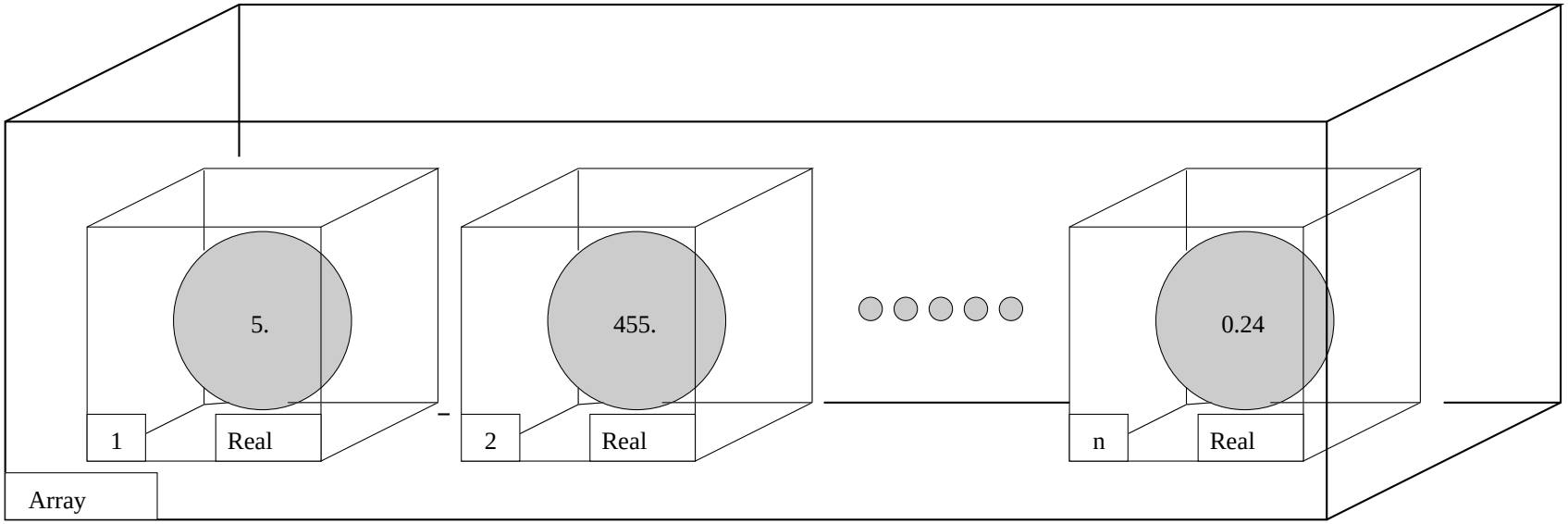
Outline

THE STORY SO FAR...

- Understand about data types and how to assign them
- Basic input/output
- How to declare, write and use Functions and Subroutines
- Use logic and loops to control the flow of your program

What is an Array?

- There are many situations where we would like to group data together



- One way to do this is to have a group or **array** of data:
  - with the same name;
  - but each with an index or subscript to identify it
  - Here we all elements have the same name "**Array**" and type "**Real**"
  - We identify indiviual elements by their subscript

How do they work?

- We can reference the element of an array by putting the index in parenthesis

Examples

```
x(1) = 1.          !  x is an array
x(2) = y(1) + y(2)   !  y is also an array
```

- Arrays are stored **contiguously** in the memory stack
- In the previous example the variable "Array" stores the position of the first element
- Referring to the element `Array(i)` tells the compiler:
  - start at position `Array`
  - move `i` positions in the memory stack
  - look for the data there

Declaring an Array

- An array may be declared in one of three ways:

```
REAL, DIMENSION(10) ::  a,b,c
INTEGER ::  d(20)
REAL ::  e(-10:10)
```

- a, b, and c each have 10 real elements
- d has 20 integer elements
- e has 21 real elements, indexed from -10 to 10

Assigning Values to an Array

- We can assign all elements in an array the same value simply by using the equals operator:

```
a = 0.          !  sets all elements in a to zero
```

- We can use **array constructors** to assign values to an array:

```
a = (/ 1,2,3,4,5,6,7,8,9,10 /)
```

- There is also a special syntax, the *implied DO*, to assign values in a neater way

```
a = (/ (i,i=1,10) /)
```

Input/Output with Arrays

- We can input and out the array elements themselves in the same way as scalar variables

```
READ *,a(1),a(2)   !  read in first two
elements of array
```

- Using only the array name will reference the entire array

```
PRINT *,a   !  print all elements of array
```

- Part of an array can be referenced using *implied DO* as before

```
PRINT *,(a(i),i=3,12,3)   !  print elements 3,
6, 9 and 12 of array
```

- The statements `PRINT *` and `READ *` are list-directed, we let the compiler choose how to deal with the data.
- We can specify a **format**, containing *edit descriptors* to have more control over how the data is input/output
- The format can be included in three ways:

```
PRINT format_string, input_list
PRINT label, input_list
```

- `format_string` is written in the form:

```
format_string = '(edit_descriptor_list)'
```

or

```
label FORMAT(edit_descriptor_list)
```

Edit Descriptors

- Edit descriptors for READ

Descriptor	Meaning
Iw	Read the next w characters as an integer
Fw.d	Read the next w characters as a real, d digits
Ew.d	after the decimal point
Aw	Read the next w characters
A	Read sufficient characters to fill the input list as characters
Lw	Read the next w characters as the representation of a logical value
nX	Ignore the next n characters
Tc	Next character to be read is at position c
TLn	Next character to be read is n characters before (TL)
TRn	or after (TR) the current position.

EXAMPLES

- Some examples using the READ statement

```
INTEGER ::  n,m,p
READ '(I4,I3,I2)', n,m,p
!    Input 246813579
```

- what are the values of n, m and p?
- What is the value of n from the following

```
READ '(2X,I4)',n
```

- What is the value of r1, r2, r3 and r4

```
REAL ::  r1,r2,r3,r4 READ
'(F3.1,F2.2,F3.0,TL6,F4.2)',r1,r2,r3,r4   !
Input 975318642
```

- Now suppose data is 1.345.67. What are the values?

EDIT DESCRIPTORS FOR PRINT

Descriptor	Meaning
Iw	Output the next w characters as an integer
Fw.d	Output a real number w positions , d digits after the decimal point
Ew.d	Output a real number in the next w positions in exponent form, with d digits
	decimal places in mantissa and four characters for exponent
Aw	Output a character string in the next w positions
A	Output a character string at next position
Lw	Output w-1 blanks with T or F in next position logical value
nX	Ignore the next n character positions
Tc	Output next item starting at position c
TLn	Next item to be output n characters before (TL)
TRn	or after (TR) the current position
"cdedfff..'	Output string in quotes
'cdefff...'	Output string in quotes
/	start on a new line

New Keywords

- Using READ and PRINT so far has printed to the screen and read from the keyboard
- If we wish to write to a file we must use the keyword WRITE.

Example statements

```
WRITE(10,'(3(I3,1X))')n,m,p   !  write out 3 ints to some file linked to the integer 10
READ(9,'(E.5)')x   !  read in the real number x from a file linked to 9
```

- The integer linked to the file is called the UNIT

Opening a File for Reading/Writing

- To open a file we use keyword `OPEN` and link it to a `UNIT`

```
OPEN(UNIT=10,FILE='results.dat',IOSTAT=ios)
IF(ios /= 0) STOP   !  stop if file not opened
```

- We can then read or write to `'results.dat'` simply by writing

```
WRITE(10,'(3(I3,1X))')n,m,p   !  write
integers to results.dat
```

or

```
READ(10,'(3(I3,1X))')n,m,p   !  read integers
from results.dat
```

Declaring a New Type

- We declare new data types before declaring variables at the top of the program
- Example: create a data type to store coordinates in the (x,y) plane

```
TYPE point
  REAL ::  x,y
END TYPE
```

- Example: create a data type to store the date in days, months, and years

```
TYPE date
  INTEGER ::  day,month,year
END TYPE
```

Using Your New Type

- To create a variable of the new type, the type specification must be at the top of each subprogram
- We declare the variable by using `TYPE(new_type)`
- We reference elements inside the new type using % sign

```
TYPE(point) ::  a
a%x = 1.  ; a%y = 2.        !     a = (1,2)
```

- You can create a new type using other types you have created so long as they have been specified before
  - a line may be made up of two points
  - an n sided polygon may be made up of n points
- Try to rewrite some of the examples using new types, for instance this type in question 8

Compiler Errors

- Compiler errors can often be cryptic
- Work your way down from the first error, recompiling until each error is removed
- Use highlighting styles to help with syntax
- Enter cryptic errors into google to look for help

- Ask for help!!

Debugging

- Attempt to program small parts of the program independently before attempting the whole problem
- Effort spent designing your programs will pay off
- It is important to leave comments in your code
- Use output to the screen to try to understand where things are going wrong
- Test the lowest level procedures in your code, then build your way upwards
- If you are using a new method, check results against old or other working codes