Computing with Fortran
Engineering Tool V
Spring 2015

Dr. Andreas Haselbacher
Institute of Energy Technology
Overview

- Course overview
- Structure of simple Fortran program
- Compiling and linking
- Intrinsic data types
- Operations
- Intrinsic procedures
- Control flow
- Input/output
Course Overview

- Course is about **computing**, not **programming** with Fortran
- Programming is means to an end, not the end itself
- Therefore, need to be effective and efficient programmer
Course Overview

- Computers are fast but stupid: Once told what to do, they can do it quickly many times
- Fundamentally, programming is nothing but explaining actions to computer
- Consequences:
  - You need to understand action before you can explain it
  - Programming forces you to understand what you do
  - If computer does not do what you want, it’s your fault
Course Overview

• Course focus on Fortran 90 (called Fortran for simplicity)
• Changes in later versions (mostly) not important for us
  ‣ Fortran 95: Minor revision of Fortran 90
  ‣ Fortran 2003: Major additions to Fortran 95
  ‣ Fortran 2008: Minor revision of Fortran 2003
• gfortran compiler:
  ‣ Fortran 95: Completely supported
  ‣ Fortran 2003: Partially supported
  ‣ Fortran 2008: Partially supported
Course Overview: Scope & Structure

- Cannot cover all of Fortran in 12 hours
- Course will only cover what I view as most important parts
- Cannot cover debuggers, unfortunately

- Planned structure:

<table>
<thead>
<tr>
<th>Lecture</th>
<th>Day 1</th>
<th>Day 2</th>
<th>Day 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lecture 1</td>
<td>Lecture/programming</td>
<td>Lecture/programming</td>
<td>Programming</td>
</tr>
<tr>
<td>Lecture 2</td>
<td>Lecture/programming</td>
<td>Lecture/programming</td>
<td>Programming</td>
</tr>
<tr>
<td>Lecture 3</td>
<td>Lecture/programming</td>
<td>Lecture/programming</td>
<td>Programming</td>
</tr>
<tr>
<td>Lecture 4</td>
<td>Programming</td>
<td>Programming</td>
<td>Programming</td>
</tr>
</tbody>
</table>
Course Overview: Goals

- Students know important Fortran commands and concepts
- Students write well-structured Fortran programs
- Students use compiler effectively
- Students have fun programming!
Course Overview: Administrative Matters

• Course grade: Pass/fail
• Determination of grade based on program submitted after end of lectures on day 3
Course Overview: Recommended Books

• Fortran:

• Programming in general:
Overview

- Course overview
- **Structure of simple Fortran program**
- Compiling and linking
- Intrinsic data types
- Operations
- Intrinsic procedures
- Control flow
- Input/output
Structure of Simple Fortran Program

PROGRAM simple_program

IMPLICIT NONE

INTEGER :: i,j

i = 1
j = 2*i

WRITE(*,*) i,j

END PROGRAM simple_program
Structure of Simple Fortran Program

```
PROGRAM simple_program

IMPLICIT NONE

INTEGER :: i,j

i = 1
j = 2*i

WRITE(*,*) i,j

END PROGRAM simple_program
```

---

**Declaration section**

```
IMPLICIT NONE

INTEGER :: i,j
```

**Executable section**

```
i = 1
j = 2*i

WRITE(*,*) i,j
```

**Termination section**

```
END PROGRAM simple_program
```
Structure of Simple Fortran Program

```fortran
PROGRAM simple_program

IMPLICIT NONE

INTEGER :: i, j

i = 1
j = 2*i

WRITE(*,*) i, j

END PROGRAM simple_program
```

Program name can consist of up to 31 letters, digits, and underscores, that is, A-Z, a-z, 0-9, _

Program name must start with letter

Fortran is case-insensitive! So

simple_program
SIMPLE_PROGRAM
sIMple_pRoGRaM
are all equivalent
Structure of Simple Fortran Program

PROGRAM simple_program

IMPLICIT NONE

INTEGER :: i,j

i = 1
j = 2*i

WRITE(*,*) i,j

END PROGRAM simple_program

Don’t worry about this for now, we’ll deal with it later
PROGRAM simple_program

IMPLICIT NONE

INTEGER :: i,j

i = 1
j = 2*i

WRITE(*,*) i,j

END PROGRAM simple_program

Fortran commands, variables, and constants are also case-insensitive!

Could just as well have written

integer :: I,J
InTEgEr :: I,j

Recommend following convention:
• Fortran commands: uppercase
• Program names: mixed case
• Variable names: mixed case

Variable names must not coincide with Fortran commands and procedures
Structure of Simple Fortran Program

PROGRAM simple_program

IMPLICIT NONE

INTEGER :: i, j

i = 1
j = 2*i

WRITE(*,*) i, j

END PROGRAM simple_program

Lines may be up to 132 characters long

Recommendation: Restrict to smaller number (say, 80) which can be printed on A4 paper with normal font size

If line is longer than your maximum length, use continuation character & and continue on next line

The maximum number of continuation lines is 39
Structure of Simple Fortran Program

PROGRAM simple_program

IMPLICIT NONE

INTEGER :: i,j

i = 1
j = 2*i

WRITE(*,*) i,j

END PROGRAM simple_program

Fortran ignores empty lines

Recommendation: Use them to make your programs more readable
Structure of Simple Fortran Program

PROGRAM simple_program

IMPLICIT NONE

INTEGER :: i,j

i = 1
j = 2*i

WRITE(*,*) i,j

END PROGRAM simple_program
PROGRAM simple_program

IMPLICIT NONE

INTEGER :: i,j

i = 1
j = 2*i

WRITE(*,*) i,j

END PROGRAM simple_program

In a simple program like this one, comments are not really necessary

But in more complicated ones, comments are very important

In Fortran, comments start with !
Structure of Simple Fortran Program

PROGRAM simple_program

IMPLICIT NONE

INTEGER :: i,j

! Define i
i = 1

j = 2*i ! Compute j

WRITE(*,*) i,j

END PROGRAM simple_program

In a simple program like this one, comments are not really necessary

But in more complicated ones, comments are very important

In Fortran, comments start with !

Comments can be placed anywhere, even on same line as statements
Overview

• Course overview
• Structure of simple Fortran program
• Compiling and linking
• Intrinsic data types
• Operations
• Intrinsic procedures
• Control flow
• Input/output
Compiling & Linking

simple_code.f90

Source code/source file
Compiling & Linking

- **simple_code.f90**
  - Source code/source file
- **simple_code.o**
  - Object code/object file

Compiling
Compiling & Linking

simple_code.f90

Source code/source file

Compiling

simple_code.o

Object code/object file

Linking

simple_code

simple_code.exe

Executable/executable file
Compiling & Linking

simple_code.f90

```
gfortran -c simple_code.f90 -o simple_code.o
```

simple_code.o

```
gfortran simple_code.o -o simple_code
```

simple_code
Compiling & Linking

```
simple_code.f90

simple_code.o

gfortran -c simple_code.f90 -o simple_code.o

gfortran simple_code.o -o simple_code
```

Compiler option indicating “compile only”

Compiler option indicating name of output

Many additional compiler options are available
We’ll talk about some of them later
Compiling & Linking

```
gfortran simple_code.f90 -o simple_code
```

simple_code.f90

simple_code
Niklaus Wirth (*1934)
• Swiss computer scientist
• Degree from ETH in 1959
• Professor at ETH from 1968-1999

Quote perfectly encapsulates two central components of a program:
› Data (and how it is stored)
› Algorithms/methods/operations that manipulate data
Overview

- Course overview
- Structure of simple Fortran program
- Compiling and linking
- **Intrinsic data types**
- Operations
- Intrinsic procedures
- Control flow
- Input/output
Intrinsic Data Types

• Fortran provides six intrinsic data types:
  ‣ INTEGER
  ‣ REAL
  ‣ DOUBLE PRECISION
  ‣ COMPLEX
  ‣ LOGICAL
  ‣ CHARACTER

• Each can be used to declare variables or constants
• Can also define your own data types
Intrinsic Data Types: INTEGER

- Integer variables declared with:
  \[
  \text{INTEGER :: } i, j, k \\
  \]
  \[
  i = 1 \\
  j = 2 \\
  k = i + j \\
  \]

- Integer constants declared with:
  \[
  \text{INTEGER, PARAMETER :: NEDGES_MAX = 1000} \\
  \]

- Constants (of any data type) often called named constants
- Constants (of any data type) cannot be changed after the declaration

Recommendation:
- Use uppercase names for \textsc{parameter}s
- Use underscores to separate words
Intrinsic Data Types: INTEGER

• Q: Why use named constants?
Intrinsic Data Types: INTEGER

- Computers represent data as strings of bits using binary number system
- Each bit assumes values 0 or 1
- Number of bits in string therefore determines range of values that can be represented
- Assuming 8-bit representation:
  \[ \begin{align*}
  00000010 &= 1 \times 2^1 + 0 \times 2^0 = 2_2 \\
  00100011 &= 1 \times 2^5 + 1 \times 2^1 + 1 \times 2^0 = 35_2
  \end{align*} \]
- Minimum and maximum number that could be represented:
  \[ \begin{align*}
  00000000 &= 0_2 \\
  11111111 &= 255_2
  \end{align*} \]
- With n bits, can represent numbers ranging from 0 to \(2^{n-1}\)
Intrinsic Data Types: INTEGER

- Q: How to represent negative integers?
- A: Reserve one bit to represent sign
- Thus can represent numbers ranging from $-2^{n-1}$ to $2^{n-1}-1$
- Fortran uses 32 bits for INTEGERs, thus can represent numbers ranging from -2147483648 to 2147483647
- Very important: Range of representable numbers is limited
- Exceeding the range leads to integer overflow or underflow
Intrinsic Data Types: REAL

- Computers represent floating-point numbers:
  \[ 132.857 = +0.132857E+3 \]
  - Sign
  - Mantissa
  - Exponent

- Range of real numbers that can be represented depends on numbers of bits used for mantissa and exponent

- Fortran uses 32 bits for REALs:
  - 1 bit for sign
  - 23 bits for mantissa
  - 8 for exponent

- Most floating-point numbers are rounded, so not exact

- Largest number: \[ 3.40282347E+38 \]

- Machine precision: \[ 1.19209290E-07 \]
Intrinsic Data Types: DOUBLE PRECISION

• If **REAL** range too small or machine precision too large, need **DOUBLE PRECISION** representation

• Fortran uses 64 bits for **DOUBLE PRECISION**:
  ‣ 1 bit for sign
  ‣ 52 bits for mantissa
  ‣ 11 for exponent

• Largest number: \(1.7976931348623157 \times 10^{308}\)

• Machine precision: \(2.2204460492503131 \times 10^{-16}\)
Intrinsic Data Types: COMPLEX

• Convenient way to represent complex numbers:

\[
\text{COMPLEX } :: \ c, d
\]

\[
c = (1.0, -2.0) \quad \text{Represents 1-2i}
\]

\[
d = \text{CMPLX}(-3, 4) \quad \text{Represents -3+4i}
\]

• Fortran provides intrinsic procedures for complex numbers, such as magnitudes, complex conjugates, etc.
Intrinsic Data Types: LOGICAL

• Logical variables declared with:
  
  LOGICAL :: testPassed, resultNegative

• Logical variables and constants can only take two values:
  
  testPassed = .TRUE.
  resultNegative = .FALSE.

• Logical constants can be declared with:
  
  LOGICAL, PARAMETER :: flagTrue = .TRUE.

• Use of logical constants limited
Intrinsic Data Types: CHARACTER

- Single characters are declared with

  \[
  \text{CHARACTER :: c}
  \]

  \[
  \begin{align*}
  c &= 'a' \\
  c &= "b" \\
  c &= '&'
  \end{align*}
  \]

  Can initialise characters with single or double quotes

- Single characters are not, of course, very useful
Intrinsic Data Types: CHARACTER

• To represent words or phrases, use strings of characters
• Example of declaration of string:
  CHARACTER(LEN=11) :: c

  c = “hello world”  Ok to initialise to string with less than 11 characters, but not more

• Careful with apostrophes:
  c = ‘what’s up?’  Gives compiler error (unbalanced quotes)
  c = “what’s up?”
  c = ‘what’’s up?’  Both ok

• Can also use shorter declaration
  CHARACTER(11) :: c
Intrinsic Data Types: CHARACTER

• Declaration of character string parameters:

```fortran
CHARACTER(6), PARAMETER :: ERROR_MESSAGE = 'ERROR!'  
INTEGER :: i, j

WRITE(*,*) 'Enter positive integer:'
READ(*,*) i
WRITE(*,*) 'Enter negative integer:'
READ(*,*) j

IF ( i < 0 ) THEN
  WRITE(*,*) ERROR_MESSAGE
  STOP
END IF

IF ( j > 0 ) THEN
  WRITE(*,*) ERROR_MESSAGE
  STOP
END IF
```

Definition of character string parameter useful if used repeatedly

If string needs to be changed, only one change is necessary
In older versions of Fortran, *implicit declaration* possible:

- Variables starting with `i`, `j`, `k`, `l`, `m`, `n` were integers by default
- All other variables were real variables by default
- Backward compatibility means can still be used

Now considered dangerous because it masks typing mistakes:

```fortran
SUBROUTINE MakeVectorUnitVector(b)
  REAL, INTENT(INOUT) :: b(2)
  REAL :: length
  length = SQRT(b(1)*b(1)+b(2)*b(2))
  b(1) = b(1)/length
  b(2) = b(2)/length
END SUBROUTINE MakeVectorUnitVector
```

Hence recommend that always use *IMPLICIT NONE*
Intrinsic Data Types

- Single variables are not very useful
- Often collection of variables in arrays convenient/needed
- Examples:
  - Solution of linear system of equations
  - Finite-difference/volume/element solutions of PDEs
  - ...

PREC
Professorship of Renewable Energy Carriers
Intrinsic Data Types: INTEGER Arrays

• Declaration and initialization of 1d INTEGER arrays:

\[
\begin{align*}
\text{INTEGER} & : : \ a(4) \\
\text{INTEGER, DIMENSION(4)} & : : \ b
\end{align*}
\]

\[
\begin{align*}
a(1) & = 1 \\
a(2) & = -4 \\
a(3) & = 37 \\
a(4) & = -121
\end{align*}
\]

\[a = (/1,-4,37,-121/)\]

\[b(1:4) = 0\]
Intrinsic Data Types: INTEGER Arrays

- By default, Fortran array elements are indexed 1, 2, ...
- Can be changed by suitable declaration and initialization:
  
  \[
  \text{INTEGER :: a}(-1:2) \\
  \text{INTEGER, DIMENSION (0:3) :: b}
  \]

Equivalent initializations of array \( a \)

\[
\begin{align*}
  a(-1) &= 1 \\
  a(0) &= -4 \\
  a(1) &= 37 \\
  a(2) &= -121
\end{align*}
\]

This initialization to a constant value works for any numbering

\[
a = (/1,-4,37,-121/)
\]

\[
b(:) = 0
\]
Intrinsic Data Types: INTEGER Arrays

• Declaration and initialization of 2d INTEGER arrays is similar:

```
INTEGER :: a(2,3)
INTEGER, DIMENSION(4,2) :: b

a(1,1) = 1
a(2,1) = -4
a(1,2) = 37
a(2,2) = -121
a(1,3) = 456
a(2,3) = -17

b(:, :) = 0
```
Intrinsic Data Types: INTEGER Arrays

- Fortran stores multi-dimensional arrays in **column-major order**:

  ```fortran
  INTEGER :: a(2,3)
  a(1,1) = 1
  a(2,1) = -4
  a(1,2) = 37
  a(2,2) = -121
  a(1,3) = 456
  a(2,3) = -17
  ```

- In general: leftmost subscript varies fastest
- This has important performance implications
- Maximum number of array dimensions: 7
Intrinsic Data Types: Other Arrays

- Arrays for other intrinsic data types defined similarly
- Careful with arrays of character strings:

  \[
  \text{CHARACTER(20)} :: a(80)
  \]

An array of 80 strings, each of which is 20 characters long
Arrays: Stepping Out-of-Bounds

- Stepping out of bounds of an array is common mistake:
  
  ```fortran
  PROGRAM bounds1d

  INTEGER :: a(4)

  a = (/1,-4,37,-121/)

  WRITE(*,*) a(0)

  END PROGRAM bounds1d
  ```

Compile with:
```
gfortran bounds1d.f90 -o bounds1d
```

Compiler produces executable, but also issues warning:
```
bounds1d.f90:10.15:
  WRITE(*,*) a(0)
  1
Warning: Array reference at (1) is out of bounds (0 < 1) in dimension 1
```
Arrays: Stepping Out-of-Bounds

• What about following program?

```fortran
PROGRAM bounds1d

IMPLICIT NONE

INTEGER :: i
INTEGER :: a(4)

a = (/1,-4,37,-121/)

WRITE(*,*) 'Enter i:'
READ(*,*) i

WRITE(*,*) a(i)

END PROGRAM bounds1d
```

Compiler cannot issue warning, as index `i` is unknown at compile time
Arrays: Stepping Out-of-Bounds

- What happens when you run this program?

<table>
<thead>
<tr>
<th>i</th>
<th>a(i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>0</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>-4</td>
</tr>
<tr>
<td>3</td>
<td>37</td>
</tr>
<tr>
<td>4</td>
<td>-121</td>
</tr>
<tr>
<td>5</td>
<td>1475824016</td>
</tr>
<tr>
<td>6</td>
<td>32767</td>
</tr>
</tbody>
</table>

Undefined behaviour if $i < 1$ or $i > 4$

“Undefined” means that behaviour may be different on different machines, with different compilers, and different each time you run program (even on same machine and with same compiler)
Arrays: Stepping Out-of-Bounds

- Compiler can help you avoid this problem
- By compiling with
  `gfortran -fbounds-check bounds1d.f90 -o bounds1d`
  get following behavior:
  
Enter i:
  0
At line 14 of file bounds1d.f90
Fortran runtime error: Index '0' of dimension 1 of array 'a'
below lower bound of 1

- Bottom line: Compiler can help you detect and find mistakes
- But you need to tell compiler to do so with appropriate options
Recommended Compiler Options

- Recommendation: while developing program, turn on all compiler options that help detect and find mistakes
- For example:

<table>
<thead>
<tr>
<th>Option</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>-g</td>
<td>Produce information for debugger</td>
</tr>
<tr>
<td>-Wall -Wextra</td>
<td>Issue (lots of) warnings</td>
</tr>
<tr>
<td>-fbounds-check</td>
<td>Check stepping out of bounds of arrays</td>
</tr>
<tr>
<td>-fbacktrace</td>
<td>Produce list of procedures called when program crashes</td>
</tr>
<tr>
<td>-ffpe-trap=zero,overflow,underflow,invalid</td>
<td>Trap division by zero, overflow, underflow, invalid operations</td>
</tr>
</tbody>
</table>

- Full list of warning options:
  gfortran --help=warnings
- These options will slow down executable, ok during development and testing
Once program working correctly, use compiler options to speed up execution:

```
gfortran -On code.f90 -o code
```

where \( n \) is number from 0 (no optimization) to 3 (most aggressive optimization)

Check that optimization does not influence results

Bottom line:

- Compiler great help in code development; make use of it
- Familiarize yourself with options of your compiler(s)
Overview

- Course overview
- Structure of simple Fortran program
- Compiling and linking
- Intrinsic data types
- Operations
- Intrinsic procedures
- Control flow
- Input/output
Arithmetic Operations

- **Binary operations:**
  - = Assignment
    - a = b
    - i = i + 1
  - + Addition
    - a + b
  - - Subtraction
    - a - b
  - / Division
    - a/b
  - * Multiplication
    - a*b
  - ** Exponentiation
    - a**b

- **Unary operations:**
  - + ?
    - a = +b
  - - Negation
    - a = -b

- **Precedence (top to bottom):**
  - ** left to right
  - * / left to right
  - + - left to right

- **Use parentheses to influence precedence:**
  - a = (a+b)**n/(c*(a-b/(e+f)))
**Arithmetic Operations: Integer/Mixed Arithmetic**

- Careful with Fortran’s integer arithmetic, i.e., arithmetic involving only integers
- Fortran’s integer arithmetic always produces integer result by truncating fractional part
- Examples:
  - \(3/4 = 0\)
  - \(4/3 = 1\)
  - \(5/3 = 1\)
  - \(6/3 = 2\)
- Looks like major inconvenience, but can be useful at times
To divide two integers, change at least one to a real number to get a real result:

\[
\begin{align*}
3/4 & = 0 \\
3.0/4 & = 0.75 \\
3/4.0 & = 0.75
\end{align*}
\]

or

\[
\begin{align*}
i/\text{REAL}(j) & \quad i/\text{DBLE}(j) \\
\text{REAL}(i)/j & \quad \text{DBLE}(i)/j
\end{align*}
\]

In mixed (or mixed-mode) arithmetic, Fortran converts integers to real numbers:

\[
\begin{align*}
1 + 3/4 & = 1 \\
1 + 3.0/4 & = 1.75 \\
1.0 + 3/4 & = 1.0
\end{align*}
\]

Recommendation: Avoid mixed arithmetic by converting integers yourself using \texttt{REAL} and \texttt{DBLE} intrinsic procedures
Relational Operations

- Binary operations:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>equal to</td>
<td>a == b</td>
</tr>
<tr>
<td>/=</td>
<td>not equal to</td>
<td>a /= b</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>a &gt; b</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
<td>a &gt;= b</td>
</tr>
<tr>
<td>&lt;</td>
<td>smaller than</td>
<td>a &lt; b</td>
</tr>
<tr>
<td>&lt;=</td>
<td>smaller than or equal to</td>
<td>a &lt;= b</td>
</tr>
</tbody>
</table>

- Precedence: Left to right

- Careful when comparing real numbers!
Logical Operations

• Can only be used with logical variables and constants
• Binary operations:

<table>
<thead>
<tr>
<th>.AND.</th>
<th>logical AND</th>
</tr>
</thead>
<tbody>
<tr>
<td>.OR.</td>
<td>logical OR</td>
</tr>
<tr>
<td>.EQV.</td>
<td>logical equivalence</td>
</tr>
<tr>
<td>.NEQV.</td>
<td>logical nonequivalence</td>
</tr>
</tbody>
</table>

• Unary operation:

<table>
<thead>
<tr>
<th>.NOT.</th>
<th>logical NOT</th>
</tr>
</thead>
</table>
## Logical Operations

- **Truth table:**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A .AND. B</th>
<th>A .OR. B</th>
<th>A .EQV. B</th>
<th>A .NEQV. B</th>
</tr>
</thead>
<tbody>
<tr>
<td>.FALSE.</td>
<td>.FALSE.</td>
<td>.FALSE.</td>
<td>.FALSE.</td>
<td>.TRUE.</td>
<td>.FALSE.</td>
</tr>
<tr>
<td>.FALSE.</td>
<td>.TRUE.</td>
<td>.FALSE.</td>
<td>.TRUE.</td>
<td>.FALSE.</td>
<td>.TRUE.</td>
</tr>
<tr>
<td>.TRUE.</td>
<td>.FALSE.</td>
<td>.FALSE.</td>
<td>.TRUE.</td>
<td>.FALSE.</td>
<td>.TRUE.</td>
</tr>
<tr>
<td>.TRUE.</td>
<td>.TRUE.</td>
<td>.TRUE.</td>
<td>.TRUE.</td>
<td>.TRUE.</td>
<td>.FALSE.</td>
</tr>
</tbody>
</table>

- **Precedence (top to bottom):**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>.NOT.</td>
<td>left to right</td>
</tr>
<tr>
<td>.AND.</td>
<td>left to right</td>
</tr>
<tr>
<td>.OR.</td>
<td>left to right</td>
</tr>
<tr>
<td>.EQV.</td>
<td>left to right</td>
</tr>
<tr>
<td>.NEQV.</td>
<td>left to right</td>
</tr>
</tbody>
</table>

.**NEQV.** is equivalent to “exclusive or”
Logical Operations: Example of a Mistake

• What I wanted the program to be:

```fortran
IF ( (flag1 .EQV. .FALSE.) .AND. (flag2 .EQV. .FALSE.) ) THEN
   Fortran statements
END IF
```

• What I actually typed:

```fortran
IF ( flag1 .EQV. .FALSE. .AND. flag2 .EQV. .FALSE. ) THEN
   Fortran statements
END IF
```

• Precedence rules mean this is equivalent to:

```fortran
IF ( (flag1 .EQV. (.FALSE. .AND. flag2)) .EQV. .FALSE. ) THEN
   Fortran statements
END IF
```

<table>
<thead>
<tr>
<th>flag1</th>
<th>flag2</th>
<th>Desired expression</th>
<th>Actual expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>.FALSE.</td>
<td>.FALSE.</td>
<td>.TRUE.</td>
<td>.FALSE.</td>
</tr>
<tr>
<td>.FALSE.</td>
<td>.TRUE.</td>
<td>.FALSE.</td>
<td>.FALSE.</td>
</tr>
<tr>
<td>.TRUE.</td>
<td>.FALSE.</td>
<td>.FALSE.</td>
<td>.TRUE.</td>
</tr>
<tr>
<td>.TRUE.</td>
<td>.TRUE.</td>
<td>.FALSE.</td>
<td>.TRUE.</td>
</tr>
</tbody>
</table>
Overview

- Course overview
- Structure of simple Fortran program
- Compiling and linking
- Intrinsic data types
- Operations
- **Intrinsic procedures**
- Control flow
- Input/output
Intrinsic procedures

• Fortran provides 108 intrinsic procedures to operate on data
• These include trigonometric procedures, for example
• We will later learn how to write our own procedures
• Following covers only most important intrinsic procedures
• For complete list, see:
  ‣ Chapman (2008), Appendix B
  ‣ Ellis et al. (1994), Appendix A
  ‣ Metcalf et al. (2004), Appendix A
Intrinsic procedures: Integer data

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS(x)</td>
<td>Absolute value of x</td>
</tr>
<tr>
<td>DBLE(x)</td>
<td>Converts x to double-precision floating-point value</td>
</tr>
<tr>
<td>MAX(a, b)</td>
<td>Picks the larger of a and b</td>
</tr>
<tr>
<td>MIN(a, b)</td>
<td>Picks the smaller of a and b</td>
</tr>
<tr>
<td>MOD(a, b)</td>
<td>Remainder (modulo) function</td>
</tr>
<tr>
<td>REAL(x)</td>
<td>Converts x to single-precision floating-point value</td>
</tr>
</tbody>
</table>
### Intrinsic procedures: Floating-point data

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS((x))</td>
<td>Absolute value of (x)</td>
</tr>
<tr>
<td>ACOS((x))</td>
<td>Inverse cosine of (x) result in radians</td>
</tr>
<tr>
<td>ACOSH((x))</td>
<td>Inverse hyperbolic cosine of (x)</td>
</tr>
<tr>
<td>ASIN((x))</td>
<td>Inverse sine of (x) result in radians</td>
</tr>
<tr>
<td>ASINH((x))</td>
<td>Inverse hyperbolic sine of (x)</td>
</tr>
<tr>
<td>ATAN((x))</td>
<td>Inverse tangent of (x) result in radians, (-\pi/2) to (\pi/2)</td>
</tr>
<tr>
<td>ATAN2((y, x))</td>
<td>Inverse tangent of (y/x) result in radians, (-\pi) to (\pi)</td>
</tr>
<tr>
<td>ATANH((x))</td>
<td>Inverse hyperbolic tangent of (x)</td>
</tr>
<tr>
<td>CMPLX((a, b))</td>
<td>Convert (a, b) to complex number (a+bi)</td>
</tr>
<tr>
<td>COS((x))</td>
<td>Cosine of (x) (x) in radians</td>
</tr>
<tr>
<td>COSH((x))</td>
<td>Hyperbolic cosine of (x)</td>
</tr>
<tr>
<td>EXP((x))</td>
<td>Exponential of (x)</td>
</tr>
<tr>
<td>INT((x))</td>
<td>Truncates (x) to integer</td>
</tr>
</tbody>
</table>
## Intrinsic procedures: Floating-point data

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOG(x)</td>
<td>Natural logarithm of x</td>
<td>x&gt;0</td>
</tr>
<tr>
<td>LOG10(x)</td>
<td>Base-10 logarithm of x</td>
<td>x&gt;0</td>
</tr>
<tr>
<td>MAX(a,b)</td>
<td>Picks the larger of a and b</td>
<td></td>
</tr>
<tr>
<td>MIN(a,b)</td>
<td>Picks the smaller of a and b</td>
<td></td>
</tr>
<tr>
<td>MOD(a,b)</td>
<td>Remainder (modulo) function</td>
<td></td>
</tr>
<tr>
<td>NINT(x)</td>
<td>Nearest integer to x</td>
<td></td>
</tr>
<tr>
<td>SIGN(a,b)</td>
<td>Value of a with sign of b</td>
<td></td>
</tr>
<tr>
<td>SIN(x)</td>
<td>Sine of x</td>
<td>x in radians</td>
</tr>
<tr>
<td>SINH(x)</td>
<td>Hyperbolic sine of x</td>
<td></td>
</tr>
<tr>
<td>SQRT(x)</td>
<td>Square root of x</td>
<td>x≥0</td>
</tr>
<tr>
<td>TAN(x)</td>
<td>Tangent of x</td>
<td>x in radians</td>
</tr>
<tr>
<td>TANH(x)</td>
<td>Hyperbolic tangent of x</td>
<td></td>
</tr>
</tbody>
</table>
### Intrinsic procedures: Complex data

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS (x)</td>
<td>Magnitude of x</td>
</tr>
<tr>
<td>AIMAG (x)</td>
<td>Imaginary part of x</td>
</tr>
<tr>
<td>CONJG (x)</td>
<td>Complex conjugate of x</td>
</tr>
<tr>
<td>REAL (x)</td>
<td>Real part of x</td>
</tr>
</tbody>
</table>
Intrinsic procedures: Character string data

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEN(x)</td>
<td>Length of x</td>
</tr>
<tr>
<td>LEN_TRIM(x)</td>
<td>Length of x, not counting trailing blank characters</td>
</tr>
<tr>
<td>TRIM(x)</td>
<td>Remove trailing blank characters</td>
</tr>
</tbody>
</table>
# Intrinsic procedures: Arrays

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOT_PRODUCT(a,b)</td>
<td>Dot product of vectors $a$ and $b$</td>
<td>$a$ and $b$ must be conforming</td>
</tr>
<tr>
<td>LBOUND(a,i)</td>
<td>Lower bound of $i$th dimension of $a$</td>
<td></td>
</tr>
<tr>
<td>MATMUL(a,b)</td>
<td>Matrix product of $a$ and $b$</td>
<td>$a$ and $b$ must be conforming</td>
</tr>
<tr>
<td>MAXLOC(a,m)</td>
<td>Location of first element of maximum value in $a$ given mask $m$</td>
<td>See any of the books for more information</td>
</tr>
<tr>
<td>MAXVAL(a,m)</td>
<td>Maximum value of elements in $a$ given mask $m$</td>
<td>See any of the books for more information</td>
</tr>
<tr>
<td>MINLOC(a,m)</td>
<td>Location of first element of minimum value in $a$ given mask $m$</td>
<td>See any of the books for more information</td>
</tr>
<tr>
<td>MINVAL(a,m)</td>
<td>Minimum value of elements in $a$ given mask $m$</td>
<td>See any of the books for more information</td>
</tr>
<tr>
<td>SIZE(a,i)</td>
<td>Size of $a$ along $i$th dimension</td>
<td></td>
</tr>
</tbody>
</table>
# Intrinsic procedures: Arrays

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUM(a,i,m)</td>
<td>Sum of elements in $a$ along dimension $i$ given mask $m$</td>
<td>See any of the books for more information</td>
</tr>
<tr>
<td>TRANSPOSE(a)</td>
<td>Transpose of $a$</td>
<td></td>
</tr>
<tr>
<td>UBOUND(a,i)</td>
<td>Upper bound of $i$th dimension of $a$</td>
<td></td>
</tr>
</tbody>
</table>
Overview

• Course overview
• Structure of simple Fortran program
• Compiling and linking
• Intrinsic data types
• Operations
• Intrinsic procedures
• Control flow
• Input/output
Control Flow

• Often necessary to:
  ‣ Take action in response to data
  ‣ Repeat actions multiple times
• Both amount to “controlling the flow” in the program
Control Flow: IF Statements

- General case ("block IF"): 
  
  ```fortran
  IF ( logical_expression ) THEN 
    Fortran statements 
  ELSE IF ( logical_expression ) THEN 
    Fortran statements 
  ELSE IF ( logical_expression ) THEN 
    Fortran statements 
  ELSE 
    Fortran statements 
  END IF 
  ```

  There can be any number of ELSE IF statements (including none)

  Important: First true expression is taken

  There can be at most one ELSE statement

  It must appear after the ELSE IF statements (if there are any)

- Special case ("logical IF"): 
  
  ```fortran
  IF ( logical_expression ) Fortran statement 
  ```
Control Flow: IF Statements

• IF statements can be nested:

```
IF ( logical_expression ) THEN
    IF ( logical_expression ) THEN
        Fortran statements
    ELSE IF
        IF ( logical_expression ) THEN
            Fortran statements
        ELSE
            Fortran statements
        END IF
    END IF
END IF
ELSE IF ( logical_expression ) THEN
    Fortran statements
ELSE
    Fortran statements
END IF
```
Control Flow: IF Statements

- IF statements can be nested:

  ```fortran
  outerIF: IF ( logical_expression ) THEN
   middleIF: IF ( logical_expression ) THEN
    Fortran statements
   ELSE IF
    innerIF: IF ( logical_expression ) THEN
     Fortran statements
    ELSE
     Fortran statements
    END IF innerIF
   ELSE IF ( logical_expression ) THEN
    Fortran statements
   ELSE
    Fortran statements
   END IF middleIF
  ELSE IF ( logical_expression ) THEN
   Fortran statements
  ELSE
   Fortran statements
  END IF outerIF
  ```

  Fortran 95 allows you to name IF statements

  Benefits:
  1. Easier for people to understand nested IFs
  2. If you accidentally delete an IF or END IF, compiler can treat it as an error
Control Flow: SELECT CASE Statement

SELECT CASE ( case_expression )
CASE ( case_selector_1 )
  Fortran statements
CASE ( case_selector_2 )
  Fortran statements
CASE ( case_selector_3 )
  Fortran statements
CASE DEFAULT
  Fortran statements
END SELECT

There must be at least one CASE statement.
If there is more than one CASE statement, the caseSelectors must be mutually exclusive.
The case selectors must be constants (integer, character, or logical).
The statements under CASE DEFAULT are executed if none of the other CASE statements are executed.
The CASE DEFAULT statement is not required.
It is recommended that you include it to catch programming mistakes.
Control Flow: SELECT CASE Statement

Example:

SELECT CASE( TRIM(line) )
  CASE ('# BC_FARF')
    CALL RFLU_ReadBcFarfSection(pRegion)
  CASE ('# BC_INFLOW_TOTANG')
    CALL RFLU_ReadBcInflowTotAngSection(pRegion)
  CASE ('# BC_INFLOW_VELTEMP')
    CALL RFLU_ReadBcInflowVelTempSection(pRegion)
  CASE ('# BC_INJECT')
    CALL RFLU_ReadBcInjectSection(pRegion)
  ...
  CASE DEFAULT
    CALL ErrorStop(global,ERR_REACHED_DEFAULT)
END SELECT ! TRIM(line)
Control Flow: SELECT CASE Statement

Example:

```
SELECT CASE ( flag )
    CASE ( 1,2 )
        DO iPatch = 1,pGrid%nPatches
            pPatch => pRegion%patches(iPatch)
        ...
        END DO ! iPatch
    CASE ( 3 )
        DO iPatch = 1,pGrid%nPatches
            pPatch => pRegion%patches(iPatch)
        ...
        END DO ! iPatch
    CASE DEFAULT
        CALL ErrorStop(global,ERR_REACHED_DEFAULT)
    END SELECT ! flag
```

- `case_selector` can consist of more than one value, separated by commas.
- `case_selector` can also specify range of values:

  ```
  CASE ( val_min:val_max )
  ```
Control Flow: SELECT CASE or IF?

<table>
<thead>
<tr>
<th>IF</th>
<th>SELECT CASE</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>logical_expression</em> may involve <em>variables</em></td>
<td><em>case_selector</em> must only involve <em>constants</em></td>
</tr>
<tr>
<td>Order of <strong>ELSE</strong> <strong>IFs</strong> <strong>does</strong> matter</td>
<td>Order of <strong>CASEs</strong> <strong>does not</strong> matter</td>
</tr>
</tbody>
</table>
Defensive Programming

- Recommendation: always include `CASE DEFAULT` statement
- If it should not be reached, write error message if reached
- Example of defensive programming
Defensive Programming

• McConnell (2004, p. 187):
  “The idea is based on defensive driving. In defensive driving, you adopt
  the mind-set that you’re never sure what other drivers are going to do.
  That way, you make sure that if they do something dangerous you won’t
  be hurt. You take responsibility for protecting yourself even when it
  might be the other driver’s fault. In defensive programming, the main
  idea is that if a routine is passed bad data, it won’t be hurt, even if the
  bad data is another routine’s fault. More generally, it is the recognition
  that programs will have problems and modifications, and that a smart
  programming will develop code accordingly.”

• My take:
  ‣ Anticipate problems (Murphy’s law…)
  ‣ Write code to handle problems (not to fix them)
  ‣ Write error messages to let you know problem occurred
KISS Principle

• KISS = Keep it simple, stupid
• Simple programs are easier to write, debug, and modify
• Often bad idea to use elegant code that is hard to understand
• Not uncommon to have difficulty understanding code that was written only a couple of weeks/months ago
• Bottom line: Write simple programs
• Does not mean that simple programs are inefficient
Control Flow: DO Statements

- **DO-loops** are used to repeat operations

- **First form of DO-loop:**

  ```fortran
  DO i = i_beg, i_end, i_inc
  Fortran statements
  END DO
  ```

  which is equivalent to:

  ```fortran
  i = i_beg
  Fortran statement
  i = i + i_inc
  Fortran statement
  ...
  i = i_end
  Fortran statement
  ```

- **Known as “iterative” or “counting” or “count-controlled” loop**

  - *i* “loop counter”, must be integer
  - *i_beg* must be integer expression
  - *i_end* must be integer expression
  - *i_inc* “loop increment” (1 if not specified)
  - *i, i_beg, i_end* must not be changed in loop
Control Flow: DO Statements

- Sometimes, need to repeat statements until condition is met, but number of repetitions is not known
- Second form of DO-loop:

  ```fortran
  DO
    Fortran statements
    IF ( logical expression ) THEN
      Fortran statements
      EXIT Causes control to be transferred to first statement after END DO
    END IF
    Fortran statements
  END DO
  ```

- Sometimes referred to as “while” loop
- Note that if `logical expression` is never true, get so-called infinite loop…
Defensive Programming

- Avoid infinite loops by introducing loop counters:

```fortran
INTEGER, PARAMETER :: MAX_LOOP_COUNTER = 1000

loopCounter = 0
DO
  loopCounter = loopCounter + 1
  ! Fortran statements
  IF (logical expression) THEN
    ! Fortran statements
    EXIT
  END IF
  IF (loopCounter == MAX_LOOP_COUNTER) THEN
    EXIT
  END IF
  ! Fortran statements
END DO
```

Maximum value of loop counter must be chosen large enough.
Defensive Programming

- Problem: Do not why loop was exited
- Checking `loopCounter` is a possible solution
- Another solution shown on next slide…
Defensive Programming

• Solution:

  LOGICAL :: flagReachedMaxCounter
  DO
    loopCounter = loopCounter + 1
    Fortran statements
    IF ( logical expression ) THEN
      Fortran statements
      flagReachedMaxCounter = .FALSE.
      EXIT
    END IF
    IF ( loopCounter == MAX_LOOP_COUNTER ) THEN
      flagReachedMaxCounter = .TRUE.
      EXIT
    END IF
    Fortran statements
  END DO
Defensive Programming

• Introduce check after loop:

...  

IF ( loopCounter == MAX_LOOP_COUNTER ) THEN  
  flagReachedMaxCounter = .TRUE.  
  EXIT  
END IF  

Fortran statements  
END DO  

IF ( flagReachedMaxCounter .EQV. .TRUE. ) THEN  
  WRITE(*,*) 'WARNING! Maximum loop counter reached!'  
END IF
Control Flow: DO Statements

- Fortran allows DO loops to be named:

```fortran
LOGICAL :: flagReachedMaxCounter
whileLoop: DO
    loopCounter = loopCounter + 1
    Fortran statements
    IF ( logical expression ) THEN
        Fortran statements
        flagReachedMaxCounter = .FALSE.
        EXIT whileLoop
    END IF
    IF ( loopCounter == MAX_LOOP_COUNTER ) THEN
        flagReachedMaxCounter = .TRUE.
        EXIT whileLoop
    END IF
END DO whileLoop
```
Control Flow: **DO Statements**

- Already discussed **EXIT** statement, which transfer control to first statement **after** **END** **DO**
- There is also the **CYCLE** statement, which transfers control **to** **END** **DO** statement (and possibly back to **DO** statement)
Overview

- Course overview
- Structure of simple Fortran program
- Compiling and linking
- Intrinsic data types
- Operations
- Intrinsic procedures
- Control flow
- Input/output
Input/Output

- Programs require input data and generate output data
- Four possibilities:

<table>
<thead>
<tr>
<th></th>
<th>Screen</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>Read from screen</td>
<td>Read from file</td>
</tr>
<tr>
<td>Write</td>
<td>Write to screen</td>
<td>Write to file</td>
</tr>
</tbody>
</table>
Output: To Screen

- Writing an integer to screen is accomplished by:
  
  ```fortran
  INTEGER :: i
  i = 5
  WRITE(*,*) i
  ```

- Similar for floating-point variables

- Writing character string accomplished by:
  
  ```fortran
  INTEGER, PARAMETER :: MAX_CHAR_LEN = 80
  CHARACTER(MAX_CHAR_LEN) :: c
  c = 'Hello!'
  WRITE(*,*) c
  ```
  
  This writes _whole_ string, including trailing blank spaces

- This can be avoided with intrinsic procedure TRIM:
  
  ```fortran
  WRITE(*,*) TRIM(c)
  ```
Output: To Screen

- Can combine writing of data types:

  INTEGER, PARAMETER :: MAX_CHAR_LEN = 80
  INTEGER :: i
  CHARACTER(MAX_CHAR_LEN) :: c
  i = 5
  c = ‘Hello!’
  WRITE(*,*) TRIM(c),i

  or

  WRITE(*,*) ‘i=‘,i
  WRITE(*,*) ‘c=‘,TRIM(c)
Input: From Screen

• Reading an integer from screen is accomplished by:
  INTEGER :: i
  READ(*,*) i

• Similarly, a single-precision floating-point variable is read by:
  REAL :: r
  READ(*,*) r

• Finally, a character string is read by:
  INTEGER, PARAMETER :: MAX_CHAR_LEN = 80
  CHARACTER(MAX_CHAR_LEN) :: c
  READ(*,*) c
Input: From Screen

- Can also read several variables together:
  
  INTEGER, PARAMETER :: MAX_CHAR_LEN = 80  
  INTEGER :: i  
  CHARACTER(MAX_CHAR_LEN) :: c  
  WRITE(*,*) 'Enter integer and string:'  
  READ(*,*) i,c  
  WRITE(*,*) i,TRIM(c)

- Get run-time error if data that does not match data type:
  
  Enter integer and string:  
  17.0 3  
  At line 11 of file readmixed.f90 (unit = 5, file = 'stdin')  
  Fortran runtime error: Bad integer for item 1 in list input
Defensive Programming

• Better way is to check for invalid input using **IOSTAT** specifier:

```fortran
INTEGER :: ef
READ(*,*,IOSTAT=ef) i,c
IF ( ef /= 0 ) THEN
    WRITE(*,*) 'ERROR - Invalid input'
    STOP
END IF ! ef
```

giving

```
Enter integer and string:
17.0 3
ERROR - Invalid input!
```

**IOSTAT** returns an integer status variable

Non-zero status variable indicates **READ** statement was unsuccessful

Values and meaning of non-zero variable compiler-dependent
Often desire to influence how output is formatted

For example, want to write certain number of decimal places,

\[ \pi = 3.14159 \]

instead of

\[ \pi = 3.141592653589793238462643383279 \]

or use engineering notation,

\[ p = 1.01325 \times 10^5 \]

instead of

\[ p = 101325.0 \]

This can be achieved using **edit descriptors**

They are used in place of second * in WRITE statement:

```
WRITE(*,'(edit descriptors)') something
```

Note apostrophes and parentheses!
Output: To Screen with Formatting

• Following only covers most important edit descriptors
• For more information, see:
  ‣ Chapman (2008): 5.3 and 14.1
  ‣ Ellis et al. (1994): 8.2 and 15.2
  ‣ Metcalf et al. (2004): 9
Output: To Screen with Formatting

- For character variables and constants, use a edit descriptor:

  \[ \text{Aw} \quad w = \text{overall width} \]

- Example code:

  ```fortran
  PROGRAM editdescriptors
  IMPLICIT NONE
  INTEGER, PARAMETER :: MAX_CHAR_LEN = 80
  CHARACTER(MAX_CHAR_LEN) :: c
  c = 'We are testing edit descriptors!' 
  WRITE(*,*) LEN_TRIM(c)
  WRITE(*,'(A10)') c
  WRITE(*,'(A32)') c
  WRITE(*,'(A33)') c
  END PROGRAM editdescriptors
  ```

- Output

  We are tes
  We are testing edit descriptors!
  We are testing edit descriptors!

  Output truncated if width too small!
Output: To Screen with Formatting

- For real and double-precision variables and constants:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_{w.\ d}$</td>
<td>$w = \text{overall width}$\n$d = \text{number of decimal places}$</td>
<td>$\pm0.\text{nnnnnn}E\pm\text{ee}$\n$123&lt;-\ d \rightarrow 4567$\n$w \geq d + 7$\nFor example: $E10.3, E13.6, E16.9$</td>
</tr>
<tr>
<td>$F_{w.\ d}$</td>
<td>$w = \text{overall width}$\n$d = \text{number of decimal places}$</td>
<td>$\pm\text{mmm.nnnnnnn}$\n$&lt;-\ d \rightarrow$</td>
</tr>
</tbody>
</table>
Output: To Screen with Formatting

- Example code:

```fortran
PROGRAM editdescriptors
IMPLICIT NONE
REAL :: r
r = -1234567890.987654321

WRITE(*,'(A,1X,E13.6)') 'E13.6 gives:', r
WRITE(*,'(A,1X,E10.3)') 'E10.3 gives:', r
WRITE(*,'(A,1X,E9.3)')  'E9.3 gives:', r
WRITE(*,'(A,1X,E8.3)')  'E8.3 gives:', r
WRITE(*,'(A,1X,E7.3)')  'E7.3 gives:', r

END PROGRAM editdescriptors
```

- Output:

```
E13.6 gives: -0.123457E+10
E10.3 gives: -0.123E+10
E9.3 gives: -.123E+10
E8.3 gives: ********
E7.3 gives: ********
```

Use \textit{nX} edit descriptor to insert \textit{n} empty spaces

Combine edit descriptors separated by commas, Must match!

If overall width too small, lose 0 and get *
Example code:

```fortran
PROGRAM editdescriptors

IMPLICIT NONE

REAL :: r

r = -12345.0

WRITE(*,'(A,1X,F8.1)') 'F8.1 gives:',r
WRITE(*,'(A,1X,F9.2)') 'F9.2 gives:',r
WRITE(*,'(A,1X,F7.1)') 'F7.1 gives:',r

END PROGRAM editdescriptors
```

Output:

```
F8.1 gives: -12345.0
F9.2 gives: -12345.00
F7.1 gives: *******
```

If overall width too small, get *
Output: To Screen with Formatting

- For integer variables and constants, use \( \_I \) edit descriptor:
  \[
  \_I \, w \quad w = \text{overall width}
  \]

- For logical variables and constants, use \( \_L \) edit descriptor:
  \[
  \_L \, w \quad w = \text{overall width}
  \]
Output: To Screen with Formatting

• Any edit descriptor can be prefaced to indicate repetition
• For example, to write out 2 integers of width 4:
  WRITE(*,'(2I4)')

• More complicated example:
  WRITE(*,'(A,3X,A,2(1X,E23.16),2(1X,I9))') s1,s2,r1,r2,i1,i2
To write to file, replace first * in WRITE statement:

```plaintext
WRITE(fileUnit,'(edit descriptors)') something
```

where `fileUnit` is an integer

Following will only cover most important aspects of writing to (and reading from) files

For more information, see:

- Chapman (2008): 5.5
- Ellis et al. (1994): 9
- Metcalf et al. (2004): 10

File must (should) be opened before can write to it

Once opened, can write data to file in same way as writing to screen
**Input/Output: File Handling**

- Files are opened with
  \( \text{OPEN}(\text{list of specifiers}) \)

- Specifiers include:

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UNIT</strong>=\textit{fileUnit}</td>
<td>\textit{fileUnit} is a positive integer</td>
<td>- \textit{fileUnit} values 0, 5, and 6 are reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- The maximum value of \textit{fileUnit} is compiler-dependent (often 99)</td>
</tr>
<tr>
<td><strong>FILE</strong>=\textit{fileName}</td>
<td>\textit{fileName} is a string</td>
<td></td>
</tr>
<tr>
<td><strong>STATUS</strong>=\textit{fileStatus}</td>
<td>\textit{fileStatus} is a string</td>
<td>Permissible values of \textit{fileStatus}:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- ‘NEW’ = File must not exist, causes error if does exist</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- ‘OLD’ = File must exist, causes error if does not exist</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- ‘REPLACE’ = Create file regardless of whether it exists</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- ‘UNKNOWN’ = File may exist, behaviour is compiler-dependent</td>
</tr>
<tr>
<td><strong>IOSTAT</strong>=\textit{errorFlag}</td>
<td>\textit{errorFlag} is an integer</td>
<td>Non-zero values of \textit{errorFlag} indicate that an error occurred on opening the file. The values and meaning are compiler-dependent.</td>
</tr>
</tbody>
</table>
The `INQUIRE` statement can be used to find out more about particular `fileUnit`

Here only illustrate use to determine whether file exists:

```
INQUIRE (UNIT=fileUnit,EXIST=existFlag, &
         IOSTAT=errorFlag)
```

If the file exists, `existFlag` is set to `.TRUE.`; otherwise to `.FALSE.`.

There are many additional specifiers, see, for example, Chapman (2008, 14.3.3)

Can be used as follows...
Defensive Programming

Opening file in Fortran:

```
INQUIRE(UNIT=fileUnit,EXIST=existFlag,IOSTAT=errorFlag)
IF ( errorFlag /= 0 ) THEN
    WRITE(*,*) ‘ERROR - INQUIRE returned non-zero IOSTAT!’
    STOP
ELSE
    IF ( existFlag .EQV. .TRUE. ) THEN
        OPEN(UNIT=fileUnit,FILE=TRIM(fileName),STATUS=‘REPLACE’,IOSTAT=errorFlag)
    ELSE
        OPEN(UNIT=fileUnit,FILE=TRIM(fileName),STATUS=‘NEW’,IOSTAT=errorFlag)
    END IF ! existFlag
    IF ( errorFlag /= 0 ) THEN
        WRITE(*,*) ‘ERROR - Could not open file!’
        STOP
    END IF ! errorFlag
END IF ! errorFlag
```

May not always be appropriate option...
Input/Output: File Handling

• Closing files is considerably easier:
  
  \[ \text{CLOSE(list of specifiers)} \]

• Specifiers include:

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIT=\text{fileUnit}</td>
<td>\text{fileUnit} is a positive integer</td>
<td></td>
</tr>
<tr>
<td>STATUS=\text{fileStatus}</td>
<td>\text{fileStatus} is a string</td>
<td>Permissible values of \text{fileStatus}:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• ‘\text{KEEP}’ = File will not be deleted after closing (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• ‘\text{DELETE}’ = File will be deleted after closing</td>
</tr>
<tr>
<td>IOSTAT=\text{errorFlag}</td>
<td>\text{errorFlag} is an integer</td>
<td>Non-zero values of \text{errorFlag} indicate that an error occurred on opening the file. The values and meaning are compiler-dependent.</td>
</tr>
</tbody>
</table>
Defensive Programming

Closing file in Fortran:

```
CLOSE(FILE=fileUnit, IOSTAT=errorFlag)
IF ( errorFlag /= 0 ) THEN
  WRITE(*,*) 'ERROR - Could not close file!'
  STOP
END IF ! errorFlag
```
Input: From File

- File must (should) be opened before can read from it
- Once opened, can read data from file in same way as reading from screen
Dirty little secret about Fortran file output:
- If file with unit \texttt{fileUnit} not opened before writing to it, Fortran will write to file called \texttt{fort.fileUnit}
- This can be very useful while developing or debugging
- This behavior is compiler-dependent, so not portable and should not be used in general
Overview

- Structuring a program
- Functions
- Makefiles
- Subroutines
- Functions & subroutines: discussion
- Modules
- Arrays
  - Dynamic memory allocation
  - Arrays as arguments
- Additional topics
Structuring a Program

- Can write complex programs with material already covered
- However, these programs will consist of one single routine
- They will be hard to understand, test, change, share…
- … which is everything we do not want programs to be
Structuring a Program

- Good approach of tackling complex problems:
  - Subdivide into smaller and simpler parts (analysis)
  - Work on smaller and simpler parts
  - Put smaller and simpler parts back together (synthesis)
- Fortran provides several ways of using this approach:
  - **Procedures**: Functions and subroutines
  - **Modules**: Data and procedures
- Functions and subroutines are an important part of programming method called **structured or modular programming**
Overview

• Structuring a program
• Functions
• Makefiles
• Subroutines
• Functions & subroutines: discussion
• Modules
• Arrays
  ‣ Dynamic memory allocation
  ‣ Arrays as arguments
• Additional topics
Functions

• Example: Fortran provides intrinsic function to compute square root, but not cube root
• Can address this problem by writing own function:

```fortran
REAL FUNCTION cbrt(x)
IMPLICIT NONE
REAL, INTENT(IN) :: x

cbrt = EXP(LOG(x)/3.0)
END FUNCTION cbrt
```

Declaration section  
Executable section  
Termination section
Functions

- Example: Fortran provides intrinsic function to compute square root, but not cube root
- Can address this problem by writing own function:

  ```fortran
  REAL FUNCTION cbrt(x)   ! Name of function and dummy argument
  IMPLICIT NONE
  REAL, INTENT(IN) :: x   ! Declaration of dummy argument(s)
  cbrt = EXP(LOG(x)/3.0)  ! Compute result, must be assigned to variable with same name as function
  END FUNCTION cbrt   ! End of function
  ```

- Will address `INTENT (IN)` attribute shortly…
Functions

• This function could now be used as follows:

```fortran
PROGRAM main
IMPLICIT NONE
REAL :: x, y, r
x = 3.0
y = 3.0
r = cbrt(x)  ! Used in assignment statement (or in other operations)
WRITE(*,*) r, cbrt(y)  ! Used in WRITE statement
END PROGRAM main
```

• Two problems:
  • Small problem: what should be improved in `cbrt`?
  • Bigger problem: this program will not compile because `main` does not “know” what `cbrt` is/does
## Functions

- Two basic ways of “telling” `main` about `cbrt`:

<table>
<thead>
<tr>
<th><code>cbrt</code> is internal procedure</th>
<th><code>cbrt</code> is external procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Put <code>cbrt</code> into same file as <code>main</code>, use <code>CONTAINS</code> statement, and compile as single file</td>
<td>Put <code>cbrt</code> and <code>main</code> into separate files, compile separately, then link</td>
</tr>
</tbody>
</table>
Functions: Internal Procedure

- Example:

```plaintext
PROGRAM main
IMPLICIT NONE
REAL :: x, r
x = 3.0
r = cbrt(x)
WRITE(*,*) r

CONTAINS

REAL FUNCTION cbrt(x)
IMPLICIT NONE
REAL, INTENT(IN) :: x
cbr = EXP(LOG(x)/3.0)
END FUNCTION cbrt

END PROGRAM main
```
Functions: Internal Procedure

• Example:

```fortran
PROGRAM main
IMPLICIT NONE
REAL :: x, r
x = 3.0
r = cbrt(x)
WRITE(*,*) r

CONTAINS

REAL FUNCTION cbrt(x)
IMPLICIT NONE
REAL, INTENT(IN) :: x
clrt = EXP(LOG(x)/3.0)
END FUNCTION cbrt

END PROGRAM main
```
Functions: Internal Procedure

• Example:

```fortran
PROGRAM main
IMPLICIT NONE
REAL :: x,r
x = 3.0
r = cbrt(x)
WRITE(*,*) r

CONTAINS
REAL FUNCTION cbrt(x)
IMPLICIT NONE
REAL, INTENT(IN) :: x
cbrt = EXP(LOG(x)/3.0)
END FUNCTION cbrt

END PROGRAM main
```
Functions: Internal Procedure

• Example:

```plaintext
PROGRAM main
IMPLICIT NONE
REAL :: x,r
x = 3.0
r = cbrt(x)
WRITE(*,*) r

CONTAINS

REAL FUNCTION cbrt(x)
IMPLICIT NONE
REAL, INTENT(IN) :: x
cbrt = EXP(LOG(x)/3.0)
END FUNCTION cbrt

END PROGRAM main
```

Note: Function appears after CONTAINS statement but before END PROGRAM statement
Functions: INTENT (IN) Attribute

- Background: Fortran uses “call by reference”, so addresses of arguments rather than argument values are passed
- Could have written `cbrt` as:

```fortran
REAL FUNCTION cbrt(x)

IMPLICIT NONE

REAL :: x

x = LOG(x)/3.0

cbrt = EXP(x)

END FUNCTION cbrt
```

This function is said to have a **side-effect**: It does not just return a result (main effect), but also modifies the dummy argument.
Functions: \texttt{INTENT(IN)} Attribute

- Effectively, functions with side effects have two (or more) outputs
- Side effects are permitted in Fortran, but considered poor programming practice
- Good programming practice: Write functions without side effects, that is, functions that do not modify dummy arguments
- Recommendation: always use \texttt{INTENT(IN)} attribute for dummy arguments in functions
- \texttt{INTENT(IN)} attribute is nothing but a sign to compiler to check if your actions and intentions are consistent
- If variable with \texttt{INTENT(IN)} modified by mistake, compiler issues error
Functions: INTENT (IN) Attribute

• Example:

```
REAL FUNCTION cbrt(x)

IMPLICIT NONE

REAL, INTENT(IN) :: x

x = LOG(x)/3.0

cbrt = EXP(x)

END FUNCTION cbrt
```

Statement changes x and hence violates INTENT (IN) attribute

Therefore, compiler will issue error
Functions: External Procedure

```
PROGRAM main
IMPLICIT NONE
REAL, EXTERNAL :: cbrt
REAL :: x,r
x = 3.0
r = cbrt(x)
WRITE(*,*) r
END PROGRAM main

REAL FUNCTION cbrt(x)
IMPLICIT NONE
REAL, INTENT(IN) :: x
cbrt = EXP(LOG(x)/3.0)
END FUNCTION cbrt
```

Compiling

```
gfortran -g -Wall -c main.f90 -o main.o
gfortran -g -Wall -c cbrt.f90 -o cbrt.o
```

Linking

```
gfortran main.o cbrt.o -o main
```
Functions: External Procedure

• Functions in files other than that containing main program must be declared with the \texttt{EXTERNAL} attribute

• Advantage: Putting functions into separate files has advantage of keeping size of main program manageable

• Disadvantage: Compiler cannot check whether function called correctly (illustrated later)

• Later, will learn how to solve this problem
Functions: Alternative Form

• Example:

    REAL FUNCTION cbrt(x)
    
    IMPLICIT NONE

    REAL, INTENT(IN) :: x

    cbrt = EXP(LOG(x)/3.0)

    END FUNCTION cbrt
Functions: Alternative Form

• Example:

```fortran
FUNCTION cbrt(x)
  IMPLICIT NONE
  REAL :: cbrt
  REAL, INTENT(IN) :: x
  cbrt = EXP(LOG(x)/3.0)
END FUNCTION cbrt
```

Can omit data type from FUNCTION statement...

... provided it is declared

Note: This form is necessary with functions that return user-defined data types
Functions: Summary

- Functions are Fortran procedures that return a single result
- Non-intrinsic functions are sometimes called user-defined functions
Overview

• Structuring a program
• Functions
• Makefiles
• Subroutines
• Functions & subroutines: discussion
• Modules
• Arrays
  ‣ Dynamic memory allocation
  ‣ Arrays as arguments
• Additional topics
Makefiles

- Manually compiling and linking of codes consisting of multiple files is tedious
- Makefiles can be used to automate compilation and linking
- Use/adapt the simple Makefile provided
### Macros

SHELL= /bin/sh
EXEC= mycode
FC= gfortran
FFLAGS= -g -Wall -Wextra -fbounds-check
MODEXT= mod
RIFFRAFF= *.bak *.bck *.ckp *.dSYM
RM= rm -rf

### Source and object section

SRCF90= ModSample.f90 main.f90

### Pattern rules

%.o: %.f90
  $(FC) $(FFLAGS) -c $< -o $@
OBJF90= $(SRCF90:.f90=.o)

### Target section

$(EXEC): $(OBJF90)
  $(FC) $(OBJF90) $(LDFLAGS) -o $@

.PHONY: clean
clean:
  $(RM) $(OBJF90) *.$(MODEXT) $(EXEC) $(RIFFRAFF)

### Dependencies section

main.o: ModSample.o

---

**Define macros for later use**

**List source files (modules first)**

**Explain to make how to produce object files from source files**

**Explain to make what to do**

**Explain to make how your files depend on each other**
### Macros

SHELL= /bin/sh
EXEC= mycode
FC= gfortran
FFLAGS= -g -Wall -Wextra -fbounds-check
#FFLAGS= -O3
MODEXT= mod
RIFFRAFF= *.bak *.bck *.ckp *.dSYM
RM= rm -rf

### Source and object section

SRCF90= ModSample.f90 main.f90

### Pattern rules

%.o: %.f90
 $(FC) $(FFLAGS) -c $< -o $@

OBJF90= $(SRCF90:.f90=.o)

### Target section

$(EXEC): $(OBJF90)
 $(FC) $(OBJF90) $(LDFLAGS) -o $@

.PHONY: clean
clean:
 $(RM) $(OBJF90) *.$(MODEXT) $(EXEC) $(RIFFRAFF)

### Dependencies section

main.o: ModSample.o
Overview

- Structuring a program
- Functions
- Makefiles
- Subroutines
- Functions & subroutines: discussion
- Modules
- Arrays
  - Dynamic memory allocation
  - Arrays as arguments
- Additional topics
Subroutines

- Dummy argument(s) to functions should not be modified
- Hence functions are limited to returning single result
- To avoid this restriction, use subroutines
- Example: Computing roots of quadratic function

```fortran
PROGRAM main

IMPLICIT NONE

REAL :: a, b, c
COMPLEX :: r1, r2

WRITE(*,*) 'Enter a, b, c:
READ(*,*) a, b, c

CALL SolveQuadratic(a, b, c, r1, r2)

WRITE(*,*) 'Solutions:', r1, r2

END PROGRAM main
```

Get coefficients

Call subroutine to compute roots

Write roots
Subroutines

- As with `cbrt` function discussed earlier, can treat `SolveQuadratic` as internal or external procedure
- First, take closer look at `SolveQuadratic`
SUBROUTINE SolveQuadratic(a,b,c,r1,r2)

IMPLICIT NONE

REAL, INTENT(IN) :: a,b,c
COMPLEX, INTENT(OUT) :: r1,r2

REAL :: d,q

d = b**2 - 4.0*a*c

IF ( d < 0.0 ) THEN
    q  = SQRT(-d)
    r1 = CMPLX(-0.5*b/a, 0.5*q/a)
    r2 = CMPLX(-0.5*b/a,-0.5*q/a)
ELSE
    q  = -0.5*(b + SIGN(1.0,b))*SQRT(d))
    r1 = CMPLX(q/a,0.0)
    r2 = CMPLX(c/q,0.0)
END IF ! d

END SUBROUTINE SolveQuadratic
SUBROUTINE SolveQuadratic(a,b,c,r1,r2)

IMPLICIT NONE

REAL, INTENT(IN) :: a,b,c
COMPLEX, INTENT(OUT) :: r1,r2

REAL :: d,q

d = b**2 - 4.0*a*c

IF ( d < 0.0 ) THEN
    q  = SQRT(-d)
    r1 = CMPLX(-0.5*b/a, 0.5*q/a)
    r2 = CMPLX(-0.5*b/a,-0.5*q/a)
ELSE
    q  = -0.5*(b + SIGN(1.0,b)*SQRT(d))
    r1 = CMPLX(q/a,0.0)
    r2 = CMPLX(c/q,0.0)
END IF ! d

END SUBROUTINE SolveQuadratic

Name of subroutine and dummy arguments
Declaration of dummy arguments
Declaration of local variables
Recommendation: Separate declarations into
• input arguments
• output arguments
• local variables
End of subroutine
SUBROUTINE SolveQuadratic(a,b,c,r1,r2)

IMPLICIT NONE

REAL, INTENT(IN) :: a,b,c
COMPLEX, INTENT(OUT) :: r1,r2

REAL :: d,q

d = b**2 - 4.0*a*c

IF ( d < 0.0 ) THEN
    q  = SQRT(-d)
    r1 = CMPLX(-0.5*b/a, 0.5*q/a)
    r2 = CMPLX(-0.5*b/a,-0.5*q/a)
ELSE
    q  = -0.5*(b + SIGN(1.0,b)*SQRT(d))
    r1 = CMPLX(q/a,0.0)
    r2 = CMPLX(c/q,0.0)
END IF ! d

END SUBROUTINE SolveQuadratic

\textbf{INTENT (OUT)} indicates that these dummy arguments become undefined on entry. It does \textbf{not} mean that compiler should check that they will be set! If these dummy arguments are not set in the subroutine, they will be undefined on exit also!
Subroutines

• Now, let’s take a look at how we can “tell” the main program about `SolveQuadratic`

• Discussed two options earlier:
  ‣ Internal procedure: Use `CONTAINS` statement
  ‣ External procedure: Separate files
PROGRAM main
IMPLICIT NONE
REAL :: a,b,c
COMPLEX :: r1,r2
WRITE(*,*) 'Enter a,b,c:'
READ(*,*) a,b,c
CALL SolveQuadratic(a,b,c,r1,r2)
WRITE(*,*) 'Solutions:',r1,r2
CONTAINS
SUBROUTINE SolveQuadratic(a,b,c,r1,r2)
...
END SUBROUTINE SolveQuadratic
END PROGRAM main
PROGRAM main
IMPLICIT NONE
REAL :: a,b,c
COMPLEX :: r1,r2
WRITE(*,*) 'Enter a,b,c:'
READ(*,*) a,b,c
CALL SolveQuadratic(a,b,c,r1,r2)
WRITE(*,*) 'Solutions:',r1,r2
CONTAINS
SUBROUTINE SolveQuadratic(a,b,c,r1,r2)
...
END SUBROUTINE SolveQuadratic
END PROGRAM main
PROGRAM main

IMPLICIT NONE

REAL :: a,b,c
COMPLEX :: r1,r2

WRITE(*,*) 'Enter a,b,c:'
READ(*,*) a,b,c

CALL SolveQuadratic(a,b,c,r1,r2)

WRITE(*,*) 'Solutions:',r1,r2

CONTAINS

SUBROUTINE SolveQuadratic(a,b,c,r1,r2)
...
END SUBROUTINE SolveQuadratic

END PROGRAM main
Subroutines: External Procedure

**main.f90**

```fortran
PROGRAM main
    IMPLICIT NONE
    REAL :: a,b,c
    COMPLEX :: r1,r2
    ...
    CALL SolveQuadratic(...)
    ...
END PROGRAM main
```

**SolveQuadratic.f90**

```fortran
SUBROUTINE ...
    ...
END SUBROUTINE ...
```

**Compiling**

```
gfortran -g -Wall -c main.f90 -o main.o
```

**Linking**

```
gfortran main.o SolveQuadratic.o -o main
```

---

**gfortran -g -Wall -c main.f90 -o main.o**

**SolveQuadratic.f90**

```fortran
SUBROUTINE ...
    ...
END SUBROUTINE ...
```

**main.o**

**SolveQuadratic.o**

**main**
Overview

• Structuring a program
• Functions
• Makefiles
• Subroutines
• Functions & subroutines: discussion
• Modules
• Arrays
  ‣ Dynamic memory allocation
  ‣ Arrays as arguments
• Additional topics
Functions & Subroutines: Discussion

• Functions and subroutines are indispensable for writing modular programs: **Use them extensively**

• Guidelines for writing one or the other:

<table>
<thead>
<tr>
<th>Write a FUNCTION when…</th>
<th>… you have one result/output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write a SUBROUTINE when…</td>
<td>… you have more than one result/output</td>
</tr>
</tbody>
</table>

• Advantages of writing modular programs:
  ‣ Program easier to understand
  ‣ Program easier to test/debug
  ‣ Parts of program can be reused/shared
  ‣ …

• With well-designed and documented interface, procedure internals are not important
Functions & Subroutines: Discussion

• Personal recommendations:
  ‣ Keep function names short (like \texttt{SIN}, \texttt{SQRT}) to keep expressions short
  ‣ Choose subroutine names that clearly indicate objective:
    \begin{verbatim}
    SolveQuadratic
    SolveCubic
    ComputeFluxes
    ComputeSolutionError
    IterateJacobi
    IterateGaussSeidel
    \end{verbatim}
  ‣ Above all, use a consistent naming scheme!
Functions & Subroutines: Discussion

- Keep in mind the following about **INTENT** attributes:

| INTENT (IN) | • Argument is defined on entry to procedure  
|            | • Argument will not be changed in procedure  
|            | • Compiler will issue error if argument is changed |
| INTENT (OUT) | • Argument becomes undefined on entry to the procedure  
|             | • Argument will be undefined on exit unless it is set in the procedure  
|             | • It is up to you to set the value of the argument!  
|             | • The compiler may not warn you if the argument is not defined! |
| INTENT (INOUT) | • Argument is defined on entry to procedure  
|               | • Argument can be changed in procedure |
Functions & Subroutines: Discussion

- What are advantages/disadvantages of internal and external procedures?

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Internal procedure</strong> (with CONTAINS statement)</td>
<td>• Simplicity: no need to compile and link multiple</td>
</tr>
<tr>
<td></td>
<td>files</td>
</tr>
<tr>
<td></td>
<td>• Single-file program may become long</td>
</tr>
<tr>
<td><strong>External procedure</strong> (separate files)</td>
<td>• Separate files easier to keep short and easy to</td>
</tr>
<tr>
<td></td>
<td>understand</td>
</tr>
<tr>
<td></td>
<td>• Need to compile and link multiple files</td>
</tr>
</tbody>
</table>

- There is another, very important advantage to internal procedures...

- Consider example on following slide
PROGRAM main
IMPLICIT NONE
REAL :: a,b,c
COMPLEX :: r1,r2
WRITE(*,*) 'Enter a,b,c:'
READ(*,*) a,b,c
CALL SolveQuadratic(a,b,c,r1,r2)
WRITE(*,*) 'Solutions:',r1,r2
CONTAINS
SUBROUTINE SolveQuadratic(a,b,c,r1,r2)
IMPLICIT NONE
REAL, INTENT(IN) :: a,b,c
COMPLEX, INTENT(OUT) :: r1,r2
REAL :: d,q
d = b**2 - 4.0*a*c
IF ( d < 0.0 ) THEN
    q  = SQRT(-d)
r1 = CMPLX(-0.5*b/a, 0.5*q/a)
r2 = CMPLX(-0.5*b/a,-0.5*q/a)
ELSE
    q  = -0.5*(b + SIGN(1.0,b)*SQRT(d))
r1 = CMPLX(q/a,0.0)
r2 = CMPLX(c/q,0.0)
END IF ! d
END SUBROUTINE SolveQuadratic
END PROGRAM main

Functions & Subroutines: Discussion

- Consider first case of internal procedure as shown on left
- Compiles and runs successfully
Functions & Subroutines: Discussion

- Now assume we made mistake and accidentally forgot to pass third argument
- Very common mistake!
Functions & Subroutines: Discussion

- This is clearly wrong because argument lists do not match

```fortran
PROGRAM main
IMPLICIT NONE
REAL :: a,b,c
COMPLEX :: r1,r2
WRITE(*,*) 'Enter a,b,c:'
READ(*,*) a,b,c
CALL SolveQuadratic(a,b,r1,r2)
WRITE(*,*) 'Solutions:',r1,r2
CONTAINS
SUBROUTINE SolveQuadratic(a,b,c,r1,r2)
IMPLICIT NONE
REAL, INTENT(IN) :: a,b,c
COMPLEX, INTENT(OUT) :: r1,r2
REAL :: d,q
d = b**2 - 4.0*a*c
IF ( d < 0.0 ) THEN
  q = SQRT(-d)
  r1 = CMPLX(-0.5*b/a, 0.5*q/a)
  r2 = CMPLX(-0.5*b/a,-0.5*q/a)
ELSE
  q = -0.5*(b + SIGN(1.0,b)*SQRT(d))
  r1 = CMPLX(q/a,0.0)
  r2 = CMPLX(c/q,0.0)
END IF ! d
END SUBROUTINE SolveQuadratic
END PROGRAM main
```
Functions & Subroutines: Discussion

Because procedure is internal, compiler can compare arguments lists

Inconsistent argument lists lead to compiler error:

```fortran
PROGRAM main
IMPLICIT NONE
REAL :: a,b,c
COMPLEX :: r1,r2
WRITE(*,*) 'Enter a,b,c:'
READ(*,*) a,b,c
CALL SolveQuadratic(a,b,r1,r2)
WRITE(*,*) 'Solutions:',r1,r2
CONTAINS

SUBROUTINE SolveQuadratic(a,b,c,r1,r2)
IMPLICIT NONE
REAL, INTENT(IN) :: a,b,c
COMPLEX, INTENT(OUT) :: r1,r2
REAL :: d,q
   d = b**2 - 4.0*a*c
   IF ( d < 0.0 ) THEN
      q = SQRT(-d)
      r1 = CMPLX(-0.5*b/a, 0.5*q/a)
      r2 = CMPLX(-0.5*b/a,-0.5*q/a)
   ELSE
      q = -0.5*(b + SIGN(1.0,b)*SQRT(d))
      r1 = CMPLX(q/a,0.0)
      r2 = CMPLX(c/q,0.0)
   END IF ! d
END SUBROUTINE SolveQuadratic
END PROGRAM main
```

gfortran -g -Wall main.f90 -o main
main.f90:12.26:

```
CALL SolveQuadratic(a,b,r1,r2)
```

Error: Type mismatch in argument 'c' at (1); passed COMPLEX(4) to REAL(4)
Functions & Subroutines: Discussion

- Now consider case of external procedure
- Compiler cannot compare argument lists
- Code compiles successfully and gives wrong answer!

- Problem is implicit interface of procedure

```fortran
PROGRAM main
IMPLICIT NONE
REAL :: a,b,c
COMPLEX :: r1,r2
WRITE(*,*) 'Enter a,b,c:'
READ(*,*) a,b,c
CALL SolveQuadratic(a,b,r1,r2)
WRITE(*,*) 'Solutions:',r1,r2
END PROGRAM main

SUBROUTINE SolveQuadratic(a,b,c,r1,r2)
IMPLICIT NONE
REAL, INTENT(IN) :: a,b,c
COMPLEX, INTENT(OUT) :: r1,r2
REAL :: d,q
  d = b**2 - 4.0*a*c
IF ( d < 0.0 ) THEN
    q  = SQRT(-d)
    r1 = CMPLX(-0.5*b/a, 0.5*q/a)
    r2 = CMPLX(-0.5*b/a,-0.5*q/a)
ELSE
    q  = -0.5*(b + SIGN(1.0,b)*SQRT(d))
    r1 = CMPLX(q/a,0.0)
    r2 = CMPLX(c/q,0.0)
END IF ! d
END SUBROUTINE SolveQuadratic
```
Functions & Subroutines: Discussion

- Problem can be solved by providing **explicit interface**
- Inconsistent argument lists lead to compiler error:

```
gfortran -g -Wall -c main.f90 -o main.o
main.f90:19.26:
    CALL SolveQuadratic(a,b,r1,r2)
              ^
Error: Type mismatch in argument 'c' at (1); passed COMPLEX(4) to REAL(4)
```

- Will not discuss explicit interfaces in detail here since there exists a better way…
## Functions & Subroutines: Discussion

- What are advantages/disadvantages of internal and external procedures?

<table>
<thead>
<tr>
<th></th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
</table>
| **Internal procedure** *(with **CONTAINS** statement)* | • Simplicity: no need to compile and link multiple files  
• Interfaces are always explicit | • Single-file program may become long  
• Inconvenient for sharing |
| **External procedure** *(separate files)*              | • Separate files easier to keep short and easy to understand  
• Easy to share | • Need to compile and link multiple files  
• Explicit interfaces must be provided |
Overview

- Structuring a program
- Functions
- Makefiles
- Subroutines
- Functions & subroutines: discussion
- Modules
- Arrays
  - Dynamic memory allocation
  - Arrays as arguments
- Additional topics
Modules

• Modules collect variables, data-type definitions, and procedures

• Advantages:
  ‣ Simplify writing of modular programs
  ‣ Module procedures automatically have explicit interfaces
  ‣ …
MODULE ModSolvePoly
CONTAINS

SUBROUTINE SolveQuadratic(a,b,c,r1,r2)
IMPLICIT NONE
REAL, INTENT(IN) :: a,b,c
COMPLEX, INTENT(OUT) :: r1,r2
REAL :: d,q

! \text{Compute discriminant}

d = b**2 - 4.0*a*c
IF ( d < 0.0 ) THEN
  q = SQRT(-d)
  r1 = CMPLX(-0.5*b/a, 0.5*q/a)
  r2 = CMPLX(-0.5*b/a,-0.5*q/a)
ELSE
  q = -0.5*(b + SIGN(1.0,b)*SQRT(d))
  r1 = CMPLX(q/a,0.0)
  r2 = CMPLX(c/q,0.0)
END IF ! d
END SUBROUTINE SolveQuadratic

END MODULE ModSolvePoly
Accessing contents of module is known as use association:

```fortran
PROGRAM main

USE ModSolvePoly

IMPLICIT NONE

REAL :: a,b,c
COMPLEX :: r1,r2

WRITE(*,*) 'Enter a,b,c:'
READ(*,*) a,b,c

CALL SolveQuadratic(a,b,c,r1,r2)

WRITE(*,*) 'Solutions:',r1,r2

END PROGRAM main
```

**USE statements must appear immediately after PROGRAM, FUNCTION, SUBROUTINE, or MODULE statement and before IMPLICIT NONE**
Modules

• Modules are compiled like other Fortran source files:
  
gfortran -g -Wall -c ModSolvePoly.f90 -o ModSolvePoly.o  
gfortran -g -Wall -c main.f90 -o main.o 
gfortran -g -Wall main.o ModSolvePoly.o -o main

• Compilation of ModSolvePoly.f90 produces
  
ModSolvePoly.o
modsolvepoly.mod

• Modules must be compiled before files in which they are used, otherwise get compiler error:

  main.f90:4.6:

    USE ModSolvePoly
        1
Fatal Error: Can't open module file 'modsolvepoly.mod' for reading at (1): No such file or directory
• **Key point:** By placing procedure `SolveQuadratic` in module, it automatically acquires explicit interface.

• **Therefore, if use incorrect argument list, such as**

```fortran
PROGRAM main
USE ModSolvePoly
IMPLICIT NONE
REAL :: a,b,c
COMPLEX :: r1,r2
WRITE(*,*) 'Enter a,b,c:'
READ(*,*) a,b,c
CALL SolveQuadratic(a,b,r1,r2)  \[Error: Type mismatch in argument 'c' at (1); passed COMPLEX(4) to REAL(4)]
WRITE(*,*) 'Solutions:',r1,r2
END PROGRAM main
```

get compiler error:

```fortran
main.f90:14.26:
CALL SolveQuadratic(a,b,r1,r2)
\[1\]
Error: Type mismatch in argument 'c' at (1); passed COMPLEX(4) to REAL(4)
```
Modules

• Modules give flexibility of splitting program into several files without having to specify explicit interfaces
• `ModSolvePoly.f90` could be made more useful by including additional routines:
  ‣ `SolveCubic(a, b, c, d, r1, r2, r3)`
  ‣ `SolveQuartic(a, b, c, d, e, r1, r2, r3, r4)`
  ‣ Versions with double-precision arithmetic
  ‣ Versions with complex coefficients `a`, `b`, and `c`
  ‣ …
Modules

- So far, only placed procedures in modules
- But modules can also contain variables and constants
- Simple examples:

```fortran
MODULE ModPhysicalConstantsSP

IMPLICIT NONE

REAL, PARAMETER :: PHYS_CONST_SP_G  = 9.81
REAL, PARAMETER :: PHYS_CONST_SP_C0 = 299792458.0

END MODULE ModPhysicalConstantsSP

MODULE ModMathematicalConstantsSP

IMPLICIT NONE

REAL, PARAMETER :: MATH_CONST_SP_PI  = 3.14159
REAL, PARAMETER :: MATH_CONST_SP_PHI = 1.61803

END ModMathematicalConstantsSP

```

- Could be used to get unique values anywhere in code
### Modules

- More powerful to combine data and procedures:

  ```fortran
  PROGRAM main
  USE ModSample
  IMPLICIT NONE
  INTEGER :: a
  IntegerPublic = 2
  a = 10
  CALL DummySubroutine(a)
  WRITE(*,*) a
  END PROGRAM main
  
  MODULE ModSample
  INTEGER :: IntegerPublic
  CONTAINS
  SUBROUTINE DummySubroutine(a)
  IMPLICIT NONE
  INTEGER, INTENT(INOUT) :: a
  WRITE(*,*) IntegerPublic,a
  a = IntegerPublic*a
  WRITE(*,*), IntegerPublic,a
  END SUBROUTINE DummySubroutine
  END MODULE ModSample
  
  • Output:
  2          10
  2
  20
  ```
Modules

• More powerful to combine data and procedures:

```fortran
PROGRAM main

USE ModSample

IMPLICIT NONE

INTEGER :: a

IntegerPublic = 2

a = 10

CALL DummySubroutine(a)

WRITE(*,*) a

END PROGRAM main

MODULE ModSample

INTEGER :: IntegerPublic

CONTAINS

SUBROUTINE DummySubroutine(a)

IMPLICIT NONE

INTEGER, INTENT(INOUT) :: a

WRITE(*,*) IntegerPublic,a

a = IntegerPublic*a

END SUBROUTINE DummySubroutine

END MODULE ModSample
```

• Output:

```
2          10
20
```
Modules

• `IntegerPublic` accessible to:
  ‣ All procedures in `ModSample` and
  ‣ All procedures that use-associate `ModSample`

• Therefore, `IntegerPublic` said to be **public variable** of module `ModSample`

• Consequence: Must be careful with naming variables and constants to avoid conflicts with public variables from modules
Modules

• For example, variable called `IntegerPublic` in `main` would conflict with `IntegerPublic` in `ModSample`...

• … and have led to compiler error:

```
main.f90:8.29:

    INTEGER :: a,IntegerPublic
    1
main.f90:4.6:

    USE ModSample
    2
Error: Symbol 'integerpublic' at (1) conflicts with symbol from module 'modsampale', use-associated at (2)
```
Modules

- To avoid conflicts:
  - Careful naming of variables and constants in procedures that use-associate variables and constants from modules
  - Restricting access to variables and constants in modules where possible: Make variables public only if necessary
- Fortran enables this with `PUBLIC` and `PRIVATE` attributes for variables and constants declared in modules: data hiding
Modules

- One solution: Public as default

```fortran
PROGRAM main

USE ModSample
IMPLICIT NONE
INTEGER :: a
IntegerPublic = 2
a = 10
CALL DummySubroutine(a)
WRITE(*,*) a
END PROGRAM main

MODULE ModSample

INTEGER :: IntegerPublic
INTEGER, PRIVATE :: IntegerPrivate
CONTAINS
SUBROUTINE DummySubroutine(a)
IMPLICIT NONE
INTEGER, INTENT(INOUT) :: a
WRITE(*,*) IntegerPublic,a
IntegerPrivate = 20
a = IntegerPublic*IntegerPrivate*a
WRITE(*,*) IntegerPublic,a
END SUBROUTINE DummySubroutine
END MODULE ModSample
```

• Better (and recommended) solution: Private as default

### PROGRAM main

```
USE ModSample
IMPLICIT NONE
INTEGER :: a
IntegerPublic = 2
a = 10
CALL DummySubroutine(a)
WRITE(*,*) a
END PROGRAM main
```

### MODULE ModSample

```
PRIVATE
INTEGER :: IntegerPrivate,IntegerPublic
PUBLIC :: IntegerPublic
PUBLIC :: DummySubroutine
CONTAINS
SUBROUTINE DummySubroutine(a)
IMPLICIT NONE
INTEGER, INTENT(INOUT) :: a
WRITE(*,*) IntegerPublic,a
IntegerPrivate = 20
a = IntegerPublic*IntegerPrivate*a
END SUBROUTINE DummySubroutine
END MODULE ModSample
```
**Modules**

- **Alternative declarations:**

```fortran
PROGRAM main
    USE ModSample
    IMPLICIT NONE
    INTEGER :: a
    IntegerPublic = 2
    a = 10
    CALL DummySubroutine(a)
    WRITE(*,*) a
END PROGRAM main

MODULE ModSample
    PRIVATE
    INTEGER, PUBLIC :: IntegerPublic
    INTEGER :: IntegerPrivate
    PUBLIC :: DummySubroutine
    CONTAINS
    SUBROUTINE DummySubroutine(a)
        IMPLICIT NONE
        INTEGER, INTENT(INOUT) :: a
        WRITE(*,*) IntegerPublic,a
        IntegerPrivate = 20
        a = IntegerPublic*IntegerPrivate*a
    END SUBROUTINE DummySubroutine
END MODULE ModSample
```
• **Note that if** `DummySubroutine` **not** PUBLIC, **that is:**

```fortran
PROGRAM main
  USE ModSample
  IMPLICIT NONE
  INTEGER :: a
  IntegerPublic = 2
  a = 10
  CALL DummySubroutine(a)
  WRITE(*,*) a
END PROGRAM main

MODULE ModSample
  PRIVATE
  INTEGER :: IntegerPrivate, IntegerPublic
  PUBLIC :: IntegerPublic
  CONTAINS
    SUBROUTINE DummySubroutine(a)
      IMPLICIT NONE
      INTEGER, INTENT(INOUT) :: a
      WRITE(*,*) IntegerPublic, a
      INTEGERPrivate = 20
      a = IntegerPublic*IntegerPrivate*a
    END SUBROUTINE DummySubroutine
END MODULE ModSample
```

• **Compiling this leads to…**

```fortran
PROGRAM main
  USE ModSample
  IMPLICIT NONE
  INTEGER :: a
  IntegerPublic = 2
  a = 10
  CALL DummySubroutine(a)
  WRITE(*,*) a
END PROGRAM main
```

```fortran
MODULE ModSample
  PRIVATE
  INTEGER :: IntegerPrivate, IntegerPublic
  PUBLIC :: IntegerPublic
  CONTAINS
    SUBROUTINE DummySubroutine(a)
      IMPLICIT NONE
      INTEGER, INTENT(INOUT) :: a
      WRITE(*,*) IntegerPublic, a
      INTEGERPrivate = 20
      a = IntegerPublic*IntegerPrivate*a
    END SUBROUTINE DummySubroutine
END MODULE ModSample
```
Modules

• Compiling this leads to:

```
gfortran -g -Wall -c ModSample.f90 -o ModSample.o
ModSample.f90:10:0: warning: 'dummysubroutine' defined but not used [-Wunused-function]
gfortran -g -Wall -c main.f90 -o main.o
gfortran ModSample.o main.o -o main
```

```
Undefined symbols for architecture x86_64:
    "dummysubroutine", referenced from:
        _MAIN__ in main.o
ld: symbol(s) not found for architecture x86_64
```
Modules: Examples

• Module for vectors:
  ‣ Define vector
  ‣ Multiply vector by scalar
  ‣ Scalar product
  ‣ Vector product
  ‣ Angle between vectors
  ‣ Rotation of vector
  ‣ …

• Module for intersections:
  ‣ Two lines
  ‣ Line and circle
  ‣ Two circles
  ‣ …
# Modules: Examples

![Image of file browser]

<table>
<thead>
<tr>
<th>Name</th>
<th>Date Modified</th>
<th>Size</th>
<th>Kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>RFLU_ModABC.F90</td>
<td>25/09/12 08:49</td>
<td>36 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModAllocateMemory.F90</td>
<td>25/09/12 08:49</td>
<td>54 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModAUSMFlux.F90</td>
<td>25/09/12 08:49</td>
<td>56 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModAxisymmetry.F90</td>
<td>25/09/12 08:49</td>
<td>21 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModBessel.F90</td>
<td>25/09/12 08:49</td>
<td>14 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModFaceGradAccessList.F90</td>
<td>25/09/12 08:49</td>
<td>20 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModBilinearPatch.F90</td>
<td>25/09/12 08:49</td>
<td>24 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModBoundaryTests.F90</td>
<td>25/09/12 08:49</td>
<td>13 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModBoundConvertCv.F90</td>
<td>25/09/12 08:49</td>
<td>18 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModBoundLists.F90</td>
<td>25/09/12 08:49</td>
<td>102 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModBoundXvUtils.F90</td>
<td>25/09/12 08:49</td>
<td>79 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModCellFaceEdgeInfo.F90</td>
<td>25/09/12 08:49</td>
<td>21 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModCellMapping.F90</td>
<td>27/11/13 15:02</td>
<td>58 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModColoring.F90</td>
<td>25/09/12 08:49</td>
<td>41 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModCommLists.F90</td>
<td>25/09/12 08:49</td>
<td>101 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModConstraintUtils.F90</td>
<td>25/09/12 08:49</td>
<td>31 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModConvertCv.F90</td>
<td>25/09/12 08:49</td>
<td>28 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModConvertGrid3DTo2D.F90</td>
<td>25/09/12 08:49</td>
<td>17 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModCopyData.F90</td>
<td>25/09/12 08:49</td>
<td>16 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModDeallocateMemory.F90</td>
<td>25/09/12 08:49</td>
<td>45 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModDifferentiationBFaces.F90</td>
<td>25/09/12 08:49</td>
<td>53 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModDifferentiationCells.F90</td>
<td>25/09/12 08:49</td>
<td>103 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModDifferentiationFaces.F90</td>
<td>25/09/12 08:49</td>
<td>43 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModDifferentiationCellFlux.F90</td>
<td>25/09/12 08:49</td>
<td>23 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModDifferentiationDimen.F90</td>
<td>25/09/12 08:49</td>
<td>47 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModEdgeList.F90</td>
<td>25/09/12 08:49</td>
<td>61 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModEntropyFixes.F90</td>
<td>25/09/12 08:49</td>
<td>6 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModExactFlow.F90</td>
<td>25/09/12 08:49</td>
<td>53 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModExplicitMultiStage.F90</td>
<td>25/09/12 08:49</td>
<td>29 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModFaceList.F90</td>
<td>25/09/12 08:49</td>
<td>130 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModFlowHardCode.F90</td>
<td>25/09/12 08:49</td>
<td>9 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModForcesMoments.F90</td>
<td>25/09/12 08:49</td>
<td>92 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModGatherData.F90</td>
<td>25/09/12 08:49</td>
<td>25 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModGENXAdmin.F90</td>
<td>25/09/12 08:49</td>
<td>133 KB</td>
<td>Fortran...e code</td>
</tr>
<tr>
<td>RFLU_ModGENXIO.F90</td>
<td>25/09/12 08:49</td>
<td>150 KB</td>
<td>Fortran...e code</td>
</tr>
</tbody>
</table>

1 of 100 selected, 1.36 TB available
Modules

- Fortran allows definition of new data types: **derived data types**
- Examples:

  ```fortran
  TYPE :: t_point
    REAL :: x, y
  END TYPE t_point
  
  TYPE :: t_line
    REAL :: intercept, slope
  END TYPE t_line
  
  TYPE :: t_circle
    REAL :: radius
    REAL :: origin(2)
  END TYPE t_circle
  
  Recommendation: Use _t prefix or postfix to distinguish type definition from variables/constants
  
  Modules are well suited to contain derived data types and the procedures to manipulate them
  
  Otherwise, would need type definition in every procedure that needs them
  ```
Example module:

MODULE ModIntersections

IMPLICIT NONE

TYPE :: t_point
  REAL :: x, y
END TYPE t_point

TYPE :: t_line
  REAL :: intercept, slope
END TYPE t_line

TYPE :: t_circle
  REAL :: radius
  REAL :: origin(2)
END TYPE t_circle

CONTAINS

SUBROUTINE IntersectLineCircle(line, circle, point1, point2)
  IMPLICIT NONE
  TYPE(t_circle), INTENT(IN) :: circle
  TYPE(t_line), INTENT(IN) :: line
  TYPE(t_point), INTENT(OUT) :: point1, point2
  ...
END SUBROUTINE IntersectLineCircle

END MODULE ModIntersections
The example definitions could be used as follows:

```fortran
PROGRAM typedemo

USE ModIntersections

IMPLICIT NONE

TYPE(t_point) :: ipoint1, ipoint2
TYPE(t_line) :: line
TYPE(t_circle) :: circle

line%intercept = 1.05
line%slope = -2.31

circle%origin(1) = -0.72
circle%origin(2) = 0.59
circle%radius = 2.17

CALL IntersectLineCircle(line, circle, ipoint1, ipoint2)

WRITE(*,*) 'Intersection point 1:', ipoint1%x, ipoint1%y
WRITE(*,*) 'Intersection point 1:', ipoint2%x, ipoint2%y

END PROGRAM typedemo
```

- **Modules**
  - Contains derived data type definition and procedures
  - Declaration with derived data type definition
  - Set components of derived data types using `%`
  - Call subroutine with derived data types as arguments

---

**PREC**
Professorship of Renewable Energy Carriers
Overview

- Structuring a program
- Functions
- Makefiles
- Subroutines
- Functions & subroutines: discussion
- Modules
- Arrays
  - Dynamic memory allocation
  - Arrays as arguments
- Additional topics
Arrays: Dynamic Memory Allocation

- Previously discussed arrays with known dimensions
- In practice, array dimensions usually problem-dependent
- Such arrays are declared with
  
  ```fortran
  REAL, ALLOCATABLE :: a(:,:,)
  ```
  
or
  ```fortran
  REAL, DIMENSION(:,:,), ALLOCATABLE :: a
  ```

- Once dimensions known, can allocate memory:
  ```fortran
  ALLOCATE(a(m,n),STAT=ef)
  IF ( ef /= 0 ) THEN
      WRITE(*,*) 'ERROR!'
      STOP
  END IF ! ef
  ```

**ALLOCATE** returns an integer status variable

Non-zero status variable indicates memory allocation was unsuccessful

Values and meaning of non-zero status variable compiler-dependent

**ef** is short for “error flag”
Arrays: Dynamic Memory Allocation

- If array no longer needed, can be deallocated:

  ```fortran
  DEALLOCATE(a, STAT=ef)
  IF ( ef /= 0 ) THEN
    WRITE(*,*) 'ERROR!'
    STOP
  END IF ! ef
  ```

  **DEALLOCATE** returns an integer status variable
  Non-zero status variable indicates memory deallocation was unsuccessful
  Values and meaning of non-zero status variable compiler-dependent

- Before allocation and after deallocation, array elements cannot be accessed!

- Consider following example…
Arrays: Dynamic Memory Allocation

PROGRAM test
IMPLICIT NONE
INTEGER :: ef,m,n
REAL, ALLOCATABLE :: a(:,:)
WRITE(*,*) 'Enter m,n:'
READ(*,*) m,n
ALLOCATE(a(m,n),STAT=ef)
IF ( ef /= 0 ) THEN
  WRITE(*,*) 'ERROR!'
  STOP
END IF ! ef
a = 1.0
DEALLOCATE(a,STAT=ef)
IF ( ef /= 0 ) THEN
  WRITE(*,*) 'ERROR!'
  STOP
END IF ! ef
WRITE(*,*) a(1,1)
END PROGRAM test

Stepping out-of-bounds leads to run-time error:

Enter m,n:
4
2
Program received signal SIGSEGV:
Segmentation fault - invalid memory reference.

Backtrace for this error:
#0 0x104d492d2
#1 0x104d49a8e
#2 0x7fff875c1f19
#3 0x104e1db4a
#4 0x104e20514
#5 0x104e2104e
#6 0x104d40d87
#7 0x104d40dce
Segmentation fault: 11

Compare this with next example...
PROGRAM test
IMPLICIT NONE
INTEGER :: ef,m,n
REAL, ALLOCATABLE :: a(:,:)
WRITE(*,*) 'Enter m,n:'
READ(*,*) m,n
ALLOCATE(a(m,n),STAT=ef)
IF ( ef /= 0 ) THEN
  WRITE(*,*) 'ERROR!'
  STOP
END IF ! ef
a = 1.0
WRITE(*,*) a(m+1,n)
DEALLOCATE(a,STAT=ef)
IF ( ef /= 0 ) THEN
  WRITE(*,*) 'ERROR!'
  STOP
END IF ! ef
END PROGRAM test

Arrays: Dynamic Memory Allocation

Stepping out-of-bounds leads to run-time error:

Enter m,n:
4
2
At line 20 of file test.f90
Fortran runtime error: Index '5' of dimension 1 of array 'a' above upper bound of 4
Arrays: Dynamic Memory Allocation

- If unsure whether a allocated, use ALLOCATED(a) function:
  - Returns logical variable, set to true if a allocated, false otherwise
  - Can only be used if a declared with ALLOCATABLE attribute
- If unsure about dimensions of a, use LBOUND(a, i) and UBOUND(a, i)
Arrays: Dynamic Memory Allocation

• Good programming practice:
  ‣ *Always* check status variables
  ‣ Write helpful error messages
Arrays: Dynamic Memory Allocation

- How could error handling in adjacent code be improved?

```fortran
PROGRAM allocdealloc
IMPLICIT NONE
INTEGER :: ef, i, j, m, n
REAL, ALLOCATABLE :: a(:, :)
WRITE(*,*) 'Enter m,n:
READ(*,*) m, n
ALLOCATE(a(m, n), STAT=ef)
IF (ef /= 0) THEN
    WRITE(*,*) 'ERROR!'
    STOP
END IF ! ef

a = 1.0

DEALLOCATE(a, STAT=ef)
IF (ef /= 0) THEN
    WRITE(*,*) 'ERROR!'
    STOP
END IF ! ef

END PROGRAM allocdealloc
```
Overview

- Structuring a program
- Functions
- Makefiles
- Subroutines
- Functions & subroutines: discussion
- Modules
- Arrays
  - Dynamic memory allocation
  - Arrays as arguments
- Additional topics
### Arrays as Arguments: Background

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array dimension</td>
<td>Number of subscripts; each subscript has lower and upper bound</td>
</tr>
<tr>
<td>Array rank</td>
<td>Same as array dimension</td>
</tr>
<tr>
<td>Dimension extent</td>
<td>Number of elements in that dimension</td>
</tr>
<tr>
<td>Array size</td>
<td>Total number of elements in array</td>
</tr>
<tr>
<td>Array shape</td>
<td>Determined by rank and dimension extents</td>
</tr>
</tbody>
</table>
## Arrays as Arguments: Background

<table>
<thead>
<tr>
<th>Array</th>
<th>Shape</th>
<th>Dimension/ Rank</th>
<th>Dimension extents</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>a(2,3)</td>
<td><img src="image" alt="Diagram" /></td>
<td>2</td>
<td>2,3</td>
<td>6</td>
</tr>
<tr>
<td>b(0:1,-1:1)</td>
<td><img src="image" alt="Diagram" /></td>
<td>2</td>
<td>2,3</td>
<td>6</td>
</tr>
<tr>
<td>c(-1:1,0:3)</td>
<td><img src="image" alt="Diagram" /></td>
<td>2</td>
<td>3,4</td>
<td>12</td>
</tr>
<tr>
<td>e(5)</td>
<td><img src="image" alt="Diagram" /></td>
<td>1</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>f(-3:1)</td>
<td><img src="image" alt="Diagram" /></td>
<td>1</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>
Arrays as Arguments

- Four types of arrays in Fortran:

<table>
<thead>
<tr>
<th>Shape</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit-shape array</td>
<td>Explicitly declared</td>
</tr>
<tr>
<td></td>
<td>As dummy argument or as local variable</td>
</tr>
<tr>
<td>Assumed-shape array</td>
<td>Not given</td>
</tr>
<tr>
<td></td>
<td>As dummy argument provided procedure has explicit interface, cannot be used in main program</td>
</tr>
<tr>
<td>Automatic array</td>
<td>Explicitly declared, at least one index bound not constant</td>
</tr>
<tr>
<td></td>
<td>Not as dummy argument, cannot be used in main program</td>
</tr>
<tr>
<td>Deferred-shape array</td>
<td>Not given, determined with ALLOCATE statement</td>
</tr>
<tr>
<td>(Allocatable array)</td>
<td>Not as dummy argument, can be used as local variable</td>
</tr>
</tbody>
</table>
Arrays as Arguments

SUBROUTINE arrayargs(m,n,a,b)

IMPLICIT NONE

! Arguments
INTEGER, INTENT(IN) :: m,n
REAL, INTENT(INOUT) :: a(m,n) ! Explicit-shape array
REAL, INTENT(INOUT) :: b(:, :) ! Assumed-shape array

! Locals
INTEGER :: r,s
REAL :: c(m,n),d(5,n) ! Automatic arrays
REAL :: e(5) ! Explicit-shape array
REAL, ALLOCATABLE :: f(:, :, :) ! Deferred-shape array

! Allocate memory
r = ...
s = ...
ALLOCATE(f(3,r,s),STAT=ef)
...
DEALLOCATE(f,STAT=ef)
...
END SUBROUTINE arrayargs
Arrays as Arguments

SUBROUTINE arrayargs(m,n,a,b)

IMPLICIT NONE

! Arguments
INTEGER, INTENT(IN) :: m,n
REAL, INTENT(INOUT) :: a(m,n) ! Explicit-shape array
REAL, INTENT(INOUT) :: b(:,,:)
  ! Assumed-shape array

! Locals
INTEGER :: r,s
REAL :: c(m,n),d(5,n)
REAL :: e(5)
REAL, ALLOCATABLE :: f(:,,:,:)
  ! Deferred-shape array

! Allocate memory
r = …
s = …
ALLOCATE(f(3,r,s),STAT=ef)
…
DEALLOCATE(f,STAT=ef)
…
END SUBROUTINE arrayargs

Assumed-shape array can only be
dummy argument in procedure with
explicit interface, so either
1. calling routine includes explicit
   interface for arrayargs, or
2. arrayargs is placed in MODULE
   and USED in calling routine
Arrays as Arguments

```fortran
PROGRAM arrayexpshape
IMPLICIT NONE
INTEGER :: a(-3:2,4:5)
a = 1
CALL sub(a)
CONTAINS
SUBROUTINE sub(b)
INTEGER, INTENT(IN) :: b(:, :)
WRITE(*, *) LBOUND(b, 1), UBOUND(b, 1)
WRITE(*, *) LBOUND(b, 2), UBOUND(b, 2)
END SUBROUTINE sub
END PROGRAM arrayexpshape
```

- Careful when passing array with non-unity lower bounds!
- Output of code on left: Not
  
  -3  2
  4  5

  but
  1  6
  1  2

- Q: Can you think of a workaround?
## Additional Topics

<table>
<thead>
<tr>
<th>Topic</th>
<th>Brief description</th>
<th>Additional information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Namelist input/output</td>
<td>Simplified input/output of group of variables</td>
<td>Chapman (2008): 14.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ellis et al. (1994): 15.7</td>
</tr>
<tr>
<td>Recursion</td>
<td>Enable procedures to call themselves</td>
<td>Chapman (2008): 13.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ellis et al. (1994): 11.5</td>
</tr>
<tr>
<td>Object-oriented programming (a little in Fortran 90/95, but mostly in Fortran 2003)</td>
<td>Enhanced modularity, flexibility, abstraction</td>
<td>Chapman (2008): 16</td>
</tr>
</tbody>
</table>
## Additional Topics

<table>
<thead>
<tr>
<th>Topic</th>
<th>Brief description</th>
<th>Additional information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameterized data types</td>
<td>Change representation of data</td>
<td>Chapman (2008): 11</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ellis et al. (1994): 10.2, 14</td>
</tr>
<tr>
<td>Optional arguments</td>
<td>Do not pass all arguments to procedures</td>
<td>Chapman (2008): 13.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ellis et al. (1994): 11.1</td>
</tr>
<tr>
<td></td>
<td>to procedures</td>
<td></td>
</tr>
<tr>
<td>Generic procedures</td>
<td>Procedures for different data types</td>
<td>Chapman (2008): 13.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ellis et al. (1994): 11.6</td>
</tr>
</tbody>
</table>