Chapter 18
An Introduction to Pointers

Not to put too fine a point on it

Charles Dickens, Bleak House

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

18.1 Introduction

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 solved 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:

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran*,
https://doi.org/10.1007/978-3-319-75502-1_18

- 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 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.

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.

Let us look at some examples to clarify these points.

18.2 Example 1: Illustrating 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 ch1801
  implicit none
  integer, pointer :: a => null(), b => null()
  integer, target :: c
  integer :: d

  c = 1
  a => c
  c = 2
  b => c
  d = a + b
  print *, a, b, c, d
end program ch1801
```

The following

```
integer , pointer :: a=>null(),b=>null()
```

is a declaration statement that 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. We also use the `null` intrinsic to set the status of the pointers `a` and `b` to disassociated. Using the `null` intrinsic means that we can test the status of a pointer variable and avoid making a number of common pointer programming errors. Note that in this case no space is set aside for the pointer variables `a` and `b`, i.e. `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 4
```

18.3 Example 2: The `associated` Intrinsic Function

The `associated` intrinsic returns the association status of a pointer variable. Consider the following example which is a simple variant on the first.

```
program ch1802
  implicit none
  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 ch1802
```

The output from running this program is shown below

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

and as you can see we therefore have a mechanism to test pointers to see if they are in a valid state before use.

18.4 Example 3: Referencing Pointer Variables Before Allocation or Pointer Assignment

Consider the following example:

```
program ch1803
  implicit none
  integer, pointer :: a => null(), b => null()
```

```
    integer, target :: c
    integer :: d

    print *, a
    print *, b
    c = 1
    a => c
    c = 2
    b => c
    d = a + b
    print *, a, b, c, d
  end program ch1803
```

Here we are actually referencing the pointers `a` and `b`, even though their status is disassociated. Most compilers generate a run time error with this example with the default compiler options, and the error message tends to be a little cryptic. It is recommended that you look at the diagnostic compilation switches for you compiler. We include some sample output below from gfortran, Intel and Nag. The error messages are now much more meaningful.

18.4.1 gfortran

Switches are

```
gfortran -W -Wall -fbounds-check -pedantic-errors
  -std=f2003 -Wunderflow
  -O -fbacktrace -ffpe-trap=zero,
  overflow,underflow -g
```

The program runs to completion with no error message. Here is the output.

```
ch1803.out
           0
           0
           2           2           2           4
```

18.4.2 Intel

Switches are

```
/check:all /traceback
```

Here is the output.

```
D:\document\fortran\newbook\examples\ch18>>
ch1803
forrtl: severe (408): fort: (7):
Attempt to use pointer A when it
is not associated with a target
Image          PC                Routine Line
Source
ch1803.exe     000000013F0AC598  Unknown Unknown
Unknown
...
ntdll.dll      0000000077096611  Unknown Unknown
Unknown
```

18.4.3 Nag

Switches are

```
-C=all -C=undefined -info -g -gline
```

Here is the output.

```
Runtime Error: ch1803.f90, line 5:
Reference to disassociated POINTER A
Program terminated by fatal error
ch1803.f90, line 5: Error occurred in CH1803
```

18.5 Example 4: Pointer Allocation and Assignment

Consider the following example:

```
program ch1804
  implicit none
  integer, pointer :: a => null(), b => null()
  integer, target :: c
  integer :: d
  allocate (a)
  a = 1
  c = 2
  b => c
  d = a + b
  print *, a, b, c, d
  deallocate (a)
end program ch1804
```

In this example we allocate `a` and then can do conventional assignment. If we had not allocated `a` the assignment would be illegal. Try out problem 18.2 to see what will happen with your compiler.

Our simple recommendation when using pointers is to nullify them when declaring them and to explicitly allocate them before conventional assignment.

18.6 Memory Leak Examples

Dynamic memory brings greater versatility but requires greater responsibility.

18.6.1 Example 5: Simple Memory Leak

```
program ch1805
  implicit none
  integer, pointer :: a => null(), b => null()
  integer, target :: c
  integer :: d

  allocate (a)
  allocate (b)
  a = 100
  b = 200
  print *, a, b
  c = 1
  a => c
  c = 2
  b => c
  d = a + b
  print *, a, b, c, d
end program ch1805
```

What has happened to the memory allocated to `a` and `b`?

18.6.2 *Example 6: More Memory Leaks*

Now consider the following example.

```
program ch1806
  implicit none
  integer :: allocate_status = 0
  integer, parameter :: n1 = 10000000
  integer, parameter :: n2 = 5
  integer, dimension (:), pointer :: x
  integer, dimension (1:n2), target :: y
  integer :: i

  do
    allocate (x(1:n1), stat=allocate_status)
    if (allocate_status>0) then
      print *, ' allocate failed. program ends.'
      stop
    end if
    do i = 1, n1
      x(i) = i
    end do
    do i = 1, n2
      print *, x(i)
    end do
    do i = 1, n2
      y(i) = i*i
    end do
    do i = 1, n2
      print *, y(i)
    end do
    x => y             ! x now points to y
    do i = 1, n2
      print *, x(i)
    end do
!   what has happened to the memory that x
!   used to point to?
  end do
end program ch1806
```

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

Under Microsoft Windows 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.

The second program can require a power off on a Windows operating system with a compiler that will remain anonymous!

18.7 Non-standard Pointer Example

Some Fortran compilers provide a non-standard `loc` intrinsic. This can be used to print out the address of the variable passed as an argument.

18.7.1 Example 7: Using the C `loc` Function

Some Fortran compilers provide non standard access to functions supported in the C language. This example uses the C `loc` function.

```
program ch1807
  implicit none
  integer, pointer :: a => null(), b => null()
  integer, target :: c
  integer :: d

  allocate (a)
  allocate (b)
  a = 100
  b = 200
  print *, a, 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 ch1807
```

Here is the output from a compiler with `loc` support.

```
         100         200
              13803552
              13803600
               2948080
               2948084
           2           2           2           4
               2948080
               2948080
               2948080
               2948084
```

This program clearly shows the memory leak.

18.8 Problems

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

18.2 Compile and run example 4 without the `allocate(a)` statement. See what happens with your compiler.

Here is the output from the Nag compiler. The first run is with the default options.

```
nagfor ch1804p.f90
NAG Fortran Compiler:
[NAG Fortran Compiler normal termination]
a.exe
```

There is no meaningful output.

The following adds the -C=all compilation option.

```
nagfor ch1804p.f90 -C=all
NAG Fortran Compiler:
[NAG Fortran Compiler normal termination]
a.exe
Runtime Error: ch1804p.f90, line 5:
Reference to disassociated POINTER
A
Program terminated by fatal error
```

We now get a meaningful error message.