# Eradication of arrays bounds violations in IFS/ARPEGE/AROME

Ryad El Khatib

27 August 2019

*From cycle 47T0, the source should be modified in such a way that all arrays bounds violations will be banned forever.*

## Origins of the arrays bounds violations issue

Models fail at runtime when the whole code is compiled with arrays bounds checking option because of several « assumed » violations.
These violations are assumed because the code has been written in such a way that it should remain flexible for scientific researches and developments, and designed at the time the Fortran standard was Fortran 77.
Here is a typical example :

```
REAL(KIND=JPRB), INTENT(IN) :: PGMVT1(NPROMA,NFLEVG,NDIM)

CALL GPINISLB(..., PGMVT1(1,1,MNHX),…)
```

Inside the subroutine GPINISLB the variable PGMV( :, :,MNHX) will be used only if the model is non-hydrostatic. Consequently the address MNHX will be uninitialized (or initialized to a value out of the bounds of the array PGMV) if the model is hydrostatic. This will be harmless … as long as PGMV( :, :,MNHX) is not used by mistake. Therefore the check against arrays bounds violation would be helpful to avoid such bugs.
Unfortunately, once we have assumed that we would not do such mistakes in a given subroutine, we have to assume that the whole subroutine concerned cannot be compiled with arrays bounds checking option, and that may hide other bugs.

As a partial workaround, we may set the default address to the value of the variable NUNDEFLD which is in namelist. Its default value is a number huge enough to cause bounds violations, but if set to 1, then the potential bounds violations will be silent … to the risk that the model returns wrong results !

With the Fortran 90, a possible solution was to use optional arguments. That solution has been used here and there, but it has its limits : the code becomes very difficult again to maintain with increasing number of options, leading to endless combinations of optional arguments.

The recent object-oriented refactoring of the code does not help against this issue, on the contrary it can create more bounds violations issues. Here is an example, where a fortran derived type has been used :

```
TYPE(TRAJ_TYPE_OOPS) :: PTRAJEC_OOPS

DO JBL=1,NGPBLKS
  CALL CPG(...,PTRAJEC_OOPS%PHYS(JBL))
ENDDO
```

In that example, PHYS may be not allocated. Then the compilation with bounds checking would cause an error at runtime.
Again, the only solution seemed either to disable the bounds checking option, or to use optional arguments, with the consequences explained above.

## Solution to the arrays bounds violations

A final solution has been adopted, will be implemented in cycle 47T0 , and then later in the common cycle 48.
The general idea is to consider that we don't want to determine by ourselves the memory address of a field (or a group of fields) but, as a user, we request a field from a fields database and we expect the system to return its memory address to us.
Following this idea we can take advantage of Fortran pointers, not used as allocatable entities but only as true pointers.
The pointers will be computed by a generic function, given the array of origin and the expected address inside.
Let's consider its application to the first example above :

```
USE SC2PRG_MOD, ONLY : SC2PRG ! Generic function

REAL(KIND=JPRB), INTENT(IN) :: PGMVT1(NPROMA,NFLEVG,NDIM)

REAL(KIND=JPRB), POINTER :: ZNHXT1(:,:) ! User pointer declaration

CALL SC2PRG(MNHX,PGMVT1,ZNHXT1) ! Pointer initialization

CALL GPINISLB(..., ZNHXT1,…) ! Pointer is used instead of the direct addressing
```

The generic fonction will check the size of the array and the given address against the array bounds. If the given address is valid (within the array bounds) and the size of the array is not zero, then the pointer is initialized to the given adress. Otherwise the pointer is initialized to null. At runtime there won't be any bounds violation but if a pointer with null address is used, then the model will abort on a segmentation violation signal.

## Implementation

This solution has been implemented on top of cycle 46t1_r1.03, and tested successfully on a toy IFS application, and on ARPEGE and AROME at high resolution. Other tests on TL and AD models at low resolution have been tested successfully, too. A test of IFS 4Dvar is expected from ECMWF.

The modifications of code have a neutral impact on the computational performance, and had a null impact on the scientific results.

36 subroutines are directly concerned by the modifications for fixing bounds violation, though not all of them had needed the generic pointer function. For safety, most of them have been transformed thanks to a basic script. The list of modified subroutines follows :

aladin/adiab/elarmesad.F90

aladin/adiab/elarmes.F90

aladin/adiab/elarmestl.F90

aladin/adiab/elarmes5.F90

arpifs/control/gp_model.F90

arpifs/phys_dmn/mf_phys.F90

arpifs/phys_dmn/mf_physad.F90

arpifs/phys_dmn/mf_phystl.F90

arpifs/module/sc2prg_mod.F90

arpifs/adiab/larmesad.F90

arpifs/adiab/call_sl_ad.F90

arpifs/adiab/cpg_drv_ad.F90

arpifs/adiab/lapinea5.F90

arpifs/adiab/larcinb.F90

arpifs/adiab/cpg_drv_tl.F90

arpifs/adiab/cpg_gp_nhee.F90

arpifs/adiab/lapinea.F90

arpifs/adiab/cpg_dia.F90

arpifs/adiab/cpg_gp_ad.F90

arpifs/adiab/larmes5.F90

arpifs/adiab/cpg_drv.F90

arpifs/adiab/cpg_gp.F90

arpifs/adiab/larmes.F90

arpifs/adiab/laitre_gfl.F90

arpifs/adiab/larmestl.F90

arpifs/adiab/cpg_gp_hyd.F90

arpifs/adiab/cpglag.F90

arpifs/adiab/cpg_gp_tl.F90

arpifs/adiab/lattex.F90

arpifs/adiab/lapineaad.F90

arpifs/adiab/lapineatl.F90

arpifs/adiab/call_sl.F90

arpifs/phys_radi/radintg.F90

arpifs/phys_ec/ec_phys_drv.F90

arpifs/adiab/postphy.F90

ifsaux/fa/farcis.F90

## Next step

From the moment this solution is implemented in the code, executions of the model with the code compiled integrally with arrays bounds checking should be regularly performed ; and no arrays bounds violations will be tolerated anymore.

There are various techniques to fix the inevitable next « assumed » arrays bounds violations in the code :

- one of them is, of course, to use or add a new procedure to the generic function, especially if the incriminated variable is a fortran structure.

- one may also consider the technique of pointer remapping (a fortran 2003 features), which has it as been applied to call_sl_ad.F90 to preserve full vectorization with minimum modifications in the code :

```
REAL(KIND=JPRB), TARGET :: PB1(NASLB1,NFLDSLB1)

REAL(KIND=JPRB), POINTER :: ZB1(:,:)

ZB1(NASLB1*NFLDSLB1,1:1) => PB1 ! Pointer remapping

<   PB1(INC(JINC,JROF,1),1)=PB1(INC(JINC,JROF,1),1)+ZINC(JINC,JROF,1)

>   ZB1(INC(JINC,JROF,1),1)=ZB1(INC(JINC,JROF,1),1)+ZINC(JINC,JROF,1)
```

## Appendix : excerpt of the generic pointer function

```
MODULE SC2PRG_MOD ! Named in memory of glorious subroutines sc2rdg and sc2wrg
REAL(KIND=JPRB), POINTER :: FAKE2(:,:) => NULL()
INTERFACE SC2PRG
MODULE PROCEDURE SC2PRG2A, ...
END INTERFACE
CONTAINS
SUBROUTINE SC2PRG2A(KBL,PG2A,PGPR)
INTEGER(KIND=JPIM),
INTENT(IN) :: KBL
REAL(KIND=JPRB),TARGET, INTENT(IN) :: PG2A(:,:,:)
REAL(KIND=JPRB),POINTER :: PGPR(:,:)
IF (SIZE(PG2A) == 0) THEN
      PGPR => FAKE2
ELSEIF ( (LBOUND(PG2A,DIM=3) <= KBL) .AND. (KBL <= UBOUND(PG2A,DIM=3)) ) THEN
      PGPR => PG2A(:,:,KBL)
ELSE
      PGPR => FAKE2
ENDIF
END SUBROUTINE SC2PRG2A
END MODULE SC2PRG_MOD
```