Université M^{ed} Boudiaf de M'sila

Faculté des Sciences

Département de Physique

OUTILS INFORMATIQUES POUR LA PHYSIQUE :

INITIATION AU FORTRAN 77 & F90 (et F95, F2003)

Dr. Abboud METATLA

(2020 - 2021)

Avant-Propos

Ce document a été initialement rédigé pour les étudiants du **Master I, Physique Appliquée,** de l'université de M'sila, comme document d'accompagnement du module **Outils Informatique pour la Physique**. Puis corrigé et remis en page pour qu'il soit à l'usage de tous les étudiants qui cherchent d'acquérir les connaissances de bases nécessaires à l'élaboration de programmes écrits sous le langage Fortran (en particulier F90).

Je tiens à remercier tous qui ont contribué de près ou de loin à l'élaboration de ce document, et surtout les étudiants du Master I, Physique Appliquée (années universitaires 2016 - 2020).

.

Table des matières

I. INITIATION AU LANGAGE FORTRAN

1. Introduction.

Le nom Fortran vient de l'expression (**For**mula **Tran**slation), qui est historiquement, le premier langage informatique de haut-niveau. Ce langage est apparu à la fin des années 1950 sous l'impulsion de John Backus.
Le langage Fortran a été standardisé en 1972 sous la forme du Fortran 66 et son efficacité dans le calcul scientifique en a fait le langage le plus utilisé dans les applications non commerciales. Puis il a été standardisé à la fin des années 1970 sous la forme Fortran77 où il a apporté d'énormes améliorations en particulier dans le traitement des chaînes de caractères.
Durant la révolution informatique des années 1980, le langage a évolué au Fortran 90, avec l'introduction des fonctionnalités nouvelles présentes dans des langages plus récents comme C, C++,…. La norme Fortran 90 a permis la modernisation du langage.
Cette nouvelle version a permis un nettoyage des éléments les plus obsolètes du fortran (format fixe par exemple, lié à l'utilisation des cartes perforées).
L'évolution du langage Fortran a continué avec le Fortran 95 (révision mineure), Fortran 2003 (le standard a été publié en novembre 2004).
La nouvelle norme Fortran 2008 constitue une évolution mineure par rapport au Fortran 2003 (introduction de notion de sous-module, des outils de programmation parallèle, de nouvelles fonctions mathématiques intrinsèques...).

1ère Séance

2. Mise en œuvre d'un programme.

En général, il y a quatre étapes pour la mise en œuvre d'un programme :

1-Edition ----- > fichier source (exemple : test1).
2-Compilation ----- > fichier objet (test1.o).
3- Edition des liens -----> fichier exécutable (test1.x).
4- Exécution ----- > (erreurs,…, il faut les corrigées).

Exemple 1: test1

```
c ---col.7--------------------format fixe-----------------------col.72
c---------- pour introduire des commentaires (et aussi !) -----
c
       program bienvenue           ! nom du programme
c
       write(*,*) "Bonjour"        ! affichage sur écran
c
       stop
        end
c-------------------------------------------------------------
```

Exemple 2: test2

```
      program bienvenue           !  nom du programme
c
      print*, "Bonjour"           !  affichage sur écran
c
      end program bienvenue
```

Exemple 3: test3

```
!------------ --------format libre--------------------col.132
       program bienvenue           !  nom du programme
!
       write(6,*)  "Bonjour"       !  affichage sur écran
!
       end program bienvenue
```

Exemple 4: test4

```
      program circle
c
      real r, area
c     this program reads a real number r and prints
c     the area of a circle with radius r.
      write (*,*) 'give radius r:'
      read (*,*) r          !   donnée introduite par clavier
c
      area = 3.14159*(r**2)
      write (*,*) 'area = ', area
c
      end program circle
```

Exemple 5: test5

```
      program circle
c
      real r, area
c     this program reads a real number r and prints
c     the area of a circle with radius r.
      write (6,*) 'Give radius r:'
      read (*,*) r          !   donnée introduite par clavier
c
      area = 3.14159*(r**2)
      write (6,*) 'area = ', area
c
      end program circle
```

2ème Séance

3. Caractères, types, variables et déclaration.

L'ensemble des caractères du fortran 90 est constitué des caractères alphanumériques, de signes de ponctuation et de quelques symboles spéciaux. Il comprend :
– les minuscules (lettres de a à z),
– les majuscules (lettres de A à Z),
– les chiffres arabes de 0 à 9,
– le caractère souligné : « _ »,
– et les symboles suivants, parmi lesquels $ et ? n'ont pas de signification particulière dans le langage :

	+	-	=	*	(	)
,	.	:	;	‘	"	/
<	>	%	!	&	$	?

Notons que la norme 2003 a complété le jeu de caractères du fortran par d'autres caractères spéciaux.
Le langage fortran (90 et plus) possède les types intrinsèques ou prédéfinis suivants :
– *CHARACTER* : chaîne de caractères
– LOGICAL : booléen
– types numériques :
  – type numérique représenté exactement :
    – INTEGER : entier
  – types numériques représentés approximativement :
    – REAL : réel en virgule flottante
    – DOUBLE PRECISION : flottant en précision étendue

– COMPLEX : nombre complexe flottant

a) Caractéristiques des types numériques prédéfinis.

Types entiers par défaut :

Sur un processeur de type PC 64 bits, les entiers par défaut peuvent être stockés sur 8 octets, (même si ce n'est pas forcément la règle), donc 63 bits pour la valeur absolue plus un bit de signe.

Nbr d'octets	Bit_Size	Digits	Huge	Range
4	32	31	2147483647 = $2^{31} - 1$	9
8	64	63	9223372036854775807 = $2^{63} - 1$	18

Voir référence [1].

Types réels par défaut :

Les réels sont, sur la plupart des machines, stockés sur 4 octets, dont 24 bits (dont 1 pour le signe) pour la mantisse et 8 bits pour l'exposant. Avec des options de compilation, on peut passer à 8 octets dont 53 bits (dont 1 pour le signe) pour la mantisse et 11 bits pour l'exposant (variante de réels aussi disponible sous le type **double precision**).

Octets	Digits	Precis.	Epsilon	Tiny	Huge
4	24	6	1.192093E-7	1.175494E-38	3.402823E+38
8	53	15	2.220446049250313E-16	2.225073858507201E-308	1.797693134862316E+308

Voir référence [1].

b) Déclaration :

Les objets doivent être déclarés en tout début de programme, sous-programme (ou subroutine), fonction ou module, précisément avant les instructions exécutables, selon les syntaxes générales dans les exemples ci-dessous.

Exemple 6: test6

```
      program circle
c
      implicit none
c
      real r, area, pi
c
      parameter (pi = 3.14159)
c
      write (*,*) 'Give radius r:'
      read (*,*) r
      area = pi*(r**2)
      write (*,*) 'Area = ', area
c     instruction selon la norme f77
      stop
      end
```

Exemple 7: test7

```
      program circle
c
      implicit none
c
      real:: r, area
c
      real, parameter:: pi = 4.*atan(1.)
```

```
c
      write (*,*) 'Give radius r:'
      read (*,*) r
      area = pi*(r**2)
      write (*,*) 'Area = ', area
c
      end program circle
```

Remarque :

Par défaut, les variables numériques non déclarées suivent un typage implicite déterminé par leur initiale : les variables dont l'initiale est I, J, K, L, M ou N sont des entiers et les autres des flottants. Il est très fortement conseillé de s'obliger à déclarer toutes les variables, sans mettre en œuvre le typage implicite, grâce à l'ordre **implicit none**, placé avant toutes les déclarations.

3ème Séance

4. Opérateurs et expressions.

a) Les opérateurs numériques

Le fortran dispose des quatre opérateurs binaires d'addition (+), de soustraction (-), de multiplication (*) et de division (/), et possède aussi l'opérateur d'élévation à la puissance, noté **.

Concernant les règles de priorité entre ces opérateurs numériques binaires, lorsque l'ordre d'évaluation n'est pas imposé par l'utilisation de parenthèses, les expressions numériques sont évaluées en respectant l'ordre de priorité suivant : (1) ** (2) * et / (3) + et -.

Remarque :

Pour éviter des erreurs d'arrondi sur les réels en précision limitée ou des dépassements de capacité sur les réels comme sur les entiers, il est donc parfois prudent d'imposer l'ordre d'évaluation par des parenthèses, voire de reformuler les expressions.

b) Les opérateurs de comparaison

Le fortran possède six opérateurs de comparaison dont le résultat est une variable booléenne. L'ancienne notation du fortran77 est toujours acceptée.

Fortran90 et plus	Fortran77	Signification
<	.LT.	inférieur à
<=	.LE.	inférieur ou égal à
==	.EQ.	égal à
>=	.GE.	supérieur ou égal à
>	.GT.	supérieur à
/=	.NE.	différent de

Exemple 8: test8

```
      program comparaison
c
      implicit none
c
      real    :: a, b, r
      integer :: i
c
      write (*,*)"entrer a"
```

```
      read (*,*) a
      if (a > 0) b = log(a) ! a > 0 est évalué à .true. si a est positif
c
      write(*,*) "b=",b
      write (*,*)"entrer i"
c
      read (*,*) i
c
      if ( i /= 0 ) r = 1. / real(i)
c
      write(*,*)"r=",r
c
      end program comparaison
```

Remarque :

Étant donné que les nombres flottants ne sont représentés qu'approximativement en machine, les tests d'égalité entre flottants sont déconseillés ; on préférera comparer les différences à un seuil, en utilisant la fonction intrinsèque EPSILON.
On remplacera par exemple :

```
if ( a == b ) then
```

Par :

```
if ( abs(a-b) <= c *abs(a) * epsilon(a) ) then
```

Où c est une constante supérieure à 1.

c) Les opérateurs booléens

Le fortran dispose de quatre opérateurs logiques binaires permettant de combiner des expressions logiques entre elles, ainsi que de l'opérateur unaire de négation.

.AND.	ET
.OR.	OU (inclusif)
.NOT.	Négation unaire
.EQV.	Équivalence
.NEQV.	OU exclusif

d) Concaténation des chaînes de caractères

L'opérateur // permet de concaténer deux chaînes de caractères.

Exemple 9: test9

```
      program concat_chaines
c
      implicit none
c
      character (len=5)  :: jour='mardi'
      character (len=4)  :: mois='mars'
      character (len=13) :: date
c
      date = jour//'-29-'//mois   ! contiendra 'mardi-29-mars'
c
      print*,date
c
      end program concat_chaines
```

4ème Séance

5. Structures de contrôle.

a) Structure if

Cette structure est utilisée pour l'exécution conditionnelle d'une instruction ou un bloc d'instructions.

Exemple 10: test10

```
      program structure_if1
c
      implicit none

      integer :: i, j
      real ::k
c
      read (*,*) i,j

      if (j /= 0) k = real(i)/real(j)  !k n'est calculé que si j
différent de 0
c
      print*, k
c
      end program structure_if1
```

Exemple 11: test11

```
      program structure_if2
c
      implicit none
c
      integer :: i
c
      read (*,*) i
c
      if (i < 10) then
      write(*,*) 'i est inférieur à 10'

      end if
      end program structure_if2
```

Exemple 12: test12

```
      program structure_if3
c
      implicit none
c
      integer :: i
c
      read (*,*) i
c
      if (i < 10) then
      write(*,*) 'i est inférieur à 10'
c
        else
      write (*,*) 'i est supérieur ou égal à 10'
c
      end if
      end program structure_if3
```

Exemple 13: test13

```
      program structure_if4
c
      implicit none
c
      real :: a,b
c
      read (*,*) a,b
c
      if (a <= 0) then
      write(*,*) 'a est inférieur ou égal 0'
c
```

```
        else
            if (a > 0 .and. b <= 0) then
              write (*,*) 'a est supérieur à 0 et b …. '
            end if
        end if
        end program structure_if4
```

b) Structure `select case`

Généralement cette structure est utilisée lorsqu'on a plusieurs choix possible pour une même expression.

Exemple 14: test14

```
      Program select_case
c
      implicit none
c
      integer :: i
      read*,i
c
      select case(i)

c
      case(0)
      write (*,*) ' i = 0'
      case (1,-1)
      write (*,*) ' i = 1 ou i = -1'
      case (2:10)
      write (*,*) ' 2 <= i <= 10'
      case (11:)
      write (*,*) ' i >= 11'
      case default
      write (*,*) 'i < -1'
      end select
c
      end program select_case
```

5ème Séance

c) Structures de boucles

Pour effectuer des itérations, le fortran dispose des boucles **do**, de la boucle **do while**, de l'instruction **where**, ou de la structure **where ... end where**.

Exemple 15: test15

```
      Program boucle_01
c
      implicit none
      integer:: i, imax = 10, n
      n = 0
      do i = 1, imax, 2        ! donc i = 1, 3, 5, 7, 9 (pas=2)
        n = n + i
      end do
c
      write (*,*) 'somme des entiers impairs inférieurs à 10',n
c
      end program boucle_01
```

Exemple 16: test16

```
      Program boucle_02
c
      implicit none
      integer:: i, imax = 10, n
c
```

```
      n = 0
      i = 1
      do while ( i <= imax )
         n = n + i
         i = i + 2      ! modification de l'expression
      end do
      write (*,*) 'somme des entiers impairs inférieurs à 10', n
c
      end program boucle_02
```

Exemple 17a: test17a

```
      Program boucle_03
c
      implicit none
      integer:: i, n
c
      n = 0
      do i = 1, 1000, 2 ! a priori de 1 à 999
         if ( i > 10 ) exit      ! limitation à 10
           n = n + i
      end do
c
      write (*,*) 'somme des entiers impairs inférieurs à 10', n
c
      end program boucle_03
```

Exemple 17b: test17b

```
      Program avec_where
c
      real, dimension(5) :: a, b, c
      a=(/1.,2.,3.,-1.,-2./)    ! aussi [1.,2.,3,-1.,-2.]
c
      where ( a > 0. ) b = sqrt(a)   ! simple instruction where
      write(*,*)b
c
      where ( a > 0. ) c = log(a)               ! bloc where
             ! pour éléments positifs du tableau a
        else where
         c = -1.e20     ! pour éléments négatifs ou nuls du tableau a
      end where
c
      write(*,*)c
c
      end program avec_where
```

d) Autres structures de contrôle

Il y a plusieurs instructions très utilisées notamment dans les anciennes versions du fortran, comme par exemple :
L'instruction vide **continue**, très utilisée pour délimiter les fins de boucles.
L'instruction **go to (étiquette_numérique)** permet le branchement vers l'instruction repérée par l'étiquette numérique.
L'instruction **stop**.
L'instruction **return** qui transfère le contrôle de la procédure (sous-programme ou fonction) où elle est située à la procédure l'ayant appelée.
Notons qu'en fortran 90, **end** est un ordre exécutable et **stop** en fin de programme principal ainsi que **return** en fin de sous-programme ou de fonction sont donc devenus facultatifs.

6ème Séance

6. Entrées et sorties.

a) Entrées et sorties standard

Nous avons déjà utilisés les instructions **read*** et **print*** (ou **read (*,*)**, **write(*,*)**) qui sont des instructions de lecture et d'affichage standard (par le terminal, clavier et écran) en format libre.
Lors de la lecture d'un enregistrement, une combinaison quelconque des saisies suivantes au clavier :
– un ou plusieurs espaces ;
– une virgule ;
– un ou plusieurs changements de ligne,
est considérée comme un séparateur de données, permettant de passer à la lecture de la variable suivante. On peut arrêter la lecture de l'enregistrement avant d'avoir épuisé la liste par la saisie du caractère (**/**) de fin d'enregistrement. Il est aussi possible de sauter une variable au sein de la liste en saisissant deux virgules successivement (**,,**). Dans ces deux cas, les variables non lues gardent leur valeur antérieure.

Exemple 18: test18

```
      Program lecture
c
      implicit none
      integer :: i,j,k   ! 3 entiers initialisés
c
      print *, 'donner 3 entiers'
      read *, i, j, k
      print'(a3,i5.3,3x,a3,i5.3,3x,a3,i5.3)', 'i=', i, 'j=', j , 'k=', k
c
      end program lecture
```

Faire donner à (i, j, k) les valeurs (1 2 3), (1 ,, 3), (1 2 /) et voir ce que affichera le programme.
Notons aussi que si on fournit plus de données que nécessaire, les données superflues sont perdues, et elles ne sont disponibles pour l'instruction de lecture suivante.

b) Instructions d'entrées-sorties

Une opération de transfert de données requiert en général une instruction **open** de connexion entre unité logique et fichier, des instructions d'échange **read** ou **write**, et enfin une instruction **close** de libération de l'unité logique. Les instructions d'entrées-sorties **open, read, write, print** et **inquire** se partagent des arguments obligatoires ou optionnels accessibles par mot-clef.

Exemple 19: test19

```
      Program lect_fichier
c
      implicit none
c
      integer, dimension(5):: ti    ! tableau de 5 entiers
      real, dimension(5) :: tr      ! tableau de 5 réels
c
      open (2,file='donnee.dat')      ! ouverture du fichier
      read(2, *) ti,tr    ! lecture
c
      write(*,*)ti,tr
c
      close(2)
      end program lect_fichier
```

Il faut créer un fichier **donnee.dat** qui contient les deux tableaux **ti** et **tr**.

Instruction `open`

L'instruction **`open`** permet de connecter un fichier à une unité logique. La syntaxe de l'instruction est :

```
open (unit=entier, file='nom du fichier', err=étiquette,
status='…',position='…',…)
```

Ici, seuls les paramètres désignés par les mots-clefs **`unit`** et **`file`** sont obligatoires, la plupart des autres paramètres sont optionnels avec des valeurs par défaut indiquées ci-dessous :

- **`unit`**=entier, désigne le numéro de l'unité logique.
- **`file`**='nom du fichier', désigne le nom du fichier à connecter.
- **`err`**=étiquette, désigne une étiquette numérique permettant un branchement en cas d'erreur.
- **`status`**='…', désigne l'état du fichier à ouvrir et peut valoir :
  - **`'old'`** dans le cas d'un fichier préexistant.
  - **`'new'`** dans le cas où le fichier n'existe pas.
  - **`'replace'`** qui permet le remplacement ou la création.
  - **`'unkhown'`** dans le cas où on ne sait pas a priori si le fichier existe.
  - **`'scratch'`** pour un fichier temporaire qui sera détruit lors de l'instruction close.
- **`position`**='…', précise la position à l'ouverture d'un fichier en accès séquentiel :
  - **`'append'`** positionne à la fin du dernier enregistrement.
  - **`'rewind'`** positionne en début de fichier.
  - **`'asis'`** ouvre le fichier à la position courante s'il est déjà connecté.

Pour plus de détails concernant des restes des paramètres optionnels voir la référence [1].

`Exemple 20`: `test20`

```
      Program lect_fichier
c
      implicit none
c
      integer, dimension(5):: ti    ! tableau de 5 entiers
      real, dimension(5) :: tr      ! tableau de 5 réels
c
      open (2,file='nom fichier')    ! ouverture du fichier
c
      read(2,*) ti      ! lecture
      read(2,*) tr
      print*,ti
      print*,tr
c
      close(2)
c
      end program lect_fichier
```


Instruction `close`

L'instruction **`close`** permet de déconnecter une unité logique d'un fichier, d'où l'unité logique sera libre, ce qui permet une nouvelle connexion. La syntaxe de l'instruction est :

```
close (unit=entier, err=étiquette, status='…', .....)
```

Les arguments de **close** sont tous optionnels sauf l'unité logique et ont la même signification que dans l'instruction **open**, sauf **status** qui désigne ici le devenir du fichier après déconnexion et peut valoir :

- **'keep'** (par défaut) pour conserver un fichier préexistant.
- **'delete'** pour détruire le fichier à la déconnexion.

Instruction read

En plus de la variante de lecture standard, la syntaxe de lecture dans un fichier est la suivante :

```
read (unit=entier, err=étiquette, end= étiquette, .....)listes d'entrées.
```

Notons que seul le premier argument, l'unité logique est positionnel.

Instruction write

La syntaxe de l'instruction **write** est la suivante :

```
write (unit=entier, err=étiquette, .....)listes de sorties.
```

Où seul le premier paramètre, le numéro de l'unité logique, est positionnel.

Exemple 21: test21

```
      Program ecrit_fichier
c
      implicit none
c
      integer, dimension(4):: ti=(/0,1,2,3/)    ! tableau de 4 entiers
      real, dimension(3) :: tr=(/0.5,1.,2.7/)   ! tableau de 3 réels
c
      open (3,file='nom fichier')    ! ouverture
c
      write(3, *) ti,tr            ! écriture dans 'resul.txt'
c
      close(3)
c
      end program ecrit_fichier
```

Exemple 22: test22

```
       Program lect_fichier
c
       implicit none
c
       integer, dimension(5):: ti    ! tableau de 5 entiers
       real, dimension(5) :: tr     ! tableau de 5 réels
c
       open (5,file='donnee.txt')    ! ouverture du fichier
       open (7,file='res.txt')
c
       read(5,*)ti
       read(5,*)tr
         write(*,100)ti
         write(*,200)tr
         write(7,100)ti
         write(7,200)tr
 100   format(i5)
 200   format(f7.4)
c
       close(5)
       close(7)
       end program lect_fichier
```

Il faut créer un fichier qui contient **ti** et **tr**.

8ème Séance

c) Descripteurs de format

Les règles de conversion entre représentation interne des données et chaîne de caractères des fichiers formatés sont déterminées par le format de lecture ou d'écriture. Le format peut être spécifié sous une des formes suivantes :

- format libre, désigné par * ;
- chaîne de caractères constante ou variable décrivant le format entre parenthèses ;
- étiquette numérique (entier positif) faisant référence à une instruction dite de format.

On distingue les descripteurs actifs, qui spécifient le mode de conversion d'une donnée en chaîne de caractères ou réciproquement, les chaînes de caractères et les descripteurs de contrôle, qui, entre autres, règlent le positionnement, gèrent les espaces et les changements d'enregistrement.

Exemple 23: test23

```
       Program descrip_format
c
       implicit none
c
       character(len=4) :: fmt1, fmt2
       integer :: j = 12
c
     write(*, *) 'bonjour'      ! format libre
c
     write (*, '("bonjour")')
c
     write (*, '(a7)') 'bonjour'   ! chaîne de caractères constante
c
     fmt1='(a7)'
     write (*, fmt1) 'bonjour'     ! chaîne de caractères variable
c
     write (*, 1000) 'bonjour'     ! étiquette numérique
1000 format(a7)
!
     write (*, *) j        ! format libre
c
     write (*, '(i2)') j   ! chaîne de caractères constante
c
     fmt2='(i2)'
     write (*, fmt2) j     ! chaîne de caractères variable
c
     fmt2='(i4)'
     write (*, fmt2) j     ! chaîne de caractères variable
c
     write (*, 2000) j ! étiquette numérique
2000 format(i2)
c
     end Program descrip_format
```

Descripteurs actifs

D'une façon générale, le premier paramètre (entier), `n`, détermine le nombre total de caractères occupés par la donnée codée. La signification de l'éventuel deuxième paramètre `(p)` dépend du type de donnée à convertir.

Si le champ est plus large que nécessaire, le champ est complété par des blancs à gauche. Si la largeur `n` du champ de sortie est insuffisante, la conversion en sortie est impossible et provoque l'écriture de n signes `*`.

- Pour un **entier** : ***In[.p]*** *(p indique nombre minimal de chiffres y compris les zéros à gauche).*
- Pour un **réel** : ***Fn.p,*** en notation décimale *(p indique nombre de chiffre après la virgule).*
  - En virgule flottante :
    En.p *(mantisse (<1) plus exposant).*
    En.p[eq] *(mantisse (<1) plus exposant avec **q** chiffres).*
    ESn.p, notation scientifique *(mantisse entre 1 et 10)*.
    ENn.p, notation ingénieur *(mantisse entre 1 et 1000)*.
- Pour un booléen : ***Ln***
- Pour une chaîne de caractères : ***An***

Descripteurs de contrôle

- ***nx*** : provoque l'écriture de ***n*** blancs en sortie et le saut de ***n*** caractères en entrée.
- ***/*** : provoque le changement d'enregistrement (changement de ligne).
- Tabulations : ***Tn, TLn, TRn***

Une liste de descripteurs, séparée par des virgules, peut être érigée en groupe de descripteurs, délimité par des parenthèses.
Un facteur de répétition (entier positif) peut être appliqué à un descripteur de format ou à un groupe de descripteurs.

Exemple 24: test24

```
       Program descrip_format
c
       implicit none
c
      integer, dimension(4):: ti=(/0,1,2,3/)    ! tableau de 4 entiers
      real, dimension(3) :: tr=(/0.5,1.,2.7/)   ! tableau de 3 réels
c
      open (3,file='nom fichier')    ! ouverture
c
      write(3, 1000) 'ti=',ti,'tr=',tr       ! écriture dans 'resul.txt'
 1000 format(a5,4(i5),5x,a5,3(f7.5))
c
      close(3)
c
      end program descrip_format
```

Refaire l'exécution du programme pour d'autres descripteurs de formats (in.p, En.p, ESn.p,…)

9ème Séance

7. Procédures.

Lorsqu'un programme commence à dépasser une page ou comporte des duplications d'instructions, le mieux est de le scinder en sous-ensembles plus élémentaires, nommés procédures, qui seront appelées par le programme principal, ceci donne au code source la structure d'une hiérarchie de procédures plus concises et modulaires, qui permet :
– l'amélioration de la lisibilité et rend la vérification et la maintenance plus facile à faire;
– la réutilisation des outils déjà mis au point par d'autres applications, au prix d'une généralisation et d'un paramétrage des procédures.

a) Programme principal

Tout programme doit contenir un programme principal. C'est par lui que commence l'exécution du programme. Un programme contient un seul programme principal.

b) Sous programmes et fonctions

On a deux sortes de procédures :

–Sous-programmes (subroutine), qui sont constitués d'une suite d'instructions qui effectuent une tâche déterminée quand on les appelle depuis une autre unité de programme via l'instruction

CALL et rendent le contrôle à l'unité appelante après exécution.

– Fonctions (functions) sont des procédures qui renvoient un résultat sous leur nom, ce résultat sera ensuite utilisé dans une expression au sein de l'unité de programme appelante.

Notons qu'une fonction pourrait être considérée comme un cas particulier d'un sous-programme, et lorsqu'il s'agit d'effectuer une action qui n'est pas un simple calcul et par exemple modifier les arguments d'appel, on préfère d'utiliser un sous-programme au lieu d'une fonction.

Sous-Programme

Un sous-programme est introduit par l'instruction *SUBROUTINE*, suivie du nom du sous-programme et de la liste éventuelle de ses arguments muets entre parenthèses et séparés par des virgules, et délimité par l'instruction *END SUBROUTINE* 'nom du sous-programme'.

Exemple 25: test25

```
      Program calcul_surface
c
      implicit none
c
      real :: b1, h1, s1, b2, h2, s2
c
      b1 = 30.
      h1 = 2.
c
      call aire(b1, h1, s1)! appel avec arguments effectifs de même type
c
      print *, ' Surface = ', s1
c
      b2 = 2.
      h2 = 5.
c
      call aire(b2, h2, s2)! appel avec arguments effectifs de même type
c
      print *, ' Surface = ', s2
c
      end program calcul_surface
c     !------------------------------------ !
      subroutine aire(base, hauteur, surface)    ! arguments muets
c
      implicit none
c
      real :: base, hauteur     ! déclaration des arguments d'entrée
      real :: surface           ! déclaration des arguments de sortie
c
      surface = base * hauteur / 2.     ! calcul de surface
c
      end subroutine aire
```

Fonctions

Une fonction est encadrée par les déclarations FUNCTION, suivie du 'nom de fonction' et de la liste éventuelle de ses arguments muets, et l'instruction END FUNCTION 'nom de fonction' à la fin. Le type du résultat de la fonction peut être soit déclaré dans l'entête, comme préfixe de *FUNCTION*, soit déclaré en même temps que les arguments.

Exemple 26: test26

```
      Program calcul_surface
c
      implicit none
c
      real :: b1, h1, b2, h2
      real :: surf,somme
      b1 = 50.
      h1 = 4.
c
      print *, ' Surface = ', surf(b1, h1)
c
      b2 = 25.
      h2 = 5.
      print *, ' Surface = ', surf(b2, h2)
c
      somme= surf(b1, h1)+ surf(b2, h2)
      print*,' somme =',somme
c
      end program calcul_surface
c     !------------------------------------ !
      real function surf(base, hauteur)
c
      implicit none
c
      real :: base, hauteur     ! déclaration des arguments d'entrée
c
      surf = base * hauteur / 2.     ! affectation du résultat
c
      end function surf
```

10ème Séance

c) Procédures internes et procédures externes

Les procédures appelées (sous-programmes et fonctions) peuvent être :
– soit incluses dans le programme appelant, appelé aussi hôte (host) de la procédure et qualifiées de procédures internes; elles sont alors introduites par la déclaration *CONTAINS* ;
– soit extérieures au programme appelant (comme en fortran 77, **Exemple 25 et 26**) et qualifiées alors de procédures externes; où il est recommandé de les insérer dans des modules.

Exemple 27: test27

```
    program calcul_interne
c
    implicit none
c
    real:: base, hauteur, surface     ! variables globales
    base = 30.
    hauteur = 2.
c
    call aire
    print *, ' Surface = ', surface
c
```

```
      base = 2.
      hauteur = 5.
c
      call aire
      print *, ' Surface = ', surface
c
      contains
c
        subroutine aire     ! version déconseillée
          ! ne pas déclarer base, hauteur et surface (variables globales)
          ! sous peine de créer des variables locales distinctes !!!
c
       surface = base * hauteur / 2.
c
       end subroutine aire
c
      end program calcul_interne
```

d) La notion d'interface

Dans le cas d'une procédure externe, la compilation séparée de la procédure appelée et de la procédure appelante ne permet pas au compilateur de connaître, sans information complémentaire, le type des arguments échangés et de vérifier la cohérence entre les arguments effectifs lors de l'appel et les arguments muets déclarés. Il est cependant possible d'assurer une visibilité de ces déclarations d'arguments à l'aide d'une interface explicite (explicit interface) insérée dans la ou les procédures appelantes. Les déclarations des arguments sont ainsi communiquées aux appelants à l'intérieur d'un bloc d'interface, délimité par les instructions *INTERFACE* et *END INTERFACE* et inséré après les déclarations de variables de l'appelant.

Exemple 28: test28

```
      program calcul_externe
c
      implicit none
c
      real :: b1, h1
c
      interface
        function surf(base, hauteur)
c
        implicit none
        real :: surf                        ! déclaration du résultat
        real :: base, hauteur   ! déclaration des arguments muets
c
       end function surf
      end interface
c
      b1 = 30.
      h1 = 2.
      Print *, ' Surface= ', surf(b1, h1)
c
      end program calcul_externe
c ------------------------------------------------
         real function surf(base, hauteur)
c
         implicit none
         real :: base, hauteur     ! déclaration des arguments d'entrée
c
         surf = base * hauteur / 2.     ! affectation du résultat
c
         end function surf
```

e) Les modules

Les modules sont des entités compilables séparément (mais non exécutables) permettant d'héberger des éléments de code ou de données qui ont vocation à être utilisés par plusieurs procédures sans nécessité de dupliquer l'information. Un module est délimité par les instructions :
`MODULE 'nom_de_module' et  END MODULE 'nom_de_module'.`
Le module peut comporter:
– des déclarations de variables, susceptibles d'être partagées entre plusieurs unités de programme,
– un bloc d'interface,
–des procédures dites de module (module procedure) introduites par l'instruction *CONTAINS*.
L'instruction `USE 'nom_de_module'`, placée avant toutes les déclarations, assure à l'unité de programme ou à la procédure où elle est placée la visibilité de toutes les entités publiques du module.

Exemple 29: test29

```
      module variables
c
      implicit none
c
      real :: b1,h1,b2,h2
c
      end module variables
c ---------------------------
      program mod_exemple
c
      implicit none
c
      use variables        ! accès au module
      real :: surf, somme
c
      b1=20
      h1=10
c
      print *, ' Surface = ', surf(b1, h1)
c
      b2=20
      h2=10
c
      print *, ' Surface = ', surf(b2, h2)
c
      somme= surf(b1, h1)+ surf(b2, h2)
c
      print*,' somme =',somme
c
      end program mod_exemple
c     !----------------------------------- !
      real function surf(base, hauteur)
c
      implicit none
c
      real :: base, hauteur
      surf = base * hauteur / 2.     ! affectation du résultat
c
      end function surf
```

11ème Séance

8. Initiation aux tableaux

Un tableau (array) est un ensemble ''rectangulaire'' d'éléments de même type (plus précisément de même variante de type), repérés au moyen d'indices entiers. Les éléments d'un tableau sont rangés selon un ou plusieurs ''axes'' appelés dimensions du tableau.

Les tableaux en fortran se caractérises par :

– rang (*rank*) d'un tableau : nombre de ses dimensions.

– étendue (*extent*) d'un tableau selon une de ses dimensions : nombre d'éléments dans cette dimension.

– bornes (*bounds*) d'un tableau selon une de ses dimensions : limites inférieure et supérieure des indices dans cette dimension. La borne inférieure par défaut vaut 1.

– profil (*shape*) d'un tableau : vecteur dont les composantes sont les étendues du tableau selon ses dimensions ; son étendue (sa taille) est le rang du tableau.

– taille (*size*) d'un tableau : nombre total des éléments qui le constituent.

– deux tableaux sont dits conformant (*conformable*) s'ils ont le même profil.

La déclaration d'un tableau s'effectue grâce à l'attribut *DIMENSION* qui indique le profil du tableau, ou éventuellement les bornes, séparées par le symbole (:) .

– MINLOC et MAXLOC fournissent le vecteur des indices du premier élément respectivement minimum ou maximum du tableau.

– SUM et PRODUCT calculent respectivement la somme et le produit des éléments du tableau.

– MAXVAL et MINVAL fournissent respectivement le maximum et le minimum d'un tableau.

– DOT_PRODUCT donne le produit scalaire de deux vecteurs :

– si u et v sont des vecteurs de réels (ou d'entiers), DOT_PRODUCT(u, v) = SUM(u*v).

– si u et v sont des vecteurs de complexes, DOT_PRODUCT(u, v) = SUM(CONJG(u)*v).

– MATMUL effectue le produit matriciel de deux tableaux.

Exemple 30: test30

```
C   ****************************************
C      PROGRAMME TEST SUR LES TABLEAUX (ARRAYS)
C      ****************************************
      PROGRAM TEST_TABLEAU01
      IMPLICIT NONE
C
      REAL,DIMENSION(5)   :: V1=[1,2,3,1,3]
      REAL,DIMENSION(5)   :: V2=[1,-2,3,1,0]
C   ************************************************************
      PRINT*,'SIZE=',SIZE(V2),'      SHAPE=',SHAPE(V2)
      PRINT*,LBOUND(V1),'///',UBOUND(V1)
C
      PRINT*,MINLOC(V1),"//",MAXLOC(V1),"//",MINLOC(V2),"//",MAXLOC(V2)
      PRINT*,DOT_PRODUCT(V1(:),V2(:)),'///',SUM(V1),"///",SUM(V2)
C
      PRINT*,PRODUCT(V1),'///',PRODUCT(V2)
C
      END PROGRAM TEST_TABLEAU01
```

Exemple 31: test31

```
C    ************************************
C       PROGRAMME TEST02 SUR LES TABLEAUX
C    ************************************
      PROGRAM TEST_TABLEAU02
      IMPLICIT NONE
```

```
C
      REAL,DIMENSION(3,4):: M1
      REAL,DIMENSION(4,3):: M2
      REAL,DIMENSION(3,3):: M
C     ************************************************************
      M1(:,1)=[-1,0,1]    !     [-1,-2,-1, 1]
      M1(:,2)=[-2,0,2]    ! M1=[ 0, 0, 1, 0]
      M1(:,3)=[-1,1,1]    !     [ 1, 2, 1,-1]
      M1(:,4)=[1,0,-1]
C
      M2(:,1)=[-1,0,1,2]   !     [-1,-2,-1]
      M2(:,2)=[-2,0,2,1]   ! M2=[ 0, 0, 1]
      M2(:,3)=[-1,1,1,0]   !     [ 1, 2, 1]
                           !     [ 2, 1, 0]
      M=MATMUL(M1(:,:),M2(:,:))
C
      PRINT*,'SIZE(M1)=',SIZE(M1),'----ET-----','SIZE(M2)=',SIZE(M2)
C
      PRINT*,'-------------------------------------'
      WRITE(*,*)'SHAPE(M1)=',SHAPE(M1),'--ET--','SHAPE(M2)=',SHAPE(M2)
C
      PRINT*,'-------------------------------------'
      WRITE(*,'((A),12(2X,F7.2),10X,(A),10X,((A),12(2X,F7.2)))')
     &'M1(3,4)=',M1(:,:),'---------ET-------------','M2(4,3)=',M2(:,:)
C
      PRINT* ,'-------------------------------------'
      WRITE(*,'((A),9(1X,ES10.2))')'M=',M
C
      PRINT*,'-------------------------------------'
      WRITE(*,'((A),12(2X,F7.2))')'TRANSPOSE(M1)=',TRANSPOSE(M1)
C
      WRITE(*,'((A),12(2X,F7.2))') 'SUM(M1,DIM=1)=',SUM(M1,DIM=1)
      WRITE(*,*)'DOT_PRODUCT(M1(:,1),M1(:,2))='
     &,DOT_PRODUCT(M1(:,1),M1(:,2))
C
      END PROGRAM TEST_TABLEAU02
```

Références

1. Introduction au Fortran 90 /95 /2003, Jaques Lefrère, Université Paris VI (2013).
2. A first course in scientific computing, Rubin H. Landau, Princeton University Press (2005).
3. Programming in Fortran 95, Rachael Padman,University of Cambridge (2007).
4. Introduction à Fortran 90, Nicolas Kielbasiewicz (2008).
5. A Physics Fortran Tutorial, Richard Kass (2004).