Introduction to
Scientific & Engineering Computing
BIL 102FE (Fortran) Course
for
Week 5

Dr. Ali Can Takinacı

Assistant Professor

in

The Faculty of Naval Architecture and Ocean Engineering

80626

Maslak – Istanbul – Turkey

CONTROLLING THE FLOW IN A PROGRAM

the concept of comparison
between two numbers or two character strings
explanations how such comparisons can be used to determine which of two, or more , alternative sections of the code obeyed.

Choice and decision-making

All the programs up to now have started execution at the beginning of the main program, and have then proceed to execute each statement one by one in turn, until the last statement of the main program executed

F uses the words *if* and *then* to alter the sequential process.
This structure is known as an *if construct*.
The way an *if* construct works is that each decision criterion is examined in turn.
If it is true the following action or “block” of F statements is executed.
If it is not true then the next criterion (if any) is examined.
If none of the criterion is found to be true the block of statements following the *else* (if there is one) is executed.

An if structure

```
if (criterion_1) then
action_1
else if (criterion_2) then
action_2
else if (criterion_3) then
action_3
else
action_3
end if
```

If there is no *else* statement then no action is taken and the program flow goes to the next statement.

```
if (criterion) then
action
end if
```

Logical expressions and logical variables

The values *true* and *false* are called *logical values.*

*A*n expression, which can take one of these two values, is called a logical expression.

The simplest forms of logical expressions are those expressing the relationship between two numeric values
thus

a > b

is true if the value of *a* is greater then the value of *b*, and

x = = y

is true if the value of x equal to the value of y.

The six relational operators exist in F, which express a relationship between two values.

```
a  <   b is true if a is less than b
a  <=  b is true if a is less than or equal to b
a  >   b is true if a is greater than b
a  >=  b is true if a is greater than or equal to b
a  ==  b is true if a is equal to b
a  /=  b is true if a is not equal to b
```

The two possible logical variables in F are written as:

```
.true.
.false.
```

The logical operators .*or*. and .*and*. are used to combine two logical expressions or values.

The effect of .*or*. gives a true result if *either* of its operand is true.

The effect of .*and*. gives a true result only if *both* are true.

L1	L2	L1 .or. L2	L1 .and. L2
true	*true*	*true*	*true*
true	*false*	*true*	*false*
false	*true*	*true*	*false*
false	*false*	*false*	*false*

Two other logical operators exist in F.

The first of these (.*eqv*.) gives a true result if its operands are *equivalent* (that is, if they both have the same logical value).

The other (.*neqv*.) is the opposite (*not equivalent*) and gives a true result if they have opposite logical values.

L1	L2	L1 .eqv. L2	L1 .neqv. L2
true	*true*	*true*	*false*
true	*false*	*false*	*true*
false	*true*	*false*	*true*
false	*false*	*true*	*false*

Essentially

these operators are used in logical expressions to simplify their structure.

the following two expressions are identical in their effect.

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

There is one further logical operator *.not.*, which unlike all the other relational and logical operator.

The *.not.* operator, that is a unary, inverts the value of the following logical expressions and has a single operand.

If the logical expression is *true* then logical expression applied to *.not.* is *false*. The following expressions are equivalent in their effect.

```
.not. (a<b .and. b<c) <==>  a>=b .or. b>=c
```

or

```
.not. (a<b .eqv. x<y) <==>  a<b .neqv. x<y
```

The if construct

The initial statement of the construct is an *if* statement which consists of the word *if* followed by a logical expression enclosed in parentheses, followed by the word *then*:

```
if (logical_expression) then
```

This is followed by a block of statements, which will be executed only if the logical expression is true. The block of statements is terminated by an *else if* statement, or an *end if* statement.

The *else if* statement has a very similar syntax to that of an *if* statement:

```
else if (logical_expression) then
```

It is followed by a block of statements which will be executed if the logical expression is true, and if the logical expression in the initial if statement, and those of any preceding *else if* statements, are false.

The block of statements is terminated by another *else if* statement, an *else* statement, or an *end if* statement.

The *else* statement introduces a final block of statements, which will be executed only if the logical expressions in all preceding *if* and *else if* statements are false.

Finally, the *end if* statement terminates the *if* construct.

```
if (logical_expression) then
   block of F statements
else if (logical_expression) then
   block of F statements
else if (logical_expression) then
      .
      .
      .
      else
block of F statements
end if
```

<u>Example</u>: Write a function, which will return the cube root of its argument.
In section 4.3 a function to meet this requirement, which was only valid for positive argument, was written. If the argument is negative the relation can be used. The zero argument situation is however, slightly more complicated, since it is not possible to calculate the logarithm of zero.

Structure plan

```
Function cube_root(x)
if x = 0
Return zero
else if x < 0
Return -exp(log(-x)/3)
else
Return  exp(log(x)/3)
end if
```

```fortran
      program test_cube_root
      real :: x
      print *," type real positive number"
      read *, x
      print *,"the cube root of x=",x," is",cube_root(x)
      end program test_cube_root
      function cube_root(x) result(root)
!     Dummy argument declaration
      real::x
!     Result variable declaration
      real::root
!     Local variable declaration
      real::log_x
!     eliminate the zero case
      if(x==0.0) then
      root = 0.0
!     Calculate cube root by using logs
      else if (x<0.0) then
!     first deal with negative arguments
      log_x = log(-x)
      root = -exp(log_x/3.0)
      else
      log_x = log(x)
      root = exp(log_x/3.0)
      end if
!     positive argument

      end function cube_root
```

<u>Comparison of character string</u>

Character strings are compared with the rule of the *collating sequence* of letters, digits, and other characters, which is based on the order of these characters in the <u>A</u>merican <u>N</u>ational <u>S</u>tandard <u>C</u>ode for <u>I</u>nformation <u>I</u>nterchange (ASCII).

F lays down six rules for this, covering letters, digits and the space or blank character.

1. The 26 upper-case letters are collated in the following order:

A B C D E F G H I J K L M N O P Q R S T U W Y Z

2. The 26 lower-case letters are collated in the following order:

a b c d e f g h i j k l m n o p q r s t u w y z

3. The 10 digits are collated in the following form:

0 1 2 3 4 5 6 7 8 9

4. A space (or blank) is collated before both letters an digits

5. Digits are all collated before the letter A.

6. Upper-case letters are all collated before any lower-case letters.

The position in the collating sequence of the other 22 characters in the F character set is determined by their position in the ASCII collating sequence.

1. If the two operands are not the same length the shorter one is treated as though it were extended on the right with blanks until it is the same length as the longer one.

- “Adam” > “Eve” is *false* because A comes before E

2. The two operands are compared character by character, starting with the left-most character, until either a difference is found or the end of the operands is reached

- “Adam” < “Adamant” is *true* because after Adam has been extended the relationship reduces to “ “ < “a” after the first four characters have been found to be the same. Since a blank comes before a letter, this is *true*

3. If a difference is found, then the relationship between the two operands with the character, which comes earlier in the collating sequence being deemed to be lesser of the two. If no difference is found then the strings are considered to be equal.

- “120” < “1201” is *true* because the first difference in the string leads to an evaluation of “ “< “1”, which is *true* since a blank also comes before a digit.

- “ADAM” < “Adam” is *true* because the first difference in the strings leads to an evaluation of “D”<”d”, which is *true* since upper-case letters come before lower-case letters.

Example 1: Write a function which takes a single character as its argument and returns a single character according to the following rules:

- If the input character is a lower-case letter then return its upper-case equivalent.
- If the input character is an upper-case letter then return its lower-case equivalent.
- If the input character is not a letter then return it unchanged.

Analysis: The major problem is establishing the relationship between upper and lower-case letters, so that conversions may be easily made.

It can be used the ASCII code to effect due to the existence of the two intrinsic functions **`ichar()`** and **`char()`**.

`Ichar()` provides the position of its character argument in he ASCII collating sequence.

`ichar(“A”)` is 65. **`Char()`** returns the character at a specified position in that sequence.

Therefore **`char(97)`** returns to the character “a”. Every lower-case character is exactly 32 positions after its upper-case equivalent in the ASCII character set.

Structure plan

```
Function change_case(ch)
if A <= ch <=Z
calculate the lower-case of the character ch
else if a <= ch <=z
calculate the upper-case of the character ch
else
return without changing
end if
```

```
 program character_converter
 character(len=1) ::input_char,change_case
 print *,"enter a character"
 read *, input_char
 print *,"input character =  ",input_char
 print *,"output character = ",change_case(input_char)
 end program character_converter
 function change_case(ch) result(ch_new)
!   this function changes the case of its argument
!               if it is alphabetic
!   Dummy arguments and result
 character (len=*), intent(in)::ch
 character (len=1) :: ch_new
!   Check if argument is upper-case - convert lower-case
 if (ch>="A" .and. ch<="Z") then
 ch_new = char(ichar(ch) + 32)
!   Check if argument is lower-case - convert upper-case
 else if (ch>="a" .and. ch<="z") then
 ch_new = char(ichar(ch) - 32)
 else
!   not alphabetic
 ch_new = ch
 end if
 end function change_case
```

Example 2: Write a program that reads the coefficient of a quadratic equation of the form $ax^2+bx+c=0$ and print its roots.

Analysis: The program will use the formula:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

There are three possible cases:

$\sqrt{b^2-4ac} > 0$	The equation will have two real roots.
$\sqrt{b^2-4ac} = 0$	The equation will have two coincident roots
$\sqrt{b^2-4ac} < 0$	The equation will have no roots. (actually in this case the roots will be imaginary but this is out of scope of this chapter)

Since the real arithmetic is an *approximation*, the equality of two real numbers should never been *tested*. If the numbers have been calculated in a different way, they will often differ very slightly. This difficulty can be avoided by comparing the difference two real numbers with a very small number. Therefore the second case can be written as follows:

$$-\text{small} \le \sqrt{b^2 - 4ac} \le \text{small}$$

where *small* is a very small number, in this case the equation will have one root.

In this case the suitable design would be:

```
read coefficients
calculate b²-4ac, and store it in d
if d > small then
• calculate and print two roots
if d < -small then
•  print message “no roots”
 if –small ≤ d ≤ small then
•  calculate and print two equal roots
  end of if block
```

```fortran
        program quadratic_equation_solution
!       A program to solve a quadratic equation using an if
!       construct to distinguish between the three cases.
!       Constant declaration
        real, parameter :: small=1.e-6
!       Variable declarations
        real :: a,b,c,d,x1,x2
!       read coefficients
        print *," Type the three coefficients a, b and c"
        read *, a,b,c
!       Calculate b^2-4ac
        d = b**2 - 4.0*a*c
!       calculate an print roots
        if (d > small) then
!       two roots case
        x1 = (-b - sqrt(d)) / (2.0*a)
        x2 = (-b + sqrt(d)) / (2.0*a)
        print *," The equation has two roots:"
        print *," x1=",x1,"  x2=",x2
        else if (-small <= d .and. d <= small) then
!       two coincident roots case
        x1 = -b / (2.0*a)
        print *," The equation has two coincident roots:"
        print *," x1=x2=",x1
        else
!       No root case
        print *," The equation has no real roots"
        end if
        end program quadratic_equation_solution
```

The case construct

F provides another form of selection, known as the **case construct**, to deal with the situation in which the various alternatives are mutually exclusive.

The initial statement of a case construct takes the form

```
select case (case_expression)
```

where *`case_expression`* is either an **integer** or a **character** expression.

When the select case statement is encountered the value of *`case_expression`* is evaluated, and its value used to determine which, if any, of the alternative blocks of statements in the **case** construct is to be executed.

```
select case (case_expression)
case (case_selector)
block of statements
case (case_selector)
block of statements
  .
end select
```

The *`case_selector`* determines, which, if any, of the blocks of statements will be obeyed

<u>Example 1</u>: Write a program that reads the date in the form of dd-mm-yyyy and prints a message to indicate whether on this date, it will be spring, summer, fall and winter.

<u>Analysis:</u> The problem is ideally suited for a case statement. The structure plan of the problem may be:

```
read date
extract month from date
select case
month is 3-5 => print “spring”
month is 6-8 => print “summer”
month is 9-10 => print “fall”
month is 11-12,1 => print “winter”
month is anything else print an error message
```

```fortran
        program seasons
!       A program to calculate in which season a specified date lies.
!       variable declarations
        character(len=10) :: date
        character(len=2) :: month
!       read date
        print *, "Please type a date in the form dd-mm-yyy"
        read *, date
!       extract month number
        month = date(4:5)
!       extract from 4th to 5th character of string date and assign them
!        to character variable month
!       print season
        select case(month)
!       case("03","04","05")
        case("03":"05")
        print *, date , " is in the spring"
        case("06","07","08")
        print *, date , " is in the summer"
        case("09","10","11")
        print *, date , " is in the fall"
        case("12","01","02")
        print *, date , " is in the winter"
        case default
        print *, date , " is invalid date"
        end select
        end program seasons
```