statement separator.

Example:

```
WRITE(unit=6,fmt="(A39)") "This line is continued &
     &on the next line"; END IF     ! end if statement
```

Now looking more closely at the code. It is delimited by

```
PROGRAM
END PROGRAM
```

statements. Between these there are two distinct areas.

- Specification Part

This gives named memory locations (variables) for use, and specifies the type of each variable.

- `IMPLICIT NONE` -- this should always be present, meaning all variables must be declared.
- `REAL :: Deg_F, Deg_C, Deg_K` -- declares three `REAL` (numeric) type variables.

Note that Fortran 95 is not case sensitive: `K` is the same as `k` and `INTEGER` is the same as `integer`.

- Execution Part

This is the part of the program that does the actual work. It reads in data, calculates the temp in ^{o}C and ^{o}K and writes out results.

- `WRITE(unit=6,fmt="(A28)",advance="no") &`
  `"Please type in the temp in F: "` -- writes the string to the screen;
- `READ*, Deg_F` -- reads a value from the keyboard and assigns it to the `REAL` variable `Deg_F`;
- `Deg_C = 5.0*(Deg_F-32.0)/9.0` -- the expression on the RHS is evaluated and assigned to the `REAL` variable `Deg_C`.

  * is the multiplication operator;
  - is the subtraction operator;
  / is the division operator;
  = is the assignment operator.

- `WRITE(unit=6,fmt="(A17,F6.1,A2)") "This is equal to ",&`
  `Deg_C, "C"` -- displays a string on the screen followed by the value of a variable (`Deg_C`) followed by a second string `("C")`.

By default, input is from the keyboard and output to the screen.

Statement Ordering

The following table details the prescribed ordering:

<table>
<tr><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE or MODULE statement</td></tr>
<tr><td colspan="3">USE statements</td></tr>
<tr><td rowspan="4">FORMAT statements</td><td colspan="2">IMPLICIT NONE</td></tr>
<tr><td>PARAMETER statements</td><td>IMPLICIT statements</td></tr>
<tr><td>PARAMETER statements</td><td>Derived-Type Definitions, Interface blocks, Type declaration and specification statements</td></tr>
<tr><td colspan="2">Executable constructs</td></tr>
<tr><td colspan="3">CONTAINS statement</td></tr>
<tr><td colspan="3">Internal or module procedures</td></tr>
<tr><td colspan="3">END statement</td></tr>
</table>

Compiling and Running the Program

This program can be compiled on holyrood by using the command:

```
holyrood$ f95 TempFtoC.f95
```

and this should produce an executable file named `a.out`

Now test the program to ensure it gives the correct answer:

```
holyrood$  a.out
Please type in the temp in F:
32
This is equal to    0.0 C
and  273.0 K
```

In addition a switch or flag can be supplied to the compiler to specify a name for the executable file. For example the following compiles `TempFtoC.f95` and sends the output to a separate file called `TempFtoC`

```
holyrood$  f95  -o TempFtoC  TempFtoC.f95

holyrood$  TempFtoC
Please type in the temp in F:
212
This is equal to  100.0 C
and  373.0 K
```

If you need more information about the compiler, give the command: `man f95`

It is important to exercise as much of the program as possible with the test data used in demonstrating correctness.

Bugs

Compile-time Errors

In the previous program, consider what would happen if we accidentally typed:

```
Dwg_C = 5.0*(Deg_F - 32.0)/9.0
```

The compiler generates a compile-time or syntax error:

```
holyrood$ f95 TempFtoC.f95

                Dwg_C = 5.0*(Deg_F - 32.0)/9.0
                ^
"TempFtoC.f95", Line = 10, Column = 4: ERROR: IMPLICIT
NONE is specified in the local scope, therefore an
explicit type must be specified for data object "Dwg_C".
```

Run-time Errors
With some compilers an expression such as

```
Deg_C = 5.0*(Deg_F - 32.0)/0.0
```

would compile but a run-time error would be generated. This might take the form:

```
holyrood$ a.out
Please type in the temp in F:
122
Arithmetic exception
```

It is also possible to write a program that runs to completion but gives the wrong results. Be particularly wary if using a program written by someone else: the original author may have thoroughly tested those parts of the program exercised by their data but been less thorough with other parts of the program.

Practical Exercise 1

Question 1: The Hello World Program
Write a Fortran 95 program to write out `Hello World` on the screen.

Question 2: Real Formatting
Write a program which uses the expression `4.0*atan2(1.0,1.0)` to evaluate π and store it in a variable. Write out this value 9 times using edit descriptors of the form `E12.d, F12.d, G12.d` with `d` taking the values 2, 4 and 6.

Question 3: Some Division One Results
A particular number can be expressed as the sum of several integers, and the sum of the reciprocals of these integers is, perhaps, surprising. Write a program to calculate the values of the two following expressions and write a short text and the results:

```
2 + 6 + 8 + 10 + 12 + 40
```

$$\frac{1}{2}+\frac{1}{6}+\frac{1}{8}+\frac{1}{10}+\frac{1}{12}+\frac{1}{40}$$

Hint: some constants are better as type `INTEGER` but some must be type `REAL`.

Now write similar results using the set of numbers `{2,3,10,24,40}`

Question 4: Area of a Circle
Write a simple program to read in the radius and calculate the area of the corresponding circle and volume of the sphere. Demonstrate correctness by calculating the area and volume using radii of 2, 5, 10 and -1.

Area of a circle:

$$area = \pi r^2$$

Volume of a sphere:

$$volume = \frac{4\pi r^3}{3}$$

Hint 1: place the `READ`, the area calculation and the `WRITE` statement in a loop as shown in the program template given below.
Hint 2: use the value 3.14159 for π.

```
PROGRAM Area_and_Vol
!...Add specification part
DO
  WRITE(unit=6,fmt="(A)") "Type in the radius, &
     &a negative value will terminate"
  READ*, radius
  IF (radius < 0) EXIT

  !...Add code to calculate area and volume

  WRITE(unit=6,fmt="(A26,F5.1,A4,F6.1)") &
     "Area of circle with radius ",&
     radius, " is ", area
  WRITE(unit=6,fmt="(A28,F5.1,A4,F6.1)") &
     "Volume of sphere with radius ",&
     radius, " is ", volume
END DO
END PROGRAM Area_and_Vol
```

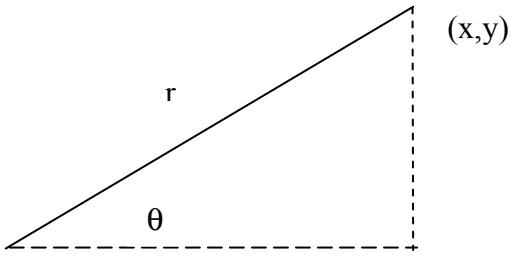
In this way when a negative radius is supplied the program will terminate.

Question 5: Point on a circle

Write a program to read in a vector defined by a length, r, and an angle, θ, in degrees which writes out the corresponding (x, y) co-ordinates. Recall that arguments to trigonometric functions are in radians.

Demonstrate correctness by finding the (x, y) co-ordinates for the following vectors:

1. r = 2, θ = 60°
2. r = 3, θ = 120°
3. r = 5, θ = 240°
4. r = 8, θ = 300°
5. r = 13, θ = 450°



Hint: remember that $\sin\theta = \frac{y}{r}$ and $\cos\theta = \frac{x}{r}$ and 180 degrees = π radians

Question 6: Filed values

Write a program to open the file names `statsa` which has been provided: `statsa` contains several values, each on a separate line (or record). Read the first value which is an integer, and is in a field of width 5. Then read the second value which is of type real, in a field of width 5 with two digits after the decimal point. Write these two values within a line of explanatory text to the screen.

2. Control Constructs and Intrinsics

Control Flow

Control constructs allow the normal sequential order of execution to be changed. Fortran 95 supports:

- conditional execution statements and constructs, (`IF ...` and `IF ... THEN ... ELSE ... END IF`);
- multi-way choice construct, (`SELECT CASE`);
- loops, (`DO ... END DO`).

IF Statement

- The basic syntax of an `IF` statement is:

`IF(<` *`logical-expression`* `>)<` *`exec-stmt`* `>`

If `<` *`logical-expression`* `>` evaluates to `.TRUE.` then execute `<` *`exec-stmt`* `>` otherwise do not.

For example:

```
IF (bool_val) a = 3
IF (x > y) Maxi = x
```

The second means `if `x` is greater than `y` then set `Maxi` to be equal to the value of `x'.

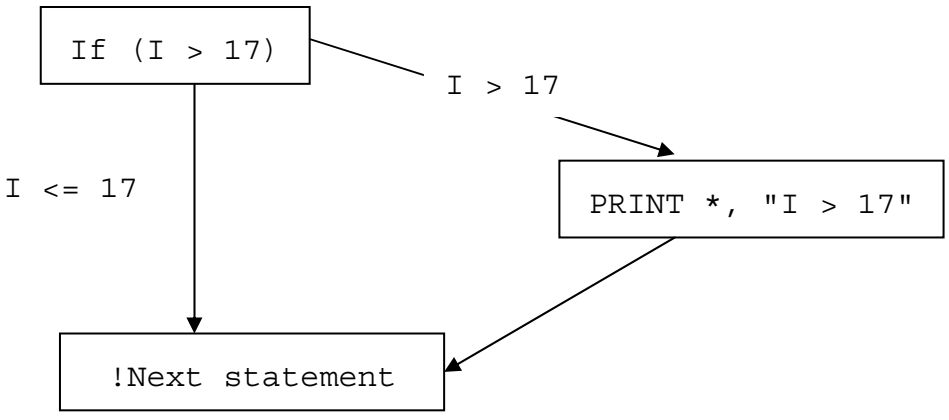
More examples:

```
IF (a*b+c /= 47) Boolie = .TRUE.
IF (i /= 0 .AND. j /= 0) k = l/(i*j)
IF ((i /= 0) .AND. (j /= 0)) k = l/(i*j)     ! same
```

The `IF` Statement can be explained by a flow structure. Consider the `IF` statement:

```
IF (I > 17) Print*, "I > 17"
```

This maps onto the following control flow structure:



When using real-valued expressions (which are approximate) `.EQ.` and `.NE.` have no useful meaning. This example shows a way of treating such a case: `Tol` has been set to a suitable small value.

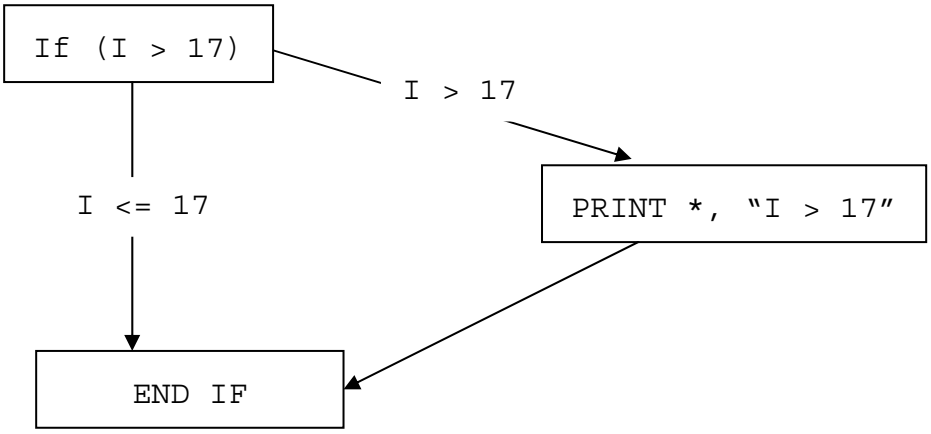
```
IF (ABS(a-b) < Tol) same = .TRUE.
```

IF ... THEN ... ELSE Construct

The block-`IF` is a more flexible version of the single line `IF`. A simple example:

```
IF (I > 17) THEN
     Print*, "I > 17"
END IF
```

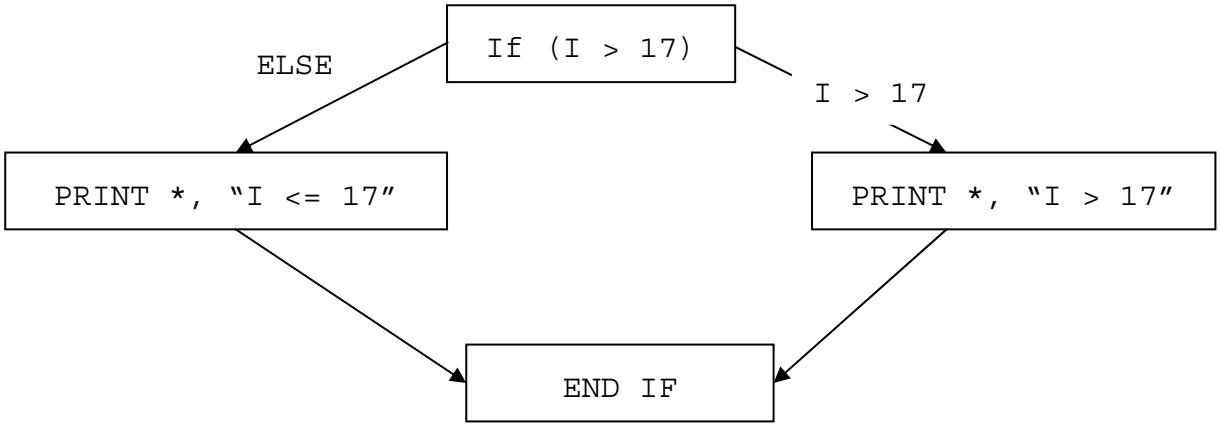
This maps onto the following control flow structure:



Consider the `IF` ... `THEN` ... `ELSE` construct:

```
IF (I > 17) THEN
     Print*, "I > 17"
ELSE
     Print*, "I <= 17"
END IF
```

Note how the indentation helps. This maps onto the following control flow structure:



IF ... THEN .... ELSEIF Construct

The `IF` construct has the following syntax:

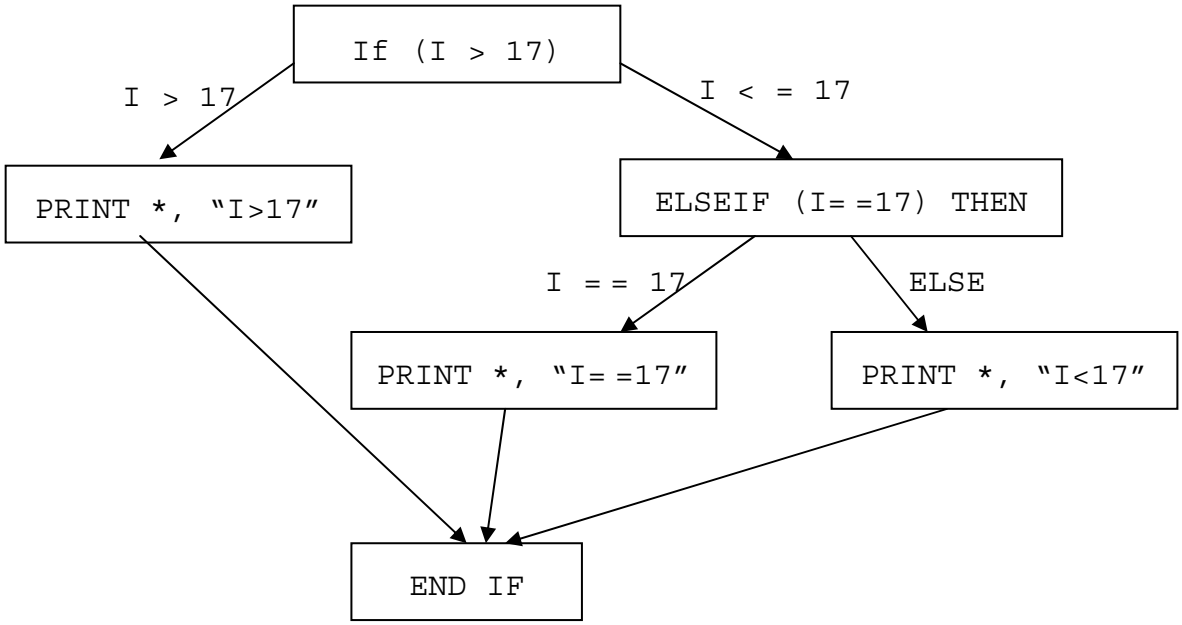
```
IF(< logical-expression >)THEN
     < then-block >
[ ELSEIF(< logical-expression >)THEN
     < elseif-block >
... ]
[ ELSE
     < else-block > ]
END IF
```

The first branch to have a true < *logical-expression* > is the one that is executed. If none is found then the < *else-block* >, if present, is executed. Each of `ELSEIF` and `ELSE` is optional.

Consider the `IF` ...`THEN` ... `ELSEIF` construct:

```
IF (I > 17) THEN
     Print*, "I > 17"
ELSEIF (I == 17) THEN
     Print*, "I == 17"
ELSE
     Print*, "I < 17"
END IF
```

This maps onto the following control flow structure:



You can also have one or more `ELSEIF` branches. `IF` blocks may also be nested. As an example consider:

```
IF (x > 3) THEN
     A = B+C*D
ELSEIF (x == 3) THEN
     A = B*C-D
ELSEIF (x == 2) THEN
     A = B*B
ELSE
     IF (y /= 0) A=B
ENDIF
```

Nested and Named IF Constructs

All control constructs can be both named and nested:

```
outa:   IF (a == 0) THEN
            WRITE(unit=6,fmt="(A5)") "a = 0"
  inna:     IF (c /= 0) THEN
               WRITE(unit=6,fmt="(A16)") "a = 0 AND c /= 0"
            ELSE
               WRITE(unit=6,fmt="(A15)") "a = 0 BUT c = 0"
            ENDIF inna
        ELSE IF (a > 0) THEN
            WRITE(unit=6,fmt="(A5)") "a > 0"
        ELSE
            WRITE(unit=6,fmt="(A13)") "a must be < 0"
        END IF outa
```

The names may only be used once per program unit.

Example Using IF constructs

A program written to calculate the roots of a quadratic equation of the form:

$$ax^2 + bx + c = 0$$

will use some of the constructs just described.
The roots are given by the following formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The algorithm consists of:

1. `READ` values of *a*, *b* and *c*;
2. if *a* is zero then stop as we do not have a quadratic;
3. calculate the value of discriminant D = $b^2 - 4ac$
4. if D is zero then there is one root: $\frac{-b}{2a}$

5. if D > 0 then there are two real roots: $\frac{-b+\sqrt{D}}{2a}$ and $\frac{-b-\sqrt{D}}{2a}$
6. if D < 0 there are two complex roots: $\frac{-b+i\sqrt{-D}}{2a}$ and $\frac{-b-i\sqrt{-D}}{2a}$
7. `WRITE` the solution.

The program for this might look like this:

```
PROGRAM QES
IMPLICIT NONE
INTEGER :: a, b, c, D
REAL :: Real_Part, Imag_Part
WRITE(unit=6,fmt="(A29)") "Type in values for a, b and c"
READ*, a, b, c
IF (a /= 0) THEN
! Calculate discriminant
  D = b*b - 4*a*c
  IF (D == 0) THEN                        ! one root
    WRITE(unit=6,fmt="(A8,F6.1)") "Root is ", -b/(2.0*a)
  ELSE IF (D > 0) THEN                    ! real roots
    WRITE(unit=6,fmt="(A9,F6.1,1X,A3,F5.1)") &
    "Roots are",(-b+SQRT(REAL(D)))/(2.0*a),&
    "and",  (-b-SQRT(REAL(D)))/(2.0*a)
  ELSE                                    ! complex roots
    Real_Part = -b/(2.0*a)
    ! D < 0 so must take SQRT of -D
    Imag_Part = (SQRT(REAL(-D))/(2.0*a))
    WRITE(unit=6,fmt="(A9,F6.1,1X,A1,F5.1,A1)") &
    "1st Root ", Real_Part, "+", Imag_Part, "i"
    WRITE(unit=6,fmt="(A9,F6.1,1X,A1,F5.1,A1)") &
    "2nd Root ", Real_Part, "-", Imag_Part, "i"
  END IF
ELSE                                      ! a == 0
  WRITE(unit=6,fmt="(A24)") "Not a quadratic equation"
END IF
END PROGRAM QES
```

The previous program introduces some new ideas:

- `&` -- means the line is continued;
- `IF` construct -- different statements are executed depending upon the value of the logical expression;
- relational operators -- `==` (is equal to) or > (is greater than);
- nested constructs -- one control construct can be located inside another;
- procedure call -- `SQRT(X)` returns the square root of `X`;
- type conversion -- in the above call, `X` must be of type `REAL`. In the program, `D` is `INTEGER`, `REAL(D)` converts `D` to be real valued. To save CPU time we calculate the discriminant once and store it in `D`.

SELECT CASE Construct

A simple example of a select case construct is:

```
SELECT CASE (i)
     CASE (2,3,5,7)
          WRITE(6,”(A10)”) "i is prime"
     CASE (10:)
          WRITE(6,”(A10)”) "i is >= 10"
     CASE DEFAULT
          WRITE(6,”(A26)”) "i is not prime and is < 10"
END SELECT
```

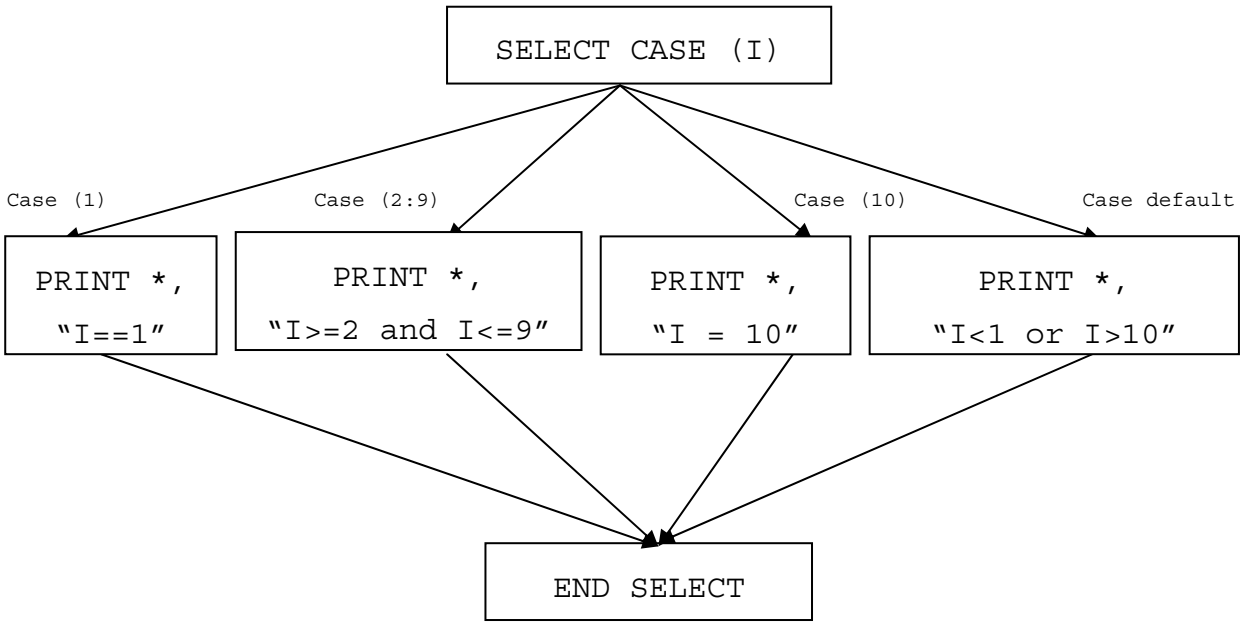
An `IF .. ENDIF` construct could have been used but a `SELECT CASE` is neater and more efficient. Here is the same example:

```
SELECT CASE(i)
     CASE (2,3,5,7)
     ! IF(i==2 .OR. i==3 .OR. i==5 .OR. i==7) THEN
          WRITE(6,”(A10)”) "I is prime"
     CASE (10:)
     ! ELSE IF(i >= 10).THEN
          WRITE(6,”(A10)”) "I is >= 10"
     CASE DEFAULT
     ! ELSE
          WRITE(6,”(A26)”) "I is not prime and is < 10"
END SELECT
     ! END IF
```

The `SELECT CASE` construct can be explained by a flow structure. Consider the `SELECT CASE` construct:

```
 SELECT CASE (I)
      CASE(1);       Print*, "I=1"
      CASE(2:9);     Print*, "I>=2 and I<=9"
      CASE(10);      Print*, "I=10"
      CASE DEFAULT; Print*, "I<1 or I>10"
 END SELECT
```

This maps onto the following control flow structure:



The `SELECT CASE` construct is useful if one of several paths must be chosen based on the value of a single expression.

The syntax is as follows:

```
[ < name >:] SELECT CASE (< case-expr >)
      [ CASE (< case-selector >)[ < name > ]
           < exec-stmts > ] ...
      [ CASE DEFAULT [ < name > ]
           < exec-stmts > ]
END SELECT [ < name > ]
```

Note:

- the `< case-expr >` must be scalar and `INTEGER, LOGICAL` or `CHARACTER` valued;
- the `< case-selector >` is a parenthesised single value or range, for example, `(.TRUE.), (1)` or `(99:101)`;
- there can only be one `CASE DEFAULT` branch;
- control cannot jump into a `CASE` construct.

Conditional Exit Loop

It is possible to set up a DO loop which is terminated by simply jumping out of it:

```
i = 0
DO
     i = i + 1
     IF (i > 100) EXIT
     WRITE(unit=6,fmt="(A4,I4)") "I is", i
END DO
! if I>100  control jumps here
WRITE(unit=6,fmt="(A27,I4)") &
     "Loop finished. I now equals", i
```

This will generate:

```
     I is 1
     I is 2
     I is 3
     ....
     I is 100
     Loop finished. I now equals 101
```

The EXIT statement tells control to jump out of the current DO loop.

Conditional Cycle Loops

You can also set up a DO loop which, on some iterations, only executes a subset of its statements. Consider:

```
i = 0
DO
     i = i + 1
     IF (i >= 50 .AND. i <= 59) CYCLE
     IF (i > 100) EXIT
     WRITE(unit=6,fmt="(A4,I4)") i
END DO
WRITE(unit=6,fmt="(A27,I4)") &
     "Loop finished. I now equals", i
```

This will generate:

```
     I is 1
     I is 2
     ....
     I is 49
     I is 60
     ....
     I is 100
     Loop finished. I now equals 101
```

CYCLE forces control to the innermost active DO statement and the loop begins a new iteration.

Named and Nested Loops

Loops can be given names and an `EXIT` or `CYCLE` statement can be made to refer to a particular loop. This is demonstrated by the code:

```
0-- outa: DO
1--  inna: DO
2--        IF (a > b) EXIT outa          ! jump to line 9
4--        IF (a == b) CYCLE outa        ! jump to line 0
5--        IF (c > d) EXIT inna          ! jump to line 8
6--        IF (c == a) CYCLE             ! jump to line 1
7--        END DO inna
8--  END DO outa
9--  ...
```

The (optional) name following the `EXIT` or `CYCLE` determines which loop the statement refers to. Note that the name given to a loop cannot be given to any other object in the program unit.

Indexed DO Loops

Loops can be written which cycle a fixed number of times. For example:

```
DO i = 1, 100, 1
     ... ! i takes the values 1,2,3,...,100
     ... ! 100 iterations
END DO
```

The formal syntax is as follows:

```
DO < DO-var > = < expr1 >, < expr2 > [,< expr3 > ]
     < exec-stmts >
END DO
```

The number of iterations, which is evaluated before execution of the loop begins, is calculated as:

```
MAX(INT(< expr2 >-< expr1 >+< expr3 >)/< expr3 >), 0)
```

If this is zero or negative then the loop is not executed.
If `< expr3 >` is absent it is assumed to be equal to 1.
The `< DO-var >` must not be explicitly modified within the `DO` construct.

Here are four examples of different loops:

Upper bound not exact

```
DO i = 1, 30, 2
     ... ! i takes the values 1,3,5,7,...,29
     ... ! 15 iterations
END DO
```

Negative stride

```
DO j = 30, 1, -2
     ... ! j takes the values 30,28,26,...,2
     ... ! 15 iterations
END DO
```

Zero-trip loop

```
DO k = 30, 1, 2
     ... ! 0 iterations
     ... ! loop skipped
END DO
```

Missing stride -- assumed to be 1

```
DO l = 1, 30
     ... ! l takes the values 1,2,3,...,30
     ... ! 30 iterations
END DO
```

DO construct index

The value of the index variable is incremented at the end of each loop ready for the next iteration of the loop: this value is available outside the loop. With a piece of code like this there are three possible outcomes for the index variable:

```
DO i = 1, n
  ...
  IF (i == k) EXIT
  ...
END DO
```

1. If, at execution time, `n` is less than `1` it is a zero-trip loop so `i` is given the value `1` and control passes to the statement following `END DO`.
2. If `n` is greater than `1` and not less than `k` then `i` will have the same value as `k` when `EXIT` transfers control to the statement following `END DO`.
3. If `n` is greater than `1` and less than `k` then the loop will be executed `n` times with `i` taking the values `1,2,...,n`. At the end of the n^{th} loop `i` will be incremented to `n+1` and will have this value when control transfers to the statement following `END DO`.

Practical Exercise 2

Question 1: Parity

Write a program to read several numbers, positive or negative, one at a time and for each to write out a line giving the number just read and a description of it as an odd or even number. Stop if the number read in is zero.

Question 2: A Triangle Program

Write a program to accept three (`INTEGER`) lengths and report back on whether these lengths could define an equilateral, isosceles or scalene triangle (3, 2 or 0 equal length sides) or whether they cannot form a triangle.

Demonstrate that the program works by classifying the following:

1. (1, 1, 1)
2. (2, 2, 1)
3. (1, 1, 0)
4. (3, 4, 5)
5. (3, 2, 1)
6. (1, 2, 4)

Hint: If three lengths form a triangle then 2 times the longest side must be less than the sum of all three sides. In Fortran 95 terms, the following must be true:
`(2*MAX(side1,side2,side3) < side1+side2+side3)`

Question 3: The Ludolphian Number

Write a program which uses 6 variables of type real; `a`, `b`, `c`, `d`, `e`, `f` (or any other names you choose). Set initial values as follows, remembering to match the type of constant to the type of variable:

$$a = 1, \quad b = \frac{1}{\sqrt{2}}, \quad c = \frac{1}{4}, \quad d = 1$$

Code these 7 lines as Fortran 95 statements (with constants of the correct type) within a loop which is to be obeyed 4 times:

$e = a$

$$a = \frac{(a+b)}{2}$$

$$b = \sqrt{b \times e}$$

$$c = c - d \times (a-e)^2$$

$d = 2d$

$$f = \frac{(a+b)^2}{4c}$$

output f

This algorithm was developed by Tamura and Kanada.

Question 4: Odd Numbers
Write a program which:
1. Asks how many odd numbers you want to use.
2. Reads in a number (16 would be sufficient to test your program).
3. Sums this many odd numbers, starting from 1 (Do not use the formula for the sum of an arithmetic progression!)
   As each number is added in, write out a count of how many odd numbers have been added in and what the sum is. So the first line will simply be:
   1 1

Question 5: Simple Sequences (symmetric, unitary, descending)
For each of these sequences set an initial value and use a DO-loop.

a) Write a program to calculate and write out each of the terms in this sequence:

1 x 1
11 x 11
111 x 111
:
11111 x 11111

Now calculate and write out the next term in this sequence. Anything strange?

b) Write a program to calculate and write out each of the terms in this sequence:

0 x 9 + 1
1 x 9 + 2
12 x 9 + 3
123 x 9 + 4
:
12345678 x 9 + 9

c) Write a program to calculate and write out each of the terms in this sequence:

1 x 8 + 1
12 x 8 + 2
123 x 8 + 3
:
123456789 x 8 + 9

Question 6: Mathematical Magic
If you take a positive integer, halve it if it is even or triple it and add one if it is odd, and repeat, then the number will eventually become one. This is known as the Syracuse algorithm.

Set up a loop containing a statement to read in a number (input terminated by zero) and a loop to write out the sequence obtained from each input. The number 13 is considered to be very unlucky and if it is obtained as part of the sequence then execution should immediately terminate with an appropriate message.

Demonstrate that your program works by outputting the sequences generated by the following sets of numbers:

a) 7
b) 106, 46, 3, 0

Question 7: Decimal to Roman Numerals Conversion

Using a `SELECT CASE` block and integer division write a program that reads in a decimal number between 0 and 999 and writes out the equivalent in Roman Numerals. Demonstrate that your program works with the numbers:

1. 888
2. 0
3. 222
4. 536

The output should contain no embedded spaces.

0					
1	i	1.	x	1..	c
2	ii	2.	xx	2..	cc
3	iii	3.	xxx	3..	ccc
4	iv	4.	xl	4..	cd
5	v	5.	l	5..	d
6	vi	6.	lx	6..	dc
7	vii	7.	lxx	7..	dcc
8	viii	8.	lxxx	8..	dccc
9	ix	9.	xc	9..	cm

Hint: Use a `CHARACTER` string (or `CHARACTER` strings) to store the number before output. The `longest' number is 888, dccclxxxviii (12 characters).

Question 8: Solving a Quadratic Equation

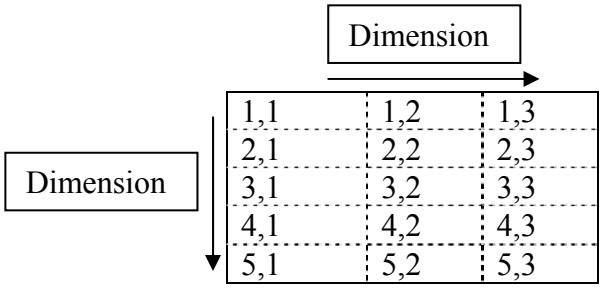
1. Using an editor, type the program QES on page 31 into a file called `QuadSolver.f95`
2. Compile and run the program. Verify the correctness of the code by supplying the following test data:
   a) a = 1, b = -3 and c = 2,
   b) a = 1, b = -2 and c = 1,
   c) a = 1, b = 1 and c = 1,
   d) a = 0, b = 2 and c = 3.
3. Copy `QuadSolver.f95` into a new file called `NewQuadSolver.f95`.
4. Define a new `REAL` variable called `sqrt_D` and where appropriate pre-calculate `SQRT(REAL(D))` and substitute this new variable in place of this expression. Compile the code to produce an executable file called `NewQuadSolver`.
5. Use a different set of test data to that given above to convince yourself that `NewQuadSolver.f95` is a correct program.
6. Delete the original program file `QuadSolver.f95` and the executable file `a.out`.

3. Arrays

Arrays (or matrices) hold a collection of different values at the same time. Individual elements are accessed by **subscripting** the array.

A 15-element array can be visualised as:

And a 5 x 3 array as:



Every array has a type and each element holds a value of that type. Here are some examples of declarations used in Fortran:

```
REAL, DIMENSION(15)       :: X
REAL, DIMENSION(1:5,1:3)  :: Y, Z      ! 5 rows, 3 columns
```

The above are *explicit-shape* arrays. Further terminology you might meet includes:

- **rank** -- number of dimensions. Rank of `X` is `1`; rank of `Y` and `Z` is `2`.
- **bounds** -- lower and upper limits of indices. Bounds of `X` are `1` and `15`; bounds of `Y` and `Z` are `1` and `5` and `1` and `3`.
- **extent** -- number of elements in dimension. Extent of `X` is `15`; extents of `Y` and `Z` are `5` and `3`.
- **size** -- total number of elements. Size of `X`, `Y` and `Z` is `15`.
- **shape** -- rank and extents. Shape of `X` is `15`; shape of `Y` and `Z` is `5,3`.
- **conformable** -- same shape. `Y` and `Z` are conformable.

Declarations

Literals and constants can be used in array declarations:

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

But note:

- the default lower bound is 1;
- bounds can begin and end anywhere.

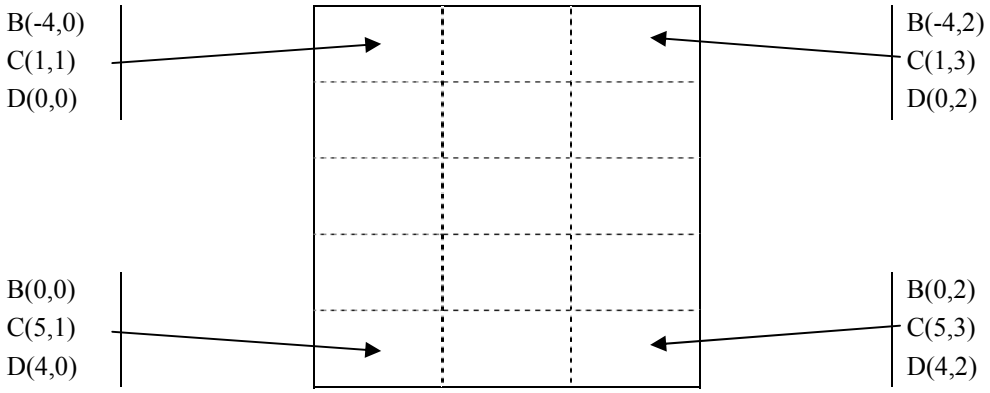
Now consider how these arrays look diagrammatically:

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

Individual array elements are denoted by subscripting the array name by an `INTEGER`, for example, `A(7)`, 7th element of `A`:


```
REAL, DIMENSION(-4:0,0:2) :: B
REAL, DIMENSION(5,3)      :: C
REAL, DIMENSION(0:4,0:2)  :: D
```

or `C(3,2)`, 3rd row, 2nd column of `C`:

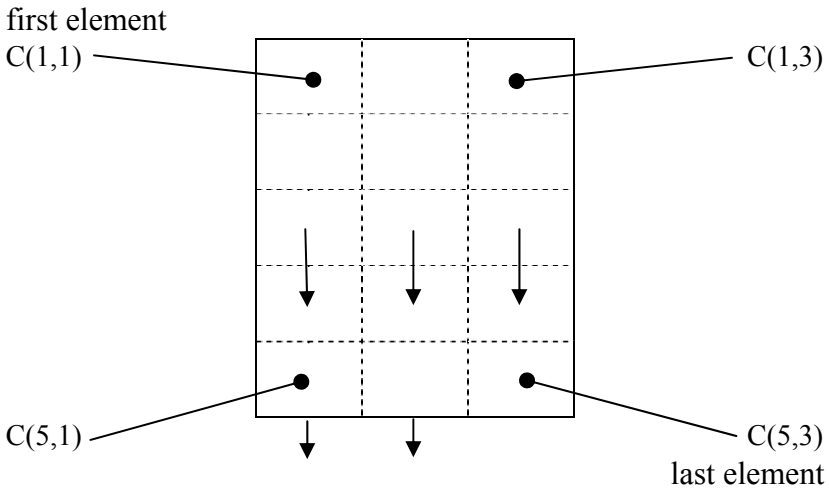


Array Element Ordering

Organisation in memory:

- Fortran 95 does not specify anything about how arrays should be located in memory. **It has no storage association.**
- Fortran 95 does define an array element ordering for certain situations, which is of column major form.

The array is conceptually ordered as:



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

Array Sections

These are specified by subscript-triplets for each dimension of the array. The general form is:

`[< bound1 >]:[< bound2 >][:< stride >]`

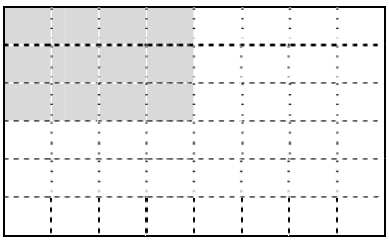
The section starts at `< bound1 >` and ends at or before `< bound2 >`.
`< stride >` is the increment by which the elements are selected.

`< bound1 >`, `< bound2 >` and `< stride >` must all be scalar integer expressions. Thus, almost all of these are valid sections:

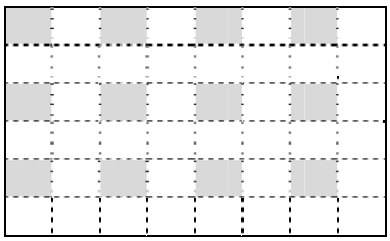
A(:)	!	the whole array
A(3:9)	!	A(3) to A(9) in steps of 1
A(3:9:1)	!	as above
A(m:n)	!	A(m) to A(n)
A(m:n:k)	!	A(m) to A(n) in steps of k
A(8:3:-1)	!	A(8) to A(3) in steps of –1
A(8:3)	!	A(8) to A(3) in steps of 1 => Zero size
A(m:)	!	from A(m) to UBOUND(A)
A(:n)	!	from LBOUND(A) to A(n)
A(::2)	!	from LBOUND(A) to UBOUND(A) in steps of 2
A(m:m)	!	1 element section of rank 1
A(m)	!	scalar element - not a section

The following statements illustrate the declaration of an array and some sections of it:

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

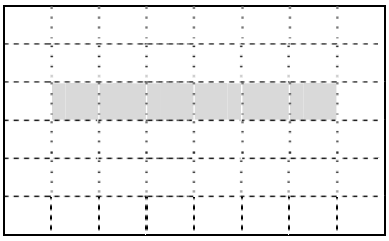


P(1:3,1:4)



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

Two sections each of shape 3,4



P(3,2:7) P(3:3,2:7

`P(3,2:7)` is a section of rank 1 whereas `P(3:3,2:7)` is a section of rank 2.

Array Conformance

Two arrays or sub-arrays are said to be conformable if they have the same shape (identical rank and extents): a scalar is conformable with any array.

Array Syntax

You can reference a particular element of an array by using the array name and a valid subscriptUsing the arrays which were declared previously:

```
REAL, DIMENSION(15)        :: A
REAL, DIMENSION(-4:0,0:2)  :: B
REAL, DIMENSION(5,3)       :: C
REAL, DIMENSION(0:4,0:2)   :: D
```

`A(1) = 0.0`	sets one element to zero,
`B(0,0) = A(3) + C(5,1)`	sets an element of `B` to the sum of an element of `A` and an element of `C`.

Whole Array Expressions

If an unary intrinsic operation is applied to an array this produces an array of the same shape where each element has a value equal to the operation being performed on the corresponding element of the operand:

```
B = SIN(C)          ! B(i,j) = SIN(C(i+5,j+1))
```

Similarly if a binary intrinsic operation is applied to two arrays of the same shape this produces an array of the same shape where each element has a value equal to the operation being performed on the corresponding element of the operand:

```
B = C + D              ! B(i,j) = C(i+5,j+1) + D(i+4,j)
```

The correspondence of elements in these operations is by position in each extent and not by subscript.

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

```
C = 1.0                ! C(i,j) = 1.0
```

Array Section Expressions
The rules which apply to whole array expressions also apply to array section expressions.

The following program performs an array operation on only certain elements and uses three functions on arrays. In each case an array section has been used: the statement, the two transformational and one elemental functions are described before the program code is given.

The Fortran standard does not prescribe the order in which scalar operations in any array expression are executed so the compiler is free to optimize such expressions.

WHERE statement and construct

The general form of the statement is:

```
WHERE (<logical-array-expr>) <array-variable> = <expr>
```

The logical array expression *<logical-array-expr>* must have the same shape as *<array-variable>*. It is evaluated first and for those elements which have the value true the corresponding elements of *<expr>* are evaluated and assigned to the corresponding elements of *<array-variable>*. For all other elements the corresponding values of *<expr>* are not evaluated and the corresponding elements of *<array-variable>* retain their existing values.
A single logical array expression can be used as a mask controlling several array assignments if all the arrays are of the same shape:

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

COUNT function

The general form of the function is:

`COUNT (<logical-array-expr>)`

This returns the integer value of the number of elements in the logical array expression `<logical-array-expr>` which have the value true. For example:

```
nonnegP = COUNT(P > 0.0)
```

SUM function

The general form of the function is:

`SUM (<array>)`

This returns the sum of the elements of an integer, real or complex `<array>`: it returns the value zero if the `<array>` has size zero. For example:

```
sump = SUM(P)
```

MOD function

The general form of the function is:

`MOD (A,P)`

This returns the remainder `A` modulo `P`, that is `A-INT(A/P) * P`. `A` and `P` must both be integer or real.

An old method of multiplying two numbers is to write down the two numbers and then generate a sequence from each in the following manner. Choose one and halve it, discarding any remainder, and continue until it becomes 1: double the other number as many times as the first was halved. Cross out the numbers in the doubling sequence which correspond to a number in the halving sequence which is even: then sum the remaining numbers. For example, to multiply 13 by 37:

13	6	3	1	
37	~~74~~	148	296	481

```
PROGRAM old_times
! An old method of multiplying two integers,
!  described on page 44 of Curious and Interesting Numbers.
! program written by Neil Hamilton-Smith, March 2006
IMPLICIT NONE
INTEGER :: n=2, p1, p2, p3, z
INTEGER, PARAMETER :: rn = 32
INTEGER, DIMENSION(1:rn) :: r1, r2

Write(unit=6,fmt="(a46)",advance="no") &
"Please give the two numbers to be multiplied: "
Read(unit=5,fmt=*) p1, p2

! store the two sequences of values in arrays r1 and r2
r1(1) = p1
r2(1) = p2
DO
  r1(n) = r1(n-1)/2
  r2(n) = r2(n-1)*2
  IF ( r1(n) == 1 ) EXIT
  n = n + 1
  IF ( n > rn ) THEN
    Write(unit=6,fmt="(a43,i3)") &
      "Arrays r1, r2 need upper bound greater than", rn
    STOP
  END IF
END DO

! cross out value in r2 if value in r1 is even
WHERE (Mod(r1(1:n),2) == 0) r2(1:n) = 0

! count the zeros in r2: equals count of evens in r1.
! For interest, not strict necessity
z = COUNT(r2(1:n) == 0)
Write(unit=6,fmt="(a42,i4)") &
  " Number of even numbers in halved row =", z

! sum the values (including zeros) in r2
p3 = SUM(r2(1:n))
Write(unit=6,fmt="(a10,i5,a4,i5,a3,i8)") "product of", &
  p1, " and", p2, " is", p3

END PROGRAM old_times
```

Suppose we have an array of numbers as shown below and wish to determine the minimum and maximum values and their positions within the array:

3	6	17	24	15
10	18	21	12	4
19	25	13	1	7
22	14	5	8	16
11	2	9	20	23

The program to find the four items will use four transformational functions.

MINVAL function

The general form of the function is:

`MINVAL (<array>)`

This returns the minimum value of an element of an integer or real `<array>`: it returns the largest positive value supported by the processor if the `<array>` has size zero.
For example:

```
minP = MINVAL(P)
```

MAXVAL function

The general form of the function is:

`MAXVAL (<array>)`

This returns the maximum value of an element of an integer or real `<array>`: it returns the largest negative value supported by the processor if the `<array>` has size zero.
For example:

```
maxP = MAXVAL(P)
```

MINLOC function

The general form of the function is:

`MINLOC (<array>)`

This returns a rank-one default integer array of size equal to the rank of `<array>`: its value is the sequence of subscripts of an element of minimum value, as though all the lower bounds of `<array>` were 1. If there is more than one such element then the first in array element order is taken.

MAXLOC function

The general form of the function is:

`MAXLOC (<array>)`

This returns a rank-one default integer array of size equal to the rank of `<array>`: its value is the sequence of subscripts of an element of maximum value, as though all the lower bounds of `<array>` were 1. If there is more than one such element then the first in array element order is taken.

The program could be:

```
program seek_extremes
implicit none
! showing use of four intrinsic array functions
integer, dimension(1:5,1:5) :: magi
integer, dimension(1:2)     :: posmax, posmin
integer :: m1, m25

! assign values to the rank-two array magi

m1 = minval(magi)
m25 = maxval(magi)
posmin = minloc(magi)
posmax = maxloc(magi)
write(6,"(a,i3,a,i2,a,i2)") "The least value,", m1, &
  ", is in row", posmin(1), " and column", posmin(2)
write(6,"(a,i3,a,i2,a,i2)") "The greatest value,", &
  m25, ", is in row", posmax(1), " and column", posmax(2)

end program seek_extremes
```

Array I/O

The conceptual ordering of array elements is useful for defining the order in which array elements are output. If `A` is a rank 2 array then

```
WRITE(*,*) A
```

would produce output in the order:

```
A(1,1),A(2,1),A(3,1),..,A(1,2),A(2,2),..
```

and

```
READ(*,*) A
```

would assign to the elements in the above order.

Using intrinsic functions such as `RESHAPE` or `TRANSPOSE` you could change this order. As an example consider the matrix `A`:

1	4	7
2	5	8
3	6	9

The following `WRITE` statements

```
  ...
  WRITE(unit=6,fmt="(A19,I2)")  'Array element =',a(3,2)
  WRITE(unit=6,fmt="(A19,3I2)") 'Array section =',a(:,1)
  WRITE(unit=6,fmt="(A19,4I2)") 'Sub-array =',a(:2,:2)
  WRITE(unit=6,fmt="(A19,9I2)") 'Whole Array =',a
  WRITE(unit=6,fmt="(A19,9I2)") 'Array Transp''d =', &
    &TRANSPOSE(a)
END PROGRAM WrtArray
```

produce on the screen:

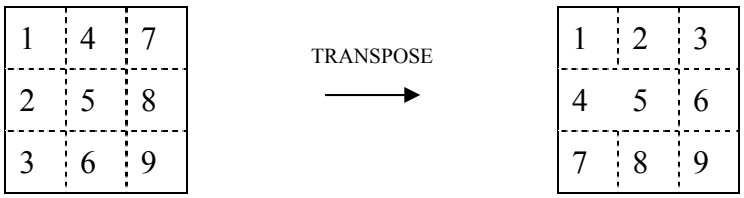
```
 Array element = 6
 Array section = 1 2 3
     Sub-array = 1 2 4 5
   Whole Array = 1 2 3 4 5 6 7 8 9
Array Transp'd = 1 4 7 2 5 8 3 6 9
```

The `TRANSPOSE` Intrinsic Function

`TRANSPOSE` is a general intrinsic function, which takes a rank-two matrix and returns a rank-two matrix of the same type which is its transpose, ie each element (i,j) is replaced by element (j,i):

```
TRANSPOSE(<matrix>)
```

For example:



Array Constructors

Used to give arrays or sections of arrays specific values. For example:

```
IMPLICIT NONE
INTEGER                             :: I
INTEGER, DIMENSION(10)              :: ints
CHARACTER(len=5), DIMENSION(3)      :: colours
REAL, DIMENSION(4)                  :: heights
heights = (/5.10, 5.6, 4.0, 3.6/)
colours = (/'RED  ','GREEN','BLUE '/)
! note padding so strings are 5 chars long
ints = (/ 100, (i, i=1,8), 100 /)
...
```

- constructors and array sections must conform.
- must be rank 1.
- for higher rank arrays use `RESHAPE` intrinsic.
- `(i, i=1,8)` is an *implied* `DO` and is `1,2,..,8`: it is possible to specify a stride.

The `RESHAPE` Intrinsic Function

`RESHAPE` is a general intrinsic function, which delivers an array of a specific shape given by the rank-one integer array <*shape*>:

```
RESHAPE(<source>, <shape>)
```

For example:

```
! declare an array A
INTEGER, DIMENSION(1:2,1:2)   :: A
! assign values to this array A
A = RESHAPE((/1,2,3,4/),(/2,2/))
```

A is filled in array element order and looks like:

```
1     3
2     4
```

Visualisation:

Named Array Constants

Named array constants may be created:

```
INTEGER, DIMENSION(3), PARAMETER :: &
          Unit_vec = (/1,1,1/)
REAL, DIMENSION(3,3), PARAMETER :: &
          Unit_matrix = &
               RESHAPE((/1,0,0,0,1,0,0,0,1/),(/3,3/))
```

Allocatable Arrays

Fortran 95 allows arrays to be created by dynamic memory allocation.

- Declaration:

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

  Note the ALLOCATABLE attribute and fixed rank. These are known as deferred-shape arrays because the actual shape is deferred until allocation.

- Allocation:

```
READ(*,*) isize
ALLOCATE(ages(isize), STAT=ierr)
IF (ierr /= 0) WRITE(6,"(A)") "ages : Allocation failed"

ALLOCATE(speed(0:isize-1,10),STAT=ierr)
IF (ierr /= 0) WRITE(6,'"(A)") "speed : Allocation failed"
```

- the optional STAT= field reports on the success of the storage request. If the INTEGER variable ierr is zero the request was successful otherwise it failed.

Deallocating Arrays

Storage can be reclaimed by using the DEALLOCATE statement:

```
IF (ALLOCATED(ages)) DEALLOCATE(ages,STAT=ierr)
```

- it is an error to deallocate an array without the ALLOCATE attribute or one that has not previously been allocated space;
- there is an intrinsic function, ALLOCATED, which returns a scalar LOGICAL value reporting on the status of an array;
- the STAT= field is optional but its use is recommended;
- if a procedure containing an allocatable array which does not have the SAVE attribute (see page 62) is exited without the array being DEALLOCATEd then this storage becomes inaccessible.

Vector and Matrix Multiplication

There are two intrinsic functions which perform vector and matrix multiplications. Each function has two arguments which are both of numeric type (integer, real or complex) or both of logical type: the result is of the same type as the multiply or logical and operation between two such scalars.

The function for vectors is:

```
dot_product(vector_a, vector_b)
```

which requires two arguments of rank-one and the same size.

- If `vector_a` is of type integer or real then the result is
  `sum(vector_a * vector_b)`
- If `vector_a` is of type complex then the result is
  `sum(conjg(vector_a) * vector_b)`
- If `vector_a` is of type logical then the result is `.true.` if any element of `vector_a .and. vector_b` is `.true.`.

If at least one argument is a rank-two matrix then the function is:

```
matmul(matrix_a, matrix_b)
```

where there are three possibilities depending on the shape of the arguments.

- If `matrix_a` has shape `(n,m)` and `matrix_b` has shape `(m,k)` then the result has shape `(n,k)` and element `(i,j)` has the value
  `sum(matrix_a(i,:) * matrix_b(:,j))`
- If `matrix_a` has shape `(m)` and `matrix_b` has shape `(m,k)` then the result has shape `(k)` and element `(j)` has the value `sum(matrix_a * matrix_b(:,j))`
- If `matrix_a` has shape `(n,m)` and `matrix_b` has shape `(m)` then the result has shape `(n)` and element `(i)` has the value `sum(matrix_a(i,:) * matrix_b)`

- If the arguments are of logical type the shapes are as for numeric arguments and the values are determined by replacing `sum` and `*` by `any` and `.and.`.

Practical Exercise 3

Question 1: Rank, Bounds etc.
Give the rank, bounds, size and shape of the arrays defined as follows:

```
REAL, DIMENSION(1:10) :: ONE
REAL, DIMENSION(2,0:2) :: TWO
INTEGER, DIMENSION(-1:1,3,2) :: THREE
REAL, DIMENSION(0:1,3) :: FOUR
```

Write down the array element order of each array.

Which two of the arrays are conformable?

Question 2: Array Sections
Declare an array which would be suitable for representing a chess board. Write a program to set all the white squares to zero and the black squares to one. (A chess board is 8×8 with alternate black and white squares.) Use formatted output to display your chess board on the screen.

Question 3: Array Constructor
Euler noted that a sequence of 40 prime numbers p starting at 41 can be found from the formula:

$$p = 41 + x + x^2, \quad \text{for } 0 \leq x \leq 39$$

Write a program using an array constructor to store this sequence of 40 primes in an array, putting the first prime in element 0 or 1. Use formatted write to output the sequence on your screen, with at most 5 primes on each row.

Question 4: Fibonacci Numbers
The Fibonacci numbers are defined as follows:

$$u_0 = 0;\ u_1 = 1;\ u_n = u_{n-2} + u_{n-1} \text{ for } n \geq 2$$

Write a program to generate and store in an array the Fibonacci numbers up to and including u_{24}.

The sum of the first n numbers is $u_{n+2} - 1$. Use the intrinsic function SUM on an array section to find the sum of the numbers u_0 to u_{22}. Compare this result with the value of $u_{24} - 1$.

The sum of the first n numbers with odd indices is:

$$u_1 + u_3 + u_5 + \ldots + u_{2n-1} = u_{2n}.$$

Use the intrinsic function SUM on an array section to find the sum of the numbers with odd indices up to u_{23}. Compare this result with the value of u_{24}.

The sum of the first n numbers with even indices is:

$$u_2 + u_4 + u_6 + \ldots + u_{2n} = u_{2n+1} - 1.$$

Use the intrinsic function SUM on an array section to find the sum of the numbers with even indices up to u_{22}. Compare this result with the value of $u_{23} - 1$.

Question 5: Magic Squares

A magic square is a set of numbers arranged in a square array so that the sum of the numbers in each row, the sum of the numbers in each column and the sum of the numbers along each diagonal are all equal. This sum is known as the magic number of this particular magic square.

Write a program to create two 3 x 3 arrays holding these magic squares:

4	9	2		9	2	7
3	5	7		4	6	8
8	1	6		5	10	3

- For each magic square write a line of text as a heading and then the magic square.
- Add the two magic squares together and save the result in a third array: write a heading and then this magic square.
- Check that this is a new magic square by comparing the sums across the first row, down the last column and along the leading diagonal.

Question 6: Symmetry

Write a program to work with the first magic square of Question 5.

- Write the square's magic number (the row, column or diagonal sum). You can check your answer because for an n x n magic square consisting of any arrangement of the integers 1 to n^2 the formula is $(n^3 + n)/2$
- Use the intrinsic function TRANSPOSE to save the transpose of the magic square in a new 3 x 3 array.
- Add the magic square to its transpose and save the result in a new array: this should be a symmetric matrix. Check that the bottom left and top right elements are equal: write out the symmetric matrix.

Question 7: More Magic

Modify the Mathematical Magic program which you wrote for Exercise 2, Question 6 to save the sequences generated in an array. Write out each sequence and find the largest value in each of these sequences and the position in the sequence at which it occurs.

Question 8: MATMUL Intrinsic

For the declarations

```
REAL, DIMENSION(8,8) :: A, B, C
```

what is the difference between `C=MATMUL(A,B)` and `C=A*B`?

Question 9: More Filed values

Modify the Filed values program which you wrote for Exercise 2, Question 7 to declare an allocatable rank one array of type real. Use the integer value which is read in from the file `statsa` as the upper bound for the array when it is allocated (and make the lower bound 1). Then fill the array with real values read from the file. (All values are in fields of width 5 with two digits after the decimal point.) Write out these real values, with 5 on each line. Deallocate the array.

4. Procedures

Program Units

Fortran 95 has two main program units:

- main `PROGRAM` is the place where execution begins and where control should eventually return before the program terminates. It may contain procedures.
- `MODULE` is a program unit, which can contain procedures and declarations. It is intended to be used by another program unit where the entities defined within it become accessible.

There are two types of procedures:

- `SUBROUTINE` is a parameterised named sequence of code which performs one or more specific tasks and can be invoked from within other program units.
- `FUNCTION` is a parameterised named sequence of code which returns a result in the function name (of any specified type and kind).

Main Program Syntax

```
PROGRAM Main
! ...
CONTAINS ! Internal Procs
    SUBROUTINE Sub1(..)
    ! Executable stmts
    END SUBROUTINE Sub1
    ! etc.
    FUNCTION Funkyn(...)
    ! Executable stmts
    END FUNCTION Funkyn
END PROGRAM Main
```

```
[ PROGRAM [ < main program name > ] ]
< declaration of local objects >
. . .
< executable statements >
. . .
[ CONTAINS
< internal procedure definitions > ]
END [ PROGRAM [< main program name > ] ]
```

As an example consider the following:

```
PROGRAM Main
     IMPLICIT NONE
     REAL :: x
          READ(*,*) x
          WRITE(unit=6,fmt="(I6)") FLOOR(x)   ! Intrinsic
          WRITE(unit=6,fmt="(F12.4)") Negative(x)
CONTAINS
     REAL FUNCTION Negative(a)
          REAL :: a
          Negative = -a
     END FUNCTION Negative
END PROGRAM Main
```

Introduction to Procedures

The first question should be: "Do we really need to write a procedure?" Functionality often exists. For instance look first at:

- intrinsics, Fortran 95 has 121;
- libraries, for example, NAG fl90 Numerical Library has 300+, BLAS, IMSL, LaPACK. Library routines are usually very fast, sometimes faster than Intrinsics.
- modules, number growing, many free! See WWW.

Subroutines

Consider the following example:

```
PROGRAM Thingy
     IMPLICIT NONE
     REAL.DIMENSION(1:5) :: NumberSet = (/1,2,3,4,5/)
     .....
     CALL OutputFigures(NumberSet)
     .....
CONTAINS
 SUBROUTINE OutputFigures(Numbers)
 REAL, DIMENSION(:) :: Numbers
 WRITE(6,"(A,(/5F12.4))") "Here are the figures", Numbers
 END SUBROUTINE OutputFigures
END PROGRAM Thingy
```

Internal subroutines lie between the `CONTAINS` and `END PROGRAM` statements and have the following syntax:

```
SUBROUTINE < procname >[ (< dummy args >) ]
     < declaration of dummy args >
     < declaration of local objects >
          . . .
     < executable stmts >
END [ SUBROUTINE [< procname > ] ]
```

Note that, in the example, the `IMPLICIT NONE` statement applies to the whole program including the `SUBROUTINE`.

Functions

Consider the following example:

```
PROGRAM Thingy
     IMPLICIT NONE
     .....
     WRITE(unit=6,fmt="(F7.3)") theta(a,b,c)
     .....
CONTAINS
     REAL FUNCTION theta(x,y,z)
     ! return the angle between sides x and y
          REAL :: x, y, z
          !  check that sides do make a triangle
          IF (2*MAX(x,y,z) < (x+y+z)) THEN
               theta = ACOS((x**2+y**2-z**2)/(2.0*x*y))
          ELSE      !  sides do not make a triangle
               theta = 0.0
          END IF
     END FUNCTION theta
END PROGRAM Thingy
```

Internal functions also lie between the `CONTAINS` and `END PROGRAM` statements. Functions have the following syntax:

```
[< prefix >] FUNCTION < procname > ( [< dummyargs >])
     < declaration of dummy args >
     < declaration of local objects >
          . . .
     < executable stmts, assignment of result >
END [ FUNCTION [< procname > ] ]
```

The function type could be declared in the declarations area instead of in the header. A function returns its result through its name, and is usually used in an expression.

Argument Association

Recall, with the `SUBROUTINE` we had an invocation:

```
CALL OutputFigures(NumberSet)
```

and a declaration:

```
SUBROUTINE OutputFigures(Numbers)
```

where `NumberSet` is an *actual argument* and is *argument associated* with the *dummy argument* `Numbers`. The actual argument must agree in type with the dummy argument.

For the above call, in `OutputFigures`, the name `Numbers` is an **alias** for `NumberSet`. Likewise, consider the two statements:

```
     WRITE(unit=6,fmt="(F7.3)") theta(a,b,c)

     REAL FUNCTION theta(x,y,z)
```

The actual arguments `a`, `b` and `c` are associated with the dummy arguments `x`, `y` and `z`. If the value of a dummy argument changes and the actual argument is a variable then so does the value of this variable.

Argument Intent

Information to the compiler can be given as to whether a dummy argument will:

- only be referenced -- `INTENT(IN)`;
- be assigned to before use -- `INTENT(OUT)`;
- be referenced and assigned to -- `INTENT(INOUT)`.

```
SUBROUTINE example(arg1,arg2,arg3)
     REAL, INTENT(IN) :: arg1
     INTEGER, INTENT(OUT) :: arg2
     CHARACTER, INTENT(INOUT) :: arg3
     REAL :: r
          r = arg1*ICHAR(arg3)
          arg2 = NINT(r)
          arg3 = CHAR(MOD(arg2,127))
END SUBROUTINE example
```

The use of `INTENT` attributes is recommended as it allows good compilers to check for coding errors, and facilitates efficient compilation and optimisation.

Note: if an actual argument is ever a literal, then the corresponding dummy argument must have the attribute `INTENT(IN)`.

If the intent of a dummy argument is `OUT` or `INOUT` then the corresponding actual argument must be a variable.

If a procedure changes the value of an argument then it is better for this procedure to be a subroutine rather than a function.

Local Objects

In the following procedure

```
SUBROUTINE Madras(i,j)
     INTEGER, INTENT(IN) :: i, j
     REAL                :: a
     REAL, DIMENSION(i,j):: x
```

`a` and `x` are known as *local objects*. They:

- are created each time the procedure is invoked;
- are destroyed when the procedure completes;
- *do not* retain their values between calls;
- do not exist in the program's memory between calls.

`x` could have a different size and shape on each call.

Scoping Rules

Fortran 95 is *not* a traditional block-structured language:

- the *scope* of an entity is the range of program unit where it is visible and accessible;
- internal procedures can inherit entities by *host association*;
- objects declared in modules can be made visible by *use association* (the `USE` statement, explained in the next chapter): useful for global data.

Host Association -- Global Data

Consider:

```
PROGRAM CalculatePay
     IMPLICIT NONE
     REAL      :: GrossPay, TaxRate, Delta
     INTEGER   :: NumberCalcsDone = 0
     GrossPay = ...; TaxRate = ... ; Delta = ...
     CALL PrintPay(GrossPay,TaxRate)
     TaxRate = NewTax(TaxRate,Delta)
     WRITE(unit=6,fmt="(a29,i2)") &
     "Number of calculations done =", NumberCalcsDone
CONTAINS
     SUBROUTINE PrintPay(Pay,Tax)
          REAL, INTENT(IN) :: Pay, Tax
          REAL :: TaxPaid
          TaxPaid = Pay * Tax
          WRITE(unit=6,fmt="(F8.3)") Pay - TaxPaid
          NumberCalcsDone = NumberCalcsDone + 1
     END SUBROUTINE PrintPay
     REAL FUNCTION NewTax(Tax,Delta)
          REAL, INTENT(IN) :: Tax, Delta
          NewTax = Tax + Delta*Tax
          NumberCalcsDone = NumberCalcsDone + 1
     END FUNCTION NewTax
END PROGRAM CalculatePay
```

`NumberCalcsDone` is a global variable and available in all procedures in this program by host association.

Scope of Names

Consider the following example:

```
PROGRAM Proggie
     IMPLICIT NONE
     REAL :: A=1.0, B, C
     CALL sub(A)
CONTAINS
     SUBROUTINE Sub(D)
          REAL :: D      ! D is dummy (alias for A)
          REAL :: C      ! local C (diff from Proggie's C)
          C = A**3       ! A cannot be changed
          D = D**3 + C   ! D can be changed
          B = C          ! B from Proggie gets new value
     END SUBROUTINE Sub
END PROGRAM Proggie
```

In `Sub`, as `A` is argument associated it may not have its value changed but may be referenced.

`C` in `Sub` is totally separate from `C` in `Proggie`, changing its value in `Sub` does not alter the value of `C` in `Proggie`.

SAVE Attribute

The `SAVE` attribute can be applied to a specified local variable in a procedure so that it and its value are not lost on return from the procedure. In the following example `NumInvocations` is initialised on first call and retains its new value between calls:

```
  SUBROUTINE Barmy(arg1,arg2)
    REAL, INTENT(IN)  :: arg1
    REAL, INTENT(OUT) :: arg2
    INTEGER, SAVE :: NumInvocations = 0
    NumInvocations = NumInvocations + 1
```

Variables with the `SAVE` attribute are *static* objects. Clearly, `SAVE` has no meaning in the main program.

Strictly the `SAVE` attribute in this example is not necessary because all variables with initial values acquire the `SAVE` attribute automatically.

Dummy Array Arguments

There are two main types of dummy array argument:

- *explicit-shape* -- all bounds specified;
  `REAL, DIMENSION(8,8), INTENT(IN) :: expl_shape`

  The actual argument that becomes associated with an explicit-shape dummy must conform in size and shape.

- *assumed-shape* -- no bounds specified, all inherited from the actual argument;
  `REAL, DIMENSION(:,:), INTENT(IN) :: ass_shape`

  The actual argument that becomes associated with an assumed-shape dummy must conform in rank. An explicit interface *must* be provided.

- dummy arguments cannot be (unallocated) `ALLOCATABLE` arrays.

Assumed-shape Arrays

Should declare dummy arrays as assumed-shape arrays:

```
PROGRAM Main
     IMPLICIT NONE
     REAL, DIMENSION(40)       :: X
     REAL, DIMENSION(40,40)    :: Y, Z
     ...
     CALL gimlet(X,Y)
     CALL gimlet(X(1:39:2),Y(2:4,4:4))
     CALL gimlet(Y(1:39:2,1),Z(2:40:2,2:40:2))
CONTAINS
     SUBROUTINE gimlet(a,b)
          REAL, INTENT(IN) :: a(:), b(:,:)
          ...
     END SUBROUTINE gimlet
END PROGRAM
```

Note:

- the actual argument cannot be a vector subscripted array;
- the actual argument cannot be an assumed-size array;
- in the procedure, bounds begin at 1.

External Functions

In an earlier example we had a program with an internal function:

```
PROGRAM Main
     IMPLICIT NONE
     REAL :: x
          READ(*,*) x
          WRITE(unit=6,fmt="(F12.4)") Negative(x)
          ...
CONTAINS
     REAL FUNCTION Negative(a)
          REAL, INTENT(IN) :: a
          Negative = -a
     END FUNCTION Negative
END PROGRAM Main
```

Sometimes a function is defined outside the body of the program unit which uses it, *ie* it is external to that unit. For example:

```
PROGRAM Main
     IMPLICIT NONE
     REAL :: x
          READ(*,*) x
          WRITE(unit=6,fmt="(F12.4)") Negative(x)
          ...
END PROGRAM Main

REAL FUNCTION Negative(a)
     REAL, INTENT(IN) :: a
     Negative = -a
END FUNCTION Negative
```

So that the compiler may know about this object it is necessary to give both its type and the fact that it is external in the specification part of the program. There are two ways of doing this:

```
PROGRAM Main
    IMPLICIT NONE
    REAL :: x, Negative     ! specify type REAL
    EXTERNAL :: Negative    ! use EXTERNAL statement
        READ(*,*) x
        ...
```

Or alternatively:

```
PROGRAM Main
    IMPLICIT NONE
    REAL :: x
    REAL, EXTERNAL :: Negative! specify type REAL and use
                              ! EXTERNAL attribute
        READ(*,*) x
        ...
```

Subroutine or Function?

It is permissible to write a function which does more than calculate its result but if it also performs action such as altering the values of arguments, input or output operations these side-effects adversely affect optimization particularly on parallel processors. Some side-effects in procedures to be aware of include:

- if a function, it does not alter the value of any dummy argument: effectively each dummy argument could have intent IN: the name of the function behaves like a dummy argument which is initially undefined and has intent OUT;
- it does not alter any variable accessed by host or use association;
- it does not contain any local variable with the attribute SAVE;
- it does not perform any operation on an external file;
- it must not contain a STOP statement.

If it is necessary for a procedure to include any side-effect then a subroutine should be written instead of a function.

Practical Exercise 4

Question 1: Simple example of a Subroutine
Write a main program and an internal subroutine that returns, as its first argument, the sum of two real numbers.

Question 2: Simple example of a Function
Write a main program and an internal function that returns the sum of two real numbers supplied as arguments.

Question 3: Switch or Stick
Write a main program and an internal subroutine with two arguments that returns, as its first argument, the smaller of two real numbers and as its second argument, the other number.

Question 4: Standard Deviation
Write a program which contains an internal function that returns the standard deviation from the mean of an array of real values. Note that if the mean of a sequence of values (x_i , $i = 1, n$) is denoted by m then the standard deviation, s, is defined as:

$$s = \sqrt{\frac{\sum_{i=1}^{n} (x_i - m)^2}{(n-1)}}$$

[Hint: In Fortran 95 `SUM(X)` is the sum of the elements of `X`.]
To demonstrate correctness write out the standard deviation of the following numbers (10 of them):

5.0 3.0 17.0 -7.56 78.1 99.99 0.8 11.7 33.8 29.6

and also the following 14:

1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 12.0 13.0 14.0

The files `statsa` and `statsb` contain these two sets of real values preceded by the relevant count (see Exercise 3, Question 9).

Question 5: Save Attribute
Write a skeleton procedure that records how many times it has been called.

5. Modules and Derived Types

Plane Geometry Program

The following program defines a simple 3 sided shape and contains two internal functions:

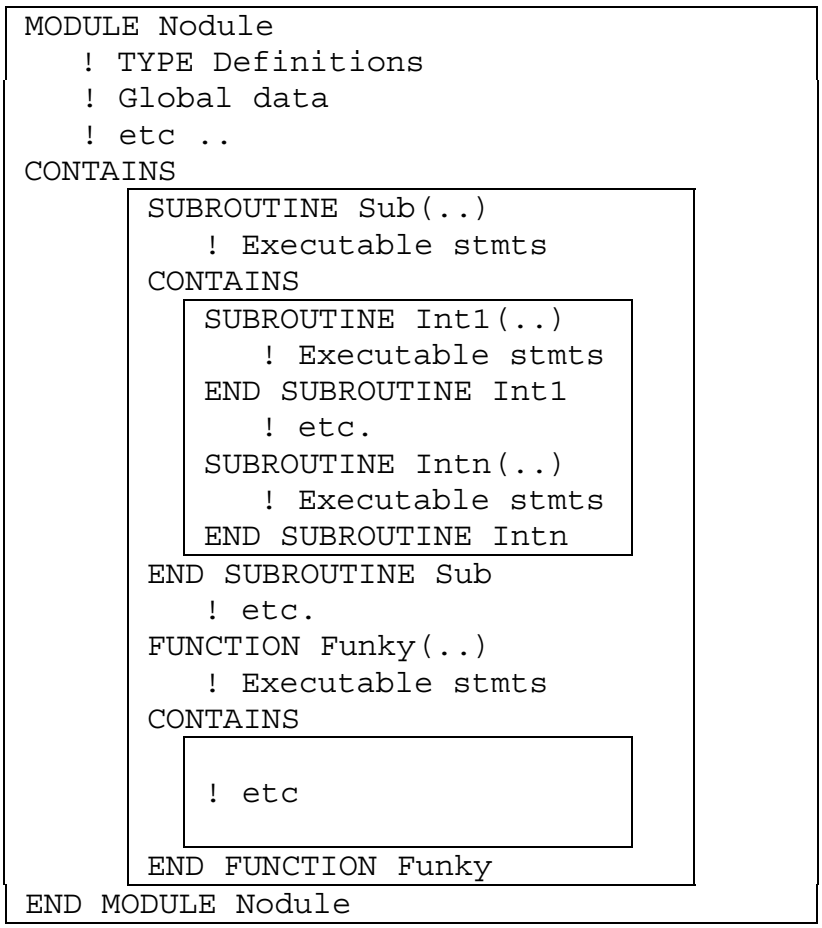
```
PROGRAM Triangle
     IMPLICIT NONE
     REAL :: a, b, c
     REAL, PARAMETER :: pi = 3.14159

     WRITE(unit=6,fmt="(a)") 'Welcome, please enter the &
     &lengths of the 3 sides.'
     READ(5,*) a, b, c
     WRITE(unit=6,fmt="(a16,f5.1)") &
     'Triangle''s area:', Area(a,b,c)

CONTAINS

FUNCTION theta(x,y,z)
     ! return the angle between sides x and y
     REAL :: theta              ! function type
     REAL, INTENT(IN) :: x, y, z
     ! check that sides do make a triangle
     IF (2*MAX(x,y,z) < (x+y+z)) THEN
          theta = ACOS((x**2+y**2-z**2)/(2.0*x*y))
     ELSE           ! sides do not make a triangle
          theta = 0.0
     END IF
END FUNCTION theta

FUNCTION Area(x,y,z)
     REAL :: Area               ! function type
     REAL, INTENT(IN) :: x, y, z
     REAL :: height
     height = x*SIN(theta(x,y,z))
     Area = 0.5*y*height
END FUNCTION Area

END PROGRAM Triangle
```

The main program can now call `Area` which uses 3 `REAL` values.

Reusability – Modules

To allow the constant `pi` and the function `Area` to be used elsewhere they should be contained in a `MODULE`. This is called encapsulation.

The general form of a module is:




```
MODULE < module name >
     < declarations and specifications statements >
[ CONTAINS
     < definitions of module procedures > ]
END [ MODULE [< module name > ] ]
```

The MODULE program unit provides the following facilities:

- global object declaration;
- procedure declaration (including operator definition);
- semantic extension;
- ability to control accessibility of above to different programs and program units;
- ability to package together whole sets of facilities.

Here is some of the code taken from the previous program example and encapsulated in a module:

```
MODULE Triangle_Operations
     IMPLICIT NONE
     REAL, PARAMETER :: pi = 3.14159
CONTAINS
FUNCTION theta(x,y,z)
     ! return the angle between sides x and y
     REAL :: theta              ! function type
     REAL, INTENT(IN) :: x, y, z
     ! check that sides do make a triangle
     IF (2*MAX(x,y,z) < (x+y+z)) THEN
          theta = ACOS((x**2+y**2-z**2)/(2.0*x*y))
     ELSE           ! sides do not make a triangle
          theta = 0.0
     END IF
END FUNCTION theta

FUNCTION Area(x,y,z)
     REAL :: Area               ! function type
     REAL, INTENT( IN ) :: x, y, z
     REAL :: height
          height = x*SIN(theta(x,y,z))
          Area = 0.5*y*height
     END IF
END FUNCTION Area

END MODULE Triangle_Operations
```

Other programs can now access Triangle_Operations. The USE statement attaches it to a program, and must precede any specification statements:

```
PROGRAM TriangUser
     USE Triangle_Operations
     IMPLICIT NONE
     REAL :: a, b, c, angle_rad, angle_deg

     WRITE(unit=6,fmt="(a)") 'Welcome, please enter the &
     &lengths of the 3 sides.'
     READ(5,*) a, b, c
     WRITE(unit=6,fmt="(a16,f5.1)") &
     'Triangle''s area:', Area(a,b,c)
     angle_rad = theta(a,b,c)
     angle_deg = angle_rad*180.0/pi
     WRITE(unit=6,fmt="(a,f6.2,a,f6.2,a)") "Angle is", &
     angle_rad, " radians or", angle_deg, " degrees"

END PROGRAM TriangUser
```

It is as if the code had been included in `TriangUser`.

Points raised:

- within a module, functions and subroutines are called *module procedures*;
- module procedures may contain internal procedures (like `PROGRAM`s);
- module objects which retain their values should be given the `SAVE` attribute;
- modules can also be `USE`d by procedures and other modules;
- modules can be compiled separately. They must be compiled **before** the program unit that uses them.

Restricting Visibility

In the example, the main program has access to `theta` and `Area`. You can prevent this by assigning visibility attributes:

```
     PRIVATE :: theta                    ! hidden
     PUBLIC  :: Area                     ! not hidden
```

`theta` is hidden, `Area` is not.

This allows `Area` to use `theta` within the module and for there to be a distinct object named `theta` defined outside this module which could be used in the program. Alternatively, you could use statements or attributes:

```
     PUBLIC                          ! set default visibility
     PRIVATE :: theta                ! hidden
     REAL, PRIVATE :: height         ! hidden
```

so, in the main `PROGRAM`:

```
     abc = Area(3,4,5)               ! OK
     height = 2.9                    ! Forbidden
```

The USE Renames Facility

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

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

Module entities can be renamed:

```
     USE Triangle_Operations, Space => Area
```

The module object `Area` is renamed to `Space` when used locally.

USE ONLY Statement

Another way to avoid name clashes is to only use those objects which are necessary. It has the following form:

```
USE < module-name > [, ONLY: < only-list >...]
```

The < *only-list* > can also contain renames (=>). For example:

```
     USE Triangle_Operations, ONLY : pi, &
          Space => Area
```

Only `pi` and `Area` are made accessible: `Area` is renamed to `Space`.

The `ONLY` statement gives the compiler the option of including only those entities specifically named.

Derived Types

It is often advantageous to express some objects in terms of aggregate structures, for example: coordinates, (`x`, `y`, `z`). Fortran 95 allows compound entities or *derived* types to be defined:

```
     TYPE COORDS_3D
          REAL :: x, y, z
     END TYPE COORDS_3D
     TYPE(COORDS_3D) :: pt1, pt2
```

Derived types definitions should be placed in a `MODULE`.

Previously defined types can be used as components of other derived types. These are sometimes known as supertypes:

```
     TYPE SPHERE
          TYPE (COORDS_3D)   :: centre
          REAL               :: radius
     END TYPE SPHERE
```

Objects of type SPHERE can be declared:

```
TYPE (SPHERE) :: bubble, ball
```

Values can be assigned to derived types either component by component or as an object.

An individual component may be selected by using the % operator:

```
pt1%x = 1.0
bubble%radius = 3.0
bubble%centre%x = 1.0
```

The whole object may be selected and assigned to by using a constructor:

```
pt1 = COORDS_3D(1.,2.,3.)
bubble%centre = COORDS_3D(1.,2.,3.)
bubble = SPHERE(bubble%centre,10.)
bubble = SPHERE(COORDS_3D(1.,2.,3.),10.)
```

The derived type component of SPHERE must also be assigned to by using a constructor. Note however, that assignment between two objects of the same derived type is intrinsically defined:

```
ball = bubble
```

Derived type objects, which do not contain pointers (or private) components, may be input or output using normal methods:

```
WRITE(unit=6,fmt="(4F8.3)") bubble
```

is exactly equivalent to:

```
WRITE(unit=6,fmt="(4F8.3)") bubble%centre%x, &
     bubble%centre%y, bubble%centre%z, &
     bubble%radius
```

Derived types are handled on a component by component basis. Their definitions should be packaged in a MODULE.

```
MODULE VecDef
     TYPE vec
          REAL :: r
          REAL :: theta
     END TYPE vec
END MODULE VecDef
```

To make the type definitions visible, the module must be used:

```
PROGRAM Up
     USE VecDef
     IMPLICIT NONE
     TYPE(vec) :: north
     CALL subby(north)
     ...
CONTAINS
     SUBROUTINE subby(arg)
          TYPE(vec), INTENT(IN) :: arg
          ...
     END SUBROUTINE subby
END PROGRAM Up
```

Type definitions can only become accessible by *host* or *use* association.

Functions can return results of an arbitrary defined type

```
FUNCTION Poo(kanga, roo)
     USE VecDef
     TYPE (vec) :: Poo
     TYPE (vec), INTENT(IN) :: kanga, roo
     Poo =...
END FUNCTION Poo
```

Recall that the definitions of `VecDef` must be made available by use or host association.

True Portability

The range and precision of the values of numeric intrinsic types are not defined in the language but are dependent upon the system used for the program. There are intrinsic integer functions for investigating these:

```
INTEGER  :: I, PR, RI, RR
REAL     :: X
RI = RANGE(I)
RR = RANGE(X);   PR = PRECISION(X)
```

If `RI` has the value `9` this means that any integer `n`
in the range $-999999999 \leq n \leq 999999999$ can be handled by the program.

If `RR` has the value `37` and `PR` has the value `6` this means that any real value in the range 10^{-37} to 10^{37} can be handled in the program with a precision of 6 decimal digits. As values of type complex consist of ordered pairs of values of type real, similar values would be returned by the inquiry functions if their arguments were of type complex instead of type real.

Intrinsic types can be parameterised by the `KIND` value to select the accuracy and range of the representation.
For type integer the function `SELECTED_INT_KIND` with a single argument of type integer giving the desired range will return the appropriate `KIND` value.

```
INTEGER, PARAMETER   :: ik9 = SELECTED_INT_KIND(9)
INTEGER(KIND=ik9)    :: I
```

If the given range can be supported, then the KIND value will be non-negative: a value of -1 indicates that the range is not supported.

For type real the function SELECTED_REAL_KIND with two arguments of type integer giving the desired precision and range will return the appropriate KIND value.

```
INTEGER, PARAMETER   :: rk637 = SELECTED_REAL_KIND(6,37)
REAL(KIND=rk637)     :: X
```

If the given precision and range can be supported, then the KIND value will be non-negative. A value of -1 indicates that insufficient precision is available, a value of -2 indicates that insufficient exponent range is available, and -3 indicates that neither is attainable.

Constants should have their KIND value attached:

```
INTEGER(KIND=ik9)       :: I=1_ik9
REAL(KIND=rk637)        :: X=2.0_rk637
COMPLEX(KIND=rk637)     :: C=(3.0_rk637,4.0_rk637)
I = I + 5_ik9
X = X + 6.0_rk637
C = C + (7.0_rk637,8.0_rk637)
```

You should make KIND value constants global by defining them in a module.

Practical Exercise 5

Question 1: Encapsulation

Define a module called `Simple_Stats` which contains encapsulated functions for calculating the mean and standard deviation of an arbitrary length `REAL` vector. The functions should have the following interfaces:

```
REAL FUNCTION mean(vec)
     REAL, INTENT(IN), DIMENSION(:) :: vec
END FUNCTION mean

REAL FUNCTION Std_Dev(vec)
     REAL, INTENT(IN), DIMENSION(:) :: vec
END FUNCTION Std_Dev
```

[Hint: In Fortran 95, `SIZE(X)` gives the number of elements in the array `X`.]
You may like to utilise your earlier code as a basis for this exercise.
Add some more code in the module, which records how many times each statistical function is called during the lifetime of a program. Record these numbers in the variables: `mean_use` and `std_dev_use`.
Demonstrate the use of this module in a test program; in one execution of the program give the mean and standard deviation of the following sequences of 10 numbers:

5.0 3.0 17.0 -7.56 78.1 99.99 0.8 11.7 33.8 29.6

and then the following 14:

1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 12.0 13.0 14.0

Write out the values of `mean_use` and `std_dev_use` for this run of the program.

Question 2: Binary Cut

Write a module containing a function which returns the position of a particular number in an array of sorted integers. The function should employ the so-called ``binary cut'' method. This method proceeds by determining in which half the number is and then concentrating on that half. It is easily implemented by using two indices to point at the low and high positions of the current area of interest. It is assumed that if there is more than one occurrence of the number then the one with the higher index will be chosen. This method is very efficient for very large arrays.

Algorithm:

- Let `i` and `j` be the indices of the low and high marks.
- Initially set `i = 1` and `j = n` (the number in the list)
- Assume `k` is the number we are trying to find
- `DO`
- `IF(i ≥ j)EXIT`
- determine the half way point $ihalf = \frac{i+j}{2}$
- `IF k` is above `ihalf` put `i = ihalf + 1`
- Otherwise put `j = ihalf`
- `END DO`
- `j` will now point at the number `k`

Question 3: Spheres Apart

Write a program to look at the relationship between all pairs of an arbitrary number of spheres defined in 3-dimensional space. Read in the number of spheres being used and read the coordinates of the centres and the lengths of the radii of these spheres into an allocatable array of a defined type variable. For spheres s_m and s_n the separation of their centres is given by the formula:

$$\sqrt{(x_m - x_n)^2 + (y_m - y_n)^2 + (z_m - z_n)^2}$$

If the centre of one sphere lies within the other then output a line stating this fact. Otherwise are the surfaces of the two spheres touching, intersecting or separate? You could try your program on spheres with these centres and radii:
(3.0,4.0,5.0), 3.0 (10.0,4.0,5.0), 4.0 (3.0,-3.0,5.0), 5.0 (3.0,4.0.8.0), 6.0

Question 4: Real Portability

Take a copy of the program you wrote in Question 3 of Exercise 2 to find the Ludolphian number. Replace the statement of the form:

```
REAL :: a, b, c, d, e, f
```

by the statements of the form:

```
INTEGER, PARAMETER :: k = SELECTED_REAL_KIND(P=15,R=31)
REAL(KIND=k)       :: a, b, c, d, e, f
```

Add a statement to check that `k > 0`, and change the kind of the constants to `k`, for example `1.0_k` Output the results with 12 decimal digits.

Run this program and compare the results with those you got earlier.

Question 5: Integer Portability

Take a copy of the program you wrote in Question 5(a) of Exercise 2 to find the first 5 terms of a sequence. Extend the range of those integers necessary to find the 6^{th} term of this sequence.

6. Bibliography

Fortran95/2003 explained
Michael Metcalf, John Reid, Malcolm Cohen.
Oxford University Press
ISBN 0 19 852693 8

A formal definition of the language.

Fortran 90 Programming
T.M.R.Ellis, Ivor R.Philips, Thomas M.Lahey
Addison-Wesley
ISBN 0-201-54446-6

A full explanation of the language.

Fortran 90/95 for Scientists and Engineers
Stephen J.Chapman
McGraw Hill
ISBN 007-123233-8

Contains good examples.