Derived Types and Pointers

Carlos Cruz
Jules Kouatchou
Bruce Van Aartsen

NASA GSFC Code 606 (ASTG)
Greenbelt Maryland 20771

October24, 2018

Derived Data Types

- A Derived Data Type is sometimes called a Data Structure. It allows you to group data objects of different types into one record.
- For instance, if you want to describe the attributes of the weather at point in time, you might create:

```
TYPE WeatherOb
   character(len=10) :: skyCond
   real :: tempC, dewptC, pressHPa
   integer :: windDir, windKt, windGust
END TYPE WeatherOb
```

- Then to use this data type, declare it with:

```
TYPE(WeatherOb) :: wx12ZKLFI
```

- Or create an array of this type with:

```
TYPE(WeatherOb), dimension(24) :: wx24OctKLFI
```

Defining Values

- You may specify default values during declaration:

```
TYPE WeatherOb
   character(len=20) :: skyCond = 'CLR'
   real :: tempC = 0., dewptC = 0., pressHPa = 1013.2
   integer :: windDir = 0, windKt = 0, windGust = 0
END TYPE WeatherOb
```

- Assign values with the Constructor syntax, in order of definition:

```
wx12ZKLFI = WeatherOb('OVC025', 20., 15., 1021.5, 210, 16, 24)
```

Or by using Keywords:

```
wx12ZKLFI = WeatherOb(skyCond='OVC025',tempC=20,dewptC=15, ...
```

Component Selection

- After variable declaration, you can access individual components by using the selector “%” followed by the component name:

```
TYPE(WeatherOb), DIMENSION(24) :: wx24OctKLFI
wx24OctKLFI(12)%tempC = 22.5
wx24OctKLFI(12)%windKt = 12

maxTemp = MAXVAL(wx24OctKLFI(:)%tempC)
```

- You can also assign values of complete derived types to others of the same type:

```
TYPE(WeatherOb), DIMENSION(24) :: wx24OctKLFI, wx24OctKORF
...
wx24OctKLFI = wx24OctKORF
```

Nesting Derived Data Types

- You can also use a Derived Data Type as a component of another Derived Data Type.

```
TYPE WindOb
   integer :: direction, speedKt, gust
END TYPE WindOb

TYPE WeatherOb
   character(len=10) :: skyCond
   real :: tempC, dewptC, pressHPa
   TYPE (WindOb) :: wind
END TYPE WeatherOb
```

- The individual WindOb components are still accessible:

```
TYPE(WeatherOb), dimension(24) :: wx24OctKIAD
wx24OctKIAD(1)%wind%speedKt = 12
```

- And the constructors would look like this:

```
wx24OctKIAD(1) = WeatherOb('OVC025', 20., 15., 1021.5, WindOb(210, 16, 24))
wx24OctKAID(1)%wind = WindOb(210, 16, 24)
```

I/O on Derived Types

- Normal I/O operations can be performed with individual components:

```
TYPE(WeatherOb) :: wx12Z
PRINT *, wx12Z%tempC
```

Results:

```
20.000000
```

- You can also print the entire structure at once:

```
PRINT *, wx24Z
```

Results:

```
OVC025 20.000000 15.000000 1021.5000 210 16 24
```

Derived Type Example

```
module NestedTypes
   TYPE WindOb
      integer :: windDir
      real    :: windMps
   END TYPE WindOb

   TYPE WeatherOb
      real :: tempK
      real :: humidity
      real :: precip
      TYPE (WindOb) :: wind
   END TYPE WeatherOb
end module Nested
```

```
program PrintObs
   use NestedTypes
   implicit none

   TYPE(WeatherOb), dimension(10) :: wxOb
   integer :: i

   open(unit=10, file='WeatherObs.txt')
   do i = 1, 10
      read(10,*) wxOb(i)%tempK, wxOb(i)%humidity,&
        & wxOb(i)%precip, wxOb(i)%wind%windMps,  &
        & wxOb(i)%wind%windDir
   end do

   print *, 'Temperature:', wxOb(5)%tempK, 'K'
   print *, 'Humidity:', wxOb(5)%humidity
   print *, 'Precip:', wxOb(5)%precip, 'cm'
   print *, 'Windspeed:', wxOb(5)%wind%windMps, &
          & 'm/s'
   print *, 'Wind from:', wxOb(5)%wind%windDir, &
          & 'degrees'

end program PrintObs
```

Hidden Components

- When used within a module, you can restrict access to components of the derived data type by declaring them private. This software engineering technique will only allow internal module procedures to modify the components, normally by using **setter** and **getter** functions.

```
MODULE Polygon
   implicit none

   TYPE :: Circle
      PRIVATE
      real :: radius
   END TYPE Circle

CONTAINS
   real function setCircleRadius(this, radius)
   ...
   real function circleArea(this)
   ...
END MODULE
```

Pointers

- In Fortran, a pointer is a data object that contains information about a particular object, like type, rank, and extents, as well as memory address.
- The two most important benefits of using pointers are:
  - Provides a more flexible alternative to allocatable arrays
  - It can enable linked lists, and other dynamic data structures
- A pointer can point to
  - An area of dynamically allocated memory.
  - A data object of the same type as the pointer, with the **TARGET** attribute

- A Fortran Pointer is declared by adding the POINTER attribute, as shown:

```
integer, POINTER :: p1                      !pointer to integer
real, POINTER, dimension(:) :: pa1          !pointer to real array
real, POINTER, dimension(:,:) :: pa2        !pointer to 2-dim real array
```

- The ALLOCATE statement is used to dynamically allocate space for a pointer object:

```
integer, POINTER :: p1
ALLOCATE(p1)
```

Targets and Association

- A target is another normal variable, with space allocated for it. A target variable must be declared with the **TARGET** attribute.
- You associate a pointer variable with a target variable using the association operator (=>):

```
integer, POINTER :: p1
integer, TARGET  :: t1
p1=>t1
```

- Now any operation performed on p1 is also performed on t1
- To remove the association, use the **NULLIFY** statement, and check the status with the **ASSOCIATED** command:

```
NULLIFY(p1)
PRINT *, ASSOCIATED(p1, t1), ASSOCIATED(p1)
```

- Result: **F F**

Pointers Example

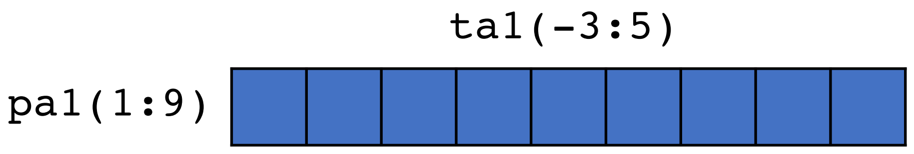
```
program PointerCheck
   implicit none

   integer, POINTER :: a, b
   integer, TARGET  :: t
   integer :: c

   t = 1
   if (.NOT. ASSOCIATED(a)) a => t
   t = 2
   b => t
   c = a + b

   print *, a, b, t, c

end program PointerCheck
```

- Result: **2 2 2 4**

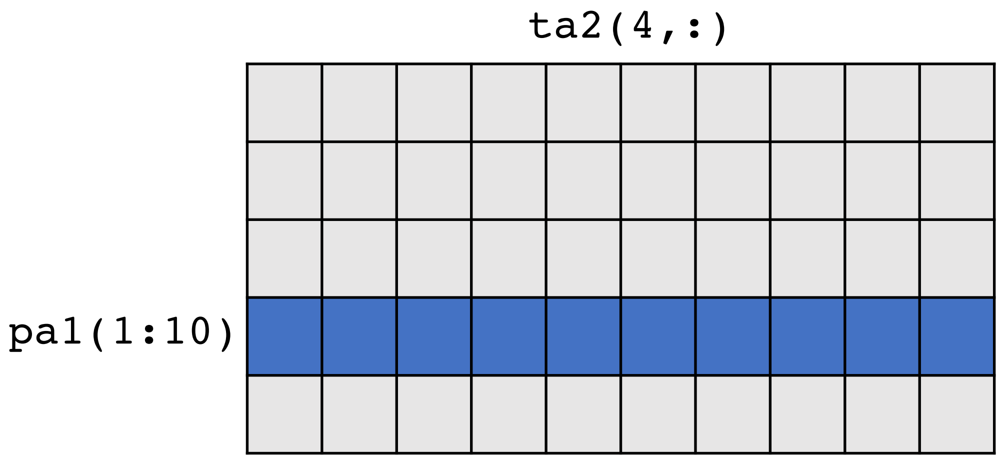
Array Pointers Example

Array pointer indices don't have to align with the array target indices, and can point to just a portion of the target array:

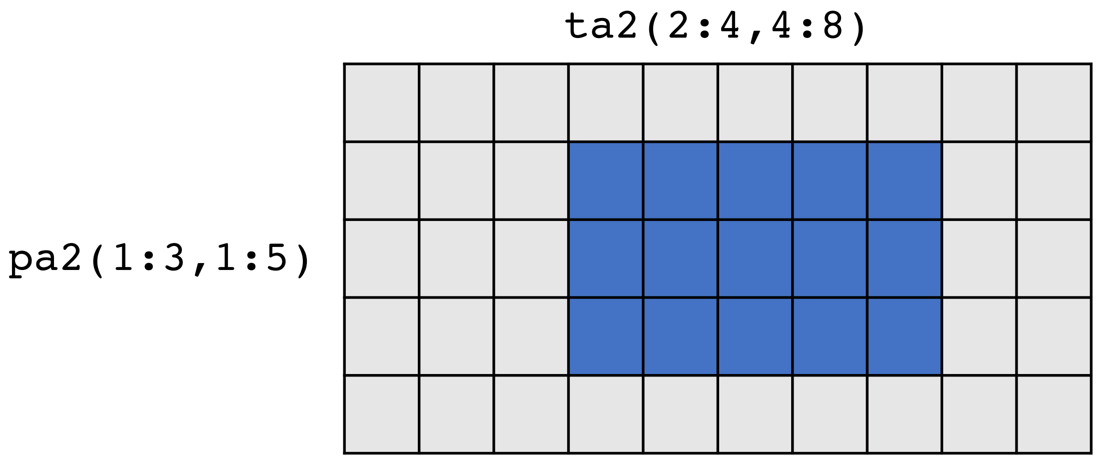
```
real, dimension(:),     POINTER :: pa1
real, dimension(:, :),  POINTER :: pa2
real, dimension(-3:5),   TARGET :: ta1
real, dimension(5, 10),  TARGET :: ta2
```




```
pa1 => ta1(:)  ! pa1 renumbers ta1
```




```
pa1 => ta2(4, :) ! pa1 points to 4th row
```




```
pa2 => ta2(2:4, 4:8) ! pa2 points to section
```

Linked List

We can now take advantage of the features of Derived Types combined with Pointers to set up and manipulate a linked list. In a linked list, the connected objects (nodes):

- are not necessarily stored contiguously,
- can be created dynamically (i.e., at execution time),
- may be inserted at any position in the list,
- may be removed dynamically.

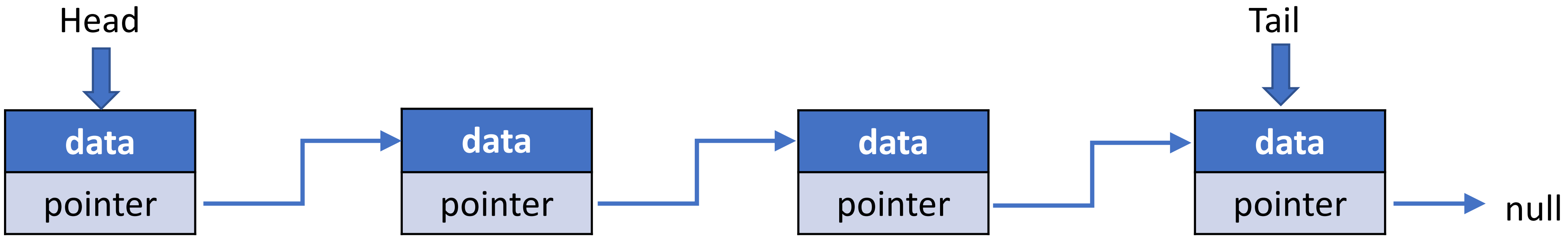
Therefore, the size of a list may grow to an arbitrary size as a program is executing.

A linked list exploits the ability of a derived type to point to an object of the same type:

```
TYPE node
   integer :: value                 ! data field
   TYPE (node), POINTER :: next     ! pointer field
END TYPE node
```

Linked List

We can diagram a linked list like this:



To build up this list, you start with declaring the HEAD node and another CURRENT node

```
TYPE (node), POINTER :: current=>null(), head=>null()
```

Then create a new current node:

```
ALLOCATE(current, STAT = status)
current%value = 1
current%next => head
head => current
```

Repeat process to create a list.

Linked List

```
program LinkedList
implicit none
TYPE node
   integer :: value
   TYPE (node), POINTER :: next
END TYPE node
integer :: num, status
TYPE (node), POINTER :: head, current

! build up the list; initially nullify head
NULLIFY(head)
do
   read *, num          ! read num from keyboard
   if (num == 0) EXIT ! until 0 is entered
   ALLOCATE(current)  ! create new node
   current%value = num
   current%next => head ! point to previous one
   head => current      ! update head of list
end do
```

```
! traverse the list & print the values
current => head     ! start at head
do
   ! exit if null pointer--end of list
   if (.NOT. ASSOCIATED(current)) EXIT
   print*, current%value

   ! make current alias of next node
   current => current%next
end do
end program LinkedList
```