Chapter 22
Data Structuring in Fortran

The good teacher is a guide who helps others to dispense with his services.

R. S. Peters, Ethics and Education

Aims

The aims of this chapter are to look at several complete examples illustrating data structuring in Fortran using the following

- Singly linked lists
- Ragged arrays
- A perfectly balanced tree
- A date data type

22.1 Introduction

This chapter looks at simple data structuring in Fortran using a range of examples. We use modules throughout to define the data structures that we will be working with. The chapter starts with a number of pointer examples.

22.2 Example 1: Singly Linked List: Reading an Unknown Amount of Text

Conceptually a singly linked list 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.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran*,
https://doi.org/10.1007/978-3-319-75502-1_22

In the diagram below we have a singly linked list that holds characters Jane. We assume that the address of the start of the list is 100. We assume 4 bytes per character (a 32 bit word) and 4 bytes per pointer.

- Element 1 is at address 100 and holds the character J and a pointer to the next element at address 108.
- Element 2 holds the character a and a pointer to the next element at address 116.
- Element 3 holds the character n and a pointer to the next element at address 124.
- Element 4 holds the character e and does not point to anything - we use the null pointer.

```
[J : 108] -> [a : 116] -> [n : 124] -> [e : null]
```

We can construct a data structure in Fortran to work with a singly linked list by defining a link data type with two components, a character variable and a pointer variable to a link data type. A complete program to do this is given below:

```
module link_module
  type link
    character (len=1) :: x
    type (link), pointer :: next => null()
  end type link
end module link_module

program ch2201
  use link_module
  implicit none
  character (len=80) :: fname
  integer :: io_stat_number = 0
  type (link), pointer :: root, current
  integer :: i = 0, n
  character (len=:), allocatable :: string

  print *, ' Type in the file name ? '
  read '(a)', fname
  open (unit=1, file=fname, status='old')

  allocate (root)

! read first data item

  read (unit=1, fmt='(a)', advance='no', &
    iostat=io_stat_number) root%x
  if (io_stat_number/=-1) then
    i = i + 1
    allocate (root%next)
```

```
  end if
  current => root

! read the rest

  do while (associated(current%next))
    current => current%next
    read (unit=1, fmt='(a)', advance='no', &
      iostat=io_stat_number) current%x
    if (io_stat_number/=-1) then
      i = i + 1
      allocate (current%next)
    end if
  end do

  print *, i, ' characters read'

  n = i
  allocate (character(len=n) :: string)
  i = 0
  current => root
  do while (associated(current%next))
    i = i + 1
    string(i:i) = current%x
    current => current%next
  end do
  print *, 'data read was:'
  print *, string
end program ch2201
```

The first thing of interest is the type definition for the singly linked list. We have

```
module link_module
  type link
    character (len=1) :: c
    type (link) , pointer   :: next => null()
  end type link
end module link_module
```

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`.

We use the intrinsic `null()` to provide an initial value for the `next` pointer.

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

```
iostat=io_stat_number
advance='no'
```

and the two options combine to provide the ability to read an arbitrary number of text from a file a character at a time. If there is data in the file we allocate `root%next` and increment the character count `i`. We then loop until we reach end of file. When end of file is reached the while loop will terminate as `next` is `null()`. 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.

When end of file is reached the while loop will terminate as `next` is `null()`. We then print out the number of characters read. We then allocate a character variable of the correct size. The next statement

```
current => root
```

means that we are back at the start of the list, and in a position to traverse the list and copy each character from the linked list to the `word` character variable.

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 amount of text from a file. Seeing is believing.

22.3 Example 2: Reading in an Arbitrary Number of Reals Using a Linked List and Copying to an Array

In this example we will look at using a singly linked list to read in an arbitrary amount of data and then allocating an array to copy it to for normal numeric calculations at run time. Here is the program.

```
module link_module
  type link
    real :: x
    type (link), pointer :: next => null()
  end type link
end module link_module

program ch2202
  use link_module
  implicit none
  character (len=80) :: fname
  integer :: io_stat_number = 0
  type (link), pointer :: root, current
  integer :: i = 0, n
  real, allocatable, dimension (:) :: y

  print *, ' Type in the file name ? '
  read '(a)', fname
  open (unit=1, file=fname, status='old')

  allocate (root)

! read first data item

  read (unit=1, fmt=*, &
    iostat=io_stat_number) root%x
  if (io_stat_number/=-1) then
    i = i + 1
    allocate (root%next)
  end if
  current => root

! read the rest

  do while (associated(current%next))
    current => current%next
    read (unit=1, fmt=*, &
```

```
        iostat=io_stat_number) current%x
      if (io_stat_number/=-1) then
        i = i + 1
        allocate (current%next)
      end if
    end do

    print *, i, ' numbers read'

    n = i
    allocate (y(1:n))
    i = 0
    current => root
    do while (associated(current%next))
      i = i + 1
      y(i) = current%x
      current => current%next
    end do
    print *, 'data read was:'
    do i = 1, n
      print *, y(i)
    end do

  end program ch2202
```

A casual visual comparison of the two examples shows many similarities.

Diff is a line-oriented text file comparison utility. It tries to determine the smallest set of deletions and insertions to create one file from the other. The diff command displays the changes made in a standard format. Given one file and the changes, the other file can be created.

Here is the output from running this utility on these two examples.

```
3c3
<     character (len=1) :: x
---
>     real :: x
8c8
< program ch2201
---
> program ch2202
15c15
<   character (len=:), allocatable :: string
---
>   real, allocatable, dimension (:) :: y
25c25
```

```
<   read (unit=1, fmt='(a)', advance='no', &
---
>   read (unit=1, fmt=*, &
37c37
<     read (unit=1, fmt='(a)', advance='no', &
---
>     read (unit=1, fmt=*, &
45c45
<   print *, i, ' characters read'
---
>   print *, i, ' numbers read'
48c48
<   allocate (character(len=n) :: string)
---
>   allocate (y(1:n))
53c53
<     string(i:i) = current%x
---
>     y(i) = current%x
57,58c57,61
<   print *, string
< end program ch2201
---
>   do i = 1, n
>     print *, y(i)
>   end do
>
> end program ch2202
```

22.4 Example 3: Ragged Arrays

Arrays in Fortran are rectangular, even when allocatable. However if you wish to set up a lower triangular matrix that uses minimal memory Fortran provides a number of ways of doing this. The following example achieves it using allocatable components.

```
module ragged_module
  implicit none
  type ragged
    real, dimension (:), allocatable :: &
      ragged_row
  end type ragged
end module ragged_module
```

```
program ch2203
  use ragged_module
  implicit none
  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 ch2203
```

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.

22.5 Example 4: Ragged Arrays and Variable Sized Data Sets

The previous example showed how to use allocatable components in a derived type to achieve ragged arrays.

In this example we are going to use data from the UK Met Office. Here is the current web address.

```
https://www.metoffice.gov.uk/public/weather/
climate-historic/#?tab=climateHistoric
```

In this example both the number of stations and the number of data items for each station is read in at run time and allocated accordingly. Notice that 0 is valid as the number of data items for a station.

```
module ragged_module
  type ragged
    real, allocatable, dimension (:) :: rainfall
  end type ragged
end module ragged_module
```

```
program ch2204
  use ragged_module
  implicit none
  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)))
    if (nc(i)==0) then
      cycle
    end if
    print *, ' Type in the values for station ', &
      i
    read *, station(i)%rainfall(1:nc(i))
  end do
  print *, '  Row    N    Data'
  do i = 1, nr
    print 100, i, nc(i), station(i)%rainfall(1: &
      nc(i))
100 format (3x, i3, 2x, i3, 2x, 12(1x,f6.2))
  end do
end program ch2204
```

Here is the input data file. It is the first 6 years rainfall data from the Met Office Cwmystwyth site.

```
6
0
0
9
 144.8
 112.5
  77.2
 130.7
  66.3
```

```
   66.1
  141.1
  149.5
  134.8
8
  117.8
   72.8
   56.7
  236.2
  218.0
   69.7
   85.2
  204.4
10
  106.2
  159.7
  126.9
  121.6
   62.9
  154.3
  165.0
  139.0
  234.4
   19.7
12
   83.1
   38.5
   67.3
   76.4
   90.4
   83.5
  177.0
  180.5
   66.0
  171.9
  174.5
  334.8
```

Here is the output.

```
enter number of stations
enter the number of data values for station  1
enter the number of data values for station  2
enter the number of data values for station  3
Type in the values for station  3
```

```
enter the number of data values for station  4
Type in the values for station  4
enter the number of data values for station  5
Type in the values for station  5
enter the number of data values for station  6
Type in the values for station  6
 Row    N    Data
   1    0
   2    0
   3    9   144.80 112.50  77.20 130.70  66.30
         66.10 141.10 149.50 134.80
   4    8   117.80  72.80  56.70 236.20 218.00
         69.70  85.20 204.40
   5   10   106.20 159.70 126.90 121.60  62.90
        154.30 165.00 139.00 234.40  19.70
   6   12    83.10  38.50  67.30  76.40  90.40
        83.50 177.00 180.50  66.00 171.90
174.50 334.80
```

22.6 Example 5: Perfectly Balanced Tree

Let us now look at a more complex example that builds a perfectly balanced tree and prints it out. A loose definition of a perfectly balanced tree is one that has minimum depth for n nodes. More accurately a tree is perfectly balanced if for each node the number of nodes in its left and right subtrees differ by at most 1:

```
module tree_node_module
  implicit none

  type tree_node
    integer :: number
    type (tree_node), pointer :: left => null(), &
      right => null()
  end type tree_node

end module tree_node_module

module tree_module
  implicit none

contains
```

```
  recursive function tree(n) result (answer)
    use tree_node_module
    implicit none
    integer, intent (in) :: n
    type (tree_node), pointer :: answer
    type (tree_node), pointer :: new_node
    integer :: l, r, x

    if (n==0) then
      print *, ' terminate tree'
      nullify (answer)
    else
      l = n/2
      r = n - l - 1
      print *, l, r, n
      print *, ' next item'
      read *, x
      allocate (new_node)
      new_node%number = x
      print *, ' left branch'
      new_node%left => tree(l)
      print *, ' right branch'
      new_node%right => tree(r)
      answer => new_node
    end if
    print *, ' function tree ends'
  end function tree

end module tree_module

module print_tree_module
  implicit none

contains

  recursive subroutine print_tree(t, h)
    use tree_node_module
    implicit none
    type (tree_node), pointer :: t
    integer :: i
    integer :: h

    if (associated(t)) then
      call print_tree(t%left, h+1)
      do i = 1, h
```

```
        write (unit=*, fmt=100, advance='no')
      end do
      print *, t%number
      call print_tree(t%right, h+1)
    end if
100 format (' ')

  end subroutine print_tree

end module print_tree_module

program ch2205
! construction of a perfectly balanced tree
  use tree_node_module
  use tree_module
  use print_tree_module
  implicit none
  type (tree_node), pointer :: root
  integer :: n_of_items

  print *, 'enter number of items'
  read *, n_of_items
  root => tree(n_of_items)
  call print_tree(root, 0)
end program ch2205
```

There are a number of very important concepts contained in this example and they include:

- The use of a module to define a type. For user defined data types we must create a module to define the data type if we want it to be available in more than one program unit .
- The use of a function that returns a pointer as a result.
- As the function returns a pointer we must determine the allocation status before the function terminates. This means that in the above case we use the `nullify(result)` statement. The other option is to target the pointer.
- The use of `associated` to determine if the node of the tree is terminated or points to another node.

Type the program in and compile, link and run it. Note that the tree only has the minimal depth necessary to store all of the items. Experiment with the number of items and watch the tree change its depth to match the number of items.

22.7 Example 6: Date Class

The following is a complete manual rewrite of Skip Noble and Alan Millers date module. Here are two urls for Alan Miller's Fortran 90 version of the code. The original Skip Noble Fortran 77 version is in Chap. 38.

```
http://jblevins.org/mirror/amiller/
http://jblevins.org/mirror/amiller/datesub.f90
```

Here are some details about the function and subroutine naming conversion.

```
Skip Noble      Alan Miller
Fortran 77      Fortran 90       Current implementation

IDAY            iday             date_to_day_in_year
IZLR            izlr             date_to_weekday_number
CALEND          calend           year_and_day_to_date
CDATE           cdate            julian_to_date
NDAYS           ndays            ndays
DAYSUB          daysub           julian_to_date_and_week_and_day
JD              jd               calendar_to_julian
```

The original worked with the built-in Fortran intrinsic data types, i.e. `year`, `month` and `day` were plain integer data types. It has been rewritten to work with a derived `date` data type.

We have also added a function to print dates out in a variety of formats. This is based on a subroutine called `date_stamp` from the original code. The first key code segment is

```
type, public :: date
  private
  integer :: day
  integer :: month
  integer :: year
end type date
```

where the `date` data type is public but its components are private. This means that access to the components must be done via subroutines and functions within the `date_module` module. The next key segment is

```
character (9) :: day(0:6) = &
  (/ 'Sunday   ', 'Monday   ', 'Tuesday  ', &
     'Wednesday', 'Thursday ', 'Friday   ', &
           'Saturday ' /)
character (9) :: month(1:12) = &
```

```
(/ 'January  ', 'February ', 'March    ', &
   'April    ', 'May      ', 'June     ', &
   'July     ', 'August   ', 'September', &
   'October  ', 'November ', 'December ' /)
```

which declares the variable day to be an array of characters of length 9. They are initialised with the names of the days. The variable day is declared in the module and is available to all contained functions and subroutines.

The variable month is an array of characters of length 9 and is initialised to the names of the months. The variable month is declared in the module and is available to all contained functions and subroutines. The next key code segment is

```
public :: &
  calendar_to_julian, &
  date_, &
  date_to_day_in_year, &
  date_to_weekday_number, &
  get_day, &
  get_month, &
  get_year, &
  julian_to_date, &
  julian_to_date_and_week_and_day, &
  ndays, &
  print_date, &
  year_and_day_to_date
```

where we explicitly make the listed subroutines and functions public, as the code segment from the top of the module,

We have to provide a user defined constructor when the components of the derived type are private. This is given below:

```
function date_(dd,mm,yyyy) result (x)
  implicit none
  type (date) :: x
  integer, intent (in) :: dd, mm, yyyy
  x = date(dd,mm,yyyy)
end function date_
```

This in turn calls the built-in constructor date. As the date_ function is now an executable statement we cannot initialise in a declaration, i.e. the following is not allowed.

```
type (date) :: date1_(11,2,1952)
```

We also provide three additional procedures to access the components of the date class:

```
get_day
get_month
get_year
```

This is common programming practice in object oriented and object based programming.

The `print_date` function also has examples of internal write statements. These are

```
  write(print_date(1:2),'(i2)')x%day
  write(print_date(4:5),'(i2)')x%month
  write(print_date(7:10) , '(i4)') x%year
write(print_date(pos:pos+1) ,'(i2)') x%day
  write(print_date(pos:pos+3) , '(i4)') x%year
```

where we construct the elements of the character variable from the integer values of the `x%day`, `x%month` and `x%year` data.

```
module date_module
  implicit none

  private

  type, public :: date
    private
    integer :: day
    integer :: month
    integer :: year
  end type date

  character (9) :: day(0:6) = (/ 'Sunday   ', &
    'Monday   ', 'Tuesday  ', 'Wednesday', &
    'Thursday ', 'Friday   ', 'Saturday ' /)
  character (9) :: month(1:12) = (/ 'January  ', &
    'February ', 'March    ', 'April    ', &
    'May      ', 'June     ', 'July     ', &
    'August   ', 'September', 'October  ', &
    'November ', 'December ' /)

  public :: calendar_to_julian, date_, &
    date_to_day_in_year, date_to_weekday_number, &
```

```
      get_day, get_month, get_year, &
      julian_to_date, &
      julian_to_date_and_week_and_day, ndays, &
      print_date, year_and_day_to_date

  contains

    function calendar_to_julian(x) result (ival)
      implicit none
      integer :: ival
      type (date), intent (in) :: x

      ival = x%day - 32075 + 1461*(x%year+4800+(x% &
        month-14)/12)/4 + 367*(x%month-2-((x%month &
        -14)/12)*12)/12 - 3*((x%year+4900+(x%month &
        -14)/12)/100)/4
    end function calendar_to_julian

    function date_(dd, mm, yyyy) result (x)
      implicit none
      type (date) :: x
      integer, intent (in) :: dd, mm, yyyy

      x = date(dd, mm, yyyy)
    end function date_

  ! functions
  ! "izlr"     date_to_day_in_year
  ! and
  ! "iday"     date_to_weekday_number
  ! are taken from remark on
  ! algorithm 398, by j. douglas robertson,
  ! cacm 15(10):918.

    function date_to_day_in_year(x)
      implicit none
      integer :: date_to_day_in_year
      type (date), intent (in) :: x
      intrinsic modulo

      date_to_day_in_year = 3055*(x%month+2)/100 - &
        (x%month+10)/13*2 - 91 + (1-(modulo(x%year &
        ,4)+3)/4+(modulo(x%year,100)+99)/100-( &
        modulo(x%year,400)+399)/400)*(x%month+10)/ &
        13 + x%day
```

```
  end function date_to_day_in_year

  function date_to_weekday_number(x)
    implicit none
    integer :: date_to_weekday_number
    type (date), intent (in) :: x
    intrinsic modulo

    date_to_weekday_number = modulo((13*( &
      x%month+10-(x%month+10)/13*12)-1)/5+x%day+ &
      77+5*(x%year+(x%month-14)/12-(x%year+ &
      (x%month-14)/12)/100*100)/4+(x%year+(x% &
      month-14)/12)/400-(x%year+(x%month- &
      14)/12)/100*2, 7)
  end function date_to_weekday_number


  function get_day(x)
    implicit none
    integer :: get_day
    type (date), intent (in) :: x

    get_day = x%day
  end function get_day

  function get_month(x)
    implicit none
    integer :: get_month
    type (date), intent (in) :: x

    get_month = x%month
  end function get_month

  function get_year(x)
    implicit none
    integer :: get_year
    type (date), intent (in) :: x

    get_year = x%year
  end function get_year

! cdate - julian_to_date
! see cacm 1968 11(10):657,
! letter to the editor by fliegel and van
! flandern.
```

```
function julian_to_date(julian) result (x)
  implicit none
  integer, intent (in) :: julian
  integer :: l, n
  type (date) :: x

  l = julian + 68569
  n = 4*l/146097
  l = l - (146097*n+3)/4
  x%year = 4000*(l+1)/1461001
  l = l - 1461*x%year/4 + 31
  x%month = 80*l/2447
  x%day = l - 2447*x%month/80
  l = x%month/11
  x%month = x%month + 2 - 12*l
  x%year = 100*(n-49) + x%year + 1
end function julian_to_date

subroutine julian_to_date_and_week_and_day(jd, &
  x, wd, ddd)
  implicit none
  integer, intent (out) :: ddd, wd
  integer, intent (in) :: jd
  type (date), intent (out) :: x

  x = julian_to_date(jd)
  wd = date_to_weekday_number(x)
  ddd = date_to_day_in_year(x)
end subroutine julian_to_date_and_week_and_day

function ndays(date1, date2)
  implicit none
  integer :: ndays
  type (date), intent (in) :: date1, date2

  ndays = calendar_to_julian(date1) - &
    calendar_to_julian(date2)
end function ndays

function print_date(x, day_names, &
  short_month_name, digits)
  implicit none
  type (date), intent (in) :: x
  logical, optional, intent (in) :: day_names, &
```

```
    short_month_name, digits
  character (40) :: print_date
  integer :: pos
  logical :: want_day, want_short_month_name, &
    want_digits
  intrinsic len_trim, present, trim


  want_day = .false.
  want_short_month_name = .false.
  want_digits = .false.
  print_date = ' '
  if (present(day_names)) then
    want_day = day_names
  end if
  if (present(short_month_name)) then
    want_short_month_name = short_month_name
  end if
  if (present(digits)) then
    want_digits = digits
  end if
  if (want_digits) then
    write (print_date(1:2), '(i2)') x%day
    print_date(3:3) = '/'
    write (print_date(4:5), '(i2)') x%month
    print_date(6:6) = '/'
    write (print_date(7:10), '(i4)') x%year
  else
    if (want_day) then
      pos = date_to_weekday_number(x)
      print_date = trim(day(pos)) // ' '
      pos = len_trim(print_date) + 2
    else
      pos = 1
      print_date = ' '
    end if
    write (print_date(pos:pos+1), '(i2)') &
      x%day
    if (want_short_month_name) then
      print_date(pos+3:pos+5) = month(x%month) &
        (1:3)
      pos = pos + 7
    else
      print_date(pos+3:) = month(x%month)
      pos = len_trim(print_date) + 2
```

```
      end if
      write (print_date(pos:pos+3), '(i4)') &
        x%year
    end if

    return
  end function print_date

! calend - year_and_day_to_date
! see acm algorithm 398,
! tableless date conversion, by
! dick stone, cacm 13(10):621.

  function year_and_day_to_date(year, day) &
    result (x)
    implicit none
    type (date) :: x
    integer, intent (in) :: day, year
    integer :: t
    intrinsic modulo

    x%year = year
    t = 0
    if (modulo(year,4)==0) then
      t = 1
    end if
    if (modulo(year,400)/=0 .and. &
      modulo(year,100)==0) then
      t = 0
    end if
    x%day = day
    if (day>59+t) then
      x%day = x%day + 2 - t
    end if
    x%month = ((x%day+91)*100)/3055
    x%day = (x%day+91) - (x%month*3055)/100
    x%month = x%month - 2
    if (x%month>=1 .and. x%month<=12) then
      return
    end if
    write (unit=*, fmt='(a,i11,a)') '$$year_and_d&
      &ay_to_date: day of the year input &
      &=', day, ' is out of range.'
  end function year_and_day_to_date
```

```
end module date_module


program ch2206
  use date_module, only: calendar_to_julian, &
    date, date_, date_to_day_in_year, &
    date_to_weekday_number, get_day, get_month, &
    get_year, julian_to_date_and_week_and_day, &
    ndays, print_date, year_and_day_to_date

  implicit none
  integer :: dd, ddd, i, mm, ndiff, wd, yyyy
  integer :: val(8)
  intrinsic date_and_time
  type (date) :: date1, date2, x

  call date_and_time(values=val)
  yyyy = val(1)
  mm = 10
  do i = 31, 26, -1
    x = date_(i, mm, yyyy)
    if (date_to_weekday_number(x)==0) then
      print *, 'Turn clocks  back to EST on: ', &
        i, ' October ', get_year(x)
      exit
    end if
  end do
  call date_and_time(values=val)
  yyyy = val(1)
  mm = 4
  do i = 1, 8
    x = date_(i, mm, yyyy)
    if (date_to_weekday_number(x)==0) then
      print *, 'Turn clocks ahead to DST on: ', &
        i, ' April   ', get_year(x)
      exit
    end if
  end do
  call date_and_time(values=val)
  yyyy = val(1)
  mm = 12
  dd = 31
  x = date_(dd, mm, yyyy)
  if (date_to_day_in_year(x)==366) then
    print *, get_year(x), ' is a leap year'
```

```
else
  print *, get_year(x), ' is not a leap year'
end if
x = date_(1, 1, 1970)
call julian_to_date_and_week_and_day &
  (calendar_to_julian(x), x, wd, ddd)
if (get_year(x)/=1970 .or. get_month(x)/=1 &
  .or. get_day(x)/=1 .or. wd/=4 .or. ddd/=1) &
  then
  print *, &
    'julian_to_date_and_week_and_day failed'
  print *, ' date, wd, ddd = ', get_year(x), &
    get_month(x), get_day(x), wd, ddd
  stop
end if
date1 = date_(22, 5, 1984)
date2 = date_(22, 5, 1983)
ndiff = ndays(date1, date2)
yyyy = 1970

x = year_and_day_to_date(yyyy, ddd)

if (ndiff/=366) then
  print *, 'ndays failed; ndiff = ', ndiff
else
  if (get_month(x)/=1 .and. get_day(x)/=1) &
    then
    print *, 'year_and_day_to_date failed'
    print *, ' mma, dda = ', get_month(x), &
      get_day(x)
  else
    print *, ' calendar_to_julian OK'
    print *, ' date_ OK'
    print *, ' date_to_day_in_year OK'
    print *, ' date_to_weekday_number OK'
    print *, ' get_day OK'
    print *, ' get_month OK'
    print *, ' get_year OK'
    print *, &
      ' julian_to_date_and_week_and_day OK'
    print *, ' ndays OK'
    print *, ' year_and_day_to_date OK'
  end if
end if
```

```
    x = date_(11, 2, 1952)

    print *, ' print_date test'
    print *, ' Single parameter       ', &
      print_date(x)
    print *, &
      ' day_names=false short_month_name=false ', &
      print_date(x, day_names=.false., &
      short_month_name=.false.)
    print *, &
      ' day_names=true  short_month_name=false ', &
      print_date(x, day_names=.true., &
      short_month_name=.false.)
    print *, &
      ' day_names=false short_month_name=true  ', &
      print_date(x, day_names=.false., &
      short_month_name=.true.)
    print *, &
      ' day_names=true  short_month_name=true  ', &
      print_date(x, day_names=.true., &
      short_month_name=.true.)
    print *, ' digits=true            ', &
      print_date(x, digits=.true.)

    print *, ' Test out a month'

    yyyy = 1970
    do dd = 1, 31
      x = year_and_day_to_date(yyyy, dd)
      print *, print_date(x, day_names=.false., &
        short_month_name=.true.)
    end do

  end program ch2206
```

There are wrap problems with some of the lengthier arithmetic expressions. The version on the web site is obviously correct.

We also have an alternate form of array declaration in this program, which is given below. It is common in Fortran 77 style code:

```
    integer :: val(8)
```

One improvement would be additional code to test the validity of dates. This would be called from within our constructor `date_`. This would mean that we could never have an invalid date when using the `date_module`. This is left as a programming exercise.

22.7.1 Notes: DST in the USA

The above program is no longer correct. Beginning in 2007, Daylight Saving Time was brought forward by 3 or 4 weeks in Spring and extended by one week in the Fall. Daylight Saving Time begins for most of the United States at 2 a.m. on the second Sunday of March. Time reverts to standard time at 2 a.m. on the first Sunday in November.

22.8 Example 7: Date Data Type with USA and ISO Support

The date derived type in this chapter handles conventional UK or world data types. To handle USA and ISO date formats we have added an extra component to this derived type. Here is the updated type.

```
type, public :: date
private
integer :: day
integer :: month
integer :: year
integer :: date_type = 1
end type date
```

When we use the default constructor we set the `date_type` to 1. An integer variable is often used in a problem like this. In the `date_iso` constructor we set `date_type` to 3 and in the `date_us` constructor set set `date_type` to 2.

The only other method we have to alter is the `print_date` method. In this method we have an `if then else` construct to choose how to print the date, based on the date type.

We have solved the problem of how to handle a variety of date formats in a simple, non object oriented fashion. First we have the date module.

```
module date_module
  implicit none

  private
```

```
  type, public :: date
    private
    integer :: day
    integer :: month
    integer :: year
    integer :: date_type = 1
  end type date

  character (9) :: day(0:6) = (/ 'Sunday   ', &
    'Monday   ', 'Tuesday  ', 'Wednesday', &
    'Thursday ', 'Friday   ', 'Saturday ' /)
  character (9) :: month(1:12) = (/ 'January  ', &
    'February ', 'March    ', 'April    ', &
    'May      ', 'June     ', 'July     ', &
    'August   ', 'September', 'October  ', &
    'November ', 'December ' /)

  public :: calendar_to_julian, date_, date_iso, &
    date_us, date_to_day_in_year, &
    date_to_weekday_number, get_day, get_month, &
    get_year, julian_to_date, &
    julian_to_date_and_week_and_day, ndays, &
    print_date, year_and_day_to_date

contains

  function date_(dd, mm, yyyy) result (x)
    implicit none
    type (date) :: x
    integer, intent (in) :: dd, mm, yyyy
    integer :: dt = 1

    x = date(dd, mm, yyyy, dt)
  end function date_

  function date_iso(yyyy, mm, dd) result (x)
    implicit none
    type (date) :: x
    integer, intent (in) :: dd, mm, yyyy
    integer :: dt = 3

    x = date(dd, mm, yyyy, dt)
  end function date_iso
```

```
  function date_us(mm, dd, yyyy) result (x)
    implicit none
    type (date) :: x
    integer, intent (in) :: dd, mm, yyyy
    integer :: dt = 2

    x = date(dd, mm, yyyy, dt)
  end function date_us

  include 'date_module_include_code.f90'

  function print_date(x, day_names, &
    short_month_name, digits)
    implicit none
    type (date), intent (in) :: x
    logical, optional, intent (in) :: day_names, &
      short_month_name, digits
    character (30) :: print_date
    integer :: pos
    logical :: want_day, want_short_month_name, &
      want_digits
    integer :: l, t
    intrinsic len_trim, present, trim

    want_day = .false.
    want_short_month_name = .false.
    want_digits = .false.
    print_date = ' '
    if (present(day_names)) then
      want_day = day_names
    end if
    if (present(short_month_name)) then
      want_short_month_name = short_month_name
    end if
    if (present(digits)) then
      want_digits = digits
    end if

!   Start of code dependent on date_type
!   day month year
    if (x%date_type==1) then
      if (want_digits) then
        write (print_date(1:2), '(i2)') x%day
        print_date(3:3) = '/'
        write (print_date(4:5), '(i2)') x%month
        print_date(6:6) = '/'
```

```
        write (print_date(7:10), '(i4)') x%year
      else
        if (want_day) then
          pos = date_to_weekday_number(x)
          print_date = trim(day(pos)) // ' '
          pos = len_trim(print_date) + 2
        else
          pos = 1
          print_date = ' '
        end if
        write (print_date(pos:pos+1), '(i2)') &
          x%day
        if (want_short_month_name) then
          print_date(pos+3:pos+5) &
            = month(x%month)(1:3)
          pos = pos + 7
        else
          print_date(pos+3:) = month(x%month)
          pos = len_trim(print_date) + 2
        end if
        write (print_date(pos:pos+3), '(i4)') &
          x%year
      end if

    else if (x%date_type==2) then
!     month day year
      if (want_digits) then
        write (print_date(1:2), '(i2)') x%month
        print_date(3:3) = '/'
        write (print_date(4:5), '(i2)') x%day
        print_date(6:6) = '/'
        write (print_date(7:10), '(i4)') x%year
      else
        pos = 1
        if (want_short_month_name) then
          print_date(pos:pos+2) = month(x%month) &
            (1:3)
          pos = pos + 4
        else
          print_date(pos:) = month(x%month)
          pos = len_trim(print_date) + 2
        end if
        if (want_day) then
          t = date_to_weekday_number(x)
          l = len_trim(day(t))
          print_date(pos:pos+l) = trim(day(t)) &
```

```
              // ' '
            pos = len_trim(print_date) + 2
          end if
          write (print_date(pos:pos+1), '(i2)') &
            x%day
          pos = pos + 3
          write (print_date(pos:pos+3), '(i4)') &
            x%year
        end if
      else if (x%date_type==3) then
!       year month day
        if (want_digits) then
          write (print_date(1:4), '(i4)') x%year
          print_date(5:5) = '/'
          write (print_date(6:7), '(i2)') x%month
          print_date(8:8) = '/'
          write (print_date(9:10), '(i2)') x%day
        else
          pos = 1
          write (print_date(pos:pos+3), '(i4)') &
            x%year
          pos = pos + 5
          if (want_short_month_name) then
            print_date(pos:pos+2) = month(x%month) &
              (1:3)
            pos = pos + 4
          else
            print_date(pos:) = month(x%month)
            pos = len_trim(print_date) + 2
          end if
          if (want_day) then
            t = date_to_weekday_number(x)
            l = len_trim(day(t))
            print_date(pos:pos+l) = trim(day(t))
            pos = pos + l + 1
          end if
          write (print_date(pos:pos+1), '(i2)') &
            x%day
        end if
      end if
      return
    end function print_date

  end module date_module
```

Note that we have put the common executable code from the earlier date module into an include file.

```
include 'date_module_include_code.f90'
```

Next we have the program that uses the module.

```
include 'ch2207_date_module.f90'

program ch2207
  use date_module, only: calendar_to_julian, &
    date, date_, date_iso, date_us, &
    date_to_day_in_year, date_to_weekday_number, &
    get_day, get_month, get_year, &
    julian_to_date_and_week_and_day, ndays, &
    print_date, year_and_day_to_date

  implicit none
  integer :: i
  integer, parameter :: n = 3
  type (date), dimension (1:n) :: x

  x(1) = date_(11, 2, 1952)
  x(2) = date_us(2, 11, 1952)
  x(3) = date_iso(1952, 2, 11)

  do i = 1, 3
    print *, print_date(x(i))
  end do

end program ch2207
```

Note that we used the alternate syntax of using the

```
include 'ch2207_date_module.f90'
```

statement in this example.

22.9 Bibliography

Chapter 2 provided details of some books that address data structuring, but mainly from an historical viewpoint.

We provide a small number of references to books that look at data structuring more generally.

Schneider G.M., Bruell S.C., Advanced Programming and Problem Solving with Pascal, Wiley, 1981.

- The book is aimed at computer science students and follows the curriculum guidelines laid down in Communications of the ACM, August 1985, Course CS2. The book is very good for the complete beginner as the examples are very clearly laid out and well explained. There is a coverage of data structures, abstract data types and their implementation, algorithms for sorting and searching, the principles of software development as they relate to the specification, design, implementation and verification of programs in an orderly and disciplined fashion — their words.

Sedgewick, Robert (1993). Algorithms in Modula 3, Addison-Wesley. ISBN 0-201-53351-0.

- The Modula 3 algorithms are relatively easy to translate into Fortran.

22.10 Problems

22.1 Compile and run the examples in this chapter with your compiler.

22.2 Using ch2202.f90 as a starting point rewrite it to work with a file of integer data. You may find the diff output useful here.

22.3 Modify the ragged array example that processes a lower triangular matrix to work with an upper triangular matrix.

22.4 Using the balanced tree example as a basis and modify it to work with a character array rather than an integer. The routine that prints the tree will also have to be modified to reflect this.

22.5 Modify the Date program to account for the current DST in the USA.

22.6 Modify ch2204 to calculate and print the average rainfall for each station.