A numerical code is like your hand-writing and if you write many codes in your career some people will recognize you, when they look at your codes.

Now when you start your career as an “Atmosphere Modeller” you have to decide which way you will go!
DO JJ = 1, EM_kz
EM_ES(JJ) = (a0 + a1 * EM_TempC(JJ)**1 + a2 * EM_TempC(JJ)**2 + a3 * &
EM_TempC(JJ)**3 + a4 * EM_TempC(JJ)**4 + a5 * EM_TempC(JJ)**5 + a6 * &
EM_TempC(JJ)**6) * 100
EM_EW(JJ) = EM_RH(JJ) * EM_ES(JJ) / 100
EM_WVM(JJ) = 18.016 / 28.97 * EM_EW(JJ) / (EM_Pres - EM_EW(JJ)) * 1000
IF (EM_EW(JJ) .GT. EM_ES(JJ)) THEN
EM_EW(JJ) = EM_ES(JJ)
ENDIF
ENDDO

DO JJ = 1, EM_kz
  ! Saturation vapour pressure of water (Pa)
  EM_ES(JJ) = (a0 + a1 * EM_TempC(JJ)**1 + a2 * EM_TempC(JJ)**2 &
               + a3 * EM_TempC(JJ)**3 + a4 * EM_TempC(JJ)**4 &
               + a5 * EM_TempC(JJ)**5 + a6 * EM_TempC(JJ)**6) * 100

  ! Water vapour pressure (Pa)
  EM_EW(JJ) = EM_RH(JJ) * EM_ES(JJ) / 100

  ! Water vapour mixing ratio (g/kg)
  EM_WVM(JJ) = 18.016 / 28.97 * EM_EW(JJ) / (EM_Pres - EM_EW(JJ)) * 1000

  ! Limit EM_EW to maximum value of EM_ES
  IF (EM_EW(JJ) .GT. EM_ES(JJ)) THEN
    EM_EW(JJ) = EM_ES(JJ)
 ENDIF
ENDDO
What is good coding

Code formatting: Be consistent!!!

- **Rule of thumb:**
  - *Consistency more important than specific formatting style*
- No matter how many SPACE's you use for an indent, use it consistently throughout the source code. SPACE's and TAB's do not mix well!
- **Indent** code to better convey the logical structure of your code. Without indenting, code becomes difficult to follow.

```plaintext
if ... then
  ... if ... then
  ... else
  ... end if
  ... else
  ... end if
end if
```

- **Establish a maximum line length** for comments and code to avoid having to scroll the window of the text editor.
- Use SPACE after each “comma” in lists, such as array values and arguments, also before and after the “equal” of an assignment.
  
  \[ \text{Energy} = 0.5 \times k_b \times \text{Temp}(i, j, k) \]

- Use empty lines to provide **organizational clues** to source code, blocks (“paragraphs”-like structure) help the reader in comprehending the logical segmenting.
What is good coding

**Code formatting (Cont.)**

- When a line is broken across several lines, make it **obvious** that the line is incomplete using indentation.

```plaintext
! Parameters to calculate the saturation vapour pressure for water
REAL, PARAMETER ::
a0 = 6.107799961, &
a1 = 4.436518524D-1, &
a2 = 1.428945805D-2, &
a3 = 2.650648471D-4, &
a4 = 3.031240396D-6, &
a5 = 2.03408948D-8, &
a6 = 6.136820929D-11
```

**Avoid** to define format statements “far away” from the READ/WRITE statement itself

```plaintext
write ( fileId, 99062) iter ...
99062 format ( ' Number of iterations = ' , i7 )
```

- Even better, avoid label at all and include the format within the statement

```plaintext
write ( fileId , "( ' Number of iterations = ' , i7 )") iter
```
Code formatting (Cont.)

- Enable syntax highlighting in your text editor.
- Use freely available program that help to indent, format, and beautify your source code automatically and consistently.
- I use Eclipse (free software) but everyone should find the best suitable for his or her need.

```fortran
! First put all emissions to zero at each time step
DO  JJ = 1,kz
   DO  II = 1,22
      EM_EMI(JJ,II) = 0.
   ENDDO
ENDDO
```

Code formatting (Cont.)

- **Break** large, complex sections of code into smaller, comprehensible modules (subroutine/functions/methods). A good rule is that modules do not exceed the size of the text editor window.

- **Arrange** and separate your source code logically between files.
Naming convention: Be consistent!

- **Choose** and **stick to** a style for naming various elements of the code, this is one of the most influential aids to understand the logical flow.
- Difficulty to find a proper name for a routine/variable may indicate a need to further analysis to define its purpose.
- A name should tell **what** rather than **how**, avoid names that expose underlying implementation.
- Ideally you would like to be able to read the code as **prose**.

Naming convention (Cont.)

```plaintext
! Output variables:
REAL, DIMENSION(kz) :: EM_Sunfrac
REAL, DIMENSION(kz) :: EM_Sun_Par
REAL, DIMENSION(kz) :: EM_Sun realpath, EM_Sun realpath
REAL, DIMENSION(kz) :: EM_Shade realpath, EM_Shade realpath

REAL(real_x) ::
  CH_O2,
  CH_R02_realpath,
&  O2 concentration in [molecules / cm^3]
&  RO2 concentration in [molecules / cm^3]
```
Naming convention: Be consistent!

- **Define** all variables and parameters in the beginning of the code.

- If available also provide the **units** – so it is easier to find errors in the code specially when results are orders of magnitude too high or too low.

---

**Naming convention (Cont.)**

---

```plaintext
<table>
<thead>
<tr>
<th>Type</th>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>DelFlag</td>
<td>0 = no emissions, 1 = old Megan code, 2 = new Megan code, 3 = SIMBIM</td>
</tr>
<tr>
<td>INTEGER</td>
<td>Year</td>
<td>Year</td>
</tr>
<tr>
<td>INTEGER</td>
<td>Mon</td>
<td>Month</td>
</tr>
<tr>
<td>INTEGER</td>
<td>Day</td>
<td>Julian day</td>
</tr>
<tr>
<td>INTEGER, PARAMETER</td>
<td>num_layer</td>
<td>Number of vertical layers inside the model (not only canopy)</td>
</tr>
<tr>
<td>INTEGER, PARAMETER</td>
<td>num_layerCanopy</td>
<td>Number of vertical layers inside the canopy</td>
</tr>
<tr>
<td>REAL</td>
<td>TT</td>
<td>Run parameter for the time</td>
</tr>
<tr>
<td>REAL</td>
<td>PAR</td>
<td>Incident photosynthetic active radiation [umol/m²/s]</td>
</tr>
<tr>
<td>REAL</td>
<td>Press</td>
<td>Pressure in the canopy (Pa)</td>
</tr>
<tr>
<td>REAL</td>
<td>SMST</td>
<td>Soil moisture (%)</td>
</tr>
<tr>
<td>REAL</td>
<td>SunST</td>
<td></td>
</tr>
<tr>
<td>REAL</td>
<td>PAR_day</td>
<td>Daily average PAR</td>
</tr>
<tr>
<td>REAL</td>
<td>SWR_day</td>
<td>Daily average short wave radiation (W/m²)</td>
</tr>
<tr>
<td>REAL</td>
<td>TempK_day</td>
<td>Daily average temperature Kelvin</td>
</tr>
<tr>
<td>REAL, DIMENSION(12)</td>
<td>LAI</td>
<td>Array of Leaf area index for each month</td>
</tr>
<tr>
<td>REAL, PARAMETER</td>
<td>P4</td>
<td>3.14159</td>
</tr>
<tr>
<td>REAL, PARAMETER</td>
<td>a_plant</td>
<td>57.29578</td>
</tr>
<tr>
<td>REAL, DIMENSION(ka)</td>
<td>k_layers</td>
<td>Array of height of the layers</td>
</tr>
<tr>
<td>REAL, DIMENSION(ka)</td>
<td>TempK</td>
<td>Temperature in Kelvin</td>
</tr>
<tr>
<td>REAL, DIMENSION(ka)</td>
<td>Wind</td>
<td>Wind speed [m/s]</td>
</tr>
<tr>
<td>REAL, DIMENSION(ka)</td>
<td>RH</td>
<td>Relative humidity in %</td>
</tr>
<tr>
<td>REAL, DIMENSION(ka)</td>
<td>SpecHum</td>
<td>Specific humidity in (g / m³)</td>
</tr>
<tr>
<td>REAL, DIMENSION(ka)</td>
<td>LAD</td>
<td>Leaf area density as (m²/1) in the canopy</td>
</tr>
</tbody>
</table>
```
Naming convention: Be consistent!

- **Avoid elusive** names, open to subjective interpretation like

  ```
  Analyse ( . . . ) // subroutine or function or method
  nnsmcomp1 // variable
  ```

- Use a **verb-noun** method to name routines that perform some **operation-on-a-given-object**. Most names are constructed by concatenating several words, use mixed-case formatting or underscore to ease reading.

  ```
  calculateKineticEnergy ( . . . )
  calculate_kinetic_energy ( . . . )
  ```

---

General comments

- Use as many as possible **comments** this will help you to understand what you have done if you read the code later and it will be much easier for others to learn your code.
Introduction to Fortran

Zhou Putian

Modified from http://www.bu.edu/tech/files/2013/09/intro2fortran.pptx

Outline

• Introduction
• Fortran History
• Basic Syntax
• Makefile
• Additional Syntax
Introduction

Two fundamentally different types of high-level languages:

- Interpreted language (MATLAB, Python, ...)
  - Translation to machine-language is performed at runtime by an interpreter
  - More convenient but slower (no need to declare variables; realize your idea quickly ...)
- Compiled language (Fortran, C, C++, ...)
  - Translation is performed once, then executable is run as frequently as needed without further translation
  - Run faster (suitable for large-scale computing ...)

However, the border between them for some languages are not clear (e.g., Java).

What Language Should I Use?

- Generally, use the language you know the best
- Interpreted languages are great for
  - Interactive applications
  - Code development and debugging
  - Algorithm development
- For major number crunching, compiled languages are preferred (Fortran, C, C++)
Fortran History

• Before Fortran, programs were written in assembly language
  • low-level commands such as “load x from memory into register 7”
    or “add values in registers 10 and 11 and write result to register 4”
• Fortran was the first widely-used high-level computer language
  • 1957
  • Developed by John Backus' team in IBM as an alternative of assembly language
  • Program written on a specially formatted green sheet, then entered as punched cards
Fortran History

- Fortran 66 (1966)
- Fortran 77 (1978)
- Fortran 90 (1991)
  - “fairly” modern (structures, etc.)
  - Current “workhorse” Fortran
- Fortran 95 (minor tweaks to Fortran 90)
- Fortran 2003
  - Gradually being implemented by compiler companies
  - Object-oriented support
  - Interoperability with C is in the standard
- Fortran 2008 (submodules, ...)
- Fortran 2018 (formerly Fortran 2015, minor revision of Fortran 2008, still in development)

- FORTRAN is a compiled language (like C) so the source code (what you write) must be converted into machine code before it can be executed (e.g. Make command)
Program Structure

PROGRAM program_name
    [USE module_names]
    IMPLICIT NONE
    [specification part]
    [execution part]
CONTAINS
    [subprogram part]
END PROGRAM program_name

Fortran Syntax (1)

- Program is contained in a text file called source code or source file
- Source code must be processed by a compiler to create an executable
- Source file suffix can vary (.for, .f, .F, .f90, .F90, ...), but we will always use ".f90"
- Since source file is simply text, it can be written using any text editor
  - emacs, vi, gedit, Notepad++, ...
Fortran Syntax (2)

```fortran
PROGRAM prog_name
  [USE module_names]
  IMPLICIT NONE
  [specification part]
  [execution part]
CONTAINS
  [subprogram part]
END PROGRAM prog_name
```

- Suggestions: Keep the source file the same name as the program:
  - prog_name.f90
- Case insensitive
- Blank spaces serve as delimiter

Fortran Syntax (3)

- **Variable types**: real, integer, complex, logical, character
- **Real** variables have decimals
  - Real can be a whole number, but decimal places are stored internally
  - Even when a real is a whole number, it's good practice to write one decimal place
  - **3.0** rather than **3**
- **Integer** variables do not have decimals
- **Logical** variables only have two values: .TRUE. and .FALSE.
Fortran Syntax (4)

- Integer arithmetic is *truncated*, not rounded
  - $3/2 = 1$
  - $2/3 = 0$
  - $5/(-2) = -2$
- If at least one of them is real, results would be also real
  - $3.0/2.0 = 1.5$
  - $2.0/3.0 = 0.6666667$
  - $5.0/(-2) = -2.50000000$
- **Character** variables contain literal text
  - Enclosed in single or double quotes
    - "A", 'Hello', "Fortran is a computer language."
  - Blank spaces within quotes are significant. They are part of the string (contains more than one characters).

Fortran Syntax (5)

- Need to **declare** the type for every variable
  
  ```fortran
  real :: velocity, mass, pi  
  integer :: imax, jdim  
  character :: p 
  ```

- Variables must start with a letter (a-z); can mix with digits (0-9); also underscores ( _ ); but no blanks; name length <= 31
- Strongly recommend to adopt the practice of declaring with **implicit none** (**MUST DURING OUR MODELING COURSE**)
  - This promises the compiler that you will declare all variables
  - This goes before any type declaration statements
Fortran Syntax (6)

- **print *, variables or write(*,*) variables**
  - “list-directed” output
  - Simple way to produce output on the screen
  - Examples,

    ```fortran
    print*, ra, 'This is my character string.'
    write(*,*) "I am at bottom.", ib
    ```

Fortran Syntax (7)

- Comment character is `!`
  - Anything from `!` to end of line ignored by the compiler
  - Use comments liberally to document source code
- Ampersand, `&`, at end of line tells compiler that statement is continued on next source line
- Spaces don’t matter except within literal character strings
  - use them liberally to make code easy to read, e.g., before and after equal signs
- Note that source lines do not end with semicolons (as in C or MATLAB)
Exercise 1

- Write a “hello world” program with your editor
  - Program should print a character string
  - Save it to a file name with a .f90 suffix (for example, hello.f90)

Solution Exercise 1

!================================
! hello world
!================================
program hello
  implicit none
  write(*,*) &
    'Hello world. '
end program hello

Compilation (1)

- A compiler is a program that reads source code and converts it to a form usable by the computer
- Internally, these steps are performed:
  - preprocess source code
  - check source code for syntax errors
  - compiler translates source code to assembly language
  - assembler translates assembly language to machine language
  - linker gathers machine-language modules and libraries
- All these steps sometimes loosely referred to as “compiling”
Compilation (2)

- Code compiled for a given processor architecture will not generally run on other processors
- However, this problem will come to you later, when you start to run your code on different machines

Compilation (3)

- Compilers have huge numbers of options
- Compile hello.f90 on your laptop
- If it simply returns a Unix prompt, it worked
- If you get error messages, read them carefully and see if you can fix the source code and re-compile
- Once it compiles correctly, type the executable name at the Unix prompt, and it will print your string

$ gfortran hello.f90
$ ./a.out
Arithmetic

- $+, -, *, /, **$ (power)
  
  $2.5^{1.5} \rightarrow 2.5^{*1.5}$

- Built-in math functions ($\sin$, $\acos$, $\exp$, $\log$, $\log10$, ...)
  
  - Arguments are in parentheses
    
    $\sin(0.6)$

- Exponential notation indicated by letter “e” or “d”
  
  $5.3*10^4 \rightarrow 5.3e4$

  $5.3*10^4 \rightarrow 5.3d4$ (double precision)

More List-Directed I/O

- $\text{read } *$ is list-directed read, analogous to $\text{print}*$$
- $\text{read}(*,*) \text{ VS write}(*,*)$
- Examples

```fortran
print *, 'Enter a float and an integer:'
read *, x, j
print *, 'float = ', x, ' integer = ', j

write(*,*) 'Enter a float and an integer'
read(*,*) x, j
write(*,*) 'float = ', x, ' integer = ', j
```
Exercise 2

- Write program to prompt for a Celsius temperature, convert it to Fahrenheit, and print the result.
  - make sure you declare all variables
  - use decimal points with all reals, even if they're whole numbers

\[
F = \frac{9}{5} \times C + 32 \\
C = \frac{5}{9} \times (F - 32)
\]

Solution Exercise 2

```fortran
!====================================
! ctof.f90
! prompt for Celcius temperature
! print Fahrenheit value
!====================================
program ctof
  implicit none
  real :: c, f

  write(*,*) 'Enter temperature in Celcius.'
  read(*,*) c
  f = (9.0/5.0)*c + 32.0
  write(*,*) 'T = ', f, 'degrees Fahrenheit'

end program ctof
```
Arrays (1)

• Specify static dimensions in declaration:
  
  real, dimension(10,3,5) :: x
  real :: m(2,3), n(100)
  integer, dimension(10) :: I

• Can also specify ranges of dimension indices
  
  integer, dimension(3:11, -15:-2) :: ival, jval

• Access array elements using parenthesis
  
  a = y(3) + y(4)

• Fortran: column-major array

Arrays (2)

• Dynamic allocation
  
  • Useful when size is not known at compile time, e.g., input value
  • Need to specify number of dimensions in declaration
  • Need to specify that it's an allocatable array
    
    real, dimension(:,:,:), allocatable :: x, y

• allocate function performs allocation
  
  allocate( x(ni,nj,nk), y(ldim,mdim,ndim) )

• When you're done with the variables, deallocate with
  
  deallocate(x, y)

  But it is not necessary at very end of code; Fortran will clean them up for you

• Avoid using “I” because it could be mistaken for “1” or “i” (You could use “L”)

• Good idea to establish your own naming conventions and follow through with them
Parameters

- If variable has known, fixed, value, declare as parameter and initialize in declaration
  
  ```fortran
  integer, parameter :: idim = 100, jdim = 200
  ```

- Compiler substitutes values wherever variables appear in code
- Efficient, since there are no memory accesses
- Often used for declaring arrays

  ```fortran
  integer, parameter :: idim = 100, jdim = 200
  real, dimension(idim, jdim) :: x
  integer, dimension(idim) :: iarray
  ```

Exercise 3

- Write a program to prompt for 2 floating-point vectors of length 3, calculate the dot product, and print the result
  
  - Don’t name the code “dot_product” or “dot”
  - Fortran has a “dot_product” intrinsic function
  - there is a Unix command called “dot”
  - Can use array name in list-directed read, and it will expect the appropriate number of values (dimension) separated by spaces or commas

  \[
  c = \sum_{i=1}^{3} a_i * b_i = a_1 * b_1 + a_2 * b_2 + a_3 * b_3
  \]
Solution Exercise 3

```fortran
! dotprod.f90
!
! calculate dot product
!
!
program dotprod
    implicit none
    real :: c
    real, dimension(3) :: a, b
    ! enter data
    !
    ! calculate dot product
    !
    c = a(1)*b(1) + a(2)*b(2) + a(3)*b(3)
    !
    ! print result
    !
    print*, 'Dot product = ', c
end program dotprod
```

Introduction to Fortran

Control

- **Do loop repeats calculation over range of indices**
  ```fortran
do i = 1, 10
    a(i) = sqrt( b(i)**2 + c(i)**2 )
  end do
  ```

- **Can use increment that is not equal to 1**
  - Goes at *end* of do statement, unlike Matlab where it is in the middle
  ```fortran
do i = 10, -10, -2
  ```
Exercise 4

- Modify dot product program to use a do loop
  - Declare scalar real variable to hold the summation
  - Initialize it to zero before the do loop

Solution Exercise 4

```fortran
program dotprod
  implicit none
  integer :: i
  real :: c
  real, dimension(3) :: a, b

  !-------------------------------
  ! enter data
  !-------------------------------
  print*, 'Enter first vector'
  read*, a
  print*, 'Enter second vector'
  read*, b

  !-------------------------------
  ! calculate dot product using do loop
  !-------------------------------
  c = 0.0
  do i = 1, 3
    c = c + a(i)*b(i)
  enddo

  !-------------------------------
  ! print result
  !-------------------------------
  print*, 'Dot product = ', c
end program dotprod
```
If-Then-Else (1)

- Conditional execution of block of source code
- Based on relational operators
  - < less than
  - > greater than
  - == equal to
  - <= less than or equal to
  - >= greater than or equal to
  - /= not equal to
  - .and.
  - .or.

If-Then-Else (2)

```fortran
if( x > 0.0  .and.  y > 0.0 ) then
   z = 1.0/(x+y)
else if ( x < 0.0  .and.  y < 0.0) then
   z = -2.0/(x+y)
else
   write(*,*) 'Error condition'
end if
```
Exercise 5

• In dot product code, check if the magnitude of the dot product is less than 1e-6 using the absolute value function \texttt{abs}. If it is, print a message. In either case, print the result as before.

Never check if a real number equals a specific value. Instead, check if it is within a certain range, e.g., with \texttt{abs}.

Solution Exercise 5

```fortran
!-------------------------------------
! dotprod.f90
! prompt for two real vectors of length 3, 
! calculate dot product; if magnitude of 
! dot product is less than 1e-6, print 
! warning, 
! then print result 
!-------------------------------------
program dotprod
  implicit none
  integer :: i
  real :: c
  real, dimension(3) :: a, b

  !----------------------------
  ! input vectors
  !----------------------------
  print*, 'Enter first vector'
  read*, a
  print*, 'Enter second vector'
  read*, b

  !----------------------------
  ! calculate dot product 
  !----------------------------
  c = 0.0
  do i = 1, 3
    c = c + a(i)*b(i)
  enddo

  !----------------------------
  ! check magnitude 
  !----------------------------
  if(abs(c) < 1.0e-6) then
    print*, '--------------------------'
    print*, 'Warning: dot product < 1e-6'
    print*, '--------------------------'
  endif

  !----------------------------
  ! print result 
  !----------------------------
  print*, 'dot product = ', c
end program dotprod
```
Array Syntax (1)

- Fortran can perform operations on entire arrays
  - Like MATLAB, unlike C.
- To add two arrays, simply use
  \[ c = a + b \]  
  \( a, b, c \) are arrays of the same shape and size
- Can also operate on array sections
  \[ c(-5:10) = a(0:15) + b(0:30:2) \]  
  must have same shape
  - Here we use \( b(0), b(2), b(4), \) etc. due to increment specification
  - Numbers of elements must be consistent
- Don’t assume that all MATLAB matrix rules apply
  \[ c = a * b \]  
  * is elemental multiply, not matrix multiply

Array Syntax (2)

- There are intrinsic functions to perform some operations on entire arrays
  - \texttt{sum(x)}: sum up all the elements in \( x \)
  - \texttt{product(x)}: multiply all the elements in \( x \)
  - \texttt{minval(x)}: minimum value in \( x \)
  - \texttt{maxval(x)}: maximum value in \( x \)
  - \texttt{matmul(x, y)}: matrix multiplication of \( x \) and \( y \)
Exercise 6

- Modify dot product code to use array syntax instead of do loop
- use "sum" intrinsic to sum components

Solution Exercise 6

```fortran
!=====================================!
! dotprod.f90
! prompt for two real vectors of length 3,
! calculate dot product using array syntax;
! magnitude of dot product is less than 1e-6,
! if magnitude of dot product is less than 1e-6,!
! print warning, then print result
!=====================================!
program dotprod
  implicit none
  integer :: i
  real :: c
  real, dimension(3) :: a, b
  !----------------------------
  ! input vectors
  !----------------------------
  print*, 'Enter first vector'
  read*, a
  print*, 'Enter second vector'
  read*, b
  !----------------------------
  ! calculate dot product
  !----------------------------
  c = sum(a*b)
  !----------------------------
  ! check magnitude
  !----------------------------
  if(abs(c) < 1.0e-6) then
    print*, '---------------------'
    print*, 'Warning: dot product < 1e-6'
    print*, '---------------------'
  endif
  !----------------------------
  ! print result
  !----------------------------
  print*, 'dot product = ', c
end program dotprod
```
Subprograms

• Calculations may be grouped into subroutines and functions to perform specific tasks such as:
  - read or write data
  - initialize data
  - solve a system of equations

• Function returns a single object (number, array, etc.), and usually does not alter the arguments
  - Fortran uses pass-by-reference; change of variables’ values passed into subprogram will be changed after returning
  - Altering certain argument’s value in a subprogram, considered a “side effect,” is bad programming practice. Changing a pre-defined constant is an example. It may either cause a segmentation fault or worse, the variable got changed.

• Subroutine transfers calculated values (if any) through arguments

Functions

• Example: Convert Celsius degree to Fahrenheit degree
  
  ```fortran
  real function fahrenheit(c)
  real :: c
  fahrenheit = (9.0/5.0)*c + 32.0 ! Convert Celsius to fahrenheit
  end function fahrenheit
  
  function fahrenheit(c) result(f)
  real :: c
  real :: f
  f = ...
  
  • Use:
    f = fahrenheit(0.0)
    f = fahrenheit(degc)
  
  • Names of dummy arguments don’t have to match actual names```
Subroutines

• Subroutine converting Celsius degree to Fahrenheit degree

```fortran
subroutine temp_conversion(celsius, fahrenheit)
    real :: celsius, fahrenheit
    fahrenheit = (9.0/5.0)*celsius + 32.0
end subroutine temp_conversion
```

• Use:

```fortran
call temp_conversion(c, f)
```

• As with function, names of dummy arguments don’t have to match actual names

Exercise 7

• Modify dot-product program to use a subroutine to compute the dot product
  • The subroutine definition may go before or after the main program in source code, but I recommend to contain them inside the program
  • Don’t forget to declare arguments
  • Give the subroutine a name different than the program
Program dotprod

```fortran
program dotprod
  implicit none
  integer :: i
  real :: c
  real, dimension(3) :: a, b

  !----------------------------
  ! input vectors
  !----------------------------
  print*, 'Enter first vector'
  read*, a
  print*, 'Enter second vector'
  read*, b

  ! calculate dot product
  call dp(a, b, c)

  !----------------------------
  ! check magnitude
  !----------------------------
  if(abs(c) < 1.0e-6) then
    print*, '--------'
    print*, 'Warning: dot product <1e-6'
    print*, '--------'
  endif

  !----------------------------
  ! print result
  !----------------------------
  print*, 'dot product = ', c
contains
  subroutine dp(x,y,d)
    implicit none
    real :: d
    real, dimension(3) :: x, y
    d = sum(x*y)
  end subroutine dp
end program dotprod
```

### Solution Exercise 7

- Modify dot-product program to use a function to compute the dot product
  - Name the function different than the program name, like dotp

- Modify the fahrenheit function into a function, `convert`, such that if input is in fahrenheit, it returns the Celsius equivalence. If input is in Celsius, it returns fahrenheit. (Hint: extra input parameter)
program dotprod
  implicit none
  integer :: i
  real :: c, dotp
  real, dimension(3) :: a, b

  !----------------------------
  ! input vectors
  !----------------------------
  print*, 'Enter first vector'
  read*, a
  print*, 'Enter second vector'
  read*, b

  !----------------------------
  ! calculate dot product
  !----------------------------
  c = dotp(a,b)

  !----------------------------
  ! check magnitude
  !----------------------------
  if(abs(c) < 1.0e-6) then
    print*, '-'
    print*, 'Warning: dot product < 1e-6'
    print*, '-'
  endif

  !----------------------------
  ! print result
  !----------------------------
  print*, 'dot product = ', c
contains
  function dotp(x,y)
    implicit none
    real :: dotp
    real, dimension(3) :: x, y
    dotp = sum(x*y)
  end function dotp
end program dotprod

real function converts(temp, mode)
  real :: temp
  integer :: mode

  if (mode == 0) then  ! Celcius to Fahrenheit
    converts = (9.0/5.0)*temp + 32.0
  else  ! Fahrenheit to Celcius
    converts = (temp - 32.0) * (5.0/9.0)
  end if
end function converts
Basics of Code Management

• Large codes usually consist of multiple files
  • Bad practice to have everything in the same file
  • I usually create a separate file for each subprogram
  • Easier to edit
  • Can recompile one subprogram at a time

• Files can be compiled, but not linked, using –c option; then object files can be linked
  gfortran –c mycode.f90
  gfortran –c myfunc.f90
  gfortran –o mycode mycode.o myfunc.o

Exercise 8

• Put dot-product subroutine and main program in separate files
• Give main program same name you have been using for code, e.g., “program dotprod” and dotprod.f90
• Give subroutine same name you used for subroutine, e.g., “subroutine dp” and dp.f90
• Compile, link, and run
Solution Exercise 8

```fortran
program dotprod
 implicit none
 integer :: i
 real :: c
 real, dimension(3) :: a, b

 !----------------------------------------
 ! prompt for two real vectors of length 3,
 ! calculate dot product using subroutine
 ! from separate file
 !----------------------------------------

 !----------------------------------------
 ! calculate dot product
 !----------------------------------------
 call dp(a,b,c)
 !----------------------------------------

end program dotprod

subroutine dp(x,y,d)
 implicit none
 real :: d
 real, dimension(3) :: x, y
 d = sum(x*y)
end subroutine dp
```

Introduction to Fortran

Makefiles (1)

- Make is a Unix utility to help manage codes
- When you make changes to files, it will
  - Automatically deduce which files need to be compiled and compile them
  - Link latest object files
- Makefile is a file that tells the make utility what to do
- Default name of file is “makefile” or “Makefile”
- Usage:
  - $ make
  - (or) $ make -f makefile
- Check the full manual of 'GNU make' at http://www.gnu.org/software/make/manual/make.html
Makefiles (2)

- Makefile contains different sections with different functions
  - The sections are *not* executed in order!

- Comment character is `#`

- There are defaults for some values, but I like to define everything explicitly

---

Makefiles (3)

- *example makefile:*

```makefile
### compiler
F90 = gfortran
COMMONFLAGS = -O3
COMPOPTFLAGS = -c \$(COMMONFLAGS)
LINKFLAGS = \$(COMMONFLAGS)

### objects
OBJ = mymain.o subl.o

### compile source code
mymain.o: mymain.f90
   \$(F90) -c $^ -o $@
subl.o: subl.f90
   \$(F90) -c $^ -o $@

### compile and link
myex: \$(OBJ)
   \$(F90) -o $@ \$(LINKFLAGS) \$(OBJ)
```
Makefiles (4)

• Variables
  • Some character strings appear repeatedly in makefiles
  • It’s convenient to give them names so if they are changed, you only have to do it in one place
  • To define variable:
    name = string
  • No quotes are required for the string
  • String may contain spaces
  • “name” is any name you want
  • Variable names are usually all capitals
  • To continue line, use \ character

Makefiles (5)

• Variables
  • To use variable, either of these works:
    $(name)
    ${name}
  • Example:
    • Define compiler
      F90 = gfortran
    • To use elsewhere in makefile:
      $(F90)
Makefiles (6)

Good practice to define compiler info in variables
F90 = gfortran
COMMONFLAGS = -O3
COMPFLAGS = -c $(COMMONFLAGS)
LINKFLAGS = $(COMMONFLAGS)

Usually define variable with all object file names
OBJ = mymain.o sub1.o

Makefiles (7)

• Finally, everything falls in place with the definition of a rule
  target: prerequisites
  <tab>recipe

• The target is any name you choose
  • Often use name of executable

• Prerequisites are files that are required by target
  • e.g., executable requires object files

• Recipe tells what you want the makefile to do
### compile source code

```makefile
mymain.o: mymain.f90
   $(F90) -c $^ -o $@
```

```makefile
sub1.o: sub1.f90
   $(F90) -c $^ -o $@
```

### compile and link

```makefile
myexe: $(OBJ)
   $(F90) -o $@ $(LINKFLAGS) $(OBJ)
```

$@, $^: Automatic variables for target, check the full list in the online manual.

---

### Makefiles (9)

- When you type “make,” it will look for a file called “makefile” or “Makefile”
- It then searches for the first target in the file
- In our example (and the usual case) the object files are prerequisites
- It checks the time stamps on the associated .o and .f90 files to see if the .f90 is newer
- If the .f90 file is newer it compiles the source code
Makefiles (10)

- Once all the prerequisites are updated as required, it performs the recipe
- In our case it links the object files and creates our executable file
- Many makefiles have an additional target, “clean,” that removes .o and other files
  
  clean:
  \<tab>\texttt{rm -f *.o}

- When there are multiple targets, specify desired target as argument to make command
  
  make clean

Exercise 9

- Create a makefile for your dot product code
- Include two targets
  - executable
  - clean
- Delete your old object files using “make clean”
- Build your code using the makefile
Solution Exercise 9

Suppose you have one main program file (dotprod.f90) and one file containing the subroutine dp (dp.f90). You can compile them in this way:

$ gfortran -c dp.f90 -o dp.o
$ gfortran -c dotprod.f90 -o dotprod.o
$ gfortran dotprod.o dp.o

or

$ gfortran -c dp.f90 -o dp.o
$ gfortran dotprod.f90 dp.o

*.o are object files, they can be linked to generate the final executable file.

---

### compiler
F90 = gfortran

### objects
OBJ = dotprod.o  dp.o

### compile and link
dotprod: $(OBJ)
<tab>$F90 $(OBJ) -o $@

dotprod.o: dotprod.f90
<tab>$F90 -c $< -o $@

dp.o: dp.f90
<tab>$F90 -c $< -o $@

### Clean object files and executable file
clean:
    rm -f *.o dotprod

---
Kind

- Declarations of variables can be modified using “kind” parameter
- Often used for precision of reals
- Intrinsic function `selected_real_kind(n)` returns kind that will have at least $n$ significant digits
  - $n = 6$ will give you “single precision”
  - $n = 12$ will give you “double precision”
- If you want to change precision, can easily be done by changing one line of code

```fortran
integer, parameter :: rk = selected_real_kind(12)
real(rk) :: x, y, z
real(rk), dimension(101,101,101) :: a
```

Exercise 10

- Modify dot-product code to use kinds to declare double-precision reals
  - Don't forget to modify all files
  - “make” will automatically compile and link
Solution Exercise 10

program dotprod
  implicit none
  !-----------------------------
  ! set kind
  !-----------------------------
  integer, parameter :: rk = selected_real_kind(12)

  integer :: i
  real(rk) :: c
  real(rk), dimension(3) :: a, b

  !-----------------------------
  ! input vectors
  !-----------------------------
  print '*', 'Enter first vector'
  read*, a
  print '*', 'Enter second vector'
  read*, b

  !-----------------------------
  ! calculate dot product
  !-----------------------------
  call dp(a,b,c)
end program dotprod

!-----------------------------
! check magnitude and print results left out here!
!-----------------------------
... end program dotprod

subroutine dp(x,y,d)
  implicit none
  !-----------------------------
  ! dot product of two vectors
  ! of length 3
  !-----------------------------
  integer, parameter :: rk = selected_real_kind(12)
  real(rk) :: d
  real(rk), dimension(3) :: x, y
  d = sum(x*y)
end subroutine dp

Modules (1)

• Program units that group variables and subprograms
• Good for global variables
• Checking of subprogram arguments
  • If type or number is wrong, linker will yell at you
• Can be convenient to package variables and/or subprograms of a given type
Modules (2)

module module-name
implicit none
... variable declarations ...
contains
... subprogram definitions ...
end module module-name

Modules (3)

• You need “contains” if module contains subprograms
• I usually name my modules (and associated files) with 
  _mod in the name, e.g., solvers_mod, solvers_mod.f90
• In program unit that needs to access components of 
  module use module-name
• use statement must be before implicit none
Modules (4)

- **use** statement may specify specific components to access by using “only” qualifier:
  
  ```fortran
  use solvers_mod, only: nvals, x
  ```

- A Fortran style suggestion:
  - Group global variables in modules based on function
  - Employ “use only” for all variables required in program unit
  - All variables then appear at top of program unit in declarations or “use” statements

Modules (5)

- **When linking object files, modules must come first in the list**
  - In my makefiles I create a MODS variable analogous to OBJS
  - Link command then contains $(MODS) $(OBJS)
Exercise 11

- Create module *prec_mod*
  - Separate file called *prec_mod.f90*
  - Parameter *rk*
  - Real kind for double
- Use this module in dot-product program units
- Modify makefile to compile module
  - add module list to dependencies and link recipe

---

Solution Exercise 11

```fortran
! dotprod.f90
! dot product using module to set precision
program dotprod
 ! use module containing precision parameter
use prec_mod, only: rk

implicit none
integer :: i
real(rk) :: c
real(rk), dimension(3) :: a, b

! input vectors
!...
print*, 'Enter first vector'
read*, a
print*, 'Enter second vector'
read*, b

!...
! calculate dot product
!...
call dp(a, b, c)

!...
! check magnitude and print results left out here
!...
end program dotprod

! dot product of two vectors of length 3
!...
subroutine dp(x, y, d)

!...
! set kind
!...
use prec_mod, only: rk
implicit none
real(rk) :: d
real(rk), dimension(3) :: x, y

d = sum(x*y)
end subroutine dp

! precision parameter
!...
single precision: selected_real_kind(6)
double precision: selected_real_kind(12)

module prec_mod
implicit none
integer, parameter :: rk = selected_real_kind(12)
end module prec_mod
```
### compiler
F90 = gfortran

### modules, objects and executable
MOD = prec_mod.o
OBJ = dotprod.o dp.o
EXE = dotprod

all: $(EXE)

%.o: %.f90
 $(F90) -c $< -o $@

### compile and link
$(EXE): $(MOD) $(OBJ)
 $(F90) $^ -o $@

clean:
 rm -f *.o *.mod $(EXE)

---

**Derived Types**

- Analogous to structures in C
- Can package a number of variables under one name

```fortran
type grid
  integer :: nvals
  real, dimension(100,100) :: x, y, jacobian
end type grid

type(grid) :: airfoil

airfoil%nvals = 20
airfoil%x = 0.0  ! array notation, initialize entire array

call calc_jacobian(airfoil)
```
Exercise 12

- Create module with definition of rvec3 type
  - Size of vector nvals = 3
  - not a parameter – can’t have parameter in derived type
  - Real 3-component vector
  - Use prec_mod
- Modify code to use rvec3
- Modify makefile to include new module

Solution Exercise 12

```fortran
! dotprod.f90
! dot product using derived type for vectors
program dotprod
  module containing precision parameter
  use prec_mod, only: rk

  module containing derived type
  use rvec3_mod, only: rvec3
  implicit none
  integer :: i
  real(rk) :: c
  type(rvec3) :: a, b !... Note type of vectors

  ! input vectors
  print*, 'Enter first vector'
  read*, a%x
  !... Note derived type notation
  print*, 'Enter second vector'
  read*, b%x

  ! calculate dot product (check and print left our here!)
  call dp(a%x, b%x, c)
end program dotprod

! module containing derived type for 3-component real vector
module rvec3_mod
  implicit none
  integer :: nvals = 3
  real(rk), dimension(3) :: x
  type rvec3
  end module rvec3_mod
```
Solution Exercise 12

### compiler
F90 = gfortran

### modules, objects and executable
OBJ = prec_mod.o rvec3_mod.o dotprod.o dp.o
EXE = dotprod

%.o: %.f90
 $(F90) -c $< -o $@

### compile and link
$(EXE): $(OBJ)
 $(F90) $(OBJ) -o $@

clean:
 rm -f *.o *.mod $(EXE)

---

I/O (1)

- List-directed output, print* or write(*,*), gives little control
- Use formatted output
  
  write(unit, format) variables
  
- Unit is a number indicating where you want to write data
  - The number 6 is std out (write to screen)
I/O (2)

* im
  - For integers
  - m is total number of places in field
  - i3 125

* am
  - For character strings
  - m is number of characters
  - a5 hello
  - Left-justifies
  - If m isn’t specified, writes number of characters in variable declaration

I/O (3)

* fm.n
  - For floating-point (real) numbers
  - m is total number of characters in field
  - n is number of decimal places
  - f5.3 1.234
  - f5.2 -1.23
  - If m is larger than required, right-justifies

* em.n
  - Exponential notation
  - e9.2 -0.23e-01
  - Always zero left of decimal
I/O (4)

• es\text{n}
  • scientific notation
  • es9.2 -2.30e-02

• In format statement, put formats within ‘()’
• Example write statement
  write(6, '(a, f6.2, i5, es15.3)') 'answers are ', x, j, y

You can check a full list of formats here:
https://www.cs.mtu.edu/~shene/COURSES/cs201/NOTES/chap05/format.html

I/O (5)

• Suppose you want to write to a file
  • open statement
    open(11, file='mydata.dat')

  • “11” is unit number
  • Don’t use 5 or 6
    • Reserved for std in, std out
  • Use this unit in your write statement
    write(11, *) 123

  • When you’re finished writing, close the file
    close(11)
I/O (6)

- Can also read from file
- `read` rather than write
- Can use * instead of format specifier

```fortran
read(11,*) j, x
```

Exercise 13

- Write your dot-product result to a file
- Only have to change very end of main program
Solution Exercise 13

```fortran
program dotprod

! input vectors
print*, 'Enter first vector'
read*, a
print*, 'Enter second vector'
read*, b

! calculate dot product
call dp(a, b, c)

! check magnitude left out here

! write result to file
open(199, file='dot.d', form='unformatted')
write(199) c
close(199)
pnint*, '
print*, 'Output written to file dot.d'
end program dotprod
```

Unformatted I/O

- Binary data take less disk space than ascii (formatted) data
- Data can be written to file in binary representation
  - Not directly human-readable
    ```fortran
    open(199, file='unf.d', form='unformatted')
    write(199) x(1:100000), j1, j2
    read(199) x(1:100000), j1, j2
    ```
  - Note that there is no format specification
- Fortran unformatted slightly different format than C binary
- Fortran unformatted contains record delimiters
Exercise 14

- Modify dot-product program to:
  - Write result to unformatted file
  - don’t write character string, just number
- After file is closed, open it back up and read result
- Print result to make sure it wrote/read correctly

Solution Exercise 14

```
! dotprod.f90
! calculate dot product and write result to
! an unformatted file, read result to check code
!
program dotprod

!------------------------------------------------------------------
! module containing precision parameter
!------------------------------------------------------------------
use prec_mod, only: rk

!------------------------------------------------------------------
! module containing derived type
!------------------------------------------------------------------
use rvecV_mod, only: rvecV
implicit none
integer :: i
real(rk) :: c, qc
type(rvecV) :: a, b

!------------------------------------------------------------------
! set nvals for each vector
!------------------------------------------------------------------
a%nvals = 3
b%nvals = 3

!------------------------------------------------------------------
! input vectors and check like in the exercises before!
!------------------------------------------------------------------
call dp9(x, b(3, x, c)

!------------------------------------------------------------------
! write result to file
!------------------------------------------------------------------
open(21, file = 'dot.d', form = 'unformatted')
write(21) c
close(21)
print*,'Output written to file dot.d'

!------------------------------------------------------------------
! read result from file and print value
!------------------------------------------------------------------
open(21, file='dot.d', form = 'unformatted')
read(21) c
print*,'Value read from file = ', c
end program dotprod
```
Integration Example

- Integration of cosine from 0 to \(\pi/2\) with mid-point rule
- Integral \(\approx\) sum of rectangles (height \(*\) width)

\[
\int_0^{\pi/2} \cos(x) \, dx \approx \sum_{i=1}^{m} \cos(x) \, dx \approx \sum_{i=1}^{m} \cos(a + (i - \frac{1}{2})h)h
\]

Exercise 15

- Write a program to perform integration of cosine using the mid-point rule
- Write a function `integral` to perform integration
  - \(m, a, h\) are input to `integral`
- The main program calls `integral` a few times (using do loop), each time with a larger \(m\) than the previous time. The purpose is to study the convergence trend. (hint: \(m=25*2^n\); \(n\) is the loop index)
Solution Exercise 15

```fortran
program integration_with_doloop
!
! This program computes the numerical integration of cosine function
!
! Written by: Kadin Tseng
! Date written: September 19, 2012
!
implicit none
real, parameter :: pi=3.141593
real :: a, b, h, integ, integral
integer :: n, m
a = 0.0 ! lower limit of integration
b = pi/2 ! upper limit of integration
do n=1,4 ! number of cases to study
  m = 25*2**(n-1) ! define number of increments as a function of n
  h = (b - a)/m ! increment length
  integ = integral(a, h, m) ! perform integration
  print*, 'No. of increments = ', m, 'Integral value is ', integ
endo
end program integration_with_doloop

real function integral(a, h, m)
!
! performs midpoint integration with forloop
 implicit none
real :: a, h, x
integer :: m, i
integral = 0.0 ! initialize integral
do i=1,m
  x = a+(i-0.5)*h ! mid-point of increment i
  integral = integral + cos(x)*h
endoeturn
end
```

References

- Lots of books available
  - “Fortran 95/2003 Explained” by Metcalf, Reid, and Cohen is good
- Gfortran
- Fortran wiki
  - http://fortranwiki.org/fortran/show/HomePage
- Use google