Les apports de Fortran 200x

Patrick Corde et Hervé Delouis

18 octobre 2002

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

Voici les principaux documents dont nous nous sommes inspirés :

- *ISO/IEC/ 1539 "WORKING DRAFT" J3/02-007R3* - 20 sept. 2002 (`http://www.j3-fortran.org`) . Publié par le comité international J3 chargé du développement de la norme Fortran 200x.
- *Object Orientation and Fortran 2002 : Part II* - Malcolm Cohen : Fortran Forum - SIGPLAN - V.18 N.1 avril 1999.
- *The New Features of Fortran 2000* - John Reid - Fortran Forum V.21 N.2 août 2002.

Table des matières

1 – Meilleure intégration à l'environnement système

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

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

- `GET_COMMAND_ARGUMENT(number, value, length, status)`
  retourne le `number`e argument de la commande (numérotés à partir de zéro).

- `GET_ENVIRONNEMENT_VARIABLE(name, value, length, status, trim_name)`
  retourne 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 entiers correspondant aux numéros logiques d'unité associés à l'**astérisque** des commandes `READ`/`WRITE` et à la sortie standard des messages d'erreurs ;

- `IOSTAT_END` et `IOSTAT_EOR` sont des entiers correspondant aux valeurs négatives prises par le paramètre `IOSTAT` des commandes d'entrée/sortie en cas de fin de fichier ou d'erreur. La portabilité de ce paramètre est ainsi assurée pour ces deux cas d'erreurs. D'autres cas d'erreurs peuvent générer une valeur positive constructeur dépendante.

2 – Interopérabilité avec C

L'accès au module **ISO_C_BINDING** permet l'**interopérabilité** pour :

- les **entités de type intrinsèques** via des constantes symboliques définissant la valeur entière du paramètre `KIND` des types/sous-types autorisés (`C_SHORT`, `C_FLOAT`, `C_DOUBLE`, etc.) ou des caractères spéciaux (`C_NULL_CHAR`, `C_NEW_LINE`, `C_FORM_FEED`, etc.) ;

- les **pointeurs C** via le type `C_PTR` et les fonctions `C_ASSOCIATED(C_PTR_1[, C_CPTR_2])` et `C_LOC(X)` 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 : `type(C_PTR), VALUE :: buf`

  Note : attributs `ALLOCATABLE`/`POINTER` exclus pour une entité de type `C_PTR`

- les **structures de données C** via l'attribut `BIND(C)`. Par exemple :
  `TYPE,BIND(C) :: struct`

  Note : attributs `ALLOCATABLE`/`POINTER` exclus pour les composantes ;

- les **tableaux C**, ainsi : `integer,dimension(18,3:7,*)` est interopérable avec le tableau C : `int b[][5][18]` ;
- les **variables globales C** : concerne les blocs `COMMON` et les variables globales des modules Fortran. Par exemple :

```
common/com/ r, s
real(kind=C_FLOAT) :: r, s
BIND(C) :: /com/
```

  interopérable en C avec : `struct {float r; s;} com` ;

- les **fonctions C** : appelées depuis Fortran ou appelant une procédure Fortran ;

Note : l'interopérabilité Fortran–C devrait être disponible avec le compilateur Fortran Nec courant 2003...

2.1 – Fonction C appelée depuis Fortran

Voici un exemple de fonction C appelée depuis Fortran :

Tout d'abord, le prototype de la fonction C appelée :

```
int C_Func(void* buf, int count, My_type x);
```

Ci-après le programme Fortran appelant :

```
program p1   ! <=== Programme Fortran appelant la fonction C
  use FTN_C  !      ************************************
  . . .
  real(C_FLOAT), dimension(100), TARGET :: tab
  integer(kind=C_INT)                   :: N, y
  type(my_type)                         :: X
  . . .
  y = C_FUNC(buf=C_LOC(tab), count=N, x=X)
  . . .

module FTN_C
  use ISO_C_BINDING
  TYPEALIAS :: my_type => integer(kind=C_INT)
  interface
    function C_FUNC(buf, count, x), BIND(C, NAME="C_Func")
      implicit none
      integer(kind=C_INT)         :: C_FUNC
      type(kind=C_PTR),    VALUE :: buf
      integer(kind=C_INT), VALUE :: count
      type(my_type),       VALUE :: x
    end function C_FUNC
  end interface
end module FTN_C
```

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

Fonction C appelée

Arguments d'appel

1	2	3	-------	100

```
real(C_FLOAT),target :: tab(100)

18 integer(C_INT) :: N
32 type(my_type) :: X
```

Arguments muets

```
void* buf
int count
My_type x
```

Stack

`type(C_PTR),VALUE :: buf`	`C_LOC(tab)`
`integer(C_INT),VALUE::count`	18
`type(my_type),VALUE :: x`	32

Bloc Interface →

Appel : `y = C_FUNC(buf=C_LOC(tab), count=N, x=X)`

2.2 – Procédure Fortran appelée depuis C

Voici un exemple de sous-programme Fortran appelé depuis un programme C :

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

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

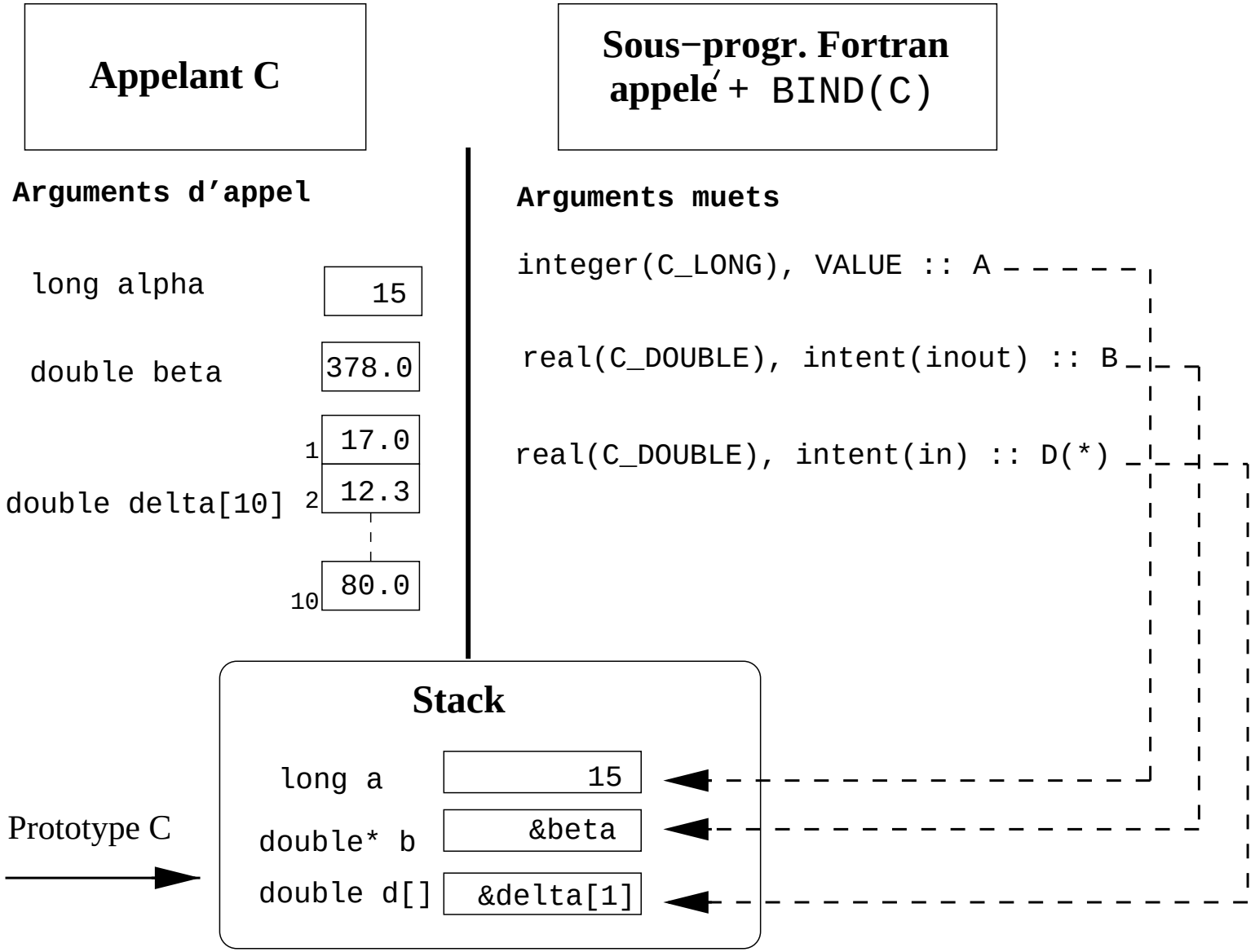
Voici le **prototype** de la fonction C correspondante :

```
void f1(long a, double *b, double d[]);
```

Voici une séquence d'appel de `f1` en C :

```
long alpha; double beta; double delta[10];
f1(alpha, &beta, 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` ;
- 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`.



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

3 – Arithmétique IEEE et traitement des exceptions

L'accès à certaines fonctionnalités de l'arithmétique IEEE ainsi qu'au contrôle des exceptions est lié à l'utilisation de 3 modules intrinsèques :

- `IEEE_EXCEPTIONS`,
- `IEEE_ARITHMETIC`,
- `IEEE_FEATURES`.

Voici une façon de les utiliser :

```
USE, INTRINSIC :: IEEE_ARITHMETIIC ;  USE, INTRINSIC :: IEEE_FEATURES
```

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

Voici quelques fonctions typiques :

- IEEE_SUPPORT_NAN([x]),
- IEEE_SUPPORT_INF([x]),
- IEEE_SUPPORT_STANDARD([x]),

- IEEE_IS_NAN(x),
- IEEE_IS_INF(x),
- IEEE_NEXT_AFTER(x,y),
- IEEE_IS_NEGATVE(x),

- IEEE_GET_FLAG(flag, flag_value),
- IEEE_SET_FLAG(flag, flag_value),
- IEEE_SET_HALTING_MODE(flag, halting),
- IEEE_GET_ROUDING_MODE(round_value),
- IEEE_SET_ROUDING_MODE(round_value).

4 – Nouveautés concernant les tableaux dynamiques

4.1 – Passage en paramètres de procédures

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

En Fortran 200x, un argument muet peut avoir l'attribut `ALLOCATABLE`, l'argument d'appel correspondant devra aussi avoir cet attribut ainsi que le même rang et le même type/sous-type. Le tableau dynamique passé en argument d'appel n'est pas nécessairement déjà alloué.

Géré par un descripteur interne analogue à celui d'un pointeur, il pourra (comme ce dernier) être une composante d'un type dérivé (cf. chapitre suivant).

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

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=8),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 ;
- les sous-objets 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

En Fortran 95, un bloc interface défini dans un module ne pouvait accéder aux autres entités (définition de type dérivé par ex.) stockées dans ce même module. L'instruction IMPORT permet l'importation d'une ou plusieurs de ces entités dans un bloc interface :

```
module truc
  type couleur
    character*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_teite
  end interface
contains
  . . . !---> Partie procédurale du module
end module truc
```

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

6 – Entrées-sorties

6.1 – Entrées-sorties asynchrones

Le nouveau paramètre `ASYNCHRONOUS='yes'` des instructions `OPEN` et `READ/WRITE` permet d'effectuer des entrées-sorties en mode asynchrone. Le programme continue sans attendre la fin de l'entrée-sortie : la nouvelle instruction `WAIT(unit=..., ...)` permettant de se resynchroniser.

À noter que les instructions `INQUIRE` et `CLOSE` permettent aussi la synchronisation.

Toute entité faisant l'objet d'entrées-sorties asynchrones récupère automatiquement un nouvel attribut `ASYNCHRONOUS` pour avertir le compilateur du risque encouru à les manipuler. Cet attribut peut aussi être explicitement spécifié :

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

6.2 – 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 : seule la position dans le fichier importe. Le fichier peut être formatté ou non.

La position courante est mesurée en octets 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 de se positionner au moment d'une entrée-sortie.

L'utilisation du paramètre `POS` est constructeur dépendante.

6.3 – Traitement des objets de type dérivé

Lors de la présence d'un objet de type dérivé au sein d'une liste de variables spécifiée pour une opération d'entrée-sortie, il est possible de faire appel à une procédure de type `SUBROUTINE`.

Concernant les entrées-sorties formattées un nouveau descripteur de format a été défini. Il s'applique à un objet de type dérivé de la liste et a pour forme :

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

La chaîne de caractères ainsi que le tableau d'entiers indiqués pour ce descripteur sont transmis en argument à la procédure appelée laquelle peut être attachée de façon générique au type dérivé de l'objet (*generic bindings*).

```
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
```

Une alternative est de définir un bloc interface :

```
INTERFACE READ(FORMATTED)
   MODULE PROCEDURE lecture_formatte1, lecture_formatte2
END INTERFACE
```

Ce type de procédures doit respecter le prototype suivant :

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

- `dtv` : objet de type dérivé qui est à l'origine de l'appel,
- `unit` : numéro de l'unité logique sur laquelle a été connecté le fichier, (`0` pour un fichier interne),
- `iotype` : chaîne de caractéres 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 formattés :

```
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
    ...
  WRITE( UNIT=1, FMT="(I2, DT 'COMPLET', 2F6.2 )", &
         IOSTAT=ios, IOMSG=message ) i, c1, x, y
  PRINT *, ios, message
  WRITE( UNIT=1, FMT="(F7.3, DT, DT 'COMPLET' )",  &
         IOSTAT=ios, IOMSG=message ) x, c1, c2
  PRINT *, ios, message
  WRITE( UNIT=1, FMT=*, IOSTAT=ios, IOMSG=message ) x, c, i, y
  PRINT *, ios, message
  WRITE( UNIT=1, NML=liste )
  PRINT *, ios, message
END PROGRAM exemple
```

```
MODULE couleur_mod
  TYPE couleur
    CHARACTER(len=16)  :: nom
    REAL, DIMENSION(3) :: compos
    CONTAINS
      GENERIC :: WRITE(FORMATTED) => ecr_format ! <=== Generic binding.
  END TYPE couleur
CONTAINS
    SUBROUTINE ecr_format( dtv, unit, iotype, v_list, iostat, iomsg )
      TYPE(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
      NAMELIST/liste/ dtv%nom, dtv%compos

      SELECT CASE( iotype )
        CASE ('DTNAMELIST'); WRITE( UNIT=unit, NML=liste, &
                                    IOSTAT=iostat, IOMSG=iomsg )
        CASE ('DTDIRECTED'); WRITE( UNIT=unit, FMT=* )             dtv%nom, dtv%compos
        CASE ('DTCOMPLET') ; WRITE( UNIT=unit, FMT='(a, 3f5.2)' ) dtv%nom, dtv%compos
        DEFAULT            ; WRITE( UNIT=unit, FMT='(3f5.2)' )    dtv%compos
      END SELECT
    END SUBROUTINE ecr_format
END MODULE couleur_mod
```

Autre exemple traitant d'un fichier binaire (non formatté) :

```
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
      GENERIC :: READ(UNFORMATTED) => lec_binaire ! <=== Generic binding.
  END TYPE couleur
CONTAINS
    SUBROUTINE lec_binaire( dtv, unit, iostat, iomsg )
      TYPE(couleur),          INTENT(in)    :: 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.

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(p1, p2, ...)
 real(kind=8), dimension(:,:), pointer, INTENT(IN)  :: p1
 real(kind=8), dimension(:,:), pointer, INTENT(OUT) :: p2
 . . . .
```

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

7.2 – Association et reprofilage

Lors de l'association, il est possible de préciser les bornes inférieures d'un tableau :

```
p(0:, 0:) => tab
```

Il est aussi possible de reprofiler les éléments d'un tableau de rang 1 :

```
p(1:n, 1:2*m) => vect(1:2*n*m)
```

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.
L'interface peut être implicite ou explicite.

Voici un exemple avec une **interface implicite** :

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

Voici un autre exemple avec l'attribut `PROCEDURE` permettant de déclarer un pointeur de procédure `p` avec le même mode d'interface implicite ou explicite que la fonction ou le sous-programme `proc` :

```
PROCEDURE(proc), POINTER :: p => NULL()
```

Voici un autre exemple de déclaration d'un pointeur de procédure `p` avec interface implicite, à l'état indéterminé, pouvant être associé à une fonction ou un sous programme :

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

Note : ces pointeurs peuvent apparaître en tant que composante d'un type dérivé ce qui permet d'*attacher* dynamiquement des méthodes à des objets (*dynamic binding*) comme on le verra plus loin.

Pour être en contexte d'**interface explicite**, on peut référencer un bloc interface virtuel (***abstract interface***) au moment de la déclaration du pointeur ou d'une procédure externe comme `proc` dans l'exemple ci-dessous.

```
ABSTRACT INTERFACE
  SUBROUTINE sub( x, y )
    REAL, intent(out) :: x
    REAL, intent(in)  :: y
  END SUBROUTINE sub
END INTERFACE
TYPE mytype
  PROCEDURE(sub), POINTER :: p
END TYPE
PROCEDURE(sub), POINTER :: p1=>NULL()
PROCEDURE(sub) :: proc
TYPE(mytype)             :: a
REAL                     :: var
p1 => proc
CALL p1( x=var, y=3.14 )
PRINT *, ASSOCIATED( p1, proc )
a%p => proc   ! <=== 'dynamic binding'
CALL a%p( x=var, y=2.718 )
```

8 – Nouveautés concernant les types dérivés

8.1 – Constructeurs de structures

Lors de la valorisation d'un type dérivé via un constructeur de structures, 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 ] )
. . .
```

À noter : le constructeur `couleur` peut être remplacé par une fonction générique personnelle de même nom.

8.2 – Paramètres d'un type dérivé

En Fortran 95, les types intrinsèques étaient déjà tous paramétrables ; ainsi, 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 (dit *nonkind*) n'est pas discriminant pour la généricité tandis que le deuxième (dit *kind*) l'est.

Les paramètres *nonkind* sont utilisés pour définir la longueur des chaînes ou les bornes des tableaux.

En Fortran 200x, les types dérivés sont paramétrables.

Par exemple :

```
type obj_mat(k, d)
  integer, KIND                  :: k
  integer, NONKIND               :: d
  integer                        :: nb_bits=k*8
  real(kind=k),dimension(d,d) :: tab
end type obj_mat
. . . .
type(obj_mat(k=8, d=1024))    :: mat
```

Les paramètres (ici **k** et **d**) sont obligatoirement de type entier avec un attribut `KIND`/`NONKIND`. `KIND` lui permet d'être discriminant au niveau de la généricité alors que `NONKIND` ne le permet pas.

S'ils ne sont pas explicitement déclarés ils sont du type entier par défaut et leur attribut `KIND`/`NONKIND` par défaut dépend du contexte de leur utilisation.

8.3 – 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 200x, la privatisation peut se gérer plus finement au niveau de chaque composante.

```
type, extensible :: base_type
  private                    ! Visibilité semi-privée par défaut
  integer          :: i      ! Composante privée
  integer, public  :: k      ! Composante publique
end type base_type
type, extends(public :: base_type) :: my_type
  private
  integer          :: l
  integer,public   :: m
end type my_type
. . . .
type(my_type) :: x
```

- `x%k` (raccourci de `x%base_type%k`) est une référence valide.
- `x%m` (raccourci de `x%my_type%m`) est une référence valide.
- `x%i` (raccourci de `x%base_type%i`) est une référence invalide.

8.4 – Composante allouable dynamiquement

L'attribut ALLOCATABLE est autorisé pour une composante

```
type obj_mat
  integer :: N, M
  real,dimension(:,:),ALLOCATABLE :: A
end type obj_mat
. . .
type(obj_mat) :: M1
. . .
read *,        M1%N, M1%M
allocate(M1%A(M1%N, M1%M))
. . .
```

8.5 – Sous-programmes de fin : destructeur

Dans une définition de type dérivé, la *directive* `FINAL` permet de spécifier une liste de sous-programmes (*final subroutines*) ayant obligatoirement un seul argument muet du type de celui défini.

Ces `destructeurs` sont automatiquement exécutés lors de la *finalisation* de toute entité du type considéré.

Une entité `POINTER` ou `ALLOCATABLE` est *finalisée* à sa désallocation tandis que les autres le sont au `RETURN`/`END`.

Voici un exemple :

```
module m
  type t(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=lbound(x,1), ubound(x,1)
      if(associated(x(i)%v)) deallocate(x(i)%v)
    end do
  end subroutine finalize_vect
end module m
```

Ainsi, l'objet `obj` déclaré via : `type(t(k=4)), dimension(:), allocatable :: obj` sera automatiquement *finalisé* via `finalize_vect` lors de sa désallocation.

8.6 – Procédures attachées à un type dérivé

Il y a trois façons d'*attacher* des procédures à un type dérivé :

- association par une **composante pointeur** de procédure (*dynamic binding*) ;
- attachement par **nom** (éventuellement générique)(*name binding*) ;
- attachement par **opérateur** (*operator binding*) ;

8.6.1 – Procédure associée par composante pointeur

Voici un exemple utilisant une composante pointeur de procédure :

```
type matrix(kind, n, m)
  integer,    kind                  :: kind
  integer, nonkind                  :: n, m
  real(kind=kind), dimension(n,m) :: A
  PROCEDURE(proc}, POINTER          :: solve !--> Interface identique à la
end type matrix                              !    procédure proc réelle ou
  . . .                                      !    virtuelle (abstract interface)
  . . .
  type(matrix(kind=kind(0.D0), n=5, m=10)) :: mat1
  . . .
  mat1%solve => my_proc  !--> Dynamic binding
  . . .
  call mat1%solve(....) !--> call my_proc(mat1, ....)
  . . .
```

8.6.2 – 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)
   type(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)
. . .
```

À noter :

- `proc` pourrait être le nom générique d'une familles de procédures : on coderait alors :
  `GENERIC :: proc => my_proc1, my_proc2, ...`
- les procédures ainsi attachées doivent être en mode d'interface explicite ;
- par défaut, l'objet `obj` est automatiquement passé comme premier argument `b` de la procédure ainsi appelée (notion de *passed-object dummy argument*). Ce premier argument muet a l'attribut `PASS` par défaut. Pour passer explicitement l'argument, il faut coder l'attribut NOPASS :
  `PROCEDURE, NOPASS :: proc => my_proc`
- on peut appliquer explicitement l'attribut `PASS` à un autre argument muet que le premier en spécifiant son nom :
  `PROCEDURE, PASS(b) :: proc => my_proc`

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

```
type matrix(kind, n, m)
  integer,    kind                  :: kind
  integer, nonkind                  :: n, m
  real(kind=kind), dimension(n,m) :: A
contains
  GENERIC :: OPERATOR(+)   => plus1
  GENERIC :: ASIGNEMENT(=) => egal1
end type matrix
  . . .
  type(matrix(kind=kind(0.D0), n=5, m=10)) :: mat1, mat2, mat3
  . . .
  mat1 = mat2 + mat3 !--> La fonction plus1 retourne le résultat de
  . . .              !    (mat2 + mat3) qui est affecté à mat1 via
  . . .              !    le sous-programme egal1.
```

Plusieurs procédures peuvent enrichir la généricité de ces opérateurs à condition que la liste des arguments muets soit discriminante (en faisant par exemple intervenir d'autres arguments qui ne sont pas de type `matrix`).

9 – Programmation orientée objet

- Extension des types dérivés
- Variables polymorphiques :
  - arguments muets polymorphiques,
  - pointeurs polymorphiques.
- Type effectif d'une variable polymorphique
- Structure de contrôle `SELECT TYPE`
- Procédures dites *type-bound* :
  - héritage d'une procédure *type-bound*,
  - surcharge d'une procédure *type-bound*,
  - procédure *type-bound* non surchargeable.

9.1 – Extension des types dérivés

Après avoir défini un type dérivé, on a la possibilité de l'étendre en y ajoutant des composantes :

```
type, extensible :: point2d
  real x
  real y
end type

type, extends(point2d) :: point3d
  real z
end type
```

Les variables de ces types sont déclarées de façon habituelle :

```
type(point2d) p2d
type(point3d) p3d
```

- les composantes de la variable `p2d` sont **x** et **y**,
- les composantes de la variable `p3d` sont **z** (l'extension), **x** et **y** ainsi que `point2d` (partie héritée).

9.2 – Variables polymorphiques

Une variable polymorphique est une variable dont le type est déterminé **à l'exécution** ; il est soit un type dérivé de base, soit l'une de ses extensions.

Une telle variable sera déclarée à l'aide du mot-clé `CLASS` avec le type de base :

```
CLASS(point2d) point
```

La variable `point` peut être :

- ☞ un **argument muet** : le type dynamique sera celui de l'argument d'appel,
- ☞ un **pointeur** : le type dynamique sera déterminé soit au moment de son association soit lors d'une allocation.

À noter : un objet (*unlimited polymorphic object*) déclaré via `CLASS(*)` pourrait prendre dynamiquement n'importe quel type extensible.

9.2.1 – Arguments muets polymorphiques

Ce type d'argument permet l'écriture de procédures qui s'appliquent à n'importe quel type étendu d'un type extensible :

```
function distance( p1, p2 )
  CLASS(point2d) p1, p2
  real          distance
  ! calcul de la distance entre les points p1 et p2
          . . .
end function distance
```

9.2.2 – Pointeurs polymorphiques

Le type dynamique d'un pointeur est celui de sa cible qui peut être définie lors d'une association ou d'une allocation dynamique via l'instruction `ALLOCATE` :

```
TYPE(point2d),  target  :: p2d
TYPE(point3d),  target  :: p3d
CLASS(point2d), pointer :: ptr2d_1, ptr2d_2
CLASS(point3d), pointer :: ptr3d
  . . . .
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.
```

9.3 – 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`.

9.4 – Structure de contrôle SELECT TYPE

Une nouvelle structure de contrôle (`SELECT TYPE`) permet de tester le type dynamique d'une variable polymorphique :

```
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 )
      TYPE IS(point2d)
        distance = sqrt( (p2%x-p1%x)**2 + (p2%y-p1%y)**2 )
      TYPE IS(point3d)
        distance = sqrt( (p2%x-p1%x)**2 + (p2%y-p1%y)**2 + (p2%z-p1%z)**2 )
      TYPE 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 *type-bound*

Les procédures *type-bound* sont déclarées au sein d'un type extensible. L'attribut `PASS` permet de leur passer implicitement en premier argument l'objet qui est à l'origine de l'appel.

```
MODULE point
  private
  TYPE, PUBLIC, EXTENSIBLE :: point2d
    REAL x, y
  CONTAINS
    PROCEDURE, PASS :: affichage => affichage_2d
  END TYPE
CONTAINS
  SUBROUTINE affichage_2d( obj, texte )
    CLASS(point2d),   intent(in) :: obj
    CHARACTER(len=*), intent(in) :: texte
    print *, texte; print *, "X = ", x, ", Y = ", y
  END SUBROUTINE affichage_2d
END MODULE point
PROGRAM prog
  USE point
  TYPE(point2d) p
  CALL p%affichage( "Voici mes coordonnées" )
END PROGRAM prog
```

9.5.1 – 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 pointcol
  USE point
  private
  TYPE, PUBLIC, EXTENDS(point2d) :: point2d_col
    REAL couleur
  END TYPE
END MODULE pointcol
PROGRAM prog
  USE pointcol
  TYPE(point2d_col) :: pcol
  call pcol%affichage( "Voici mes coordonnées" )
END PROGRAM prog
```

9.5.2 – 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
  private
  TYPE, PUBLIC, EXTENDS(point2d) :: point3d
    REAL z
  CONTAINS
    PROCEDURE, PASS :: affichage => affichage_3d
  END TYPE
CONTAINS
  SUBROUTINE affichage_3d( obj, texte )
    CLASS(point3d),   intent(in) :: obj
    CHARACTER(len=*), intent(in) :: texte
    print *, texte; print *, "X = ", x, ", Y = ", y, ", Z = ", 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 : dans un tel cas le compilateur sait quelle procédure appeler, il n'y a rien de dynamique !

Voici un exemple permettant de bénéficier complè tement du polymorphisme :

```
PROGRAM dynamic
  USE point
  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" )
  END SUBROUTINE affiche
END PROGRAM dynamic
```

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

9.5.3 – Procédure *type-bound* non surchargeable

On peut préciser l'attribut `NON_OVERRIDABLE` à la déclaration d'une procédure *type-bound* pour interdire toute surcharge.

```
TYPE, extensible :: mycomplex
  REAL theta, magnitude
CONTAINS
  PROCEDURE, PASS, NON_OVERRIDABLE :: real => real_part
  PROCEDURE, PASS, NON_OVERRIDABLE :: imag => imag_part
END TYPE
...
FUNCTION real_part( a )
  CLASS(mycomplex), intent(in) :: a
  REAL  real_part
  real_part = a%magnitude*cos(a%theta)
END FUNCTION
```

10 – En conclusion

- La phase d'élaboration technique de la norme Fortran 200x se termine en 2002. Le document *WORKING DRAFT* est consultable en ligne à l'adresse :
  `http://www.j3-fortran.org`
  à la rubrique Fortran 200x.

- La phase de validation et d'approbation par votes du comité J3-Fortran du NCITS (*National Committee for Information Technology Standards*) va prendre deux années supplémentaires...

- Les premiers compilateurs Fortran 200x ne seront pas disponibles avant 2004 !

- En attendant, certains fournisseurs comme IBM, Nec , Nag, N.A. Software, ... intégrent progressivement certains aspects de cette future norme.