Computer Science 1510

Lecture 19

Lecture Outline

- Pointers

Pointers

- Consider an INTEGER variable a, declared using,

  ```
  INTEGER::a
  ```

  The compiler allocates memory for the variable a, to store an integer value.

- The variable name can then be used to access this location in memory, to either assign a value to that variable, or retrieve the value stored there.

- The variable name a can only ever be used to refer to the specific memory location assigned to it by the compiler.

- A `POINTER` allows a program to reference a variable by a different name. It acts as an alias.

- Whereas a variable name always refers to the same location in memory, a pointer can be reassigned to point to different locations.

Pointers

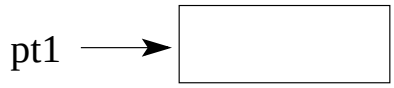
- Syntax:

  ```
  type,POINTER::pointer-variable
  ```

  where `pointer-variable` can be used to access a memory location where a value of the specified type can be stored.

- When a pointer variable is declared it is initially undefined, ie. it does not point to any location in memory.

- The `ALLOCATE` statement can be used to acquire memory locations to associate with pointer variables.



  This memory location is referred to as a *target*.

- Example: `ALLOCATE(pt1)`

Pointers

- Pointers variables are in one of three states:
  - Undefined - As when they are declared.
  - Associated - When pointing to a target.
  - Disassociated - When the association is broken. A pointer in this state is said to be *null*.

- The Fortran function `ASSOCIATED` can be used to test if a pointer is associated with a target.
  Syntax:

  ```
  ASSOCIATED(pointer-variable)
  ```

  This function returns true if `pointer-variable` is associated with a target, and false otherwise.

- The status of a pointer variable can be changed to null using the `NULLIFY` function.
  Syntax:

  ```
  NULLIFY(pointer-variable)
  ```

  Once the association is broken, the memory location pointed to by the pointer variable can no longer be accessed unless pointed to by another pointer variable.

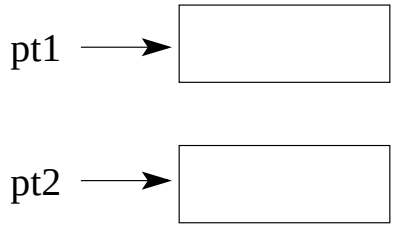
Pointers

- If two pointer variables have the same type, then a pointer assignment statement,

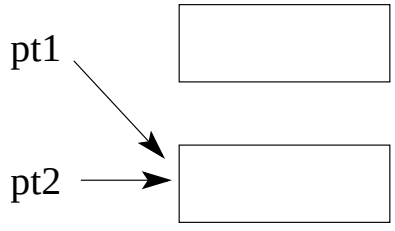
  ```
  pt1 => pt2
  ```

  results in `pt1` pointing to the same memory location as `pt2`.

- Prior to the assignment:



  After the assignment:



Pointers

- Pointer assignment may also be of the form,

```
pointer-variable => target-variable
```

  where `target-variable` has the same type as `pointer-variable`, but it is not a pointer.

- The `target-variable` must have the `TARGET` attribute to allow a pointer to point to its memory location:

```
INTEGER, TARGET :: target_variable
```

- The `ASSOCIATED` function can also be used in the form:

```
ASSOCIATED(pointer,target)
```

  to check whether `pointer` points to `target`.
  It can also be used in the form

```
ASSOCIATED(pointer1,pointer2)
```

  to check whether `pointer1` and `pointer2` point to the same target.

Pointers

- When pointers are used in expressions, they are automatically dereferenced, ie. the value that is stored in the location pointed to by the pointer is used.

- Therefore, if a pointer is assigned a value, the value in its associated memory location is changed.

- Consider the following:

```
INTEGER, TARGET :: i=5
INTEGER, POINTER :: j
j => i
WRITE(*,*) i,j  ! i is 5, what is j?
j=6
WRITE(*,*) i,j  ! j is now 6, what is i?
```

  What is the final value of `i`? of `j`? Why?

Example: Pointers

```
 1 PROGRAM Pt_example
 2    IMPLICIT NONE
 3    INTEGER,POINTER::pt1,pt2
 4    INTEGER,TARGET::a,b
 5    a=10
 6    b=8
 7    pt1=>a
 8    WRITE(*,*) a,b,pt1  ! Prints 10 8 10
 9    a=12
10    WRITE(*,*) a,b,pt1  ! Prints 12 8 12
11    pt1=>b
12    WRITE(*,*) a,b,pt1  ! Prints 12 8 8
13    pt2=>a
14    WRITE(*,*) pt2 ! Prints 12
15    pt1=5
16    WRITE(*,*) a,b,pt1,pt2  ! Prints 12 5 5 12
17    pt2=pt1
18    WRITE(*,*) a,b,pt1,pt2  ! Prints 5 5 5 5
19    a=4
20    WRITE(*,*) a,b,pt1,pt2  ! Prints 4 5 5 4
21    pt2=>pt1
22    WRITE(*,*) a,b,pt1,pt2  ! Prints 4 5 5 5
23    NULLIFY(pt1)
24    WRITE(*,*) pt1 ! Prints 0
25 END PROGRAM Pt_example
```

Line numbers have been added to the program above for referencing individual lines below.

Line 3:

- Both pt1 and pt2 are declared as integer pointers, and are thus not regular variables. They are

currently not associated with a memory location.

- A pointer cannot store a value.
- A pointer can only reference a value of the same type (ex. INTEGER).

Line 4:

- `INTEGER,TARGET::a,b` creates variables that can store integer values, but that also allows pointers to reference (or *point to*) the memory locations where those integers are stored.
- `TARGET` modifies a regular variable such that both the variable name (`a` in this case), and a pointer can access the value stored in memory.

Lines 5 and 6:

- The variable `a` is assigned a value of 10, and `b` is assigned a value of 8.

Line 7:

- Here `pt1` is associated with the memory location allocated for `a`.
- We say that `pt1` *points to* `a`.

Line 8:

- This `WRITE` statement prints out `10 8 10` since `a` was set to 10 (line 5), `b` was set to 8 (line 6), and `pt1` is pointing to the memory location where the value of `a` is stored.

Line 9:

- The variable `a` is reassigned a value of 12.

Line 10:

- This `WRITE` statement prints out `12 8 12` since `a` was set to 12 (line 9), `b` was set to 8 (line 6), and `pt1` is still pointing to the memory location where the value of `a` is stored.

- Recall that a pointer never stores a value, and will therefore print the value currently stored in the memory location that it is pointing to.

Line 11:

- `pt1` is set to point to the value of `b`.

Line 12:

- This `WRITE` statement prints out `12 8 8` since `a` was set to 12 (line 9), `b` was set to 8 (line 6), and `pt1` is pointing to the memory location where the value of `b` is stored.

Line 13:

- `pt2` is set to point to the value of `a`.

Line 14:

- This `WRITE` statement prints out `12` since `pt2` is pointing to the memory location where the value of `a` is stored.

Line 15:

- The value in the memory location referenced by `pt1` (which is `b`) is set to 5.

Line 16:

- This `WRITE` statement prints out `12 5 5 12` since `a` was set to 12 (line 9), `b` was set to 5 (line 15), `pt1`

is pointing to the memory location where the value of `b` is stored, and `pt2` is pointing to the memory location where the value of `a` is stored.

Line 17:

- The value referenced by `pt2` (which is `a`) is assigned the value referenced by `pt1` (which is `b`), ie. `a` is assigned a value of 5.

Line 18:

- This `WRITE` statement prints out `5 5 5 5` since `a` was set to 5 (line 17), `b` was set to 5 (line 15), `pt1` is pointing to the memory location where the value of `b` is stored, and `pt2` is pointing to the memory location where the value of `a` is stored.

Line 19:

- `a` is assigned a value of 4.

Line 20:

- This `WRITE` statement prints out `4 5 5 4` since `a` was set to 4 (line 19), `b` was set to 5 (line 15), `pt1`

is pointing to the memory location where the value of `b` is stored, and `pt2` is pointing to the memory location where the value of `a` is stored.

Line 21:

- `pt2` is set to point to the same location in memory as `pt1`, that is, to the `b` variable.

Line 22:

- This `WRITE` statement prints out `4 5 5 5` since `a` was set to `4` (line 19), `b` was set to `5` (line 15), and both `pt1` and `pt2` are pointing to the memory location where the value of `b` is stored.

Line 23:

- `pt1` is set to null, ie. it does not point to anything.

Line 24:

- This `WRITE` statement prints out `0` since the pointer was nullified on line 23.

Pointers to data structures

- Recall that a pointer variable can be used to access a memory location where a value having the specified type (and attributes) can be stored.

- Previously we have seen pointers for integer numbers.

- A declaration like

  ```
  CHARACTER(8), POINTER :: StringPtr
  ```

  can be used only to access memory locations in which character strings of length 8 reside.

- ```
  TYPE Inventory_Info
  INTEGER :: Number
  REAL :: Price
  END TYPE Inventory_Info

  TYPE(Inventory_Info), POINTER :: InvPtr
  ```

  declares that InvPtr is a pointer variable that can be used to point to locations where structures of type `Inventory_Info` are stored.
  ```

Example: Memory allocation

- Suppose that we would like to be able to increase the size of an array as necessary during run-time.
- Consider the case where we allocate an array of a certain size and double the size if and when we run out of space.
- In this example, we will allocate an array with 5 elements.
- The user will be asked to enter positive integers which will be stored in the array.
- If the user enters a sixth integer then the array will be increased in size to 10 elements.
- If the user enters an 11th element then the array will be increased in size to 20 elements, and so on.
- The program will stop and print out the values entered when the user enters a value of -1.

Example: ALLOCATE

```
PROGRAM Infinite_array
   IMPLICIT NONE
   INTEGER,DIMENSION(:),POINTER::array,newarray
   INTEGER::i,j,value,AllocateStatus

   ALLOCATE(array(1:5),STAT=AllocateStatus)
   IF (AllocateStatus/=0) THEN
      WRITE(*,*) 'Unable to allocate necessary memory'
      STOP
   END IF
   WRITE(*,*) 'Please enter a positive integer, enter -1 to stop'
   READ(*,*) value

   i=1
   DO WHILE (value/=-1)
      IF (i>SIZE(array)) THEN ! The array is too small
         ! Allocate a new array that is twice the size of the old one.
         ALLOCATE(newarray(1:2*SIZE(array)),STAT=AllocateStatus)
         IF (AllocateStatus/=0) THEN
            WRITE(*,*) 'Unable to allocate necessary memory'
            STOP
         END IF
         WRITE(*,*) 'New memory allocated of size ',SIZE(newarray)
         ! Copy the elements from the old array to the new one.
         DO j=1,SIZE(array)
            newarray(j)=array(j)
         END DO
         ! Deallocate the old array
         DEALLOCATE(array,STAT=AllocateStatus)
         IF (AllocateStatus/=0) THEN
            WRITE(*,*) 'Unable to free memory'
            STOP
         END IF
         ! Point the old array pointer to the new array.
         array=>newarray
```

```
      END IF
      ! Add the current value to the array.
      array(i)=value
      i=i+1
      WRITE(*,*) 'Please enter a positive integer, enter -1 to stop'
      READ(*,*) value
   END DO
   WRITE(*,*) (array(i),i=1,SIZE(array))
END PROGRAM Infinite_array
```