Introduction à Fortran 90

Nicolas Kielbasiewicz*

19 mars 2008

Fortran pour FORmula TRANslator est comme son nom l'indique dédié au calcul scientifique car très performant de ce point de vue. Fortran 77 est encore à l'heure actuelle l'un (pour ne pas dire le) des langages les plus présents parmi la communauté scientifique. Il présente néanmoins quelques inconvénients et contraintes majeurs (à l'époque, les ordinateurs fonctionnaient avec des cartes perforées) et c'est la raison pour laquelle Fortran 90 est apparu.

L'objet de ce document n'est pas de faire un cours de Fortran , mais de rassembler sous forme d'un aide-mémoire les commandes essentielles.

Table des matières

*Unité de Mathématiques Appliquées, École Nationale Supérieure de Techniques Avancées

1 Types de données et attributs

1.1 Les types de données

1.1.1 Les types INTEGER, REAL et COMPLEX

Ce sont les types qui permettent de représenter des entiers relatifs, des nombres flottants et des nombres complexes. Il convient d'introduire ici la notion de précision. FORTRAN propose en effet la simple précision et la double précision.

	INTEGER	REAL	COMPLEX
simple précision	2 octets	4 octets	4 octets
double précision	4 octets	8 octets	8 octets

TAB. 1 – Espace mémoire réservé suivant la précision

Comment manipuler la précision ? Par défaut, les types sont en simple précision. Si toutefois, on veut la double précision, on va alors utiliser la syntaxe suivante :

```
real (kind=8) :: x,y
integer (kind=4) :: i,j
complex(kind=8) :: c
```

1.1.2 Le type LOGICAL

C'est le type qui permet de gérer les booléens en FORTRAN . Les deux valeurs possibles sont .true. et .false.

1.1.3 Le type CHARACTER

C'est le type qui permet de gérer les chaînes de caractères. Les chaînes de caractères sont écrites indifféremment entre guillemets ou entre apostrophes.

```
character (LEN:20) :: s,t
s='toto'
t="Ce texte est&
        &écrit sur&
        &plusieurs lignes"
s(2:2) = 'a'
s(3:4)='ut'
```

1.1.4 Le type TYPE

C'est l'équivalent des structures en C. Il permet de définir son propre type de données.

```
type voiture
        character (len=20) :: marque
        character (len=20) :: modele
        integer :: annee
end type voiture

type(voiture) :: lamienne

lamienne%marque = Peugeot
lamienne%modele = 308
lamienne%annee = 2008
```

1.2 Les attributs

1.2.1 L'attribut PARAMETER

Cet attribut permet de spécifier que la variable ainsi définie est une constante dont la valeur ne pourra être modifiée au cours du programme.

1.2.2 L'attribut DIMENSION

C'est l'attribut qui permet de définir des tableaux, et plus précisément sa taille. Si on précise un entier, il s'agira de la taille du tableau et les indices commenceront à 1. Si on précise une plage de valeurs, il s'agira des valeurs des indices. Si on précise plusieurs éléments, il s'agira d'un tableau à plusieurs dimensions.

```
real, dimension(11) :: a
integer, dimension(-10:5) :: b
character (len = 10), dimension(5,-4:12) :: c
```

1.2.3 Les attributs POINTER et TARGET

En FORTRAN , on distingue les *pointeurs* et les *cibles*, les premiers poitant sur les derniers.

```
integer, pointer :: ptr
integer, target :: n
n = 10
ptr => n
```

Modifier la valeur d'un pointeur modifie en réalité la valeur de la cible qui lui est associée. FORTRAN nous permet ainsi d'utiliser les pointeurs de la même manière que des variables standard (initialisation, affectation, ...). Nous verrons plus loin que l'on peut se passer de la cible et manipuler des pointeurs de la même manière qu'en C.

2 Structures de contrôle et expressions logiques

2.1 IF

```
if ( x < 3) then
        x=x+3
else if (x > 0 ) then
        x=0
else
        x=x-3
end if
```

2.2 SELECT CASE

```
select (y)
        case (0)
                y=y+2
        case (1,3)
                y=0
        case (4:7)
                y=y+3
        case default
                y=1
end select
```

2.3 DO et DO WHILE

L'instruction DO remplit le double rôle d'une boucle pour et d'une boucle tant que.

```
do i = 1, 10
        x=x+2
end do
```

```
integer :: s
s=0
do
        if(s>=10) exit
        s=s+1
end do
```

Dans ce dernier exemple, on utilise en réalité une syntaxe de boucle infinie dans laquelle on va nécessairement inclure une condition d'arrêt ? C'est ce que l'on appelle une boucle "jusqu'à".

On peut également définir un boucle "tant que".

```
integer :: s
s=0
do while (s<10)
        s=s+1
end do
```

2.4 Les instructions GOTO et STOP

L'instruction GOTO est un reliquat de FORTRAN 77. Elle permet d'aller directement à la ligne de code correspondant à l'étiquette donnée en argument.

```
if (x>0.0) then
        y=log(x)
else
        goto 99
endif
print *, "log(x) = ", x
...
99 print*, "x est negatif"
...
```

Nous avons vu précédemment l'instruction EXIT qui permet de sortir d'une boucle tant que. L'instruction STOP permet d'arrêter totalement l'exécution du programme. On l'utilise donc essentiellement dans la gestions des erreurs.

3 Structure d'un programme

3.1 Programmes, procédures et fonctions

3.1.1 Le programme

Toute programme commence et se termine par :

```
program nom_programme
...
end program nom_programme
```

3.1.2 Les procédures ou subroutines

Les procédures sont des sous-programmes qui prennent des arguments d'entrée et et des arguments de sortie :

```
program toto
...
call tata(a,b,c)
...
end program toto

subroutine tata(a,b,c)
real, intent(in) :: a ! argument d'entree a ne pas modifier dans la routine
real, intent(inout) :: b ! argument d'entree et de sortie
real, intent(out) :: c ! argument en sortie  a lui affecter obligatoirement
    une valeur
end subroutine tata
```

On peut les définir à l'intérieur d'un programme, après le mot clé dédié **CONTAINS**. On parlera dans ce cas de procédures internes.

3.1.3 Les fonctions

A la différences des procédures, les fonctions ne retournent qu'un seul argument qui porte le nom de la fonction. L'appel se fait aussi de manière différente :

```
program toto
...
real :: y
```

```
y= tata(a,b,c)
...
end program toto

function tata(a,b,c)
real, intent(in) :: a,b,c
real :: tata
tata=a+b*c
end function tata
```

Tout comme les procédures, les fonctions peuvent être définies de manière interne.

3.1.4 Les fonctions récursives

Ce sont des fonctions qui s'appellent elle-mêmes. Leur syntaxe est un peu particulière, comme on peut le voir avec la définition de la fonction factorielle :

```
recursive function fact(n) result(res)
integer, intent(in) :: n
real :: res
if (n <=1) then
        res=1
else
        res=fact(n-1)*n
endif
end function fact
```

3.2 Modules, interfaces et généricité

3.2.1 Les interfaces

Les interfaces jouent un rôle très important et quasi systématique en FORTRAN 90. Elles permettent de définir les prototypes des fonctions et subroutines définies de manière externe. Dans ce rôle, elles sont les analogues des inclusions de fichiers de header en C.

```
function toto(a,b)
  implicit none
  real, intent(in) :: a, b
  real :: toto
end function toto

program test
  implicit none
  ...
  interface
    function toto(a,b)
      real, intent(in) :: a, b
      real :: toto
    end function toto
  end interface
  ...
end program test
```

Les interfaces permettent également de surcharger des méthodes pour plusieurs types d'arguments. Le bloc interface associé sera de la forme :

```
interface toto
  subroutine totoint(n)
    integer, intent(in) :: n
  end subroutine totoint
  subroutine totoreal(r)
    ireal, intent(in) :: r
  end subroutine totoreal
end interface
```

3.2.2 Les modules

Il s'agit d'un autre point fort de FORTRAN 90, à savoir la possibilité de programmer des modules, c'est-à-dire des librairies particulières dont on peut choisir d'utiliser tout ou partie.

```
module mon_module
  type toto
    ...
  end type toto
  interface operator(.titi.) ! surcharge d'operateur
     module procedure tata
   end interface
   contains
     subroutine tata(a,b)
       ...
     end subroutine tata
     function tutu(a,b)
       ...
     end function tutu
end module mon_module

program test
  use mon_module, only : tutu ! chargement de la fonction tutu du module
  implicit none
  ...
end program
```

Quelques remarques :

- Dans cet exemple, on montre l'utilisation de **ONLY**. Si l'on veut charger tout un module, on écrira **USE** mon_module.
- Le chargement d'un ou plusieurs modules se fait en tout début de programme, avant la commande **IMPLICIT NONE** et les déclarations et initialisations de variables.
- Cet exemple montre également la notion de surcharge (ou surdéfinition) d'opérateur. Possibilité est donnée non seulement de surcharger les opérateurs existants (+, -, *, /, .and., <, ...) à condition que ce soit pour des types différents que ceux pour lesquels ces opérateurs sont déjà définis, mais aussi de définir de nouveaux opérateurs dont le nom sera de la forme .NOM. .

4 Compléments sur les tableaux

4.1 Manipuler des tableaux

4.1.1 Sections de tableaux

Encore un point fort de FORTRAN , la possibilité de manipuler des sections de tableaux avec des syntaxes de la forme a(debut:fin:pas). Si le pas est omis, il a la valeur 1 par défaut.

4.1.2 initialisation / affectation

L'exemple ci-dessous montre les deux manières standard d'initialiser un tableau, la manière globale et la manière élémentaire :

```
real, dimension(5) :: a, b
a=(/ 2., 3.2, -1.0, 5.9, 4.5 /) # initialisation globale
b=(/ (2*i+1, i=1,5)/) # initialisation globale
a(2)=-1.5 # initialisation d'un element
```

La "manière"globale" ne fonctionne que pour des tableaux de rang 1. Pour des tableaux multidimensionnels, on est donc amené à utiliser la manière élémentaire ou d'utiliser un tableau temporaire de rang 1 que l'on redimensionnera à l'aide de la fonction `reshape` :

```
program init_tab
  implicit none
  real, dimension(6) :: t1
  real, dimension(2,3) :: t2
  t1 = (/ 2., -1.5, 3.7, 0., -2.9, 1. /)
  t2=reshape(t1, (/ 2, 3 /) )
end program init_tab
```

t1 =

2.0
-1.5
3.7
0.0

t2 =

2.0	3.7
-1.5	0.0

4.1.3 Opérateurs élémentaires

Tout opérateur défini sur des types donnés fonctionnent avec des tableaux de mêmes types. Ces opérateurs fonctionneront alors élément à élément, à la condition que les tableaux impliqués soient de même profils pour les opérateurs binaires (+, -, *, /, ...).

Une conséquence pratique est la possibilité d'affecter une valeur à chacun des éléments d'un tableau avec l'instruction a=0..

Il existe par ailleurs un certain nombre de fonctions élémentaires :

- sum(t1), product(t1) ! calcule la somme (le produit) de tous les éléments de t1
- size(t1,i) ! donne l'étendue du tableau t2 selon la dimension i
- maxval(t1), minval(t1) ! fournit la plus grande (petite) valeur de t1
- dot_product(t1,t2) ! calcule le produit scalaire de deux tableaux de rang 1 et de même taille t1 et t2, qui peuvent être des tableaux de réels, d'entiers, de complexes et même de booléens.
- matmul(A,B) ! calcule le produit A*B de deux tableaux de rang 2 et dont les dimensions sont compatibles.

4.1.4 L'instruction WHERE

C'est la généralisation du IF pour des tableaux, dans le sens où on va effectuer une opération logique sur un tableau et suivant le résultat élément par élément, effectuer une affectation spécifique. L'exemple suivant montre comment récupérer la partie positive d'un tableau, c'est-à-dire effectuer l'opération b=max(a,0) :

```
where(a>0.)
  b=a
else where
  b=0.
end where
```

On pourrait dans l'exemple précédent écrire :

```
b=0.
where(a>0.)
  b=a
end where
```

ou

```
b=0.
where(a>0.) b=a
```

4.2 Passage en arguments - profils implicites et automatiques

J'ai inclus ici les chaînes de caractères qui se comportent quelque peu comme des tableaux de caractères.

Lorsqu'on déclare un argument d'entrée d'une fonction ou d'une procédure, on peut souhaiter ne pas définir explicitement le profil d'un tableau. FORTRAN 90 autorise la définition implicite. Pour les tableaux en argument de sortie, on parlera de profil automatique.

```
function fct(a,b)
  real, dimension(:,:), intent(in) :: a
  character(len=*), intent(in) :: b
  real, dimension(size(a,1),size(a,2)) :: fct
  ...
end function fct
```

4.3 Tableaux dynamiques - attribut ALLOCATABLE ou POINTER

Une autre possibilité est de déclarer un tableau et d'allouer dynamiquement la mémmoire au moment où on en a besoin. Dans ce cas, on déclare un tableau de profil implicite avec l'attribut ALLOCATABLE.

```
program toto
  implicit none
  real, dimension(:,:), allocatable :: tab
  ...
  allocate(tab(10,20))
  ...
  deallocate(tab)
end program toto
```

Une remarque importante, la désallocation d'un tableau dynamique doit se faire dans le même bloc programme (un programme, une fonction, ou une procédure) que la déclaration et l'allocation. Cela a pour conséquence fondamentale que l'on ne peut déclarer un tableau dynamique comme argument de sortie d'une fonction ou d'une procédure, du moins de manière rigoureuse, car certains compilateurs l'autorisent.

La solution à adopter est d'utiliser un pointeur de tableau. Comme annoncé plus haut dans le paragraphe concernant les pointeurs, on peut se passer d'une cible et d'allouer la mémoire avec la fonction ALLOCATE, comme on le ferait en C avec la fonction `malloc`.

```
program toto
  implicit none
  real, dimension(:,:), pointer :: tab
  ...
  allocate(tab(10,20))
  ...
  deallocate(tab)
end program toto
```

Dans ce cas, on peut passer un tableau "dynamique" en argument de sortie.

5 Les formats

5.1 Les différents formats

An représente n caractères.

In représente un entier sur n caractères.

Fn.p représente un réel en notation flottante avec n caractères dont p décimales. La virgule compte pour un caractère.

En.p représente un réel en notation exponentielle avec n caractères au total dont p décimales dans la mantisse.

nX représente n espaces.

Ln représente un booléen sur n caractères.

Tn représente un positionnement au n-ième caractère.

/ représente un passage à la ligne.

***** représente un format libre.

5.2 Formats et entrées / sorties standard

L'exemple suivant montre la syntaxe des entrées / sorties standard, avec différents formats, y compris par le biais d'étiquette et de la fonction `format` :

```
program iostd
  implicit none
  integer :: n=5
  real :: x=0.5
  character(len=6) :: s="test :"
  print *, s, n, x
  print "(1x,a6,i4,1x,f8.3)", s, n, x
  print 199, n, x
  199 format(1x,'test :',i4,1x,f8.3)
end program iostd
```

6 La gestion des entrées / sorties

6.1 Entrées / sorties séquentielles

Les entrées / sorties séquentielles représentent l'utilisation des fichiers la plus courante et la plus élémentaire, à savoir le parcours d'un fichier ligne après ligne.

```
program ioseqfile
  implicit none
  integer :: a
  real :: r
  open(unit=12, form='formatted', file='toto.txt')
    read(12,*) a
  close(12)
  ...
  open(unit=25, form='unformatted', file='tata.txt', &
       status='new')
  write(25) r
  close(25)
end program ioseqfile
```

6.1.1 OPEN et CLOSE

Les arguments les plus courants des commandes OPEN et CLOSE sont :

unit= (obligatoire en premier argument) la valeur 5 est réservée pour l'entrée standard, la valeur 6 est réservée pour la sortie standard. De manière générale, $1 < \text{unit} < 99$.

status= permet de spécifier s'il s'agit d'un fichier existant ('old'), à créer ('new'), à remplacer ('replace'), temporaire ('scratch') ou de statut quelconque ('unknown')

form= (open uniquement) permet de spécifier si le fichier ouvert est binaire ('unformatted') ou ASCII ('formatted'), c'est à dire lisible par tout éditeur de texte.

iostat=val la variable entière *val* sera initialisée à une valeur strictement positive en cas d'erreur d'entrée / sortie, ou nulle sinon.

err=etiq permet de se positionner à l'étiquette *etiq* en cas d'erreur de lecture / écriture.

position= (open uniquement) permet de se positionner dans le fichier ouvert, au début ('rewind'), à la fin ('append'), ou à la dernière position en date si fichier déjà ouvert ('asis').

6.1.2 READ et WRITE

Les arguments les plus courants des commandes READ et WRITE sont :

unit (obligatoire en 1er argument) pour spécifier sur quel fichier on lit / écrit.

fmt (fichier formaté uniquement) (obligatoire en 2ème argument) pour spécifier le formatage.

iostat=val la variable entière *val* sera initialisée à une valeur strictement positive en cas d'erreur d'entrée / sortie, ou nulle sinon.

err=etiq permet de se positionner à l'étiquette *etiq* en cas d'erreur de lecture / écriture.

end=etiq (read uniquement) permet de se positionner à l'étiquette *etiq* en cas de fin de fichier atteint prématurément.

eor=etiq (read uniquement) permet de se positionner à l'étiquette *etiq* en cas de fin d'enregistrement (ligne) atteint prématurément.

6.2 Entrées / sorties à accès direct

A la différence des entrées / sorties séquentielles, FORTRAN permet de gérer des fichiers à accès direct, c'est-à-dire des fichiers où l'on peut accéder à une ligne en particulier sans avoir parcouru les précédentes. Une des conséquence est que l'on peut lire et écrire en même temps dans un fichier à accès direct, ce qui ne peut être le cas des entrées / sorties séquentielles.

```
program acces_direct
  implicit none
  character(len=12) :: prenom='nicolas',pre
  character(len=20) :: nom='kielbasiewicz',n
  integer :: annee=2008,a
  open(unit=15, file='table.txt', access='direct', &
       recl=44, form='unformatted',status='new')
  write(15,rec=2) nom, prenom, annee
  read(15,rec=1) n,pre,a
  close(15)
end program acces_direct
```

Les arguments spécifiques sont :

access='direct' pour spécifier que le fichier est en accès direct

recl=44 pour spécifier la taille en octet (ou caractère) d'un enregistrement

rec=2 pour spécifier le numéro d'enregistrement.

6.3 Entrées / sorties internes

Cela correspond à la manipulation de chaînes de caractères comme de simples fichiers

```
program iostr
  implicit none
  character(len=10) :: a='0123456789'
  integer :: n, p=0
  real :: x
  read(a, '(i3, f6.2)') n, x
  print *, n, x
  print '(i3,1x,f7.2)', n, x
  write(a(2:3),'(i2)') p
  print *, a
end program iostr
```