Fortran pointers

Victor Eijkhout, Susan Lindsey

COE 322 Fall 2020

Pointers are aliases

- Pointer points at an variable
- Access variable through pointer
- You can change what variable the pointer points at.

Setting the pointer

- You have to declare that a variable is pointable:

```
real,target :: x
```

- Set the pointer with => notation (New! Note!):

```
point_at_real => x
```

Dereferencing

Fortran pointers are often automatically *dereferenced*: if you print a pointer you print the variable it references, not some representation of the pointer.

Code:

```
real,target :: x
real,pointer :: realp

print *,"Pointer starts as not set"
if (.not.associated(realp)) &
   print *,"Pointer not associated"
x = 1.2
print *,"Set pointer"
realp => x
if (associated(realp)) &
   print *,"Pointer points"
print *,"Unset pointer"
nullify(realp)
if (.not.associated(realp)) &
   print *,"Pointer not associated"
```

Output
[pointerf] basicp:

```
 1.20000005
```

Pointer example

Code:

```
real,target :: x,y
real,pointer :: that_real

x = 1.2
y = 2.4
that_real => x
print *,that_real
that_real => y
print *,that_real
y = x
print *,that_real
```

Output

[pointerf] realp:

```
 1.20000005
 2.40000010
 1.20000005
```

1. The pointer points at `x`, so the value of `x` is printed.
2. The pointer is set to point at `y`, so its value is printed.
3. The value of `y` is changed, and since the pointer still points at `y`, this changed value is printed.

Assign pointer from other pointer

```
real,pointer :: point_at_real,also_point
point_at_real => x
also_point => point_at_real
```

Now you have two pointers that point at `x`.

Very important to use the =>, otherwise strange memory errors

Assignment subtleties

Assign underlying variables:

```
real,target :: x,y
real,pointer :: p1,p2

x = 1.2
p1 => x
p2 => y
p2 = p1
print *,p2
```

Crash because lhs pointer unassociated:

```
real,target :: x
real,pointer :: p1,p2

x = 1.2
p1 => x
p2 = p1
print *,p2
```

Pointer status

- `Nullify`: zero a pointer
- `Associated`: test whether assigned

Code:

```
real,target :: x
real,pointer :: realp

print *,"Pointer starts as not set"
if (.not.associated(realp)) &
   print *,"Pointer not associated"
x = 1.2
print *,"Set pointer"
realp => x
if (associated(realp)) &
   print *,"Pointer points"
print *,"Unset pointer"
nullify(realp)
if (.not.associated(realp)) &
   print *,"Pointer not associated"
```

Output
[pointerf] statusp:

```
Pointer starts as not set
Pointer not associated
Set pointer
Pointer points
Unset pointer
Pointer not associated
```

Dynamic allocation

Pointers are Fortran's way of making dynamic allocation:

```
Integer,Pointer,Dimension(:) :: array_point
Allocate( array_point(100) )
```

This is automatically deallocated when control leaves the scope.
No memory leaks.

Exercise 1

Write a routine that accepts an array and a pointer, and on return has that pointer pointing at the largest array element:

Code:

```
real,dimension(10),target :: array &
     = [1.1, 2.2, 3.3, 4.4, 5.5, &
        9.9, 8.8, 7.7, 6.6, 0.0]
real,pointer :: biggest_element

print '(10f5.2)',array
call SetPointer(array,biggest_element)
print *,"Biggest element is",
    biggest_element
biggest_element = 0
print '(10f5.2)',array
```

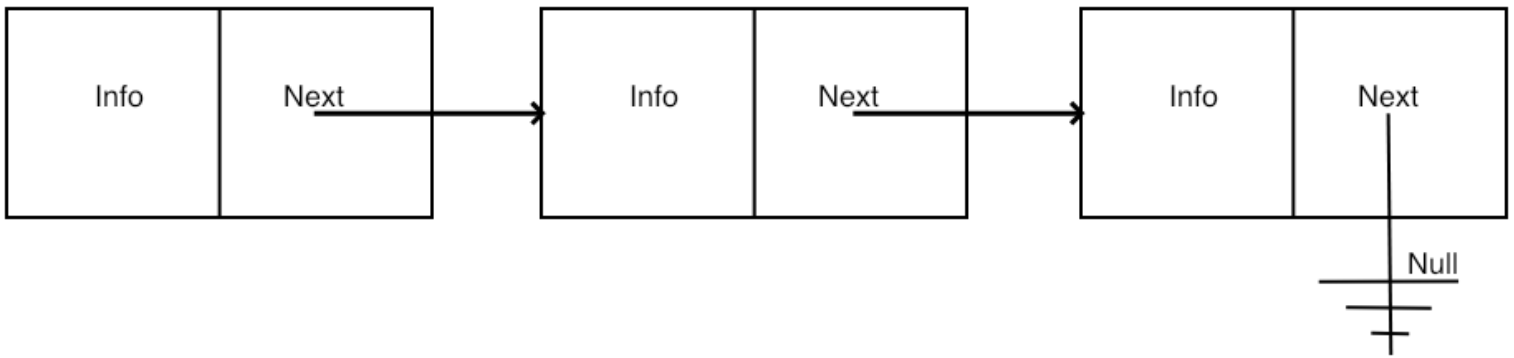
Output
[pointerf] arpointf:

```
 1.10 2.20 3.30 4.40 5.50 9.90 8
Biggest element is   9.8999996
 1.10 2.20 3.30 4.40 5.50 0.00 8
```

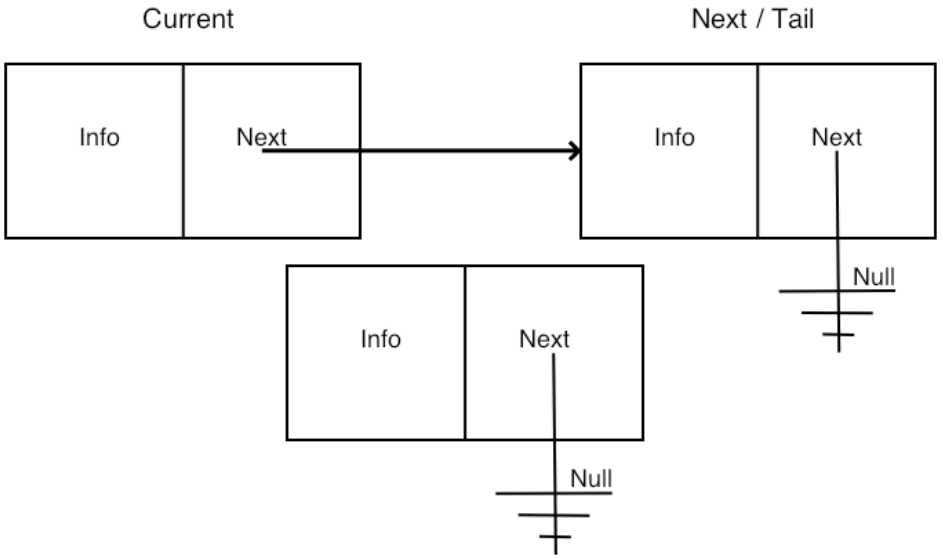
Linked list

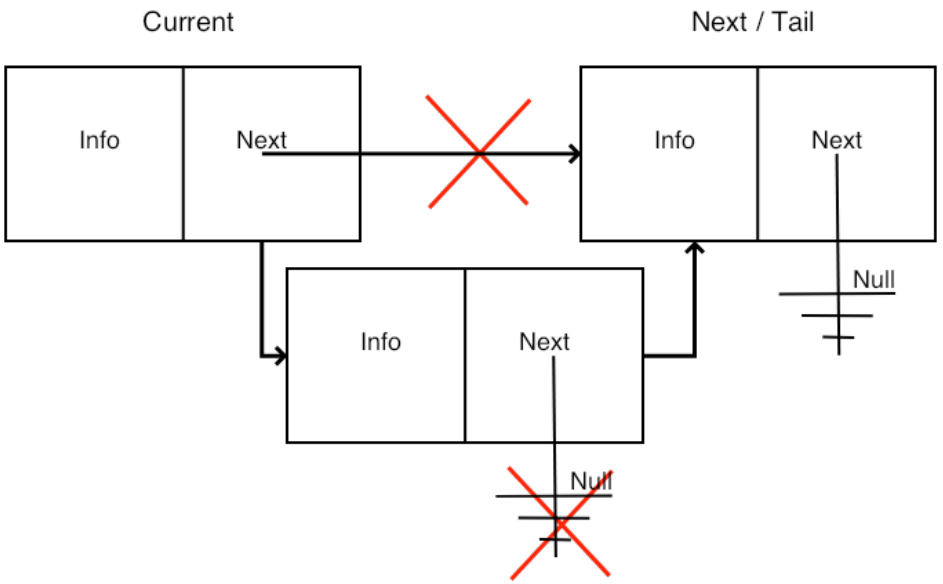
- Linear data structure
- more flexible for insertion / deletion
- . . . but slower in access

Linked list



Insertion


Current
Next / Tail
Info
Next
Info
Next
Null
Info
Next
Null


Current
Next / Tail
Info
Next
Info
Next
Null
Info
Next
Null

Linked list datatypes

- Node: value field, and pointer to next node.
- List: pointer to head node.

```
type node
   integer :: value
   type(node),pointer :: next
end type node

type list
   type(node),pointer :: head
end type list

type(list) :: the_list
nullify(the_list%head)
```

List initialization

First element becomes the list head:

```
allocate(new_node)
new_node%value = value
nullify(new_node%next)
the_list%head => new_node
```

Attaching a node

Keep the list sorted: new largest element attached at the end.

```
allocate(new_node)
new_node%value = value
nullify(new_node%next)
current%next => new_node
```

Exercise 2

Write function `attach` that takes an integer, and attaches a new node at the end of the list with that value.

Inserting 1

Find the insertion point:

```
current => the_list%head ; nullify(previous)
do while ( current%value<value &
     .and. associated(current%next) )
   previous => current
   current => current%next
end do
```

Inserting 2

The actual insertion requires rerouting some pointers:

```
allocate(new_node)
new_node%value = value
new_node%next => current
previous%next => new_node
```

Exercise 3

Write a `print` function for the linked list.
For the simplest solution, print each element on a line by itself.

More sophisticated: use the `Write` function and the `advance` keyword:

```
write(*,'(i1",")',advance="no") current%value
```

Exercise 4

Write a *length* function for the linked list.
Try it both with a loop, and recursively.