# Guide to create or adapt column dominate physical parameterization in BRAMS (Brazilian RAMS)

Álvaro L. Fazenda; Jairo Panetta
INPE/CPTEC
Saulo Freitas; Pedro L. Silva Dias
USP/IAG

RAMS memory allocation mechanism has dramatically improved from RAMS version 4.X to RAMS version 5.0. A software modernization process replaced the old and faithful "A" array by a series of data types that implement some level of object orientation, allowing state-of-the-art dynamic memory allocation and deallocation.

This new software structure poses a problem for RAMS contributors, which face an unfamiliar set of commands. This document tries to alleviate the problem by describing the new software structure and applying it on a procedure to insert physical modules in RAMS 5.X. The insertion procedure is limited to modules that operate over a column of the atmosphere at a time (that is, no horizontal interaction is allowed). A case study, the insertion of Shallow Cumulus Parameterization, is presented. This physical module was moved, into RAMS 5.0, from its original integration at RAMS 4.3, where the "A" array was used.

We describe the insertion procedure at BRAMS 1.0, which is a joint project of ATMET, IME/USP, IAG/USP and CPTEC, aimed to produce a version of RAMS tailored to the tropics. This first version (BRAMS 1.0) is just RAMS 5.0 with the inclusion of modeling of physical phenomena such as Shallow Cumulus, improvements in software quality (leading to binary reproducibility and higher portability), a higher resolution vegetation data file, etc. The insertion procedure applies, without modification, to RAMS 5.0.

To include new procedures or parameterizations in BRAMS, a decision procedure should be followed:

1. If the new procedure uses only existent fields and local (to the procedure) scalar storage, it suffices to insert the new subroutine call at the appropriated point at routine *timestep;*

2. On the other hand, if the new procedure requires timestep-independent local storage (as scratch space, for example), than the local storage can be acquired as Fortran 90 *automatic* (or *allocatable*) arrays or re-using the scratch arrays structure of BRAMS;
3. But if the new procedure requires history-carrying storage (such as new meteorological variables), we strongly recommend the use of a structure similar to the one presented at BRAMS. This structure and organization are depicted in the sections bellow.

This document is organized in nine sections. Section 1 introduces the concept of a data type in BRAMS and lists the default data types. Section 2 shows how the default data types manage storage for a single grid. Section 3 extends this concept to multiple grids. Section 4 shows how VTABLE information is included for these data types. Section 5 presents the standard structure of a data type definition file, summarizing the material of all previous sections. Section 6 shows the sequence of calls to allocate and return default data.

The material of sections 1-6 is used, in section 7, to insert the Shallow Cumulus data type in BRAMS. Section 8 describes how to insert the call to Shallow Cumulus at *timestep*, and section 9 state some conclusions and remarks.

## *1- Data Types in BRAMS 1.0*

BRAMS 1.0 have a new way of allocating memory to fields. Previously, fields were allocated as sections of the "A" array. The dedicated section was accessible by pointing to its first position. Currently, fields are components of a DATA TYPE, dynamically allocated at initialization.

There are 14 DATA TYPES in BRAMS 1.0 (12 in RAMS 5.0). Each data type contains all fields required to represent a certain theme, for a certain grid. The structures are all located at the ./BRAMS/src/rams/5.0/modules directory, all coded as Fortran-90 MODULES. They are:

- Basic data (DATA TYPE *basic_vars* at file *mem_basic.f90*)
    - 3D arrays: up, uc, vp, vc, wp, wc, pp, pc, rv, theta, thp, rtp, pi0, th0, dn0, dn0u, dn0v
    - 2D arrays: fcoru, fcorv, cputime

- Standard cumulus parameterization data (DATA TYPE *cuparm_vars* at file *mem_cuparm.f90*)
    - 3D arrays: thsrc, rtsrc
    - 2D arrays: aconpr,conprr
- Vegetation and Soil ("Leaf") data (DATA TYPE *leaf_vars* at file *mem_leaf.f90*)
    - 4D arrays: soil_water, soil_energy, soil_text; sfcwater_mass, sfcwater_energy, sfcwater_depth
    - 3D arrays: ustar,tstar,rstar, veg_fracarea,veg_ lai,veg_rough, veg_height, veg_albedo,veg_tai, patch_area, patch_rough, patch_wetind, leaf_class, soil_rough, sfcwater_nlev, stom_resist, ground_rsat,ground_rvap, veg_water, veg_temp, can_rvap, can_temp, veg_ndvip, veg_ndvic, veg_ndvif
    - 2D arrays: snow_mass, snow_depth, seatp,seatf
- Microphysics data (DATA TYPE *micro_vars* at file *mem_micro.f90*)
    - 3D arrays: rcp, rrp, rpp, rsp, rap, rgp, rhp, ccp, crp, cpp, csp, cap, cgp, chp, cccnp, cifnp, q2, q6, q7
    - 2D arrays: accpr, accpp, accps, accpa, accpg, accph, pcprr, pcprp, pcprs, pcpra, pcprg, pcprh, pcpg, qpcpg, dpcpg
- Radiation data (DATA TYPE *radiate_vars* at file *mem_radiate.f90*)
    - 3D array: fthrd
    - 2D arrays: rshort, rlong, rlongup, albedt, cosz
- Turbulent phenomena data (DATA TYPE *turb_vars* at file *mem_turb.f90*)
    - 3D arrays: tkep, epsp, hkm, vkm, vkh
    - 2D arrays: sflux_u, sflux_v, sflux_w, sflux_t, sflux_r
- General variable data (DATA TYPE *varinit_vars* at file *mem_varinit.f90*)
    - 3D arrays: varup, varvp, varpp, vartp, varrp, varuf, varvf, varpf, vartf, varrf, varwts
- Grid data (DATA TYPE *grid_vars* at file *mem_grid.f90*)
    - 3D arrays: aru, arv, arw, volu, volv, volw, volt
    - 2D arrays: topt, topu, topv, topm, topma, topta, rtgt, rtgu, rtgv, rtgm, f13t, f13u, f13v, f13m, f23t, f23u, f23v, f23m, dxt, dxu, dxv, dxm, dyt, dyu, dyv, dym, fmapt, fmapu, fmapv, mapm, fmapti, fmapui, fmapvi, fmapmi, glat, glon, topzo;  lpu, lpv, lpw

- Scalar variables and tendencies data (DATA TYPE *scalar_vars* at file *mem_scalar.f90*)
    - 3D array: sclp
    - 2D array: drydep
    - 1D array: sclt
    - Scalars: n1, n2, n3, naddsc, nsc
- Tendencies data (DATA TYPE *tend_vars* at file *mem_tend.f90*)
    - 1D array: ut, vt, wt, pt, tht, rtt, rct, rrt, rpt, rst, rat, rgt, rht, cct, crt, cpt, cst, cat, cgt, cht, cccnt, cifnt, tket, epst
- Scratch data (DATA TYPE *scratch_vars* at file *mem_scratch.f90*)
    - 1D arrays: scr1, scr2, vt3da, vt3db, vt3dc, vt3dd, vt3de, vt3df, vt3dg, vt3dh, vt3di, vt3dj, vt3dk, vt3dl, vt3dm, vt3dn, vt3do, vt3dp, vt2da, vt2db, vt2dc, vt2dd, vt2de, vt2df (to save data related to 3D arrays); vt2da, vt2db, vt2dc, vt2dd, vt2de, vt2df (to save data related to 2D arrays)
- Nested boundary interpolation data (DATA TYPE *nest_bounds* at file *mem_nestb.f90*)
    - 4D arrays: bsx, bsy, bsz
    - 3D arrays: bux, buy, buz, bvx, bvy, bvz, bwx, bwy, bwz, bpx, bpy, bpz

Remaining data continues to be allocated statically and communicated by COMMONs. Take, for example, file /BRAMS/src/rams/5.0/include/rcommons.h. It contains declaration of arrays such as *vctr1*, *tnudcent* and *nnqparm*, that are accessible to RAMS modules through *INCLUDE 'rcommons.h'*. This data arrangement will be used for a while. However, the DATA TYPE method is more appropriated for history carying fields (that is, variables that should keep their values among type stetps).

*2 – Dynamic Allocation of a Single Grid*

Take any of the above DATA TYPES. For example, *basic_vars.* The data type declaration at file /BRAMS/src/rams/5.0/modules/mem_basic.f90 is:

```
TYPE basic_vars

   ! Variables to be dimensioned by (nzp,nxp,nyp)
   REAL, POINTER, DIMENSION(:,:,:) :: &
        up,uc,vp,vc,wp,wc,pp,pc  &
        ,rv,theta,thp,rtp &
        ,pi0,th0,dn0,dn0u,dn0v

   ! Variables to be dimensioned by (nxp,nyp)
   REAL, POINTER, DIMENSION(:,:) :: &
        fcoru,fcorv,cputime

END TYPE basic_vars
```

Observe that each variable of type *basic_vars* has 17 three-dimensional fields and 3 two dimensional fields. For example, the declaration

```
TYPE (basic_vars) :: example
```

states that variable *example* is of type *basic_vars.* Consequently, it contains all 20 fields. To access one field (say, *fcoru*) it suffices to write *example%fcoru.* Element (i,j) of this array is accessed by *example%fcoru(i,j).*

Declaring a variable of type *basic_vars* do not reserve memory for its allocatable components (the fields). An explicit memory allocation command *(ALLOCATE)* should be used for each component.

Each DATA TYPE in BRAMS has an initialization routine that allocates all allocatable components of a variable of that type. Taking again *basic_vars* as an example, the corresponding allocation procedure is

```
SUBROUTINE alloc_basic(basic,n1,n2,n3)

  IMPLICIT NONE
  TYPE (basic_vars) :: basic
  INTEGER, INTENT(in) :: n1,n2,n3

  ALLOCATE (basic%up(n1,n2,n3))
  ALLOCATE (basic%uc(n1,n2,n3))
  ALLOCATE (basic%vp(n1,n2,n3))
  ALLOCATE (basic%vc(n1,n2,n3))
  ...
  ALLOCATE (basic%fcoru(n2,n3))
  ALLOCATE (basic%fcorv(n2,n3))
  ALLOCATE (basic%cputime(n2,n3))
```

```
    RETURN
  END SUBROUTINE alloc_basic
```

Assume that variable *example* represents a grid of size (10, 50, 8). Then, the invocation

```
  CALL alloc_basic(example, 10, 50, 8)
```

allocates all 20 fields with the desired size.

*3 – Allocating Multiple Grids*

A scalar variable like *example* suffices to allocate a single grid. To allocate multiple grids, either one creates a new variable name for each grid (which is, at least, inconvenient) or one creates an array of elements, indexed by grid number.

Consequently, the declaration

```
  TYPE (basic_vars), ALLOCATABLE :: basic_g(:)
```

defines *basic_g* as an array of *basic_vars,* representing multiple grids. The array has to be allocated (by an *ALLOCATE* statement) and all fields of each array entry also have to be allocated (by invoking *alloc_basic(basic_g(i),.....)*).

The actual declaration of the array that represents all grids is (see file *mem_basic.f90*):

```
  TYPE (basic_vars), ALLOCATABLE :: basic_g(:), basicm_g(:)
```

While *basic_g* contains the actual fields, *basicm_g* contains temporal means, if required.

Besides the types and basic grid variables declarations, the file also contains procedures to dynamically allocate the components of each variable of the type (the *alloc_basic* routine) and procedures to deallocate and nullify all components:

```
  SUBROUTINE nullify_basic(basic)

    IMPLICIT NONE
```

```
  TYPE (basic_vars) :: basic

  IF (ASSOCIATED(basic%up   ))    NULLIFY (basic%up   )
  IF (ASSOCIATED(basic%uc   ))    NULLIFY (basic%uc   )
  IF (ASSOCIATED(basic%vp   ))    NULLIFY (basic%vp   )
  IF (ASSOCIATED(basic%vc   ))    NULLIFY (basic%vc   )
  ...
  IF (ASSOCIATED(basic%cputime)) NULLIFY (basic%cputime )

  RETURN
END SUBROUTINE nullify_basic

SUBROUTINE dealloc_basic(basic)

  IMPLICIT NONE
  TYPE (basic_vars) :: basic

  IF (ASSOCIATED(basic%up  ))    DEALLOCATE (basic%up)
  IF (ASSOCIATED(basic%uc   ))    DEALLOCATE (basic%uc)
  IF (ASSOCIATED(basic%vp   ))    DEALLOCATE (basic%vp)
  IF (ASSOCIATED(basic%vc   ))    DEALLOCATE (basic%vc)
  ...
  IF (ASSOCIATED(basic%cputime))DEALLOCATE(basic%cputime)

  RETURN
END SUBROUTINE dealloc_basic
```

*4 – Substituting VTABLES*

Another novelty of BRAMS (and RAMS 5.0) is the replacement of the VTABLE file by an array. The software structure is identical to the one depicted above: a data type for each entry of an array. A subroutine called *filltab_basic* fills the array, invoking *vtables2* to fill one entry of the array. These are all included in the *mem_basic* file; it contains, for example:

```
SUBROUTINE filltab_basic(basic,basicm,imean,n1,n2,n3,ng)

  USE var_tables

  IMPLICIT NONE
  TYPE (basic_vars) :: basic,basicm
  INTEGER, INTENT(in) :: imean,n1,n2,n3,ng
  INTEGER :: npts
  REAL, POINTER :: var,varm

  ! Fill pointers to arrays into variable tables
```

```
    npts=n1*n2*n3

    IF (ASSOCIATED(basic%up))  &
         CALL vtables2 (basic%up(1,1,1),basicm%up(1,1,1)  &
         ,ng, npts, imean,  &
         'UP :3:hist:anal:mpti:mpt3:mpt2')
    IF (ASSOCIATED(basic%vp))  &
         CALL vtables2 (basic%vp(1,1,1),basicm%vp(1,1,1)  &
         ,ng, npts, imean,  &
         'VP :3:hist:anal:mpti:mpt3:mpt2')
    IF (ASSOCIATED(basic%wp))  &
         CALL vtables2 (basic%wp(1,1,1),basicm%wp(1,1,1)  &
         ,ng, npts, imean,  &
         'WP :3:hist:anal:mpti:mpt3:mpt2')
    ...
    npts=n2*n3
    IF (ASSOCIATED(basic%fcoru)) &
         CALL vtables2(basic%fcoru(1,1),basicm%fcoru(1,1) &
         ,ng, npts, imean,  &
         'FCORU :2:mpti')
    IF (ASSOCIATED(basic%fcorv)) &
         CALL vtables2(basic%fcorv(1,1),basicm%fcorv(1,1) &
         ,ng, npts, imean,  &
         'FCORV :2:mpti')
    ...
    RETURN
  END SUBROUTINE filltab_basic
```

that fills all entries of the VTABLES array. The *vtables2* subroutine fills a single table entry with a string like:

"UP :3:hist:anal:mpti:mpt3:mpt2"

that has the same meaning than one entry of the old VTABLE file. Semantics of the sub-strings is:

```
# Tables:
#--------
# hist - write to history file
# anal - write to analysis file
# lite - write to analysis "lite" file
# Parallel tables:
#-----------------
#  mpti - initialization, full sub-domain master to node
#  mpt1 - long timestep, subdomain boundaries node to node
#  mpt2 - small timestep, subdomain boundaries node to node
#  mpt3 - full sub-domain node to master for output
# description
```

```
#  1     2 3+
#--------------------------------------------------
#  UP :  3:hist:anal:mpti:mpt3:mpt2
# 1. variable name
# 2. dimensionality ( 3-> 3d; 2-> 2d; s->soil)
# 3. + list of tables
```

The VTABLES mechanism provides an automatic input/output mechanism that can recover or save information in history or analyses files, as well as distinguished parallel communication.

Of course, the above *filltab_basic* routine fills the VTABLES array just for variables of type *basic_var*. That is the reason why it is contained at file *mem_basic*. Each file that contains a new data type has to contain a similar routine.

*5– Final DATA TYPE file structure*

All 12 data types are contained in files with the same strucure: a data type declaration followed by procedures to:

1. Allocate components of a variable of that data type
2. Deallocate (and nullify) components of the same variable
3. Fill the VTABLES entry for variables of the same type

How important is to know (and maintain) this file structure? Supose the insertion of a new physical parametrization module in BRAMS. If this new module requires history carying variables, then a new data type, allocation, deallocation, nullify and VTABLES procedures ought to be created, specifically for variables of the new data type.

In other words, it is central (to the health of BRAMS) to keep this structure. But that is not enough.

*6– Global Memory Allocation*

There is a single procedure in BRAMS that performs all memory allocation and initialization: that is *rams_mem_alloc* at file BRAMS/src/rams/5.0/model/alloc.f90. It allocates data for all grids, nullify components, call *alloc_basic* to allocate component arrays for each grid (with the appropriated dimensions) and fill the information tab (*filltab_basic*) for the just allocated arrays:

```
...
! Allocate Basic variables data type
PRINT*,'start basic alloc'
ALLOCATE(basic_g(ngrids),basicm_g(ngrids))
DO ng=1,ngrids
   CALL nullify_basic(basic_g(ng))
   CALL nullify_basic(basicm_g(ng))
   CALL alloc_basic(basic_g(ng), nmzp(ng), nmxp(ng),  &
                    nmyp(ng), ng)
   IF (imean == 1) THEN
      CALL alloc_basic(basicm_g(ng), nmzp(ng),        &
                       nmxp(ng), nmyp(ng), ng)
   ELSEIF (imean == 0) THEN
      CALL alloc_basic(basicm_g(ng),1,1,1,ng)
   ENDIF

   CALL filltab_basic(basic_g(ng),basicm_g(ng),imean  &
        ,nmzp(ng),nmxp(ng),nmyp(ng),ng)
ENDDO
...
```

But it does that not only for variables of type *basic_vars*; it does for all global variables of all types. Consequently, if a new type is to be created (maintaining the file structure just shown) then the above code has to be replicated for the variables of the just-created type. That is the case for the shallow cumulus parameterization.

*7 – Dynamic Allocation of new data for Shallow Cumulus parameterization*

Since the Shallow Cumulus parameterization requires history-carrying fields, a new MODULE called *mem_shcu* was created, similar to the previous one, to insert three new arrays:

```
MODULE mem_shcu
  TYPE shcu_vars

     ! Variables to be dimensioned by (nzp,nxp,nyp)
     REAL, POINTER, DIMENSION(:,:,:) :: &
          THSRCSH, RTSRCSH

     ! Variables to be dimensioned by (nxp,nyp)
     REAL, POINTER, DIMENSION(:,:) :: &
          SHMF
```

```
  END TYPE shcu_vars

  TYPE (shcu_vars), ALLOCATABLE :: shcu_g(:), shcum_g(:)

CONTAINS

  SUBROUTINE alloc_shcu(shcu,n1,n2,n3,ng)
    IMPLICIT NONE
    TYPE (shcu_vars) :: shcu
    INTEGER, INTENT(in) :: n1,n2,n3,ng
    ...
  END SUBROUTINE alloc_shcu

  SUBROUTINE nullify_shcu(shcu)
    IMPLICIT NONE
    TYPE (shcu_vars) :: shcu
    ...
  END SUBROUTINE nullify_shcu

  SUBROUTINE dealloc_shcu(shcu)
    IMPLICIT NONE
    TYPE (shcu_vars) :: shcu
    ...
  END SUBROUTINE dealloc_shcu

  SUBROUTINE filltab_shcu(shcu,shcum,imean,n1,n2,n3,ng)
    USE var_tables
    IMPLICIT NONE
    TYPE (shcu_vars) :: shcu, shcum
    INTEGER, INTENT(in) :: imean,n1,n2,n3,ng
    ...
  END SUBROUTINE filltab_shcu
END MODULE mem_shcu
```

Since a new module has been created, it is necessary to invoke its memory allocation and initialization routines at *rams_mem_alloc*, just like in section 6 for *basic_vars* (the user need also to remember to put the properly *USE* statement for Shallow Cumulus in the begging of *rams_mem_alloc* routine):

```
SUBROUTINE rams_mem_alloc(proc_type)

  USE mem_all
  USE node_mod

  USE mem_shcu   ! needed for Shallow Cumulus

  IMPLICIT NONE
```

```
  ...

  ! Allocate nested boundary interpolation arrays. All
  ! grids will be allocated.

  PRINT*,'start nestb alloc'
  IF (proc_type == 0 .OR. proc_type == 2) THEN
     DO ng=1,ngrids
        IF(nxtnest(ng) == 0 ) THEN
          CALL alloc_nestb(ng,1,1,1)
        ELSE
          CALL alloc_nestb(ng,nnxp(ng),nnyp(ng),nnzp(ng))
        ENDIF
     ENDDO
  ENDIF

  !-----------------------------------------------------------
  ! Allocate data for Shallow Cumulus

  DO ng=1, ngrids
     IF (NNSHCU(ng) == 1) Alloc_ShCu_Flag = 1
  ENDDO
  IF (Alloc_ShCu_Flag == 1) THEN
     PRINT*,'start ShCu alloc'
     ALLOCATE(shcu_g(ngrids),shcum_g(ngrids))
     DO ng=1,ngrids
        CALL nullify_shcu(shcu_g(ng))
        CALL nullify_shcu(shcum_g(ng))
        CALL alloc_shcu(shcu_g(ng),nmzp(ng),  &
                        nmxp(ng),nmyp(ng),ng)
        IF (imean == 1) THEN
           CALL alloc_shcu(shcum_g(ng),nmzp(ng),  &
                           nmxp(ng),nmyp(ng),ng)
        ELSEIF (imean == 0) THEN
           CALL alloc_shcu(shcum_g(ng),1,1,1,ng)
        ENDIF

        CALL filltab_shcu(shcu_g(ng),shcum_g(ng),imean,  &
                          nmzp(ng),nmxp(ng),nmyp(ng),ng)
     ENDDO
  ENDIF

  ! Set "Lite" variable flags according to namelist input
  ! LITE_VARS.

  IF (proc_type == 0 .OR. proc_type == 2) THEN
     CALL lite_varset()
  ENDIF

  RETURN
END SUBROUTINE rams_mem_alloc
```

*8 – Computing the shallow cumulus parameterization*

The final step is to include the invocation of the new parametrization routines at *timestep* (at file /BRAMS/src/rams/5.0/model/rtimh.f90). Take, for example, the Shallow Cumulus Parametrization invocation:

```
...
!  Get the overlap region between parallel nodes
!-------------------------------------------------
t1=cputime(w1)
IF(ipara == 1) THEN
   CALL node_getlbc()
   IF (ngrid  ==  1) CALL node_getcyclic(1)
ENDIF
CALL acctimes('accu',13,'GETlbc',t1,w1)

!  Sub-grid diffusion terms
!-----------------------------------------
t1=cputime(w1)
CALL diffuse ()
CALL acctimes('accu',12,'DIFFUSE',t1,w1)

!  Velocity advection
!-----------------------------------------
t1=cputime(w1)
CALL ADVECTc('T',mzp,mxp,myp,ia,iz,ja,jz,izu,jzv,mynum)
CALL acctimes('accu',15,'ADVECTs',t1,w1)

!srf-----------------------------------------------------
!srf - Shallow  cumulus parameterization
t1=cputime(w1)
IF(NSHCU.EQ.1) CALL SHCUPA()  ! Shallow Cumulus param.
CALL acctimes('accu',19,'SHCUPARM',t1,w1)
!srf-----------------------------------------------------

!  Update scalars
!-----------------------------------------
t1=cputime(w1)
CALL PREDTR()
CALL acctimes('accu',16,'PREDTR',t1,w1)
...
```

File BRAMS/src/rams/5.0/braz_modules/shallow_cum/rshcupar.f90 contains subroutine *SHCUPA()*, that uses previously allocated data for shallow cumulus and other required data present owned by others MODULEs:

```
SUBROUTINE SHCUPA()

  ! USE Modules for 5.0
  USE mem_basic
  USE mem_micro
  USE mem_grid
  USE mem_turb
  USE mem_tend
  USE node_mod,  &
     ONLY : MXP, MYP, MZP, IA, IZ, JA, JZ, I0, J0

  USE mem_shcu ! USE Module for Shallow Cumulus
  IMPLICIT NONE
  INCLUDE 'rcommons.h'
  ...
```

The *node_mod* MODULE contains the current grid dimensions at one parallel node (specified at the *USE ONLY* statement above).

*9 – Conclusions and Remarks*

RAMS is going through a continuous software modernization process, triggered by ANSI/OSI Fortran enhancements. It should result a more robust and reliable product (when it comes to software), without loosing processing speed. BRAMS is in pace with this process.

This document is designed to help users insert new procedures in BRAMS. But it does not touch other software engineering issues that should be obeyed. Among them, we strongly recommend the declaration of all variables - experience has shown the importance of using IMPLICIT NONE.

Another important point is that dynamic memory allocation has a strong impact on non-standard practices related to local variables initialization and SAVEs. Users tend to rely on automatic initialization (to zero) of all requested memory; that is, they do not explicitly initialize all local variables. Users also rely on static placement of variables into memory – which means that the value of a local variable will be kept among procedure calls. Although that is absolutely false in Fortran 77 standard-compliant programs (otherwise the SAVE command will be useless), Fortran 77 processors make this true in practice.

Both practices do not hold for dynamically allocated memory. Due to that, we strongly recommend users to initialize all local variables, as well as use the SAVE attribute on local variables that should keep their value among procedure invocations.