University of Maryland College Park

Institute for Advanced Computer Studies TR–2003–90
Department of Computer Science TR–4522

Memory Leaks in Derived Types Revisited*

G. W. Stewart†

August 2003

ABSTRACT

In a note in the *Fortran Forum*, Markus describes a technique for avoiding memory leaks with derived types. In this note, we show by a simple example that this technique does not work when the object in question is a parameter in nested subprogram invocations. A fix is proposed and illustrated with code from MATRAN, a Fortran 95 package for performing matrix manipulations.

*This report is available by anonymous ftp from `thales.cs.umd.edu` in the directory `pub/reports` or on the web at `http://www.cs.umd.edu/~stewart/`.

†Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742 (`stewart@cs.umd.edu`). This work was supported in part by the National Science Foundation under grant CCR0204084.

Memory Leaks in Derived Types Revisited

G. W. Stewart

ABSTRACT

In a note in the *Fortran Forum*, Markus describes a technique for avoiding memory leaks with derived types. In this note, we show by a simple example that this technique does not work when the object in question is a parameter in nested subprogram invocations. A fix is proposed and illustrated with code from MATRAN, a Fortran 95 package for performing matrix manipulations.

In a note in the *Fortran Forum*, Markus (2003) describes a technique for avoiding memory leaks with derived types. Such leaks may occur when a temporary type is generated in the course of evaluating an expression. For example, suppose we have a derived type `Rmat` implementing a matrix stored in a real rectangular pointer array and we have overloded `+` and `=` so that

```
C = A + B
```

computes the matrix sum of `A` and `B` and assigns it to `C`. The function implementing `+` will create a temporary `Rmat` to hold the sum pending assignment. Fortran will remove this temporary after the assignment, but it will not deallocate the storage for the matrix. Hence the memory leak.

Markus's solution to this problem is to augment the defined type in question with a logical component:

```
logical :: tmp = .false.
```

The function implementing `+` sets the `tmp` component of the result to `.true.`. The subroutine implementing the assignment tests the component and if it is `.true.` deallocates the storage for the matrix. (Markus also points out that this dodge will not be necessary in Fortran 2000, where defined types can have allocatable components.)

Unfortunately, this simple solution does not work when nested subprogram invocations are involved. For example, consider what happens when Markus's procedure is used in the following code fragment.

```
type Rmat :: A, B, C
call foo(A + B, C)
...
```

```
subroutine foo(X, Y)
type Rmat intent(in) :: X
type Rmat intent(inout) :: Y
Y = X
Y = X + Y
...
```

According to Markus's procedure, in the subroutine `foo` the parameter `X` is flagged as temporary, since the actual parameter is the result of evaluating an expression. Hence, the pointer array for `X` will be deallocated by the subroutine implementing the assignment `Y = X`, and `foo` will fail trying to compute

```
Y = X + Y
```

Thus any solution to the problem of temporaries must take into account nested invocations of subroutines and functions.

In this note we present the solution used in MATRAN (pronounced MAYtran), a Fortran 95 wrapper that implements matrix operations and decompositions using the BLAS and LAPACK. (Stewart, 2003. A preliminary version of MATRAN can be obtained via the author's home page `www.cs.umd.edu/~stewart`.) The idea is to maintain a counter that tracks the depth of subprogram invocations.

Here is the type `Rmat` from MATRAN.

```
type Rmat
   real(wp), pointer &              ! The matrix array
           :: a(:,:) => null()     !
   integer :: nrow = 0             ! Number of rows in the matrix
   integer :: ncol = 0             ! Number of columns in the matrix
   integer :: narow = 0            ! Number of rows in the array
   integer :: nacol = 0            ! Number of columns in the array
   character(2) &                  ! Type of matrix
           :: tag = 'GE'           !
   logical :: adjustable =.true.   ! Adjustable array
   integer, pointer &              ! Intermediate value
           :: temporary => null()
end type Rmat
```

The component we are interested in is the integer pointer `temporary`. It is initialized to `null()`, which indicates that the `Rmat` is question is not a temporary. We will explain in a moment why we use an integer pointer rather than an integer.

The status of temporary can be set using the following subroutine.

```
subroutine SetTemp(A)
   type(Rmat), intent(inout) :: A
   if(.not.associated(A%temporary)) &
        allocate(A%temporary)
   A%temporary = 1
end subroutine SetTemp
```

When a subroutine or function is invoked, one executes the following subroutine for each `Rmat` that is possibly a temporary

```
subroutine GuardTemp(A)
   type(Rmat) :: A
   integer, pointer :: t
   if(associated(A%temporary)) then
      t=>A%temporary
      t = t + 1
   end if
end subroutine GuardTemp
```

This subroutine simply increases the depth count by one. Just before returning one calls the following subroutine, once for each `Rmat` for which `GuardTemp` was called.

```
subroutine CleanTemp(A)
   type(Rmat) :: A
   integer, pointer :: t
   real(wp), pointer :: s(:,:)
   if(associated(A%temporary)) then
      t=>A%temporary
      if(t > 1)  t = t - 1
      if (t == 1) then
         s=>A%a
         deallocate(s)
         deallocate(t)
      end if
   end if
end subroutine CleanTemp
```

If `A` is temporary, the depth count is decreased by one (if it is not already one). If, after decrementation, the depth count is one, the pointers in `A` are deallocated. Note that if there is only one level of nesting, the call to `GuardTemp` can be skipped, though it is not recommended practice.

The reason for using an integer pointer for the depth count is that in subprograms implementing operations and assignments, some of the parameters must be specified as having `intent(in)`. As it turns out, some compilers are quite assiduous in tracking down violations of intent. However, while it is not permitted to change a pointer component of a parameter with `intent(in)`, it is permissible to change the object that it points to.

To illustrate the use of these routines, here is the subroutine that implements matrix assignments in MATRAN.

```
subroutine RmEqualsRm(A, B)
   type(Rmat), intent(inout) :: A
   type(Rmat), intent(in) :: B
   call GuardTemp(B)
   call ReshapeAry(A, B%nrow, B%ncol)
   A%a(1:A%nrow, 1:A%ncol) = B%a(1:B%nrow,1:B%ncol)
   A%tag = B%tag
   call CleanTemp(B)
end subroutine RmEqualsRm
```

The subroutine is straightforward. After the call to `GuardTemp`, the call to `ReshapeAry` makes sure that `A` has enough storage allocated to receive `B`. Then `B`'s array is copied to `A`'s, and `A` inherits `B`'s `tag` component, which tells whether the matrix is general, triangular, symmetric, etc. `CleanTemp` then deallocates `B`, if necessary.

This scheme for avoiding memory leaks has been tested on a number of different compilers and appears to work satisfactorily.

I would like to thank John Reid for his useful correspondence on this topic. Part of this work was performed as faculty appointee at the Mathematical and Computational Sciences Division of the National Institute for Standards and Technology.

References

A. Markus. Avoiding memory leaks with derived types, Fortran Forum 22:2 (2003) 1–6.

G. W. Stewart. MATRAN: A Fortran 95 Matrix Wrapper, University of Maryland, Department of Computer Science Technical Report 4522 (2003).