Avoiding memory leaks with derived types

Arjen Markus[1]
WL | Delft Hydraulics
PO Box 177
2600 MH Delft
The Netherlands

Abstract

In this short note a solution is presented for one particular type of memory leaks that can occur with derived types. With the advent of Fortran 2X and before that the adoption of Technical Report ISO/IEC 15581: 1998(E) (the "allocatable array extension") this solution will be superfluous, nevertheless it seems worthwhile to describe it, as it can solve the problem in the short term.

Introduction

In well-known modules, such as the module to support strings of varying length and the varying-precision arithmetic module (Schonfelder, 1994, 2000), derived types are introduced that hold a pointer component. This pointer is necessary because the contents may change in size without bounds. Because there are also functions defined that return these derived types as a result, which often implement *operators* like + or assigments via =, memory leaks will appear: the function result is assigned to an ordinary variable – but the allocated memory becomes inaccessible (*cf.* Metcalf and Reid, 1999).

One alternative to avoid this is to use *subroutines* instead of functions and assignments, so that there are no intermediate results, but this causes a rather awkward way of working. Compare:

```
call sum( a, b, c )
```

with:

```
a = b + c    ! Resolves into the function call below
```

or:

```
a = sum( b, c )
```

Another solution is to *mark* the derived types, so that the allocated memory can be deallocated when it is no longer needed. This solution will be illustrated using a simple (if impractical) module, which concatenates arrays of integers.

Sample module: chains

The purpose of the module *chains* is to store and manipulate integer arrays – manipulation being limited to assignment and concatenation:

```
module chains
   type chain
      integer, dimension(:), pointer :: values => null()
   end type chain

   interface assignment(=)
      module procedure assign_chain
      module procedure assign_array
```

[1] E-mail address: arjen.markus@wldelft.nl

```
   end interface assignment(=)

   interface operator(.concat.)
      module procedure concat_chain
   end interface operator(.concat.)

contains

subroutine assign_array( ic, jc )
   type(chain),intent(out) :: ic
   integer, dimension(:)   :: jc

   if ( associated( ic%values ) ) deallocate( ic%values )
   allocate( ic%values(1:size(jc)) )
   ic%values = jc
end subroutine assign_array

subroutine assign_chain( ic, jc )
   type(chain), intent(inout) :: ic
   type(chain)                :: jc

   if ( associated( ic%values ) ) deallocate( ic%values )
   allocate( ic%values(1:size(jc%values)) )
   ic%values = jc%values

end subroutine assign_chain

function concat_chain( ic, jc )
   type(chain), intent(in) :: ic
   type(chain), intent(in) :: jc
   type(chain)             :: concat_chain
   integer :: nic
   integer :: njc

   nic = size(ic%values)
   njc = size(jc%values)

   allocate( concat_chain%values(1:nic+njc) )
   concat_chain%values(1:nic)         = ic%values(1:nic)
   concat_chain%values(nic+1:nic+njc) = jc%values(1:njc)

end function concat_chain

end module chains
```

Whenever assigning a new value to a variable of this type, any old memory must be deallocated and new memory of the right size allocated (as shown in the subroutines *assign_array* and *assign_chain*). Otherwise memory would be referenced twice or get lost.[2]

The following statement presents a problem:

```
kc = ic .concat. jc
```

because the intermediate result from the concatenation operator can not be deallocated – the memory leak we are trying to avoid.

The root cause is that we do not know that the data that are being copied are in fact temporary results. So the solution is to mark the result of any function as *temporary*. We modify the definition of the derived type slightly:

```
type chain
```

[2] As Fortran 90 does *not* allow automatic initialisation of derived types, especially ones with pointers, variables must be initialised explicitly. In the rest of this note we assume Fortran 95 to keep the discussion clear.

```
      integer, dimension(:), pointer :: values => null()
      logical                        :: tmp    = .false.
   end type chain
```

With this new type the function *concat_chain()* can mark its result as temporary. All functions in the module now check whether their arguments are temporary and clean them up if that is the case, *as they will not be used anymore*:

```
   function concat_chain( ic, jc )
      type(chain), intent(in) :: ic
      type(chain), intent(in) :: jc
      type(chain)             :: concat_chain
      integer :: nic
      integer :: njc

      nic = size(ic%values)
      njc = size(jc%values)

      allocate( concat_chain%values(1:nic+njc) )
      concat_chain%values(1:nic)         = ic%values(1:nic)
      concat_chain%values(nic+1:nic+njc) = jc%values(1:njc)

      concat_chain%tmp = .true.     ! Mark as temporary

      call cleanup( ic, .true. )    ! Clean up temporary arguments
      call cleanup( jc, .true. )
   end function concat_chain

   end module chains
```

Sample program: a test

As a small test of the above ideas, here is a complete program that uses an extra field to identify which “chain” variables are created and subsequently cleaned up again (this is the task of the routine *cleanup()* that hides the details of the process and can be used by the test program too):

```
! test_chain --
!    Test program to see if memory leaks originating from derived-types
!    can be circumvented
!    The idea:
!    - The user is responsible for cleaning up his/her own variables
!    - The module is responsible for cleaning up its intermediate
!      results (flagged as "temporary")
!
!    Note:
!    - seqno and alloc_seq are only used for debugging purposes
!
module chains
   type chain
      integer, dimension(:), pointer :: values => null()
      logical                        :: tmp    =  .false.
      integer                        :: seqno  =  0
   end type chain

   integer :: seqno = 0
   logical, dimension(1:100) :: alloc_seq = .false.

   interface assignment(=)
      module procedure assign_chain
      module procedure assign_array
   end interface

   interface operator(.concat.)
      module procedure concat_chain
   end interface
```

```
contains

subroutine assign_array( ic, jc )
   type(chain),intent(out)           :: ic
   integer, dimension(:), intent(in) :: jc

   call cleanup( ic, .false. )

   allocate( ic%values(1:size(jc)) )
   seqno = seqno + 1
   alloc_seq(seqno) = .true.
   ic%values = jc
   ic%seqno  = seqno
   ic%tmp    = .false.

end subroutine assign_array

subroutine assign_chain( ic, jc )
   type(chain), intent(inout) :: ic
   type(chain), intent(in)    :: jc

   call cleanup( ic, .false. )

   allocate( ic%values(1:size(jc%values)) )
   seqno = seqno + 1
   alloc_seq(seqno) = .true.
   ic%values = jc%values
   ic%seqno  = seqno
   ic%tmp    = .false.

   call cleanup( jc, .true. )

end subroutine assign_chain

function concat_chain( ic, jc )
   type(chain), intent(in) :: ic
   type(chain), intent(in) :: jc
   type(chain)             :: concat_chain
   integer                 :: nic
   integer                 :: njc

   nic = size(ic%values)
   njc = size(jc%values)

   allocate( concat_chain%values(1:nic+njc) )
   seqno = seqno + 1
   alloc_seq(seqno) = .true.

   concat_chain%values(1:nic)         = ic%values(1:nic)
   concat_chain%values(nic+1:nic+njc) = jc%values(1:njc)
   concat_chain%seqno  = seqno
   concat_chain%tmp    = .true.

   call cleanup( ic, .true. )
   call cleanup( jc, .true. )

end function concat_chain

subroutine cleanup( ic, only_tmp )
   type(chain) :: ic
   logical, optional :: only_tmp
   logical :: clean_tmp

   clean_tmp = .false.
   if ( present(only_tmp) ) clean_tmp = only_tmp

   if ( .not. clean_tmp .or. ic%tmp ) then
      if ( associated( ic%values) ) deallocate( ic%values )
```

```
      if ( ic%seqno .gt. 0 ) alloc_seq(ic%seqno) = .false.
   endif

end subroutine cleanup

subroutine report_chain
   integer :: i
   integer :: count

   count = 0
   do i = 1,size(alloc_seq)
      if ( alloc_seq(i) ) then
         write(*,*) 'Allocated item', i
         count = count + 1
      endif
   enddo
   write(*,*) 'Number of allocated items:', count

end subroutine report_chain

end module chains

program test_chain
   use chains

   type(chain) :: ic
   type(chain) :: jc
   type(chain) :: kc

   ic = (/1,2,3/)              ! Create item 1
   jc = (/4,5/)                ! Create item 2
   kc = ic .concat. jc          ! Function result is item 3, assigned to item
4
   call report_chain
   kc = jc .concat. ic          ! Function result is item 5, assigned to item
6
   call report_chain
   call cleanup(ic)
   call cleanup(jc)
   call cleanup(kc)
   call report_chain

end program test_chain
```

The output from this program is:

```
 Allocated item           1
 Allocated item           2
 Allocated item           4
 Number of allocated items:           3
 Allocated item           1
 Allocated item           2
 Allocated item           6
 Number of allocated items:           3
 Number of allocated items:           0
```

The source code can be found at: http://ftp.cac.psu.edu/pub/ger/fortran/Markus/noleaks.f90 (courtesy of H.D. Knoble)

Discussion

To effectively avoid all memory leaks using this technique puts some burden on the programmer of these modules and on the user too, as both must ensure that variables are appropriately initialised and memory is released when it can be done. Still, this is no worse than in most other languages.

One situation remains whre memory leaks can not be avoided: if the function result is used directly in a write statement like:

```
write(*,*) ic .concat. jc
```

The practical advantages of the described method are that it requires no extension to the standard and that it is completely safe to be used in a multiprocessing environment.

This note has not discussed whether *elemental* functions are possible that use this technique. It would probably involve "fooling" the compiler via *pure* interfaces to the cleanup routine, as this modifies the intent(in) arguments.

Literature

Metcalf, M. and J. Reid (1999)
Fortran 90/95 explained
Oxford University Press, second edition, 1999

Schonfelder, J.L. (1994)
The Fortran 90 module "ISO_VARYING_STRING"
http://www.pcweb.liv.ac.uk/jls/is1539-2-99.htm

Schonfelder, J.L. (2000)
The Fortran 95 module "VARYING_PRECISION_ARITHMETIC"
http://www.fortran.com/fortran/free.html