Langage Fortran
(F95-1)

Anne Fouilloux
Patrick Corde

anne.fouilloux@ecmwf.int

Patrick.Corde@idris.fr

28 septembre 2012

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

Table des matières I

1. Introduction
   - Historique
   - bibliographie
   - documentation
2. Généralités
   - Bases de numération
   - Représentation des données
     - Représentation des entiers
     - Représentation des réels
     - Représentation des complexes
     - Représentation des logiques
     - Représentation des caractères
   - Jeu de caractères
   - Notion d'unité de programme
   - Éléments syntaxiques
     - Format libre
     - Commentaires
3. Déclarations
   - Identificateurs
   - Différents types
   - Syntaxe
   - Le type CHARACTER
   - Instruction IMPLICIT NONE
   - Constantes littérales
     - Constantes entières

Table des matières II

- Constantes réelles simple précision
- Constantes réelles double précision
- Constantes complexes
- Constantes chaînes de caractères

Initialisation
- L'instruction DATA
- Le symbole " = "

Constantes symboliques

Instruction EQUIVALENCE

4 Opérateurs et expressions

Opérateurs arithmétiques
- Les opérateurs
- Conversion implicite

Opérateurs relationnels

Opérateurs logiques
- Les tables de vérité

Opérateur de concaténation

Opérateur d'affectation
- syntaxe générale
- Règles de typage

Priorité des Opérateurs

5 Structures de contrôle

Les tests
- Le bloc IF
- Le bloc SELECT-CASE

Table des matières III

Les itérations
- L'instruction GOTO
- Les bloucles DO

6 Tableaux

Déclaration

Définitions (rang, profil, étendue)

Initialisation
- Le symbole "="
- Le constructeur de vecteurs
- L'instruction DATA

Manipulation de tableaux
- Expressions de type tableau
- Sections de tableaux

7 Entrées-Sorties

Introduction

Accès séquentiel
- Fichier binaire séquentiel
- Fichier texte séquentiel

Accès direct
- Fichier binaire à accès direct
- Fichier texte à accès direct

Fichier temporaire

Destruction d'un fichier

Fichier interne

Instructions de positionnement

Table des matières IV

Instruction INQUIRE
Formats BOZ
Unités standards
Instruction FORMAT
Mots clés END=, ERR=

8 Procédures
Arguments
Subroutines
Fonctions
Arguments de type chaîne de caractères
Arguments de type tableau
Arguments de type procédure
Procédures internes
Durée de vie et visibilité des identificateurs
Procédures intrinsèques

9 Common
L'instruction COMMON
Common blanc
Common étiqueté
Initialisation : BLOCK DATA
Instruction SAVE et COMMON
Règles et restrictions

10 Include
La directive INCLUDE

1 Introduction
Historique
bibliographie
documentation

2 Généralités

3 Déclarations

4 Opérateurs et expressions

5 Structures de contrôle

6 Tableaux

7 Entrées-Sorties

8 Procédures

9 Common

10 Include

- `Code machine` (notation numérique en octal) ;
- `Assembleurs` de codes mnémoniques ;
- 1954 : projet création du premier `langage symbolique` FORTRAN par John Backus d'`IBM` (*Mathematical* ***FOR****mula* ***TRAN****slating System*) :
  - Efficacité du code généré (performance) ;
  - Langage quasi naturel pour scientifiques (productivité, maintenance, lisibilité).
- 1957 : Livraison des premiers compilateurs ;
- 1958 : **Fortran II** (`IBM`) ⇒sous-programmes compilables de façon indépendante.
- Généralisation aux autres constructeurs mais :
  - divergences des extensions ⇒nécessité de **normalisation** ;
  - `ASA` *American Standards Association* (`ANSI` *American Nat. Standards Institute*). Comité chargé du développement d'une norme Fortran.
- 1966 : **Fortran IV** (Fortran 66) ;
- Évolution par extensions divergentes. . .
- 1977 : **Fortran V** (Fortran 77).
  quasi compatible :
  aucune itération des boucles *nulles* (`DO I=1,0`)
  - **Nouveautés principales** :
    - type caractère ;
    - `IF-THEN-ELSE` ;
    - `E/S` accès direct et `OPEN`.

- Travail des comités `X3J3/ANSI` et `WG5/ISO` pour moderniser Fortran 77 :
  - Standardisation : inclusion d'extensions ;
  - Développement : nouveaux concepts déjà exploités par langages plus récents `APL, Algol, PASCAL, Ada` ; . . .
  - Performances en calcul scientifique ;
  - Totalement compatible avec Fortran 77.
- 1991/1992 : Norme **Fortran 90** (`ISO` et `ANSI`) ;
- 1994 : Premiers compilateurs Fortran 90 `Cray` et `IBM` ;
- 1997 : Norme **Fortran 95** (`ISO` et `ANSI`) ;
- 1999 : Premiers compilateurs Fortran 95 sur `Cray T3E` puis `IBM RS/6000` ;
- septembre 2004 : Norme **Fortran 2003** (`ISO` et `ANSI`) ;
- octobre 2010 : Norme **Fortran 2008** (`ISO` et `ANSI`).

- Adams, Brainerd, Hendrickson, Maine, Martin, Smith, *The Fortran 2003 Handbook*, Springer, 2009, (712 pages), ISBN 978-1-84628-378-9 ;
- Adams, Brainerd, Martin, Smith et Wagener, *Fortran 95 Handbook*, MIT Press, 1997, (711 pages), ISBN 0-262-51096-0 ;
- Brainerd, Goldberg, Adams, *Programmer's guide to Fortran 90*, 3e édit. Unicomp, 1996, (408 pages), ISBN 0-07-000248-7 ;
- Chamberland Luc, *Fortran 90 : A Reference Guide*, Prentice Hall, ISBN 0-13-397332-8 ;
- Delannoy Claude, *Programmer en Fortran 90 – Guide complet*, Eyrolles, 1997, (413 pages), ISBN 2-212-08982-1 ;
- Dubesset M., Vignes J., *Les spécificités du Fortran 90*, Éditions Technip, 1993, (400 pages), ISBN 2-7108-0652-5 ;
- Ellis, Phillips, Lahey, *Fortran 90 Programming*, Addisson-Wesley, 1994, (825 pages), ISBN 0-201-54446-6 ;
- Hahn B.D., *Fortran 90 for the Scientist & Engineers*, Edward Arnold, London, 1994, (360 pages), ISBN 0-340-60034-9 ;
- Kerrigan James F., *Migrating to Fortran 90*, O'Reilly & Associates Inc., 1994, (389 pages), ISBN 1-56592-049-X ;

- Lignelet P., *Fortran 90 : approche par la pratique*, Éditions Studio Image (série informatique), 1993, ISBN 2-909615-01-4 ;
- Lignelet P., *Manuel complet du langage Fortran 90 et Fortran 95*, calcul intensif et génie logiciel, Col. Mesures physiques,Masson, 1996, (320 pages), ISBN 2-225-85229-4 ;
- Lignelet P., *Structures de données et leurs algorithmes avec Fortran 90 et Fortran 95*, Masson, 1996, (360 pages), ISBN 2-225-85373-8 ;
- Morgan and Schoenfelder, *Programming in Fortran 90*, Alfred Waller Ltd., 1993, ISBN 1-872474-06-3 ;
- Metcalf M., Reid J.,
  - Fortran 90 explained, Science Publications, Oxford, 1994, (294 pages), ISBN 0-19-853772-7, Traduction française par Pichon B. et Caillat M., Fortran 90 : les concepts fondamentaux, Éditions AFNOR, 1993, ISBN 2-12-486513-7 ;
  - Fortran 90/95 explained, Oxford University Press, 1996, (345 pages), ISBN 0-19-851888-9 ;
  - Fortran 95/2003 explained, Oxford University Press, 2004, (416 pages), ISBN 0-19-852693-8 ;
- Olagnon Michel, Traitement de données numériques avec Fortran 90, Masson, 1996, (364 pages), ISBN 2-225-85259-6 ;

- Redwine Cooper, *Upgrading to Fortran 90*, Springer, 1995, ISBN 0-387-97995-6 ;
- *International Standard* ISO/IEC 1539-1:1997(E) *Information technology - Progr. languages - Fortran - Part1 : Base language*. Disponible auprès de l'AFNOR.

- Documentation IBM/SP6 :
  - XL Fortran Language Reference
  - XL Fortran USER's Guide
  - ESSL - Engineering and Scientific Subroutine Library Guide
- Disponibles sur le serveur Web IDRIS à l'adresse :
  http://www.idris.fr/su/Scalaire/vargas/doc-ibm.html
- Documentation IDRIS IBM/SP6 :
  - descriptif matériel et logiciel,
  - supports de cours,
  - FAQ,
- Disponibles sur le serveur Web IDRIS à l'adresse :
  http://www.idris.fr/su/Scalaire/vargas
- Documentation générale
  - Supports de cours Fortran 95 IDRIS :
    http://www.idris.fr/data/cours/lang/fortran/choix_doc.html
  - Manuel "Fortran 77 pour débutants" (en anglais) :
    http://www.idris.fr/data/cours/lang/fortran/choix_doc.html
  - *Fortran Market Index* :
    http://www.swcp.com/~walt/
  - État d'avancement de l'intégration de la norme Fortran 2003 :
    http://fortranwiki.org/fortran/show/Fortran+2003+status
  - État d'avancement de l'intégration de la norme Fortran 2008 :
    http://fortranwiki.org/fortran/show/Fortran+2008+status

1 Introduction

2 Généralités
- Bases de numération
- Représentation des données
  - Représentation des entiers
  - Représentation des réels
  - Représentation des complexes
  - Représentation des logiques
  - Représentation des caractères
- Jeu de caractères
- Notion d'unité de programme
- Éléments syntaxiques
  - Format libre
  - Commentaires

3 Déclarations

4 Opérateurs et expressions

5 Structures de contrôle

6 Tableaux

7 Entrées-Sorties

8 Procédures

9 Common

10 Include

Soit un nombre `n` dont l'écriture en base `b` est de la forme :

$$(u_p u_{p-1} \ldots u_1 u_0)_b$$

avec :

$$\forall i \in \{0, 1, ..., p\} \quad 0 \leq u_i < b$$

La valeur du nombre `n` en base 10 est :

$$n_{10} = \sum_{i=0}^{p} u_i b^i$$

Les ordinateurs ne savent calculer qu'en base 2, de ce fait les données stockées dans la mémoire le sont sous la forme d'une suite de chiffres binaires 0 et 1 appelés **bits** abréviation de ***bi**nary dig**its***. Un ensemble de 8 bits s'appelle un **octet**.
L'écriture des données en base 2 se révèle fastidieuse. Par commodité, on adopte plutôt la base 8 (base octale) ou la base 16 (hexadécimale) pour les définir.

L'écriture d'un nombre en octal s'effectuera à l'aide des chiffres de 0 à 7.
L'écriture d'un nombre en hexadécimal s'effectuera à l'aide des chiffres de 0 à 9 auxquels on ajoute les lettres de `a` à `f`.
Supposons que l'on dispose de l'écriture d'un nombre en base 2. Sa conversion en octal peut être faite en découpant le motif binaire par tranches de 3 bits en partant de la droite, puis en convertissant en base 10 chaque groupe obtenu.
Sa conversion en hexadécimal pourra s'effectuer de la même manière à l'aide d'un découpage par tranches de 4 bits.

Exemple

1001110101_2 = $1 * 2^0$ + $1*2^2$ + $1*2^4$ + $1*2^5$ + $1*2^6$ + $1*2^9$
= 629_{10}
1001110101_2 = $1|001|110|101_2 = 1165_8$
1001110101_2 = $10|0111|0101_2 = 275_{16}$

Représentation des entiers

Dans la mémoire de l'ordinateur, les données numériques sont représentées à l'aide d'un motif binaire de longueur 32, 64 voire 128 bits.
La représentation en machine d'un nombre entier positif correspond à son écriture en base 2. Pour l'obtenir, il suffit de procéder à des divisions successives par 2.
Les nombres entiers négatifs sont représentés en **complément vrai** ou **complément à 2** qui consiste, à partir du motif binaire du nombre positif, à inverser tous les bits puis d'ajouter 1.
De ce fait, sur `n` bits, les nombres représentables sont les suivants :

$$-2^{n-1} \leq i \leq 2^{n-1} - 1$$

Exemple

$+5_{10}$ = $00000000000000000000000000000101_2$
-5_{10} = $11111111111111111111111111111010_2$ + 1
-5_{10} = $11111111111111111111111111111011_2$
-5_{10} = $FFFFFFFB_{16}$

Représentation des réels

Un nombre réel ou *flottant* est caractérisé par :

1. son signe ;
2. son exposant ou caractéristique ;
3. sa mantisse.

Son mode de représentation est un motif binaire respectant la norme IEEE.

Représentation d'un nombre réel sur 32 bits

Ce type de réel, appelé réel simple précision, admet un motif binaire de la forme :

seeeeeeeem——m

avec :

- **s** : bit de signe ;
- **e** : exposant sur 8 bits à excédent 127 ;
- **m** : mantisse sur 23 bits.

Le nombre représenté correspond à $\Rightarrow$ $r = s1.m \times 2^{e-127}$

Ce type de représentation permet de représenter les nombres :

$$1.2\times10^{-38} \leq |r| \leq 3.4\times10^{+38}$$

avec 6 chiffres significatifs.

Représentation d'un nombre réel sur 64 bits

Ce type de réel, appelé réel double précision, admet un motif binaire de la forme :

seeeeeeeeeeem——m

avec :

- **s** : bit de signe,
- **e** : exposant sur 11 bits à excédent 1023,
- **m** : mantisse sur 52 bits.

Le nombre représenté correspond à $\Rightarrow$ $r = s1.m \times 2^{e-1023}$
Ce type de représentation permet de représenter les nombres :

$$2.2\times10^{-308} \leq |r| \leq 1.8\times10^{+308}$$

avec 15 chiffres significatifs.

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

On peut écrire ce réel sous la forme suivante :

$$10,4 = \frac{104}{10} = \frac{52}{5} = \frac{110100_2}{101_2}$$

Voici ce que donne la division binaire :

110100	101
00110	1010,0110011001100110011...
00001000	
000000110	
000000001	

L'écriture en binaire du nombre réel 10,4 est finalement :

$$10,4_{10} = 1010,0110011001100110011..._2 = 1,0100110011001100110011...\times2^3$$
$$= 1,0100110011001100110011...\times2^{130-127}$$

On en déduit alors le motif binaire suivant :

$$\underbrace{01000001}_{\text{exposant 130}}\underbrace{00100110011001100110011}_{\text{mantisse (23bits)}} = 41266666_{16}$$

Représentation du réel 10,4 sur 64 bits

Pour le codage de ce réel sur 64 bits il suffit de prolonger son écriture binaire d'autant.
On obtient :

$$\begin{aligned} 10,4_{10} &= 1010,0110011001100110011001100110011001100110011001100\ldots \\ &= 1,0100110011001100110011001100110011001100110011001100\ldots \times 2^{3} \\ &= 1,0100110011001100110011001100110011001100110011001100\ldots \times 2^{1026-1023} \end{aligned}$$

On en déduit alors sa représentation interne :

$$10,4 \blacktriangleright\ 0\ \underbrace{10000000010}_{\text{exposant 1026}}\ \underbrace{0100110011001100110011001100110011001100110011001100}_{\textit{mantisse}\ (52\textit{bits})}$$

$$\blacktriangleright\quad 4024CCCCCCCCCCCC_{16}$$

Remarque :

- En vérité, le motif en mémoire est 4024CCCCCCCCCCCD car lors de la troncature le 1er bit perdu à pour valeur 1.
- Sur 32 bits ce n'était pas le cas car le bit perdu à pour valeur 0.

Représentation des complexes

Un nombre complexe est une paire de nombres réels, simple ou double précision, correspondant aux parties réelle et imaginaire.

Soit le nombre complexe : 1.5-1.5i
Sa représentation en simple précision nécessite 2 réels sur 32 bits :

Partie réelle	0 01111111 1000...000$_2$	= 3FC00000$_{16}$	= +1.5$_{10}$
Partie imaginaire	1 01111111 1000...000$_2$	= BFC00000$_{16}$	= -1.5$_{10}$

Représentation des logiques

Un logique est une entité qui peut prendre comme valeur :

- `.TRUE.`
- `.FALSE.`

Il est représenté en général sur 32 bits (4 octets). Il peut exister des variantes codées sur 1, 2 voire 8 octets. Tous les bits sont positionnés à 0 sauf le bit le plus à droite qui pour la valeur `.TRUE.` est positionné à 1.

Représentation des caractères

Un caractère est codé sur 1 octet. Sa représentation interne respecte un codage appelé codage `ASCII`.
Il existe 128 caractères différents dont les représentations sont indiquées dans une table dite table `ASCII`.
Dans cette table les caractères numériques ainsi que les caractères alphabétiques (majuscules et minuscules) sont rangés consécutivement et en ordre croissant.
On appelle chaîne de caractères une suite de caractères rangés de façon consécutive en mémoire.

Table 1: *table des codes ASCII des caractères*

Caract.	déc.	hex	oct.	Caract.	déc.	hex	oct.
C-@ (NUL)	0	0x00	000	espace	32	0x20	040
C-a (SOH)	1	0x01	001	!	33	0x21	041
C-b (STX)	2	0x02	002	"	34	0x22	042
C-c (ETX)	3	0x03	003	#	35	0x23	043
C-d (EOT)	4	0x04	004	$	36	0x24	044
C-e (ENQ)	5	0x05	005	%	37	0x25	045
C-f (ACK)	6	0x06	006	&	38	0x26	046
C-g (BEL)	7	0x07	007	’	39	0x27	047
C-h (BS)	8	0x08	010	(	40	0x28	050
C-i (HT)	9	0x09	011	)	41	0x29	051
C-j (LF)	10	0x0a	012	*	42	0x2a	052
C-k (VT)	11	0x0b	013	+	43	0x2b	053
C-l (FF)	12	0x0c	014	,	44	0x2c	054
C-m (CR)	13	0x0d	015	-	45	0x2d	055
C-n (SO)	14	0x0e	016	.	46	0x2e	056
C-o (SI)	15	0x0f	017	/	47	0x2f	057
C-p (DLE)	16	0x10	020	0	48	0x30	060
C-q (DC1)	17	0x11	021	1	49	0x31	061
C-r (DC2)	18	0x12	022	2	50	0x32	062
C-s (DC3)	19	0x13	023	3	51	0x33	063
C-t (DC4)	20	0x14	024	4	52	0x34	064
C-u (NAK)	21	0x15	025	5	53	0x35	065
C-v (SYN)	22	0x16	026	6	54	0x36	066
C-w (ETB)	23	0x17	027	7	55	0x37	067
C-x (CAN)	24	0x18	030	8	56	0x38	070
C-y (EM)	25	0x19	031	9	57	0x39	071
C-z (SUB)	26	0x1a	032	:	58	0x3a	072
C-[(ESC)	27	0x1b	033	;	59	0x3b	073
C-\ (FS)	28	0x1c	034	<	60	0x3c	074
C-] (GS)	29	0x1d	035	=	61	0x3d	075
C-$ (RS)	30	0x1e	036	>	62	0x3e	076
C-_ (US)	31	0x1f	037	?	63	0x3f	077

Table 2: *table des codes ASCII des caractères (suite)*

Caract.	déc.	hex	oct.	Caract.	déc.	hex	oct.
@	64	0x40	100	‘	96	0x60	140
A	65	0x41	101	a	97	0x61	141
B	66	0x42	102	b	98	0x62	142
C	67	0x43	103	c	99	0x63	143
D	68	0x44	104	d	100	0x64	144
E	69	0x45	105	e	101	0x65	145
F	70	0x46	106	f	102	0x66	146
G	71	0x47	107	g	103	0x67	147
H	72	0x48	110	h	104	0x68	150
I	73	0x49	111	i	105	0x69	151
J	74	0x4a	112	j	106	0x6a	152
K	75	0x4b	113	k	107	0x6b	153
L	76	0x4c	114	l	108	0x6c	154
M	77	0x4d	115	m	109	0x6d	155
N	78	0x4e	116	n	110	0x6e	156
O	79	0x4f	117	o	111	0x6f	157
P	80	0x50	120	p	112	0x70	160
Q	81	0x51	121	q	113	0x71	161
R	82	0x52	122	r	114	0x72	162
S	83	0x53	123	s	115	0x73	163
T	84	0x54	124	t	116	0x74	164
U	85	0x55	125	u	117	0x75	165
V	86	0x56	126	v	118	0x76	166
W	87	0x57	127	w	119	0x77	167
X	88	0x58	130	x	120	0x78	170
Y	89	0x59	131	y	121	0x79	171
Z	90	0x5a	132	z	122	0x7a	172
[	91	0x5b	133	{	123	0x7b	173
\	92	0x5c	134	|	124	0x7c	174
]	93	0x5d	135	}	125	0x7d	175
^	94	0x5e	136	∼	126	0x7e	176
_	95	0x5f	137	C-?	127	0x7f	177

Jeu de caractères

- 26 lettres de l'alphabet ;
- chiffres 0 à 9 ;
- caractères spéciaux :

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

- le caractère espace ;
- le caractère _ (*underscore*).

Remarque :

les caractères minuscules sont convertis en majuscules par le compilateur

Un programme source `Fortran` est composé de parties indépendantes appelées **unités de programme** (*scoping unit*).
Chaque partie est compilée de façon indépendante. Chacune admet son propre environnement. Il sera cependant possible que ces parties communiquent entre elles.
Les différentes unités de programme sont :

1. le programme principal ;
2. les sous-programmes :
   - de type *subroutine* ;
   - de type *function*.
3. les modules ;
4. les *block data*.

Chaque unité comprend une partie déclarative (déclaration des variables locales, ...) suivie d'une partie comportant des instructions exécutables parmi lesquelles peut apparaître l'instruction `STOP` qui provoque l'interruption du programme.

Éléments syntaxiques

Dans le mode « **format libre** »les lignes peuvent être de longueur quelconque à concurrence de 132 caractères.
Il est également possible de coder plusieurs instructions sur une même ligne en les séparant avec le caractère « **;** ».

Exemple

```
print *, " Entrez une valeur :"; read *,n
```

Une instruction peut être codée sur plusieurs lignes : on utilisera alors le caractère « **&** ».

Exemple

```
print *, "Montant  HT :", montant_ht ,  &
         "        TVA :", tva        ,  &
         "Montant TTC :", montant_ttc
```

Lors de la coupure d'une chaîne de caractères la suite de la chaîne doit obligatoirement être précédée du caractère « **&** ».

Exemple

```
print *, "Entrez un nombre entier &
         &compris entre 100 & 199"
```

Remarque : il existe aussi le « **Format fixe** », considéré maintenant comme obsolète dont la structure d'une ligne est :

1. zone étiquette (colonnes 1 à 5) ;
2. zone instruction (colonnes 7 à 72) ;
3. colonne suite (colonne 6)

Le caractère « **!** »rencontré sur une ligne indique que ce qui suit est un commentaire. On peut évidemment écrire une ligne complète de commentaires : il suffit pour cela que le 1er caractère non blanc soit le caractère « **!** ».

Exemple

```
if (n < 100 .or. n > 199) ! Test cas d'erreur
   . . . .
! On lit l'exposant
read *,x
! On lit la base
read *,y
if (y <= 0) then    ! Test cas d'erreur
    print *," La base doit être un nombre > 0"
else
    z = y**x        ! On calcule la puissance
end if
```

Remarque :

En format fixe, les lignes qui commencent par **C**, c, * ou **!** en colonne 1 sont des commentaires.

1 Introduction

2 Généralités

3 Déclarations
- Identificateurs
- Différents types
- Syntaxe
- Le type CHARACTER
- Instruction IMPLICIT NONE
- Constantes littérales
  - Constantes entières
  - Constantes réelles simple précision
  - Constantes réelles double précision
  - Constantes complexes
  - Constantes chaînes de caractères
- Initialisation
  - L'instruction DATA
  - Le symbole " = "
- Constantes symboliques
- Instruction EQUIVALENCE

4 Opérateurs et expressions

5 Structures de contrôle

6 Tableaux

7 Entrées-Sorties

8 Procédures

9 Common

10 Include

Un identificateur permet de donner un nom à :

- une **variable** ;
- une **constante** ;
- une **procédure**.

Il est défini par :

- une suite de caractères **alphanumériques** (lettres non accentuées, chiffres, *underscore*) ;
- le premier caractère doit être une **lettre** ;
- la longueur est limitée à **31 caractères** ;
- on ne distingue pas les lettres **majuscules** des **minuscules**.

Exemple

```
compteur
Compteur
fin_de_fichier
montant_annee_1993
```

Le type d'une variable détermine :

- le nombre d'octets à réserver en mémoire ;
- un mode de représentation interne ;
- l'ensemble des valeurs admissibles ;
- l'ensemble des opérateurs qui peuvent lui être appliqués.

Types prédéfinis

Mot clé.	Type
`INTEGER`	: entier
`CHARACTER`	: caractère
`LOGICAL`	: deux valeurs `.TRUE.`, `.FALSE.`
`REAL`	: réel simple précision
`DOUBLE PRECISION`	: réel double précision
`COMPLEX`	: complexe simple précision

Remarque :

la précision d'un réel simple est de 7 chiffres décimaux significatifs alors que celle d'un double est de 15.

Attributs

Chaque type peut être surchargé d'attributs dont voici un extrait :

Attributs

Attribut	Signification
`PARAMETER`	: constante symbolique
`DIMENSION`	: taille d'un tableau
`SAVE`	: objet statique
`EXTERNAL`	: procédure externe
`INTRINSIC`	: procédure intrinsèque

Syntaxe d'une déclaration :

```
type[, liste_attributs ::] liste_identificateurs
```

Exemple

```
PROGRAM declaration
  INTEGER, SAVE ::  compteur
  INTEGER       ::  temperature
  LOGICAL       ::  arret_boucle
                ...
END PROGRAM declaration
                ...
PROGRAM declaration
  INTEGER     indice_boucle
  SAVE        indice_boucle
                ...
END PROGRAM declaration
```

Pour déclarer une chaîne de caractères on précise de plus sa longueur. Si elle n'est pas indiquée elle est égale à 1 :

```
CHARACTER(len=n) ch_car
CHARACTER c
```

L'ancienne syntaxe suivante est toujours disponible mais déclarée obsolète :

```
CHARACTER*n ch_car
```

Exemple

```
PROGRAM declaration
  CHARACTER(LEN=11)  chaine1
  CHARACTER*11       chaine2
                ...
END PROGRAM declaration
```

Par défaut, les variables dont l'identificateur commence par les caractères `I` à `N` sont de type `INTEGER`.
Toutes les autres sont de type `REAL`.
L'instruction `IMPLICIT NONE` change cette règle car elle impose à l'utilisateur la **déclaration** de chaque variable.
Cette instruction est vivement **recommandée** car elle permet la détection d'un certain nombre d'erreurs à la compilation.

- `IMPLICIT NONE` se place avant les déclarations des variables,
- L'instruction ne s'applique qu'à l'unité de programme qui la contient.

- une suite de chiffres en base 10,
- une suite de chiffres en base 2 encadrée par des quotes, le tout précédé du caractère **B**,
- une suite de chiffres en base 8 encadrée par des quotes, le tout précédé du caractère **O**,
- une suite de chiffres en base 16 encadrée par des quotes, le tout précédé du caractère **Z**.

Une valeur négative sera précédée du signe `-`.

Exemple

```
1
123
-28
B'11011011100'
O'3334'
Z'6DC'
```

Remarque :

Les constantes écrites en base 2, 8 ou 16 s'appellent des constantes `BOZ`. Elles ne peuvent figurer que dans les instructions d'initialisation de type `DATA`.

Une constante de type REAL doit obligatoirement comporter :

- soit le point décimal, même s'il n'y a pas de chiffres après la virgule ;
- soit le caractère E pour la notation en virgule flottante.

Pour les nombres écrits 0.xxxxx, on peut omettre le 0 avant le point décimal.

Exemple

```
0.
1.0
1.
3.1415
31415E-4
1.6E-19
1E12
.001
-36.
```

Une constante double precision doit obligatoirement être écrite en virgule flottante, le E étant remplacé par un D.

Exemple

```
0D0
0.D0
1.D0
1d0
3.1415d0
31415d-4
1.6D-19
1d12
-36.d0
```

Une constante de type COMPLEX est obtenue en combinant deux constantes réelles entre parenthèses séparées par une virgule : 2.5+i s'écrira (2.5,1.)

Exemple

```
(0.,0.)
(1.,-1.)
(1.34e-7, 4.89e-8)
```

Une constante chaînes de caractères est une suite de caractères encadrée par le délimiteur « ' » ou bien « " ».
Si parmi la suite des caractères figure le caractère délimiteur, il devra être doublé.

Exemple

```
'La somme des n premiers entiers est : '
'l''étendue désirée est : '
"l'étendue désirée est : "
```

À partir d'une variable chaîne de caractères on peut extraire une suite de caractères contigus. Pour cela on spécifie le nom de la variable suivi entre parenthèses d'un couple d'entiers « n:m » indiquant les rangs de début et de fin d'extraction.

Exemple

```
CHARACTER(LEN=10) :: ch

ch = "Bonjour"; ch(4:7) = "soir"
```

Une initialisation pourra s'effectuer au moyen de l'instruction suivante :

```
DATA liste1/init1/[, ..., listei/initi/, ...]
```

- `listei` fait référence à une liste de variables à initialiser,
- `initi` indique les valeurs d'initialisation,
- le type des valeurs d'initialisation doit respecter les règles suivantes :
  - pour un objet de type caractère ou logique, la constante d'initialisation doit être de même type,
  - pour un objet de type entier, réel ou complexe, la constante d'initialisation peut être de l'un de ces types.

Exemple

```
REAL        a, b, c
INTEGER     n, m
LOGICAL     arret
DATA        a, b, n/1.0, 2.0, 17/
DATA        c/2.6/, m/3/
DATA        arret/.FALSE./
```

Remarques :

- cette instruction peut apparaître après des instructions exécutables, mais la norme F95 a déclaré cette possibilité comme obsolète ;
- les variables initialisées par ce moyen héritent de l'attribut `SAVE` : elles sont alors permanentes (cf. chapitre `Procédures`, section `Durée de vie des identificateurs`).

Il n'est pas rare de trouver ce type d'initialisation lors de la déclaration comme dans l'exemple suivant (ce n'est pas conseillé car cela ne fait pas partie de la norme donc non portable) :

Extension

```
REAL        a/3.14/, b/2.718/
INTEGER     n/1/, m/4/
LOGICAL     arret/.false./
```

Fortran permet d'initialiser une variable lors de sa déclaration à l'aide du symbole « = ». Dans ce contexte, les caractères « :: » sont obligatoires :

```
TYPE[, attributs] :: v1=c1[, ..., vi=ci, ...]
```

où v_i est le nom de la variable à initialiser et c_i sa valeur.

Exemple

```
PROGRAM initialisation
  INTEGER   :: debut   = 100
  REAL      :: valeur  = 76.3
  LOGICAL   :: drapeau = .TRUE.
                ...
END PROGRAM initialisation
```

Note : ces variables héritent alors de l'attribut SAVE, ce qui implique que leur emplacement mémoire est permanent. Pour plus de détails, se reporter page 181 du support.

L'attribut PARAMETER permet de donner un nom symbolique à une constante littérale :

```
TYPE, PARAMETER :: n1=c1[, ..., ni=ci, ...]
```

où n_i est le nom donné à une constante et c_i sa valeur.
La notation suivante est aussi utilisable :

```
PARAMETER ( n1=c1[, ..., ni=ci, ...] )
```

Exemple

```
PROGRAM constante
  LOGICAL, PARAMETER :: VRAI=.TRUE., FAUX=.FALSE.

  DOUBLE PRECISION   :: PI, RTOD
  PARAMETER (PI=3.14159265d0, RTOD=180.d0/PI)
                ...
END PROGRAM constante
```

- L'instruction EQUIVALENCE permet à des variables de partager la même zone mémoire au sein d'une unité de programme ;
- il n'y a pas de conversion de type ;
- chaque variable garde les propriétés de son type ;
- le type CHARACTER ne peut pas être associé à d'autres types.

Syntaxe générale :

EQUIVALENCE(v_1, v_2)[,..., (v_{i-1}, v_i),...]

où les v_i sont des scalaires (variables simples ou éléments de tableaux).

Exemple

```
PROGRAM correspondance
  COMPLEX        cmplx(2)
  REAL           temp(4)
  EQUIVALENCE  (temp(1), cmplx(1))
               ...
END PROGRAM correspondance
```

Agencement en mémoire :

```
|--------cmplx(1)-------|--------cmplx(2)-------|
|-----------|-----------|-----------|-----------|
|--temp(1)--|--temp(2)--|--temp(3)--|--temp(4)--|
```

Exemple

```
PROGRAM correspondance
  CHARACTER(LEN=4)                  :: A, B
  CHARACTER(LEN=3)                  :: C(2)
  CHARACTER(LEN=10)                 :: chaine
  CHARACTER(LEN=1), DIMENSION(10) :: tab_car
  EQUIVALENCE       (A,C(1)),(B,C(2))
  EQUIVALENCE       (chaine,tab_car(1))
                ...
END PROGRAM correspondance
```

Agencement en mémoire :

-
```
| 01 | 02 | 03 | 04 | 05 | 06 | 07 |
|---------A---------|
|-----C(1)-----|-----C(2)-----|
               |---------B---------|
```
-
```
| 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 |
|-------------------- chaine ----------------------|
  |                              |
  |--> tab_car(1)                |--> tab_car(7)
```


1 Introduction

2 Généralités

3 Déclarations

4 Opérateurs et expressions
- Opérateurs arithmétiques
  - Les opérateurs
  - Conversion implicite
- Opérateurs relationnels
- Opérateurs logiques
  - Les tables de vérité
- Opérateur de concaténation
- Opérateur d'affectation
  - syntaxe générale
  - Règles de typage
- Priorité des Opérateurs

5 Structures de contrôle

6 Tableaux

7 Entrées-Sorties

8 Procédures

9 Common

10 Include

Les opérateurs

Table 3: *Opérateurs arithmétiques*

Symbole	Expression	Interprétation
+	o_1 + o_2	ajoute o_2 à o_1
+	+ o_1	égal à o_1
-	o_1 - o_2	soustrait o_2 à o_1
-	- o_1	inverse le signe de o_1
*	o_1 * o_2	multiplie o_1 par o_2
/	o_1 / o_2	o_1 divisé par o_2
**	o_1**o_2	élève o_1 à la puissance o_2

Les opérandes o_1 et o_2 peuvent être :

- une constante numérique ;
- une variable numérique, précédée ou non d'un opérateur unaire (+ ou -) ;
- une expression arithmétique entre parenthèses.

Exemple

```
3.14159
K
(A + B) * (C + D)
-1.0 / X + Y / Z ** 2
-2.0 * 3.14159 * RADIUS
```

Le type d'une expression arithmétique dépend des types de ses opérandes.
Dans le cas d'opérateurs binaires :

1. si les 2 opérandes sont du même type alors l'expression arithmétique résultante sera de ce type.
2. si les deux opérandes ne sont pas du même type alors l'expression arithmétique sera évaluée dans le type le plus fort relativement à la hiérarchie suivante :
   INTEGER < REAL < DOUBLE PRECISION < COMPLEX

Expression	Valeur	Type du résultat
99/100	0	INTEGER
7/3	2	INTEGER
(100*9)/5	180	INTEGER
(9/5)*100	100	INTEGER
99./100	0.99	REAL
99./100d0	0.99d0	DOUBLE PRECISION
(1.,2.)+1	(2.,2.)	COMPLEX

Attention

Soit l'expression `d = 1.d0+5.**0.5` avec la variable `d` déclarée en `DOUBLE PRECISION`. La sous-expression `5.**0.5` est évaluée dans le type `REAL` car les opérandes de l'opérateur `**` le sont. Le reste de l'évaluation s'effectuera ensuite dans le type `DOUBLE PRECISION`, le résultat étant finalement stocké dans la variable `d`.

Mais cette variable `d` bien que du type `DOUBLE PRECISION` hérite d'un calcul qui a commencé dans le type `REAL`, d'où une perte de précision.

Cela peut induire par la suite des comportements inattendus lors de l'évaluation d'expressions dans lesquelles figurent cette variable (problème de convergence dans des processus itératifs comme dans l'exercice 3).

En conclusion, lors de l'écriture d'expressions avec présence de constantes réelles que l'on désire évaluer en `DOUBLE PRECISION`, il est impératif d'écrire ces constantes dans ce type. Ce qui donne pour l'expression précédente :

```
d = 1.d0+5.d0**0.5d0
```

Table 4: *Opérateurs relationnels*

Opérateur	Opération
.LT. ou <	strictement plus petit
.LE. ou <=	inférieur ou égal
.EQ. ou ==	égal
.NE. ou /=	non égal
.GT. ou >	strictement plus grand
.GE. ou >=	supérieur ou égal

Ces opérateurs admettent des opérandes de type `INTEGER`, `REAL` ou `CHARACTER`. Seuls les opérateurs ==, /= peuvent s'appliquer à des expressions de type `COMPLEX`.

Exemple

```
N .GE. 0
X .LT. Y
Z /= 3.7
(B**2 - 4*A*C) .GT. 0.
```

Les opérandes des opérateurs logiques doivent être des expressions de type LOGICAL.

Table 5: *Opérateurs logiques*

Opérateur	Opération
.NOT.	négation logique
.AND.	conjonction logique
.OR.	disjonction inclusive
.EQV.	équivalence logique
.NEQV.	non-équivalence logique

Table 6: *Opérateur de négation*

l	.NOT.l
.true.	.false.
.false.	.true.

Table 7: *Autres opérateurs*

l_1	l_2	l_1.AND.l_2	l_1.OR.l_2	l_1.EQV.l_2	l_1.NEQV.l_2
.true.	.true.	.true.	.true.	.true.	.false.
.true.	.false.	.false.	.true.	.false.	.true.
.false.	.true.	.false.	.true.	.false.	.true.
.false.	.false.	.false.	.false.	.true.	.false.

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

L'opérateur de concaténation n'admet que des expressions de type CHARACTER.

Expression	Interprétation
c_1 // c_2	concatène c_1 avec c_2

Exemple

```
CHARACTER(LEN=10) :: ch

ch = "BON" // "JOUR"     ! <-- Affectation de la chaîne "BONJOUR"
ch = "BON"
ch = ch // "JOUR"        ! <-- Inopérant !!!
ch = TRIM(ch) // "JOUR"  ! <-- OK
```

Remarques :

1. lorsque la chaîne réceptrice est plus grande que celle affectée, elle est complétée à l'aide du caractère espace ;
2. reportez-vous à la page 227 pour plus d'informations concernant la procédure TRIM utilisée dans l'exemple ci-dessus.

```
variable = expression
```

où `expression` est une expression arithmétique, logique ou relationnelle.

- une valeur de type `CHARACTER` ne peut pas être affectée à une variable numérique ou *vice-versa*,
- une valeur de type `INTEGER` peut être affectée à une variable de type `REAL`,
- une valeur de type `REAL` peut également être affectée à une variable de type `INTEGER`. Mais dans ce cas, la valeur est alors tronquée en supprimant la partie fractionnaire.

En supposant dans les expressions suivantes, les variables `x` de type `REAL` et `n, m` de type `INTEGER` :

Expression	Interprétation
`x = 5`	`x = 5.0`
`n = 0.9999`	`n = 0`
`m = -1.9999`	`m = -1`

Table 8: *Ordre de priorité des opérateurs*

Opérateur	Associativité
**	D → G
* et /	G → D
+ et -	G → D
//	G → D
<, <=, ==	G → D
/=, >, >=	
.NOT.	///////
.AND.	G → D
.OR.	G → D
.EQV. et .NEQV.	G → D

En supposant dans les expressions suivantes, les variables `a, b, c, d` de type `REEL` et `e, f, g` de type `LOGICAL` :

Expression	Interprétation
`2**3**2`	`2**(3**2) = 512`
`5.+4.*9.**2`	`5.+(4.*(9.**2)) = 329.`
`e.OR.f.AND.g`	`e.OR.(f.AND.g)`
`a**b+c.GT.d.AND.e`	`(((a**b)+c).GT.d).AND.e`

1 Introduction

2 Généralités

3 Déclarations

4 Opérateurs et expressions

5 Structures de contrôle
- Les tests
  - Le bloc IF
  - Le bloc SELECT-CASE
- Les itérations
  - L'instruction GOTO
  - Les bloucles DO

6 Tableaux

7 Entrées-Sorties

8 Procédures

9 Common

10 Include

Le bloc IF

```
[nom_bloc: ] IF( exp_1 ) THEN
                   bloc_1
             [ELSE IF( exp_2 ) THEN [nom_bloc]
                   bloc_2
              ELSE IF( exp_3 ) THEN [nom_bloc]
                   bloc_3
                   ...
              ELSE [nom_bloc]
                   bloc_n]
              END IF [nom_bloc]
```

- nom_bloc une étiquette facultative : si elle est présente elle doit figurer au niveau de l'instruction END IF et peut apparaître à la suite des éventuelles instructions ELSE, ELSE IF ;
- exp_i une expression de type LOGICAL ;
- $bloc_i$ une suite d'instructions **Fortran**.

En l'absence de clause ELSE lorsque $bloc_1$ est réduit à une seule instruction, la structure IF se simplifie en :

```
IF (exp) instruction
```

Exemple

```
PROGRAM structure_if
  REAL A,B,SUM
                ...
  IF (A.LT.B) THEN
    SUM = SUM + A
    IF (SUM > 0.) PRINT *, SUM
  END IF
                ...
END PROGRAM structure_if
```

Exemple

```
PROGRAM structure_if
  REAL A,HRS
                ...
  IF (HRS.LE.40.0) THEN
    A = HRS*150.0
  ELSE IF (HRS.LE.50.) THEN
    A = (HRS-40.0)*150.0*1.5
  ELSE
    A = (HRS-50.0)*150.0*2.0
  END IF
END PROGRAM structure_if
```

L'instruction SELECT CASE permet des branchements multiples qui dépendent de la valeur d'une expression scalaire de type entier, logique ou chaîne de caractères.

```
[ nom_bloc: ] SELECT CASE(expression)
                CASE(liste) [ nom_bloc ]
                   bloc_1
                     ...
              [ CASE DEFAULT[ nom_bloc ]
                   bloc_n ]
              END SELECT[ nom_bloc ]
```

- nom_bloc est une étiquette,
- expression est une expression de type INTEGER, LOGICAL ou CHARACTER,
- liste est une liste de constantes du même type que expression,
- bloc$_i$ est une suite d'instructions **Fortran**.

Exemple

```
PROGRAM structure_case
  integer :: mois, nb_jours
  logical :: annee_bissext
            ...
  SELECT CASE(mois)
    CASE(4, 6, 9, 11)
      nb_jours = 30
    CASE(1, 3, 5, 7:8, 10, 12)
      nb_jours = 31
    CASE(2)
  !-----------------------------------
       fevrier: select case(annee_bissext)
          case(.true.)
            nb_jours = 29
          case(.false.)
            nb_jours = 28
       end select fevrier
  !-----------------------------------
    CASE DEFAULT
      print *, " Numéro de mois invalide"
  END SELECT
END PROGRAM structure_case
```

L'instruction GOTO permet d'effectuer un branchement à un endroit particulier du code :

```
GOTO étiquette
```

Cette instruction est à éviter car elle peut générer des programmes **illisibles** et **difficiles à corriger**.

Exemple

```
PROGRAM iteration_goto
   REAL diviseur, valeur, facteur
             ...
   valeur = 0. ; diviseur = 360.
69 IF (diviseur .NE. 0.) THEN
     valeur   = valeur + facteur / diviseur
     diviseur = diviseur - 10.
     GOTO 69
   END IF
             ...
END PROGRAM iteration_goto
```

Cet exemple peut être remplacé par une boucle itérative de type DO WHILE.

Il existe plusieurs types de boucles itératives qui sont toutes de la forme :

```
[ nom_bloc: ] DO [contrôle_de_boucle]
                bloc
              END DO [ nom_bloc ]
```

- nom_bloc est une étiquette,
- contrôle_de_boucle définit les conditions d'exécution et d'arrêt de la boucle,
- bloc est une suite d'instructions **Fortran**.

1re forme : DO indéxé

contrôle_de_boucle est de la forme :

variable = $expr_1$, $expr_2$ [,$expr_3$]

avec :

- variable est une variable de type INTEGER,
- $expr_1$, $expr_2$ et $expr_3$ sont des expressions arithmétiques de type INTEGER.

Le nombre d'itérations est évalué avant le démarrage de la boucle.

Exemple

```
PROGRAM iteration_do
  INTEGER i, somme, n
          ...
! affectation de n
  somme=0
  DO i=1,n,2
    somme=somme+i
  END DO
          ...
END PROGRAM iteration_do
```


2re forme : DO WHILE

contrôle_de_boucle est de la forme :

WHILE (expression)

avec expression de type scalaire logique.
Le corps de la boucle est exécuté tant que l'expression est vraie.

Remarque : pour pouvoir sortir de la boucle, il faut que expression puisse prendre la valeur .FALSE. dans le bloc.

Sommation de la série $\sum_{n\geq 1} 1/n^2$ jusqu'à ce que le terme général soit inférieur à ϵ fois la somme partielle courante :

Exemple

```
PROGRAM iteration_while
  INTEGER                          :: n
  DOUBLE PRECISION                 :: terme, somme
  DOUBLE PRECISION, PARAMETER      :: epsilon = 1.d-3
  LOGICAL                          :: fini

! Initialisation
  n=0
  somme=0.d0
  fini=.FALSE.
  DO WHILE (.not. fini)
    n=n+1
    terme = 1d0/n**2
    somme=somme + terme
    fini=(terme .LT. epsilon*somme)
  END DO
  print *,"Nombre d'itérations : ", n
  print *,"Somme = ", somme
END PROGRAM iteration_while
```

3re forme : DO

Ce sont des boucles DO sans contrôle de boucle. Pour en sortir, on utilise une instruction conditionnelle avec une instruction EXIT.
bloc est de la forme :

```
bloc_1
IF (expression) EXIT
bloc_2
```

avec :

- expression une expression de type LOGICAL,
- $bloc_i$ des séquences de code **Fortran**.

Notons que la condition IF peut être remplacée par une instruction de type SELECT CASE.

Exemple

```
PROGRAM iteration_exit
  REAL             :: valeur
  REAL             :: x, xlast
  REAL, PARAMETER  :: tolerance = 1.0e-6

  valeur = 50.

  x = 1.0          ! valeur initiale (diff. 0)

  DO
    xlast = x
    x = 0.5 * (xlast + valeur/xlast)
    IF (ABS(x-xlast)/x < tolerance) EXIT
  END DO
END PROGRAM iteration_exit
```

Instruction CYCLE

bloc$_i$ peut aussi contenir une instruction CYCLE :

```
IF (expression) CYCLE
```

CYCLE permet d'abandonner le traitement de l'itération courante et de passer à la suivante.
Là aussi, l'instruction IF peut être remplacée par une instruction de type SELECT CASE.

Exemple

```
PROGRAM iteration_cycle
  INTEGER :: annee

  DO
    READ(*,*) annee
    IF (annee .LE. 0) EXIT
! On élimine les années bissextiles.
    IF( ((annee/4*4     .EQ. annee) .AND. &
         (annee/100*100 .NE. annee)) .OR. &
         (annee/400*400 .EQ. annee) ) CYCLE
    PRINT*,"Traitement des années non-bissextiles"
      ...
  END DO
END PROGRAM iteration_cycle
```

1 Introduction

2 Généralités

3 Déclarations

4 Opérateurs et expressions

5 Structures de contrôle

6 Tableaux
- Déclaration
- Définitions (rang, profil, étendue)
- Initialisation
  - Le symbole "="
  - Le constructeur de vecteurs
  - L'instruction DATA
- Manipulation de tableaux
  - Expressions de type tableau
  - Sections de tableaux

7 Entrées-Sorties

8 Procédures

9 Common

10 Include

Un tableau est un ensemble d'éléments de même type contigus en mémoire.
Pour déclarer un tableau, il est recommandé d'utiliser l'attribut DIMENSION :

```
TYPE, DIMENSION(expr1, ...,exprn) :: liste_tab
```

avec :

- $n \leq 7$ i.e un tableau peut avoir jusqu'à 7 dimensions
- $\texttt{expr}_i$ sert à indiquer l'**étendue** dans la dimension correspondante. C'est une expression qui peut être spécifiée à l'aide :
  - d'une constante entière (littérale ou symbolique) ; dans ce cas, la borne inférieure du tableau est 1,
  - d'une expression de la forme $\texttt{cste}_1\texttt{:cste}_2$ avec $\texttt{cste}_1$, $\texttt{cste}_2$ des constantes entières telles que $\texttt{cste}_1$ <= $\texttt{cste}_2$,
- liste_tab est une liste de tableaux.

Exemple

```
INTEGER, PARAMETER                :: lda = 6
REAL, DIMENSION(0:lda-1)          :: Y
REAL, DIMENSION(1+lda*lda,10)     :: Z

REAL, DIMENSION(100)              :: R
REAL, DIMENSION(1:5,1:5, 1:5)     :: S
REAL, DIMENSION(-10:-1)           :: X
```

Remarque : les dimensions d'un tableau peuvent aussi être spécifiées sans l'attribut DIMENSION :

Exemple

```
REAL :: T(10,10), U(4,2), G(-1:10,4:9,1:3)
```

Attention, cette notation n'est pas recommandée !

- Le **rang** (*rank*) d'un tableau est son nombre de dimensions.
- Le nombre d'éléments dans une dimension s'appelle l'**étendue** (*extent*) du tableau dans cette dimension.
- Le **profil** (*shape*) d'un tableau est un vecteur dont chaque élément est l'**étendue** du tableau dans la dimension correspondante.
- La **taille** (*size*) d'un tableau est le produit des éléments du vecteur correspondant à son **profil**.
- Deux tableaux sont dits **conformants** s'ils ont le même **profil**.

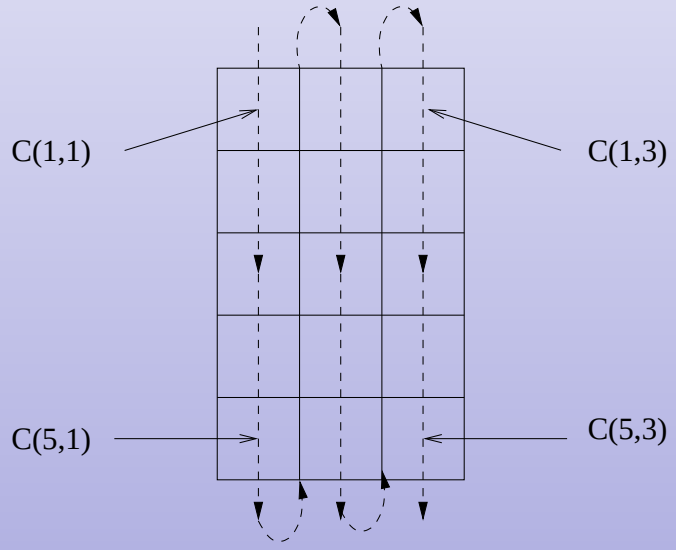
Exemple

```
REAL, DIMENSION(15)         :: X
REAL, DIMENSION(1:5,1:3)    :: Y
REAL, DIMENSION(-1:3,0:2)   :: Z
```

- Le tableau `X` est de rang `1`, `Y` et `Z` sont de rang `2` ;
- L'étendue de `X` est `15`, `Y` et `Z` ont une étendue de `5` et `3` ;
- Le profil de `X` est le vecteur `(/ 15 /)`, celui de `Y` et `Z` est le vecteur `(/ 5,3 /)`
- La taille des tableaux `X`, `Y` et `Z` est `15`.
- Les tableaux `Y` et `Z` sont conformants.

Ordre des éléments

En mémoire la notion de tableau n'existe pas : les éléments sont rangés les uns à la suite des autres.
Pour accéder à ces éléments, dans l'ordre mémoire, Fortran fait d'abord varier le premier indice, puis le second et ainsi de suite.
Par exemple, les éléments d'un tableau à deux dimensions sont ordonnés comme suit :
`REAL, DIMENSION(5,3) :: C` ⇒ `C(1,1),C(2,1),...,C(5,1),C(1,2),C(2,2),...,C(5,3)`



Fortran permet de manipuler globalement l'ensemble des éléments d'un tableau.
On pourra alors utiliser le symbole "=" comme pour l'initialisation d'une variable scalaire.
Pour initialiser à 3 l'ensemble d'un vecteur :

Exemple

```
REAL, DIMENSION(100) :: X = 3.
```

Un constructeur de vecteur est un **vecteur de scalaires** dont les valeurs sont encadrées par les caractères « (/ » et « /) » :

`tableau = (/ expr1, expr2, ..., exprn /)`

- `tableau` est un tableau de **rang** 1,
- $\texttt{expr}_i$ est :
  - un scalaire,
  - une boucle `DO` **implicite** de la forme `(expr_scalaire, variable = m1,m2[,m3])` avec `variable` une variable `INTEGER` correspondant à l'indice de cette boucle et m_1, m_2, m_3 des constantes entières délimitant la boucle (voir boucle `DO`).
- Le constructeur et le tableau `tableau` doivent être conformants.

Exemple

```
IMPLICIT NONE

REAL, DIMENSION(4)              :: heights =      &
                 (/ 5.10, 5.6, 4.0, 3.6 /)

CHARACTER(len=5), DIMENSION(3) :: colours =      &
                 (/ "RED  ", "GREEN", "BLUE " /)

INTEGER                         :: i
INTEGER, DIMENSION(10)          :: ints    =      &
                 (/ 100, (i, i=1,8), 100 /)
```

L'instruction DATA

Comme pour les variables simples, on peut utiliser l'instruction DATA pour initialiser les tableaux lors de leur déclaration. Elle permet d'initialiser tout ou partie de tableaux à l'aide d'une liste de constantes encadrée par le caractère « / » (la notation **n*valeur** peut être employée pour répéter une même valeur).
Les éléments à initialiser peuvent être spécifiés au moyen d'une boucle DO **implicite** : (tab(i), i = m_1,m_2[,m_3]).

Exemple

```
IMPLICIT NONE
INTEGER                    :: i, j

REAL,    DIMENSION(20)     :: A, B
INTEGER, DIMENSION(10)     :: ints
REAL,    DIMENSION(2,3)    :: MAT

DATA A/20*7.0/, B(1), B(7), B(10)/10., 3.87, 10.5/
DATA (ints(i),i=2,8,2)/4*6/, A(10)/-1.0/
DATA MAT/1., 2., 3., 4., 5., 6./
DATA ((MAT(i,j),j=1,3),i=1,2)/1., 2., 3., &
                              4., 5., 6./
```


Les tableaux peuvent être utilisés en tant qu'opérandes dans une expression :

- les opérateurs intrinsèques sont applicables à des tableaux conformants,
- les fonctions élémentaires s'appliquent à l'ensemble du tableau.

Dans ces cas, les fonctions ou opérateurs sont appliqués à chaque élément du tableau (log, sqrt, ...).

Exemple

```
REAL, DIMENSION(-4:0,0:2) :: B
REAL, DIMENSION(5,3)      :: C
REAL, DIMENSION(0:4,0:2)  :: D
      ...
B = C * D - B**2
B = SIN(C)+COS(D)
```

Les **sections régulières** de tableaux sont obtenus en faisant varier le ou les indices à l'aide d'un triplet de la forme :

`[limite1]:[limite2][:pas]`

- cette notation est équivalente à une pseudo-boucle ;
- une section de tableau est aussi un tableau ;
- le **rang** d'une section de tableau est inférieur ou égal à celui du tableau global ;
- un tel triplet indique que l'indice correspondant débute avec la valeur `limite1` et se termine à une valeur $\leq$ `limite2` ;
- pas est l'incrément de l'indice.

Exemple

```
A(:)        ! Le tableau global
A(3:9)      ! A(3) à A(9) par pas de 1
A(3:9:1)    ! Idem
A(m:n)      ! A(m) à A(n)
A(m:n:k)    ! A(m) à A(n) par pas de k
A(8:3:-1)   ! A(8) à A(3) par pas de -1
A(8:3)      ! A(8) à A(3), pas de 1 => taille nulle
A(m:)       ! de A(m) à la borne supérieure de A
A(:n)       ! de la borne inférieure de A à A(n)
A(::2)      ! borne inf. de A à borne sup., pas de 2
A(m:m)      ! section constituée de 1 élément
            ! (ce n'est pas un scalaire !)
A(m)        ! section équivalente à un scalaire
```

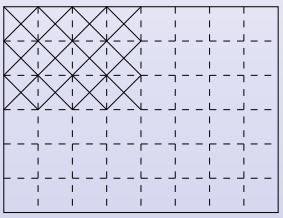
Seules les opérations sur des sections conformantes sont valides :

Exemple

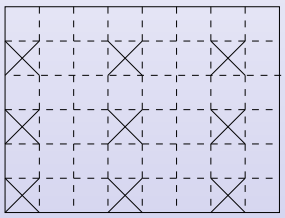
```
REAL, DIMENSION(1:6,1:8) :: P

P(1:3,1:4)      = P(1:6:2,1:8:2) ! VALIDE
P(2:8:2,1:7:3)  = P(1:3,1:4)     ! INVALIDE
P(2:6:2,1:7:3)  = P(2:5,7)       ! INVALIDE
```

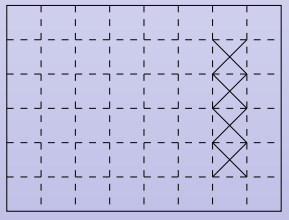
```
REAL, DIMENSION(1:6,1:8) :: P
```



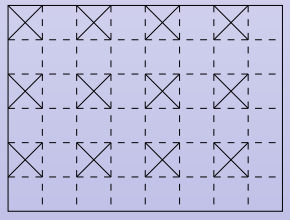
P(1:3,1:4)



P(2:6:2,1:7:3)



P(2:5,7), P(2:5,7:7)



P(1:6:2,1:8:2)

Attention, `P(2:5,7)` est une section 1D tandis que `P(2:5,7:7)` est une section 2D : ces 2 tableaux ne sont donc pas conformants.

1 Introduction

2 Généralités

3 Déclarations

4 Opérateurs et expressions

5 Structures de contrôle

6 Tableaux

7 Entrées-Sorties
- Introduction
- Accès séquentiel
  - Fichier binaire séquentiel
  - Fichier texte séquentiel
- Accès direct
  - Fichier binaire à accès direct
  - Fichier texte à accès direct
- Fichier temporaire
- Destruction d'un fichier
- Fichier interne

Instructions de positionnement
Instruction INQUIRE
Formats BOZ
Unités standards
Instruction FORMAT
Mots clés END=, ERR=

8 Procédures

9 Common

10 Include

Introduction

On appelle *entrée-sortie*, un transfert d'informations entre la mémoire de l'ordinateur et l'un de ses périphériques (un disque le plus souvent).
Une *entrée* se traduit par une *lecture* d'informations du périphérique vers la mémoire, tandis qu'une *sortie* implique une écriture de la mémoire vers le périphérique.
Ces informations sont stockées dans un fichier qui possède un nom.
L'unité de transmission entre la mémoire et le périphérique s'appelle le **bloc**. Il permet d'effectuer le traitement en passant par une zone intermédiaire dite *zone tampon* (*buffer*) permettant ainsi de limiter le nombre de transferts entre la mémoire et le périphérique : opération coûteuse.
L'unité de traitement est l'enregistrement logique : il correspond à la longueur des données traitées lors d'une opération de lecture-écriture.

L'exploitation d'un fichier au sein d'un programme nécessite au préalable son ouverture qui, en `Fortran`, est faite au moyen de l'instruction `OPEN`.
Cette instruction permet notamment :

- de connecter le fichier à un numéro d'*unité logique* : c'est celui-ci que l'on indiquera par la suite pour toute opération de lecture-écriture,
- de spécifier le mode désiré : lecture, écriture ou lecture-écriture,
- d'indiquer le mode de transfert : avec ou sans conversion en caractères,
- d'indiquer le mode d'accès au fichier : séquentiel ou direct.

Si l'ouverture du fichier est fructueuse, des lectures-écritures pourront être lancées à l'aide des instructions `READ/WRITE` par l'intermédiaire du numéro d'*unité logique*.
Une fois le traitement du fichier terminé, on le fermera au moyen de l'instruction `CLOSE`.

Accès séquentiel

On dit qu'un fichier est séquentiel lorsqu'il est nécessaire d'avoir traité les enregistrements précédant celui auquel on désire accéder.
Pour un fichier en lecture le paramètre `IOSTAT` de l'instruction `READ` permet notamment de gérer la fin de fichier ; celui-ci fait référence à une variable entière qui est valorisée à l'issue de la lecture comme suit :

- à `0` si la lecture s'est bien déroulée ;
- à une valeur positive si une erreur s'est produite ;
- à une valeur négative si la fin de fichier ou une fin d'enregistrement a été rencontrée.

On prendra soin de tester la valeur de cette variable **immédiatement après chaque lecture**.
Pour une analyse plus fine des erreurs de lecture se reporter à la page 159.

Fichier binaire séquentiel

On appelle fichier *binaire* un fichier dans lequel on stocke les informations telles qu'elles sont représentées en mémoire.
C'est au moment de l'ouverture du fichier que l'on indique le type de fichier à traîter.

Exemple

```
real, dimension(100) :: tab
integer              :: i
real                 :: r
integer              :: ios

OPEN( UNIT=1,              FILE="data_bin_seq", &
      FORM="unformatted", ACCESS="sequential", &
      ACTION="read",       POSITION="rewind",   &
      IOSTAT=ios )
if ( ios /= 0 ) stop "Problème à l'ouverture"
READ( UNIT=1, IOSTAT=ios ) tab, i, r
do while ( ios == 0 )
    ...
  READ( UNIT=1, IOSTAT=ios ) tab, i, r
end do
CLOSE( UNIT=1 )
```


On demande l'ouverture du fichier dont le nom est `data_bin_seq`. C'est un fichier binaire séquentiel (`unformatted, sequential`) que l'on désire lire depuis le début (`rewind`).
Ce fichier est connecté à l'unité logique dont le numéro est `1`. C'est ce numéro que l'on indique au moment de la lecture des variables `tab, i, r`, ainsi qu'à la fermeture du fichier.
En cas d'erreur lors de l'ouverture d'un fichier, en l'absence du mot-clé `IOSTAT` le programme s'interrompt avec édition d'un message résumant l'erreur. Ici ce mot-clé est précisé. Il permet de personnaliser le traitement en cas d'erreur. Il fait référence à une variable entière (ici la variable `ios`) laquelle, à l'issue de l'`OPEN`, est valorisée de la façon suivante :

- à zéro si tout s'est bien passé,
- à une valeur non nulle si une erreur s'est produite.

Il faudra évidemment la tester avant d'entreprendre toute opération d'entrée-sortie sur le fichier.
Ce même mot-clé est précisé au sein de l'instruction `READ`. Il référence la même variable `ios` qui reflètera l'état de la lecture une fois celle-ci effectuée. Elle est utilisée pour notamment gérer la fin de fichier.

Fichier texte séquentiel

Dans un fichier texte les données sont stockées sous forme de caractères. De ce fait :

- lors d'une lecture, elles sont converties en binaire avant d'être rangées en mémoire,
- lors d'une écriture, elles sont converties en caractères avant d'être écrites dans le fichier.

Cette opération de conversion est signalée au sein des instructions `READ/WRITE` :

- à l'aide d'une chaîne de caractères appelée **format d'édition** (paramètre `FMT=`),
- ou bien en utilisant un nom de liste (`NAMELIST`) regroupant les variables que l'on désire exploiter (paramètre `NML=`).

Formats d'édition

Pour que la conversion puisse être faite, il est nécessaire de connaître le type de la donnée à convertir.
Pour cela le **format d'édition** contient des **descripteurs** :

- descripteur `I` pour le type `INTEGER`,
- descripteurs `F, E` pour le type `REAL`,
- descripteur `L` pour le type `LOGICAL`,
- descripteur `A` pour le type `CHARACTER`.

Exemple

```
PROGRAM texte_sequentiel
    real, dimension(10) :: tab
    integer             :: i
    real                :: r
    integer             :: ios

    OPEN( UNIT=1,             FILE="data_txt_seq", &
          FORM="formatted",   ACCESS="sequential", &
          STATUS="old",       ACTION="write",      &
          POSITION="rewind", IOSTAT=ios )
    if ( ios /= 0 ) then ! Problème à l'ouverture
        ...
    else
      WRITE( UNIT=1, FMT="(10F8.4,I3,F6.3)") tab, i, r
        ...
    endif
        ...
    CLOSE( UNIT=1 )
END PROGRAM texte_sequentiel
```

Dans cet exemple, on demande l'ouverture du fichier dont le nom est `data_txt_seq`. C'est un fichier texte séquentiel (`formatted, sequential`) existant (`old`) que l'on désire écraser (`rewind`).
Comme précédemment, à l'issue de l'`OPEN` on teste la valeur de retour contenue dans l'entier `ios`.
Si l'ouverture s'est bien passée on lance, à l'aide de l'instruction `WRITE`, l'écriture en caractères d'un enregistrement comportant un tableau de réels (`tab`) suivi d'un entier puis d'un réel (`i, r`).
Le format d'édition spécifié sous la forme d'une constante chaîne de caractères (`'(10F8.4,I3,F6.3)'`) permet de convertir en caractères les variables ci-dessus :

- `10F8.4` : écriture des 10 éléments du tableau `tab`. Chacun a un gabarit de 8 caractères avec 4 chiffres en partie décimale,
- `I3` : écriture de l'entier `i` sur 3 caractères,
- `F6.3` : écriture du réel `r` sur 6 caractères avec 3 chiffres en partie décimale.

Formats d'édition en lecture

- `I`w permet la conversion des `w` caractères suivants dans le type `INTEGER`,
- `F`w`.`d : permet la conversion des `w` caractères suivants dans le type `REAL`. Si le point décimal n'est pas présent alors les `d` derniers caractères désignent la partie fractionnaire,
- `E`w`.`d : permet la conversion des w caractères suivants (interprétés comme un nombre réel en notation exponentielle) dans le type `REAL`,
- `L`w : permet la conversion des w caractères suivants dans le type `LOGICAL`,
- `A[`w`]` : permet de lire des caractères.

Format d'édition `I` en lecture

Dans ce qui suit le caractère ^ représente l'espace. Les différentes entrées présentées sont supposées figurer dans un fichier texte séquentiel connecté à l'unité 1 après un `OPEN`.

Exemple

```
INTEGER i, j
        ...
READ( UNIT=1, FMT="(I5,I4)" ) i, j
        ...
```

Entrées	Affectations
^^45^^^9^	i=45, j=9
^-24^10^^	i=-24, j=10

À noter : dans un champ en entrée, l'espace est ignoré.

Formats d'édition F, E et D en lecture

Ce format à la forme générale : F$w.d$, E$w.d$ ou bien D$w.d$.
Le nombre réel à lire peut être soit en notation virgule fixe, soit exponentielle avec, dans ce dernier cas, l'exposant préfixé de la lettre E ou D.
Le point décimal peut ne pas figurer :

- s'il est spécifié alors le nombre indiqué par d est ignoré, c'est le nombre de décimales figurant en entrée qui est pris en compte,
- s'il est omis c'est le nombre indiqué par d qui est considéré.

Exemple

```
REAL x, y
   ...
READ( UNIT=1, FMT="(F4.1,F6.2)" ) x, y
   ...
```

Entrées	Affectations
^3.1-3.141	x=3.1, y=-3.141
^123^^5678	x=12.3, y=56.78

Exemple

```
REAL x
   ...
READ( UNIT=1, FMT="(E12.6)" ) x
! ou bien
READ( UNIT=1, FMT="(F12.6)" ) x
   ...
```

Entrées	Affectations
2.718281^^^^	x=2.718281
2718281^^^^^	x=2.718281
27.18281d-1^	x=2.718281
.2718281e+1^	x=2.718281
.2718281^e^1	x=2.718281
^^^^^2718281	x=2.718281
^^2^7^18^281	x=2.718281
^^^^^^^^^271	x=0.000271

Format d'édition L en lecture

Ce format à la forme générale : Lw
Ce type de format permet la lecture de valeurs logiques.
Le caractère w indique comme précédemment la largeur du champ en entrée. Celui-ci doit comporter comme premier caractère non blanc l'une des lettres `F`, `f`, `T` ou `t` éventuellement précédée du caractère `'.'`. N'importe quels caractères peuvent compléter le champ.

Exemple

```
LOGICAL l1, l2
   ...
READ( UNIT=1, FMT="(L6,L7)" ) l1, l2
   ...
```

Entrées	Affectations
.true..false.	l1=.TRUE., l2=.FALSE.
^^^^t..FACILE	l1=.TRUE., l2=.FALSE.
t^^^^^F^^^^^^	l1=.TRUE., l2=.FALSE.

Format d'édition A en lecture

Ce format à la forme générale : A[w]
Ce type de format permet la lecture de caractères. La valeur w indique le nombre de caractères que l'on désire traiter en entrée.

- si la valeur de w est plus grande que la longueur `l` de la chaîne réceptrice, ce sont les `l` caractères les plus à droite qui seront lus,
- si elle est plus petite, w caractères seront lus et stockées dans la chaîne réceptrice qui sera complétée à droite par des blancs,
- si elle n'est pas précisée, c'est la longueur spécifiée lors de la déclaration de la chaîne de caractères qui indiquera le nombre de caractères à lire. Si la fin de l'enregistrement est atteinte avant la fin de la lecture, la chaîne est complétée par des blancs.

Exemple

```
CHARACTER(len=7) :: ch1, ch2
   ...
READ( UNIT=1, FMT="(A6,A8)" ) ch1, ch2
READ( UNIT=1, FMT="(A6,A8)" ) ch1, ch2
READ( UNIT=1, FMT="(A,A)"   ) ch1, ch2
   ...
```

Entrées	Affectations
BACH^^^^BACH^^	ch1="BACH^^^^",ch2="^BACH^^"
MOZARTHAENDEL^	ch1="MOZART^",ch2="AENDEL^"
MOZARTHAENDEL^	ch1="MOZARTH",ch2="AENDEL^"

Formats d'édition en écriture

- `I`w`[.`d`]` permet l'édition d'une variable de type `INTEGER` sur w caractères. S'il est présent d indique le nombre minimum de chiffres édités : si nécessaire des 0 apparaîtront en tête du nombre,
- `F`w`.`d : permet l'édition d'une variable de type `REAL` sur w caractères comprenant le point décimal suivi de d chiffres pour la partie fractionnaire,
- `E`w`.`d : idem format `F` mais la sortie est faite en notation exponentielle,
- `L`w : permet l'édition d'une variable de type `LOGICAL` sur w caractères,
- `A[`w`]` : permet l'édition d'une variable de type `CHARACTER`.

Format d'édition I en écriture

Exemple

```
INTEGER i, j, k

i = -125
j = 1756
k = 1791
WRITE( UNIT=1, FMT="(I4,I4,I4)" ) i, j, k
WRITE( UNIT=1, FMT="(I5,I6,I6)" ) i, j, k
WRITE( UNIT=1, FMT="(I6.5,I6.6,I6.6)" ) i, j, k
```

Sorties
-12517561791
^-125^^1756^^1791
-00125001756001791

Format d'édition F en écriture

Exemple

```
REAL x, y, z

x = 3.14159
y = -15.137
z = 799.7432
WRITE( UNIT=1, FMT="(F7.5,F8.3,F9.4)" ) x, y, z
WRITE( UNIT=1, FMT="(F6.2,F9.4,F10.5)" ) x, y, z
```

Sorties
3.14159^-15.137^799.7432
^^3.14^-15.1370^799.74323

En ce qui concerne les formats[1] d'écriture `I`, `B`, `O`, `Z` et `F`, lorsque le gabarit de la zone réceptrice est insuffisant celle-ci est remplie par le caractère `*`. Depuis la norme `Fortran 95` il est possible d'éviter cela en indiquant `0` comme largeur de zone.

Exemple

```
PROGRAM gabarit
  INTEGER I
  REAL    R

  I = 129876
  R = -2345.78

  WRITE( UNIT=1, FMT="(I4, F6.3)" ) I, R
  WRITE( UNIT=1, FMT="(I0, F0.3)" ) I, R
END PROGRAM gabarit
```

Sorties
****
129876-2345.780

[1] formats `B`, `O`, `Z` : c.f. Remarques en fin de chapitre.

Format d'édition `E` et `D` en écriture

Avec les format `E`w`.`d, `D`w`.`d on obtiendra en sortie le motif :

```
S0.XXXXXXXESXX, S0.XXXXXXXDSXX
   <--d-->         <--d-->
<-----w------>  <-----w------>
```

Le caractère `S` indique une position pour le signe.
Un facteur d'échelle peut précéder ce type de format. Il s'écrit sous la forme `kP` et permet d'éditer le nombre avec `k` chiffres avant le point décimal (modifie en conséquence la valeur de l'exposant).
Si `-d` < `k` $\leq$ `0`, la partie décimale sera constituée de `|k|` zéros suivis de `d-|k|` chiffres significatifs.
Si `0` < `k` < `d+2`, le nombre en sortie sera constitué de `k` chiffres significatifs avant le point décimal et de `d-k+1` chiffres significatifs en partie décimale.
Toute autre valeur de `k` est invalide : dans ce cas, la zone en sortie sera remplie par le caractère `*`.
Celui-ci s'applique à tous les formats `E` qui suivent. Pour retrouver le comportement par défaut il suffit de préciser le facteur `0P`.

Exemple

```
REAL x, y, z
DOUBLE PRECISION xd, yd, zd

x = 3.14159;    xd = 3.14159d0
y = -15.137;    yd = -15.137d0
z = 799.74328; zd = 799.74328d0
WRITE( UNIT=1, FMT="(D12.6,E13.5,E15.8)" ) x, y, z
WRITE( UNIT=1, FMT="(4P,D12.6,E13.5,0P,E10.3)" ) x, y, z
WRITE( UNIT=1, FMT="(D12.6,E13.5,E15.8)" ) xd, yd, zd
WRITE( UNIT=1, FMT="(4P,D12.6,E13.5,0P,E15.8)" ) xd, yd, zd
```

Sorties
0.314159D+01^-0.15137E+02^0.79974329E+03
3141.590D-03^-1513.70E-02^0.800E+03
0.314159D+01^-0.15137E+02^0.79974328E+03
3141.590D-03^-1513.70E-02^0.79974328E+03

Il existe deux variantes du format E qui correspondent aux descripteurs EN et ES. Le facteur d'échelle kP ne peut être appliqué dans ce contexte.
Le descripteur EN permet la sortie d'un réel en notation dite *ingénieure*. Dans ce type de représentation, la valeur absolue de la partie entière est supérieure ou égale à 1 et inférieure à 1000 et l'exposant est divisible par 3.

Exemple

```
REAL x, y, z, t

x = 6.421
y = -.5
z = .00217
t = 4721.3
WRITE( UNIT=1, FMT="(EN12.3,EN12.3,EN12.3,EN12.3)" ) x, y, z, t
```

Sorties
^^^6.421E+00-500.000E-03^^^2.170E-03^^^4.721E+03

Le descripteur ES permet la sortie d'un réel en notation dite *scientifique*. Dans ce type de représentation, la valeur absolue de la partie entière est supérieure ou égale à 1 et inférieure à 10.

Exemple

```
REAL x, y, z, t

x = 6.421
y = -.5
z = .00217
t = 4721.3
WRITE( UNIT=1, FMT="(ES12.3,ES12.3,ES12.3,ES12.3)" ) x, y, z, t
```

Sorties
^^^6.421E+00^^-5.000E-01^^^2.170E-03^^^4.721E+03

Par défaut, la présence du signe + pour les données numériques positives dépend du compilateur. Il existe des descripteurs permettant de forcer sa présence ou bien de le supprimer :

- SS (*sign suppress*) : le signe + n'apparaît pas,
- SP (*sign print*) : le signe + précède la donnée,
- S : restauration du mode par défaut.

Exemple

```
INTEGER i, j
REAL x, y

i = 1756
j = -69
x = 6.421
y = .00217
WRITE( UNIT=1, FMT="(SP,F7.3,SS,ES12.3,I4,SP,I6)" ) x, y, j, i
```

Sorties
^+6.421^^^2.170E-03^-69^+1756

Format d'édition L en écriture

Ce format à la forme générale : Lw
Ce type de format permet l'écriture de valeurs logiques.
En sortie on obtiendra `w-1` blancs suivis de la lettre `T` pour une valeur `.true.` et `F` pour une valeur `.false.`.

Exemple

```
LOGICAL l1/.true./, l2/.false./
   ...
WRITE( UNIT=1, FMT="(L6,L7)" ) l1, l2
   ...
```

Sorties
^^^^^T^^^^^^F

Format d'édition A en écriture

Le format `A[`w`]` permet la sortie de chaînes de caractères. La valeur w est facultative. Si elle est précisée, elle indique la largeur de la zone réceptrice.

- si la valeur de w est plus grande que la longueur `l` de la chaîne, en sortie celle-ci apparaîtra précédée de w-`l` blancs,
- si elle est plus petite, seuls les w caractères les plus à gauche de la chaîne seront écrits,
- si la valeur w est absente, c'est la longueur de la chaîne spécifiée à la déclaration qui indique la largeur du champ en sortie.

Exemple

```
CHARACTER(len=9) :: ch1, ch2, ch3
   ...
ch1 = "BEETHOVEN"
ch2 = "PUCCINI"
ch3 = "VERDI"
WRITE( UNIT=1, FMT="(A9,A8,A6,A)" ) ch1, ch2, ch3, ch3
WRITE( UNIT=1, FMT="(A10)" ) ch3
ch1 = "Ludwig"
ch2 = " Van"
ch3 = "BEETHOVEN"
WRITE( UNIT=1, FMT="(A,A,A,A)" ) trim(ch1), trim(ch2), " ", ch3
   ...
```

Sorties
BEETHOVENPUCCINI^VERDI^VERDI^^^^
^VERDI^^^^
Ludwig^Van^BEETHOVEN

Format d'édition : *Litteral string*

Si une constante littérale de type chaîne de caractères est spécifiée dans un format, celle-ci est reproduite telle quelle en sortie.

Exemple

```
CHARACTER(len=9) :: ch1, ch2, ch3
   ...
ch1 = "Ludwig"
ch2 = " Van"
ch3 = "BEETHOVEN"
WRITE( UNIT=1, FMT="('NOM : ',A,', PRÉNOM : ',A,A)" ) &
       ch3, trim(ch1), trim(ch2)
     ...
```

Sortie
NOM^:^BEETHOVEN,^PRÉNOM^:^Ludwig^Van

Descripteurs de contrôle

- descripteurs de positionnement :
  - n`X` : ignore (en entrée), saute (en sortie) les n caractères suivants,
  - `T`c : permet de se positionner au caractère de rang c,
  - `TL`n : permet de se positionner au caractère situé n positions à gauche par rapport à la position courante,
  - `TR`n : permet de se positionner au caractère situé n positions à droite par rapport à la position courante.
- descripteurs de gestion des blancs dans les champs numériques en entrée :
  - `BN` (*Blank Null*) : ignore les blancs,
  - `BZ` (*Blank Zero*) : interprète le caractère blanc comme un `0`.

Exemple

```
INTEGER i, j, k, l
   ...
READ( UNIT=1, FMT="(I4,3X,I2,T12,I3,TL4,I1)" ) i, j, k, l
      ...
```

Entrées	Affectations
1901^1254^4361	i=1901,j=54,k=361,l=4

Exemple

```
INTEGER i, j, k
   ...
READ( UNIT=1, FMT="(I3,BZ,I2,BN,I3)" ) i, j, k
      ...
```

Entrées	Affectations
^8^2^4^^	i=8,j=20,k=4

Un autre descripteur, /, provoque le passage à l'enregistrement suivant :

- en entrée : abandon de l'enregistrement courant et positionnement au début du suivant,

Exemple

```
INTEGER i, j

READ( UNIT=1, FMT="(I4,/,I4)") i, j
```

Entrées	Affectations
1756^1254	i=1756,j=1791
1791	

- en sortie : écriture du caractère *newline*.

Exemple

```
CHARACTER(len=9) :: ch1, ch2, ch3

ch1 = "Ludwig"
ch2 = " Van"
ch3 = "BEETHOVEN"
WRITE( UNIT=1, FMT="('NOM     : ',A,/,'PRÉNOM : ',A,A)" ) ch3, trim(ch1), trim(ch2)
```

Sortie
NOM^^^^:^BEETHOVEN
PRÉNOM^:^Ludwig^Van

Facteur de répétition

Lorsqu'une liste de descripteurs identiques figure dans un format il est possible de les regrouper au moyen d'un facteur de répétition indiqué sous la forme d'une constante littérale entière.

Exemple

```
INTEGER i, j, k
INTEGER t(3)
   ...
READ( UNIT=1, FMT="(I4,I4,I4)" ) i, j, k
READ( UNIT=1, FMT="(3I4)" ) t
   ...
WRITE(6, "(3(1X,I4))") t
```

Entrées	Affectations
^^45^^^9^^10	i=45, j=9, k=10
^-24^10^^^99	t(1)=-24, t(2)=10, t(3)=99

Sortie
^^-24^^^10^^^99

Réexploration d'un format

L'ensemble des variables spécifiées dans une instruction READ/WRITE s'appelle la liste d'entrée-sortie. Chaque élément de cette liste est associé à un descripteur du format.

- si le nombre de descripteurs correspond au nombre de variables de la liste, ceux-ci s'appliquent aux éléments successifs de cette liste,
- s'il est plus grand, les suivants sont ignorés (le format est abandonné),
- s'il est plus petit, le format est réexploré. Cette réexploration entraîne le passage à l'enregistrement suivant.

Conclusion

En Fortran la liste d'entrée-sortie est toujours satisfaite.

La règle de réexploration est la suivante :

- si le format ne contient aucun groupe de descripteurs entre parenthèses, alors il est réexploré depuis son début,
- sinon, la réexploration est faite à partir du groupe de niveau 1 le plus à droite. S'il est précédé d'un facteur de répétition, il est pris en compte.

Le caractère « | » dans les exemples suivant indique l'endroit à partir duquel la réexploration est effectuée :

```
"( I6, 10X,I5, 3F10.2 )"
    | ⇐
"( I6, 10X,I5, (3F10.2) )"
                | ⇐
"( I6,(10X,I5), 3F10.2 )"
       | ⇐
"( F6.2, (2F4.1,2X,I4, 4(I7,F7.2)) )"
          | ⇐
"( F6.2, 2(2F4.1,2X,I4), 4(I7,F7.2) )"
                           | ⇐
"( F6.2,(2(2F4.1,2X,I4), 4(I7,F7.2)) )"
         | ⇐
```

Exemple

```
INTEGER :: i=100, j=200 ! depuis Fortran 90.
INTEGER :: k, l
DATA    k/300/, l/400/  ! ancienne forme
                        ! d'initialisation.
REAL    t(3,4)
DATA    t/  1.,   4.,   9., &
           16.,  25.,  36., &
           49.,  64.,  81., &
          100., 121., 144. /

WRITE(6, "( 4i5, (t25,4f8.2) )") i, j, k, l, &
                                 ((t(i,j),j=1,4),i=1,3)
```

Sorties
^^100^^200^^300^^400^^^^^^^1.00^^^16.00^^^49.00^^100.00
^^^^^^^^^^^^^^^^^^^^^^^^^^^^4.00^^^25.00^^^64.00^^121.00
^^^^^^^^^^^^^^^^^^^^^^^^^^^^9.00^^^36.00^^^81.00^^144.00

Format libre

En Fortran il existe un format implicite appelé **format libre** (*list-directed formatting*).
Dans l'instruction READ/WRITE, on spécifie alors le caractère * à la place du format.
Dans ce contexte, les enregistrements sont interprétés comme une suite de valeurs séparées par des caractères appelés **séparateurs**. C'est le type des variables auxquelles ces valeurs vont être affectées qui détermine la conversion à effectuer.
Les caractères interprétés comme des séparateurs sont :

- la virgule (,) ;
- le blanc (espace).

Une chaîne de caractères contenant un caractère séparateur doit être délimitée soit par des quotes (') soit par des guillemets (").
En entrée, plusieurs valeurs identiques peuvent être regroupées à l'aide d'un facteur de répétition sous la forme `n*valeur`.

Une constante complexe est codée comme 2 nombres réels entre parenthèses séparés par une virgule. Les parenthèses ainsi que la virgule peuvent être suivies ou précédées de blancs.
Une fin d'enregistrement (*newline*) a le même effet qu'un blanc. Elle peut apparaître :

- au sein d'une chaîne de caractères (délimitée par des quotes (') ou par des guillemets (")),
- entre la partie réelle et la virgule ou entre la virgule et la partie imaginaire d'une constante complexe.

Si une chaîne de caractères a été déclarée avec une longueur plus petite que celle de la valeur lue alors seuls les caractères les plus à gauche de la valeur seront stockés dans la chaîne. Sinon, celle-ci est complétée par des blancs.
Si dans l'enregistrement en entrée apparaissent deux virgules à la suite (éventuellement avec des blancs entre) alors l'élément correspondant de la liste ne sera pas modifié.

Exemple

```
INTEGER i/100/, &
        j/200/, &
        k/300/
REAL    t(3)
COMPLEX c
CHARACTER(len=18) ch
    ...
READ ( UNIT=1, FMT=* ) i, j, k, t, c, ch
WRITE( UNIT=2, FMT=* ) i, j, k, c, ch(1:8)
WRITE( UNIT=2, FMT=* ) t
```

Entrées	Affectations
150	i=150
260,,	j=260
3*2.718	t=(/ 2.718,2.718,2.718 /)
(^2.	c=(2.,3.)
,^3.^)^^^'Wolfgang^	ch='Wolfgang Amadeus M'
Amadeus Mozart'	

Sorties
^150^260^300^(2.000000000,3.000000000)^Wolfgang
^2.717999935^2.717999935^2.717999935

namelist

On peut regrouper les variables que l'on désire lire ou écrire au sein d'une liste à laquelle on donne un nom.
Ce regroupement s'effectue au moyen de l'instruction NAMELIST :

```
NAMELIST/nom_liste/liste_variables
```

- `nom_liste` est le nom de la NAMELIST,
- `liste_variables` est une liste de variables précédemment déclarées.

Au niveau de l'instruction READ/WRITE la *namelist* remplace le format ainsi que la liste de variables qui, dans ce contexte, devient inutile.
l'enregistrement correspondant (lu ou écrit) respecte le format suivant :

```
&nom_liste liste d'affectations /
```

La liste d'affectations concerne les variables de la *namelist* qui peuvent apparaître dans n'importe quel ordre, certaines pouvant être omises. Les différentes affectations sont séparées par des caractères séparateurs (, ou blancs).
Le caractère / indique la fin de la *namelist*. En entrée, les éventuels caractères qui suivent sont ignorés.
En entrée les chaînes de caractères doivent être délimitées à l'aide du caractère ' ou ".
En sortie celles-ci apparaissent par défaut sans délimiteur. C'est le paramètre DELIM= de l'instruction OPEN qui permet de le définir.
Dans l'instruction READ/WRITE, la *namelist* est indiquée à l'aide du paramètre NML= (à la place de FMT=).
L'utilisation des *namelist* est un moyen très commode d'effectuer des entrées-sorties sans avoir à définir de format.

Exemple

```
INTEGER i/100/, j/200/, k/300/
INTEGER t(3)
CHARACTER(len=11) ch
NAMELIST/LISTE/i,j,k,t,ch
     ...
READ ( UNIT=1, NML=liste )
WRITE( UNIT=2, NML=liste )
     ...
```

Entrées
&LISTE t=3*2,i=1, k=4 ch="Rythm&Blues" /
Affectations
i=1,k=4,t(1)=t(2)=t(3)=2,ch="Rythm&Blues"

Sorties
^&LISTE
^I=1,^J=200,^K=4,^T=2,^2,^2,^CH=Rythm&Blues
^/

Accès direct

À la différence d'un fichier séquentiel, il est possible d'accéder à un enregistrement d'un fichier à accès direct sans avoir traité les précédents.
Chaque enregistrement est repéré par un numéro qui est son rang dans le fichier. Leur taille est fixe.
Au sein de l'instruction OPEN :

- le paramètre RECL= est obligatoire, sa valeur indique la taille des enregistrements (en caractères pour les fichiers textes, dépend du processeur pour les fichiers binaires),
- le paramètre POSITION= est invalide,
- si le paramètre FORM n'est pas précisé, c'est la valeur *unformatted* qui est prise en compte.

Le rang de l'enregistrement que l'on désire traiter doit être spécifié à l'aide du paramètre REC= de l'instruction READ/WRITE. Un enregistrement ne peut pas être détruit mais par contre il peut être réécrit. Dans ce contexte, les *namelist* ainsi que le format libre sont interdits.

Fichier binaire à accès direct

Exemple

```
real, dimension(100) :: tab
integer ios, n

OPEN( UNIT=1,           FILE="data_bin_direct", &
      ACCESS="direct", ACTION="read",          &
      STATUS="old",     RECL=400 )

OPEN( UNIT=2,         FILE="data_txt_seq", &
      ACTION="read", STATUS="old" )
  ...
READ( UNIT=2, FMT=* ) n
READ( UNIT=1, REC=n, IOSTAT=ios ) tab
if ( ios > 0 ) then
  print "(a,i0,/,a)", "Erreur lors du traitement de &
                      &l'enregistrement de rang ", rang, &
                      "    ==> peut-être un problème d'existence."
  stop 4
end if
    ...
CLOSE( UNIT=2 )
CLOSE( UNIT=1 )
```


Le fichier dont le nom est `data_bin_direct` est connecté à l'unité logique numéro 1. C'est un fichier binaire à accès direct (`ACCESS="direct"` et paramètre `FORM` absent donc considéré égal à *unformatted*). Chaque enregistrement fait 400 octets (`RECL`=400). On accède à l'enregistrement de rang `n`, valeur préalablement lue dans le fichier texte séquentiel de nom `data_txt_seq` connecté à l'unité logique numéro 2. Le paramètre `IOSTAT` de l'instruction `READ` permet de récupérer l'état de la lecture dans l'entier `ios` : une valeur non nulle positive signale une erreur du type enregistrement inexistant par exemple.

Fichier texte à accès direct

Exemple

```
real, dimension(100) :: tab
integer ios, n

OPEN( UNIT=1,            FILE="data_txt_direct", &
      FORM="formatted", ACCESS="direct",         &
      ACTION="read",     STATUS="old",           &
      RECL=800 )

OPEN( UNIT=2,         FILE="data_txt_seq", &
      ACTION="read", STATUS="old" )

READ( UNIT=2, FMT=* ) n
READ( UNIT=1, REC=n, IOSTAT=ios, FMT="(100F8.4)" ) tab
if ( ios > 0 ) then
  print "(a,i0,/,a)", "Erreur lors du traitement de &
                      &l'enregistrement de rang ", rang, &
                      "    ==> peut-être un problème d'existence."
  stop 4
end if
     ...
CLOSE( UNIT=2 ); CLOSE( UNIT=1 )
```


www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

On a adapté l'exemple précédent à un fichier texte à accès direct : le paramètre FORM="formatted" a donc été précisé.
La valeur du paramètre RECL correspond à la taille en caractères de chacun des enregistrements qui correspond ici au format indiqué au niveau de l'instruction READ (100*8 = 800).

Notes :

- il n'est pas permis de lire un fichier texte à accès direct au moyen d'un format libre,
- un fichier dans lequel on écrit un seul enregistrement de rang n, contiendra en réalité n enregistrements avec les n-1 premiers ayant un contenu indéterminé. Lors de la lecture d'un tel fichier, se pose le problème de la reconnaissance de la nature de celui-ci. On pourra résoudre ce problème en adjoignant aux données une marque sous la forme d'une variable logique par exemple.

Dans l'exemple suivant des enregistrements de rang aléatoire sont créés. Pour les repérer on ajoute en tête des données une variable logique *flag*.

Exemple

```
program dir
  implicit none
  integer, parameter    :: n = 100, m = 1000
  real, dimension(n,m) :: matrice
  real, dimension(n)    :: vec
  integer i,j, rang, ios, size
  logical flag

  inquire( iolength=size ) flag, matrice(:,1)
  open( unit=1,                &
        file="direct.file", &
        form="unformatted", &
        access="direct",     &
        action="readwrite", &
        status="unknown",    &
        recl=size )
  flag = .false.
  do j=1,m
    write( unit=1, rec=j ) flag
  end do
  call random_number( matrice )
  flag = .true.
  do j=1,m
    if ( matrice(1,j) > matrice(n,j) ) &
      write( unit=1, rec=j ) flag, matrice(:,j)
  end do
```


Exemple (suite)

```
  do
    print *,"Entrez un rang : "
    read( *, *, iostat=ios ) rang
    if ( ios < 0 ) exit
    if ( ios > 0 ) then
      print *,"Erreur de saisie. Veuillez recommencer."
      cycle
    end if
    read( unit=1, rec=rang, iostat=ios ) flag, vec
    if ( ios > 0 ) then
      print "(a,i0,/,a)", "Erreur lors du traitement de &
                          &l'enregistrement de rang ", rang, &
                          "    ==> peut-être un problème d'existence."
      cycle
    end if
    if ( flag ) then
      print *,"vec(1) = ", vec(1), "vec(n) = ", vec(n)
    else
      print *,"Enregistrement ", rang, " indéfini."
    end if
  end do
  close(1)
end program dir
```

Notes :

- l'instruction `INQUIRE( iolength=size ) ...` permet de récupérer dans l'entier *size* la taille de la liste de variables fournies qui servira à renseigner, par la suite, le paramètre `RECL` de l'instruction `OPEN`. Comme cette valeur est exprimée dans une unité dépendant du compilateur (pour les fichiers binaires en accès direct), cette nouvelle forme de l'instruction `INQUIRE`[2], introduite par la norme `Fortran 90`, permet de la calculer en s'affranchissant de l'unité ;
- l'instruction `call random_number( matrice )` fait appel à la procédure intrinsèque `Fortran 90 RANDOM_NUMBER` qui valorise le tableau transmis à l'aide de nombres aléatoires générés dans l'intervalle $[0., 1.]$.

[2] cette instruction est détaillée plus loin.

Fichier temporaire

Si à l'ouverture d'un fichier on positionne le mot-clé `STATUS` à la valeur `"scratch"` alors celui-ci sera temporaire et détruit à sa fermeture. Un tel fichier est anonyme : le paramètre `FILE` de l'instruction `OPEN` ne doit donc pas être spécifié dans ce contexte.

Exemple

```
OPEN( UNIT=1,             &
      FORM="formatted",   &
      ACCESS="direct",    &
      ACTION="write",     &
      STATUS="scratch",   &
      RECL=1200 )
   ...
CLOSE( UNIT=1 )
```

Destruction d'un fichier

L'instruction CLOSE admet le paramètre à mot-clé STATUS qui par défaut (pour un fichier non temporaire) prend la valeur "KEEP" permettant ainsi de conserver le fichier après fermeture. Si on désire le supprimer il suffira de préciser la valeur "DELETE" pour ce paramètre.

Exemple

```
logical condition

OPEN( UNIT=1,           &
      FILE="data_seq", &
      ACTION="read",    &
      STATUS="old" )
    ...
if ( condition ) then
  CLOSE( UNIT=1, STATUS="delete" )
else
  CLOSE( UNIT=1 )
end if
```

Fichier interne

On appelle fichier interne un fichier dont les enregistrements sont en mémoire. Ce type de fichier induit des échanges entre zones de la mémoire et non plus entre un support externe et la mémoire.
Ces fichiers sont pré-connectés : il n'y a donc aucune ouverture ni fermeture à effectuer.
Dans les instructions READ/WRITE, à la place du numéro d'unité logique on indique une variable de type chaîne de caractères. C'est celle-ci qui fait référence à l'enregistrement en mémoire.
Seul l'accès séquentiel formaté est permis dans ce cas. Les *namelist* sont interdites.
Lors d'une écriture il faut s'assurer que la chaîne de caractères réceptrice est de taille suffisante.
Lors d'une lecture, la fin de fichier est atteinte lorsqu'on essaie d'accéder aux caractères situés au-delà de la chaîne qui fait référence à l'enregistrement.

Fichier interne

Exemple

```
INTEGER, PARAMETER     :: n = 4, m = 6
REAL, DIMENSION(n,m)   :: tab
CHARACTER(LEN=8)       :: format = "(  F8.3)"
INTEGER                :: i, j

WRITE( format(2:3), "(I2)" ) n ! fichier interne
OPEN( UNIT=1,                &
      FILE="data_txt_seq", &
      POSITION="rewind",   &
      ACTION="write",      &
      STATUS="new" )
WRITE( UNIT=1, FMT=format ) ((tab(i,j),i=1,n),j=1,m)
CLOSE( UNIT=1 )
```

Dans un format le facteur de répétition doit obligatoirement être précisé à l'aide d'une constante littérale. Cet exemple montre comment le générer dynamiquement en utilisant un fichier interne.

Voici un autre exemple dans lequel on traîte une centaine de fichiers dont les noms sont de la forme FILExxx avec xxx prenant les valeurs de 001 à 100.

Exemple

```
PROGRAM groupe_fichiers
  implicit none
  character(len=7) :: nom_fichier = "FILExxx"
  integer i

  do i=1,100
    write( nom_fichier(5:7), "(i3.3)" ) i ! et non pas "(i3)".
    open ( unit=1, file=nom_fichier, ... )
        ...
    close( unit=1 )
  end do
END PROGRAM groupe_fichiers
```

Exemple

```
PROGRAM fichier_interne
  CHARACTER(len=80) enreg
  INTEGER           ios
  REAL              x, y, z
  NAMELIST/liste/x, y, z

  OPEN( UNIT=1,           FILE="data_txt_seq", &
        FORM="formatted", ACTION="read",       &
        STATUS="old",     POSITION="rewind" )

  DO
    READ( UNIT=1, FMT="(a)", IOSTAT=ios ) enreg
    if ( ios /= 0 ) exit
    IF ( VERIFY( enreg, " ,+-0123456789.eEdD" ) == 0 ) THEN
      READ( enreg, FMT=*, iostat=ios ) x, y, z
      !---------------------------------------
      WRITE( UNIT=*, NML=liste )
    END IF
  END DO
  if ( ios > 0 ) stop "Erreur lors de la lecture."
  CLOSE( UNIT=1 )
       ...
END PROGRAM fichier_interne
```

Dans cet exemple on lit un fichier en ne traitant que les enregistrements constitués de réels et en ignorant tous les autres.

Instructions de positionnement

Toute opération de lecture-écriture dans un fichier est effectuée par rapport à la position courante dans ce fichier. À l'ouverture celle-ci peut être précisée à l'aide du paramètre `POSITION`. Dans un fichier séquentiel toute lecture-écriture d'un enregistrement de rang `n` implique le positionnement à l'enregistrement de rang `n+1`.
Trois instructions `BACKSPACE`, `REWIND` et `ENDFILE` permettent de modifier la position :

- `BACKSPACE` force la position au début de l'enregistrement précédent,
- `REWIND` force la position au début du fichier,
- `ENDFILE` écrit un enregistrement de type fin de fichier. Il est alors nécessaire d'exécuter ensuite l'une des deux instructions précédentes.

Ces instructions admettent en paramètre le numéro de l'unité logique auquel le fichier est connecté.

Exemple : troncature contrôlée d'un fichier

Exemple

```
program troncature
  REAL, dimension(100) :: tab
  INTEGER              :: ios
  LOGICAL              :: flag = .false.

  OPEN( UNIT=1,               &
        FILE="data_txt_seq", &
        ACTION="readwrite",  &
        POSITION="append",   &
        STATUS="old" )

  tab(:) = acos(-1.)
  WRITE( UNIT=1, FMT="(100F6.3)" ) tab
  REWIND( UNIT=1 )
```

Le fichier dont le nom est `data_txt_seq` est ouvert avec un positionnement en fin de fichier (`POSITION="append"`). Après écriture d'un enregistrement, on se repositionne en tête (`REWIND`).

Exemple (suite)

```
  READ( UNIT=1, FMT="(100F6.3)", IOSTAT=ios ) tab
  DO WHILE( ios == 0 )
    if ( flag ) then
      BACKSPACE( UNIT=1 )
      ENDFILE( UNIT=1 )
      BACKSPACE( UNIT=1 )
    END IF
       ...
    READ( UNIT=1, FMT="(100F6.3)",IOSTAT=ios ) tab
  END DO
  CLOSE( UNIT=1 )
end program troncature
```

Ensuite on relit le fichier et si la variable `flag` contient la valeur `.TRUE.` on le tronque avant le dernier enregistrement lu. (Instructions `BACKSPACE` et `ENDFILE`).

Instruction INQUIRE

L'instruction d'interrogation INQUIRE permet de récupérer un certain nombre d'informations concernant un fichier ou un numéro d'unité logique.
Elle permet par exemple de tester si un fichier existe, s'il est connecté et dans ce cas de connaître les valeurs des paramètres positionnés lors de son ouverture via OPEN.
Cette interrogation peut être faite en indiquant soit le numéro d'unité logique soit le nom du fichier.

Exemple

```
program inquire
  LOGICAL existe
  INTEGER ios
  CHARACTER(len=3)  :: form
  CHARACTER(len=10) :: acces

  INQUIRE( FILE="data_txt_seq", EXIST=existe )
```

Exemple (suite)

```
  if ( existe ) then
    OPEN( UNIT=1,             FILE="data_txt_seq", &
          POSITION="rewind", ACTION="read",       &
          IOSTAT=ios )
    if ( ios /= 0 ) then ! erreur à l'ouverture
       ...
    else
      INQUIRE( UNIT=1,          &
               FORMATTED=form, &
               ACCESS=acces )
    end if
       ...
    CLOSE( UNIT=1 )
  end if
end program inquire
```

Dans les variables caractères `form` et `acces` on récupère respectivement les valeurs `"YES"` et `"SEQUENTIAL"` (si le fichier `data_txt_seq` existe).

Formats BOZ

Les spécificateurs de format `Bw[.d]`, `Ow[.d]` et `Zw[.d]` permettent la conversion de données entières sous forme binaire, octale et hexadécimale respectivement.

Exemple

```
PROGRAM boz
  INTEGER I, J, K

  I =  1415; J =  1515; K =  1715
  WRITE( UNIT = 1, &
         FMT  = "(B32.32,'|',O11.11,'|',Z8.8)" ) I, J, K
  I = -1415; J = -1515; K = -1715
  WRITE( UNIT = 1, FMT="(B32.32)" ) I
  WRITE( UNIT = 1, FMT="(O11.11)" ) J
  WRITE( UNIT = 1, FMT="(Z8.8)"   ) K
END PROGRAM boz
```

Sorties
00000000000000000000010110000111|00000002753|000006B3
11111111111111111111101001111001
37777775025
FFFFF94D

Unités standards

Les fichiers associés au clavier et à l'écran d'une session interactive sont pré-connectés en général aux numéros d'unités logiques 5 et 6 respectivement : en lecture pour le premier, en écriture pour le second.
Dans un souci de portabilité, on préférera utiliser dans les instructions READ/WRITE le caractère « * » à la place du numéro de l'unité logique pour référencer l'*entrée standard* (READ) ou la *sortie standard* (WRITE). C'est la valeur par défaut du paramètre UNIT.
L'instruction PRINT remplace l'instruction WRITE dans le cas où celui-ci n'est pas précisé.

Exemple

```
  CHARACTER(LEN=8)       :: ft = "(F8.3)"
  READ( UNIT=5, FMT=ft) ...
  READ( UNIT=*, FMT=ft) ...
  READ ft, ...
       ...
  WRITE( UNIT=6, FMT=ft) ...
  WRITE( UNIT=*, FMT=ft) ...
  PRINT ft, ...
```

Instruction FORMAT

Le format d'édition peut être défini en dehors des instructions d'entrées-sorties READ/WRITE. Dans ce cas le paramètre FMT= est positionné à un numéro (étiquette) renvoyant à une instruction de définition de format (FORMAT).

Exemple

```
      REAL, DIMENSION(5,6) :: tab
      INTEGER n, i
      CHARACTER(len=10) :: ch
          ...
      PRINT "( I4,A,(T20,F8.3) )", n, ch, (tab(i,:),i=1,5)
      PRINT 100, n, ch, (tab(i,:),i=1,5)
100   FORMAT( I4,A,(T20,F8.3) )
```

En Fortran l'ouverture d'un fichier séquentiel est facultative. À défaut, l'ouverture du fichier est faite implicitement lors de l'exécution de la première instruction d'entrée-sortie. Le compilateur attribue au fichier un nom de la forme fort.i (i étant le numéro de l'unité logique indiqué dans l'instruction READ/WRITE). L'ouverture est faite en mode *formatted* ou *unformatted* suivant qu'un format a été ou non précisé.

Mots clés END=, ERR=

Le paramètre END de l'instruction READ offre un autre moyen de tester une fin de fichier pour un accès séquentiel. On lui indique le numéro (étiquette) de l'instruction à laquelle on désire poursuivre le traitement. Ce paramètre n'est pas valide dans le cas d'un fichier à accès direct.
De même, le paramètre ERR permet de se débrancher à une instruction dans le cas d'une erreur de lecture (problème de format pour un fichier formaté, enregistrement inexistant pour un fichier à accès direct ...).

Exemple

```
      INTEGER :: date
      DO
        PRINT *, "Saisie d'une date :"
        READ( *, "(i4)", END=1, ERR=2 ) date
        PRINT "(i4)", date
        CYCLE
2       PRINT *, "Saisie invalide. Veuillez recommencer."
      END DO
1     PRINT *, "Arrêt de la saisie."
```

Remarque : au clavier, la saisie du caractère Ctrl-D après le caractère *newline* (touche *Enter*) indique une fin de fichier.

Mots clés END=, ERR=

Lors d'une lecture, lorsqu'une erreur se produit (problème de format s'agissant d'un fichier texte, détection d'une fin de fichier ...) un traitement par défaut est activé par le compilateur, lequel traitement peut fournir des informations précieuses quant à la nature de l'erreur. L'utilisation du paramètre IOSTAT désactive ce traitement, ce qui peut être considéré comme un inconvénient. De ce fait, pour tester une fin de fichier il peut être préférable d'utiliser le mot-clé END. Celui-ci a l'avantage de ne pas masquer le traitement par défaut concernant les erreurs éventuelles.

Exemple

```
PROGRAM lecture
  INTEGER date

  OPEN( UNIT=1, FILE="data", ACTION="read" )
  DO
    READ( UNIT=1, FMT="(i4)", END=1 ) date
    PRINT *, date
  END DO
1 CLOSE( UNIT=1 )
END PROGRAM lecture
```

Mots clés END=, ERR=

Exemple

```
Contenu du fichier "data"
1685
1756
17a1

Sorties produites
 1685
 1756

"lecture.f90", line 11: 1525-097 A READ statement
 using decimal base input found the invalid digit
 'a' in the input file. The program will recover
 by assuming a zero in its place.
 1701
```

Mots clés END=, ERR=

Exemple

```
PROGRAM lecture
  INTEGER date, ios

  OPEN( UNIT=1, FILE="data", ACTION="read" )
  DO
    READ( UNIT=1, FMT="(i4)", IOSTAT=ios ) date
    if ( ios /= 0 ) exit
    PRINT *, date
  END DO
  CLOSE( UNIT=1 )
  if ( ios < 0 ) print *,"Fin de fichier atteinte"
  if ( ios > 0 ) print *,"Erreur de lecture détectée"
END PROGRAM lecture

Sorties produites
 1685
 1756
 Erreur de lecture détectée
```

1 Introduction

2 Généralités

3 Déclarations

4 Opérateurs et expressions

5 Structures de contrôle

6 Tableaux

7 Entrées-Sorties

8 Procédures
- Arguments
- Subroutines
- Fonctions
- Arguments de type chaîne de caractères
- Arguments de type tableau
- Arguments de type procédure
- Procédures internes
- Durée de vie et visibilité des identificateurs

Procédures intrinsèques

9 Common

10 Include

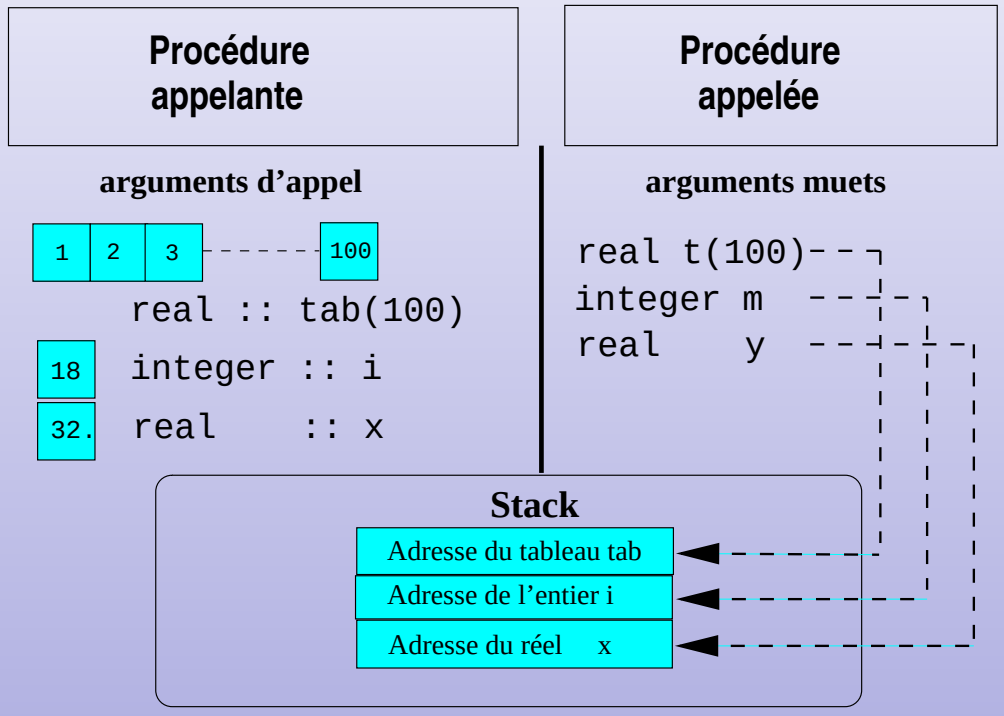
Arguments

Très souvent, dans un programme, on a besoin d'effectuer un même traitement plusieurs fois avec des valeurs différentes. La solution est de définir ce traitement une seule fois à l'aide d'une unité de programme de type procédure (`SUBROUTINE` ou `FUNCTION`).
Les unités de programmes désirant effectuer ce traitement feront appel à cette procédure en lui transmettant des valeurs via des variables appelées **arguments d'appel** (*actual-arguments*). La procédure appelée récupère les valeurs qu'on lui a transmises via des variables appelées **arguments muets** (*dummy-arguments*).
En Fortran le passage de ces valeurs s'effectue par **référence** ;

- les adresses des **arguments d'appel** sont transmises à la procédure appelée,
- dans la procédure appelée, les **arguments muets** sont des alias des **arguments d'appel**.

Schéma passage arguments

Figure 1: *Schéma passage arguments*



Subroutines

L'appel d'une procédure de type `SUBROUTINE` s'effectue à l'aide de l'instruction `CALL` suivie du nom de la procédure à appeler avec la liste des arguments d'appels entre parenthèses. Au sein de celle-ci l'instruction `return` permet de forcer le retour à la procédure appelante.

Exemple

```
REAL, DIMENSION(100) :: tab
REAL                 :: moyenne, maximum
    ...
CALL SP( tab, moyenne, maximum )
PRINT *,moyenne, maximum
END
SUBROUTINE SP( t, moy, max )
  REAL, DIMENSION(100) :: t
  REAL                 :: moy, max
  INTEGER              :: i
  max = t(1); moy = t(1)
  DO i=2,100
    IF ( t(i) > max ) max = t(i)
    moy = moy + t(i)
  END DO
  moy = moy/100
END SUBROUTINE SP
```

Fonctions

Un autre moyen de transmettre des valeurs à une unité de programme est l'utilisation d'une procédure de type `FUNCTION`.
À la différence d'une `SUBROUTINE`, une `FUNCTION` retourne une valeur ; celle-ci est donc typée. De plus, son appel s'effectue en indiquant uniquement son nom suivi entre parenthèses de la liste des arguments d'appels.
Au sein de la fonction l'instruction `return` sert à transmettre à la procédure appelante la valeur à retourner. Celle-ci n'est nécessaire que dans le cas où on désire effectuer ce retour avant la fin de la définition de la fonction.
Dans la procédure appelante l'expression correspondant à l'appel de la fonction est remplacée par la valeur retournée.

Fonctions

Exemple

```
REAL, DIMENSION(100) :: tab
REAL                 :: moyenne, maximum
REAL                 :: maxi
    ...
maximum = maxi( tab, moyenne )
PRINT *,moyenne, maximum
    ...
END
FUNCTION maxi( t, moy )
  REAL, DIMENSION(100) :: t
  REAL                 :: moy, maxi
  INTEGER              :: i
  maxi = t(1); moy = t(1)
  DO i=2,100
    IF ( t(i) > maxi ) maxi = t(i)
    moy = moy + t(i)
  END DO
  moy = moy/100
END FUNCTION maxi
```

Arguments de type chaîne de caractères

Lorsqu'une chaîne de caractères est transmise en argument, `Fortran` passe également sa longueur de façon implicite.
Dans la procédure appelée, celle-ci peut être récupérée à l'aide de la fonction intrinsèque `LEN`.
La déclaration de la chaîne de caractères au sein de la procédure appelée est faite en spécifiant le caractère « * » à la place de la longueur.
La procédure appelée fait alors référence à une chaîne de caractères à taille implicite (*assumed-size string*).

Arguments de type chaîne de caractères

Dans l'exemple ci-dessous les fonctions `IACHAR/ACHAR` permettent de mettre en relation un caractère et son rang dans la table des caractères `ASCII`.

Exemple

```
program arg_chaine
  implicit none
  character(len=10) :: ch

  read '(a)',ch
  call conv( ch )
  print *,ch
end program arg_chaine
subroutine conv( chaine )
  implicit none
  character(len=*) :: chaine
  integer i, j

  do i=1,len(chaine)
    if( iachar( chaine(i:i) ) < 97 .or. &
        iachar( chaine(i:i) ) > 122 ) cycle
    j = iachar( chaine(i:i) ) - 32
    chaine(i:i) = achar( j )
  end do
end subroutine conv
```

Arguments de type tableau

Lorsque l'on transmet un tableau en argument il est commode de transmettre également ses dimensions afin de pouvoir déclarer l'argument muet correspondant au sein de la procédure appelée à l'aide de celles-ci ; de ce fait le tableau est ajustable.

Exemple

```
program arg_tab
  implicit none
  integer, parameter    :: n = 3, m = 2
  real, dimension(n,m) :: tab
  real                  :: somme
  read *,tab; print *,somme( tab, n, m )
end program arg_tab
real function somme( t, n, m )
  integer                 :: n,m,i,j
  real, dimension(n,m) :: t
  somme = 0.
  do i=1,n
    do j=1,m
      somme = somme + t(i,j)
    end do
  end do
end function somme
```


Arguments de type tableau

Lorsqu'un tableau est passé en argument c'est l'adresse de son premier élément qui est transmise.
La procédure appelée doit posséder les informations lui permettant d'adresser les différents éléments de ce tableau.
De façon générale, supposons que l'on dispose d'un tableau `tab` à 2 dimensions constitué de `n` lignes et `m` colonnes. L'adresse de l'élément `tab(i,j)` est :

`@tab(i,j) = @tab(1,1) + [n×(j-1)+(i-1)]×taille(élément)`

Le nombre de colonnes `m` n'intervient pas dans ce calcul.
Souvent en `Fortran`, lors de l'appel d'une procédure seule la première dimension d'un tableau à 2 dimensions est transmise.
Dans la procédure appelée celui-ci est déclaré en indiquant le caractère « * » à la place de la deuxième dimension. On fait alors référence à un tableau à `taille implicite` (*assumed-size array*).
Dans un tel cas, il faut faire preuve d'une certaine prudence car dans la procédure appelée on ne maîtrise pas l'espace mémoire total occupé par le tableau.

Arguments de type tableau

Exemple

```
program taille_implicite
  implicit none
  integer, parameter    :: n = 5, m = 6
  real, dimension(n,m) :: tab
  real                  :: somme, som

  read *, tab
  somme = som( tab, n )
  print *,somme
end program taille_implicite
real function som( t, lda )
  implicit none
  integer                 :: lda
  real, dimension(lda,*) :: t
  integer                 :: i,j

  som = 0.
  do i=1,lda
    do j=1,lda
      som = som + t(i,j)
    end do
  end do
end function som
```

Arguments de type procédure

Une procédure peut être transmise à une autre procédure. Il est nécessaire de la déclarer dans la procédure appelante avec l'attribut `EXTERNAL` ou `INTRINSIC` si elle est intrinsèque.

Exemple

```
program arg_proc
  implicit none
  double precision b_inf, b_sup, aire
  double precision integrale
  integer pas
  double precision, external :: carre

  b_inf = 1.
  b_sup = 6.
  pas = 200000
  aire = integrale( b_inf, b_sup, pas, carre )
  print "('Aire : ', f11.6)", aire
end program arg_proc
```

Arguments de type procédure

Exemple

```
function integrale( borne_i, borne_s, pas, f )
  implicit none
  double precision borne_i, borne_s
  double precision integrale
  integer          pas, i
  double precision h, f

  h = (borne_s - borne_i)/pas
  integrale = 0.
  do i=0, pas-1
    integrale = integrale + h*f(borne_i+i*h)
  end do
end function integrale

function carre( x )
  implicit none
  double precision x
  double precision carre

  carre = x*x
end function carre
```


Procédures internes

En Fortran une procédure peut en contenir d'autres. Ces procédures sont appelées procédures internes. Elles ne peuvent être appelées que depuis la procédure les contenant.
Les définitions des procédures internes sont faites dans la procédure les incluant après l'instruction CONTAINS.
Il n'y a pas d'imbrications possibles : une procédure interne ne peut pas elle-même en contenir.

Exemple

```
program proc_interne
  implicit none
  character(len=10) :: ch

  read "(a)",ch
  call conversion( ch )
  print *,ch
end program proc_interne
```

Procédures internes

Exemple

```
subroutine conversion( chaine )
  implicit none
  character(len=*) :: chaine
  integer i

  do i=1,len(chaine)
    if( ichar( chaine(i:i) ) < 97 .or.
        ichar( chaine(i:i) ) > 122 ) cycle
    chaine(i:i) = car_majuscule( chaine(i:i) )
  end do
  CONTAINS
  function car_majuscule( c )
    character(len=1) :: c, car_majuscule
    integer          :: i

    i = ichar( c ) - (ichar("a") - ichar("A"))
    car_majuscule = achar( i )
  end function car_majuscule
end subroutine conversion
```


Procédures internes

Dans une procédure interne, toute variable déclarée dans l'unité de programme qui la contient est accessible, à moins qu'elle n'ait fait l'objet d'une redéclaration.

Exemple

```
program visibilite
  implicit none
  integer i, j
  i = 10; j = 20
  call sub
  print "('i =',i3,', j =',i3)", i, j
  contains
  subroutine sub
    integer j    ! masque le "j" de l'appelant
    j = 100; i = i + 1
  end subroutine sub
end program visibilite
```

Sortie
i^=^11,^j^=^20

Durée de vie et visibilité des identificateurs

On appelle durée de vie d'un identificateur le temps pendant lequel il existe en mémoire. Il est visible s'il existe en mémoire et est accessible, car il peut exister mais être masqué par un autre de même nom (c.f. procédure interne).

- Par défaut, une variable a une durée de vie limitée à celle de l'unité de programme dans laquelle elle a été définie,
- l'attribut `SAVE` permet de prolonger la durée de vie à celle de l'exécutable : on parle alors de variable permanente ou statique,
- dans une unité de programme l'instruction `SAVE` sans spécification de liste de variables indique que toutes les variables de cette unité sont permanentes,
- une compilation effectuée en mode *static* force la présence de l'instruction `SAVE` dans toutes les unités de programme, ce qui implique que toutes les variables sont permanentes,
- par contre si elle est faite en mode *stack*, les variables permanentes sont :
  - celles pour lesquelles l'attribut `SAVE` a été précisé,
  - celles initialisées à la déclaration (via l'instruction `DATA` ou à l'aide du signe « = »).

Durée de vie et visibilité des identificateurs

Exemple

```
program newton
  double precision :: valeur = 50.d0
  double precision :: tolerance
  double precision :: racine, x

  tolerance = 1.0d-1
  x = racine( valeur, tolerance )
  print "('Racine de ',f5.1,' = ', d16.8)", valeur, x
  tolerance = 1.0d-6
  x = racine( valeur, tolerance )
  print "('Racine de ',f5.1,' = ', d16.8)", valeur, x
end program newton
function racine( valeur, tol )
  double precision :: valeur, tol
  double precision :: racine
  double precision :: x = 1.0d0, x_prev
  integer          :: nb_iterations

  nb_iterations = 0
  do
    nb_iterations = nb_iterations + 1
    x_prev = x
    x = 0.5 * (x_prev + valeur/x_prev)
    if ( abs(x-x_prev)/x < tol ) exit
  end do
  print *,"Nombre d'itérations = ", nb_iterations
  racine = x
end function racine
```

Procédures intrinsèques

Le compilateur `Fortran` dispose d'une bibliothèque de procédures couvrant différents domaines : mathématique, conversion de type, manipulation de chaînes de caractères, comparaison de chaînes de caractères, ...
Pour une procédure donnée, le nom d'appel diffère suivant le type des arguments transmis. Un nom générique permet de s'affranchir de ces types : c'est la solution fortement conseillée car elle facilite la portabilité.
Par exemple un appel à la procédure générique `ABS`, retournant la valeur absolue de son argument, est transformé par le compilateur en un appel à la procédure :

- `IABS` pour un argument entier,
- `ABS` pour un argument réel simple précision,
- `DABS` pour un argument réel double précision,
- `CABS` pour un argument complexe.

Une liste des procédures intrinsèques est fournie en annexe `B`.

1 Introduction

2 Généralités

3 Déclarations

4 Opérateurs et expressions

5 Structures de contrôle

6 Tableaux

7 Entrées-Sorties

8 Procédures

9 Common
- L'instruction COMMON
- Common blanc
- Common étiqueté
  - Initialisation : BLOCK DATA
  - Instruction SAVE et COMMON

Règles et restrictions

10 Include

L'instruction COMMON

L'instruction COMMON permet le regroupement de zones mémoires pouvant être partagées par différentes unités de programme (fonctions, procédures).
La syntaxe d'une instruction COMMON est la suivante :

```
COMMON [/[nom_common]/] liste_variables
```

Le COMMON est dit étiqueté si nom_common est précisé. S'il n'a pas de nom on l'appelle COMMON blanc.
Les différentes zones regroupées au sein du bloc COMMON sont adressées via des variables dont les noms sont indiqués dans la partie liste_variables de l'instruction COMMON.
Pour qu'une unité de programme ait accès à ces zones, il est nécessaire qu'elle contienne l'instruction COMMON les référençant.

Common blanc

Lors de la déclaration d'un COMMON blanc nom_common est omis et les deux *slashes* "/" sont alors optionnels.
Ses particularités sont :

- un COMMON blanc est permanent (il hérite de l'attribut SAVE),
- les variables apparaissant dans un COMMON blanc ne peuvent pas être initialisées lors de leur déclaration. Ce type de bloc est initialement constitué de *bits* à 0. En conséquence :
  - les données numériques sont initialisées à 0,
  - les données logiques sont initialisées à la valeur .FALSE.,
  - les données de type chaîne de caractères représentent des chaînes vides.
- un COMMON blanc peut ne pas avoir la même taille d'une unité de programme à une autre, c'est alors la taille maximum qui sera retenue pour l'ensemble du programme.

Common blanc

Exemple

```
PROGRAM common_blanc
  INTEGER                 :: i
  INTEGER, DIMENSION(6)   :: itab
  LOGICAL, DIMENSION(3)   :: ltab
  COMMON                     itab,ltab

  DO i=1,6
     itab(i) = i
  END DO
  ltab(1) = .true.
  ltab(2) = .false.
  ltab(3) = .true.
  CALL sub
END PROGRAM common_blanc
!**********************************
SUBROUTINE sub
  INTEGER, DIMENSION(6) :: itab
  LOGICAL, DIMENSION(3) :: ltab
  COMMON                   itab,ltab

  PRINT*,"Tableau entier  = ", itab
  PRINT*,"Tableau logique = ", ltab
END SUBROUTINE sub
```

Common étiqueté

Un COMMON est dit étiqueté lorsque nom_common est précisé entre caractères /.
Ses particularités sont :

- il peut être initialisé par l'intermédiaire d'une unité de programme de type BLOCK DATA,
- un bloc COMMON défini dans le programme principal reçoit implicitement l'attribut SAVE. S'il ne l'est pas, l'attribut SAVE est nécessaire si on désire le rendre permanent.

Exemple

```
INTEGER, DIMENSION(6)  :: itab
REAL,    DIMENSION(12) :: rtab

COMMON /TAB/itab, rtab
```

Initialisation : BLOCK DATA

BLOCK DATA est une unité de programme qui permet d'initialiser des objets déclarés dans des COMMONs étiquetés :

```
BLOCK DATA [nom_block_data]
   bloc_init
END BLOCK DATA [nom_block_data]
```

- nom_block_data est le nom du BLOCK DATA
- bloc_init est une suite :
  - de déclarations de type (INTEGER, REAL, ...),
  - de déclarations de zones communes (COMMON),
  - d'initialisations statiques (DATA).

Un bloc COMMON ne peut apparaître que dans un seul BLOCK DATA.
On peut se définir plusieurs unités BLOCK DATA, chacune regroupant les COMMON qui ont un lien logique entre eux.

Initialisation : BLOCK DATA

Exemple

```
BLOCK DATA INIT
   REAL                      :: A, B, C, D
   REAL, DIMENSION(10,15)    :: MATRICE
   INTEGER, DIMENSION(20)    :: VECTEUR

   COMMON /BLOC1/ MATRICE, A, B
   COMMON /BLOC2/ VECTEUR, C, D

   DATA A /-1./, B /3.14/, C /0./, D /7.1/
   DATA MATRICE /150 * 50.0/, VECTEUR /20 * 4/

END BLOCK DATA INIT
```

Instruction SAVE et COMMON

Les valeurs des variables d'un COMMON étiqueté deviennent indéfinies quand une procédure se termine (retour à l'unité appelante) sauf s'il existe une autre unité de programme active le référençant.
Le cas échéant, on lui appliquera l'instruction SAVE pour conserver son contenu :

```
SAVE /nom_common/
```

- Un COMMON qui reçoit l'attribut SAVE dans une fonction ou procédure devra toujours être déclaré avec ce même attribut dans toutes les autres unités de programme,
- Il est inutile de spécifier l'attribut SAVE si le COMMON a été déclaré dans le programme principal.

Instruction SAVE et COMMON

Exemple

```
PROGRAM common_save
  call first
  call second
END PROGRAM common_save
!************************************
SUBROUTINE first
  REAL, DIMENSION(6)     :: rtab
  LOGICAL                :: drapeau
  COMMON/BLOC/ rtab,drapeau
  SAVE  /BLOC/

  CALL random_number(rtab)
  PRINT*,"Afficher le tableau (true/false)"
  READ(*,*) drapeau
END SUBROUTINE first
!************************************
SUBROUTINE second
  REAL, DIMENSION(6)     :: rtab
  LOGICAL                :: drapeau
  COMMON /BLOC/            rtab,drapeau
  SAVE   /BLOC/

  IF (drapeau) PRINT*,"Tableau de réels  = ", rtab
END SUBROUTINE second
```

Règles et restrictions

- Un COMMON ne peut pas contenir :
  - les noms de procédures (sous-programmes, fonctions),
  - les arguments de procédures,
  - les constantes symboliques (ayant l'attribut PARAMETER).
- une même variable ne peut pas apparaître dans deux COMMONs de noms différents,
- la taille d'un bloc COMMON étiqueté doit être la même dans chaque unité de programme le référençant,
- Fortran 90 permet désormais le mélange de données numériques et caractères au sein d'un bloc COMMON,
- d'une unité de programme à une autre, les variables de la liste peuvent porter des noms différents,
- au sein d'une même unité de programme, un bloc COMMON (étiqueté ou non) peut être référencé plusieurs fois : les différentes listes de variables sont alors ajoutées les unes aux autres.

Règles et restrictions

- un bloc COMMON peut être découpé différemment d'une unité de programme à une autre, c'est-à-dire référencé à l'aide de variables de types différents à condition que ce découpage soit cohérent (les zones numériques doivent correspondre à des zones numériques et de même pour les zones caractères),
- une instruction EQUIVALENCE ne peut pas :
  - associer des variables déclarées dans des blocs COMMON différents,
  - avoir pour effet de prolonger le COMMON à sa gauche. Dans l'exemple suivant, on ne peut pas associer la variable scalaire A au scalaire B(2) :

Exemple

```
REAL                 :: A
REAL, DIMENSION(2)   :: B

COMMON /X/A
EQUIVALENCE (A,B(2))    ! INVALIDE
EQUIVALENCE (A,B(1))    ! VALIDE
```

Règles et restrictions

Exemple

```
program common_mixte
  complex, dimension(10) :: c
  character(len=100)     :: chaine
  COMMON/MIXTE/c, chaine
     . . .
  call sub
end program common_mixte
subroutine sub
  real, dimension(20)              :: tab_reels
  character(len=1), dimension(100) :: tab_car
  COMMON/MIXTE/tab_reels, tab_car
     . . .
  ! impression des parties réelles
  print *,tab_reels(1:20:2)
  print *,tab_car(1), tab_car(10)
end subroutine sub
```

Recommandations :

1. il est préférable de déclarer un bloc COMMON de la même façon dans les différentes unités de programme et donc d'employer la technique de l'exemple précédent avec modération,
2. l'instruction INCLUDE ci-après favorise le bon emploi des blocs COMMON.

1 Introduction
2 Généralités
3 Déclarations
4 Opérateurs et expressions
5 Structures de contrôle
6 Tableaux
7 Entrées-Sorties
8 Procédures
9 Common
10 Include
La directive INCLUDE

La directive INCLUDE

Elle demande au compilateur d'inclure un fichier contenant des instructions Fortran :

```
INCLUDE 'nom_de_fichier'
```

Exemple

```
PROGRAM inclusion
  IMPLICIT NONE
  INTEGER :: i
  INCLUDE "inclusion.inc"

  DO i=1,6
     rtab(i) = PI
  END DO
  ltab(1) = .true.
  ltab(2) = .false.
  ltab(3) = .true.
  CALL sub
END PROGRAM inclusion
```

La directive INCLUDE

Exemple

```
SUBROUTINE sub
  IMPLICIT NONE
  INCLUDE "inclusion.inc"

  PRINT*,"Tableau réels   = ", rtab
  PRINT*,"Tableau logique = ", ltab
END SUBROUTINE sub
```

Le fichier `inclusion.inc` contient les déclarations de COMMONs et de paramètres :

Exemple (suite)

```
  DOUBLE PRECISION, PARAMETER     :: PI=3.14159265d0
  DOUBLE PRECISION, DIMENSION(6) :: rtab
  LOGICAL, DIMENSION(3)           :: ltab
  COMMON rtab,ltab
```

Remarques :

- L'inclusion du contenu du fichier se fait là où est placée la directive INCLUDE,
- Il est possible d'imbriquer plusieurs directives INCLUDE. Le nombre d'imbrications possible dépend du compilateur.
- La localisation du fichier à inclure peut être précisée à l'aide de l'option -I du système de compilation.

11 Annexe A : entrées-sorties - syntaxes

12 Annexe B : procédures intrinsèques

13 Annexe C : aspects obsolètes

14 Annexe D : système de compilation

15 Annexe E : exercices

Instruction OPEN

Note : les valeurs par défaut sont soulignées.

```
OPEN( [UNIT=]u,          &      ! entier
      IOSTAT=ios,        &      ! entier
      FILE=filename,     &      ! chaîne de caractères
      STATUS=st,         &      ! chaîne de caractères
      ACCESS=acc,        &      ! chaîne de caractères
      FORM=mode,         &      ! chaîne de caractères
      RECL=reclen        &      ! entier
      POSITION=pos,      &      ! chaîne de caractères
      ACTION=act,        &      ! chaîne de caractères
      BLANK=blk,         &      ! chaîne de caractères
      DELIM=del,         &      ! chaîne de caractères
      PAD=padding )             ! chaîne de caractères
```

- UNIT : numéro de l'unité logique sur laquelle on désire connecter le fichier,
- IOSTAT : entier valorisé une fois l'ouverture effectuée. Il est nul si l'opération s'est bien passée, non nul sinon,
- FILE : nom du fichier à ouvrir,

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

Instruction OPEN

- STATUS : état du fichier avant l'ouverture,
  - OLD : le fichier doit exister,
  - NEW : le fichier ne doit pas exister, il sera créé,
  - UNKNOWN : état dépendant du compilateur,
  - REPLACE : si le fichier n'existe pas, il est créé et hérite de l'état "OLD", sinon, il est détruit et un nouveau fichier est créé,
  - SCRATCH : un fichier temporaire anonyme est créé, il sera détruit à la fin du programme ou au moment du CLOSE.
- ACCESS : mode d'accès au fichier,
  - DIRECT : accès direct,
  - SEQUENTIAL : accès séquentiel.

Instruction OPEN

- FORM : type du transfert,
  - FORMATTED : mode caractère (avec format),
  - UNFORMATTED : mode binaire (sans format).
- RECL : longueur des enregistrements logiques en accès direct ou de l'enregistrement maximum en accès séquentiel. L'unité de longueur est le caractère pour les fichiers textes et dépend du processeur pour les fichiers binaires,
- POSITION : positionnement dans le fichier après ouverture,
  - ASIS : positionnement inchangé si le fichier est déjà connecté, indéfini sinon,
  - REWIND : positionnement en tête de fichier,
  - APPEND : positionnement en fin de fichier pour extension,
- ACTION : mode d'ouverture.
  - READ : seule la lecture est possible,
  - WRITE : seule l'écriture est possible,
  - READWRITE : la lecture ainsi que l'écriture sont permises,

Instruction OPEN

- BLANK : indique la façon dont les blancs sont interprétés lors d'une opération d'entrée-sortie,
  - NULL : ils sont ignorés,
  - ZERO : ils sont interprétés comme des zéros,
- DELIM : indique le caractère délimiteur utilisé pour les constantes chaînes de caractères écrites en format libre ou via une NAMELIST,
  - APOSTROPHE : délimiteur → ",
  - QUOTE : délimiteur → ',
  - NONE : aucun délimiteur,
- PAD : permet de gérer le *padding* lors d'une lecture,
  - YES : la liste de variables spécifiée dans l'instruction de lecture peut être plus grande que l'enregistrement logique, les variables non valorisées lors de la lecture sont remplies par des zéros pour les variables numériques ou logiques, par des blancs pour les variables de type chaîne de caractères,
  - NO : aucun *padding* n'est autorisé. La taille de l'enregistrement logique doit être suffisante pour permettre la valorisation des variables de la liste.

Instruction READ

```
READ( [UNIT=]u,          &      ! entier
      [FMT=]label|ch,    &      ! label ou bien chaîne de caractères
      [NML=]nom,         &      ! chaîne de caractères
      ADVANCE=ad,        &      ! chaîne de caractères
      END=label,         &      ! label
      EOR=label,         &      ! label
      ERR=label,         &      ! label
      IOSTAT=st,         &      ! entier
      REC=n,             &      ! entier
      SIZE=n )                  ! entier
```

- UNIT : numéro de l'unité logique sur laquelle le fichier est connecté,
- FMT : format de conversion pour les fichiers textes spécifié sous forme :
  - d'une constante chaîne de caractères,
  - d'une variable chaîne de caractères,
  - d'une étiquette d'instruction FORMAT.
- NML : nom d'une *namelist*,

Instruction READ

- ADVANCE : positionnement à partir duquel s'effectue l'entrée-sortie suivante :
  - YES → enregistrement suivant,
  - NO → suite de l'enregistrement courant,
- END : étiquette de l'instruction à exécuter en cas de fin de fichier,
- EOR : étiquette de l'instruction à exécuter en cas de fin d'enregistrement,
- ERR : étiquette de l'instruction à exécuter en cas d'erreur,
- IOSTAT : entier valorisé une fois l'entrée-sortie effectuée. Il est nul si l'opération s'est bien passée. Une valeur positive indique une erreur. Une valeur négative signale une fin de fichier dans le cas d'un accès séquentiel et l'absence d'enregistrement du rang spécifié si l'accès est direct,
- REC : numéro de l'enregistrement à traiter pour un fichier à accès direct,
- SIZE : entier récupérant le nombre de caractères traités si la fin d'enregistrement a été atteinte.

Instruction WRITE

```
WRITE( [UNIT=]u,          &      ! entier
       [FMT=]label|ch,    &      ! label ou bien chaîne de caractères
       [NML=]nom,         &      ! chaîne de caractères
       ADVANCE=ad,        &      ! chaîne de caractères
       ERR=label,         &      ! label
       IOSTAT=st,         &      ! entier
       REC=n )                   ! entier
```

- UNIT : numéro de l'unité logique sur laquelle le fichier est connecté,
- FMT : format de conversion pour les fichiers textes spécifié sous forme :
  - d'une constante chaîne de caractères,
  - d'une variable chaîne de caractères,
  - d'une étiquette d'instruction FORMAT.
- NML : nom d'une *namelist*,

Instruction WRITE

- ADVANCE : positionnement à partir duquel s'effectue l'entrée-sortie suivante :
  - YES → enregistrement suivant,
  - NO → suite de l'enregistrement courant,
- ERR : étiquette de l'instruction à exécuter en cas d'erreur,
- IOSTAT : entier valorisé une fois l'entrée-sortie effectuée. Il est nul si l'opération s'est bien passée, non nul sinon,
- REC : numéro de l'enregistrement à traiter pour un fichier à accès direct.

Instruction INQUIRE

```
INQUIRE( [UNIT=]u,          &    ! entier
         ACCESS=acc,        &    ! chaîne de caractères
         ACTION=act,        &    ! chaîne de caractères
         BLANK=bl,          &    ! chaîne de caractères
         DELIM=del,         &    ! chaîne de caractères
         DIRECT=dir,        &    ! chaîne de caractères
         ERR=label,         &    ! label
         EXIST=ex,          &    ! logique
         FILE=file,         &    ! chaîne de caractères
         FORM=form,         &    ! chaîne de caractères
         FORMATTED=form,    &    ! chaîne de caractères
         IOSTAT=ios,        &    ! entier
         NAME=name,         &    ! chaîne de caractères
         NAMED=named,       &    ! logique
         NEXTREC=next,      &    ! entier
         NUMBER=num,        &    ! entier
         OPENED=op,         &    ! logique
         PAD=pad,           &    ! chaîne de caractères
         POSITION=pos,      &    ! entier
         READ=r,            &    ! chaîne de caractères
         READWRITE=rw,      &    ! chaîne de caractères
         RECL=n,            &    ! entier
         SEQUENTIAL=seq,    &    ! chaîne de caractères
         UNFORMATTED=un,    &    ! chaîne de caractères
         WRITE=wr )              ! chaîne de caractères
```


Instruction INQUIRE

- UNIT : numéro de l'unité logique sur laquelle le fichier est connecté,
- ACCESS : méthode d'accès
  - SEQUENTIAL si fichier connecté pour un accès séquentiel,
  - DIRECT si fichier connecté pour un accès direct,
  - UNDEFINED si fichier non connecté,
- ACTION : type d'accès
  - READ si fichier connecté en lecture,
  - WRITE si fichier connecté en écriture,
  - READWRITE si fichier connecté en lecture/écriture,
  - UNDEFINED si fichier non connecté,
- BLANK :
  - NULL si les blancs sont ignorés,
  - ZERO si les blancs sont interprétés comme des 0,
  - UNDEFINED si le fichier n'est pas connecté en mode formaté ou bien n'est pas connecté du tout,

Instruction INQUIRE

- DELIM : délimiteur pour les chaînes de caractères en sortie en format libre ou via *namelist*
  - APOSTROPHE délimiteur → ",
  - QUOTE délimiteur → ',
  - UNDEFINED si le fichier n'est pas connecté en mode formaté ou bien n'est pas connecté du tout,
- DIRECT : accès direct
  - YES l'accès direct est un mode d'accès permis,
  - NO l'accès direct n'est pas un mode d'accès permis,
  - UNKNOWN le compilateur ne sait pas si l'accès direct est permis ou non,
- ERR : étiquette de l'instruction à exécuter en cas d'erreur,
- EXIST :
  - .true. si le fichier ou l'unité logique existe,
  - .false. si le fichier ou l'unité logique n'existe pas,
- FILE : nom du fichier pour lequel on souhaite avoir des informations,

Instruction INQUIRE

- FORM : type d'accès
  - FORMATTED si le fichier est connecté en mode formaté,
  - UNFORMATTED si le fichier est connecté en mode binaire,
  - UNDEFINED si le fichier n'est pas connecté,
- FORMATTED :
  - YES si le traitement du fichier en mode formaté est valide
  - NO si le traitement du fichier en mode formaté n'est pas valide,
  - UNKNOWN lorsque le compilateur ne sait pas si le traitement du fichier en mode formaté est permis ou non,
- IOSTAT : valeur de retour
  - >0 si une erreur s'est produite,
  - =0 si aucune erreur ne s'est produite,
- NAME : nom du fichier connecté s'il a un nom, sinon valeur indéfinie,

Instruction INQUIRE

- NAMED :
  - .true. si le fichier a un nom,
  - .false. si le fichier est anonyme,
- NEXTREC :
  - renvoie le numéro du prochain enregistrement à traiter si le fichier est ouvert en accès direct (1 si aucun enregistrement n'a déjà été traité,
  - renvoie une valeur indéfinie si le fichier n'a pas été ouvert en accès direct,
- NUMBER : renvoie le numéro de l'unité logique sur laquelle le fichier est connecté (-1 s'il ne l'est pas),
- OPENED :
  - .true. si le fichier est ouvert,
  - .false. si le fichier n'est pas ouvert,
- PAD :
  - NO le fichier a été ouvert avec le paramètre PAD="NO",
  - YES le fichier a été ouvert avec le paramètre PAD="YES" ou bien le fichier n'est pas connecté,

Instruction INQUIRE

- POSITION :
  - REWIND si fichier ouvert avec un positionnement en tête,
  - APPEND si fichier ouvert avec un positionnement en fin,
  - ASIS si fichier ouvert sans changement de la position,
  - UNDEFINED si fichier non connecté ou bien connecté en accès direct,
- READ :
  - YES un accès en lecture est permis,
  - NO un accès en lecture n'est pas permis,
  - UNKNOWN le compilateur ne sait pas si un accès en lecture est permis ou non,
- READWRITE :
  - YES un accès en lecture/écriture est permis,
  - NO un accès en lecture/écriture n'est pas permis,
  - UNKNOWN le compilateur ne sait pas si un accès en lecture/écriture est permis ou non,
- RECL : renvoie la taille de l'enregistrement logique maximum ou une valeur indéfinie si le fichier n'existe pas,

Instruction INQUIRE

- SEQUENTIAL :
  - YES l'accès séquentiel est un mode d'accès permis,
  - NO l'accès séquentiel n'est pas un mode d'accès permis,
  - UNKNOWN le compilateur ne sait pas si l'accès séquentiel est permis ou non,
- UNFORMATTED :
  - YES si le traitement du fichier en mode binaire est valide
  - NO si le traitement du fichier en mode binaire n'est pas valide,
  - UNKNOWN lorsque le compilateur ne sait pas si le traitement du fichier en mode binaire est permis ou non,
- WRITE :
  - YES un accès en écriture est permis,
  - NO un accès en écriture n'est pas permis,
  - UNKNOWN le compilateur ne sait pas si un accès en écriture est permis ou non.

Instruction CLOSE

```
CLOSE( [UNIT=]u,       &
       IOSTAT=ios,     &
       STATUS=st )
```

- UNIT : numéro de l'unité logique du fichier à fermer,
- IOSTAT : entier valorisé une fois la fermeture effectuée. Il est nul si l'opération s'est bien passée, non nul sinon,
- STATUS : état du fichier après fermeture,
  - DELETE : le fichier est détruit après fermeture. C'est la valeur par défaut si l'ouverture a été faite avec le paramètre STATUS="scratch",
  - KEEP : le fichier est conservé après fermeture. C'est la valeur par défaut si l'ouverture a été faite avec le paramètre STATUS différent de "scratch".

Exemple

```
integer ios
   ...
OPEN( UNIT=1,               &
      FILE="donnees",       &
      FORM="unformatted",   &
      ACTION="read",        &
      POSITION="rewind",    &
      IOSTAT=ios )
if ( ios /= 0 ) ! Problème à l'ouverture
   ...
CLOSE( UNIT=1 )

OPEN( UNIT=2,               &
      FORM="formatted",     &
      ACCESS="direct",      &
      STATUS="scratch",     &
      ACTION="write",       &
      LRECL=9600,           &
      IOSTAT=ios )
if ( ios /= 0 ) ! Problème à l'ouverture
   ...
CLOSE( UNIT=2 )
```


11 Annexe A : entrées-sorties - syntaxes

12 Annexe B : procédures intrinsèques

13 Annexe C : aspects obsolètes

14 Annexe D : système de compilation

15 Annexe E : exercices

Principales procédures intrinsèques

- `ABS` : retourne la valeur absolue de son argument. Pour un complexe, retourne sa norme : $\sqrt{x^2 + y^2}$.
  `ABS(-1) = 1; ABS(-1.5) = 1.5; ABS((3.,4.)) = 5.0`
- `ACHAR` : retourne le caractère de la table `ASCII` dont le rang est transmis en argument.
  `ACHAR(88) = 'X'; ACHAR(42) = '*'`
- `ACOS` : retourne l'arc cosinus en radians de son argument réel.
  `ACOS(0.54030231) = 1.0`
- `ADJUSTL` : cadre à gauche la chaîne passée en argument : supprime les blancs en tête ; complète à droite par des blancs.
  `ADJUSTL('^^Fortran') = 'Fortran^^'`
- `ADJUSTR` : cadre à droite la chaîne passée en argument : supprime les blancs en fin ; complète à gauche par des blancs.
  `ADJUSTR('Fortran^^') = '^^Fortran'`

Principales procédures intrinsèques

- `AIMAG` : retourne la partie imaginaire du complexe passé en argument.
  `AIMAG((2.,3.)) == 3.0`
- `AINT` : tronque le réel passé en argument.
  `AINT(2.783) = 2.0; AINT(-2.783) = -2.0`
- `ANINT` : retourne, sous forme d'un réel, l'entier le plus proche du réel transmis.
  `ANINT(2.783) = 3.0; ANINT(-2.783) = -3.0`
- `ASIN` : retourne l'arc sinus en radians de son argument réel.
  `ASIN(0.84147098) = 1.0`
- `ATAN` : retourne l'arc tangente en radians de son argument réel.
  `ATAN(1.5574077) = 1.0`
- `BIT_SIZE` : retourne le nombre de bits utilisés pour la représentation de l'entier passé en argument.
  `BIT_SIZE(1) = 32`

Principales procédures intrinsèques

- BTEST : permet de tester la valeur d'un bit d'un entier : l'entier ainsi que la position du bit à tester sont passés en argument.
  `BTEST(8,3) = .true.; BTEST(10,2) = .false.`
- CEILING : retourne l'entier immédiatement supérieur au réel transmis en argument.
  `CEILING(3.7) = 4, CEILING(-3.7) = -3`
- CMPLX : retourne un complexe dont les parties réelle et imaginaire sont transmises en argument.
  `CMPLX(-3.) = -3.0+0.i; CMPLX(2,4.) = 2.0+4.0i`
- CONJG : retourne le complexe conjugué de celui passé en argument.
  `CONJG((-3.0,4.0)) = -3.0-4.0i`
- COS : retourne le cosinus de l'angle passé en argument (exprimé en radians).
  `COS(1.0) = 0.54030231`
- COSH : retourne le cosinus hyperbolique.
  `COSH(1.0) = 1.5430806`

Principales procédures intrinsèques

- DBLE : convertit en double précision l'argument transmis.
- EXP : retourne l'exponentiel de l'argument transmis.
  `EXP(1.0) = 2.7182818`
- FLOOR : retourne l'entier immédiatement inférieur au réel transmis en argument.
  `FLOOR(3.7) = 3, FLOOR(-3.7) = -4`
- IACHAR : retourne le rang dans la table ASCII du caractère transmis en argument.
  `IACHAR('X') = 88; IACHAR('*') = 42`
- IAND : retourne l'entier dont la représentation binaire est obtenue en combinant à l'aide d'un "et logique" les bits des deux entiers transmis en argument.
  `IAND(1,3) = 1; IAND(10,10) = 10`
- IBCLR : permet de forcer à zéro un bit d'un entier : l'entier ainsi que la position du bit à forcer sont passés en argument.
  `IBCLR(14,1) = 12`
  `IBCLR((/ 1,2,3,4 /), 31) = (/ 29,27,23,15 /)`

Principales procédures intrinsèques

- IBITS : permet l'extraction d'une séquence de bits d'un entier. L'entier suivi de la position ainsi que la longueur pour effectuer l'extraction sont passés en argument.
  `IBITS(14,1,3) = 7`
- IBSET : permet de forcer à 1 un bit d'un entier : l'entier ainsi que la position du bit à forcer sont passés en argument.
  `IBSET(12,1) = 14`
  `IBSET((/ 1,2,3,4 /), 0) = (/ 1,3,3,5 /)`
- IEOR : retourne l'entier dont la représentation binaire est obtenue en combinant à l'aide d'un "ou exclusif" les bits des deux entiers transmis en argument.
  `IEOR(1,3) = 2; IEOR(10,10) = 0`
- INDEX : retourne la position d'une sous-chaîne dans une chaîne. La chaîne suivie de la sous-chaîne et du sens de la recherche sont fournis en argument.
  `INDEX('FORTRAN','R') = 3`
  `INDEX('FORTRAN','R',BACK=.true.) = 5`

Principales procédures intrinsèques

- INT : convertit en entier l'argument transmis.
  `INT(-3.7) = -3; INT(9.1/4.0) = 2`
- IOR : retourne l'entier dont la représentation binaire est obtenue en combinant à l'aide d'un "ou logique" les bits des deux entiers transmis en argument.
  `IOR(1,3) = 3`
  `IOR((/ 3,2 /),(/ 1,10 /)) = (/ 3,10 /)`
- ISHFT : permet d'effectuer un décalage des bits de l'entier passé en premier argument. Le deuxième argument indique le nombre de bits à décaler : son signe indique le sens du décalage (positif = gauche, négatif = droite). Les bits sortants sont perdus, les positions vacantes sont mises à zéro.
  `ISHFT(3,1) = 6; ISHFT(3,-1) = 1`
- ISHFTC : idem ISHFT à la différence que le décalage est circulaire et s'effectue sur les n bits de droite de l'entier , n étant fourni en troisième argument (s'il est absent il est considéré égal au nombre de bits de l'entier).
  `ISHFT(3,2,3) = 5; ISHFT(3,-2) = -1073741824`

Principales procédures intrinsèques

- LEN : retourne la longueur de la chaîne de caractères transmise en argument.
  `CHARACTER(len=10) CH; LEN(CH) = 10`
- LEN_TRIM : retourne la longueur de la chaîne de caractères transmise en argument sans considérer les blancs de fin.
  `LEN_TRIM('^^FORTRAN^^^') = 9; LEN_TRIM('^^^') = 0`
- LGE : compare les deux chaînes de caractères transmises en argument : retourne `.true.` si la première chaîne est supérieure ou égale à la deuxième, `.false.` sinon.
  `LGE('MANET','MONET') = .false.`
  `LGE('MANET Edouard','MANET') = .true.`
- LGT : compare les deux chaînes de caractères transmises en argument : retourne `.true.` si la première chaîne est supérieure strictement à la deuxième, `.false.` sinon.
  `LGT('MANET','MANET') = .false.`
- LLE : compare les deux chaînes de caractères transmises en argument : retourne `.true.` si la première chaîne est inférieure ou égale à la deuxième, `.false.` sinon.
  `LLE('MANET','MONET') = .true.`
  `LLE('MANET','MANET') = .true.`
- LLT : compare les deux chaînes de caractères transmises en argument : retourne `.true.` si la première chaîne est inférieure strictement à la deuxième, `.false.` sinon.
  `LLT('MANET','MANET') = .false.`

Principales procédures intrinsèques

- LOG : retourne le logarithme népérien de l'argument transmis.
  `LOG(2.7182818) = 1.0; LOG(10.0) = 2.3025851`
- LOG10 : retourne le logarithme décimal de l'argument transmis.
  `LOG10(10.0) = 1.0; LOG10(10.E10) = 11.0`
- MAX : retourne le maximum des nombres passés en argument.
  `MAX(-9.0,7.0,2.0) = 7.0`
- MIN : retourne le minimum des nombres passés en argument.
  `MIN(-9.0,7.0,2.0) = -9.0`
- MOD : retourne le reste de la division effectuée à l'aide des deux arguments fournis.
  `MOD(3.0,2.0) = 1.0; MOD(-8,5) = -3`
- NOT : retourne l'entier dont la représentation binaire est obtenue en inversant les bits de l'entier transmis en argument.
  `NOT(10) = -11`
- REAL : convertit en réel l'argument transmis.
  `REAL(3) = 3.0`
- REPEAT : permet de concaténer `n` fois une chaîne de caractères.
  `REPEAT('A',10) = 'AAAAAAAAAA'`
- SCAN : retourne la position du premier caractère d'une chaîne figurant parmi un ensemble de caractères donné. La recherche peut être faite dans les deux sens.
  `SCAN('RENOIR','OI') = 4`
  `SCAN('RENOIR','OI',BACK=.true.) = 5`

Principales procédures intrinsèques

- `SIGN` : retourne le nombre dont la valeur absolue est celle du premier argument et le signe celui du deuxième.
  `SIGN(-3.0,2.0) = 3.0`
- `SIN` : retourne le sinus de l'angle passé en argument (exprimé en radians).
  `SIN(1.0) = 0.84147098`
- `SINH` : retourne le sinus hyperbolique.
  `SINH(1.0) = 1.1752012`
- `SQRT` : retourne la racine carré de son argument.
  `SQRT(5.0) = 2.236068010`
- `TAN` : retourne la tangente de l'angle passé en argument (exprimé en radians).
  `TAN(1.0) = 1.5574077`
- `TANH` : retourne la tangente hyperbolique.
  `TANH(1.0) = 0.76159416`
- `TRIM` : retourne la chaîne de caractères transmise débarrassée de ses blancs de fin.
  `TRIM('PICASSO^^^^^') = 'PICASSO'`
- `VERIFY` : retourne la position du premier caractère d'une chaîne ne figurant pas parmi un ensemble de caractères donné. La recherche peut être faite dans les deux sens.
  `VERIFY('RENOIR','OI') = 1`
  `VERIFY('RENOIR','OI',BACK=.true.) = 6`

11 Annexe A : entrées-sorties - syntaxes

12 Annexe B : procédures intrinsèques

13 Annexe C : aspects obsolètes

14 Annexe D : système de compilation

15 Annexe E : exercices

Aspects obsolètes I

1. IF arithmétique : `IF (ITEST) 10,11,12`
   ==> `IF--THEN--ELSE IF--ELSE--ENDIF`
2. Branchement au `END IF` depuis l'extérieur `(*)`
   ==> se brancher à l'instruction suivante.
3. Boucles `DO` pilotées par réels : `DO 10 R=1., 5.7, 1.3 (*)`
4. Partage d'une instruction de fin de boucle :

```
      DO 1 I=1,N
        DO 1 J=1,N
          A(I,J)=A(I,J)+C(J,I)
    1 CONTINUE
```

   ==> autant de `CONTINUE` que de boucles.
5. Fins de boucles autres que `CONTINUE` ou `END DO`

Aspects obsolètes II

6. `ASSIGN` et le `GO TO` assigné : `(*)`

```
      ASSIGN 10 TO intvar
      ....
      ASSIGN 20 TO intvar
      ....
      GO TO intvar
```

   ==> `SELECT CASE` ou `IF/THEN/ELSE` `(*)` : aspects obsolètes déclarés hors norme par Fortran 95
7. `ASSIGN` d'une étiquette de `FORMAT` : `(*)`

```
  ASSIGN 2 TO NF              CHARACTER(7),DIMENSION(4)::C
2 FORMAT (F9.2)       ==>     I = 2; C(2) = '(F9.2)'
  PRINT NF,TRUC               PRINT C(I),TRUC
```

8. `RETURN` multiples :

Aspects obsolètes III

```
      CALL SP1(X,Y,*10,*20)
           ...
10         ...
           ...
20         ...
           ...
      SUBROUTINE SP1(X1,Y1,*,*)
           ...
      RETURN 1
           ...
      RETURN 2
           ...
```

==> SELECT CASE sur la valeur d'un argument retourné

9. PAUSE 'Montez la bande 102423 SVP' (*)
   ==> READ qui attend les données
10. FORMAT(9H A éviter) (*)
    ==> Constante littérale : FORMAT(' Recommandé')

(*) : aspects obsolètes déclarés hors norme par Fortran 95

Aspects obsolètes Fortran 95

1. Le "format fixe" du source
   ==> "format libre".
2. Le GO TO calculé (GO TO (10,11,12,...), *int_expr*)
   ==> select case.
3. L'instruction DATA placée **au sein** des instructions exécutables
   ==> **avant** les instructions exécutables.
4. *Statement functions* (sin_deg(x)=sin(x*3.14/180.))
   ==> procédures internes.
5. Le type CHARACTER*... dans les déclarations
   ==> CHARACTER(LEN=...)
6. Le type CHARACTER(LEN=*) de longueur implicite en retour d'une fonction
   ==> CHARACTER(LEN=len(str)).

11 Annexe A : entrées-sorties - syntaxes

12 Annexe B : procédures intrinsèques

13 Annexe C : aspects obsolètes

14 Annexe D : système de compilation

15 Annexe E : exercices

Système de compilation

La commande f90 permet de générer un exécutable à partir de fichiers sources `Fortran`.
Celle-ci appelle un système de compilation faisant successivement appel à :

- un préprocesseur,
- un compilateur,
- un *loader* ou éditeur de liens.

La composante préprocesseur transforme le source `Fortran` en entrée au moyen de directives.
La composante compilateur analyse le source `Fortran` fourni (éventuellement transformé à l'étape précédente) avec :

- détection des erreurs de syntaxe,
- traduction du source en langage machine plus ou moins optimisé,
- production d'un module objet.

Enfin la dernière composante fait appel au *loader* qui récupère les modules objets précédemment créés et les regroupe pour produire un module exécutable.
Les différentes unités de programme constituant une application `Fortran` peuvent figurer dans un même fichier ou bien être réparties dans plusieurs fichiers. Ceux-ci doivent être suffixés par .f ou .f90 .

Système de compilation

Le compilateur suppose que le source est écrit avec le format fixe si le fichier est suffixé par .f et avec le format libre si le fichier est suffixé par .f90. Ce format d'écriture peut être explicité au moyen d'une option indépendamment du suffixe employé.
Les fichiers correspondant aux modules objets sont suffixés par .o.
Par défaut le module exécutable est stocké dans un fichier de nom `a.out` qu'il est possible de renommer à l'aide de l'option -o nom_exécutable_désiré.
L'option -c permet de conserver le ou les modules objets produits par le compilateur et d'inhiber l'étape du *loader*.
C'est la technique employée par l'utilitaire `make` qui, automatiquement, lance les compilations des différents fichiers source constituant l'application. Les modules objets obtenus sont *in fine* fournis au système de compilation pour la production du module exécutable.

Exemple

```
$ f90 -c source1.f90
$ f90 -c source2.f90
      ...
$ f90 -c sourcen.f90
$ f90 *.o -o a.exe
```

Système de compilation

Il est possible d'archiver les fichiers *.o à l'aide de l'utilitaire ar dans le but de les regrouper dans un seul fichier dont le nom est de la forme libxxx.a.
Cela permet la constitution de bibliothèques de modules objets lesquelles sont transmises à la composante *loader* à l'aide des options -L et -l permettant de spécifier l'endroit où celles-ci sont stockées et leur noms.

Exemple

```
$ f90 -c source1.f90
$ f90 -c source2.f90
      ...
$ f90 -c sourcen.f90
$ ar -rv libexemple.a *.o
$ mv libexemple.a $HOME/lib
$ f90 -L$HOME/lib -lexemple -o a.exe
```

11 Annexe A : entrées-sorties - syntaxes

12 Annexe B : procédures intrinsèques

13 Annexe C : aspects obsolètes

14 Annexe D : système de compilation

15 Annexe E : exercices

Énoncés

Exercice 1

Écrire un programme permettant de résoudre le système de 2 équations à 2 inconnues :

$$\begin{cases} u_1 x + v_1 y = w_1 \\ u_2 x + v_2 y = w_2 \end{cases}$$

On pourra imprimer les solutions à l'aide de l'instruction : `PRINT *, 'X = ', X, ', Y = ', Y`

Exercice 2

Écrire un programme permettant de calculer les racines du trinôme du 2^{nd} degré : $ax^2 + bx + c$. On s'assurera que a est non nul. Les racines, si elles existent, pourront être imprimées à l'aide de l'instruction :
`PRINT *, 'X1 = ', X1, ', X2 = ', X2`

Exercice 3

Écrire un programme calculant le nombre d'Or. Celui-ci peut être obtenu à partir de la suite de Fibonnacci u_n définie par :

$$\begin{aligned} u_0 &= 1 \\ u_1 &= 1 \\ &\dots \\ u_{n+1} &= u_n + u_{n-1} \end{aligned}$$

La suite $(\frac{u_{n+1}}{u_n})$ converge vers le nombre d'Or.

Énoncés

Exercice 4

Écrire un programme permettant de déterminer les nombres premiers dans l'intervalle **[1,n]** à l'aide du crible d'Ératosthène. Il consiste à former une table avec tous les entiers naturels compris entre **2** et **n** et à rayer (mise à zéro), les uns après les autres, les entiers qui ne sont pas premiers de la manière suivante : dès que l'on trouve un entier qui n'a pas encore été rayé, il est déclaré premier, et on raye tous les multiples de celui-ci.
À la fin du procédé, les nombres non barrés sont des nombres premiers.
On tiendra compte du fait qu'un nombre donné peut déjà avoir été éliminé en tant que multiple de nombres précédents déjà testés.
Par ailleurs, on sait que l'on peut réduire la recherche aux nombres de **2** à $\sqrt{n}$ (si un entier non premier est strictement supérieur à $\sqrt{n}$ alors il a au moins un diviseur inférieur à $\sqrt{n}$ et aura donc déjà été rayé).

Exercice 5

Écrire un programme permettant de trier un vecteur de nombres en ordre croissant puis décroissant. On s'appuiera sur l'algorithme appelé *tri à bulle* qui consiste à comparer 2 éléments consécutifs et à les intervertir si nécessaire.
Si après avoir terminé l'exploration du tableau au moins une interversion a été effectuée, on renouvelle l'exploration, sinon le tri est terminé.

Exercice 6

Écrire un programme permettant d'effectuer le produit de 2 matrices `A` et `B`. Leurs profils seront définis à l'aide de constantes symboliques. La matrice résultat `C` sera imprimée à l'écran ligne par ligne avec l'instruction `PRINT` puis stockée dans un fichier binaire que l'on nommera « **`exo6.matrice`** ».

Exercice 7

Le fichier texte séquentiel « **`musiciens`** » est constitué de plusieurs enregistrements, chacun contenant un nom de musicien suivi de ses années de naissance et de mort.
Écrire un programme dont le but est de lire le fichier « **`musiciens`** » et de stocker les enregistrements lus dans un fichier binaire à accès direct que l'on nommera « **`musiciens.bin`** ».

Énoncés

Exercice 8

Imprimer l'enregistrement du fichier « **`musiciens`** » dont le rang est entré au clavier. Son extraction sera effectuée à partir d'un fichier temporaire à accès direct, image du précédent.
On permettra la saisie de plusieurs rangs.

Exercice 9

Les enregistrements des fichiers séquentiels
« **`index_naissance.dat`** » et « **`index_deces.dat`** » sont constitués d'une date de naissance (ou de décès) d'un musicien suivi de son rang dans le fichier « **`musiciens.bin`** » créé à l'exercice 7.
Écrire un programme permettant d'imprimer le ou les musiciens dont la date de naissance ou de mort est saisie au clavier. Le type de date désirée sera préalablement déterminé.
La sélection des enregistrements répondant aux choix spécifiés, s'effectuera par l'intermédiaire du fichier d'index correspondant au type de date.
On offrira la possibilité d'effectuer plusieurs recherches.

Exercice 10

Le but de cet exercice est de transformer la matrice stockée dans le fichier binaire « **`exo6.matrice`** ». Cette transformation consiste à modifier chaque élément à l'aide d'une fonction paramétrable de la forme $y = f(x)$.
On définira plusieurs fonctions de ce type. La valeur d'un entier lu dans une *namelist* indiquera la fonction à transmettre en argument de la procédure chargée d'effectuer la transformation.

Exercice 11

Trier les vecteurs lignes puis les vecteurs colonnes d'une matrice en utilisant l'algorithme *tri à bulle* et la matrice stockée dans le fichier binaire « **`exo6.matrice`** ».
On se définira une procédure effectuant le tri (croissant ou décroissant) des différents vecteurs au moyen d'une procédure interne.

Corrigé de l'exercice 1

```
program systeme
  implicit none
  real u1,u2
  real v1,v2
  real w1,w2
  real delta, delta_x, delta_y
  real x,y

                ! Valorisation des coefficients.
  u1 = 2; u2 = 4
  v1 = 5; v2 = 11
  w1 = 7; w2 = 6

              ! Calcul du déterminant principal.
  delta = u1*v2 - u2*v1
  if ( delta == 0. ) then
    print *, "Le système n'a pas de solution unique."
    stop 4
  end if
                ! Calcul du déterminant en x.
  delta_x = w1*v2 - w2*v1
                ! Calcul du déterminant en y.
  delta_y = u1*w2 - u2*w1
                 ! calcul des solutions.
  x = delta_x/delta
  y = delta_y/delta
                 ! Impression des solutions.
  print *, "x = ", x, ", y = ", y
end program systeme
```

Corrigé de l'exercice 2

```
program trinome
  implicit none
  real a, b, c
  real delta, r_delta, x1, x2

              ! Valorisation des coefficients.
  a = 3.; b = 7.; c = -11.

                ! a doit être non nul.
  if ( a == 0. ) stop "a doit être non nul."

                ! calcul du déterminant.
  delta = b*b - 4*a*c
              ! cas du déterminant négatif.
  if( delta < 0. ) stop "Pas de racine réelle."

                ! cas du déterminant nul.
  if ( delta == 0. ) then
    x1 = -b/(2*a); x2 = x1
  else           ! cas du déterminant positif.
    r_delta = sqrt( delta )
    x1 = (-b - r_delta)/(2*a); x2 = (-b + r_delta)/(2*a)
  end if

                 ! Impression des racines.
  print *,"x1 = ", x1, ", x2 = ", x2
end program trinome
```

Corrigé de l'exercice 2 (solutions dans le corps des complexes)

```
program trinome
  implicit none
  real    a, b, c
  real    delta, expr1, expr2
  complex x1, x2

            ! Valorisation des coefficients.
  a = 1.4; b = 2.7; c = 2.9

            ! a doit être non nul.
  if ( a == 0. ) stop "a doit être non nul."
            ! calcul du déterminant.
  delta = b*b - 4*a*c

  expr1 = -b/(2*a)
  expr2 = abs(delta)**0.5d0/(2*a)

  if( delta < 0. ) then ! racines complexes
    x1 = cmplx( expr1,  expr2 )
    x2 = cmplx( expr1, -expr2 )
    print *, "Les racines sont complexes => x1=", x1, " x2=", x2
  else                  ! racines réelles
    x1 = expr1 + expr2
    x2 = expr1 - expr2
    print *, "Les racines sont réelles => x1=", real(x1), " x2=", real(x2)
  end if
end program trinome
```


Corrigé de l'exercice 3

```
program nombre_dor
  implicit none
  real, parameter :: epsilon = 1.e-5
  real            :: u_prec, u_cour
  real            :: v_prec, v_cour
  real            :: somme
  real            :: nombre_or

  nombre_or = (1. + 5.**0.5)/2.

  u_prec = 1.; u_cour = 1.
  do
    v_prec = u_cour/u_prec
    somme  = u_cour + u_prec
    u_prec = u_cour
    u_cour = somme
    v_cour = u_cour/u_prec
    if ( abs( (v_cour-v_prec)/v_prec ) < epsilon ) exit
  end do

  print*, "Limite de la suite (vn) : ", v_cour, &
          "Nombre d'or : ", nombre_or
end program nombre_dor
```

Corrigé de l'exercice 3 (autre solution)

```
program nombre_dor
  implicit none
  double precision, parameter :: epsilon = 1.d-11
  double precision            :: u_prec, u_cour
  double precision            :: v_cour
  double precision            :: somme
  double precision            :: nombre_or

  ! Attention : une partie de l'expression suivante est
  ! ---------   évaluée en simple précision. Ce qui fait
  !             diverger le calcul itératif qui suit!
  !nombre_or = (1.d0 + 5.**0.5)/2.d0

  nombre_or = (1.d0 + 5.d0**0.5d0)/2.d0

  u_prec = 1.d0; u_cour = 1.d0
  do
    somme  = u_cour + u_prec
    u_prec = u_cour
    u_cour = somme
    v_cour = u_cour/u_prec
    if ( abs( (v_cour-nombre_or) < epsilon ) exit
  end do

  print*, "Limite de la suite (vn) : ", v_cour, &
          "Nombre d'or : ", nombre_or
end program nombre_dor
```


Corrigé de l'exercice 4

```
program eratosthene
  implicit none
  integer, parameter     :: n = 1000
  integer, dimension(n) :: tab_nombres
  integer                :: imax
  integer i, j

  tab_nombres(:) = (/ (i,i=1,n) /)

  imax = int(sqrt(real(n)))
  do i=2,imax
    if( tab_nombres(i) /= 0 ) then
      do j=i+1,n
        if ( tab_nombres(j) /= 0 .and. &
             mod( tab_nombres(j), i ) == 0 ) &
          tab_nombres(j) = 0
      end do
    end if
  end do

  print *,"Les nombres premiers entre 1 et ", n, " sont :"
  do i=2,n
    if ( tab_nombres(i) /= 0 ) print *,tab_nombres(i)
  end do
end program eratosthene
```

Corrigé de l'exercice 4 (autre solution)

```
program eratosthene
  implicit none
  integer, parameter    :: n = 1000
  logical, dimension(n) :: tab_nombres
  integer               :: imax
  integer i, j

  tab_nombres(:) = .true.

  imax = n**0.5
  do i=2, imax
    if( tab_nombres(i) ) then
      do j=i+1,n
        if ( tab_nombres(j) .and. j/i*i == j ) &
          tab_nombres(j) = .false.
      end do
    end if
  end do

  print *,"Les nombres premiers entre 1 et ", n, " sont :"
  do i=2,n
    if ( tab_nombres(i) ) print *,i
  end do
end program eratosthene
```


Corrigé de l'exercice 5

```
program triabulle
  implicit none
  integer, parameter :: croissant=1, decroissant=2, n=10
  real, dimension(n) :: tab
  real               :: temp
  logical            :: tri_termine, expr1, expr2
  integer            :: sens, i
             ! Valorisation du vecteur
  data tab/0.76, 0.38, 0.42, 0.91, 0.25, &
           0.13, 0.52, 0.69, 0.76, 0.98/
  do sens=croissant, decroissant             ! Sens du tri
    do                                       ! Tri
     tri_termine = .true.
     do i=2,n
       expr1 = sens == croissant   .and. tab(i-1) > tab(i)
       expr2 = sens == decroissant .and. tab(i-1) < tab(i)
       if (expr1 .or. expr2) then
         tri_termine = .false.
         temp = tab(i-1); tab(i-1) = tab(i); tab(i) = temp
       end if
     end do
     if (tri_termine) exit
    end do
           ! Impression du vecteur trié
    if (sens == croissant)   print*, "Tri croissant "
    if (sens == decroissant) print*, "Tri décroissant "
    print*, tab
  end do
end program triabulle
```

Corrigé de l'exercice 6

```
program produit_matrice
  implicit none
  integer, parameter    :: n = 10, m = 5, p = 3
  real, dimension(n,m) :: a
  real, dimension(m,p) :: b
  real, dimension(n,p) :: c
  integer               :: i,j,k

          ! Valorisation des matrices A et B
  data          ((a(i,j),j=1,m),i=1,n)/ &
         0.00, 0.38, 0.42, 0.91, 0.25, &
         0.13, 0.52, 0.69, 0.76, 0.98, &
         0.76, 0.83, 0.59, 0.26, 0.72, &
         0.46, 0.03, 0.93, 0.05, 0.75, &
         0.53, 0.05, 0.85, 0.74, 0.65, &
         0.22, 0.53, 0.53, 0.33, 0.07, &
         0.05, 0.67, 0.09, 0.63, 0.63, &
         0.68, 0.01, 0.65, 0.76, 0.88, &
         0.68, 0.38, 0.42, 0.99, 0.27, &
         0.93, 0.07, 0.70 ,0.37, 0.44/

  data ((b(i,j),j=1,p),i=1,m)/ &
           0.76, 0.16, 0.9047, &
           0.47, 0.48, 0.5045, &
           0.23, 0.89, 0.5163, &
           0.27, 0.90, 0.3190, &
           0.35, 0.06, 0.9866/
```


Corrigé de l'exercice 6 (suite)

```
                ! Produit de matrice.
  do i=1,n
    do j=1,p
      c(i,j) = 0.
      do k=1,m
         c(i,j) = c(i,j) + a(i,k) * b(k,j)
      end do
    end do
  end do

            ! Impression de la matrice c.
  do i=1,n
    print *, c(i,:)
  end do

            ! Écriture de la matrice c dans un fichier.
  open( unit=1,           file="exo6.matrice", &
        status="replace", form="unformatted",  &
        action="write" )
  write( unit=1 ) c
  close( unit=1 )
end program produit_matrice
```

Corrigé de l'exercice 7

```
program ecriture_musiciens
  character(len=80) :: mus
  integer           :: ios_mus, lrecl
  integer           :: numrec
!  Ouverture du fichier des musiciens
!  ainsi que d'un fichier en écriture
!  à accès direct dans lequel on
!  va recopier le fichier précédent.
  open( unit=1,              file="musiciens",      &
        form="formatted",    status="old",          &
        action="read",       position="rewind" )
! Calcul de la taille des enregistrements du fichier
! binaire en sortie. (dépend du compilateur).
  inquire( iolength=lrecl ) mus
  open( unit=2,              file="musiciens.bin", &
        status="replace",                          &
        form="unformatted", access="direct",       &
        action="write",      recl=lrecl )
! On effectue la copie.
  numrec = 0
  read( unit=1, fmt="(a)", iostat=ios_mus ) mus
  do while ( ios_mus == 0 )
    numrec = numrec + 1
    write( unit=2, rec=numrec) mus
    read( unit=1, fmt="(a)", iostat=ios_mus ) mus
  end do
  close( unit=1 ); close( unit=2 )
  if ( ios_mus > 0 ) print *,"Erreur lecture sur l'unité 1."
end program ecriture_musiciens
```


Corrigé de l'exercice 8

```
program musiciens
  implicit none
  integer, parameter   :: lrecl=80
  character(len=lrecl) :: mus
  integer              :: ios_mus, ios_stdin
  integer              :: numrec, rang

! Ouverture du fichier des musiciens
! ainsi que d'un fichier temporaire
! à accès direct dans lequel on
! va recopier le fichier précédent.
  open( unit=1,              file="musiciens",   &
        form="formatted",    status="old",       &
        action="read",       position="rewind" )
  open( unit=2,              status="scratch",   &
        form="formatted",    access="direct",    &
        action="readwrite", recl=lrecl )
! On effectue la copie.
  numrec = 0
  do
    read( unit=1, fmt="(a)", END=1 ) mus
    numrec = numrec + 1
    write( unit=2, rec=numrec, fmt="(a)" ) mus
  end do
1 close( unit=1 )
```

Corrigé de l'exercice 8 (suite)

```
! On demande un rang de musicien.

  do
    print *,"Entrez le rang d'un musicien :"
    read( unit=*, fmt=*, iostat=ios_stdin ) rang
    do while ( ios_stdin == 0 )
      read( unit=2, rec=rang, fmt="(a)", iostat=ios_mus ) mus
      if ( ios_mus /= 0 ) then
        print *,"Le musicien de rang ", &
                rang, "n'existe pas"
      else
        print "('musicien de rang',i3,' ==> ', a)", &
              rang,trim(mus)
      end if
      print *,"Entrez le rang d'un musicien :"
      read( unit=*, fmt=*, iostat=ios_stdin ) rang
    end do
    if ( ios_stdin > 0 ) then
      print "(a,/)","Erreur lors de la saisie."
    else
      exit
    end if
  end do
  close( unit=2 )
end program musiciens
```


Corrigé de l'exercice 9

```
program sequentiel_indexe
  implicit none
  character(len=19), dimension(2), parameter :: f_index = &
        (/ "index_naissance.dat", "index_deces.dat    " /)
  character(len=80) :: musicien
  integer           :: lrecl

  ! Ouverture du fichier des musiciens en accès direct
  ! ainsi que des fichiers d'index.
  open ( unit=1,          file = f_index(1),               &
         status="old",    form="formatted", action="read" )
  open ( unit=2,          file = trim(f_index(2)),         &
         status="old",    form="formatted", action="read" )
  ! Calcul de la taille des enregistrements du fichier
  ! créé à l'exercice 7. (dépend du compilateur).
  inquire( iolength=lrecl ) musicien
  open ( unit=3,          file="musiciens.bin",            &
         status="old",    form="unformatted",              &
         access="direct", action="read", recl=lrecl )

  call recherche_musicien

  close( unit=1 )
  close( unit=2 )
  close( unit=3 )
end program sequentiel_indexe
```

Corrigé de l'exercice 9 (suite)

```
subroutine recherche_musicien
  implicit none
  integer, parameter :: EOF = -1
  character(len=50)  :: prompt_date, message
  character(len=4)   :: date_saisie, date_lue
  integer            :: unite_index

  do
    ! Type de date : naissance ou décès
    if ( choix_date() == EOF ) exit
    ! Lecture d'une date.
    if ( lecture_date() == EOF) exit
    ! Recherche du ou des musiciens
    if ( .not. musicien_trouve() ) &
      print *,"Aucun musicien ne répond à la demande"
    print *
  end do

contains
```

Corrigé de l'exercice 9 (suite)

```
  integer function choix_date()
    integer   ios_stdin, reponse
    character tabulation

    tabulation = achar(9)
    do
      print"(2a)",tabulation,"--------------------------------"
      print"(2a)",tabulation,"Choix du critère de recherche : "
      print"(2a)",tabulation,"- par date de naissance (1)"
      print"(2a)",tabulation,"- par date de décès     (2)"
      print"(2a)",tabulation,"- QUITTER               (3)"
      print"(2a)",tabulation,"--------------------------------"
      read(*, *, IOSTAT=ios_stdin) reponse
      if ( ios_stdin < 0 ) then
        choix_date = EOF
        return
      else if ( ios_stdin > 0 ) then
        print "(/,a,/)", "Erreur dans la saisie"
      else
        if ( reponse < 1 .or. reponse > 3 ) then
          print *,"Choix invalide."
        else
          exit
        end if
      end if
    end do
```

Corrigé de l'exercice 9 (suite)

```
  select case (reponse)
    case(1) ! Recherche par date de naissance.
      prompt_date = "Entrer une date de naissance d'un musicien"
      message = "Liste des musiciens nés en"
    case(2) ! Recherche par date de décès.
      prompt_date = "Entrer une date de décès d'un musicien"
      message = "Liste des musiciens morts en"
    case(3) ! Quitter
      choix_date = EOF
      return
  end select
  unite_index = reponse
  rewind( unit=unite_index )
  choix_date = 0
end function choix_date
```

Corrigé de l'exercice 9 (suite)

```
function lecture_date()
  integer lecture_date
  integer ios_stdin

  do
    print "(/,a)", trim(prompt_date)
    read(*, *, IOSTAT=ios_stdin) date_saisie
    if( ios_stdin < 0 ) then
      lecture_date = EOF
      return
    else if( ios_stdin > 0 ) then
      print "(/,a,/)", "Date erronée!"
    else
      message = trim(message)//" "//date_saisie
      exit
    end if
  end do
  lecture_date = 0
end function lecture_date
```

Corrigé de l'exercice 9 (suite)

```
  function musicien_trouve()
    logical musicien_trouve
    ! Déclarations locales
    character(len=80) :: musicien
    character(len=11) :: fmt = "(/,a,/,  a)"
    integer           :: i, taille_message, numrec

    taille_message = len_trim(message)

    write( fmt(8:9), "(i2)" ) taille_message
    print fmt, message, ("-",i=1,taille_message)

    musicien_trouve = .false.
    do
      read( unit=unite_index, fmt=*, END=1 ) date_lue, numrec
      if ( date_lue == date_saisie ) then
        ! On lit l'enregistrement correspondant.
        musicien_trouve = .true.
        read( unit=3, rec=numrec ) musicien
        print "(2a)"," - ", trim(musicien)
      end if
    end do
1   return
  end function musicien_trouve
end subroutine recherche_musicien
```

Corrigé de l'exercice 10

```
program mat_transf
  implicit none
  integer, parameter   :: n = 10, m = 3
  real, dimension(n,m) :: mat
  integer              :: choix_methode, ios, num_ligne
  real, external       :: carre, identite, logarithme
  real, intrinsic      :: sqrt
  namelist/methode/choix_methode
        ! Ouverture du fichier contenant la matrice.
  open( unit=1,              file="exo6.matrice", &
        form="unformatted", action="read",       &
        status="old",        position="rewind",   &
        iostat=ios )
  if (ios /= 0) &
    stop "Erreur à l'ouverture du fichier 'exo6.matrice'"
              ! Lecture de la matrice.
  read(1) mat
  close(1)
          ! Ouverture du fichier contenant
              ! la namelist "methode".
  open( unit=1,            file="exo10.namelist", &
        form="formatted", action="read",         &
        status="old",      position="rewind",     &
        iostat=ios )
  if (ios /= 0) &
    stop "Erreur à l'ouverture du fichier 'exo10.namelist'"
  read( unit=1, nml=methode )
  close( unit=1 )
```

Corrigé de l'exercice 10 (suite)

```
        ! Transformation de la matrice à l'aide
                ! de la méthode choisie.
  select case( choix_methode )
    case (1)
      call transform( mat, n, m, identite )
    case (2)
      call transform( mat, n, m, carre )
    case (3)
      call transform( mat, n, m, sqrt )
    case (4)
      call transform( mat, n, m, logarithme )
  end select
        ! Sauvegarde de la matrice transformée dans
                ! le fichier "exo6_matrice_transf".
  open( unit=1,           file="exo6_matrice_transf", &
        form="formatted", action="write",             &
        status="replace", iostat=ios )

  if ( ios /= 0 ) &
    stop "Erreur lors de l''ouverture &
         &du fichier ""exo6_matrice_transf"""

  do num_ligne=1,n
    write( unit=1, fmt="(3f10.6)" ) mat(num_ligne,:)
  end do
  close( unit=1 )
end program mat_transf
```


Corrigé de l'exercice 10 (suite)

```
              ! Procédure de transformation.
subroutine transform( t, n, m, f )
  implicit none
  integer               :: n, m, i, j
  real, dimension(n,m) :: t
  real                  :: f

  do i=1,n
    do j=1,m
      t(i,j) = f(t(i,j))
    end do
  end do
end subroutine transform
       ! Définitions des fonctions de transformation.
function identite(x)
  implicit none
  real x, identite
  identite = x
end function identite
function carre(x)
  implicit none
  real x, carre
  carre = x*x
end function carre
function logarithme(x)
  implicit none
  real x, logarithme
  logarithme = log(x)
end function logarithme
```

Corrigé de l'exercice 11

```
program tri_matrice
  implicit none
  integer, parameter    :: n=10, m=3
  real, dimension(n,m) :: mat
  integer               :: ios, i, j
            ! Lecture de la matrice à trier.
  open( unit=1,                &
        file="exo6.matrice", &
        form="unformatted",  &
        status="old",        &
        action="read",       &
        position="rewind",   &
        iostat=ios )
  if ( ios /= 0 ) stop "Erreur à l'ouverture du fichier &
                       &""exo6.matrice"""
  read( unit=1 ) mat; close( unit=1 )
  call tri( mat, n, m ) ! Tri de la matrice lue.
             ! Écriture de la matrice triée.
  open( unit=1,           file="exo11.matrice_triee", &
        form="formatted", status="replace",           &
        action="write",   position="rewind",          &
        iostat=ios )
  if ( ios /= 0 ) stop "Erreur à l'ouverture du fichier &
                       &""exo11.matrice_triee"""
  do i=1,n
    write( unit=1, fmt="(3F7.3)" ) mat(i,:)
  end do
  close( unit=1 )
end program tri_matrice
```


Corrigé de l'exercice 11 (suite)

```
                  ! Procédure de tri.
subroutine tri( mat, n, m )
  implicit none
  integer               :: n, m, ligne, col
  real, dimension(n,m) :: mat

  do ligne=1,n       ! Tri des lignes.
    call tri_vecteur( mat(ligne,:), m )
  end do
  do col=1,m         ! Tri des colonnes.
    call tri_vecteur( mat(:,col), n )
  end do
  contains ! Procédure de tri d'un vecteur.
    subroutine tri_vecteur( v, n )
      integer              :: n, i
      real, dimension(n) :: v
      logical              :: tri_termine
      do
       tri_termine = .true.
       do i=2,n
         if ( v(i) > v(i-1) ) then
           ! Utilisation de sections régulières pour effectuer l'interversion.
           tri_termine = .false.; v(i-1:i) = v(i:i-1:-1)
         end if
       end do
       if (tri_termine) exit
      end do
    end subroutine tri_vecteur
end subroutine tri
```

– Symboles –

assumed-size array 173
assumed-size string 171

– A –

accès
  direct 137, 139, 141
    exemple 143
  séquentiel 97, 99, 101
ANSI 9
ar 235
argument
  chaîne de caractères 171
  procédure 175
  tableau 173
argument procédure 177
arguments d'appel 165, 167, 169
arguments muets 165
ASA 9
assign 229
attribut 37
  DIMENSION 81

– B –

BACKSPACE 151
base
  hexadécimale 17
  octal 17
bases de numération 17
bibliographie 11
bibliothèque 235
bloc 95
Block data 189
  exemple 191
buffer 95

– C –

CALL 161
caractère

  représentation en mémoire 25
  table ASCII 27
CHARACTER* 231
CLOSE 97
  exemple 217
  IOSTAT= 215
  STATUS= 215
  UNIT= 215
code source
  jeu de caractères 29
Common 185, 187, 189, 191, 193, 195
  étiqueté 189
    attribut save 191, 193
    initialisation 189, 191
  blanc 185, 187
    exemple 187
    initialisation 187
  règles et restrictions 193, 195
  syntaxe 185
compilateur 233
complexe
  représentation en mémoire 23
constantes
  chaînes de caractères 45
  complexes 45
  entières 41
  littérales 41, 43, 45
  réelles double précision 43
  réelles simple précision 43
  symboliques 49
CONTAINS 177

– D –

déclaration
  attribut PARAMETER 49
  character 39
  implicit none 41
  initialisation avec '=' 49
  instruction equivalence 51
    exemple 51, 52
  syntaxe 39

tableau 81
DATA 231
DELIM= 135
descripteurs de format 101
Durée de vie et visibilité des identificateurs 181

– E –

ENDFILE 151
enregistrement logique 95
entier
représentation en mémoire 19
entrée standard 157
equivalence 51
exercices
énoncés
exercice 1 237
exercice 10 239
exercice 11 239
exercice 2 237
exercice 3 237
exercice 4 239
exercice 5 239
exercice 6 239
exercice 7 239
exercice 8 239
exercice 9 239
corrigés
exercice 1 241
exercice 10 259
exercice 11 263
exercice 2 ..... 241, 243
exercice 3 243
exercice 3 (autre solution) 245
exercice 4 245
exercice 4 (autre solution) 247
exercice 5 247
exercice 6 249
exercice 7 251
exercice 8 251
exercice 9 25
EXTERNAL 175

– F –

fichier
binaire 99, 139
destruction 147
interne 147
temporaire 145
texte 101, 141
FMT= 101
fonction
statement function 231
format
Litteral string 123
BOZ 157
descripteur A en écriture 121
descripteur A en lecture 109
descripteur EN en écriture 117
descripteur ES en écriture 119
descripteur F en écriture 113
descripteur I en écriture 113
descripteur I en lecture 105
descripteur L en écriture 121
descripteur L en lecture 109
descripteurs
en écriture 111
en lecture 105
descripteurs / 127
descripteurs de contrôle 125
descripteurs E, D en écriture 115
descripteurs F en lecture 107
descripteurs F, E, D en lecture 107
descripteurs SS,SP,S 119
facteur de répétition 127
gabarit indéfini 115
instruction 159
libre 131
réexploration 129
format fixe 231
format libre 231
formats d'édition 101
Fortran
documentation 13

fortran 2003 . . . 9
fortran 2008 . . . 9
fortran 66 . . . 9
fortran 77 . . . 9
fortran 90 . . . 9
fortran 95 . . . 9
fortran IV . . . 9
fortran V . . . 9
Fortran 95
aspects obsolètes . . . 229, 231
FUNCTION . . . 169

– G –

GO TO calculé . . . 231

– I –

identificateur . . . 35
identificateurs
durée de vie . . . 181
visibilité . . . 181
INCLUDE . . . 197
initialisation
instruction DATA . . . 47
INQUIRE . . . 155
ACCESS= . . . 209
ACTION= . . . 209
BLANK= . . . 209
DELIM= . . . 211
DIRECT= . . . 211
ERR= . . . 211
EXIST= . . . 211
FILE= . . . 211
FORM= . . . 211
FORMATTED= . . . 211
IOSTAT= . . . 211
NAME= . . . 211
NAMED= . . . 213
NEXTREC= . . . 213
NUMBER= . . . 213
OPENED= . . . 213

PAD= . . . 213
POSITION= . . . 213
READ= . . . 213
READWRITE= . . . 213
RECL= . . . 213
SEQUENTIAL= . . . 215
syntaxe . . . 209
UNFORMATTED= . . . 215
UNIT= . . . 209
WRITE= . . . 215
inquire
iolength= . . . 145
instruction
GO TO . . . 71
include . . . 197
instructions
positionnement . . . 151
INTRINSIC . . . 175
IOSTAT . . . 97

– L –

LEN= . . . 171
loader . . . 233
logique
représentation en mémoire . . . 25

– M –

make . . . 235
module exécutable . . . 233
module objet . . . 233

– N –

NAMELIST . . . 101, 135
NML= . . . 101, 135

– O –

opérateurs

arithmétiques
conversion implicite 57
concaténation 61
d'affectation 63
logiques 61
relationnels 59
table des priorités 63
OPEN 97, 99
ACCESS= 201
ACTION= 203
BLANK= 203
DELIM= 203
exemple 217
FILE= 201
FORM= 203
IOSTAT= 201
PAD= 203
POSITION= 203
RECL= 203
STATUS= 201
syntaxe 201
UNIT= 201

– P –

préprocesseur 233
PRINT 157
procédure
fonction 169
interne 177
intrinsèque 183
subroutine 167
procédures intrinsèques
ABS 219
ACHAR 171, 219
ACOS 219
ADJUSTL 219
ADJUSTR 219
AIMAG 219
AINT 219
ANINT 219
ASIN 219

Annexe E : exercices

ATAN 219
BIT_SIZE 219
BTEST 221
CEILING 221
CMPLX 221
CONJG 221
COS 221
COSH 221
DBLE 221
EXP 221
FLOOR 221
IACHAR 171, 221
IAND 221
IBCLR 221
IBITS 223
IBSET 223
IEOR 223
INDEX 223
INT 223
IOR 223
ISHFT 223
ISHFTC 223
LEN 225
LEN_TRIM 225
LGE 225
LGT 225
LLE 225
LLT 225
LOG 225
LOG10 225
MAX 225
MIN 225
MOD 225
NOT 225
REAL 225
REPEAT 225
SCAN 225
SIGN 227
SIN 227
SINH 227
SQRT 227

Anne Fouilloux Patrick Corde (anne.fouilloux@ecmwf Langage Fortran(F95-1) 28 septembre 2012 262 / 262

TAN 227
TANH 227
TRIM 227
VERIFY 227

– R –

réel
précision 37
représentation en mémoire 19
READ
ADVANCE= 205
END= 159, 161, 205
EOR= 205
ERR= 159, 205
FMT= 205
IOSTAT= 163, 205
NML= 205
REC= 205
SIZE= 205
syntaxe 205
UNIT= 205
REC= 137, 139, 141
représentation en mémoire
des caractères 25
des complexes 23
des entiers 19
des logiques 25
des réels 19
return 167, 169
REWIND 151

– S –

scratch 145
sortie standard 157
structures de contrôle
DO
syntaxe 71
DO indéxé 73
DO WHILE 75
DO/CYCLE 77

DO/EXIT 75
IF 67
SELECT CASE 69
syntaxe
commentaire 33
format fixe 31
format libre 31

– T –

tableau
étendue 83
agencement en mémoire 85
conformance 83
constructeur de vecteur 87
déclaration 81
expression 89
initialisation
instruction DATA 89
symbole '=' 85
profil 83
rang 83
section régulière 91
tampon 95
tests
IF 67
SELECT CASE 69
Types prédéfinis 37

– U –

UNIT=* 157
unité logique 97
unités de programme 29

– V –

variable
type 37

– W –

WRITE

ADVANCE= 207
ERR= 207
FMT= 207
IOSTAT= 207
NML= 207
REC= 207
syntaxe 207
UNIT= 207

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com