

Introduction to Modern Fortran

Data Types and Basic Calculation

Nick Maclaren

Computing Service

nmm1@cam.ac.uk, ext. 34761

November 2007

Data Types (1)

- **INTEGER** for exact **whole numbers**

e.g. 1, 100, 534, -18, -654321 etc.

In maths, an approximation to the ring \mathbb{Z}

- **REAL** for approximate, **fractional numbers**

e.g. 1.1, 3.0, 23.565, π , $\exp(1)$, etc.

In maths, an approximation to the field \mathbb{R}

- **COMPLEX** for complex, **fractional numbers**

e.g. (1.1, -23.565), etc.

In maths, an approximation to the field \mathbb{C}

Data Types (2)

- **LOGICAL** for **truth values**

These may have only values **true** or **false**

e.g. **.TRUE.** , **.FALSE.**

These are often called **boolean** values

- **CHARACTER** for **strings** of characters

e.g. **'?'**, **'Albert Einstein'**, **'X + Y = '**, etc.

The **string length** is part of the **type** in Fortran

There is no separate **character type**, unlike **C**

There is more on this later

Integers (1)

- **Integers** are restricted to lie in a finite range

Typically ± 2147483647 (-2^{31} to $2^{31}-1$)

Sometimes $\pm 9.23 \times 10^{17}$ (-2^{63} to $2^{63}-1$)

A compiler **may** allow you to select the range

Often including ± 32768 (-2^{15} to $2^{15}-1$)

Older and future systems may have other ranges

There is more on the arithmetic and errors later

Integers (2)

Fortran uses **integers** for:

- **Loop counts** and **loop limits**
- An **index** into an **array** or a position in a list
- An **index** of a **character** in a **string**
- As **error codes**, **type categories** etc.

Also use them for purely **integral values**

E.g. calculations involving **counts** (or money)

They can even be used for **bit masks** (see later)

Integer Constants

Usually, an optional **sign** and one or more **digits**

e.g. 0, 123, -45, +67, 00012345

E.g. '60' in `minutes = minutes + 60*hours`

Also allowed in **binary**, **octal** and **hexadecimal**

e.g. B'011001', O'35201', Z'a12bd'

- As with names, the case is irrelevant

There is a little more, which is covered later

Reals

- **Reals** are held as **floating-point** values
These also have a finite **range and precision**

It is essential to use **floating-point** appropriately

- Much of the Web is misleading about this
This course will mention only the bare minimum
See “**How Computers Handle Numbers**”

Reals are used for **continuously varying** values
Essentially just as you were taught at A-level

IEEE 754

You can assume a variant of **IEEE 754**

You should almost always use **IEEE 754** 64-bit

There is information on how to select it later

IEEE 754 32-, 64- and 128-bit formats are:

10^{-38} to 10^{+38} and 6–7 decimal places

10^{-308} to 10^{+308} and 15–16 decimal places

10^{-4932} to 10^{+4932} and 33–34 decimal places

Older and future systems may be different

Real Constants

- Real constants **must** contain a **decimal point** or an **exponent**

They can have an optional **sign**, just like **integers**

The basic **fixed-point form** is anything like:

123.456, -123.0, +0.0123, 123., .0123
0012.3, 0.0, 000., .000

- Optionally followed **E** or **D** and an exponent
1.0E6, 123.0D-3, .0123e+5, 123.d+06, .0e0

1E6 and **1D6** are also valid Fortran **real constants**

Complex Numbers

This course will generally ignore them
If you don't know what they are, don't worry

These are (**real**, **imaginary**) pairs of **REALs**
I.e. **Cartesian** notation, as at A-level

Constants are **pairs of reals** in **parentheses**
E.g. **(1.23, -4.56)** or **(-1.0e-3, 0.987)**

Declaring Numeric Variables

Variables hold values of different types

INTEGER :: count, income, mark

REAL :: width, depth, height

You can get all **undeclared variables** diagnosed
Add the statement **IMPLICIT NONE** at the start
of every **program**, **subroutine**, **function** etc.

If not, variables are **declared implicitly** by use

Names starting with **I–N** are **INTEGER**

Ones with **A–H** and **O–Z** are **REAL**

Implicit Declaration

- This is a common source of errors

`REAL :: metres, inches`

`inch3s = meters*30.48`

The effects can be even worse for **function calls**

Ask offline if you want to know the details

- Also the default `REAL` type is a disaster

Too inaccurate for practical use (see later)

- You should **always** use `IMPLICIT NONE`

Important Warning!

- I shall **NOT** do that myself

I warned you about this in the previous lecture
The problem is fitting all the text onto a slide
I shall often rely on **implicit typing** :-(

- **Do what I say, don't do what I do**

If I omit it in example files, it is a **BUG**

Assignment Statements

The general form is

`<variable> = <expression>`

This is actually very powerful (see later)

This **first** evaluates the **expression** on the **RHS**

It **then** stores the result in the **variable** on the **LHS**

It replaces whatever **value** was there before

For example:

`Max = 2 * Min`

`Sum = Sum + Term1 + Term2 + (Eps * Err)`

Arithmetic Operators

There are five built-in numeric operations

- + addition
 - subtraction
 - * multiplication
 - / division
 - ** exponentiation (i.e. raise to the power of)
- Exponents can be any arithmetic type:
INTEGER, REAL or COMPLEX
- Generally, it is best to use them in that order

Examples

Some examples of **arithmetic expressions** are

$A * B - C$

$A + C1 - D2$

$X + Y / 7.0$

$2 ** K$

$A ** B + C$

$A * B - C$

$(A + C1) - D2$

$A + (C1 - D2)$

$P ** 3 / ((X + Y * Z) / 7.0 - 52.0)$

Operator Precedence

Fortran uses normal mathematical conventions

- Operators bind according to **precedence**
- And then generally, from **left to right**

The **precedence** from highest to lowest is

****** exponentiation

***** / multiplication and division

+ - addition and subtraction

- Parentheses (‘(’ and ‘)’) are used to control it
- Use them whenever the **order matters** or it is **clearer**

Examples

$X + Y * Z$ is equivalent to $X + (Y * Z)$

$X + Y / 7.0$ is equivalent to $X + (Y / 7.0)$

$A - B + C$ is equivalent to $(A - B) + C$

$A + B ** C$ is equivalent to $A + (B ** C)$

$- A ** 2$ is equivalent to $-(A ** 2)$

$A - (((B + C)))$ is equivalent to $A - (B + C)$

- You can force any order you like

$(X + Y) * Z$

Adds X to Y and **then** multiplies by Z

Warning

$X + Y + Z$ may be evaluated as any of
 $X + (Y + Z)$ or $(X + Y) + Z$ or $Y + (X + Z)$ or . . .

Fortran defines **what** an expression means
It does not define **how** it is calculated

They are all **mathematically** equivalent
But may sometimes give slightly different results

See “**How Computers Handle Numbers**” for more

Precedence Problems

Mathematical conventions vary in some aspects

$A / B * C$ – is it $A / (B * C)$ or $(A / B) * C$?

$A ** B ** C$ – is it $A ** (B ** C)$ or $(A ** B) ** C$?

Fortran specifies that:

$A / B * C$ is equivalent to $(A / B) * C$

$A ** B ** C$ is equivalent to $A ** (B ** C)$

- Yes, ****** binds from **right to left**!

Parenthesis Problems

Always use **parentheses** in ambiguous cases
If only to imply “**Yes, I really meant that**”

And to help readers used to different rules
Programming languages vary in what they do

Be careful of over-doing it – what does this do?

$((A+(P*R+B)/2+B**3)/(4/Y)*C+D))+E$

- Several, simpler statements is better style

Integer Expressions

I.e. ones of **integer constants** and **variables**

```
INTEGER :: K, L, M  
N = K*(L+2)/M**3-N
```

These are evaluated in **integer arithmetic**

- Division always truncates **towards zero**

If $K = 4$ and $L = 5$, then $K+L/2$ is 6
 $(-7)/3$ and $7/(-3)$ are both -2

Mixed Expressions

INTEGER and **REAL** is evaluated as **REAL**
Either and **COMPLEX** goes to **COMPLEX**

Be careful with this, as it can be deceptive

```
INTEGER :: K = 5  
REAL :: X = 1.3  
X = X+K/2
```

That will add **2.0** to **X**, not **2.5**
K/2 is still an **INTEGER** expression

Conversions

There are several ways to force conversion

- Intrinsic functions **INT**, **REAL** and **COMPLEX**

$$X = X + \text{REAL}(K)/2$$
$$N = 100 * \text{INT}(X/1.25) + 25$$

You can use appropriate constants

You can even add zero or multiply by one

$$X = X + K/2.0$$
$$X = X + (K + 0.0)/2$$

The last isn't very nice, but works well enough

And see later about **KIND** and precision

Mixed-type Assignment

<real variable> = <integer expression>

- The RHS is converted to **REAL**

Just as in a mixed-type expression

<integer variable> = <real expression>

- The RHS is **truncated** to **INTEGER**

It is always truncated **towards zero**

Similar remarks apply to **COMPLEX**

- The **imaginary** part is discarded, quietly

The **RHS** is evaluated **independently** of the **LHS**

Examples

INTEGER :: K = 9, L = 5, M = 3, N

REAL :: X, Y, Z

X = K ; Y = L ; Z = M

N = (K/L)*M

N = (X/Y)*Z

N will be 3 and 5 in the two cases

$(-7)/3 = 7/(-3) = -2$ and $7/3 = (-7)/(-3) = 2$

Numeric Errors

See “How Computers Handle Numbers”

This a a **very** minimal summary

- **Overflowing the range** is a serious error
As is **dividing by zero** (e.g. **123/0** or **0.0/0.0**)
Fortran does not define what those cases do
- Each **numeric type** may behave differently
Even different compiler options will, too
- And do **not** assume results are predictable

Examples

Assume the **INTEGER** range is ± 2147483647

And the **REAL** range is $\pm 10^{38}$

- Do you know what this is defined to do?

```
INTEGER :: K = 1000000  
REAL :: X = 1.0e20  
PRINT *, (K*K)/K, (X*X)/X
```

- The answer is “**anything**” – and it means it
Compilers optimise on the basis of no errors
Numeric errors can cause **logical errors**

Numeric Non-Errors (1)

- Conversion to a **lesser** type loses information
You will get no warning of this, unfortunately

REAL \Rightarrow **INTEGER** truncates towards zero

COMPLEX \Rightarrow **REAL** drops the imaginary part

COMPLEX \Rightarrow **INTEGER** does both of them

That also applies when dropping in **precision**

E.g. assigning a **64-bit** real to a **32-bit** one

Numeric Non-Errors (2)

Fortran does **NOT** specify the following
But it is true on all systems you will use

Results too small to represent are not errors

- They will be replaced by zero if necessary
- **Inexact results** round to the nearest value

That also applies when dropping in **precision**

Intrinsic Functions

Built-in functions that are always available

- No declaration is needed – just use them

For example:

```
Y = SQRT(X)
```

```
PI = 4.0 * ATAN(1.0)
```

```
Z = EXP(3.0*Y)
```

```
X = REAL(N)
```

```
N = INT(X)
```

```
Y = SQRT(-2.0*LOG(X))
```

Intrinsic Numeric Functions

REAL(n) ! Converts its argument n to REAL
INT(x) ! Truncates x to INTEGER (to zero)
AINT(x) ! The result remains REAL
NINT(x) ! Converts x to the nearest INTEGER
ANINT(x) ! The result remains REAL
ABS(x) ! The absolute value of its argument
! Can be used for INTEGER, REAL or COMPLEX
MAX(x,y,...) ! The maximum of its arguments
MIN(x,y,...) ! The minimum of its arguments
MOD(x,y) ! Returns x modulo y

And there are more – some are mentioned later

Intrinsic Mathematical Functions

SQRT(x)	! The square root of x
EXP(x)	! e raised to the power x
LOG(x)	! The natural logarithm of x
LOG10(x)	! The base 10 logarithm of x
SIN(x)	! The sine of x, where x is in radians
COS(x)	! The cosine of x, where x is in radians
TAN(x)	! The tangent of x, where x is in radians
ASIN(x)	! The arc sine of x in radians
ACOS(x)	! The arc cosine of x in radians
ATAN(x)	! The arc tangent of x in radians

And there are more – see the references

Bit Masks

As in **C** etc., **integers** are used for these
Use is by weirdly-named functions (historical)

Bit indices start at **zero**

Bit **K** has value 2^K (little-endian)

As usual, stick to **non-negative** integers

- A little tedious, but very easy to use

Bit Intrinsic

<code>BIT_SIZE(x)</code>	! The number of bits in x
<code>BTEST(x, n)</code>	! Test bit n of x
<code>IBSET(x, n)</code>	! Set bit n of x
<code>IBCLR(x, n)</code>	! Clear bit n of x
<code>IBITS(x, m, n)</code>	! Extract n bits
<code>NOT(x)</code>	! NOT x
<code>IAND(x, y)</code>	! x AND y
<code>IOR(x, y)</code>	! x OR y
<code>IEOR(x, y)</code>	! x (exclusive or) y
<code>ISHFT(x, n)</code>	! Logical shift
<code>ISHFTC(x, n, [k])</code>	! Circular shift

Logical Type

These can take only two values: **true** or **false**

.TRUE. and **.FALSE.**

- Their type is **LOGICAL** (not **BOOL**)

LOGICAL :: red, amber, green

IF (red) THEN

PRINT *, 'Stop'

red = .False. ; amber = .True. ; green = .False.

ELSIF (red .AND. amber) THEN

. . .

Relational Operators

- **Relations** create **LOGICAL** values

These can be used on **any other** built-in type

== (or **.EQ.**) equal to

/= (or **.NE.**) not equal to

These can be used only on **INTEGER** and **REAL**

< (or **.LT.**) less than

<= (or **.LE.**) less than or equal

> (or **.GT.**) greater than

>= (or **.GE.**) greater than or equal

See “**How Computers Handle Numbers**” for more

Logical Expressions

Can be as complicated as you like

Start with `.TRUE.`, `.FALSE.` and `relations`

Can use `parentheses` as for numeric ones

`.NOT.`, `.AND.` and `.OR.`

`.EQV.` must be used instead of `==`

`.NEQV.` must be used instead of `/=`

- Fortran is `not` like `C`-derived languages
`LOGICAL` is not a sort of `INTEGER`

Short Circuiting

LOGICAL :: flag

```
flag = ( Fred( ) > 1.23 .AND. Joe( ) > 4.56 )
```

Fred and Joe may be called in either order

If Fred returns 1.1, then Joe may not be called

If Joe returns 3.9, then Fred may not be called

Fortran expressions define the answer only

The behaviour is up to the compiler

One of the reasons that it is so optimisable

Character Type

Used when **strings of characters** are required
Names, descriptions, headings, etc.

- Fortran's basic type is a **fixed-length string**
Unlike almost all more recent languages

- **Character constants** are **quoted strings**

```
PRINT *, 'This is a title'
```

```
PRINT *, "And so is this"
```

The **characters** between **quotes** are the **value**

Character Data

- The **case of letters** is significant in them
Multiple spaces are not equivalent to one space
Any **representable character** may be used

The **only** Fortran syntax where the above is so
Remember the line joining method?

In `'Time^=^13:15'`, with '^' being a space
The character string is of length **15**
Character **1** is **T**, **8** is a space, **10** is **1** etc.

Character Constants

- "This has UPPER, lower and MiXed cases"
- 'This has a double quote (") character'
- "Apostrophe (') is used for single quote"
- "Quotes (""") are escaped by doubling"
- 'Apostrophes (') are escaped by doubling'
- 'ASCII ' , |, ~, ^, @, # and \ are allowed here'

- "Implementations may do non-standard things"
- 'Backslash (\) MAY need to be doubled'
- "Avoid newlines, tabs etc. for your own sanity"

Character Variables

CHARACTER :: answer, marital_status

CHARACTER(LEN=10) :: name, dept, faculty

CHARACTER(LEN=32) :: address

answer and marital_status are each of length 1

They hold precisely one character each

answer might be blank, or hold 'Y' or 'N'

name, dept and faculty are of length 10

And address is of length 32

Another Form

```
CHARACTER :: answer*1,      &  
           marital_status*1, name*10,    &  
           dept*10, faculty*10, address*32
```

While this form is historical, it is more compact

- Don't mix the forms – this is an abomination

```
CHARACTER(LEN=10) :: dept, faculty, addr*32
```

- For obscure reasons, using **LEN=** is cleaner
It avoids some arcane syntactic “gotchas”

Character Assignment

```
CHARACTER(LEN=6) :: forename, surname  
forename = 'Nick'  
surname = 'Maclaren'
```

`forename` is padded with spaces ('Nick^^')
`surname` is truncated to fit ('Maclar')

- Unfortunately, you won't get told
But at least it won't overwrite something else

Character Concatenation

Values may be **joined** using the `//` operator

```
CHARACTER(LEN=6) :: identity, A, B, Z  
identity = 'TH' // 'OMAS'  
A = 'TH' ; B = 'OMAS'  
Z = A // B
```

Sets `identity` to 'THOMAS'
But `Z` is set to 'TH' – why?

`//` does not remove **trailing spaces**
It uses the whole length of its inputs

Substrings

If `Name` has length **9** and holds `'Marmaduke'`

`Name(1:1)` would refer to `'M'`

`Name(2:4)` would refer to `'arm'`

`Name(6:)` would refer to `'duke'` – **note the form!**

We could therefore write statements such as

```
CHARACTER :: name*20, surname*18, title*4
```

```
name = 'Dame Edna Everage'
```

```
title = name(1:4)
```

```
surname = name(11:)
```

Example

This is not an example of good style!

```
PROGRAM message
  IMPLICIT NONE
  CHARACTER :: mess*72, date*14, name*40
  mess = 'Program run on'
  mess(30:) = 'by'
  READ *, date, name
  mess(16:29) = date
  mess(33:) = name
  PRINT *, mess
END PROGRAM message
```


Warning – a “Gotcha”

CHARACTER substrings look like array sections
But there is no equivalent of array indexing

```
CHARACTER :: name*20, temp*1  
temp = name(10)
```

- `name(10)` is an implicit function call
Use `name(10:10)` to get the tenth character

CHARACTER variables come in various lengths
`name` is **not** made up of 20 variables of length 1

Character Intrinsic

LEN(c)	! The STORAGE length of c
TRIM(c)	! c without trailing blanks
ADJUSTL(C)	! With leading blanks removed
INDEX(str,sub)	! Position of sub in str
SCAN(str,set)	! Position of any character in set
REPEAT(str,num)	! num copies of str, joined

And there are more – see the references

Examples

```
name = ' Bloggs  '  
newname = TRIM(ADJUSTL(name))
```

newname would contain 'Bloggs'

```
CHARACTER(LEN=6) :: A, B, Z  
A = 'TH' ; B = 'OMAS'  
Z = TRIM(A) // B
```

Now Z gets set to 'THOMAS' correctly!

Collation Sequence

This controls whether "ffred" < "Fred" or not

- Fortran is **not** a **locale**-based language
It specifies **only** the following

'A' < 'B' < ... < 'Y' < 'Z'	These ranges
'a' < 'b' < ... < 'y' < 'z'	will not
'0' < '1' < ... < '8' < '9'	overlap
' ' is less than all of 'A', 'a' and '0'	

A **shorter** operand is extended with **blanks** (' ')

Named Constants (1)

- These have the **PARAMETER** attribute

```
REAL, PARAMETER :: pi = 3.14159  
INTEGER, PARAMETER :: maxlen = 100
```

They can be used anywhere a **constant** can be

```
CHARACTER(LEN=maxlen) :: string  
circum = pi * diam  
IF (nchars < maxlen) THEN  
    . . .
```

Named Constants (2)

Why are these important?

They reduce mistyping errors in long numbers

Is `3.14159265358979323846D0` correct?

They can make formulae etc. much clearer

Much clearer **which** constant is being used

They make it easier to modify the program later

`INTEGER, PARAMETER :: MAX_DIMENSION = 10000`

Named Character Constants

```
CHARACTER(LEN=*), PARAMETER ::      &  
    author = 'Dickens', title = 'A Tale of Two Cities'
```

`LEN=*` takes the length from the data

It is permitted to define the **length** of a constant
The data will be **padded** or **truncated** if needed

- But the above form is generally the best

Named Constants (3)

- Expressions are allowed in constant values

```
REAL, PARAMETER :: pi = 3.14135, &  
    pi_by_4 = pi/4, two_pi = 2*pi
```

```
CHARACTER(LEN=*), PARAMETER :: &  
    all_names = 'Pip, Squeak, Wilfred', &  
    squeak = all_names[6:11]
```

Generally, anything reasonable is allowed

- It must be determinable at compile time

Initialisation

- Variables start with **undefined** values
They often vary from run to run, too
- **Initialisation** is very like defining constants
Without the **PARAMETER** attribute

```
INTEGER :: count = 0, I = 5, J = 100
```

```
REAL :: inc = 1.0E5, max = 10.0E5, min = -10.0E5
```

```
CHARACTER(LEN=10) :: light = 'Amber'
```

```
LOGICAL :: red = .TRUE., blue = .FALSE., &  
          green = .FALSE.
```

Information for Practicals

A **program** has the following basic **structure**:

```
PROGRAM name  
Declarations  
Other statements  
END PROGRAM name
```

Read and write data from the terminal using:

```
READ *, variable [ , variable ]...  
PRINT *, expression [ , expression ]...
```