21

An Introduction to Pointers

"The question naturally arises whether the analogy can be extended to a data structure corresponding to recursive procedures. A value of such a type would be permitted to contain more than one component that belongs to the same type as itself; in the same way that a recursive procedure can call itself recursively from more than one place in its own body."

C.A.R. Hoare, *Structured Programming*

Aim

The primary aim of the chapter is to introduce some of the key concepts of pointers in Fortran.

21 An Introduction to Pointers

All of the data types introduced so far, with the exception of the allocatable array, have been static. Even with the allocatable array a size has to be set at some stage during program execution. The facilities provided in Fortran by the concept of a pointer combined with those offered by a user defined type enable us to address a completely new problem area, previously extremely difficult to solve in Fortran. There are many problems where one genuinely does not know what requirements there are on the size of a data structure. Linked lists allow sparse matrix problems to be soved with minimal storage requirements, two-dimensional spatial problems can be addressed with quad-trees and three-dimensional spatial problems can be addressed with oct-trees. Many problems also have an irregular nature, and pointer arrays address this problem.

First we need to cover some of the technical aspects of pointers. A pointer is a variable that has the POINTER attribute. A pointer is associated with a target by allocation or pointer assignment. A pointer becomes associated as follows:

- The pointer is allocated as the result of the successful execution of an ALLOCATE statement referencing the pointer

or

- The pointer is pointer-assigned to a target that is associated or is specified with the TARGET attribute and, if allocatable, is currently allocated.

A pointer shall neither be referenced nor defined until it is associated. A pointer is disassociated following execution of a DEALLOCATE or NULLIFY statement, following pointer association with a disassociated pointer, or initially through pointer initialisation.

A pointer may have a pointer association status of associated, disassociated, or undefined. Its association status may change during execution of a program. Unless a pointer is initialised (explicitly or by default), it has an initial association status of undefined. A pointer may be initialised to have an association status of disassociated.

Let us look at some examples to clarify these points.

21.1 Some basic pointer concepts

With the introduction of pointers as a data type into Fortran we also have the introduction of a new assignment statement — the pointer assignment statement. Consider the following example:

```
PROGRAM C2101
   INTEGER , POINTER :: A,B
```

```
    INTEGER , TARGET :: C
    INTEGER :: D
    C = 1
    A => C
    C = 2
    B => C
    D = A + B
    PRINT *,A,B,C,D
END PROGRAM C2101
```

The first declaration defines A and B to be variables, with the POINTER attribute. This means we can use A and B to refer or point to integer values. Note that in this case no space is set aside for the pointer variables A and B. A and B should not be referenced in this state.

The second declaration defines C to be an integer, with the TARGET attribute, i.e., we can use pointers to refer or point to the value of the variable C.

The last declaration defines D to be an ordinary integer variable.

In the case of the last two declarations space is set aside to hold two integers.

Let us now look at the various executable statements in the program, one at a time:

C = 1	This is an example of the normal assignment statement with which we are already familiar. We use the variable name C in our program and whenever we use that name we get the *value* of the variable C.
A => C	This is an example of a pointer assignment statement. This means that both A and C now refer to the same value, in this case 1. A becomes associated with the target C. A can now be referenced.
C = 2	Conventional assignment statement, and C now has the value 2.
B = > C	Second example of pointer assignment. B now points to the value that C has, in this case 2. B becomes associated with the target C. B can now be referenced.
D = A + B	Simple arithmetic assignment statement. The value that A points to is added to the value that B points to and the result is assigned to D.

The last statement prints out the values of A, B, C and D.

The output is

```
 2  2  2  4
```

21.2 The ASSOCIATED intrinsic function

The ASSOCIATED intrinsic returns the association status of a pointer variable. Consider the following example:

```
PROGRAM C2102
INTEGER , POINTER :: A,B
INTEGER , TARGET :: C
INTEGER :: D
   PRINT *,ASSOCIATED(A)
   PRINT *,ASSOCIATED(B)
   C = 1
   A => C
   C = 2
   B => C
   D = A + B
   PRINT *,A,B,C,D
   PRINT *,ASSOCIATED(A)
   PRINT *,ASSOCIATED(B)
END PROGRAM C2102
```

The output from running this program with a number of compilers is shown below.

21.2.1 CVF 6.6C

```
 F
 F
              2              2              2
4
 T
 T
```

21.2.2 Intel, Windows, 8.1

```
F
F
2  2  2  4
T
T
```

21.2.3 Lahey, Windows 5.70f

```
F
F
2  2  2  4
T
```

```
T
```

21.2.4 NAG, Windows, 4.2

```
 T
 T
 2  2  2  4
 T
 T
```

21.2.5 Salford 4.6.0

```
    T
    T
                2               2               2
4
    T
    T
```

We have some differences, and the actual answer as to why is rather subtle. The standard says that the ASSOCIATED function must not be called with a pointer whose status is undefined. So in this program we have declared the pointers A and B but their initial status is undefined. So in a sense all of the above could be regarded as correct, as the program breaks the standard!

The next example is a simple variant.

21.3 Referencing A and B before assignment

Consider the following example:

```
PROGRAM C2103
INTEGER , POINTER :: A,B
INTEGER , TARGET :: C
INTEGER :: D
   PRINT *,ASSOCIATED(A)
   PRINT *,ASSOCIATED(B)
   PRINT *,A
   PRINT *,B
   C = 1
   A => C
   C = 2
   B => C
   D = A + B
   PRINT *,A,B,C,D
   PRINT *,ASSOCIATED(A)
```

```
   PRINT *,ASSOCIATED(B)
END PROGRAM C2103
```

Here we are actually referencing the pointer, even though its status is undefined. Most compilers generate a run time error with this example. However, the error message tends to be a little cryptic. Some sample outputs with the default compilation options follow

21.3.1 CVF

```
 F
 F
forrtl: severe (157): Program Exception - access
violation
Image                  PC          Routine
Line          Source
ch2003cvf.exe        00401098   C2003
7   ch2003.f90
ch2003cvf.exe        004266A9   Unknown
Unknown   Unknown
ch2003cvf.exe        0041D9E4   Unknown
Unknown   Unknown
kernel32.dll         7C816D4F   Unknown
Unknown   Unknown
```

21.3.2 Intel, Windows 8.1

```
 F
 F
forrtl: severe (157): Program Exception - access
violation
Image                  PC          Routine
Line          Source
ch2003intel.exe     0040106E   Unknown
Unknown   Unknown
ch2003intel.exe     0043DE2D   Unknown
Unknown   Unknown
ch2003intel.exe     00430D60   Unknown
Unknown   Unknown
kernel32.dll         7C816D4F   Unknown
Unknown   Unknown
```

21.3.3 Lahey, Windows 5.70f

```
F
F
jwe0019i-u The program was terminated abnormally with
Exception Code EXCEPTION_ACCESS_VIOLATION.
 Error occurs at or near line 6 of _MAIN__
error summary (Fortran)
error number   error level   error count
  jwe0019i           u             1
total error count = 1
```

21.3.4 NAG, Windows 4.2

```
 T
 T
Segmentation fault (core dumped)
```

21.3.5 Salford 4.6.0

```
    T
    T
   -2130131837
               1
               2              2              2
4
    T
    T
```

Some of the compilers give a clue with a line number. The Salford output is interesting as the program actually ran to completion. Try and find compiler options that will provide better diagnostic error messages with your compiler.

21.4 The NULL intrinsic

Fortran 95 introduced the NULL intrinsic. The three previous examples are to a degree examples of Fortran 90 style programming:

```
PROGRAM C2104
INTEGER , POINTER :: A=>NULL(),B=>NULL()
INTEGER , TARGET :: C
INTEGER :: D
  PRINT *,ASSOCIATED(A)
  PRINT *,ASSOCIATED(B)
  C = 1
  A => C
```

```
   C = 2
   B => C
   D = A + B
   PRINT *,A,B,C,D
   PRINT *,ASSOCIATED(A)
   PRINT *,ASSOCIATED(B)
END PROGRAM C2104
```

All compilers tested gave the same correct result. The recommendation is therefore to always use the NULL intrinsic to provide pointer variables with a known value of disassociated, rather than undefined.

21.5 Assignment via =

Consider the following two examples:

```
PROGRAM C2105
INTEGER , POINTER :: A=>NULL(),B=>NULL()
INTEGER , TARGET :: C
INTEGER :: D
   C = 1
   A = 21
   C = 2
   B => C
   D = A + B
   PRINT *,A,B,C,D
END PROGRAM C2105
```

and

```
PROGRAM C2106
INTEGER , POINTER :: A=>NULL(),B=>NULL()
INTEGER , TARGET :: C
INTEGER :: D
   C = 1
   A => C
   C = 2
   B = A
   D = A + B
   PRINT *,A,B,C,D
END PROGRAM C2106
```

Both of these will compile but both will generate run time errors. In the first program the problems lies with the statement

```
   A = 21
```

and in the second case the problem lies with the statement

```
   B = A
```

Below are the corrected versions of the programs

```
PROGRAM C2107
INTEGER , POINTER :: A=>NULL(),B=>NULL()
INTEGER , TARGET :: C
INTEGER :: D
   ALLOCATE(A)
   C = 1
   A = 21
   C = 2
   B => C
   D = A + B
   PRINT *,A,B,C,D
END PROGRAM C2107
```

and

```
PROGRAM C2108
INTEGER , POINTER :: A=>NULL(),B=>NULL()
INTEGER , TARGET :: C
INTEGER :: D
   ALLOCATE(B)
   C = 1
   A => C
   C = 2
   B = A
   D = A + B
   PRINT *,A,B,C,D
END PROGRAM C2108
```

Our recommendation when using pointers is to nullify them when declaring them and to explicitly allocate them before using them when assigning a value via normal assignment.

21.6 Singly linked list

Conceptually a singly linked lists consists of a sequence of boxes with compartments. In the simplest case the first compartment holds a data item and the second contains directions to the next box.

We can construct a data structure in Fortran to work with a singly linked list by combining the concept of a record from the previous chapter with the new concept of a pointer. A complete program to do this is given below:

```
PROGRAM C2109
TYPE Link
   CHARACTER :: C
   TYPE (Link) , POINTER    :: Next
END TYPE Link
TYPE (Link) , POINTER :: Root , Current
INTEGER :: IO_Stat_Number=0
   ALLOCATE(Root)
   READ (UNIT = *, FMT = '(A)' , ADVANCE = 'NO' , &
     IOSTAT = IO_Stat_Number) Root%C
   IF (IO_Stat_Number == -1) THEN
      NULLIFY(Root%Next)
   ELSE
      ALLOCATE(Root%Next)
   ENDIF
   Current => Root
   DO WHILE (ASSOCIATED(Current%Next))
      Current => Current%Next
      READ (UNIT=*,FMT='(A)',ADVANCE='NO', &
      IOSTAT=IO_Stat_Number) Current%C
      IF (IO_Stat_Number == -1) THEN
         NULLIFY(Current%Next)
      ELSE
         ALLOCATE(Current%Next)
      ENDIF
   END DO
   Current => Root
   DO WHILE (ASSOCIATED(Current%Next))
      PRINT * , Current%C
      Current => Current%Next
   END DO
END PROGRAM C2109
```

The behaviour of this program is system specific. You will have to look at your compiler documentation regarding the IO_Stat_Number. The first thing of interest is the type definition for the singly linked list. We have

```
TYPE Link
   CHARACTER :: C
   TYPE (Link) , POINTER :: Next
END TYPE Link
```

and we call the new type `Link`. It comprises two component parts: the first holds a character `C`, and the second holds a pointer called `Next` to allow us to refer to another instance of type `Link`. Remember we are interested in joining together several boxes or `Links`.

The next item of interest is the variable definition. Here we define two variables `Root` and `Current` to be pointers that point to items of type `Link`. In Fortran when we define a variable to be a pointer we also have to define what it is allowed to point to. This is a very useful restriction on pointers, and helps make using them more secure.

The first executable statement

```
ALLOCATE(Root)
```

requests that the variable `Root` be allocated memory. At this time the contents of both the character component and the pointer component are undefined.

The next statement reads a character from the keyboard. We are using a number of additional features of the READ statement, including

```
ADVANCE='NO'
IOSTAT=IO_Stat_Number
```

and the two options combine to provide the ability to read an arbitrary amount of text from the user per line, and terminate only when end of file is encountered as the only input on a line, typically by typing CTRL Z. Note that the numbers returned by the IOSTAT option are implementation specific. A small program would have to be written to test the values returned for each platform.

If an end of file is reached then the pointer `Root%Next` is nullified using the NULLIFY statement. This gives the pointer a status of disassociated, and this is a convenient way of saying that it doesn't point to anything valid.

If the end of file is not detected then the next link in the chain is created.

The statement

```
Current => Root
```

means that both `Current` and `Root` point to the same physical memory location, and this holds a character data item and a pointer. We must do this as we have to know where the start of the list is. This is now our responsibility, not the compilers. Without this statement we are not able to do anything with the list except fill it up — hardly very useful.

The WHILE loop is then repeated until end of file is reached. If the user had typed an end of file immediately then `Current%Next` would not be ASSOCIATED, and the WHILE loop would be skipped.

This loop allocates memory and moves down the chain of boxes one character at a time filling in the links between the boxes as we go. We then have

```
Current => Root
```

and this now means that we are back at the start of the list, and in a position to traverse the list and print out each character in the list.

There is thus the concept with the pointer variable Current of it providing us with a window into memory where the complete linked list is held, and we look at one part of the list at a time.

Both WHILE loops use the intrinsic function ASSOCIATED to check the association status of a pointer.

It is recommended that this program be typed in, compiled and executed. It is surprisingly difficult to believe that it will actually read in a completely arbitrary number of characters from the user. Seeing is believing.

21.7 Reading in an arbitrary quantity of numeric data

In this example we will look at using a singly linked list to read in an arbitrary quantity of data and then allocating an array to copy it to for normal numeric calculations at run time:

```
PROGRAM C2110

TYPE Link
   REAL :: N
   TYPE (Link) , POINTER    :: Next
END TYPE Link

TYPE (Link) , POINTER :: Root, Current
```

```
INTEGER :: I=0
integer :: error=0
INTEGER :: IO_Stat_Number=0
integer :: blank_lines=0

real , allocatable , dimension(:) :: x

   ALLOCATE(Root)
   READ (UNIT = *, FMT = *, IOSTAT = IO_Stat_Number)
Root%N
   IF (IO_Stat_Number > 0) THEN
      error=error+1
   else if (io_stat_number == -1) then
      NULLIFY(Root%Next)
   else if (io_stat_number == -2) then
      blank_lines=blank_lines+1
   ELSE
      i=i+1
      ALLOCATE(Root%Next)
   ENDIF

   Current => Root

   DO WHILE (ASSOCIATED(Current%Next))

      Current => Current%Next

      READ (UNIT=*,FMT=*, IOSTAT=IO_Stat_Number)
Current%N

      IF (IO_Stat_Number > 0) THEN
         error=error+1
      else if (io_stat_number == -1) then
         NULLIFY(current%Next)
      else if (io_stat_number == -2) then
         blank_lines=blank_lines+1
      ELSE
         i=i+1
         ALLOCATE(current%Next)
      ENDIF
```

```
   END DO

   print *,i,' items read'
   print *,blank_lines,' blank lines'
   print *,error,' items in error'

   allocate(x(1:i))
   i=1
   Current => Root

   DO WHILE (ASSOCIATED(Current%Next))
      x(i)=current%n
      i=i+1
      PRINT * , Current%N
      Current => Current%Next
   END DO

   print *,x

END PROGRAM C2110
```

Below is a variant on this using the NAG compiler. Note the use of a module and meaningful names for the status of the read:

```
PROGRAM C2111

use f90_iostat

TYPE Link
   REAL :: N
   TYPE (Link) , POINTER    :: Next
END TYPE Link

TYPE (Link) , POINTER :: Root, Current

INTEGER :: I=0
INTEGER :: IO_Stat_Number=0

   ALLOCATE(Root)
   READ (UNIT = *, FMT = *, IOSTAT = IO_Stat_Number)
Root%N
   if (io_stat_number == ioerr_eof) then
      NULLIFY(Root%Next)
```

```
   ELSE if(io_stat_number == ioerr_ok) then
      i=i+1
      ALLOCATE(Root%Next)
   ENDIF

   Current => Root

   DO WHILE (ASSOCIATED(Current%Next))

      Current => Current%Next

      READ (UNIT=*,FMT=*, IOSTAT=IO_Stat_Number)
Current%N

      if (io_stat_number == ioerr_eof) then
         NULLIFY(current%Next)
      ELSE if(io_stat_number == ioerr_ok) then
         i=i+1
         ALLOCATE(current%Next)
      ENDIF

   END DO

   print *,i,' items read'

   Current => Root

   DO WHILE (ASSOCIATED(Current%Next))
      PRINT * , Current%N
      Current => Current%Next
   END DO

END PROGRAM C2111
```

21.8 Arrays of pointers

Arrays in Fortran are rectangular, even when allocatable. So if you wish to set up a lower triangular matrix that uses minimal memory you have to use arrays of pointers. The following examples show how to do this.

```
PROGRAM C2112
IMPLICIT NONE
TYPE Ragged
```

```
   REAL , DIMENSION(:) , POINTER :: Ragged_row
END TYPE
INTEGER :: i
INTEGER , PARAMETER :: n=3
TYPE (Ragged) , DIMENSION(1:n) :: Lower_Diag
   DO i=1,n
      ALLOCATE(Lower_Diag(i)%Ragged_Row(1:i))
      PRINT *,' Type in the values for row ' , i
      READ *,Lower_Diag(I)%Ragged_Row(1:i)
   END DO
   DO i=1,n
      PRINT *,Lower_Diag(i)%Ragged_Row(1:i)
   END DO
END PROGRAM C2112
```

The type Ragged has a component that is a pointer to an array. Within the first DO loop we allocate a row at a time and each time we go around the loop the array allocated increases in size.

21.9 Arrays of pointers and variable sized data sets — 1

In this example we use a parameter statement to set up the number of stations:

```
PROGRAM C2113
IMPLICIT NONE
TYPE Ragged
  REAL , DIMENSION(:) , POINTER :: rainfall
END TYPE
INTEGER :: i
INTEGER , PARAMETER :: nr=5
INTEGER , DIMENSION (1:nr) :: nc
TYPE (ragged) ,  DIMENSION(1:nr) :: station
  DO i=1,nr
     PRINT *,' enter the number of data values' &
      ' for station ',i
     READ *,nc(i)
     ALLOCATE(station(i)%rainfall(1:nc(i)))
     PRINT *,' Type in the values for station ' , i
     READ *,station(i)%rainfall(1:nc(i))
  END DO
  DO i=1,nr
     PRINT *,station(i)%rainfall(1:nc(i))
  END DO
```

```
END PROGRAM C2013
```

We read in the dimension or number of values for each station at run time, and allocate the space at run time.

21.10 Arrays of pointers and variable sized data sets — 2

In this example the number of stations is read in at run time:

```
PROGRAM C2114
IMPLICIT NONE
TYPE Ragged
  REAL , DIMENSION(:) , POINTER :: rainfall
END TYPE
INTEGER :: i
INTEGER :: nr
iNTEGER , ALLOCATABLE , DIMENSION (:) :: nc
TYPE (ragged) , ALLOCATABLE ,  DIMENSION(:) :: station
  PRINT *,' enter number of stations'
  READ *,nr
  ALLOCATE(station(1:nr))
  ALLOCATE(nc(1:nr))
  DO I=1,Nr
    PRINT *,' enter the number of data values ' &
     ' for station ',i
    READ *,nc(i)
    ALLOCATE(station(i)%rainfall(1:nc(i)))
    PRINT *,' Type in the values for station ' , I
    READ *,station(i)%rainfall(1:nc(i))
  END DO
  DO i=1,nr
    PRINT *,station(i)%rainfall(1:nc(i))
  END DO
END PROGRAM C2114
```

In this example both the number of stations and the dimension for each station is read in at run time and allocated accordingly.

21.11 Memory leak examples

Dynamic memory brings greater versatility but requires greater responsibility. Consider the following example:

```
PROGRAM C2115
IMPLICIT NONE
INTEGER :: Allocate_status=0
REAL , DIMENSION(:) , POINTER :: X
REAL , DIMENSION(1:10) , TARGET :: Y
INTEGER , PARAMETER :: SIZE=10000000
INTEGER :: I
   ALLOCATE(X(1:SIZE),STAT=Allocate_status)
   IF (allocate_status > 0) THEN
      PRINT *,' Allocate failed. Program ends.'
      STOP
   ENDIF
! initialise the memory that x points to
    DO I=1,SIZE
       X(I)=I
    END DO
! print out the first 10 values
    DO I=1,10
       PRINT *,X(I)
    END DO
! initialise the array y
    DO I=1,10
       Y(I)=I*I
    END DO
! print out y
    DO I=1,10
       PRINT *,Y(I)
    END DO
! x now points to y
    X=>Y
! print out what x now points to
    DO I=1,10
       PRINT *,X(I)
    END DO
! what has happened to the memory that x
! used to point to?
END PROGRAM C2115
```

The next is a simple variant on the above:

```
PROGRAM C2116
IMPLICIT NONE
INTEGER :: Allocate_status=0
```

```
REAL , DIMENSION(:) , POINTER :: X
REAL , DIMENSION(1:10) , TARGET :: Y
INTEGER , PARAMETER :: SIZE=10000000
INTEGER :: I
  DO
    ALLOCATE(X(1:SIZE),STAT=Allocate_status)
    IF (allocate_status > 0) THEN
      PRINT *,' Allocate failed. Program ends.'
      STOP
    ENDIF

! initialise the memory that x points to
    DO I=1,SIZE
      X(I)=I
    END DO
! print out the first 10 values
    DO I=1,10
      PRINT *,X(I)
    END DO
! initialise the array y
    DO I=1,10
      Y(I)=I*I
    END DO
! print out y
    DO I=1,10
      PRINT *,Y(I)
    END DO
! x now points to y
    X=>Y
! print out what x now points to
    DO I=1,10
      PRINT *,X(I)
    END DO
! what has happened to the memory that x
! used to point to?
  end do
END PROGRAM C2116
```

Before running this example we recommend starting up a memory monitoring program.

Under Microsoft Windows XP Professional holding [CTRL] + [ALT] + [DEL] will bring up the Windows Task Manager. Choose the [Performance] tab to get a

screen which will show CPU usage, PF Usage, CPU Usage History and Page File Usage History. You will also get details of Physical and Kernel memory usage.

Under Linux type

```
top
```

in a terminal window.

In these examples we also see the recommended form of the ALLOCATE statement when working with arrays. This enables us to test if the allocation has worked and take action accordingly. A positive value indicates an allocation error, zero indicates OK.

21.12 Nonstandard pointer examples

Some Fortran compilers provide a LOC intrinsic. The description from the CVF on line documentation follows:

result = LOC (x)
x (Input) is a variable, an array or a record field reference, a procedure, or a constant; it can be of any data type. It must not be the name of an internal procedure or statement function. If it is a pointer, it must be defined and associated with a target.

This returns the address of the variable passed. Below are four examples that show some of what is happening behind the scenes when using pointer variables. We have also included some sample output:

```
PROGRAM C2117
INTEGER , POINTER :: A,B
INTEGER , TARGET :: C
INTEGER :: D
   PRINT *,LOC(a)
   PRINT *,LOC(b)
   PRINT *,LOC(c)
   PRINT *,LOC(d)
   C = 1
   A => C
   C = 2
   B => C
   D = A + B
   PRINT *,A,B,C,D
   PRINT *,LOC(a)
   PRINT *,LOC(b)
   PRINT *,LOC(c)
```

```
   PRINT *,LOC(d)
END PROGRAM C2117
```

CVF Output:

```
              0
              0
      4424172
      4424168
              2             2             2
4
      4424172
      4424172
      4424172
      4424168
```

Lahey Output:

```
0
0
4456968
4456972
2  2  2  4
4456968
4456968
4456968
4456972
```

The value zero is often used to signify a special memory value in computing. After the pointer assignments it is clear that all three variables point to the same value:

```
PROGRAM C2018
INTEGER , POINTER :: A=>NULL(),B=>NULL()
INTEGER , TARGET :: C
INTEGER :: D
   PRINT *,LOC(a)
   PRINT *,LOC(b)
   PRINT *,LOC(c)
   PRINT *,LOC(d)
   C = 1
   A => C
   C = 2
   B => C
   D = A + B
```

```
   PRINT *,A,B,C,D
   PRINT *,LOC(a)
   PRINT *,LOC(b)
   PRINT *,LOC(c)
   print *,loc(d)
END PROGRAM C2018
```

CVF Output:

```
              0
              0
      4424168
      4424164
              2              2              2
4
      4424168
      4424168
      4424168
      4424164
```

Lahey Output:

```
0
0
4456968
4456972
2  2  2  4
4456968
4456968
4456968
4456972
```

We have again the use of zero as the special memory value:

```
PROGRAM C2119
INTEGER , POINTER :: A=>NULL(),B=>NULL()
INTEGER , TARGET :: C
INTEGER :: D
   PRINT *,LOC(a)
   PRINT *,LOC(b)
   ALLOCATE(a)
   ALLOCATE(b)
   PRINT *,LOC(a)
   PRINT *,LOC(b)
```

```
   PRINT *,LOC(c)
   PRINT *,LOC(d)
   C = 1
   A => C
   C = 2
   B => C
   D = A + B
   PRINT *,A,B,C,D
   PRINT *,LOC(a)
   PRINT *,LOC(b)
   PRINT *,LOC(c)
   PRINT *,LOC(d)
END PROGRAM C2119
```

CVF Output:

```
              0
              0
      3292304
      3292328
      4424152
      4424148
              2              2              2
4
      4424152
      4424152
      4424152
      4424148
```

Lahey Output:

```
0
0
8915968
8916160
4457148
4457152
2  2  2  4
4457148
4457148
4457148
4457152
```

In this example we actually use the ALLOCATE statement to set aside space for the pointers. What is interesting in this example is that the original space set aside becomes lost after the pointer assignments. This indicates a small memory leak:

```
PROGRAM C2020
INTEGER , POINTER :: A=>NULL(),B=>NULL()
INTEGER , TARGET :: C
INTEGER :: D
   PRINT *,LOC(a)
   PRINT *,LOC(b)
   ALLOCATE(a)
   ALLOCATE(b)
   PRINT *,LOC(a)
   PRINT *,LOC(b)
   PRINT *,LOC(c)
   PRINT *,LOC(d)
   C = 1
   A = 21
   C = 2
   B = A
   D = A + B
   PRINT *,A,B,C,D
   PRINT *,LOC(a)
   PRINT *,LOC(b)
   PRINT *,LOC(c)
   PRINT *,LOC(d)
END PROGRAM C2020
```

CVF Output:

```
              0
              0
      3292304
      3292328
      4424152
      4424148
            21            21             2            42
      3292304
      3292328
      4424152
      4424148
```

Lahey Output:

```
0
0
8915968
8916160
4457152
4457156
21 21 2 42
8915968
8916160
4457152
4457156
```

In this case the addresses for A and B remain the same as for a normal assignment, not a pointer assignment.

21.13 Problems

1. Compile and run all of the example programs in this chapter with your compiler and examine the output.

2. There are a number of ways of handling exceptions with the READ statement, and we have used the IOSTAT option in this chapter. Consider the following program:

```
PROGRAM C2102p
INTEGER :: IO_Stat_Number=0
INTEGER :: I
   DO
      READ (UNIT=* , FMT=10 , ADVANCE='NO' &
      , IOSTAT=IO_Stat_Number) I
      10 FORMAT(I3)

! 0   = no error
!       no end of file (eof)
!       no end of record (eor)
! -   = eor or eof
! +   = an error occurred

      PRINT *,' iostat=',IO_Stat_Number
      PRINT *,I
   END DO
END PROGRAM C2102p
```

This program is a simple test of the IOSTAT values of whatever system you work on. Try typing in a variety of values including minimally:

- A valid three-digit number + [RETURN] key.
- A three-digit number with an embedded blank, e.g., 1 2 + [RETURN] key.
- [RETURN] key only.
- [CTRL] + Z.
- Any other non-numeric character on the keyboard.
- 100200300 + [RETURN] key.
- [CTRL] + C

This will enable us to program exactly the kind of behaviour we want from I/O and can be used as a code segment for other programs.