A generic linked list implementation in Fortran 95

Article *in* ACM SIGPLAN Fortran Forum · April 2009

DOI: 10.1145/1667140.1667141

Jason R Blevins
The Ohio State University

A Generic Linked List Implementation in Fortran 95[*]

JASON R. BLEVINS[†]
Department of Economics, Duke University

May 18, 2009

Abstract. This paper develops a standard conforming generic linked list in Fortran 95 which is capable of storing data of an any type. The list is implemented using the `transfer` intrinsic function, and although the interface is generic, it remains relatively simple and minimizes the potential for error. Although linked lists are the focus of this paper, the generic programming techniques used are very general and broadly-applicable to other data structures and procedures implemented in Fortran 95 that need to be used with data of an unknown type.

Keywords: Fortran 95, generic programming, data structures, scientific computing.

1. INTRODUCTION

A linked list, or more specifically a singly-linked list, is a list consisting of a series of individual node elements where each node contains a data member and a pointer that points to the next node in the list. This paper describes a *generic* linked list implementation in standard Fortran 95 which is able to store arbitrary data, and in particular, pointers to arbitrary data types. While linked lists are the focus of this paper, the underlying idea can be easily used to create more general generic data structures and procedures in Fortran 95. The `generic_list` module is developed in detail below, followed by an example control program which illustrates the list interface, showing how to use Fortran's `transfer` intrinsic function to store and retrieve pointers to derived type data variables.

This module has several advantages: it is written in standard Fortran 95 to ensure portability, it is self-contained and does not require the use of a preprocessor or `include` statements, and the `transfer` function only needs to be called once by the user, resulting in clean code and minimizing the potential for errors. Furthermore, simple wrapper functions can be written for specific types so that `transfer` would not need to be called at all by the user.

[*]The author benefited greatly from Arjen Markus's open source FLIBS project at `http://flibs.sourceforge.net/`, which illustrates several generic programming techniques, and from reading Richard Maine's descriptions of using `transfer` to pass pointers, posted to `comp.lang.fortran`.

[†]Email: `jason.r.blevins@duke.edu`

Several other generic list implementations have been offered, each with its own advantages and disadvantages. The generic list of McGavin and Young (2001) is very similar to the one presented here, but it is not standard-conforming due to an assumption about the physical representation of pointers to derived types. The method described below avoids this problem by writing the list interface in such a way that the compiler automatically determines the exact amount of storage required for any arbitrary data type the user wishes to store in the list.

The FLIBS project (Markus, 2008) provides a generic linked list in the form of an include file. The user then creates a new module for each data type that will be stored in a list. The module then includes the generic list code. This approach is clean in that the compiler can perform type checking to reduce errors, however, it requires the user to create potentially many new modules and each list can only store data of a single type. The method described in this paper does not require the user to write any additional code.

The `generic_list` module it is not, however, without flaws. In particular, the `transfer` function can be confusing for new Fortran programmers and must be used with care to avoid subtle errors. Furthermore, a disadvantage with respect to type-aware lists is that when storing pointers to data elements in the generic list, the elements must be allocated and deallocated by the user. This is the case with all dynamically allocated memory in Fortran, but it is nonetheless an area where care should be taken when implementing the generic programming techniques used herein.

While the `generic_list` implementation in this paper uses the `transfer` function, other generic programming techniques are also possible. A preprocessor such as the C preprocessor, m4, or a Fortran-specific preprocessor such as Forpedo (McCormack, 2005) can be used to automatically generate specific code from generic code. Markus (2001) discusses generic programming in Fortran 90 using both text substitution and implementation-hiding. The former is a form of preprocessing while the latter is accomplished by writing, say, a quicksort (Hoare, 1961) routine in such a way that it requires passing only two *procedures* for swapping and comparing data elements, rather than passing the data array itself. The swap and compare procedures have a generic interface and so the resulting sort routine is type-independent.

It is also worth nothing that with the introduction of Fortran 2003, the methods discussed in this paper will no longer be necessary. Instead, new features such as unlimited polymorphic (`class(*)`) objects or C pointers (variables of `type(c_ptr)`) can be used to implement generic data structures in a more natural way. However, until Fortran 2003 compliant compilers become widely available, Fortran 95 techniques such as these will remain useful.

In the following, we first briefly review Fortran pointers and the `transfer` intrinsic before presenting the `generic_list` module and an example program.

2. POINTERS AND THE TRANSFER INTRINSIC

Fortran 95 has no syntax for constructing arrays of pointers and it requires some additional work to manipulate pointers themselves rather than the pointer targets. A common idiom involves constructing a derived type containing only a pointer to the data in question. The same approach can be used to store pointers in the generic list presented here. Consider, for example, a representative data type called `data_t` and an associated derived type for storing `data_t` pointers called `data_ptr`.

```
! A representative derived type for storing data
module data
  implicit none

  private
  public :: data_t
  public :: data_ptr

  ! Data is stored in data_t
  type :: data_t
     real :: x
  end type data_t

  ! A container for storing data_t pointers
  type :: data_ptr
     type(data_t), pointer :: p
  end type data_ptr

end module data
```

The `data_t` type is very simple, holding only a single real variable, but such data container types can easily become very complex in specific applications. As such, rather than using `transfer` to copy entire `data_t` structures in and out of the list, it is sometimes desirable to copy the smaller `data_ptr` structures instead. The `data_ptr` type also allows us to create arrays of pointers:

```
type(data_ptr), dimension(100) :: pointer_array
```

In most cases, it is these pointers that we would like to store in the linked list nodes.

Rather than build a new list for each possible type we might want to store, we use the `transfer` function to convert data of an arbitrary type to an array of integers before storing it in the list. We thus only have to write one linked-list module which is capable of handling rank-one integer arrays.

The `transfer` function introduced in Fortran 90 can be used to move data of one type through procedures and variables that were expecting data of some other type. In this sense, `transfer` can approximate the behavior of void pointers in C. Essentially, it copies the bits in memory representing a variable `source` to a scalar or array of the same type as a given `mold`. The syntax for `transfer` is

```
result = transfer(source, mold[, size])
```

where `source` and `mold` are scalars or arrays of any type and `size` is an optional scalar of type `integer`. The `result` is of the same type as `mold`. If `size` is given, `result` is a rank-one array of length `size`. If `size` is omitted but `mold` is an array, then `result` is an array just large enough to represent the `source`. Finally, if `size` is omitted and `mold` is a scalar, then `result` is a scalar.

Returning to our `data_ptr` pointer example above, we can convert pointers to integer arrays by using `transfer` as follows:

```
program transfer_ptr
  use data
  implicit none

  type(data_ptr) :: ptr
  type(data_t), target :: dat
  integer, dimension(:), allocatable :: iarr
  integer :: len

  dat%x = 3.1416
  ptr%p => dat

  len = size(transfer(ptr, iarr))
  allocate(iarr(len))
  iarr = transfer(ptr, iarr)
  deallocate(iarr)
end program transfer_ptr
```

The above code makes two calls to `transfer`: first, to probe the size of the array needed to store the pointer in integer form, and second, to actually transfer the data. We must check the size for generality. It might be the case that the `data_t` pointer uses the same amount of storage as a single default `integer`, but the Fortran standard does not guarantee this. Furthermore, we might wish to store other types of data in the list, not just pointers.

On the author's system, for one run of the `transfer_ptr` program above, the values of `len` and `iarr` were 2 and $(1054311824, 32767)$ respectively. Note that the value of `iarr` represents the location of `dat` in memory and will likely differ each time the program is executed.

The `transfer` construction above is verbose, error prone, and inefficient. The interface of the `generic_list` implementation is designed to minimize the inconvenience of using `transfer` so that only a single call to `transfer` is required to store or retrieve data from the list. Instead of manually allocating the temporary array, we can use the `transfer` statement as an argument expression, along with assumed-shape arrays in the list interface, and let the compiler determine the appropriate size, thus avoiding one call each to `size` and `transfer`. Although the compiler will create a temporary array, it replaces the manually-allocated array in the above example which will not be required.

3. IMPLEMENTATION

The `generic_list` module below defines the `list_node_t` type, from which the list is constructed, and the related procedures for initializing and freeing the list and manipulating list nodes. Although the `data` element of each list node is defined as a pointer to an array of integers, the list is not intended to simply store integers. Rather, Fortran's `transfer` intrinsic will be used to "encode" any arbitrary data type so that it can be represented as an array of integers. When data needs to be accessed later, `transfer` is used again to "decode" it, returning it to its original type. Using the methods of the previous section, we can also store pointers to arbitrary data types in the list.

The module first defines a few types including the `list_node_t` type which stores the `data` and well as a variable `list_data` which is defined for convenience to be used as a mold for `transfer`.

```
module generic_list
  implicit none

  private
  public :: list_node_t, list_data
  public :: list_init, list_free
  public :: list_insert, list_put, list_get, list_next

  ! A public variable used as a MOLD for transfer()
  integer, dimension(:), allocatable :: list_data

  ! Linked list node
  type :: list_node_t
     private
     integer, dimension(:), pointer :: data => null()
     type(list_node_t), pointer :: next => null()
  end type list_node_t

contains
```

```
  ! Procedures to initialize and free generic_list objects
  include 'list_init.f90'
  include 'list_free.f90'

  ! Basic list operations such as insert and next
  include 'list_operations.f90'

  ! List data accessors
  include 'list_accessors.f90'

end module generic_list
```

Two life-cycle procedures are defined to initialize memory for the list (`list_init`) and to free the memory once the list is no longer needed (`list_free`). The `list_init` subroutine allocates memory for a "head node" and nullifies the `next` pointer. It also takes an optional `data` argument to allow the first data object to be stored upon initialization.

```
! Initialize a head node SELF and optionally store the provided DATA.
subroutine list_init(self, data)
  type(list_node_t), pointer :: self
  integer, dimension(:), intent(in), optional :: data

  allocate(self)
  nullify(self%next)

  if (present(data)) then
     allocate(self%data(size(data)))
     self%data = data
  else
     nullify(self%data)
  end if
end subroutine list_init
```

The `list_free` procedure traverses the list, deallocating list nodes and their data until the end of the list is reached, as indicated by a node with a `null` `next` pointer.

```
! Free the entire list and all data, beginning at SELF
subroutine list_free(self)
  type(list_node_t), pointer :: self
  type(list_node_t), pointer :: current
  type(list_node_t), pointer :: next

  current => self
  do while (associated(current))
```

```
     next => current%next
     if (associated(current%data)) then
        deallocate(current%data)
        nullify(self%data)
     end if
     deallocate(current)
     nullify(current)
     current => next
  end do
end subroutine list_free
```

Since the list node data is encapsulated by the `private` attribute in order to hide the implementation, three simple accessor procedures `list_put`, `list_get`, and `list_next` are also defined. `list_put` stores encoded data in a particular list node and `list_get` retrieves data. The `list_next` function returns the next node in the list.

```
! Store the encoded DATA in list node SELF
subroutine list_put(self, data)
  type(list_node_t), pointer :: self
  integer, dimension(:), intent(in) :: data

  if (associated(self%data)) then
     deallocate(self%data)
     nullify(self%data)
  end if
  self%data = data
end subroutine list_put

! Return the DATA stored in the node SELF
function list_get(self) result(data)
  type(list_node_t), pointer :: self
  integer, dimension(:), pointer :: data
  data => self%data
end function list_get

! Return the next node after SELF
function list_next(self)
  type(list_node_t), pointer :: self
  type(list_node_t), pointer :: list_next
  list_next => self%next
end function list_next
```

Although many additional list operations could be defined, we provide only a single example `list_insert` procedure which inserts a new node after the current one.

```
! Insert a list node after SELF containing DATA (optional)
subroutine list_insert(self, data)
  type(list_node_t), pointer :: self
  integer, dimension(:), intent(in), optional :: data
  type(list_node_t), pointer :: next

  allocate(next)

  if (present(data)) then
     allocate(next%data(size(data)))
     next%data = data
  else
     nullify(next%data)
  end if

  next%next => self%next
  self%next => next
end subroutine list_insert
```

Finally, we provide a simple control program which illustrates how to initialize and free the list and how to store and retrieve pointers to `data_t` objects using `transfer`.

```
program test_list
  use generic_list
  use data
  implicit none

  type(list_node_t), pointer :: list => null()
  type(data_ptr) :: ptr

  ! Allocate a new data element
  allocate(ptr%p)
  ptr%p%x = 2.7183

  ! Initialize the list with the first data element
  call list_init(list, transfer(ptr, list_data))
  print *, 'Initializing list with data:', ptr%p

  ! Allocate a second data element
  allocate(ptr%p)
  ptr%p%x = 0.5772
  ! Insert the second into the list
  call list_insert(list, transfer(ptr, list_data))
  print *, 'Inserting node with data:', ptr%p

  ! Retrieve data from the second node and free memory
  ptr = transfer(list_get(list_next(list)), ptr)
  print *, 'Second node data:', ptr%p
  deallocate(ptr%p)

  ! Retrieve data from the head node and free memory
  ptr = transfer(list_get(list), ptr)
  print *, 'Head node data:', ptr%p
  deallocate(ptr%p)

  ! Free the list
  call list_free(list)
end program test_list
```

The test program produces the following output:

```
 Initializing list with data:   2.7183001
 Inserting node with data:  0.57720000
 Second node data:  0.57720000
 Head node data:   2.7183001
```

4. CONCLUSION

The generic programming methods described in this paper provide a relatively simple way to structure Fortran 95 modules and procedures that need to be used with data of an unknown type. These methods are general and broadly-applicable to data structures and procedures far beyond the simple linked list considered in this paper. In specific applications, a few simple wrapper functions can be written for the required data types so that the resulting interface is clean and simple, entirely avoiding direct use of `transfer` by the user. Until the new language features of Fortran 2003 become more accessible, generic Fortran 95 techniques such as these can provide much of the same convenience and functionality at the cost of a slightly more complicated user interface.

REFERENCES

Hoare, C. A. R. (1961). Quicksort: Algorithm 64. *Communications of the ACM 4*, 321–322. [2]

Markus, A. (2001). Generic programming in Fortran 90. *ACM SIGPLAN Fortran Forum 20*(3), 20–23. [2]

Markus, A. (2008). FLIBS - a collection of Fortran modules. http://flibs.sourceforge.net/. Version 0.9. [2]

McCormack, D. (2005). Generic programming in Fortran with Forpedo. *ACM SIGPLAN Fortran Forum 24*(2), 18–29. [2]

McGavin, P. and R. Young (2001). A generic list implementation. *ACM SIGPLAN Fortran Forum 20*(1), 16–20. [2]