'SELECT CASE' Construct & Exam

Handout Nine

1 'SELECT CASE' Construct

The 'SELECT CASE' Construct is another decision making or branching construct to control the flow of your codes.

1.1 The 'IF THEN ELSEIF' construct revisited.

Earlier in the course you were introduced to the 'IF' construct. The 'IF' construct allows 'overlap' of ranges in tests made in the segments of the construct. For example consider the following piece of code.

```
PROGRAM example_if
!**  A code to demonstrate the importance of ordering and
!**  the allowed overlapping of the test ranges in an IF
!**  construct.

  IMPLICIT NONE

  INTEGER :: i

  PRINT*,"Enter an INTEGER"
  READ*,i  !*** Get the user to input i

  !**** IF construct with overlapping of test ranges.
  !**** Therefore ORDERING is important

  IF (i .EQ. 2) THEN
    PRINT*,"Integer = 2"
  ELSEIF (i .EQ. 7) THEN
    PRINT*,"Integer = 7"
  ELSE IF (i .GE. 2 .AND. i .LE. 8) THEN
    PRINT*,"Integer is greater than one and less than nine"
  ELSE IF (i .GE. 6 .AND. i .LE. 10) THEN
    PRINT*,"Integer is larger than five and less than eleven"
  ELSE
    PRINT*,"No Match"
  END IF
END PROGRAM example_if
```

The execution of this construct involves testing the logical expression of each branch from top to bottom in turn. If one of the logical expressions evaluates to '.TRUE.' then the statements in that branch are executed and the construct exited. Note how the ranges overlap so the order that the programmer places the branches in the construct are significant.

Now consider the following piece of code.

```
PROGRAM example_if2

  IMPLICIT NONE

  INTEGER :: i

  PRINT*,"Enter an INTEGER"
  READ*,i  !*** Get the user to input i

  !**** IF construct with no overlapping of test ranges.
  !**** Therefore ORDERING is NOT important

  IF (i .EQ. 2) THEN
    PRINT*,"Integer = 2"
  ELSE IF (i .EQ. 7) THEN
    PRINT*,"Integer = 7"
  ELSE IF (i .GT. 2 .AND. i .LT. 7) THEN
    PRINT*,"Integer is greater than 2 an less than 7"
  ELSE IF (i .GT. 7 .AND. i .LE. 10) THEN
    PRINT*,"Integer is greater than 7 and less than 11"
  ELSE
    PRINT*,"No Match"
  END IF
END PROGRAM example_if2
```

Exercise One : In a '`handout9/exercise1`' directory copy the above two codes from;

```
fortran1 exercise1> cp ~/info/examples/handout9/exercise1/example_if.f90 .
fortran1 exercise1> cp ~/info/examples/handout9/exercise1/example_if2.f90 .
```

Remember the 'period' at the end is required as it tells Linux you want to copy the file to your current directory. Compile and run the codes making sure you understand the results .

Notice in the second code there is no overlapping of the ranges in the logical test expressions for the various branches of the construct, so each test is unique. As there is no overlapping, the order in which the programmer chooses to place each branch in the construct is no longer an issue. In such a situation the '`SELECT CASE`' construct can be used instead of the more general '`IF THEN ELSEIF`' construct.

2 'SELECT CASE' Construct

The 'SELECT CASE' Construct is particularly useful if one of several paths through an algorithm must be chosen based on the value of a single expression.

```
SELECT CASE < case-expression >
  CASE < case-selector >
      < exec-statements >
  CASE < case-selector >
      < exec-statements >
  CASE DEFAULT
      < exec-statements >
END SELECT
```

The above construct shows two 'CASE' blocks however you can have as many as you need but ONLY one 'CASE DEFAULT' block. The 'CASE DEFAULT' block is actually optional, if left out and none of the conditions to enter a 'CASE' block are evaluated to '.TRUE' then the 'CASE' construct does nothing.

- The < case-expression > must be scalar and of 'INTEGER', 'LOGICAL' or 'CHARACTER' type.
- There may be any number of general 'CASE' statements but only one 'CASE DEFAULT' branch.
- the < case-selector > must be a single item or a comma separated list of items. Each item can be a parenthesised single value or a range (section with a stride of one), for example, '(.TRUE.)' or '(99:101)'. A range specifier is a lower and upper limit separated by a single colon. One or other of the bounds is optional giving an open ended range specifier.
- All the < case-selector > items in the construct must be mutually exclusive of each other (no overlapping).
- The < case-expression > is evaluated and compared with each< case-selector > in turn.
- If no branches are are matched then the 'CASE DEFAULT' is executed (if present).
- When the < exec-statements > in a matched branch have been executed, control jumps out of the 'CASE' construct (via the 'END SELECT' statement).
- As with other similar structures it is not possible to jump into a 'CASE' construct.

Consider the example below where the '< case-expression >' is simply an 'INTEGER' 'i'. The construct simply picks out any prime numbers for that are less than ten. Note how the first 'CASE' demonstrates the use of a comma separated list. Note the '< case-selector>', for the second 'CASE' block, '(10:)' means any integer greater than 100.

```
SELECT CASE (i)
  CASE (3,5,7)
    PRINT*,"i is prime"
  CASE (10:)
    PRINT*,"i is > 10"
  CASE DEFAULT
    PRINT*, "i is not prime and is < 10"
END SELECT
```

The next example is slightly more complicated. Note the '< case-selector>', for the third 'CASE' block, '(100:)' means any integer greater than 100.

```
SELECT CASE (num)
  CASE (6,9,99,66)      !** IF(num==6.OR. .. .OR.num==66) THEN
    PRINT*, "Woof woof"
  CASE (10:65,67:98)    !** ELSEIF((num.GE.10.AND.num.LE.65) .OR. ...
    PRINT*, "Bow wow"
  CASE (100:)           !** ELSEIF (num.GE.100) THEN
    PRINT*, "Bark"
  CASE DEFAULT          !** ELSE
    PRINT*, "Meow"
END SELECT
! ENDIF
```

Exercise Two : In a '`handout9/exercise2`' directory rewrite the second code from 'exercise one' using a '`SELECT CASE`' construct instead of an '`IF THEN ELSEIF`' construct.

---

3 The Exam

Programming is different than other areas of mathematics, you can be convinced you have done it correctly and then the compiler spits error messages at you. In the exam there is no compiler, only you. So try not to lose easy marks that you could have gained with a little more care.
The marking scheme is not 'overly' strict with trivial syntax errors or spelling mistakes but you must try to make sure that the keyword order and logical structure of your statements are correct. For example in the declaration of an '2D' array to accommodate a matrix.

```
REAL, DIMENION(4,4) ::  matrix1
```

is logically correct but has a trivial spelling error, '`DIMENION`' instead of `DIMENSION` so, in a case like this, you would not lose marks, however,

```
REAL, DIMENSION(4) ::  matrix1
```

is logically incorrect because a two dimensional array is needed to accommodate a matrix.

3.1 Exam Structure and Hints

- Answer all questions.
- In **all** questions where you are required to write code you should follow the rules you have been taught for neat, structured and indented code. You should also use '`IMPLICIT NONE`' where appropriate and **all** dummy arguments should be declared with the correct '`INTENT`' attribute.
- Remember you are not typing into 'nano', there are NO [Backspace] or [Delete] keys. This means you should be sure of something when you write it down, use rough paper to test your thoughts. There is no substitute for practice here.
- Leave some space between the lines of your written code so if, later on, you wish to change something or add something you can do so more easily.
- If you are having difficulty getting started with a code start by writing down the '`PROGRAM`' and '`MODULE`' names and the '`FUNCTION`' and '`SUBROUTINE`' headers is a start, writing down the structure of your code in this way may help get you started and may get you some marks. Declaring the variables needed and, if dummy variables, their correct '`INTENT`' may also get you marks. If you know you need a '`DO`' loop or an '`IF`' structure but are not sure what to put in the structure body, just putting in the empty '`DO`' loop and '`IF`' structures may get you marks.

- If you are finding that you are really hitting a brick wall with a question, it may be worth moving onto another question and then returning to the tricky one. It is often the case that temporarily switching your train of thoughts to another problem does the trick.

3.2 Revision

You should revise:

- ALL the work covered in the all the handouts except the non-Fortran sections of handout one. That is you will not be examined on Linux, nano and the '`PRINT`' formatting descriptors.
- Anything on Fortran that has been mentioned in class.
- Everything you have been asked to do in your class-projects.
- All of the exercises in the all of the handouts after and including 'handout2'.
- All of the work set in, and notes handed out with, all your assignments.
- You may be asked to code a method you have not seen in this course but it will be well described in the exam.
- I recommend that you read through the two previous papers on the exam archive.
- Make sure you understand the numerical methods underlying all the codes you have seen and written through the duration of the course. Understanding the methodology is crucial to clear, correct and efficient programming.
- Make sure you practice writing down your codes on paper! Then go through it carefully with a printout of the solutions to see where you seem to make mistakes and be aware of these possible mistakes in the exam.
- You will not be examined on the '`PRINT`' statement in any great detail. You will only only be required to use the default formatting '`*`'. This means you do not need to learn all the '`formatting`' descriptors.
- In the exam a question may direct you to formulate your answer in a certain way. You should follow this advice. Note that, for example, you may be asked to create certain variables for a piece of code. You will need to do this but it may not be all the variables you need, you may need to declare other variables not hinted at in the question. However, if you are given something more concrete like the function header including its argument list then this will be exactly what you need to use for that function.

4 Revision Check List

Here is a list of what you could be examined on, it is not necessarily comprehensive but covers nearly everything you have done.

- **Main program structure** - Handout [2] page [1]
- **Numeric declarations** - Handout [2] page [2]
- **Numeric operators** - Handout [2] page [3], in particular make sure you know the order of precedence of the operators.
- **Character declarations** - Handout [2] page [3]
- **Logical declarations** - Handout [2] page [4]
- **Programming Style** - Handout [2] page [4]
- **The '`IF`' statement and construct** - Handout [2] page [4 & 7] ALL the forms of this!

– **Relational & Logical Operators & Expressions** - Handout [2] page [5]
– **The Looping 'DO' Construct** - Handout [2] page [9]
– **The Conditional 'DO' Construct** - Handout [2] page [10]
– **Numeric Arrays : Part One** - Handout [2] page [11]
– **Printing Arrays to the screen** - Handout [2] page [11]
– **Expression evaluation** - Handout [3] page [1]
– **Integer division** - Handout [3] page [2]
– **Data Casting** - Handout [3] page [2,3]
– **Intrinsic Functions** - Handout [3] page [3]
– **Data Type Limitations** - Handout [3] page [4]
– **Complex Arithmetic** - Handout [3] page [4]
– **The true meaning of '=' in Fortran** - Handout [3] page [4]
– **Use of the '&' character and blank spaces** - Handout [3] page [4]
– **Simple Input & Output 'PRINT*' and 'READ*'** - Handout [3] page [5,6]
– **Characters & Strings : declaration** - Handout [4] page [1]
– **Concatenation & substrings** - Handout [4] page [1]
– **Character intrinsic functions** - Handout [4] page [2]
– **Internal functions and subroutines** - Handout [5] page [1]
– **Argument lists** - Handout [5] page [2]
– **Local and Dummy Variables & Call by Reference** - Handout [5] page [3]
– **Scope** - Handout [5] page [4]
– **Functions** - Handout [5] page [6]
– **The 'INTENT' attribute** - Handout [5] page [7] Learn all three aspects of this attribute including where and how they should be used!
– **Array Terminology** - Handout [5] page [8]
– **Array references** - Handout [5] page [8]
– **Array construction** - Handout [5] page [9]
– **Array Syntax and Expressions** - Handout [5] page [9]
– **Assumed shape arrays** - Handout [5] page [10]
– **Automatic arrays** - Handout [5] page [10] : Learn also the use of automatic arrays in conjunction with the 'SIZE' function.
– **'SIZE' Function** - Handout [5] page [12]
– **'MODULE's (essential)** - Handout [6] page [1] Essential part of the course
– **module structure** - Handout [6] page [1]
– **'USE' statement** - Handout [6] page [2,3] Learn not only its basic use but also how to 'limit' the entities made available by the 'USE' statement. Also make sure you know how to **'rename'** entities within the 'USE' statement.
– **Modules to hold common data** - Handout [6] page [4] : Remember 'MODULES' can be used to hold data (REAL, INTEGER, arrays etc) and they can be made visible to other program units via the 'USE' statement.
– **Reasons for Using Modules** - Handout [6] page [4]
– **Array element by element arithmetic** - Handout [6] page [6] (Points to note about the code)

- **Initial value problems for ODE's** - Assignment Two : Make sure you know how to code up the methods introduced here. If you are asked to write code for these in the exam the required formulae will be given but if you practice coding up the methods you will benefit greatly in the exam.
- **Eulers Method** - Handout [Assignment Two] page [1,2]
- **Heun's Method** - Handout [Assignment Two] page [2]
- **Nystom's Method** - Handout [Assignment Two] page [2] : Remember this method is 'not' self starting!
- **3rd & 4th Order Runge-Kutta Method** - Handout [Assignment Two] page [4]
- **2,3,4 step Adams-Bashforth Methods** - Handout [Assignment Two] page [4,5]
- **Simple Predictor Corrector 2nd Order Method** - Handout [Assignment Two] page [4]
- **Milne's Method & Modified version** - Handout [Assignment Two] page [5]
- **Hammings's Method & Modified version** - Handout [Assignment Two] page [6]
- **1D Array Construction** - Handout [7] page [1] : Assigning values to 1D arrays using other 1D arrays, or segments of 1D arrays or 1D segments of multidimensional arrays.
- **2D Array Construction and the 'RESHAPE' intrinsic function** - Handout [7] page [1]
- **Array Syntax and Expressions** - Handout [7] page [2]
- **Some Array Intrinsics** - Handout [7] page [3]
- **Dynamic Allocation of Arrays** - Handout [7] page [4]
- **Keyword arguments** - Handout [7] page [5]
- **Optional arguments** - Handout [7] page [6]
- **Basic file I/O : 'OPEN' Statement** - Handout [8] page [1]
- **The 'CLOSE' statement** - Handout [8] page[2]
- **The 'INQUIRE' Statement** - Handout [8] page[2]
- **Formatted Files (text files)** - Handout [8] page[3]
- **Unformatted Files (binary files)** - Handout [8] page[4]
- **SELECT CASE construct** - Handout [9] page[1-4]
- **All your exercise, assignment and class project codes : Quadratic, Maclaurin, Isotope Code etc**

+++++ THE END +++++

Best of luck in the exam and I hope you gained some useful programming experience from the course.