Intro to Fortran

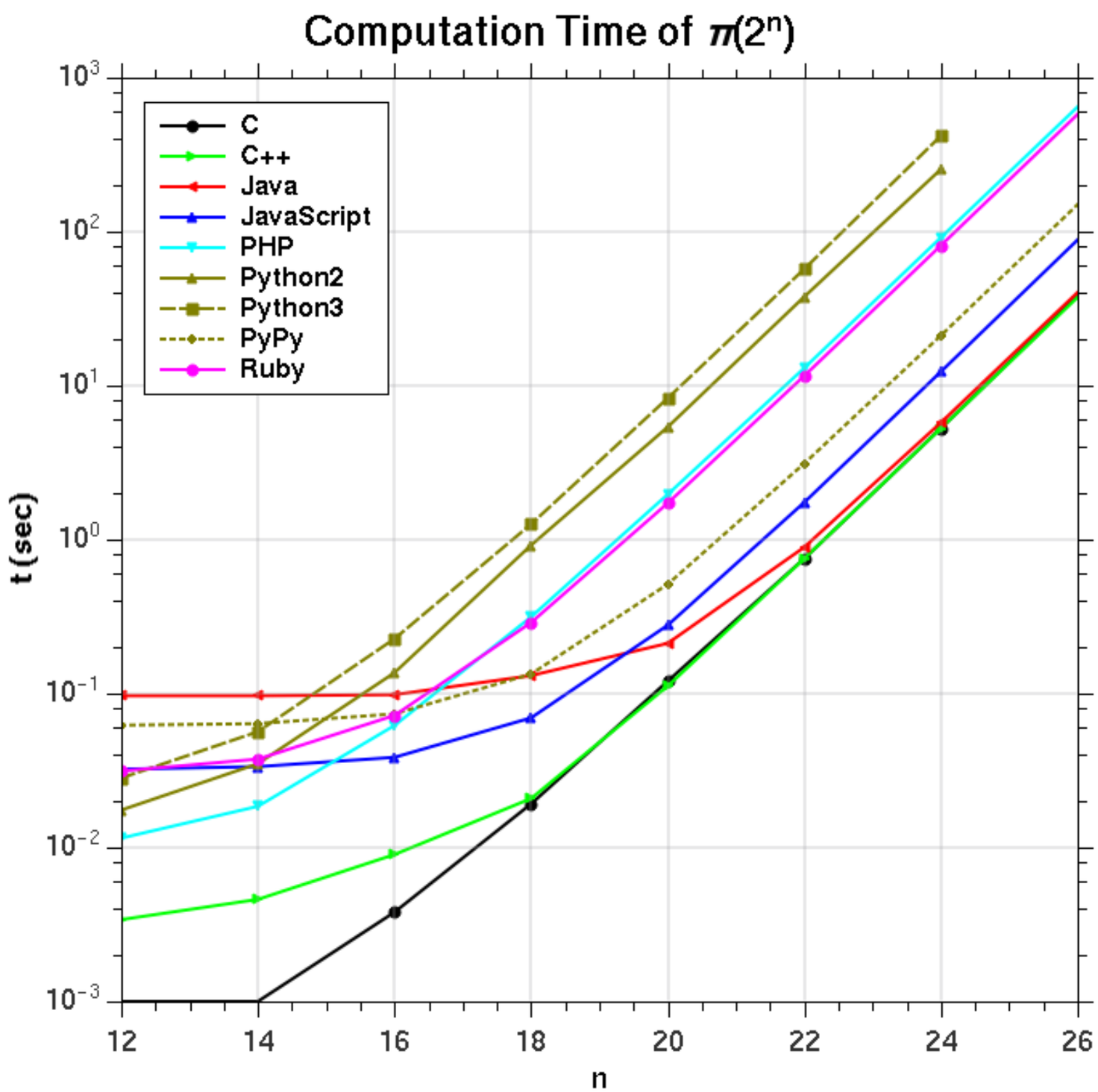
Warwick RSE

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

Programming languages

- Fortran (Not FORTRAN in the end, but the same basic idea)
- Fairly low level but not as close to the machine as C
- The “normal” way of programming Fortran makes a lot of restrictions
  - Ones that are generally not troublesome for scientific code
- Much harder to accidentally slow down your code because of these restrictions

Fortran as First Language

- Fortran is fairly forgiving for a high performance compiled language
- The built in features of Fortran are intended to work with arrays and make working with arrays easier than most other compile language
- Can have “one-liner” code comparable to interpreted languages (at least for the “right” type of arrays)
- **array = [array,new_element]** - grow an array
- **b = pick(a,a>0)** - create an array containing only >0 elements of array **a**
- **array = [(i**2,i=1,25)]** - Create an array containing the values from 1^2 to 25^2

Fortran as First Language

- Fortran is a general purpose language, but arrays are the real crown of the core language
- Other data structures are available as libraries, such as the "Fortran Standard Library" https://github.com/fortran-lang/stdlib
- **NB** Fortran has powerful "one-line" code for handling arrays, but unlike interpreted languages it is not **generally** faster to use them

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

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

Language Standards

- You might well find yourself working with Fortran 66 or 77 (or even older)
- We have a cheat sheet that will help you read the older code if you know modern Fortran
- https://warwick.ac.uk/research/rtp/sc/rse/training/cheatsheets/old_fortran_cheat_sheet.pdf
- We **strongly** recommend that you don't write any code using these old methods!

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.

Lines

- Fortran has a maximum line length of 132 characters
  - Many compilers support more but not required in the standard
- Putting an & on the end of a line joins the next line on
  - Can have up to 255 continuation lines
  - Can put an & on the start of the next line but this is only required if you break a line inside a string
- This is massively relaxed in F23 and some very new compilers now support much longer lines (if you need them)

Intrinsic Functions

- Fortran has a very wide range of **intrinsic functions**
  - Numerical functions (see https://warwick.ac.uk/research/rtp/sc/rse/training/cheatsheets/numerical_operators_cheat_sheet.pdf)
  - String handling functions
  - Array handling functions
  - Various others
- We will mention the ones that we use here but there are **lots** of others

Special Functions

- One of the most useful additions to Fortran 2008 is **special functions**
  - We don't know why they took so long to make it in!
- Bessel functions of the first and second kinds
- Gamma and log gamma function
- Error functions
- These are available in library form too

Libraries

- Fortran has been around for a long time
  - Modern Fortran is able to interoperate with code from the 60s
- Massive array of libraries for all sorts of purposes
  - http://fortranwiki.org/fortran/show/Libraries
  - Many more than those but that's a good starting point
  - Includes modern things like machine learning (https://arxiv.org/pdf/1902.06714.pdf)
- Can also call C code from Fortran (we're not covering it but it is briefly in the extras)

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

```fortran
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

```fortran
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!

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 SCRTP
- Fortran files have the .f90 extension by convention.
  - .f is also used but is outdated and may not work properly

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

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

```fortran
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

```fortran
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

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

```fortran
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

```fortran
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

```fortran
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 is 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

Subroutines and Functions

- In most languages there are **function**s.
  - 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

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

```fortran
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
```

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

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

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

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

Function Arguments

```fortran
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
```

```fortran
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

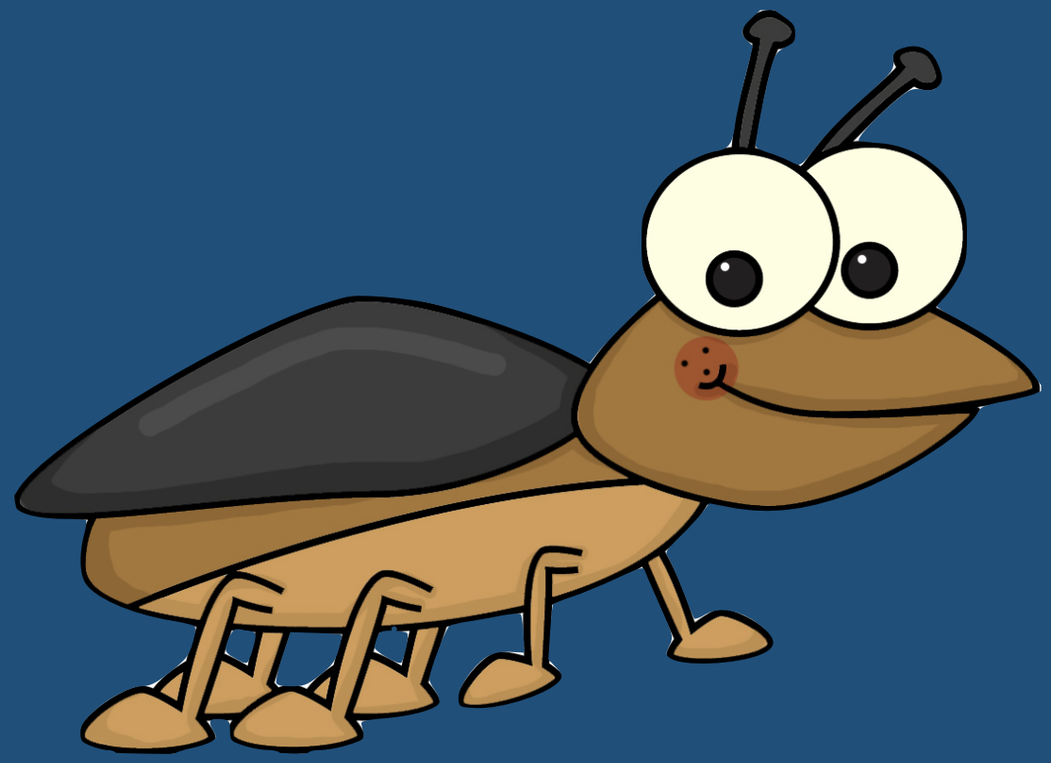
```
PROGRAM driver
  USE mymodule
  IMPLICIT NONE
  INTEGER :: a, b

  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



The End