Pointers and allocatables

Arjen Markus

Deltares

July 4, 2020

Pointers and allocatables

In many cases pointers are not necessary:

For allocating memory, allocatables are quite suitable and they are *automatically* cleaned up – no memory leaks!

Pointers are suitable if you need to select a non-contiguous part of an array:

```
real, dimension(:,:), target  :: array
real, dimension(:,:), pointer :: p

p => array(1:10:2,2:30:4)     ! Contrived, but possible
```

Or with dynamic data structures like a linked list or a binary tree:

```
type linked_list
    type(element_data)         :: data
    type(linked_list), pointer :: next => null()
end type
```

Cleaning up of allocatables

Allocatable arrays (and scalars) are automatically deallocated, when the routine containing them returns.
It is a matter of taste whether you do it yourself:

```
! Local array
real, dimension(:,:), allocatable :: array

allocate( array(10000) )
... do some work
deallocate( array ) ! Not really necessary
return
```

This is possible because the lifetime is well defined.

That is not the case with pointers

Pointers in Fortran and C++

Pointers in Fortran carry a lot of information:

```
real, dimension(:,:), target  :: array
real, dimension(:,:), pointer :: p

p => array(1:10:2,2:30:4)    ! Contrived, but possible
```

The size of p and the shape are respectively 40 and `[5,8]`

In C and C++ a pointer is simply the starting address. There is no information about the size.

As a consequence working with pointers in Fortran is much safer – even though they can often be avoided.

Limitations of pointers in Fortran

Pointers in Fortran have several limitations with respect to C/C++ pointers:

- There is no pointer arithmetic in Fortran
- It is harder/impossible to convert pointers for one data type to pointers for another type. You need to use such objects as unlimited polymorphic variables.