ISO/IEC JTC1/SC22/WG5 **N1187**

03 June 1996

Requirement for Cleaner & Producer in Fortran 90

This document describes the need of a kind of a destructor in Fortran 90. It also suggests a way how a - what we call - *cleaner* and *producer* in Fortran could be implemented, as well as the functionalities of a *cleaner* and a *producer*. The cleaner is described in I., the basic functionality of the producer is described in II., and an example which discusses and shows the need of a cleaner is described in III. as an appendix.

I. Cleaner in Fortran 90

- A cleaner is called for each variable of user defined type and for each pointer variable and array pointer after the end of its scope. The functionality of a cleaner is to destroy data when it's not needed anymore.
- The user defined cleaner is implicitly called to avoid memory leaks produced by local variables and function results containing dynamic components and pointers in Fortran 90
- A cleaner is necessary to beware a user to implement complicated deallocation mechanisms.

A cleaner should be called once for every object generated during program execution. An object can be a local variable, a global variable, or a function result.

- **Problem I.a. :** Function results containing pointers somehow are allocated within the function and require large management mechanisms to be able to deallocate them if they are not needed anymore. These mechanisms have to be implemented by the user.

  Allocation and deallocation of data has always got to be understood as a pair. Sometimes deallocation of data is not visible, normally this is done at the end of the program, on stacks etc. As long as no pointers are involved in function results there are no problems.

  As soon as function results which have pointer components (scalar pointer, array pointer, or derived types containing pointer components) are allocated, they have

to be deallocated somewhere.

→ *Each allocation of an object must have a corresponding deallocation of the object.*

Example I.1.:

```
c = fu(a,b)        ! ordinary function call
```

The allocated function result of `fu` can be deallocated within the assignment operation, if it can be checked somehow, wether the argument on the right hand side is a function result or not. This of course is possible if the function result is of derived type by simply adding a flag as a component. If the function result is a scalar or array pointer, the user has no possibilities to distinguish between function results and arrays defined e.g. by the user and which are not to be destroyed.

Therefore one could think of data objects with target attribute, having an additional counter counting the number of pointers (links) pointing to it, or better, the number of objects using it (not to forget the target itself).

also ok for

```
b = fu(a,b)    !* old 'b' could simply be reallocated if
                    !* necessary, and overwritten
```

not ok for

```
b => fu(a,b)   !* never get rid of old b
```

Work around :

```
!* possibility to work round problem above ...
!* ... but definitly not very elegant
   do
     deallocate(c)
     c => fu(a,b)
     b = c
   enddo
```

Example I.2.:

```
c = fu2(fu(a,b),r) ! nested function calls with function result as
                   ! argument
call sub(fu(a,b),u)! same problem with function result as argument
                   ! in subroutine
```

If the function result allocated by `fu` would be recognized to be a function result, it could be destroyed at the end of the procedure using this function result, which is `fu2` or `sub` in the examples above. Therefore the user has to do a lot of management. It at least is possible, if derived types are used as function results, because a flag telling wether the variable is a function result or not could be implemented. But if the function result of `fu` is an array pointer, there's no possibility to deallocate the function result of `fu` when it's not needed anymore.

Having the destroying mechanisms of arguments in procedures is not always possible. If the function `fu` is taken out of a library, there is no possibility for the user to define the result variable to be function results. For `sub` taken out of a library there is no problem, because the arguments are directly passed.

Example I.3.:

```
print*, fu(a,b)    ! function result to be destroyed after print
write*, fu(a,b)    ! function result to be destroyed after write
```

This is just a special case of Example I.1, but even with a complicated derived type it is never possible to access the function result of `fu(a,b)` to destroy any pointer components.

One could remark at this point, that an I/O list like this, if the function result has pointer components somehow, is not valid anyway, and that one would provide a procedure which does the print job for the function result. This then could look like the following

```
call print_object(fu(a,b))
```

The implementation of such a procedure would have to contain the print mechanism for the components of the function result of `fu` plus a mechanism to destroy the function results components after it has been printed like described in Example I.2, which is not what we expect from a printing routine.

It could also be possible that in future stardards definitions the `print` function can be overloaded (No 17).

Example I.4.:

```
c => fu(a,b)    ! function result related to left hand side
                !  via pointer assignment.
```

A pointer assignment statement is recognized by the compiler and therefore no cleaner is called for the result variable of `fu(a,b)`.

One more problem is shown by this example. It is necessary for the user to be able to overload the pointer assignment statement for the management. In the case above, the number of links to the function result has to be increased with the pointer assignment. It is even worse if the statement is simpler like in the following case

```
c => a
```

Example and Discussion for memory leaks produced by function results : see III. Appendix : *Discussion of Deallocation Problem with Function Results* in this document.

- **Problem I.b.:**

A user should not be concerned with administration of data, because deallocation mechanisms can be complex, even if the scope of the variables is one (sub-)program.

Example I.5 :

```
subroutine cgs(A,x,y)
  type(matrix),pointer :: A
  type(vector),pointer :: x,y
  !*
  type(vector),pointer :: d,u,z
          ...

  !* clean up local variables
  call deallocate_vector(d)  !*
  call deallocate_vector(u)  !* error prone, should not have
  call deallocate_vector(z)  !*  to be done by the user !!
  deallocate(d,u,z)          !*
end subroutine cgs
```

- A cleaner is similar to a destructor in C++
- A cleaner is a user written program which is accessed via a special interface like e.g.

```
interface CLEANER
  module procedure cleaner_mytype
end interface
```

- A cleaner should be defined for each dynamic variable and array type and for each user defined type.
- A cleaner should contain counting meachanisms for links to targets and possibilities to destroy targets in a clean way if no links to the target exist anymore.

  A mechanism to count links on each target is required, but should better be implemented by the user because of performance problems. This is necessary for writing Fortran 90 cleaners or destructors, because a target must not be destroyed with a pointer if there still exist links to other pointers.
- A cleaner is data type specific
- A cleaner is called implicitly $\rightarrow$ The actual function or subroutine does not have to be changed if cleaners are activated.
- The moment to call the cleaner for a variable depends on the scope of this variable

  A Cleaner is called at the end of a scope

  - for local variables at the end of the function or subroutine
  - for function results at the end of the whole statement the function is involved

    Example :

    ```
    x = f(g(a,b),h(b,c))
    ```

    In the example the function result for `h(b,c)` is destroyed after the function result of `f` is assigned to `x`.

    Example :

```
call sub(fu1(a,b),fu2(c,d))
```

The function results for `fu1` and `fu2` are destroyed right after returning from the subroutine `sub`.

This enables further optimization possibilities for the compiler if a function is defined to be pure.

Example :

```
call sub(fu1(a,b),fu2(fu1(a,b),c)) !* mark a
```

In the statement (mark a), the function result of `fu1(a,b)` can be reused if `fu1` is a pure function, which means that it has not got to be computed twice.

- for function results in print, write statements after the print, write in which the function is involved is finished. (special case from the case above)
- for global variables at the end of the program

II. Producer in Fortran 90

- A Fortran 90 producer is used to optimize memory usage and to suppress time consuming allocation-deallocation mechanisms. It is not a constructor in the C++ sense. Initialization of static derived type components can already be done in type declarations but initialization of dynamic arrays should be done by the producer. A producer is used to allocate memory if objects have to be generated. It has to be written by the user and also has to be explicitly called by the user.
- Using a producer optimizes memory usage by avoiding new allocation if possible. If a producer is used, the compiler decides wether new storage for a function result has to be allocated or not.
- It is a compilers work to optimize the memory management

There is no exact specification for the producer yet, but there is a main functionality and sense of the producer which will be specified below. We still discuss on how the producer could be realized.

Procedure oriented (structured) programming in FORTRAN 77 was mainly based on subroutines. But improved programming paradigms like modular, object oriented, or component oriented programming, which are supported by Fortran 90 are based on functions instead of subroutines. Therefore the meaning of functions has grown.

A mechanism for the compiler to see in which context a function turns up is required. There are three (and only three) possibilities.

1. The function is an argument of another function or subroutine.
   Examples : `c=f(g(a),b), print, write, call sub(f(a,b)), ...`
2. The function is the right hand side of an assignment to a variable of the same type.
   Example : `x = f(y,z)`
3. The function is the right hand side of a pointer assignment statement.
   Example : `x => f(y,z)`

To distinguish between these three cases won't cause a runtime problem, because the compiler can do the job. If the cases are recognized, the producer can be activated for the second case (2) and take the left hand side of the assignment (`x`) as function result variable for `f(y,z)` to avoid the allocation of a temporary function result and a copy of the temporary to `x`.
This improves efficiency very much, especially if complicated data structures and/or large arrays are used.

Task of a Producer in Fortran 90

- The producer is activated if the function and the left handside of an assignment have the same data type. The function on the right hand side overturns the data structure on the left handside of the assignment as its result variable. Therefore the left hand side of the assignment is reallocated if necessary.
  The assignment step is done before and while the function result is produced.
  → **preassignment**
- If the producer is not activated, the variable for the result value is allocated in the usual way, the assignment therefore normally is overloaded. The function result is computed and stored locally before the assignment to the variable on the left handside of the assignment.
  → **postassignment**
- A Fortran 90 producer is data type specific, but function related.
- Examples for usage of *preassignment*, *postassignment*, and *pointer assignment* :
  1. *preassignment* in function `f` for

     ```
     c = f(a,b)      !* case (B)
     c = f(a,g(b,c)) !* case (B)
     ```

  2. *postassignment* if function in

     ```
     function call   !* case (A)
     subroutine call !* case (A)
     print, write    !* case (A)
     c = f(a,c)      !* special case (B)
     ```

  3. *pointer assignment* (Allocation of extra function result)

     ```
     c => f(a,c)
     ```

- A Fortran 90 producer uses allocated storage if possible and allocates storage if necessary. It can also be used for initialization, but the main functionality is memory allocation.
- The producer of a certain data type allocates its components in the right size. It is an allocation routine written by the user which has to be explicitly called by the user.
- List of assignment priorities :

Priority	Assignment Type
1	Pointer Assignment (F90)
2	Preassignment (new)
3	Postassignment (overloaded assignment)
4	Implicit Assignment (F77)

Using a producer (*preassignment*), one allocation (and one copy) less has to be performed (temporary result variable). Also the deallocation of the result variable is avoided.

III. Appendix : Discussion of Deallocation Problem with Function Results.

Problem : Deallocation of pointer valued function results

The following example shows a typical case for the use of pointer valued function results. The program shows the multiplication of two periodic Toeplitz matrices.

```
    module routines

    contains

    !***********************************************

      function toeplitz_mult(xx,yy) result(zz)

        real,pointer,dimension(:)     :: xx,yy,zz  !** fmark 1
        integer                       :: xx_dim

        !* next statement is necessary without the possibility
        !* to deallocate the array within this subroutine, because
        !* it is the function result

        if(size(xx) /= size(yy)) stop ' Error in toeplitz_mult'

        xx_dim = size(xx)
        allocate(zz(xx_dim))                      !** fmark 2

        i=1
        do k=1,xx_dim
        ind3=mod(k-i+xx_dim,xx_dim)+1
        zz(ind3)=0.                               !** fmark 3
         do j=1,xx_dim
          ind1=mod(j-i+xx_dim,xx_dim)+1
          ind2=mod(k-j+xx_dim,xx_dim)+1
          zz(ind3)=zz(ind3)+xx(ind1)*yy(ind2)     !** fmark 4
         enddo
        enddo

        !* no deallocation of zz

      end function toeplitz_mult

    !***********************************************

    end module routines

!*********************************************************************
```

```
program iterate

use routines

integer,parameter          :: nmax = 100
integer,parameter          :: itmax=24
integer                    :: iter
real,pointer,dimension(:)  :: aa,bb,cc      !** mark 1

allocate(aa(nmax),bb(nmax))                 !** mark 2

... !* (Initialization of aa)

bb=0.; bb(1)=1.                             !** mark 3

do iter=1,itmax
  cc => toeplitz_mult(bb,aa)                !** mark 4
  bb = cc                                   !** mark 5
  deallocate(cc)                            !** mark 6
                  !* without this statement, cc always exists
                  !* but is not accessible anymore.
enddo

call output(cc)

end

!*************************************************
```

- In numerical programs it is very important to have the possibility to create large arrays dynamically. Most times, these arrays are part of complex data structures.

  In the above case, the actual and dummy arguments for the function `toeplitz_mult` as well as the function result variable of function `toeplitz_mult` are array pointers. The arguments have been allocated before they were passed to the function. The function result array has to be allocated via an allocate statement (fmark 2) within the function.
  This array cannot be deallocated in the same program unit as it has been allocated because it is needed as function result for further operations.
  In the loop in the main program, the array pointer cc points to the function result which is also declared to be an array pointer. Therefore no values have to be copied, the two pointers point to the same target (mark 4).
  The values are copied in the statement in (mark 5).
  In the next loop iteration, the array pointer cc points to another location, where the new function result of `toeplitz_mult` is located. Without the deallocate statement in the main programs loop (mark 6), the space of the function result variable would not be given free, but also could not be accessed anymore.

  If nmax and imax were very large numbers this reults in large memory leaks.

- If the statement in mark 4 was changed into

```
cc = toeplitz_mult(bb,aa)                !** mark 4.2
```

either an allocate statement for `cc` in the main program to be able to do proper array syntax, or an assignment subroutine is needed. With an allocate statement for `cc` in the main program before the loop, the problem with the allocated space for the function result variable is just the same, because the assignment is a copy of array values, and not just setting a pointer. Therefore the function result can not be accessed after the assignment (mark 4.2) to be deallocated.
With an assignment subroutine, the existence of cc can be checked by the instrinsic function associated, and after the assignment of the values, the right hand side can be deallocated within the assignment subroutine. But if the right hand side consists of at least 2 function calls to the function `toeplitz_mult` like in mark 4.2b

```
cc = toeplitz_mult(toeplitz_mult(bb,aa),aa)  !** mark 4.2b
```

there is at least one function result for which space has been allocated which cannot be accessed anymore.
Of course it is possible to check the arguments within the function `toeplitz_mult` via special flags, if they have also been function results before, and deallocate them in this case at the end of the function. But therefore a more complex data structure is needed, containing at least a flag apart from the values array, telling if the object itself is a function result or not, and possibly a counter, which tells the number of pointers pointing to the same target.
On the whole, this variation would result in a large management expense causing slower program executions. This is definitly not the intention of object oriented programming.

- In numerical programs dynamic components as parts of complex data structures are very important. Arrays like in the example above normally are only parts of complex data structures. Therefore the variables aa,bb, and cc could also be of derived type, containing pointer components. A very simple derived type is the one below.

```
type toeplitz_matrix
  integer           :: dimension
  real,dimension(:),pointer  :: values
end type toeplitz_matrix

type(toeplitz_matrix)  :: aa,bb,cc
```

The dummy arguments and result variable types in the function `toeplitz_mult` would be changed into `type(toeplitz_matrix)`,too. Derived types for numerical data are mostly much more complex.

For the assignment of data structures with pointer components we would need an assignment subroutine. Handling data structures lead to exactly the same problems as described above. This is not for the ordinary components, but for the pointer components.

If communication between program units was done via copying to and from stack as it is done with simple variables or arrays without pointer attribute, the problem above would not exist in that way. But the only way to have dynamic components within data structures is to have pointer components. And pointer assignments are much cheaper than copies and therefore very attractive.

The same problem of course turns up for overloaded operators which are more natural to be used in a sequence of operations like

```
cc = (bb * aa) * aa
```