

ICHEC

Mr. Alastair McKinstry, Mr Alin Elena & Dr. Ruairi Nestor,
ICHEC Computational Scientists



An Introduction to Fortran

Fortran is a computer programming language that is popular for numerically intensive tasks. Fortran was originally developed in the 1950s and continues to be developed today. The most recent Fortran standard is Fortran 2003. This document is intended to serve as a brief introduction and/or a quick reference to the features of modern Fortran.

A simple Fortran program

The following is a simple Hello World program written in Fortran. This example illustrates the structure of a basic Fortran program and how it should be compiled and run. The file is saved as a text file named hello.f90.

```
program helloWorld
! This is a comment - it doesn't do anything
! Declare an integer variable called i
integer :: i
real :: a ! Declare a real variable a

i = 5 ! Assign values to i and a
a = 3.142

! Print a message to the screen
write (*,*) 'Hello World'
write (*,*) 'This is an integer: ',i
write (*,*) 'This is a real: ',a
end program helloWorld
```

On ICHEC systems, this program can be compiled into an executable by firstly loading the Fortran compiler:

```
user@stokes:> module load intel-fc
```

and then compiling the program into an executable named hello:

```
user@stokes:> ifort hello.f90 -o hello
```

An Introduction to Fortran

A simple fortran program	1
Variables, data types and operators	2
Implicit none statement & Operators	2
Flow control & Loops	3
Subprograms, Functions & Subroutines	4
Modules & Procedure overloading	5
Arrays & Input and Output	6
File I/O	8
Obsolete features	10

To run the program, type:

```
user@stokes:> ./hello
```

The program will produce output similar to the following:

```
Hello World
This is an integer: 5
This is a real: 3.1420000
```

Fortran files are identified by a file extension, which tells the compiler what type of file it is. Older Fortran files were fixed form, meaning that some columns of the file were reserved for certain characters or statements. Fixed form files usually have the one of the following extensions: .f, .f77 or .for. Modern versions of Fortran permit the use of free form files, which place no restrictions on column usage. All modern source files should be written in free form, and saved with the extension .f90 or .f95.

Variables, data types and operators

In modern Fortran, five intrinsic data types are available in addition a user may define their own data types.

Logical

Logical variables can have only have one of two

```
logical :: a,b
a = .true.
b = .false.
```

values: .true. or .false.:

Character

A character variable can store a single character. For

```
!a can store a single character
character :: a
!b can store a string of max 20 characters
character(len=20) :: b
a = 'R'
b = 'Hello'
```

storing a string of characters, it is possible to specify a length:

Numeric data types

Three numeric data types are available in Fortran: integer, real and complex. An optional kind argument may be supplied to an integer declaration, which allows the programmer to set the range of the integer variable. For real or complex data types, the precision and range of the variable may be set using the "selected real kind" function:

```
! 'double' is a parameter: a new value cannot be
! assigned to it later
integer, parameter :: double=selected_real_kind(15 ,307)
! i is in the range -10^8 to 10^8
integer(kind=8) :: i
! a has 15 digits precision and exponent range +/-30
real(double) :: a
complex (double) :: c
```

User-defined data types

The basic data types are often inadequate for describing more complex data. For these situations, Fortran allows the programmer to create their own types. The following program illustrates the creation of a user-defined type to store information about a person:

```
program userType
  implicit none
  !Definition of a new data type called 'personType'
  type :: personType
    character (len=100) :: name
    integer :: age
    real :: weight
  end type personType
  ! Declare an instance of personType named A
  type (personType) :: A
  ! The individual elements of A are accessed
  ! assigned using the \% operator
  A%name = 'John Joe'
  A%age = 45
  A%weight = 65.0
  print*, A%name
end program userType
```

Implicit none statement

By default, Fortran allows variables to be used without declaration. One undesirable side effect of this behaviour is that spelling errors in variable names may not be found by the compiler. As a result, difficult-to-find runtime errors can occur. Therefore, it is considered good programming practice to include the implicit none statement at the start of each source file. This forces the programmer to explicitly declare all variables.

Operators

• The arithmetic operators available in Fortran are summarised in the following table:

Arithmetic operators in Fortran	
Operator	Description
$a + b$	Add a and b
$a - b$	Subtract b from a
$a * b$	Multiply a and b
a / b	Divide a by b
$a ** b$	Raise a to the power of b

Logic operators are also available, the result of which is `.true.` or `.false.` depending on the values of their operands:

Logic operators in Fortran	
Operator	Description
<code>a == b</code>	Equal to
<code>a /= b</code>	Not equal to
<code>a > b</code>	Greater than
<code>a >= b</code>	Greater than or equal to
<code>a < b</code>	Less than
<code>a <= b</code>	Less than or equal to

Note the distinction between the relational operator (`=`) and the equal-to logic operator (`==`).

Flow control

Like other programming languages, Fortran offers constructs which allow the programmer to control the flow of the program as required. Both conditional and looping constructs are available. If statements are used when a block of code should or should not be executed, depending on the evaluation of a logical expression. For example, suppose we have a variable, which if negative causes a message to be printed to the screen:

```
if (a < 0) then
  write (*,*) 'a is less than zero'
endif
```

Now suppose we wish to extend the above and print a suitable message if `a` turns out to be zero or greater:

```
if (a < 0) then
  write (*,*) 'a is less than zero'
else
  ! a is not less than zero - it is zero or greater
  write (*,*) 'a is zero or greater'
endif
```

In the above example, the code inside the `else` block is executed only if the logical expression `(a < 0)` is false. There may be at most one `else` statement in an `if` block. To evaluate a second logical expression in addition to `(a < 0)` we use the `elseif` statement – continuing the above example, suppose we wish to print a message if `a` is exactly equal to zero:

```
if (a < 0) then
  write (*,*) 'a is less than zero'
elseif (a == 0) then
  ! a is exactly equal to zero
  write (*,*) 'a is equal to zero'
else
  ! a is definitely greater than zero
  write (*,*) 'a is greater than zero'
endif
```

There may be any number of `elseif` statements in an `if` block, and thus any number of logical expressions can be evaluated.

Loops

Loops are used when a block of code needs to be executed several times. In Fortran, there are two basic types of loop: while loops and iterative loops. While loops continue to repeat indefinitely until some condition is satisfied, whereas iterative loops repeat the contained block of code a specified number of times. The following is an example of a while loop in Fortran:

```
program loop
  implicit none
  integer :: i,n
  i=1
  n=5

  do
    write (*,*) i, i**2
    i = i+1 ! Increment counter i by 1
    if (i>n) then ! if i > n then quit the loop
      exit ! 'exit' causes the loop to terminate
    endif
  enddo
end program loop
```

Program Output:

```
1      1
2      4
3      9
4     16
5     25
```

An equivalent form of the above can be written as:

```
program loop
  implicit none
  integer :: i,n
  n=5
  i=1

  do while (i<=n) !Loop executes while i<=n
    write (*,*) i, i**2
    i = i+1 !Increment counter i
  enddo
end program loop
```

For the other type of loop, the iterative loop, the number of times the loop will execute is known beforehand. This type of loop is written in Fortran as

```
program loop
  implicit none
  integer :: i,n=5

  do i=1,n,1 !Loop from i=1 to i=n in steps of 1
    write (*,*) i, i**2
  enddo
end program loop
```

Each time the loop repeats, the index i increases by 1. This process is repeated until i reaches the maximum number of iterations n . This program produces the same output as the first two examples.

In the first loop example above, the exit statement was used when it was necessary to stop executing the code in the loop. 'Cycle' is another useful statement which may be used in do loops. When a cycle statement is encountered, the remaining code in the loop is skipped and the next loop iteration starts. For example:

```
program loop
  implicit none
  integer :: i,n=5

  do i=1,n,1 ! Loop from i=1 to i=n in steps of 1
    if (i == 3)
      cycle ! Skip iteration 3
    endif
    write (*,*) i, i**2
  enddo
end program loop
```

Fortran also offers another looping construct for certain operations the implied loop. Implied loops are a shorthand way of looping over an index in read, write or array definition statements. The general form of an implied loop is:

(object , index = beginning , end , increment)

As in iterative loops, the increment is optional and may be omitted. As an example, the following code prints a list of integers from 1 to n .

```
write (*,*) (i, i=1,n)
```

Implied loops can only be used for the following:

- Read, write and print procedures
- Setting array elements
- Variable definition

In addition, implied loops cannot be nested, i.e. one cannot be placed inside another.

Subprograms

In the design of larger programs, the typical approach is to divide the program into several smaller, more manageable tasks. In Fortran and other programming languages, this corresponds to creating a set of subprograms which make up over- all program. The two basic types of subprograms in Fortran are functions and subroutines. Variables declared within a function or subroutine are usually local, i.e. they are not visible to the main program or other subprograms

Functions

Functions are subprograms which take several arguments as input and return a single output value or array. The following example illustrates the use and definition of a user-defined function to compute the distance between a point (x,y) and the origin $(0,0)$.

```
program functionExample
  implicit none

  real :: a,b, result
  a = 1.5
  b = 2.0

  ! Call function "dist". The result is placed in
  ! the variable "distance"
  result = dist(a,b)
  write (*,*) result

!The subprograms are defined after "contains"
contains
  function dist(x,y)
    ! All arguments to the function must be defined
    ! at the start of the function
    ! The variable with the same name as the function
    ! is the returned value:
    real :: dist
    ! x,y are inputs and can't be modified in the
    ! function:
    real, intent(in) :: x,y

    dist = sqrt(x*x + y*y)
  end function dist
end program functionExample
```

```
Program Output :
  2.5000000
```

Subroutines

Fortran subroutines are similar to functions, except that they can return more than one variable or array to the calling program. Another difference is that the output values are returned via the arguments to the subroutine.

```

program subExample
  implicit none

  real :: a,b,result
  a = 1.5
  b = 2.0

  ! Subroutine is called using the "call" keyword
  call dist(result,a,b)
  write (*,*) result

  ! The subprograms are defined after "contains"
contains
  subroutine dist(r,x,y)
    ! Define all arguments at the top of the
    ! subroutine. r is the output: its value may be
    ! changed in the subroutine:
    real, intent(out) :: r
    ! x,y are inputs and can't be modified in the
    ! subroutine
    real, intent(in) :: x,y

    r = sqrt(x*x + y*y)
  end subroutine dist
end program subExample

Program Output :
2.5000000

```

Modules

Modules are program units which contain both the data and subprograms associated with a certain task. An example, shown below, is a vector module. This module defines a user type (vectorType) containing three variables: the number of components, the vector itself and the vector

```

module vector
  implicit none
  private ! private statement causes all data and
           ! subprograms to be hidden from calling
           ! program, unless otherwise specified

  ! Create a new type which will be visible to
  ! the calling program
  type,public :: vectorType
    ! The number of components in the vector
    integer :: n=3
    real :: a(3) ! The vector components
    real :: magnitude ! The vector magnitude
  end type vectorType

  ! Make subroutine magnitude public, i.e. visible
  ! to calling program:
  public :: magnitude

contains
  subroutine magnitude (x)
    type(vectorType), intent(inout) :: x
    integer :: i

    x%magnitude = 0.0
    do i = 1,x%n
      x%magnitude = x%magnitude + x*a(i)*x*a(i)
    enddo
    x%magnitude = sqrt(x%magnitude)
  end subroutine magnitude
end module vector

```

magnitude. A function is then de- fined which computes the magnitude of the array. Additional functionality could be added by including subprograms which compute the dot product of two vectors for example.

The module may be used in a program as follows:

```

program vectorTest
  ! make the data and functions in vector module
  ! available to this program
  use vector
  implicit none

  ! Create a new vector
  type(vectorType) :: v1

  ! Set some test values
  v1%a(1) = 1.0; v1%a(2) = 1.0; v1%a(3) = 0.0

  ! Compute the magnitude of v1.
  call magnitude(v1)
  write (*,*) 'Vector Magnitude:'
  write (*,*) (v1%magnitude)
end program vectorTest

Program Output:
Vector Magnitude:
1.4142135

```

Procedure overloading

Overloading means that several procedures, each with different arguments of different types, can be referred to using a single name. The compiler then decides which procedure should be called on the basis of the number and/or type of arguments supplied. Procedure overloading is useful because it promotes readability and brevity in the calling program. As an example, consider overloading the assignment operator (=) so that we can use this operator between two user-defined types (vectorType in this case):

```

module vector
  implicit none
  private

  type,public :: vectorType
    integer :: n=3
    real :: a(3)
    real :: magnitude
  end type vectorType

  ! Allow assignment operator to be used between
  ! vector types
  interface assignment(=)
    module procedure equalVectorType
  end interface

contains
  subroutine equalVectorType (left,right)
    type(vectorType),intent(in) :: right
    type(vectorType),intent(inout) :: left
    integer :: i

    left%n = right%n
    left%magnitude = right%magnitude
    do i = 1,right%n
      left%a(i) = right%a(i)
    enddo
  end subroutine equalVectorType
end module vector

```

This module can be used in the main program as follows:

```

program overloadTest
! make the data and functions in vector module
! available to this program
use vector
implicit none

! Create two new vectors
type(vectorType) :: v1, v2
integer :: i

! Set some test values
v1%a(1) = 1.0; v1%a(2) = 1.0; v1%a(3) = 0.0
v2%a(1) = 0.0; v2%a(2) = 0.0; v2%a(3) = 0.0

! Set v2 equal to v1
v2 = v1
write (*,*) (v2%a(i), i = 1,v2%n)
end program overloadTest

```

Program Output:

```

1.00000000    1.00000000    0.00000000

```

Arrays

An array is an indexed group of variables of the same type that are referred to by a single name. Arrays are useful for storing lists of related data. The individual elements of an array are accessed by using a subscript. The following Fortran example illustrates how to declare a 1D array with 3 entries and set each of its elements to zero:

```

integer :: i
real :: a(3) ! A 1D array with 3 entries

do i = 1,3 ! Loop over each entry and set to zero
  a(i) = 0.0
enddo

```

To define a 2D array with 3 rows and 3 columns and set all of its elements to zero:

```

! A 2D array with 3 rows and 3 columns
real :: a(3,3)

! Loop over each entry and set to zero
do i = 1,3 ! Loop over rows
  do j = 1,3 ! Loop over each column within the row
    a(i,j) = 0.0
  enddo
enddo

```

Arrays in Fortran can have a maximum of 7 dimensions. In most practical applications, the dimensions of an array will not be known before the program is run. If the array is used for storing a matrix for example, the dimensions of the matrix will probably vary from problem to problem. For this reason, it is possible to define dynamic arrays whose dimensions can be set or change while the program is running. This accomplished in Fortran as follows:

```

program dynamic
implicit none
real, allocatable :: matrix(:, :)
! Matrix is dynamic - will specify size below
integer :: components
integer :: rows, cols
integer :: info
integer :: i, j
rows = 3
cols = 3

! Allocate the required memory for this matrix
allocate(matrix(rows, cols), stat=info)
! Check status of allocation: Error if info not
! equal to 0
if (info /= 0) then
  write (*,*) 'Error in memory allocation'
  exit
endif

! Initialise all matrix elements to zero
do i=1,cols
  do j=1,rows
    matrix(j,i) = 0.0
  enddo
enddo

! Perform other computations here...

! Deallocate matrix - return memory to system
! First check if memory has been allocated to
! matrix
if (allocated(matrix)) then
  deallocate(matrix)
endif
end program dynamic

```

The status of `allocate()` should always be checked to ensure that the memory has been successfully allocated. Memory allocation can fail if for example an attempt is made to allocate more memory than is available on the system. In addition, allocated memory should always be deallocated so that the memory is returned to the system.

Input and output

In the examples shown previously, program output has been performed using `write(*,*)` statements. The first field inside the brackets (an asterisk in this case) of this write statement is called the output unit. Putting an asterisk in this field indicates that the output should be directed to standard output, i.e. the screen. The second field inside the brackets is the format specifier, which indicates how the resulting output will appear. An asterisk in this field indicates that the output should be written in free format, i.e. with default formatting. Reading input data is performed using the `read(*,*)` statement, which is analogous to `write(*,*)`. In this section, some of the possible values for the unit and format fields of read and write statements are described.

By providing a format specifier to the write function, the appearance of the resulting output can be controlled. Different format specifiers are used for different data types. Some of the possible format specifiers for integer output are described in the following example.

```

program formatIntegerOut
  implicit none
  integer :: i,j

  i = 12345678
  j = 10

  ! Print a ruler:
  ! The 'a' format descriptor indicates that a string will be printed:
  write (*,'(a)') "    5   10   15   20   25   30   35"
  write (*,'(a)') "----|----|----|----|----|----|----|"

  ! Field width of zero means that the field expands to fit the output
  ! The output is left-justified:
  write (*,'(i0)') i
  write (*,'(i0)') j

  ! write out two integers with one space between them:
  write (*,'(2(i0,1x))') i,j

  ! Specify a field width of 10: output will be right-justified:
  write (*,'(i10)') i

  ! write 2 integers with field width 10 and 5 spaces between them:
  ! Output of each will be right-justified:
  write (*,'(2(i10,5x))') i,j

  ! Specify a field width that is too small: Outputs asterisks
  write (*,'(i2)') i

  ! Print another ruler:
  write (*,'(a)') "----|----|----|----|----|----|----|"
  write (*,'(a)') "    5   10   15   20   25   30   35"
end program formatIntegerOut

```

Program Output:

```

    5   10   15   20   25   30   35
----|----|----|----|----|----|----|
12345678
10
12345678 10
 12345678
12345678          10
**
----|----|----|----|----|----|----|
    5   10   15   20   25   30   35

```

For floating point numbers, a couple of different choices are possible for the output format. These include floating point and exponential notation. Some sample format specifiers for real numbers are given in the following example:

```

program formatRealOut
  implicit none
  real :: a,b

  a = 21.2345
  b = 125.1234

  ! Print a ruler:
  write (*,'(a)') "    5   10   15   20   25   30   35"
  write (*,'(a)') "----|----|----|----|----|----|----|"

  ! 1: Total field width of 10, 4 digits after the decimal point
  write (*,'(f10.4)') a

  ! 2: Expand field width to fit the number, 8 digits after decimal point
  write (*,'(f0.4)') a

  ! 3: 3 digits after decimal point: Note how the output is rounded
  write (*,'(f0.3)') a

  ! Next format some numbers in exponential notation
  ! 4: e format specifier gives a number between 0.1 and 1 times a power of 10
  ! This example prints a number with total field width 10
  write (*,'(e10.4)') a

```

```

! 5: es format specifier gives a number between 1.0 and 10.0 times a power of 10
! This example prints two values separated by 5 spaces:
write (*,'(2(es10.4,5x))') a,b

! 6: field width too small - prints asterisks
write (*,'(es5.4)') a

! Print another ruler:
write (*,'(a)') "----|----|----|----|----|----|"
write (*,'(a)') "    5  10  15  20  25  30  35"
end program formatRealOut

```

Program Output:

```

    5  10  15  20  25  30  35
----|----|----|----|----|----|
 21.2345
21.2345
21.235
0.2123E+02
2.1235E+01    1.2512E+02
*****
----|----|----|----|----|
    5  10  15  20  25  30  35

```

File I/O

In many practical applications, program input and output are performed by reading from or writing to files. Before a file can be accessed from a Fortran program, it must be opened. The following example shows how a plain text file named out.txt may firstly be opened for writing and subsequently opened for reading:

```

program fileIO
  implicit none
  integer :: istat
  real :: a,b
  a=3.142

  ! Open a file called out.txt for writing. Its unit number is 101
  open(101, file="out.txt", status="unknown", action="write", form="formatted", IOSTAT=istat)

  ! Check the file was opened successfully:
  if (istat /= 0) then
    write (*,*) "Error opening file"
    exit
  endif
  write (101, '(f0.3)') a

  ! Once finished with the file it should be closed:
  close(101)

  ! Open the file for reading - a different unit number (102) is used,
  ! but this is not necessary since we have closed the file
  ! associated with unit 101
  open(102, file="out.txt", status="old", action="read", form="formatted", IOSTAT=istat)
  ! Check the file was opened successfully:
  if (istat /= 0) then
    write (*,*) "Error opening file"
    exit
  endif

  ! read the value from file and assign the value to b
  ! note that a total field width (5 in this case) must be
  ! supplied to the read function:
  read (102, '(f5.3)') b
  write (*,'(f5.3)') b

  close(102)
end program fileIO

```

Program Output

```
3.142
```

Arguments to the open function with possible values and resulting behaviour		
Status	old	File must already exist, otherwise open will fail with an error.
	new	File must not already exist, otherwise open will fail with an error.
	unknown	File may or may not exist. This is the default behaviour.
Action	write	File is opened for writing only. Subsequent attempts to read from the file result in an error
	read	File is opened for reading only. Subsequent attempts to write to the file result in an error.
	readwrite	File is opened for reading or writing. This is the default behaviour.
Form	formatted	ASCII (text) output (human readable).
	unformatted	Binary output (not human readable).

The following example shows how a binary file may be written in Fortran:

```

program fileIOBin
  implicit none
  integer :: istat
  real :: a,b

  a=3.142

  ! Open a file called out.dat for writing. Its unit number is 101
  open(101, file="out.dat", status="unknown", action="write", form="unformatted",
  IOSTAT=istat)
  ! Check the file was opened successfully; if not quit the program (exit):
  if (istat /= 0) then
    write (*,*) "Error opening file"
    exit
  endif
  ! Note: In unformatted output mode, there should be no format specifier
  write (101) a

  ! Once finished with the file it should be closed:
  close(101)

  ! Open the binary file for reading
  open(101, file="out.dat", status="unknown", action="read", form="unformatted",
  IOSTAT=istat)
  if (istat /= 0) then
    write (*,*) "Error opening file"
    exit
  endif
  read(101,iostat=istat) b
  write (*,*) b ! Write the variable to the screen

  close(101)
end program fileIOBin

```

Program Output:
3.1420000

Obsolete Features

Because of its long history, modern Fortran standards retain features that were designed with old programming methodologies and hardware in mind. The use of these features in modern Fortran is discouraged. They are included here because they may be encountered in the modification or maintenance of older Fortran source code. In this section, some of these obsolete features are described and modern replacements are suggested.

Common Blocks

Common blocks are a feature that allow a variable or variables to be allocated to consecutive regions in memory. For example, it is possible to declare a common block named temperature in a main program, the contents of which will be accessible to program units declaring a common block with the name temperature:

```
program common
  implicit none
  real :: a
  real :: b(100)
  common /temperatures/ a, b

  a = 10

  call test()
  write (*,*) a

  contains
    subroutine test()
      real :: c
      real :: d(100)
      common /temperatures/ c, d

      c = 20
    end subroutine test
end program common
```

Program Output:
20.000000

Common blocks may be problematic because they are associated with memory rather than with data types. Suppose that, by mistake, the common block in the subroutine test() was declared as:

```
real :: c
real :: d(100)
common /temperatures/ d, c
```

Then the variable d(1) would actually refer to the same memory location as the variable a in the main program. This is most likely not the desired behaviour. However, an error would not be generated by the compiler because the variables are associated with memory locations only, not data types. If it's necessary to share data in the manner facilitated by common blocks, modules with public data should be used instead:

```
module temperatures
  implicit none
  public
  real :: a
  real :: b(100)
end module temperatures

!In the main program:
program commonReplace
  use temperatures
  implicit none

  a = 20
  write (*,*) a
end program commonreplace
```

Go To

The go to statement has the form `go to <label>` where <label> is an integer appearing at the start of an executable Fortran statement. In the past, go to statements were combined with if statements to create program branches. For example:

```
program goto
  implicit none
  integer :: i

  i = 1

  if (i /= 0) go to 100
  go to 110

  100 write (*,*) 'i does not equal zero'
  105 go to 120
  110 write (*,*) 'i equals zero'
  120 write (*,*) 'End Program'

end program goto
```

Program Output:
i does not equal zero
End Program

Excessive use of the go to statements makes it difficult to follow the flow of logic in the code, and their use should therefore be avoided where possible. The go to statements in the above example could be replaced with an if/else block, as follows:

```
program gotoReplace
  implicit none
  integer :: i

  i = 1

  if (i /= 0) then
    write (*,*) 'i does not equal zero'
  else
    write (*,*) 'i equals zero'
  endif

  write (*,*) 'End Program'

end program gotoReplace
```

Statement Functions

Statement functions are a shorthand way of writing functions. For example, to define a function that returns the square of a number:

```
real :: x1, x2, y1, y2
quad(x) = x**2

x1 = 3.0
x2 = 4.0

y1 = quad(x1)      !y1 = 3.0**2
y2 = quad(x2)      !y2 = 4.0**2
```

Statement functions have been dropped in recent Fortran standards because they can be easily confused with array assignments.

Arithmetic If

The arithmetic if statement takes the following form:

```
integer :: p, q
...
...
if (p-q) 10, 20, 30
10 <Code for the case when (p-q) is negative>
20 <Code for the case when (p-q) equals zero>
30 <Code for the case when (p-q) is positive>
```

In modern Fortran programs, the logical block if structure should be used instead of the arithmetic if statement.

Obsolete Do Loop Structures

Before the Fortran 90 standard, Do loops had no End Do statement and were instead supplemented by a label. The following is an example of an early Fortran Do loop:

```
do 100 i = startIndex, endIndex
a(i) = 0.0
100 b(i) = 0.0      ! Loop end at label 100
```

An equivalent, more readable form of the above is the following:

```
do 100 i = startIndex, endIndex
  a(i) = 0.0      ! Note: Indented code
  b(i) = 0.0
100 continue
! Loop end at label 100, continue statement does
! nothing
```

It was also possible to share labels between nested loops:

```
do 100 i = istartIndex, iendIndex
  do 100 j = jstartIndex, jendIndex
    a(i,j) = 0.0
    b(i,j) = 0.0
  100 continue
! Both loops end at label 100, continue statement
! does nothing
```

Furthermore, in the Fortran 77 standard it was possible to have real numbers as indices for loops. These do loop forms are confusing and should not be used in modern Fortran programs.



References

S. J. Chapman,
Fortran 95/2003 for Scientists and Engineers,
McGraw-Hill, 3rd Ed., 2008.

M. Metcalf, J. Reid and M. Cohen,
Fortran 95/2003 Explained,
Oxford University Press, 3rd Ed., 2004.