

An Introduction to Fortran Pointer Techniques

A. Stock

November 16, 2009

Contents

1	Pointers	1
2	Linked Lists	3
2.1	Formal Definition of a Linked List	3
2.2	Programming a Linked List	4
3	Pointer Arrays	6
3.1	Motivation	6
3.2	Creating a Pointer Array	6
4	Appendix	8
4.1	Code: Linked List	8
4.2	Code: Pointer Array	9

1 Pointers

There are two concepts of variables in FORTRAN:

- the static variable, and
- the dynamic pointer.

Whereas a static variable is declared before the program is executed, a pointer can pointer point to any variable during the execution of the program. A static variable is defined in the beginning of the program:

```
PROGRAM LinkedListExample
  IMPLICIT NONE
  REAL          :: static_variable
END PROGRAM
```

Before executing the program a fixed address in the memory is allocated to contain the data of the variable. During the runtime this address does not change and all operations including this variable have to access the same address in the memory. Figure 1 illustrates the difference.



Figure 1: Static variable vs. dynamic pointer.

A pointer can be defined in the following way:

```
PROGRAM LinkedListExample
  IMPLICIT NONE
  REAL, TARGET      :: static_variable1
  REAL, TARGET      :: static_variable2
  REAL, POINTER     :: pointer1
END PROGRAM
```

When running the program the pointer can be pointed on the targets `static_variable1` or `static_variable2`, such as:

```
static_variable1 = 1
static_variable2 = 2
pointer1 => static_variable1
WRITE(*,*) pointer1
pointer1 => static_variable2
WRITE(*,*) pointer1
```

Figure 2 illustrates that issue. The pointer changes the address during the runtime.

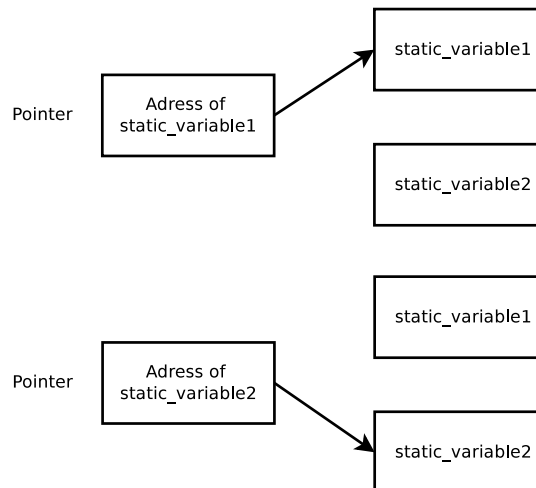


Figure 2: One pointer pointing on two different variables during runtime.

2 Linked Lists

2.1 Formal Definition of a Linked List

The ability to point on different static variables during the runtime seems to be a nice feature but still it is not clear how to use the pointer concept for a more complex problem, such as an array. The array could contain the ID-number of the element in our CFD code (1,2,3,4,5,...). Usually we defined this array statically as:

```
REAL, DIMENSION(5) :: Elements
```

which is a vector of the length five. We can store five ID-numbers of the Elements. The question now is:

How can we create this array with pointers?

In the case of pointers we do not talk about an array any more but about a linked list. Figure 3 shows a linked list based on pointers. We start with the `firstElem` pointer pointing on the

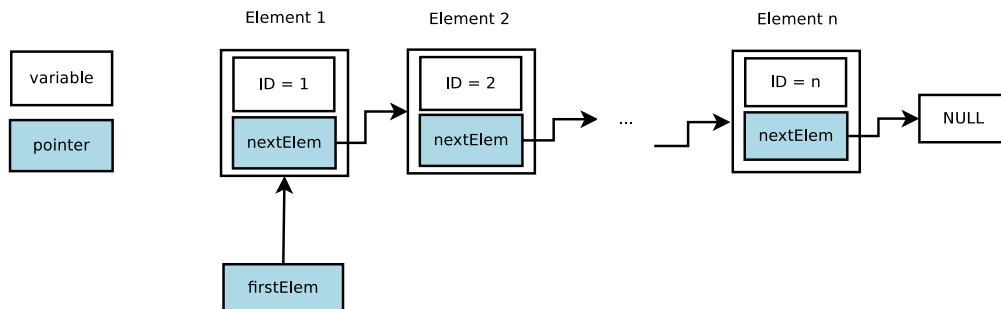


Figure 3: A linked list based on pointers

first element in the list. Each element contains a `nextElem` pointer to the next element and an integer containing the ID of the element. The `nextElem` pointer and the ID integer belong to a type. This might sound confusing, but can be explained in the following way. Usually a the definition of a pointer tells on what type of variable the pointer points. For an integer target this can be:

```
INTEGER, POINTER      :: pointer1
```

For a real variable this is:

```
REAL, POINTER        :: pointer1
```

Since our element contains different types of variables - integer and pointer - we have to define a new type of variable that contains both an integer and an pointer as target of the pointer. Since we are defining elements we call the type `tElem`. The type contains the integer ID and the pointer to the next element:

```

TYPE tElem
  INTEGER          :: ID
  TYPE(tElem), POINTER :: nextElem
END TYPE tElem

```

Since the `nextElem` pointer points on an element it has to be of the type `tElem`. The `firstElem` pointer points on an element. So it is of the type `tElem`. It either can read the ID:

```
firstElem%ID
```

or it can point on the next element:

```
firstElem%nextElem
```

The `firstElem` pointer is not declared in the type `tElem` but of the type `tElem`:

```
TYPE(tElem), POINTER :: firstElem
```

The pointers we have defined so far are static pointers that are pointing on the same targets during runtime. For a dynamic access to the elements we need another pointer, `aElem`. Technically it does the same as the `firstElem` pointer but it is used to dynamically change its target during runtime. The declaration is analogue:

```
TYPE(tElem), POINTER :: aElem
```

To start to create a linked list we point `aElem` on `firstElem`

```
aElem => firstElem
```

Now `aElem` can access data of the first element, such as the ID:

```
aElem%ID
```

To access the next element we have to point `aElem` on the `nextElem` pointer:

```
aElem => aElem%nextElem
```

Now `aElem` points on the next Element and can access data of this element. We can redo this procedure until we reach the end of the linked list. In every element we can access data that we need to compute certain algorithms in our CFD code.

2.2 Programming a Linked List

So far we have introduced the concept of a linked list. Even though we used some code to explain the structure of the linked list we neglected some important formal aspects of programming a linked list. The questions we will answer in this section will be:

- How do we create an element of a linked list?

- How do we define the end of a linked list?
- How can we access the data of a linked list once they are stored in it?

To define an element in our linked list we have to access the memory and write the value of the variable of the type definition into it. Starting with the `firstElem` pointer we allocate the memory where it points to:

```
ALLOCATE( firstElem, STAT=AllocStat)
```

Now we can write data to the memory where the `firstElem` pointer is pointing to:

```
firstElem%ID = 1
```

In order to create a new element we have to allocate the `nextElem` pointer by:

```
ALLOCATE(firstElem%nextElem)
```

Now we can write the ID of the next element into the memory by:

```
firstElem%nextElem%ID = 2
```

However, this is not what we are doing if we have an entire list of element. There we start to use the `aElem` pointer to go through our list. We point it on the first element by:

```
aElem => firstElem
```

`aElem` now points on the `firstElem` pointer which points on the address in the memory which contains the information of the first element and which contains a pointer to the next element. Since we already defined the first element we want to define the next element. Since this operation is the same in each element we can make loop over all elements:

```
DO i = 2, 5
  ALLOCATE(Elem%nextElem)
  Elem%nextElem%ID = i
  Elem => Elem%nextElem
END DO
```

In the last element the `nextElem` pointer has not been linked with an element. To make sure that it does not point on a random part in the memory we have to nullify it by:

```
NULLIFY(aElem%nextElem)
```

We can now test if the `nextElem` pointer of an element is associated by:

```
ASSOCIATED(aElem%nextElem)
```

If we have nullified it the answer will be false (F). If we would not have nullified it `nextElem` could still be pointing on a random position in the memory. In FORTRAN this can happen if

pointers are not deallocated. If we shorten a list for example this might occur. The pointer is not used any more but still there. Thus it is a good practice guideline to nullify the pointer. If we do not take care that the new pointer is nullified it might happen that the information of the old pointer still is used in the new pointer, which creates a wrong linking in the list. This issue is critical since FORTRAN does not given error message if this error occurs, but goes on with the execution of the program. If we are lucky a segmentation fault occurs due to pointing on a dead memory area. If we are not lucky the program goes on and we receive wrong results. Thus we have to delete the old pointer precautionary by nullifying it.

Once we are sure that our list has a beginning and an end we can read the data by going through it in the same fashion as for the creating of the list:

```
aElem => firstElem
DO WHILE (ASSOCIATED(aElem))
  WRITE(*,*) aElem%ID
  aElem => aElem%nextElem
END DO
```

By controlling if `aElem` is associated we do not need to know how big the list is. Since we ensured that the last `nextElem` pointer is nullified will receive a false for the associated inquiry and the loop will end. This is an advantage since we do not need to know the length of the list. We can add or subtract elements of the list without changing the definition of the loop. This allows us to dynamically create elements or delete them. Mesh refinement is a typical application for this feature.

3 Pointer Arrays

3.1 Motivation

In this section we will describe the application of a pointer array. A pointer array is a combination of the new pointer techniques with the standard array architecture. We use a static array to store pointers. These pointers are pointing on elements. This concept implies that we lose the ability of dynamically creating and deleting elements, as we can do for the linked list. Thus we have to motivate why we are doing the backward step.

To read the data of a specific element in the linked list we have to start at the first element and go through the list until we reach the element. If we need to read 5 not sequent elements out of a list of 1000 elements, we have to go 5 times from the first element through the list. It is obvious that this procedure is inefficient. To avoid this problem we can create a pointer for each element and store them in an pointer array. This is useful for the implementation of boundary condition. If only a certain number of elements are located at a boundary we can directly go to them and apply the boundary conditions. Another issue is parallelization. Most domain decomposition routines only work with arrays.

3.2 Creating a Pointer Array

To create a pointer array we have to define this array. In FORTRAN we can only create an pointer array for a special type of pointers. Thus we have to create a new type of pointers. The

type is named `tElemPtr`. The pointers in this type definition are of the type `tElem`. We declare them by:

```
TYPE tElemPtr
  TYPE(tElem), POINTER      :: Elem
END TYPE tElemPtr
```

The pointer array is defined by:

```
TYPE(tElemPtr), ALLOCATABLE :: Elems(:)
```

The array is of the type `tElemPtr` containing pointer defined in this type, but being of type `tElem`. This construction links the pointer array type with the elements type.

The entire variable declaration is given by:

```
TYPE tElem
  INTEGER                :: ID
  TYPE(tElem), POINTER   :: nextElem
END TYPE tElem

TYPE tElemPtr
  TYPE(tElem), POINTER   :: Elem
END TYPE tElemPtr

TYPE(tElemPtr), ALLOCATABLE :: Elems(:)
TYPE(tElem), POINTER        :: firstElem, aElem
INTEGER                      :: i, AllocStat, nElems
```

We extend the method for linked lists to create a pointer array. We have to allocate the first element and to define the length of the pointer array. Then we need to allocate it.

```
ALLOCATE( firstElem, STAT=AllocStat)
nElems = 15
ALLOCATE(Elems(1:nElems))
```

The first element is defined manually again:

```
firstElem%ID = 1
aElem => firstElem
Elems(1)%Elem => aElem
```

To store a pointer in the array we point with the `Elems(1)%Elem` pointer on the `aElem` of our current element. The loop over all elements is analogue to the linked list, but we now also store the pointer in the array:

```
DO i = 2,15
  ALLOCATE(aElem%nextElem)
  aElem%nextElem%ID = i
```

```

    aElem => aElem%nextElem
    Elems(i)%Elem => aElem
END DO

```

To read the data from the array we use a loop over the entries:

```

DO i = 1,nElems
    WRITE(*,*) Elems(i)%Elem%ID
END DO

```

4 Appendix

4.1 Code: Linked List

```

PROGRAM PtrLinkedList

IMPLICIT NONE
! -----!
! Basic definition of Pointers:
TYPE tElem
    INTEGER          :: ID
    TYPE(tElem), POINTER :: nextElem
END TYPE tElem
! -----!
TYPE(tElem), POINTER :: firstElem, aElem
INTEGER          :: i, AllocStat

ALLOCATE( firstElem, STAT=AllocStat)

firstElem%ID = 1

! Point Elem Pointer on first Elem:
aElem => firstElem

DO i = 2, 15
    ALLOCATE(aElem%nextElem)
    aElem%nextElem%ID = i
    aElem => aElem%nextElem
END DO
! Make shure that all memory connections are deleted:
NULLIFY(aElem%nextElem)

WRITE(*,*) 'Loop over all Elements:'

aElem => firstElem
DO WHILE(ASSOCIATED(aElem))
    WRITE(*,*) aElem%ID

```

```

        aElem => aElem%nextElem
    END DO

END PROGRAM

```

4.2 Code: Pointer Array

```

PROGRAM PtrArray

    IMPLICIT NONE
    ! -----!
    ! Basic definition of Pointers:
    TYPE tElem
        INTEGER          :: ID
        TYPE(tElem), POINTER :: nextElem
    END TYPE tElem
    TYPE tElemPtr ! For use of MPI-Comm-Elem-Pointer-List
        TYPE(tElem), POINTER :: Elem          ! Pointer to an Element
    END TYPE tElemPtr
    ! -----!
    TYPE(tElem), POINTER      :: firstElem, aElem, tmpElem
    INTEGER                   :: i, AllocStat, nElems
    TYPE(tElemPtr), ALLOCATABLE :: Elems(:) ! Array of Pointers

    ALLOCATE( firstElem, STAT=AllocStat)

    nElems = 15
    ALLOCATE(Elems(1:nElems))

    firstElem%ID = 1

    ! Point Elem Pointer on first Elem:
    aElem => firstElem
    Elems(1)%Elem => aElem

    DO i = 2,15
        ALLOCATE(aElem%nextElem)
        aElem%nextElem%ID = i
        aElem => aElem%nextElem
        Elems(i)%Elem => aElem
    END DO

    ! Make shure that all memory connections are deleted:
    NULLIFY(aElem%nextElem)

    WRITE(*,*) 'Loop over all Elements:'

```

```
DO i = 1,nElems
  WRITE(*,*) Elems(i)%Elem%ID
END DO
END PROGRAM
```