

Fortran pointers

Victor Eijkhout, Susan Lindsey

Fall 2022

last formatted: November 15, 2022

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 pointable:

```
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:

```
real,target :: x
real,pointer :: point_at_real

x = 1.2
point_at_real => x
print *,point_at_real
```

Output:

1.20000005

4. 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:

```
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 $p2 \Rightarrow p1$
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:

```
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:

```
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:

```
Real,pointer :: x_ptr,y_ptr  
allocate(x_ptr)  
y_ptr => x_ptr  
x_ptr = 6  
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:

```
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  
print *, "checking pointerhood:",&  
    associated(biggest_element)  
biggest_element = 0  
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
```

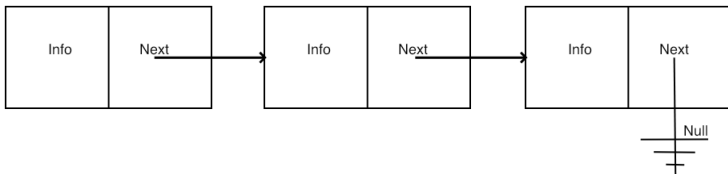
You can base this off the file `arpointf.F90` in the repository

Linked lists

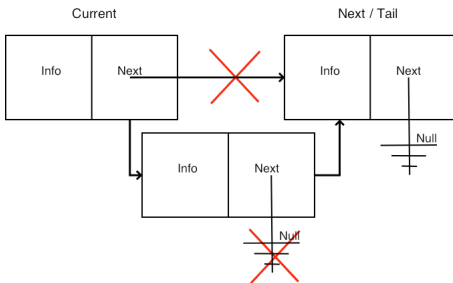
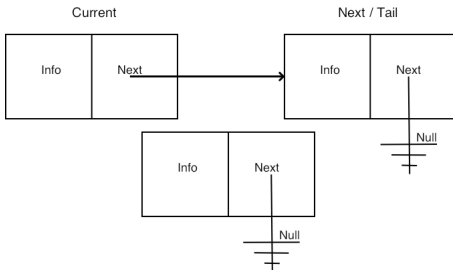
9. Linked list

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

Linked list



Insertion



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:

```
integer,parameter :: listsize=7
type(list) :: the_list
integer,dimension(listsize) ::
    inputs = &
    [ 62, 75, 51, 12, 14, 15, 16 ]
integer :: input,input_value

nullify(the_list%head)
do input=1,listsize
    input_value = inputs(input)
    call attach(the_list,input_value)
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
```

13. Attaching a node

New element attached at the end. **missing snippet**
listattachrecur

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.

14. Main for inserting

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

Code:

```
allocate(node_ptr); node_ptr%value  
    = 1  
call insert(the_list,node_ptr)  
allocate(node_ptr); node_ptr%value  
    = 5  
call insert(the_list,node_ptr)  
allocate(node_ptr); node_ptr%value  
    = 3  
call insert(the_list,node_ptr)  
  
call print(the_list)
```

Output:

List: [1,3,5,]

Exercise 3

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

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

Exercise 4

Modify your code from exercise 3 so that the new node is not allocated in the main program.

Instead, pass only the integer argument, and use `allocate` to create a new node when needed.

```
call insert(the_list,1)
```

```
call insert(the_list,5)
```

```
call insert(the_list,3)
```