Introduction to Modern Fortran

Data Pointers

Nick Maclaren

nmm1@cam.ac.uk

March 2014

Data Pointers

- Fortran pointers are unlike C/C++ ones

Not like Lisp or Python ones, either

- Errors with using pointers are rarely obvious

This statement applies to almost all languages

- Fortran uses a semi–safe pointer model

Translation: your footgun has a trigger guard

Use pointers only when you need to

Pointer and Allocatable

Pointers are a sort of changeable allocation
In that use, they almost always point to arrays
For example, needed for non–rectangular arrays

Always try to use allocatable arrays first
Only if they really aren't adequate, use pointers

ALLOCATABLE was restricted in Fortran 95
Fortran 2003 removed almost all restrictions
You may come across POINTER in old code
It can usually be replaced by ALLOCATABLE

Pointer-Based Algorithms

Some genuinely pointer–based algorithms
Fortran is not really ideal for such uses

- But don't assume anything else is any better!

There are NO safe pointer–based languages
Theoretically, one could be designed, but ...

In Fortran, see if you can use integer indices
That has software engineering advantages, too
If you can't, you may have to use pointers

Pointer Concepts

Pointer variables point to target variables
In almost all uses, pointers are transparent

- You access the target variables they point to

Dereferencing the pointer is automatic

- Special syntax for meaning the pointer value

The POINTER attribute indicates a pointer
The TARGET attribute indicates a target
No variable can have both attributes

Example

```
PROGRAM fred
    REAL, TARGET :: popinjay = 0.0
    REAL, POINTER :: arrow
    arrow => popinjay
    ! arrow now points to popinjay
    arrow = 1.23
    PRINT *, popinjay
    popinjay = 4.56
    PRINT *, arrow
END PROGRAM fred
```

```
 1.2300000
 4.5599999
```

Pointers and Target Arrays

```
REAL, DIMENSION(20), TARGET :: array
REAL, DIMENSION(:), POINTER :: index
```

Pointer arrays must be declared without bounds
They will take their bounds from their targets

- Pointer arrays have just a rank
Which must match their targets, of course

Very like allocatable arrays

Use of Targets

Treat targets just like ordinary variables

The ONLY difference is an extra attribute
Allows them on the RHS of pointer assignment

Valid targets in a pointer assignment?
If OK for INTENT(INOUT) actual argument
Variables, array elements, array sections etc.

```
REAL, DIMENSION(20, 20), TARGET :: array
REAL, DIMENSION(:, :), POINTER :: index
index => array(3:7:2, 8:2:−1)
```

Initialising Pointers

Pointer variables are initially undefined

- Not initialising them is a Bad Idea

- You can use the special syntax => null()

To initialise them to disassociated (*sic*)

```
REAL, POINTER :: index => null()
```

- Or you can point them at a target, ASAP

Note that null() is a disassociated target

Pointer Assignment

You use the special assignment operator =>
Note that using = assigns to the target

```
PROGRAM fred
    REAL, TARGET :: popinjay
    REAL, POINTER :: arrow
    arrow => popinjay       ! POINTER assignment
    ! arrow now points to popinjay
    arrow = 1.23            ! TARGET assignment
    PRINT *, popinjay
    popinjay = 4.56         ! TARGET assignment
    PRINT *, arrow
    arrow => null()         ! POINTER assignment
END PROGRAM fred
```

Pointer Expressions

Also pointer expressions on the RHS of =>
Currently, only the results of function calls

```
FUNCTION select (switch, left, right)
    REAL, POINTER :: select, left, right
    LOGICAL switch
    IF (switch) THEN
        select => left
    ELSE
        select => right
    END IF
END FUNCTION select

new_arrow => select(A > B, old_arrow, null())
```

ALLOCATE

You can use this just as for allocatable arrays
This creates some space and sets up array

```
REAL, DIMENSION(:, :), POINTER :: array
ALLOCATE(array(3:7:2, 8:2:−1), STAT=n)
```

If you can, stick to using ALLOCATABLE

Do you get the idea I don't like pointers much?
At the end, I mention why you may need them

DEALLOCATE

- Only on pointers set up by ALLOCATE

```
DEALLOCATE(array, STAT=n)
```

array now becomes disassociated
Other pointers to its target become undefined

- Don't DEALLOCATE undefined pointers

That is undefined behaviour

Previous Pointer Values

New pointer value overwrites the previous one
Applies to both assignment and ALLOCATE
Well, it is a sort of assignment ...

- Does not affect other pointers to the target

But DEALLOCATE makes other pointers undefined
Also happens if the target goes out of scope

- That causes the dangling pointer problem

And assignment can break the last link

- Memory leaks and (rarely) worse problems

ASSOCIATED

- Can test if pointers are associated

```
IF (ASSOCIATED(array)) . . .
IF (ASSOCIATED(array, target)) . . .
```

Works if array is associated or disassociated
Latter tests if array is associated with target

- Don't use it on undefined pointers

That is undefined behaviour

A Nasty “Gotcha”

Fortran 95 forbids POINTER and INTENT

- Fortran 2003 applies INTENT to the link

```
subroutine joe (arg)
    real, target :: junk
    real, pointer, intent(in) :: arg
    allocate(arg)        ! this is ILLEGAL
    arg => junk          ! this is ILLEGAL
    arg = 4.56           ! but this is LEGAL :-(
end subroutine joe
```

Irregular Arrays

- Fortran does not support them

This is how you do the task, if you need to

```
TYPE Cell
    REAL, DIMENSION(:), ALLOCATABLE :: column
END TYPE Cell

TYPE(Cell), DIMENSION(:), ALLOCATABLE :: matrix
```

matrix can be a non−rectangular matrix
Note that pointers are not needed in this case

Example

```
TYPE Cell
      REAL, DIMENSION(:), ALLOCATABLE :: column
END TYPE Cell

TYPE(Cell), DIMENSION(:), ALLOCATABLE :: matrix

INTEGER, DIMENSION(100) :: rows
READ *, N, (rows(K), K = 1,N)
ALLOCATE(matrix(1:N))
DO K = 1,N
      ALLOCATE(matrix(K)%column(1:rows(K)))
END DO
```

Arrays of Pointers

- Fortran does not support them

This is how you do the task, if you need to

```
TYPE Cell
    REAL, DIMENSION(:), POINTER :: column
END TYPE Cell

TYPE(Cell), DIMENSION(100) :: matrix
```

Remember Trees?

This was the example we used in derived types

```
TYPE :: Leaf
    CHARACTER(LEN=20) :: name
    REAL(KIND=dp), DIMENSION(3) :: data
END TYPE Leaf
TYPE :: Branch
    TYPE(Leaf), ALLOCATABLE :: leaves(:)
END TYPE Branch
TYPE :: Trunk
    TYPE(Branch), ALLOCATABLE :: branches(:)
END TYPE Trunk
```

Recursive Types

We can do this more easily using recursive types

```
TYPE :: Node
    TYPE(Node), POINTER :: subnodes(:)
    CHARACTER(LEN=20) :: name
    REAL(KIND=dp), DIMENSION(3) :: data
END TYPE Node
```

Recursive components must be pointers
Fortran 2008 will allow allocatable
Obviously a type cannot include itself directly

More Complicated Structures

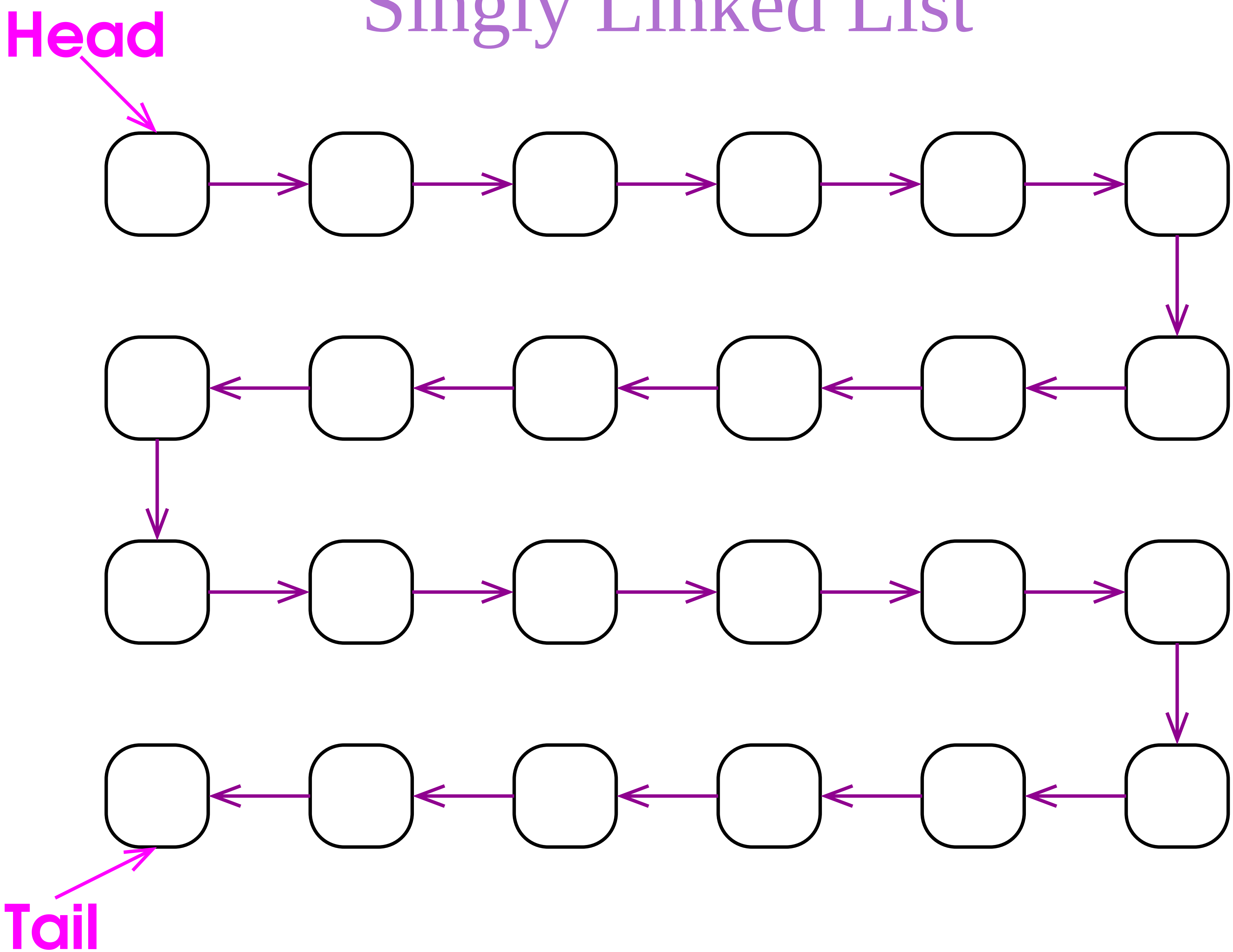
In mathematics, a graph is a set of linked nodes
Common forms include linked lists, trees etc.

A tree is just a hierarchy of objects
We have already covered these, in principle

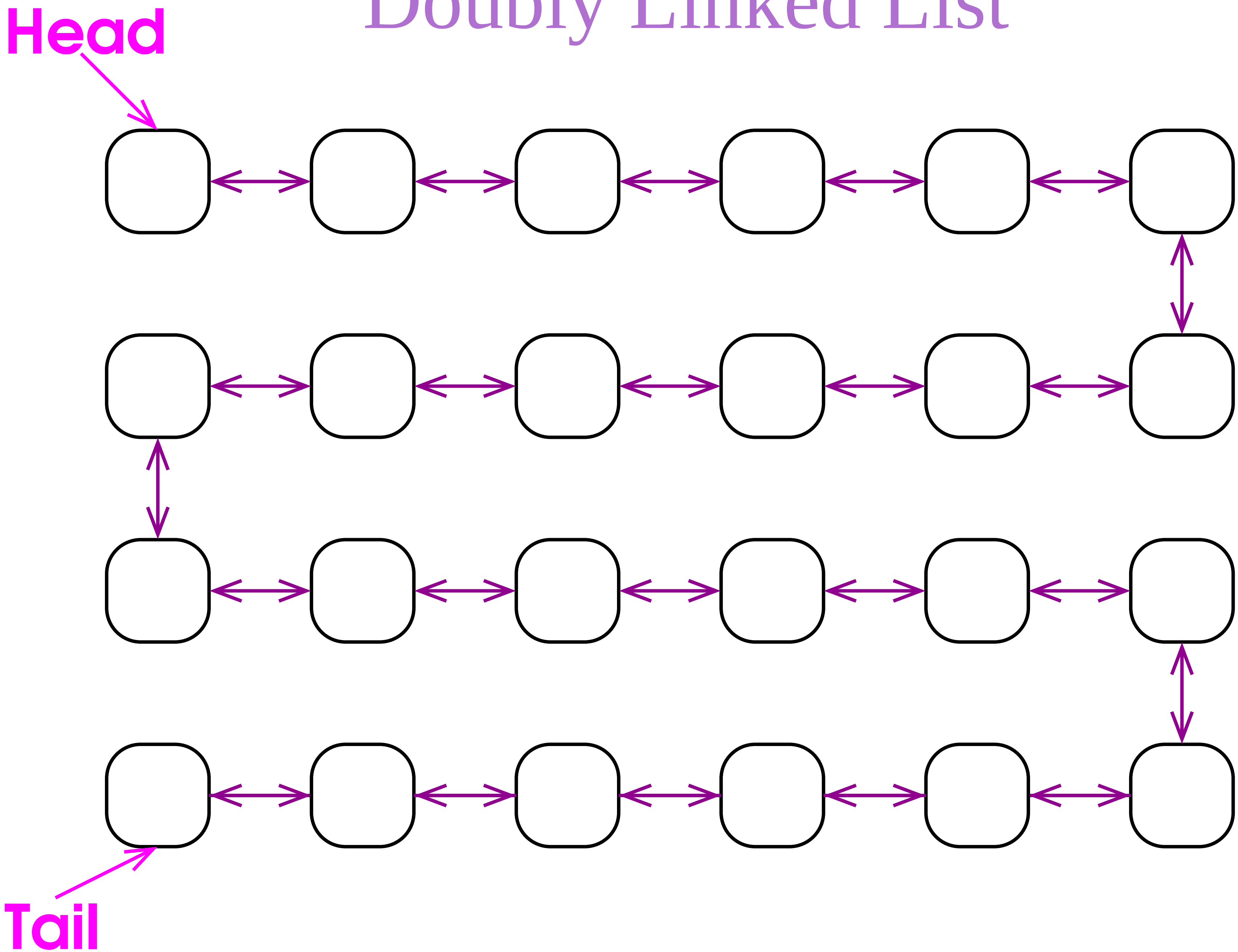
Linked lists (also called chains) are common
And there are lots of more complicated structures

Those are very painful to handle in old Fortran
So most Fortran programmers tend to avoid them
But they aren't difficult in modern Fortran

Singly Linked List



Doubly Linked List



Linked Lists

You can handle linked lists in a similar way
And any other graph–theoretic data structure, too

```
TYPE Cell
    CHARACTER(LEN=20) :: node_name
    REAL :: node_weight
    TYPE(Cell), POINTER :: next, last, &
        first_child, last_child
END TYPE Cell
```

Working with such data structures is non–trivial
Whether in Fortran or any other language

Graph Structures

Using pointers in Fortran is somewhat tedious
But it is as easy as in C++ and a little safer

Graph structures are in computer science
linked lists are probably the only easy case
Plenty of books on them, for example:

Cormen, T.H. et al. Introduction to Algorithms
Knuth,D.E. The Art Of Computer Programming
Also Sedgewick, Ralston, Aho et al. etc.

Procedure Pointers

Fortran 2003 allows them, as well as data pointers

Don't go there

This has absolutely nothing to do with Fortran
They are a nightmare in all languages, including C++
They are almost impossible to use safely
A fundamental problem in any scoped language

- Very rarely need them in clean code, anyway

Passing procedures as arguments is usually enough
Or one procedure calling a fixed set of others