Climatescience.org.au

Fortran Introduction

Holger Wolff

26/10/2015

Background

In this opening section, I will talk briefly about

- The History of Fortran
- Fortran 77 vs. Fortran 90
- Example programs

History of Fortran

- **For**mula **Tran**slation
- Around since 1952
- Good for number crunching
- Bad for text manipulation
- Procedural Programming Paradigm
- Versions usually denoted by year of publication, eg. 66, 77, 90, 95, 2003
- This course deals with versions 77 to 95.

Example Fortran77 Code

```
1       PROGRAM HELLO
2       IMPLICIT NONE
3       INTEGER i
4 C     This is a comment
5       DO 100 i = 1, 10
6       PRINT *, "I am in iteration ", i,
7      & " of the loop."
8  100  CONTINUE
9       END PROGRAM
```

Fortran 77

Fixed Form:

- The position in the line of a character has meaning.
- Anything in first position (other than digit): Line is Comment.
- A number in postions 1-5: Label that can be referenced somewhere else in the code
- Anything in position 6: Continuation line, this code continues from last line
- Code between position 7 and 72 (inclusive)

Example Fortran90 Code

```
1  program hello
2      implicit none
3      integer :: i   ! This is a comment
4      do i = 1, 10
5          print *, "I am in iteration ", i, &
6                   " of the loop."
7      end do
8  end program hello
```

Fortran 90

Free Form: Position in the line no longer relevant. Some added features:

- DO - END DO
- Modules
- Comments now denoted with Exclamation Mark
- But still mostly backwards compatible.

Fortran 95

Fortran 95 removed items from the syntax that have been deprecated. Examples:

- DO loops with floating point iterators
- Calculating Jump destinations

But also new features were introduced, for example:

- Implicit Loops with FORALL and WHERE
- PURE, ELEMENTAL, and RECURSIVE procedures

Newer Fortran Versions

- Fortran 2003
  - Some Object Oriented Features
  - Input/Output Streams
- Fortran 2008
  - Easier parallelisation with DO CONCURRENT
  - Submodules
- Fortran 2015
  - ???

Creating an Executable

In this brief section, I will talk about

- How to write code
- How to compile code
- Compiler Warnings
- Debug Settings

Writing Source Code

Source Code is written with a Text Editor.
Not to be confused with Word Processors. Basically: If you can easily change the Font, you can't use it to write code.
Examples for good text editors:

- Windows: Notepad++
- MacOS: TextWrangler
- Linux GUI: gedit
- Linux console: vim, emacs, nano

Compiling Simple

Let's consider this very simple "Hello World" program:

```
1 program hello
2     implicit none
3     print *, "Hello World"
4 end program hello
```

Such a program can be compiled in a single step:

```
$ gfortran -o hello hello.f90
```

Compiling Multiple Files

If your code spans several files, you have to independently compile each into object files. Often you need to do that in a certain order. Then you link the object files together into the executable.

```
$ gfortran -c -o mod_hello.o mod_hello.f90
$ gfortran -c -o hello.o hello.f90
$ gfortran -o hello hello.o mod_hello.o
```

Note the -c flag in the first two lines.

Compiler Warnings

Sometimes your code is syntactically correct, but doesn't do what you think it does. There are some warning signs that the compiler can pick up, such as Variables that are declared but never used, or blocks of code that are never reached.
It is good programming practise to always enable all warnings, and then eliminate the causes for these warnings.

- **Intel Fortran:** use option `-warn all`
- **GNU Fortran:** use option `-Wall`

Compiler Debugging Settings

Some compiler options reduce performance, but provide useful feedback during programming and debugging.

	Intel	**GNU**
Traceback	`-traceback`	`-fbacktrace`
Checking	`-check all`	`-fbounds-check`

Once you're done debugging, compile it without these options.

Variables

In this section, I will introduce

- The basic program structure
- The different variable types
- Kinds
- Custom Types
- Arrays

Basic Program Structure

```
1 program <name>
2   implicit none
3   <variable declaration>
4   <executable statements>
5 end program <name>
```

Variable Types

Fortran	Other	Examples
`LOGICAL`	bool(ean)	`.TRUE.` or `.FALSE.`
`INTEGER`	int	`0`, `10`, `-20`
`REAL`	float	`0.0`, `1.`, `.5`, `1.4e3`
`COMPLEX`		`(0.0, 1.0)`
`CHARACTER`	char, String	`"A"`, `"Hello World"`

Declaration

```
1 INTEGER :: int_var
2 REAL :: real_var1, real_var2
3 REAL :: real_var3
```

Double colon is optional, but preferred.

Parameters

Parameters, called constants in many other languages, are like variables, except their value cannot ever change.

```
1 REAL, PARAMETER :: pi = 3.141592
2 REAL :: e
3 PARAMETER(e = 2.71)
```

KIND

Variable KINDS are flavours of variables that change the precision and range of values that can be stored in them.
Usually, the KIND is the number of bytes required to store the data, though not all numbers are allowed.

```
1 REAL*8 :: var1
2 REAL(KIND=8) :: var2
3
4 INTEGER, PARAMETER :: dp = selected_real_kind(P = 15)
5 REAL(KIND=dp) :: var3
```

Character

Fortran can only understand fixed length Strings.

```
1  CHARACTER*20 :: A
2  CHARACTER(LEN=10) :: B
3
4  A = "Hello World" ! Actually stores "Hello World     "
5  B = A             ! Actually stores "Hello Worl"
```

Working with Characters

```
1 print *, String1 // String2
2 print *, trim(String1) // " " // trim(String2)
3 print *, "Length of String1 is ", len(String1)
4 print *, "Length of String2 w/o trailing spaces is ", &
5          len_trim(String1)
```

Types

You can create your own Variable types.

```
1  TYPE :: my_type
2    INTEGER :: my_int
3    REAL :: my_real
4  END TYPE my_type
5
6  TYPE(my_type) :: t
7
8  t % my_int = 5
9  t % my_real = 1.2
```

Arrays

Several values of the same type can be taken together in Arrays.

```
1 INTEGER :: a(10)
2 INTEGER, DIMENSION(10) :: b
3 INTEGER, DIMENSION(1:10) :: c
```

All three lines above do the same thing: They declare an array of 10 integers, with indices running from 1 to 10.
To access certain values inside the array, you write the index in parentheses after the array name:

```
print *, "The 5th element of a is ", a(5)
```

Arrays (continued)

```
1 INTEGER, DIMENSION(-5:5) :: a
2 INTEGER, DIMENSION(10, 20) :: b
3 INTEGER, DIMENSION(:), ALLOCATABLE :: c
```

Line 1 shows how to create an array with indices that run from -5 to 5.
Line 2 creates a two-dimensional array with 10 by 20 elements.
Line 3 creates an allocatable array. In this case, it has one dimension, but at compile time we don't yet know how big it needs to be. Before c can be used, it first has to be allocated, like so:

```
allocate(c(100))
deallocate(c)
```

Basic Statements

In this section, I will explain

- How to assign values to variables
- Common sources of errors
- Loops
- Conditional Code

Basic Assignment

```
1  a = 3 + 4 * 2
2  b = (3 + 4) * 2
3  c = sqrt(2.0)
4  i = i + 1
```

Single equals sign is assignment operator.
Right side is evaluated, assigned to variable on left.
Line 4 is a common line that increments the value of `i` by one.

Assignment Gotcha

```
1  REAL :: a
2
3  a = 3 / 4
4  print *, a ! prints 0.0
```

3/4 is evaluated using integer division, ignoring the remainder. Then 0 is assigned to a, changed into a `REAL` type.

Solution

Enforce the division to be a floating point division by turning at least one of numerator and denominator into `REAL`:

```
1  a = 3.0 / 4
2  a = float(3) / 4
3  a = 3 / (4 * 1.0)
```

What doesn't work is converting the result of the division to `REAL`:

```
a = float(3 / 4) ! Doesn't work
```

Loops

Loops repeat the same code several times, often have an iterator that changes its value every iteration.

```
1  ! Old Style
2  do 100 i = 1, 10
3    print *, i
4  100 continue
5
6  ! New Style
7  do i = 1, 10
8    print *, i
9  end do
```

These loops print the numbers from 1 to 10.

Stride

Iterators don't have to increment by one every iteration.

```
1  do i = 5, 20, 3
2  do i = 20, 10, -1
```

The third number is the stride, the amount by which the iterator changes in every iteration of the loop.

DO WHILE

You can also repeat a loop while a condition is still valid:

```
1  do while(j > 10)
2    j = j - 2
3  end do
```

Ensure that the loop will terminate eventually, or your program will hang.

EXIT

The `exit` statement interrupts the loop and moves straight to the first statement after the loop:

```
1  do i = 1, 10
2    if (i > 5) exit
3  end do
```

CYCLE

The `cycle` statement interrupts the current iteration of the loop and immediately starts with the next one.

```
1  do i = 1, 100
2    if (mod(i, 3) == 0) cycle ! Don't print multiples of 3
3    print *, i
4  end do
```

Named Loops

You can name loops. In this case, the `end do` statement has to repeat the name of the loop.

```
1  print_loop : do i = 1, 10
2    print *, i
3  end do print_loop
```

This helps with readability when you have long and/or nested loops.
This also helps you to clear which loop to `EXIT` or `CYCLE`.

Nested Loops and Multi-Dimensional Arrays

Often nested loops are used to loop over all the indices of multi-dimensional arrays.

```
1 do k = 1, 100
2     do j = 1, 100
3         do i = 1, 100
4             a(i, j, k) = 3.0 * b(i, j, k)
5         end do
6     end do
7 end do
```

For performance reasons, it is always best to have the **innermost** loop over the **first** index, and so on out.
This has to do with the way Fortran stores multi-dimensional arrays.

Conditional

```
if (<condition>) <statement>
```

The `if` statement is used for conditional execution. Example:

```
if (i < 5) print *, "i is less than 5"
```

`<condition>` must evaluate to `.TRUE.` or `.FALSE.`, and only in the former case is the `<statement>` executed.

IF Block

```
if (<condition>) then
  <statement 1>
  <statement 2>
end if
```

When the execution of several statements must be dependend on a condition, the keyword `THEN` is added after the condition.
In that case, all statements until the `END IF` statement are executed.

IF ELSE Block

```
if (<condition>) then
  <true statement>
else
  <false statement>
end if
```

An `IF` block can be extended with an `ELSE` block that gets executed if and only if the initial condition evaluated to `.FALSE.`.

ELSEIF Block

```
if (<condition 1>) then
  <statement 1>
elseif (<condition 2>) then
  <statement 2>
else
  <statement 3>
end if
```

If the initial condition is `.FALSE.` we can check for other conditions using the `ELSEIF` clause.
Note here that when `condition 1` is `.TRUE.`, only `statement 1` is executed and `condition 2` never even checked.
There is no limit on the number of `ELSEIF` clauses.

Conditionals (cont'd)

At the end, `<condition>` has to be of type `LOGICAL`.
But they can be combined (`A` and `B` are of type `LOGICAL`):

Code	Explanation
`.not. A`	inverts A
`A .or. B`	`.TRUE.` if at least one of `A` or `B` is.
`A .and. B`	`.TRUE.` if both `A` and `B` are.

Conditionals (cont'd #2)

Variables of all types can be compared to get `LOGICAL`:

F77 Code	F90 Code	`.TRUE.` if
`A .lt. B`	`A < B`	`A` is less than `B`
`A .gt. B`	`A > B`	`A` is greater than `B`
`A .le. B`	`A <= B`	`A` is less than or equal to `B`
`A .eq. B`	`A == B`	`A` is equal to `B`
`A .neq. B`	`A <> B`	`A` is not equal to `B`
`A .ge. B`	`A >= B`	`A` is greater than or equal to `B`

```
1  if ( (i == 5) .and. (.not. (mod(j, 5) == 0))) then
2    print *, "i is 5 and j is not a multiple of 5"
3  end if
```

Floating Point Warning

Floating point numbers have limited precision. This leads to rounding errors. In the familiar decimal system, we see this for example here:
$1 = 1/3 + 1/3 + 1/3 = 0.3333 + 0.3333 + 0.3333 = 0.9999$
Computers use binary, so their rounding errors might come as a surprise:

```
if (.not. ((0.1 + 0.2) == 0.3)) print *, "Rounding
    Error!"
```

CASE

Commonly, one wants to execute different statements depending on a single variable. This could be done with lots of `ELSEIF` statements, but there is a more conveniant way:

```
1  select case (var)
2    case (-10)
3      <what to do if var is equal to -10>
4    case (10:)
5      <what to do if var is more than or equal to 10>
6    case default
7      <what to do if none ot the other cases matched>
8  end select
```

Procedures

In this section, I will talk about the two types of procedures that Fortan offers:

1. Subroutines
2. Functions

I will talk about how to write and call them.

Motivation for Procedures

Procedures help by

- Separating Code into smaller, easier to understand chunks.
- Improving Code Re-Usability.
- Allowing for independent testing of Code

Subroutines

Subroutines look very similar to a program:

```
1 subroutine <routine name> (<arguments>)
2     implicit none
3     <declarations>
4     <executable code>
5 end subroutine <routine name>
```

New are the `arguments`. This is a list of variables that the caller has to pass to the subroutine in order for the subroutine to work correctly.

Subroutine Example

```
1 subroutine greet (cname)
2     implicit none
3     character(len=*) :: cname
4     print *, "Hello " // trim(cname)
5 end subroutine greet
```

Here, the name of the subroutine is `greet`, and it takes one argument: `cname`, which is declared in line 3 as being a character with unknown length.
This is one of the few times where Fortran doesn't need to know the length of the String. `(len=*)` means that the routine should just take the string length as is.

Calling a Subroutine

```
call greet("World")

call greet(cname = "Tony")

my_name = "Holger"
call greet(my_name)
```

These are three ways how to call the subroutine.

Intent

A procedure can have any number of arguments, separated by commata.
It is good programming practise to give all arguments an `INTENT`:

```
1  subroutine my_routine (a, b, c)
2      implicit none
3      integer, intent(in) :: a
4      integer, intent(out) :: b
5      integer, intent(inout) :: c
6
7      b = a + c
8      c = 2 * a
9  end subroutine my_routine
```

Intent (cont'd)

INTENT	Explanation
`IN`	Procedure will not change the value of this variable
`OUT`	Procedure will set this to a new value
`INOUT`	Procedure might read this value and might set it to a new one

This helps the compiler to spot bugs and to improve performance.

Functions

Functions are similar to subroutines, except that they return a value themselves.

```
1  function my_abs(n)
2      implicit none
3      integer, intent(in) :: n
4      integer :: my_abs ! Declare the type of the function
5      if (n >= 0) then
6          my_abs = n    ! Use name of function as variable
7      else
8          my_abs = -n
9      end if
10 end function my_abs
```

Calling a function

Since functions return a value, when called, that value has to be placed somewhere.

```
b = my_abs(a)
print *, "The absolute value of -10 is ", my_abs(-10)
```

RETURN

Sometimes, you want to return from a procedure early.

```
1 subroutine report_error(error_code)
2     ! Prints the error code to screen
3     ! only if the error code is not 0
4     implicit none
5     integer, intent(in) :: error_code
6     if (error_code == 0) return
7     print *, "Encountered error: ", error_code
8     print *, "Please check program"
9 end subroutine report_error
```

In the case that the error code is 0, line 6 will return early and not print the error messages.

Independend Procedures

Procedures can be placed outside of the program code, in a different file altogether.

```
1 subroutine hello()
2     implicit none
3     print *, "Hello World!"
4 end subroutine hello
```

```
1 program main
2     implicit none
3     call hello()
4 end program main
```

This is inferior to other methods.

CONTAINS

Programs can contain subroutines. In this case, the subroutine has full access to the programs variables at the time it is called.

```
1 program main
2     implicit none
3     call greet()
4 contains
5     subroutine greet()
6         implicit none
7         print *, "Hello World"
8     end subroutine greet
9 end program main
```

MODULE

Procedures can also be placed in modules. Modules are very similar to a program, in that they (can) have variables, types, and can contain procedures.

```
1 module <module_name>
2     implicit none
3     <declaration of variables>
4 contains
5     <procedures>
6 end module <module_name>
```

What they don't have is executable code outside of the contained procedures.

MODULE Example

Here's an example for a module for a greeter.

```
1  module greet_mod
2      implicit none
3      character(len=20) :: greeting = "Hello"
4  contains
5      subroutine greet(cname)
6          implicit none
7          character(len=*), intent(in) :: cname
8          print *, trim(greeting) // " " // trim(cname)
9      end subroutine greet
10 end module greet_mod
```

Note that the subroutine has access to the module variable `greeting`

USE of Modules

To use a module, either in a program or another procedure, the `USE` keyword is used **before** the `IMPLICIT NONE`.

```
1 program hello_world
2     use greet_mod
3     implicit none
4
5     greeting="G'Day"
6     call greet("World")
7 end program hello_world
```

Note that the program doesn't have to declare the `greeting`, and can still change its value.

Output and Input

This section deals with the Input and Output of Data.

- Writing to Standard Output
- Reading from Standard Input
- Formatting Input and Output
- Writing to and reading from files
- Namelists

PRINT

The first way to output data has been featuring prominently on most slides:

```
print *, <data>
```

This statement dumps the data to the standard output. The formatting isn't nice, but it's really easy.

WRITE

To get more control over the output, the `WRITE` statement is used.

```
write(*, *) <data>
```

This line is actually equivalent to the `print` statement.
But note the two asterisks:
The first one declares the destination of the write statement (* is standard output), the second one the format (* is standard format).

Format

The format can be given in three ways:
Directly as a string:

```
1 write(*, '(A6, I4)') "Year: ", 2015
```

Placed in a string variable:

```
1 fmt = "(A6, I4)"
2 write(*, fmt) "Year: ", 2015
```

Or by giving the label of a `FORMAT` statement:

```
1 write(*, 1001) "Year: ", 2015
2 1001 format(A6, I4)
```

Format Examples

ID	Explanation
`In.w`	Integer, n is total length, w is min length padded with 0
`Fn.w`	Float, n is total length, w is number of digits after the comma
`En.w`	Float in notation 2.41E4, n is total length, w is number of digits after the comma
`An`	String, n is length

There are lots more, google "Fortran Format" for complete list.

READ

The converse of the `WRITE` statement is the `READ` statement. It follows the same rules and uses the same formatting syntax.

```
read(*, '(I4, F8.2)') n, r
```

This command reads from standard input (keyboard) first a 4 digit integer, then a floating point number in the format xxxxx.xx

Writing to CHARACTER

The first argument of the `WRITE` and `READ` statements is the destination/source. The asterisk refers to standard input/output. But we can also direct it to something else.
For example a character variable:

```
1 character(len=10) :: t
2 write(t, '(A5, I5.5)') "file_", 4
```

The above snippet would store the text "file_00004" in the variable `t`.

OPEN

To write to (or read from) a file, the file needs to be opened and closed.

```
1 integer, parameter :: my_unit = 300 ! Some number > 10
2 integer :: io_err
3
4 open(unit=my_unit, file="file.txt", action="WRITE", &
5      status="REPLACE", iostat=io_err)
6 write(my_unit, '(A)') "This text is written to the file"
7 close(my_unit)
```

There are a lot more optional arguments to `OPEN`, google "Fortran OPEN" for a complete list.

NAMELIST

A NAMELIST is a collection of variables that can be read from and written to a file very easily.
First, the variables have to be declared and declared to be part of a namelist:

```
1  integer :: a
2  real :: r
3  real, dimension(10) :: t
4
5  namelist /NLIST/ a, r, t
6
7  open(unit=my_unit, file='file.txt', action='READ')
8  read(my_unit, nml=NLIST)
9  close(my_unit)
```

NAMELIST (cont'd)

In order to be read by the code on the previous page, the file "file.txt" must obey a certain format:

```
1 &NLIST
2   a = 6,
3   r = 4.51,
4   t = 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0
5 /
```

A namelist like this can also be written by a `WRITE` statement.

Deprecated Features

This section explains a few features that are deprecated.
You usually wouldn't want to use any of these features yourself, however you might come across them when looking at old code.

CONTINUE

The `CONTINUE` statement does nothing at all.
It was usually used in conjunction with labels to remove any ambiguity as to whether the line with the label was to be executed or not.

COMMON

Common blocks were used to share variables between different procedures without having to pass them through as arguments.
It was important that all procedures that share these variables use the exact same variable names, and the same `COMMON` statement.
Use `MODULES` instead.

EQUIVALENCE

`EQUIVALENCE` means that different variables should literally share the same memory location.
This was originally necessary to reuse memory on computers that had kilobytes at best.
Modern computers have more than enough memory, so this need is no longer there.
Some people claim that they've found other uses for `EQUIVALENCE` but you really shouldn't bother with that.

DATA

`DATA` was used to initialise variables, that is to give them their initial values.
Just assign the values at the beginning of your execution block.

GOTO

`GOTO` is used with a label. Execution of the program jumps directly to the line with the label.

```
1 print *, "This line is executed."
2 goto 1001
3 print *, "This line is not."
4 1001 CONTINUE
5 print *, "This line is executed again."
```

`GOTO` can jump both forwards and backwards. It interrupts the code flow and is therefore despised by many programmers.
Using loops and `IF` blocks can often achieve the same thing in a more elegant way.
That said: "Real Programmers Aren't Afraid of GOTO."

Coding Standards

Most large projects have so-called 'Coding Standards'.
These Standards explain how code on this project should be written. Amongst the things typically included in Coding Standards are:

- Naming Conventions for Variables and Procedures
- Indentation Guidelines
- Comments and Documentation Guidelines

If you want to contribute to a major project, you need to follow these Coding Standards if you want to contribute.

Thanks

And questions?