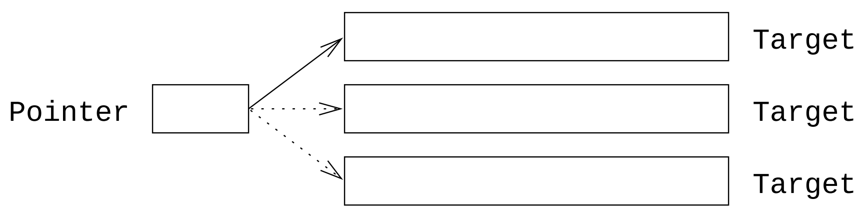
Lecture 7:

Pointers and Derived Types

Pointers and Targets

It is often useful to have variables where the space referenced by the variable can be changed as well as the values stored in that space.



- The pointer often uses less space than the target.

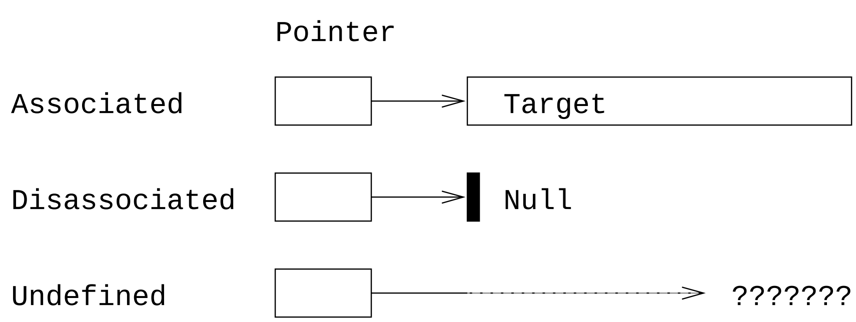
- A reference to the pointer will in general be a reference to the target, (pointers are automatically dereferenced).

Terminology

A pointer has 3 possible states:

- if a pointer has a particular target then the pointer is said to be *associated* with that target,
- a pointer can be made to have no target — the pointer is *disassociated*,
- the initial status of a pointer is *undefined*,

Visualisation,



Use `ASSOCIATED` intrinsic to get the (association) status of a pointer.

Pointer Declaration

A `POINTER`:

- is a variable with the `POINTER` attribute;

- has static type, kind and rank determined by its declaration, for example,

```
REAL, POINTER :: Ptor
REAL, DIMENSION(:,:), POINTER :: Ptoa
```

  - `Ptor` is a pointer to a scalar real target,
  - `Ptoa` is a pointer to a 2-D array of reals.

So,

- the declaration fixes the type, kind and rank of the target;

- pointers to arrays are declared with deferred-shape array specifications;

- the rank of a target is fixed but the shape may vary.

Target Declaration

Targets of a pointer must have the `TARGET` attribute.

```
REAL, TARGET :: x, y
REAL, DIMENSION(5,3), TARGET :: a, b
REAL, DIMENSION(3,5), TARGET :: c
```

With these declarations (and those from the previous slide):

- `x` or `y` may become associated with `Ptor`;
- `a`, `b` or `c` may become associated with `Ptoa`.

Pointer Manipulation

The following operators manipulate pointers:

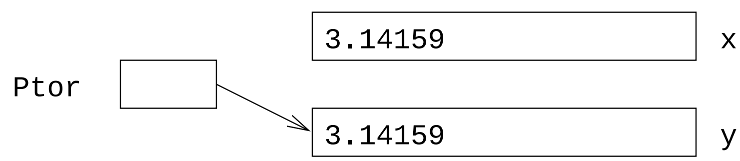
- =>, *pointer assignment* — alias a pointer with a given target;
- =, *'normal' assignment* — assign a value to the space pointed at by the pointer.

Pointer assignment makes the pointer and the variable reference the same space while the normal assignment alters the value contained in that space.

Pointer Assignment

Consider,

```
x = 3.14159
Ptor => y
Ptor = x
```



- `x` and `Ptor` have the same value.

- `Ptor` is an alias for `y` so the last statement sets `y = 3.14159`.

- if the value of `x` is subsequently changed, the value of `Ptor` and `y` do not.

Coding,

```
Ptor = 5.0
```

sets `y` to `5.0`.

Dynamic Targets

Targets can also be created dynamically by allocation. `ALLOCATE` can make space become the target of a pointer.

```
ALLOCATE(Ptor,STAT=ierr)
ALLOCATE(Ptoa(n*n,2*k-1),STAT=ierr)
```

- □ the first statement allocates a single real as the target of `Ptor`.

- □ the second allocates a rank 2 real array as the target of `Ptoa`.

It is not an error to allocate an array pointer that is already associated.

Association Status

The status of a defined pointer may be tested by an intrinsic function:

```
ASSOCIATED(Ptoa)
```

If `Ptoa` is defined and associated then this will return `.TRUE.`; if it is defined and disassociated it will return `.FALSE.`. If it is undefined the result is also undefined.

The target of a defined pointer may also be tested:

```
ASSOCIATED(Ptoa, arr)
```

If `Ptoa` is defined and currently associated with the specific target, `arr`, then the function will return `.TRUE.`, otherwise if it will return `.FALSE.`.

Pointer Disassociation

Pointers can be disassociated with their targets by:

- nullification

  ```
  NULLIFY(Ptor)
  ```

  - breaks the connection of its pointer argument with its target (if any),
  - disassociates the pointer.

  Note: it is good practise to nullify *all* pointers before use.

- deallocation

  ```
  DEALLOCATE(Ptoa, STAT=ierr)
  ```

  - breaks the connection between the pointer and its target,
  - deallocates the target.

Practical Example

Pointers can be of great use in iterative problems. Iterative methods:

- make guess at required solution;
- use guess as input to an equation to produce better approximation;
- use new approximation to obtain better approximation;
- repeat until degree of accuracy is obtained;

```
REAL, DIMENSION(100,100), TARGET :: app1, app2
REAL, DIMENSION(:,:), POINTER :: prev_app, &
                                 next_app, swap
prev_app => app1
next_app => app2
prev_app = initial_app(.....)
DO
 next_app = iteration_function_of(prev_app)
 IF(ABS(MAXVAL(next_app-prev_app))<0.0001)EXIT
 swap => prev_app
 prev_app => next_app
 next_app => swap
END DO
```

Using pointers here avoids having to copy the large matrices.

Derived Types

It is often advantageous to express some objects in terms of aggregate structures, for example: coordinates, (x, y, z).

Fortran 90 allows compound entities or *derived types* to be defined:

```
TYPE COORDS_3D
 REAL :: x, y, z
END TYPE COORDS_3D
TYPE(COORDS_3D) :: pt1, pt2
```

Derived types definitions should be placed in a `MODULE`.

Supertypes

Previously defined types can be used as components of other derived types,

```
TYPE SPHERE
 TYPE (COORDS_3D) :: centre
 REAL             :: radius
END TYPE SPHERE
```

Objects of type `SPHERE` can be declared:

```
TYPE (SPHERE) :: bubble, ball
```

Derived Type Assignment

Values can be assigned to derived types in two ways:

- component by component;
- as an object.

An individual component may be selected, using the % operator:

```
pt1%x = 1.0
bubble%radius = 3.0
bubble%centre%x = 1.0
```

The whole object may be selected and assigned to using a constructor:

```
pt1 = COORDS_3D(1.,2.,3.)
bubble%centre = COORDS_3D(1.,2.,3.)
bubble = SPHERE(bubble%centre,10.)
bubble = SPHERE(COORDS_3D(1.,2.,3.),10.)
```

The derived type component of SPHERE must also be assigned to using a constructor.

Assignment between two objects of the same derived type is intrinsically defined,

```
ball = bubble
```

Derived Type I/O

Derived type objects which do not contain pointers (or private) components may be input or output using 'normal' methods:

```
PRINT*, bubble
```

is exactly equivalent to

```
PRINT*, bubble%centre%x, bubble%centre%y, &
        bubble%centre%z, bubble%radius
```

Derived types are handled on a component by component basis.

POINTER Components of Derived Types

- ALLOCATABLE arrays cannot be used as components in a derived type,

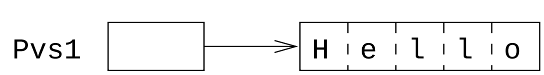
- POINTERs can be,

Dynamically sized structures can be created and manipulated, for example,

```
TYPE VSTRING
 CHARACTER, DIMENSION(:), POINTER :: chars
END TYPE VSTRING
```

this has a component which is a pointer to a 1-D array of characters.

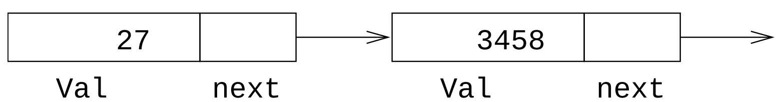
```
TYPE(VSTRING) :: Pvs1
 ...
ALLOCATE(Pvs1%chars(5))
Pvs1%chars = (/"H","e","l","l","o"/)
```



Pointers and Recursive Data Structures

- Derived types which include pointer components provide support for recursive data structures such as linked lists.

```
TYPE CELL
 INTEGER :: val
 TYPE (CELL), POINTER :: next
END TYPE CELL
```



- Assignment between structures containing pointer components is subtlely different from normal,

```
TYPE(CELL) :: A
TYPE(CELL), TARGET :: B
A = B
```

is equivalent to:

```
A%val = B%val
A%next => B%next
```

Practical Example of Linked Lists

The following fragment would create a linked list of cells starting at **head** and terminating with a cell whose **next** pointer is null (disassociated).

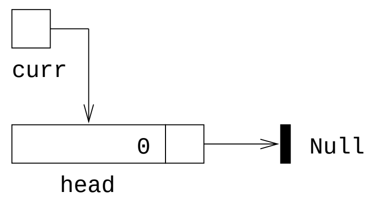
```
PROGRAM Thingy
 IMPLICIT NONE
 TYPE (CELL), TARGET  :: head
 TYPE (CELL), POINTER :: curr, temp
 INTEGER              :: k
 head%val = 0             ! listhead = default
 NULLIFY(head%next)       ! un-undefine
 curr => head             ! curr head of list
 DO
  READ*, k                ! get value of k
  ALLOCATE(temp)          ! create new cell
  temp%val = k            ! assign k to new cell
  NULLIFY(temp%next)      ! set disassociated
  curr%next => temp       ! attach new cell to
                          ! end of list
  curr => temp            ! curr points to new
                          ! end of list
 END DO
END PROGRAM Thingy
```

Example (Continued)

The statements,

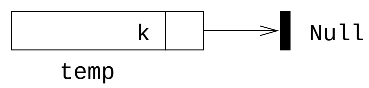
```
head%val = 0; NULLIFY(head%next); curr => head
```

give,



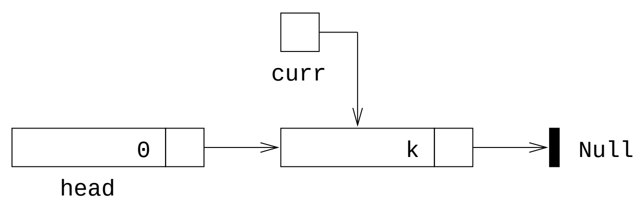

```
ALLOCATE(temp); temp%val = k; NULLIFY(temp%next)
```

give,

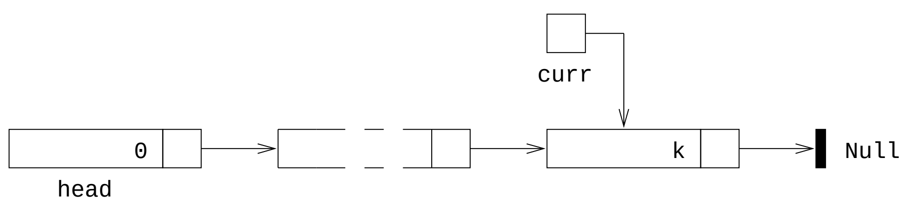



```
curr%next => temp; curr => temp
```

give,



The final list structure is,



Example (Continued)

A “walk-through’ the previous linked list could be written as follows:

```
curr => head
DO
 PRINT*, curr%val
 IF(.NOT.ASSOCIATED(curr%next)) EXIT
 curr => curr%next
ENDDO
```

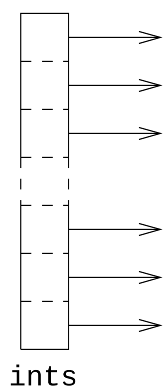
All sorts of multiply linked lists can be created and manipulated in analogous ways.

Arrays of Pointers

It is possible to create what are in effect arrays of pointers:

```
TYPE iPTR
 INTEGER, POINTER :: compon
END TYPE iPTR
TYPE(iPTR), DIMENSION(100) :: ints
```

Visualisation,



Cannot refer to a whole array of pointers,

```
ints(10)%compon ! valid
ints(:)%compon  ! not valid
```

If desired `ints` could have been `ALLOCATABLE`.