Les apports de Fortran 2003

Patrick Corde et Hervé Delouis

13 février 2008

Avertissement : vous trouverez ici un certain nombre de notions introduites par la norme Fortran. Le choix des sujets est arbitraire et non exhaustif. C'est une exploration des principales nouveautés. Les exemples proposés n'ont pu être passés au crible d'un compilateur ; il est donc fort possible que des erreurs ou des interprétations approximatives de la norme s'y soient glissées...

Voici les principaux documents dont nous nous sommes inspirés :

- *Working Draft J3/04-007* - 10 mai 2004 (`www.j3-fortran.org`). Publié par le comité international J3 chargé du développement de la norme Fortran (585 pages)
- *Fortran 95/2003 explained* - Michael METCALF, John REID, Malcolm COHEN - 2004 - OXFORD University Press - ISBN 0-19-852693-8 (416 pages).
- *The Future of Fortran* - John Reid - juillet/août 2003 - Scientific Programming (`www.computer.org/cise/cs2003/c4059.pdf`).
- *The New Features of Fortran 2000* - John Reid - Fortran Forum V.21 N.2 août 2002.
- *Object Orientation and Fortran 2002 : Part II* - Malcolm Cohen : Fortran Forum - SIGPLAN - V.18 N.1 avril 1999.

Table des matières

1 – Meilleure intégration à l'environnement système 11

2 – Interopérabilité avec C 13

2.1 – Entités de type intrinsèques 14

2.2 – Tableaux C 16

2.3 – Variables globales C 17

2.4 – Les pointeurs 19

2.5 – Fonctions C et procédures Fortran 22

2.5.1 – Exemple de fonction C appelée depuis Fortran 24

2.5.2 – Exemple de procédure Fortran appelée depuis C 27

2.5.3 – Interopérabilité entre pointeurs : le type `C_PTR` 31

2.6 – Structures de données C 45

3 – Arithmétique IEEE et traitement des exceptions 55

3.1 – Standard IEEE-754 56

3.1.1 – Valeurs spéciales 56

3.1.2 – Exceptions 57
3.1.3 – Mode d'arrondi 58
3.2 – Intégration standard IEEE : modules intrinsèques 59
3.3 – Fonctions d'interrogation 60
3.4 – Procédures de gestion du mode d'arrondi 65
3.5 – Gestion des exceptions 66
3.6 – Procédures de gestion des interruptions 70
3.7 – Procédures de gestion du contexte arithmétique 71
3.8 – Exemple complémentaire sur les exceptions 73
3.9 – Modules intrinsèques 74
3.9.1 – Module IEEE_EXCEPTIONS 75
3.9.2 – Module IEEE_ARITHMETIC 77
3.9.3 – Module IEEE_FEATURES 81
3.10 – Documentations 83

4 – Nouveautés concernant les tableaux dynamiques 85

4.1 – Passage en argument de procédure 86

4.2 – Composante allouable d'un type dérivé 88

4.3 – Allocation d'un scalaire `ALLOCATABLE` 89

4.4 – Allocation/réallocation via l'affectation 90

4.5 – Sous-programme `MOVE_ALLOC` de réallocation 92

5 – Nouveautés concernant les modules 95

5.1 – L'attribut `PROTECTED` 95

5.2 – L'instruction `IMPORT` du bloc interface 96

5.3 – `USE` et renommage d'opérateurs 97

6 – Entrées-sorties 99

6.1 – Nouveaux paramètres des instructions `OPEN/READ/WRITE` 99

6.2 – Entrées-sorties asynchrones 101

6.3 – Entrées-sorties en mode *stream* 103

6.4 – Traitement personnalisé des objets de type dérivé 106
7 – Pointeurs 116
7.1 – Vocation (`INTENT`) des arguments muets pointeurs 116
7.2 – Association et reprofilage 117
7.3 – Pointeurs de procédures 119
7.3.1 – Pointeurs de procédure : interface implicite 120
7.3.2 – Pointeurs de procédure : interface explicite 122
8 – Nouveautés concernant les types dérivés 126
8.1 – Composante pointeur de procédure 126
8.2 – Paramètres d'un type dérivé 128
8.3 – Constructeurs de structures 130
8.4 – Visibilité des composantes 135
8.5 – Extension d'un type dérivé 137
9 – Programmation orientée objet 140

9.1 – Variable polymorphique 141
9.1.1 – Argument muet polymorphique 142
9.1.2 – Variable polymorphique : attribut `POINTER`, `ALLOCATABLE` 145
9.2 – Construction `SELECT TYPE` 146
9.3 – Pointeurs génériques 148
9.4 – Type effectif d'une variable polymorphique 149
9.5 – Procédures attachées à un type (*type-bound procedures*) 151
9.5.1 – Procédure attachée par nom (*name binding*) 152
9.5.2 – Procédure attachée par nom générique (*generic binding*) 153
9.5.3 – Procédure attachée par opérateur (*operator binding*) 154
9.5.4 – Procédure attachée via le mot-clé `FINAL` (*final binding*) 157
9.6 – Héritage 159
9.6.1 – Héritage d'une procédure *type-bound* 159
9.6.2 – Surcharge d'une procédure *type-bound* 161

9.6.3 – Procédure *type-bound* non surchargeable 163
9.7 – Type abstrait 164
10 – En conclusion 169

Merci d'envoyer toutes remarques concernant ce document à :

`Patrick.Corde@idris.fr`

IDRIS – Bât. 506 – BP167 – 91403 ORSAY – FRANCE

Notes personnelles

1 – Meilleure intégration à l'environnement système

Voici des procédures intrinsèques donnant accès aux arguments de la ligne de commande ou à des variables d'environnement.

- `GET_COMMAND(command, length, status)`
  sous-progr. retournant dans `command` la commande ayant lancé le programme.

- `COMMAND_ARGUMENT_COUNT()`
  fonction retournant le nombre d'arguments de la commande.

- `GET_COMMAND_ARGUMENT(number, value, length, status)`
  sous-progr. retournant dans `value` le `number`^e argument de la commande (numérotés à partir de zéro).

- `GET_ENVIRONMENT_VARIABLE(name, value, length, status, [trim_name])`
  sous-progr. retournant dans `value` la valeur de la variable d'environnement spécifiée en entrée via `name`.

Un nouveau module intrinsèque ISO_FORTRAN_ENV donne accès à des entités publiques concernant 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.

2 – Interopérabilité avec C

L'interopérabilité Fortran–C 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 qui seront ainsi connues et contrôlées comme telles à l'étape de compilation Fortran.

L'accès (`USE`) au module **ISO_C_BINDING** permet l'**interopérabilité** avec C pour un certain nombre d'entités que nous allons passer en revue...

Note : l'interopérabilité Fortran–C est disponible avec le compilateur Fortran `IBM depuis la version 9.1.0`.

2.1 – Entités de type intrinsèques

Via des constantes symboliques définissant :

1. la valeur entière du paramètre `KIND` des types/sous-types autorisés. Par exemple :

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
`CHARACTER(kind=C_CHAR)`	char
etc.	...

Note : pour le type `CHARACTER`, C ne supportant que les variables de longueur 1 (gérées sous forme de tableaux), on pourra déclarer une chaîne sous la forme :

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

2. des caractères spéciaux :

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

2.2 – Tableaux C

Un tableau Fortran est interopérable s'il est d'un type interopérable 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]
```

2.3 – Variables globales C

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

2.4 – 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)` fonction retournant un scalaire de type `C_PTR` contenant l'adresse de la *cible* `X` (au sens de l'opération unaire `&X` selon la norme C) ; par exemple :

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

À noter que la cible `X` doit avoir l'attribut `TARGET` ; elle peut désigner aussi bien un scalaire, un tableau statique, un tableau dynamique alloué, ou même un pointeur associé à un scalaire.

Nous verrons plus loin comment la fonction `C_LOC` facilite l'interopérabilité avec des pointeurs C.

☞ `C_F_POINTER(CPTR, FPTR [,SHAPE])` : 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_CPTR_2])` vérifie que deux *pointeurs C* (de type `C_PTR`) sont identiques ou que le premier est à l'état nul.

Notes :

1. si l'interopérabilité concerne un **pointeur de fonction** (*C function pointer type*), on utilisera alors les entités équivalentes :
   - type dérivé semi-privé `C_FUNPTR`,
   - fonction `C_FUNLOC(X)` retournant l'adresse C d'une procédure `X` dans un scalaire de type `C_FUNPTR`,
   - sous-programme `C_F_PROCPOINTER(CPTR, FPTR)` : convertit `CPTR`, un pointeur de procédure de type `C_FUNPTR`, en un pointeur Fortran `FPTR` .
2. ces entités permettent l'interopérabilité des tableaux dynamiques : un tableau Fortran alloué peut être passé à C et un tableau alloué en C peut être associé à un pointeur Fortran (cf. exemple en fin de chapitre).

2.5 – Fonctions C et procédures Fortran

Nouveautés syntaxiques Fortran :

- ☞ 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 à la définition d'une procédure Fortran interopérable (ou du bloc interface associé à une fonction C dont le nom peut être spécifié avec le sous-paramètre `NAME=`).

L'interface de procédure Fortran est constituée des informations exploitables par Fortran pour définir et contrôler l'interopérabilité avec un prototype de fonction C. Selon les cas, elle est constituée de :

Fortran ⇒ C l'appel procédural de la fonction C et le bloc interface associé ;

C ⇒ Fortran la partie déclarative de la procédure Fortran appelée.

Quelques règles à respecter :

- ☞ 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).

2.5.1 – Exemple de fonction C appelée depuis Fortran

Dans cet exemple, Fortran passe à C deux arguments :
- un tableau passé classiquement par référence,
- une variable entière passée par valeur.

Tout d'abord, voici la fonction C appelée et son **prototype** :

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

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

  return somme ;
}
```

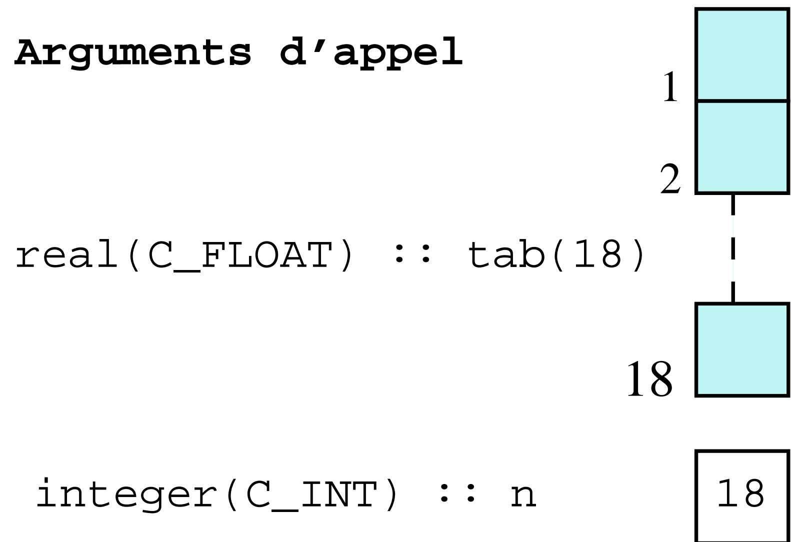
Ci-après, voici :
- ☞ le bloc interface associé à la fonction C (dans un module),
- ☞ le programme Fortran appelant,
- ☞ un schéma récapitulatif du passage des arguments.

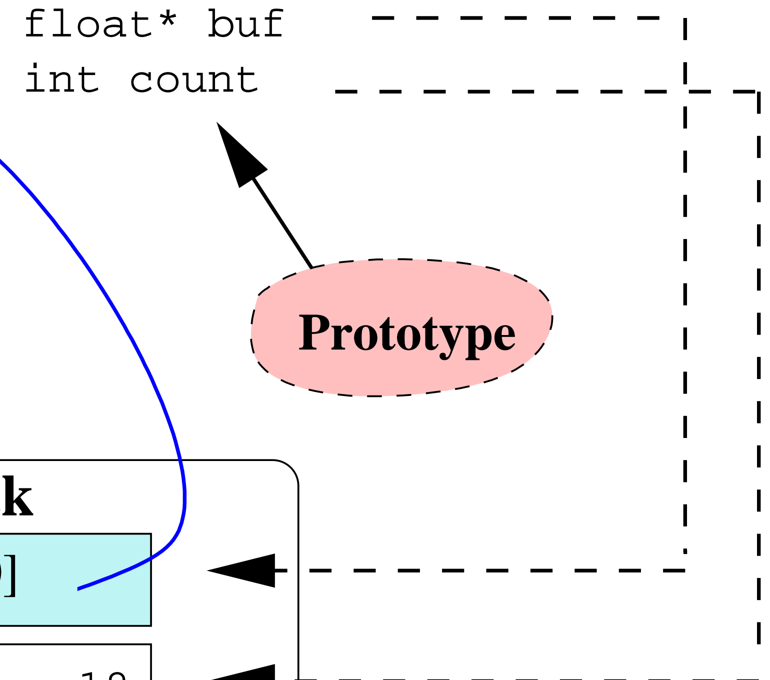
```
module FTN_C
  use, intrinsic :: ISO_C_BINDING
  interface
    function C_FUNC(array, N) BIND(C, NAME="C_Func")
      import C_INT, C_FLOAT
      implicit none
      real(kind=C_FLOAT)                :: C_FUNC
      real(kind=C_FLOAT), dimension(*) :: array
      integer(kind=C_INT), VALUE      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

program p1
  use FTN_C
  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 p1
```

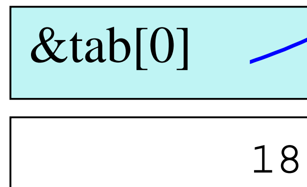
Exemple : fonction C appelée depuis Fortran

Appelant Fortran
+ B.I. + `BIND(C)`

Fonction C appelée







Bloc Interface

```
integer(C_INT), VALUE :: N
real(C_FLOAT), dimension(*) :: array
```

Appel : `y = C_FUNC(array=tab, N=n)`

2.5.2 – Exemple de procédure Fortran appelée depuis C

Dans cet exemple, Fortran reçoit de C trois arguments :

– une variable entière passée par valeur,

– une variable réelle passée par référence,

– un tableau à taille implicite passé par référence.

Voici le **prototype** C de la fonction correspondante et le programme C avec une séquence d'appel de `f1` :

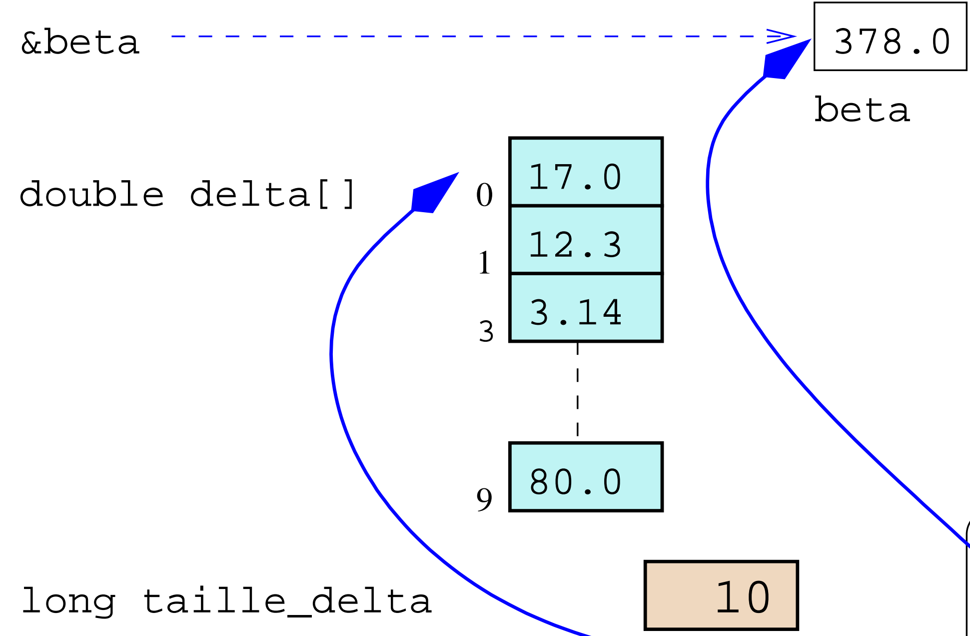
```
void f1( double *b, double d[], long taille_d );
main()
{
  double beta = 378.0 ;
  double delta[] = { 17.,   12.3,  3.14,  2.718, 0.56,
                     22.67, 25.8, 89.,   76.5,  80. } ;
  long   taille_delta = sizeof delta / sizeof delta[0] ;

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

Voici le source du sous-programme Fortran `F1` :

```
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
```

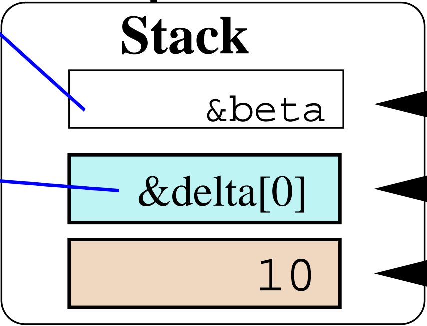
Exemple : procédure Fortran appelée depuis C

Appelant C	Sous–progr. Fortran appelé `BIND(C)`



Arguments muets

```
real(C_DOUBLE), intent(inout) :: B
real(C_DOUBLE), intent(in) :: D(*)
integer(C_LONG), VALUE :: TAILLE_D
```



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

- ☞ Le prototype de la fonction C associée au sous-programme `F1` indique qu'aucune valeur n'est retournée (`void`).
- ☞ Le 1er argument muet `A` de type `INTEGER(C_LONG)` avec l'attribut `VALUE` correspond au paramètre formel `a` du prototype ; il reçoit la **valeur** de `alpha` copiée dans le *stack*. Attention : l'attribut `INTENT` (vocation), n'ayant aucun sens dans ce cas, est interdit !
- ☞ Le 2e argument muet `B` de type `REAL(C_DOUBLE)` correspond au paramètre formel `b` (pointeur typé `double`) du prototype ; il reçoit l'**adresse** de `beta` (`&beta`).
- ☞ Le 3e argument muet `D` de type `REAL(C_DOUBLE)` est un tableau de taille implicite correspondant au paramètre formel `d` du prototype ; il reçoit l'**adresse** du 1er élément du tableau `delta`.

2.5.3 – Interopérabilité entre pointeurs : le type C_PTR

Argument Fortran de type C_PTR		
	Passage par valeur (VALUE)	**Passage par référence**
Fortran ⇓ C	• 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, • Bloc interface obligatoire. ⟹ cf. exemple 1 ci-après	• La fonction C récupère l'adresse de la composante adresse encapsulée dans l'argument de type `C_PTR` **non** nécessairement associé, • Bloc interface obligatoire. ⟹ cf. exemple 2 ci-après
C ⇓ Fortran	• C passe à Fortran un pointeur déjà valorisé, • `intent(OUT/INOUT)` interdit. ⟹ cf. exemple 4 ci-après	• C doit passer l'adresse (`&p`) d'un pointeur qui pourra donc être associé dans la procédure Fortran. ⟹ cf. exemples 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.

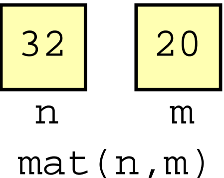
```
program EXEMPLE_1
  use, intrinsic :: ISO_C_BINDING
  real(kind=C_FLOAT), dimension(:,:), allocatable, target :: mat
  integer(kind=C_INT)                                      :: n, m
  interface !------------------------------------------!
    subroutine c_func( n, m, v ) bind(C, name="fct")
      import C_INT, C_PTR
      integer(kind=C_INT), VALUE :: n, m
      type(C_PTR), VALUE         :: v
    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) )
  print *, "SOMME = ", sum( array=mat, dim=1 )
  deallocate( mat )
end program EXEMPLE_1
-----------------------------------------------------------------
void fct( int n, int m, float *vec )          void fct( int n, int m, float mat[m][n] )
{                                           ! {
  float **mat;                              !   for( int i=0; i<m; i++ )
  int     i, j;                             !     mat[i][n-1] *= 2.;
  mat = malloc( m*sizeof(float *) );        !
  for( i=0, j=0; i<m; i++, j+=n )           !   return;
    mat[i] = vec+j, mat[i][n-1] *= 2.;      ! }
  free( mat );                              !
}                                           !
```

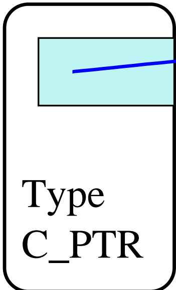
Exemple 1 : Fortran ==> C (argument C_PTR passé par valeur)

Appelant Fortran + **B.I.** + `BIND(C)`	**Fonction C appelée**

Arguments d'appel

```
integer(C_INT) :: n, m
```

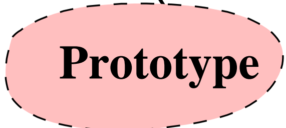




C_LOC(mat)

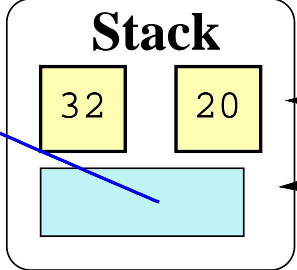
Arguments muets

```
float *vec
int n
int m
```



Bloc Interface

```
integer(C_INT), VALUE :: n, m
type(C_PTR), VALUE :: v
```



Appel : `call c_func(n, m, C_LOC(mat))`

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.

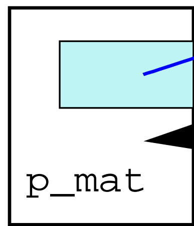
```
program exemple2
  use ISO_C_BINDING
  integer(kind=C_INT)                             :: n, m
  type(C_PTR)                                     :: pointeurC
  real(kind=C_FLOAT), dimension(:,:), pointer :: p_mat
  interface
    subroutine c_alloc( ptrC, n, m ) bind(C, name="C_alloc")
      import 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 ; }
```

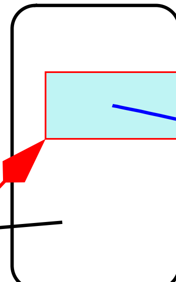
Exemple 2 : Fortran ==> C (argument C_PTR passé par référence)

Appelant Fortran + B.I. + `BIND(C)`	Fonction C appelée

Arguments d'appel

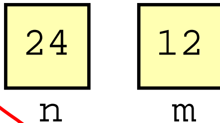
`type(C_PTR) :: pointeurC`





C_F_POINTER

pointeurC



n m

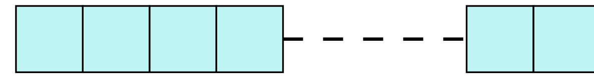
`integer(C_INT) :: n, m`

Arguments muets

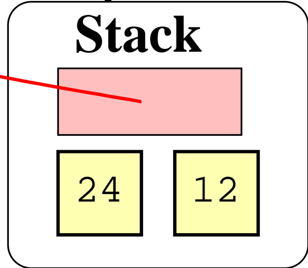
`float **p`

`int n`
`int m`

Prototype



`*p=malloc(n*m*sizeof(float))`



Bloc Interface

```
type(C_PTR), intent(out) :: ptrC
integer(C_INT), VALUE :: n, m
```

Appel : `call c_alloc(pointeurC, n, m)`

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`.

Voici successivement :

- le programme C appelant avec le prototype des sous-programmes Fortran,
- les sous-programmes Fortran,
- un schéma du passage des arguments du sous-programme `for_alloc`.

```
#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;
}
```

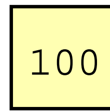
```
module creer_liberer
  use ISO_C_BINDING
  real(kind=C_FLOAT), dimension(:), allocatable, target :: vecteur
contains
  subroutine for_alloc( pointeurC, n ) BIND(C, name="F_alloc")
    type(C_PTR), intent(out)   :: pointeurC
    integer(kind=C_INT), VALUE :: n
    allocate( vecteur(n) )
    call random_number(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

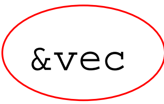
Exemple 3 : C ==> Fortran (argument C_PTR passé par référence)

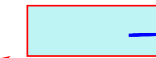
Appelant C	Sous–programme Fortran appelé

Arguments d'appel

`const int n=100;`

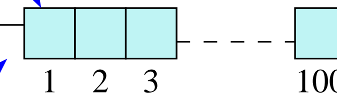


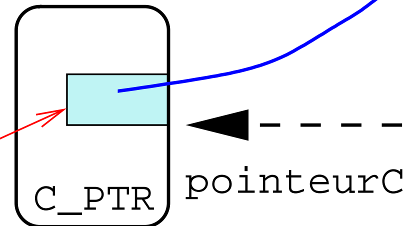


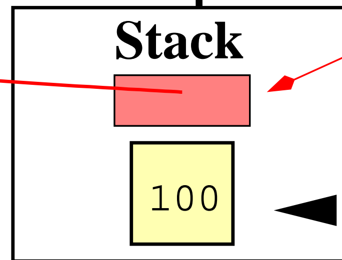


Arguments muets

```
integer(C_INT), VALUE :: n
type(C_PTR), intent(out) :: pointeurC
```





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` car il n'est pas possible de le libérer via un pointeur associé passé en argument du sous-programme `for_free`.

- ☞ L'entité `vecteur` pourrait être déclarée avec l'attribut `pointer` (au lieu de `allocatable`) ; il ne serait alors plus nécessaire de le *globaliser* pour pouvoir désallouer la cible dynamique anonyme qui lui serait associée.

  Par contre, sa conversion en pointeur C interopérable ne pourrait se faire que sous la forme `C_LOC(vecteur(1))` car, selon la norme, l'argument de `C_LOC` ne peut être un pointeur Fortran associé à un tableau.

Exemple 4 : C $\Longrightarrow$ Fortran (argument C_PTR par référence et par valeur)

Dans cet exemple, C sollicite des méthodes Fortran pour manipuler des objets de type dérivé non interopérables adressés via un pointeur de type C_PTR.

1. C déclare deux pointeurs,
2. C appelle la méthode INIT pour créer et valoriser deux objets de ce type (passage par référence),
3. C appelle la méthode ADD pour effectuer un traitement sur ces deux objets (passage par valeur).

Notes :

– dans la méthode INIT on utilise la fonction C_LOC pour retourner à la fonction C l'adresse de l'objet alloué.

– dans la méthode ADD on utilise la procédure C_F_POINTER afin de convertir l'objet de type C_PTR en pointeur Fortran permettant de manipuler l'objet transmis.

```
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
```

```
|-----------------------------------
   /* Fonction C appelante   */
   main()
   {
     void *p, *q ;
       ...
     init( &p, 100 ) ;
     init( &q, 200 ) ;
       ...
     add( p, ... ) ;
     add( q, ... ) ;

    return 0 ;
   }
|-----------------------------------
```

2.6 – Structures de données C

Via l'attribut `BIND(C)`.

Exemple : 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_struc` :

```
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 : les entités `ALLOCATABLE` et les procédures sont exclues pour les composantes qui doivent bien sûr toutes être interopérables ; par contre, les types `C_PTR` et `C_FUNPTR` permettent d'y stocker l'adresse C de telles entités (cf. exemple ci-après).

Exemple Fortran $\Longrightarrow$ C : Fortran passe par valeur à une fonction C une structure de données (type *vecteur*) contenant une composante *pointeur* interopérable associée à une cible dynamique déjà allouée et valorisée. Via un argument muet d'entrée défini comme une structure C équivalente, la fonction C peut accéder à la cible dynamique.

Voici tout d'abord un module contenant la définition du type dérivé Fortran *vecteur* et le bloc interface de la fonction C appelée :

```
module inter
  use ISO_C_BINDING
  type, BIND(C) :: vecteur
    integer(kind=C_INT) :: n
    type(C_PTR)         :: pointeur_C ! Composante dynamique type "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 inter
```

Voici le programme Fortran appelant la fonction C ; il lui passe le *vecteur* `v` encapsulant le tableau dynamique `tab` :

```
program interoper
  use inter
  implicit none
  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 interoper
```

Voici la définition de la fonction `C_moyenne` qui récupére en argument le vecteur passé par Fortran pour accéder au tableau alloué dans le but de calculer la moyenne de ses éléments :

```
#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)

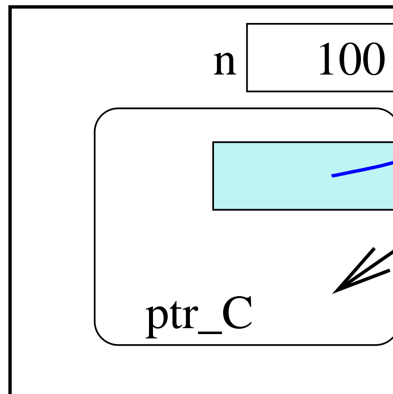
Appelant Fortran
+ B.I. + `BIND(C)`

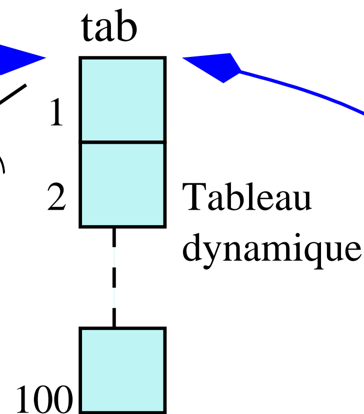
Fonction C appelée

`Arguments d'appel`

`Arguments muets`

Structure
v





`vecteur vec`

Prototype

Bloc Interface

`type(vecteur), VALUE :: vec`

Stack

len	100
p	`&tab[0]`

Appel : `call moyenne(vec=v)`

Exemple C ⟹ Fortran : 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. Voici le sous-programme Fortran `valorisation` :

```
module m
  use ISO_C_BINDING
  implicit none
  type, BIND(C)           :: vecteur
    integer(kind=C_INT) :: n
    type(C_PTR)           :: pointeurC  ! Composante dynamique type "pointeur C"
  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
```

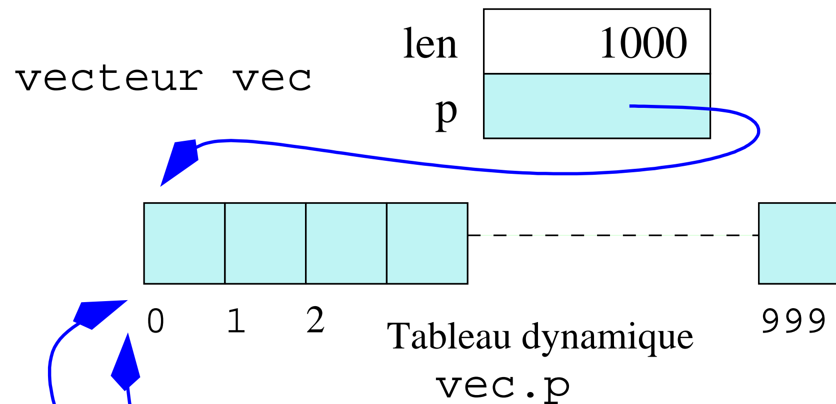
Programme C et prototype de la fonction Fortran `F_val` appelée :

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

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

void F_val(vecteur v);

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)

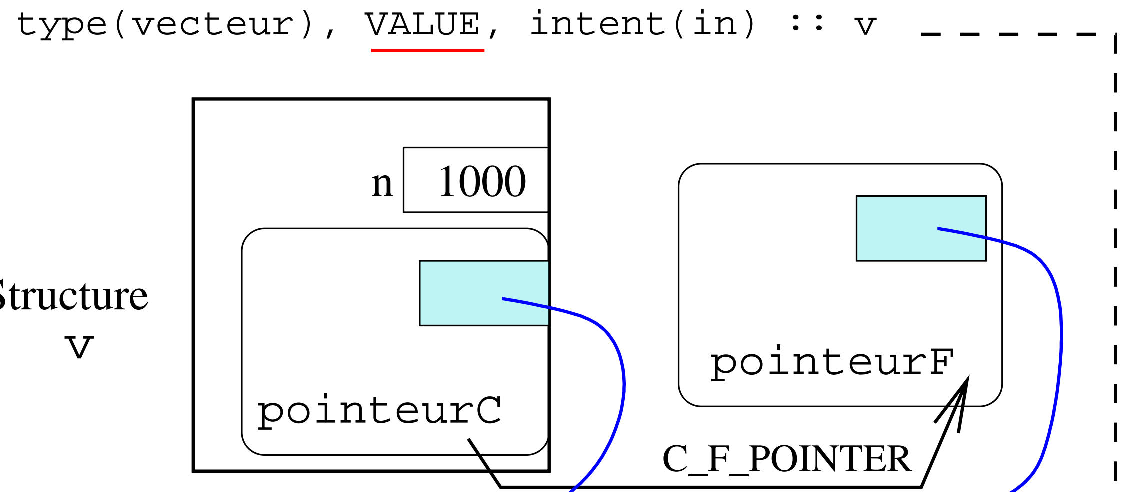
Appelant C	Fonction Fortran appelée

Arguments d'appel

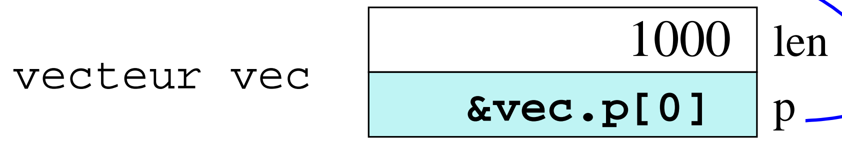


Arguments muets

`type(vecteur), VALUE, intent(in) :: v`



Prototype C



Appel : `F_val(vec)`

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

Voici les modifications à apporter :

```
void F_val(vecteur *);

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

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

Notes personnelles...

3 – Arithmétique IEEE et traitement des exceptions

3.1 – Standard IEEE-754

Le standard `IEEE-754` concernant l'arithmétique réelle flottante ainsi que le traitement des exceptions définit un système de représentation des nombres flottants.

3.1.1 – Valeurs spéciales

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.

3.1.2 – 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.

3.1.3 – 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 !

3.2 – Intégration standard IEEE : 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.

3.3 – 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`.

```
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
```

3.4 – 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)`,

```
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 )
```

3.5 – 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 d'utilisation

```
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)`

```
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 )
```

3.6 – 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)

```
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
```

3.7 – 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.

Voici un exemple montrant comment sauvegarder puis restaurer l'ensemble de ces drapeaux :

```
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 )
!       . . .
```

3.8 – 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 ) !<== Procédure élémentaire
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
```

3.9 – 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.

3.9.1 – 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

3.9.2 – 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 suivants :

- ☞ 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

3.9.3 – 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'intruction `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
```

3.10 – 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
```

Note : fonctionnalités faisant déjà partie des extensions du compilateur `Fortran d'IBM` depuis la version `9.1.0`.

Notes personnelles...

4 – Nouveautés concernant les tableaux dynamiques

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 est donc normal qu'un tableau dynamique ait maintenant les avantages du pointeur vis-à-vis du passage en paramètre de procédure et des composantes dynamiques de structures de données ainsi que de nouvelles possibilités abordées 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, ...

4.1 – Passage en argument de procédure

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 pouvant 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. Voici un exemple :

```
program alloca
  real,dimension(:), ALLOCATABLE :: tab
  call sp(tab)
CONTAINS
  subroutine sp(t)
    real,dimension(:), ALLOCATABLE, intent(inout) :: t
    ALLOCATE(t(256))
    call random_number(t)
  end subroutine sp
end program alloca
```

Remarques :

- 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).
- Comme c'était déjà le cas pour les pointeurs de tableaux en `Fortran 95`, les bornes inférieures/supérieures des dimensions d'un tableau allouable passé en argument sont récupérables dans l'unité appelée dans le cas où celui-ci a été préalablement alloué dans l'unité appelante.
- Quand un objet de type dérivé est désalloué, toute composante ayant l'attribut `ALLOCATABLE` est automatiquement désallouée. Si un destructeur (*final subroutine* – cf. §8.5) est attaché à l'objet, il est appliqué avant la désallocation de cette composante.

4.2 – Composante allouable d'un type dérivé

L'attribut ALLOCATABLE est autorisé pour une composante. Voici un exemple :

```
type obj_mat
  integer :: N, M
  real,dimension(:,:),ALLOCATABLE :: A
end type obj_mat
. . .
type(obj_mat) :: MAT1
. . .
read *,       MAT1%N, MAT1%M
allocate(MAT1%A(MAT1%N, MAT1%M))
. . .
```

C'est bien entendu le descripteur du tableau allouable qui sera stocké dans la composante A de MAT1.

4.3 – Allocation d'un scalaire ALLOCATABLE

L'attribut `ALLOCATABLE` peut dorénavant s'appliquer à un **scalaire**. Voici un exemple :

```
character(len=:), allocatable :: ch
integer                       :: unit, taille
  . . .
read( UNIT=unit ) taille
allocate( character(len=taille) :: ch )
read( UNIT=unit ) ch
  . . .
```

Dans cet exemple, la taille de la chaîne de caractères `ch` est définie à l'exécution et fournie ensuite au niveau de l'instruction `ALLOCATE` en explicitant le type.

4.4 – 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    !<-- Scalaire character(len=:)
 . . . .
NAME = 'chaine_de_car'//'acteres'; . . . .;  NAME = 'FIN'
```

La variable scalaire `NAME` de type `CHARACTER` sera allouée lors de la première affectation avec une longueur `LEN=20`. Lors de la 2e affectation, elle sera désallouée puis réallouée avec une longueur `LEN=3`.

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 se préoccuper du nombre d'éléments vrais de `masque`.

Note : ce processus d'allocation/réallocation automatique peut être inhibé en codant par exemple :

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

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

4.5 – Sous-programme MOVE_ALLOC de réallocation

`MOVE_ALLOC(FROM, TO)` réalloue à TO l'objet alloué passé via FROM.

- FROM entité allouable de n'importe quel type/rang. Sa vocation est INTENT(INOUT),
- TO entité allouable compatible (type et rang) avec FROM. Sa vocation est INTENT(OUT).

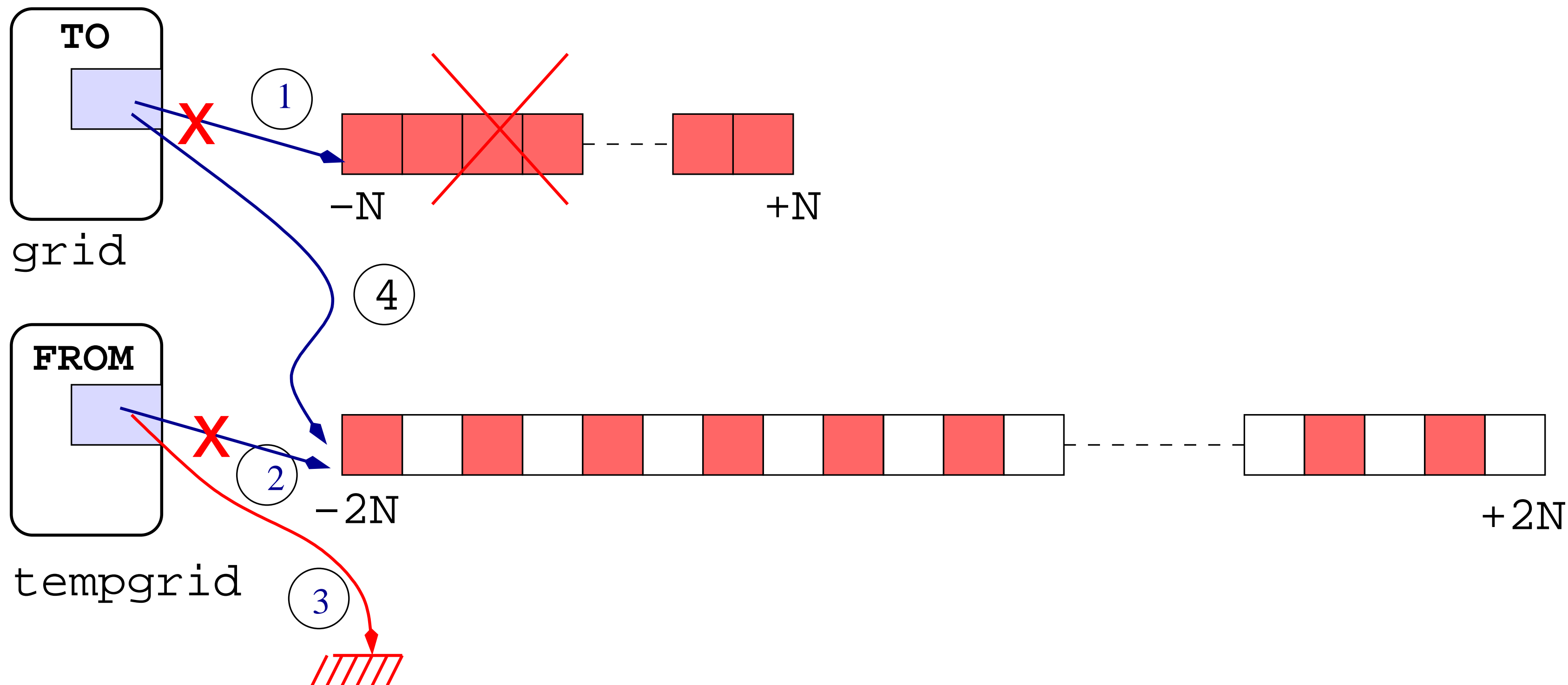
En retour de ce sous-programme, le tableau allouable TO désigne le tableau allouable FROM ; la zone mémoire préalablement désignée par TO est désallouée.

En fait, c'est une *méthode* permettant de *nettoyer* 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.

– Si TO a l'attribut TARGET, tout pointeur initialement associé à FROM devient associé à TO.

```
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)
```



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(:) )
```

5 – Nouveautés concernant les modules

5.1 – 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.

5.2 – 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 :

```
module truc
  type couleur
    character(len=16)  :: nom
    real, dimension(3) :: compos
  end type couleur

  interface
    function demi_teinte(col_in)
      IMPORT :: couleur
      type(couleur), intent(in) :: col_in
      type(couleur)             :: demi_teinte
    end function demi_teinte
  end interface
contains
  . . . !---> Partie procédurale du module
end module truc
```

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

5.3 – 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.)
```

Notes personnelles...

6 – Entrées-sorties

6.1 – 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 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.

```
INTEGER(kind=2) :: unit, iostat
READ( UNIT=unit, ..., IOSTAT=iostat ) ...
IF( is_iostat_end( iostat ) ) then
     . . .
END IF
```

6.2 – 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 effectué.

Une synchronisation peut être demandée explicitement à l'aide de l'instruction `WAIT(unit=..., ...)`. Celle-ci est implicite à la rencontre d'un `INQUIRE` ou d'un `CLOSE` sur le fichier.

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
```

6.3 – 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 =` octet ou 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` permet d'indiquer 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.
    . . .
write( UNIT=1, POS=rang ) d+1 ! par rapport à la position contenue dans rang.
```

Note : sous `UNIX/LINUX` les fichiers ne contiennent pas de marque physique de fin de fichier. C'est une notion purement logique. De ce fait le dernier enregistrement d'un fichier accédé en mode *stream* est considéré comme incomplet. En conséquence, si on lit plus de données que cet enregistrement n'en contient une fin de fichier sera détecté comme le montre l'exemple suivant :

```
program lire
  character(len=80) enreg
  integer ios

  open( UNIT=1,        FILE="data", ACCESS="stream", &
        ACTION="read", FORM="formatted" )
  do
    read( unit=1, fmt='(a)', iostat=ios ) enreg
    if ( ios /= 0 ) exit
    print '(a)', trim(enreg)
  end do
  close( unit=1 )
end program lire
```

Contenu supposé du fichier `data` :

```
Wolfgang Amadeus Mozart
Ludwig-Van Beethoven
Gustav Mahler
```

Ce programme produit la sortie suivante :

```
Wolfgang Amadeus Mozart
Ludwig-Van Beethoven
```

6.4 – 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 d'insérer un bloc interface au niveau de la définition du type :

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

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 facultativement indiqués 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 procédures doit respecter le prototype suivant :

```
SUBROUTINE lecture_format  (dtv, unit, iotype, v_list, iostat, iomsg )
SUBROUTINE ecrit_nonformat (dtv, unit,                 iostat, iomsg )
```

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

unit : 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,

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

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

Voici un exemple avec des enregistrements formatés :

```
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_clair", compos=[ 0.8, 0.8, 0.8 ] )
  c2 = couleur( nom="gris_fonce", compos=[ 0.2, 0.2, 0.2 ] )
  i = 1756; x = 9.81; y = 2.718
  WRITE( UNIT=1, FMT="(I2, DT 'COMPLET'(6), 2F6.2 )", &
         IOSTAT=ios, IOMSG=message ) i, c1, x, y
  if( ios /= 0 ) PRINT *, ios, message
  WRITE( UNIT=1, FMT="(F7.3, DT(7), DT 'COMPLET'(5) )",  &
         IOSTAT=ios, IOMSG=message ) x, c1, c2
  if( ios /= 0 ) PRINT *, ios, message
  WRITE( UNIT=1, FMT=*, IOSTAT=ios, IOMSG=message ) x, c1, i, y
  if( ios /= 0 ) PRINT *, ios, message
  WRITE( UNIT=1, NML=liste )
  if( ios /= 0 ) PRINT *, ios, message
END PROGRAM exemple
```

```
MODULE couleur_mod
  TYPE couleur
    CHARACTER(len=16) :: nom; REAL, DIMENSION(3) :: compos
    CONTAINS
      PROCEDURE :: ecr_format
      GENERIC :: WRITE(FORMATTED) => ecr_format ! <=== Generic binding.
  END TYPE couleur
CONTAINS
  SUBROUTINE ecr_format( dtv, unit, iotype, v_list, iostat, iomsg )
    CLASS(couleur),        INTENT(in)    :: dtv
    INTEGER,               INTENT(in)    :: unit
    CHARACTER(len=*),      INTENT(in)    :: iotype
    INTEGER, DIMENSION(:), INTENT(in)    :: v_list
    INTEGER,               INTENT(out)   :: iostat
    CHARACTER(len=*),      INTENT(inout) :: iomsg
    CHARACTER(len=10), DIMENSION(2)      :: fmt
    write( fmt(1), '(a, i2, a)' ) "(a,3f", v_list(1), ".2)"
    write( fmt(2), '(a, i2, a)' ) "(3f", v_list(1), ".2)"
    SELECT CASE( iotype )
      CASE ('NAMELIST')    ; WRITE( UNIT=unit, FMT=* )      dtv%nom, dtv%compos
      CASE ('LISTDIRECTED'); WRITE( UNIT=unit, FMT=* )      dtv%nom, dtv%compos
      CASE ('DTCOMPLET')   ; WRITE( UNIT=unit, FMT=fmt(1) ) dtv%nom, dtv%compos
      CASE DEFAULT         ; WRITE( UNIT=unit, FMT=fmt(2) ) dtv%compos
    END SELECT
  END SUBROUTINE ecr_format
END MODULE couleur_mod
```

Exemple de traitement d'un fichier binaire

```
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 *, message
    STOP 4
  END IF
END PROGRAM exemple
```

```
MODULE couleur_mod
  TYPE couleur
    CHARACTER(len=16)   :: nom
    REAL, DIMENSION(3) :: compos
    CONTAINS
      PROCEDURE :: lec_binaire
      GENERIC :: READ(UNFORMATTED) => lec_binaire ! <=== Generic binding.
  END TYPE couleur
CONTAINS
    SUBROUTINE lec_binaire( dtv, unit, iostat, iomsg )
      CLASS(couleur),         INTENT(inout) :: dtv
      INTEGER,                INTENT(in)    :: unit
      INTEGER,                INTENT(out)   :: iostat
      CHARACTER(len=*),       INTENT(inout) :: iomsg

      READ( UNIT=unit, IOSTAT=iostat, IOMSG=iomsg ) dtv%nom, dtv%compos
    END SUBROUTINE lec_binaire
END MODULE couleur_mod
```

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, unit, iotype, v_list, iostat, iomsg )
    CLASS(node),           INTENT(in)    :: dtv
    INTEGER,               INTENT(in)    :: unit
    CHARACTER(len=*),      INTENT(in)    :: iotype
    INTEGER, DIMENSION(:), INTENT(in)    :: v_list
    INTEGER,               INTENT(out)   :: iostat
    CHARACTER(len=*),      INTENT(inout) :: iomsg

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

Ce module pourra s'employer de la façon suivante :

```
PROGRAM liste
  USE liste_module
  INTEGER    :: unit, iostat
  TYPE(node) :: racine
      . . .
  ! Création d'une liste chaînée avec racine comme noeud primaire.
      . . .
  ! Impression de la liste.
  WRITE( UNIT=unit, FMT='(dt)', IOSTAT=iostat ) 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=unit, FMT='(/,dt)', IOSTAT=iostat ) dtv%next_node
```

Notes personnelles...

7 – Pointeurs

7.1 – 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 ;
- `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.

```
subroutine sp(p1, p2, ...)
 real, dimension(:,:), pointer, INTENT(IN)  :: p1
 real, dimension(:,:), pointer, INTENT(OUT) :: p2
 . . . .
```

Note : fonctionnalité faisant partie des extensions du compilateur Fortran d'IBM depuis la version 95.

7.2 – Association et reprofilage

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`.

Exemples

```
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
```

```
! 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( nl, nc )
  real, dimension(nl*nc), target :: vec
  real, dimension(:,:), pointer  :: matrice
  real, dimension(:), pointer    :: diag

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

7.3 – 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.

7.3.1 – Pointeurs de procédure : interface implicite

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

☞ Première possibilité (pour une fonction seulement) avec l'attribut POINTER :

```
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(TYPE(obj_mat(k=8, dim=256, d=128))), POINTER :: p
```

p ne pourra alors être associé qu'à une fonction retournant un objet de type obj_mat paramétré comme indiqué (cf. §8.2 - Paramètres d'un type dérivé).

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.

7.3.2 – Pointeurs de procédure : interface explicite

☞ Attribut `POINTER` appliqué à une déclaration de procédure effectuée à l'aide d'un bloc interface :

```
module m
  POINTER :: sp
  interface
    subroutine sp( a, b )
      real, intent(inout) :: a
      real, intent(in)    :: b
    end subroutine sp
  end interface
contains
  subroutine trace( u, v )
    real, intent(inout) :: u
    real, intent(in)    :: v
      . . .
  end subroutine trace
end module m
program prog
  use m
  sp => trace
  call sp( ... )
end program prog
```

☞ 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) :

```
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***) comme celui-ci :

```
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 :

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

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

Notes personnelles...

8 – Nouveautés concernant les types dérivés

8.1 – 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*). 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.

```
module m
  TYPE mytype
    private
      real    :: x
      integer :: i
      PROCEDURE(proc), public, POINTER, PASS :: p
  END TYPE mytype
  abstract interface
    subroutine proc( this, r, i )
      type(mytype), intent(inout) :: this
      real,         intent(in)  :: r
      integer,      intent(in)  :: i
    end subroutine proc
  end interface
end module m
program prog
  use m
  TYPE(mytype) :: a

  a%p => p1
  call a%p( 3.14,  100 ) ! équivalent à "call p1( a, 3.14, 100 )"
end program prog
```

8.2 – Paramètres d'un type dérivé

En Fortran 95, les types intrinsèques étaient déjà paramétrables ; en particulier le type `CHARACTER(LEN= , KIND= )` avait 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 doit être connu à la compilation, tandis que le deuxième l'est ; Fortran 2003 généralise cette distinction aux paramètres des types dérivés qui pourront être déclarés avec l'**attribut** `LEN` ou `KIND`.

Les paramètres `LEN/KIND` peuvent être utilisés pour définir la longueur de chaînes ou les bornes de tableaux, mais seuls ceux ayant l'attribut `KIND` (dont la valeur doit être connue à la compilation) peuvent intervenir dans une expression d'initialisation ou la valorisation d'un sous-type (*kind-selector*).

Des valeurs par défaut peuvent être définies.

Voici un exemple de type dérivé paramétrable :

```
type obj_mat(k, dim, d)
  !---------------------------------------
  integer, KIND                       :: k, dim
  integer, LEN                        :: d
  !---------------------------------------
  integer                             :: nb_bits=k*8
  real(kind=k),dimension(d,d)          :: tab
  real(kind=k),dimension(dim)         :: vect
end type obj_mat
. . . .
type(obj_mat(k=8, dim=256, d=128)) :: mat
. . . .
```

Les paramètres (ici **k**, **dim** et **d**) 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*.

8.3 – 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 :

```
type couleur
  character*16 :: nom
  real,dimension(3) :: compos
end type couleur
. . .
type(couleur) :: c
. . .
c=couleur(nom='rose_saumon', compos=[ 0.72, 0.33, 0.05 ] )
. . .
```

Notez aussi la nouvelle notation du constructeur de vecteur encadré par des crochets (au lieu des caractères `(/` et `/)`).

Une procédure générique peut avoir le même nom qu'un constructeur de structure. Les constituants de la famille générique ont priorité sur le constructeur en cas d'ambiguité. Cette technique peut être employée afin de surcharger un constructeur, comme le montre l'exemple suivant :

```
type mycomplex
  real :: rho, theta                          !<=== Coordonnées polaires
end type mycomplex
interface mycomplex
  type(mycomplex) function complex_to_mycomplex(c)
    complex, intent(in) :: c                  !<=== Type intrinsèque Fortran
  end function complex_to_mycomplex
  type(mycomplex) function two_reals_to_mycomplex(x, y)
    real, intent(in)           :: x           !
    real, intent(in), optional :: y           !<=== Coordonnées cartésiennes
  end function two_reals_to_mycomplex
end interface
  . . .
type(mycomplex) :: a, b, c
complex         :: w
  . . .
a = mycomplex( theta=5.6, rho=1.0 )           ! Appel constructeur intrinsèque
b = mycomplex( w )                            ! Appel à complex_to_mycomplex
c = mycomplex( x=0.0, y=1.0 )                 ! Appel à two_reals_to_mycomplex
```

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.

Exemple

```
type real_liste
  real                        :: valeur
  type(real_liste), pointer :: next => null()
end type real_liste
    . . .
type(real_liste) :: x = real_liste(3.14)
    . . .
```

Exemple d'un type paramétré

```
type champ_vecteur(n,d,k)
  integer, kind                  :: k = kind(1.0)
  integer, len                   :: d = 2, n
  real(kind=k), dimension(d,n) :: champ
end type champ_vecteur
   . . .
type(champ_vecteur(k=selected_real_kind(9,99), n=100)) :: c
   . . .
c = champ_vecteur(k=selected_real_kind(9,99), n=100)(champ=real(1., kind=c%k))
```

`NOTE` : on remarque qu'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
  real(selected_real_kind(9,99)), dimension(10) :: t

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

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 par exemple :

```
subroutine impression_champ( c )
  type(champ_vecteur(k=selected_real_kind(9,99), d=*, n=*)) :: c
    . . .
  print *,c%d, c%n
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 :

```
type(champ_vecteur(k=selected_real_kind(9,99), d=:, n=:)),   pointer :: p
type(champ_vecteur(k=selected_real_kind(9,99), d=3, n=200)), target  :: cible
  . . .
p => cible
```

8.4 – Visibilité des composantes

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

```
type struct_publique
  integer               :: i, j
  real,dimension(160) :: tab
end type struct_publique
```

```
type, private :: struct_privee
  integer               :: i, j
  real,dimension(160) :: tab
end type struct_privee
```

```
type struct_semi_privee
  private
  integer               :: i, j
  real,dimension(160) :: tab
end type struct_semi_privee
```

En Fortran 2003, la privatisation peut se gérer plus finement au niveau de chaque composante.

```
module m
  type t
    private
    integer          :: i
    integer, public :: j
    real             :: k
  end type t
    . . .
end module m
program p
  use m
  type(t) obj

  obj%i = ...     ! invalide : i est privé au module
  obj%j = ...     ! valide   : j est publique
  print *, obj%k ! invalide : k est privé au module
    . . .
end program p
```

`Remarque` : les identificateurs `i` et `k` ne sont accessibles qu'au sein du module `m`.

8.5 – Extension d'un type dérivé

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`.

```
type base_type
  integer :: i
  integer :: j
end type base_type
type, extends(base_type) :: my_type
  real    :: r
  logical :: l
end type my_type
  . . .
type(my_type) :: x
  . . .
x%base_type%i = ... ! peut s'écrire : x%i = ...
x%r = ...
x%l = .false.
  . . .
```

La notation `x%base_type%i = ...` pourrait s'avérer utile pour lever une ambiguïté d'homonymie de nom de composantes après héritage...

Extension d'un type paramétré

Lorsqu'un type paramétré est étendu, le nouveau type hérite des paramètres du type qu'il complète et peut en définir de nouveaux.

```
type champ_vect(n,d,k)
  integer, kind                :: k = kind(1.0)
  integer, len                 :: d = 2, n
  ! - - - - - - - - - - - - - -
  real(kind=k), dimension(d,n) :: champ = 0.
end type champ_vect

type, extends(champ_vect) :: champ_vect_label(ln)
  integer, len      :: ln
  ! - - - - - - - -
  character(len=ln) :: label=""
end type champ_vect_label
  . . .
type(champ_vect_label(k=kind(1.d0), d=3, n=100, ln=30)) :: c1
  . . .
c1 = champ_vect_label(k=kind(1.d0), d=3, n=100, ln=30)( label="champ de vitesses" )
  . . .
```

Notes personnelles...

9 – Programmation orientée objet

9.1 – Variable polymorphique

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.

```
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.

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.

9.1.1 – 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. De façon plus générale, voici les règles d'association **arguments réels** <==> **arguments muets** suivant que ceux-ci sont polymorphiques ou non, illustrées à l'aide de l'exemple ci-dessous :

```
module m
  type point2d
    real x, y
  end type point2d
  type, extends(point2d) :: point2d_coul
    real, dimension(3) :: compos_rvb
  end type point2d_coul
  type, extents(point2d) :: point3d
    real z
  end type point3d
  type, extends(point3d) :: point3d_coul
    real, dimension(3) :: compos_rvb
  end type point3d_coul
```

```
contains
  function distance2d_poly( p1, p2 ) ! Calcul de la distance entre les points p1 et p2
    CLASS(point2d) p1, p2
    real          distance2d
       . . .
  end function distance2d
  function distance3d_poly( p1, p2 ) ! Calcul de la distance entre les points p1 et p2
    CLASS(point3d) p1, p2
    real          distance3d
       . . .
  end function distance3d
  function distance2d_fixe( p1, p2 ) ! Calcul de la distance entre les points p1 et p2
    TYPE(point2d) p1, p2
    real           distance2d
       . . .
  end function distance3d
  function distance3d_fixe( p1, p2 ) ! Calcul de la distance entre les points p1 et p2
    TYPE(point3d) p1, p2
    real           distance3d
       . . .
  end function distance3d
end module m
```

```
program p
  use m
  TYPE(point2d) :: A_p2d, B_p2d; CLASS(point2d) :: A_p2d_poly, B_p2d_poly
  TYPE(point3d) :: A_p3d, B_p3d; CLASS(point3d) :: A_p3d_poly, B_p3d_poly
  real distance
  ! Argument muet polymorphique, argument réel d'un type fixé :
  distance = distance2d_poly( A_p2d, B_p2d ) %{\overlay{1}\rnode{NodeA}}
  distance = distance2d_poly( A_p3d, B_p3d )                          ! valide
  distance = distance3d_poly( A_p2d, B_p2d )                          ! invalide
  distance = distance3d_poly( A_p3d, B_p3d )                          ! valide
  ! Argument muet d'un type fixé, argument réel polymorphique :
  distance = distance2d( A_p2d_poly, B_p2d_poly )                     ! valide
  distance = distance2d( A_p3d_poly, B_p3d_poly )                     ! invalide
  distance = distance2d( A_p3d_poly%point2d, B_p3d_poly%point2d )     ! valide
  distance = distance3d( A_p2d_poly, B_p2d_poly )                     ! invalide
  select type( A_p2d_poly )
    class is ( point3d )
      distance = distance3d( A_p2d_poly, B_p2d_poly )                 !
    class default
  end select
  distance = distance3d( A_p3d_poly, B_p3d_poly )                     ! valide
  ! Argument muet et argument réel polymorphiques :
  distance = distance2d_poly( A_p2d_poly, B_p2d_poly )                ! valide
  distance = distance2d_poly( A_p3d_poly, B_p3d_poly )                ! valide
  distance = distance3d_poly( A_p2d_poly, B_p2d_poly )                ! invalide
end program p
```

9.1.2 – Variable polymorphique : attribut POINTER, ALLOCATABLE

Pointeur polymorphique : variable polymorphique ayant l'attribut POINTER ; son type dynamique est celui de sa cible qui peut être définie lors d'une association ou d'une allocation dynamique (ALLOCATE).

De même, le type dynamique d'une variable polymorphique ayant l'attribut ALLOCATABLE est celui fourni lors de son allocation.

```
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      ! Le type dynamique de ptr2d_1 est TYPE(point2d)
ptr2d_2 => p3d      ! Le type dynamique de ptr2d_2 est TYPE(point3d)
ptr3d   => p3d      ! Le type dynamique de ptr3d   est TYPE(point3d)
ptr2d_2 => ptr2d_1  ! Le type dynamique de ptr2d_2 est celui de ptr2d_1
ptr3d   => ptr2d_1  ! Interdit
ALLOCATE( ptr2d_1 ) ! Alloue un objet de type dynamique TYPE(point2d)
                    ! et associe ptr2d_1 avec.
ALLOCATE( TYPE(point3d)::ptr2d_2 ) ! Alloue un objet de type dynamique TYPE(point3d)
                                   ! et associe ptr2d_2 avec.
ALLOCATE( TYPE(point3d_coul) :: point )
```

9.2 – Construction SELECT TYPE

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

```
subroutine sp( a, b )
  use m
  CLASS(point2d) :: a, b
     . . .
  select type (a)
    type is(point2d_coul)
          . . .
    class is(point3d)
          . . .
    class is(point3d_coul)
          . . .
    class default
          . . .
  end select
end subroutine sp
```

C'est le type dynamique de l'objet `a` 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, 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 riche qui est exécuté,
- ☞ sinon, s'il existe une instruction `CLASS DEFAULT`, c'est son bloc qui est exécuté.

9.3 – Pointeurs génériques

Il est possible de définir un pointeur générique ou *unlimited polymorphic pointer* en précisant le caractère $*$ à la place du nom du type au niveau du mot-clé `CLASS`. Celui-ci peut ensuite être associé à tout type de cible.

```
function  func( p, ... )
  CLASS(*), intent(in), pointer :: p
    . . .
  select type( p )
    type is( selected_int_kind(9) )
       . . .
    type is( real )
       . . .
    type is( double precision )
       . . .
    type is( character(len=*) )
       . . .
    class default
      print *,"Type invalide."
  end select
end function func
```

9.4 – 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 )
```

Retourne **vrai** si `a` et `b` ont le même type dynamique.

```
EXTENDS_TYPE_OF( a, mold )
```

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
```

9.5 – 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.

9.5.1 – Procédure attachée par nom (*name binding*)

```
module truc
  type T
    . . . !--> Déclaration des composantes
    . . . !--> du type dérivé T
  contains
    PROCEDURE :: proc => my_proc
  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 truc

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

9.5.2 – Procédure attachée par nom générique (*generic binding*)

```
module m
  type matrix(k, n, m)
    integer, kind :: k
    integer, len  :: n, m
    real(kind=k), dimension(n,m) :: A
  contains
    PROCEDURE, PRIVATE :: max4, max8
    GENERIC :: max => max_4, max_8
  end type matrix
contains
  real(kind=4) function max_4( this )
    class(matrix(k=4, n=*, m=*), intent(int) :: this
    max_4 = MAXVAL( array=this%A )
  end function max_4
  real(kind=8) function max_8( this )
    class(matrix(k=8, n=*, m=*), intent(int) :: this
    max_8 = MAXVAL( array=this%A )
  end function max_8
end module m
program prog
  type(matrix(k=4, n=10, m=20) :: obj1; type(matrix(k=8, n=20, m=50) :: obj2
  real(kind=4) max4; real(kind=8) max8
  max4 = obj1%max(); max8 = obj2%max()
end program prog
```

9.5.3 – Procédure attachée par opérateur (*operator binding*)

```
module m
  type matrix(k, n, m)
    integer, kind                 :: k
    integer, len                  :: n, m
    real(kind=k), dimension(n,m) :: A
  contains
    PROCEDURE, PRIVATE :: add4, add8, affect4, affect8
    GENERIC :: OPERATOR(+)     => add4, add8
    GENERIC :: ASSIGNMENT(=) => affect4, affect8
  end type matrix
contains
  function add4( a, b )
    class(matrix(k=4, n=*, m=*), intent(in)  :: a, b
    type(matrix(k=4, n=:, m=:), allocatable  :: add4 ! "type" et non "class".
    if( shape(a%A) /= shape(b%A) ) stop "Erreur : objets non conformants"
    allocate( matrix(k=4, n=a%n, m=a%m) :: add4 )
    add4%A(:,:) = a%A(:,:) + b%A(:,:)
  end function add4
```

```
  function add8( a, b )
    class(matrix(k=8, n=*, m=*), intent(in) :: a, b
    type(matrix(k=8, n=a%n, m=a%m)          :: add8 ! "type" et non "class".
    if( shape(a%A) /= shape(b%A) ) stop "Erreur : objets non conformants"
    add8%A(:,:) = a%A(:,:) + b%A(:,:)
  end function add8
  !---------------------------------
  subroutine affect4( a, b )
    class(matrix(k=4, n=*, m=*), intent(inout) :: a
    class(matrix(k=4, n=*, m=*), intent(in)    :: b
    if( shape(a%A) /= shape(b%A) ) stop "Erreur : objets non conformants"
    a%A(:,:) = b%A(:,:)
  end function affect4
  !---------------------------------
  subroutine affect8( a, b )
    class(matrix(k=8, n=*, m=*), intent(inout) :: a
    class(matrix(k=8, n=*, m=*), intent(in)    :: b
    if( shape(a%A) /= shape(b%A) ) stop "Erreur : objets non conformants"
    a%A(:,:) = b%A(:,:)
  end function affect8
end module m
```

```
program prog
  use m
  type(matrix(k=kind(0.d0), n=5,  m=10)) :: mat1d, mat2d, mat3d
  type(matrix(k=kind(0.),   n=20, m=60)) :: mat1s, mat2s, mat3s
      . . .
  mat1s = mat2s + mat3s ! Appel à add4 puis affect4
      . . .
  mat1d = mat2d + mat3d ! Appel à add8 puis affect8
end program prog
```

9.5.4 – Procédure attachée via le 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.

```
module m
  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=4)) :: x
    if( associated(x%v) ) deallocate(x%v)
  end subroutine finalize_scal
  subroutine finalize_vect(x) !--> Arg. vecteur
    type(t(k=4)), dimension(:) :: x
    do i=1,size(x)
      if( associated(x(i)%v) ) deallocate(x(i)%v)
    end do
  end subroutine finalize_vect
end module m
```

Ainsi, lors de la désallocation de l'objet `obj` déclaré ainsi :

```
type(t(k=4)), dimension(:), allocatable :: obj
```

c'est le destructeur `finalize_vect` qui sera exécuté.

9.6 – Héritage

9.6.1 – Héritage d'une procédure *type-bound*

Un type étendu d'un type extensible hérite à la fois de ses composantes mais également de ses procédures *type_bound*.

```
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
```

Voici un exemple d'héritage de procédure (ici `affichage`) :

```
module pointcoul
  use point
  private
  type, public, extends(point2d) :: point2d_coul
    real, dimension(3) :: compos_rvb
  end type
end module pointcoul
!----------------------------
program prog
  use pointcoul
  type(point2d_coul) :: pcoul
  call pcoul%affichage( "Voici mes coordonnées" )
end program prog
```

9.6.2 – Surcharge d'une procédure *type-bound*

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

```
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 prog
  use pointext
  type(point3d) p
  call p%affichage( "Voici mes coordonnées" )
end program prog
```

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.

```
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
    call p%affichage( "Voici mes coordonnées" ) !--> Polymorphisme dynamique
  end subroutine affiche
end program dynamic
```

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

9.6.3 – Procédure *type-bound* 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.

```
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
    real_part = a%rho*sin(a%theta)
  end function imag_part
end module m
```

9.7 – 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. 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 numerique
  type, abstract :: mon_type_numerique
  contains
    procedure(func_oper), PASS(b), DEFERRED :: add
    procedure(func_oper), DEFERRED :: mult
    procedure(sub_oper),  DEFERRED :: affect
    generic, public :: operator(+)    => add
    generic, public :: operator(*)    => mult
    generic, public :: assignment(=) => affect
  end type mon_type_numerique
  abstract interface
    function func_oper( a, b ) result(r)
      import mon_type_numerique
      class(mon_type_numerique), intent(in)   :: a, b
      class(mon_type_numerique),  allocatable :: r
    end function func_oper
    subroutine sub_oper( a, b )
      import mon_type_numerique
      class(mon_type_numerique), intent(inout) :: a
      class(mon_type_numerique), intent(in)    :: b
    end subroutine sub_oper
  end interface
end module numerique
```

```
module entier
  use numerique
  type, extends(mon_type_numerique) :: mon_entier
    integer, private :: valeur
  contains
    procedure, PASS(b) :: add    => add_mon_entier
    procedure :: mult   => mult_mon_entier
    procedure :: affect => affect_mon_entier
  end type mon_entier
contains
  function mon_entier( val )
    integer, intent(in) :: val
    type(mon_entier) :: mon_entier

    mon_entier%valeur = val
  end function mon_entier
```

```
function add_mon_entier( a, b ) result(r)
  class(mon_type_numerique), intent(in)  :: a
  class(mon_entier),         intent(in)  :: b
  class(mon_type_numerique), allocatable :: r
  allocate( mon_entier :: r )
  select type(a)
    type is( mon_entier )
      select type(r)
        type is( mon_entier )
          r%valeur = a%valeur + b%valeur
      end select
  end select
end function add_mon_entier
function mult_mon_entier( a, b ) result(r)
  class(mon_entier),          intent(in)  :: a
  class(mon_type_numerique),  intent(in)  :: b
  class(mon_type_numerique),  allocatable :: r
  allocate( mon_entier :: r)
  select type(b)
    type is( mon_entier )
      select type(r)
        type is( mon_entier )
          r%valeur = a%valeur * b%valeur
      end select
  end select
end function mult_mon_entier
```

```
  subroutine affect_mon_entier( a, b )
    class(mon_entier), intent(inout) :: a
    class(mon_entier), intent(in)    :: b

    a%valeur = b%valeur
  end subroutine affect_mon_entier
  subroutine imp( a )
    class(mon_entier), intent(in)    :: a

    print *, "Valeur = ", a%valeur
  end subroutine imp
end module entier

program prog
  use entier
  type(mon_entier) :: int1, int2, int3

  int1 = mon_entier( 2 )
  int2 = mon_entier( 1756 )
  int3 = mon_entier( 1791 )
  int1 = (int2 + int3)*int1 ! Implique l'appel aux procédures :
                            !   add_mon_entier, mult_mon_entier puis affect_mon_entier
  call imp( int1 )
end program prog
```

10 – En conclusion

- ☞ La phase d'élaboration technique de la norme Fortran 2003 est terminée ainsi que la phase ISO FCD (*Final Committee Draft*) dont le document ISO-IEC/JTC1/SC22/WG5/N1578 (draft 04-007) est consultable en ligne à l'adresse :
  `http://www.j3-fortran.org`
  à la rubrique Fortran 2003.

- ☞ La norme Fortran 2003 est désormais officielle depuis septembre 2004.

- ☞ En attendant l'arrivée des premiers compilateurs Fortran 2003, certains fournisseurs comme IBM, Nec, Nag, N.A. Software, ... intègrent progressivement certains aspects de cette norme.

Index

– Symboles –

type-bound procedure : : generic binding 153
type-bound procedure : : name binding 152
type-bound procedure : : operator binding 154
type-bound procedure 151

– A –

ABSTRACT INTERFACE 124, 127
ACCESS - STREAM 105
adresse cible : C_LOC(X) 20
affectation et allocation automatique 90, 91
ALLOCATABLE 85
allocatable 45
ALLOCATABLE - argument de procédure 86
ALLOCATABLE - composante type dérivé 88
ALLOCATABLE - scalaire 89
ALLOCATABLE : argument INTENT(OUT) 87
ALLOCATABLE : argument SAVE 87
allocation via affectation 90
argument de procédure : ALLOCATABLE 86
arguments - ligne de commande 11
arrondi 58
association : pointeur de procédure 119
asynchrones - E./S. 102
ASYNCHRONOUS 102

– B –

BIND(C) 18, 22, 23, 45, 46, 50
bloc interface : IMPORT 96
bound procedure - surcharge 161

– C –

C_ALERT 15
C_ASSOCIATED 21
C_BACKSPACE 15
C_CARRIAGE_RETURN 15
C_CHAR 14
C_DOUBLE 14
C_F_POINTER 47
C_F_PROCPOINTER 21
C_FLOAT 14
C_FORM_FEED 15
C_FUNLOC 21
C_FUNPTR 21, 45
C_HORIZONTAL_TAB 15
C_INT 14
C_LOC 20, 21, 47
C_LONG 14
C_NEW_LINE 15
C_NULL_CHAR 15
C_PTR 19–21, 45, 46, 50
C_SHORT 14
C_VERTICAL_TAB 15
calloc 48, 51
CHARACTER : interopérabilité 14
CLASS 141
CLASS() 161
CLOSE 102
COMMAND_ARGUMENT_COUNT 11
common : interopérabilité 18

composante ALLOCATABLE : destructeur 87
constructeur de type dérivé 130

– D –

dénormalisée 58
declared type 141
descripteur 85
descripteur de format DT 107
destructeur : composante ALLOCATABLE 87
destructeur type dérivé 157
division par zéro 57
DT - descripteur format 107
dtio-generic-pec 107
dynamic type 141

– E –

Entrées/sorties 102
environnement 11
environnement - variables d' 12
ERROR_UNIT 12
EXTENDS() 161
EXTENDS_TYPE_OF 149

– F –

FINAL 157
final subroutine 87
fonction C 47, 48, 51

– G –

GENERIC 153
GENERIC : : ASSIGNMENT 156
GENERIC : : OPERATOR 156
generic binding 106
generic binding : type dérivé 153
GET_COMMAND 11
GET_COMMAND_ARGUMENT 11
GET_ENVIRONMENT_VARIABLE 11

– H –

héritage - procédure 160

– I –

IEEE_ALL 67, 70–72
IEEE_ARITHMETIC 59
IEEE_CLASS 62
IEEE_CLASS_TYPE 64
IEEE_DIVIDE_BY_ZERO 66, 70
IEEE_EXCEPTIONS 59
IEEE_FEATURES 59
IEEE_GET_FLAG 68, 70
IEEE_GET_HALTING_MODE 70
IEEE_GET_ROUNDING_MODE 65
IEEE_GET_STATUS 72
IEEE_GET_UNDERFLOW_MODE(gradual) 69
IEEE_INEXACT 66, 68
IEEE_INVALID 66
IEEE_IS_FINITE 62
IEEE_IS_NAN 62
IEEE_IS_NEGATIVE 62
IEEE_IS_NORMAL 62
IEEE_NEGATIVE_DENORMAL 61, 64
IEEE_NEGATIVE_INF 61, 64
IEEE_NEGATIVE_NORMAL 61, 64
IEEE_NEGATIVE_ZERO 61, 64
IEEE_OTHER_VALUE 61

IEEE_OVERFLOW 66
IEEE_POSITIVE_DENORMAL 61, 64
IEEE_POSITIVE_INF 61, 64
IEEE_POSITIVE_NORMAL 61, 64
IEEE_POSITIVE_ZERO 61, 64
IEEE_QUIET_NAN 61
IEEE_ROUND_VALUE 65
IEEE_SET_FLAG 68
IEEE_SET_HALTING_MODE 70
IEEE_SET_ROUNDING_MODE 65
IEEE_SET_STATUS 72
IEEE_SET_UNDERFLOW_MODE(gradual) 69
IEEE_SIGNALING_NAN 61
IEEE_STATUS_TYPE 72
IEEE_SUPPORT_DATATYPE 60, 64
IEEE_SUPPORT_DENORMAL 60
IEEE_SUPPORT_INF 60
IEEE_SUPPORT_NAN 60
IEEE_SUPPORT_STANDARD 60
IEEE_UNDERFLOW 66
IEEE_UP 65
IEEE_USUAL 67
IEEE_VALUE 62, 64
IMPORT : bloc interface 96
INPUT_UNIT 12
INQUIRE 105
INTENT - pointeur 116
INTENT(OUT) : argument ALLOCATABLE 87
interface implicite 120
Interface procédure Fortran 22
interopérabilité Fortran-C 13, 18
IOSTAT 12
IOSTAT_END 12
IOSTAT_EOR 12
iso-IEC-1539 2
ISO_C_BINDING 13, 20, 45
ISO_FORTRAN_ENV 12

– K –

KIND : paramètre type dérivé 128, 129
kind type parameters 129

– L –

LEN : paramètre type dérivé 128, 129
length type parameters 129
ligne de commande : arguments 11

– M –

module : importation d'entités 96
module : protection d'entités 95
module : renommage d'opérateurs 97
module : USE 97
MOVE_ALLOC 93

– N –

NAME= 22, 23
NaN 56, 57
NON_OVERRIDABLE 163

– O –

opérateur : renommage via USE 97
operator binding : type dérivé 156
OPERATOR() 97
OUTPUT_UNIT 12
overflow 57
override - type bound procedure 161

– P –

paramètres d'un type dérivé 128
PASS, NOPASS 126
POINTER 85
pointer : interopérabilité 19
pointeur 85
pointeur - procédure 126
pointeur - reprofilage 118
pointeur : vocation 116
pointeur C : valeur 20
pointeur de procédure 21, 119–121, 124
pointeur de procédure : association 119
pointeur générique 148
pointeur polymorphique 145
pointeurs : interopérabilité 19
polymorphique - pointeur 145
polymorphisme 162
POS - READ/WRITE 105
PRIVATE - type dérivé 135
procédure 23
procédure - pointeur 126
procédure : pointeur 21
procédure : pointeur de 119–121, 124
PROCEDURE : attribut 120, 121, 124
PROCEDURE() - attribut pointeur 124
PROTECTED 95
PUBLIC - type dérivé 135

– R –

réallocation et affectation automatique 90, 91
réallocation via affectation 90
READ 102
reprofilage et association 118

– S –

SAME_TYPE_AS 149, 150
SAVE : argument ALLOCATABLE 87
scalaire : ALLOCATABLE 89
SELECT TYPE 146, 150
select type 146
standard IEEE 56, 57
STREAM - ACCESS 105
struct 45
structure : interopérabilité 46, 50
structure C 45
surcharge - type bound procedure 161

– T –

tableau dynamique 85
tableaux : interopérabilité 18
tableaux dynamiques 90
toward $+\infty$ 58
toward $-\infty$ 58
toward nearest 58
toward zero 58
traitement des interruptions 70
traitement exception 68
type bound procedure 160
type dérivé 45
type dérivé : composante ALLOCATABLE 87, 88
type dérivé : constructeur 130
type dérivé : destructeur 157
type dérivé : E./S. 106, 107
type dérivé : GENERIC 106
type dérivé : generic binding 153
type dérivé : operator binding 156
type dérivé : paramètre KIND 128, 129
type dérivé : paramètre LEN 128, 129

type dérivé : paramètres 128
type dérivé : PRIVATE 135
type dérivé : PUBLIC 135
type dérivé : visibilité 135
TYPE IS 150
typedef 45
typedef struct 48, 51

– U –

underflow 57, 58
USE - module, renommage opérateur 97

– V –

valeurs spéciales 56, 57
VALUE 46, 50
VALUE : attribut 22, 23
variable d'environnement 12
variable polymorphique 141
variable polymorphique : ALLOCATABLE 145
variable polymorphique : argument muet 144
variable polymorphique : POINTER 145
visibilité : type dérivé 135
void 23

– W –

WAIT 102
Working draft 2, 169
WRITE 102