Introduction à FORTRAN 90

Alain Soyer

Séminaire du 2 février 1995

Introduction à FORTRAN 90

par Alain Soyer
Laboratoire de Minéralogie-Cristallographie
Université Paris 6 , CNRS UMR 7590

Comme son titre l'indique, ce document n'est qu'une introduction au langage de programmation FORTRAN 90.
Ce n'est pas un cours, car il ne fait pas une description rigoureuse et complète du langage.
Son but est de présenter les principales nouveautés de la norme FORTRAN 90 par rapport à la norme précédente : FORTRAN 77.
Il s'adresse donc à des personnes ayant une connaissance minimum de FORTRAN 77.

Plan :

Mais, avant d'entrer dans le vif du sujet, rassurons ceux qui pourraient être inquiets du devenir de leurs programmes actuels.

Comme le montre le tableau ci-dessous, le FORTRAN 90 est formé à environ 62 % du FORTRAN 77 standard.

Domaine	%
FORTRAN 77	62
Traitement des tableaux	8.7
Pointeurs	6.1
Traitement numérique	3.1
Types et opérations définis par l'utilisateur	3.1
Modules	3.1
Procédures	3.1
Structures de données	2.9
Structures de contrôle	2.8
Entrées / Sorties	2.6
Forme du code source	1.4
Constructions obsolètes	1.1

Les pourcentages sont proportionnels aux nombres de nouveautés introduites dans les différents domaines indiqués. On constate par exemple l'importance du traitement des tableaux, ainsi que des pointeurs.

1 - Les éléments d'un programme source :

On va partir des plus petits, les caractères, pour remonter vers les plus grands, jusqu'aux unités de programme.

- Le jeux de caractères :

Caractères alpha-numériques :

Lettres ABCDEFGHIJKLMNOPQRSTUVWXYZ
Chiffres 0123456789
Blanc-souligné _

Caractères spéciaux :

Graphisme	Nom	Graphisme	Nom
	Blanc	:	Deux points
=	Égal	!	Point d'exclamation
+	Plus	"	Guillemet double
-	Moins	%	Pour-cent
*	Étoile	&	Et commercial
/	Slash	;	Point virgule
(	Parenthèse gauche	<	Inférieur
)	Parenthèse droite	>	Supérieur
,	Virgule	?	Point d'interrogation
.	Point	$	Dollar
'	Apostrophe		

Si les minuscules sont permises elles sont équivalentes aux majuscules, sauf dans les chaines de caractères :

MY_VARIABLE <=> My_Variable <=> my_variable
mais 'Coucou' est différent de 'coucou'

En option une machine peut supporter d'autres jeux de caractères (Grecs, Kanji ...) mais ils ne peuvent être utilisés que dans les constantes caractères et les commentaires.

- Les mots du langage (lexical tokens) :

formés à partir du jeu de caractères FORTRAN, ce sont:

les étiquettes : 1 à 5 caractère(s) numérique(s)
99999 ...

les mots clés :
IF THEN GOTO …

les constantes :
1 1.85 .TRUE. 'salut' ...

les opérateurs :
+ - * …

les séparateurs :
; (/ /) ...

les noms : 31 caractères alpha-numériques maximum
doivent commencer par une lettre
NOMBRE_D_ATOMES CAS128

- Les instructions :

formée de “lexical tokens” une instruction peut s’étendre sur 40 lignes maximum. Le caractère & placé en fin de ligne est utilisé pour indiquer que l’instruction continue sur la ligne suivante.

Chaque ligne ne doit pas dépasser 132 caractères.

Les blancs peuvent être utilisés librement:
Y=A*X+B est identique à Y = A * X + B

Le caractère ! placé n’importe où dans une ligne indique un début de commentaire : tous les caractères de la ligne situés après le ! sont ignorés.

- Unité de programme :

Une unité de programme est une suite d'instructions, terminée par l'instruction :

```
END
```

Le caractère ; est un séparateur d'instructions : plusieurs instructions courtes peuvent être placées sur la même ligne séparées par un ; (mais je déconseille cette pratique).

Des instructions situées dans un autre fichier peuvent être inclues dans le fichier source courant par :

```
INCLUDE  'NomDeFichier'
```

Un fichier source peut être écrit dans une des *formes* suivantes.

- Forme fixe comme en FORTRAN 77 (héritée des cartes perforées)
  étiquettes en colonnes 1 à 5
  instructions en colonnes 7 à 72
  un caractère en colonne 6 indique une « carte suite »
  un C ou une * en colonne 1 indique une ligne commentaire

- Forme libre (nouvelle forme plus adaptée au travail sur terminal)

```
Y = A * X    &    ! Commentaire
+ B
```

mais ceci est incorrect :

```
Y = A * X   ! Commentaire   &
+ B

NOM = 'Alain &
    Soyer'       <=> NOM = 'Alain     Soyer'
NOM = 'Alain &
   &Soyer'       <=> NOM = 'Alain Soyer'
```

- Forme mixte
  à l'avantage d'être équivalente aux deux formes précédentes
  si une instruction s'étend sur deux lignes on place un & colonne 73
  et un autre & ligne suivante en colonne 6

2 - Les déclarations :

Les déclarations sont utilisées pour spécifier les attributs (type, kind, dimension ...) des entités d'un programme (variables, constantes, fonctions ...), ainsi que leurs relations (COMMON, EQUIVALENCE ...).

On peut se forcer à déclarer toutes les variables en plaçant au début de chaque unité de programme l'instruction :

```
IMPLICIT NONE
```

Ceci ne relève pas d'une pratique informaticienne un peu masochiste, mais est au contraire une bonne habitude à prendre : elle peut éviter des erreurs, surtout maintenant que le FORTRAN permet, comme nous le verrons plus loin, de créer des sous-programmes internes et d'importer des variables en utilisant des modules.

Nouvelle forme de déclaration en FORTRAN 90 :
l'idée est de regrouper en un seul endroit tout ce qui concerne une variable.

Type, attribut_1, ... attribut_n :: Nom = Valeur_Initiale

exemples :

```
INTEGER, DIMENSION (3) :: IARRAY = (/ 1, 2, 3 /)
CHARACTER (LEN = 12), SAVE :: PROMPT = 'Password ?'
```

- Types :

Le type est l'attribut le plus important d'une entité.

<table>
<tr><td rowspan="6">FORTRAN 90
data types</td><td rowspan="4">Intrinsic types</td><td rowspan="3">Numeric types</td><td>Integer</td></tr>
<tr><td>Real</td></tr>
<tr><td>Complex</td></tr>
<tr><td rowspan="2">Non numeric
types</td><td>Logical</td></tr>
<tr><td>Character</td></tr>
<tr><td rowspan="2">Derived
types</td><td>. . . .</td><td>. . . .</td></tr>
<tr><td>. . . .</td><td>. . . .</td></tr>
</table>

-Type dérivé:

Une grande nouveauté en FORTRAN 90 est la possibilité pour l'utilisateur de se définir ses propres types, appelés types dérivés, à partir des types intrinsèques.

Exemple :

```
TYPE MENDELEEV
     CHARACTER *12 NAME
     CHARACTER *2  SYMBOL
     INTEGER       NUMBER
     REAL          WEIGHT
END TYPE MENDELEEV
```

ainsi définit, le nouveau type dérivé peut être utilisé dans des déclarations:

```
INTEGER, PARAMETER :: ELEMENT_NBR = 106
TYPE (MENDELEEV) PERIODIC_TAB (ELEMENT_NBR)
```

puis les éléments de PERIODIC_TAB, appelés structures, sont utilisés:

```
PERIODIC_TAB(2) = MENDELEEV ('Helium','He',2,4.0026)
```

le séparateur % sert à manipuler les membres d'une structure individuellement:

```
PERIODIC_TAB(1)%NAME = 'Hydrogen'
WRITE(*,*) 'Masse du Lithium=', PERIODIC_TAB(3)%WEIGHT
```

autre exemple :

```
TYPE MACHIN
     CHARACTER *16    TRUC (10)
     TYPE (MENDELEEV) BIDON
END TYPE MACHIN

TYPE (MACHIN) BIDULE (5)
```

Ceci :	Désigne cela :
BIDULE	l'ensemble du tableau BIDULE
BIDULE(I)	tous les membres du Ième élément de BIDULE
BIDULE(I)%TRUC	l'ensemble du tableau TRUC du Ième

	élément de BIDULE
BIDULE%TRUC(K)	les éléments K des tableaux TRUC dans tous les éléments de BIDULE
BIDULE(I)%TRUC(K)	le Kième élément du tableau TRUC dans le Ième élément de BIDULE
BIDULE(I)%BIDON%NAME(J:K)	les caractères J à K du membre NAME de BIDON dans le Ième élément de BIDULE

A priori l'ordre de déclaration des composants (ou membres) d'un type dérivé n'implique rien concernant l'ordre de stockage en mémoire, sauf si on ajoute la déclaration 'SEQUENCE' en tête.

- "Kind" :

Sur une machine donnée, chaque type intrinsèque peut avoir plus d'une représentation qui est repérée par un entier appelé "kind type parameter".

Par exemple la norme impose qu'il existe au moins deux sortes de réels correspondants à la simple et à la double précision.

La fonction intrinsèque KIND retourne le "kind type parameter" de son argument:
Par exemple si I est déclaré INTEGER, KIND (I) retourne le "kind" des entiers par défaut.
KIND (0.0) retourne le "kind" des réels par défaut.

Le programmeur peut dans une déclaration de variable indiquer la précision et l'étendue dont il a besoin:

```
INTEGER, PARAMETER :: SMALL = SELECTED_INT_KIND (3)
INTEGER (KIND = SMALL) :: ISMALL
```

on a demandé 3 chiffres significatifs, donc on devra avoir :
-999 <= ISMALL <= 999

de même on pourra définir la constante 123 dans le type entier sélectionné :
`123_SMALL`

```
INTEGER, PARAMETER :: BIG = SELECTED_REAL_KIND (15,307)
REAL (KIND = BIG) :: PI = 3.14159265358979_BIG
```

on a demandé 15 chiffres significatifs et une étendue pouvant aller jusqu'à 10^{307}

Pour une constante caractère, le kind doit être placé devant :

```
INTEGER, PARAMETER :: GREEK = 5
WRITE (*,*)  GREEK_'αβχδ'
```

- Il existe beaucoup d'autres attributs qui peuvent apparaitre dans les déclarations; certains seront décrits dans la suite (TARGET, POINTER, INTENT ...) .

- "Namelist" :

Un ensemble de variables peut être regroupé dans une "liste nommée" en vue de faciliter les lectures / écritures:

```
INTEGER, DIMENSION (2) :: MY_INT = (/ 2, 1 /)
REAL                   :: MY_REAL = 123.456
CHARACTER (LEN=16)     :: MY_STRING = "L'aventure"
NAMELIST / MY_LIST / MY_INT, MY_REAL, MY_STRING
```

Les variables de la liste nommée ne sont pas forcément rangées séquentiellement en mémoire.

La liste est lue par :

```
OPEN (UNIT=IIN, ....
READ (UNIT=IIN, NML=MY_LIST)
```

les données se présentant sous la forme suivante:

```
&MY_LIST MY_INT(1) = 0  MY_STRING = 'Coucou'
  MY_REAL = 789.0 /
```

On remarquera que l'ordre est sans importance, puisque les noms des variables servent de mots clés, et que MY_INT(2) garde sa valeur initiale car il n'apparait pas dans les données.

l'ordre d'écriture :

```
OPEN  (UNIT=IOUT, ....
WRITE (UNIT=IOUT, NML=MY_LIST)
```

donnera alors le résultat :

```
&MY_LIST  MY_INT = 0  1 , MY_REAL = 789.00000,
 MY_STRING = Coucou /
```

Les "Namelist" ont deux avantages:
- une grande souplesse d'utilisation,
- des fichiers de données plus clairs car les noms des variables précèdent leur valeur, et leur ordre est sans importance.

3 - Contrôle de l'ordre d'exécution des instructions :

Un programme se limite rarement à une suite linéaire d'instructions ; on a souvent besoin d'effectuer certaines actions suivant le résultat d'un test, ou de répéter un certain nombre de fois une série d'instructions.
Ceci peut se faire à l'aide de blocs IF, de blocs SELECT CASE et de boucles DO.

- Nouvelle syntaxe possible pour les tests :

F77 ou F90	F90
.EQ.	==
.NE.	/=
.GE.	>=
.GT.	>
.LT.	<
.LE.	<=

- Possibilité de donner un nom aux blocs :

```
INTEGER           :: ATOM_TYPE
CHARACTER         :: ATOM_LABEL
CHARACTER (LEN=6) :: COLOR
. . .
FIND_TYPE:  IF (ATOM_TYPE >= 1) THEN
     FIND_COLOR:  IF (ATOM_LABEL == 'C') THEN
          COLOR = 'BLUE'
     ELSE IF (ATOM_LABEL /= 'O') THEN FIND_COLOR
          COLOR = 'RED'
     ELSE FIND_COLOR
          COLOR = 'GREEN'
     END IF FIND_COLOR
ELSE  FIND_TYPE
     COLOR = 'GRAY'
END IF FIND_TYPE
```

- "Select Case" :

```
IFORCE = AMOR ( 'Roméo', 'Juliette')
SELECT CASE (IFORCE)
     CASE ( :1)                    ! Anything <= 1
          AIME = 'Un peu'
     CASE ( 100: )                 ! Anything >= 100
          AIME = 'A la folie'
     CASE DEFAULT
          AIME = 'Beaucoup'
END SELECT
```

- Nouvelles formes de la boucle DO :

```
IO_STATUS = 0
DO WHILE (IO_STATUS == 0)
     READ (UNIT=IIN, FMT=*, IOSTAT=IO_STATUS) XX
END DO

YEAR = 0
AGE: DO
     YEAR = YEAR + 1
     IF ( YEAR > 130) EXIT AGE          ! Exit loop
     IF ( MOD (YEAR, 10) /= 0) THEN
          CYCLE AGE                     ! Iterate
     ELSE
          DECADE = DECADE + 1
     END IF
     ....
END DO AGE
```

4 - Manipulation des tableaux :

En FORTRAN 90 un tableau peut être considéré comme un tout et manipulé dans son entier ou par morceaux (sections).

- Initialisation :

```
REAL, DIMENSION (5)      :: ARRAY = 0.0
INTEGER, DIMENSION (3)   :: IT = (/ 1, 3, 5 /)
INTEGER, DIMENSION (80) :: EVEN = (/ (2*I, I=1,80) /)
LOGICAL, DIMENSION (20) :: LA
INTEGER, DIMENSION (3,3):: MAT
....
LA = .TRUE.
MAT = RESHAPE ( (/ 1,2,3,4,5,6,7,8,9 /), (/ 3,3 /) )
```

- Sections de tableau (array sections) :

Syntaxe similaire à celle utilisée pour les sous-chaines de caractères

```
INTEGER, PARAMETER            :: N = 5
INTEGER, DIMENSION (N,N)      :: ARRAY
INTEGER, DIMENSION (N-2,N-2) :: INTERIOR
INTEGER, DIMENSION (N)        :: COLUMN1, ROW2, DIAGO
INTEGER, DIMENSION (N*N)      :: LINEAR
....
INTERIOR = ARRAY (2:N-1 , 2:N-1)
COLUMN1  = ARRAY (: , 1)
ROW2     = ARRAY (2 , :)
LINEAR   = PACK ( ARRAY, .TRUE. )
DIAGO    = LINEAR (1 : N*N : N+1)
```

- Fonctions d'interrogation :

```
INTEGER ARRAY (-3:4 , -5:6)
INTEGER ARRAY_SHAPE (2)
INTEGER LOWER (2), UPPER (2)
INTEGER ARRAY_SIZE
...
ARRAY_SHAPE = SHAPE (ARRAY)  retournera {8, 12}
LOWER = LBOUND (ARRAY)       retournera {-3, -5}
UPPER = UBOUND (ARRAY)       retournera {4, 6}
ARRAY_SIZE = SIZE (ARRAY)    retournera 96
```

- Indices vectoriels (vector subscript) :

```
REAL, DIMENSION (6,8)  :: A
INTEGER, DIMENSION (4) :: IT = (/ 2, 5, 2, 4 /)
....
A (IT, 3)  est un tableau de rang 1 constitué de:
A(2,3), A(5,3), A(2,3) et A(4,3)
```

- "WHERE construct" :

```
REAL ARRAY (50)
CHARACTER*5 INDIC (50)
.....
ARRAY (I) = ....
.....
WHERE (ARRAY < 0)  ARRAY = - ARRAY
....
WHERE ( ARRAY > 100)
     INDIC = 'Big'
ELSE WHERE
     INDIC = 'Small'
END WHERE
```

- Autres fonctions :

La plupart des fonctions intrinsèques acceptent des tableaux en argument :

```
REAL, DIMENSION (100) :: ARRAY
...
ARRAY = COS (ARRAY)
```

Il existe de nombreuses nouvelles fonctions intrinsèques pour manipuler des tableaux, comme par exemple :

```
CSHIFT      permutation circulaire des éléments
MINVAL      plus petit élément d'un tableau
MINLOC      position du plus petit élément
SUM         somme des éléments d'un tableau
PRODUCT     produit des éléments d'un tableau
DOT_PRODUCT produit scalaire de deux vecteurs
MERGE       combine deux tableaux (suivant un masque)
TRANSPOSE   transpose une matrice
MATMUL      produit de deux matrices
....
```

- Passage de tableaux et sections en argument :

Nouvelle façon de passer des tableaux en argument d'un sous-programme :

```
SUBROUTINE ASHAPE (TABLE)
REAL, DIMENSION ( : , : )  :: TABLE
. . . .
```

ici TABLE est un "assumed shape array" de rang 2.

Le programme appelant peut se présenter de la manière suivante :

```
PROGRAM PASSE_TABLE
INTERFACE
     SUBROUTINE ASHAPE (ARRAY)
     REAL, DIMENSION ( : , : )  :: ARRAY
END INTERFACE
REAL, DIMENSION (2,3)  :: TAB
....
CALL ASHAPE (TAB)
....
```

Passage de sections en argument d'un sous-programme :

```
PROGRAM ESS1
   INTERFACE
      SUBROUTINE SUB1 (TAB)
         REAL TAB (:)
      END SUBROUTINE SUB1
   END INTERFACE
   INTEGER, PARAMETER :: N = 10
   REAL               :: MAT (N,N)
   INTEGER            :: IL, I
   DO IL = 1, N
      MAT (IL,:) = (/ (N*(IL-1)+I, I=1,N) /)
   END DO
   DO IL = 1, N
      CALL SUB1 (MAT(IL,1:IL:2))
   END DO
END PROGRAM ESS1
SUBROUTINE SUB1 (TAB)
   REAL TAB (:)
   INTEGER M, I
   M = SIZE (TAB)
   WRITE (*,*) (TAB(I), I=1,M)
END SUBROUTINE SUB1
```

5 - “Program units” :

- Généralités :

Un programme doit contenir un (et un seul) programme principal. Il peut aussi comporter d’autres “unités de programmes” qui sont:
- les procédures, c’est-à-dire:
  - les “subroutines” (externes ou internes),
  - les fonctions (externes ou internes);
- les modules;
- les “block-data”.

Organisation d’un programme principal :

```
PROGRAM GENIAL
    partie spécification (déclarations)
    partie exécutable (instructions)
    CONTAINS
    procédures internes
END PROGRAM GENIAL
```

Procédure externe:

```
SUBROUTINE SUPER (...)
    partie spécification (déclarations)
    partie exécutable (instructions)
    CONTAINS
    REAL FUNCTION DIVINE (....)
        ....
        DIVINE = ...
        ...
    END FUNCTION DIVINE
END SUBROUTINE SUPER
```

Les procédures internes (comme la fonction DIVINE) ne sont visibles et donc appelables que dans l’unité de programme qui les contient (ici SUPER) appelée “host”.

- Nouveautés concernant les procédures :

Possibilité de préciser le type d'utilisation des arguments :

```
SUBROUTINE SUPER (ARG1, ARG2, ARG3)
INTEGER, INTENT (IN)                :: ARG1
REAL, DIMENSION (:), INTENT (OUT) :: ARG2
REAL, INTENT (INOUT)                :: ARG3
```

Arguments optionnels et passage d'arguments par mots clés :

```
SUBROUTINE SECUR (CLE, PRUDENT, DEBUG)
INTEGER, DIMENSION (: , :)      :: CLE
CHARACTER (*), OPTIONAL         :: PRUDENT
LOGICAL, OPTIONAL, INTENT (IN) :: DEBUG
...
IF (PRESENT (DEBUG) ) THEN
    IF (DEBUG) THEN
        WRITE (*,*) CLE
    ELSE
        IF (PRESENT (PRUDENT)) WRITE (*,*) PRUDENT
    END IF
END IF
....
```

Le programme appelant doit contenir une interface (ici via l'INCLUDE).
L'appel à SECUR peut se faire de différentes manières :

```
PROGRAM ESPION
INCLUDE 'interface.f'
CHARACTER * 80              :: LINE
LOGICAL, PARAMETER          :: DBX = .TRUE.
INTEGER, DIMENSION (10,20) :: ITAB

CALL SECUR (ITAB)
CALL SECUR (ITAB, PRUDENT=LINE)
CALL SECUR (ITAB, DEBUG=DBX)
CALL SECUR (ITAB, DEBUG=DBX, PRUDENT=LINE)
```

Récursivité (directe ou indirecte):

La valeur de la fonction qui était désignée par son nom en FORTRAN 77, doit maintenant être précisée par un autre nom dans RESULT.

```
RECURSIVE FUNCTION FACTORIEL (N) RESULT (FCT)
  INTEGER, INTENT (IN) :: N
  INTEGER              :: FCT

  IF (N <= 1) THEN
    FCT = 1
  ELSE
    FCT = N * FACTORIEL (N - 1)
  ENDIF
END FUNCTION FACTORIEL
```

- Interface :

Une interface explicite peut servir à :

- préciser les arguments (et leurs attributs) de procédures externes, pour permettre au compilateur de vérifier que les appels à ces procédures sont corrects, ou parce qu'on utilise des arguments optionnels :

dans l'exemple précédent le fichier inclus "interface.f" doit contenir :

```
INTERFACE
   SUBROUTINE SECUR (CLE, PRUDENT, DEBUG)
      INTEGER, DIMENSION (: , :)      :: CLE
      CHARACTER (*), OPTIONAL         :: PRUDENT
      LOGICAL, OPTIONAL, INTENT (IN) :: DEBUG
   END SUBROUTINE SECUR
END INTERFACE
```

- créer un sous-programme générique, c'est-à-dire appelable avec des arguments de type différent :

```
INTERFACE INCREMENT
   SUBROUTINE I_INC ( ARG )
      INTEGER, INTENT (INOUT) :: ARG
   END SUBROUTINE I_INC
   SUBROUTINE R_INC ( ARG )
      REAL, INTENT (INOUT) :: ARG
   END SUBROUTINE R_INC
END INTERFACE
....
INTEGER II
REAL RR
....
CALL INCREMENT (II)
....
CALL INCREMENT (RR)
....
```

- créer ses propres opérateurs :

```
....
INTERFACE OPERATOR ( .FUSION. )
   FUNCTION FUSION_NUCLEAIRE ( P1, P2)
      TYPE (MENDELEEV), INTENT (IN)   :: P1
      TYPE (MENDELEEV), INTENT (IN)   :: P2
      TYPE (MENDELEEV) :: FUSION_NUCLEAIRE
   END FUNCTION FUSION_NUCLEAIRE
END INTERFACE
....
TYPE (MENDELEEV) PERIODIC (106)
TYPE (MENDELEEV) NEW
....
NEW = PERIODIC(78) .FUSION. PERIODIC(54)
....
```

- définir un nouveau sens à l'affectation (=) :

```
....
INTERFACE ASSIGNMENT ( = )
   SUBROUTINE EXTRA (RESULTAT, ENTREE)
      TYPE (MENDELEEV), INTENT (OUT) :: RESULTAT
      TYPE (MACHIN), INTENT (IN)     :: ENTREE
   END SUBROUTINE EXTRA
END INTERFACE
....
TYPE (MENDELEEV) :: RES
TYPE (MACHIN)    :: DONNEE
....
DONNEE = ....
RES = DONNEE
....
```

- Modules :

Le “module” est un nouveau concept de FORTRAN 90 qui est amené à jouer un grand rôle. Un module est une unité de programme qui peut contenir :

- des déclarations de types dérivés qui seront accessibles dans les procédures qui “utilisent” le module : de ce point de vue le module peut remplacer un INCLUDE ;

- des variables qui seront globales : le module peut donc aussi remplacer des COMMON ;

- des constantes et initialisations de données : le module joue alors le rôle de BLOCK-DATA ;

- des interfaces pour des sous-programmes et des nouveaux opérateurs ;

- des procédures : le module peut constituer une bibliothèque.

Les modules permettent de regrouper des données et des procédures manipulant ces données. Certaines de ces données et/ou procédures peuvent être déclarées “privées” et rendues invisibles de l’extérieur. Les modules facilitent donc l’organisation, la sécurité, le découpage et l’indépendance des différentes parties d’un programme.

```
MODULE POLAR_COORDINATES
   TYPE POLAR
      PRIVATE
      REAL RHO, THETA
   END TYPE POLAR
   INTERFACE OPERATOR (*)
      MODULE PROCEDURE POLAR_MULT
   END INTERFACE
   CONTAINS
   TYPE (POLAR) FUNCTION POLAR_MULT (P1, P2)
      TYPE (POLAR), INTENT(IN) :: P1, P2
      POLAR_MULT = POLAR (P1%RHO * P2%RHO, &
      P1%THETA + P2%THETA)
   END FUNCTION POLAR_MULT
END MODULE POLAR_COORDINATES
```

Utilisation du module :

```
SUBROUTINE COORD ( .... )
USE POLAR_COORDINATES
TYPE (POLAR) :: POL1, POL2, POL3
....
POL3 = POL1 * POL2
....
```

- Portée (Scope) et déclarations implicites :

La portée ou le "scope" d'une entité (variable, constante, procédure, opérateur) est la partie du programme où elle est connue, et donc peut être utilisée, définie ou référencée.

La notion de "portée" existait déjà en FORTRAN 77 mais ne posait pas de problème ; en FORTRAN 90, avec l'apparition des procédures internes et des modules, le programmeur doit faire plus attention :

```
PROGRAM HOST
   USE EXT_DATA  ! Accesses integer X, real Y
   REAL A, B
   . . .
   READ *, P     ! declares P implicitly
   . . .
   CALL CALC (Z) ! declares Z implicitly
   . . .
 CONTAINS
   SUBROUTINE CALC (X)
      REAL B
      . . .
      X = A + B + P + Q + Y
      ! X (dummy argument) is local
      ! A and P are host associated
      ! B explicitly declared is local
      ! Q implicitly declared is local
      ! Y is used associated from the module
      . . .
   END SUBROUTINE CALC
   . . .
END PROGRAM HOST
```

6 - Entrées / Sorties :

- NAMELIST : voir 2

- Nouvelles options de l'OPEN :

(Ces options ont leur correspondant dans l'INQUIRE)
La valeur par défaut est indiquée en premier:

```
ACTION = 'READWRITE'
         'WRITE'
         'READ'
                                        en écriture :
DELIM = 'NONE'                             chaine
        'QUOTE'                            «chaine»
        'APOSTROPHE'                       'chaine'

POSITION = 'ASIS'
           'APPEND'
           'REWIND'

PAD =  'YES'      (utilisé en lecture)
       'NO'

RECL =    peut maintenant être utilisé pour les fichiers formatés

STATUS =       nouvelle possibilité : 'REPLACE'
```

Remarque: il est pratique d'utiliser l'INQUIRE pour calculer la taille d'enregistrement d'un fichier à accès direct :

```
CHARACTER *25    C
DOUBLE PRECISION D
REAL             R
INTEGER          I, IOL
TYPE (MENDELEEV) DERIV
....
INQUIRE (IOLENGTH = IOL) C, D, I, R, DERIV
```

- Nouveaux FORMAT :

B	Binaire	B'10110101'	
O	Octal	O'265'	
Z	Hexadécimal	Z'B5'	
EN	Engineering	75.09E+03	(exposant = multiple de 3)
ES	Scientifique	7.509E+04	(chiffre non nul avant le .)
G	Général		

- Lecture/Ecriture sans changer d'enregistrement (NonAdvancing) :

En sortie :

en interactif permet de poser une question avec le curseur positionné en fin de ligne :

```
WRITE (UNIT=*, FMT='(A)', ADVANCE='NO') ' Filename ? '
```

dans un fichier:

```
WRITE (UNIT=IOUT,FMT=15,ADVANCE='NO') ' Comment va tu'
WRITE (UNIT=IOUT,FMT=15) 'yau de poele ?'
15 FORMAT (A)
```

écrira un seul enregistrement: Comment va tuyau de poele ?

En entrée :

permet de lire par champs dans un enregistrement.

```
INTEGER IIN, BYTE, STATUS, I
INTEGER RECORD        ! compteur d'enregistrements
INTEGER FIELD         ! compteur de champs
REAL A (...)
. . . .
OPEN (UNIT=IIN, ...
I = 1
RECORD = 1
DO
  FIELD = 1
  DO
    READ (UNIT=IIN, FMT=10, ADVANCE='NO', &
    SIZE=BYTE, IOSTAT=STATUS, EOR=20, END=80) A(I)
    IF (STATUS /= 0) THEN
      WRITE (*,*) 'Error read record number: ', &
      RECORD
      WRITE (*,*) 'Error in field number: ', FIELD
      WRITE (*,*) 'Error character number: ', BYTE
      ....
    END IF
    I = I + 1
    FIELD = FIELD + 1
    ....
  END DO
  ...
  20 CONTINUE
  RECORD = RECORD + 1
END DO
80 CONTINUE
CLOSE (UNIT=IIN, ...
...
```

7 - Allocation dynamique, Pointeurs et Cibles (Targets) :

- Allocation dynamique de mémoire :

```
SUBROUTINE  SWAP_ARRAYS (A, B)
REAL, DIMENSION ( : )               :: A, B
REAL, DIMENSION ( : ), ALLOCATABLE :: TEMP
INTEGER  :: ERR_CODE, NA, NB
....
NA = SIZE (A)
NB = SIZE (B)
IF (NA /= NB) THEN
     ....
     RETURN
END IF
ALLOCATE (TEMP (NA), STAT=ERR_CODE)
IF ( ERR_CODE /= 0) THEN
     ....
     RETURN
END IF
TEMP = A
A = B
B = TEMP
DEALLOCATE (TEMP)
END SUBROUTINE SWAP_ARRAYS
```

- Pointeurs et Cibles (targets) :

Un **pointeur** peut être considéré comme un **descripteur** qui contient des informations sur le type, le rang, les dimensions et l'adresse d'une **cible**. (Donc un pointeur vers un scalaire de type réel n'est pas identique à un pointeur vers un tableau de type dérivé).

Pour pouvoir être la cible d'un pointeur, une variable ou un tableau doit avoir l'attribut TARGET dans sa déclaration.

```
REAL, DIMENSION (100), TARGET  :: TAB1, TAB2
REAL, DIMENSION ( : ), POINTER :: PT1, PT2
INTEGER ERR_CODE

! Associe PT1 à TAB1
! PT1 devient un synonyme de TAB1
PT1 => TAB1
WRITE (*,*) ASSOCIATED (PT1)  ! Write .TRUE.

PT2 => TAB2

! Attention le comportement des pointeurs est
! différent du langage C : ceci recopie le contenu
! de la cible de PT1 dans la cible de PT2
PT2 = PT1                     ! Same as TAB2 = TAB1

NULLIFY (PT1)
WRITE (*,*) ASSOCIATED (PT1)  ! Write .FALSE.
PT1 => PT2                 ! So PT1 points to TAB2
...

! On peut aussi s'allouer de la mémoire
ALLOCATE (PT1(20), STAT = ERR_CODE)
```

! Attention: surtout ne pas faire de NULLIFY de PT1
! sinon la mémoire allouée est perdue !
! il faut faire un DEALLOCATE avant

- Gestion de liste à l'aide de pointeurs :

Exemple d'une liste de villes.

```
MODULE VOYAGE

   TYPE VILLE
      TYPE (VILLE), POINTER :: PRECEDENTE
      CHARACTER (LEN=25)    :: NOM_VILLE
      TYPE (VILLE), POINTER :: SUIVANTE
   END TYPE VILLE
   TYPE (VILLE), POINTER :: DEBUT
   TYPE (VILLE), POINTER :: COURANT
   TYPE (VILLE), POINTER :: FIN

CONTAINS

SUBROUTINE DESCENDRE
  ! Parcours de la liste
  WRITE (*,*) 'Sens du voyage: descente'
  IF (.NOT. ASSOCIATED (DEBUT)) RETURN
  COURANT => DEBUT
  DO
    WRITE (*,*) COURANT%NOM_VILLE
    IF (.NOT. ASSOCIATED (COURANT%SUIVANTE)) EXIT
    COURANT => COURANT%SUIVANTE
  END DO
END SUBROUTINE DESCENDRE

SUBROUTINE MONTER
  ! Parcours inverse de la liste
  WRITE (*,*) 'Sens du voyage: montée'
  IF (.NOT. ASSOCIATED (FIN)) RETURN
  COURANT => FIN
  DO
    WRITE (*,*) COURANT%NOM_VILLE
    IF (.NOT.ASSOCIATED (COURANT%PRECEDENTE)) EXIT
    COURANT => COURANT%PRECEDENTE
  END DO
END SUBROUTINE MONTER

END MODULE VOYAGE
```

```
PROGRAM VACANCES
USE VOYAGE
CHARACTER * 25      :: NOM_LU
INTEGER, PARAMETER :: IIN = 7
INTEGER             :: IR
! Initialisations
NULLIFY (DEBUT)
NULLIFY (COURANT)
NULLIFY (FIN)
...
! Lecture et rangement du premier nom de ville
READ (UNIT=IIN, FMT=1000, END=100) NOM_LU
1000 FORMAT (A)
ALLOCATE (DEBUT, STAT=IR)
IF (IR /= 0) THEN
    ...
END IF
NULLIFY (DEBUT%PRECEDENTE)
DEBUT%NOM_VILLE = NOM_LU
NULLIFY (DEBUT%SUIVANTE)
COURANT => DEBUT
FIN => DEBUT
....
DO
    ! Lecture et rangement des villes suivantes
    READ (UNI=IIN, FMT=1000, END=100) NOM_LU
    ALLOCATE (FIN%SUIVANTE, STAT=IR)
    IF (IR /= 0) THEN
       ....
    END IF
    FIN => FIN%SUIVANTE
    FIN%PRECEDENTE => COURANT
    FIN%NOM_VILLE = NOM_LU
    NULLIFY (FIN%SUIVANTE)
    COURANT => FIN
END DO
100 CONTINUE
....
! Affichages de la liste
CALL DESCENDRE
CALL MONTER
....
```

- pointeurs : passage en argument d'un sous-programme

```
PROGRAM PTR1
   INTERFACE
      SUBROUTINE SUBA (ARGA)
         REAL, DIMENSION(:) :: ARGA
      END SUBROUTINE SUBA
      SUBROUTINE SUBP (ARGP)
         REAL, DIMENSION(:), POINTER :: ARGP
      END SUBROUTINE SUBP
   END INTERFACE
   REAL, DIMENSION(15,20), TARGET :: TAB
   REAL, DIMENSION (:), POINTER   :: PTR
   . . .
   PTR => TAB(:,3)
   CALL SUBA (PTR)
   . . .
   CALL SUBP (PTR)
   . . .
END PROGRAM PTR1
!=====================
SUBROUTINE SUBA (ARGA)
   REAL, DIMENSION(:) :: ARGA
   . . .
   ARGA (I) = . . .
END SUBROUTINE SUBA
!=====================
SSUBROUTINE SUBP (ARGP)
   REAL, DIMENSION(:), POINTER :: ARGP
   . . .
   ARGP=> . . .
END SUBROUTINE SUBP
```

8 - Constructions obsolètes :

Arithmetic IF	IF (X), 10, 20, 30
Alternate return	CALL SUB (X, Y, *400, *500)
ASSIGN	ASSIGN 600 TO I
Assigned format	ASSIGN 700 TO I ; WRITE (6,I)
Assigned GOTO	ASSIGN 800 TO I ; GOTO I, (800, 900)
DO loop without integers limits	DO 10, X = 1.1, 9.9
DO loop without CONTINUE	DO 11, i=1,5 ; 11 x=x+1.0
Branch inside IF block	
H edit descriptor	1H+ or 11Hhello world
PAUSE	Pause 'mount tape'

Lors de la prochaine révision du standard les constructions obsolètes vont être supprimées, et un certain nombre de choses généralement redondantes, vont probablement être déclarées obsolètes.

Ceci	Peut-être remplacé par
Fixed source form	Free (or mixed) source form
DOUBLE PRECISION	REAL (KIND = KIND(1.0D0))
Computed GOTO	CASE
CHARACTER *8	CHARACTER (LEN=8)
EQUIVALENCE	Allocation dynamique ou pointeur
COMMON	MODULE
BLOCK-DATA	MODULE
ENTRY	MODULE PROCEDURE
INCLUDE	USE d'un module

9- Compilation :

Règle: Il faut compiler les modules avant les unités de programme qui les utilisent par l'instruction USE.
En effet en plus du fichier objet .o le compilateur génère pour chaque module un fichier .mod ("module symbole file") qui contient des informations nécessaires lors de la compilation des unités de programme qui utilisent le module (USE).

Sur IBM RS6000 :
xlf: appel du compilateur avec des options par défaut donnant une compatibilité avec l'ancien FORTRAN 77.

xlf90: appel du compilateur avec des options par défaut pour FORTRAN 90 (donne accès à l'ensemble du langage avec certaines extensions IBM).

options:	-qlanglvl=90std	FORTRAN 90 standard (sans extensions)
	-qlanglvl=90pure	FORTRAN 90 - constructions obsolètes
	-qfixed	si source en forme fixe

10- Bibliographie :

Aberti	Fortran 90 initiation à partir du F	1992
Adams, J	Fortran 90 : a programmer's guide	1990
Adams, J	Fortran 90 Handbook	1990
Metcalf, M	Fortran 90 explained	1990
Ellis, T	Fortran 90 programming	1994
Couniham, M	Fortran 90	1991
Smith, I.M	Programming in Fortran 90	1994
Berlinger, E	Understanding Fortran 77 and 90	1993
Edgar, S	Fortran for the 90's	1992
Kerrigan, J	Migrating to Fortran 90	1993
Metcalf, M	Fortran 90 les concepts fondamentaux	1993
Delannoy, C	Programmer en Fortran 90	1992
Dubesset, M	Specificités du Fortran 90	1993
Lignelet, P	Fortran 90 approche par la pratique	1993