INSTITUT DU DÉVELOPPEMENT ET DES RESSOURCES EN INFORMATIQUE SCIENTIFIQUE

Langage Fortran (Expert)

Patrick Corde
Hervé Delouis

Patrick.Corde@idris.fr

10 mai 2019

Table des matières I

Table des matières II

Table des matières III

Table des matières IV

Avertissement

Parmi les points abordés dans ce cours, certains nécessitent une version récente du compilateur employé.
Afin de connaitre le niveau d'intégration des normes 2003 et 2008 au sein des compilateurs usuels se reporter aux *URLs* suivantes :

- état d'avancement de l'intégration de la norme Fortran 2003 :
  `http://fortranwiki.org/fortran/show/Fortran+2003+status`
- état d'avancement de l'intégration de la norme Fortran 2008 :
  `http://fortranwiki.org/fortran/show/Fortran+2008+status`

Meilleure intégration à l'environnement système

Il est désormais possible d'accéder de façon portable aux arguments de la ligne de commande et aux variables d'environnement à l'aides de procédures.

```
GET_COMMAND( [command] [,length] [,status] )
```

- **command** est une chaîne de caractères dans laquelle sera stocké le nom de la commande (y compris les arguments s'ils existent) qui a servi à lancer l'exécutable. Elle sera valorisée avec des blancs si la récupération est impossible ;
- **length** est un entier qui sera valorisé à la longueur de la chaîne de caractères ci-dessus. Si impossibilité, 0 sera retournée ;
- **status** est un entier qui sert de code retour. Sa valeur sera :
  - 0 si l'exécution s'est bien déroulée ;
  - -1 si l'argument command a été précisée avec une taille inférieure à la longueur de la commande ;
  - > 0 si l'exécution s'est terminée en erreur.

Exemple

```
program p
  implicit none
  character(len=:), allocatable :: commande
  integer long
  call GET_COMMAND( LENGTH=long )
  allocate( character(len=long) :: commande )
  call GET_COMMAND( COMMAND=commande )
  print *, "Commande lancée ==> ", commande
  deallocate( commande )
end program p
```



```
GET_COMMAND_ARGUMENT( number [,value] [,length] [,status] )
```

- **number** est un entier qui est fourni en entrée. Il indique le numéro de l'argument désiré parmi ceux spécifiés lors du lancement de l'exécutable. La valeur 0 fait référence au nom de l'exécutable et la fonction COMMAND_ARGUMENT_COUNT retourne le nombre d'arguments qui suivent ;
- **value** est une chaîne de caractères dans laquelle sera retournée l'argument désigné ci-dessus. Une chaîne à blancs sera renvoyée si la valorisation est impossible ;
- **length** est un entier qui sera valorisé à la longueur de la chaîne de caractères ci-dessus. Si impossibilité, 0 sera retournée ;
- **status** est un entier qui sert de code retour qui est valorisé de la même façon que pour la procédure GET_COMMAND.

Exemple

```
program p
  implicit none
  character(len=:), allocatable :: arg
  integer long, i
  do i=0, COMMAND_ARGUMENT_COUNT()
    call GET_COMMAND_ARGUMENT( NUMBER=i, LENGTH=long )
    allocate( character(len=long) :: arg )
    call GET_COMMAND_ARGUMENT( NUMBER=i, VALUE=arg )
    print *,"Argument de rang ", i, " ==> ", arg
    deallocate( arg )
  end do
end program p
```

```
GET_ENVIRONMENT_VARIABLE( name [,value] [,length] [,status] [,trim_name] )
```

- **name** est une chaîne de caractères valorisée au nom de la variable d'environnement dont on désire le contenu ;
- **value** est une chaîne de caractères dans laquelle sera retournée le contenu de la variable d'environnement fournie ci-dessus. Une chaîne à blancs sera renvoyée dans les cas suivant :
  - la variable d'environnement indiquée n'existe pas ;
  - la variable d'environnement existe mais son contenu est vide ;
  - la notion de variable d'environnement n'existe pas sur la plateforme utilisée.
- **length** est un entier qui sera valorisé à la longueur de la valeur de la variable d'environnement fournie si celle-ci existe et a un contenu défini. Sinon 0 sera la valeur retournée.
- **status** est un entier qui sert de code retour qui est valorisé de façon suivante :
  - si la variable d'environnement existe avec un contenu vide ou bien admet une valeur retournée avec succès, 0 sera la valeur retournée ;
  - si la variable d'environnement existe et admet un contenu dont la taille est supérieure à celle de la chaîne de caractères fournie via le paramètre `value`, -1 sera retournée ;
  - ce code de retour est valorisé à 1 si la variable d'environnement n'existe pas et à 2 si il n'y a pas de notion de variable d'environnement sur la plateforme utilisée.
- **trim_name** est un logique. S'il est fourni avec `.FALSE.` comme valeur alors les caractères blancs situés à la fin du paramètre **name** seront considérés comme significatifs. Ils seront ignorés dans les autres cas.

Exemple

```
program p
  implicit none
  character(len=:), allocatable :: path
  integer long

  call GET_ENVIRONMENT_VARIABLE( NAME="PATH", LENGTH=long )
  allocate( character(len=long) :: path )
  call GET_ENVIRONMENT_VARIABLE( NAME="PATH", VALUE=path )
  print *, "PATH=", path
  deallocate( path )
end program p
```

Introduction

En **Fortran 95**, du fait des insuffisances notoires des tableaux dynamiques (attribut ALLOCATABLE), on leur substituait souvent les pointeurs plus puissants, mais présentant des inconvénients en terme de performance.

En **Fortran 2003**, les tableaux allouables sont désormais gérés par un descripteur interne analogue à celui d'un pointeur. Ce descripteur peut être vu comme un type dérivé semi-privé contenant, entre autres, l'adresse d'une zone dynamique anonyme.
Il en résulte que l'on pourra désormais tirer profit des avantages jusque-là réservés aux tableaux déclarés avec l'attribut POINTER. Ces avantages sont abordés ci-après.

On réservera alors l'usage des pointeurs aux fonctionnalités qui leur sont propres : notion d'alias dynamique d'entités éventuellement complexes, gestion de listes chaînées, pointeurs de procédures dans les types dérivés, ...

En **Fortran 95**, un tableau allouable ne pouvait être passé en argument d'appel que s'il était déjà alloué. Au sein de la procédure appelée, il était considéré comme un simple tableau à profil implicite (sans l'attribut ALLOCATABLE).

En **Fortran 2003** et en contexte d'interface explicite, un argument muet peut avoir l'attribut ALLOCATABLE.
L'argument d'appel correspondant devra aussi avoir cet attribut ainsi que le même rang et le même type/sous-type, sans être nécessairement déjà alloué.
Dans le cas où le tableau est alloué avant d'être transmis, il sera possible de récupérer ses bornes au sein de la procédure appelée comme c'était le cas auparavant au moyen de l'attribut POINTER.

Exemple

```
program alloca
  implicit none
  real, dimension(:), ALLOCATABLE :: vec_x, vec_y
  allocate( vec_y(-2:100) )
  call sp( vec_x, vec_y )
  print *, vec_x((/ lbound(vec_x), ubound(vec_x) /)), vec_y((/ -2, 100 /))
  deallocate( vec_x ); deallocate( vec_y )
CONTAINS
  subroutine sp( v_x, v_y )
    real, dimension(:), ALLOCATABLE, intent(inout) :: v_x, v_y
    ALLOCATE(v_x(256)); call random_number(v_x); call random_number(v_y)
    print *, "Bornes inf/sup(v_y) : ", lbound( v_y ), ubound( v_y )
  end subroutine sp
end program alloca
```

Remarque :

- Si un argument muet a la vocation INTENT(OUT), l'argument d'appel correspondant est automatiquement désalloué à l'appel de la procédure (s'il était alloué).

Composante allouable d'un type dérivé

L'attribut ALLOCATABLE est désormais autorisé au niveau d'une composante d'un type dérivé. En Fortran-95, seul l'attribut POINTER permettait de gérer des composantes dynamiques.

Exemple

```
program alloca
  type obj_mat
    integer :: N, M
    real, dimension(:,:), ALLOCATABLE :: A
  end type obj_mat
  type(obj_mat) :: mat

  read *, mat%N, mat%M
  allocate( mat%A(mat%N, mat%M) ); call random_number( mat%A )
  print *, mat%A( (/ 1, mat%N /), (/ 1, mat%M /) )
  deallocate( mat%A )
end program alloca
```

Remarques :

- C'est bien entendu le descripteur du tableau allouable qui sera stocké dans la composante A de mat.
- Quand un objet de type dérivé est désalloué, toute composante ayant l'attribut ALLOCATABLE est automatiquement désallouée. Si un destructeur (*final subroutine* – voir page 103) est attaché à l'objet, il est appliqué avant la désallocation de cette composante.

Allocation d'un scalaire ALLOCATABLE

L'attribut ALLOCATABLE peut dorénavant s'appliquer à un **scalaire**, notamment à une chaîne de caractères.

Exemple

```
module m
  implicit none
contains
  function strdup( ch_in )
    character(len=*) :: ch_in
    character(len=:), ALLOCATABLE :: strdup
    allocate( character(len=len_trim(ch_in)) :: strdup )
    strdup = trim( ch_in )
  end function strdup
end module m
program alloca_char
  use m
  implicit none
  character(len=256) :: ch_in
  character(len=:), ALLOCATABLE :: ch_out
  read( *, "(A)" ) ch_in; ch_out = strdup( ch_in )
  print "(i0,1x,a)", len(ch_out), ch_out
  deallocate( ch_out )
end program alloca_char
```

Dans cet exemple, la fonction strdup retourne une chaîne de caractères allouable dont la longueur est celle de la chaîne passée en argument débarrassée des blancs de fin. L'allocation est effectuée à l'aide de l'instruction ALLOCATE en explicitant le type et la longueur désirée.

Allocation/réallocation via l'affectation

Une allocation/réallocation d'une entité `var_alloc` (scalaire ou tableau ayant l'attribut `ALLOCATABLE`) peut se faire implicitement lors d'une opération d'affectation du type :

`var_alloc = expression`

1. Si `var_alloc` est déjà allouée, elle est automatiquement désallouée si des différences concernant le profil ou la valeur des *length type parameters* – (cf. §8.2) existent entre `var_alloc` et *expression*.
2. Si `var_alloc` est ou devient désallouée, alors elle est réallouée selon le profil et les paramètres de type de *expression*.

Voici un exemple permettant le traitement d'une chaîne de caractères de longueur variable :

```
character(:), ALLOCATABLE :: NAME
 . . . .
NAME = "Beethoven"; ...;  NAME = "Ludwig-Van Beethoven"
```

La variable scalaire `NAME` de type `CHARACTER` sera allouée lors de la première affectation avec une longueur `LEN=9`. Lors de la 2e affectation, elle sera désallouée puis réallouée avec une longueur `LEN=20`.

À noter que cette possibilité de réallocation dynamique facilite la gestion des chaînes dynamiques ainsi que le respect de la contrainte de conformance lors d'une affectation de tableaux. Ainsi par exemple :

```
real, ALLOCATABLE, dimension(:) :: x
     ...
!--allocate( x(count( masque )) ) <--- Devient inutile !
     ...
x = pack( tableau, masque )
```

Le tableau `x` est automatiquement alloué/réalloué avec le bon profil sans que l'on ait à se préoccuper du nombre d'éléments vrais de `masque`.

Note :

- ce processus d'allocation/réallocation automatique peut être inhibé en mentionnant explicitement tout ou partie de l'objet :

```
NAME(:)                 = "chaine_de_car"//"acteres"
x(1:count( masque )) = pack( tableau, masque )
```

  À gauche de l'affectation, la présence du caractère « : » signifie qu'on fait référence à un sous-ensemble d'une entité (`NAME` ou `x`) qui doit exister et donc être déjà allouée.

Procédure MOVE_ALLOC de réallocation

`MOVE_ALLOC(FROM, TO)`

Ce sous-programme permet de transférer une allocation d'un objet allouable à un autre.

- FROM fait référence à une entité allouable de n'importe quel type/rang. Sa vocation est INTENT(INOUT) ;
- TO fait référence à une entité allouable de type compatible avec FROM et de même rang. Sa vocation est INTENT(OUT).

En retour de ce sous-programme, le tableau allouable TO désigne le tableau allouable FROM ; la zone mémoire préalablement désignée par TO est désallouée.
En fait, c'est un moyen permettant de *vider* le descripteur de FROM après l'avoir recopié dans celui de TO.

- Si FROM n'est pas alloué en entrée, TO devient non alloué en sortie ;
- sinon, TO devient alloué avec les mêmes caractéristiques (type dynamique, paramètres de type, bornes de tableau, valeurs) que FROM avait en entrée. Parce que la vocation de TO est INTENT(OUT), il y aura désallocation de celui-ci ;
- Si TO a l'attribut TARGET, tout pointeur initialement associé à FROM devient associé à TO.

Exemple

```
real, ALLOCATABLE, dimension(:) :: grid, tempgrid
      ...
ALLOCATE(grid(-N:N))           ! Allocation initiale de grid
      ...
ALLOCATE(tempgrid(-2*N:2*N)) ! Allocation d'une grille plus grande
      ...                      !
tempgrid(::2) = grid           ! Redistribution des valeurs de grid
      ...                      !
call MOVE_ALLOC( TO=grid, FROM=tempgrid )
```

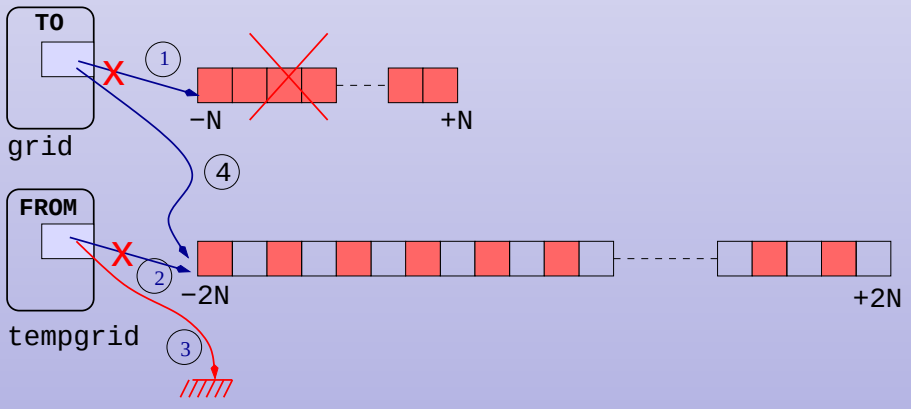


FIGURE 1 : Schéma correspond au move_alloc précédent

Exemple précédent en Fortran 95

```
real, ALLOCATABLE, dimension(:) :: grid, tempgrid
      ...                       !
ALLOCATE(grid(-N:N))            ! Allocation initiale de grid
      ...                       !
ALLOCATE(tempgrid(-2*N:2*N))    ! Allocation d'une grille plus grande
      ...                       !
tempgrid(::2) = grid            ! Redistribution des valeurs de grid
      ...                       !
DEALLOCATE( grid(:) )
ALLOCATE( grid(-2*N:2*N) )
grid(:) = tempgrid(:)
DEALLOCATE( tempgrid(:) )
```

L'attribut PROTECTED

De même que la vocation `INTENT(in)` protège les arguments muets d'une procédure, l'attribut `PROTECTED` protège les entités déclarées avec cet attribut **dans un module** ; elles sont exportables (*use association*) mais pas modifiables en dehors du module où est faite la déclaration :

```
real(kind=my_prec), PROTECTED, dimension(10,4) :: tab
```

- cet attribut n'est spécifiable que dans le module où est faite la déclaration, pas dans ceux qui l'importent (`USE`) ;
- les sous-objets éventuels d'un objet protégé reçoivent l'attribut `PROTECTED` ;
- pour un pointeur, c'est l'association et non la cible qui est protégée.

L'instruction IMPORT du bloc interface

En Fortran 95, un bloc interface ne pouvait accéder aux entités (définition de type dérivé par ex.) de l'unité hôte (module ou unité de programme). L'instruction IMPORT permet l'importation (*host association*) de ces entités dans un bloc interface :

Exemple

```
module truc
  type couleur
    character(len=25)      :: nom
    integer, dimension(3) :: code_rvb
  end type couleur
  interface
    function demi_teinte(col_in)
      IMPORT :: couleur ! Importation du type couleur
      type(couleur), intent(in) :: col_in
      type(couleur)             :: demi_teinte
    end function demi_teinte
  end interface
contains
      ...
end module truc
```

IMPORT sans liste permet l'importation de toutes les entités *vues* par *host association*.

USE et renommage d'opérateurs

Les **opérateurs** non intrinsèques d'un module peuvent être renommés au moment de leur importation via l'instruction USE. Voici un exemple :

```
USE my_module, OPERATOR(.MY_OPER.) => OPERATOR(.OPER.)
```

Notion de sous-module (SUBMODULE)

La norme 2008 définit la notion de « submodule », procédé permettant d'étendre un module existant. Le module étendu est le *hôte* ou le *parent* du sous-module appelé le *descendant*. La définition d'un sous-module s'effectue à l'aide de la syntaxe suivante :

```
SUBMODULE(parent) nom_sous-module
...
END SUBMODULE [nom_sous-module]
```

Un module peut être étendu et avoir plusieurs descendants ; la syntaxe est alors :

```
SUBMODULE(parent:nom_sous-module1) nom_sous-module2
...
END SUBMODULE [nom_sous-module2]
```

Ce procédé peut être employé pour séparer l'interface d'un traitement avec son implémentation ; l'interface sera défini dans le module parent et l'implémentation dans un sous-module : voir l'exemple suivant.

Exemple

```
module mod1
  type mus
    private
    character(len=:), allocatable :: nom
    integer dates(2)
  end type mus

  interface
    logical module function mus_plus_ancien(m1, m2)
      type(mus), intent(in) :: m1, m2
    end function mus_plus_ancien
    logical module function mus_mort_plus_jeune(m1, m2)
      type(mus), intent(in) :: m1, m2
    end function mus_mort_plus_jeune
    module subroutine imp(m)
      type(mus), intent(in) :: m
    end subroutine imp
  end interface

  interface mus
    module procedure construct_mus
  end interface mus
```

Exemple (suite)

```
contains
  function construct_mus(nom, dates)
    character(len=*),       intent(in) :: nom
    integer, dimension(:), intent(in) :: dates
    type(mus) :: construct_mus

    construct_mus%nom = nom
    construct_mus%dates = dates
  end function construct_mus
end module mod1
```

Rappel :

- la fonction *construct_mus* permet de définir un constructeur pour le type *mus* ; les composantes de celui-ci étant privées, le constructeur intrinsèque ne peut être appelé.

Exemple (suite)

```
submodule(mod1) mod2
contains
  logical module function mus_plus_ancien(m1, m2)
    type(mus), intent(in) :: m1, m2

    mus_plus_ancien = m1%dates(1) < m2%dates(1)
  end function mus_plus_ancien
  logical module function mus_mort_plus_jeune(m1, m2)
    type(mus), intent(in) :: m1, m2

    mus_mort_plus_jeune = &
      (m1%dates(2) - m1%dates(1)) < (m2%dates(2) - m2%dates(1))
  end function mus_mort_plus_jeune
  module subroutine imp(m)
    type(mus), intent(in) :: m

    print *, "NOM   = ", m%nom
    print *, "DATES = ", m%dates
  end subroutine imp
end submodule mod2
```

Exemple (suite)

```
program prog
  use mod1
  type(mus) m1, m2

  m1 = mus("Wolfgang Amadeus Mozart", [1756, 1791])
  m2 = mus("Ludwig Van Beethoven",    [1770, 1827])
  call imp(m1)
  call imp(m2)
  print *, "m1 plus ancien que m2 = ",      &
                    mus_plus_ancien(m1, m2)
  print *, "m1 mort plus jeune que m2 = ", &
                    mus_mort_plus_jeune(m1, m2)
end program prog
```

Notes :

- le module mod1 définit l'interface (appelé *separate interface*) et le sous-module mod2 l'implémentation ;
- le programme principal n'a besoin que du module mod1 ;
- si le module et le sous-module sont dans des fichiers distincts, une modification du sous-module ne nécessite pas la recompilation du programme principal (dans le cas bien sûr où cette modification n'entraine pas un changement de l'interface).

Constantes d'environnement pour les entrées_sorties

Le compilateur `Fortran` 2003 est livré avec plusieurs modules intrinsèques notamment `ISO_FORTRAN_ENV` permettant l'accès à des entités publiques qui concernent l'environnement :

- `INPUT_UNIT`, `OUTPUT_UNIT` et `ERROR_UNIT` sont des constantes symboliques de type entier correspondant aux numéros des unités logiques relatifs à **l'entrée standard**, **la sortie standard** et à **la sortie d'erreur**. Ils remplacent avantageusement l'**astérisque** employé traditionnellement au niveau du paramètre `UNIT` des instructions `READ`/`WRITE` .
- `IOSTAT_END` et `IOSTAT_EOR` sont des constantes symboliques de type entier correspondant aux valeurs négatives prises par le paramètre `IOSTAT` des instructions d'entrée/sortie en cas de fin de fichier ou fin d'enregistrement. Cela permet d'enrichir la portabilité d'un code `Fortran`. Cependant, les cas d'erreurs génèrent une valeur positive restant dépendante du constructeur.

Nouveaux paramètres des instructions OPEN/READ/WRITE

- IOMSG : ce paramètre des instructions READ/WRITE identifie une chaîne de caractères récupérant un message si une erreur, une fin de fichier ou une fin d'enregistrement intervient à l'issue de l'entrée-sortie.
- ROUND : lors d'une entrée-sortie formatée le mode d'arrondi peut être contrôlé à l'aide du paramètre ROUND de l'instruction OPEN qui peut prendre comme valeurs : "up", "down", "zero", "nearest", "compatible" ou "processor_defined". Cette valeur peut être changée au niveau des instructions READ/WRITE à l'aide du même paramètre ou via les spécifications de format ru, rd, rz, rn, rc et rp. La valeur par défaut dépend du processeur utilisé.
- SIGN : ce paramètre a été ajouté à l'instruction OPEN pour la gestion du signe « + » des données numériques en sortie. Il peut prendre les valeurs : "suppress", "plus" ou "processor_defined". Cette valeur peut être changée au moment de l'instruction WRITE à l'aide du même paramètre, ou bien à l'aide des spécifications de format ss, sp et s. La valeur par défaut est "processor_defined".
- IOSTAT : deux nouvelles fonctions élémentaires is_iostat_end et is_iostat_eor permettent de tester la valeur de l'entier référencé au niveau du paramètre IOSTAT de l'instruction READ. Elles retournent la valeur vraie si une fin de fichier ou une fin d'enregistrement a été détectée.
  Remarque : les paramètres comme IOSTAT peuvent dorénavant référencer tout type d'entier.

Exemple

```
program io
  use ISO_FORTRAN_ENV
  implicit none
  character(len=256) :: message
  integer(kind=2)    :: stat
  integer            :: i
  real               :: a, b
  namelist /var/i, a, b

  open( UNIT=1,              FILE="new_param_data", &
        FORM="formatted",    ACTION="read",         &
        ACCESS="sequential", ROUND="up" )
  open( UNIT=OUTPUT_UNIT, SIGN="plus" )

  do
    read( UNIT=1, FMT="(i4,1x,rd,f8.6,1x,rn,f10.1)", &
          IOSTAT=stat, IOMSG=message ) i, a, b
    if ( is_iostat_end( stat ) ) exit
    if ( stat > 0 ) then
      write( OUTPUT_UNIT, * ) trim(message); exit
    end if
    write( OUTPUT_UNIT, NML=var )
  end do

  close( UNIT=1 )
end program io
```

Entrées-sorties asynchrones

Les entrées-sorties peuvent être faites en mode asynchrone, permettant ainsi au programme de continuer son exécution pendant que l'entrée-sortie est en cours. Ce mode de fonctionnement n'est possible que pour les fichiers externes ouverts avec le paramètre ASYNCHRONOUS='yes'. Ce même paramètre sera fourni également au niveau de l'instruction READ/WRITE si l'on désire lancer une telle entrée-sortie, sinon, par défaut ou en précisant le paramètre ASYNCHRONOUS='no', elle sera synchrone quel que soit le mode d'ouverture effectuée.
Une synchronisation peut être demandée explicitement à l'aide de l'instruction WAIT. Celle-ci est implicite à la rencontre d'un INQUIRE ou d'un CLOSE sur le fichier.
Pour cette instruction on retrouve certains paramètres des instructions READ/WRITE tels que :

UNIT, END, EOR, ERR, IOSTAT, IOMSG, ERR, ID

Le paramètre ID fait référence à une variable entière qui a été prélablement valorisée à l'issue d'une opération d'entrée-sortie. S'il n'est pas spécifié au niveau de l'instruction WAIT, celle-ci concerne toutes les entrées-sorties en cours.

Exemple

```
program io
  use ISO_FORTRAN_ENV
  implicit none
  character(len=256)          :: message
  integer(kind=2)             :: etat
  integer                     :: i, ident
  real                        :: a, b, c
  real, dimension(1024*1024) :: tab
  namelist /var/i, a, b, c

  open( UNIT=1,               FILE="new_param_data", FORM="formatted", ACTION="read", &
        ACCESS="sequential", ASYNCHRONOUS="yes",    ROUND="up" )
  open( UNIT=OUTPUT_UNIT, SIGN="plus" )
  do
    read( UNIT=1,       FMT="(i4,1x,rd,f8.6,1x,rn,f10.1)", &
          IOSTAT=etat, ASYNCHRONOUS="yes",
          ID=ident,     IOMSG=message ) i, a, b
    if (etat /= 0) then; call trait_erreur; exit; end if
    call random_number(tab)
    write( UNIT=OUTPUT_UNIT, FMT=* ) tab( [lbound(tab), ubound(tab)] )
    WAIT( UNIT=1, ID=ident, IOSTAT=etat, IOMSG=message )
    if (etat /= 0) then; call trait_erreur; exit; end if
    c = i*a+b; write( OUTPUT_UNIT, NML=var )
  end do
  close( UNIT=1 )
contains
  subroutine trait_erreur
    if ( is_iostat_end(etat) ) write( OUTPUT_UNIT, "('Fin de fichier atteinte.')" )
    if (etat > 0) write( OUTPUT_UNIT, * ) trim(message)
  end subroutine trait_erreur
end program io
```

- Toute entité faisant l'objet d'entrées-sorties asynchrones récupère automatiquement un nouvel attribut ASYNCHRONOUS dans le but d'avertir le compilateur du risque encouru à optimiser des portions de code les manipulant.
  En effet, le résultat de cette optimisation pourrait être un déplacement d'instructions référençant ces entités avant une instruction de synchronisation.
- On peut préciser explicitement cet attribut lors de la déclaration :

```
INTEGER, ASYNCHRONOUS, DIMENSION(10,40) :: TAB
```

Entrées-sorties en mode *stream*

Le paramètre ACCESS de l'instruction OPEN admet une troisième valeur "STREAM" permettant d'effectuer des entrées-sorties en s'affranchissant de la notion d'enregistrement : le fichier est considéré comme étant une suite d'unités (unité représentant l'octet ou le mot suivant l'environnement). L'entrée-sortie est faite soit relativement à la position courante, soit à une position donnée.

- Le fichier peut être formaté ou non.
- La position courante est mesurée en unités en partant de 1. Le paramètre POS de l'instruction INQUIRE permet de la connaître.
- Le paramètre POS (expression entière) des instructions READ/WRITE indique la position dans le fichier à partir de laquelle s'effectuera l'entrée-sortie.
- Cette nouvelle méthode d'accès facilite notamment l'échange de fichiers binaires entre Fortran et C.

Exemple

```
double precision d
integer          rang
      ...
open(UNIT=1, ..., ACCESS="STREAM", form="unformatted")
      ...
inquire(UNIT=1, POS=rang) ! sauvegarde de la position courante.
write(UNIT=1) d           ! par rapport à la position courante.
      ...
! par rapport à la position sauvegardée dans rang.
write(UNIT=1, POS=rang) d+1
```

Exemple de création d'un fichier en mode stream pour être relu en C

```
program ecrire
  implicit none
  integer, parameter  :: n = 3
  real, dimension(10) :: v
  integer un, i

  open(NEWUNIT=un,      FILE="bin.data",     ACCESS="stream", &
       ACTION="write", FORM="unformatted", STATUS="replace")
  do i=1,n
    call random_number(v)
    write(unit=un) v
  end do
  close(unit=un)
end program ecrire
```

Vocation (INTENT) des arguments muets pointeurs

Contrairement à Fortran 95, il est possible de définir la vocation (attribut INTENT) des arguments muets pointeurs au sein d'une procédure. C'est l'association qui est concernée et non la cible.

- INTENT(IN) : le pointeur ne pourra ni être associé, ni mis à l'état nul, ni alloué au sein de la procédure ;
- INTENT(OUT) : le pointeur est forcé à l'état indéfini à l'entrée de la procédure ;
- INTENT(INOUT) : le pointeur peut à la fois transmettre une association préétablie et retourner une nouvelle association.

Exemple

```
module m
  implicit none
  type vecteur
    integer n
    real, dimension(:), allocatable :: v
  end type vecteur
contains
  subroutine copy(v2, v1)
    type(vecteur), POINTER, INTENT(INOUT) :: v2
    type(vecteur), POINTER, INTENT(IN)    :: v1
    if( associated(v2) .AND. associated(v1, v2) ) return
    if( .not.associated(v2) ) allocate(v2)
    v2 = v1
    print *, lbound(v2%v), ubound(v2%v) ! lbound(v1%v), ubound(v1%v)
  end subroutine copy
end module m
```

Exemple (suite)

```
program copie
  use ISO_FORTRAN_ENV, only : OUTPUT_UNIT
  use m
  implicit none
  integer n

  type(vecteur), pointer :: vec1, vec2
  allocate(vec1)
  read *, vec1%n
  allocate(vec1%v(-1:vec1%n-2))
  call random_number(vec1%v)
  call copy(vec1, vec1)
  call copy(vec2, vec1)
  write(UNIT=OUTPUT_UNIT, FMT=*) vec2%v([ 1, ubound(vec2%v)])
  deallocate(vec1%v)  ! inutile
  deallocate(vec2%v)  ! inutile
  deallocate(vec1)    ! vec1%v sera désalloué
  deallocate(vec2)    ! vec2%v sera désalloué
end program copie
```

Association et reprofilage : norme 2003

Lors de l'association d'un pointeur à une cible il est dorénavant possible :

- d'indiquer un nouveau profil au niveau du pointeur tout en respectant le rang de la cible ;
- de modifier de plus le profil uniquement dans le cas d'une cible de rang 1.

Exemple

```
program profil
  integer, parameter              :: N=100, M=200
  integer, parameter              :: NXL=5, NXU=20, NYL=10, NYU=30
  real, dimension(N,M), target    :: mat
  real, dimension(:,:), pointer   :: bloc_mat

  call random_number(mat)
  bloc_mat(0:, 0:) => mat(NXL:NXU, NYL:NYU)
  print *, "Bornes sup de bloc_mat : ", ubound(bloc_mat)
end program profil
```

Exemple (suite)

```
! Le sous-programme suivant récupère un tableau de rang 1
! (champ_vecteurs) supposé contenir les coordonnées d'une
! série de vecteurs dans un espace à 3 dimensions.
subroutine sp(champ_vecteurs, n)
  real, dimension(:),   target  :: champ_vecteurs
  integer                       :: n, nb_vecteurs
  real, dimension(:,:), pointer :: matrice

  nb_vecteurs = size(champ_vecteurs)/3
  matrice(1:3,1:nb_vecteurs) => champ_vecteurs
              ...
end subroutine sp
! Sous-programme dans lequel on désire avoir une vue matricielle
! du vecteur "vec" puis considérer la diagonale de cette matrice.
subroutine sp(n)
  integer n
  real, dimension(n*n), target  :: vec
  real, dimension(:,:), pointer :: matrice
  real, dimension(:),   pointer :: diag

  call random_number(vec)
  ! matrice(n, n) => vec (INVALIDE)
  matrice(1:n, 1:n) => vec; diag => vec(1::n+1)
end subroutine sp
```


Association et reprofilage : norme 2008

Un nouvel attribut CONTIGUOUS peut être précisé lors de la déclaration :

- d'un argument muet de type tableau à profil implicite ;
- d'un pointeur de tableau.

Dans le premier cas, cela indique que l'argument doit désigner un tableau d'éléments contigus. Dans le deuxième, le pointeur ne pourra être associé qu'à un tableau dont les éléments sont contigus.

Le reprofilage d'un tableau n'est plus limité aux tableaux de rang 1 comme le stipule la norme 2003 mais peut s'appliquer à tout tableau à condition que celui-ci référence des éléments contigus (ayant l'attribut CONTIGUOUS par exemple).

L'exemple suivant montre comment interpréter en tant que vecteur un argument muet désignant un tableau de rang quelconque. Cette technique permet de faire l'économie mémoire qu'aurait suscité un appel à la fonction RESHAPE.

Association et reprofilage : norme 2008

Exemple de reprofilage

```
module reprof
  interface sp
    module procedure sp_2D, sp_3D, ...
  end interface sp
contains
  subroutine sp_2D(tab)
    real, dimension(:,:), contiguous, target, intent(inout) :: tab
    real, dimension(:), pointer :: p

    p(1:size(tab)) => tab
        ...
  end subroutine sp_2D
  subroutine sp_3D(tab)
    real, dimension(:,:,:), contiguous, target, intent(inout) :: tab
    real, dimension(:), pointer :: p

    p(1:size(tab)) => tab
        ...
  end subroutine sp_3D
    ...
end module reprof
```

Remarque :

- concernant l'argument muet tab, les compilateurs peuvent décider d'effectuer une copie, si nécessaire, lors de l'appel à la procédure sp.

Divers

Déclaration de procédures

Lorsque l'on désire transmettre une procédure en argument, il est nécessaire d'informer le compilateur que celui-ci est de type procédural, ceci au moyen de l'attribut EXTERNAL.
Il est désormais possible de le déclarer de façon beaucoup plus précise grâce à l'instruction PROCEDURE.
Cette instruction peut servir également à déclarer des pointeurs de procédures (voir ci-après) ainsi que toute procédure non nécessairement transmise en argument afin, par exemple, de lui associer par la suite un pointeur.
La forme la plus simple (et de ce fait la plus pauvre) de déclaration de procédure est :

```
PROCEDURE() sub
```

Elle est équivalente à la déclaration `EXTERNAL sub` employée auparavant. Rien n'est précisé sur le type de procédure : c'est le mode de l'interface implicite.

Pour une procédure de type FUNCTION, on peut indiquer le type retourné comme dans la déclaration :

```
PROCEDURE(REAL) :: func
```

Ici, func désigne une procédure de type FUNCTION retournant un résultat de type REAL. Rien de plus n'est précisé, on est toujours en mode d'interface implicite.
Si l'on désire utiliser le mode d'interface explicite, on indiquera un nom de procédure ou bien le nom d'un bloc interface abstrait servant de modèle.

Exemple

```
module m
  implicit none
  abstract interface
    function func(x)
      double precision, intent(in) :: x
      double precision func
    end function func
  end interface
contains
  function integrale( borne_i, borne_s, pas, f )
    double precision :: borne_i, borne_s, integrale, h
    integer          :: pas, i
    PROCEDURE(func)  :: f

    h = (borne_s - borne_i)/pas
    integrale = 0.
    do i=0, pas-1
      integrale = integrale + h*f(borne_i+i*h)
    end do
  end function integrale
  function carre( x )
    double precision, intent(in) :: x
    double precision carre
    carre = x*x
  end function carre
end module m
```


Exemple (suite)

```
program p
  use m
  implicit none
  double precision :: b_inf, b_sup, aire
  integer          :: pas

  b_inf = 1.
  b_sup = 6.
  pas = 200000
  aire = integrale( b_inf, b_sup, pas, carre )
  print "('Aire : ', f11.6)", aire
end program p
```

Remarques :

- le bloc interface est un bloc interface *abstrait* qui permet de définir un modèle de procédure pour des déclarations ultérieures,
- la déclaration de l'argument muet *f* au sein de la fonction *integrale* est effectuée au moyen de l'instruction PROCEDURE à laquelle on a indiqué le nom du modèle ci-dessus permettant ainsi de fiabiliser l'appel à *f* effectué plus loin.

Pointeurs de procédures

Les pointeurs peuvent être associés à des cibles de type procédure. On parlera alors de pointeurs de procédures, assimilables aux pointeurs de fonctions en langage C.
Voici quelques aspects concernant ces pointeurs :

- l'interface peut être implicite ou explicite (cf. ci-après) ;
- une fonction peut retourner un pointeur de procédure ;
- au moment de l'association `p => proc` d'un pointeur de procédure `p` à l'aide de l'opérateur classique « => », le compilateur vérifie (si l'interface est explicite) la compatibilité des interfaces procédurales comme pour l'appel classique d'une procédure. L'opérande de droite peut être au choix :
  - une procédure ;
  - un pointeur de procédure ;
  - une fonction retournant un pointeur de procédure.

Interface implicite

Voici trois possibilités de les déclarer en mode d'interface implicite :

- `Première possibilité` (pour une fonction seulement) avec l'attribut `EXTERNAL` :

```
REAL, EXTERNAL, POINTER :: p
REAL, EXTERNAL :: f, g
...
p => f
print *, p(..., ..., ...)
...
p => g
print *, p(..., ..., ...)
```

- `Deuxième possibilité` avec les attributs `PROCEDURE()` et `POINTER` ; dans l'exemple, `p` est un pointeur en mode d'interface implicite, à l'état indéterminé, pouvant être associé à une fonction ou un sous-programme :

```
PROCEDURE(), POINTER :: p
```

- `Troisième possibilité` (pour une fonction seulement) avec l'attribut `PROCEDURE` référençant un type pour *expliciter* partiellement un pointeur de fonction en le déclarant par exemple sous la forme : `PROCEDURE(complex), POINTER :: p`
  `p` ne pourra alors être associé qu'à une fonction retournant un objet de type `complex`.

Bien entendu, comme pour les appels de procédures, il est plutôt conseillé d'utiliser le mode d'**interface explicite** afin que le compilateur puisse contrôler la cohérence des associations de pointeurs de procédures... La fiabilité est à ce prix !
Voyons quelles sont les possibilités de les déclarer en mode d'interface explicite.

Interface explicite

- Attribut PROCEDURE faisant référence à une procédure *modèle* existante. Voici par exemple la déclaration de p initialement à l'état nul avec la même interface que celle du sous-programme proc (obligatoirement en mode d'interface explicite) :

Exemple

```
module m
contains
  subroutine proc( a, b )
    real, dimension(:), intent(in)  :: a
    real, dimension(:), intent(out) :: b
      . . .
  end subroutine proc
end module m
program prog
  use m
  PROCEDURE(proc), POINTER :: p => NULL()
      . . .
end program prog
```

- À défaut d'une procédure pouvant servir de modèle pour expliciter l'interface, il est aussi possible de définir un **bloc interface virtuel** (*abstract interface*) tel que :

```
ABSTRACT INTERFACE
  SUBROUTINE sub( x, y )
    REAL, intent(out) :: x
    REAL, intent(in) :: y
  END SUBROUTINE sub
END INTERFACE
```

Ce bloc interface virtuel peut être utilisé (via l'attribut PROCEDURE) au moment de la déclaration d'un pointeur de procédure comme p1 ou même d'une procédure externe comme proc dans l'exemple ci-dessous :

Exemple

```
PROCEDURE(sub), POINTER :: p1=>NULL()
PROCEDURE(sub)          :: proc
REAL                    :: var

p1 => proc
 . . .
CALL p1( x=var, y=3.14 )
PRINT *, ASSOCIATED( p1, proc )
```

Constructeurs de structures

Lors de la valorisation d'un type dérivé via un *constructeur de structure*, il est désormais possible d'affecter les composantes par mots clés :

Exemple

```
type couleur
  character(len=25)     :: nom
  integer,dimension(3)  :: code_rvb
end type couleur
    ...
type(couleur) :: c
    ...
c = couleur(nom="rose_saumon", code_rvb=[ 250, 128, 114 ])
    ...
```

Remarques

- Si une composante d'un type dérivé à une valeur par défaut, l'argument correspondant au niveau du constructeur se comporte comme un argument optionnel,
- Lors de l'appel au constructeur intrinsèque, le type ainsi que chaque composante explicitement précisée devront être accessible au sein de l'unité de programme dans laquelle cet appel apparaît,
- Pour une composante comportant l'attribut `pointer`, c'est une association (au sens du symbole « => ») qui s'effectuera avec l'argument précisé au niveau du constructeur. On pourra indiquer la fonction `NULL` pour forcer la composante à l'état nul de façon explicite,
- Pour une composante comportant l'attribut `allocatable`, l'argument précisé au niveau du constructeur devra être de même type et de même rang. S'il a l'attribut `allocatable`, il peut être ou non alloué. Comme pour une composante `pointer`, on pourra préciser la fonction `NULL` mais sans argument : la composante sera alors dans l'état "non alloué".

Exemple

```
module m
  type t
    real, dimension(:),   pointer     :: p => NULL()
    character(len=:),     allocatable :: chaine
    real, dimension(:,:), allocatable :: matrice
  end type t
end module m
program construct
  use ISO_FORTRAN_ENV; use m; implicit none
  type(t) u
  real, dimension(:),   pointer     :: q
  real, dimension(20),  target      :: v
  real, dimension(2,3)              :: mat1
  real, dimension(:,:), allocatable :: mat2
  integer i
  call random_number(v); call random_number(mat1); q => v
  u = t(p=v, chaine="Mozart", matrice=mat1)
  write( OUTPUT_UNIT, *) u%p, u%chaine, len(u%chaine)
  if (allocated(u%matrice)) call sortie         ! <== .true.
  u = t(p=q, chaine="Beethoven", matrice=mat2)
  write(OUTPUT_UNIT, *) u%p, u%chaine, len(u%chaine)
  if (allocated(u%matrice)) call sortie         ! <== .false.
  u = t(p=v, chaine="Verdi", matrice=NULL())
  write(OUTPUT_UNIT, *) u%p, u%chaine, len(u%chaine)
  if (allocated(u%matrice)) call sortie         ! <== .false.
  u = t(chaine="Wagner", matrice=NULL())
  write(OUTPUT_UNIT, *) u%chaine, len(u%chaine)
  mat2 = mat1; u = t(p=q, chaine="Schubert", matrice=mat2)
  write(OUTPUT_UNIT, *) u%p, u%chaine, len(u%chaine)
  if (allocated(u%matrice)) call sortie         ! <== .true.
contains
  subroutine sortie
    do i=1,size(u%matrice,1)
      write(OUTPUT_UNIT, *) u%matrice(i,:)
    end do
  end subroutine sortie
end program construct
```


Une procédure générique peut avoir le même nom qu'un constructeur de structure. Un appel valide à l'un des constituants de la famille générique ne doit pas rentrer en conflit avec l'appel du constructeur. Si aucun constituant ne fait l'affaire, c'est le constructeur intrinsèque qui sera activé. Cette technique peut être employée afin de surcharger un constructeur, comme le montre l'exemple suivant :

Exemple

```
module m
  type mycomplex
    real :: rho, theta  !<=== Coordonnées polaires
  end type mycomplex
  interface mycomplex
    module procedure complex_to_mycomplex, two_reals_to_mycomplex
  end interface mycomplex
contains
  function complex_to_mycomplex(c) result(myc)
    complex, intent(in) :: c
    type(mycomplex)     :: myc
    real, parameter     :: PI = acos(-1.)
    real                :: x, y

    myc%rho = abs(c) ; x = real(c); y = aimag(c)
    ! myc%theta ∈ ]] − π ; π]]
    myc%theta = atan(y, x)
    if (x < 0. .AND. y < 0.) myc%theta = myc%theta - PI
    if (x < 0. .AND. y > 0.) myc%theta = myc%theta + PI
  end function complex_to_mycomplex
```

Exemple

```
  function two_reals_to_mycomplex(x,y) result(myc)
    real, intent(in) :: x, y
    type(mycomplex)  :: myc

    myc%rho = sqrt(x*x + y*y)
    myc%theta = atan2(y, x) ! ∈ ]−π;π]
  end function two_reals_to_mycomplex
end module m
program p
  use, intrinsic :: ISO_FORTRAN_ENV, only : OUTPUT_UNIT
  use m
  type(mycomplex) :: a, b, c
  complex         :: w

  w = (0.,1.)
  a = mycomplex(theta=5.6, rho=1.0)       ! Appel constructeur intrinsèque
  b = mycomplex(w)                        ! Appel à complex_to_mycomplex
  c = mycomplex(x=0.5, y=3.**.5/2.)       ! Appel à two_reals_to_mycomplex
  write(OUTPUT_UNIT, *) "A(rho,theta) : ", a%rho, a%theta
  write(OUTPUT_UNIT, *) "B(rho,theta) : ", b%rho, b%theta
  write(OUTPUT_UNIT, *) "C(rho,theta) : ", c%rho, c%theta
end program p
```

Visibilité des composantes

En Fortran 95, un type dérivé pouvait seulement être public, privé ou semi-privé :

Exemple

```
!------ type publique -------
type struct_publique
  integer              :: i, j
  real,dimension(160)  :: tab
end type struct_publique
!-------- type privé ---------
type, private :: struct_privee
  integer              :: i, j
  real,dimension(160)  :: tab
end type struct_privee
!----- type semi-privé ------
type struct_semi_privee
  private
  integer              :: i, j
  real,dimension(160)  :: tab
end type struct_semi_privee
```

En Fortran 2003, il est désormais possible de déterminer quelles seront les composantes privées et quelles sont celles qui seront publiques. Les mots-clés private et public s'emploient comme pour un module : l'instruction private positionne un accès par défaut, celui-ci pouvant être surchargé à l'aide de l'attribut public/private indiqué lors de la définition des composantes :

Exemple : module m1

```
module m1
  type t
    private
    integer          :: i
    integer, public  :: j
    real             :: r
  end type t
end module m1
```

Exemple : module m2

```
module m2
  type t
    private
    integer :: i, j
    real    :: r
  end type t
  interface t
    module procedure construct_t
  end interface t
contains
  function construct_t(i, j, r)
    integer, intent(in) :: i, j
    real,    intent(in) :: r
    type(t)             :: construct_t

    construct_t%i = i
    construct_t%j = j
    construct_t%r = r
  end function construct_t
  subroutine sortie(obj)
    use, intrinsic :: ISO_FORTRAN_ENV, &
                      only : OUTPUT_UNIT
    type(t), intent(in) :: obj

    write(OUTPUT_UNIT, *) obj
  end subroutine sortie
end module m2
```


Exemple (suite)

```
program p
  use m1
  type(t) obj

  obj%i = 1                        ! invalide : i est privé au module
  obj%j = 2                        ! valide   : j est public
  print *, obj%r                   ! invalide : r est privé au module
  obj = t(i=1, j=2, r=2.718)       ! invalide : i et r sont privés
  call sub
contains
  subroutine sub
    use m2
    type(t) :: obj
    obj = t(i=1, j=2, r=3.)   ! valide
    call sortie(obj)
  end subroutine sub
end program p
```

Note :

- dans la procédure sub, la fonction t appelée est bien la fonction *construct_t* définie dans le module m2 comme une surcharge du constructeur intrinsèque, ceci au moyen d'une interface générique qui porte le même nom que le type : il n'y a aucune ambiguité avec l'appel du constructeur intrinsèque car les composantes du type étant privées, celui-ci n'est pas activable explicitement.

Paramètres d'un type dérivé

En Fortran 95, les types intrinsèques étaient déjà paramétrables ; en particulier le type CHARACTER(LEN= , KIND=) a deux paramètres à valeur entière pour spécifier le nombre de caractères et le sous-type.
Nuance importante, le premier n'est pas discriminant pour la généricité et pas nécessairement connu à la compilation, tandis que le deuxième l'est.
Fortran 2003 permet de définir pour un type dérivé des composantes particulières. On leur indiquera l'**attribut** LEN ou KIND, elles serviront à paramétrer le type et pourront être utilisées lors de la définition des autres composantes. Il est possible de leur donner une valeur qui servira de valeur par défaut lors de la déclaration d'objets de ce type.

Remarque sur leur emploi au sein du type :

- ceux avec l'**attribut** LEN ne peuvent apparaître qu'au niveau d'une *specification-expression* permettant de définir par exemple une longueur de chaînes ou des bornes de tableaux,
- ceux avec l'**attribut** KIND peuvent de plus apparaître au niveau d'une *initialization-expression*.

Exemple

```
type obj_mat(K, DIM, LN)
  !-----------------------------------------
  integer, KIND :: K=4
  integer, LEN  :: DIM, LN = 100
  !-----------------------------------------
  real(kind=K), dimension(DIM,DIM) :: tab
  character(len=LN)                :: type_mat
end type obj_mat
     ...
type(obj_mat(DIM=256)) :: mat1
type(obj_mat(K=8, DIM=512, LN=30)) :: mat2
     ...
```

- Les paramètres K, DIM et LN sont obligatoirement de type entier avec un attribut KIND/LEN.
- Seuls ceux ayant l'attribut KIND sont discriminants au niveau de la généricité des fonctions.
- Ceux avec l'attribut LEN font partie des *length type parameters* par opposition aux *kind type parameters*.
- Pour une meilleure lisibilité, on a précisé ces paramètres en majuscules afin de mieux les différencier des composantes du type, lesquelles apparaissent en minuscules.

Constructeur et type paramétrable

```
type champ_vecteur(K,DIM1,DIM2)
  integer, KIND :: K = kind(1.0)
  integer, LEN  :: DIM1 = 2, DIM2
  real(kind=K), dimension(DIM1,DIM2) :: champ
end type champ_vecteur
integer, parameter :: prec = selected_real_kind(9,99)
type(champ_vecteur(K=prec, DIM2=100)) :: c
   . . .
c = champ_vecteur(K=prec, DIM2=100)(champ=1._prec)
```

Note : il est possible d'adresser les composantes d'un type dérivé ayant l'attribut `LEN` ou `KIND` à l'aide du symbole « % ». Ce procédé s'applique également aux types intrinsèques :

```
subroutine sub(ch, len)
  character(len=*), intent(inout) :: ch
  integer,          intent(in)    :: len
  integer, parameter              :: prec = selected_real_kind(9,99)
  real(kind=prec), dimension(10)  :: t

  print *, len, ch%len, t%kind ! len(ch) ambigu ici, kind(t) est valide.
end subroutine
```


Assumed-type-parameter et *deffered-type-parameter*

Par analogie avec les tableaux à taille implicite (*assumed-size-array*), lors d'une déclaration, il est possible d'introduire le caractère « * » pour les paramètres ayant l'attribut `LEN` (*assumed-type-parameter*). Cette forme peut être employée pour un argument muet d'une procédure :

```
subroutine impression_champ(c)
  integer, parameter :: prec = selected_real_kind(9,99)
  type(champ_vecteur(K=prec, DIM1=*, DIM2=*)) :: c
          ...
  print *,c%DIM1, c%DIM2
end subroutine impression_champ
```

De même, toujours pour ces paramètres, le caractère « : » peut être utilisé (*deffered-type-parameter*), comme pour la longueur d'une chaîne de caractères ou les dimensions d'un tableau :

```
integer, parameter :: prec = selected_real_kind(9,99)
type(champ_vecteur(K=prec, DIM1=:, DIM2=:)),   pointer :: p
type(champ_vecteur(K=prec, DIM1=3, DIM2=200)), target  :: cible
       ...
p => cible
```

Extension d'un type dérivé non paramétré

Un type existant peut être enrichi. Pour cela, on définit un nouveau type en référençant le type que l'on désire étendre au moyen de l'attribut EXTENDS. Un type comportant l'attribut SEQUENCE ou BIND n'est pas un type extensible.

Exemple

```
module m
  type coul
    integer, dimension(3) :: code_rvb
  end type coul
  type, extends(coul) :: coul_ext
    character(len=:), allocatable :: nom
  end type coul_ext
end module m
program p
  use m
  type(coul_ext) :: c

  c%coul%code_rvb = [255, 255, 0] ! ou c%code_rvb = [255, 255, 0]
  c%nom = "Jaune"
  print *, c%code_rvb, c%nom
end program p
```


Constructeur d'un type dérivé étendu

Lors de l'extension d'un type dérivée, le ou les types étendus possèdent leur contructeur du nom du type étendu. Reprenons l'exemple précédent avec appel du constructeur pour le type étendu.

Exemple

```
module m
  type coul
    integer, dimension(3) :: code_rvb
  end type coul
  type, extends(coul) :: coul_ext
    character(len=:), allocatable :: nom
  end type coul_ext
end module m
program p
  use m
  type(coul_ext) :: ce; type(coul) :: c
  ce = coul_ext(code_rvb=[ 255, 0, 0 ], nom="Rouge")
  print *, ce%code_rvb, ce%nom
  ce = coul_ext(coul=coul([ 0, 255, 0 ]), nom="Vert")
  print *, ce%code_rvb, ce%nom
  c = coul(code_rvb=[ 0, 0, 255 ])
  ! le cas suivant est correct même si les
  ! composantes du type "coul" sont privées.
  ce = coul_ext(coul=c, nom="Bleu")
end program p
```

Type dérivé paramétré

Lorsqu'un type paramétré est étendu, le nouveau type hérite des paramètres du type qu'il complète et peut en définir de nouveaux, comme le montre l'exemple suivant :

```
module m
  use ISO_FORTRAN_ENV, only : REAL64
  type champ_vect(K,DIM1,DIM2)
    integer, kind :: K = REAL64
    integer, len  :: DIM1, DIM2
    real(kind=K), dimension(DIM1,DIM2) :: champ
  end type champ_vect
  type, extends(champ_vect) :: champ_vect_label(LN)
    integer, len  :: LN
    character(len=LN) :: label
  end type champ_vect_label
end module m
program p
  use m
  implicit none
  call sp(d1=2, d2=100, texte="champ de forces électromagnétiques")
  call sp(d1=3, d2=200, texte="champ de vitesses")
contains
  subroutine sp(d1, d2, texte)
    integer,          intent(in) :: d1, d2
    character(len=*), intent(in) :: texte
    type(champ_vect_label(DIM1=d1, DIM2=d2, LN=len(texte))) :: c
    real(kind=c%K), dimension(c%DIM1,c%DIM2) :: mat
    call random_number(mat)
    c = champ_vect_label(DIM1=c%DIM1, DIM2=c%DIM2, LN=c%LN)(champ=mat, label=texte)
  end subroutine sp
end program p
```


Variable polymorphique : définition

C'est une variable dont le type peut varier au cours de l'exécution. Celle-ci doit avoir l'attribut `POINTER` ou `ALLOCATABLE` ou bien être un argument muet d'une procédure. Pour sa déclaration, on spécifie le mot-clé `CLASS` à la place de `TYPE`.

Exemple

```
type point
  real :: x, y
end type point
CLASS(point), pointer :: p
```

Dans cet exemple, le pointeur p pourra être associé à un objet de type `point` et à toutes les extensions éventuelles de ce type.
Le type indiqué au niveau du mot-clé `CLASS` doit forcément être un type dérivé extensible, ce qui exclut les types intrinsèques et les types dérivés pour lesquels on a précisé l'attribut `sequence` ou `bind`.

Variable polymorphique : remarques

- Une variable polymorphique ne peut figurer comme membre de gauche de l'instruction d'affectation intrinsèque ;
- On appelle *declared type* le type indiqué à la déclaration au moyen du mot-clé `CLASS` et *dynamic type* le type réel de l'objet à l'exécution ;
- On a la possibilité de se définir également des variables sans aucun type à la déclaration permettant ainsi de leur donner n'importe quel type dynamique : on les appelle *unlimited polymorphic* et on leur spécifie `CLASS(*)` lors de la déclaration.

Argument muet polymorphique

Son type dynamique est celui de l'argument réel fourni à l'appel. Cela permet d'appliquer à tous les types étendus une procédure définie pour le type de base.

Exemple

```
module m
  type point2d
    real x, y
  end type point2d
  type, extends(point2d) :: point2d_coul
    integer, dimension(3) :: code_rvb
  end type point2d_coul
  type, extends(point2d) :: point3d
    real z
  end type point3d
  type, extends(point3d) :: point3d_coul
    integer, dimension(3) :: code_rvb
  end type point3d_coul
contains
  ! Calcul de la distance entre les points p1 et p2
  function distance(p1, p2)
    CLASS(point2d) p1, p2
    real distance
        ...
  end function distance
end module m
```

Les exemples suivant présentent la façon dont l'association entre les arguments réels et muets s'effectue dans le cas de variable polymorphique.
L'association entre argument réel (*actual-argument*) et argument muet (*dummy-argument*) s'effectue selon la règle suivante :

- si l'argument muet est d'un type fixé, le type déclaré de l'argument réel doit être le même ;
- si l'argument muet est polymorphique, le type déclaré de l'argument réel doit être le même que celui de l'argument muet ou bien d'un type étendu ;
- si l'argument muet est polymorphique et admet l'attribut ALLOCATABLE, l'argument réel doit également être polymorphique avec le même type déclaré que celui de l'argument muet.

Exemple

```
module m
  type point2d; real x, y; end type point2d
  type, extends(point2d) :: point3d
    real z
  end type point3d
contains
  ! procédures scl2d, scl3d, st2d, st3d listées ci-dessous
end module m
```

```
subroutine scl2d(arg)
 CLASS(point2d) arg
       ...
end subroutine scl2d
```

```
subroutine scl3d(arg)
 CLASS(point3d) arg
       ...
end subroutine scl3d
```

```
subroutine st2d(arg)
 TYPE(point2d) arg
       ...
end subroutine st2d
```

```
subroutine st3d(arg)
 TYPE(point3d) arg
       ...
end subroutine st3d
```


Exemple

```
subroutine sp(cl2d, cl3d)
  use m
  CLASS(point2d) cl2d; CLASS(point3d) cl3d
  TYPE(point2d)  t2d;  TYPE(point3d)  t3d
  Argument muet polymorphique, argument réel d'un type fixé
  call scl2d(t2d)          ! valide
  call scl2d(t3d)          ! valide
  call scl3d(t2d)          ! invalide
  call scl3d(t3d)          ! valide
  Argument muet d'un type fixé, argument réel polymorphique
  call st2d(cl2d)          ! valide
  call st2d(cl3d)          ! invalide
  call st2d(cl3d%point2d) ! valide
  call st3d(cl2d)          ! invalide
  select type(cl2d)
    class is(point3d)
      call st3d(cl2d)      ! valide
       ...
  end select
  call st3d(cl3d)          ! valide
  Argument muet et argument réel polymorphiques
  call scl2d(cl2d)         ! valide
  call scl2d(cl3d)         ! valide
  call scl3d(cl2d)         ! invalide
end subroutine sp
```

Variable polymorphique : attributs POINTER, ALLOCATABLE

Une variable polymorphique ayant l'attribut POINTER (ou pointeur polymorphique) a pour type dynamique celui de sa cible. De même, le type dynamique d'une variable polymorphique ayant l'attribut ALLOCATABLE est celui fourni lors de son allocation.

Exemple

```
program p
  use m
  type(point2d),  target      :: p2d
  type(point3d),  target      :: p3d
  CLASS(point2d), pointer     :: ptr2d_1, ptr2d_2
  CLASS(point3d), pointer     :: ptr3d
  CLASS(point2d), allocatable :: point

  ptr2d_1 => p2d                    ! type dynamique de ptr2d_1 ⇒ point2d
  ptr2d_2 => p3d                    ! type dynamique de ptr2d_2 ⇒ point3d
  ptr3d   => p3d                    ! type dynamique de ptr3d   ⇒ point3d
  ptr2d_2 => ptr2d_1                ! type dynamique de ptr2d_2 ⇒ point2d
  ptr3d   => ptr2d_1                ! Interdit
  ALLOCATE(ptr2d_1)                 ! type dynamique de la cible ⇒ point2d
  ALLOCATE(point3d :: ptr2d_2)      ! type dynamique de la cible ⇒ point3d
  ALLOCATE(point3d_coul :: point)
      ...
end program p
```


Une variable polymorphique peut également être allouée en faisant référence à une variable non polymorphique préexistante. Celle-ci sert de modèle, son type sera celui de la variable allouée et son contenu sera copié. Pour effectuer ce type d'allocation, on spécifie le mot-clé « SOURCE= » au niveau de l'instruction allocate permettant de préciser l'objet source désiré.

Exemple

```
program p
  use m
  type(point2d) :: p2d
  type(point3d) :: p3d
  class(point2d), allocatable :: ptr

  p2d = point2d(x=1.5, y=2.)
  p3d = point3d(x=1.5, y=2., z=3.)
  ALLOCATE(ptr, SOURCE=p2d)
        ...
  deallocate(ptr)
  ALLOCATE(ptr, SOURCE=p3d)
        ...
  deallocate(ptr)
end program p
```

Construction SELECT TYPE

Cette construction permet l'exécution de blocs d'instructions en fonction du type dynamique d'une variable polymorphique.

Exemple

```
module m
  use, intrinsic :: ISO_FORTRAN_ENV, only : OUTPUT_UNIT
  type couleur
    integer, dimension(3) :: code_rvb
  end type couleur
  type, extends(couleur) :: coul_label
    character(len=:), allocatable :: nom
  end type coul_label
contains
  subroutine output(obj)
    CLASS(couleur), intent(in) :: obj

    select type(obj)
      CLASS is(couleur)
        write(OUTPUT_UNIT, *) "Composantes : ", obj%code_rvb
      CLASS is(coul_label)
        write(OUTPUT_UNIT, *) "Nom : ",           trim(obj%nom), &
                              ", composantes : ", obj%code_rvb
    end select
  end subroutine output
```


Exemple (suite)

```
  subroutine init(obj, rvb, nom)
    CLASS(couleur),              intent(out) :: obj
    integer, dimension(:),       intent(in)  :: rvb
    character(len=*), optional, intent(in)  :: nom

    obj%code_rvb = rvb
    select type(obj)
      CLASS is(coul_label)
        if (.not. present(nom)) then
          obj%nom="Noir"
        else
          obj%nom=nom
        end if
    end select
  end subroutine init
end module m
program p
  use m
  CLASS(couleur), allocatable :: c
  type(couleur)               :: coul

  allocate(coul_label :: c)
  call init(obj=c, rvb=[ 158, 158, 158 ], nom="Gris souris")
  call output(c); deallocate(c)
  coul = couleur(code_rvb=[ 158, 158, 158 ])
  allocate(c, source=coul)
  call output(c); deallocate(c)
end program p
```

```
 Nom : Gris souris, composantes : 158 158 158
 Composantes : 158 158 158
```

Exemple : variable polymorphique générique

```
module m
  type t
    integer i
  end type t
  type tt
    real x
  end type tt
contains
  subroutine sp(obj)
    class(*), intent(out) :: obj ! unlimited polymorphic object

    select type(obj)
      type is(t)
        obj%i = 1756
      type is(tt)
        obj%x = acos(-1.)
      type is(real)
        obj = exp(1.)
      type is(character(len=*))
        obj = "Pression initiale"
    end select
  end subroutine sp
end module m
program unlim
  use m
  type(t)             obj_t
  type(tt)            obj_tt
  real                x
  character(len=100)  texte

  call sp(obj_t);  print *, obj_t
  call sp(obj_tt); print *, obj_tt
  call sp(x);      print *, x
  call sp(texte);  print *, trim(texte)
end program unlim
```


Notes :

C'est le type dynamique de l'objet qui est analysé. La sélection du bloc d'instructions à exécuter se fait d'après les règles suivantes :

- s'il existe une instruction `TYPE IS` correspondant au type, le bloc qui suit est exécuté ;
- sinon, s'il existe une seule instruction `CLASS IS` répondant au type (l'objet est du type ou d'un type étendu de celui spécifié), le bloc qui suit est exécuté ;
- sinon, s'il existe plusieurs instructions `CLASS IS` correspondant au type, c'est le bloc de l'instruction `CLASS IS` référençant le type le plus proche de celui de l'objet qui est exécuté ;
- sinon, s'il existe une instruction `CLASS DEFAULT`, c'est son bloc qui est exécuté.

Type effectif d'une variable polymorphique

Deux nouvelles fonctions intrinsèques permettent de déterminer le type d'une variable polymorphique au moment de l'exécution :

- `SAME_TYPE_AS( a, b )`

  Cette fonction retourne `vrai` si `a` et `b` ont le même type dynamique.

- `EXTENDS_TYPE_OF( a, mold )`

  Celle-ci retourne `vrai` si le type dynamique de `a` est une extension de celui de `mold`.

Exemple

```
function distance(p1, p2)
  CLASS(point2d) p1, p2
  real           distance

  ! Calcul de la distance entre les points p1 et p2
  if (SAME_TYPE_AS(p1, p2)) then
    SELECT TYPE(p1)
      CLASS IS(point2d)
        distance = sqrt( (p2%x-p1%x)**2 + (p2%y-p1%y)**2 )
      CLASS IS(point3d)
        distance = sqrt( (p2%x-p1%x)**2 + (p2%y-p1%y)**2 + &
                                          (p2%z-p1%z)**2 )
      CLASS DEFAULT
        print *, "Erreur : type non reconnu"; distance = -1.
    END SELECT
  else
    print *, "Erreur : les objets p1 et p2 &
             &doivent être de même type"
    distance = -2.
  endif
end function distance
```

Variable polymorphique et opérateur d'affectation

Une variable polymorphique ne peut figurer comme opérande de gauche de l'opérateur d'affectation intrinsèque. Mais s'il s'agit d'une surcharge de cet opérateur l'opération devient valide comme le montre l'exemple suivant :

Exemple

```
module m
  type base
    integer i
  end type base
  type, extends(base) :: t
    real r
  end type t
  interface assignment(=)
    module procedure affect
  end interface assignment(=)
contains
 subroutine affect(a, b)
   class(base), allocatable, intent(out) :: a
   class(base), intent(in)  :: b

   allocate(a, source=b)
 end subroutine affect
end module m
program p
  use m
  class(base), allocatable :: obj
  type(t) source

  source%i = 1756; source%r = acos(-1.)
  obj = source ! Valide sans surcharge depuis le standard 2008.
end program p
```


Composante pointeur de procédure

- Un pointeur de procédure peut apparaître en tant que composante d'un type dérivé.
- L'association puis l'appel de la procédure cible s'effectue à partir d'un objet du type dérivé au moyen du symbole « % » comme pour l'accès aux composantes habituelles.
- Par défaut, l'objet qui est à l'origine de l'appel est transmis implicitement comme premier argument à la procédure cible (*passed-object dummy argument*). Celui-ci doit obligatoirement être déclaré comme objet polymorphique donc avec l'attribut `CLASS`.
- L'attribut `NOPASS`, indiqué lors de la déclaration de la composante, empêche cette transmission implicite : dans ce cas, si on désire transmettre l'objet, on le fera explicitement.
- L'attribut `PASS` peut être indiqué soit pour confirmer le mode par défaut soit pour transmettre implicitement l'objet vers un autre argument que le premier.

Exemple

```
module m
  use, intrinsic :: ISO_FORTRAN_ENV, only : OUTPUT_UNIT
  TYPE mytype
    private
      real    :: x
      integer :: i
      PROCEDURE(proc), public, POINTER, PASS   :: p
      PROCEDURE(func), public, POINTER, NOPASS :: q
  END TYPE mytype
  abstract interface
    subroutine proc(this, r, i)
      import mytype
      CLASS(mytype), intent(inout) :: this
      real,          intent(in)    :: r
      integer,       intent(in)    :: i
    end subroutine proc
    function func(x)
      real x, func
    end function func
  end interface
contains
  subroutine set(this, r, i)
    CLASS(mytype), intent(inout) :: this
    real,          intent(in)    :: r
    integer,       intent(in)    :: i

    this%x = r; this%i = i
    write(OUTPUT_UNIT, "('THIS%X = ', f6.2,/,'THIS%I = ', i6)") this%x, this%i
  end subroutine set
  function f(x)
    real x, f

    f = x*exp(x)
  end function f
end module m
```


Exemple (suite)

```
program prog
  use m
  TYPE(mytype) :: a
  real         :: y

  a%p => set
  call a%p(3.14, 100) ! équivalent à "call set(a, 3.14, 100)"
  a%q => f
  y = a%q(Log(3.14))
  write(OUTPUT_UNIT, "('Y = ', f6.2)") y
end program prog
```

Remarque :

- En Fortran 95, un bloc interface ne pouvait accéder aux entités (définition de type dérivé par ex.) de l'unité hôte (module ou unité de programme). L'instruction IMPORT permet l'importation (*host association*) de ces entités dans un bloc interface. D'où la présence de l'instruction **import mytype** au sein de l'*abstract interface*. IMPORT sans aucune spécification permet l'importation de toutes les entités *vues* par *host association*.

Procédures attachées à un type (*type-bound procedures*)

Souvent, en programmation orientée objet, on désire appeler une procédure pour effectuer un traitement dont la nature dépend du type dynamique d'un objet polymorphique. Ceci est mis en œuvre au moyen de procédures attachées à un type dérivé (*type-bound procedures*) qui récupèrent en entrée l'objet à l'origine de l'appel défini comme argument muet polymorphique. Celles-ci peuvent faire l'objet d'une surcharge lors d'extensions du type.
Dans d'autres langages comme C++ on les appelle des **méthodes** ou **services** ; leur invocation est vue comme l'envoi d'un message à un objet dont la nature peut être résolue à l'exécution (**polymorphisme dynamique**) ou à la compilation (**polymorphisme statique**).
Ce type de procédure peut évidemment s'employer pour un type dérivé simple non étendu.
Elles doivent être définies en contexte d'interface explicite.
Au sein du type, elles apparaîssent après le mot-clé `CONTAINS` lequel annonce la *type-bound procedure section*. Par défaut, leur accès est public indépendamment du mode d'accès des composantes du type. Comme pour les composantes l'instruction `private` en tête de section permet de changer le mode d'accès par défaut, sachant qu'il sera possible de le préciser au niveau de chaque procédure comme attribut.

Pour attacher des procédures (*type-bound procedures*) à un type on emploie la ou les instructions `PROCEDURE` et `GENERIC`.
Cet attachement s'effectue à l'aide d'un *binding-name*. Le nom de la procédure peut être celui-ci ou bien un nom précisé à la suite des caractères « => » au niveau de l'instruction `PROCEDURE`.
La syntaxe de ces instructions est la suivante :

- ❶ `PROCEDURE [, binding-attr-list ::] binding-name [=> procedure-name]`
  ❷ `PROCEDURE(interface-name), binding-attr-list :: binding-name`
- `GENERIC [, access-spec] :: generic-spec => binding-name-list`

binding-attr-list : `PASS`, `NOPASS`, `NON_OVERRIDABLE`, `DEFERRED` ou *access-spec*
access-spec : `PUBLIC` ou `PRIVATE`

Notes :

- la deuxième syntaxe de l'instruction `PROCEDURE` concerne les types abstraits, l'attribut `DEFERRED` est alors obligatoire (voir page 121) ;
- l'instruction `GENERIC` permet de faire de la généricité de la surcharge d'opérateurs ou des entrés-sorties portant sur des objets de type dérivé (voir page 151).

Procédure attachée par nom (*specific binding*)

Exemple

```
module m
  type T
    ... !--> Déclaration des composantes
    ... !--> du type dérivé T
  contains
    PROCEDURE :: proc => my_proc ! Procédure publique
  end type T
contains
  subroutine my_proc(b, x, y)
    class(T), intent(inout) :: b !--> Passed-object dummy argument
    real,     intent(in)    :: x, y
        . . .
  end subroutine my_proc
end module m

type(T) :: obj
real    :: x1, y1
. . .
call obj%proc(x1, y1)  !--> call my_proc(obj, x1, x2)
. . .
```


Une procédure externe peut être attachée à un type. Dans ce cas, évidemment, on précisera obligatoirement l'attribut NOPASS au moment de l'attachement. On fournira de plus un bloc interface afin que l'interface soit explicite.

Exemple

```
subroutine toto( ... )
   ...
end subroutine toto
module m
  type T
    private
    integer i
    contains
      private
      PROCEDURE, PUBLIC, NOPASS :: toto
  end type T

  interface
    subroutine toto( ... )
       ...
    end subroutine toto
  end interface
end module m
```

Procédure attachée par nom générique (*generic binding*)

Exemple

```
module mod
  use ISO_FORTRAN_ENV, only : REAL32, REAL64
  type matrix(K, N, M)
    integer, kind :: K; integer, len  :: N, M
    real(kind=K), dimension(N,M) :: A
  contains
    PROCEDURE, PRIVATE :: max32
    PROCEDURE, PRIVATE :: max64
    GENERIC :: max => max32, max64
  end type matrix
contains
  real(kind=REAL32) function max32(this)
    class(matrix(K=REAL32, N=*, M=*)), intent(in) :: this

    max32 = MAXVAL(array=this%A)
  end function max32
  real(kind=REAL64) function max64(this)
    class(matrix(K=REAL64, N=*, M=*)), intent(in) :: this

    max64 = MAXVAL(array=this%A)
  end function max64
end module mod
program prog
  use ISO_FORTRAN_ENV, only : REAL32, REAL64
  use mod
  type(matrix(K=REAL32, N=10, M=20)) :: obj1
  type(matrix(K=REAL64, N=20, M=50)) :: obj2
  real(kind=REAL32) maxs
  real(kind=REAL64) maxd
     . . .
  maxs = obj1%max(); maxd = obj2%max()
end program prog
```


Procédure attachée par opérateur (*operator binding*)

Exemple

```
module mod
  use ISO_FORTRAN_ENV, only : REAL32, REAL64
  type matrix(K, N, M)
    integer, kind                :: K
    integer, len                 :: N, M
    real(kind=K), dimension(N,M) :: A
  contains
    PROCEDURE, PRIVATE :: add32
    PROCEDURE, PRIVATE :: add64
    PROCEDURE, PRIVATE :: affect32
    PROCEDURE, PRIVATE :: affect64
    GENERIC :: OPERATOR(+)    => add32,    add64
    GENERIC :: ASSIGNMENT(=) => affect32, affect64
  end type matrix
contains
  function add32(a, b)
    class(matrix(K=REAL32, N=*, M=*)), intent(in ) :: a, b
    ! "type" et non "class".
    type(matrix(K=REAL32, N=:, M=:)), allocatable :: add32

    if( any(shape(a%A) /= shape(b%A)) ) stop "Erreur : objets non conformants"
    allocate(matrix(K=REAL32, N=a%N, M=a%M) :: add32)
    add32%A(:,:) = a%A(:,:) + b%A(:,:)
  end function add32
  function add64(a, b)
    class(matrix(K=REAL64, N=*, M=*)), intent(in) :: a, b
    ! "type" et non "class".
    type(matrix(K=REAL64, N=a%N, M=a%M)) :: add64

    if( any(shape(a%A) /= shape(b%A)) ) stop "Erreur : objets non conformants"
    add64%A = a%A(:,:) + b%A(:,:)
  end function add64
```

Exemple (suite)

```
  subroutine affect32(a, b)
    class(matrix(K=REAL32, N=*, M=*)), intent(inout) :: a
    class(matrix(K=REAL32, N=*, M=*)), intent(in)    :: b

    if( any(shape(a%A) /= shape(b%A)) ) &
                  stop "Erreur : objets non conformants"
    a%A(:,:) = b%A(:,:)
  end subroutine affect32
  !--------------------------------
  subroutine affect64(a, b)
    class(matrix(K=REAL64, N=*, M=*)), intent(inout) :: a
    class(matrix(K=REAL64, N=*, M=*)), intent(in)    :: b

    if( any(shape(a%A) /= shape(b%A)) ) &
                  stop "Erreur : objets non conformants"
    a%A(:,:) = b%A(:,:)
  end subroutine affect64
end module mod
program prog
  use mod
  type(matrix(K=REAL64, N=5,  M=10)) :: Amat64, Bmat64, Cmat64
  type(matrix(K=REAL32, N=20, M=60)) :: Amat32, Bmat32, Cmat32
        ...
  Amat32 = Bmat32 + Cmat32 ! call Amat32%affect32(Bmat32%add32(Cmat32))
        ...
  Amat64 = Bmat64 + Cmat64 ! call Amat64%affect64(Bmat64%add64(Cmat64))
end program prog
```


Procédure attachée mot-clé FINAL (*final binding*)

- C'est une procédure de type *subroutine* qui s'exécute lorsqu'un objet cesse d'exister. Pour cela, au sein du type dérivé correspondant à l'objet, on spécifie le mot-clé FINAL auquel on associe une liste de sous-programmes (*final subroutines*) appelés destructeurs ;
- Ceux-ci admettent un seul argument muet du type de celui défini ;
- Pour un objet alloué dynamiquement à l'aide de l'instruction ALLOCATE, le destructeur est appelé au moment de sa désallocation effectuée au moyen de l'instruction DEALLOCATE ;
- Pour un objet automatique, le destructeur est appelé lorsque l'unité de programme, au sein de laquelle l'objet est défini, est désactivée ;
- Par contre, si le programme s'arrête suite à une erreur ou bien lors de l'exécution de l'instruction STOP ou END de l'unité principale, aucune procédure de finalisation n'est exécutée.

Exemple

```
module mod
  use ISO_FORTRAN_ENV, only : REAL32
  implicit none
  type t(K)
    integer, KIND :: K
    real(kind=K), pointer, dimension(:) :: v => null()
  contains
    FINAL :: finalize_scal , finalize_vect
  end type t
contains
  subroutine finalize_scal(x) !--> Arg. scalaire
    type(t(K=REAL32)) :: x

    if( associated(x%v) ) deallocate(x%v)
  end subroutine finalize_scal
  subroutine finalize_vect(x) !--> Arg. vecteur
    type(t(K=REAL32)), dimension(:) :: x
    integer i

    do i=1,size(x)
      call finalize_scal(x(i))
    end do
  end subroutine finalize_vect
end module mod
```


Exemple (suite)

```
program prog
  use ISO_FORTRAN_ENV, only : REAL32
  use mod
  implicit none
  type(t(K=REAL32)), dimension(:), allocatable :: obj
  integer i

  allocate(obj(100))
  do i=1,100
    allocate(obj(i)%v(100))
    call random_number(obj(i)%v)
    print *, obj(i)%v(100)
  end do
  deallocate(obj)
end program prog
```

Remarque :

- Ainsi, lors de la désallocation de l'objet `obj` c'est le destructeur `finalize_vect` qui sera exécuté.

Pour un type étendu, lors de la destruction d'un objet de ce type :

- si le type étendu contient une procédure de finalisation, celle-ci est d'abord exécutée,
- puis la procédure de finalisation du type parent, si elle existe, est exécutée avec en argument la composante de l'objet correspondant au type parent.

Exemple

```
module m
  type base
    real, dimension(:), pointer :: p => NULL()
    contains
      FINAL :: fin_base
  end type base
  type, extends(base) :: t
    real, dimension(:), pointer :: q => NULL()
    contains
      FINAL :: fin_t
  end type t

contains
```

Exemple (suite)

```
  subroutine fin_base(this)
    type(base) :: this
    print *, "Finalisation du type base."
    if (associated(this%p)) deallocate(this%p)
  end subroutine fin_base

  subroutine fin_t(this)
    type(t) :: this
    print *, "Finalisation du type t."
    if (associated(this%q)) deallocate(this%q)
  end subroutine fin_t
end module m

program type_extent_final
  use m
  type(t), allocatable :: obj

  allocate(obj); allocate(obj%p(100))
  call random_number(obj%p)
  print *, obj%p([ 1, 50, 100 ])
  allocate(obj%q(200))
  call random_number(obj%q)
  print *, obj%q([ 1, 50, 100, 150, 200 ])
  deallocate(obj)
end program type_extent_final
```

L'exemple précédent produit les résulats suivant :

```
 0.7826369256E-05 0.4364113808 0.4153946042
 0.5373039246 0.4885145426 0.8246973753 0.6163503528 0.7947697639
 Finalisation du type t.
 Finalisation du type base.
```

Remarques :

- Si on déclare l'objet `obj` tel que `type(t) :: obj`, aucune procédure de finalisation n'est exécutée, cette déclaration figurant dans l'unité de programme principale.
- Pour une hierarchie de types, ce processus de finalisation s'applique de façon récursive en partant du type le plus riche en remontant successivement au type de base. Les procédures de finalisation sont alors exécutées dans cet ordre sur la partie de l'objet concernée.
- Les composantes possédant l'attribut `ALLOCATABLE` sont automatiquement libérées par le compilateur lors de la destruction d'un objet : il est alors inutile de prévoir une procédure de finalisation pour effectuer explicitement cette libération mémoire.

Héritage d'une procédure attachée à un type

Un type étendu d'un type extensible hérite à la fois de ses composantes mais également de ses procédures qui lui sont attachées.

Exemple

```
module point
  private
  type, public :: point2d
    real x, y
  contains
    PROCEDURE, PASS :: affichage => affichage_2d
  end type
contains
  subroutine affichage_2d(this, texte)
    CLASS(point2d),  intent(in) :: this
    CHARACTER(len=*), intent(in) :: texte
    print *, texte; print *, "X = ", this%x; print *, "Y = ", this%y
  end subroutine affichage_2d
end module point
module pointcoul
  use point
  private
  type, public, extends(point2d) :: point2d_coul
    integer, dimension(3) :: code_rvb
  end type point2d_coul
end module pointcoul
program p
  use pointcoul
  type(point2d_coul) :: pcoul
  call pcoul%affichage( "Voici mes coordonnées" )
end program p
```

Surcharge d'une procédure *type-bound* (*specific binding*)

Lors de la définition d'un type étendu d'un type extensible il est possible d'étendre ou surcharger (*override*) les procédures *type_bound*.

Exemple

```
module pointext
  use point
  type, public, extends(point2d) :: point3d
    real z
  contains
    PROCEDURE, PASS :: affichage => affichage_3d
  end type
contains
  subroutine affichage_3d(this, texte)
    CLASS(point3d),   intent(in) :: this
    CHARACTER(len=*), intent(in) :: texte
    call this%point2d%affichage(texte)
    print *, "Z = ", this%z
  end subroutine affichage_3d
end module pointext
program p
  use pointext
  type(point3d) pt
  call pt%affichage("Voici mes coordonnées")
end program p
```


Surcharge d'une procédure *type-bound* (*generic binding*)

Exemple

```
module mod
  use ISO_FORTRAN_ENV, only : REAL32, REAL64
  implicit none
  type matrix(K, N, M)
    integer, kind :: K; integer, len :: N, M
    real(kind=K), dimension(N,M) :: a ! ici, initialisation invalide.
  contains
    PROCEDURE, PRIVATE :: max32
    PROCEDURE, PRIVATE :: max64
    PROCEDURE, PRIVATE :: sortie32
    PROCEDURE, PRIVATE :: sortie64
    GENERIC :: max => max32, max64
    GENERIC :: sortie => sortie32, sortie64
  end type matrix
!
  type, extends(matrix) :: matrix_e(LN)
    integer, len :: LN = 30
    character(len=LN) :: nom
  contains
    PROCEDURE, PRIVATE :: sortie32 => sortie32e
    PROCEDURE, PRIVATE :: sortie64 => sortie64e
  end type matrix_e
```

Exemple (suite)

```
contains
  subroutine sortie32(this)
    class(matrix(K=REAL32, N=*, M=*)), intent(in) :: this
    print "(2(a,i0),a)", "ORDRE = [ ", this%N, ",", this%M, " ]"
    print *, "dernier élément => ", &
             this%a( ubound(this%a,1), ubound(this%a,2) )
  end subroutine sortie32
  !-------------------------
  subroutine sortie64(this)
    class(matrix(K=REAL64, N=*, M=*)), intent(in) :: this
    print "(2(a,i0),a)", "ORDRE = [ ", this%N, ",", this%M, " ]"
    print *, "dernier élément => ", &
             this%a( ubound(this%a,1), ubound(this%a,2) )
  end subroutine sortie64
  !-------------------------
  subroutine sortie32e(this)
    class(matrix_e(K=REAL32, N=*, M=*, LN=*)), intent(in) :: this
    call this%matrix%sortie
    print *, "nom = ", this%nom
  end subroutine sortie32e
  !---------------------------
  subroutine sortie64e( this )
    class(matrix_e(K=REAL64, N=*, M=*, LN=*)), intent(in) :: this
    call this%matrix%sortie
    print *, "nom = ", this%nom
  end subroutine sortie64e
```


Exemple (suite)

```
  real(kind=REAL32) function max32(this)
    class(matrix(K=REAL32, N=*, M=*)), intent(in) :: this

    max32 = MAXVAL(array=this%a)
  end function max32
  !-----------------
  real(kind=REAL64) function max64(this)
    class(matrix(K=REAL64, N=*, M=*)), intent(in) :: this

    max64 = MAXVAL(array=this%a)
  end function max64
end module mod
!
program prog
  use mod
  integer, parameter :: N1=10, M1=20, n2=20, M2=50
  type(matrix  (K=REAL32, N=N1, M=M1)) :: obj32
  type(matrix_e(K=REAL64, N=N2, M=M2)) :: obj64e
  real(kind=REAL32), dimension(N1,M1)  :: mat1
  real(kind=REAL64), dimension(N2,M2)  :: mat2

  mat1 = reshape( source=[ (i,i=1,N1*M1) ], shape=shape(mat1) )
  mat2 = reshape( source=[ (i,i=1,N2*M2) ], shape=shape(mat2) )
  obj32  = matrix  (K=REAL32, N=N1, M=M1)(a=mat1)
  obj64e = matrix_e(K=REAL64, N=N2, M=M2)(a=mat2, nom="matrice générale")
  call obj32%sortie; call obj64e%sortie
  print *, "MAX32 = ", obj32%max(), ", MAX64E = ", obj64e%max()
end program prog
```

Polymorphisme dynamique

Dans les exemples précédents, les objets à partir desquels les procédures *type_bound* sont appelées sont d'un **type fixé** : le compilateur sait alors quelle procédure appeler. Si l'on désire bénéficier du **polymorphisme dynamique**, on déclare l'objet p de l'exemple précédent comme argument muet polymorphique. Cela permet d'appeler la procédure affichage correspondant au type dynamique de l'objet qui est à l'origine de l'appel.

Exemple

```
program dynamic
  use pointext
  type(point2d) p2d
  type(point3d) p3d
        ...
  call affiche(p2d); call affiche(p3d)
contains
  subroutine affiche(p)
    CLASS(point2d), intent(in) :: p
    !------- Polymorphisme dynamique --------
    call p%affichage("Voici mes coordonnées")
  end subroutine affiche
end program dynamic
```

La fonction affiche s'appliquera alors à tout nouveau type étendu ultérieurement défini **sans avoir à la recompiler**.

Procédure attachée à un type non surchargeable

L'attribut NON_OVERRIDABLE à la déclaration d'une procédure *type-bound* permet d'interdire toute surcharge lors d'extensions éventuelles de ce type.

Exemple

```
module m
  type mycomplex
    real theta, rho
  contains
    PROCEDURE, PASS, NON_OVERRIDABLE :: real => real_part
    PROCEDURE, PASS, NON_OVERRIDABLE :: imag => imag_part
  end type
contains
  function real_part(a)
    class(mycomplex), intent(in) :: a
    real real_part
    real_part = a%rho*cos(a%theta)
  end function real_part
  function imag_part(a)
    class(mycomplex), intent(in) :: a
    real imag_part
    imag_part = a%rho*sin(a%theta)
  end function imag_part
end module m
```

Dans cet exemple les opérateurs « + » et « = » sont surchargés au sein des deux types « base » et « etendu » (extension du précédent). On remarque que les interfaces des fonctions de surcharge doivent être les mêmes hormis le *passed-object dummy argument*.

Exemple récapitulatif

```
module surcharge
  use, intrinsic :: ISO_FORTRAN_ENV, only : OUTPUT_UNIT
  implicit none
  type base
    real x
    contains
      procedure, private :: plus     => plus_base
      procedure, private :: egal     => egal_base
      procedure          :: affiche => affiche_base
      generic :: operator(+)    => plus
      generic :: assignment(=) => egal
  end type base
  type, extends(base) :: etendu
    integer i
    contains
      procedure, private :: plus     => plus_etendu
      procedure, private :: egal     => egal_etendu
      procedure          :: affiche => affiche_etendu
  end type etendu
```


Exemple récapitulatif (suite)

```
contains
  function plus_base(this, b)
    class(base), intent(in)  :: this, b
    class(base), allocatable :: plus_base

    allocate( base :: plus_base ); plus_base%x = this%x + b%x
  end function plus_base
  function plus_etendu(this, b)
    class(etendu), intent(in) :: this        ! doit respecter l'interface
    class(base),   intent(in) :: b           ! de la fonction "plus_base"
    class(base), allocatable  :: plus_etendu ! hormis l'argument this.

    if ( SAME_TYPE_AS(b, this) ) then
      allocate( etendu :: plus_etendu )
    else
      allocate( base :: plus_etendu )
    end if
    select type (plus_etendu)
      type is (base)
        plus_etendu = this%base + b
      type is (etendu)
        plus_etendu%base = this%base + b
        select type (b)
          type is (etendu)
            plus_etendu%i = this%i + b%i
        end select
    end select
  end function plus_etendu
```

Exemple récapitulatif (suite)

```
  subroutine egal_base(this, b)
    class(base), intent(out) :: this
    class(base), intent(in)  :: b

    this%x = b%x
  end subroutine egal_base
  subroutine egal_etendu(this, b)
    class(etendu), intent(out) :: this
    class(base),   intent(in)  :: b

    this%base = b
    select type (b)
      type is (etendu)
        this%i = b%i
    end select
  end subroutine egal_etendu
  subroutine affiche_base(this)
    class(base), intent(in) :: this

    write(OUTPUT_UNIT, "(a, f0.3)") "X = ", this%x
  end subroutine affiche_base
  subroutine affiche_etendu(this)
    class(etendu), intent(in) :: this

    call this%base%affiche; write(OUTPUT_UNIT, "(a, i0)") "I = ", this%i
  end subroutine affiche_etendu
end module surcharge
```


Exemple récapitulatif (suite)

```
program prog
  use surcharge
  implicit none
  real, parameter :: pi = acos(-1.)
  type(etendu)    :: e1, e2, e3
  type(base)      :: b1, b2

  b1 = base  (x=pi)
  b2 = base  (x=pi/2.)
  e1 = etendu(x=pi/3., i=2)
  e2 = etendu(x=pi/6., i=3)

  e3 = e1 + e2
  call e3%affiche

  e3  = b1 + e2
  call e3%affiche

  e3  = e2 + b1
  call e3%affiche

  e3  = b1 + b2
  call e3%affiche

  b2  = e1 + e3
  call b2%affiche
end program prog
```

Type abstrait

C'est un type qui sert de base à de futures extensions. Parmi les procédures qui lui sont attachées certaines peuvent être déclarées avec l'attribut DEFERRED indiquant qu'elles seront définies au sein de types étendus. Pour celles-ci :

- il sera nécessaire de fournir un nom d'interface entre parenthèses à la suite du mot-clé PROCEDURE ;
- aucun nom de procédure n'est fourni, seule la présence d'un *binding-name* est nécessaire ;
- L'attribut DEFERRED doit être précisé si et seulement si un nom d'interface est fourni ;
- La définition d'un type abstrait s'effectue en précisant le mot-clé ABSTRACT ;
- Il n'est pas possible de déclarer des objets de ce type. Par contre, celui-ci peut être utilisé pour la déclaration de variables polymorphiques ;
- L'extension d'un tel type peut se faire au moyen d'un type étendu normal ou d'un nouveau type abstrait.

Exemple

```
module fig
  use, intrinsic :: ISO_FORTRAN_ENV, only : OUTPUT_UNIT
  type, abstract :: figure_geo
    character(len=10) type
    contains
      procedure :: aire
      procedure :: volume
      procedure(aff), deferred :: affichage_constituants
  end type figure_geo

  type, extends(figure_geo) :: point
    private
    double precision x, y
    contains
      procedure :: affichage_constituants => &
                                    affichage_constituants_point
  end type point

  type, extends(point) :: cercle
    private
    double precision rayon
    contains
      procedure :: aire                    => aire_cercle
      procedure :: affichage_constituants => affichage_constituants_cercle
  end type cercle
```

Exemple (suite)

```
  type, extends(cercle) :: cylindre
    private
    double precision hauteur
    contains
      procedure :: aire                    => aire_cylindre
      procedure :: volume                  => volume_cylindre
      procedure :: affichage_constituants => affichage_constituants_cylindre
  end type cylindre

  type, extends(cercle) :: sphere
    contains
      procedure :: aire   => aire_sphere
      procedure :: volume => volume_sphere
  end type sphere

  abstract interface
    subroutine aff(this)
      import figure_geo
      class(figure_geo), intent(in) :: this
    end subroutine aff
  end interface
```

Exemple (suite)

```
  interface point
    module procedure construct_point
  end interface point

  interface cercle
    module procedure construct_cercle
  end interface cercle

  interface cylindre
    module procedure construct_cylindre
  end interface cylindre

  interface sphere
    module procedure construct_sphere
  end interface sphere

contains ! Constructeurs

  function construct_point(x, y)
    double precision, intent(in) :: x, y
    type(point) :: construct_point

    construct_point%type = "POINT"
    construct_point%x = x; construct_point%y = y
  end function construct_point
```

Exemple (suite)

```
function construct_cercle(x, y, r)
  double precision, intent(in) :: x, y, r
  type(cercle) :: construct_cercle

  construct_cercle%point = point(x, y)
  construct_cercle%type = "CERCLE"
  construct_cercle%rayon = r
end function construct_cercle

function construct_cylindre(x, y, r, h)
  double precision, intent(in) :: x, y, r, h
  type(cylindre) :: construct_cylindre

  construct_cylindre%cercle = cercle(x, y, r)
  construct_cylindre%type = "CYLINDRE"
  construct_cylindre%hauteur = h
end function construct_cylindre

function construct_sphere(x, y, r)
  double precision, intent(in) :: x, y, r
  type(sphere) :: construct_sphere

  construct_sphere%cercle = cercle(x, y, r)
  construct_sphere%type = "SPHERE"
end function construct_cercle
```

Exemple (suite)

```
function aire(this)
  class(figure_geo), intent(in) :: this
  double precision              :: aire
  aire = 0.d0
end function aire

function volume(this)
  class(figure_geo), intent(in) :: this
  double precision              :: volume
  volume = 0.d0
end function volume

function aire_cercle(this)
  class(cercle), intent(in) :: this
  double precision          :: aire_cercle
  aire_cercle = acos(-1.d0)*this%rayon*this%rayon
end function aire_cercle

function aire_cylindre(this)
  class(cylindre), intent(in) :: this
  double precision            :: aire_cylindre
  aire_cylindre = 2.d0*this%cercle%aire() + &
                  2.d0*acos(-1.d0)*this%rayon*this%hauteur
end function aire_cylindre
```

Exemple (suite)

```
  function aire_sphere(this)
    class(sphere), intent(in) :: this
    double precision          :: aire_sphere
    aire_sphere = 4.d0*this%cercle%aire()
  end function aire_sphere

  function volume_cylindre(this)
    class(cylindre), intent(in) :: this
    double precision            :: volume_cylindre

    volume_cylindre = this%cercle%aire()*this%hauteur
  end function volume_cylindre

  function volume_sphere(this)
    class(sphere), intent(in) :: this
    double precision          :: volume_sphere
    double precision          :: rayon

    rayon = this%cercle%rayon
    volume_sphere = 4.d0/3.d0*acos(-1.d0)*rayon*rayon*rayon
  end function volume_sphere
```

Exemple (suite)

```
  subroutine affichage_constituants_point(this)
    class(point), intent(in) :: this
    write(OUTPUT_UNIT, '(2a,/,3(:,a,f0.2))')  &
                    "===> ", trim(this%type), &
                    "Coordonnees : [ ", this%x, ",", this%y, " ]"
  end subroutine affichage_constituants_point

  subroutine affichage_constituants_cercle(this)
    class(cercle), intent(in) :: this
    call this%point%affichage_constituants
    write(OUTPUT_UNIT, "(a, f10.2)") "Rayon   : ", this%rayon
  end subroutine affichage_constituants_cercle

  subroutine affichage_constituants_cylindre(this)
    class(cylindre), intent(in) :: this
    call this%cercle%affichage_constituants
    write(OUTPUT_UNIT, "(a, f10.2)") "Hauteur : ", this%hauteur
  end subroutine affichage_constituants_cylindre
end module fig
```

Remarque :

- les constructeurs ont été redéfinis car ceux par défaut ne peuvent être appelés pour valoriser des composantes qui sont privées.

Exemple (suite)

```
program prog
  use fig
  type tab_ptr
    class(figure_geo), allocatable :: figure
  end type tab_ptr
  type(point) p ; type(cercle) c ; type(cylindre) cyl ; type(sphere) sph

  p   = point   ( x=1.d0, y=5.d0 )
  c   = cercle  ( x=0.d0, y=5.d0, r=10.d0 )
  cyl = cylindre( x=0.d0, y=5.d0, r=10.d0, h=100.d0 )
  sph = sphere  ( x=0.d0, y=5.d0, r=10.d0 )

  call tableau
contains
  subroutine tableau
    type(tab_ptr), dimension(4) :: tab_figures
    integer n

    allocate( tab_figures(1)%figure, source=p )
    allocate( tab_figures(2)%figure, source=c )
    allocate( tab_figures(3)%figure, source=cyl )
    allocate( tab_figures(4)%figure, source=sph )
    do n=1,size(tab_figures,1)
      call sorties_figure(tab_figures(n)%figure)
      deallocate(tab_figures(n)%figure)
    end do
  end subroutine tableau
```


Exemple (suite)

```
  subroutine sorties_figure(f)
    use, intrinsic :: ISO_FORTRAN_ENV, only : OUTPUT_UNIT
    CLASS(figure_geo), intent(in) :: f

    call f%affichage_constituants
    write(OUTPUT_UNIT, "(a, f10.2)") "Aire   : ", f%aire()
    write(OUTPUT_UNIT, "(a, f10.2)") "Volume : ", f%volume()
  end subroutine sorties_figure
end program prog
```

Reprise de l'exemple précédent en séparant la partie interface de la partie implémentation grâce à la notion de *submodule* définie par la norme 2008.
Le source de l'exemple est réparti en plusieurs fichiers ; de plus un fichier `Makefile` est proposé : celui-ci fait référence à l'utilitaire `makedepf90` permettant de générer les dépendances induites par les modules `Fortran`.

base.f90

```
module base
  type, abstract :: figure_geo
    character(len=10) type
    contains
      procedure :: aire
      procedure :: volume
      procedure(aff), deferred :: affichage_constituants
  end type figure_geo

  abstract interface
    subroutine aff(this)
      import figure_geo
      class(figure_geo), intent(in) :: this
    end subroutine aff
  end interface
```

base.f90 (suite)

```
  interface
    module function aire(this)
      class(figure_geo), intent(in) :: this
      double precision              :: aire
    end function aire

    module function volume(this)
      class(figure_geo), intent(in) :: this
      double precision              :: volume
    end function volume
  end interface
end module base
```

sbase.f90

```
submodule(base) sbase
contains
  module function aire(this)
    class(figure_geo), intent(in) :: this
    double precision              :: aire

    aire = 0.d0
  end function aire

  module function volume(this)
    class(figure_geo), intent(in) :: this
    double precision              :: volume

    volume = 0.d0
  end function volume
end submodule sbase
```

point.f90

```
module mpoint
  use base
  type, extends(figure_geo) :: point
    private
    double precision x, y
    contains
      procedure :: affichage_constituants => affichage_constituants_point
  end type point

  interface point
    module function construct_point(x, y)
      double precision, intent(in) :: x, y
      type(point) :: construct_point
    end function construct_point
  end interface point

  interface
    module subroutine affichage_constituants_point(this)
      class(point), intent(in) :: this
    end subroutine affichage_constituants_point
  end interface
end module mpoint
```

spoint.f90

```
submodule(mpoint) spoint
  use, intrinsic :: ISO_FORTRAN_ENV, only : OUTPUT_UNIT
contains
  module function construct_point(x, y)
    double precision, intent(in) :: x, y
    type(point) :: construct_point

    construct_point%type = "POINT"
    construct_point%x = x; construct_point%y = y
  end function construct_point

  module subroutine affichage_constituants_point(this)
    class(point), intent(in) :: this

    write(OUTPUT_UNIT, '(2a,/,3(:,a,f0.2))') &
                    "===> ", trim(this%type), &
                    "Coordonnees : [ ", this%x, ",", this%y, " ]"
  end subroutine affichage_constituants_point
end submodule spoint
```

cercle.f90

```
module mcercle
  use mpoint
  type, extends(point) :: cercle
    private
    double precision :: rayon
    contains
      procedure :: get_rayon
      procedure :: aire                    => aire_cercle
      procedure :: affichage_constituants => affichage_constituants_cercle
  end type cercle

  interface cercle
    module function construct_cercle(x, y, r)
      double precision, intent(in) :: x, y, r
      type(cercle) :: construct_cercle
    end function construct_cercle
  end interface cercle
```

cercle.f90 (suite)

```
  interface
    module function get_rayon(this)
      class(cercle), intent(in) :: this
    end function get_rayon

    module function aire_cercle(this)
      class(cercle), intent(in) :: this
      double precision          :: aire_cercle
    end function aire_cercle

    module subroutine affichage_constituants_cercle(this)
      class(cercle), intent(in) :: this
    end subroutine affichage_constituants_cercle
  end interface
end module mcercle
```

scercle.f90

```
submodule(mcercle) scercle
  use, intrinsic :: ISO_FORTRAN_ENV, only : OUTPUT_UNIT
contains

  module function construct_cercle(x, y, r)
    double precision, intent(in) :: x, y, r
    type(cercle) :: construct_cercle

    construct_cercle%point = point(x, y)
    construct_cercle%type = "CERCLE"
    construct_cercle%rayon = r
  end function construct_cercle

  module function get_rayon(this)
    class(cercle), intent(in) :: this

    get_rayon = this%rayon
  end function get_rayon
```

scercle.f90 (suite)

```
  module function aire_cercle(this)
    class(cercle), intent(in) :: this
    double precision          :: aire_cercle

    aire_cercle = acos(-1.d0)*this%rayon*this%rayon
  end function aire_cercle

  module subroutine affichage_constituants_cercle(this)
    class(cercle), intent(in) :: this

    call this%point%affichage_constituants
    write(OUTPUT_UNIT, '(a, f10.2)') "Rayon       : ", this%rayon
  end subroutine affichage_constituants_cercle
end submodule scercle
```

cylindre.f90

```
module mcylindre
  use mcercle
  type, extends(cercle) :: cylindre
    private
    double precision hauteur
    contains
      procedure :: aire                 => aire_cylindre
      procedure :: volume               => volume_cylindre
      procedure :: affichage_constituants => affichage_constituants_cylindre
  end type cylindre

  interface cylindre
    module function construct_cylindre(x, y, r, h)
      double precision, intent(in) :: x, y, r, h
      type(cylindre) :: construct_cylindre
    end function construct_cylindre
  end interface cylindre
```

cylindre.f90 (suite)

```
  interface
    module function aire_cylindre(this)
      class(cylindre), intent(in) :: this
      double precision            :: aire_cylindre
    end function aire_cylindre

    module function volume_cylindre(this)
      class(cylindre), intent(in) :: this
      double precision            :: volume_cylindre
    end function volume_cylindre

    module subroutine affichage_constituants_cylindre(this)
      class(cylindre), intent(in) :: this
    end subroutine affichage_constituants_cylindre
  end interface
end module mcylindre
```

scylindre.f90

```
submodule(mcylindre) scylindre
  use, intrinsic :: ISO_FORTRAN_ENV, only : OUTPUT_UNIT
contains
  module function construct_cylindre(x, y, r, h)
    double precision, intent(in) :: x, y, r, h
    type(cylindre) :: construct_cylindre

    construct_cylindre%cercle = cercle(x, y, r)
    construct_cylindre%type = "CYLINDRE"
    construct_cylindre%hauteur = h
  end function construct_cylindre

  module function aire_cylindre(this)
    class(cylindre), intent(in) :: this
    double precision            :: aire_cylindre

    aire_cylindre = 2.d0*this%cercle%aire() + &
                    2.d0*acos(-1.d0)*this%get_rayon()*this%hauteur
  end function aire_cylindre
```

scylindre.f90 (suite)

```
  module function volume_cylindre(this)
    class(cylindre), intent(in) :: this
    double precision            :: volume_cylindre

    volume_cylindre = this%cercle%aire()*this%hauteur
  end function volume_cylindre

  module subroutine affichage_constituants_cylindre(this)
    class(cylindre), intent(in) :: this

    call this%cercle%affichage_constituants
    write(OUTPUT_UNIT, '(a, f10.2)') "Hauteur     : ", this%hauteur
  end subroutine affichage_constituants_cylindre
end submodule scylindre
```

sphere.f90

```
module msphere
  use mcercle
  type, extends(cercle) :: sphere
    contains
      procedure :: aire   => aire_sphere
      procedure :: volume => volume_sphere
  end type sphere

  interface sphere
    module function construct_sphere(x, y, r)
      double precision, intent(in) :: x, y, r
      type(sphere) :: construct_sphere
    end function construct_sphere
  end interface sphere

  interface
    module function aire_sphere(this)
      class(sphere), intent(in) :: this
      double precision          :: aire_sphere
    end function aire_sphere

    module function volume_sphere(this)
      class(sphere), intent(in) :: this
      double precision          :: volume_sphere
    end function volume_sphere
  end interface
end module msphere
```

ssphere.f90

```
submodule(msphere) ssphere
contains

  module function construct_sphere(x, y, r)
    double precision, intent(in) :: x, y, r
    type(sphere) :: construct_sphere

    construct_sphere%cercle = cercle(x, y, r)
    construct_sphere%type = "SPHERE"
  end function construct_sphere

  module function aire_sphere(this)
    class(sphere), intent(in) :: this
    double precision          :: aire_sphere

    aire_sphere = 4.d0*this%cercle%aire()
  end function aire_sphere

  module function volume_sphere(this)
    class(sphere), intent(in) :: this
    double precision          :: volume_sphere
    double precision          :: rayon

    rayon = this%get_rayon()
    volume_sphere = 4.d0/3.d0*acos(-1.d0)*rayon*rayon*rayon
  end function volume_sphere
end submodule ssphere
```


prog.f90

```
program prog
  use, intrinsic :: ISO_FORTRAN_ENV, only : OUTPUT_UNIT
  use base
  use mpoint
  use mcercle
  use mcylindre
  use msphere

  type tab_ptr
    class(figure_geo), allocatable :: figure
  end type tab_ptr

  type(point)    p
  type(cercle)   c
  type(cylindre) cyl
  type(sphere)   sph

  p   = point   ( x=1.d0, y=5.d0 )
  c   = cercle  ( x=0.d0, y=5.d0, r=10.d0 )
  cyl = cylindre( x=0.d0, y=5.d0, r=10.d0, h=100.d0 )
  sph = sphere  ( x=0.d0, y=5.d0, r=10.d0 )

  call tableau
```

prog.f90 (suite)

```
contains
  subroutine tableau
    type(tab_ptr), dimension(4) :: tab_figures
    integer n

    allocate( tab_figures(1)%figure, source=p )
    allocate( tab_figures(2)%figure, source=c )
    allocate( tab_figures(3)%figure, source=cyl )
    allocate( tab_figures(4)%figure, source=sph )
    do n=1,size(tab_figures,1)
      call sorties_figure(tab_figures(n)%figure)
      deallocate(tab_figures(n)%figure)
    end do
  end subroutine tableau

  subroutine sorties_figure(f)
    CLASS(figure_geo), intent(in) :: f

    call f%affichage_constituants
    write(OUTPUT_UNIT, '(a, f10.2)') "Aire         : ", f%aire()
    write(OUTPUT_UNIT, '(a, f10.2)') "Volume       : ", f%volume()
  end subroutine sorties_figure
end program prog
```

Makefile

```
SRCS=$(wildcard *.f90)
FOBJ=${SRCS:.f90=.o}
# FC = the compiler to use
FC=ifort
#FC=gfortran

# Compiler options
FFLAGS=-stand f08
#FFLAGS=-std=f2008 -O3 -g  -frounding-math -m64

# List libraries used by the program here
LIBS=

# Suffix-rules:  Begin by throwing away all old suffix-
# rules, and then create new ones for compiling
# *.f90-files.
.SUFFIXES:
.SUFFIXES: .f90 .o

#all : $(FOBJ)

.f90.o:
        $(FC) -c $(FFLAGS) $<
```

Makefile (suite)

```
# Include the dependency-list created by makedepf90 below
include .depend

# target 'clean' for deleting object- *.mod- and other
# unwanted files
clean:
        rm -f *.o *.mod *.smod core .depend

# Create a dependency list using makedepf90.  All files
# that needs to be compiled to build the program,
# i.e all source files except include files, should
# be given on the command line to makedepf90.
#
# The argument to the '-o' option will be the name of the
# resulting program when running 'make'.
depend .depend:
        makedepf90 -o prog *.f90 > .depend
```

Divers

Traitement personnalisé des objets de type dérivé

Au-delà du traitement standard, il devient possible de contrôler les opérations d'entrée-sortie portant sur un objet de type dérivé (public ou semi-privé) via une ou plusieurs procédures de type SUBROUTINE.
Il existe 4 catégories de procédures (lecture/écriture et avec/sans format) qui peuvent être attachées de façon générique (*generic bindings*) au type dérivé de l'objet via des instructions du type :

```
GENERIC :: READ(FORMATTED)    => lecture_format1, lecture_format2
GENERIC :: READ(UNFORMATTED)  => lecture_nonformat1, lecture_nonformat2
GENERIC :: WRITE(FORMATTED)   => ecriture_format1, ecriture_format2
GENERIC :: WRITE(UNFORMATTED) => ecriture_nonformat1, ecriture_nonformat2
```

insérées au sein de la définition du type comme nous le verrons dans les exemples plus loin.
À droite des flèches, on trouve le nom des procédures qui seront discriminées à la compilation en fonction du type dérivé de l'objet traité et de la valeur effective des sous-types (paramètre KIND) de ses composantes (paramétrables à la déclaration).

Une alternative à l'attachement générique précédent est de regrouper l'ensemble des différentes procédures (concernant éventuellement plusieurs types dérivés) au sein d'un bloc interface :

```
INTERFACE READ(FORMATTED)
  MODULE PROCEDURE lecture_format1, lecture_format2, ...
END INTERFACE READ(FORMATTED)
```

Concernant les entrées-sorties **avec format**, un nouveau descripteur de format DT a été défini. Il s'applique à un objet de type dérivé de la liste d'entrée-sortie et a pour forme :

```
DT ['chaîne de caractères'] [(liste d'entiers)]
```

La chaîne de caractères et le tableau d'entiers, qui peuvent apparaître avec ce descripteur, sont automatiquement transmis en argument d'entrée de la procédure appelée. Ils sont à la libre disposition du programmeur pour paramétrer le traitement.

Note :

- toutes ces notions sont repérables dans la documentation anglaise sous les mots-clés *dtio-generic-spec* et *dtv-type-spec*.

Ce type de procédure doit respecter le prototype suivant :

```
SUBROUTINE lecture_format (dtv, unite, iotype, v_list, etat, msg )
SUBROUTINE ecrit_nonformat(dtv, unite,                 etat, msg )
```

dtv : objet de type dérivé qui est à l'origine de l'appel (discriminant en cas d'attachement générique de plusieurs procédures),

unite : numéro de l'unité logique sur laquelle a été connecté le fichier (0 pour un fichier interne),

iotype : chaîne de caractères indiquée au niveau du descripteur de format DT,

v_list : tableau d'entiers indiqué au niveau du descripteur de format DT,

etat : reflète en retour l'état de l'entrée-sortie,

msg : contient en retour le texte d'un message d'erreur si etat est non nul.

Exemple

```
module couleur_mod
  type couleur
    character(len=:), allocatable :: nom
    integer, dimension(3)         :: code_rvb
    contains
      procedure :: ecr_format
      generic :: write(formatted) => ecr_format ! <=== generic binding.
  end type couleur
contains
  subroutine ecr_format(dtv, unite, iotype, v_list, etat, msg)
    class(couleur),         intent(in)    :: dtv
    integer,                intent(in)    :: unite
    character(len=*),       intent(in)    :: iotype
    integer, dimension(:), intent(in)    :: v_list
    integer,                intent(out)   :: etat
    character(len=*),       intent(inout) :: msg
    character(len=13), dimension(4)       :: formats

    formats(1) = '(A, *(1X,I0))'
    if (size(v_list) > 0) then
      write(formats(2), "(A, I2.2, A)") "(A,3I", v_list(1), ")"
      write(formats(3), "(A, I2.2, A)")   "(3I", v_list(1), ")"
    end if
    formats(4) = MERGE(TSOURCE=formats(1), &
                       FSOURCE=MERGE(TSOURCE=formats(2), &
                                     FSOURCE=formats(3), &
                                     MASK=iotype=='DTCOMPLET'), &
                       MASK=size(v_list)==0)
    if (iotype == 'DTCOMPLET') write(unit=unite, fmt=formats(4)) dtv%nom, dtv%code_rvb
    if (iotype /= 'DTCOMPLET') write(unit=unite, fmt=formats(4)) dtv%code_rvb
  end subroutine ecr_format
end module couleur_mod
```


Exemple (suite)

```
PROGRAM exemple
  USE couleur_mod
  IMPLICIT NONE
  TYPE(couleur)       :: c1, c2
  REAL                :: x, y
  INTEGER             :: i, ios
  CHARACTER(len=132)  :: message
  NAMELIST/liste/ x, y, c1

  c1 = couleur(nom="gris_souris",     code_rvb=[158, 158, 158])
  c2 = couleur(nom="gris_anthracite", code_rvb=[48, 48, 48])
  i = 1756; x = 9.81; y = 2.718

  WRITE(UNIT=1, FMT="(I0, /, DT, /, 2F6.2, /)", &
        IOSTAT=ios, IOMSG=message) i, c1, x, y
  if(ios /= 0) PRINT *, ios, trim(message)
  WRITE(UNIT=1, FMT="(I0, /, DT 'COMPLET'(6), /, 2F6.2, /)", &
        IOSTAT=ios, IOMSG=message) i, c1, x, y
  if(ios /= 0) PRINT *, ios, trim(message)
  WRITE(UNIT=1, FMT="(F7.3, /, DT(7), /, DT 'COMPLET'(5), /)", &
       IOSTAT=ios, IOMSG=message) x, c1, c2
  if(ios /= 0) PRINT *, ios, trim(message)
  WRITE(UNIT=1, FMT=*, IOSTAT=ios, IOMSG=message) x, c1, i, y
  if(ios /= 0) PRINT *, ios, trim(message)
  WRITE(UNIT=1, NML=liste)
  if(ios /= 0) PRINT *, ios, trim(message)
END PROGRAM exemple
```

Exemple de traitement d'un fichier binaire

```
MODULE couleur_mod
  TYPE couleur
    CHARACTER(len=25)      :: nom
    INTEGER, DIMENSION(3) :: code_rvb
    CONTAINS
      PROCEDURE :: lec_binaire
      ! === Generic binding ===
      GENERIC   :: READ(UNFORMATTED) => lec_binaire
  END TYPE couleur
CONTAINS
  SUBROUTINE lec_binaire(dtv, unite, etat, msg)
    CLASS(couleur),    INTENT(inout) :: dtv
    INTEGER,           INTENT(in)    :: unite
    INTEGER,           INTENT(out)   :: etat
    CHARACTER(len=*), INTENT(inout) :: msg

    READ(UNIT=unite, IOSTAT=etat, IOMSG=msg) dtv%nom, &
                                             dtv%code_rvb
  END SUBROUTINE lec_binaire
END MODULE couleur_mod
```

Exemple de traitement d'un fichier binaire (suite)

```
PROGRAM exemple
  USE couleur_mod
  IMPLICIT NONE

  TYPE(couleur)       :: c
  REAL                :: x, y
  INTEGER             :: i, ios
  CHARACTER(len=132) :: message
    ...
  READ(UNIT=1, IOSTAT=ios, IOMSG=message) i, c, x, y
  DO WHILE ( ios == 0 )
          ...
    READ(UNIT=1, IOSTAT=ios, IOMSG=message) i, c, x, y
  END DO
  IF (ios > 0) THEN
    PRINT *, trim(message)
    STOP 4
  END IF
END PROGRAM exemple
```

Remarque : cette nouveauté permet de bénéficier du concept d'abstraction des données.

Exemple de traitement récursif

```
MODULE list_module
  TYPE node
    integer                :: value = 0
    type(node), pointer    :: next_node => null()
    CONTAINS
      PROCEDURE :: pwf
      GENERIC   :: WRITE(FORMATTED) => pwf ! <=== Generic binding.
  END TYPE node
CONTAINS
  RECURSIVE SUBROUTINE pwf(dtv, unite, iotype, v_list, etat, msg)
    CLASS(node),             INTENT(in)    :: dtv
    INTEGER,                 INTENT(in)    :: unite
    CHARACTER(len=*),        INTENT(in)    :: iotype
    INTEGER, DIMENSION(:),   INTENT(in)    :: v_list
    INTEGER,                 INTENT(out)   :: etat
    CHARACTER(len=*),        INTENT(inout) :: msg

    WRITE(UNIT=unite, FMT="(I9)", IOSTAT=etat) dtv%value
    if (etat /= 0) return
    if ( ASSOCIATED(dtv%next_node) ) &
      WRITE(UNIT=unite, FMT="(/,dt)", IOSTAT=etat) dtv%next_node
  END SUBROUTINE pwf
END MODULE list_module
```


Ce module pourra s'employer de la façon suivante :

Exemple de traitement récursif (suite)

```
PROGRAM liste
  USE liste_module
  INTEGER     :: unite, etat
  TYPE(node) :: racine
        ...
  ! Création d'une liste chaînée avec racine comme noeud primaire.
        ...
  ! Impression de la liste.
  WRITE(UNIT=unite, FMT="(dt)", IOSTAT=etat) racine
END PROGRAM liste
```

Note : dans ce dernier exemple, une fin d'enregistrement est générée uniquement lors de l'écriture effectuée au sein du programme principal. Par contre, concernant les écritures de plus bas niveau (à savoir celles effectuées dans la procédure `pwf` rattachée au type `node`) une fin d'enregistrement devra être traîtée explicitement. Ce qui explique la présence du caractère « / » dans le format de l'instruction :

```
WRITE(UNIT=unite, FMT="(/,dt)", IOSTAT=etat) dtv%next_node
```

Si l'on désire préciser un objet de type dérivé au sein d'une liste d'entrées-sorties, il est obligatoire de définir une procédure d'entrées-sorties pour ce type dans le cas où il contient une composante avec l'attribut `allocatable` ou `pointer`. C'est également le cas si l'objet précisé est polymorphique.

Exemple incorrect

```
module m
  use ISO_FORTRAN_ENV, only : OUTPUT_UNIT
  type cel
    integer i; real r
    contains
      procedure :: affichage
  end type cel
contains
  subroutine affichage(this)
    class(cel), intent(in) :: this

    write(OUTPUT_UNIT, *) this
  end subroutine affichage
end module m
program p
  use m
  type(cel) :: c=cel(r=acos(-1.), i=1756)
  call c%affichage
end program p
```


Exemple correct

```
module m
  use ISO_FORTRAN_ENV, only : OUTPUT_UNIT
  type cel
    integer i ; real r
    contains
      procedure :: ecriture
      GENERIC   :: WRITE(FORMATTED) => ecriture ! <=== Generic binding.
      procedure :: affichage
  end type cel
contains
  subroutine ecriture(dtv, unite, iotype, v_list, etat, msg)
    class(cel),             intent(in)    :: dtv
    integer,                intent(in)    :: unite
    character(len=*),       intent(in)    :: iotype
    integer, dimension(:), intent(in)    :: v_list
    integer,                intent(out)   :: etat
    character(len=*),       intent(inout) :: msg

    write(unite, *) dtv%i, dtv%r
  end subroutine ecriture
  subroutine affichage(this)
    class(cel), intent(in) :: this

    write(OUTPUT_UNIT, *) this
  end subroutine affichage
end module m
program p
  use m
  type(cel) :: c=cel(r=acos(-1.), i=1756)

  call c%affichage
end program p
```

Interopérabilité entre entités C et Fortran

L'interopérabilité entre une entité Fortran et une entité C existe lorsqu'il est possible de les déclarer de part et d'autre de façon à ce qu'elles puissent être mise en relation. L'interopérabilité suppose évidemment que l'on ne manipule que des entités (variables, fonctions, concepts, ...) communes aux deux langages, ce qui impose un certain nombre de restrictions et de contraintes.

Pour faciliter le travail du programmeur et améliorer la portabilité de son code, la norme Fortran 2003 fournit un certain nombre de nouveaux éléments syntaxiques nécessaires pour faciliter la définition d'entités interopérables .

Le module intrinsèque ISO_C_BINDING contient des constantes symboliques qui permettent l'**interopérabilité** avec C pour un certain nombre d'entités que nous allons passer en revue . . .

Les types intrinsèques

Le tableau suivant présente des correspondances entre entités de type intrinsèques au moyen de constantes symboliques définies dans le module ISO_C_BINDING :

Type/sous-type en Fortran	Type correspondant en C
INTEGER(kind=C_INT)	int
INTEGER(kind=C_SHORT)	short int
INTEGER(kind=C_LONG)	long int
REAL(kind=C_FLOAT)	float
REAL(kind=C_DOUBLE)	double
COMPLEX(kind=C_FLOAT_COMPLEX)	float _Complex
LOGICAL(kind=C_BOOL)	_Bool
CHARACTER(kind=C_CHAR)	char
etc.	...

Note : une chaîne de caractères étant considérée en C comme un tableau de caractères (avec comme dernier élément le caractère NULL), elle est interopérable avec l'identificateur déclaré en Fortran sous la forme :

```
CHARACTER(kind=C_CHAR), dimension(*) :: chaine
```

D'une façon générale, une variable scalaire est interopérable si son type ainsi que les éventuels paramètres de ce type sont interopérables et que celle-ci n'admet ni l'attribut POINTER ni l'attribut ALLOCATABLE.

D'autres constantes symboliques permettent de faire référence à des caractères spéciaux comme :

Nom	Signification en C	Valeur ASCII	Équivalent C
C_NULL_CHAR	*null character*	achar(0)	\0
C_ALERT	*alert*	achar(7)	\a
C_BACKSPACE	*backspace*	achar(8)	\b
C_HORIZONTAL_TAB	*horizontal tab*	achar(9)	\t
C_NEW_LINE	*line feed/new line*	achar(10)	\n
C_VERTICAL_TAB	*vertical tab*	achar(11)	\v
C_FORM_FEED	*form feed*	achar(12)	\f
C_CARRIAGE_RETURN	*carriage return*	achar(13)	\r

Les tableaux

Un tableau Fortran est interopérable s'il est d'un type interopérable (ainsi que les éventuels paramètres de ce type) et de profil explicite ou de taille implicite.
De plus pour les tableaux multidimensionnés, l'ordre des indices doit être inversé.

Ainsi les tableaux Fortran :

```
integer(kind=C_INT), dimension(18,3:7,*)     ::   t1
integer(kind=C_INT), dimension(18,3:7,100)   ::   t2
```

sont interopérables avec les tableaux ainsi déclarés en C :

```
int t1[][5][18]
int t2[100][5][18]
```

Les variables globales

Une variable externe C peut interopérer avec un bloc COMMON ou avec une variable déclarée dans un module Fortran.

Exemple

```
module variables_C
  use, intrinsic :: ISO_C_BINDING
  integer(C_INT), bind(C)  :: c_extern
  integer(C_LONG)          :: fort_var
  BIND( C, NAME="C_var" )  :: fort_var
  common/COM/ r, s
  common/SINGLE/ t
  real(kind=C_FLOAT) :: r, s, t
  bind(C) :: /COM/, /SINGLE/
end module variables_C
```

Ces variables Fortran sont interopérables avec celles définies en C au niveau externe par :

```
int           c_extern;
long          C_var;
struct {
  float r, s;
} com;
float         single;
```

Remarques :

- une variable globale Fortran doit avoir été déclarée avec l'attribut BIND(C) pour pouvoir être mise en correspondance avec une variable externe C,
- si cet attribut a été spécifié sans le paramètre NAME, une référence externe est générée entièrement en minuscules,
- si le paramètre NAME a été précisé, sa valeur correspond au nom de la référence externe générée, en respectant les minuscules et/ou majuscules employées.

Exemple

```
module m
  use ISO_C_BINDING
  interface
    subroutine sp(mat1, mat2, n, chaine) bind(C)
      import C_FLOAT, C_DOUBLE, C_INT, C_CHAR
      real(kind=C_FLOAT),  dimension(3,6)   :: mat1
      real(kind=C_DOUBLE), dimension(5,*)   :: mat2
      integer(kind=C_INT)                   :: n
      character(kind=C_CHAR), dimension(*) :: chaine
    end subroutine sp
  end interface

  integer(kind=C_INT), bind(C) :: entier_externe
  real(kind=C_DOUBLE), bind(C) :: double_externe
end module m
```

Exemple (suite)

```
program prog
  use m
  use ISO_FORTRAN_ENV
  real(kind=C_FLOAT),  dimension(3,6)      :: m1
  real(kind=C_DOUBLE), dimension(5,7)      :: m2
  character(kind=C_CHAR,len=*), parameter :: chaine = &
             "Interoperabilite Fortran/C"//C_NULL_CHAR
  integer i

  entier_externe = 1756
  double_externe = acos(-1._C_DOUBLE)
  call random_number(m1); call random_number(m2)
  call sp(m1, m2, 7_C_INT, chaine)
  write(OUTPUT_UNIT, "(/, 2a, 'Matrices Fortran 2003',/)") &
               C_HORIZONTAL_TAB, C_HORIZONTAL_TAB
  do i=1, size(m1,1)
    write(OUTPUT_UNIT, "(6f9.6)") m1(i,:)
  end do
  print *
  do i=1, size(m2,1)
    write(OUTPUT_UNIT, "(7f9.6)") m2(i,:)
  end do
end program prog
```


Exemple (suite)

```
#include <stdio.h>
int entier_externe;
double double_externe;

void sp(float m1[6][3], double m2[][5], int *n, char *ch)
{
  int i, j;

  printf("\t\t%s\n\n", ch);
  printf("entier_externe = %d\n", entier_externe);
  printf("double_externe = %f\n", double_externe);
  printf("\t\tMatrices C\n\n");
  for(j=0; j<3; j++) {
    for(i=0; i<6; i++)
      printf("%f ", m1[i][j]);
    printf("\n");
  }
  printf("\n");
  for(j=0; j<5; j++) {
    for(i=0; i<*n; i++)
      printf("%f ", m2[i][j]);
    printf("\n");
  }
  return;
}
```

Types dérivés Fortran/structures de données C

Un objet de type C_struct ainsi défini en C :

```
typedef struct
{
  int m, n;
  float r;
} C_struct;
```

est interopérable avec une structure Fortran du type F_struct suivant :

```
use, intrinsic :: ISO_C_BINDING
type, BIND(C) :: F_struct
  integer(kind=C_INT) :: m, n
  real(kind=C_FLOAT)  :: r
end type F_struct
```

Note :

- un type est interopérable si toutes ses composantes le sont : les entités ALLOCATABLE en sont donc exclues, de même aucune procédure ne peut lui être attachée. Dans le cas contraire, les types C_PTR et C_FUNPTR peuvent être employés afin de considérer un objet de ce type comme une entité opaque. (cf. l'exemple 4 page 195).

Les pointeurs

Les pointeurs C, quels qu'ils soient, sont interopérables avec des pointeurs Fortran particuliers du type dérivé semi-privé C_PTR dont une composante privée contient l'adresse *cachée* d'une cible.

On retrouve là l'analogie avec le descripteur du pointeur Fortran qui est sous-jacent à l'attribut POINTER. Le pointeur en Fortran est un concept abstrait et puissant n'autorisant pas (fiabilité oblige) la manipulation arithmétique directe de l'adresse qui reste *cachée*.

La nécessité de définir les pointeurs de type C_PTR, souvent appelés *pointeurs C* par opposition aux pointeurs Fortran, se justifie en partie par le fait que contrairement à ces derniers ils ne peuvent/doivent pas désigner une zone mémoire non contiguë. De plus, le type C_PTR est utilisable dans un contexte d'interopérabilité avec tout type de pointeur C (typé ou non – void*).

Toutes les manipulations relatives aux *pointeurs C* se font via des opérateurs ou des procédures (*méthodes*), ainsi :

- `C_LOC(X)`

  cette fonction retourne un scalaire de type `C_PTR` renfermant l'adresse de son argument `X` au sens de l'opération unaire « & » du langage `C`. L'argument `X` doit avoir l'attribut `POINTER` ou `TARGET` et être soit :
  - une variable de type et paramètres de type interopérables ;
  - une variable scalaire non polymorphique sans *length type parameters*.

  de plus :
  - si la variable `X` admet l'attribut `ALLOCATABLE`, elle devra être allouée ;
  - si elle admet l'attribut `POINTER` elle devra être associée ;
  - si c'est un tableau, il devra être de taille non nulle.

```
use ISO_C_BINDING
real(C_FLOAT), dimension(10), target   ::   X
type(C_PTR)                            ::   buf

buf = C_LOC(X)
```

  Remarque : pour un argument non interopérable, cette fonction peut être employée pour transmettre à une fonction `C` l'adresse d'un objet opaque. Cette fonction n'aura alors d'autre choix que d'interpréter cette adresse comme étant de type *void ** : aucune indirection ne sera alors possible.

- `C_F_POINTER(CPTR, FPTR [,SHAPE])`

  cette procédure convertit `CPTR` de type `C_PTR` en un pointeur Fortran `FPTR` (`SHAPE` à spécifier si la cible est un tableau) ;

- `C_ASSOCIATED(C_PTR_1[, C_PTR_2])`

  cette procédure vérifie que deux *pointeurs C* (de type `C_PTR`) sont identiques ou que le premier est à l'état nul.

Notes :

- Ces entités permettent l'interopérabilité des tableaux dynamiques : un tableau Fortran alloué peut être adressé en `C` et un tableau alloué en `C` peut ensuite être associé à un pointeur Fortran.

Arguments de types intrinsèques

- attribut `VALUE` pour les arguments muets scalaires. L'argument d'appel correspondant n'est plus passé par référence (adresse), mais via une copie temporaire dans le *stack*. À noter que la copie en retour n'est pas faite, ce qui est exclusif de `intent(OUT/INOUT)` !
- attribut `BIND(C [,NAME=...])` obligatoire :
  - lors de la définition d'une procédure Fortran interopérable laquelle est appelée en `C`,
  - à la définition du bloc interface associé à la fonction `C` appelée depuis `Fortran`.

  Le paramètre `NAME` permet de donner un nom à la référence externe comme pour les variables globales déjà vues ;
- interface explicite et attribut `BIND(C)` obligatoire,
- arguments muets tous interopérables (non optionnels) et en cohérence avec ceux du prototype `C` ;
- une fonction Fortran doit retourner un scalaire interopérable et un sous-programme doit correspondre à un prototype `C` retournant le type `void` ;
- un argument du prototype `C` de type pointeur peut être associé à un argument muet Fortran classique **sans** l'attribut `VALUE` (cf. exemple suivant) ;
- un argument du prototype `C` qui n'est pas de type pointeur doit être associé à un argument muet **avec** l'attribut `VALUE` (cf. exemple suivant).

Dans un premier exemple, Fortran appelle une fonction `C` à laquelle il transmet deux arguments :

- un tableau passé classiquement par référence suivi d'une variable entière passée par valeur.

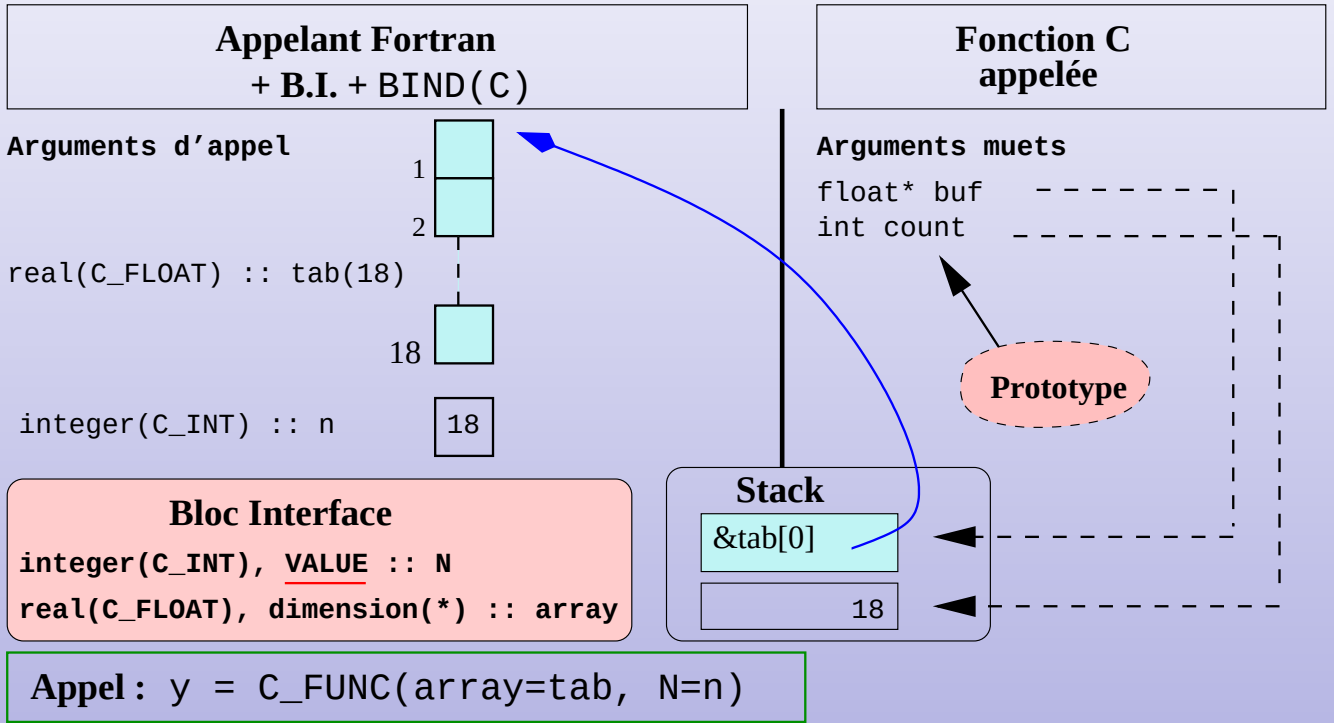
Exemple : fonction C appelée depuis Fortran

```
module m
  use, intrinsic :: ISO_C_BINDING
  interface
    function C_FUNC(array, N) BIND(C, NAME="C_Func")
      import C_INT, C_FLOAT
      implicit none
      real(kind=C_FLOAT)                 :: C_FUNC
      real(kind=C_FLOAT), dimension(*) :: array
      integer(kind=C_INT), VALUE         :: N
    end function C_FUNC
  end interface
end module m
```

Exemple : fonction C appelée depuis Fortran (suite)

```
program p
  use m
  integer(kind=C_INT), parameter :: n = 18
  real(C_FLOAT), dimension(n)    :: tab
  real(kind=C_FLOAT)             :: y

  call random_number(tab)
  y = C_FUNC(array=tab, N=n)
  print *, "Val. retournée par la fonction : ", y
end program p

float C_Func(float *buf, int count)
{
  float somme = 0. ;

  for(int i=0; i<count; i++) somme += buf[i] ;

  return somme ;
}
```

Exemple : fonction C appelée depuis Fortran



Dans ce deuxième exemple, C appelle une procédure Fortran à laquelle il transmet trois arguments :

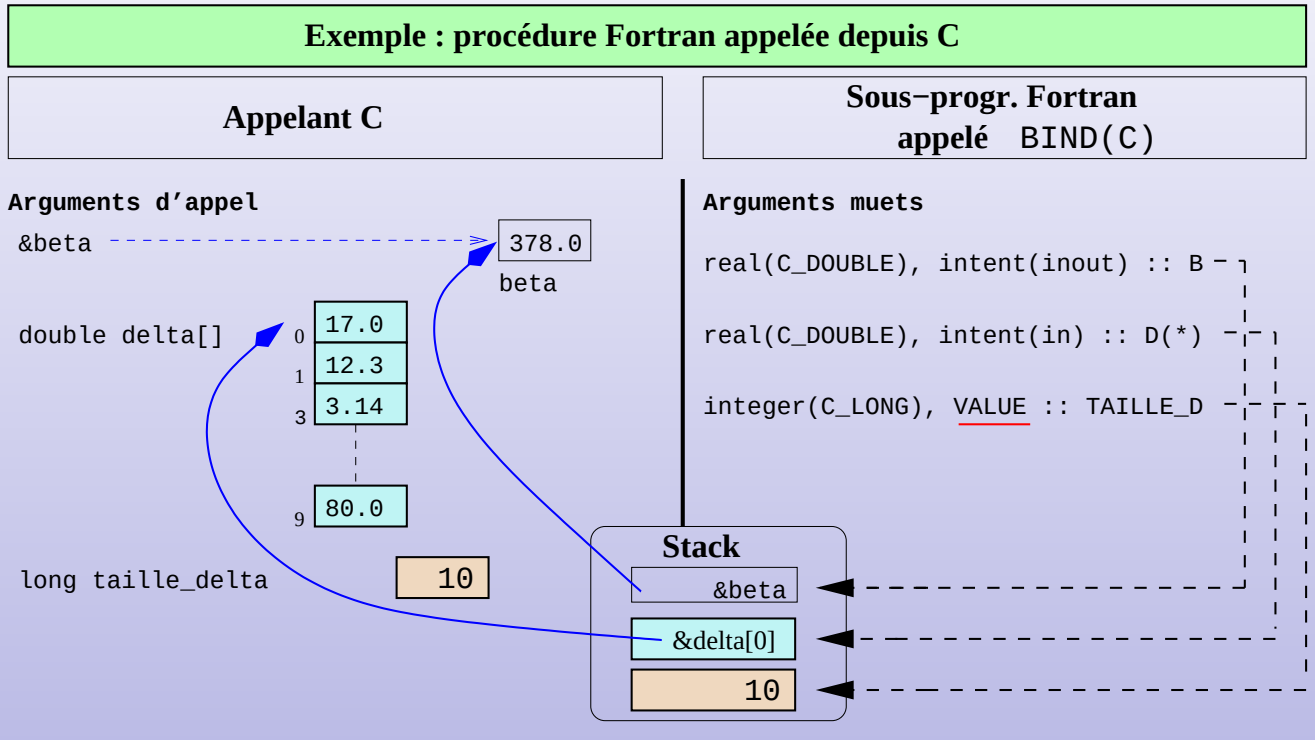
- une variable entière passée par valeur ;
- une variable réelle passée par référence ;
- un tableau à taille implicite passé par référence.

Exemple : procédure Fortran appelée depuis C

```c
void f1(double *b, double d[], long taille_d);
int main()
{
  double beta = 378.0 ;
  double delta[] = { 17.,   12.3,  3.14,  2.718,  0.56,
                     22.67, 25.8,  89.,   76.5,   80. } ;
  long   taille_delta = sizeof delta / sizeof delta[0] ;

  f1(&beta, delta, taille_delta) ;
  return 0;
}
```

```fortran
subroutine F1(B, D, TAILLE_D) BIND(C, NAME="f1")
  use, intrinsic :: ISO_C_BINDING
  implicit none
  real(C_DOUBLE), intent(inout)             :: B
  real(C_DOUBLE), dimension(*), intent(in)  :: D
  integer(kind=C_LONG), VALUE               :: TAILLE_D
  print *, "B=", B, "D(", TAILLE_D, ")=", D(TAILLE_D)
end subroutine F1
```


Exemple : procédure Fortran appelée depuis C



Appel : `f1(&beta, delta, taille_delta)`

Arguments de types pointeurs : le type C_PTR

- Fortran ⇒ C

Passage par valeur (VALUE)

- La fonction C récupère le contenu de la composante adresse encapsulée dans l'argument de type C_PTR nécessairement déjà associé ;
- intent(OUT/INOUT) interdit.

exemple 1 ci-après

Passage par référence

- La fonction C récupère l'adresse de la composante adresse encapsulée dans l'argument de type C_PTR **non** nécessairement associé ;

exemple 2 ci-après

- C ⇒ Fortran

Passage par valeur (VALUE)

- C passe à Fortran un pointeur déjà valorisé ;
- intent(OUT/INOUT) interdit.

exemple 4 ci-après

Passage par référence

- C doit passer l'adresse (&p) d'un pointeur qui pourra donc être associé dans la procédure Fortran.

exemple 3 et 4 ci-après

Exemple 1 : Fortran ⇒ C (argument C_PTR passé par valeur)

- dans cet exemple, Fortran alloue et valorise un tableau à deux dimensions. Son adresse traduite en un pointeur C à l'aide de la fonction C_LOC est transmise par valeur à une fonction C ;
- de plus la fonction C récupère les dimensions du tableau qui lui ont été passées par valeur.

Notes :

- le tableau Fortran déclaré avec l'attribut ALLOCATABLE n'est pas interopérable. Le type C_PTR employé ici permet de contourner ce problème ;
- on donne deux versions de la fonction C, l'une respectant la norme C89, l'autre la norme C99.

Exemple1 : Fortran ⇒ C (argument C_PTR passé par valeur)

```
program EXEMPLE_1
  use, intrinsic :: ISO_C_BINDING
  real(kind=C_FLOAT), dimension(:,:), allocatable, target :: mat
  integer(kind=C_INT)                                     :: n, m
  interface !---------------------------------------!
    subroutine c_func(n, m, v) bind(C, name="fct")
      import C_INT, C_PTR
      integer(kind=C_INT), VALUE :: n, m
      type(C_PTR), VALUE :: v
      ! ou (*) real(kind=C_FLOAT), dimension(*) :: v
    end subroutine c_func
  end interface !-------------------------------------!
  read *, n, m ; allocate(mat(n,m))
  call random_number(mat)
  call c_func(n, m, C_LOC(mat)) ! ou (*) call c_func(n, m, mat)
  print *, "SOMME = ", sum(array=mat, dim=1); deallocate(mat)
end program EXEMPLE_1
```

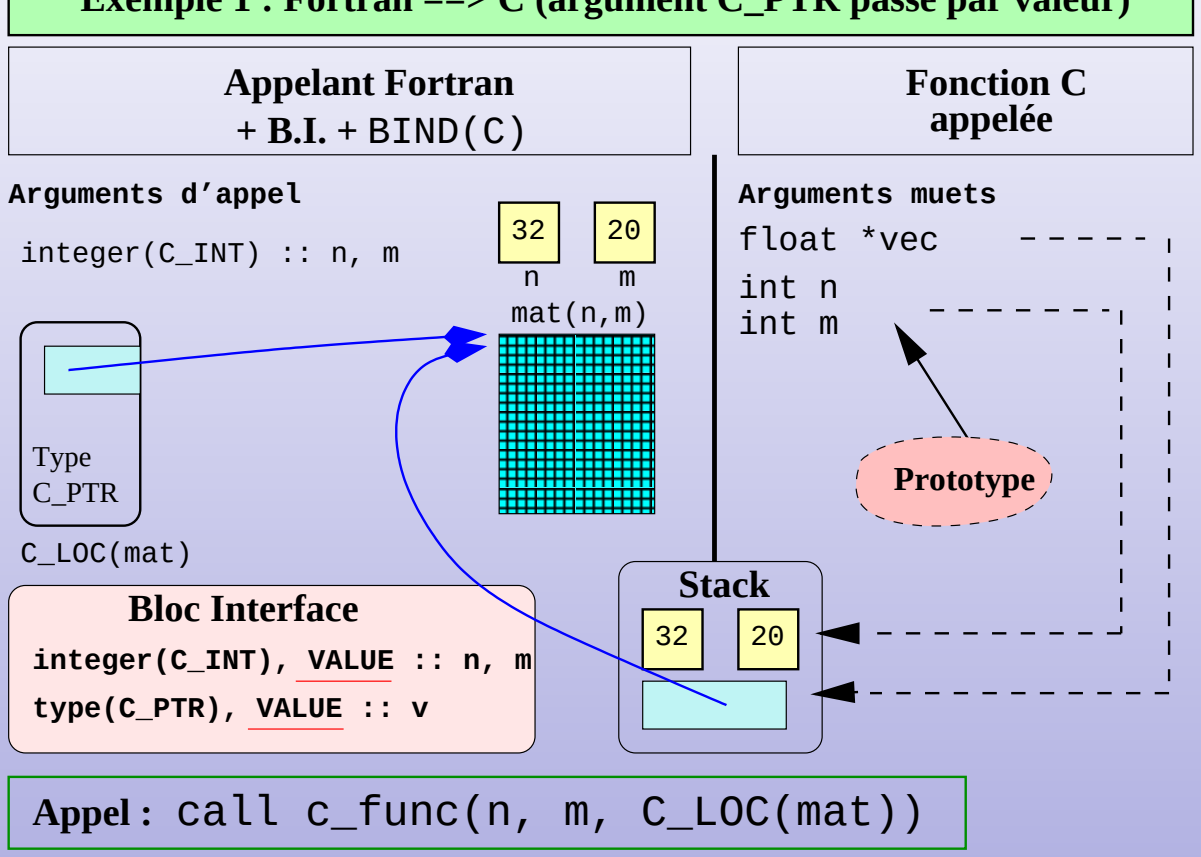
Version C89

```
void fct(int n, int m, float *vec)
{
  float **mat;
  int     i, j;
  mat = malloc(m*sizeof(float *));
  for(i=0, j=0; i<m; i++, j+=n)
    mat[i] = vec+j, mat[i][n-1] *= 2.;
  free( mat );
}
```

Version C99

```
void fct(int n, int m, float mat[m][n])
{
  for(int i=0; i<m; i++)
    mat[i][n-1] *= 2.;

  return;
}
```


Exemple 1 : Fortran ==> C (argument C_PTR passé par valeur)



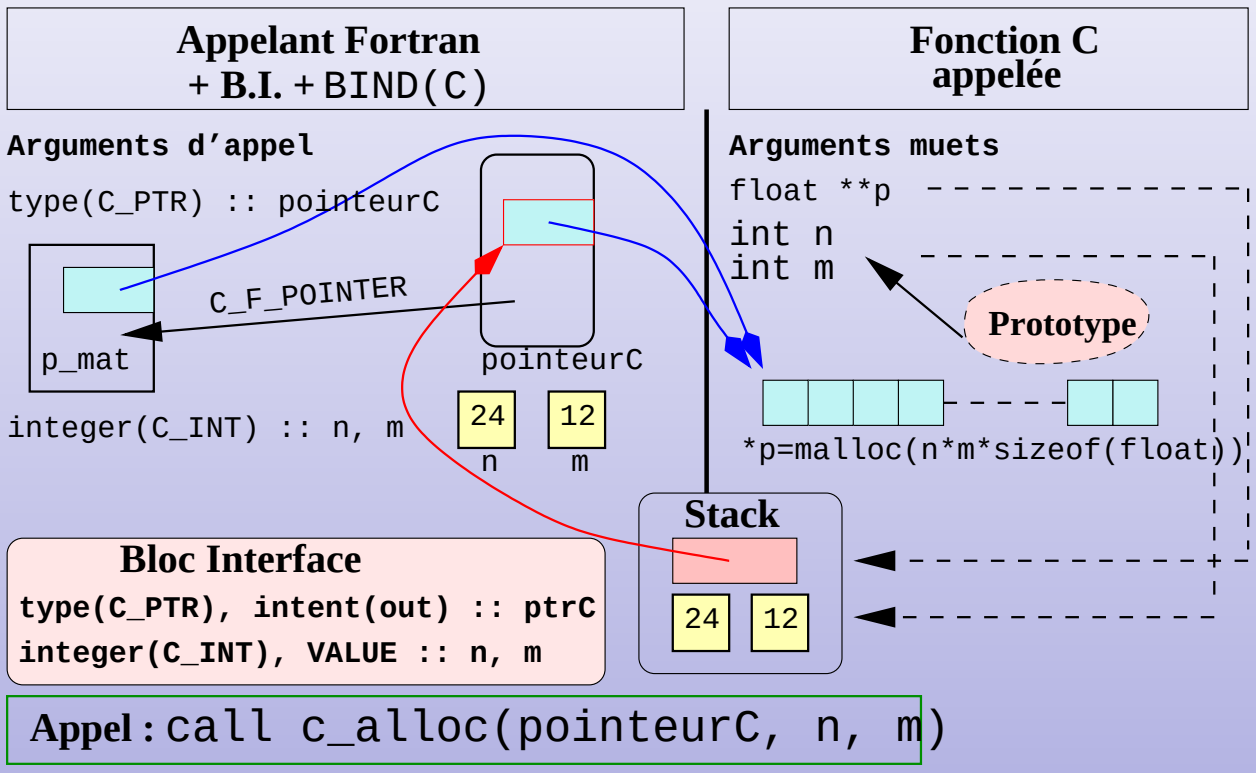
Exemple 2 : Fortran ⇒ C (argument C_PTR passé par référence)

- dans cet exemple, Fortran souhaite sous-traiter à une fonction C l'allocation d'une matrice n*m qu'il référencera ensuite via un pointeur Fortran ;
- à l'appel de la fonction C, Fortran passe par valeur les deux dimensions n et m désirées et passe par référence un pointeur interopérable non encore associé ;
- la fonction C appelée alloue une zone mémoire de n*m réels. Son adresse est stockée dans l'objet Fortran pointeurC de type C_PTR ;
- en retour de la fonction C, Fortran convertit l'objet pointeurC en un pointeur Fortran classique (via la procédure C_F_POINTER) qui devient ainsi associé à la matrice allouée en C ;
- ensuite cet objet pointeurC est transmis par valeur à une autre fonction C afin de libérer la zone allouée.

Exemple2 : Fortran ⇒ C (argument C_PTR passé par référence)

```
program exemple2
  use ISO_C_BINDING
  integer(kind=C_INT)                              :: n, m
  type(C_PTR)                                      :: pointeurC
  real(kind=C_FLOAT), dimension(:,:), pointer :: p_mat
  interface
    subroutine c_alloc(ptrC, n, m) bind(C, name="C_alloc")
      import C_PTR, C_INT
      type(C_PTR), intent(out) :: ptrC
      integer(kind=C_INT), VALUE :: n, m
    end subroutine c_alloc
    subroutine c_free(ptrC) bind(C, name="C_free")
      import C_PTR
      type(C_PTR), VALUE :: ptrC
    end subroutine c_free
  end interface
  read *, n, m ; call c_alloc(pointeurC, n, m)
  call C_F_POINTER(CPTR=pointeurC, FPTR=p_mat, shape=[ n, m ])
  call random_number(p_mat)
  print *, "SOMME = ", sum(array=p_mat, dim=1)
  call c_free(pointeurC)
end program exemple2

void C_alloc(float **p, int n, int m)
{ *p = malloc(n*m*sizeof(float)); return ; }

void C_free(float *p) { free(p); return ; }
```

Exemple 2 : Fortran ==> C (argument C_PTR passé par référence)



Exemple 3 : C ⇒ Fortran (argument C_PTR passé par référence)

- dans cet exemple, C souhaite sous-traiter à des sous-programmes Fortran la gestion (allocation, valorisation, traitement, libération) d'un vecteur de 100 réels ;
- à l'appel du sous-programme `for_alloc`, C passe en argument l'adresse `&vec` d'un pointeur de réels ;
- dans le sous-programme `for_alloc`, l'argument `pointeurC` muet correspondant est un pointeur de type `C_PTR` de vocation `INTENT(OUT)` sans l'attribut `VALUE` ;
- `Fortran` alloue un tableau de la taille requise dont l'adresse est retournée à C via l'argument de sortie `pointeurC` valorisé à l'aide de la fonction `C_LOC` ;
- en retour du sous-programme Fortran, C peut accèder à la zone dynamique par l'intermédiaire du pointeur `vec`.

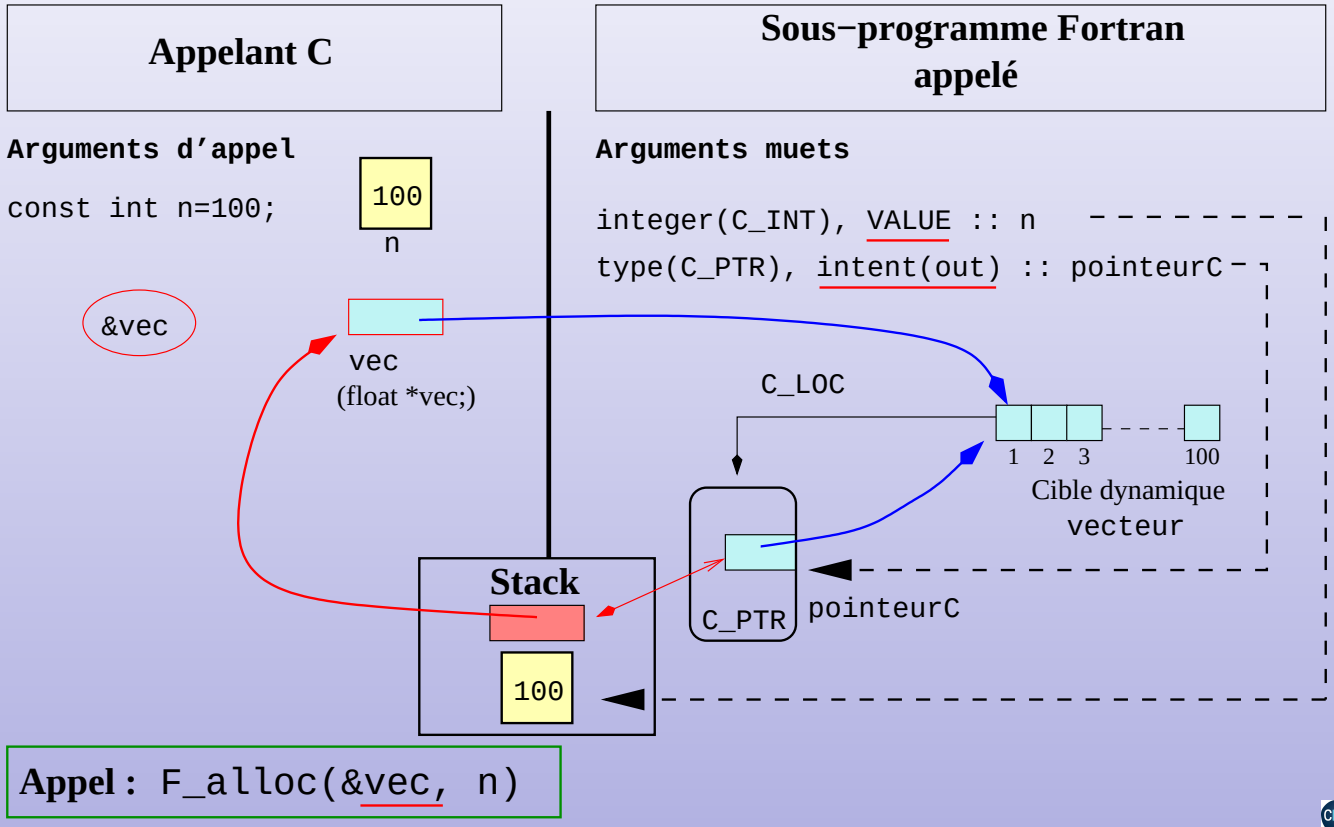
Exemple 3 : C ⇒ Fortran (argument C_PTR passé par référence)

```c
#include <stdio.h>

void F_alloc   (float **, int);
void F_moyenne (float * , int);
void F_free    (void);

int main()
{
  const int  n = 100;
  float     *vec;

  F_alloc(&vec, n);
  printf("vec[50] = %f\n", vec[50]);
  F_moyenne(vec, n);
  F_free();

  return 0;
}
```

Exemple 3 : C ⇒ Fortran (argument C_PTR passé par référence) (suite)

```fortran
module creer_liberer
  use ISO_C_BINDING
  real(kind=C_FLOAT), dimension(:), pointer :: vecteur
contains
  subroutine for_alloc(pointeurC, n) BIND(C, name="F_alloc")
    type(C_PTR), intent(out)   :: pointeurC
    integer(kind=C_INT), VALUE :: n
    allocate(vecteur(n))
    call random_number(vecteur)
    pointeurC = C_LOC(vecteur)
  end subroutine for_alloc
  subroutine for_free() BIND(C, name="F_free")
    if (associated(vecteur)) deallocate(vecteur)
  end subroutine for_free
end module creer_liberer

module calculs
  use ISO_C_BINDING
contains
  subroutine moyenne(pointeurC, n) BIND(C, name="F_moyenne")
    type(C_PTR),    VALUE                    :: pointeurC
    integer(C_INT), VALUE                    :: n
    real(kind=C_FLOAT), dimension(:), pointer :: p
    call C_F_POINTER(CPTR=pointeurC, FPTR=p, shape=[n])
    print *, "MOYENNE = ", sum(p)/n
  end subroutine moyenne
end module calculs
```

Exemple 3 : C ==> Fortran (argument C_PTR passé par référence)



Appel : `F_alloc(&vec, n)`

Notes sur l'exemple 3 ci-dessus :

- le tableau dynamique `vecteur` (`allocatable`) est déclaré comme entité globale dans le module `creer_liberer`. Dans le cas contraire (déclaré localement à la procédure `for_alloc`) il ne sera plus possible de le libérer au sein de la procédure `for_free` car la libération ne peut se faire qu'en mentionnant le nom du tableau ;
- l'entité `vecteur` pourrait être déclarée avec les attributs `allocatable, target` à la place de `pointer` ;
- si cette entité avait été déclarée localement à la procédure `for_alloc` avec l'attribut `pointer`, elle ne pourrait être libérée au sein de la procédure `for_free` qu'à l'aide d'un pointeur global lequel aurait été associé lors de l'allocation : pour la bonne raison que l'information indiquant qu'il s'agit d'une cible dynamique est stockée dans le descripteur de pointeur.
  Cependant le plus simple, dans le cas d'une déclaration locale, est d'allouer et de désallouer l'espace au sein de la même procédure.

Exemple 4 : C ⇒ Fortran (argument C_PTR passé par référence et par valeur)

```
module gestion_cellule
  use ISO_C_BINDING
  type pass
    integer n
    real, allocatable, dimension(:) :: a
        ...
  end type pass
  type(pass), pointer :: p_cel
contains
  subroutine init(data, n) BIND(C)
    type(C_PTR), intent(out) :: data
    integer(C_INT), VALUE    :: n
    allocate(p_cel)
    p_cel%n = n
    allocate(p_cel%a(n))
    data = C_LOC(p_cel)
  end subroutine init
  subroutine add(data, ... ) BIND(C)
    type(C_PTR), VALUE :: data
        . . .
    call C_F_POINTER(data, p_cel)
        . . .
  end subroutine add
end module gestion_cellule
```


Exemple 4 : C ⇒ Fortran (argument C_PTR passé par référence et par valeur)

```
int main()
{
  void *p, *q ;
      ...
  init(&p, 100) ;
  init(&q, 200) ;
      ...
  add(p, ... ) ;
  add(q, ... ) ;

  return 0;
}
```

Remarque :

- dans cet exemple, la fonction `C_LOC` est utilisée pour transmettre à la fonction `main` l'adresse d'un objet opaque. Celle-ci considère cette adresse comme étant de type `void *` : elle ne peut donc effectuer aucune indirection. Cette adresse est ensuite transmise à la procédure `Fortran`, `add` ;
- la fonction `main` gère de façon abstraite un travail sous la responsabilité de `Fortran`.

Arguments de types structure de données

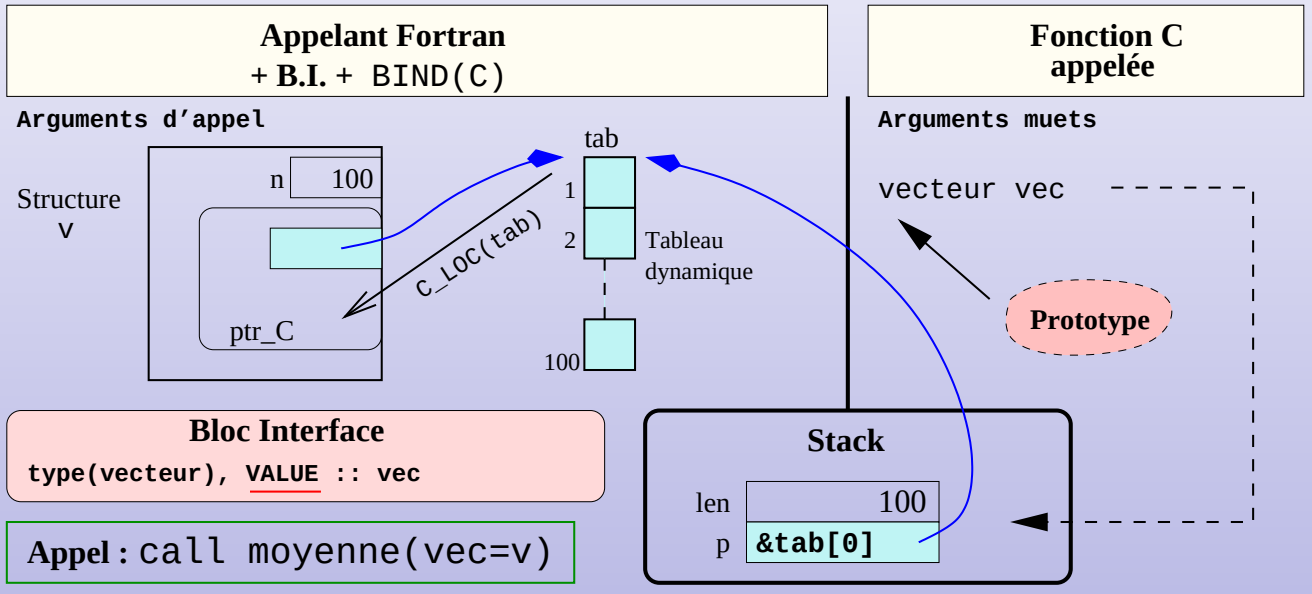
Exemple Fortran ⇒ C

```
module m
  use ISO_C_BINDING
  type, BIND(C) :: vecteur
    integer(kind=C_INT) :: n
    type(C_PTR)         :: pointeur_C
  end type vecteur
  interface  ! <---------- Bloc interface de la fonction C
    subroutine moyenne(vec) BIND(c, name="C_moyenne")
      import vecteur
      type(vecteur), VALUE :: vec
    end subroutine moyenne
  end interface
end module m
program p
  use m
  type(vecteur) :: v
  real(C_FLOAT), allocatable, dimension(:), target :: tab

  v%n = 100; allocate(tab(v%n)); call random_number(tab)
  v%pointeur_C = C_LOC(tab);
  call moyenne(vec=v)  ! <---Appel fonction C
  deallocate(tab)
end program p
```


Exemple Fortran ⇒ C (suite)

```
#include <stdio.h>

typedef struct
{
    int    len;
    float *p;
} vecteur;

void C_moyenne(vecteur vec)
{
  float moy ;

  printf("Le vecteur vec a %d éléments.\n", vec.len) ;
  moy = 0. ;
  for(int i=0; i<vec.len; i++) moy += vec.p[i] ;
  moy /= vec.len ;
  printf("Moyenne = %f\n", moy) ;
  return ;
}
```

Exemple 1 : Fortran ==> C (structure de données en argument avec composante pointeur)



Interopérabilité d'une structure de données passée par valeur et contenant une composante *pointeur* associée à une cible dynamique. Dans cet exemple la cible est allouée en C puis valorisée dans le sous-programme Fortran appelé.

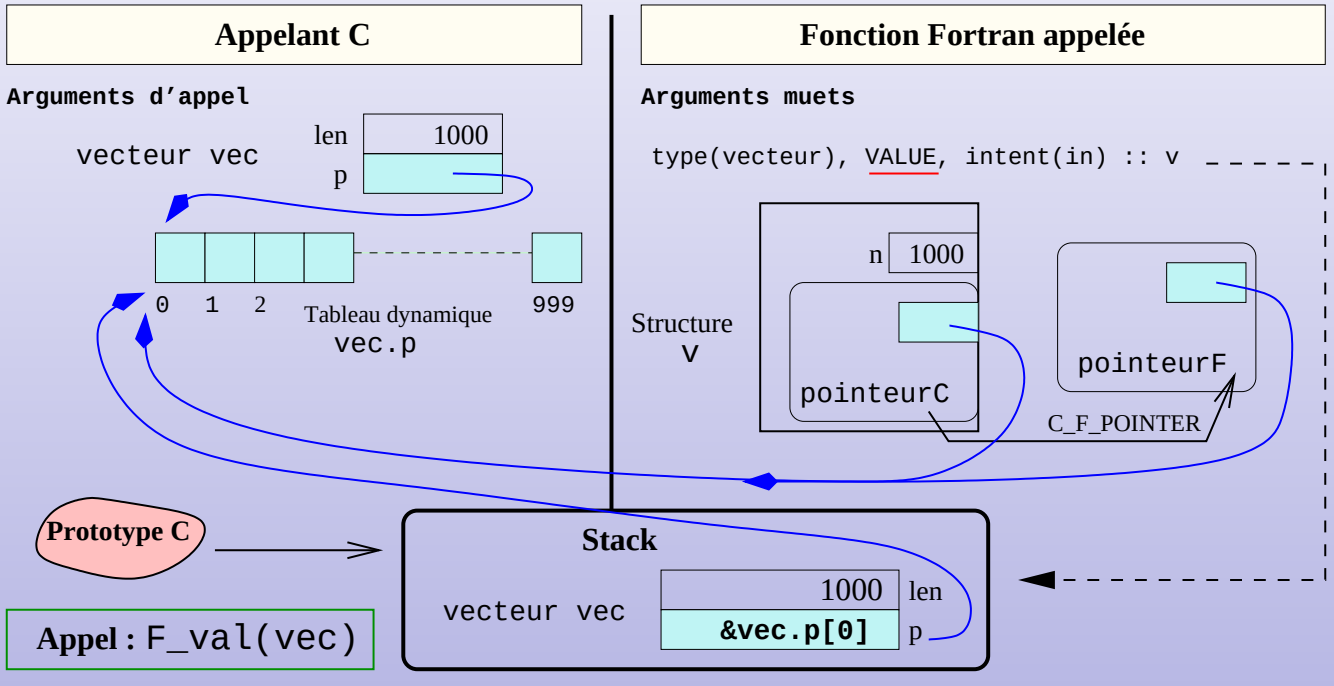
Exemple C ⇒ Fortran

```
module m
  use ISO_C_BINDING
  implicit none
  type, BIND(C)           :: vecteur
    integer(kind=C_INT)   :: n
    ! Composante dynamique type "pointeur C"
    type(C_PTR)           :: pointeurC
  end type vecteur
contains
  subroutine valorisation(v) BIND(C, NAME="F_val")
    type(vecteur), VALUE                        :: v
    real(kind=C_FLOAT), dimension(:), pointer :: pointeurF

    print *, "Taille du vecteur alloué en C :", v%n
    !-- Conversion du  "pointeur C" en pointeur Fortran
    call C_F_POINTER(CPTR=v%pointeurC, FPTR=pointeurF, &
                     SHAPE=[ v%n ])
    call random_number(pointeurF)
  end subroutine valorisation
end module m
```

Exemple C ⇒ Fortran (suite)

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct
{
    int    len;
    float *p;
} vecteur;

void F_val(vecteur v);

int main()
{
  float   moy ;
  vecteur vec ;

  moy = 0. ;
  vec.p = (float *)calloc(vec.len=1000, sizeof(float)) ;
  F_val(vec) ;
  for(int i=0; i<vec.len; i++) moy += vec.p[i] ;
  moy /= vec.len ; printf("Moyenne = %f\n", moy) ;
  free(vec.p) ;
  return 0 ;
}
```


Exemple 2 : C ==> Fortran (structure de données en argument avec composante pointeur)



Reprenons ce même exemple en passant cette fois-ci le vecteur vec par référence.

Exemple C ⇒ Fortran

```
void F_val(vecteur *);

int main()
{
  . . .
  F_val(&vec) ;
  . . .
}

module m
  . . .
contains
  subroutine valorisation(v) BIND(C, NAME="F_val")
    type(vecteur), intent(in) :: v
                . . .
  end subroutine valorisation
end module m
```

Exemple d'interopérabilité avec des types dérivés/structures

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

typedef struct
{
  int i, j;
} Pair;

typedef struct
{
  int     n;
  double  r;
  int     n_ch;
  char  **tab_chaines;
  int     nb_pairs;
  Pair   *p;
} Cel;
```

Exemple d'interopérabilité avec des types dérivés/structures (suite)

```
int main()
{
  void Ap_For(Cel);
  Cel c;

  c.n = 1756, c.r = acos(-1.), c.n_ch = 10;
  c.tab_chaines = calloc(c.n_ch, sizeof(char *));
  for (int n=0; n<c.n_ch; n++)
  {
    char buffer[] = "File numberxx";

    sprintf(buffer+11, "%02d", n);
    c.tab_chaines[n] = strdup(buffer);
  }
  c.p = calloc(c.nb_pairs=3, sizeof(Pair));
  for (int n=0; n<c.nb_pairs; n++)
    c.p[n].i = n+1000, c.p[n].j = n+2000;

  Ap_For(c);

  for (int n=0; n<c.n_ch; n++) free(c.tab_chaines[n]);
  free(c.tab_chaines);
  free(c.p);

  return 0;
}
```


Exemple d'interopérabilité avec des types dérivés/structures (suite)

```
module m
  use ISO_C_BINDING
  type, bind(C) :: pair
    integer(kind=C_INT) i
    integer(kind=C_INT) j
  end type pair

  type, bind(C) :: cellule
    integer(kind=C_INT) n
    real(kind=C_DOUBLE) r
    integer(kind=C_INT) n_ch
    type(C_PTR)         p
    integer(kind=C_INT) n_pairs
    type(C_PTR)         q
  end type cellule

  type(C_PTR), dimension(:), pointer :: tab_ptr
  character(len=:,kind=C_CHAR), pointer :: chaine
  type(pair), dimension(:), pointer :: p_pair
```

Exemple d'interopérabilité avec des types dérivés/structures (suite)

```
contains
  subroutine sp(c) bind(C, name="Ap_For")
    use ISO_FORTRAN_ENV
    type(cellule), VALUE :: c
    integer i, n_char

    call C_F_POINTER(CPTR=c%p, FPTR=tab_ptr, shape=[ c%n_ch ])
    do i=1,c%n_ch
      call C_F_POINTER(CPTR=tab_ptr(i), FPTR=chaine)
      n_char = 1
      do while(chaine(n_char:n_char) /= C_NULL_CHAR)
        n_char = n_char + 1
      end do
      write(OUTPUT_UNIT, *) chaine(1:n_char)
    end do

    call C_F_POINTER(CPTR=c%q, FPTR=p_pair, shape=[ c%n_pairs ])
    do i=1,c%n_pairs
      write(OUTPUT_UNIT, *) p_pair(i)%i, p_pair(i)%j
    end do
  end subroutine sp
end module m
```


Pointeur de fonction/pointeur de procédure : le type C_FUNPTR

Au même titre que les pointeurs C, les pointeurs de fonction sont interopérables avec les pointeurs de procédures au moyen du type opaque C_FUNPTR. Pour gérer l'adresse qui y est encapsulée, on dispose des services suivant :

- fonction `C_FUNLOC(X)` qui retourne l'adresse C d'une procédure Fortran X (ayant obligatoirement l'attribut BIND(C)) dans un scalaire de type C_FUNPTR,
- sous-programme `C_F_PROCPOINTER(CPTR, FPTR)` qui convertit un pointeur de fonction de type C_FUNPTR (CPTR) en un pointeur de procédure Fortran (FPTR).

Exemple

```
module m
  use ISO_C_BINDING
  implicit none

  interface
    subroutine appel_C() bind(C)
    end subroutine appel_C
  end interface
  type(C_FUNPTR), bind(C) :: addr, addr_func

contains

  function carre(x) bind(C)
    real(C_FLOAT), VALUE, INTENT(IN) :: x
    real(C_FLOAT) :: carre

    carre = x*x
  end function carre
end module m
```

Exemple (suite)

```
program p
  use m
  implicit none
  procedure(carre), pointer :: ptr_proc, fonction

  ptr_proc => carre
  addr = C_FUNLOC(carre)
  !addr = C_FUNLOC(ptr_proc)

  call appel_C

  call C_F_PROCPOINTER(CPTR=addr_func, FPTR=fonction)
  print *, fonction(2.718);
end program p
```

Exemple (suite)

```
#include <stdio.h>

float (*addr)(float);
float (*addr_func)(float);

float f(float x)
{
  return x*x*x;
}

void appel_c(void)
{
  printf("%f\n", addr(3.14f));
  addr_func = f;

  return;
}
```

Arithmétique IEEE et traitement des exceptions

Le standard `IEEE-754` concernant l'arithmétique réelle flottante ainsi que le traitement des exceptions définit un système de représentation des nombres flottants. Ce système permet la représentation des `valeurs spéciales` suivantes :

- `NaN` (*Not a Number*) : valeur d'une expression mathématique indéterminée comme $0/0$, $0 * \infty$, ∞/∞, $\sqrt{-1}$;
- `+INF` $(+\infty)$, `-INF` $(-\infty)$;
- 0^+, 0^- ;
- *dénormalisées* : concernent les très petites valeurs.

Dans tout système de représentation, en l'occurrence celui défini par le standard `IEEE`, l'ensemble des réels représentables est un ensemble fini.

Exceptions

Dans des cas extrêmes, une opération arithmétique peut produire comme résultat une des valeurs spéciales indiquées ci-dessus ou bien une valeur en dehors de l'ensemble des valeurs représentables. De tels cas génèrent des **événements de type exception**.

Le standard `IEEE` définit 5 classes d'exception :

- *overflow* : valeur calculée trop grande,
- *underflow* : valeur calculée trop petite,
- division par zéro,
- opération invalide : valeur calculée égale à `NaN`,
- opération inexacte : valeur calculée non représentable exactement (implique un arrondi).

Dans le cas d'un *underflow*, la valeur calculée est soit une valeur *dénormalisée* (*gradual underflow*) soit 0 (*abrupt underflow*) selon le choix du programmeur.
Lorsqu'une exception se produit, un ***flag* spécifique** est positionné.

Mode d'arrondi

Lorsque la valeur calculée n'est pas représentable, une exception de type *opération inexacte* est générée et le calcul se poursuit avec une valeur approchée (arrondie).
Le standard `IEEE` définit **4 modes d'arrondi** :

- *toward nearest* (défaut sur `IBM xlf`) ;
- *toward zéro* ;
- *toward* `+INF` ($+\infty$) ;
- *toward* `-INF` ($-\infty$).

Note : aucune valeur par défaut n'est prévue par la norme !

Modules intrinsèques

Trois modules intrinsèques permettent l'accès aux fonctionnalités définies par le standard IEEE :

- IEEE_ARITHMETIC,
- IEEE_EXCEPTIONS,
- IEEE_FEATURES.

Ces modules contiennent des définitions :

- de types,
- de constantes symboliques,
- de procédures.

Fonctions d'interrogation

Ce sont des fonctions d'interrogation sur l'environnement utilisé afin de savoir s'il est conforme en tout ou partie au standard IEEE :

- IEEE_SUPPORT_STANDARD(x),
- IEEE_SUPPORT_DATATYPE(x),
- IEEE_SUPPORT_DENORMAL(x),
- IEEE_SUPPORT_INF(x),
- IEEE_SUPPORT_NAN(x),

Elles retournent une valeur logique indiquant si l'environnement utilisé respecte le standard ou un aspect du standard pour le type de l'argument réel x fourni.

Il existe des fonctions permettant de connaître la **classe** ou le type de valeur d'un réel x (NaN, ∞, négatif, positif, nul, ...). L'appel à ces fonctions n'est possible que si la fonction IEEE_SUPPORT_DATATYPE appliquée à ce réel retourne la valeur vraie.
Les classes sont définies via des constantes symboliques d'un type prédéfini (IEEE_CLASS_TYPE) dont voici la liste :

- IEEE_SIGNALING_NAN (NaNS),
- IEEE_QUIET_NAN (NaNQ),
- IEEE_NEGATIVE_INF,
- IEEE_POSITIVE_INF,
- IEEE_NEGATIVE_DENORMAL,
- IEEE_POSITIVE_DENORMAL,
- IEEE_NEGATIVE_NORMAL,
- IEEE_NEGATIVE_ZERO,
- IEEE_POSITIVE_NORMAL,
- IEEE_POSITIVE_ZERO,
- IEEE_OTHER_VALUE

Ces fonctions sont les suivantes :

- IEEE_CLASS(x),
- IEEE_IS_NAN(x),
- IEEE_IS_FINITE(x),
- IEEE_IS_NEGATIVE(x),
- IEEE_IS_NORMAL(x).

De plus la fonction `IEEE_VALUE(x, class)` génère un réel d'un type (celui de x) et d'une classe donnés.
L'exemple qui suit permet de récupérer la classe d'un réel x lu dans le fichier fort.1.

Exemple

```
program class
  use IEEE_ARITHMETIC
  implicit none
  type(IEEE_CLASS_TYPE) :: class_type
  real                  :: x, y
  read( unit=1 )x
  if(IEEE_SUPPORT_DATATYPE(x)) then
    class_type = IEEE_CLASS(x)
    if (IEEE_SUPPORT_NAN(x)) then
      if (class_type == IEEE_SIGNALING_NAN) &
        print *,"X is a IEEE_SIGNALING_NAN"
      if (class_type == IEEE_QUIET_NAN)     &
        print *,"X is a IEEE_QUIET_NAN"
    end if
    if (IEEE_SUPPORT_INF(x)) then
      if (class_type == IEEE_NEGATIVE_INF) &
        print *,"X is a IEEE_NEGATIVE_INF number"
      if (class_type == IEEE_POSITIVE_INF) &
        print *,"X is a IEEE_POSITIVE_INF number"
    end if
    if (IEEE_SUPPORT_DENORMAL(x)) then
      if (class_type == IEEE_NEGATIVE_DENORMAL) &
        print *,"X is a IEEE_NEGATIVE_DENORMAL number"
      if (class_type == IEEE_POSITIVE_DENORMAL) &
        print *,"X is a IEEE_POSITIVE_DENORMAL number"
    end if
    if (class_type == IEEE_NEGATIVE_NORMAL) &
      print *,"X is a IEEE_NEGATIVE_NORMAL number"
    if (class_type == IEEE_POSITIVE_NORMAL) &
      print *,"X is a IEEE_POSITIVE_NORMAL number"
    if (class_type == IEEE_NEGATIVE_ZERO) &
      print *,"X is a IEEE_NEGATIVE_ZERO number"
    if (class_type == IEEE_POSITIVE_ZERO) &
      print *,"X is a IEEE_POSITIVE_ZERO number"
  end if
  y = IEEE_VALUE(x, class_type); print *,y
end program class
```

Procédures de gestion du mode d'arrondi

Le **mode d'arrondi** utilisé lors de calculs est géré par deux sous-programmes (les différents modes sont définis à l'aide des constantes symboliques IEEE_NEAREST, IEEE_UP, IEEE_DOWN, IEEE_TO_ZERO du type prédéfini IEEE_ROUND_VALUE) :

```
IEEE_GET_ROUNDING_MODE(round_value)
IEEE_SET_ROUNDING_MODE(round_value)
```

Exemple

```
use IEEE_ARITHMETIC
implicit none
type(IEEE_ROUND_TYPE) :: round_value

! Sauvegarde du mode d'arrondi courant.
call IEEE_GET_ROUNDING_MODE(round_value)
! Positionnement du mode d'arrondi toward +INF
call IEEE_SET_ROUNDING_MODE(IEEE_UP)
          ...
          ...
! Restauration du mode d'arrondi sauvegardé.
call IEEE_SET_ROUNDING_MODE(round_value)
```


Gestion des exceptions

Le programmeur, après avoir détecté la survenance d'une exception, a la possibilité de lancer un traitement personnalisé.

À chacune des 5 classes d'exception correspondent 2 drapeaux :

- l'un indiquant si l'événement relatif à l'exception s'est réalisé ;
- l'autre signalant si ce type d'exception provoque une interruption du programme.

Ces différents drapeaux sont référencés à l'aide de constantes symboliques d'un type prédéfini (IEEE_FLAG_TYPE) :

- IEEE_UNDERFLOW,
- IEEE_OVERFLOW,
- IEEE_DIVIDE_BY_ZERO,
- IEEE_INVALID
- IEEE_INEXACT

De plus, deux constantes symboliques de type tableau (IEEE_USUAL, IEEE_ALL) permettent de référencer tout ou partie de ces drapeaux :

```
IEEE_USUAL = (/ IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, IEEE_INVALID /)
IEEE_ALL   = (/ IEEE_USUAL, IEEE_UNDERFLOW, IEEE_INEXACT /)
```

Ces constantes symboliques sont principalement utilisées comme argument de fonctions telles que :

```
IEEE_GET_FLAG(flag, flag_value)
IEEE_SET_FLAG(flag, flag_value)
```

Ces procédures retournent dans l'argument flag_value un logique signalant l'état de l'exception indiquée en 1er argument sous forme d'une des constantes symboliques précédentes.

Exemple

```
program except
  use IEEE_EXCEPTIONS
  implicit none
  real    valeur_calculee
  logical flag_value

  call IEEE_SET_FLAG(IEEE_ALL, .false.)

  valeur_calculee = 2.453*4.532
  print *, "valeur_calculée : ", valeur_calculee
  call IEEE_GET_FLAG(IEEE_INEXACT, flag_value)
  if(flag_value) print *, "Valeur calculée inexacte."
end program except
```

Note :

- lors de l'appel et du retour d'une procédure, le contexte de gestion de l'arithmétique IEEE est sauvegardé puis restauré. Pour des raisons de performance, ce processus peut être inhibé sur certains environnements via une option du compilateur (cf. option -qnostrictieeemod de xlf sur IBM).

Lorsqu'une exception de type *underflow* se produit, le résultat du calcul est soit 0 (*abrupt underflow*) soit un nombre dénormalisé si le processeur supporte de tels nombres (*gradual underflow*).
Les deux sous-programmes suivants permettent de gérer le mode d'*underflow* désiré (*gradual underflow* ou *abrupt underflow*) :

```
IEEE_GET_UNDERFLOW_MODE(gradual)
IEEE_SET_UNDERFLOW_MODE(gradual)
```

Exemple

```
use IEEE_ARITHMETIC
implicit none
logical :: save_underflow_mode

! Sauvegarde du mode d'underflow courant.
call IEEE_GET_UNDERFLOW_MODE(GRADUAL=save_underflow_mode)
! Positionnement du mode abrupt underflow
CALL IEEE_SET_UNDERFLOW_MODE(GRADUAL=.false.)
! Calculs dans le mode abrupt underflow ; une valeur
! trop petite est alors remplacée par zéro.
          . . .
! Restauration du mode d'underflow sauvegardé.
CALL IEEE_SET_UNDERFLOW_MODE(GRADUAL=save_underflow_mode)
```


Procédures de gestion des interruptions

Lorsqu'une exception est générée, le programme peut s'arrêter ou bien continuer.
Ce mode de fonctionnement est contrôlé par les sous-programmes suivants :

```
IEEE_GET_HALTING_MODE(flag, halting)
IEEE_SET_HALTING_MODE(flag, halting)
```

Exemple

```
use IEEE_ARITHMETIC
implicit none
real                    :: x, zero
logical, dimension(5)   :: flags
logical                 :: arret

read *,zero, arret ! zero = 0.
! Mode d'interruption suite à une division par zéro.
call IEEE_SET_HALTING_MODE(IEEE_DIVIDE_BY_ZERO, arret)
x = 1./zero; print *,x
call IEEE_GET_FLAG(IEEE_ALL, flags)
print *, flags
```

Procédures de gestion du contexte arithmétique

Deux autres sous-programmes gèrent l'état de l'environnement relatif à l'arithmétique flottante (drapeaux d'exceptions et d'interruptions, mode d'arrondi) :

```
IEEE_GET_STATUS(status_value)
IEEE_SET_STATUS(status_value)
```

L'argument `status_value` est du type `IEEE_STATUS_TYPE` ; il contient en entrée (`SET`) ou en sortie (`GET`) l'état de tous les drapeaux relatifs à l'arithmétique flottante.

Exemple

```
use IEEE_EXCEPTIONS
implicit none
type(IEEE_STATUS_TYPE) status_value
!
! Sauvegarde de tout le contexte flottant IEEE.
call IEEE_GET_STATUS(status_value)
!
! Mettre tous les "drapeaux" de type exception à faux.
call IEEE_SET_FLAG(IEEE_ALL, .false.)
!
!   Calculs avec traitement des exceptions éventuelles.
        ...
! Restauration de tout le contexte flottant IEEE précédent.
call IEEE_SET_STATUS(status_value)
        ...
```

Exemple complémentaire sur les exceptions

```
use IEEE_EXCEPTIONS
implicit none
type(IEEE_FLAG_TYPE), dimension(2), parameter :: &
   out_of_range = (/ IEEE_OVERFLOW, IEEE_UNDERFLOW /)
logical, dimension(2)                         :: flags_range
logical, dimension(5)                         :: flags_all

call IEEE_SET_HALTING_MODE(IEEE_ALL, .false.)
           ...
           ...
call IEEE_GET_FLAG(out_of_range, flags_range)
if (any(flags_range)) then
! Une exception du type "underflow" ou "overflow" s'est produite.
           ...
end if
           ...
           ...
call IEEE_GET_FLAG(IEEE_ALL, flags_all) !<== Procédure élémentaire
if (any(flags_all)) then
! Une exception quelconque s'est produite.
           ...
end if
```

Modules intrinsèques

La disponibilité des modules IEEE_ARITHMETIC, IEEE_EXCEPTIONS et IEEE_FEATURES dépend de l'environnement utilisé, de même que les constantes symboliques définies dans le module IEEE_FEATURES, dans le cas où celui-ci est fourni.

Le module IEEE_ARITHMETIC se comporte comme s'il contenait une instruction use IEEE_EXCEPTIONS.

Si, dans une unité de programme, le module IEEE_ARITHMETIC ou IEEE_EXCEPTIONS est accessible, les fonctionnalités IEEE_OVERFLOW et IEEE_DIVIDE_BY_ZERO sont supportées dans cette unité pour tout type de réels et de complexes. On utilisera la fonction IEEE_SUPPORT_FLAG afin de savoir si les autres fonctionnalités le sont.

Ces modules définissent 5 types dérivés dont les composantes sont privées et un ensemble de procédures.

Module IEEE_EXCEPTIONS

- Il définit les types :
  - IEEE_FLAG_TYPE permettant d'identifier un type d'exception particulier. Les valeurs possibles sont les constantes symboliques suivantes :
    - IEEE_INVALID,
    - IEEE_OVERFLOW,
    - IEEE_DIVIDE_BY_ZERO,
    - IEEE_UNDERFLOW,
    - IEEE_INEXACT
  - IEEE_STATUS_TYPE pour la sauvegarde de l'environnement flottant.
- Il définit les fonctions d'interrogations suivantes :
  - IEEE_SUPPORT_FLAG,
  - IEEE_SUPPORT_HALTING,
- Il définit les sous-programmes élémentaires suivants :
  - IEEE_GET_FLAG,
  - IEEE_GET_HALTING_MODE,
- Il définit les sous-programmes non élémentaires suivants :
  - IEEE_GET_STATUS,
  - IEEE_SET_FLAG,
  - IEEE_SET_HALTING_MODE,
  - IEEE_SET_STATUS

Module IEEE_ARITHMETIC

- Il définit les types :
  - IEEE_CLASS_TYPE permettant d'identifier la classe d'un réel. Les valeurs possibles sont les constantes symboliques suivantes :
    - IEEE_SIGNALING_NAN,
    - IEEE_QUIET_NAN,
    - IEEE_NEGATIVE_INF,
    - IEEE_NEGATIVE_NORMAL,
    - IEEE_NEGATIVE_DENORMAL,
    - IEEE_NEGATIVE_ZERO,
    - IEEE_POSITIVE_ZERO,
    - IEEE_POSITIVE_DENORMAL,
    - IEEE_POSITIVE_NORMAL,
    - IEEE_POSITIVE_INF,
    - IEEE_OTHER_VALUE
  - IEEE_ROUND_TYPE permettant d'identifier le mode d'arrondi. Les valeurs possibles sont les constantes symboliques suivantes :
    - IEEE_NEAREST,
    - IEEE_TO_ZERO,
    - IEEE_UP,
    - IEEE_DOWN,
    - IEEE_OTHER

De plus, il surdéfinit les opérateurs == et /= pour deux valeurs d'un de ces types.

- Il définit les fonctions d'interrogation suivantes :
  - IEEE_SUPPORT_DATATYPE,
  - IEEE_SUPPORT_DENORMAL,
  - IEEE_SUPPORT_DIVIDE,
  - IEEE_SUPPORT_INF
- Il définit les fonctions élémentaires suivantes :
  - IEEE_CLASS,
  - IEEE_COPY_SIGN,
  - IEEE_IS_FINITE,
  - IEEE_IS_NAN,
  - IEEE_IS_NORMAL,
  - IEEE_IS_NEGATIVE,
  - IEEE_LOGB,
  - IEEE_NEXT_AFTER,
  - IEEE_REM,
  - IEEE_RINT,
  - IEEE_SCALEB,
  - IEEE_UNORDERED,
  - IEEE_VALUE

- Il définit la fonction de transformation suivante :
  - IEEE_SELECTED_REAL_KIND
- Il définit les sous-programmes non élémentaires suivants :
  - IEEE_GET_ROUNDING_MODE,
  - IEEE_GET_UNDERFLOW_MODE,
  - IEEE_SET_ROUNDING_MODE,
  - IEEE_SET_UNDERFLOW_MODE

Module IEEE_FEATURES

• Il définit les constantes symboliques (du type IEEE_FEATURES_TYPE) associées à des fonctionnalités IEEE :

- IEEE_DATATYPE,
- IEEE_DENORMAL,
- IEEE_DIVIDE,
- IEEE_HALTING,
- IEEE_INEXACT_FLAG,
- IEEE_INF,
- IEEE_INVALID_FLAG,
- IEEE_NAN,
- IEEE_ROUNDING,
- IEEE_SQRT,
- IEEE_UNDERFLOW_FLAG

Pour un processeur donné, une partie de ces fonctionnalités peuvent être *naturelles* et seront donc mises en œuvre en l'absence du module IEEE_FEATURES. Pour ce processeur, le fait de coder l'instruction use IEEE_FEATURES dans une unité de programme aura pour effet de solliciter d'autres fonctionnalités au prix d'un surcoût.
Le programmeur peut demander l'accès, à l'aide de la clause ONLY de l'instruction use précédente, à une fonctionnalité particulière laquelle peut être :

- naturelle ;
- génératrice d'un surcoût ;
- non disponible, un message d'erreur sera alors émis par le compilateur.

Exemple

```
USE, INTRINSIC :: IEEE_FEATURES, ONLY: IEEE_DIVIDE
```

Documentations

- `http://www.dkuug.dk/jtc1/sc22/open/n3661.pdf` ⇒ Exceptions and IEEE arithmetic ;
- `http://www-1.ibm.com/support/docview.wss?uid=swg27003923&aid=1` ⇒ Chapter 16. Floating-point Control and Inquiry Procedures ;
- `http://cch.loria.fr/documentation/IEEE754/ACM/goldberg.pdf`

Énumération

Une énumération est un ensemble de constantes symboliques appelées énumérateurs. Ces énumérations facilitent l'interopérabilité avec celles du langage C. Cependant on peut les utiliser en dehors de ce contexte.

```
ENUM, BIND(C)
 ENUMERATOR [::] liste
     ...
END ENUM
```

Remarques :

- aucun nouveau type n'est créé selon ce procédé ;
- la présence de BIND(C) est obligatoire ;
- la valeur d'un énumérateur est :
  - celle indiquée lors d'une éventuelle initialisation ;
  - si aucune initialisation n'est fournie, sa valeur est 0 s'il est le premier, sinon celle du précédent augmenté de 1.

Exemple

```
enum, bind(C)
  enumerator :: red=4, blue=9
  enumerator yellow
end enum
```

Cet exemple permet de définir un ensemble de constantes symboliques red, blue et yellow qui auront les valeurs 4, 9 et 10 respectivement.

Bloc associate

La construction `associate` permet d'associer un nom à une variable ou à une expression au sein d'un bloc.

Exemple

```
    ...
associate(z => exp(-(x**2+y**2))*cos(theta))
  print *, a+z, a-z
end associate
    ...
associate(xc => ax%b(i,j)%c)
  xc%dv = xc%dv + product(xc%ev(1:n))
end associate
    ...
associate(array => ax%b(i,:)%c)
  array(n)%ev = array(n-1)%ev
end associate
    ...
associate(w => result(i,j)%w, zx => ax%b(i,j)%d, zy => ay%b(i,j)%d)
  w = zx*x + zy*y
end associate
```

Attribut volatile

Ce nouvel attribut indiqué lors de la déclaration d'une variable indique au compilateur que celle-ci peut à tout moment être modifiée en dehors du programme `Fortran`. Toute référence à cette variable obligera le compilateur à charger son contenu de la mémoire et non d'un registre. De ce fait, l'optimisation est désactivée lors de l'utilisation de telles variables.

La raison d'être de cet attribut est d'interopérer avec des bibliothèques de traitement parallèles telles `MPI` lesquelles permettent d'effectuer des transferts de données asynchrones d'un processeur vers un autre.

Dans l'exemple qui suit, une demande de transfert de la donnée `data` est effectuée en mode asynchrone via l'appel à `mpi_isend`, la synchronisation est faite ensuite à l'aide de `mpi_wait`.
Le compilateur peut, à des fins d'optimisation, déplacer l'instruction d'affectation « `data = newdata` » avant la demande de synchronisation, ce qui pourrait avoir pour effet de transférer la nouvelle valeur de la variable `data`, le transfert étant asynchrone. Pour éviter toute optimisation de ce genre, on précise l'attribut `volatile` lors de la déclaration de la variable `data`.

Exemple

```
subroutine transfert( ... )
  use mpi
  double precision, allocatable, dimension(:)           :: newdata
  double precision, allocatable, dimension(:), volatile :: data
        ...
  call mpi_isend(data, size(data), mpi_double_precision, &
                 dest, tag, comm, request, ierr)
        ...
  call mpi_wait(request, status)
  data = newdata
        ...
end subroutine transfert
```

Longueurs des identificateurs et des instructions

La longueur maximum pour un identificateur est passée de 31 à 63 caractères. En Fortran 90/95 une instruction est limitée à 40 lignes (20 en format fixe). Cette limite passe dorénavant à 256 lignes quel que soit le format. Une instruction peut donc désormais admettre 255 lignes suites. La raison de cette augmentation est de pouvoir analyser certains programmes Fortran générés de façon automatique.

Constantes binaires, octales et hexadécimales

Précédemment, les constantes binaires, octales et hexadécimales (ou *boz constants*) ne pouvaient apparaître qu'au sein d'instructions de type DATA. Il est maintenant possible de les spécifier lors d'un appel aux fonctions intrinsèques **cmplx**, **dble**, **real** et **int**.

Exemple

```
integer :: i, j
data i/z"3f7"/

j = int( z"3f7" )
```

Nouveautés concernant certaines fonctions intrinsèques

Les fonctions intrinsèques **max**, **maxval**, **min** et **minval** admettent désormais des arguments de type CHARACTER.
Un nouvel argument kind peut être précisé pour les fonctions intrinsèques **count**, **iachar**, **ichar**, **index**, **lbound**, **len**, **len_trim**, **maxloc**, **minloc**, **scan**, **shape**, **size**, **ubound** et **verify**. Cela peut être très utile dans le cas où l'entier par défaut n'a pas le gabarit suffisant pour récupérer le résultat de ces fonctions (sur des machines 64-bits par exemple).

Exemple

```
program p
  use ISO_FORTRAN_ENV, only : OUTPUT_UNIT
  real, dimension(:,:,:,:), allocatable :: a

  allocate(a(64,1024,1024,1024))
    ...
  write(OUTPUT_UNIT, *) size(a, kind=selected_int_kind(12))
    ...
end program p
```

Messages d'erreurs

Lors de l'utilisation des instructions ALLOCATE ou DEALLOCATE il est possible de personnaliser le traitement d'erreur en spécifiant le mot-clé STAT=. Mais l'inconvénient est que le message produit par l'action standard en l'absence de ce mot-clé n'apparaît plus. Il est maintenant possible de le récupérer au moyen d'une nouvelle clause (ERRMSG=) que l'on peut indiquer au niveau de ces instructions.

Exemple

```
program p
  use ISO_FORTRAN_ENV, only : OUTPUT_UNIT
  character(len=256) :: error_message
  integer            :: etat
       ...
  allocate(x(n), STAT=etat, ERRMSG=error_message)
  if (etat > 0) then
    write(OUTPUT_UNIT, *) &
        "Erreur lors de l'allocation de X : ", trim(error_message)
        ...
  end if
       ...
end program p
```


Constantes complexes

On peut écrire des constantes complexes à l'aide de constantes symboliques réelles ou entières.

Exemple

```
real,    parameter :: zero = 0, one = 1
complex, parameter :: i = (zero, one)
```

Exercice 1

Corrigez l'erreur que comporte le programme suivant.
Piste ⇒ celle-ci intervient au moment de la désallocation : la modification d'un mot-clé suffit.

```
program prog
  implicit none
  type t
    real, pointer :: p
  end type t
  real, pointer :: pr
  type(t) :: obj

  allocate( pr )
  pr = 3.14
  call sp( pr )
  print *,obj%p
  call libere
contains
  subroutine sp( r )
    real, target :: r

    obj%p => r
  end subroutine sp

  subroutine libere
    integer ierr

    !deallocate( obj%p, stat=ierr )
    !print *, "ierr = ", ierr
    deallocate( obj%p )
  end subroutine libere
end program prog
```

Exercice 2

Complétez le programme suivant qui se propose de copier le fichier `exo2.data`. Son nom ainsi que celui en sortie seront fournis au lancement de l'exécutable. Chaque enregistrement du fichier en entrée est lu par blocs successifs de longueur fournie dans le fichier `exo2.nml`.

```
program prog
  use ISO_FORTRAN_ENV
  implicit none
  character(len=:), allocatable :: nom_fichier_acopier, nom_fichier_copie
  character(len=:), allocatable :: nom_executable, bloc, buffer
  integer taille_bloc, in_unit, out_unit, ios
  namelist /param/ taille_bloc

  ! Récupération du nom de l'exécutable puis du nom du fichier à copier ainsi que
  ! celui du fichier à créer fournis sur la ligne de commande. Pensez à proposer
  ! un "usage" si le nombre des arguments fourni n'est pas adéquate.
  open( newunit=in_unit, file="exo2.nml", &
        action="read",  status="old",    &
        form="formatted" )
  read( unit=in_unit, nml=param ); close( unit=in_unit )
  open( newunit=in_unit, file=nom_fichier_acopier, &
        action="read",  status="old",              &
        form="formatted" )
  open( newunit=out_unit, file=nom_fichier_copie, &
        action="write",  status="replace",       &
        form="formatted" )
  write(OUTPUT_UNIT, *)"La lecture s'effectuera avec des blocs de ", taille_bloc, "octets."
  allocate( character(len=taille_bloc) :: bloc )
  do
    ios = lire_enreg( ZONE=buffer )
    if ( is_iostat_end( ios ) ) exit
    write( unit=out_unit, fmt="(a)" ) trim(buffer)
  end do
  close( unit=in_unit ); close( unit=out_unit )
CONTAINS
  ! Ecriture de la fonction "lire_enreg" laquelle valorise son argument avec le
  ! contenu d'un enregistrement complet. Cette fonction retourne l'état de la lecture.
    ...
  ! Ecriture de la procédure "usage" appelé lorsque le nombre d'arguments sur la
  ! ligne de commandes est invalide.
end program prog
```

Exercice 3

On définit le menu suivant :

```
1 - AJOUTS d'éléments dans une liste chaînée.
2 - AFFICHAGE de la liste chaînée.
3 - TRI de la liste chaînée.
4 - SUPPRESSION d'éléments dans la liste.
5 - VIDER la liste.
6 - ARRÊT du programme.
```

Le but de cet exercice est d'afficher ce menu à l'écran et de proposer la saisie d'un des numéros de la liste.
Ce menu sera affiché jusqu'à ce que le choix 6 ait été saisi, lequel permet au programme de s'arrêter.

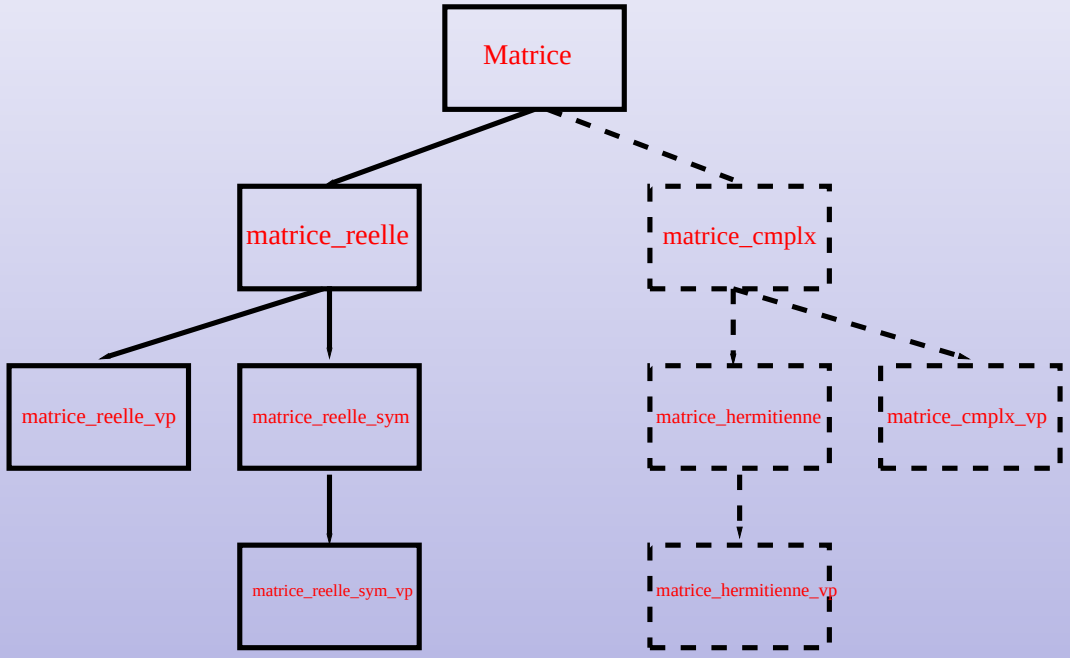
Le traitement correspondant au choix effectué se limitera à :

1. l'affichage du texte associé ;
2. l'affichage d'un message indiquant la nature du traitement désiré.

Remarque :
Le menu pourra être défini comme étant un tableau d'éléments d'un `type dérivé` à définir, lequel sera constitué d'une chaine de caractères pour y mettre le texte affiché ainsi que d'un pointeur d'une procédure chargée de l'action à exécuter.

Exercice 4

Cet exercice propose de calculer les valeurs et/ou vecteurs propres de différents types de matrices. Pour cela, il s'appuie sur le concept de polymorphisme dynamique attaché à la hierarchie de classes suivantes :



On demande d'étendre l'application traîtant les matrices réelles en développant la partie du graphe en pointillés concernant les matrices complexes.

L'application existante est constitué du programme principal ainsi que de deux modules stockés dans les fichiers sources `prog_matrice.f90`, `mat_base.f90` et `mat_real.f90` :

- Le module `mat_base` définit le type abstrait `matrice` avec ses *type_bound procedures* à définir lors de la dérivation, ainsi que deux procédures :
  - **poly_assign** : procédure de surcharge de l'affectation entre objets polymorphes ;
  - **calculs** : procédure effectuant le calcul des valeurs et/ou vecteurs propres de l'objet polymorphe transmis en argument.
- Le module `mat_reel` définit les types étendus :
  - `matrice_reelle` ;
  - `matrice_reelle_vp` ;
  - `matrice_reelle_sym` ;
  - `matrice_reelle_sym_vp`.

  ainsi que les constructeurs et les procédures rattachées à ces types.

`NOTES :`

- L'application est interfacée avec la bibliothèque `lapack95` pour le calcul des valeurs et des vecteurs propres. Les sous-programmes utilisés sont `LA_GEEV`, `LA_SYEV` et `LA_HEEV` (Pour leurs prototypes se reporter à l'annexe B page 275).
- Pour compiler et exécuter cet exercice lancez la commande `make`.

Exercice 5

Soit les deux fonctions suivantes écrites en langage C :

```c
#include <string.h>
#include <stdlib.h>

void c_chaine(char **chaine)
{
  *chaine = strdup("Wolfgang Amadeus Mozart");

  return;
}

void c_chaine_free(char *chaine)
{
  free( chaine );

  return;
}
```

Complétez la partie Fortran suivante contenant l'appel aux deux fonctions précédentes :

```fortran
program inter
  use ISO_C_BINDING, only : C_PTR, C_CHAR, C_NULL_CHAR, C_F_POINTER
  implicit none

  interface
    ! Interface Fortran pour les fonctions
    ! C : c_chaine et c_chaine_free
        ...
  end interface

  ! Déclaration des variables "ptr" et "chaine"
  ! utilisées ci-dessous
      ...

  call c_chaine(ptr)
  chaine = recup_chaine(ptr)
  print *, "Longueur chaine : ", len(chaine)
  print *, "chaine          : ", chaine
contains
  ! Ecriture de la fonction "recup_chaine".
  ! Celle-ci retourne une copie de la chaîne allouée en C.
  ! Pensez à la desallouer une fois la copie effectuée.
      ...
end program inter
```

Corrigé de l'exercice 1

```
program prog
  implicit none
  type t
    real, pointer :: p
  end type t
  real, pointer :: pr
  type(t) :: obj

  allocate(pr)
  pr = 3.14
  call sp(pr)
  print *,obj%p
  call libere
contains
  subroutine sp(r)
    ! Il faut indiquer ici l'attribut POINTER afin de transmettre le descripteur de
    ! pointeur lequel indique que la cible est dynamique (information utilisée lors
    ! de l'exécution de l'instruction deallocate).

    real, pointer :: r

    obj%p => r ! Copie des descripteurs.
  end subroutine sp

  subroutine libere
    integer ierr

    ! deallocate(obj%p, stat=ierr)
    ! print *, "ierr = ", ierr
    deallocate(obj%p) ! L'information est présente.
  end subroutine libere
end program prog
```


Corrigé de l'exercice 2

```
program prog
  use ISO_FORTRAN_ENV
  implicit none
  character(len=:), allocatable :: nom_fichier_acopier, nom_fichier_copie
  character(len=:), allocatable :: nom_executable, bloc
  integer taille_bloc, in_unit, out_unit, long
  namelist /param/ taille_bloc

  !Récupération du nom de l'exécutable.
  call get_command_argument(number=0, length=long)
  allocate(character(len=long) :: nom_executable)
  call get_command_argument(number=0, value=nom_executable)
  if (command_argument_count() /= 2) call usage
  !Récupération du nom du fichier à copier.
  call get_command_argument(number=1, length=long)
  allocate(character(len=long) :: nom_fichier_acopier)
  call get_command_argument(number=1, value=nom_fichier_acopier)
  !Récupération du nom du fichier de sortie.
  call get_command_argument(number=2, length=long)
  allocate(character(len=long) :: nom_fichier_copie)
  call get_command_argument(number=2, value=nom_fichier_copie)
```

Corrigé de l'exercice 2 (suite)

```
  open( newunit=in_unit, file="exo2.nml", &
        action="read",  status="old",     &
        form="formatted" )
  read(unit=in_unit, nml=param)
  close(unit=in_unit)

  open( newunit=in_unit, file=nom_fichier_acopier, &
        action="read",  status="old",              &
        form="formatted" )
  open( newunit=out_unit, file=nom_fichier_copie, &
        action="write",  status="replace",       &
        form="formatted" )

  write(OUTPUT_UNIT, *) "La lecture s'effectuera avec des blocs de ", &
                        taille_bloc, "octets."
  allocate(character(len=taille_bloc) :: bloc)

  block
    character(len=:), allocatable :: buffer
    integer ios
    do
      call lire_enreg(ZONE=buffer)
      if (is_iostat_end(ios)) exit
      write(unit=out_unit, fmt="(a)") trim(buffer)
    end do
  end block
  close(unit=in_unit)
  close(unit=out_unit)
```


Corrigé de l'exercice 2 (suite)

```
contains
  subroutine lire_enreg(zone)
    character(len=:), allocatable :: zone
    character(len=:), allocatable :: zone_etendue
    integer :: nb_blocs, nb_car

    ! On reste positionné dans l'enregistrement pour
    ! pouvoir lire le bloc suivant s'il existe.
    read(unit=in_unit, fmt="(a)", advance="no", iostat=ios) bloc
    if (is_iostat_end(ios)) return ! Si fin de fichier
    zone = bloc
    nb_blocs = 1
    ! Tant que la fin d'enregistrement n'est pas atteinte
    do while (.not.is_iostat_eor(ios))
      nb_blocs = nb_blocs + 1
      ! On étend la zone d'un bloc
      allocate(character(len=nb_blocs*taille_bloc) :: zone_etendue)
      nb_car = len(zone)
      zone_etendue(1:nb_car) = zone
      call MOVE_ALLOC(TO=zone, FROM=zone_etendue)
      read(unit=in_unit, fmt="(a)", advance="no", iostat=ios) zone(nb_car+1:)
    end do
  end subroutine lire_enreg
  subroutine usage
    write(ERROR_UNIT, "(3a)") "Usage : ", nom_executable, " fichier1 fichier2"
    stop 4
  end subroutine usage
end program prog
```

Corrigé de l'exercice 3

```
module GestionMenu
  use ISO_FORTRAN_ENV, only : INPUT_UNIT, ERROR_UNIT
  implicit none
  abstract interface
    subroutine modele
    end subroutine modele
  end interface

  type menu
    character(:),           allocatable :: texte
    procedure(modele), nopass, pointer :: action
  contains
    procedure, pass :: affichage_texte
  end type menu
```

Corrigé de l'exercice 3 (suite)

```
contains
  function SelectionMenu(tab_menu, NbChoix) result(choix)
    type(menu), dimension(:), intent(in) :: tab_menu
    integer,                  intent(in) :: NbChoix
    integer choix
    integer choix_saisi, ios

    do
      print '(//,a)', "Liste des choix :"
      do choix=1, NbChoix
        print *, tab_menu(choix)%texte
      end do
      read( INPUT_UNIT, *, iostat=ios) choix_saisi
      if (ios == 0) then
        if(choix_saisi >= 1 .AND. choix_saisi <= NbChoix) then
          choix = choix_saisi
          exit
        end if
      elseif (ios < 0) then
        choix = NbChoix
        exit
      end if
      write( ERROR_UNIT, '(/,a)') "ERREUR - choix invalide."
    end do
  end function SelectionMenu
```

Corrigé de l'exercice 3 (suite)

```
  subroutine ajouts
    print '(/,a,/)', 'Choix "AJOUTS" selectionne.'
  end subroutine ajouts
  subroutine affichage
    print '(/,a,/)', 'Choix "AFFICHAGE" selectionne.'
  end subroutine affichage
  subroutine tri
    print '(/,a,/)', 'Choix "TRI" selectionne.'
  end subroutine tri
  subroutine suppression
    print '(/,a,/)', 'Choix "SUPPRESSION" selectionne.'
  end subroutine suppression
  subroutine vider
    print '(/,a,/)', 'Choix "VIDER" selectionne.'
  end subroutine vider
  subroutine arret
    print '(/,a,/)', 'Choix "ARRET" selectionne.'
    stop
  end subroutine arret
  subroutine affichage_texte(this)
    class(menu), intent(in) :: this

    print '(//,a)', this%texte
  end subroutine affichage_texte
end module GestionMenu
```


Corrigé de l'exercice 3 (suite)

```
program TestMenu
  use GestionMenu
  implicit none
  type proc
    procedure(modele), nopass, pointer :: p
  end type proc
  integer,    parameter           :: NbChoix = 6
  type(menu), dimension(NbChoix) :: tab_menu
  character(len=*), parameter, dimension(NbChoix) :: tab_chaines = &
            [ "1 - AJOUTS d'éléments dans une liste chaînée.", &
              "2 - AFFICHAGE de la liste chaînée.           ", &
              "3 - TRI de la liste chaînée.                 ", &
              "4 - SUPPRESSION d'éléments dans la liste.    ", &
              "5 - VIDER la liste.                          ", &
              "6 - ARRÊT du programme.                      " ]
  type(proc), dimension(:), allocatable :: tab_procs
  integer choix

  tab_procs = [ proc(ajouts),      proc(affichage), proc(tri), &
                proc(suppression), proc(vider),     proc(arret) ]
  do choix=1, NbChoix
    tab_menu(choix) = menu(tab_chaines(choix), tab_procs(choix)%p)
  end do
  do
    choix = SelectionMenu(tab_menu, NbChoix)
    call tab_menu(choix)%affichage_texte
    call tab_menu(choix)%action
  end do
end program TestMenu
```

Corrigé de l'exercice 4

```
module mat_cmplx
  use la_precision, only: WP => DP
  use f95_lapack,   only: GEEV => LA_GEEV, HEEV => LA_HEEV
  use mat_base
  implicit none

  type, extends(matrice) :: matrice_cmplx
    private
    complex(kind=WP), dimension(:,:), allocatable :: mat
    complex(kind=WP), dimension(:),   allocatable :: valp
    character(len=100) :: format_sortie_mat = &
                  '(*(:,"(",f6.3,",",f6.3,")",1x))'
    character(len=100) :: format_sortie_valp = &
                  '("Valeur propre N.",i2," : (",es10.3,"  , ",es10.3,")")'
  contains
    procedure :: propres => propres_cmplx
    procedure :: sorties => sorties_cmplx
  end type matrice_cmplx

  type, extends(matrice_cmplx) :: matrice_cmplx_vecp
    private
    complex(kind=WP), dimension(:,:), allocatable :: vecp
    character(len=100) :: format_sortie_vecp = &
                  '(*(:,"(",f6.3,",",f6.3,")",1x))'
  contains
    procedure :: propres => propres_cmplx_vecp
    procedure :: sorties => sorties_cmplx_vecp
  end type matrice_cmplx_vecp
```


Corrigé de l'exercice 4 (suite)

```
  type, extends(matrice_cmplx) :: matrice_hermitienne
    private
  contains
    procedure :: propres => propres_hermitienne
  end type matrice_hermitienne

  type, extends(matrice_hermitienne) :: matrice_hermitienne_vecp
    private
    complex(kind=WP), dimension(:,:), allocatable :: vecp
    character(len=100) :: format_sortie_vecp = &
                  '(*(:,"(",f6.3,",",f6.3,")",1x))'
  contains
    procedure :: propres => propres_hermitienne_vecp
    procedure :: sorties => sorties_hermitienne_vecp
  end type matrice_hermitienne_vecp

  interface matrice_cmplx
    module procedure matrice_cmplx_f
  end interface matrice_cmplx
  interface matrice_cmplx_vecp
    module procedure matrice_cmplx_vecp_f
  end interface matrice_cmplx_vecp
  interface matrice_hermitienne
    module procedure matrice_hermitienne_f
  end interface matrice_hermitienne
  interface matrice_hermitienne_vecp
    module procedure matrice_hermitienne_vecp_f
  end interface matrice_hermitienne_vecp
```

Corrigé de l'exercice 4 (suite)

```
contains
  function matrice_cmplx_f(fichier)
    character(len=*), intent(in) :: fichier
    type(matrice_cmplx)          :: matrice_cmplx_f
    integer un, n, i, j

    associate( t => matrice_cmplx_f )
      open(newunit=un,      file=fichier, &
           action="read", form="formatted", access="sequential")
      read(unit=un, fmt="(a, /, i4)") type_matrice, n
      t%type = type_matrice
      t%ordre = n
      allocate(t%mat(n, n), t%valp(n))
      read(unit=un, fmt=*) ((t%mat(i,j),j=1,n),i=1,n)
      close(unit=un)
      call matrice_cmplx_f%sorties(init=.true.)
    end associate
  end function matrice_cmplx_f

  function matrice_cmplx_vecp_f(fichier)
    character(len=*), intent(in) :: fichier
    type(matrice_cmplx_vecp)     :: matrice_cmplx_vecp_f
    integer n

    matrice_cmplx_vecp_f%matrice_cmplx = matrice_cmplx(fichier)
    n = matrice_cmplx_vecp_f%get_ordre()
    allocate(matrice_cmplx_vecp_f%vecp(n,n))
  end function matrice_cmplx_vecp_f
```


Corrigé de l'exercice 4 (suite)

```
  function matrice_hermitienne_f(fichier)
    character(len=*), intent(in) :: fichier
    type(matrice_hermitienne)    :: matrice_hermitienne_f

    matrice_hermitienne_f%matrice_cmplx = matrice_cmplx(fichier)
  end function matrice_hermitienne_f

  function matrice_hermitienne_vecp_f(fichier)
    character(len=*), intent(in)   :: fichier
    type(matrice_hermitienne_vecp) :: matrice_hermitienne_vecp_f
    integer n

    matrice_hermitienne_vecp_f%matrice_hermitienne = &
                                  matrice_hermitienne(fichier)
    n = matrice_hermitienne_vecp_f%get_ordre()
    allocate(matrice_hermitienne_vecp_f%vecp(n,n))
  end function matrice_hermitienne_vecp_f

  subroutine propres_cmplx(this)
    class(matrice_cmplx), intent(inout) :: this

    call geev(this%mat, this%valp)
  end subroutine propres_cmplx

  subroutine propres_cmplx_vecp(this)
    class(matrice_cmplx_vecp), intent(inout) :: this

    call geev(this%mat, this%valp, VR=this%vecp)
  end subroutine propres_cmplx_vecp
```

Corrigé de l'exercice 4 (suite)

```
subroutine propres_hermitienne(this)
  class(matrice_hermitienne), intent(inout) :: this
  REAL(WP), DIMENSION(:), ALLOCATABLE :: W
  integer n

  n = this%get_ordre()
  allocate(W(n))
  call heev(this%mat, w)
  this%valp(:) = W(:)
  deallocate(W)
end subroutine propres_hermitienne

subroutine propres_hermitienne_vecp(this)
  class(matrice_hermitienne_vecp), intent(inout) :: this
  REAL(WP), DIMENSION(:), ALLOCATABLE :: W
  integer n

  n = this%get_ordre()
  allocate(W(n))
  call heev(this%mat, w, JOBZ='V')
  this%valp(:) = W(:)
  this%vecp(:,:) = this%mat(:,:)
  deallocate(W)
end subroutine propres_hermitienne_vecp
```

Corrigé de l'exercice 4 (suite)

```
subroutine sorties_cmplx(this, init)
  class(matrice_cmplx), intent(in) :: this
  logical,     optional, intent(in) :: init
  logical init_loc
  integer i

  if (     present(init)) init_loc = init
  if (.NOT.present(init)) init_loc = .false.

  if (init_loc) then
    print '(//, t20, a, /)', this%type
    do i=1,size(this%mat,1)
      print trim(this%format_sortie_mat), this%mat(i,:)
    end do
  else
    print '(//, t20, "Valeurs propres", /)'
    do i=1,size(this%valp)
      print trim(this%format_sortie_valp), i, this%valp(i)
    end do
  end if
end subroutine sorties_cmplx
```

Corrigé de l'exercice 4 (suite)

```
  subroutine sorties_cmplx_vecp(this, init)
    class(matrice_cmplx_vecp), intent(in) :: this
    logical,          optional, intent(in) :: init
    integer i

    call this%matrice_cmplx%matrice_cmplx%sorties(init)
    print '(//, t20, a, /)', "Vecteurs propres"
    do i=1,size(this%vecp,1)
      print trim(this%format_sortie_vecp), this%vecp(i,:)
    end do
  end subroutine sorties_cmplx_vecp

  subroutine sorties_hermitienne_vecp(this, init)
    class(matrice_hermitienne_vecp), intent(in) :: this
    logical,                optional, intent(in) :: init
    integer i

    call this%matrice_cmplx%matrice_cmplx%sorties(init)
    print '(//, t20, a, /)', "Vecteurs propres"
    do i=1,size(this%vecp,1)
      print trim(this%format_sortie_vecp), this%vecp(i,:)
    end do
  end subroutine sorties_hermitienne_vecp
end module mat_cmplx
```

Corrigé de l'exercice 5

```
program inter
  use ISO_C_BINDING, only : C_PTR, C_CHAR, C_NULL_CHAR, C_F_POINTER
  implicit none

  interface
    subroutine C_chaine(p) bind(C)
      import C_PTR
      type(C_PTR) :: p ! Par référence
    end subroutine C_chaine
    subroutine C_chaine_free(p) bind(C)
      import C_PTR
      type(C_PTR), value :: p ! Par valeur
    end subroutine C_chaine_free
  end interface

  type(C_PTR) :: ptr
  character(len=:,kind=C_CHAR), allocatable :: chaine
  ! Le pointeur C ptr doit être passé par référence,
  ! car c'est la fonction c_chaine qui le valorise.
  call c_chaine(ptr)
  chaine = recup_chaine(ptr)
  print *, "Longueur chaine : ", len(chaine)
  print *, "chaine          : ", chaine
contains
```

Corrigé de l'exercice 5 (suite)

```
  function recup_chaine(ptr)
    type(C_PTR) :: ptr
    character(len=:,kind=C_CHAR), allocatable :: recup_chaine
    character(len=:,kind=C_CHAR), pointer     :: chaine
    integer i, ierr

    ! Conversion du pointeur C ptr en un pointeur Fortran chaine.
    call C_F_POINTER(CPTR=ptr, FPTR=chaine)
    ! Recherche du caractère '\0' fin de chaîne C
    i = 1
    do while (chaine(i:i) /= C_NULL_CHAR)
      i = i + 1
    end do
    recup_chaine = chaine(1:i-1)
    call c_chaine_free(ptr)
  end function recup_chaine
end program inter
```

Prototype GEEV

```
SUBROUTINE GEEV( A, <w>, VL=vl, VR=vr, INFO=info )
     <type>(<wp>), INTENT(INOUT) :: A(:,:)
     <type>(<wp>), INTENT(OUT) :: <w(:)>
     <type>(<wp>), INTENT(OUT), OPTIONAL :: VL(:,:), VR(:,:)
     INTEGER, INTENT(OUT), OPTIONAL :: INFO
où
     <type> ::= REAL | COMPLEX
     <wp>   ::= KIND(1.0) | KIND(1.0D0)
     <w>    ::= WR, WI | W
     <w(:)> ::= WR(:), WI(:) | W(:)
```

A (entrée/sortie) matrice carrée réelle ou complexe dont on désire les valeurs et/ou vecteur propres. En sortie son contenu est détruit.

<w> (sortie) vecteur(s) réels ou complexe dans lesquels seront stockées les valeurs propres.
Dans le cas réel, on précise deux vecteurs WR et WI dans lesquels seront stockées respectivement les parties réelles et imaginaires des valeurs propres. Une valeur propre et sa conjuguée sont stockées de façon consécutive, celle ayant la partie imaginaire positive avant l'autre.
Dans le cas complexe, un seul vecteur W de type complexe est nécessaire, lequel est valorisé à l'aide des valeurs propres.

Prototype GEEV (suite)

VL,VR (sortie/optionnels) matrices réelles ou complexes valorisées respectivement à l'aide des vecteurs propres à gauche et à droite.
Le stockage est effectué dans l'ordre des valeurs propres.
Chaque vecteur propre est calibré de sorte que sa norme euclidienne est égale à 1 et sa partie réelle plus grande que sa partie imaginaire.
Dans le cas d'une matrice réelle, les valeurs propres ainsi que les vecteurs propres sont conjugués deux à deux. Les valeurs de deux vecteurs propres conjugués sont données par :

```
V(j)   = VL(VR)(:,j) + i*VL(VR)(:,j+1)
V(j+1) = VL(VR)(:,j) - i*VL(VR)(:,j+1)
```

Dans le cas complexe :

```
V(j) = VL(VR)(:,j)
```

INFO (sortie) entier reflétant l'état du traitement :
= 0 : avec succès,
< 0 : si INFO = -i, le ième argument à une valeur invalide,
> 0 : si INFO = i, l'algorithme QR utilisé n'est pas arrivé à calculer toutes les valeurs propres. Dans ce cas aucun vecteur propre n'a été calculé.
Les éléments [i+1:n] du vecteur <w> contiennent les valeurs propres qui ont convergé (n désignant l'ordre de la matrice A).
Si cet argument est absent et qu'une erreur se produit, le programme s'arrête avec un message d'erreur.

Prototype SYEV/HEEV

```
SUBROUTINE SYEV / HEEV( A, W, JOBZ, UPLO, INFO )
     <type>(<wp>), INTENT(INOUT) :: A(:,:)
     REAL(<wp>), INTENT(OUT) :: W(:)
     CHARACTER(LEN=1), INTENT(IN), OPTIONAL :: JOBZ, UPLO
     INTEGER, INTENT(OUT), OPTIONAL :: INFO
  où
     <type> ::= REAL | COMPLEX
     <wp>   ::= KIND(1.0) | KIND(1.0D0)
```

```
A       (entrée/sortie) matrice carrée réelle ou complexe dont on désire
        les valeurs et/ou vecteur propres.
        En entrée seul la partie triangulaire supérieure (UPLO="U") ou inférieure
        (UPLO="L") est considérée.
        En sortie :
           - si JOBZ = "V" et INFO = 0, A contient les vecteurs propres
             orthogonaux,
           - si JOBZ = "N", la partie de A considérée est détruite.

W       (sortie) vecteur réel. Si INFO = 0, il contient les valeurs propres
        stockées en ordre croissant,
```

Prototype SYEV/HEEV (suite)

```
JOBZ    (entrée/optionnel) type CHARACTER(len=1)
           = "N" : seules les valeurs propres sont calculées,
           = "V" : calcul des valeurs et vecteurs propres.
        Si JOBZ est absent, JOBZ="N" est pris par défaut,

UPLO    (entrée/optionnel) type CHARACTER(len=1)
           = "U" : seule la partie triangulaire supérieure de la matrice A
                   est analysée,
           = "L" : seule la partie triangulaire inférieure de la matrice A
                   est analysée,
        Si UPLO est absent, UPLO="U" est pris par défaut,

INFO    (sortie) entier reflétant l'état du traitement :
        = 0 : avec succès,
        < 0 : si INFO = -i, le ième argument à une valeur invalide,
        > 0 : si INFO =  i, l'algorithme utilisé diverge.
        Si cet argument est absent et qu'une erreur se produit, le programme
        s'arrête avec un message d'erreur.
```

– Symboles –

– A –

– B –

– C –

– D –

– E –

– F –

– G –

– H –

– I –

– K –

– L –

– M –

– N –

– O –

– P –

– R –

– S –

– T –

– U –

– V –

– W –