Fortran pointers

Victor Eijkhout, Susan Lindsey

Fall 2022

last formatted: March 27, 2023

1. Fortran Pointers

- A pointer is a variable that points at a variable of some type: elementary, or derived types. (but not pointers)
- You can access and change the value of a variable through a pointer that points at it.
- You can change what variable the pointer points at.
- A pointer acts like an alias:
  no explicit dereference needed.

2. Setting the pointer

- You have to declare that a variable is point-able:

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

- Declare a pointer:

```
real,pointer :: point_at_real
```

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

```
point_at_real => x
```

3. 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:

```
1 real,target :: x
2 real,pointer :: point_at_real
3
4 x = 1.2
5 point_at_real => x
6 print *,point_at_real
```

Output:

```
 1.20000005
```

4. Pointer example

Code:

```
1  real,target :: x,y
2  real,pointer :: that_real
3
4  x = 1.2
5  y = 2.4
6  that_real => x
7  print *,that_real
8  that_real => y
9  print *,that_real
10 y = x
11 print *,that_real
```

Output:

```
1.20000005
2.40000010
1.20000005
```

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

5. 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

6. Assignment subtleties

What happens if you want to write *p2*=>*p1*
but you write *p2*=*p1*?
The second one is legal, but has different meaning:

Assign underlying variables:

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

x = 1.2
p1 => x
p2 => y
p2 = p1 ! same as y=x
print *,p2 ! same as print y
```

Crash because *p2* pointer unassociated:

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

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

7. Pointer status

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

Code:

```
1  real,target :: x
2  real,pointer :: realp
3
4  print *,"Pointer starts as not set"
5  if (.not.associated(realp)) &
6     print *,"Pointer not associated"
7  x = 1.2
8  print *,"Set pointer"
9  realp => x
10 if (associated(realp)) &
11    print *,"Pointer points"
12 print *,"Unset pointer"
13 nullify(realp)
14 if (.not.associated(realp)) &
15    print *,"Pointer not associated"
```

Output:

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

8. Pointer allocation

If you want a pointer to point at something,
but you don't need a variable for that something:

Code:

```
1 Real,pointer :: x_ptr,y_ptr
2 allocate(x_ptr)
3 y_ptr => x_ptr
4 x_ptr = 6
5 print *,y_ptr
```

Output:

```
   6.00000000
```

(Compare `make_shared` in C++)

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:

```
1  real,dimension(10),target :: array &
2       = [1.1, 2.2, 3.3, 4.4, 5.5, &
3          9.9, 8.8, 7.7, 6.6, 0.0]
4  real,pointer :: biggest_element
5
6  print '(10f5.2)',array
7  call
       SetPointer(array,biggest_element)
8  print *,"Biggest element
       is",biggest_element
9  print *,"checking pointerhood:",&
10      associated(biggest_element)
11 biggest_element = 0
12 print '(10f5.2)',array
```

Output:

```
 1.10 2.20 3.30 4.40
     5.50 9.90 8.80
     7.70 6.60 0.00
 Biggest element is
     9.89999962
 checking pointerhood: T
 1.10 2.20 3.30 4.40
     5.50 0.00 8.80
     7.70 6.60 0.00
```

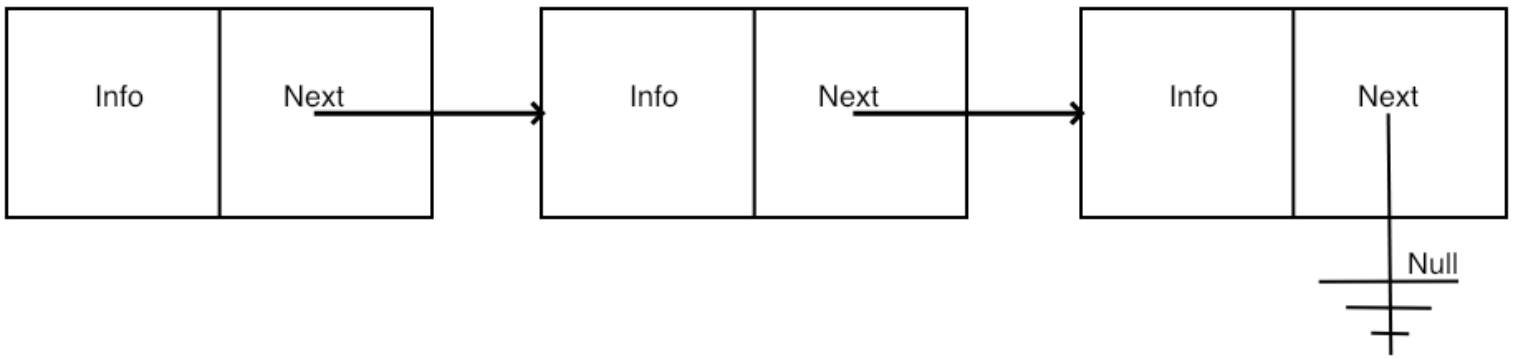
You can base this off the file `arpointf.F90` in the repository

Linked lists

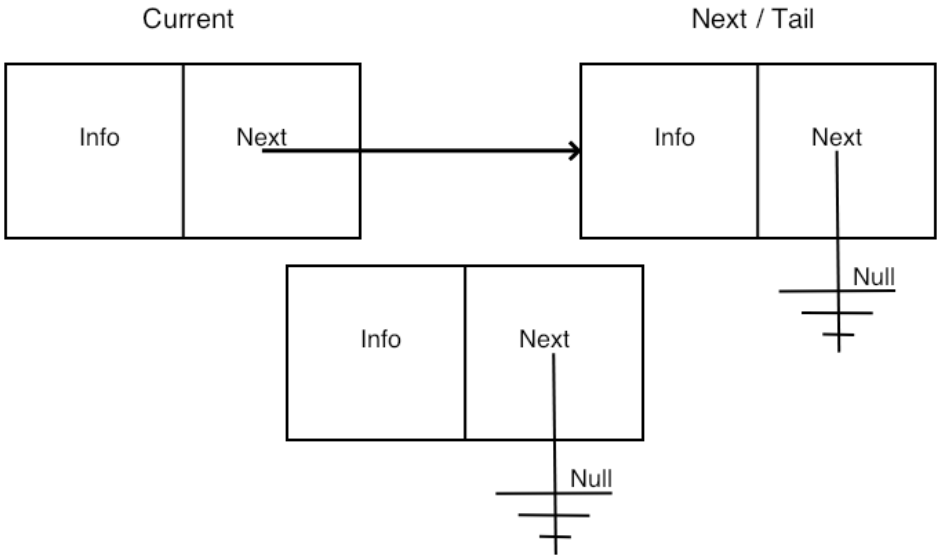
9. Linked list

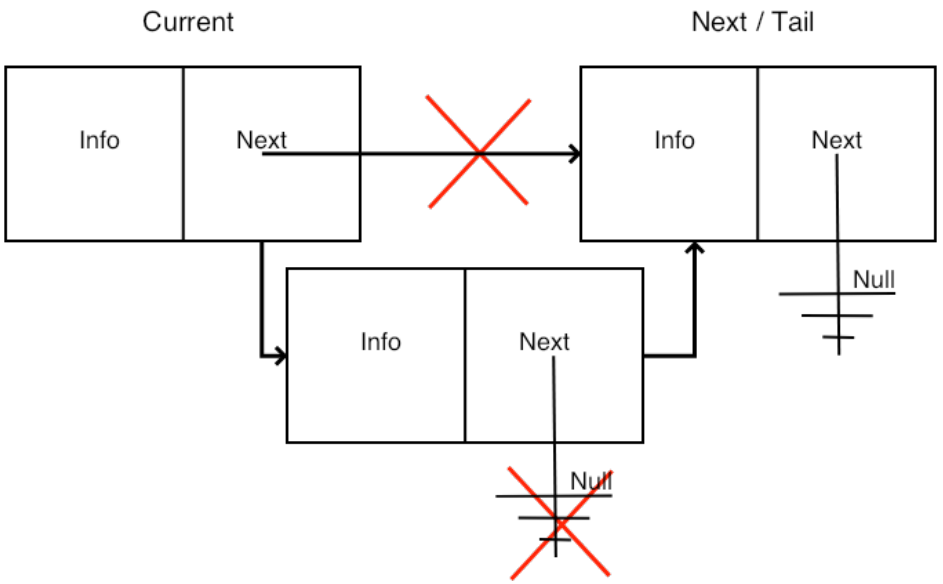
- Linear data structure
- more flexible than array for insertion / deletion
- ... but slower in access

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

10. 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
```

11. Sample main

Our main program will create three nodes, and append them to the end of the list:

Code:

```
1  integer,parameter :: listsize=7
2  type(list) :: the_list
3  integer,dimension(listsize) ::
       inputs = &
4       [ 62, 75, 51, 12, 14, 15, 16 ]
5  integer :: input,input_value
6
7  nullify(the_list%head)
8  do input=1,listsize
9     input_value = inputs(input)
10    call attach(the_list,input_value)
11 end do
```

Output:

```
List: [
      62,75,51,12,14,15,16,
      ]
```

12. List initialization

```
subroutine attach( the_list,new_value )
  implicit none
  ! parameters
  type(list),intent(inout) :: the_list
  integer,intent(in) :: new_value
```

First element becomes the list head:

```
! if the list has no head node, attached the new node
if (.not.associated(the_list%head)) then
   allocate( the_list%head )
   the_list%head%value = new_value
else
   call node_attach( the_list%head,new_value )
end if
```

13. Attaching a node

New element attached at the end.

```
recursive subroutine node_attach( the_node,new_value )
!! ...
  if ( .not. associated(the_node%next) ) then
     allocate( the_node%next )
     the_node%next%value = new_value
  else
     call node_attach( the_node%next,new_value )
  end if
```

Exercise 2

Take the recursive code for attaching an element, and turn it into an iterative version, that is, use a `while` loop that goes down the list till the end.

You may do the whole thing in the *attach* routine for the list head.

14. Main for inserting

Almost the same as before, but now keep the list sorted:

Code:

```
1 do in=1,listsize
2    in_value = inputs(in)
3    call insert(the_list,in_value)
4    call print(the_list)
5 end do
```

Output:

```
List: [  62  ]
List: [  62  75  ]
List: [  51  62  75  ]
List: [  12  51  62  75
      ]
List: [  12  14  51  62
      75  ]
List: [  12  14  15  51
      62  75  ]
List: [  12  14  15  16
      51  62  75  ]
```

Exercise 3

Copy the *attach* routine to *insert*, and modify it so that inserting a value will keep the list ordered.

You can base this off the file `listfappendalloc.F90` *in the repository*