

Chapter 7

OO Data Structures

7.1 Data Structures

We have seen that F90 has a very strong intrinsic base for supporting the use of subscripted arrays. Fortran arrays can contain intrinsic data types as well as user defined types (i.e., ADT's). One can not directly have an array of pointers but you are allowed to have an array contain defined types that are pointers or that have components that are pointers. Arrays offer an efficient way to contain information and to insert and extract information. However, there are many times when creating an efficient algorithm dictates that we use some specialized storage method, or *container*, and a set of operations to act with that storage mode. The storage representation and the set of operations that are allowed for it are known as a *data structure*. How you store and retrieve an item from a container is often independent of the nature of the item itself. Thus, different instances of a data structure may produce containers for different types of objects. Data structures have the potential for a large amount of code reuse, which is a basic goal of OOP methods. In the following sections we will consider some of the more commonly used containers.

7.2 Stacks

A stack is a data structure where access is restricted to the last inserted object. It is referred to as a *last-in first-out* (LIFO) container. In other words, a stack is a container to which elements may only be inserted or removed at one end of the container, called the *top* of the stack. It behaves much like a pile of dinner plates. You can place a new element on the pile (widely known as a *push*), remove the top element from the pile (widely known as a *pop*), and identify the element on the top of the pile. You can also have the general concept of an empty pile, and possibly a full pile if it is associated with some type of restrictive container. Since at this point we only know about using arrays as containers we will construct a stack container by using an array.

Assume that we have defined the attributes of the "Object" that is to use our container by building a module called `object_type`. Then we could declare the array implementation of a stack type to be:

```
module stack_type
  use object_type ! to define objects in the stack
  implicit none

  integer, parameter :: limit = 999 ! stack size limit

  type stack
  private
    integer      :: size      ! size of array
    integer      :: top       ! top of stack
    type (Object) :: a(limit) ! stack items array
  end type stack
end module stack_type
```

The interface contract to develop one such stack support system (or ADT) is given as:

```

module stack_of_objects
implicit none
  public :: stack, push_on_Stack, pop_from_Stack, &
         is_Stack_Empty, is_Stack_Full

interface ! for a class_Stack contract

  function make_Stack (n) result (s) ! constructor
    use stack_type ! to define stack structure
    integer, optional :: n ! size of stack
    type (stack) :: s ! the new stack
  end function make_Stack

  subroutine push_on_Stack (s, item) ! push item on top of stack
    use stack_type ! for stack structure
    type (stack), intent(inout) :: s
    type (Object), intent(in) :: item
  end subroutine push_on_Stack

  function pop_from_Stack (s) result (item) ! pop item from top
    use stack_type ! for stack structure
    type (stack), intent(inout) :: s
    type (Object) :: item
  end function pop_from_Stack

  function is_Stack_Empty (s) result (b) ! test stack
    use stack_type ! for stack structure
    type (stack), intent(in) :: s
    logical :: b
  end function is_Stack_Empty

  function is_Stack_Full (s) result (b) ! test stack
    use stack_type ! for stack structure
    type (stack), intent(in) :: s
    logical :: b
  end function is_Stack_Full

end interface
end module stack_of_objects

```

In the interface we see that some of the member services (`is_Stack_Empty` and `is_Stack_Full`) are independent of the contained objects. Others (`pop_from_Stack` and `push_on_Stack`) explicitly depend on the Object utilizing the container. Of course, the constructor (here `make_Stack`) always indirectly relates to the Object being contained in the array. The full details of a `Stack` class are given in Fig. 7.1.

For a specific implementation test we will simply utilize objects that have a single integer attribute. That is, we define the object of interest by a code segment like:

```

module object_type
  type Object
    integer :: data ; end type ! one integer attribute
end module object_type

```

Obviously, there are many other types of objects that one may want to create and place in a container like a stack. At the present one would have to edit the above segment to define all the attributes of the object. (Begin to think about how you might seek to automate such a process.) The new `Stack` class is tested in Fig. 7.2, while a history of the example stack is sketched in Fig. 7.3. The only part of that code that depends on a specific object is in line 7 where the (public) intrinsic constructor, `Object`, was utilized rather than some more general constructor, say `Object_`.

In Fig. 7.1 note that we have used an alternate syntax and specified the type of function result (logical, Object, or stack) as a prefix to the function name (lines 16, 28, 36, 40). The author thinks that the form used in the interface contract is easier to read and understand since it requires an extra line of code, however some programmers prefer the condensed style of Fig. 7.1. Later we will examine an alternate implementation of a stack by using a linked list.

The stack implementation shown here is not complete. For example, some programmers like to include a member, say `show_Stack_top`, to display the top element on the container without removing it from the stack. Also we need to be concerned about *pre-conditions* that need to be satisfied for a member and may require that we throw an exception message. You can not pop an item off of an empty stack, nor can you push an item onto the top of a full stack. Only the member `pop_from_Stack` does such pre-condition checking in the sample code. Note that members `is_Stack_Empty` and `is_Stack_Full`

are called *accessors*, as would be `show_Stack_top`, since they query the container but do not change it.

```
[ 1] module class_Stack
[ 2]   implicit none
[ 3]   use exceptions ! to warn of errors
[ 4]   use object_type
[ 5]   public :: stack, push_on_Stack, pop_from_Stack, &
[ 6]           is_Stack_Empty, is_Stack_Full
[ 7]   integer, parameter :: limit = 999 ! stack size limit
[ 8]
[ 9]   type stack
[10]     private
[11]       integer      :: size      ! size of array
[12]       integer      :: top       ! top of stack
[13]       type (Object) :: a(limit) ! stack items array
[14]   end type
[15] contains ! encapsulated functionality
[16]
[17]   type (stack) function make_Stack (n) result (s)      ! constructor
[18]     integer, optional :: n ! size of stack
[19]     s%size = limit ; if ( present (n) ) s%size = n
[20]     s%top = 0      ! object array not initialized
[21]   end function make_Stack
[22]
[23]   subroutine push_on_Stack (s, item) ! push item on top of stack
[24]     type (stack), intent(inout) :: s
[25]     type (Object), intent(in)   :: item
[26]     s%top = s%top + 1 ; s%a(s%top) = item
[27]   end subroutine push_on_Stack
[28]
[29]   type (Object) function pop_from_Stack (s) result (item) ! off top
[30]     type (stack), intent(inout) :: s
[31]     if ( s%top < 1 ) then
[32]       call exception ("pop_from_Stack", "stack is empty")
[33]     else
[34]       item = s%a(s%top) ; s%top = s%top - 1
[35]     end if ; end function pop_from_Stack
[36]
[37]   logical function is_Stack_Empty (s) result (b)
[38]     type (stack), intent(in) :: s
[39]     b = ( s%top == 0 ) ; end function is_Stack_Empty
[40]
[41]   logical function is_Stack_Full (s) result (b)
[42]     type (stack), intent(in) :: s
[43]     b = ( s%top == s%size ) ; end function is_Stack_Full
[44]
[45] end module class_Stack
```

Figure 7.1: A Typical Stack Class

```

[ 1] include 'class_stack.f'      ! previous figure
[ 2] program main
[ 3] use class_stack
[ 4] implicit none
[ 5] type (stack) :: b
[ 6] type (object) :: value, four, five, six
[ 7]
[ 8]     four = Object(4) ; five = Object(5) ; six = Object(6) ! initialize
[ 9]
[10]     b = make_stack(3)                ! private constructor
[11]     print *, is_stack_empty(b), is_stack_full(b) ! b = [], empty
[12]
[13]     call push_on_stack (b, four)      ! b = [4]
[14]     call push_on_stack (b, five)    ! b = [5,4]
[15]     call push_on_stack (b, six)     ! b = [6,5,4], full
[16]     print *, is_stack_empty(b), is_stack_full(b) ! F T
[17]
[18]     value = pop_from_stack (b) ; print *, value ! b = [5,4]
[19]     print *, is_stack_empty(b), is_stack_full(b) ! F F
[20]
[21]     value = pop_from_stack (b) ; print *, value ! b = [4]
[22]     print *, is_stack_empty(b), is_stack_full(b) ! F F
[23]
[24]     value = pop_from_stack (b) ; print *, value ! b = [], empty
[25]     print *, is_stack_empty(b), is_stack_full(b) ! T F
[26]
[27]     value = pop_from_stack (b)      ! nothing to pop
[28] end program main ! running gives:
[29] ! T F ! F T
[30] ! 6   ! F F
[31] ! 5   ! F F
[32] ! 4   ! T F
[33] ! Exception occurred in subprogram pop_from_stack
[34] ! With message: stack is empty

```

Figure 7.2: Testing a Stack of Objects

Full ?	F	F	F	T	F	F	F	F
Empty ?	T	F	F	F	F	F	T	T
Error ?	N	N	N	N	N	N	N	Y

Stack:		4	5	6	5	4		
			4	5	4			
	---	---	---	---	---	---	---	---
(Line)	9	12	13	14	17	20	23	26

Figure 7.3: Steps in the Stack Testing

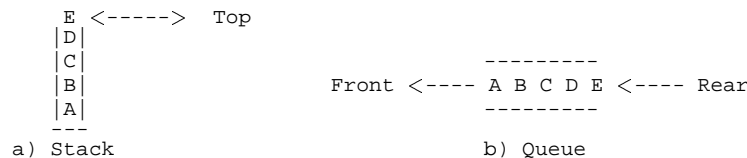


Figure 7.4: Simple Containers

7.3 Queues

A comparison of a stack and another simple container, a *queue*, is given in Fig. 7.4. Its name queue comes from the British word which means waiting in a line for service. A queue is a container into which elements may be inserted at one end, called the *rear*, and leave only from the other end, called the *front*. The first element in the queue expects to be the first serviced and, thus, be the first out of line. A queue is a *first-in first-out* (FIFO) container system. In planning our first queue container we will again make use of an array of objects. Doing so one quickly finds that you are much less likely to encounter a full queue if it is stored as a so-called fixed circular array with a total of `Q_Size_Limit` storage slots. At this point we define the structure of our queue to be:

```
module Queue_type
! A queue stored as a so-called fixed circular array with a total
! of Q_Size_Limit storage slots; requires remainder function, mod.
! (version 1, i.e., without allocatable arrays and pointers)
use object_type ! to define objects in the Container
implicit none

integer, parameter :: Q_Size_Limit = 999

type Queue
private
integer :: head ! index of first element
integer :: tail ! index of last element
integer :: length ! size of used storage
type (Object) :: store (Q_Size_Limit) ! a circular array
end type Queue
end module Queue_type
```

An interface contract that will allow us to build a typical queue is:

```
module Queue_of_Objects
implicit none
public :: Queue, Add_to_Q, Create_Q, Get_Front_of_Q, Is_Q_Empty, &
Is_Q_Full, Get_Length_of_Q, Remove_from_Q

interface ! for a class_Queue contract

subroutine Add_to_Q (Q, item) ! add to tail of queue
use Queue_type ! for Queue structure
type (Queue), intent(inout) :: Q
type (Object), intent(in) :: item ; end Subroutine Add_to_Q

function Create_Q (N) result (Q) ! manual constructor
use Queue_type ! for Queue structure
integer, intent(in) :: N ! size of the new array
type (Queue) :: Q ; end function Create_Q

function Get_Capacity_of_Q (Q) result (item)
use Queue_type ! for Queue structure
type (Queue), intent(in) :: Q
type (Object) :: item ; end function Get_Capacity_of_Q

function Get_Front_of_Q (Q) result (item)
use Queue_type ! for Queue structure
type (Queue), intent(in) :: Q
type (Object) :: item ; end function Get_Front_of_Q

function Is_Q_Empty (Q) result(B)
use Queue_type ! for Queue structure
type (Queue), intent(in) :: Q
logical :: B ; end function Is_Q_Empty

function Is_Q_Full (Q) result(B)
use Queue_type ! for Queue structure
type (Queue), intent(in) :: Q
logical :: B ; end function Is_Q_Full

function Get_Length_of_Q (Q) result (N)
use Queue_type ! for Queue structure
type (Queue), intent(in) :: Q
integer :: N ; end function Get_Length_of_Q

subroutine Remove_from_Q (Q) ! remove from head of queue
use Queue_type ! for Queue structure
type (Queue), intent(inout) :: Q ; end subroutine Remove_from_Q

end interface
end module Queue_of_Objects
```

For a specific version we provide full details for objects containing an integer in Fig. 7.5, and test and display the validity of the implementation in Fig. 7.6, where again the objects are taken to be integers (lines 15, 19, 20).

```
[ 1] module class_Queue                                ! file: class_Queue.f90
[ 2]
[ 3] ! A queue stored as a so-called fixed circular array with a total of
[ 4] ! Q_Size_Limit storage slots; requires remainder function, mod.
[ 5] ! (i.e., without allocatable arrays and pointers)
[ 6]
[ 7] use exceptions                                  ! inherit exception handler
[ 8] implicit none
[ 9]
[10] public :: Queue, Add_to_Q, Create_Q, Get_Front_of_Q
[11]         Is_Q_Full, Get_Length_of_Q, Remove_from
[12]
[13] integer, parameter :: Q_Size_Limit = 3
[14]
[15] type Queue
[16]   private
[17]   integer :: head          ! index of first element
[18]   integer :: tail         ! index of last element
[19]   integer :: length       ! size of used storage
[20]   integer :: store (Q_Size_Limit) ! a circular array of elements
[21] end type Queue
[22]
[23] contains                                          ! member functionality
[24]
[25] Subroutine Add_to_Q (Q, item)                    ! add to tail of queue
[26]   type (Queue), intent(inout) :: Q
[27]   integer,          intent(in)  :: item
[28]
[29]   if ( Is_Q_Full(Q) ) call exception ("Add_to_Q","full Q")
[30]   Q%store (Q%tail) = item
[31]   Q%tail      = 1 + mod (Q%tail, Q_Size_Limit)
[32]   Q%length    = Q%length + 1 ; end Subroutine Add_to_Q
[33]
[34] type (Queue) function Create_Q (N) result (Q)    ! manual constructor
[35]   integer, intent(in) :: N ! size of the new array
[36]   integer              :: k ! implied loop
[37]
[38]   if (N > Q_Size_Limit) call exception("Create_Q","increase size")
[39]   Q = Queue (1, 1, 0, (/ (0, k=1,N) /)) ! intrinsic constructor
[40] end function Create_Q
[41]
[42] integer function Get_Capacity_of_Q (Q) result (item)
[43]   type (Queue), intent(in) :: Q
[44]
[45]   item = Q_size_Limit - Q%length ; end function Get_Capacity_
[46]
[47] integer function Get_Front_of_Q (Q) result (item)
[48]   type (Queue), intent(in) :: Q
[49]
[50]   if (Is_Q_Empty(Q)) call exception("Get_Front_of_Q","em
[51]   item = Q%store (Q%head) ; end function Get_Front_of_Q
[52]
[53] logical function Is_Q_Empty (Q) result(B)
[54]   type (Queue), intent(in) :: Q
[55]
[56]   B = (Q%length == 0) ; end function Is_Q_Empty
[57]
[58] logical function Is_Q_Full (Q) result(B)
[59]   type (Queue), intent(in) :: Q
[60]
[61]   B = (Q%length == Q_Size_Limit) ; end function Is_Q_Full
[62]
[63] integer function Get_Length_of_Q (Q) result (N)
[64]   type (Queue), intent(in) :: Q
[65]   N = Q%length ; end function Get_Length_of_Q
[66]
[67] subroutine Remove_from_Q (Q)                    ! remove from head of queue
[68]   type (Queue), intent(inout) :: Q
[69]
[70]   if (Is_Q_Empty(Q)) call exception("Remove_from_Q","empty Q")
[71]   Q%head = 1 + mod (Q%head, Q_Size_Limit)
[72]   Q%length = Q%length - 1 ; end subroutine Remove_from_Q
[73]
[74] end module class_Queue                                ! file: class_Queue.f
```

Figure 7.5: A Typical Queue Class

```

[ 1] program main
[ 2]   use class_Queue ! inherit its methods & class global constants
[ 3]   implicit none
[ 4]
[ 5]   type (Queue) :: C, B ! not used, used
[ 6]   integer :: value, limit = 3 ! work items
[ 7]
[ 8]   C = Create_Q (limit) ! private constructor
[ 9]   print *, "Length of C = ", Get_Length_of_Q (C)
[10]   print *, "Capacity of C = ", Get_Capacity_of_Q (C)
[11]   print *, "C empty? full? ", is_Q_Empty (C), is_Q_Full (C) !
[12]
[13]   B = Create_Q (3) ! private constructor
[14]   print *, "B empty? full? ", is_Q_Empty (B), is_Q_Full (B) !
[15]
[16]   call Add_to_Q (B, 4); print *, "B = [4]"
[17]   print *, "Length of B = ", Get_Length_of_Q (B)
[18]   print *, "B empty? full? ", is_Q_Empty (B), is_Q_Full (B) !
[19]
[20]   call Add_to_Q (B, 5); print *, "B = [4,5]"
[21]   call Add_to_Q (B, 6); print *, "B = [4,5,6], full"
[22]   print *, "Length of B = ", Get_Length_of_Q (B)
[23]   print *, "B empty? full? ", is_Q_Empty (B), is_Q_Full (B) !
[24]   print *, "Capacity of B = ", Get_Capacity_of_Q (B)
[25]
[26]   value = Get_Front_of_Q (B); print *, "Front Q value = ", value
[27]
[28]   call Remove_from_Q (B); print *, "Removing from B"
[29]   print *, "Length of B = ", Get_Length_of_Q (B)
[30]   print *, "B empty? full? ", is_Q_Empty (B), is_Q_Full (B) !
[31]   value = Get_Front_of_Q (B); print *, "Front Q value = ", value
[32]
[33]   call Remove_from_Q (B); print *, "Removing from B"
[34]   print *, "Length of B = ", Get_Length_of_Q (B)
[35]   print *, "B empty? full? ", is_Q_Empty (B), is_Q_Full (B) !
[36]
[37]   call Remove_from_Q (B); print *, "Removing from B"
[38]   print *, "Length of B = ", Get_Length_of_Q (B)
[39]   print *, "B empty? full? ", is_Q_Empty (B), is_Q_Full (B) !
[40]
[41]   print *, "Removing from B"; call Remove_from_Q (B)
[42]   call exception_status
[43] end program main ! running gives:
[44] ! Length of C = 0 ! Capacity of C = 3 ! C empty? full? T, F
[45] ! B empty? full? T, F
[46] ! B = [4] ! Length of B = 1 ! B empty? full? F, F
[47] ! B = [4,5]
[48] ! B = [4,5,6], full ! Length of B = 3 ! B empty? full? F, T
[49] ! Capacity of B = 0 ! Front Q value = 4 ! Removing from B
[50] ! Length of B = 2 ! B empty? full? F, F ! Front Q value = 5
[51] ! Removing from B ! Length of B = 1 ! B empty? full? F, F
[52] ! Removing from B ! Length of B = 0 ! B empty? full? T, F
[53] ! Removing from B
[54] ! Exception Status Thrown
[55] ! Program :Remove_from_Q
[56] ! Message :empty Q
[57] ! Level : 5
[58] !
[59] ! Exception Summary:
[60] ! Exception count = 1
[61] ! Highest level = 5

```

Figure 7.6: Testing of the Queue Class

7.4 Linked Lists

From our limited discussion of stacks and queues it should be easy to see that to try to insert or remove an object at the middle of a stack or queue is not an efficient process. *Linked lists* are containers which make it easy to perform the operations of insertion and deletion. A linked list of objects can be thought of as a group of boxes, usually called *nodes*, each containing an object to be stored and a *pointer*, or reference, to the box containing the next object in the list. In most of our applications a list is referenced by a special box, called the *header* or *root* node, which does not store an object but serves mainly to point to the first linkable box, and thereby produces a condition where the list is never truly empty. This simplifies the insertion scheme by removing an algorithmic special case. We will begin our introduction of these topics with a *singly linked list*, also known as a simple list. It is capable of being traversed in only one direction, from the beginning of the list to the end, or vice versa.

As we have seen, arrays of data objects work well so long as we know, or can compute, in advance the amount of data to be stored. The data structures (linked lists and trees) to be considered here employ *pointers* to store and change data objects when we do not know the required amount of storage in advance. During program execution linked lists and trees allow separate memory allocations for each individual data object. However, they do not permit direct access to an arbitrary object in the container. Instead some searching must be performed and thus they incur an execution time penalty for such an access operation. That penalty is smaller in tree structures than in linked lists (but is smallest of all in arrays).

Linked lists and trees must use pointer (reference) variables. Fortran pointers can simply be thought of as an alias for other variables of the same type. We are beginning to see that pointers give a programmer more power. However, that includes more power to “shoot yourself in the foot”; they make it hard to find some errors; and can lead to new types of errors such as the so called *memory leaks*. Recall that each pointer must be in one of three states: undefined, null, or associated. As dummy arguments within routines pointer variables cannot be assigned the INTENT attribute. That means they have a greater potential for undesired *side effects*. To avoid accidentally changing a pointer it is good programming practice to clearly state in comments the INTENT of all dummy pointer arguments and to immediately copy those with an INTENT IN attribute. Thereafter working with the copied pointer guarantees that an error or later modification of the routine can not produce a side effect on the pointer. We also want to avoid a *dangling pointer* which is caused by a deallocation that leaves its target object forever inaccessible. A related problem is a memory leak or *unreferenced storage* such as the program segment:

```
real, pointer :: X_ptr (:)
  allocate ( X_ptr(Big_number) )
  . . . ! use X_ptr
  nullify ( X_ptr ) ! dangling pointer
```

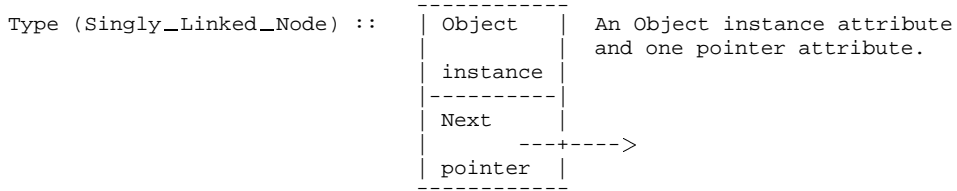
because now there is no way to release memory for X_ptr. To avoid this we need to free the memory before the pointer is nullified, so the segment becomes:

```
real, pointer :: X_ptr (:)
  allocate ( X_ptr(Big_number) )
  . . . ! use X_ptr
  deallocate ( X_ptr ) ! memory released
  nullify ( X_ptr )
```

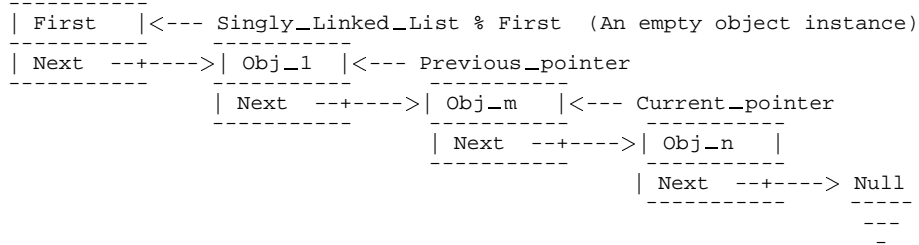
Remember that in F95 the memory is automatically deallocated at the end of the scope of the variable, unless one retains the variable with a SAVE statement (and formally deallocates it elsewhere).

7.4.1 Singly Linked Lists

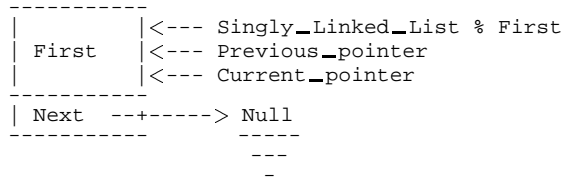
We begin the study of the singly linked list by showing the notations employed in Fig. 7.7. From experience we have chosen to have a dummy first node, called `first`, to simplify our algorithms so that a list is never truly empty. Also as we scan through a list we will use one pointer, called `current`, to point to the current object in the list and a companion, called `previous`, to point to the directly preceding object (if any). If no objects have been placed in the list then both of these simply point to the `first` node. The end of the list is denoted by the `next` pointer attribute taking on the `null` value. To insert or delete objects one must be able to rank two objects. This means that in order to have a generic linked list one must overload the relational operators, (`<` and `==`) when the object to be placed in the container is defined. Since most objects have different types of attributes the overloading process is clearly application



a) Singly linked node



b) List of singly linked nodes



c) An 'empty' (one node) singly linked list

Figure 7.7: Singly linked list terminology

dependent. The process for inserting an object is sketched in Fig. 7.8 while that for deleting an object is in Fig. 7.9.

The `Singly_Linked_List` class is given in Fig. 7.10. It starts with the definition of a singly linked node (lines 4-8) that has an object attribute and a pointer attribute to locate the next node. Then a list is begun (lines 10-13) by creating the dummy first node that is consider to represent an empty list. The object deletion member must employ an overloaded operator (line 28), as must the insertion member (line 52). Observe that a list never gets "full", unless the system runs out of memory. The empty list test member (line 62) depends on the pointer status, but is independent of the objects stored. The constructor for a list (line 68) simply creates the first node and nullifies it. The printing member (line 74) is called an `iterator` since it runs through all objects in the list. The testing program for this container type and its output results are given in Fig. 7.11. In order to test such a container it is necessary to have an object type defined. Here an object with a single integer value was selected, and thus it was easy to overload the relational operators with a clear meaning as shown in Fig. 7.12.

INSERT

Figure 7.8: Inserting an Object in a Singly Linked List

DELETE

Figure 7.9: Deleting an Object from a Singly Linked List

```

[ 1] module singly_linked_list
[ 2]   use class_Object
[ 3]   implicit none
[ 4]
[ 5]   type S_L_node                                ! Singly Linked Node
[ 6]     private
[ 7]     type (Object)                               :: value          ! Object attribute
[ 8]     type (S_L_node), pointer :: next          ! Pointer to next node
[ 9]   end type S_L_node
[10]
[11]   type S_L_list                                ! Singly Linked List of Nodes
[12]     private
[13]     type (S_L_node), pointer :: first ! Dummy first object in list
[14]   end type S_L_list
[15]
[16] contains
[17]   subroutine S_L_delete (links, Obj, found)
[18]     type (S_L_list), intent (inout) :: links
[19]     type (Object),   intent (in)    :: Obj
[20]     logical,         intent (out)   :: found
[21]     type (S_L_node), pointer       :: previous, current
[22]
[23]     ! find location of Obj
[24]     previous => links%first          ! begin at top of list
[25]     current  => previous%next       ! begin at top of list
[26]     found = .false.                ! initialize
[27]     do
[28]       if ( found .or. (.not. associated (current))) return ! list end
[29]       if ( Obj == current%value ) then ! *** OVERLOADED ***
[30]         found = .true. ; exit ! this location search
[31]       else ! move the next node in list
[32]         previous => previous%next
[33]         current  => current%next
[34]       end if
[35]     end do ! to find location of node with Obj
[36]     ! delete if found
[37]     if ( found ) then
[38]       previous%next => current%next ! redirect pointer
[39]       deallocate ( current )       ! free space for node
[40]     end if
[41]   end subroutine S_L_delete
[42]

```

Fig. 8.5, A Typical Singly Linked List Class of Objects (continued)

```

[43] subroutine S_L_insert (links, Obj )
[44]   type (S_L_list), intent (inout) :: links
[45]   type (Object),  intent (in)    :: Obj
[46]   type (S_L_node), pointer       :: previous, current
[47]
[48]   ! Find location to insert a new object
[49]   previous => links%first           ! initialize
[50]   current  => previous%next        ! initialize
[51]   do
[52]     if ( .not. associated (current) ) exit ! insert at end
[53]     if ( Obj < current%value ) exit      ! *** OVERLOADED ***
[54]     previous => current                ! insert before current
[55]     current  => current%next           ! move to next node
[56]   end do ! to locate insert node
[57]   ! Insert before current (duplicates allowed)
[58]   allocate ( previous%next )          ! get new node space
[59]   previous%next%value = Obj           ! new object inserted
[60]   previous%next%next => current       ! new next pointer
[61] end subroutine S_L_insert
[62]
[63] function is_S_L_empty (links) result (t_or_f)
[64]   type (S_L_list), intent (in) :: links
[65]   logical                    :: t_or_f
[66]   t_or_f = .not. associated ( links%first%next )
[67] end function is_S_L_empty
[68]
[69] function S_L_new () result (new_list)
[70]   type (S_L_list) :: new_list
[71]   allocate ( new_list%first )          ! get memory for the object
[72]   nullify ( new_list%first%next )     ! begin with empty list
[73] end function S_L_new
[74]
[75] subroutine print_S_L_list (links)
[76]   type (S_L_list), intent (in) :: links
[77]   type ( S_L_node), pointer    :: current
[78]   integer                      :: counter
[79]   current => links%first%next
[80]   counter = 0 ; print *, 'Link   Object Value'
[81]   do
[82]     if ( .not. associated (current) ) exit ! list end
[83]     counter = counter + 1
[84]     print *, counter, ' ', current%value
[85]     current => current%next
[86]   end do
[87] end subroutine print_S_L_list
[88] end module singly_linked_list

```

Figure 7.10: A Typical Singly Linked List Class of Objects

```

[ 1] program main ! test a singly linked object list
[ 2] use singly_linked_list
[ 3] implicit none
[ 4] type (S_L_list) :: container
[ 5] type (Object)   :: Obj_1, Obj_2, Obj_3, Obj_4
[ 6] logical        :: delete_ok
[ 7]
[ 8]   Obj_1 = Object(15) ; Obj_2 = Object(25) ! constructor
[ 9]   Obj_3 = Object(35) ; Obj_4 = Object(45) ! constructor
[10]   container = S_L_new()
[11]   print *, 'Empty status is ', is_S_L_empty (container)
[12]   call S_L_insert (container, Obj_4) ! insert object
[13]   call S_L_insert (container, Obj_2) ! insert object
[14]   call S_L_insert (container, Obj_1) ! insert object
[15]   call print_S_L_list (container)
[16]
[17]   call S_L_delete (container, obj_2, delete_ok)
[18]   print *, 'Object: ', Obj_2, ' deleted status is ', delete_ok
[19]   call print_S_L_list (container)
[20]   print *, 'Empty status is ', is_S_L_empty (container)
[21]
[22]   call S_L_insert (container, Obj_3) ! insert object
[23]   call print_S_L_list (container)
[24]   call S_L_delete (container, obj_1, delete_ok)
[25]   print *, 'Object: ', Obj_1, ' deleted status is ', delete_ok
[26]   call S_L_delete (container, obj_4, delete_ok)
[27]   print *, 'Object: ', Obj_4, ' deleted status is ', delete_ok
[28]   call print_S_L_list (container)
[29]   print *, 'Empty status is ', is_S_L_empty (container)
[30]
[31]   call S_L_delete (container, obj_3, delete_ok)
[32]   print *, 'Object: ', Obj_3, ' deleted status is ', delete_ok
[33]   print *, 'Empty status is ', is_S_L_empty (container)
[34]   call print_S_L_list (container)
[35] end program ! running yields
[36] ! Empty status is T
[37] ! Link   Object Value
[38] ! 1      15
[39] ! 2      25
[40] ! 3      45
[41] ! Object: 25 deleted status is T
[42] ! Link   Object Value
[43] ! 1      15
[44] ! 2      45
[45] ! Empty status is F
[46] ! Link   Object Value
[47] ! 1      15
[48] ! 2      35
[49] ! 3      45
[50] ! Object: 15 deleted status is T
[51] ! Object: 45 deleted status is T
[52] ! Link   Object Value
[53] ! 1      35
[54] ! Empty status is F
[55] ! Object: 35 deleted status is T
[56] ! Empty status is T
[57] ! Link   Object Value

```

Figure 7.11: Testing the singly linked list with integers

```

[ 1] module class_Object
[ 2] implicit none
[ 3] type Object ! An integer object for testing lists
[ 4]   integer :: data ; end type Object
[ 5]
[ 6] interface operator (<) ! for sorting or insert
[ 7]   module procedure less_than_Object ; end interface
[ 8] interface operator (==) ! for sorting or delete
[ 9]   module procedure equal_to_Object ; end interface
[10]
[11] contains ! overload definitions only
[12] function less_than_Object (Obj_1, Obj_2) result (Boolean)
[13]   type (Object), intent(in) :: Obj_1, Obj_2
[14]   logical :: Boolean
[15]   Boolean = Obj_1%data < Obj_2%data ! standard (<) here
[16] end function less_than_Object
[17] function equal_to_Object (Obj_1, Obj_2) result (Boolean)
[18]   type (Object), intent(in) :: Obj_1, Obj_2
[19]   logical :: Boolean
[20]   Boolean = Obj_1%data == Obj_2%data ! standard (==) here
[21] end function equal_to_Object
[22] end module class_Object

```

Figure 7.12: Typical object definition to test a singly linked list


```

[ 1] module doubly_linked_list
[ 2] use class_Object
[ 3] implicit none
[ 4] type D_L_node
[ 5]   private
[ 6]     type (Object)           :: Obj
[ 7]     type (D_L_node), pointer :: previous
[ 8]     type (D_L_node), pointer :: next
[ 9]   end type D_L_node
[10]
[11] type D_L_list
[12]   private
[13]     type (D_L_node), pointer :: header
[14]   end type D_L_list
[15]
[16] contains
[17]
[18] function D_L_new () result (new_list)      ! constructor
[19]   type (D_L_list) :: new_list
[20]   allocate (new_list % header)
[21]   nullify (new_list % header % previous)
[22]   nullify (new_list % header % next)
[23] end function D_L_new
[24]
[25] subroutine destroy_D_L_List (links)      ! destructor
[26]   type (D_L_list), intent (in) :: links
[27]   type (D_L_node), pointer     :: current
[28]   do
[29]     current => links % header % next
[30]     if ( .not. associated ( current ) ) exit
[31]     current % previous % next => current % next
[32]     if ( associated ( current % next ) ) then
[33]       current % next % previous => current % previous
[34]     end if
[35]     nullify ( current % previous )
[36]     nullify ( current % next )
[37]     print *, 'Destroying object ', current % Obj
[38]     deallocate ( current )
[39]   end do
[40]   deallocate ( links % header )
[41]   print *, 'D_L_List destroyed'
[42] end subroutine destroy_D_L_List
[43]

```

Fig. 7.14, A Typical Doubly Linked List Class of Objects (continued)

with it. The printing member (line 90) is called an *iterator* since it runs through all objects in the list. The testing program for this container type and its output results are given in Fig. 7.15. Here an object with a single integer value was selected, and thus it was easy to overload the relational operators with a clear meaning as shown in Fig. 7.12.

7.5 Direct (Random) Access Files

Often it may not be necessary to create special object data structures such as those outlined above. From its beginning Fortran has had the ability to create a sophisticated random access data structure where the implementation details are hidden from its user. This was necessary originally since the language was utilized on computers with memory sizes that are considered tiny by today's standard (e.g., 16 Kb), but it was still necessary to efficiently create and modify large amounts of data. The standard left the actual implementation details to the compiler writers. That data structure is known as a "direct access file". It behaves like a single subscript array in that the object at any position can be read, modified, or written at random so long as the user keeps up with the position of interest. The user simply supplies the position, known as the record number, as additional information in the read and write statements. With today's hardware, if the file is stored on a virtual disk (stored in random access memory) there is practically no difference in access times for arrays and direct files.

It should be noted here that since pointers are addresses in memory they can not be written to any type of file. That, of course, means that no object having a pointer as an attribute can be written either. Thus in some cases one must employ the other types of data structures illustrated earlier in the chapter.

To illustrate the basic concepts of a random access file consider the program called `random_access_file` which is given in Fig. 7.16. In this case the object is simply a character string, as

```

[ 43] subroutine D_L_insert_before (links, values)
[ 44]   type (D_L_list), intent (in) :: links
[ 45]   type (Object),   intent (in) :: values
[ 46]   type (D_L_node), pointer      :: current      ! Temp traversal pointer
[ 47]   type (D_L_node), pointer      :: trailing     ! Preceding node pointer
[ 48]   ! Find location to insert new node, in ascending order
[ 49]   trailing => links % header      ! initialize
[ 50]   current  => trailing % next     ! initialize
[ 51]   do
[ 52]     if (.not. associated (current)) exit      ! insert at end
[ 53]     if (values < current % Obj ) exit        ! insert before current
[ 54]     trailing => current                  ! move to next node
[ 55]     current  => current % next           ! move to next node
[ 56]   end do
[ 57]   ! Insert before current (duplicates allowed)
[ 58]   allocate (trailing % next)              ! get new node space
[ 59]   trailing % next % Obj = values          ! new object inserted
[ 60]   ! Insert the new pointers
[ 61]   if (.not. associated (current)) then     ! End of list (special)
[ 62]     nullify (trailing % next % next)
[ 63]     trailing % next % previous => trailing
[ 64]   else
[ 65]     trailing % next % next => current      ! Not the end of the list
[ 66]     trailing % next % previous => trailing
[ 67]     current % previous => trailing % next
[ 68]   end if
[ 69] end subroutine D_L_insert_before
[ 70]
[ 71] function Get_Obj_at_Ptr (ptr_to_Obj) result ( values)
[ 72]   type (D_L_node), intent (in) :: ptr_to_Obj
[ 73]   type (Object)                :: values    ! intent out
[ 74]   values = ptr_to_Obj % Obj
[ 75] end function Get_Obj_at_Ptr
[ 76]
[ 77] function Get_Ptr_to_Obj (links, values) result (ptr_to_Obj)
[ 78]   type (D_L_list), intent (in) :: links    ! D_L_list header
[ 79]   type (Object),   intent (in) :: values   ! Node identifier Object
[ 80]   type (D_L_node), pointer      :: ptr_to_Obj ! Pointer to the Object
[ 81]   type (D_L_node), pointer      :: current  ! list traversal pointer
[ 82]   current => links % header % next
[ 83]   do ! Search list, WARNING: runs forever if values not in list
[ 84]     if (current % Obj == values) exit      ! *** OVERLOADED ***
[ 85]     current => current % next
[ 86]   end do
[ 87]   ptr_to_Obj => current                    ! Return pointer to node
[ 88] end function Get_Ptr_to_Obj
[ 89]
[ 90] subroutine print_D_L_list ( links )
[ 91]   type (D_L_list), intent (in) :: links
[ 92]   type (D_L_node), pointer      :: current ! Node traversal pointer
[ 93]   integer                       :: counter ! Link position
[ 94]   ! Traverse the list and print its contents to standard output
[ 95]   current => links % header % next
[ 96]   counter = 0 ; print *, 'Link   Object Value'
[ 97]   do
[ 98]     if (.not. associated (current)) exit
[ 99]     counter = counter + 1
[100]     print *, counter, ' ', current % Obj
[101]     current => current % next
[102]   end do
[103] end subroutine print_D_L_list
[104] end module doubly_linked_list

```

Figure 7.14: A Typical Doubly Linked List Class of Objects

defined in line 4. The hardware transportability of this code is assured by establishing the required constant record with the intrinsic given in line 10. It is then used in opening the file, which is designated as a direct file in line 12. Lines 16–24 create the object record numbers in a sequential fashion. They also define the new object to be stored with each record. In lines 27–32 the records are accessed in a backwards order, but could have been accessed in any random or partial order. In line 35 a random object is given a new value. Finally, the changes are output in a sequential order in lines 37–42. Sample input data and program outputs are included as comments at the end of the program.

```

[ 1] program main
[ 2] use doubly_linked_list
[ 3] implicit none
[ 4]   type (D_L_list)           :: container
[ 5]   type (Object)            :: Obj_1, Obj_2, Obj_3, Obj_4
[ 6]   type (Object)            :: value_at_pointer
[ 7]   type (D_L_node), pointer :: point_to_Obj_3
[ 8]
[ 9]   Obj_1 = Object(15) ; Obj_2 = Object(25)
[10]   Obj_3 = Object(35) ; Obj_4 = Object(45)
[11]   container = D_L_new()
[12]   ! print *, 'Empty status is ', is_D_L_empty (container)
[13]   call D_L_insert_before (container, Obj_4)
[14]   call D_L_insert_before (container, Obj_2)
[15]   call D_L_insert_before (container, Obj_1)
[16]   call D_L_insert_before (container, Obj_3)
[17]   call print_D_L_list (container)
[18]
[19]   ! find and get Obj_3
[20]   point_to_Obj_3 = Get_Ptr_to_Obj (container, Obj_3)
[21]   value_at_pointer = Get_Obj_at_Ptr (point_to_Obj_3)
[22]   print *, 'Object: ', Obj_3, ' has a value of ', value_at_pointer
[23]   call destroy_D_L_List (container)
[24] end program main                               ! Running gives:
[25] ! Link      Object Value
[26] ! 1         15
[27] ! 2         25
[28] ! 3         35
[29] ! 4         45
[30] ! Object:   35 has a value of 35
[31] ! Destroying object 15
[32] ! Destroying object 25
[33] ! Destroying object 35
[34] ! Destroying object 45
[35] ! D_L_List destroyed

```

Figure 7.15: Testing a Partial Doubly Linked List

```

[ 1] program random_access_file
[ 2] ! create a file and access or modify it randomly
[ 3] implicit none
[ 4] character(len=10) :: name
[ 5] integer :: j, rec_len, no_name, no_open
[ 6] integer :: names = 0, unit = 1
[ 7]
[ 8] ! find the hardware dependent record length of the object
[ 9] ! to be stored and modified. Then open a binary file.
[10] inquire (iolen = rec_len) name
[11] open (unit, file = "random_list", status = "replace",
[12]       access = "direct", recl = rec_len,
[13]       form = "unformatted", iostat = no_open)
[14] if ( no_open > 0 ) stop 'open failed for random_list'
[15]
[16] ! read and store the names sequentially
[17] print *, ' '; print *, 'Original order'
[18] do ! forever from standard input
[19]   read (*, '(a)', iostat = no_name) name
[20]   if ( no_name < 0 ) exit ! the read loop
[21]   names = names + 1 ! record number
[22]   write (unit, rec = names) name ! save record
[23]   print *, name ! echo
[24] end do
[25] if ( names == 0 ) stop 'no records read'
[26]
[27] ! list names in reverse order
[28] print *, ' '; print *, 'Reverse order'
[29] do j = names, 1, -1
[30]   read (unit, rec = j) name
[31]   print *, name
[32] end do ! of random read
[33]
[34] ! change the middle name in random file
[35] write (unit, rec = (names + 1)/2) 'New_Name'
[36]
[37] ! list names in original order
[38] print *, ' '; print *, 'Modified data'
[39] do j = 1, names
[40]   read (unit, rec = j) name
[41]   print *, name
[42] end do ! of random read
[43]
[44] close (unit) ! replace previous records and save
[45] end program random_access_file
[46] ! Running with input of:   Name_1
[47] !                           B_name
[48] !                           3_name
[49] !                           name_4
[50] !                           Fifth
[51] ! Yields:
[52] ! Original order  Reverse order  Modified data
[53] ! Name_1          Fifth          Name_1
[54] ! B_name          name_4         B_name
[55] ! 3_name          3_name         New_Name
[56] ! name_4          B_name         name_4
[57] ! Fifth           Name_1         Fifth

```

Figure 7.16: Utilizing a Random Access File as a Data Structure

7.6 Exercises