Data Types and Basic Calculation
Intrinsic Data Types

Fortran supports five intrinsic data types:

1. **INTEGER** for exact whole numbers
   - e.g., 1, 100, 534, -18, -654321, etc.

2. **REAL** for approximate, fractional numbers
   - e.g., 1.1, 3.0, 23.565, 3.1415, \( \exp(1) \), etc.

3. **COMPLEX** for complex, fractional numbers
   - e.g., (1.1, -23.565), etc.
4. **LOGICAL** for *truth values* (boolean)
   These may only have values of true or false e.g., `.TRUE.`, `.FALSE.`.

5. **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 special *character type* (unlike C).
Integers (1)

- Integers are restricted to lie in a finite range.
  Typically \( \pm 2^{147483647} \) (\(-2^{31}\) 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 \( \pm 32768 \) (\(-2^{15}\) to \(2^{15}-1\))

- More on 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**
Example: Calculations involving counts
Reals

- **Reals** are used for continuously varying values.
- **Reals** are stored as floating-point values. They also have a finite range and precision.

**THEY ARE INEXACT**

- It is essential to use floating-point appropriately.
Floating Point Standard

- The Institute of Electrical and Electronics Engineers (IEEE) has produced a standard for floating point arithmetic. IEEE 754-1985.
- This defines 32-bit and 64-bit floating point representations.
- **32-bit**: $10^{-38}$ to $10^{+38}$ and 6-7 decimal places
- **64-bit**: $10^{-308}$ to $10^{+308}$ and 15-16 decimal places
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 by E or D and an exponent
  
  1.0D6, 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)

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
Names starting with A-H and O-Z are REAL
YOU SHOULD ALWAYS USE IMPLICIT NONE
Assignment Statements

The general form is:

\[ \text{<variable>} = \text{<expression>} \]

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:

\[
\begin{align*}
\text{xyMax} & = 2 \times \text{xyMin} \\
\text{mySum} & = \text{mySum} + \text{Term}_1 + \text{Term}_2 + (\text{Eps} \times \text{Err})
\end{align*}
\]
There are **five** built-in numeric operations:

- **+** addition
- **-** subtraction
- **\*** multiplication
- **/** division
- **** exponentiation

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:

\[ A*B-C \]
\[ A + C1 - D2 \]
\[ X + Y/7.0 \]
\[ 2**K \]
\[ A**B + C \]
\[ (A + C1) - D2 \]
\[ A + (C1 - D2) \]
\[ P**3/((X+Y*Z)/7.0-52.0) \]
Operator Precedence

Fortran uses normal mathematical conventions

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

The **precedence** from **highest** to **lowest** is:

- **** exponentiation
- * / multiplication and division
- + - addition and subtraction

Parentheses are used to control it. Use them whenever the **order matters** or it is **clearer**.
Examples

\[ X + Y \times Z \] is equivalent to \[ X + (Y \times Z) \]
\[ X + Y / 7.0 \] is equivalent to \[ X + (Y / 7.0) \]
\[ A - B + C \] is equivalent to \[ (A - B) + C \]
\[ A + B \times C \] is equivalent to \[ A + (B \times C) \]
\[ -A \times 2 \] is equivalent to \[ -(A \times 2) \]
\[ A - (((B + C))) \] is equivalent to \[ A - (B + C) \]

You can force any order you like:
\[ (X + Y) \times Z \]
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

The are all mathematically equivalent
But may sometimes give slightly different results
Integer Expressions

Expressions involving integer constants and variables. These are evaluated in integer arithmetic. Division always truncates toward zero.

```
INTEGER :: K, L, N
N = K+L/2
If K = 4 and L = 5 then N = 6

(-7)/3 and 7/(-3) are both -2
```
Mixed Expressions

In the CPU calculations must be performed between objects of the same type, so if an expression mixes type some objects must change type.

Default types have an implied ordering:

1. INTEGER (lowest)
2. REAL
3. COMPLEX (highest)

The result of an expression is always of the highest type. e.g., INTEGER * REAL gives a REAL

Be careful with this as it can be deceptive!
Conversions

There are several ways to force conversion

- **Intrinsic functions** INT, REAL and COMPLEX
  \[ X = X + \text{REAL}(K)/2 \]
  \[ N = 100 \times \text{INT}(X/1.25)+25 \]

- Use the appropriate constants. (You can even add zero or multiply by one
  \[ X = X + K/2.0 \]
  \[ X = X+(K+0.0)/2 \]

The second method isn’t very nice but works well enough. (See later about **KIND** and precision)
Mixed-type Assignment

\[ \text{<real variable>} = \text{<integer expression>} \]

- The RHS is converted to \text{REAL}
- Just as in a mixed-type expression

\[ \text{<integer variable>} = \text{<real expression>} \]

- The RHS is truncated to \text{INTEGER}
- It is always truncated \text{toward zero}

Similar remarks apply to \text{COMPLEX}

The \text{RHS} is evaluated \text{independently} of the \text{LHS}

Example: \text{mixedassigned.f90}
Intrinsic Functions

**Built-in functions** that are always available
- No *declaration* is needed -- just use them!

Examples:

- \( Y = \text{SQRT}(X) \)
- \( \text{PI} = 4.0 \times \text{ATAN}(1.0) \)
- \( Z = \text{EXP}(3.0 \times Y) \)
- \( X = \text{REAL}(N) \)
- \( N = \text{INT}(X) \)
- \( Y = \text{SQRT}(-2.0 \times \text{LOG}(X)) \)
Intrinsic Numeric Functions

REAL(n) ! Converts its argument 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 of x
LOG(x)    ! The natural logarithm of x
LOG10(x)  ! The base 10 logarithm of x

SIN(x)    ! The sine of x (x in radians)
COS(x)    ! The cosine of x (x in radians)
TAN(x)    ! The tangent of x (x in radians)
ASIN(x)   ! The arc sine of x (x in radians)
ACOS(x)   ! The arc cosine of x (x in radians)
ATAN(x)   ! The arc tangent of x (x in radians)
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.
ELSE IF (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 to
  > (or .GT.) greater than
  >= (or .GE.) greater than or equal to
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.` can be used instead of `==

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

Fortran is not like C-derived languages

`LOGICAL` is not a sort of `INTEGER`
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

In ‘Time^^=^^13:15’, with ‘^’ being a space
The character string is of length 14
Character 1 is T, 8 is a space, 10 is I, etc.

Example program: charstrings.f90
**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. *Address* is of length 32.
Another Form

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

While this form is historical it is more compact

Don’t mix the forms -- that is an abomination

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

For some obscure reasons using LEN= is cleaner
It avoids some arcane syntactic “gotchas”
Character Assignment

\[
\text{CHARACTER(LEN=6) :: firstname, lastname} \\
\text{firstname = ‘Mark’ ; lastname = ‘Branson’}
\]

\text{firstname} \text{ is padded with spaces (‘Mark^^’)}  \\
\text{lastname} \text{ is truncated to fit (‘Branso’)}

Unfortunately you won’t get told  
But at least it won’t overwrite something else
Character Concatenation

Values may be joined using the `//` operator

```fortran
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 used the whole length of its inputs
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*15, lastname*7, title*3
   name = ‘Mr. Joe Johnson’
   title = name(1:3)
   lastname = name(9:;)

Substrings
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 10th character

CHARACTER variables come in various lengths
name is not made up of 20 variables of length 1
Intrinsic Character Functions

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

```fortran
name = ' Smith
newname = TRIM(ADJUSTL(name))
```

newname would contain ‘Smith’

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

Now Z gets set to ‘THOMAS’ correctly
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 equations 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

```fortran
REAL, PARAMETER :: pi = 3.1415, &
   pi_by_4 = pi/4, two_pi = 2*pi

CHARACTER(LEN=*), PARAMETER :: &
   all_names = 'Bob, Jennifer, Karen', &
   karen = all_names(16:20)
```

Generally anything reasonable is allowed
It must be determinable at compile time
Initialization

Variables start with **undefined** values
They often vary from run to run, too

**Initialization** is much 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.
```