Object-oriented approach in Fortran 95: avoiding memory leaks with derived data types

Anna Kelbert

June 4, 2009

Abstract

This document describes our approach to the typical memory leaks and other problems commonly encountered in Fortran 95 when dealing with derived data types that contain allocatable/pointer components. When the Technical Report ISO/IEC 15581 (the "allocatable array extension" to F95) is adopted, these solutions will hopefully become obsolete. Currently, most state of the art compilers (g95, ifort, pgf95) do not support these extensions, making a temporary solution such as that described here crucial. We strongly recommend that these practices are carefully followed for all future development of this code, until they are proven to be unnecessary. In the following, when we refer to a data type, we mean a data type with a pointer component. Creating such a data type involves a memory allocation procedure.

Rule 1

Interface assignment (a = b) has to be overloaded.

Reason: When not overloaded, the pointers inside a point to the same memory location as those in b. Thus, vectors are not copied, and deallocating b later in the code will create an unusable a structure. Very dangerous.

Rule 2

Functions that output a data type have to

1. **allocate the output, and**
2. **mark the output as temporary.**

Use subroutines over functions where ever efficiency is an issue.

Reason: On input to the function, the output data type is completely undefined. Thus, checking it's `allocated`/`associated` status is meaningless. It therefore has to be allocated inside the function. Once allocated, the result of the function is stored in a temporary variable, call it RHS. Since the equals (=) sign is overloaded, it is then passed as input to the user-defined copy subroutine that overloads the assignment. The LHS is computed, but the RHS is still stored in memory. Moreover, once we exit the copy subroutine, it can no longer be accessed or deallocated. Most compilers (certainly g95 and ifort at the time of writing) do not deallocate these temporary variables. This creates very real memory leaks, i.e. every time such a function is called, a copy of the data type is left behind in system memory. This can very easily result in unusable code.

Example: `a = add(b,c)` OR `a = b + c` OR `a = zero(b)`

Solution: One possibility is staying away from functions (and hence, from overloaded operators!). We adopt a different approach. Every data type contains a logical variable `temporary`, `.false.` by default. It is only `.true.` for a function output. The `copy` subroutine deallocates it's input if and only if it is "temporary". This can be done be tricking the compiler: the `deall` subroutine does not have an intent for it's input argument (note that `intent(inout)` would make the compiler bomb due to the `intent(in)` specification of the variable in the `copy` subroutine). This works and gets rid of the memory leaks; it also allows us to safely use functions. When this trick becomes unnecessary as the compilers develop, it would be easy to get rid of it in the code.

Restrictions:
With this approach, usage of the functions prohibits the following.

Example: `write(0,*) add(b,c)`

Reason: The temporary variable is created and not deallocated, causing a memory leak.

Example: `a = b + c*d`

Reason: The temporary variable c*d is created and not deallocated, causing a memory leak. So, no concatenation of operators! This would be allowed if the above approach is extended to delete temporary input variables in all functions. However, that requires a bit of extra coding. Besides, for efficiency, this usage is discouraged.

Example: usage of functions in the lower level code

Reason: All the temporary variables that are created in the process may affect the efficiency of the code. Thus, good programming disallows usage of any functions in a subroutine that is going to be called many times in the code. Only use functions and/or operators in the upper level code, where readability is more important than efficiency. Use subroutines (such as `linComb`) otherwise.

Rule 3

No data type intent(out) dummy arguments in subroutines.

Reason: For a mysterious reason that I do not understand, the compilers also create a temporary variables for such a data type (just like for function outputs), that cannot be deallocated after exiting the subroutine, creating a steady memory leak.

Solution: Use `intent(inout)` instead.

Rule 4

Basic subroutines require that intent(inout) dummy arguments are allocated and of the correct size.

Reason: Variables are passed by reference in Fortran. Thus, in a call such as `linComb(a1,v1,a2,v2,v1)` that overwrites its own input variable, any modification (or deallocation) of the output `v1` inside the subroutine also modifies (or deallocates) the input `v1`. Mysterious errors result. Not a memory leak issue, but may be worse. Impossible for the compiler to track.

Solution: Such subroutines may not reuse their output. So, no allocation/deallocation or recursion in subroutines that may potentially overwrite their inputs. This is our solution for this code.

An alternative solution: Require the output to NOT exist at the entrance to the subroutine. Drawbacks of this approach include inefficiency, since no outputs may overwrite the inputs. Hence, more temporary variables will be floating around, resulting in multiple additional allocation/deallocation calls. However, programming is safer.

Rule 5

Always deallocate temporary data types before exiting a subroutine.

Reason: To avoid multiple memory leaks. A data type with pointer components is not automatically deallocated on exit from a subroutine.

Rule 6

Only allocate a pointer if it's not already associated.

Reason: Statements such as `allocate(vector,STAT=istat)` do not save you from allocating the memory again. If the vector was already allocated, this statement results in a memory leak. The only fail proof check is `associated(vector)`.