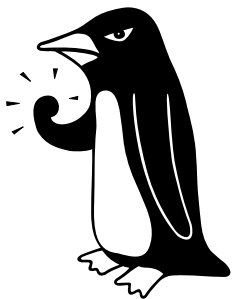


Intro to Fortran

“The Angry Penguin”, used under creative commons licence
from Swantje Hess and Jannis Pohlmann.



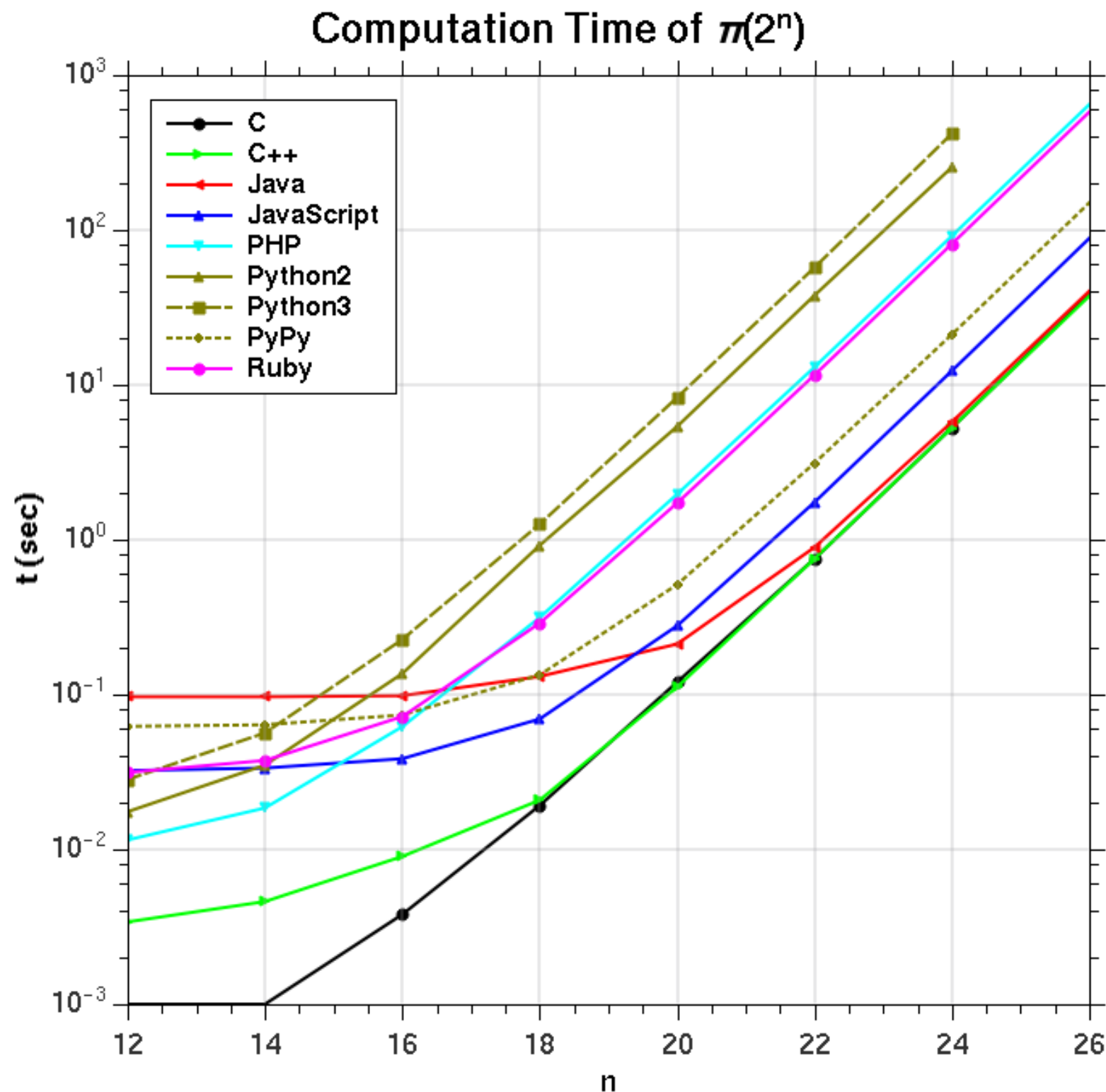
Warwick RSE

25th Sept 2023

I don't know what the programming language
of the year 2000 will look like, but I know it will
be called FORTRAN

Charles Anthony Richard Hoare, circa 1982

Why Fortran?



- Faster code than Python/R/Matlab
- Comparable speed to C
- Easier to write than C
- Lots of libraries
- Lots of experienced people

Language Standards

- All languages are defined by standards which say what is and isn't valid in a language
- Compilers will list themselves as supporting or not supporting a given language standard and may also have their own "extensions" beyond the standard
- **Pick a standard and stick to it!**

Language Standards

- FORTRAN 66 and FORTRAN 77 - Old Fortran. Still common in the wild. Fully supported by compilers but **very** old fashioned. Similar but different to what we are teaching
- Fortran 90 and Fortran 95 - Early modern Fortran. Fully supported by almost all compilers. Has a few odd restrictions and limitations
- Fortran 2003 - Very well supported by compilers (except for support for international character sets)
- Fortran 2008 - Now very well supported by modern compilers
- Fortran 2018 - New standard compiler support is now good, but most of the features are only for “power users”
- Fortran 2023 - Just about to be ratified. Some useful stuff but not game changing

Case Sensitivity

- Fortran is case insensitive
 - "var" and "Var" are the same variable
- Means that you have a lot of flexibility in how you want to use case to make your code look
 - Lots of different opinions on what's best
- We'll show our preferred version but lots of people disagree

White space

- Fortran doesn't use whitespace as a core part of the language
- By convention Fortran code is block indented
 - Loops and conditionals increase the indent level
- Only restriction is that you **HAVE** to use spaces, not tabs
- How many spaces you use is entirely up to you
 - We think that 2 spaces per level is a nice balance of readability and not using too much space.

Function Passing

- When you call a function in a language the values that you give the function have to get into the function somehow
- Two general approaches
 - Pass by value - copy the values into the function. Changing that copy doesn't change the original value (C/C++)
 - Pass by reference - the function is given a reference to the value - changing the value inside the function changes the original value (Fortran)
 - Slight wrinkle if you pass the same variable to two arguments (don't do this)

Compiling Fortran



Compiling

- Fortran is a **compiled** language
 - The source code has to be run through a compiler before it can be executed
- If you are familiar with C/C++ then the compile lines for Fortran are very similar, just change the compiler name
 - gfortran - GNU
 - ifort - Intel
 - Many others but these are the ones we use in SC RTP
- Fortran files have the .f90 extension by convention.
 - .f is also used but is outdated and may not work properly

"As If" compilation

- Compilers always try and make your code faster, by **optimisation**
 - You have a choice of how much optimisation is applied but there is always some (for most codes -O3 is most optimised and -O0 is least optimised)
- Modern compilers perform "as if" compilation
- They produce code that gives the same effects "as if" they had run your code, but otherwise don't have to have any particular connection to your code at all
- This is why language standards are important - if you write code that is not permitted ("undefined behaviour") then the compiler can chose to do literally anything that it likes when it encounters that code!

Compiling

- Single line compilation
 - `gfortran file1.f90 file2.f90 file3.f90 -o output_prog`
 - Files have to be in needed order. If file2.f90 uses stuff from file1.f90 then file1.f90 has to be before file2.f90 in the list
 - Final executable program is "output_prog" and can just be run as `./output_prog`
- Usually want to specify `-O3` as a command line option to the compiler to tell the compiler to optimise the code as much as possible

Compiling

- Multi line compilation
 - gfortran -c file1.f90
gfortran -c file2.f90
gfortran -c file3.f90
gfortran file1.o file2.o file3.o -o output_prog
 - First lines create .o intermediate files from the .f90 files
 - Final line builds “output_prog” again from those files. The .o files don’t need to be in any particular order on the compile line

Hello World

Program “Entry Point”

- All programs start from an entry point
 - Indicates where to start when they run
- For Pythonites, simple scripts just start at line 1
- Simple modules use the `if __name__ == "__main__":` construct
- For C/C++ programmers the entry point is the `main` function

Simplest possible Fortran program

```
PROGRAM boilerplate  
  
END PROGRAM boilerplate
```

- The entry point for Fortran is a PROGRAM block
- You can call your program anything you like
- Has to match in the start and end commands - can be omitted from END

Fortran Hello World

```
PROGRAM hello_world  
  IMPLICIT NONE  
  PRINT *, "Hello World!"  
END PROGRAM hello_world
```

- Same "PROGRAM", "END PROGRAM" pair
- Fortran "PRINT" statement prints to screen
- The "*" means "print using default format"
- Other formats are available

Fortran Hello World

```
PROGRAM hello_world  
  IMPLICIT NONE  
  PRINT *, "Hello World!"  
END PROGRAM hello_world
```

- “IMPLICIT NONE” turns off an old and nasty Fortran feature that allows variables to be created implicitly
- Means that if you forget to define the type of a variable it will still exist with a type depending on the first letter of its name
 - NOT USEFUL!

Variable creation

Types

- Fortran is a **strongly compile time typed** language
- Every variable has a type that is known when the code is compiled
- Every function has to take a list of parameters that have known types at compile time
- If functions return a type that also has to be known at compile time
- Often a fair chunk of a code is made up of type declarations

Declarations

- Fortran variables **must** be declared and given a **type** before they are used (once you have turned off implicit variables)
- In Fortran all declarations have to come before any other code in a function (not quite true but you won't often encounter the other bits)
- This can be feel a bit restrictive but it does help make code substantially more readable

Fortran variables

- Fortran basic types are
 - CHARACTER - A default character (as in a string, unlike C you can't use Fortran CHARACTER as a very short integer)
 - Mostly try to avoid string handling in Fortran, not the language's strength
 - LOGICAL - A true or false variable type
 - INTEGER - A default integer
 - REAL - A default floating point number
 - COMPLEX - A default complex number

Fortran variables

```
PROGRAM variables
  IMPLICIT NONE
  CHARACTER(LEN=20) :: mystring
  INTEGER, PARAMETER :: myint = 10
  REAL :: myfloat1, myfloat2
  COMPLEX :: mycomplex
END PROGRAM variables
```

- Type name
- Optionally parameters to the type in round brackets
- Optionally comma separated attributes (such as PARAMETER)
- Double colon
- Comma separated list of names

Fortran variables attributes

- All variables in Fortran can also have additional attributes added to them
- You put them as part of a comma separated list before the "::" in the variable declaration
- You'll meet quite a few of them, but the simplest is **PARAMETER**
 - **PARAMETER** means that you are assigning this variable a value now and you can't change it when the code runs
 - Have to give a value to the variable on the line where you create it

Loops

Fortran DO loop

```
PROGRAM simple_loop

  IMPLICIT NONE
  INTEGER :: loop

  DO loop = 0, 9
    PRINT *, "Hello from loop ", loop
  END DO

END PROGRAM simple_loop
```

- Fortran loops are most commonly DO loops
- DO variable = start, end, {stride}
- ...
END DO
- Loop variable will run from start to end inclusive

Fortran DO loop

- Loop runs from the start value until the end value is passed
- Optionally, can specify another **stride** value to specify how to increment the loop variable every iteration
- IMPORTANT NOTES
 - You can't change the loop variable inside the loop
 - Can't change the end value of a loop while the loop is running
 - Loop variable has to be an integer

Fortran DO WHILE loop

```
PROGRAM simple_loop
  IMPLICIT NONE
  INTEGER :: loop

  loop = 0
  DO WHILE(loop <= 9)
    PRINT *, "Hello from loop ", loop
    loop = loop + 1
  END DO
END PROGRAM simple_loop
```

- DO WHILE gives you more control
- Loop iterates so long as the condition in the WHILE function is .TRUE.
- WHILE is checked fresh at the start of every iteration so you can change the termination condition

Loops

- Loops have additional commands that allow you to exit a loop early or have it immediately go on to the next iteration of the loop
- EXIT - Leave the loop that you are currently in immediately. Do not execute any other commands in the loop. Do not collect £200
- CYCLE - Immediately exit this iteration of the loop and start the next iteration. If this is the last iteration of the loop exit the loop
- They have their place but they can make code hard to read. If you can set your loop condition to avoid using EXIT in particular you probably should

Conditionals

A solid blue geometric shape is located in the bottom right corner of the slide. It consists of a horizontal line segment on the left, followed by a diagonal line segment sloping down to the right, then a diagonal line segment sloping up to the right, and finally a diagonal line segment sloping down to the right, ending in a horizontal line segment on the right.

Fortran IF syntax

```
PROGRAM simple_if
```

```
  IMPLICIT NONE
```

```
  INTEGER :: test, candidate
```

```
  test = 2
```

```
  candidate = 4
```

```
  IF (test /= candidate) PRINT *, 'Test ', test, &  
    ' not equal to candidate ', candidate
```

```
END PROGRAM simple_if
```

Fortran IF syntax

- $a == b$ - test for a exactly equal to b
- $a /= b$ - test for a not exactly equal to b
- $a < b$ - test for a less than b
- $a > b$ - test for a greater than b
- $a <= b$ - test for a less than or equal to b
- $a >= b$ - test for a greater than or equal to b
- Remember that testing for exact equality of real numbers is very risky
 - Usually want $ABS(a-b) < threshold$

Fortran IF syntax

```
PROGRAM block_if
```

```
  IMPLICIT NONE
```

```
  INTEGER :: test, candidate
```

```
  test = 2
```

```
  candidate = 4
```

```
  IF (test == candidate) THEN
```

```
    PRINT *, 'Test ', test, ' equal to candidate ', candidate
```

```
  ELSE
```

```
    PRINT *, 'Test ', test, ' not equal to candidate ', &  
      candidate
```

```
  END IF
```

```
END PROGRAM block_if
```

Fortran IF syntax

- Can have IF attached to an ELSE statement
- IF (condition) THEN
ELSE IF (condition2)
END IF
- More complex expressions should use **SELECT** rather than a chain of if's
- Usually called **CASE** or **SWITCH** statements in other languages

Fortran IF syntax

```
PROGRAM simple_if
```

```
  IMPLICIT NONE
```

```
  INTEGER :: test, candidate
```

```
  test = 2
```

```
  candidate = 4
```

```
  IF (test < candidate) THEN
```

```
    PRINT *, 'Test ', test, ' less than candidate ', candidate
```

```
  ELSE IF (test == candidate) THEN
```

```
    PRINT *, 'Test ', test, ' equal to candidate ', candidate
```

```
  ELSE
```

```
    PRINT *, 'Test ', test, ' greater than candidate ',
```

```
    candidate
```

```
  END IF
```

```
END PROGRAM simple_if
```

Fortran Logical Types

- All Boolean operations return a **LOGICAL** that represents a true/false value
- So **(a==b)** always is of type **LOGICAL** whatever the types of **a** and **b**
- **LOGICAL** constants are **.TRUE.** and **.FALSE.**
- Logicals have new operations

Fortran Logical Types

- **.NOT. A** - Invert truth (.TRUE. <-> .FALSE.)
- **A .OR. B** - Returns .TRUE. if either A or B is .TRUE.
- **A .AND. B** - Returns .TRUE. if A and B are both .TRUE.
- **A .EQV. B** - Returns .TRUE. if A and B are both in the same state
- **A .NEQV. B** - Returns .TRUE. if A and B are in different states.
For those familiar with Boolean operations, note that this is XOR under a different name
 - A lot of compilers support .XOR. but it isn't in the standard

Fortran Logical Combination

- **Remember that you can only combine logicals with the logical operators**
- So IF (a > b .AND. a < c) PRINT *, 'Condition' is fine
- IF (a>b .AND. < c) PRINT *, 'Condition' won't work

Logical Short Circuiting

- Many languages have defined **logical short circuiting** behaviour but Fortran doesn't
- In a short circuiting language a condition **`if (val1 and val2 and val3)`** will stop evaluating when it encounters the first **false** value in that chain because it *knows* that the condition must be false
- Same for other logical operators, it will stop as soon as it knows the answer

Logical Short Circuiting

- The Fortran standard actually doesn't specify any particular short circuiting behaviour (yet?) so compilers may or may not short circuit
- Only matters under certain conditions
- When it matters you have to write code that can cope with either behaviour since both short circuiting and non short circuiting behaviours exist in common compilers

Modules

Program Segmentation

- It is useful to split your code up into chunks that are logically connected in some way
 - cf Java packages or Python modules (imports)
- In Fortran the concept is that of a **Module**
- Modules contain variables and functions/subroutines
 - You can put variables and functions/subroutines in other places but they are much less common and often hang-overs from FORTRAN 77

Simple Module

```
MODULE mymodule
```

```
  IMPLICIT NONE  
  SAVE
```

```
  INTEGER :: module_int  
  INTEGER, PARAMETER :: module_parameter = 10  
  REAL :: module_real
```

```
END MODULE mymodule
```

- MODULE {name} and END MODULE {name}
- Name is used when you access the module so must be unique and should be memorable
- IMPLICIT NONE again - Always put in every module

Simple Module

```
MODULE mymodule
```

```
    IMPLICIT NONE  
    SAVE
```

```
    INTEGER :: module_int  
    INTEGER, PARAMETER :: module_parameter = 10  
    REAL :: module_real
```

```
END MODULE mymodule
```

- SAVE - Keep the value of variables defined in this module for as long as the program runs (mostly not needed in practice but strictly required per standard)
- Define variables same as in the PROGRAM

USEing Modules

```
MODULE mymodule

    IMPLICIT NONE
    SAVE

    INTEGER :: module_int
    INTEGER, PARAMETER :: module_parameter = 10
    REAL :: module_real


END MODULE mymodule

PROGRAM driver

    USE mymodule
    IMPLICIT NONE
    INTEGER :: myint

    PRINT *, module_parameter
    myint = module_parameter
    module_int = module_parameter
    module_real = REAL(module_int)

END PROGRAM driver
```



USEing Modules

- To get access to the contents of a module you **USE** the module by name
- You can **USE** modules in **programs** and other **modules** (technically also in subroutines/functions but that's rarely done)
- Makes everything in the USEd module available at the level where it was USEd **and all levels below it**
- USE statements go before IMPLICIT NONE and hence before variable declarations

USEing Modules

- Once a module is USEd the name of everything in the module are mixed with the names of all the other USEd modules and the names in the place where the USE statement occurs
 - If you know C++ this is “using namespace”, in Python this behaviour would be “import * from module”
 - There is no equivalent of the “module.item” or “namespace::item” Python or C++ syntax
 - This is coming in an upcoming Fortran standard almost certainly (Not Fortran 2023, the next one)
- Old school approaches of modname_functionname names do work if you really need to avoid collisions
 - This can be annoying because you have to remember/type more

USEing Modules

- Names of variables, functions and subroutines must be unique within the scope of all USEd modules
- You can have two functions called "test_object" in different modules so long as you don't USE both of the modules in the same place
- If this happens there are ways round it (see the formal definition of the USE command for USE, ONLY or USE, {*rename=>to*})
- Might be a sign that your design is a bit messy

USEing Modules

- USE is effectively recursive
- If you USE module B that itself USEs module A you have effectively USEd module A as well
 - Slight caveats to do with public/private declarations
- This includes all of the problems with name collisions so be careful with your USE chains
- Some people say that you should always USE, ONLY and only import what you actually use but this can get very annoying if you use most of a module

Functions and Subroutines

A decorative blue zigzag shape is located in the bottom right corner of the slide, extending from the dark blue header area into the white footer area.

Subroutines and Functions

- In most languages there are **functions**.
 - Parts of the code that take arguments operate on them and may or may not return values
- Fortran makes a distinction between
 - **FUNCTIONs** that return a value
 - **SUBROUTINEs** that do not return a value
- Collectively they are called **SUBPROGRAMS** in the Fortran documentation

Pass by Reference

- Fortran passes variables to subprograms *by reference*
- That means that the arguments that you have inside the subprogram **are exactly the same variables that you called the subprogram with, NOT copies of them**
- in C/C++ they are copies unless explicitly made pointers or references
- in Python they are also copies but **mutable** containers when copied hold the **same** items so you can change the items but not the containers

Simple Module

```
MODULE mymodule

    IMPLICIT NONE
    SAVE

    INTEGER :: module_int
    CONTAINS

    SUBROUTINE hello_world_sub()
        PRINT *, "Hello World"
        PRINT *, "Module_int is", module_int
    END SUBROUTINE hello_world_sub

END MODULE mymodule

PROGRAM driver
    USE mymodule
    IMPLICIT NONE

    module_int = 1234
    CALL hello_world_sub()
END PROGRAM driver
```

Introducing SUBROUTINE

- Subroutines and Functions are normally defined in Modules in modern Fortran
- They are separated from variable declarations by the "**CONTAINS**" keyword
- Modules should not have the "**SAVE**" keyword if they contain no variables and should not have the "**CONTAINS**" keyword if they contain no functions or subroutines

Subroutine arguments

```
SUBROUTINE hello_world_sub(arg_int)
  INTEGER :: arg_int
  PRINT *, "Hello World"
  PRINT *, "Module_int is", module_int
  PRINT *, "arg_int is ", arg_int
END SUBROUTINE hello_world_sub
```

- Arguments are passed to a subroutine by specifying their names in the brackets after the SUBROUTINE statement
- The type of the argument is then specified as a variable within the subroutine, just like any other variable
- There are some things that you can do with a argument that you can't do with a normal variable and vice versa

Arguments

- Inside a function/subroutine the arguments passed to the function are technically called “dummy arguments”
- Because they are not “real” variables but are references to the “actual arguments” that are provided then you **CALL** the function
- Only Fortran uses “dummy” as a description, in general computer science terminology they are called “formal arguments”
- **Important reminder! The dummy arguments inside the function are exactly the same as the actual arguments passed to the call! Changing one changes the other!**

Terminology

```
MODULE mymodule
```

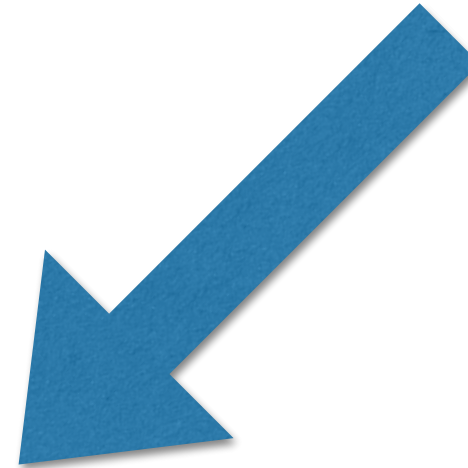
```
CONTAINS
```

```
  SUBROUTINE hello_world_sub(arg_int)
    INTEGER :: arg_int
    PRINT *, "arg_int is ", arg_int
  END SUBROUTINE hello_world_sub
```

```
END MODULE mymodule
```

```
PROGRAM driver
  USE mymodule
  IMPLICIT NONE
```

```
  CALL hello_world_sub(1234)
END PROGRAM driver
```



- Dummy argument
- Placeholder name for the data passed into the function

Terminology

```
MODULE mymodule
```

```
CONTAINS
```

```
  SUBROUTINE hello_world_sub(arg_int)  
    INTEGER :: arg_int  
    PRINT *, "arg_int is ", arg_int  
  END SUBROUTINE hello_world_sub
```

```
END MODULE mymodule
```

```
PROGRAM driver  
  USE mymodule  
  IMPLICIT NONE
```

```
  CALL hello_world_sub(1234)  
END PROGRAM driver
```

- Actual argument
- The value that will be available inside the function when it is called



Multiple arguments

```
MODULE mymodule
```

```
CONTAINS
```

```
  SUBROUTINE hello_world_sub(arg1, arg2)
```

```
    INTEGER :: arg1, arg2
```

```
    PRINT *, "arg1 is ", arg1
```

```
    PRINT *, "arg2 is ", arg2
```

```
  END SUBROUTINE hello_world_sub
```

```
END MODULE mymodule
```

```
PROGRAM driver
```

```
  USE mymodule
```

```
  IMPLICIT NONE
```

```
  CALL hello_world_sub(1234, 5678)
```

```
END PROGRAM driver
```

Intent

- The **INTENT** statement tells the compiler what you are intending to do with a dummy variable
 - INTENT(IN) - I want to use the value that this variable has when my subroutine is called inside my subroutine. This is the **only** intent that allows passing a literal value or the result of another function
 - INTENT(OUT) - I want to set the value of this variable. I neither want nor have access to it's value at the calling point of my subroutine
 - INTENT(INOUT) - I want to both know the value of this variable when the subroutine is called and change the value. Almost but not quite the same as not specifying intent

Introducing FUNCTION

- The only differences between a function and a subroutine are
 - Rather than starting and ending a **SUBROUTINE**, you start and end a **FUNCTION**
 - Functions return values
 - Rather than calling a function using CALL you "capture" the return value by putting it into a variable using "="

Function Returns

```
FUNCTION simple_func()  
  INTEGER :: simple_func  
  simple_func = 10  
END FUNCTION simple_func
```

- In Fortran you don't call a RETURN intrinsic with an argument to return a value like you do in a lot of languages
 - Note that there is a RETURN intrinsic in Fortran to leave a function and it can (sometimes) take parameters so you might be able to get this wrong and have your code compile! It just won't do what you want
- You create a variable with the same name as the function to define the return type of the function
- And then you set that variable to have the return value for the function

Function Returns

```
FUNCTION simple_func()  
    INTEGER :: simple_func  
    simple_func = 10  
END FUNCTION simple_func
```

```
FUNCTION simple_func() RESULT(val)  
    INTEGER :: val  
    val = 10  
END FUNCTION simple_func
```

```
INTEGER FUNCTION simple_func()  
    simple_func = 10  
END FUNCTION simple_func
```

Function Arguments

```
MODULE mymodule
```

```
CONTAINS
```

```
FUNCTION diff(param1, param2)
  INTEGER, INTENT(IN) :: param1
  INTEGER, INTENT(IN) :: param2
  INTEGER :: diff
  diff = param1 - param2
END FUNCTION diff
```

```
END MODULE mymodule
```

```
PROGRAM driver
```

```
USE mymodule
```

```
IMPLICIT NONE
```

```
INTEGER :: a, b
```

```
b = 5678
```

```
a = diff(1234,b)
```

```
PRINT *, 'A is ', a
```

```
END PROGRAM driver
```


Calling with keywords

```
PROGRAM driver
  USE mymodule
  IMPLICIT NONE
  INTEGER :: a, b

  b = 5678
  a = diff(param1 = 1234, param2 = b)

  PRINT *, 'A is ', a
END PROGRAM driver
```

Calling with keywords

```
PROGRAM driver
  USE mymodule
  IMPLICIT NONE
  INTEGER :: a, b

  b = 5678
  a = diff(param2 = b, param1 = 1234)

  PRINT *, 'A is ', a
END PROGRAM driver
```

Subprogram prefixes

- Subprograms can have extra attributes specified as prefixes before the FUNCTION/SUBROUTINE declaration. If there are more than one they are separated by spaces
 - Return type (already seen)
 - PURE - Has no **side effects**
 - ELEMENTAL - Acts on arrays element by element (must be **PURE** until F2008)
 - RECURSIVE - Subprogram can call itself
 - IMPURE (F2008) - Opposite of pure

Arrays

- Idea of an array is very simple
 - Pack together a set of N items of the same type and allow access to each individual element by a numerical index, c.f a vector of elements
 - Arrays are guaranteed to be contiguous in the underlying memory
- Are also higher **rank** arrays (2D, 3D etc.) but we'll start with simple 1D arrays
- You can declare an array anywhere you can declare a normal variable

Arrays

```
PROGRAM array_test
  IMPLICIT NONE
  !Array runs from 1->10
  !Different to most other languages
  !that run 0->9
  INTEGER, DIMENSION(10) :: array
  INTEGER :: i

  !SIZE is an intrinsic function that
  !tells me how large my array is
  DO i = 1, SIZE(array)
    array(i) = i
  END DO

  !Can print whole arrays although large
  !arrays are hard to read
  PRINT *, array

END PROGRAM array_test
```

Arrays

- Creating a fixed dimension array is very simple
 - Give your variable the **DIMENSION** attribute
 - Inside the brackets of the **DIMENSION** attribute specify the number of elements in the array
- Previous example showed how to set the elements of the array to the numbers 1,2,3, ... using a loop
- There are other ways

Array literals

```
PROGRAM array_test
  IMPLICIT NONE
  !Array runs from 1->10
  !Different to most other languages
  !that run 0->9
  INTEGER, DIMENSION(10) :: array
  INTEGER :: i

  !Can also use an array literal
  !Surround a comma separated list of
  !numbers with square brackets
  array = [1,2,3,4,5,6,7,8,9,10]

  !Alternate (older) style of array literal uses (/ /)
  array = (/1,2,3,4,5,6,7,8,9,10/)

  !Can print whole arrays although large
  !arrays are hard to read
  PRINT *, array

END PROGRAM array_test
```

Implied DO loops

```
PROGRAM array_test
  IMPLICIT NONE
  !Array runs from 1->10
  !Different to most other languages
  !that run 0->9
  INTEGER, DIMENSION(10) :: array
  INTEGER :: i

  !Can use an IMPLIED DO LOOP
  !to set up this array (c.f. linspace etc.)
  array = [(i,i=1,SIZE(array))]

  !Can print whole arrays although large
  !arrays are hard to read
  PRINT *, array

END PROGRAM array_test
```


Arrays

- By default Fortran arrays run from 1 to the number of elements in the array
- Most other languages tend to run from 0 to $n - 1$
- Unlike most other languages you can freely change this by specifying (lower_bound:upper_bound) in **DIMENSION**

Arrays

```
PROGRAM array_test
  IMPLICIT NONE
  !Runs 1->10
  INTEGER, DIMENSION(10) :: array1
  !Runs 1->10 but now made explicit
  INTEGER, DIMENSION(1:10) :: array2
  !Runs 0->9
  INTEGER, DIMENSION(0:9) :: array3
  !Runs -5->4
  INTEGER, DIMENSION(-5:4) :: array4

  PRINT *, 'Array1 :', SIZE(array1), LBOUND(array1), UBOUND(array1)
  PRINT *, 'Array2 :', SIZE(array2), LBOUND(array2), UBOUND(array2)
  PRINT *, 'Array3 :', SIZE(array3), LBOUND(array3), UBOUND(array3)
  PRINT *, 'Array4 :', SIZE(array4), LBOUND(array4), UBOUND(array4)

END PROGRAM array_test
```

Arrays

```
PROGRAM array_test
  IMPLICIT NONE
  !Runs 1->10
  INTEGER, DIMENSION(10) :: array1
  !Runs 1->10 but now made explicit
  INTEGER, DIMENSION(1:10) :: array2
  !Runs 0->9
  INTEGER, DIMENSION(0:9) :: array3
  !Runs -5->4
  INTEGER, DIMENSION(-5:4) :: array4

  PRINT *, 'Array1 :', SIZE(array1), LBOUND(array1), UBOUND(array1)
  PRINT *, 'Array2 :', SIZE(array2), LBOUND(array2), UBOUND(array2)
  PRINT *, 'Array3 :', SIZE(array3), LBOUND(array3), UBOUND(array3)
  PRINT *, 'Array4 :', SIZE(array4), LBOUND(array4), UBOUND(array4)

END PROGRAM array_test
```

Array1 :	10	1	10
Array2 :	10	1	10
Array3 :	10	0	9
Array4 :	10	-5	4

Arrays

- Note the intrinsic functions
 - SIZE - Total number of elements in an array
 - LBOUND - Lower bound of array, technically returns an array!
 - UBOUND - Upper bound of array, technically returns an array!

Arrays

- You've seen that you can use numerical literal values to specify array bounds, but what else can you use?
- Literal expressions (i.e. $1+10$ is fine)
- You can use INTEGER, PARAMETER variables to define bounds and expressions involving them
- You can use INTEGER dummy variables to a function to define bounds
- **ABSOLUTELY NOT normal variables or expressions containing normal variables**

Allocatable Arrays

- You can have arrays that change size depending on normal variables and they are called **Allocatable Arrays**
 - Technical reasons for why they are different but we won't go into that
- Another attribute is added to the variable declaration
 - **ALLOCATABLE**
 - Also put a ":" into the **DIMENSION** attribute to indicate that the size isn't known yet

Allocatable Arrays

- There are then two intrinsic functions that allocate and deallocate memory for allocatable arrays
 - **ALLOCATE(array_name(range_spec))**
 - **DEALLOCATE(array_name)**
- **range_spec** can be anything that is valid for a constant size array but can use normal variables
 - So you can specify upper and lower bounds etc
 - **range_spec** is evaluated when the ALLOCATE command happens

Allocatable Arrays

```
PROGRAM array_test
  IMPLICIT NONE
  INTEGER, DIMENSION(:), ALLOCATABLE :: array
  INTEGER :: array_count

  array_count = 10
  ALLOCATE(array(array_count))
  PRINT *, 'V1 :', SIZE(array), LBOUND(array), UBOUND(array)
  DEALLOCATE(array)
  ALLOCATE(array(0:array_count-1))
  PRINT *, 'V2 :', SIZE(array), LBOUND(array), UBOUND(array)
  DEALLOCATE(array)

END PROGRAM array_test
```

V1 :	10	1	10
V2 :	10	0	9

Allocatable Arrays

- Once an array is allocated you have to deallocate it before you can allocate it again
- This is **good** because most compilers will detect this so it means that you can't leak memory by accidentally reallocating memory without freeing it
- Allocatable arrays defined in functions are deallocated when you leave the function (unless they have the **SAVE** attribute (see later for this))
- Different rules apply for arrays that are passed into a function as an argument

Allocatable Arrays

- Allocatable arrays have an “allocated” property that you can check with the “**ALLOCATED**” function
- Before you allocate an array is not allocated
- After you allocate it it is allocated
- After you deallocate it it is not allocated
- Lots of array functions like SIZE etc. will give strange answers on unallocated arrays

Allocatable Arrays

```
PROGRAM alloc_test

    INTEGER, DIMENSION(:), ALLOCATABLE :: array

    PRINT *, 'Before allocation ', ALLOCATED(array)
    ALLOCATE(array(10))
    PRINT *, 'After allocation ', ALLOCATED(array)
    DEALLOCATE(array)
    PRINT *, 'After deallocation ', ALLOCATED(array)

END PROGRAM alloc_test
```

```
Before allocation  F
After allocation   T
After deallocation F
```

Implicit Allocation

- When an allocatable array has another array assigned to it with **=** it will reallocate itself to the size of that array
- This is why **array = [array,new_element]** that we introduced at the start works, so long as **array** is an allocatable array
- You create a new element containing all of the existing elements of the array and then a new element, and then assign it back to the array which is reallocated to be size of the newly formed array
- This only applies to assignment to the whole array, assigning to a **subsection** (see later) doesn't change the array's size (even if the subsection is the whole array)

Arrays and Subprograms

- You pass an array into a subprogram by specifying an array type variable as a dummy argument to the subprogram in the normal way
 - Just add the **DIMENSION** attribute to the dummy variable
 - Even if you are working with **ALLOCATABLE** arrays you only **have** to flag the dummy variable in a subprogram with **ALLOCATABLE** if you want to allocate or deallocate the array inside the subprogram
 - There are other reasons that you might want to do that

Arrays and Subprograms

```
MODULE arraymod
CONTAINS
SUBROUTINE test_array(array_in)
    INTEGER, DIMENSION(10), INTENT(IN) :: array_in

    PRINT *, 'Array is : ', SIZE(array_in), &
        LBOUND(array_in), UBOUND(array_in)

END SUBROUTINE test_array
END MODULE arraymod
```

```
PROGRAM test

    USE arraymod
    IMPLICIT NONE

    INTEGER, DIMENSION(10) :: array
    CALL test_array(array)

END PROGRAM test
```

```
Array is :           10           1           10
```

Arrays and Subprograms

- That works fine for arrays where you know the size when you are writing the code but not if you are working with allocatable arrays where you don't know the size until the code runs
- You can write functions that take arguments that are called "deferred shape arrays" that take on any size that they need to when the code runs
 - Sounds fancy, but just replace the array size in the **DIMENSION** statement with a ":" again
- You can still use the **SIZE** etc. functions to find out how large they are

Arrays and Subprograms

```
MODULE arraymod
CONTAINS
SUBROUTINE test_array(array_in)
    INTEGER, DIMENSION(:), INTENT(IN) :: array_in

    PRINT *, 'Array is : ', SIZE(array_in), &
        LBOUND(array_in), UBOUND(array_in)

END SUBROUTINE test_array
END MODULE arraymod

PROGRAM test

    USE arraymod
    IMPLICIT NONE

    INTEGER, DIMENSION(10) :: array
    CALL test_array(array)

END PROGRAM test
```


Bounds problems in subprograms

- There is one really, really annoying feature of Fortran arrays that is a hold over from FORTRAN 77
- When you pass an array into a subprogram it's lower bound always maps back to one
- **JUST THE NUMBERING**
- All of your array is still there but the lowest element is now accessed through the value 1 rather than the previous lower bound

Bounds problems in subprograms

- You can change this behaviour by
 - You explicitly set the lower bound in declaration of the dummy variable to the subprogram
 - You give the dummy variable either the **ALLOCATABLE** or **POINTER** attribute (we cover **POINTER** in the extra materials)

Arrays and Subprograms

```
MODULE arraymod
CONTAINS
SUBROUTINE test_array(array_in)
    INTEGER, DIMENSION(10), INTENT(IN) :: array_in

    PRINT *, 'Array is : ', SIZE(array_in), &
        LBOUND(array_in), UBOUND(array_in)

END SUBROUTINE test_array
END MODULE arraymod

PROGRAM test

    USE arraymod
    IMPLICIT NONE

    INTEGER, DIMENSION(0:9) :: array
    CALL test_array(array)

END PROGRAM test
```

Arrays and Subprograms

Array is : 10 1 10

- Bounds of the array as specified are ignored!
- The array is mapped back to having a lower bound of 1

Arrays and Subprograms

```
MODULE arraymod
CONTAINS
SUBROUTINE test_array(array_in)
    INTEGER, DIMENSION(0:9), INTENT(IN) :: array_in

    PRINT *, 'Array is : ', SIZE(array_in), &
        LBOUND(array_in), UBOUND(array_in)

END SUBROUTINE test_array
END MODULE arraymod

PROGRAM test

    USE arraymod
    IMPLICIT NONE

    INTEGER, DIMENSION(0:9) :: array
    CALL test_array(array)

END PROGRAM test
```

Arrays and Subprograms

```
MODULE arraymod
CONTAINS
SUBROUTINE test_array(array_in)
    INTEGER, DIMENSION(0:9), INTENT(IN) :: array_in

    PRINT *, 'Array is : ', array_in, &
        LBOUND(array_in), UBOUND(array_in)

END SUBROUTINE test_array
END MODULE arraymod
```

```
PROGRAM test

    USE arraymod
    IMPLICIT NONE

    INTEGER, DIMENSION(0:9) :: array
    CALL test_array(array)

END PROGRAM test
```

```
Array is :          10          0          9
```

Arrays and Subprograms

```
MODULE arraymod
CONTAINS
SUBROUTINE test_array(lower, array_in)
    INTEGER, INTENT(IN) :: lower
    INTEGER, DIMENSION(lower:lower+9), INTENT(IN) :: array_in

    PRINT *, 'Array is : ', SIZE(array_in), &
        LBOUND(array_in), UBOUND(array_in)

END SUBROUTINE test_array
END MODULE arraymod

PROGRAM test

    USE arraymod
    IMPLICIT NONE

    INTEGER, DIMENSION(0:9) :: array
    CALL test_array(0, array)

END PROGRAM test
```

Arrays and Subprograms

- What about deferred shape arrays?
- Still have the lower bound problem
- You can still do the same tricks to reset the lower bounds but now you just don't specify the upper bound at all so the call looks like
 - **INTEGER, DIMENSION(lower:)**

Arrays and Subprograms

```
MODULE arraymod
CONTAINS
SUBROUTINE test_array(lower, array_in)
    INTEGER, INTENT(IN) :: lower
    INTEGER, DIMENSION(lower:), INTENT(IN) :: array_in

    PRINT *, 'Array is : ', SIZE(array_in), &
        LBOUND(array_in), UBOUND(array_in)

END SUBROUTINE test_array
END MODULE arraymod

PROGRAM test

    USE arraymod
    IMPLICIT NONE

    INTEGER, DIMENSION(0:9) :: array
    CALL test_array(0, array)

END PROGRAM test
```

Arrays and Subprograms

- So I mentioned that if you specify the **ALLOCATABLE** attribute to the dummy variable to a function then the bounds will be correctly passed through to the function
- This also allows you to `ALLOCATE` and `DEALLOCATE` the variable inside the function (unless it is specified `INTENT(IN)` because then you can't change it at all)
- Problem is that the function will then **ONLY** work with allocatable variables

Arrays and Subprograms

```
MODULE arraymod
CONTAINS
SUBROUTINE test_array(array_in)
    INTEGER, DIMENSION(:), ALLOCATABLE, INTENT(IN) :: array_in

    PRINT *, 'Array is : ', SIZE(array_in), &
        LBOUND(array_in), UBOUND(array_in)

END SUBROUTINE test_array
END MODULE arraymod

PROGRAM test

    USE arraymod
    IMPLICIT NONE

    INTEGER, DIMENSION(:), ALLOCATABLE :: array
    ALLOCATE(array(0:9))
    CALL test_array(array)

END PROGRAM test
```

Arrays and Subprograms

```
MODULE arraymod
CONTAINS
SUBROUTINE test_array(array_in)
    INTEGER, DIMENSION(:), ALLOCATABLE, INTENT(IN) :: array_in

    PRINT *, 'Array is : ', SIZE(array_in), &
        LBOUND(array_in), UBOUND(array_in)

END SUBROUTINE test_array
END MODULE arraymod
```

```
PROGRAM test

    USE arraymod
    IMPLICIT NONE

    INTEGER, DIMENSION(:), ALLOCATABLE :: array
    ALLOCATE(array(0:9))
    CALL test_array(array)

END PROGRAM test
```

```
Array is :           10           0           9
```

Arrays and Subprograms

- There's also one final wrinkle to putting **ALLOCATABLE** as an attribute for a dummy variable
- If you specify that the variable is **INTENT(OUT)** then it assumes that you don't want any information from outside ***so it will deallocate the array that you pass in***
- While you can see why that was the decision that the standards body came to it isn't ***obvious*** that this will happen so watch out for it

Arrays and Subprograms

- There isn't any good solution to the lower bound problem of passing arrays to subprograms in Fortran
- I tend to favour
 - setting the lower bound to be a constant if it's the same for all arrays that will be passed to the function
 - passing the extra parameter when it isn't
- I try to avoid writing functions that "cope" with changing the lower bound to 1 since it is often confusing and inelegant
- One popular solution is to encapsulate the array in a **derived type** (see later) and pass the type to your subprogram

Multidimensional Arrays

- Fortran is one of comparatively few languages that have multidimensional arrays as a core language feature
- Generally the number of dimensions that an array has is called its **rank** and this term is very commonly used in Fortran
- So the arrays that we have been working with are **rank 1** arrays
- 2D arrays are **rank 2** etc.

Multidimensional Arrays

```
PROGRAM ranktest
```

```
    INTEGER, DIMENSION(0:9, -1:10) :: array1  
    INTEGER, DIMENSION(:, :), ALLOCATABLE :: array2
```

```
    ALLOCATE(array2(-5:4, 1:10))  
    array2(3, 7) = 4
```

```
END PROGRAM ranktest
```

- Syntax for higher rank arrays is very similar to rank 1 arrays
- Just comma separate the bounds and the indices and off you go!
- Access is (a, b) rather than multiple brackets (c.f. C)
 - Any valid range etc. is valid for each rank separately (e.g array(3, :))

Multidimensional Arrays

- Fortran up to Fortran 2008 has a maximum rank of 7
 - Fortran 2008 increased this to 15 but compiler support is still patchy for this although it is improving
- Mostly 7D arrays (i.e. array (a, b, c, d, e, f, g)) will be enough
- Higher rank arrays are passed to functions in exactly the same way as rank 1 arrays
 - Just create a dummy variable **matching the rank** of the array that you want to pass in

Multidimensional Arrays

- Functions in Fortran can only be written to take arrays of a specific rank
 - Until Fortran 2018 but at the moment it is hard to use
 - Fortran 2023 is adding more tools for doing this
- Usually simpler anyway because writing a generic routine that works for any rank of array but performs well is very hard
- If you really need to cope with this then using a 1D array with an indexing function to “mock up” higher ranks is the easiest solution

Multidimensional Intrinsics

- The intrinsic array functions change slightly for higher rank arrays
 - SIZE - Returns the total number of elements in the array
 - LBOUND - Returns an array of the lower bounds in each rank. Optional **DIM** integer parameter to specify which rank you want the lower bound for. If **DIM** is specified then returns a simple INTEGER
 - UBOUND - See LBOUND
 - SHAPE - Returns an array of the number of elements in each rank of the array

Array bound checking

- Most Fortran compilers can turn on *array bounds checking* that will check if you are accessing any rank outside of the specified bounds
- Slows down the code so not on by default, usually "-C" compiler option to turn it on
- Tools like Valgrind can detect accesses that are completely outside of an array
- Can't detect just overrunning one rank, only accessing outside array

Whole Array Operations

- In Fortran most intrinsic operations and functions can be applied to whole arrays
 - They apply element by element
- You can validly add two arrays, multiply two arrays or take the sine of an array etc.
- Arrays must be the same size and rank for these to work (excepting automatic reallocation!)
 - Although you can do "array = scalar" to set all the array values to the single scalar value

Whole Array Operations

```
PROGRAM whole_array
```

```
REAL, PARAMETER :: pi = 3.14159
```

```
REAL, DIMENSION(10) :: a, b
```

```
INTEGER :: i
```

```
!Set a so that the elements go pi/2, pi, 3pi/2 etc
```

```
a=[(pi*REAL(i)/2.0,i=1,SIZE(a))]
```

```
b = SIN(a)
```

```
PRINT *, b
```

```
END PROGRAM whole_array
```

1.000000000	2.53518169E-06	-1.000000000	-5.07036339E-06
1.000000000	7.60554485E-06	-1.000000000	-1.01407268E-05
1.000000000	1.26759078E-05		

Whole Array Operations

```
ELEMENTAL SUBROUTINE double(a)
  !NOTE this function takes a single scalar NOT an array
  INTEGER, INTENT(INOUT) :: a
  a = a * 2
END SUBROUTINE double
```

- Some functions treat arrays like vector or matrices such as **DOT_PRODUCT** or **MATMUL**
- If you want to create your own functions to operate on arrays element by element then use **ELEMENTAL** subprograms
- Rules in general are quite complicated but the idea is simple

Array subsections

- Can also refer to subsections of an array
- Use `array(lbound:ubound)` to refer to an array subsection
- Same for multiple rank arrays, can put in a range or a single value for any index that you like
- Can use array subsections exactly like you can arrays
- Set them equal to things, pass them to functions, everything you can do with an array you can do with an array subsection

Array subsections

- If you want an array section to go to the lower bound, simply don't specify the lower index
 - `array(:10)`
- If you want an array section to go to the upper bound, simply don't specify the upper index
 - `array(10:)`
- You can validly specify the whole array using just **(:)** although there are only a few reasons to (preventing implicit reallocation is one!)
- Less commonly seen you can also specify a stride
 - `array(1:100:5)` specifies every 5th element of the array from 1 to 100

MOVE_ALLOC

```
PROGRAM move_test
```

```
  INTEGER, DIMENSION(:), ALLOCATABLE :: a, b  
  INTEGER :: i
```

```
  ALLOCATE(a(-5:10))  
  DO i = -5, 10  
    a(i) = i  
  END DO
```

```
  CALL MOVE_ALLOC(FROM = a, TO = b)  
  PRINT *, LBOUND(b), UBOUND(b)  
  PRINT *, MINVAL(b), MAXVAL(b)  
  PRINT *, ALLOCATED(a), ALLOCATED(b)
```

```
END PROGRAM move_test
```

- Move data between allocatable arrays
- TO must start deallocated
- FROM becomes deallocated
- Faster than setting = because memory is **moved** not **copied**