Extras - Other bits of Fortran

Warwick RSE

I'm never finished with my paintings; the further I get, the more I seek the impossible and the more powerless I feel.

-

Claude Monet - Artist

What's in here?

- This is pretty much all of the common-ish bits of Fortran that we haven't covered here

- None of it is essential which is why it's in the extras but a lot of it is useful

- All of the stuff covered here is given somewhere in the example code that we use (albeit sometimes quite briefly)

Fortran Pointers

Pointers

- Many languages have some kind of concept of a **pointer** or a **reference**

- This is usually implemented as a variable that holds the location in memory of another variable that holds data of interest

- C like languages make very heavy use of pointers, especially to pass actual variables (rather than copies) of arrays to a function

  - Fortran avoids this because the pass by reference subroutines mean that you are always working with a reference and the fact that arrays are an intrinsic part of the language

Pointers

- Fortran pointers are quite easy to use but are a model that is not common in other languages

- You make a variable a pointer by adding the "**POINTER**" attribute to the definition

- In Fortran a pointer is the same as any other variable **except when you do specific pointer-y things to it**

- Formally this behaviour is called "automatic dereferencing"

  - When you do anything with a pointer that isn't specifically a pointer operation the pointer is automatically converted (the reference is followed, hence **dereference**) into the actual variable without you having to do anything to it

Pointer Analogy

- Imagine that you have a phone with your friend Jim's phone number in it
    - To call Jim normally you just look up Jim and press dial
        - The name (pointer) Jim automatically converts to a phone number and you don't need to worry about it
- This is like Fortran

Pointer Analogy

- Imagine now that you're somewhere where your network has no signal so you have to borrow someone else's phone
    - You now look up Jim in your phone book, get the phone number from it and type it into the other phone
- This is more like C style pointers where you have to actively convert a pointer into what it points to to use it

Pointers

- There are really only four new concepts for Fortran pointers
  - "**=>**" the points to operator. Tells a pointer to point to a variable
  - The "**TARGET**" attribute for a variable. Fortran pointers can **only** point to other pointer variables or target variables
  - The "**NULL()**" function which sets the pointer to state where it is pointing to nothing
  - The "**ASSOCIATED**" function that tests if a pointer is not in the **NULL** state (or tests if it is pointing to a specific variable)

Pointers

```
PROGRAM pointer_test

  INTEGER, TARGET :: actual_value
  INTEGER, POINTER :: ptr

  ptr => NULL()
  PRINT *,'Association test before', ASSOCIATED(ptr)
  actual_value = 5
  ptr => actual_value
  PRINT *,'Association test after', ASSOCIATED(ptr)
  PRINT *,'Association test specific', ASSOCIATED(ptr, &
      TARGET=actual_value)
  PRINT *,'Pointer value after pointing is ', ptr
  ptr = 10

  PRINT *, 'Actual value after changing pointer is ', actual_value

END PROGRAM pointer_test
```

Pointers

```
Association test before F
Association test after T
Association test specific T
Pointer value after pointing is              5
Actual value after changing pointer is              10
```

Pointers

- Note that I had to explicitly assign the pointer to have a NULL value.
    - By default pointers are created pointing to nowhere sensible at all
        - In this state they generally will report ASSOCIATED as true but are not usable or sensible
- I can use the pointer to both read data from and write data to the actual variable

Allocating Pointers

- As well as pointing pointers to existing variables I can allocate pointers.
    - This creates a new instance and points the pointer to it
- I can allocate pointer arrays, just like with allocatable arrays
    - Syntax is exactly the same as for allocatables, just switch **ALLOCATABLE** to **POINTER**
- I can also allocate a single instance of a pointer variable

Allocating Pointers

```
PROGRAM pointer_alloc

  INTEGER, POINTER :: iptr

  ALLOCATE(iptr)
  iptr = 10

  PRINT *, 'iptr is ', iptr
  DEALLOCATE(iptr)

END PROGRAM pointer_alloc
```

- After I have allocated iptr I can use it just like I could any other integer variable

- **BUT** I have to make sure that I deallocate it when I'm finished with it

Allocating Pointers

- Pointers are much more dangerous than allocatables

- I can keep allocating a pointer over and over again without deallocating it and nothing will stop me

- Each time I do so I lose track of the memory that the pointer previously pointed to so that I can't deallocate it

- **Memory leak - will eventually run out of memory**

Why pointers?

- Fortran actually avoids needing pointers very well compared to C etc.

- You'll mainly see them in two contexts

  - Advanced data structures (trees, linked lists etc.)

  - Until Fortran 2003 you couldn't have ALLOCATABLE arrays in TYPEs

  - Using a pointer to avoid an "IF" in a loop (for example)

Procedure pointers

Procedure Pointers

- You can have a pointer to a function or subroutine - typically called a procedure pointer in Fortran (often a function pointer in other languages)

- You can point it to any function and then call that function using the pointer

- Allows you to quickly switch between using different functions

  - Many other cool tricks

- Function pointers have to be matched to the parameters of the function that you want to point them to

Function Pointers

```
MODULE fnptr

  IMPLICIT NONE
  SAVE

  INTEGER, PARAMETER :: INT32 = SELECTED_INT_KIND(9)

  !This abstract interface defines a type of function that I can now get a
  !pointer to
  ABSTRACT INTERFACE
    FUNCTION opfn(val1, val2)
      !Have to IMPORT int32 to get it from the containing module
      !this keeps the function declaration and the encompassing module
      !separate
      IMPORT INT32
      INTEGER(INT32), INTENT(IN) :: val1, val2
      INTEGER(INT32) :: opfn
    END FUNCTION opfn
  END INTERFACE

  CONTAINS

    !These functions must match variables in kind, intent and type
    !But don't have to have the same variable names etc.
    !Attributes like allocatable and pointer must match as well
    FUNCTION add(val1, val2)
      INTEGER(INT32), INTENT(IN) :: val1, val2
      INTEGER(INT32) :: add

      add = val1 + val2
    END FUNCTION add

    FUNCTION minus(val1, val2)
      INTEGER(INT32), INTENT(IN) :: val1, val2
      INTEGER(INT32) :: minus

      minus = val1 - val2
    END FUNCTION minus

    FUNCTION do_op(val1, val2, op)
      INTEGER(INT32), INTENT(IN) :: val1, val2
      PROCEDURE(opfn) :: op
      INTEGER(INT32) :: do_op

      do_op = op(val1, val2)

    END FUNCTION do_op

END MODULE fnptr
```

Procedure definition

Function Pointers

```
MODULE fnptr

  IMPLICIT NONE
  SAVE

  INTEGER, PARAMETER :: INT32 = SELECTED_INT_KIND(9)

  !This abstract interface defines a type of function that I can now get a
  !pointer to
  ABSTRACT INTERFACE
    FUNCTION opfn(val1, val2)
      !Have to IMPORT int32 to get it from the containing module
      !this keeps the function declaration and the encompassing module
      !separate
      IMPORT INT32
      INTEGER(INT32), INTENT(IN) :: val1, val2
      INTEGER(INT32) :: opfn
    END FUNCTION opfn
  END INTERFACE

  CONTAINS

  !These functions must match variables in kind, intent and type
  !But don't have to have the same variable names etc.
  !Attributes like allocatable and pointer must match as well
  FUNCTION add(val1, val2)
    INTEGER(INT32), INTENT(IN) :: val1, val2
    INTEGER(INT32) :: add

    add = val1 + val2
  END FUNCTION add

  FUNCTION minus(val1, val2)
    INTEGER(INT32), INTENT(IN) :: val1, val2
    INTEGER(INT32) :: minus

    minus = val1 - val2
  END FUNCTION minus

  FUNCTION do_op(val1, val2, op)
    INTEGER(INT32), INTENT(IN) :: val1, val2
    PROCEDURE(opfn) :: op
    INTEGER(INT32) :: do_op

    do_op = op(val1, val2)

  END FUNCTION do_op

END MODULE fnptr
```

Actual functions matching definition

Function Pointers

```
MODULE fnptr

  IMPLICIT NONE
  SAVE

  INTEGER, PARAMETER :: INT32 = SELECTED_INT_KIND(9)

  !This abstract interface defines a type of function that I can now get a
  !pointer to
  ABSTRACT INTERFACE
    FUNCTION opfn(val1, val2)
      !Have to IMPORT int32 to get it from the containing module
      !this keeps the function declaration and the encompassing module
      !separate
      IMPORT INT32
      INTEGER(INT32), INTENT(IN) :: val1, val2
      INTEGER(INT32) :: opfn
    END FUNCTION opfn
  END INTERFACE

  CONTAINS

    !These functions must match variables in kind, intent and type
    !But don't have to have the same variable names etc.
    !Attributes like allocatable and pointer must match as well
    FUNCTION add(val1, val2)
      INTEGER(INT32), INTENT(IN) :: val1, val2
      INTEGER(INT32) :: add

      add = val1 + val2
    END FUNCTION add

    FUNCTION minus(val1, val2)
      INTEGER(INT32), INTENT(IN) :: val1, val2
      INTEGER(INT32) :: minus

      minus = val1 - val2
    END FUNCTION minus

    FUNCTION do_op(val1, val2, op)
      INTEGER(INT32), INTENT(IN) :: val1, val2
      PROCEDURE(opfn) :: op
      INTEGER(INT32) :: do_op

      do_op = op(val1, val2)

    END FUNCTION do_op

END MODULE fnptr
```

Function that takes a procedure as an argument

Function Pointers

```
PROGRAM test_ptr

  USE fnptr
  IMPLICIT NONE

  PROCEDURE(opfn), POINTER :: ptr => NULL()

  !Use the function pointer with add
  ptr => add
  PRINT *, ptr(1,2)

  !And the same with minus
  ptr => minus
  PRINT *, ptr(1,2)

  !Now use the function with a function as a parameter
  PRINT *, do_op(1,2,add)
  PRINT *, do_op(1,2,minus)

END PROGRAM
```

Pointer to a procedure

Optional Arguments

Optional Arguments

```
MODULE optmod

  IMPLICIT NONE

  CONTAINS

  SUBROUTINE optional_argument(name)
    CHARACTER(LEN=*), INTENT(IN), OPTIONAL :: name

    IF (PRESENT(name)) THEN
      PRINT *, 'Hello ' // TRIM(name)
    ELSE
      PRINT *, 'Hello world!'
    END IF

  END SUBROUTINE optional_argument

END MODULE optmod
```

Optional Arguments

- It is important to note that optional arguments that are not present are special
- The only thing that you can validly do under the Fortran standards to an optional argument that is not present is test it with the PRESENT function

Function Overloading

Function Overloading

- Quite often you want to write a function that does the same job but on different parameters
  - Old school way “do_thing_int”, “do_thing_float” etc.
- Function overloading - one name, different interfaces. Compiler selects function/subroutine that matches the parameters given when the function is called
- Quite strict rules on what are “different interfaces”

Function Overloading

```
MODULE overload
  IMPLICIT NONE

  INTERFACE anyprint
    MODULE PROCEDURE print_int
    MODULE PROCEDURE print_real
  END INTERFACE anyprint

  CONTAINS

  SUBROUTINE print_int(i)
    INTEGER, INTENT(IN) :: i
    PRINT *, 'INTEGER ', i
  END SUBROUTINE print_int

  SUBROUTINE print_real(r)
    REAL, INTENT(IN) :: r
    PRINT *, 'REAL ', r
  END SUBROUTINE print_real

END MODULE overload
```

```
PROGRAM test

  USE overload
  IMPLICIT NONE

  CALL anyprint(1)
  CALL anyprint(1.0)

END PROGRAM test
```

Function Overloading

- You can extend an interface later if you want
  - Add extra versions of a function that can only be defined in other places
- Simply create a new interface block with the same name and it extends the interface

Encapsulation and Object Orientation

Fortran Encapsulation

- Fortran MODULEs
  - Store Data
  - Contain functions
    - Allow you to specify PUBLIC/PRIVATE properties for their contents
- You can (and should) use MODULEs to keep data and associated functions that operate on the data together
  - Only make data PUBLIC that things outside should be able to access

Fortran Encapsulation

- Simplest form you just have module variables and the functions that operate on them in a module

- Consider the supplied “random_mod” random number generator module

- It has to store quite a lot of state information but the user doesn’t really want to know about it

- Use private variables to the module to hold that state and just have public functions to get random numbers etc.

Fortran Object Orientation

- Sometimes you want to have more than one set of data
    - Not possible with modules since you can only have one *instance* of each module
- Generalisation is **Object Orientation**
    - Use a derived TYPE and bind functions/ subroutines to it
        - Generally called **methods**

Object Orientation

```
MODULE oo

  IMPLICIT NONE

  TYPE :: ootype
    INTEGER :: intval = 0
    CONTAINS
    PROCEDURE :: set_int
    PROCEDURE :: get_int
  END TYPE ootype
  CONTAINS

  SUBROUTINE set_int(this, value)
    CLASS(ootype), INTENT(INOUT) :: this
    INTEGER, INTENT(IN) :: value
    this%intval = value
  END SUBROUTINE set_int

  FUNCTION get_int(this)
    CLASS(ootype), INTENT(IN) :: this
    INTEGER :: get_int
    get_int = this%intval
  END FUNCTION get_int

END MODULE oo
```

```
PROGRAM ootest

  USE oo
  IMPLICIT NONE
  TYPE(ootype) :: instance

  CALL instance%set_int(10)
  PRINT *, instance%get_int()

END PROGRAM ootest
```

- "**this**" parameter to the functions is object that the function is being called from

- Called anything you like

Fortran Object Orientation

- Lot more to object orientation that just having functions attached to objects
  - Could give an entire course on OO design
  - Another on OO Fortran
- Fortran (2003+) really is object oriented
  - Encapsulation
  - Inheritance
  - Polymorphism

Fortran/C Interoperability

Fortran / C Interoperability

- Fortran KINDs matching C variable types
- BIND(C) attribute for functions and subroutines to flag them as being C interoperable
- BIND(C) attribute for TYPEs to flag them as being C interoperable (match layout of C struct)
- VALUE attribute for dummy arguments to make them pass by value rather than pass by reference
  - Without this all arguments to Fortran functions are pointers in C

Fortran/C Interoperability

```fortran
USE ISO_C_BINDING

TYPE, BIND(C) :: c_type
  INTEGER(C_INT) :: myint
  REAL(C_FLOAT), DIMENSION(10) :: myfloat
END TYPE c_type
```

```c
struct c_type{
  int myint;
  float myfloat[10];
};
```

Fortran / C Interoperability

- Can both
  - Call C functions from Fortran
  - Call Fortran functions from C
- Subprograms can only have **dummy** arguments of types that have C equivalents (extended in F2018)
  - Simple variables
  - Non allocatable, non pointer arrays (actual arguments can be allocatable or pointer but not dummy arguments)
  - BIND(C) TYPEs

Fortran/C Interoperability

```
MODULE c_interop

  USE ISO_C_BINDING
  IMPLICIT NONE

  CONTAINS

  SUBROUTINE get_array_max(sz, items) BIND(C)
    INTEGER(C_INT), INTENT(IN), VALUE :: sz
    INTEGER(C_INT), DIMENSION(sz), INTENT(IN) :: items

    PRINT *, MAXVAL(items)
  END SUBROUTINE get_array_max

END MODULE c_interop
```

```
extern void get_array_max(int, int*);
int main()
{
  int u[10] = {7,2,9,4,5,128,7,9,10,65};
  get_array_max(10, u);
}
```

Fortran/C Interoperability

```c
int get_random_number()
{
  /*Random number generated by a fair die roll*/
  return 5;
}
```

Fortran/C Interoperability

```
MODULE c_call

  USE ISO_C_BINDING
  IMPLICIT NONE

  INTERFACE
    FUNCTION get_random_number() BIND(C)
      INTEGER(C_INT) :: get_random_number
    END FUNCTION get_random_number
  END INTERFACE

END MODULE c_call

PROGRAM test

  USE c_call
  IMPLICIT NONE

  PRINT *, get_random_number()

END PROGRAM test
```

Fortran/C Interoperability

- TYPE(C_PTR) - Fortran type corresponding to C pointer - no checking of C type

- C_LOC - Get C pointer from Fortran TARGET/ POINTER variable

- C_F_POINTER - Get Fortran pointer from C pointer
  - Works with arrays but you have to specify the shape

- C_FUNLOC / C_F_PROCPOINTER same for functions

What's not in here?

- This now covers over 90% of Modern Fortran. The rest is a bit more unusual and a bit more involved so we aren't going to cover it at all

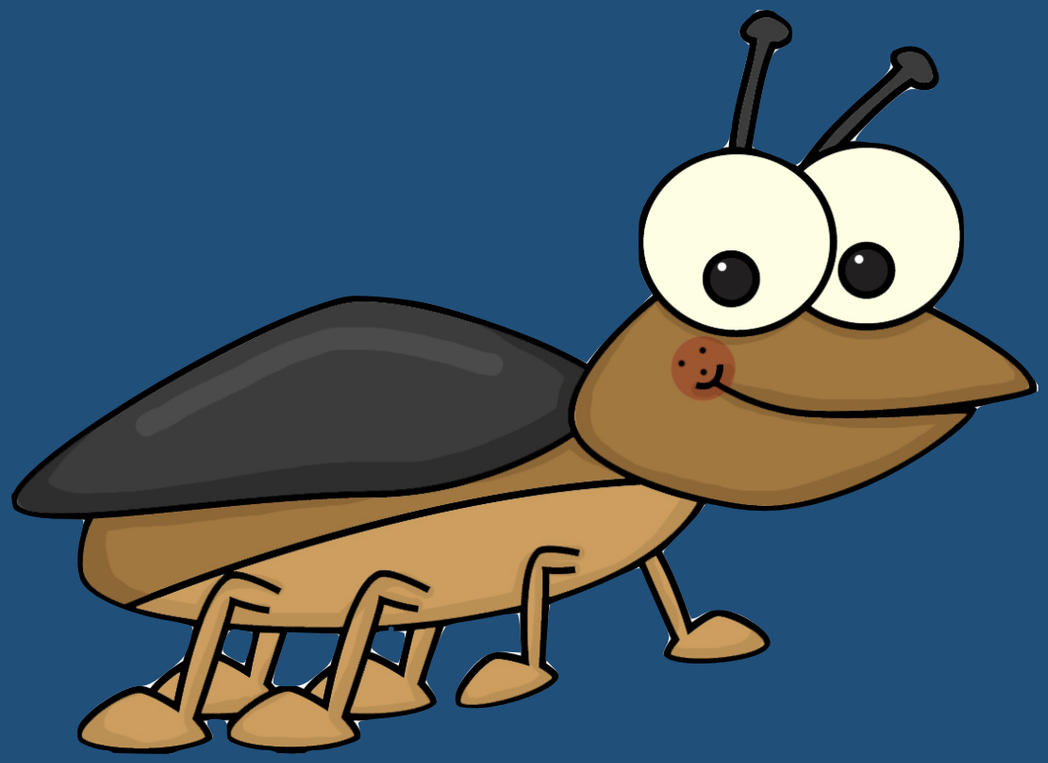
- The major remaining bits are

  - Coarray Fortran and Teams - A mechanism for writing parallel code natively in Fortran

  - Submodules - Effectively a way of splitting modules to make compiling faster

  - Parameterised derived types - write derived types that can take kind parameters like primitive types can (and length parameters like strings can)

  - Derived Type IO - write routines to allow you to pass a derived type to PRINT, WRITE or READ routines



The End