Introduction to Fortran

Drew Schmidt

January 16, 2014

Contents

Contents

What is Fortran?

- **For**mula **tran**slation system.
- General purpose programming language.
- Well-suited for mathematics and engineering.
- Compiled (rather than interpreted)
- Portable.

History

- Created at IBM in 1954 as an alternative to assembly language.
- Debuted in the days of punch cards.
- Regularly gets updated to a new standard.

What people think Fortran is like

```
1121  FORMAT(I4,F8.3)
3298  CONTINUE
      IF(MOD(I,A)
     $.EQ.Z) THEN
      GOTO 2359
      ELSE IF(MOD(I,B)
     $.EQ.Z) THEN
      GOTO 8125
      ELSE
      WRITE (*,2930) I
      GOTO 7365
      END IF
7235  FORMAT(A,F5.3)
7356  CONTINUE
      I=I
     $+1
      GOTO 1249
2930  FORMAT(I4,$)
2359  CONTINUE
```

Standards

Fortran standards are denoted by year.

- Fortran 66
- Fortran 77
- Fortran 90
- Fortran 95
- Fortran 2003
- Fortran 2008
- Fortran 2015

Language Features

Fortran essentially has two formats: Fortran 77 and "modern".
All versions have:

- Matrices.
- Complex numbers.
- Good interoperability with C.
- REALLY good compilers.

Modern variants of Fortran additionally have:

- Dynamic memory allocation.
- Pointers.
- OOP.

Applications for Fortran

Bad applications:

- A website
- An OS kernel
- Text processing
- Random number generators

Good applications:

- Mathematics
- Engineering
- Statistics
- Scientific computing

Compilers

- GNU `gfortran` (free)
- Intel `ifortran`
- Portland Group `pgfortran`
- ...

Why Fortran?

- High level language
- Avoids fiddly memory management like in C
- Fortran binaries are VERY fast
- Good HPC support (MPI, OpenMP, ...)

Hello world example

hello.f

```
1       program helloworld
2       print *, "Hello world"
3       end program helloworld
```

Hello world example

hello.f90

```
1 program helloworld
2     print *, "Hello world"
3 end program helloworld
```

Comments

- `.f` is for F77, `.f90` is for F90+
- Don't use F77.
- Fortran is case insensitive.
- `print` and `write` print/write output
- `read` reads input

Contents

Basic Type

- Integer
- Real/double
- Complex/double complex
- Logical
- Character

Integer

Whole numbers.

- 1
- -517
- 0
- Not 1.2
- Not π
- Not 0.0

Real and Double

Floating point

- 1.1
- 3.14159
- Not 1
- Not $\sqrt{-1}$

Basic Numeric Operations

- `a=b`: assign the value of `b` to `a`
- `a+b`: add `a` and `b`
- `a-b`: subtract `b` from `a`
- `a*b`: multiply `a` and `b`
- `a/b`: divide `b` into `a`
- `a**b`: raise `a` to the power `b`

Quick example

```
1 program arithmetic
2     integer :: a = 2, b = 3
3
4     print *, a+b
5     print *, a**b
6     print *, a/b
7 end program
```

```
           5
           8
           0
```

Logical

Logical variables can take two values:

- `.true.`
- `.false.`

Comparing Logicals

- `.eqv.` tests if two logical expressions are equivalent
- `.neqv.` tests if two logical expressions are *not* equivalent
- `.and.` the and operator
- `.or.` the or operator
- `.not.` the negation operator

Comparing Numerics

- `a<b` or `a.lt.b`: `a` less than `b`
- `a<=b` or `a.le.b`: `a` less than or equal to `b`
- `a>b` or `a.gt.b`: `a` greater than `b`
- `a>=b` or `a.ge.b`: `a` greater than or equal to `b`
- `a==b` or `a.eq.b`: `a` equal to `b`
- `a/=b` or `a.ne.b`: `a` not equal to `b`

The type of a numeric comparison is of type `logical`.

Comparing Numerics

Note: The output of a comparison of numerics is logical:

- `a < b < c` makes no sense (types mismatch)
- Instead: `a < b .and. b < c`

Quick example

```
1 program logicals
2     integer :: a = 2, b = 3, c = 1
3
4     print *, a < b
5     print *, a /= b .and. a < c
6 end program
```

```
 T
 F
```

Implicit Declaration

- In Fortran, variables may be used *implicitly*.
- Do not get into the habit of doing this.
- You can turn this off in a program (function, subroutine) by declaring `implicit none`.
- Declaring `implicit none` is generally recommended.

Implicit variables quick example

Compiles:

```
1 program implicit_declaration
2     a = 2
3     b = 3
4     print *, a+b
5 end program
```

Fails to compile:

```
1 program implicit_declaration
2     implicit none
3     a = 2
4     b = 3
5     print *, a+b
6 end program
```

Example

Go to example.

Not covered:

- complex
- character
- kind/precision

Contents

if-then-else

Conditionals can take a few different forms:

```
1 if (condition) one-liner
```

```
1 if (condition 1) then
2     statement
3 else if (condition 2) then
4     statement
5 else if (...) then
6     ...
7 else
8     statement
9 end if
```

Note: the `else if` and `else` pieces are optional

if-then-else quick example

```
1  integer :: score
2  character(len=1) :: grade
3
4  if (score < 60) then
5      grade = "F"
6  else if (score < 70) then
7      grade = "D"
8  else if (score < 80) then
9      grade = "C"
10 else if (score < 90) then
11     grade = "B"
12 else
13     grade = "A"
14 end if
```

do Loops

```
1 do index = first, last, step
2     ! statements
3 end do
```

- index, first, last are integers
- step is a non-zero integer
- step can be omitted (and in this case, the step is 1)

do loops quick example

```
1  integer :: factorial, i, n
2
3  print *, "Give me a positive integer integer:"
4  read *, n
5
6  factorial = 1
7
8  do i = 2, n
9      factorial = factorial * i
10 end do
11
12 print *, n, "! = ", factorial
```

do Loops

```
1 do
2     ! statements
3 end do
```

- statements are executed repeatedly
- To exit the loop, use exit
- To jump to the next iteration, use cycle

do loops quick example

```
1  integer :: f, i, n
2
3  print *, "Give me an integer:"
4  read *, n
5
6  f = 1
7  i = 2
8  do
9      if (i <= n) then
10         f = f * i
11         i = i + 1
12     else
13         exit
14     end if
15 end do
16
17 print *, n, "! = ", f
```

Example

Go to example.

Not covered: `do-while` loops

Contents

Functions

```
1 ! Declaration
2 function foo(bar)
3     type :: foo
4     ! statements
5 end function
6
7 ! Invocation
8 a = foo(b)
```

- Can take variety of inputs.
- Returns single output.

Functions quick example

```
1  function circumference(r)
2      implicit none
3      real :: pi = 3.14159
4      real :: r
5      real :: circumference
6      circumference = 2.0*pi*r
7  end function circumference
8
9  program circles
10     real :: r = 2.0
11     real :: circumference
12     print *, circumference(r)
13 end program circles
```

```
   12.5663605
```

Intrinsics

- Built-in functions.
- Casting.
- Basic math utilitis.
- Bit-shifting.

Intrinsic Examples

Intrinsic	Effect	Intrinsic	Effect
`int`	Convert to `integer`	`mod`	Modular arithmetic
`real`	Convert to `real`	`abs`	Absolute value
`floor`	Greatest integer below	`sqrt`	Square root
`ceiling`	Smallest integer above	`exp`	Exponential

Intrinsics quick example

```
1 integer :: a = 2, b = 3
2
3 print *, mod(3, 2)
4 print *, real(a/b)
5 print *, real(a) / real(b)
```

```
           1
  0.00000000
  0.666666687
```

Subroutines

```
1 ! Declaration
2 subroutine foo(bar, baz)
3     type :: bar, baz
4     ! statements
5 end subroutine
6
7 ! Invocation
8 call foo(a, b)
```

- Can take variety of inputs.
- Returns a variety of outputs (modifying values of inputs).
- Equivalent in C is `void` function.

Intent

- Can declare intention of use for variable.
- `intent(in)`, `intent(out)`, `intent(inout)`.
- Like `implicit none`, not strictly necessary, but useful.

```
1 subroutine foo(a, b)
2     integer, intent(in) :: a
3     integer, intent(out) :: b
4     ! statements
5 end subroutine
```

Subroutine quick example

```
1  subroutine circ_stuff(r, circumference, area)
2      implicit none
3      real :: pi = 3.14159
4      real, intent(in) :: r
5      real, intent(out) :: circumference, area
6      circumference = 2.0 * pi * r
7      area = pi * r * r
8  end subroutine circ_stuff
9
10 program circles
11     real :: r = 2.0, circumference, area
12     call circ_stuff(r, circumference, area)
13     print *, "Circumference = ", circumference
14     print *, "Area = ", area
15 end program circles
```

```
 Circumference =     12.5663605
 Area =      12.5663605
```

Example

Go to example.

Contents

Precision

In C, there are two floating point types

- `float`
- `double`

In elder Fortran, the corresponding types are

- `real`
- `double precision`

Scientific Notation and Precision

$$\text{number} = \text{mantissa} \times 10^{\text{exponent}}$$

$$1.234 = 1234 \times 10^{-4}$$

- `real`: 23 bit of mantissa, 8 bits of exponent, and 1 sign bit.
- `double precision`: 52 bit of mantissa, 11 bits of exponent, and 1 sign bit.
- `real`: $\approx$ 6 decimal digits of precision
- `double precision`: $\approx$ 15 decimal digits of precision

Kind

- `kind` is a parameter that specifies the storage/precision of a type (beyond its default)
- `real(kind=4)`
- Different compilers assign different meaning to `kind`.
- Avoid this complexity with the intrinsic functions `selected_<type>_kind`

- `real(kind=selected_real_kind(6))`
- `real(kind=selected_real_kind(15))`
- `...`

Quick example

```
1  program kind_example
2      implicit none
3      integer, parameter :: r15 = selected_real_kind(15)
4      real :: x
5      real(kind=r15) :: y
6
7      x = 1.0
8      y = 1.0
9
10     print *, x
11     print *, y
12
13 end program
```

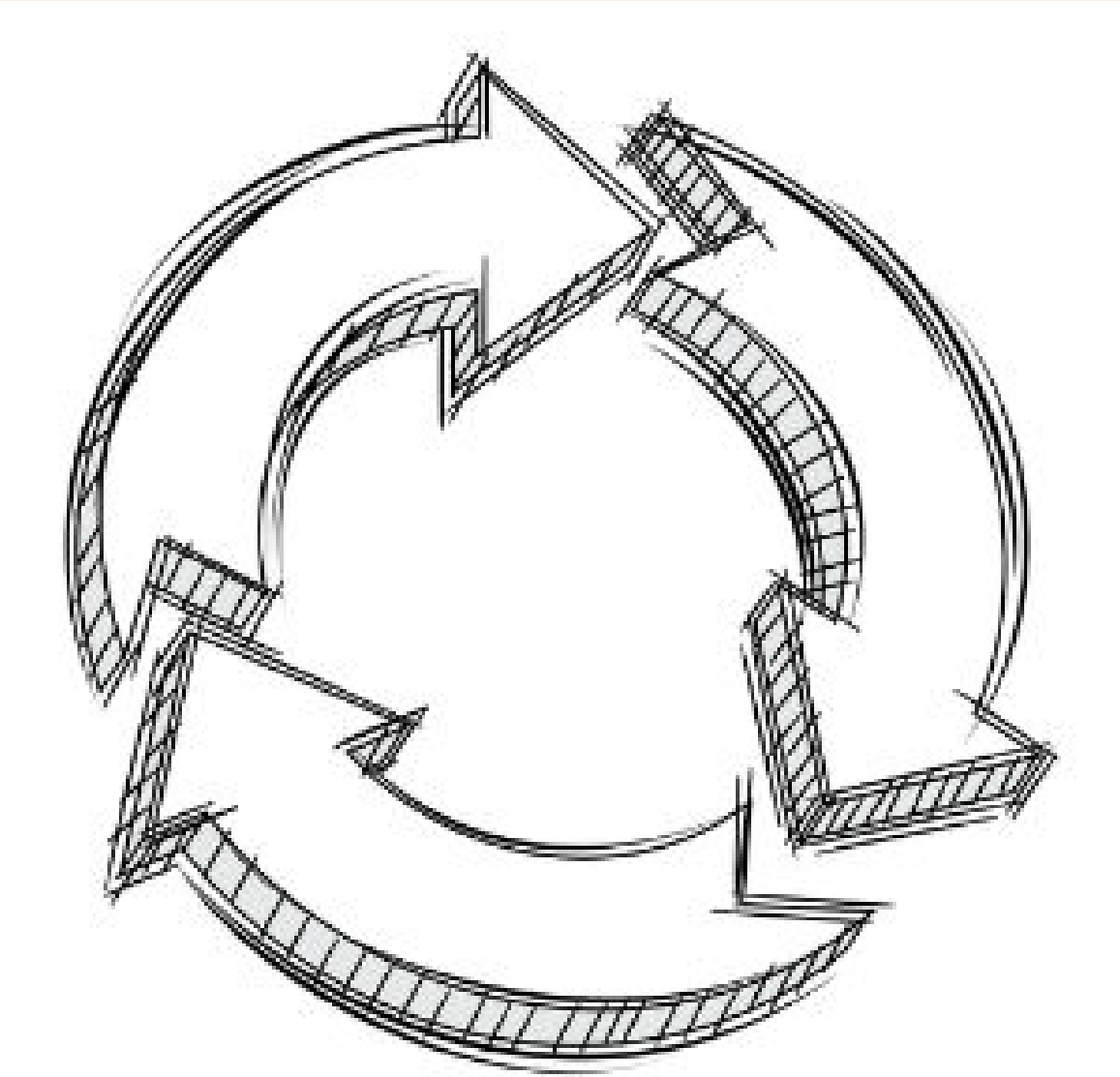
```
   1.00000000
   1.000000000000000
```

Contents

Other Important Topics Not Discussed Here

- Debugging
- Arrays
- Pointers
- Interfacing to C
- Tabs versus spaces

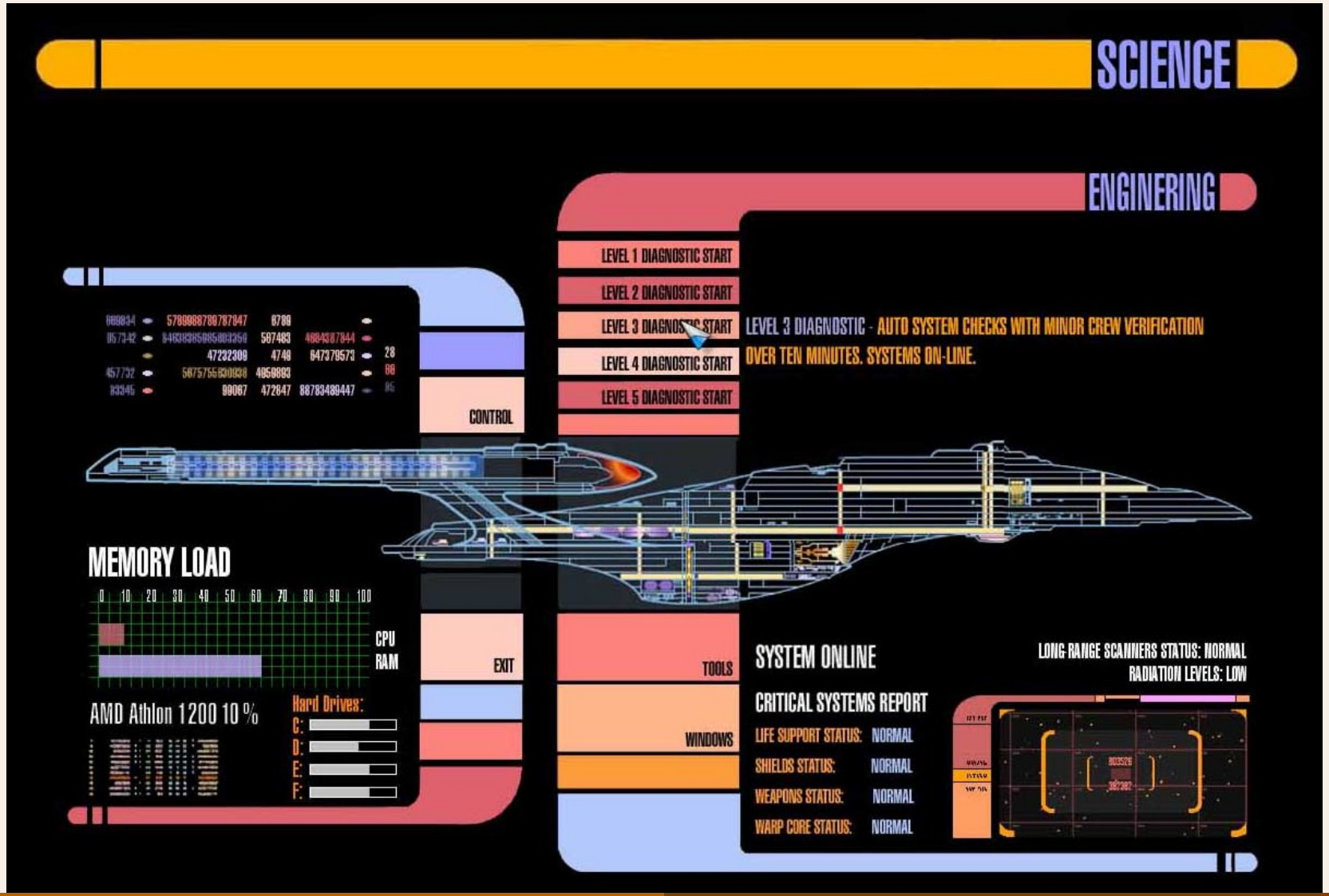
Some languages come and go



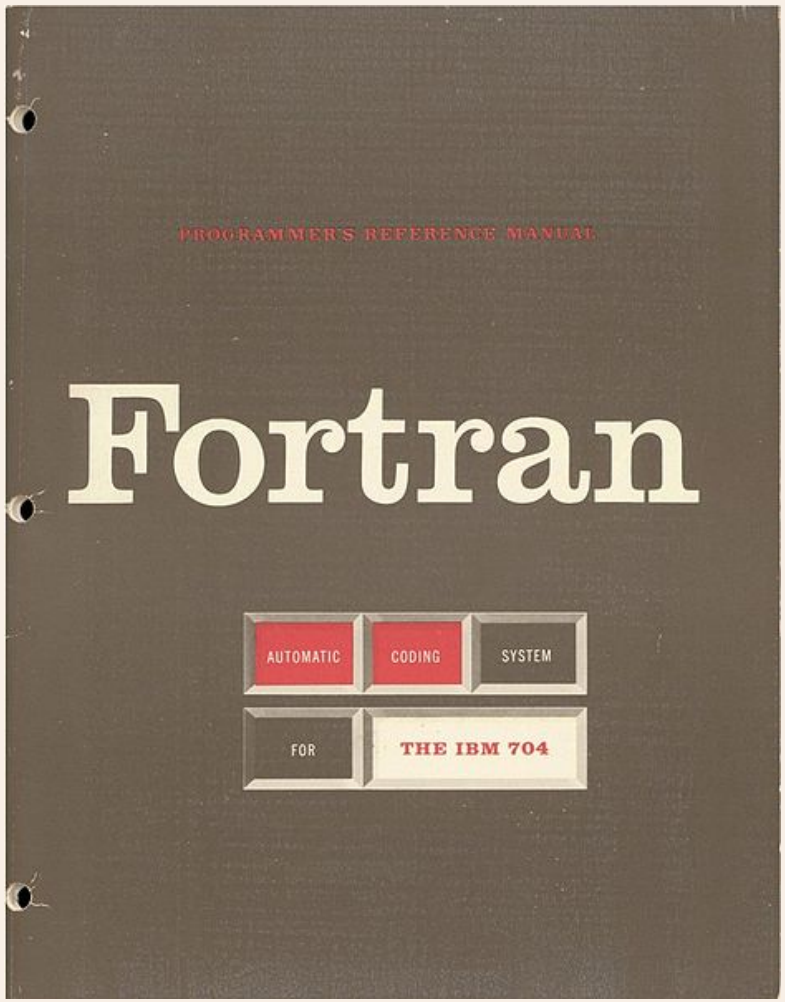
But with Fortran...

Because when the USS Enterprise makes her maiden voyage

Beneath the fancy blinking screens



You can bet that their crucial systems are powered by Fortran written in the 1970's



Thanks for coming!

Questions?