



Fortran basics

A short introduction to Fortran and how it is used in FLUKA

Outline

- Introduction to Fortran
 - Differences between Fortran 77 and 90+
 - Fortran in the user routines of FLUKA
- Fortran quick start guide
 - Variables, arrays and strings
 - Logical operators
 - Conditional (**if**) and loop (**do**) constructs
 - Procedures
 - File operations

History of Fortran

Fortran born in the early 1950s, and the first compiler was released in 1957

Standards:

- Fortran 66 – The first standard
 - Fortran 77 – Extension on Fortran 66

 - Fortran 90 – Dynamic memory allocation
 - Fortran 95 – High performance Fortran specification
- } Introduction of the *Free* format
-
- Fortran 2003 – Object oriented programming
 - Fortran 2008 / 2018 – Extensions of Fortran 2003
- } “Modern” Fortran

File format

- Fortran 77 uses the *Fixed* file format (extensions: **.f** or **.for**):
 - Maximum 72 characters in one line
 - First 6 are reserved for special function:
 - If the first character is 'c', 'C' or '*', then the line is a comment
 - The 1st – 5th characters can be used for statement labels
 - If the 6th position is not empty, then the line is treated as a continuation of the previous one (Often the '&' character is used)

```
*...5...0...5...0...5  
  program hello  
c This is a comment  
  print *, 'Hello,  
& World!'  
  end program hello
```

File format

- Fortran 90 introduced the *Free* format (extensions: **.f90**, [**.f95**, etc.]):
 - Code can start at the 1st column
 - Inline comments with ‘!’
 - Continuation lines

```
program hello
  print *, 'Hello, &
           & World!' ! This is a comment
end program hello
```

- *Note:* It is not possible to mix both in single source file.
The compiler expects the “correct” format based on the file extension.

Variable and procedure names

- Fortran 77:
 - Limited to 6 alphanumerical characters
 - Have to start with a letter
 - Case insensitive
- Starting with Fortran 90:
 - Can be up to 31 character long
 - Can contain letters, numbers and underscore ('_')
 - Have to start with a letter
 - Case insensitive
- *Note:* Try to use descriptive names, to make code readable

Variable declaration

- Fortran by default uses *implicit declaration*, which means the type of the variable (**integer**, **real**, etc.) is determined by a preset rule.
- The default rule is:
 - If the variable starts with the letter 'I', 'J', 'K', 'L', 'M', or 'N' it is an **integer**
 - Otherwise, it is a **real** (single precision float)
- It is possible (and necessary) to overwrite this with *explicit declaration*, where you manually specify another variable type, like:

```
double precision :: my_number  
logical :: my_flag
```

Issues with implicit declaration

- Typos remain hidden

If you have a typo in a variable name, the compiler won't raise an error

It is a different, but valid variable usually without a value

Using it in calculations will lead to unexpected results

- Unexpected type conversion

For example: Information is lost if you want to assign a **double precision** number to **integer** variable

- Solution

Force explicit declaration with the statement:

```
implicit none
```


Comparison of Fortran 77 and 90+

	Fortran 77		Fortran 90+
Format	Fixed (.f, .for)	→	Free (.f90, .f95, ...)
Maximum line length	72		132
Variable name max. length	6		31
Variable declaration (usually)	implicit	→ *	forced explicit

- FLUKA user routines are somewhere in-between
 - ★ Implicit declaration using **double precision** numbers instead of **reals**
- Modernization effort for a future release
 - ★ A new version of the source routine is already available (fixed format, forced explicit declaration)

Outline

- Introduction to Fortran
 - Differences between Fortran 77 and 90+
 - Fortran in the user routines of FLUKA
- Fortran quick start guide
 - Variables, arrays and strings
 - Logical operators
 - Conditional (**if**) and loop (**do**) constructs
 - Procedures
 - File operations

Variables

- Declaration:

```
integer :: amount, counter  
real    :: pi, sqrt_two  
double precision :: energy  
complex :: frequency  
character :: initial  
logical :: okay
```

- Assignment:

```
amount = 10  
pi = 0.3141592e1  
energy = 1.0d-3  
frequency = (1.0, -0.5)  
initial = 'F' ! Or "F"  
okay = .true. ! Or .false.
```

Arrays and strings

- Arrays:

```
! 1D integer array  
integer, dimension(10) :: array1  
  
! An equivalent array declaration  
integer :: array2(10)  
  
! 2D real array  
real, dimension(10, 10) :: array3  
  
! Custom lower and upper  
! index bounds  
real :: array4(0:9)  
real :: array5(-5:5)
```

- Strings:

```
character(len=10) :: string1  
  
! Or  
character(10) :: string2  
  
string2 = 'FLUKA'
```

Note: Strings are padded with “space” to the specified length, i.e. 'FLUKA '.

To omit the padding use the `trim()` function

Logical operators

- Relational operators:

Equal:

`a .eq. b` `a == b`

Not equal:

`a .ne. b` `a /= b`

Greater than:

`a .gt. b` `a > b`

Less than:

`a .lt. b` `a < b`

Greater than or equal:

`a .ge. b` `a >= b`

Less than or equal:

`a .le. b` `a <= b`

- Logical operators:

`.true.` if both operands are `.true.`:

`a .and. b`

`.true.` if one of operands is `.true.`:

`a .or. b`

`.true.` if the operand is `.false.`:

`.not. a`

`.true.` if the operands are the same:

`a .eqv. b`

`.true.` if the operands are the opposite:

`a .neqv. b`

Conditional (**if**) and loop (**do**) constructs

- Conditional (**if**) construct:

```
if (angle < 90.0) then
  print *, 'Angle is acute'
else if (angle > 180.0) then
  print *, 'Angle is reflex'
else
  print *, 'Angle is obtuse'
end if
```

- Conditional loop (**do while**):

```
i = 1
do while (i < 11)
  print *, i
  i = i + 1
end do
```

- Loop (**do**) construct:

```
integer :: i

do i = 1, 10
  print *, i
end do
```

- Loop with skip:

```
do i = 1, 10, 2
  ! Print only odd numbers
  print *, i
end do
```

Procedures

- Functions:

Invoked within an expression or assignment
Returns a value

```
integer function cube(i)
  integer :: i

  cube = i**3
end function cube
```

```
program main
  integer :: cube
  integer :: i, j

  i = 3
  j = cube(i)
end program main
```

- Subroutines:

Invoked by a **call** statement
No return value

```
subroutine print_mx(n, m, A)
  integer :: n, m
  integer :: i
  real :: A(n, m)

  do i = 1, n
    print *, A(i, 1:m)
  end do
end subroutine print_mx
```

```
real :: mat(3, 4)
...
call print_mx(3, 4, mat)
```

Passing arguments to procedures

- Many programming languages by default only pass the values of the arguments to the procedures.
Meaning, changing the value in the procedure doesn't have any effect on the value of the original argument.
- However in Fortran, the arguments by themselves are passed to the procedures. This means, the changes made to the values of the arguments will remain after the procedure completes.
- Useful when more than one value must be returned.
- *Safe practice:* Only use functions which don't change the arguments. Otherwise use subroutines.

Save statement

- Variables declared with the **save** statement retain their value between calls to procedures

```
integer, save :: amount  
real, dimension(10), save :: array
```

- This allows to create sections of code which only executed at the first call

```
logical, save :: lfirst = .true.  
integer, save :: reg_number  
integer :: ierr  
  
if (lfirst) then  
    call geon2r('TARGET ', reg_number, ierr)  
    lfirst = .false.  
end if
```

Opening files

- To open a file in Fortran:


```
open(unit=<unit>, file='<filename>', status='<status>', form='<form>')
```

Unit number: used to reference the file in the read/write comments

- Some units numbers are predefined
 - *FLUKA specific*: Unit numbers ≤ 20 and the ones in scorings can't be used
- FLUKA subroutine: Looks for the file in multiple directories

```
call oauxfi('<filename>', <unit>, '<form_and_status>', <ierr>)
```

- FLUKA **OPEN** card:

 OPEN	Unit: 21 ASC ▼	Status: OLD ▼
	File: input.dat ▼	

Input from files

- Reading from a file:

```
read(<source>, 'format') a, b, ...
```

Source: Unit number or a string

Format: Use the default *. Fortran will try to figure it out based on the type of the variables

```
real, dimension(20) :: a, b
integer :: i

open(unit=21, file='input.dat', status='old', form='formatted')

do i = 1, 20
    read(21, *) a(i), b(i)
end do
```

Output to files

- Writing to a file:

```
write (<target>, 'format') a, b, ...
```

Target: Unit number or string

Format: The default is * for automatic formatting

```
integer :: i

open(unit=22, file='output.txt', status='new', form='formatted')

do i = 1, 10
    write(22, *) i, cube(i)
end do
```

Predefined units for writing to the FLUKA output files:

.out file:

```
write(lunout, *) a, ...
```

.err file:

```
write(lunerr, *) a, ...
```

.log file:

```
write(*, *) a, ...
```

I/O formatting

- The format string lists the format specifiers for the printed variables and it is enclosed in round brackets:

```
`(A10, 5X, I4, /, F8.3, E15.7)`
```

- Integer:

```
`(Iw)`
```

w characters long

- Real:

```
`(Fw.d)`
```

w characters long,
fractional part **d** characters

```
`(Ew.d)`
```

Exponential form, **w** characters long,
fractional part **d** characters

- String:

```
`(Aw)`
```

w characters long

- Blank space:

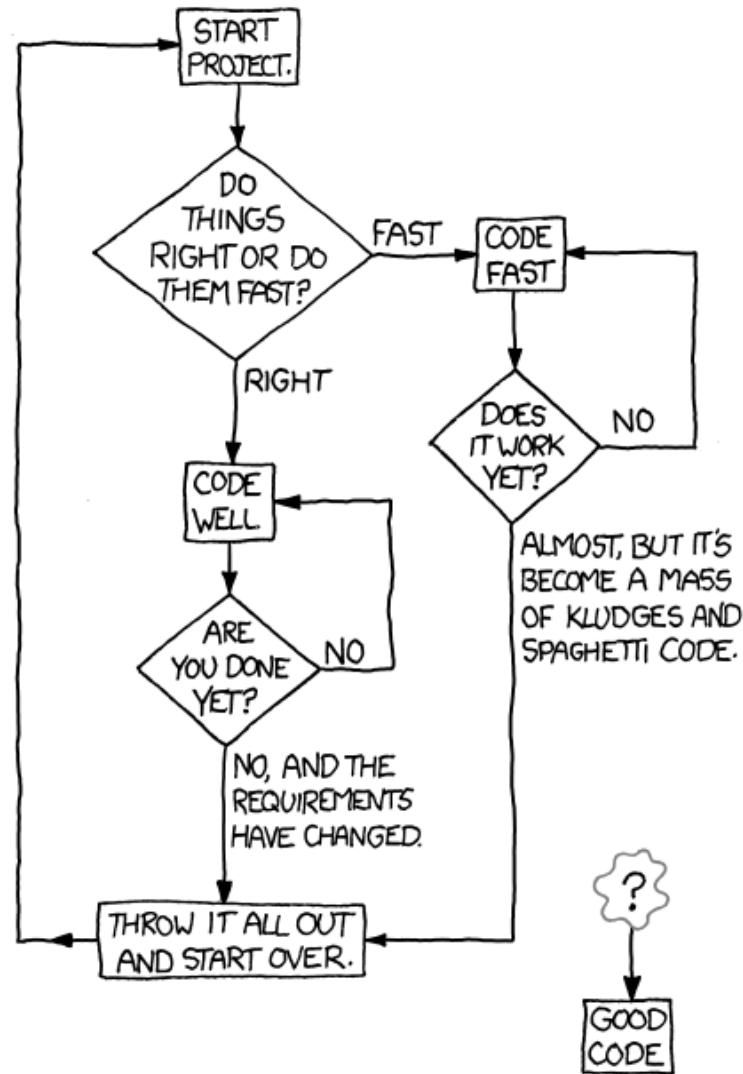
```
`(nX)`
```

n characters long

- New line:

```
`(/)`
```

HOW TO WRITE GOOD CODE:



xkcd.com/844

