Le Langage
FORTRAN

Version 2 Alpha

Fortran 77, 90, 95

J.J. HUNSINGER

Cet abrégé de FORTRAN est un des supports de cours de l'Unité de Valeur AG43 (Bases de l'informatique à l'usage de l'ingénieur) UTBM. Il peut être diffusé librement en l'état. Il s'agit actuellement de la Version 2 Alpha, en cours d'élaboration, dont le contenu n'est pas encore figé, et les fautes pas encore corrigées.
La date de dernière mise à jour est : 13/04/2005 10:42

JJH, 05/2005

INTRODUCTION

1.1 Historique

En Novembre 1954, L'informatique commence à prendre de l'ampleur. Jusqu'alors, les langages de programmation sont restés proches du langage machine. La compagnie International Business Machines (Les initiales ne vous rappellent-elles rien ?) publie la description d'un langage de programmation évolué appelé alors système de FORmula TRANslator (traducteur d'équations. Ce nouveau langage, plus condensé, avait pour vocation d'ouvrir l'utilisation des ordinateurs aux scientifiques. C'est en 1956 qu'est apparu le premier manuel de référence de chez IBM. Ce manuel décrivait le FORTRAN I. Evidemment, ce langage était appelé à évoluer. C'est ainsi que, successivement, sont apparus:

- 1957: FORTRAN II
- 1958: FORTRAN III (Resté plus ou moins confidentiel)
- 1962: FORTRAN IV (Il aura régné 16 ans)
- 1978: FORTRAN V (Normalisation ANSI et AFNOR)

Durant toute cette évolution FORTRAN a respecté toutes les anciennes normes de manière à assurer une transplantation rapide des anciens programmes sur les nouveaux compilateurs.

Le présent abrégé portera sur le FORTRAN 77, ainsi que le FORTRAN 90. Le premier est proche du FORTRAN V mais comporte en plus des instructions de structuration, le second se rapproche d'avantage des langages modernes et est orienté vers la parrallélisation des calculs. On trouve le langage FORTRAN sous différents systèmes d'exploitation comme MS-DOS (ordinateurs compatibles IBM PC), UNIX / LINUX, WINDOWS 9x, Me, NT et XP.

En fait, Chaque décennie a vu la naissance d'une nouvelle version de Fortran. Les dernières versions sont Fortran 90, Fortran 95 et Fortran 2003. On peut se demander s'il reste opportun de continuer à utiliser le Fortran, ou se tourner vers un autre langage pour le calcul scientifique. La réponse est claire si l'on utilise déjà le Fortran : On conserve ce langage, et on évolue avec lui. Pour quelqu'un qui cherche à utiliser ponctuellement un langage pour résoudre un petit problème ou pour faire une série de calculs, le BASIC semble bien indiqué, puisqu'on le trouve même intégré à des tableurs. Mais la richesses des bibliothèques de calculs que l'on peut trouver et l'évolution du langage maintiennent le Fortran comme l'outil de calcul scientifique et technique par excellence.

1.2 Elaboration d'un programme

Un programme Fortran nécessite trois types de fichiers pour son élaboration :

- Les fichiers source (extension .FOR, .F90 sous MS-DOS ou WINDOWS, .f sous UNIX[1])

[1]MS-DOS et UNIX sont ce qu'on appelle des systèmes d'exploitation. Ils permettent aux éditeurs de logiciels d'adapter plus facilement les langages aux différents types de machines. Ainsi, par exemple, pour créer, copier,

- Les fichiers objet (extension .OBJ sous MS-DOS, .o sous UNIX)
- Le fichier exécutable (extension .EXE sous MS-DOS ou WINDOWS, définie par l'utilisateur sous UNIX).

Le programmeur écrit le fichier source à l'aide d'un éditeur de texte de son choix. Il s'agit d'un texte clair, contenant toutes les instructions du programme. Ce fichier peut être envoyé sur une imprimante ou affiché sur l'écran. C'est ce que fait l'éditeur de texte.

Le programme source peut être écrit de deux façon :

- Sous la forme formattée, avec des colonnes d'usage dédié, héritage de l'époque des cartes perforées,
- Sous la forme libre, conforme à la plupart des autres langages. l'extension du fichier doit alors être f90.

Les contraintes de la forme formattée seront expliquées dans le paragraphe x*x.

Le fichier source doit ensuite être compilé, c'est à dire traduit en langage machine. C'est le rôle du compilateur. Chaque langage possède d'ailleurs un compilateur propre. Le fichier obtenu après compilation est un fichier objet, pas encore exécutable. Le fichier objet possède la particularité de pouvoir être relié à d'autres fichiers du même genre si l'utilisateur le désire. Ainsi, comme nous le verrons, les sous-programmes utiles et universels comme par exemple un sous-programme de résolution de zéro de fonction, pourra être appelé à partir d'un programme principal sans pour autant figurer dans le même fichier que ce dernier. L'éditeur de liens (LINK ou ld comme "loader") se chargeant de les lier et de les rendre exécutables.

Toute modification, correction d'un programme passe tout d'abord par la modification du fichier source. Ce fichier **doit** être ensuite **recompilé**, ce qui permet d'obtenir un fichier objet mis à jour. Attention, tout fichier exécutable comportant le module objet modifié n'est plus à jour. Il faut dans ce cas procéder à une nouvelle édition des liens.

Dans un cas simple, sur un compatible PC, on écrit le programme que nous appellerons par exemple PROG.FOR à l'aide d'un éditeur de texte au choix. Attention de ne pas prendre un traitement de texte. On sauvegarde le fichier source sur le disque dur en lui donnant un nom respectant les conditions du système d'exploitation utilisé (DOS, UNIX[2] en général). L'extension utilisée est généralement .FOR sous MS-DOS, et .f sous UNIX. Actuellement, sur des ordinateurs fonctionnant sous WINDOWS, l'extension .f90 est actuellement utilisée avec le compilateur Fortran fourni par COMPAQ, qui n'est autre que l'ancien Fortran Microsoft revendu, car pas assez rentable du point de vue commercial.

Le compilateur traduit les instructions qui ont été tapées par le programmeur et produit, si aucune erreur n'a été faite, en langage machine. La traduction est placée dans un fichier **objet** dont le nom est identique à celui du fichier source, mais dont l'extension est cette fois **.OBJ** sous DOS, et .o sous UNIX.

Attention, dans certaines configurations, l'éditeur de liens est automatiquement appelé et rend le programme exécutable.

---

modifier, enregistrer des fichiers, il n'est pas indispensable de connaître le type exact du support sur lequel on effectue les stockages. MS-DOS a totalement disparu actuellement.

[2]Se référer au manuel correspondant au système d'exploitation

Deux cas sont possibles :

Votre fichier programme contient tout ce qui est nécessaire, programme et sous-programmes. Dans ce cas, le fichier objet est pratiquement complet. Il ne lui manque que l'entête de démarrage, la mise en place des zones mémoire pour les variables, les fonctions intrinsèques (par ex. sin()), et les adresses relatives des différentes sections appelées. L'éditeur de liens se charge de résoudre ces problèmes qui sont en fait indépendants (plus ou moins) du type de langage utilisé. On peut, en prenant toutefois certaines précautions décrites dans les manuels, lier des modules objet créés à partir du FORTRAN à d'autres créés à partir du PASCAL et du C. L'éditeur de liens travaille indépendamment du langage utilisé, puisque toute notion de langage (à part le langage machine, évidemment) disparaît après l'opération de compilation.

L'éditeur de liens charge à partir d'un fichier appelé **librairie**[3] ces modules nécessaires, met bout à bout tous ces modules, y compris les modules objet produits par le programmeur, calcule les adresses relatives et met à jour toutes les instructions provoquant un branchement à une autre adresse.

Le fortran Microsoft Version 5.1 qui implémente déjà certaines spécificités du FORTRAN 90, est livré avec un environnement de développement incorporé. Les opérations d'édition, de compilation et d'édition des liens sont lancées depuis l'environnement de développement que l'on lance au début d'une session de travail. En 2004, L'ensemble de développement Fortran est distribué par COMPAQ sous l'appellation « COMPAQ Visual Fortran 6».

On trouve actuellement, gratuit, mais d'usage moins convivial, le Fortran 77 GNU, appelé g77. Il fonctionne sous Windows en mode console. Pour l'obtenir, faire une recherche sur internet avec les mots-clé FORTRAN et GNU. La version g95 est en cours de mise au point.

Les principales avancées du FORTRAN 90 concernent les manipulations de tableaux, une augmentation substantielle des procédures intrinsèques[4], la possibilité de définition de nouveaux types de variables (STRUCTURES), les boucles DO WHILE, les constructions Cas où (CASE)... L'instruction GOTO, toujours disponible, est à proscrire.

Un programme est souvent composé de plusieurs modules (sous-programmes), chacun pouvant se trouver dans des fichiers séparés.

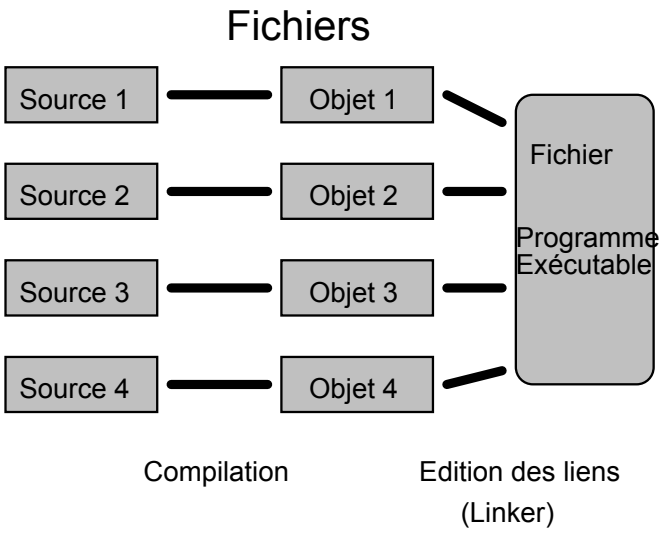
L'application complète comportera tous les modules liés. Tout d'abord, il conviendra de compiler séparément sans édition des liens chaque module. A l'issue de cette opération, on obtiendra des modules objets, c'est à dire en langage machine, mais sans adresse d'implantation en mémoire. On les reliera tout en fixant les adresses à l'aide de l'éditeur de liens LINK.EXE. Le fichier obtenu sera le programme exécutable.

Ces opérations mettant en jeu de multiples fichiers sont grandement facilitées par l'emploi de fichier **projet**. Les environnements de développement permettent la gestion de projets.

Un fichier projet est un fichier comportant toutes les informations nécessaires pour reconstituer un programme exécutable, ou une librairie. La constitution d'un fichier projet est

[3]Une librairie est un assemblage de modules objet indépendants, ce qui évite l'obligation de manipuler une quantité impressionnante de fichiers.
[4]Fonctions math ou autres disponibles avec la librairie standard du FORTRAN

extrêmement simple. Cette action est décrite dans le manuel d'utilisation de l'environnement de développement.



2. Structure des instructions

2.1 Structure d'un programme

Un programme comporte généralement plusieurs modules. Ces modules peuvent figurer dans un même fichier ou dans des fichiers différents. Lorsque les modules figurent dans des fichiers individuels, il est possible, et c'est même fortement conseillé, de les réutiliser dans d'autres programmes. L'inconvénient (mineur) est la nécessité d'emploi d'un fichier projet, chargé de communiquer à l'éditeur de liens les noms des différents modules à lier.

Un ensemble de modules doit, pour pouvoir être exécuté, comporter un programme principal. Le programme principal peut faire appel à des procédures et à des fonctions qui peuvent faire de même.

Un module comporte différentes parties :

- L'entête ou le nom du module. En FORTRAN on peut trouver:
  - **PROGRAM** *nom du programme* pour le programme principal
  - **SUBROUTINE**(*Liste des argumentss*) pour un sous-programme
  - **FUNCTION**(*Liste des arguments*) pour une fonction
- Les déclarations des arguments, s'il y a lieu
- Les déclarations des variables
- Les initialisations et déclarations particulières. Cette section est un peu particulière et sera traitée en fin de manuel.
- Les instructions exécutables. C'est la traduction en langage (FORTRAN) de l'algorithme.
- l'instruction FIN de module. En FORTRAN, il s'agit de **END**.

Attention, dans un fichier comportant plusieurs modules, il y aura autant d'instructions FIN que de modules.

Pratiquement chaque module possède ses variables. Vous pouvez choisir leur nom sans tenir compte de ceux que vous avez déjà utilisés dans d'autres modules. En effet, la seule possibilité de communication de variables entre différents modules se fait par les **arguments** d'appel et par les blocs **COMMON**.

2.2 Démarrage rapide - Deux instructions utiles : READ et WRITE

Pour réaliser son premier programme, il suffit de connaître très peu d'instructions. Avec quatre instructions, on peut s'essayer aux joies de la compilation : PROGRAM, READ, WRITE, END.

2.2.1 Utilisation de READ et WRITE

Ce sont, on l'aura devinné, des instructions le lecture et d'écriture. Elles sont plus amplement décrites dans le § 6.1. Dans un premier temps, on se contentera des instructions de lecture/écriture sur la console (écran – clavier), sans mise en forme (format libre). Si A et B sont des variables, l'entrée de leur valeur respective à partir du clavier s'écrira :

```
READ(*,*) A,B
```

La liste (*,*) est passe-partout, et sera employée pour des saisies rapides et des affichages rapides pour vérification et mise au point.
Les valeurs de A et B seront entrées au clavier, sur la même ligne, séparées par un espace, une virgule, ou une tabulation. L'appui sur la touche ENTER (retour chariot ou entrée) valide la saisie.
L'affichage sur l'écran du contenu des variables A et B sera écrit :

```
WRITE(*,*) A,B
```

Le format d'affichage est libre, et il s'adapte en fonction des variables.

2.2.2 Le premier programme

Le premier programme sera évidemment simple et parfaitement inutile. Il sera écrit en format libre (l'extension du fichier source sera ".f90", la plupart des compilateurs comprenant alors qu'il n'y a pas de format standard respecté, comme décrit paragraphe 2.6).

```
PROGRAM exemple1
     READ(*,*) A,B
     C=A*A+B*B
     WRITE(*,*)C
END PROGRAM exemple1
```

En exécutant le programme, on tombe sur une fenêtre vide, dans laquelle rien ne semble se passer. En réalité, le programme attend les valeurs pour A et B. on les donne en tapant au clavier les valeurs 4 et 5, séparées par un espace, et validées par un appui sur la touche entrée. Voici ce que l'on obtient :

```
4 5                              On entre ces valeurs et on valide
   41.00000                      C'est la réponse de l'ordinateur
Press any key to continue        Vous devez appuye une touche pour fermer
```

On peut faire quelques remarques :
A, B et C sont des variables. Le compilateur a estimé que ces variables étaient des nombres réels. En fait, il a procédé à une déclaration implicite (voir § 3.3.2). Au moment de l'éxécution, l'apparition d'un écran vide peut surprendre, et il serait plus judicieux d'afficher un texte. L'instruction PROGRAM n'est pas obligatoire, mais on prend l'habitude de l'utiliser.
Dans l'exemple qui suit, les deux valeurs entrées sont séparées par une virgule. La virgule ne sépare pas les décimales de la partie entière d'un nombre, mais sert de séparateur de valeurs.
L'instruction PROGRAM a également été volontairement omise. L'instruction END reste obligatoire.

```
WRITE(*,*) 'Donner A et B :'
READ(*,*) A,B
C=A*A+B*B
WRITE(*,*)C
END

Donner A et B :
2,4
   20.00000
Press any key to continue
```

Le premier WRITE affiche un texte qui est une constante alphanumérique (chaîne de caractères). la constante chaîne est encadrée par deux apostrophes. Attention à ne pas introduire des apostrophes supplémentaires par inadvertance (voir § 3.1.6).

2.3 Eléments de base du langage

Lorsque l'on écrit un programme, on utilise les lettres de l'alphabet, les chiffres et quelques signes. Les caractères reconnus par le compilateur sont :

- Les 36 caractères alphanumériques:
les 10 chiffres décimaux
les 26 lettres de l'alphabet

- Les caractères spéciaux:

```
  espace typographique
= signe egal
+ plus
- moins
* etoile
/ barre de fraction
( parenthèses
)
, virgule
. point
' apostrophe
: double point
```

Chaque instruction, sauf l'instruction d'affectation (ex. : `VALEUR = 10 )`, commence par un mot-clé du FORTRAN.

2.4 Les mots-clé du langage

Généralités
Leur nombre est limité. Ils constituent le vocabulaire reconnu par le compilateur. Toute autre chaîne de caractères est considérée comme nom de procédure, de constante, de variable ou bien comme commentaire. L'annexe A comporte la liste alphabétique des mots-clé, ainsi qu'une description succincte.

Exemple de mots-clé:

```
DATA, READ, PRINT, CONTINUE, COMMON.
```

Instructions d'interface
Elles sont essentielles, car elles permettent de communiquer avec l'utilisateur. Pour l'instant, le seul moyen pratique et simple de communication se passe au travers de la console (écran-clavier).

2.5 Les séparateurs

- Les parenthèses ( )

Elles encadrent une liste ou un indice associé à un mot-clé ou à un nom de variable (dans le cas de tableaux).

- Le signe =

1. Il constitue le symbole d'affectation
2. Il précède la valeur dans la déclaration d'une constante
3. Il introduit les paramètres dans une boucle DO

- Les barres de fraction / /

Elles encadrent une liste associée à certains mots-clé comme par exemple DATA, COMMON,…

- L'astérisque *

Elle désigne la transmission hors format de données ou l'utilisation d'un périphérique standard d'entrée-sortie (écran-clavier). Elle désigne aussi l'opérateur arithmétique de multiplication et d'élévation à une puissance.

- Les apostrophes ' '

Elles encadrent les constantes de type alphanumérique

- Le point-virgule ;

Le point-virgule sépare plusieurs instructions situées sur la même ligne (F90).

Cas particulier: le caractère espace. Le compilateur l'ignore, sauf s'il se situe dans une constante ou variable alphanumérique.

Dans l'exemple ci-dessous, les trois lignes sont rigoureusement identiques, et l'absence ou la présence d'espace laisse le compilateur totalement indifférent. Seul le lecteur peut être gêné lors de la lecture de la dernière ligne.

```
TOTAL = PUHT * NBRE * TVA
TOTAL=PUHT*NBRE*TVA
T O TAL= PU HT*NBRE * T V A
```

Il est donc possible, en FORTRAN 90, de placer plusieurs instructions par ligne, à condition de les séparer par des points-virgule, mais réservez cette possibilité uniquement aux affectations d'initialisation.

Exemple :

```
A=1; B=-4.0; EPSILON=.0001
```

2.6 Mise en place des instructions

2.6.1 Structure d'une ligne FORTRAN de type classique (formatée)

De nos jours, les programmes sont tous écrits sur la console, à l'aide d'un éditeur de texte. Pour les versions de FORTRAN antérieures à 90, une ligne se décompose en quatre groupes de colonnes (total de colonnes: 80):

Colonnes 1 a 5	Réservées aux étiquettes
Colonne 6	Réservée au caractère de continuation de ligne
Colonnes 7 a 72	Instruction du programme
Colonnes 73 a 80	Ignorées par le compilateur

Attention : *Le non respect de l'usage de ces quatre groupes de colonnes est une cause fréquente d'erreurs que commet un débutant.*

2.6.2 Description des zones

Les étiquettes servent à référencer les lignes pour par exemple un saut (GOTO) ou un format d'entrée/sortie (FORMAT). Il n'est absolument pas nécessaire de respecter un ordre croissant de ces étiquettes. Elles se placent toujours à partir de la première colonne.

Les lignes de continuation sont utilisées lorsque l'on risque de dépasser la colonne 72 lors de l'écriture d'une déclaration de liste de variables ou lors d'une instruction trop longue (cas de formules complexes). Le nombre de lignes de continuation ne peut en aucun cas excéder 19. Pour mettre en place une ligne de continuation, il suffit de faire figurer dans la colonne 6 un chiffre quelconque ou tout simplement un point. Une autre possibilité est de placer au bout de l'instruction que l'on désire continuer sur la ligne suivante un caractère '&'.

Exemple :

```
      WRITE(*,*) &
       'Cette ligne ne rentre pas sur 80 colonnes, on coupe.'
```

Les colonnes 73 à 80 étaient utilisées lorsque les programmes étaient écrits sur cartes perforées pour repérer les cartes les unes par rapport aux autres. Imaginez un lâcher inopportun d'un paquet de 500 cartes sur le plancher...

Aucun signe particulier marque la fin d'une instruction. Il ne peut y avoir qu'une seule instruction par ligne. On veillera à ne pas couper un nom de variable ou un mot réservé en passant à une ligne de continuation.

Attention : *Une erreur fréquente est le dépassement de la colonne 72. Le compilateur ne le signale pas directement. Il se contente de remarquer soit une instruction incompréhensible, soit une nouvelle variable (en tronquant automatiquement la fin du mot). Le réflexe du programmeur devra être de vérifier en premier lieu la position des fins d'instruction dès qu'une erreur signalée par le compilateur paraît incompréhensible.*

Les compilateurs **FORTRAN 90** (par exemple le compilateur Microsoft) admettent le format libre de ligne. Aucune colonne n'est réservée, un commentaire commence par un point d'exclamation, la colonne 72 n'est plus une limite. Avec la plupart des compilateurs Fortran,

le fait de préciser .f90 pour l'extension d'un fichier source au lieu de .for, implique l'utilisation du format libre. Il est de même possible de placer plusieurs instructions sur la même ligne, en prenant soin de les séparer par un point-virgule (;).

2.7 Commentaires

Lorsque le caractère C ou * figure dans la première colonne, la ligne entière est supposée être un commentaire, et le compilateur l'ignore. Une ligne vide est également considérée comme commentaire par le compilateur.

Il est conseillé de faire usage autant que possible des commentaires, afin de rendre le programme plus lisible. La notice d'utilisation du module figurera sous forme de commentaire en en-tête, et de ce fait ne quittera pas le programme.

Le **FORTRAN 90** admet également des commentaires en fin de ligne d'instruction. Dans ce cas, ils commencent par un **point d'exclamation**.

2.8 Un exemple en FORTRAN 77

```
C  Voici un exemple de programme FORTRAN respectant les regles :
C
      PROGRAM BIDON   ! L'instruction PROGRAM n'est pas obligatoire
C
C Normalement, il faudrait declarer ici variables et parametres,
C mais comme il n'y en a pas...
C
      WRITE(*,*) 'Voici un message qui va etre affiche sur l ecran'
C
C Notez que l'on a volontairement oublie l'apostrophe, car elle
C delimite les constantes alphanumeriques, ce qui, dans le cas
C present, aurait provoque une erreur signalee par le compilateur
C
      WRITE(*,*)
     1 'Ceci est en fait une seule ligne (ligne de continuation)'
C
      END
C Faut pas l'oublier...
```

2.9 Les identificateurs

Un identificateur est un nom que le programmeur attribue à une variable, un sous-programme, une fonction, ou un bloc. En principe, il ne peut comporter plus de 6 caractères, mais certains compilateurs en admettent d'avantage. Si l'on utilise un identificateur faisant plus de 6 caractères, les derniers sont au pire ignorés par le compilateur, et il peut confondre deux variables distinctes. Il ne faut pas choisir pour nom de variable un mot réservé . Les mots réservés sont les instructions de structuration et des noms de fonctions standard.

Ils utilisent toutes les lettres de l'alphabet, ainsi que les 10 chiffres, mais ils ne commenceront jamais par un chiffre. On peut utiliser indifféremment majuscules et minuscules.
Les variables sont placées dans des zones mémoire repérées par leur adresse. Pour le programmeur, le fait d'ignorer l'adresse physique en mémoire d'une variable n'es pas du tout

un handicap, d'autant plus qu'il n'en désire connaître que le nom. Le compilateur se charge de faire correspondre à un identificateur une adresse mémoire. Un identificateur peut-être un nom de fonction, de variable, ou de procédure.

> Un conseil : *Pour créer un identificateur, n'utilisez que les 26 lettres de l'alphabet, les chiffres et si vous désirez inclure un espace, prenez le caractère souligné ' '.*

Les majuscules et minuscules sont confondues par le langage. On peut donc styliser les noms de variables par un usage combiné de majuscules et de minuscules. Les règles élémentaires à suivre pour la constitution de noms de variables seront décrites en annexe.

Exemples :

```
TOTAL
RACINE
I1
```

Les noms suivants ne sont pas valables (Fortran 77):

```
SOUSTOTAL     (Admissible, mais comporte plus de 6 car.)
3I9           (Commence par un chiffre)
A.B           (Comporte un point)
PAUSE         (Mot reserve du langage)
```

Pour le Fortran 90, A.B est admis dans le cas des variables structurées (STRUCTURE), voir dans le paragraphe 8.4.

3. Constantes et Variables

3.1 Constantes

3.1.1 Généralités

Les constantes apparaissent dans un programme sous la forme d'une valeur numérique ou alphanumérique. On ne peut pas leur attribuer un nom. Elles sont utilisées dans les intructions DATA et dans les initialisations.

3.1.2 Constantes entières

Définition :

C'est une suite de chiffres précédée ou non d'un signe, ne pouvant comporter aucun autre caractère.

Exemples :

```
123
-18
+4
```

Contre-exemples :

`3 14`	Présence d'un espace (séparateur)
`3.14`	Présence d'un point décimal
`2,71828`	Présence d'une virgule (séparateur)

3.1.3 Constantes réelles

On peut les trouver sous la forme simple précision ou double précision. On peut les présenter de deux manières :

- constante réelle de base : mantisse

C'est une chaîne de chiffres comportant obligatoirement un seul point décimal, pouvant être précédée d'un signe.

Exemples :

```
3.14
-6.28
.7
0.
```

- Constante réelle de base suivie d'un exposant.

La présentation reste identique à la précédente, mais on rajoute en fin de chaîne un exposant comportant pour la simple précision la lettre E suivie d'une constante entière signée de deux chiffres. La mantisse peut comporter 7 chiffres significatifs. En ce qui concerne la double précision, la mantisse comportera 15 à 16 chiffres significatifs, l'exposant commencera par la lettre D et pourra comporter 3 chiffres. Les limites sont indiquées dans le chapitre de présentation des variables.

Exemples :

```
128.34E+02
-.432E-06
1.23267453274D+03
```

Contre-exemples :

```
1,5E+01
-.23E
```

3.1.4 Constantes complexes

Une constante complexe se présente sous la forme d'un doublet de deux réels placé entre parenthèses. Les deux réels sont séparés par une virgule, le premier représentant la partie réelle, le second la partie imaginaire.

Exemples :

```
(20.0, -3.14)
(0.,0.)
(1.E01,1.23E-02)
```

Les réels formant le complexe doivent être de même nature, soit en simple ou en double précision.

3.1.5 Constantes logiques

Un constante logique n'a que deux valeurs possibles :

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

3.1.6 Constantes chaînes de caractères

C'est une suite de caractères quelconques encadrée par l'apostrophe '. On utilise ces constantes pour composer des messages ou initialiser des chaînes.

Attention : *Le français fait un usage courant de l'apostrophe, ceci peut poser un problème lors de l'affichage de certains messages.*

Exemples :

```
WRITE(*,*)'LUNDI 1 JANVIER 1990'
```

Contre-exemple :

```
WRITE(*,*)'L'affichage d'apostrophes va poser des problemes!'
```

Ce problème peut être réglé par le double apostrophage :

```
      WRITE(*,*)'L''affichage d''apostrophes ne pose plus de
probleme!'
```

3.1.7 Déclaration des constantes

En FORTRAN, les constantes n'ont pas de nom. Les déclarations de constantes n'existent donc pas. Cependant, il peut-être utile de nommer des constantes, ce qui permet de les décrire en début de programme, et de les localiser tout de suite pour une modification. On utilise pour celà la déclaration **PARAMETER.**

Exemple :

```
PARAMETER (PI=3.14159,FLAG=.TRUE.,NOMFICHIER='resultats.txt')
```

Une constante réelle se distingue d'une constante entière par le point décimal.

> **Remarque :**
> *4. sera interprété comme une constante réelle, alors que 4 sera considéré comme constante entière. Cette particularité peut conduire à des erreurs (voir § 3.3.2)*

3.2 Les variables

3.2.1 Variables simples

On représente les variables simples à l'aide de leur identificateur. Le compilateur fait correspondre à chaque identificateur une zone mémoire dépendant du type de la variable qu'il représente.

3.2.2 Les tableaux (Variables indicées)

On représente une variable indicée ou un tableau par un identificateur, suivi d'indices placés entre parenthèses et séparés par des virgules :

Identificateur(I1, I2, ...)

Exemples :

```
TABLE(I,J)
PNT(2,K,L)
A(I)
```

L'indice peut être une variable ou une expression mathématique de type réel ou entier. S'il est de type réel, il sera converti en entier. *On évitera d'utiliser un indice réel, pour des raisons de compatibilité avec d'autres langages ainsi que pour des erreurs dues aux arrondis dans les calculs*.

Exemples :

```
ALPHA(5)
ELEM(3,J+3)
MAT(I*2, I+IMAT(J,I))
```

3.3 Déclaration des variables

3.3.1 Les types de variables

FORTRAN est un langage permettant l'utilisation de 5 types de variables intrinsèques:

```
REAL                reels
INTEGER             entiers
LOGICAL             logiques
COMPLEX             complexes
CHARACTER           chaines de caracteres
```

et dans une certaine mesure :

```
EXTERNAL            identificateurs de sous-programmes
```

3.3.2 Déclaration implicite des variables

Deux types de variables sont privilégiés en FORTRAN (déclaration implicite) :

- Les variables entières
- Les variables réelles

> **IMPORTANT :**
> *Le compilateur détermine le type des variables non déclarées à partir de leur lettre initiale. Toute variable dont le nom commence par la lettre I, J, K, L, M, N est considérée comme entière et les autres comme étant réelles. Ceci reste également valable pour les tableaux. Il s'agit là de la déclaration implicite des types.*

Exemple :

```
PROGRAM exemple2
IA=3
IB=2
IC=IA/IB
WRITE(*,*) 'Division de ',IA, ' par',IB,' donne :',IC
END PROGRAM exemple2
```

Aucune variable est déclarée. Les noms des variables commencent par la lettre I, le compilateur les considère alors comme entières. IC sera une valeur entière, et le résultat affiché sera entier :

```
Division de            3  par           2  donne :           1
Press any key to continue
```

On remarquera, au passage, que la mise en forme n'est pas bonne. L'utilisation d'une instruction FORMAT (§ 6.1.5) améliorerait la présentation. En modifiant les noms des variables :

```
PROGRAM exemple2
AA=3
AB=2
AC=AA/AB
WRITE(*,*) 'Division de ',AA, ' par',AB,' donne :',AC
END PROGRAM exemple2
```

On obtient un résultat plus conforme :

```
Division de    3.000000      par   2.000000      donne :
1.500000
Press any key to continue
```

Dans le paragraphe 3.1.7, il a été évoqué la possibilité d'erreurs lors d'une mauvaise representation des constants. Voici un exemple simple :

```
PROGRAM exemple2
AA=3/2
IA=3/2
WRITE(*,*) AA,IA
AA=3./2
IA=3./2
WRITE(*,*) AA,IA
END PROGRAM exemple2
```

En sortie, on aura :

```
   1.000000              1
   1.500000              1
Press any key to continue
```

AA est considérée comme réelle, et IA comme entière. 3/2 est considéré comme le quotient de deux entiers, ce qui doit donner un résultat entier, c'est ce qui est imprimé par le premier WRITE. On remarque bien que AA est interprété comme réel et IA comme entier par le format automatique de sortie de l'instruction WRITE. L'opération 3./2 est le quotient d'un réel par un entier, ce qui donne un résultat réel. Le second WRITE montre que AA est correct, et que IA n'a pris en compte que la partie entière du résultat.

> **En conclusion :** *Une mauvaise représentation des valeurs constantes, l'utilisation des déclarations implicites (même involontairement) peut conduire à des erreurs parfois difficiles à repérer. Vérifier que toute constante réelle possède bien un point décimal.*

On peut toutefois modifier cette coutume en utilisant la déclaration
`IMPLICIT` :

`IMPLICIT` *attribut (lettre), attribut (lettre),...*

attribut peut être INTEGER, REAL, CHARACTER, COMPLEX, ...

La lettre qui suit l'attribut désignera alors le type de variable dont elle est initiale.

Exemple:

```
IMPLICIT LOGICAL(K)
```

Dans ce cas, les variables `KIND, K2, KM` sont de type logique, c'est à dire qu'elles ne peuvent avoir que deux valeurs, `.FALSE.` ou `.TRUE.` (remarquer la présence de 2 points encadrant ces valeurs).

Les déclarations de type se placent en tête de programme, avant toute instruction exécutable:

```
INTEGER A,B,C,D
REAL MAT,MU
DOUBLE PRECISION DMAT,DMU
```

Il est normalement inutile de déclarer par exemple la variable INDEX entière, étant donné qu'elle commence par la lettre I. Par contre toutes les variables double précision, logiques ou complexes doivent être déclarées.

Afin de prendre de bonnes habitudes, des langages comme PASCAL et C nécessitant la déclaration préalable de toute variable, on fera de même en FORTRAN, même si cela est parfois inutile.
En fortran 90, il est possible d'imposer l'obligation de déclaration de toutes les variables utilisées à l'aide de l'instruction IMPLICIT NONE.

CONSEIL :
Imposez-vous la déclaration initiale des variables en mentionnant en début de programme la déclaration : **IMPLICIT NONE.** *Toute variable non déclarée, donc par exemple susceptible d'être une variable déclarée, mais comportant une faute de frappe, sera signalée lors de la compilation.* ***Cette précaution vous évitera quelques heures de tâtonnements pour retrouver des erreurs.***

3.3.3 Déclaration explicite des variables simples

Les déclarations des variables se placent toujours au début du module, avant toute instruction exécutable.

3.3.3.1 Variables entières

INTEGER, INTEGER*1, INTEGER*2, et INTEGER*4

Leur domaine d'utilisation est :

- -128 à 127 (1 octet) pour INTEGER*1

- -32 768 à 32 767 (2 octets) pour INTEGER*2
- -2 147 483 648 à 2 147 483 647 (4 octets) pour INTEGER*4

Au niveau des entrées, ces constantes seront interprétées en base 10. Si l'on veut préciser la base, on peut les écrire sous la forme :

```
[signe][[base]#]valeur
```

3.3.3.2 Variables réelles

REAL ou REAL*4, DOUBLE PRECISION ou REAL*8

Domaine d'utilisation :

- -3.4028235E+38 à -1.1754944E-38, 0, 1.1754944E-38 à 3.4028235E+38 pour un réel simple précision

- -1.797693134862316D+308 à -2.225073858507201D-308, 0, 2.225073858507201D-308 à 1.797693134862316D+308 pour un réel double précision.

La forme d'une valeur réelle est :

`[signe][entier][.[fraction]][Eexposant]` pour le type REAL*4

`[signe][entier][.[fraction]][Dexposant]` pour le type REAL*8

Les réels double précision, la place mémoire nécessaire est plus importante, les temps de calcul plus longs, mais cette forme est souvent utilisée pour des problèmes demandant un grand nombre d'opérations successives, tendant à faire perdre de la précision.

La déclaration REAL*4 signifie une demande de codage d'un réel sur 4 octets; REAL*8 appelle au codage d'un réel sur 8 octets (double précision).

La déclaration est faite de la manière suivante :

REAL *liste de variables*

exemple :

```
REAL SIGMAXX, SIGMAYY, SIGMAZZ

IMPLICIT REAL*8 (D)
```

Après la déclaration IMPLICIT, toutes les variables dont le nom commence par D sont en double précision.

3.3.3.3 Variables logiques

LOGICAL ou LOGICAL*1, LOGICAL*2, LOGICAL*4

3.3.3.4 Variables complexes

```
COMPLEX ou COMPLEX*8, DOUBLE COMPLEX ou COMPLEX*16
```

Les complexes sont des doublets de réels. On les utilise sous la forme suivante :

[signe](partie réelle, partie imaginaire)

Un nombre complexe est représenté par sa partie réelle et sa partie imaginaire:

```
COMPLEX CX
CX=(5.0,1.0)
```

3.3.3.5 Variables de caractères

CHARACTER[*n] où 1 <= n <= 32767

Si l'on ignore la longueur de la chaîne, on pourra écrire la déclaration sous la forme :

```
CHARACTER*(*) chaine
```

On pourra déclarer alphanumériques toutes les variables dont le nom commencera par la lettre C en mentionnant en début de programme la déclaration suivante:

```
IMPLICIT CHARACTER (C)
```

Dans cette déclaration les chaînes sont déclarées de longueur 1. Or dans la plupart des cas, notamment dans les programmes orientés gestion, il est intéressant de pouvoir manipuler des mots. On peut déclarer des variables alphanumériques de longueur différente de 1 en écrivant:

```
CHARACTER *20 NOM,PRENOM
```

NOM et PRENOM sont deux chaînes de caractères pouvant comporter au plus 20 caractères.

On peut également déclarer plusieurs variables comme étant de longueurs différentes:

```
CHARACTER*8 NOM1, NOM2, ORIGIN*15, NOM3
```

Attention dans l'exemple ci-dessus, NOM1, NOM2 ET NOM3 ont une longueur maximale de 8 caractères, et ORIGIN est défini comme étant de longueur 15.

Généralement, on place la déclaration CHARACTER après la déclaration IMPLICIT.

3.3.3.6 L'instruction DATA

L'instruction DATA devient désuette. Elle sert à initialiser une variable ou un tableau. Il est maintenant possible d'initialiser une variable au moment de sa déclaration.
La syntaxe de l'instruction DATA est la suivante :

```
DATA liste de variables /liste de valeurs/
```

Les données à l'intérieur de chaque liste sont séparées par des virgules.
Il existe trois façons d'initialiser des variables :

- Par une affectation
- A l'aide de l'instruction DATA
- Au moment de la déclaration (FORTRAN90)

L'affectation est une commande exécutable. On opère en général une affectation par ligne

```
PROGRAM exemple1
REAL*4 A,X,Y
INTEGER*2 I,J,K
A=20.5
X=-3.0
Y=0.23
I=-4
J=-4
K=-4
PRINT*, A,X,Y,I,J,K
END PROGRAM
```

Avec l'instruction DATA, le compilateur initialise directement les variables à la valeur indiquée au moment de la compilation. L'écriture est également plus compacte. L'exemple ci-dessus devient :

```
PROGRAM exemple1b
REAL*4 A,X,Y
INTEGER*2 I,J,K
DATA A,X,Y,I,J,K /20.5,-3.0,0.23,3*-4/
PRINT*, A,X,Y,I,J,K
END PROGRAM
```

Il est également possible de grouper les variables de même valeur et de condenser la valeur d'initialisation comme il a été fait dans l'exemple pour les variables I, J, K.

3.3.4 Cas des variables indicées

Une variable indicée est constituée du nom suivi du ou des indices (de 1 à 7) placés entre parenthèses. Ces indices sont des variables entières ou des constantes entières. Les indices des tableaux commencent tous à 1, et doivent être positifs.

> **Attention :** *Une erreur fréquente est l'indiçage involontaire par rapport à 0. Le compilateur ne peut pas signaler cette erreur, sauf si l'indice est une constante. Penser à vérifier la valeur des indices.*

Ex : ELEM(I,J)=X(I)*J

Un tableau apparaît toujours avec ses indices sauf:

⊙ Dans une liste d'initialisation DATA ou d'entrée-sortie où elle désigne tout le tableau,

```
PROGRAM exemple3
REAL*4 A(3)
DATA A /1., 2., 3./
     WRITE(*,*) A
END PROGRAM exemple3
```

on aura comme résultat :

```
   1.000000       2.000000       3.000000
Press any key to continue
```

La ligne DATA initialise le tableau A entier, et la ligne WRITE écrit le tableau entier.

⊙ Lorsqu'elle est placée dans une liste d'arguments d'un sous-programme, dans quel cas elle indique l'adresse du premier élément,

Exemple :

```
PROGRAM exemple4
REAL*4 A(4,4)
DATA A /16*1.0/
     WRITE(*,*) TRACE(A,4)
END PROGRAM exemple4

REAL*4 FUNCTION TRACE(X,N)
REAL*4 X
DIMENSION X(N,N)
INTEGER I,N

     TRACE=0
     DO I=1,N
          TRACE=TRACE+X(I,I)
     END DO
END
```

⊙ Dans une déclaration de type ou de commun, la dimension étant déclarée dans une autre instruction, voir exemple précédent.

La règle de définition du type par défaut est respectée dans le cas d'une variable indicée.

Attention: *Il faut prévenir le compilateur de la place mémoire à réserver à une variable indicée à l'aide de l'instruction DIMENSION.*

Exemple:

```
DIMENSION ELEM(20,20), X(20)
```

On indique au compilateur la valeur maximale que pourra atteindre l'indice. Il est IMPOSSIBLE dans un programme principal de placer une variable dans une instruction de dimensionnement, comme par exemple:

DIMENSION A(N,N), l'allocation mémoire n'étant pas dynamique. Par contre si N a été défini par paramétrage préalable, cette écriture est alors possible.

On peut remarquer dans l'exemple précédent l'instruction `DIMENSION X(N,N)`. Elle semble aller à l'encontre de ce qui vient d'être mentionné. Mais dans cet exemple, X est un argument, ce qui signifie que ce tableau existe, donc qu'il a déjà été créé. La déclaration DIMENSION, dans ce cas, précise au compilateur simplement que X est un tableau à deux dimensions.

Il est logique de placer la déclaration de type AVANT la déclaration de dimension, le type de la variable définissant la quantité mémoire
nécessaire à un élément:

```
REAL*8 A
DIMENSION A(50)
```

On peut également condenser:

```
REAL*8 TABL(10)
```

La valeur d'un indice ne peut être ni négative, ni nulle. En principe, le programmeur n'a pas à se préoccuper de la distribution en mémoire des variables, aussi cet aspect sera passé sous silence.

Les choses évoluant, FORTRAN 90 autorise des références d'indiçage quelconques : On peut indiquer au moment de la déclaration des tableaux, les bornes d'indiçage.

Le FORTRAN 90 gère l'allocation dynamique de mémoire. Elle est particulièrement utile pour les grands tableaux. Elle permet d'utiliser au maximum la mémoire disponible. On se reportera au paragraphe 3.4 traitant de l'allocation dynamique de mémoire.

3.3.4.1 Cas des chaînes de caractères

Forme générale de la déclaration :

```
CHARACTER *n a(k1), b(k2), ...
```

ou bien :

```
CHARACTER a(k1)*n1, b(k2)*n2, ...
```

où :

a, b, ... sont des noms de variables

k1, k2, ... sont des listes d'indices maxi lorsqu'on utilise des tableaux de chaînes de caractères.

*n nombre de caractères de toutes les chaînes qui suivent

*n1, *n2 nombre de caractères de la chaîne qui les précède directement.

Exemple :

```
CHARACTER T*1, Y*3, Z(5)*10
```

T est un seul caractère
Y est une chaîne de 3 caractères
Z est un tableau de 5 chaînes de 10 caractères chacune.

3.3.4.2 Groupements de Variables : Les déclarations STRUCTURE et RECORD
Les déclarations STRUCTURE et RECORD permettent de composer des ensembles de variables faciles à manipuler. Il s'agit d'un regroupement organisé de variables de différents types sous une seule appellation. On se reportera au § 8.4 pour une étude de ces déclarations.

3.4 Allocation dynamique de mémoire : ALLOCATE

3.4.1 Généralités sur l'allocation dynamique de mémoire

La réservation en mémoire de la place occupée par les variables a lieu lors de la compilation. Il est donc IMPOSSIBLE de réserver des tableaux dont on ne connaît pas à l'avance les dimensions. Par conséquent,

```
REAL*8 TABLEAU(L,M)
```

est illicite dans un programme principal. FORTRAN 90 va plus loin, en considérant, toujours dans un sous programme ou une fonction, la variable tableau comme n'existant que pendant la durée d'utilisation du sous-programme (voir dans le paragraphe 3.5).
On peut cependant trouver ce genre de déclaration dans un sous-programme en FORTRAN77, à condition qu'il s'agisse de la déclaration d'un paramètre. Dans ce cas le tableau a déjà été créé dans le programme appelant. La déclaration donne alors au compilateur les indications nécessaires pour être en mesure de recalculer les adresses de chaque élément, et non pour créer la place mémoire.
Le FORTRAN offre maintenant une autre possibilité de gestion de la mémoire: L'allocation dynamique de la mémoire.

3.4.2 Déclaration

Dans le cas de l'allocation dynamique de la mémoire, il faut toujours déclarer le tableau, mais cette fois, on ne donne aucune dimension précise, sauf le nombre d'indices de ce tableau. La déclaration doit toujours indiquer le type du tableau (réel, entier, ...) suivie de la mention **ALLOCATABLE** ainsi que d'une indication donnant le nombre d'indices. La déclaration d'un tableau dynamique prend la forme suivante :

type nom [ALLOCATABLE] (:,:,...)

Si le tableau risque de dépasser 65536 octets, il faut le déclarer HUGE :

```
type nom [ALLOCATABLE,HUGE] (:,:,...)
```

Ceci n'est valable que pour un compilateur travaillant sous MS-DOS.
exemple pour un tableau de réels à 3 indices :

```
REAL*8 TAB [ALLOCATABLE] (:,:,:)
```

3.4.3 Allocation

Un tableau créé par allocation dynamique de mémoire autorise un indiçage quelconque, donc ne commençant pas forcément à 1. Cette possibilité est rarement utile, aussi on pourra consulter le manuel fourni avec le langage pour une utilisation éventuelle.

L'allocation de la mémoire se passe en faisant appel à la procédure **ALLOCATE**. On l'utilise comme suit :

```
ALLOCATE(tab1(dim1, dim2,...),tab2(dim3,...),...,STAT=erreur)
```

`erreur` est une variable entière qui contiendra un code erreur égal à 0 si tout s'est bien passé. Cette possibilité est facultative, mais dans le cas d'une omission, la gestion des erreurs est plus difficile.

> **Attention :** *Les tableaux dynamiques ne peuvent pas apparaître dans des déclarations comme `AUTOMATIC`, `COMMON`, `DATA`, `EQUIVALENCE`, `STRUCTURE`. Un tableau de structures peut être du type dynamique.*

3.4.4 Libération de la mémoire

L'intérêt majeur de l'allocation dynamique de la mémoire est la possibilité de la libérer pour la réaffecter à un autre tableau. la libération mémoire se fait à l'aide de la procédure `DEALLOCATE`. L'usage est le suivant :

```
      DEALLOCATE(liste tableaux, STAT=erreur)
```

La variable erreur est un entier, elle indique surtout un essai de libération d'une variable n'ayant jamais existé.

Exemple :

```
      REAL*4 tableau [ALLOCATABLE](:,:)
      INTEGER dim_x, dim_y, erreur
c  Ouverture du fichier de donnees
      OPEN(12, FILE='ENTREE.DAT',STATUS='OLD)
c  Lecture des dimensions
      READ(12,*) dim_x, dim_y
c  Allocation
      ALLOCATE(tableau(dim_x, dim_y),STAT=erreur)

      IF(erreur.NE.0) STOP 'Erreur Allocation memoire'
```

```
c  Lecture de tous les termes du tableau
       DO I=1,dim_x
          READ(12,*)(tableau(I,J),J=1,dim_y)
       END DO
       .....

c  Liberation memoire
       DEALLOCATE(tableau)
```

Lorsque l'on alloue dynamiquement de l'espace mémoire à une variable dans un sous-programme, cet espace est automatiquement libéré lorsque l'on quitte le sous programme, s'il est compilé en FORTRAN95.

3.5 Les nouvelles formes de déclaration des variables

Avec le FORTRAN 90, la déclaration des variables s'est un peu modifiée. Les déclarations classiques du FORTRAN 77 restent utilisables. FORTRAN 90 admet une forme un peu différente, simple à assimiler. Les types de variables restent identiques. Les déclarations suivantes sont similaires :

```
REAL*4 A,B,X
DIMENSION A(9), B(9,9)
INTEGER*2 I,J,K
```

Peut s'écrire :

```
REAL(4),DIMENSION(9):: A
REAL(4),DIMENSION(9,9):: B
INTEGER(2):: I,J,K
```

Ou encore :

```
REAL(4):: A(9),B(9,9)
INTEGER(2):: I,J,K
```

FORTRAN90 offre des possibilités de gestion de mémoire beaucoup plus performantes que ses prédécesseurs. Il peut automatiquement créer l'espace mémoire nécessaire au moment de l'entrée dans un sous-programme ou dans une fonction, et libérer cet espace au moment de la sortie du module. Il faut, dans ce cas, déclarer les tableaux en utilisant les arguments d'appel.

Exemple :

```
SUBROUTINE TRAITEMENT(I,J)
REAL(4), DIMENSION(I*2,J) ::B
```

L'espace occupé par le tableau B est automatiquement libéré à la sortie du module. L'allocation dynamique de la mémoire est une méthode préférable. Elle s'utilise, compte tenu des nouvelles notations, comme suit :

```
REAL(4), DIMENSION( :,:), ALLOCATABLE :: B
ALLOCATE(B(I*2,J))
…
DEALLOCATE(B)
```

```
…
ALLOCATE(B(10,30))
….
DEALLOCATE(B)
```

La première déclaration peut être écrite plus simplement :
```
REAL(4), ALLOCATABLE :: B(:,:)
```

3.6 Les pointeurs

Les pointeurs sont des variables contenant l'adresse d'une autre variable, souvent d'un tableau. Ils ont en outre la particularité de pouvoir soit prendre l'adresse d'un tableau classique, déjà existant, ou se comporter comme une variable dynamique (allocation dynamique de mémoire).

Exemple :

```
REAL(4), DIMENSION(:),POINTER :: PNTR_TAB
REAL(4), DIMENSION(50),TARGET :: TABLE
…
PNTR_TAB => TABLE
….
NULLIFY(PNTR_TAB)
…
ALLOCATE(PNTR_TAB(100))
…
DEALLOCATE(PNTR_TAB)
```

On attribue l'adresse au pointeur à l'aide de l''opérateur => qui signifie « pointer sur ». Le pointeur est libéré par l'instruction NULLIFY. L'instruction ASSOCIATED vérifie si un pointeur est bien associé avec sa cible.

3.7 Déclaration des sous-programmes et fonctions passés en argument

Lorsqu'un argument d'appel est une fonction, le processus de passage d'argument entre la routine appelante et la routine appelée n'est pas identique à celui du passage de variables. Il faut prévenir le compilateur que tel ou tel argument est un nom de sous-programme ou de fonction, ce qui est fait à l'aide d'une déclaration EXTERNAL toujours placée dans le programme appelant.

La forme générale d'une telle déclaration est :

```
EXTERNAL a, b, c, ...
```

a, b, c, .. sont les noms des sous-programmes ou de fonctions.

3.8 Les Interfaces

Les interfaces sont des modules descriptifs, notamment de sous programmes et de fonctions, destinés au compilateur. Ces modules décrivent la manière d'appeler un sous programme ou une fonction, en énumérant la liste des paramètres. Le compilateur sait alors si les appels faits aux différents sous-programmes sont corrects ou non.

Un bloc interface commence par l'instruction INTERFACE et se termine par END INTERFACE. Généralement, on place les blocs Interface dans un fichier. Ce fichier est chargé en plaçant l'instruction USE tout de suite après la ligne de début du programme ou sous-programme. L'interface décrit le sous-programme ou la fonction simplement en reprenant la première et la dernière ligne (par un simple copier-coller). Entre ces deux lignes, on place les déclarations des arguments.
Le bloc INTERFACE se place en tête de programme ou sous-programme. Un moyen plus puissant consiste à placer tous les blocs INTERFACE dans un module et de le sauver avec une extension .f90. Cette méthode permettra également d'éviter l'utilisation de blocs COMMON.
Exemple d'emploi d'un bloc interface :
Le programme appelle un sous programme qui remplit deux tableaux (angle et sinus correspondant).

```
PROGRAM exemple9
REAL(4),ALLOCATABLE::TABLE(:),ANGLES(:)
INTEGER(2)::Nb,I

     Nb=5
     ALLOCATE(TABLE(Nb),ANGLES(Nb))
     CALL TABLESIN(ANGLES,TABLE,Nb)
     DO I=1,Nb
          WRITE(*,*)i,ANGLES(I),TABLE(I)
     END DO
     DEALLOCATE(TABLE,ANGLES)
END PROGRAM

SUBROUTINE TABLESIN(A,V,N)
INTEGER(2):: N
REAL(4):: A(N),V(N)
REAL(4):: PIS2, INCR
INTEGER(2)::I
     PIS2=ATAN(1.0)*2
     INCR=PIS2/(N-1)
     DO I=1,N
          A(I) = (I-1.)/(N-1)*90
          V(I) = SIN((I-1)*INCR)
     END DO
END SUBROUTINE
```

Le programme est le sous-programme sont placés dans des fichiers différents et compilés. Il n'y a pas d'erreur signalée à la compilation. Le résultat affiché par le programme donne :

```
       1  0.0000000E+00  0.0000000E+00
       2  0.3926991      0.3826835
       3  0.7853982      0.7071068
       4   1.178097      0.9238795
       5   1.570796       1.000000
Press any key to continue
```

En changeant la ligne d'appel du sous programme de la manière suivante, la compilation et l'éxécution du programme se passent correctement :

```
     CALL TABLESIN(ANGLES,TABLE,5)
```

Le dernier paramètre est cette fois une constante. Si l'on écrit ce dernier argument sous la forme d'un réel , il n'y a toujours pas d'erreur de compilation :

```
     CALL TABLESIN(ANGLES,TABLE,5.)
```

mais, les résultats sont un peu différents :

```
      1 -4.3160208E+08 -4.3160208E+08
      2 -4.3160208E+08 -4.3160208E+08
      3 -4.3160208E+08 -4.3160208E+08
      4 -4.3160208E+08 -4.3160208E+08
      5 -4.3160208E+08 -4.3160208E+08
Press any key to continue
```

Cette erreur est typique d'un défaut de passage d'arguments. En effet, normalement, le sous programme attend comme dernier argument une valeur sur 2 octets, INTEGER(2), or, le compilateur a interprêté la valeur 5. comme un nombre réel, codé sur 4 octets. On remarque qu'il est aisé de commettre une erreur de taille sur les arguments. Une erreur fréquente consiste à confondre INTEGER(2), INTEGER(4) et INTEGER. La déclaration INTEGER produit un entier sur 4 octets, ou rarement sur 2, selon les paramétrages du compilateur. Il est important de surveiller le type et l'ordre des arguments. En ajoutant un bloc INTERFACE, le compilateur détecte une incohérence :

```
PROGRAM exemple9
INTERFACE
     SUBROUTINE TABLESIN(A,V,N)
     INTEGER(2):: N
     REAL(4):: A(N),V(N)
     END SUBROUTINE
END INTERFACE
REAL(4),ALLOCATABLE::TABLE(:),ANGLES(:)
INTEGER(2)::Nb,I
     Nb=5
     ALLOCATE(TABLE(Nb),ANGLES(Nb))
     CALL TABLESIN(ANGLES,TABLE,5.)
     DO I=1,Nb
          WRITE(*,*)i,ANGLES(I),TABLE(I)
     END DO
     DEALLOCATE(TABLE,ANGLES)
END PROGRAM

exemple9.f90(12) : Error: The type of the actual argument
differs from the type of the dummy argument.   [5.]
 CALL TABLESIN(ANGLES,TABLE,5.)
---------------------------^
```

Mais l'interet n'est pas evident : Ecrire 6 lignes de programme supplémentaires pour éviter une erreur dans une ligne… En fait, on place les blocs INTERFACE dans un module que l'on appelle par l'instruction USE.

3.9 Les Modules

Les modules sont des unités de programme, comme les programmes, les sous-programmes et les fonctions. Ils sont destinés à remplacer les blocs COMMON, Permettent ainsi de partager des variables entre différents modules éxécutables (programme, sous-programmes et

fonctions). On y retrouve les interfaces des différents sous-programmes et fonctions utilisés dans le projet. Une simple ligne d'instruction USE suffit pour les utiliser.
La forme générale d'un module est :
MODULE nom_module
Déclarations de variables globales
Déclarations d'interfaces
END MODULE nom_module

4. Programmes et sous-programmes

4.1 Principe général

Le programme principal et chaque sous-programme sont analysés séparément par le compilateur. Ils peuvent donc figurer soit dans un même fichier, soit dans des fichiers séparés. Il est préférable de faire un fichier par sous-programme. Au moment de la mise au point de petits programmes, il est plus facile de faire figurer le programme principal et les sous programmes ensemble, puis de les séparer lorsque la compilation et les essais d'exécution ont réussi.

L'éditeur de liens réalise la construction globale du programme et des sous-programmes. Il fait également des tests de compatibilité au niveau des passages des arguments.

Dans le cas où l'on fait figurer dans un même fichier programme et sous-programmes, le programme principal doit figurer en tête, suivi des sous-programmes, chacun se terminant par END, pour bien délimiter les blocs.

> **Attention :** *Prendre l'habitude de faire suivre l'instruction PROGRAM, SUBROUTINE ou FUNCTION par l'instruction IMPLICIT NONE. Elle rend obligatoire la déclaration de toute variable utilisée dans le module. Cette précaution évite souvent des erreurs difficiles à détecter.*

On distingue 4 types de modules :

PROGRAM : Programme principal, tout module dont la première instruction n'est pas SUBROUTINE, FUNCTION ou BLOCK DATA. Ce module peut comporter comme première instruction PROGRAM, mais ce n'est pas obligatoire.

SUBROUTINE : Ce module est un sous-programme toujours appelé par un CALL à partir d'un autre module de type programme ou sous-programme.

FUNCTION : Ce module est simplement appelé par son nom à partir d'un autre module.

BLOCK DATA : Ce module initialise des variables placées dans un COMMON nommé. Se référer au chapitre des déclarations.

4.2 Programme principal

Un programme comporte 2 parties :

- Les déclarations concernant les variables,
- Les instructions exécutables.

On peut placer au début du programme l'instruction PROGRAM *Nom du programme*

Cette instruction est facultative, le compilateur n'en a nullement besoin, mais en fin de programme, on fera **toujours** figurer l'instruction END.

Structure globale :

```
PROGRAM nom du programme

Déclarations des variables utilisées

Instructions exécutables

END
```

Attention : *On ne peut pas placer de déclaration après une instruction exécutable.*

Arrêt du programme : Instruction **STOP**

Cette instruction est facultative pour une fin normale de programme. Elle signifie fin de l'exécution. La main est alors retournée au shell (superviseur). On peut placer des instructions STOP n'importe où dans le programme, mais de préférence on n'en placera qu'une juste avant l'instruction END.

Fin du programme : instruction **END**.

L'instruction STOP n'est pas obligatoire. En revanche, l'instruction END permet au compilateur de délimiter les différents modules (programme et sous-programmes) figurant dans un même fichier. Si l'on omet l'instruction END à la fin d'un module, le compilateur indiquera une erreur. L'instruction END est exécutable depuis la version 77 du FORTRAN. Elle remplace l'instruction STOP à la fin du programme principal, et l'instruction RETURN à la fin d'une fonction ou d'un sous-programme.

Conseil : *N'utilisez plus les instructions* **STOP** *et* **RETURN**

Le programme principal se voit assigner le nom `_main`, aussi si l'éditeur de liens vous signale pour le FORTRAN Microsoft, l'absence de `_main`, le programme principal a tout simplement été oublié.

4.3 Sous-Programmes

Il existe en FORTRAN quatre sortes de sous-programmes :

⊙ Les fonctions internes, qui ne sont pas réellement des sous-programmes, car intérieurs à un programme donné.

```
PROGRAM exemple5
REAL F,X
INTEGER I
     F(X)=X*X-1
     DO I=-5,5
          X=.5*I
          WRITE(*,*) X,F(X)
```

```
      END DO
END
```

- Les fonctions qui sont appelées par leur référence dans une expression mathématique et fournissent un résultat numérique. Elles sont perçues comme des variables "furtives".

- Les fonctions implicites qui font partie du langage. Elles se trouvent dans les librairies du FORTRAN. Ce sont par exemple les fonctions mathématiques du genre SIN(x), SQRT(x), etc.

- Les sous-programmes qui sont appelés par un CALL et peuvent fournir plusieurs résultats. Ces résultats figurent obligatoirement parmi les arguments.

4.3.1 Structure d'un sous-programme type SUBROUTINE

```
SUBROUTINE nom(liste des arguments)

Déclarations des arguments

Déclarations des variables

Instructions exécutables

RETURN

END
```

L'instruction **RETURN** redonne la main au programme appelant. On peut placer plusieurs instructions RETURN dans un même module, mais pour des raisons de structuration et de lisibilité, on évitera cette pratique. Faire figurer cette instruction de préférence à la fin du module, ou alors l'oublier, car elle n'est plus nécessaire pour les versions actuelles du FORTRAN.

Il faut déclarer les arguments comme s'il s'agissait de variables. Le compilateur ne réserve pas la place mémoire, puisque ceci a été fait dans un module hiérarchiquement supérieur. En fait, quand le nom d'une variable figure simultanément dans la liste des arguments en entrée et dans une déclaration, le compilateur utilise uniquement les informations de type fournies par la déclaration.

Arguments et variables peuvent figurer dans une même déclaration. Il est préférable, uniquement pour des raisons de clarté du module, de séparer les déclarations des arguments et des variables.

La forme générale de l'appel d'un sous-programme du type SUBROUTINE sera :

```
CALL nom du sous-programme(liste des arguments)
```

Les arguments peuvent être :

- des constantes de tous types

- des expressions arithmétiques ou logiques
- des noms de fonctions ou de sous-programmes
- des variables simples ou indicées (éléments de tableaux)
- des noms de tableaux

En sortie, le sous-programme peut modifier :

- les variables simples
- les variables indicées

> **Attention :** *Au moment de l'appel, les arguments doivent être identiques en nombre, ordre, type, aux arguments du sous-programme.*

Le passage d'un tableau entier comme paramètre, se fait en donnant uniquement son nom, non muni de parenthèses et d'indices. Dans ce cas, le tableau doit être redéclaré dans le sous-programme.

Exemple d'appel de sous-programme :

```
CALL SPROG(A,MAT,10)
```

Cas où un paramètre est un nom de fonction ou de sous-programme :

Lors d'un passage d'arguments, il faut préciser au compilateur la nature des arguments passés (s'agit-il de simples variables ou de fonctions ?). On utilise alors la déclaration **`EXTERNAL`** en tête du programme appelant.

Exemple :

```
        EXTERNAL COS, SIN
...
...
        CALL INTEG(1., 2., COS, SOM1)
        CALL INTEG(0., 1., SIN, SOM2)
...
```

Dans l'exemple ci-dessus, on précise que COS et SIN ne sont pas des variables, mais des fonctions.

4.3.2 Structure d'un sous-programme du type FUNCTION

```
type FUNCTION nom(liste d'arguments)

Déclarations

Instructions

nom = ... ! Instruction 'Retourner' en algo
```

```
RETURN ! Facultatif

END
```

Remarquer 2 choses :

- L'entête avec sa déclaration de type, car une fonction retourne une valeur dont il faut déclarer le type

- L'instruction de mise en place de la valeur retournée : (nom = ...). Elle correspond à l'instruction `retourner resultat` de l'algorithmique.

L'instruction `END` délimite les blocs, et l'instruction `RETURN` n'est plus obligatoire pour les versions actuelles du FORTRAN.

4.4 Variables et arguments

Un sous-programme ou une fonction peut reprendre des étiquettes et des noms déjà utilisés par ailleurs, sans pour autant affecter le comportement du programme. En quelques mots, un sous-programme peut être glissé après n'importe quel autre programme sans précaution particulière comme par exemple une modification des étiquettes. La communication entre le programme appelant et le sous-programme est assurée au moyen des arguments d'appel et de l'instruction `COMMON`.

Le passage des arguments dans la liste se fait par référence. Le sous-programme peut donc travailler directement dans la mémoire du programme appelant.

Deux variables de même nom situées dans deux procédures différentes seront indépendantes, sauf si elles sont liées par un passage de paramètre, une déclaration `COMMON`, ou une déclaration `EQUIVALENCE`.

Dans un sous-programme, on utilise deux types de variables :

1. Les variables propres au module:

Le compilateur réserve en mémoire la place que nécessitent les variables locales. Il convient donc de les déclarer (surtout quand le type de la variable ne correspond pas avec le caractère initial). En fait, pour une plus grande rigueur de programmation, elles devraient toutes être déclarées. Utiliser la déclaration `IMPLICIT NONE` pour forcer la déclaration des variables. Les tableaux locaux devront donc être déclarés avec une dimension numérique ou déclarés dynamiques.

2. Les arguments :

Ils sont fournis par le module appelant. Leur place mémoire n'est donc plus à réserver, le type de passage de paramètre étant par référence (par adresse). Le sous-programme travaille donc directement dans la mémoire du programme principal lorsqu'il utilise les arguments.

Mais la déclaration reste cependant nécessaire, elle servira à fixer le type de chaque variable s'il peut y avoir ambiguïté. Une déclaration concernant un paramètre ne provoquera dont pas une réservation mémoire. En ce qui concerne les tableaux, il faut obligatoirement les dimensionner. Le dimensionnement d'un tableau à un seul indice ne pose pas de problème, car tous ses éléments se suivent en mémoire. La redéclaration d'un tel tableau peut se faire comme suit :

```
DIMENSION A(1)
```

Le compilateur ne se préoccupe pas de la dimension réelle dans ce cas. Tout élément A(I) pourra être retrouvé de manière simple, étant donné la disposition linéaire en mémoire des éléments. On peut donc déclarer un tableau simple en indiquant qu'il ne contient qu'un seul élément. Le problème est tout autre pour un tableau à deux dimensions ou plus. L'accès à un élément a(i,j) dépend du nombre de lignes et de colonnes. La déclaration peut alors être faite en utilisant des variables en indice, mais uniquement dans les sous-programmes. Attention, ces variables doivent également figurer dans la liste des arguments.

Exemple pour le cas d'une matrice carrée passée en paramètre :

```
C Programme principal
      PROGRAM TOTO
C     ...
      DIMENSION A(50,50)
      N = 50
C     ...
      CALL SPROG(A,N)
C     ...
      END

C Sous-programme
      SUBROUTINE SPROG(B,M)
      DIMENSION B(M,M)
C     ...
      RETURN
      END
```

Attention M devra être rigoureusement identique à la valeur de la dimension déclarée dans le programme principal.

Dans la liste des arguments, les bornes des tableaux doivent précéder les tableaux auxquels elles sont associées.

Ex: Calcul d'une intégrale par la méthode des trapèzes.

L'intégrale s'exprime sous la forme:

$$\Delta x\left[\tfrac{1}{2}(f(a)+f(b))+\sum f(a+i\Delta x)\right]$$

Le calcul de l'intégrale est effectué par une fonction que l'on appellera TRAP. Cette fonction pourra être compilée séparément et liée au programme appelant au moment de l'édition des liens.

Le programme qui suit sera appelé PROG.FOR:

```
        FUNCTION GAUS(T)
        GAUS= EXP(-T*T/2.)
        RETURN
        END
c
        EXTERNAL GAUS
c
        WRITE (*,*) 'Donner EPS1 et EPS2...'
        READ (*,*) EPS1,EPS2
        Y=TRAP (-10.,10.,GAUS,EPS1)
        Z=TRAP (-10.,10.,GAUS,EPS2)
        WRITE(*,10) Y,Z
10      FORMAT (2F12.4)
        END
```

On peut dissocier le module fonction, et le compiler séparément sans problème particulier. On pourra essayer de déplacer la déclaration EXTERNAL avant la fonction et surveiller les messages d'erreur qui ne manqueront pas d'apparaître. Si l'on oublie de mentionner GAUS comme étant externe, la fonction TRAP ne recevra pas les données nécessaires au calcul de la fonction GAUS et le programme "plantera", même si aucun message d'erreur n'est apparu durant la compilation ou l'édition des liens.

Comme déjà dit, on aura même intérêt à compiler séparément la fonction d'intégration par la méthode des trapèzes judicieusement appelée TRAP() et la réunir au programme appelant seulement au moment de l'édition des liens (LINK ou ld). Si l'on dispose d'une foule de petits utilitaires, on aura certainement avantage à se créer sa propre librairie de fonctions et sous-programmes à l'aide d'un libraire, mais ceci est une autre histoire...

La fonction TRAP(), qui sera nommée TRAP.FOR est donnée ci-dessous. On la compilera séparément:

```
        FUNCTION TRAP(A,B,F,DX)
c================================================
c       A et B sont les bornes d'integration
c       F est la fonction a integrer
c       DX est le pas de l'integration
c================================================
c
        N=(B-A)/DX-.5
c       N est le nombre de pas necessaires (entier >= 1)
        TRAP=(F(A) + F(B))/2.
        X=A
c
        DO 10 I=1,N
        X=X+DX
```

```
        TRAP=TRAP+f(X)
10      CONTINUE
        TRAP=TRAP*DX
        RETURN
        END
```

Le lecteur compilera les différentes parties séparément, puis essaiera de les rassembler en un seul bloc pour les compiler en une seule fois. Prendre soin de noter les différences au niveau de la programmation. A titre d'exercice, refaire la manipulation en adoptant la méthode de SIMPSON.

En général, le FORTRAN est fourni avec un environnement de développement qui permet l'édition des différents modules et la gestion de projet. Le descripteur de projet fait apparaître tous les fichiers source (.for et .f90) qui composent le projet.

5. Algorithmique et FORTRAN

5.1 Noms de blocs

En algorithmique, on décrit les programmes à l'aide de différents modules. Ces modules sont des programmes, des sous-programmes ou des fonctions. Ils sont délimités par les déclarations de début et de fin. Les déclarations des variables et arguments suivent obligatoirement la déclaration de début de module et **précèdent** toute instruction exécutable.

> **Attention :** *Une erreur fréquente signalée par le compilateur est l'apparition d'une déclaration après une instruction exécutable.*

5.2 Déclarations

5.2.1 Les déclarations de modules

Elles sont déjà connues :

```
PROGRAM, FUNCTION, SUBROUTINE, BLOCK DATA
```

5.2.2 Les déclarations de type

Elles sont utilisées aussi bien pour les variables locales que pour les arguments. Citons pour mémoire les plus fréquentes :

```
LOGICAL, CHARACTER, INTEGER, REAL, COMPLEX, RECORD
```

5.2.3 les déclarations d'agencement mémoire

Ce sont les déclarations `EQUIVALENCE, COMMON, DIMENSION, STRUCTURE, ALLOCATABLE...` Elles influent l'organisation en mémoire des variables.

5.3 Constructions

5.3.1 Constructions séquentielles

5.3.2 Constructions alternatives

5.3.2.1 Si... Alors... Sinon... Fin Si

L'action conditionnelle simple: l'instruction **IF**.

Le résultat d'un test IF est logique. Sa valeur peut être .TRUE. ou .FALSE. . L'action suivant le test est donc soit ignorée, soit exécutée. Le test simple se présente sous la forme suivante:

```
IF (expression à résultat logique) instruction
```

Si l'expression à résultat logique donne un résultat vrai, alors l'instruction qui la suit est exécutée, sinon elle est ignorée. L'instruction à exécuter doit être placée sur la même ligne. **Il ne faut pas mettre de END IF**.

Exemple :

```
IF(I.GE.N) STOP 1
```

Cependant, il arrive qu'une instruction soit insuffisante. Pour éviter des blocs d'instructions délimités par des GOTO, comme c'était souvent le cas dans les anciennes versions de FORTRAN, on dispose de la séquence `IF .... THEN / ENDIF`

`THEN` marque le début d'une séquence d'instructions délimitée par l'instruction ENDIF

Exemple :

```
        IF(I.LE.N) THEN
                I=I+1
                MOT=''
        ENDIF
```

On l'utilise donc sous la forme:

```
        IF (Expression à résultat logique) THEN
                Séquence d'instructions
        ENDIF
```

Le THEN est placé sur la même ligne que le IF. Sur les lignes suivantes est disposée la séquence d'instructions à exécuter si la condition du IF est vraie. Cette séquence se termine impérativement par un END IF.
Un enrichissement complémentaire: `ELSE` (sinon). L'algorithme d'utilisation est:

```
Si (expression vraie) alors faire:

        Sequence 1

Sinon faire:

        Sequence 2

Fin de la boucle conditionnelle.
```

Il s'agit là d'une séquence alternative. On peut donner comme exemple:

```
        READ(*,*) N1, N2
        IF (N2.EQ.0) THEN
                WRITE(*,*) 'ATTENTION, LE DIVISEUR EST NUL'
        ELSE
                WRITE(*,*) 'LE QUOTIENT DE',
1               N1,'PAR',N2,'VAUT',N1/N2
        ENDIF
```

On peut également imbriquer les constructions alternatives.

5.3.2.2 Séquence Cas où

La séquence cas où existe en FORTRAN 90 sous la forme suivante :

```
SELECT CASE (selecteur)
    CASE (Expr.1)
        séquence 1
    CASE (Expr.2)
        séquence 2
    ...
    [CASE DEFAULT
        séquence défaut]
END SELECT
```

Dans ce cas, les expressions peuvent couvrir différentes formes. On peut combiner des intervalles de valeur dans un cas où :

```
CASE (10:14, 'A':'Z', '_')
```

Si le sélecteur possède une valeur comprise entre 10 et 14, ou entre les codes ASCII de A et Z ou est égal au code ASCII de _, la séquence suivant l'instruction CASE sera exécutée.

5.3.3 Constructions itératives

5.3.3.1 Boucle Pour... Fin Pour

Une boucle est une suite d'instructions que le programme va parcourir un certain nombre de fois. Le mot-clé d'une boucle est: **DO**

Syntaxe:

```
DO étiquette I = N1,N2,INCR

....

étiquette CONTINUE
```

Les éléments d'une boucle DO sont :

étiquette : Elle détermine l'étendue de la boucle. Souvent elle pointe l'instruction CONTINUE, mais ceci n'est plus obligatoire.

I : C'est le compteur. Cette variable doit obligatoirement être entière. Il s'incrémentera de 1 à chaque bouclage, partant de N1 et s'arrêtant à la valeur N2.

N1 et N2 : Bornes du compteur de boucle.

INCR : Un compteur de boucle peut évoluer non seulement par incrément de 1, mais aussi se décrémenter, ou évoluer par pas entiers constants, mais quelconques. Cet élément est facultatif. Son absence sous entend un incrément de 1.

Attention, les boucles multiples doivent être imbriquées et les indices différents.

Exemple d'utilisation d'une boucle

```
c  boucle elementaire

        DO 10 I = 1, 10
                WRITE(*,*) I
10      CONTINUE

c lecture simple d'un tableau dans un fichier sur unite
c 7

        DO 100 I = 1,L
                DO 50 J = 1,M
                        READ(7,*)A(I,J)
50              CONTINUE
100     CONTINUE
```

On peut également trouver la boucle Pour sous la forme :

```
DO index = min, max [,pas]

        instructions

ENDDO
```

L'exemple précédent s'écrit alors :

```
        DO I = 1,L
                DO J = 1,M
                        READ(7,*)A(I,J)
                END DO
        END DO
```

Dans les boucles do l'index peut être un réel ou un entier. Mais pour des raisons de purisme, on n'utilisera que des index entiers.

5.3.3.2 Boucle Tant... Fin Tant Que

Cette boucle traduit la construction suivante :

```
Tant Que condition vraie faire

     Instructions à répéter

Fin Tant Que.
```

On peut la trouver sous deux formes, quand elle existe :

```
     DOWHILE (expression conditionnelle)
```

```
          instructions
     ENDDO
```

ou bien :

```
     WHILE etiquette (expression conditionnelle)
          instructions
xx   CONTINUE
```

Cette construction n'existe pas sur tous les compilateurs. On peut alors la créer en se basant sur les boucles Itérer :

```
Tant Que (condition pour rester dans la boucle)
     Instructions
Fin Tant que
```

est équivalent à :

```
Iterer
     Sortir si (condition de sortie de boucle)
     Instructions
Fin Iterer
```

On la traduira en FORTRAN par une boucle Itérer :

```
C       Tant que non condition de sortie
et1     IF (condition de sortie) GOTO et2
                ...
                Instructions de boucle
                ...
                GOTO et1
et2     CONTINUE
C       Fin Tant que
```

et1 et et2 sont des étiquettes numériques

Les commentaires sont utiles pour mettre en évidence la construction.

5.3.3.3 Boucle Faire ... tant que ou jusqu'à ce que

La construction `Faire ... Tant Que` n'existe pas en FORTRAN. Dans ce cas, il conviendra également de la construire à l'aide des instructions `GOTO` et `IF`:

```
Faire

        Instructions de boucle

Tant que condition vraie
```

En FORTRAN :

```
C       Faire
et1     CONTINUE
     ...
        instructions
        ...
        IF (condition vraie) GOTO et1
C       Tant Que condition vraie
```

5.3.3.4 Boucles indéterminées ou boucles ITERER

Il est possible en, FORTRAN 90, d'écrire des boucles iterer généralisées. La condition de sortie se posera alors sous la forme :

```
       IF(condition de sortie) EXIT
```

La boucle suivante :

```
     DO WHILE(I .LE. 100)
        sequence
     END DO
```

devient alors :

```
     DO
     IF(I .GT. 100) EXIT
        sequence
     END DO
```

6. Les Entrées-Sorties

6.1 Instructions de lecture et d'écriture simples

6.1.1 Généralités

Elles revêtent un caractère important, car elles permettent la communication avec l'utilisateur ou avec des unités particulières comme les MODEMS (liaisons téléphoniques), fichiers sur disque, ou sur des consoles particulières.

Quatre questions essentielles se posent pour cette opération:

1. Dans quel sens (entrée ou sortie)?
2. Sur quel périphérique?
3. Faut-il une conversion, un formatage?
4. Echanger quoi?

Une entrée de données correspond à l'instruction READ, une sortie à WRITE, comme nous avons déjà pu le constater.

Le fortran attribue à chaque périphérique un numéro d'unité logique. Fortran attribue le caractère * à la console.

Le format d'édition qui peut-être considéré comme étant la mise en forme est conditionné par l'instruction FORMAT qui est référencée par une étiquette.

L'instruction d'entrée-sortie est suivie de la liste des données à transmettre.

Exemple:

```
READ(unité logique, étiquette du format) liste de variables

WRITE(unité logique, étiquette du format) liste de variables
```

Les assignations standard des unités logiques varient selon le type de compilateur. Pour les FORTRAN Microsoft ou COMPACQ, elles sont:

- * ou Astérisque, représente la console
- 0 Représente initialement la console, mais peut également représenter un fichier sur le disque
- 5 Représente initialement le clavier
- 6 Représente initialement l'écran

La constitution des formats sera vue plus loin.

On peut utiliser le signe * pour désigner l'unité logique standard qui est la console et le FORMAT libre ou automatique. C'est généralement ce que fera le débutant, ou lorsque l'on veut écrire rapidement un programme.

Exemples de lecture et d'écriture standard :

```
READ(*,*)A,B,C
WRITE(*,*)A,B,C
```

En conclusion, l'utilisation des instructions READ et WRITE standard est courante lors de la réalisation premier jet d'un programme.

6.1.2 Instruction de lecture

La forme générale simple d'un ordre de lecture est :

READ(*constante entière*,*) {*liste des variables destination*

La constante entière désigne le fichier source d'où proviennent les données. Rappelons que le système considère également les périphériques (console) comme étant un fichier déjà ouvert. Cette constante est comprise entre 1 et 99. Voir le paragraphe précédent pour les assignations standard.

A chaque ordre READ, une nouvelle ligne du fichier à lire est examinée.

Le signe * signifie que les valeurs à lire peuvent avoir été écrites sans contrainte de cadrage et de présentation. Il suffit de faire apparaître les données dans l'ordre de lecture, et de les séparer par des blancs, tabulations, ou par des virgules. On peut cependant effectuer des lectures sur des présentations plus complexes, mais dans ce cas, il faut utiliser une référence à une ligne FORMAT à la place de l'étoile.

La liste des variables définit celles qui seront affectées lors de l'opération de lecture, les valeurs étant attribuées dans l'ordre de l'apparition des variables.

Les données entrées devront être du même type que les variables destination.

> **Attention :** *Lorsque l'on utilise une entrée formatée, les caractères blancs sont lus comme étant des zéros, ce qui peut avoir de graves conséquences lorsque ces blancs sont situés après les chiffres exprimés*.

Exemple :

On fait une lecture avec la spécification I4 (voir dans le paragraphe FORMAT)

(Le caractère ‘espace’ est matérialisé par le caractère b)

```
1 2 3 4
b 1 2 b
```

Au lieu de lire 12 comme espéré, on lit 120 ! On préférera le format libre obtenu avec *.

6.1.3 Instruction d'écriture

La forme générale de l'ordre d'écriture au format standard est :

```
WRITE(constante entière,*) liste de données
```

Les données sont des constantes ou des variables séparées par une virgule.

A chaque instruction WRITE, une nouvelle ligne est écrite sur le fichier destination. Le fichier destination est repéré par une constante entière (voir l'instruction READ).

Le signe * signifie qu'une présentation standard automatique est désirée (FORMAT standard). Il est utilisé pendant la mise au point des programmes. Lors de la finition et de l'amélioration de la présentation, l'instruction d'écriture sera toujours référencée à une ligne FORMAT.

Pour une sortie formatée de résultats, on utilisera la forme suivante :

```
WRITE(numéro unité, étiquette ligne FORMAT) liste variables
```

Exemple :

```
      IMP=2
      WRITE (IMP,100) I,J,A,B
100   FORMAT (2I4,F12.4,3X,E15.8)
```

On peut également utiliser une forme plus condensée, mais identique :

```
      IMP=2
      WRITE (IMP,'(2I4,F12.4,3X,E15.8)') I,J,A,B
```

6.1.4 Les boucles implicites dans les instructions le lecture-écriture

Le FORTRAN offre une facilité en ce qui concerne les entrées-sorties de valeurs de type tableau : les boucles implicites.
Les boucles implicites s'utilisentde la manière suivante :

```
WRITE(*,*) (TABLE(I),I=1,N)
```

Une autre façon d'affichage du tableau est :

```
DO I=1,N
     WRITE(*,*) TABLE(I)
END DO
```

Chaque fois qu'un WRITE est réalisé, il y a saut de ligne, ce qui signifie que dans le second exemple, TABLE est affiché en colonne, c'est-à-dire avec une valeur par ligne. Avec la boucle implicite, l'instruction WRITE apparaît comme étant suivie par la liste complète des

éléments du tableau. Pour l'affichage de tableaux complets, et de taille raisonnable, on peut utiliser la construction suivante :

```
DO I=1,M
     WRITE(*,*) (TABLE(I,J), J=1,N)
END DO
```

Les boucles implicites se rencontrent également avec l'instruction READ. La lecture des données en tableau se fait souvent avec une boucle implicite placée à l'interieur d'une boucle explicite.
On peut également touver des boucles implicites dans une ligne d'instruction DATA.

6.1.5 Formats de lecture-écriture

Les entrées sorties peuvent se faire en format libre, ou référencées à des lignes FORMAT. Ces lignes indiquent en fait un canevas à suivre pour la présentation des résultats.

Les lignes FORMAT contiennent les spécifications de conversion, des messages, des commandes d'imprimante ou de gestion simple du curseur.

Elles peuvent se placer n'importe où dans le programme, mais on les placera de préférence à la fin du module, juste avant l'ordre END.

Les spécifications de conversion sont de la forme :

> ***n*I*m*** **:** Cette spécification s'applique aux variables entières. m indique le nombre de caractères à imprimer, ou le nombre de caractères à lire sur le fichier. n représente le nombre de répétitions de cette spécification. Dans l'exemple précédent, 2I4 s'applique aux variables I et J; ces quantités étant imprimées avec un maximum de quatre chiffres.

On aurait pu écrire la ligne FORMAT de la manière suivante:

```
100 FORMAT (I4, I4, F12.4, 3X, E15.8)
```

> ***n*F*l.d*** **:** Cette spécification s'applique aux données réelles. n garde le même rôle que précédemment. l représente le nombre de caractères maximal, y compris le signe et le point. d représente le nombre de chiffres à imprimer après la virgule.

Exemple :

Soit le caractère -123.4567 à imprimer.

Nous obtiendrons:

```
F9.4    -123.4567
F11.4   bb-123.4567
```

```
F8.4     ******** erreur
F13.6    bb-123.456700
F6.0     b-123.
```

Dans le morceau de programme précédent, F12.4 s'applique à la variable A.

***n*E*l.d* :** Lorsque l'on ne connaît pas l'ordre de grandeur de la valeur, il est préférable de demander en impression la notation scientifique. A l'affichage apparaît un E suivi de l'exposant de la puissance de Dix. Il faut que l soit >= d+7.

Exemple : pour obtenir un affichage du type 0.xxxx, on utilise le format passe-partout E15.8.

***n*D*l.d* :** Est l'équivalent du précédent, mais pour des quantités données en double précision. La formule passe-partout serait: D23.16.

***n*G*l.d* :** Recouvre I, F, E ou D. la sortie se fait dans le format convenant le mieux à la valeur.

Pour les nombres complexes, on emploie deux spécifications de type F, E, ou G.

***n*A*l* :** Permet la manipulation de chaînes de caractères. l représente le nombre de caractères à afficher.

***n*X** : Insère n blancs ou permet de sauter n caractères à la lecture.

***n*H<*Caractères*>** n désigne le nombre de caractères et Caractères est une chaîne de n caractères directement accolée à H.

Ex : Supposons que les variables A et B valent respectivement 3.14159 et 9.81. Les lignes de programme suivantes:

```
      WRITE (IMP,100) A,B
100   FORMAT (8H*** A= ,F5.2,3X,3HB= ,F5.2)
```

donneraient comme résultat à l'imprimante:

```
*** A= 3.14 B= 9.81
```

On veillera à ce que n représente la longueur exacte de la chaîne de caractères sous peine de modifier complètement les formats d'édition suivants s'ils existent. Une mauvaise valeur provoquera certainement un message d'erreur.

Le caractère / figurant dans un format provoquera un changement de ligne.

Ex :

```
      WRITE(IMP,10) A,B
10    FORMAT (I5 / F12.5)
```

On peut répéter certains caractères en procédant de la manière suivante:

`FORMAT (1X,19H******************/)` peut être remplacé par:

`FORMAT (1X,19(1H*)/)` où 19 est un multiplicateur.

Certains compilateurs autorisent l'usage du signe $ qui empèche le passage à la ligne suivante à la fin d'une instruction WRITE.

Le fait que parfois une instruction format contienne plus de spécifications que nécessaire ne gêne pas. Seules seront utilisées dans ce cas les premières. Si à l'inverse, le nombre de spécifications est trop court, le format est parcouru une seconde fois, voire plus encore. En ce qui concerne l'impression, uniquement pour certains compilateurs, le papier dans l'imprimante peut être contrôlé en utilisant les caractères suivants:

- + provoque une surimpression
- 1 fait aller à la page
- 0 fait sauter une ligne

Tout autre caractère fait aller à la ligne. Ainsi si l'on veut imprimer
`RESULTATS DU CALCUL`, en écrivant

```
      WRITE (IMP,10)
10    FORMAT (19HRESULTATS DU CALCUL)
```

fait aller à la ligne(le premier caractère est différent de +,1,0), et ce caractère est mangé!:

```
ESULTATS DU CALCUL
```

Il aurait fallu provoquer un saut de page et écrire le titre au début de la nouvelle page de la manière suivante:

```
10 FORMAT (20H1RESULTATS DU CALCUL)
```

6.2 Echanges avec les fichiers

6.2.1 Définition de fichier

Il sera intéressant d'envoyer certains résultats dans des fichiers, pour une utilisation ultérieure par un programme différent. Ces fichiers pourront être utilisés soit pour la lecture de données (entrée), soit pour le stockage de données (sortie). Lorsque les résultats d'un programme sont nombreux, il est pratique de pouvoir les consulter dans un fichier, à l'aide d'un éditeur de texte. Le fichier dans ce cas là, sera un fichier texte, à accès séquentiel.
On utilise souvent un fichier résultat pour passer des données à un autre programme de post-traitement. Ce cas peut alors être assimilé à une forme particulière de passage d'arguments.

Un fichier est un ensemble de données inscrites sur un support de masse. Le fichier est désigné par un nom suivi d'une extension facultative. L'extension donne en général des informations sur la provenance ou la destination des données. Un fichier contenant du texte

aura comme extension « .txt », ce qui en permettra l'ouverture à l'aide du bloc-notes par un double-clic sur le nom du fichier. Si le fichier contient des données binaires, on choisira une extension autre que « .txt », par exemple « .bin » ou « .dat ».

6.2.2 Contenu d'un fichier

Le contenu d'un fichier est décidé par le concepteur du programme. On distingue deux genres de fichiers :

1. Les fichiers Formatés. Ce sont des fichiers texte, qui peuvent être ouverts par un éditeur de texte et qui peuvent être affichés sans décodage particulier. Le FORTRAN considère par défaut un fichier comme Formaté, c'est-à-dire comme fichier texte.
2. Les fichiers Non Formatés. Ce sont en général des fichiers binaires. Seul un programme spécialement développé en permettra la lecture. L'intérêt de ce genre de fichier est la compacité. Les données y sont inscrites exactement comme elles figurent en mémoire : sous la forme binaire.

6.2.3 Modes d'accès à un fichier

Il existe deux modes d'accès :

1. Le mode Séquentiel. C'est le mode d'accès le plus courant. Dans ce mode, les données sont placées dans le fichier dans l'ordre d'apparition des instructions d'écriture. Le fichier ne peut être modifié que par l'ajout de données en fin de fichier. Si l'on veut insérer des données à un endroit quelconque du fichier, sans modifier les données existantes, il faut passer par un fichier intermédiaire. L'écriture dans un fichier séquentiel efface toutes les données qui sont situées après les données que l'on vient d'écrire.
2. Le mode Direct. Ce mode permet l'accès direct à n'importe quel enregistrement du fichier, que ce soit en lecture ou en écriture, sans affecter les autres enregistrements. La modification d'un enregistrement quelconque est possible sans faire appel à des manipulations de fichiers et de données. L'inconvénient majeur de ce mode est l'obligation d'avoir des enregistrements de longueur identique.

6.2.4 Mode opératoire

L'utilisation d'un fichier necessite trois étapes essentielles :

1. Ouverture du fichier (instruction OPEN)
2. Accès au fichier (instruction READ ou WRITE)
3. Fermeture du fichier (Instruction CLOSE)

Il est possible d'ouvrir simultanément plusieurs fichiers, pourvu que le numéro d'unité soit différent. Il est raisonnable de fermer le fichier lorsque l'on en a plus besoin.

6.3 Les instructions relatives aux manipulations de fichiers

6.3.1 Ouverture d'un fichier

```
OPEN ([UNIT=]unite
                    [,ACCESS=acces]
                    [,BLANK=blanks]
                    [,ERR=etiquette]
                    [,FILE=fichier]
```

```
                    [,FORM=format]
                    [,IOSTAT=erreur]
                    [,RECL=lg_rec]
                    [,STATUS=status])
```

Notations : les spécifications entre crochets sont facultatives. Une instruction standard d'ouverture de fichier a généralement la forme suivante :

```
OPEN(unite, FILE = 'nom du fichier', STATUS =
                                                    'OLD'
                                                    'NEW'
                                                    'UNKNOWN')
```

Chaque paramètre peut apparaître dans un ordre quelconque. Si l'on omet UNIT, la valeur unité doit apparaître en premier. C'est cette forme qui est en général adoptée.

6.3.1.1 Description de chaque paramètre

unité : nombre entier qui sera choisi entre 10 et 99. Il est possible de prendre des valeurs inférieures à 10, au risque d'avoir des ennuis à l'éxécution, et éventuellement une incompatibilité avec d'autres compilateurs. Ce sera le numéro d'accès au fichier. Ce numéro est arbitraire, mais il sera réservé à ce fichier durant toute la durée de son ouverture.

acces : 3 types d'accès seront possibles. 'SEQUENTIAL', 'DIRECT' ou 'APPEND' (pas implémenté sur tous les compilateurs). Le mode d'accès par défaut est 'SEQUENTIAL'.

blanks : Option qui permet d'évaluer les espaces comme étant 'NULL' (par défaut) ou 'ZERO'

etiquette : Etiquette d'une ligne instruction qui sera appelée en cas de problème

fichier : Expression alphanumérique signifiant le nom du fichier.

format : Expression qui sera soit 'FORMATTED', soit 'UNFORMATTED'. Si l'accès est séquentiel, le format par défaut sera 'FORMATTED'.

erreur : Variable entière, nulle si pas d'erreur, négative si fin de fichier rencontré, valeur positive non nulle figurant l'erreur si problème.

lg-rec : Expression entière précisant le nombre de bytes d'un enregistrement pour un fichier à accès direct. Ignoré pour un séquentiel.

status : Peut être 'OLD', 'NEW', 'SCRATCH', 'UNKNOWN', 'APPEND'.

OLD : Le fichier doit déjà exister
NEW : Le fichier est à créer .

SCRATCH : Le fichier est détruit à la fin du programme.
APPEND : Le fichier existe déjà, et on se place à la fin pour rajouter des données.
UNKNOWN : NEW ou OLD si le fichier existe ou non.

Exemples :

```
c   Ouverture d'un fichier existant deja
      OPEN (11, FILE='DONNEES.DAT')

      ...

c Ouverture d'un nouveau fichier pour y placer des
c resultats
      OPEN (12, FILE='RESULT.DAT',ACCESS = 'SEQUENTIAL',
     +     STATUS='NEW')
```

6.3.2 Fermeture d'un fichier

La fermeture d'un fichier se fait automatiquement lorsque le programme se termine normalement. Ne pas utiliser le signe * dans l'instruction de fermeture de fichier.

Syntaxe :

```
CLOSE([UNIT=]unite
                  [,ERR=etiquette]
                  [,IOSTAT=erreur]
                  [,STATUS=status])
```

Si l'on omet de mentionner UNIT=, le numéro de unité doit figurer en premier. L'ordre des arguments n'a pas d'importance.

Description de chaque paramètre :

unité : Nombre entier spécifiant l'unité externe. Aucune erreur ne sera signalée si cette unité n'a pas été ouverte.

etiquette : étiquette d'une instruction exécutable dans le même programme. Cette instruction sera exécutée si une erreur d'accès au fichier se produit.

erreur : Variable entière ou élément de tableau qui se verra affecter le numéro d'erreur si celle-ci se produit.

status : Peut être 'KEEP' (conserver) ou 'DELETE' (effacer). Les fichiers ouverts sans nom auront un status par défaut 'DELETE' qui signifie effacer après fermeture. Ces fichiers sont considérés comme fichiers temporaires. Le status par défaut pour les autres fichiers sera 'KEEP'.

Exemple :

```
c     fermeture et effacement du fichier 8
      CLOSE(8, STATUS='DELETE)
```

6.3.3 Enregistrement "fin de fichier"

Cette instruction est peu utilisée.

```
ENDFILE unite
ou
ENDFILE ([UNIT=]unite
                     [,ERR=etiquette]
                     [,IOSTAT=erreur])
```

Les arguments sont identiques à ceux déjà vus pour OPEN et CLOSE. Après avoir écrit l'enregistrement fin de fichier, L'instruction ENDFILE empêche tout transfert supplémentaire de données si l'on n'a pas auparavant utilisé l'instruction REWIND ou BACKSPACE. Lorsque l'on opère sur un fichier à accès direct, les enregistrements situés après l'enregistrement fin de fichier seront effacés.

Exemple d'utilisation sur un fichier ouvert sur le numéro 7:

```
        WRITE (7,*) resultat
        ENDFILE 7
        REWIND 7
        READ (7,*) donnee
```

6.3.4 Positionnement d'un enregistrement en arrière sur un fichier

```
BACKSPACE unite
ou

BACKSPACE ([UNIT=]unite
                       [,ERR=etiquette]
                       [,IOSTAT=erreur])
```

Les arguments ne demandent pas d'explication particulière.

Exemple d'utilisation pour le fichier 7 :

```
     BACKSPACE 7
  ou
     BACKSPACE (7)
  ou
     UNITE = 7
     BACKSPACE UNITE
  ou
     BACKSPACE (UNIT=unite, ERR=100, IOSTAT=code_erreur)
```

6.3.5 Repositionnement d'un fichier sur le premier enregistrement

```
REWIND unité ou
REWIND ([UNIT=]unité
                    [,ERR=etiquette]
                    [,IOSTAT=erreur])
```

Cette commande rappelle les inscriptions que l'on peut voir sur un magnétophone non francisé. Elle provient de l'époque où les mémoires de masse de type disque n'existaient pas encore. Les bandes magnétiques servaient alors de moyen de stockage de l'information. Ce moyen est encore utilisé pour stocker des données lorsque elles sont trop nombreuses pour pouvoir figurer sur un disque dur. La lecture était plus lente.
Cette instruction est toujours valable. Si l'on veut se repositionner au début d'un fichier séquentiel ouvert, l'instruction REWIND convient parfaitement. Attention, si l'on procède à une écriture apres un REWIND, toutes les données situées après celles que l'on vient d'écrire seront perdues.

6.3.6 Lecture des particularités d'une unité ou d'un fichier

```
INQUIRE ([UNIT=]unité ou FILE='nom_de_fichier'
                                                [,ACCESS=acces]
                                                [,BLANK=blanks]
                                                [,DIRECT=direct]
                                                [,ERR=étiquette]
                                                [,EXIST=exist]
                                                [,FORM=forme]

[,FORMATTED=formatte]
                                                [,IOSTAT=erreur]
                                                [,NAME=nom]
                                                [,NAMED=nomme]
                                                [,NEXTREC=nextr]
                                                [,NUMBER=numero]
                                                [,OPENED=ouvert]
                                                [,RECL=lg]

[,SEQUENTIAL=seq]

[,UNFORMATTED=unf])
```

Cette instruction est utilisée dans les sous-programmes devant acéder à un fichier déjà ouvert par ailleurs.

6.3.6.1 Description des arguments

acces : Variable alphanumérique qui contiendra au retour une des chaines de caractères suivantes :
'SEQUENTIAL'

'DIRECT'

blanks : Variable alphanumérique qui se verra affecter au retour soit 'NULL', soit 'ZERO'.

direct : Variable alphanumérique, qui contiendra selon le type d'accès 'YES' pour direct ou 'NO' ou 'UNKNOWN'.

exist : Variable logique qui contiendra .TRUE. si le fichier existe, sinon .FALSE..

forme : Variable alpha. Si l'unité ou le fichier est connecté à une entrée-sortie formatée, forme contiendra 'FORMATTED', sinon 'UNFORMATTED'

formatte : Variable alpha, réponse à la question FORMATTED?. Contiendra 'YES', 'NO', 'UNKNOWN'.

nom : Variable alphanumérique qui contiendra le nom du fichier connecté à unité.

nomme : Variable logique. .FALSE. si le fichier est un fichier temporaire (sans nom ou scratch).

nextr : variable entière, numéro du prochain enregistrement pour un fichier à accès direct.

numero : Variable entière qui contiendra le numéro de l'unité connecté au fichier dont le nom devra être spécifié.

ouvert : Variable logique qui sera .TRUE. si un fichier est effectivement connecté à l'unité précisée.

lg : Variable entière qui contiendra le nombre d'octets d'un enregistrement pour un fichier à accès direct.

seq : Variable alphanumérique. Contient 'YES' si le fichier précisé est à accès séquentiel.

unf : Variable alphanumérique. Contient 'YES' si le fichier est de forme non formatée.

Exemple d'utilisation :

Vérification de l'existence d'un fichier.

```
      CHARACTER*12 nom
      LOGICAL existe
c  verification de l'existence :
      INQUIRE(FILE=nom, EXIST=existe)
c  entree du nom de fichier
100   WRITE(*,*)'Donnez le nom du fichier :'
      READ(*,*) nom

      IF(.NOT.existe) THEN
          WRITE(*,*)'Le fichier n existe pas.'
```

```
        GOTO 100
    END IF
```

6.3.7 Lecture et écriture des données dans un fichier

La lecture et l'écriture de données dans un fichier se fait avec les mêmes instructions que pour la console :

`READ(`*`unité`*`,` *`format`*`)` *`liste de variables`*

`WRITE(`*`unité`*`,` *`format`*`)` *`liste de variables`*

L'unité utilisée sera le numéro affecté au fichier lors de l'ouverture de celui-ci.

6.4 Impression de résultats

La plupart des systèmes d'exploitation considèrent les imprimantes comme un fichier possédant un nom générique, comme PRN pour MS-DOS, ou WINDOWS. Aussi, si l'on désire imprimer des résultats directement sur une imprimante, il faut l'ouvrir comme un fichier et y écrire les données comme d'habitude :

```
...
OPEN(9,FILE='PRN')
WRITE(9,*)'Ce texte est destine a l imprimante'
...
```

6.5 Structures des fichiers et des enregistrements

L'accès à des données rassemblées dans un fichier dépend de la forme de stockage et du mode d'accès. On accède à des données d'un fichier par groupe de données. Ce groupe est appelé "enregistrement". Les méthodes de stockage des données dans le fichier peuvent être :

- Formatées
- Non formatées
- Binaires

On peut accéder à un enregistrement selon deux modes :

- Séquentiel
- Direct

L'accès séquentiel à un enregistrement N implique la lecture des N-1 enregistrements précédents, alors que l'accès direct permet d'accéder en lecture comme en écriture à n'importe quel enregistrement. Attention dans ce cas, la longueur d'un enregistrement doit être connue et constante.

On peut combiner les méthodes de stockage et les modes d'accès, ce qui donne 6 façons de procéder.

6.5.1 Les enregistrements formatés

On crée des fichier formatés en spécifiant à l'ouverture FORM='FORMATTED'. Les enregistrements sont alors stockés sous forme ASCII, c'est à dire directement affichables. Les nombres, utilisés en mémoire sous forme binaire, sont convertis en forme ASCII. Chaque enregistrement se termine par les caractères ASCII Retour chariot et Saut de ligne (CR et LF).

6.5.1.1 Les fichiers séquentiels formatés

Les fichiers séquentiels formatés comportent des enregistrements qui doivent être lus de façon séquentielle, chaque enregistrement est de longueur quelconque. Attention, il faut s'assurer, lors de la lecture de bien pouvoir lire un enregistrement complet. Chaque enregistrement est séparé par une suite CR LF code ASCII 0DH, 0AH.

6.5.1.2 Fichiers formatés à accès direct

Tous les enregistrements possèdent la même longueur. Chaque enregistrement peut directement être lu ou écrit, sans passer par les précédents enregistrements. La longueur d'un enregistrement est spécifiée au moment de l'ouverture par `RECL=valeur`. En réalité, il faut ajouter 2 caractères CR et LF, qui servent de séparateur. Si l'on écrit un enregistrement de longueur inférieure à celle spécifiée, il est automatiquement comblé par des espaces typographiques (code ASCII 32).

L'écriture du 7ème enregistrement suivi du 5ème se fera par exemple ainsi :

```
OPEN(13,FILE='FORDIR',FORM='FORMATTED',ACCESS='DIRECT', RECL=15)
...
WRITE(13,'(A)',REC=7) 'Coucou'
WRITE(13,'(I6)',REC=5)  12756
```

Nous aurons sur le disque :

```
    ... enregistrement 4 (17 octets) 12756          [CR/LF]enregistrement
6(17octets)Coucou         [CR/LF]...
```

La lecture se fait directement, de façon identique. Il est inutile de lire tous les enregistrement précédant l'enregistrement souhaité.

6.5.2 Les enregistrements non formatés.

Les enregistrements non formatés contiennent les données brutes, sous une forme identique que celle utilisée en mémoire. Par exemple, un réel simple précision occupe 4 octets en mémoire. Ecrit dans un fichier sous la forme non formatée, il occupera également 4 octets, mais il est inutile de chercher à le relire au moyen d'un éditeur de texte. Dans un fichier à enregistrements formatés, un réel occupe autant d'octets que de caractères nécessaires pour le représenter, étant donné qu'il sera exprimé sous la forme ASCII. Donc, premier avantage : un gain de place certain. La transcription binaire/ASCII aussi bien au niveau de la lecture que de l'écriture demande également du temps. Ce temps est économisé dans le cas des enregistrements non formatés, puisque la forme de stockage est identique à celle du stockage en mémoire. Les échanges de données sont alors beaucoup plus rapides.

6.5.2.1 Les enregistrements séquentiels non formatés

Les fichiers de ce type sont particuliers au FORTRAN. Les fichiers sont organisés en blocs de 130 octets ou moins. Ces blocs sont appelés "blocs physiques". Chaque bloc comporte les données ou une partie des données que l'on a enregistrées, encadrées deux octets contenant la longueur utile du bloc. Bien entendu, le compilateur se charge de mettre en place et à jour ces 2 octets. La figure suivante indique la structure d'un bloc physique.

/L/.....bloc...../L/

Lorsque l'on désire enregistrer un ensemble de données dépassant la capacité maximale d'un bloc physique, le compilateur place les octets d'encadrement L à 129 pour indiquer que le bloc physique courant n'est pas terminé.

Structure d'un enregistrement logique de plus de 128 octets :

/129/ ... 128 octets de données.../129//12/ ...12 octets de données.../12/

Cette opération de partitionnement en blocs de 128 octets est transparente pour le programmeur. En effet, il peut créer des enregistrements de longueur quelconque.

Les fichiers séquentiels non formatés commencent par un octet dont la valeur est 75 et se terminent par un octet dont la valeur est 130

Exemple d'utilisation de fichier séquentiel non formaté :

```
       CHARACTER XYZ(3)
       INTEGER*4 IDATA(35)
       DATA IDATA /35 * -1/, XYZ / 'x','y','z' /
c
c -1 est encode par FFFFFFFF en hexadecimal
c
       OPEN(33, FILE='SEQNF',FORM='UNFORMATTED')

c  Le mode sequentiel est obtenu par defaut.
c
c   On ecrit un enregistrement de 140 octets, suivi par un
c   enregistrement de 3 octets
       WRITE(33) IDATA
       WRITE(33) XYZ
       CLOSE(33)
       END
```

Le fichier aura alors l'allure suivante :

/75/129/ FF....128 octets à FF/129/12/ FF 12 octets à FF/12/3/xyz/3/130/

6.5.2.2 Les fichiers à accès direct et à enregistrements non formattés

Comme tous les fichiers à accès direct, l'utilisateur peut accéder en lecture ou en écriture à chaque enregistrement, dans n'importe quel ordre. Ceci impose que la longueur d'un enregistrement est connue et constante, c'est évident. La longueur d'un enregistrement est spécifiée dans l'instruction OPEN par RECL= valeur.

Exemple de création de fichier à accès direct non formaté :

```
       OPEN(33,FILE='DIRNF',RECL=10,FORM='UNFORMATTED',
```

```
     + ACCESS='DIRECT')
       WRITE(33,REC=3).TRUE.,'abcdef'
       WRITE(33, REC=1)2049
       CLOSE(33)
       END
```

L'allure du fichier sera la suivante :

/01 08 00 00??????/??????????/01 00 00 00 abcdef/

6.5.3 Les enregistrements binaires

Cette forme est la plus recommandée, car elle n'est pas propre au FORTRAN. On peut donc relire et créer des fichiers en provenance ou à destination d'autres langages. Les fichiers binaires n'ont pas de structure interne particulière. Le programmeur doit donc être parfaitement au courant de la disposition des données dans le fichier. Aucun séparateur ne permet de délimiter un quelconque bloc ou groupe de variables, sauf si le programmeur décide d'en placer de son propre gré. Les données étant enregistrées sous forme binaire, sans séparateur, le gain de place est encore (légèrement) plus important. On préférera ce type de fichiers pour le stockage d'un nombre important de données. Attention, un échange de données avec d'autres logiciels est plus problématique, mais, grâce aux normes d'encodage des valeurs réelles par exemple, ces fichiers sont souvent lisibles par d'autres langages. Cependant, pour un transfert de données vers un tableur par exemple, il est préférable de passer par un fichier séquentiel formaté. Pour travailler en mode binaire, il suffit de préciser au moment de l'ouverture FORM='BINARY'.

6.5.3.1 Les fichiers séquentiels binaires

L'écriture et la lecture de fichiers binaires sont régentés par le type des variables. Inutile de préciser une longueur d'enregistrement. Pour accéder à des données enregistrées, on relit simplement celles qui ont été placées avant, en respectant leur type.

Exemple de création de fichier binaire à accès séquentiel :

```
$STORAGE:4
       INTEGER*1 BELLS(4)
       INTEGER IWYS(3)
       CHARACTER*4 CVAR
       DATA BELLS /4*7/
       DATA CVAR /' is '/,IWYS/'What',' you',' see'/
c On va ecrire  'What you see is what you get!' suivi de 3 beeps
       OPEN(33,FILE='SEQBIN',FORM='BINARY')
       WRITE(33) IWYS,CVAR
       WRITE(33) 'what ','you get!'
       WRITE(33) BELLS
       CLOSE(33)
       END
```

6.5.3.2 Les fichiers binaires à accès direct

Il faut spécifier au moment de l'ouverture le type d'accès, ainsi que la longueur d'un enregistrement. Le nombre d'octets lus ou écrits dépendent toujours du type de la variable.

Contrairement au cas des fichiers non formatés, la tentative d'écriture d'un enregistrement de longueur supérieure à celle déclarée par RECL=longueur, ne provoque pas d'erreur, mais déborde sur l'enregistrement suivant. Si l'on ne mentionne pas de numéro d'enregistrement (REC=n dans l'instruction d'écriture ou de lecture) les accès se font alors de manière séquentielle à partir de l'enregistrement courant.

Exemple de création de fichier binaire à accès direct

```
$STORAGE:4
       OPEN(33,FILE='DIRBIN', RECL=10, ACCESS='DIRECT',
      +FORM='BINARY')
       WRITE(33, REC=1)'abcdefghijklmno'
       WRITE(33)4, 5
       WRITE(33, REC=40) 'pq'
       CLOSE(33)
       END
```

Le fichier ainsi obtenu prend la forme :

/abcdefghij/klmno?????/04 00 00 00 05 00 00 00??/.../pq????????/

Quelques exemples

Lorsqu'un débutant veut s'essayer aux manipulations de fichiers, il est conseillé de faire les essais avec peu de données, et en mode séquentiel formaté. Dans un premier temps, Il se contentera d'afficher à l'écran les données au moyen de l'instruction WRITE(*,*). Lorsque ce qui est affiché à l'écran convient, il suffit de placer l'instruction suivante avant la première instruction WRITE :

```
OPEN(10,FILE='premier.txt',STATUS='UNKNOWN')
```

Le fichier s'appellera "premier.txt". S'il n'existait pas auparavant, il sera créé. L'écriture dans le fichier est simple : il suffit de dupliquer toutes les lignes comportant une instruction WRITE et de remplacer la première étoile par la valeur 10.

On placera après le dernier WRITE l'instruction suivante :

```
CLOSE(10)
```

7. Traitement de l'information

7.1 Expressions arithmétiques

7.1.1 Définition

Une expression arithmétique est un ensemble formé de constantes, variables et fonctions séparées par des opérateurs et des parenthèses ; elle est destinée à fournir une valeur numérique.

7.1.2 Opérateurs

+ Addition
- Soustraction

* Multiplication
/ Division

** Exponentiation

Ces opérateurs sont présentés par ordre de priorité croissante.

Exemple:

TOTAL=TOTHT+TVA

Les opérations arithmétiques simples sont évaluées de la gauche vers la droite. Par contre, deux exponentiations successives sont évaluées de la droite vers la gauche.

7.1.3 Type des résultats obtenus par +, -, *, /

Les types sont classés par ordre croissant du rang:

1. INTEGER*2 rang le plus faible
2. INTEGER*4
3. REAL*4 (REAL)
4. REAL*8 (DOUBLE PRECISION)
5. COMPLEX*8 (COMPLEX)
6. COMPLEX*16 rang le plus élevé

C'est l'opérande de rang le plus élevé qui fixe le type du résultat.

Une petite exception: Une opération entre un COMPLEX*8 et un REAL*8 génère un COMPLEX*16 (Il s'agit de la double précision). Rappelons qu'un complexe est composé d'une partie réelle et d'une partie imaginaire et que ces deux parties sont toujours du même type.

7.1.4 Type des résultats obtenus par l'opérateur puissance

On peut résumer sous la forme d'un tableau le type du résultat de l'élévation de X à la puissance Y :

		Premier Opérande (X)			
		Entier	Réel	Double	Complexe
Exposant Y	Entier	*Entier*	*Réel*	*Double*	
	Réel	*Réel*	*Réel*	*Double*	
	Double	*Double*	*Double*	*Double*	
	Complexe	*Complexe*			

7.1.5 Règle d'homogénéité

Au cours d'un calcul, si les opérandes sont de types différents, l'opérande du type le plus faible (occupant le moins de place mémoire) sera converti dans le type de l'opérande le plus fort.

7.2 Traitement des chaînes de caractères

7.2.1 Accès à une partie d'une chaîne

On peut, si on le désire, accéder à une partie d'une chaîne de caractères en jouant sur les indices suivant le nom de la variable:

`nom(d:f)` "nom" représente le nom de la variable

d : indique le rang du premier caractère concerné (le premier de la chaîne par défaut).

f : indique le rang du dernier caractère concerné.

Exemple:

```
CHARACTER*10 MOT

MOT='CONTORSION'
```

`MOT(5:6)` vaut `'OR'`

`MOT(4:)` vaut `'TORSION'`

`MOT(:3)` vaut `'CON'` (excusez...)

On peut modifier MOT:

```
MOT(4:5)='VE'
```

Après exécution de cette instruction, MOT contiendra 'CONVERSION'.

L'expression `MOT(:4)=MOT(3:6)` est interdite, car de part et d'autre du symbole d'affectation figurent en commun `MOT(3:4)`. Dans cet exemple, il aurait fallu écrire:

```
CHARACTER*4 TMOT
...
TMOT=MOT(3:6)
MOT(:4)=TMOT
```

Pour mettre une variable alphanumérique à blanc, il suffit de lui attribuer la chaîne de caractères nulle :

```
MOT=''
```

Si une suite de caractères à attribuer est trop longue pour la variable de destination, seuls les premiers sont attribués, dans la limite des places disponibles.

7.2.2 Opérateur de concaténation de chaîne

Il est constitué de deux barres de fraction successives (//).

7.3 Fonctions numériques standard

On peut citer les plus usuelles:

SIN	sinus
COS	cosinus
ALOG	logarithme népérien
EXP	exponentielle
SQRT	racine carrée
ATAN	arc tangente
TANH	tangente hyperbolique
ABS	valeur absolue d'un réel
IABS	valeur absolue d'un entier
FLOAT	convertit d'entier à réel
IFIX	convertit de réel à entier
SIGN	transfert de signe
CHAR	conversion du code ASCII en caractère
ICHAR	conversion inverse de la précédente
NINT	fonction arrondi
MOD	fonction modulo

Les fonctions incorporées au langage font appel à des mots-clé.
Exemples :

Racine carrée :

Mot-clé: `SQRT`

```
X1 = -B + SQRT( B**2 - 4*A*C)
```

Code ASCII d'un caractère :

`ICHAR('0')` donne `48`.

Réciproquement, `CHAR(I)`, I étant un entier, renvoie le caractère dont le code ASCII est I (`CHAR(48)` donnera le caractère 0).

Exemple d'utilisation de la fonction CHAR:

Emission d'un Bip sonore.

On déclare IBELL comme paramètre, et, plus loin dans le programme, on l'envoie à la console:

```
      PARAMETER (IBELL=7)

 ....

      WRITE (*,*) CHAR(IBELL)
```

7.4 Expressions logiques

7.4.1 Opérateurs de relation arithmétique

Une opération de comparaison port sur deux opérandes et donne un résultat logique. Naturellement, les deux opérandes doivent être du même type. On ne peut comparer une variable alphanumérique à une variable numérique.

Les opérateurs de comparaison sont:

FORTRAN 77		FORTRAN 90
.GT.	Plus grand que	>
.GE.	Plus grand ou égal que	>=
.EQ.	Egal à	= =
.NE.	Différent de	!=
.LE.	Inférieur ou égal à	<=
.LT.	Inférieur à	<

Remarque:
Deux complexes ne peuvent être que égaux ou différents, donc seuls les opérateurs .NE. et .EQ. sont utilisables.

On peut également effectuer des comparaisons sur deux chaînes de caractères. Dans ce cas, la comparaison se fait de la gauche vers la droite, caractère après caractère. La différence est faite dès l'apparition d'un premier caractère différent de son homologue en position. C'est

alors celui qui possède le code ASCII le plus élevé qui sera considéré comme supérieur. Signalons que le code ASCII respecte l'ordre alphabétique.

Exemple:

`'HENRY'` est supérieur à `'HENRI'`

7.4.2 Opérateurs logiques

En logique, il n'existe que deux états:

.TRUE.	VRAI
.FALSE.	FAUX

Les opérateurs logiques sont:

.NOT.	NON
.AND.	ET
.OR.	OU
.EQV.	EQUIVALENCE
.NEQV.	OU EXCLUSIF

Le lecteur pourra reconstituer la table de vérité.

Ordre d'évaluation des diverses opérations vues:

1. Opérateurs arithmétiques
2. Concaténation
3. Opérateurs de comparaison
4. Opérateurs logiques

Exemple:

`I.EQ.0.OR.J.GT.I+3` sera évalué comme:

```
(I.EQ.0).OR.(J.GT.(I+3))
```

7.5 Opérateurs de relations alphanumériques

On peut également comparer deux chaînes de caractères entre elles. Les opérateurs seront :

```
.GT., .GE., .EQ., .NE., .LE., .LT.
```

Une chaîne de caractères est évaluée à partir de la valeur en code ASCII des caractères. Dans ce cas, la comparaison se fait de la gauche vers la droite, caractère après caractère. la différence est faite dès l'apparition d'un premier caractère différent de son homologue en

position. C'est alors celui qui possède le code ASCII le plus élevé qui sera considéré comme supérieur. Signalons que le code ASCII respecte l'ordre alphabétique.

Exemple:

`'HENRY'` est supérieur à `'HENRI'`

8. Gestion mémoire

Déclarations COMMON, EQUIVALENCE, ALLOCATE

8.1 Variables globales : La déclaration COMMON

La déclaration COMMON était fréquemment utilisée pour effectuer un passage des arguments, avec d'autres effets sur les variables qu'il n'est pas forcément judicieux d'exploiter comme par exemple le groupement des variables en mémoire dans l'ordre défini dans la liste COMMON.

Syntaxe de la déclaration COMMON :

```
COMMON /nom1 bloc/liste1 variables/nom2 bloc/liste2...
```

La déclaration COMMON a donc deux effets principaux :

- Elle groupe les variables dans l'ordre de chaque liste,
- Elle permet de les rendre visibles (globales) d'un sous-programme pourvu qu'il possède une déclaration COMMON correspondante. Une déclaration COMMON peut donc servir au passage des arguments.

Le nom de bloc n'est pas obligatoire. Dans ce cas, il s'agit d'un COMMON blanc. Son utilisation est à éviter.

Exemple d'utilisation de COMMON :

```
        COMMON /BLK1/A,B,C /BLK2/T,U,I //Z
```

// définit un COMMON blanc.

Il est impossible de faire figurer dans un COMMON des éléments disjoints d'un tableau :

```
        COMMON A(1), A(3)
```

n'a pas de sens.

On peut grouper les instructions COMMON et DIMENSION :

```
        COMMON A(100)
```

équivaut à

```
        DIMENSION A(100)
        COMMON A
```

Lorsque le COMMON est utilisé pour un passage des arguments, on le retrouve dans le programme et le sous-programme :

```
C Programme principal
...
       COMMON /XXX/A,I,T
...
       CALL SPROG(...)
...
       END

C Sous-programme
       SUBROUTINE SPROG(...)
       COMMON /XXX/Z,L,U
...
       RETURN
       END
```

On ne peut pas employer dans un COMMON une variable figurant déjà dans la liste des arguments :

```
       SUBROUTINE SPROG(A,B,C)
       COMMON X,Y,C
```

est incorrect !

8.2 Le partage mémoire : La déclaration EQUIVALENCE

La déclaration EQUIVALENCE permet d'affecter à plusieurs variables la même adresse. Sa syntaxe se présente sous la forme :

```
       EQUIVALENCE (liste de variables),(liste de variables),...
```

Exemple :

```
       EQUIVALENCE (A,B,C),(Y,TAB(3))
```

A, B, C auront la même adresse mémoire, c'est à dire que ces variables ne formeront qu'une. Y aura la même implantation que le troisième élément de TAB.

Si l'on place dans une liste d'équivalence des éléments de tableaux distincts :

```
       EQUIVALENCE (A(5), B(7))
```

non seulement A(5) serait égal à B(7), mais aussi A(4) à B(6), etc.

En conséquence, la déclaration suivante

```
       EQUIVALENCE (A(5),B(7)),(A(1),B(1))
```

est illicite.

En couplant les déclarations EQUIVALENCE et COMMON, on peut étendre les équivalences:

```
DIMENSION A(5)
COMMON X,Y,Z
EQUIVALENCE (Y,A(2))
```

Nous aurons dans ce cas en plus l'équivalence entre X et A(1), Z et A(3).

En suivant cette logique, les déclarations suivantes sont illicites :

```
COMMON A,B
EQUIVALENCE (A,B)
```

L'usage de l'instruction EQUIVALENCE permet d'économiser de la place mémoire. Elle permet également de "bidouiller" les variables, car on peut déclarer équivalentes des variables de types différents. Cette dernière façon de faire est cependant à éviter si l'on ne sait pas trop ce que l'on fait...

8.3 Variables situées dans la pile : la déclaration AUTOMATIC

Le FORTRAN fait appel uniquement à des variables statiques, c'est à dire que ces variables occupent de la place mémoire durant toute la durée du programme. D'autres langages comme le C procèdent de manière différente : une variable n'occupe de la mémoire que lorsque le le module dans lequel elle est déclarée est en cours d'exécution. La variable est placée dans ce que l'on appelle "la pile" ou "le tas". L'avantage est immédiat : gain de place mémoire, surtout pour les tableaux. nous sommes en présence de variable à existence "dynamique".

On peut suggérer au FORTRAN 90 l'utilisation de ce type de variables à l'aide de la déclaration AUTOMATIC.
L'usage est le suivant :

```
AUTOMATIC liste de variables
```

Si la déclaration `AUTOMATIC` apparaît seule dans une ligne, toutes les variables qui suivent sont placées dans la pile. La déclaration `SAVE` permet d'introduire des exceptions pour certaines variables. Le paragraphe 3.5 indique comment indiquer au compilateur de gérer un tableau comme variable automatique sans déclaration spécifique.

8.4 Les Structures de variables

Dans le cas de descriptions d'objets complexes, par exemple d'individus ou de véhicules, il est souvent nécessaire d'utiliser plusieurs types de variables. Par exemple, si l'on veut établir des fiches techniques pour des voitures particulières, on peut tenir compte des caractéristiques suivantes :

Paramètre	Type
Marque	Chaîne de caractères (Maximum 25)
Modèle	Chaîne de caractères (Maximum 15)
Date d'achat	Tableau de 3 entiers (x 2 octets)
Puissance Fiscale	Nombre entier (2 octets)

Masse à vide	Nombre entier (2 octets)
Charge utile	Nombre entier (2 octets)
Coef. de prix d'usage	Réel (4 octets)

Si, par exemple, on veut gérer un parc de voitures, il serait intéressant de grouper ces données sur une seule variable, tout en gardant la possibilité d'accéder individuellement à chaque composante. De plus, si cette variable « complexe » pouvait être manipulée comme un élément de tableau, dans le cadre de la gestion d'un vaste parc de véhicules, la manipulation des données serait grandement facilitée.

8.4.1 Déclarations STRUCTURE et RECORD

La déclaration STRUCTURE permet de réaliser ce groupement.
Utilisation de la déclaration :

```
STRUCTURE /nom_structure/
     déclarations des éléments
      . . .
END STRUCTURE
```

nom_structure : il s'agit d'un nom arbitraire que l'on donne à la structure dont la description suit. Attention au choix du nom. Ne pas utiliser de nom de fonctions intrinsèques, de variables déjà utilisées, de types de variables, etc...

Déclarations des éléments : On déclare les variables qui sont en général des variables classiques utilisées, comme INTEGER*2, REAL*4,... On peut aussi utiliser une structure déjà déclarée, car le nom de la structure est considéré comme étant un descripteur de type de variable.

Exemple appliqué au tableau ci-dessus :

```
STRUCTURE  /VEHICULE/
    CHARACTER*25 marque
    CHARACTER*15 modèle
    INTEGER*2 date_achat(3)
    INTEGER*2 puissance
    INTEGER*2 masse
    INTEGER*2 charge
    REAL*4 coef
END STRUCTURE
```

Il n'est pas possible d'inclure dans les variables de structure des tableaux dynamiques, ce qui est compréhensible, leur taille étant alors variable.

Une déclaration similaire à STRUCTURE est également utilisée en FORTRAN 90. Ils s'agit de la déclaration TYPE que nous verrons plus loin.

Nous venons donc, ci-dessus, de définir un nouveau **type** de variable et non de la déclarer. A partir de cet instant, nous pouvons utiliser le nom de ce type pour déclarer des variables « complexes » (à ne pas confondre avec COMPLEX !) et travailler avec ce type de variable. Commençons par la déclaration :

```
RECORD /VEHICULE/ parc1(50), parc2(50), voiture
```

parc1 et parc2 sont deux structures déclarées de 50 véhicules chacune (tableau de structures) et voiture est une variable simple.

La déclaration RECORD se place avec les autres déclarations de variables simples.

Evidemment, le bloc de description STRUCTURE doit figurer avant la déclaration RECORD.

Comment utiliser une variable de type structure ? Plusieurs possibilités s'offrent à nous : on peut manipuler une entité entière lors des affectations :

```
...
parc1(I)=voiture
```

tous les élements de la variable voiture sont copiés dans la Ieme structure de parc1. Si l'on veut accéder à un élément de voiture, il faut décrire la structure en insérant des points :

```
write(*,*) 'indiquez la marque du vehicule :'
read(*,*) voiture.marque
parc1(10).marque=voiture.marque
```

Si par exemple, on utilise une structure à l'interieur d'une autre :

```
STRUCTURE /moteur/
    integer*2 cylindree
    integer*2 Pdin
    integer*2 nb_cyl
END STRUCTURE
```

si l'on inclut cette déclaration dans la structure véhicule décrite précédemment :

```
STRUCTURE  /VEHICULE/
    ...
    REAL*4 coef
    RECORD /moteur/ motor
END STRUCTURE
```

on peut accéder à la cylindrée par :

```
write(*,*) voiture.motor.cylindree
```

Comme on peut le voir, l'usage est facile à assimiler.

Il est fort probable que l'on ait à sauver des elements de structure sur le disque. Dans ce cas, on est obligé, lorsque l'on travaille sur une structure entière, d'accéder à un fichier binaire ou non formaté. Il est possible de lire un élément d'une structure dans un fichier formaté.

Rappelons qu'un fichier binaire ou non formaté est une représentation exacte de la forme de la variable en mémoire, alors qu'un fichier formaté est la représentation ASCII lisible des variables.

8.4.2 La déclaration TYPE

Il s'agit d'une déclaration similaire, mais plus puissante. Nous ne verrons pas en détail les caractéristiques de cette déclaration, mais décrirons les points essentiels.

Syntaxe :

TYPE nom_type

declarations internes
END TYPE nom_type

nom_type est le nom du nouveau type de variable que nous venons de créer. Utilisons cette déclaration dans l'exemple vu dans le paragraphe précédent :

```
TYPE  VEHICULE
    CHARACTER*25 marque
    CHARACTER*15 modèle
    INTEGER*2 date_achat(3)
    INTEGER*2 puissance
    INTEGER*2 masse
    INTEGER*2 charge
    REAL*4 coef
END TYPE VEHICULE
```

On peut également utiliser l'instruction RECORD, liée à la déclaration STRUCTURE à l'interieur de cette construction. Pour créer une variable structurée, il faut utiliser la syntaxe suivante :

TYPE (nom_type)[,attributs] : : variable1, variable2,...

l'attribut peut avoir plusieurs fonctions. S'il s'agit de PARAMETER, la variable créée est alors une constante. Les attributs peuvent aussi être PUBLIC ou PRIVATE, selon la visibilité désirée pour les modules. Mais ceci est une autre histoire...

exemple :

TYPE (VEHICULE) : : parc1(50),parc2(50),voiture

```
write(*,*) 'indiquez la marque du vehicule :'
read(*,*) voiture%marque
voiture=VEHICULE('Peugeot','Clio',30,2,1990,5,1200,1900,0.75
)
parc1(10).marque=voiture%marque
write(*,*)voiture
```

ANNEXE A

CLASSEMENT DES INSTRUCTIONS PAR CATEGORIE

DECLARATIONS

AUTOMATIC	Les variables sont placées de préférence dans la pile
BLOCK DATA	Identifie un sous-programme bloc de données dans lequel on peut initialiser variables et tableaux
BYTE	déclaration, équivalent à INTEGER*1, variable sur 1 octet
CHARACTER	Déclaration pour variables alphanumériques
COMMON	Variables globales, partage par plusieurs modules
COMPLEX	Déclaration pour variables complexes
DATA	Initialisation de variables
DIMENSION	Déclaration pour tableaux
DOUBLE ...	Déclaration double précision
EQUIVALENCE	Partage location mémoire identique pour plusieurs variables de nom et de type différents
EXTERNAL	Identifie un nom comme étant un sous-programme ou une fonction
IMPLICIT	Attribue un type implicite à certaines variables
INTEGER	Déclaration pour variables entières
INTRINSIC	Déclaration pour fonctions intrinsèques
LOGICAL	Déclaration pour variables logiques
MAP	Débute un groupe de variables dans une STRUCTURE
NAMELIST	déclare un nom de groupe pour un ensemble de variables
PARAMETER	Donne un nom à une constante
REAL	Déclaration de type réel
RECORD	Attribue une structure à un nom de variable
STRUCTURE	Déclaration d'une structure de différents types de variables
UNION	Partage de mêmes emplacements mémoire pour plusieurs variables

PROGRAMMES ET SOUS-PROGRAMMES

PROGRAM	Début de programme
FUNCTION	Sous-programme de type fonction
SUBROUTINE	nom de Sous-Programme
CALL	Appelle et éxécute un sous-programme
RETURN	Retour de sous-programme ou fonction

FICHIERS

BACKSPACE	Positionne le pointeur de fichier sur l'enregistrement précédent
CLOSE	Fermeture de fichier
ENDFILE	Mise en place caractère fin de fichier

INQUIRE	Examen propriétés de fichiers
LOCKING	Verrouillage de fichiers
OPEN	Ouverture de fichier
READ	Lecture
REWIND	Pointe sur le début du fichier
WRITE	Ecriture

STRUCTURES ALTERNATIVES

CASE	Cas où (F90)
CASE DEFAULT	Idem (F90)
ELSE	Sinon de la structure SI... ALORS... SINON
END IF	Fin construction alternée SI...
END SELECT	Fin d'un bloc Cas où (F90)
IF	Structure alternative SI
SELECT CASE	Cas OU

BOUCLES

CONTINUE	Instruction désuette, sans effet.
CYCLE	Effectue un saut au début d'une boucle (F90)
DO	Boucle **Pour ou Iterer**
DO WHILE	Boucle **Tant Que** (F90)
END DO	Fin de boucle **pour** ou **tant que** ou **iterer** (F90)
EXIT	Sortie prématurée d'une boucle **DO**

ALLOCATION MEMOIRE

ALLOCATE	Allocation dynamique de mémoire (F90)
DEALLOCATE	Libère l'espace réservé par ALLOCATE (F90)

ENTREES-SORTIES

READ	Lecture
WRITE	Ecriture

ANNEXE B

CLASSEMENT DES INSTRUCTIONS ORDRE ALPHABETIQUE

Note

Voici la liste des instructions du langage FORTRAN. Lorsque l'instruction est particulière au FORTRAN 90, on trouvera la mention F90. Cette liste n'a pour but que de rappeler l'existence d'une instruction, mais pas d'en donner l'usage. L'utilisateur aura avantage à consulter l'aide en ligne fournie avec les compilateurs FORTRAN.

ALLOCATE	Allocation dynamique de mémoire (F90)
ASSIGN	Attribue la valeur d'une étiquette à un entier
AUTOMATIC	Les variables sont placées de préférence dans la pile (F90)
BACKSPACE	Positionne le pointeur de fichier sur l'enregistrement précédent
BLOCK DATA	Identifie un sous-programme bloc de données dans lequel on peut initialiser variables et tableaux
BYTE	Déclaration, équivalent à INTEGER*1, variable sur 1 octet (F90)
CALL	Appelle et exécute un sous-programme
CASE	Cas où (F90)
CASE DEFAULT	Idem (F90)
CHARACTER	Déclaration pour variables alphanumériques
CLOSE	Fermeture de fichier
COMMON	Variables globales, partage par plusieurs modules
COMPLEX	Déclaration pour variables complexes
CONTINUE	Instruction désuète, sans effet.
CYCLE	Effectue un saut au début d'une boucle (F90)
DATA	Initialisation de variables
DEALLOCATE	Libère l'espace réservé par ALLOCATE (F90)
DIMENSION	Déclaration pour tableaux
DO	Boucle Pour
DO WHILE	Boucle Tant Que (F90)
DOUBLE ...	Déclaration double précision
ELSE	Sinon de la structure SI... ALORS... SINON
END	Fin de module (programme, sous-programme,..)
END DO	Fin de boucle pour ou tant que (F90)
ENDFILE	Mise en place caractère fin de fichier
END IF	Fin construction alternée SI...
END MAP	
END SELECT	Fin d'un bloc Cas où (F90)
END STRUCTURE	Fin d'une structure (F90)
END UNION	Fin d'une instruction de partage mémoire UNION (F90)
ENTRY	Spécifie un point d'entrée secondaire dans un sous-programme (à éviter)
EQUIVALENCE	Partage location mémoire identique pour plusieurs variables de nom et de type différents
EXIT	Sortie prématurée d'une boucle DO

Supprimé : ¶

JJH, 05/2005

EXTERNAL	Identifie un nom comme étant un sous-programme ou une fonction
FORMAT	Format de lecture ou écriture
FUNCTION	Sous-programme de type fonction
GOTO	Saut
IF	Structure alternative SI
IMPLICIT	Attribue un type implicite à certaines variables
INCLUDE	Lecture et insertion de fichiers dans le programme au moment de la compilation (F90)
INQUIRE	Examen propriétés de fichiers
INTEGER	Déclaration pour variables entières
INTERFACE TO	Déclaration permettant au compilateur de vérifier les instructions d'appel à des sous-programmes (F90)
INTRINSIC	Déclaration pour fonctions intrinsèques passées en argument
LOCKING	Verrouillage de fichiers (F90)
LOGICAL	Déclaration pour variables logiques
MAP	Débute un groupe de variables dans une STRUCTURE (F90)
NAMELIST	déclare un nom de groupe pour un ensemble de variables (F90)
OPEN	Ouverture de fichier
PARAMETER	Donne un nom à une constante
PAUSE	Arrêt temporaire du programme
PRINT	Sortie écran
PROGRAM	Début de programme
READ	Lecture
REAL	Déclaration de type réel
RECORD	Attribue une structure à un nom de variable (F90)
RETURN	Retour de sous-programme ou fonction
REWIND	Pointe sur le début du fichier
SAVE	Maintien des variables dans des sous-programmes
SELECT CASE	Cas OU (F90)
STOP	Arrêt programme
STRUCTURE	Déclaration d'une structure de différents types de variables (F90)
SUBROUTINE	Nom de Sous-Programme
UNION	Partage de mêmes emplacements mémoire pour plusieurs variables (F90)
WRITE	Ecriture

ANNEXE C

FONCTIONS INTRINSEQUES

CLASSEMENT PAR USAGE DEDIE

Types de données et abréviations :

gen	plusieurs types possibles
i	entier 1, 2, 4 octets
i2	entier 2 Octets
i4	entier 4 octets
r	réel 4 ou 8 octets (simple ou double précision)
r4	réel 4 octets
r8	réel 8 octets
l	logique 1, 2, 4 octets
l1	logique 1 octet
l2	logique 2 octets
l4	logique 4 octets
cx	complexe 8 ou 16 octets
cx8	complexe simple précision
cx16	complexe double précision
cr	caractère

Conversion de type

Fonction	Définition	Types Argument / fonction
CHAR(i)	conversion de type	i / cr
CMPLX(gen,gen)	conversion en complexe	i, r, cx / cx
DBLE(gen)	conversion de type	i, r, cx / r8
DCMPLX(genA[,genB])	conversion de type	i, r, cx / cx16
DFLOAT(gen)	conversion de type	i, r, cx / r8
DREAL(cx16)	conversion de type	cx16 / r8
FLOAT(i)	conversion de type	i / r4
HFIX(gen)	conversion de type	i, r ou cx / i2
ICHAR(cr)	conversion de type	cr / i
IDINT(r8)	conversion de type	r8 / i
IFIX(r4)	conversion de type	r4 / i
INT(gen)	conversion de type	i, r ou cx / i
INT1(gen)	conversion de type	i, r ou cx / i1
INT2(gen)	conversion de type	i, r ou cx / i2
INT4(gen)	conversion de type	i, r ou cx / i4
JFIX(gen)	conversion de type	i, r, ou cx / i4
REAL(gen)		conversion de type i, r, cx / r4
SNGL(r8)	conversion de type	r8 / r4

Chaînes de caractères

Fonction	Définition	Types Argument / fonction
INDEX(crA, crB[,l])	recherche chaîne B dans A	cr / int
LEN(cr)	longueur d'une chaîne	cr / i
LGE(crA, crB)	comparaison chaine A >= B	cr / l
LGT(crA, crB)	A > B	cr / l
LLE(crA, crB)	A <= B	cr / l
LLT(crA, crB)	A < B	cr / l
SCAN(crA, crB[,l])	recherche caractère	cr / i
VERIFY(crA, crB[,l])	non présence d'une chaîne	cr / i

Opérations binaires

Fonction	Définition	Types Argument / fonction
BTEST(i, i)	test sur bit	i / l
IAND(iA, iB)	produit logique	i / identique
IBCHNG(iA, iB)	change valeur bit	i / identique
IBCLR(iA, iB)	RàZ bit	i / identique
IBSET(iA, iB)	Rà1 bit	i / identique
IEOR(iA, iB)	ou exclusif	i / identique
IOR(iA, iB)	ou	i / identique
ISHA(iA, iB)	décalage arithmétique	i / identique
ISHC(iA, iB)	rotation	i / identique
ISHFT(iA, iB)	décalage logique	i / identique
ISHL(iA, iB)	décalage logique	i / identique

Fonctions trigonométriques

Fonction	Définition	Types Argument / fonction
ACOS(r)	Arc cosinus	r / identique
ASIN(r)	arc sinus	r / identique
ATAN(r)	arc tangente	r / identique
ATAN2(rA, rB)	arc tangente(A/B)	r,r / identique
CCOS(cx8)	cosinus	cx8 / cx8
CDCOS(cx16)	cosinus	cx16 / cx16
CDSIN(cx16)	sinus	cx16 / cx16
COS(gen)	cosinus	r ou cx / identique
COTAN(r)	cotangente	r / identique
CSIN(cx8)	sinus	cx8 / cx8
DACOS(r8)	arc cos	r8 / r8
DASIN(r8)	arc sin	r8 / r8
DATAN(r8)	arc tangente	r8 / r8

DATAN2(r8A, r8B)	arc tan A/B	r8 / r8
DCOS(r8)	cos	r8 / r8
DCOTAN(r8)	cotangente	r8 / r8
DSIN(r8)	sinus	r8 / r8
DTAN(r8)	tangente	r8 / r8
SIN(gen)	sinus	r, cx / identique
TAN(r)	tangente	r / identique

Mis en forme : Français (France)

Trigonométrie hyperbolique

Fonction	Définition	Types Argument / fonction
COSH(r)	cos hyperbolique	r / identique
DCOSH(r8)	cos hyperbolique	r8 / r8
DSINH(r8)	sinus hyperbolique	r8 / r8
DTANH(r8)	tangente hyperbolique	r8 / r8
SINH(r)	sinus hyperbolique	r / r
TANH(r)	tangente hyperbolique	r / identique

Gestion mémoire

Fonction	Définition	Types Argument / fonction
ALLOCATED(tableau)	status d'allocation dynamique	gen / l
LOC(gen)	Adresse	tout / i2 ou i4
LOCFAR(gen)	Adresse segmentée	tout / i4
LOCNEAR(gen)	Adresse non segmentée	tout / i2

Numériques

Fonction	Définition	Types Argument / fonction
ABS(gen)	Valeur Absolue	i, r, cx / identique
AIMAG(cx8)	partie imaginaire de cx8	cx8 / r4
AINT(r)	Valeur tronquée	r / identique
ALOG(r4)	Log naturel	r4 / r4
ALOG10(r4)	Log décimal	r4 / r4
AMAX0(i, i, ...)	Maxi	i / r4
AMAX1(r4, r4,...)	Maxi	r4 / r4
AMIN0,AMIN1	cf. AMAX	
AMOD(r4, r4)	reste de division	r4 / r4
ANINT(r)	arrondi	r / identique
CABS(cx)	valeur absolue	cx / r
CDABS(cx16)	valeur absolue	cx16 / r8
CDEXP(cx16)	exponentielle	cx16 / cx16
CDLOG(cx16)	logarithme naturel	cx16 / cx16
CDSQRT(cx16)	racine carrée	cx16 / cx16
CEXP(cx8)	exponentielle	cx8 / cx8
CLOG(cx8)	log naturel	cx8 / cx8

CONJG(cx8)	complexe conjugué	cx8 / cx8
CSQRT(cx8)	racine carrée	cx8 / cx8
DABS(r8)	valeur absolue	r8 / r8
DCONJG(cx16)	complexe conjugué	cx16 / cx16
DDIM(r8A, r8B)	différence positive	r8 / r8
DEXP(r8)	exponentielle	r8 / r8
DIM(genA, genB)	différence positive	i ou r / identique
DIMAG(cx16)	partie imaginaire complexe	cx16 / r8
DINT(r8)	troncature	r8 / r8
DLOG(r8)	log naturel	r8 / r8
DLOG10(r8)	log décimal	r8 / r8
DMAX1(r8A, r8B, ...)	Maxi	r8, r8
DMIN1(r8A, r8B, ...)	Mini	r8 / r8
DMOD(r8A, r8B)	reste	r8 /r8
DNINT(r8)	arrondi	r8 / r8
DPROD(r4A, r4B)	produit double précision	r4 / r8
DSIGN(r8A, r8B)	transfert de signe	r8 / r8
DSQRT(r8)	racine carrée	r8 / r8
EOF(i)	fin de fichier	i / l
EPSILON(gen)	plus petit incrément	r / r
EXP(gen)	exponentielle	r ou cx / identique
HUGE(gen)	plus grand nombre positif	i ou r / identique
IABS(i)	valeur absolue	i / i
IDIM(iA, iB)	différence positive	i / i
IDNINT(r8)	arrondi	r8 / i
IMAG(cx)	partie imaginaire	cx / r
ISIGN(iA, iB)	transfert de signe	i / i
LOG(gen)	logarithme	r / identique
LOG10(gen)	log décimal	r / identique
MAX(genA, genB[,...])	maximum	i ou r / identique
MAX0(iA, iB[,...])	maximum	i / i
MAX1(r4A, r4B[,...])	maximum	r4 / i
MAXEXPONENT(r)	plus grand exposant pour type	r / r
MIN(genA, genB[,...])	minimum	i ou r / identique
MIN0(iA, iB[,...])	minimum	i / i
MIN1(r4A, r4B[,...])	minimum	r4 / i
MINEXPONENT(r)	plus petit exposant pour type	r / r
MOD(genA, genB)	reste	i ou r / identique
NEAREST(r, directeur)		
NINT(r)	arrondi	r / i
NOT(i)	complément logique	i / identique
PRECISION(gen)	nombre de chiffres significatifs	r / i
SIGN(genA, genB)	transfert de signe	i, r / identique
SQRT(gen)	racine carrée	r ou cx / identique
TINY(r)	plus petite valeur positive	r / r

Mis en forme : Français (Canada)