3. Programming Languages - II

[1]

[1] M. Ganesh, MATH440/540, SPRING 2018

Fortran 95/2003

[2]A typical F90/95/2003 code to solve $ax^2 + bx + c = 0$:

```
PROGRAM roots !  This program solves a * x**2 + b * x + c = 0.
IMPLICIT NONE
!  Declare the variables used in this program
REAL ::  a,b,c !  Coefficients of the equation
REAL ::  disc, imag_part, real_part, x1, x2 !New variables
!  Read coefficients
WRITE (*,*) 'Enter the coefficients A, B, and C:'
READ (*,*) a, b, c
!  Check data entry
WRITE (*,*) 'The coefficients A, B, and C are:  ', a, b, c
disc = b**2 - 4.*a*c
IF ( disc > 0.0 ) THEN ! there are two real roots X1 and X2:
X1 = ( -b + sqrt(disc) ) / ( 2.0*a ) !  Note X1 is same as x1
X2 = ( -b - sqrt(disc) ) / ( 2.0*a )
WRITE (*,*) 'This equation has two real roots X1 and X2:'
WRITE (*,*) 'X1 = ', x1
WRITE (*,*) 'X2 = ', x2
```

[2]M. Ganesh, MATH440/540, SPRING 2018

```
ELSE IF ( disc == 0.0 ) THEN ! there is one repeated root x1:
x1 = ( -b ) / ( 2.0*a )
WRITE (*,*) 'This equation has two identical real roots:'
WRITE (*,*) 'X1 = X2 = ', x1
ELSE ! the roots are complex roots:
real_part = ( -b ) / ( 2.  * a )
imag_part = sqrt ( abs (disc) ) / ( 2.0*a )
WRITE (*,*) 'This equation has complex roots:'
WRITE (*,*) 'X1 = ', real_part, ' +i ', imag_part
WRITE (*,*) 'X2 = ', real_part, ' -i ', imag_part
END IF
END PROGRAM roots
```

[3]

- **Save the above code as** `roots.f90`

- **On Sayers Lab machines using the GNU Fortran compiler (gfortran) we may creat an executable version, say,** `roots_exe`**, by typing the command**
  `gfortran -o roots_exe roots.f90`

- **You may also use the Portland Group (PG) compiler in the lab. For PG compiler, replace** `gfortran` **in the above command with** `pgf90` **or** `pgf95`**, or** `pghpf`**.**

- **To execute the file, type the command** `./roots_exe`

- **On MIO/AuN, you may use** `gfortran`**,** `pgf90`**,** `pgf95`**,** `pghpf`**, or the Intel Fortran Compiler** `ifort`**.**

[3]M. Ganesh, MATH440/540, SPRING 2018

- [4]**We use the notation F90+ to denote Fortran 90/95/2003**
- **In F90+ the 26 uppercase letters `A` to `Z` are same as the corresponding 26 lowercase letters `a` to `z`**
- **In addition, the F90+ character set consists of digits (`0` to `9`), underscore character (_), arithmetic symbols (`+`, `-`, `*`, `/`, `**`), miscellaneous symbols:`( ) = , ' $ : ! '' % & ; < > ?` and blank**
- **Fortran 2003 character-set includes additional symbols:**

  `~ \ [ ] ‘ ^ { } | # @`

- **F90+ statements in a line may be up to 132 characters long**
- **A statement can be continued in the next line by ending the current line with an ampersand (`&`), with a maximum continuation of up to 40 lines**
- **Fortran 2003 allows continuation up to 256 lines**

[4]M. Ganesh, MATH440/540, SPRING 2018

- **F90+ statements can be numbered (between 1 and 99999)**
- **The statements numbers are unique within a program**
- **Avoid using statement numbers in F90+**
- **Any characters followed by `!` are comments**
- **In general, main F90+ programs have three sections: declaration, execution, and termination**
- **The first statement of the declaration section is a nonexecutable `PROGRAM` statement (first one) that specifies the name of the program**

- **The names/variables of F90+ programs may each have up to 31 characters. (In Fortran 2003, these can be up to 63 characters)**
- **The termination section consists of an `END PROGRAM` that specifies the name of the program to be terminated**
- **The `STOP` statement can be used anywhere between the second and last line of a program to break a program due to, for example, encountering some unwanted values or ....**

- [5]In F90+, there are five built-in type constants and variables: `INTEGER`, `REAL`, `COMPLEX`, `CHARACTER` , `LOGICAL`
- `INTEGER`:
  - ⋆ **An `INTEGER` constant does not contain a decimal point**
  - ⋆ **An `INTEGER` variable takes a value of integer data type**
  - ⋆ **Most compilers support 16-bit and 32-bit integers, leading to maximum integer constant/variable value being** $2^{31} - 1 = 2147483647 = 2.147483647 \times 10^9$
  - ⋆ **F90+ allows choice of 16-bit or 32-bit kinds of integers**
  - ⋆ **The kind number can be obtained for a particular choice, system, and compiler using the built-in function** `SELECTED_INT_KIND`
  - ⋆ **The statement** `kind_number = SELECTED_INT_KIND(r)` **returns the smallest kind of integer value with maximum storage of** r **digits**
  - ⋆ **On Sayers Lab machines and MIO/AuN, with GFORTRAN, PG, and INTEL compilers, the useful** `kind_number` **values are 1, 2, 4, and 8 respectively for** $r = 1, 2$, $r = 3, 4$, $r = 5, 6, 7, 8$, **and** $r = 10$ **to** 16

[5]M. Ganesh, MATH440/540, SPRING 2018

- Use `INTEGER :: val` or `INTEGER(KIND=8) :: val` or `INTEGER(KIND = SELECTED_INT_KIND(10)) :: val`[6]
- **Beware of integer arithmetic:** **involves integer data and produces only an integer result**
- **The statements**

```
INTEGER, PARAMETER :: master = 0 !Note: PARAMETER
INTEGER(KIND = SELECTED_INT_KIND(10)) :: a, b
a = 5
b = 8
c = a/b
```

- **will lead to the value of** `c = 0.0`
- **Avoid integer arithmetic, especially division, whenever possible**
- **If the type of a variable is not explictly specified and if the variable name begins with one the letters** `I, J, K, L, M, N,` **then the variable is assumed to be of** `INTEGER` **type; otherwise assumed to be of** `REAL` **type**

[6] M. Ganesh, MATH440/540, SPRING 2018

[7]`REAL constants and variables :`

- **A `REAL` constant contains a decimal point**
- **A `REAL` variable takes a value of real data type**
- **Storage of data type, say x, is done in three parts: sign, mantissa, and exponent, of the form**
  - ⋆ $x = s(d_0 \cdot d_1 d_2 \cdots d_{p-1})_\beta \beta^e = s(d_0 + d_1\beta^{-1} + d_2\beta^{-2} + \cdots + d_{p-1}\beta^{-(p-1)})\ \beta^e$
  - ⋆ **sign** $s = +$ **or** $-$
  - ⋆ **digits** $d_0, d_1, \ldots, d_{p-1}, \qquad 0 \leq d_j \leq \beta - 1$
  - ⋆ **precision** p
  - ⋆ **base** β
  - ⋆ **exponent** e **with** $e_{\min} \leq e \leq e_{\max}$
  - ⋆ **exponent (in bits with width w)** $e = (e_1 e_2 \cdots e_w)_\beta$
  - ⋆ **mantissa (in bits with precision value p)** $(d_0 \cdot d_1 d_2 \cdots d_{p-1})_\beta$
  - ⋆ **In IEEE Standard 754, the p-digit mantissa is stored using $p-1$ bits, because the first digit in a normalized binary number is always 1 (and de-normalized numbers, `Inf`, `NaN` are taken care of using certain biased representation for the exponent in IEEE 754)**

[7]M. Ganesh, MATH440/540, SPRING 2018

[8]5.3 IEEE Standard 754

	Single Precision	Double precision
β	**2**	**2**
p	**24**	**53**
w	**8**	**11**
$e_{\max}$	$2^7 - 1 = 127$	$2^{10} - 1 = 1023$
$e_{\min}$	$2 - 2^7 = -126$	$2 - 2^{10} = -1022$
ϵ	$2^{-23} \approx 1.2 \times 10^{-7}$	$2^{-52} \approx 2.2 \times 10^{-16}$

Width in Bits

	Single	Double
sign	**1**	**1**
exponent	**8**	**11**
mantissa	**23**	**52**
Total	**32**	**64**

[8]M. Ganesh, MATH440/540, SPRING 2018

- [9]**The standard dictates a biased representation for the exponent with**

$$e = E - e_{\max} = E - (2^{w-1} - 1).$$

- **Notice that**

$$e = e_{\max} + 1 \qquad \textbf{when} \qquad E = 2e_{\max} + 1 = 2^w - 1 = (1 \cdots 1)_2$$

$$e = e_{\min} - 1 \qquad \textbf{when} \qquad E = e_{\max} + e_{\min} - 1 = 0 = (0 \cdots 0)_2$$

biased exponent	**signed exponent**	**mantissa**	**number**
$1 \le E \le 2^w - 2$	$e_{\min} \le e \le e_{\max}$	$0 \le M \le 2^{p-1} - 1$	$1.m \times 2^e$
$E = 0$	$e = e_{\min} - 1$	$\begin{cases} M = 0 \\ M \ne 0 \end{cases}$	0 $0.m \times 2^{e_{\min}}$
$E = 2^w - 1$	$e = e_{\max} + 1$	$\begin{cases} M = 0 \\ M \ne 0 \end{cases}$	`Inf` `NaN`

[9]M. Ganesh, MATH440/540, SPRING 2018

- **E is the value of exponent bits and M is value of mantissa bits**
- **The IEEE 754 standard leads to the following in double precision:**
  - ⋆ **Largest positive normalized machine number is:**
    $(\beta - \beta^{1-p})\beta^{e_{\max}} = 1.797693134862316\ldots \times 10^{308}$
  - ⋆ **Smallest positive normalized machine number is:**
    $\beta^{e_{\min}} = 2.225073858507201 \times 10^{-308}$

- [10]**A simple approach to create double precision real variables, say `a`, `b`, `pi`, in F90+ is to use the statements**

```
double precision ::  a, b
double precision, parameter ::  pi = 3.14159265358979 3d0
```

- **To create single precision variables `a` and `b`, replace `double precision` with `single precision`**
- **The simple approach assumes that single precision variables are stored in 32-bits and double precision variables are stored in 64-bits**
- **However, some 64-bit processors use 64-bits for single precision and 128-bits for double processors**
- **In view of expected substantial changes in processors and compilers, avoid the above simple approach/assumption**
- **Fortran 95/2003 provides a way to avoid the above assumption using the intrinsic `KIND` functions and `KIND` type parameters**
- **The `kind` numbers depend also on compilers**

[10]M. Ganesh, MATH440/540, SPRING 2018

- [11]**The processor and compilers dependent kind number, say** `MINE`**, that facilitates declaration of** `real` **variables with a decimal digits accuracy in mantissa and exponent range between** $10^{-\mathrm{b}}$ **and** 10^{b} **can be created using** `SELECTED_REAL_KIND(a,b)` **or** `SELECTED_REAL_KIND(p=a,r=b)` **or** `SELECTED_REAL_KIND(precision=a,range=b)`
- **For example, the integer parameter** `MINE` **with** $p = 15, r = 30$ **can be created using the statement**
  `INTEGER, PARAMETER ::  MINE = SELECTED_REAL_KIND(15,30)`
- **On Sayers Lab machines (with GNU, PG compilers) and on MIO/AuN (with gfortran, PG, INTEL compilers), we get** `MINE = 8`
- **The above statement may be followed by**
  `REAL (KIND=MINE) ::  e, f, g`
  `REAL (KIND=MINE), PARAMETER ::  h = 2.74_MINE`
  to create real variables with kind-type `MINE`
- **The statement** `e_type = KIND(e)` **will give** `e_type = 8`
- **A simple approach to obtain single and double precision kind numbers is to seek output of** `kind(0.0)` **and** `kind(0.d0)`**.**
  (Resp. `4` **and** `8` **on MIO/AuN.)**

[11] M. Ganesh, MATH440/540, SPRING 2018

- [12]**Recall that, according to IEEE standard 754, in double precision (with** 64**-bits) the largest positive normalized machine number is:** $(\beta - \beta^{1-p})\beta^{e_{\max}} = 1.797693134862316\ldots \times 10^{308}$
- **For "**`val = 1.5e+39`**", why does an F90+ compiler gives an overflow error despite choosing correct precision and range for** `val`**?**
- **In our derivation of** $1.797693134862316\ldots \times 10^{308}$**, we used the IEEE 754 standard of how the** 64**-bits are allocated in double precision**
- **In particular, the exponent part has** 11 **bits, hence** $e_{\max} = 2^{10} - 1 = 1023$ **and the remaining** 53 **bits are for the non-exponent part, leading to machine epsilon** $2^{-52} = 2.2204 \times 10^{-16}$**, that is,** 15 **to** 16 **digit accuracy.**
- **F90+ standard does not specify how bits must be used**
- **In particular, because** $2^{-55} = 2.7756 \times 10^{-17}$**, to achieve** 16 **to** 17 **accuracy, one may allocate** 56 **bits for the non-exponent part and the remaining** $64 - 56 = 8$ **bits (as in single precision) for the exponent part**
- **In this case, we get** $e_{\max} = 2^7 - 1 = 127$**,** $e_{\min} = 2 - 2^7 = -126$**. Thus the range of the exponent is from** 10^{-38} **to** 10^{38}**, because, with** $p = 56, e_{\max} = 127, e_{\min} = -126$**,** $(\beta - \beta^{1-p})\beta^{e_{\max}} = 3.4028 \times 10^{38}$ **and** $\beta^{e_{\min}} = 1.1755 \times 10^{-38}$

[12]M. Ganesh, MATH440/540, SPRING 2018

`COMPLEX` constants and variables :[13]

- **A `COMPLEX` constant contains two numeric constants, separated by a comma and enclosed in parenthesis**
- **The first and second numeric constants in the parenthesis are respectively the real and imaginary part of the complex constant**
- **A `COMPLEX` variable takes a value of complex data type**
- **The following statements illustrate initializing and using complex constants and variables:**
-
```
COMPLEX ::  a = (3, 5)
INTEGER, PARAMETER ::  MINE = SELECTED_REAL_KIND(15,30)
REAL(KIND=MINE) ::  a1 = 3.1423_MINE
REAL(KIND=MINE) ::  a2 = 4.7453_MINE
COMPLEX(KIND=MINE) ::  b, c
COMPLEX ::  d
b = CMPLX(a1, a2, MINE)
c = CONJG(a)
d = CABS(a)
```

[13]M. Ganesh, MATH440/540, SPRING 2018

[14]A shorter F90+ code to solve $ax^2 + bx + c = 0$ using CMPLX:

```
PROGRAM roots !  This program solves a * x**2 + b * x + c = 0.
IMPLICIT NONE
!  Declare the variables used in this program
REAL (KIND = SELECTED_REAL_KIND(15,100)) ::  a,b,c, disc
COMPLEX (KIND = SELECTED_REAL_KIND(15,100)):: x1, x2
!  Read coefficients
Write (*,*) 'Enter the coefficients A, B, and C:'
READ (*,*) a, b, c
!  Check data entry
WRITE (*,*) 'The coefficients A, B, and C are:  ', a, b, c
disc = b**2 - 4.*a*c
x1 = ( -b + sqrt( CMPLX(disc, 0.0) ) / ( 2.0*a )
x2 = ( -b - sqrt(CMPLX(disc, 0.0) ) / ( 2.0*a )
WRITE (*,*) 'This roots are:'
WRITE (*,*) 'x1 = ', REAL(x1), ' +i ', AIMAG(x1)
WRITE (*,*) 'x1 = ', REAL(x2), ' -i ', AIMAG(x2)
END PROGRAM roots
```

[14]M. Ganesh, MATH440/540, SPRING 2018

[15]`Character constants and variables :`

- **A `character` constant is a string of characters enclosed in single or double quotes**
- **A `character` variable takes a value of character data type that consists of strings and alphanumeric characters**
- **To create three character variables, say `first_name`, `mid_ini`, `last_name`, with `mid_ini` of length one and rest of length 25, we may use**

```
CHARACTER :: mid_ini
CHARACTER(len=25) :: first_name, last_name
```
or
```
CHARACTER(25) :: first_name, last_name
```

- **Specifying length of a declared character may be avoided:**

```
CHARACTER, PARAMETER :: MY_ERROR = 'Don''t know how now, fix later'
```

- **Fortran 2003 allows a new function `SELECTED_CHAR_KIND(name)` where, for example, `name` can be `DEFAULT`, `ASCII`, `ISO_9660` or ..., to get an integer `kind_numb` that can then be used (if ≥ 1), for example, as**

```
CHARACTER(KIND = kind_numb, len = 25) :: first_name, last_name
```

[15]M. Ganesh, MATH440/540, SPRING 2018

[16] Write basic F90+ codes

- **As illustrated in the code for solving a general quadratic equation, to write a code to solve/simulate a chosen problem/model first identify various formulas/algorithms to solve the problem**
- **Start with a `PROGRAM` statement using a name for the code**
- **Use `IMPLICIT NONE` statement to turn off default types in Fortran**
- **Use the built-in type constants and variables (such as `INTEGER`, `REAL`, ...) to declare various variables required to solved the problem. Use `SELECTED_INT_KIND`, `SELECTED_REAL_ KIND`, ... to make the code portable to various computing environments**
- **If required, use `READ` and `WRITE` to read and write/validate any input data**
- **If required, use `STOP` statement to stop execution of the program. (It is common to use CHARACTER type to print out error messages, using `WRITE` or `PRINT`, before using a `STOP` statement in between the code.)**
- **End the code using `END PROGRAM` statement using the chosen name**

[16] M. Ganesh, MATH440/540, SPRING 2018

[17] `Logical constants, variables, and operators` :

- **A** `logical` **constant can take one of the two values:** `.TRUE.` **or** `.FALSE.`

- **A** `logical` **variable takes a value of the logical data type**
- **Logical variables, say,** `a, b, c`**, can be specified using the statement:** `LOGICAL :: a, b, c`
- **Arithmetic expressions and operators are used for real variables**
- **Similarly, logical variables are usually created using logical expressions that require logic (relational and combinational) operators**
- **The relational operators in F90+ are:**
  `==, /=, >, >=, <, <=,` **where the meaning of** `/=` **is** ***not equal to.***
  (In earlier Fortran versions : `.EQ., .NE., .GT., .GE., .LT., .LE.`**)**
- **The combinational operators in F90+ are:**
  `.AND., .OR., .EQV., .NEQV., .NOT.`
- **In particular, the statement** `a_new = a .NEQV. b`
  yields a new logical variable `a_new` **with value** `.TRUE.`**, if the logical variables** `a` **and** `b` **do not have the same value**

[17] M. Ganesh, MATH440/540, SPRING 2018

[18] IF statements

In F90+, `IF` statements take the following form:

```
IF <logical expression>1 THEN
     <statements>1
      ...
ELSE IF <logical expression>2 THEN
     <statements>2
      ...
ELSE
     <statements>3
      ...
 END IF
```

- `<statements>`$_i$ **are executed if** `<logical expression>`$_i$ **is** `.TRUE.`
- **More than one `ELSE IF` branch is permitted. The `ELSE IF` and `ELSE` branches may be omitted and statement lists may be empty**

[18] M. Ganesh, MATH440/540, SPRING 2018

- [19]**In F90+, a block of branching and loop statements can be named to facilitate ease in tracking nested branching and loop statements**
- **The format for a named IF construct is:**

```
name_value: IF <logical expression>1 THEN
               <statements>1
                 ...
            ELSE IF <logical expression>2 THEN
               <statements>2
                 ...
            ELSE
               <statements>3
                 ...
            END IF name_value
```

[19]M. Ganesh, MATH440/540, SPRING 2018

For example, in a nested class of `IF statements`, it is useful to have

```
my_outer: IF <logical expression>_i THEN
            <statements>_i
                        ...
            my_middle: IF <logical expression>_j THEN
                        <statements>_j
                        ...
                     my_last: IF <logical expression>_z THEN
                               <statements>_z
                          END IF my_last
                        ...
                     END IF my_middle
                        ...
         END IF my_outer
```

[20] `SELECT CASE statements`

- **These are branching construct in F90+, to execute a block of codes based on the value of a single integer, character, or logical expression**

- **The format for a named `SELECT CASE` construct is:**

```
name_value: SELECT CASE <case_expression>
CASE <case_value>_1
             <statements>_1
               ...
CASE <case_value>_2
             <statements>_2
               ...
CASE DEFAULT
             <statements>_default
               ...
           END SELECT name_value
```

[20] M. Ganesh, MATH440/540, SPRING 2018

- [21]<statements>$_i$ **are executed if** <case_expression> **is in the range of values included in the** <case_value>$_i$
- **The** CASE DEFAULT **block is optional and if included the** <statements>$_{default}$ **are executed only if** <case_expression> **does not match any of the range of values included in** <case_value>$_i$ **for** $i = 1, 2, \cdots$
- **The** name_value: **part is optional and if desired can be included also in the line** CASE <case_value>$_i$. **E.g.,** CASE <case_value>$_i$ name_value
- **The colon operator ":" is useful to specify** <case_value>$_i$
- **The statements in the block**
  CASE (low_val:high_val)
  is executed if low_val $\leq$ <case_expression> $\leq$ high_val
- **The statements in the block**
  CASE (:high_val)
  is executed if <case_expression> $\leq$ high_val
- **The statements in the block**
  CASE (low_val:)
  is executed if <case_expression> $\geq$ low_val

[21]M. Ganesh, MATH440/540, SPRING 2018

[22] Loops

In F90+, we have a few types of loops.
First we consider the while-type loop of the form:

```
DO
     <statements>
      ...
     IF <logical expression> EXIT
      ...
 END DO
```

It is better to use the above form rather than the following form:

```
DO WHILE <logical expression>
     <statements>
      ...
 END DO
```

[22]M. Ganesh, MATH440/540, SPRING 2018

- [23]**The iterative/counting loop in F90+ is of the form:**

```
DO index = istart, iend, incr
    <statements>
     ...
  END DO
```

- CYCLE **statements provide a mechanism to skip over some parts**
- **For example, to skip certain work described in** <statements>$_z$ **for a particular index (if encountered), say** 5**, within a loop, one may use**

```
DO index = istart, iend, incr
    <statements>
     ...
    IF (index == 5) CYCLE
    <statements>_z
 END DO
```

- **To exit out of the loop if index** 5 **is encountered, replace the** CYCLE **statement with** IF (index == 5) EXIT

[23] M. Ganesh, MATH440/540, SPRING 2018

[24]As in `IF` statements, `DO` loops can be named and these are particularly useful for nested loops:

```
my_outer: DO i = 1, 1000, 4
          <statements>_i
                    ...
          my_middle: DO p = i, 2000, 2
                    <statements>_j
                    ...
                    my_last: DO w = i*p, 6000, 3
                            <statements>_z
                         END DO my_last
                    ...
                    END DO my_middle
                    ...
       END DO my_outer
```

[24]M. Ganesh, MATH440/540, SPRING 2018

INPUT/OUTPUT statements[25] :

- **General READ/WRITE statements are of the form**

  `READ(s, f) data`   `WRITE(s, f) data`

  where `s` is the unit from/to which the `data` to be read/written using the format specified in `f`

- **In particular, the statements**
  `READ(*, *) data`   `WRITE(*, *) data`
  correspond to read/write from/to the standard input/output unit using a default format. These are unformatted I/O statements

- **Examples of formatted statements to print integer data `ival1, ival2` (using field width of size 8) and real data `rval1, rval2, rval3` (using field width of size 8 in characters and 4 digits reserved right of decimal place) are:**

```
WRITE(*, 100) ival1, ival2, rval1, rval2, rval3
```

  or

```
PRINT 100, ival1, ival2, rval1, rval2, rval3
100 FORMAT(2I8, 3F8.4)
```

[25] M. Ganesh, MATH440/540, SPRING 2018

- [26]The F format (rFw.d) displays the real data in decimal notation
- The E format (rEw.d) displays the real data in exponential notation
- The G format (rGw.d) displays the real data in E or F format (In E form if the exponent in the data is negative or bigger than d.)
- The ES format (rESw.d) displays the real data in scientific notation
- The EN format (rENw.d) displays the real data in engineering notation
- For the above E, G, ES, EN formats, $w \geq d + 7$, with two digits reserved for the exponent part
- To reserve e digits for the exponent part, use rEw.dEe, rGw.dEe, rESw.dEe, rENw.dEe
- The L format (rLw) is useful for logical data
- The A format (rAw) is useful for character data
- The X format (nX) is useful for skipping n spaces
- The T format (Tc) is useful to move to column c
- / is useful to move down one line

[26]M. Ganesh, MATH440/540, SPRING 2018

- [27]**In order to read/write data from/to a specific disk file instead of the standard I/O unit, use** `OPEN` **and** `CLOSE` **statements with a specific unit number and use the unit number to replace * in** `READ(*,100)` **or** `WRITE(*,100)`**:**

- **Suppose that we want our program to read some data from an existing input data file** `input_data.txt` **and that some output to be written in a new file** `output_data.txt`

- **We need to assign an integer, say** 9**, for the input data file and a different integer, say** 14**, for the output file and these are the unit numbers to be used.**

- **After opening these two files within the program, we may use** `READ(9,100)` **or** `WRITE(14,100)`**:**

- **The opening statments before the above statements are typically of the form**

```
OPEN(UNIT=9, FILE='input_data.txt', STATUS='old', ACTION = 'read',
IOSTAT=read_error)
OPEN(UNIT=14, FILE='output_data.txt', STATUS='new', ACTION = 'write',
IOSTAT=write_error)
```

[27]M. Ganesh, MATH440/540, SPRING 2018

- **[28]In the above `OPEN` statements, `ACTION` (specifying action should be either read or write or readwrite) and `IOSTAT` (returning value 0 to the assigned integer variables `read_error` and `write_error` if successful or 1 if not successful) may be omitted from the list**
- **Fortran 2003 provides a new option `IOMSG` that allows specifying a character that will be unchanged if successful and will be changed to an error message if unsuccessful**
- **The general form of `OPEN` statement is `OPEN(my_list)`, where `my_list`, may include**
  - ⋆ `UNIT=` **(a non-negative integer value)**
  - ⋆ `FILE=` **(a character value)**
  - ⋆ `STATUS=` **('old' or 'new' or 'replace' or 'scratch' or 'unknown')**
  - ⋆ `ACTION=` **(a character 'read' or 'write' or 'readwrite')**
  - ⋆ `IOSTAT=` **(an integer variable)**
  - ⋆ `IOMSG=` **(a character variable)**
  - ⋆ ...

[28]M. Ganesh, MATH440/540, SPRING 2018

- [29]**A simple approach to closing the opened units** 9 **and** 14 **are the statements**
  `CLOSE(9)`   `CLOSE(UNIT= 14)`

- **More generally, one may use**
  ```
  CLOSE(UNIT=9, STATUS='delete', IOSTAT = close_err)
  CLOSE(UNIT=14, STATUS='keep', IOMSG = 'ok_end')
  ```

- `REWIND (UNIT=9)` **is useful to position the file to allow the next** `READ` **statement to read from the first line in the file**

- `BACKSPACE (UNIT=9, IOSTAT= my_err)` **is useful to position the file back by one line for the next** `READ` **statement**

- `ENDFILE (UNIT=14, IOSTAT= end_err)` **statement is useful to make sure that no further** `READ`**s or** `WRITE`**s is possible until either the** `REWIND` **or** `BACKSPACE` **statements are executed for the unit** 14

- **Fortran 2003 allows further options such as** `WAIT, FLUSH, ...`

- `INQUIRE` **statements are useful to check the status or properties of a file before or after** `OPEN` **statements**

[29]M. Ganesh, MATH440/540, SPRING 2018

- [30]For example, use `INQUIRE (FILE='input_data.txt', EXIST = my_chk)` to get existence of the input data file in the working directory.
- Depending on the value of the logical variable `my_chk` several `IF` or other type statments can be used to proceed accordingly (such as creating new data, destroying old data, or ....)
- We may be interested in selecting certain group of variables within a code and write just that group to some output file
- This can be achieved by using `NAMELIST` statement of the form
  `NAMELIST / special_list / val1, val22, valxx`
  and then to write this to an already opened unit, say 15, use
  `WRITE(UNIT=15, NML=special_list)`
- The statements
  `WRITE(UNIT=14,FMT=100, IOSTAT=write_err) ...`
  `100 FORMAT (...)` create formatted output that is easy for us to read but may take substantial CPU time to read, if the file size is large
- For large output files (and that does not require you to see it and instead machine to access fast later), use **UNFORMATTED** output files. For example use `WRITE(UNIT=14, IOSTAT=write_err) ...`

[30]M. Ganesh, MATH440/540, SPRING 2018