Université Paris XIII

Formation d'Ingénieurs de l'Institut Galilée MACS-2

FORTRAN 77

(en attendant mieux)

Ph. d'Anfray
philippe.anfray@cea.fr

(Version 0.2 : Juin 1998)
modifications mineures 2002, 2003

INTRODUCTION

Le langage `Fortran`

Il s'agit de présenter le langage, donc l'outil de programmation `Fortran 77` avec une vision, disons plus moderne. On essaiera donc de montrer comment représenter et exprimer en `Fortran` les «entités usuelles» de la programmation : données et procédures. On essaiera aussi d'insister sur les points forts et les points faibles -donc les dangers- de ce langage, le tout dans une perspective réalisation de grand logiciels scientifiques.

Cette présentation se veut une sorte de «guide» auquel on peut se référer rapidement. Ainsi chaque trait du langage est illustré par un ou plusieurs exemples.

Depuis l'invention du langage `Fortran` dans les années 50 les calculateurs ont largement évolué proposant à leurs utilisateurs de nouveaux modèles de programmation liés à des modifications architecturales : machines *vectorielles*, *parallèles* ; un nouveau domaine des sciences de l'ingénieur a aussi vu le jour il s'agit du **Génie Logiciel**, ses techniques visent à maitriser le développement des applications de grande taille que l'on ne pouvait pas imaginer il y a quarante ans, citons seulement *l'approche orientée objet*.

Curieusement, dans le domaine du calcul scientifique, `Fortran` est resté l'outil de base. Le langage lui-même et les pratiques de programmation ont bien peu évolué au regard de l'environnement informatique. On explique cela (partiellement) par la très grande quantité de logiciels déjà écrits en `Fortran` et par la puissance des compilateurs donc l'efficacité du code généré sur les calculateurs cible.

`Fortran 77` est *en apparence* un langage simple à utiliser. En effet, il n'est pas «contraignant» pour le programmeur et il est possible de développer rapidement une petite application «numérique». Normal, c'est fait pour cela. En caricaturant on pourrait dire que `Fortran` est fait pour traiter des problèmes à une dimension «**1D**». Si l'on veut passer au «**2D**», la situation se dégrade et lorsque l'on arrive au «**3D**» l'ensemble devient souvent ingérable surtout si il est le résultat d'une «conception incrémentale» comme c'est généralement le cas…C'est la rançon de l'apparente simplicité, car il n'y a pas dans `Fortran` d'outils pour

gérer la complexité d'un logiciel de grande taille.

Réaliser un projet en Fortran implique donc d'utiliser d'autres outils tout au long de ce qu'il est convenu d'appeler le *cycle de vie* du logiciel.

Les équipes de développement se dotent en premier lieu d'un catalogue de normes de programmation : éviter d'utiliser les constructions dangereuses du langage etc...c'est un peu le but de cet ouvrage. Ensuite des analyseurs de code «statiques» permettent de vérifier la cohérence des différentes unités de programmation, enfin des analyseurs «dynamiques» qui travaillent sur des codes instrumentés aident à la mise au point et à l'optimisation finale.

Néanmoins la maintenance et les évolutions des grands logiciels écrits en Fortran restent des opérations délicates dont les coûts réels sont largement supérieurs aux coûts de développement et d'ailleurs rarement estimés à leur juste valeur...Typiquement, il est très difficile d'évaluer l'impact d'une modification dans un grand logiciel écrit en Fortran.

Le langage Fortran pourtant évolue. Il a été bien normalisé dans les années 60 (Fortran 66), puis timidement étendu dans les années 70 (Fortran 77). Fortran 77 est encore le dialecte *industriel* le plus utilisé de nos jours. Une nouvelle norme restée longtemps en gestation sous la dénomination Fortran 8x[1], est apparue récemment sous le nom de Fortran 90, quelques apports de cette nouvelle norme étaient déjà reconnus, à titre d'extensions, par de nombreux compilateurs Fortran 77. Les autres sont plus novateurs, certains restant dans la «philosophie» Fortran d'autre non. Fortran 95 introduit quelques aménagements mineurs à la norme 90 en reprenant des notions jugées utiles du futur High Performance Fortran (ou HPF), tout cela devant être revu pour une hypothétique norme Fortran 2000 dont il est peu probable qu'elle tentera, plus que les autres, de rompre avec un héritage trop lourd. L'ultime (?) avatar défini dans les années 90, HPF, intégre de nouvelles caractéristiques visant à tirer parti des calculateurs massivement parallèles.

Enfin, nous proposons donc d'utiliser Fortran 77 plus un certain nombre d'extensions «universelles» qui sont maintenant dans la norme Fortran 90. Pour promouvoir l'écriture de programmes portables et faciles à maintenir, on trouvera au fil des pages de nombreux conseils méthodologiques qui souvent invitent à ne pas utiliser des caractéristiques «obsolètes» ou dangereuses du langage.

[1] elle devait voir le jour dans les années 80...

Table des matières

Chapitre 1

Les données

1.1 Identificateurs

La norme `Fortran 77` limite à **six** le nombre de caractères alphanumériques utilisables pour un identificateur. Le premier doit être alphabétique. Il est difficile de donner des *noms parlants* d'autant plus que `Fortran` ne distingue pas minuscules et majuscules. Ces identificateurs :

```
      xmin,  Xmin,  XMIN,  xMIN
```

sont tous **identiques**.

En fait la plupart des compilateurs `Fortran 77` admettent l'extension `Fortran 90` pour nommer les identificateurs. Un identificateur est alors composé d'au plus **31** caractères alphabétiques, numériques ou _ (blanc souligné)[1]. Le premier doit être alphabétique ou blanc souligné (_).

Voici des identificateurs valides dans ce contexte :

```
      ..., x_min, methode_1, ff_, x123, ...
```

Attention :

▷ dans tous les cas on ne distingue pas minuscules et majuscules.

Conseils méthodologiques :

► utiliser l'extension `Fortran 90` pour donner des noms parlants aux identificateurs ;

[1] Comme en langage **C**

- il n'y a pas de «mots réservés» en `Fortran` mais **ne pas** utiliser les mots-clef du langage comme identificateurs ;
- tout écrire avec les mêmes caractères et de préférence en minuscules plus lisibles (sauf peut être les mots-clef).

1.2 Le type entier

La déclaration standard est :

```
integer i, j, un_entier, t00
```

Il est possible de préciser la longueur occupée en mémoire par une variable de type entier mais il n'est pas du tout garanti que le compilateur utilisera effectivement l'indication donnée par le programmeur. La taille standard, elle, sera le plus souvent de 4 octets mais parfois de 8 selon les machines.

Les variantes suivantes sont à éviter :

```
integer*2  ptit_i, ptit_j
integer*4  i, j
```

Une déclaration `integer*2` qui avait tout son sens à l'époque ou l'on cherchait à tout prix à économiser de la mémoire sera souvent traduite par un compilateur récent en longueur standard (4 voire 8). De plus, si la longueur standard des variables est de (par exemple) 8 octets : taille des «mots» en mémoire, on pourra observer une dégradation des performances si l'on essaie d'accéder à des variables codées sur des fractions de mots (2 ou 4 octets).

Conseil méthodologique :

- utiliser `integer` sans indication de taille pour déclarer des variables entières.

Les constantes entières sont . . .des entiers signés ou non :

```
...
integer i,j
i=10
j=-123
...
```

1.3 Les types réels

Pour les variables de type réel, `Fortran` offre deux possibilités :
– `real`, la simple **précision machine** ;
– `double precision`, bien sûr la double **précision machine**.

```
      real              x_1,  y_1,  z_1
      double precision  xx_1, yy_1, zz_1
```

C'est très simple en apparence mais la réalité est plus complexe ! En effet, pour le programmeur, généralement, simple précision et double précision sous-entendent des variables réelles codées sur 4 octets ou 8 octets : **précision utilisateur**. Or la simple précision machine peut correspondre à 4 octets ou à 8 octets. Dans le second cas, la double précision machine correspondra à des variables codées sur 16 octets (parfois appellé quadruple précision. . .pour l'utilisateur) ce qui est rarement l'effet recherché.

Pour tout compliquer, il est possible, comme pour les entiers de préciser la taille occupée par une variable de type `real` et bien sûr, le compilateur n'est pas obligé de tenir compte de cette information.

Il semble que pour des raisons de portabilité du code, il faille se résoudre à utiliser les déclarations `real*4` et `real*8`, `double précision` causant trop de soucis si la taille standard est de 8.

```
c ..........  utiliser de preference
      real*4 x,  y,  z
      real*8 xx, yy, zz
```

En pratique, si la taille occupée en mémoire par un `real*8` est bien 8 octets (double précision utilisateur), la taille occupée en mémoire par un `real*4` est simplement inférieure ou égale à celle occupée par un `real*8` (comme les `float` et `double` du langage **C**).

Conseils méthodologiques :

- utiliser `real*4` pour demander la simple précision «utilisateur» si elle existe sur la machine ;
- utiliser `real*8` pour demander la double précision «utilisateur» (ce peut être la simple précision machine) ;

► ne pas utiliser `real` (tout seul) et `double precision` pour déclarer des variables réelles.

Les constantes réelles s'écrivent éventuellement avec un exposant.
– si il n'y a pas d'exposant, la constante est de type `real` ;
– si il y a un exposant noté **e**, la constante est de type `real` ;
– si il y a un exposant noté **d**, la constante est de type `double precision` ;

Attention :
▷ le paragraphe précédent invite à utiliser plutôt `real*4` et `real*8`, précision «utilisateur» plutôt que `real` et `double precision`, précision «machine» malheureusement cela rend **non portable** l'écriture des constantes réelles. . .Néanmoins, les conversions automatiques devraient fonctionner de manière satisfaisantes ;
▷ certains compilateurs possèdent des options telles que `-r8` ou `-dp` qui indiquent que l'on considère tous les réels comme étant des `real*8` ou des `double precision`. À utiliser avec précautions ! ces options ne changent pas, bien sûr, les noms de fonctions souvent basés sur la précision machine.

Conseils méthodologiques :
► bien expliquer dans l'en-tête du programme les choix retenus ;
► commenter et regrouper au maximum les parties non-portables.

1.4 Le type complexe

La norme définit seulement le type complexe simple précision «machine» formé de deux réels simple précision «machine» : la partie réelle et la partie imaginaire. De nombreux compilateurs introduisent le type `double complex`, formé de deux réels double précision «machine». Encore une fois on peut préciser la longueur occupée en mémoire par une variable de type `complex`. Nous sommes confrontés aux mêmes problèmes que pour les réels.

```
c .......... simple precision machine
      complex         a,  b,  c
c .......... double precision machine: hors norme
      double complex  aa, bb, cc
```

La même analyse que pour les types réels amène aux mêmes conclusions :

Conseils méthodologiques :

- utiliser `complex*8` pour demander la simple précision «utilisateur» si elle existe sur la machine (i.e. deux `real*4`) ;
- utiliser `complex*16` pour demander la double précision «utilisateur» (i.e. deux `real*8`, ce peut être la simple précision machine) ;
- ne pas utiliser `complex` (tout seul) et `double complex` pour déclarer des variables complexes.

```
c ..........  utiliser de preference
      complex*8   c1,  c2,  c3
      complex*16  dc1, dc2, dc3
```

Attention :

- la double précision «machine» pour les variables de type complexes est une extension à la norme, elle n'est pas disponible sur tous les systèmes.

1.4.1 Quelques fonctions

La bibliothèque standard `Fortran` comporte des fonctions pour accéder aux parties réelles et imaginaires des variables de type complexe. Ces fonctions ont des noms génériques `real` et `imag` qui ne dépendent pas de la «précision machine» utilisée. Une autre fonction `conjg` retourne la conjuguée d'une variable complexe.

```
      ...
      real*4       a, b
      complexe*8   ca, cb
c ..........
      a = real (ca)
      b = imag (ca)
c ..........
      cb = conjg (ca)
      ...
```

```
      ...
      real*8       x, y
c ... (complexe*16: hors norme)
      complexe*16  cx, cy
c ..........
      x = real (cx)
      y = imag (cx)
c ..........
      cy = conjg (cx)
      ...
```

Conseils méthodologiques :

- utiliser seulement les noms «génériques» : `real`, `imag` et `conjg`. Il existe aussi dans la bibliothèque standard `Fortran` des fonctions spécifiques dont les noms dépendent de la précision «machine» des variables utilisées, par exemple `aimag` travaille sur un `complex` et retourne un résultat `real` etc. . . ;
- de toutes façons il est préférable de ne pas mélanger les précisions.

Les constantes complexes sont formées de deux constantes réelles, entre parenthèses séparées par une virgule. Tout ce qui a été dit sur les constantes réelles s'applique donc !

```
      ...
      complexe*8  c_1, c_2
      c_1= (+1e4    , -234.67e-2)
      c_2= (12.5e+23,  2.3      )
      ...
```

1.5 Le type logique

Le type de données `logical` sert, en `Fortran` à manipuler des variables booléennes.

```
      logical fini, ok
```

Comme pour les autres types, il est possible de préciser la longueur occupée en mémoire par une variable de type logique, la taille standard sera elle de 4

ou 8 (mais oui ! !) octets selon les machines. Si l'on précise la taille on aura les variantes :

```
c .... a eviter
      logical*1 ok
      logical*2 fini
      logical*4 test
```

encore une fois, il n'est pas du tout garanti que le compilateur utilisera effectivement l'indication donnée par le programmeur, et les déclarations «exotiques» seront souvent traduites en taille standard.

Conseil méthodologique :

- utiliser `logical` sans indication de taille pour déclarer des variables booléennes.

Les constantes logiques peuvent prendre deux valeurs prédéfinies `.true.` (vrai) ou `.false.` (faux) :

```
      ...
      logical ok, fini
      fini = .false.
      ok   = .true.
      ...
```

Conseils méthodologiques :

- utiliser seulement les formes `.true.` ou `.false.` ;
- utiliser les variables logiques uniquement pour y stocker les valeurs `.true.` ou `.false.` (les normes précédentes de `Fortran` permettaient d'y mettre à peu près n'importe quoi...).

1.6 Les chaînes de caractères

Ce type de données a été introduit seulement dans `Fortran 77`. Il permet de manipuler des chaînes de caractères. La taille **maximale** de ces chaînes doit être explicitement déclarée. La taille par défaut est de **1**.

```
c ... la taille par defaut est 1.
      character    a, b
      character*1  c
      character*20 mot_1, mot_2
      character*80 ligne
```

La norme ne précise absolument pas la taille occupée réellement en mémoire par une chaîne de caractère. Souvent, un caractère occupe un octet mais la taille totale est généralement arrondie à un nombre entier de mots mémoire pour en optimiser la gestion.

De plus la taille **déclarée** d'une chaîne est mémorisée et donc aussi stockée. Elle peut être récupérée via une fonction standard de la bibliothèque `Fortran` : **`len`** .

```
      character*20 mot
      integer l
      ...
      l = len(mot)
c ....donc 20, taille declaree de mot
      ...
```

Ainsi, dans l'exemple ci-dessus, quelque soit la longueur «utilisateur» de la chaîne stockée dans la variable `mot`, cette fonction retournera toujours, pour `mot`, la valeur `20`.

Remarque :

⇒ la norme prévoit que l'on peut insérer une virgule avant la liste d'identificateurs lorsque l'on déclare des variables de type `character`. Ainsi `character*60, ligne` est une déclaration valable mais cette virgule ne sert à rien. Ne pas l'utiliser.

Les constantes chaînes de caractères s'écrivent entre **simples quotes**. Si l'on veut faire figurer une quote dans la chaîne, il suffit de la doubler :

```
    ...
    character*15 chaine1, chaine2, chaine3
    chaine1 = 'petite chaine'
    chaine2 = 'p''tite chaine'
    chaine3 = 'une tres grande chaine'
    ...
```

`chaine1` contient `petite chaine` et `chaine2` `p'tite chaine` , si la constante est plus petite que la chaîne, on «remplit» à droite avec des blancs. En revanche, `chaine3`, elle, est trop petite pour contenir la constante `une tres grande chaine` . Dans ce cas on tronque simplement à la longueur déclarée de `chaine3` qui contiendra seulement `une tres grande` .

Remarques :

⇒ l'opérateur `len` lui renverra toujours la valeur `15` pour `chaine1`, `chaine2` et `chaine3`, c'est la dimension déclarée d'une chaîne de caractères qui compte ;

⇒ pour initialiser «à blanc» une chaîne de longueur quelconque il suffit donc de faire :

```
...
chaine = ' '
...
```

Conseil méthodologique :

► ne pas utiliser d'autres délimiteurs pour les constantes chaînes de caractères, notamment la double quote (**"**).

1.7 Déclarations

Le langage `Fortran` n'impose pas que toutes les entités manipulées au cours d'un programme soient explicitement déclarées. Une variable est déclarée implicitement dès qu'elle est rencontrée par le compilateur. Pire, selon la première lettre du nom de l'indentificateur ce dernier effectue un typage implicite :

– la variable sera de type `integer` si le nom commence par **`i, j, k, l, m`** ou **`n`**.

– la variable sera de type `real` si le nom commence par tout autre lettre (de **`a`** à **`h`** et de **`o`** à **`z`**)

Il n'est donc pas obligatoire d'utiliser les déclarations présentées aux paragraphes précédents. Néanmoins cela est très dangereux, une faute d'orthographe ou une faute de frappe pouvant conduire à un programme syntaxiquement correct mais . . .**faux** !

Conseil méthodologique :

► toujours déclarer les variables et. . .lire la suite.

1.7.1 La déclaration `implicit`

Il est possible de changer le typage implicite des variables non déclarées à l'aide de la déclaration `implicit`. Cette déclaration est suivie, entre parenthèses, d'une liste de lettres ou d'intervalles de lettres tels que **`a-z`** qui signifie «de **`a`** à **`z`**».

```
implicit character*20 (c)
implicit logical      (k, l)
implicit real*8       (a,b,d-h, o-z)
```

Dans l'exemple ci-dessus, après les déclarations `implicit`, les variables non déclarées (explicitement...) et dont le nom commence par **`k`** ou **`l`** seront implicitement typées en `logical`, celles dont le nom commence par **`c`** en `character*20`, celles dont le nom commence par **`a`**, **`b`**, une lettre entre **`d`** et **`h`** ou une lettre entre **`o`** et **`z`** en `real*8`. Rien n'est changé pour les variables dont les noms commencent par **`i, j, m, n`** qui sont donc typées implicitement en `integer`.

Ces déclarations `implicit` doivent se trouver avant toutes les (éventuelles) déclarations de variables.

Conseil méthodologique :

- ne **jamais** utiliser cette forme de la déclaration `implicit` et...lire la suite.

1.7.2 Pas d' «implicite»

Une forme plus intéressante de la déclaration `implicit` mais qui malheureusement est une extension `Fortran 90` est :

```
...
implicit none
...
```

qui impose de **déclarer toutes les variables** utilisées dans le programme. Ouf ! la déclaration `implicit none` fait parti des extensions qualifiées ici d'«universelles».

Conseil méthodologique :

- toujours utiliser `implicit none` et donc déclarer, par la suite, toutes les variables.

1.8 Déclarer des constantes

La déclaration `parameter` permet de déclarer qu'un identificateur représente une constante tout en fixant sa valeur. Toute tentative ultérieure pour modifier cette valeur provoquera une erreur soit à la compilation (dans la même unité de programmation, le compilateur peut «voir» l'erreur) soit à l'exécution.

La déclaration `parameter` est suivie, entre parenthèses, d'une liste d'affectations selon le schéma `nom_de_variable = expression_constante`.

```
...
implicit   none
integer    dimension
parameter (dimension=100)
real*8     zero,       un
parameter (zero=0.0d0, un=1.0d0)
...
```

Conseil méthodologique :

- ► déclarer et initialiser avec `parameter` **toutes** les constantes utilisées dans une unité de programmation.

La partie droite `expression_constante` peut être une véritable expression qui doit pouvoir être évaluée à la compilation. Ainsi, les termes doivent être soit des constantes, soit des identificateurs déjà définis dans un ordre `parameter`.

```
...
integer     nbe, dim_1, dim_2
parameter  (nbe  =100)
parameter  (dim_1=nbe+2), dim_2=nbe*(nbe+1))
...
```

Ici, par exemple, `dim_1` et `dim_2` serviront à «dimensionner» les tableaux d'une application en fonction des caractéristiques (ici le paramètre `nbe`) du problème à traiter. La seule modification de `nbe` mettra à jour l'ensemble des constantes du programme.

Remarque :

- ⇒ bien sûr, `nbe`, `dim_1` et `dim_2` pourraient être initialisés dans le même ordre `parameter` mais il est plus lisible d'isoler, comme dans l'exemple ci-dessus, les valeurs destinées à être modifiées.

1.9 Le constructeur tableau

Le tableau est le seul type de données structuré défini en `Fortran 77`. Les dimensions sont des constantes entières c'est à dire soit des entiers soit des variables entières définies à l'aide de `parameter`.

```
      ...
      integer    taille
      parameter (taille=2000)
      integer    table(100)
      real*8     x (taille), y(100,100)
      ...
```

Les éléments sont accédés via une notation indicée entre parenthèses. Les indices sont des entiers (variables, constantes ou expressions entières) et la numérotation des éléments **commence à `1`**. Ainsi, avec les déclarations ci-dessus, l'on peut écrire :

```
      ...
c .........  acces aux elements
      ...
      table(10)=...
      ...      = y(10, 20)
      ...
```

En `Fortran`, les tableaux à deux dimensions sont rangés *par colonnes*. Ainsi, dans la mémoire de l'ordinateur, les éléments sont rangés dans l'ordre suivant :

```
Rangement des elements de I(20,50) en Fortran:

 I(1,1 ) I(2,1)  ... I(20,1 ) I(1,2)  ...I(20,2) ...
 I(1,50) ...     ... I(20,50)
```

C'est la contraire de ce qui se passe avec le langage **`C`** pour lequel ils sont rangés par *lignes*. D'une façon plus générale, pour le stockage des tableaux multidimensionnels en `Fortran`, c'est l'indice le plus à gauche qui varie en premier.

Attention :

- ▷ en `Fortran`, par défaut, la numérotation des éléments commence à **1** ;
- ▷ une déclaration de tableau sert simplement à réserver, dans la mémoire de l'ordinateur la place correspondante et à indiquer le nombre de dimensions. Il n'y a pas en `Fortran` de controle du «dépassement de tableau». Le programme suivant se compile sans erreur et pire peut s'exécuter sans erreur apparente :

```
      program tableau
      implicit none
      integer x(10)
      integer i
c ... ERREUR: debordement de tableau
      do i=1, 20
         x(i)=i
      enddo
      end
```

  Que se passe-t-il ? En traitant la déclaration, le compilateur a reservé un emplacement pour 10 entiers. Si l'on écrit dans les élements suivants, on «écrase» les valeurs des autres variables qui sont stockées à coté dans la mémoire puis, si on déborde vraiment «beaucoup» on essaiera d'écrire dans une zone contenant les instruction du programme ou des données d'autres applications. Si l'on se contente d'écraser simplement d'autres données du même programme, cela ne provoque pas forcément une erreur à l'exécution. Le petit programme ci-dessus s'exécute «normalement» sur la plupart des systèmes. Les erreurs dues aux débordements de tableau peuvent donc être extrêmement difficiles à détecter. Notons que certains compilateurs proposent, pour valider les programmes, une option de compilation qui permet d'activer à l'exécution un contrôle de débordement ;
- ▷ le nombre maximum de dimensions d'un tableau est de `7`.

Conseil méthodologique :

- ► en `Fortran 77`, il n'y a pas d'allocation dynamique de la mémoire. Tous les tableaux doivent être dimensionnés à la compilation. Il est préférable d'utiliser pour cela des constantes déclarées avec `parameter` ce qui permet de regrouper en un même endroit du programme des valeurs que l'on peut être amené à modifier.

1.9.1 La déclaration `dimension`

Une variante, pour déclarer un tableau consiste à utiliser la déclaration `dimension` :

```
real*8    x
dimension x(100)
...
```

Si l'on choisit cette technique, utiliser un tableau nécessite deux déclarations. Cela ne contribue pas à la lisibilité du programme et encore moins à le rendre compréhensible si les deux déclarations ne sont pas à la suite l'une de l'autre.

Conseil méthodologique :

- ne pas utiliser l'ordre `dimension` pour déclarer un tableau.

1.9.2 Préciser les bornes

Il est possible enfin de préciser la plage de variation de l'indice. Au lieu de la dimension on indique alors les valeurs extrêmes de l'indice séparées par le catactère `:` (deux points).

```
...
integer    table(0:99)
real*8     x    (-10:+9, 10:29)
...
```

Dans cet exemple, `table` est un tableau de `100` éléments mais numérotés à partir de `0`. Ils seront donc accédés par `table(0)`, `table(1)`, ...jusqu'à `table(99)`. De même, le premier indice de `x` varie entre `-10` et `+9` et le second entre `+10` et `+29`.

Conseil méthodologique :

- est-ce vraiment utile ? sans doute parfois mais il faut utiliser cette facilité avec précautions car ce n'est pas dans les «habitudes `Fortran`» et donc, à coup sûr, source de confusions.

1.10 Initialisations

La norme `Fortran` ne prévoit pas d'initialisations par défaut pour les variables d'un programme. Dans certains dialectes, les variables sont initialisées (à zéro) au moment de la déclaration mais il n'est pas possible d'écrire un code portable en s'appuyant sur cette possibilité.

Attention :

▷ avant d'être affectée, la valeur d'une variable est indéterminée.

1.11 (pas de) `data`

Il existe en `Fortran` une instruction `data` qui permet, à la compilation, d'initialiser des variables.

```
      ...
      ...
      real*8     epsil
      data epsi  /1.0d-6/
      ...
```

Ici `epsi` sera initialisé à `1.0d-6`, rien n'empêche, par la suite, de modifier cette valeur.

Remarque :

⇒ l'instruction `data` peut prendre des formes variées qu'il est difficile de décrire de façon synthétique mais que l'on peut encore rencontrer, voici quelques exemples :

```
      ...
      ...
      real*8     epsil
      integer    i, j
c .... "data" regroupes sur une meme ligne.
      data epsi  /1.0d-6/ i, j  /100, 5/
c .... (on peut mettre une virgule avant le i)
      ...
      ...
```

```
      ...
      ...
      real*8     x(100)
      integer    i,j,k
c .... plusieurs fois la meme valeur
c .... (la virgule avant le i est optionnelle)
      data       x/100*0/, i,j,k /3*0/
      ...
      ...
```

```
      ...
      real*8     x(10,10)
      integer    i,j
c .... boucles "implicites" 1 sur la diagonale
c .... des zero ailleurs
      data ((x(i,j),i=1,10),j=  1,i-1)/45*0.0d0/
     &     ((x(i,j),i=1,10),j=i+1, 10)/45*0.0d0/
     &      (x(i,i),i=1,10)           /10*1.0d0/
      ...
```

etc. . .tout cela manque un peu de rigueur.

Conseil méthodologique :

- **ne pas utiliser** l'instruction `data` et encore moins les variantes exposées dans la *remarque* ci dessus. Si l'idée était (probablement) à l'époque de gagner du temps en n'effectuant pas certaines affectations à l'exécution, cette instruction est maintenant inutile ;
- pour initialiser des **constantes**, utiliser la déclaration `parameter`, pour initialiser des **variables** , utiliser une affectation :

```
      ...
c .... declaration de constante
      real*8     epsil
      parameter (epsi= 1.0d-6)
      real*8     x(100)
      integer    i, j, k
c .... initialisations des variables
      i =100
      j =  5
c .... boucle "pour", voir plus loin
      do k=1, 100
         x(k)=100.0d0
      enddo
c .... suite du programme
      ...
      ...
```

Chapitre 2

Les opérateurs

Le langage `Fortran` définit quatre classes d'opérateurs : arithmétiques, relationnels, logiques et enfin spécifiques aux chaînes de caractères. Pour écrire des expressions correctes, il est indispensable de connaître les caractéristiques de chaque opérateur. Une de ces caractéristique est par exemple *l'arité* qui définit le nombre des opérandes d'un opérateur. On différencie, par exemple, l'expression `-z` qui utilise un opérateur **unaire** de l'expression `x-y` où l'on utilise un opérateur **binaire**. Mais les caractéristiques qui nous intéressent ici sont surtout :

l'associativité : l'associativité décrit l'ordre **conventionnel** d'évaluation des expressions. Les opérateurs arithmétiques sont en général associatifs à gauche ainsi l'expression `x+y+z` sera évaluée de gauche à droite comme `(x+y)+z` . Une exception, l'opérateur *élévation à la puissance* noté `**` en `Fortran`, est associatif à droite. L'expression `x**y**zz` sera évaluée de droite à gauche comme `x**(y**z)`;

la précédence : ou priorité relative. C'est encore **une convention** qui permet de déterminer quels sont les opérateurs utilisés en premier lors de l'évaluation d'une expression. Par exemple l' opérateur *multiplication* **`*`** est traditionnellement prioritaire sur l' opérateur *addition* **`+`** et l'expression `x+y*z` est évaluée comme `x+(y*z)` . Pour des opérateurs différents mais de même précédence, on utilise l'associativité, ainsi dans `x/y*z` , les opérateurs *diviser* (`/`) et *multiplier* (`*`) ont la même priorité mais sont associatifs à gauche on évalue donc cette dernière expression comme : `(x/y)*z` .

2.1 Opérateurs arithmétiques

Ces opérateurs s'appliquent aux types numériques (entiers, réels, complexes). L'opérateur **`**`** est l' exponentiation.

Opérateurs unaires	`+` `-`	associatifs à droite.
Opérateurs binaires	`**` `*` `/` `+` `-`	associatif à droite. associatifs à gauche.

Les priorités sont les suivantes, on applique :

1. `**` *puis*
2. `*` , `/` et `%` *et enfin*
3. `+` *et* `-` *unaires ou binaires.*

Attention :

- ▷ l'opérateur `**` est associatif à droite. `x**y**z` est évalué comme `x**(y**z)` ;
- ▷ l'opérateur `/` appliqué à des variables de type `integer` **est** la division euclidienne (`5/2` vaut `2`) ;
- ▷ on considère généralement que les opérateurs unaires ont une plus forte priorité mais ce n'est pas le cas en `Fortran`.
  Ainsi `-1.0**2` vaut `-1.0` , pour obtenir une puissance de `-1` , il faudra écrire : `(-1.0)**2` (qui ouf ! vaut bien `1.0`).

Conseil méthodologique :

- ► éviter lorsque c'est possible d'utiliser l'opérateur `**`. La multiplication est plus précise et plus rapide :

```
      ...
c ..........  ne pas ecrire
c     y = x**2
c     z = t**3
c ...........  mais plutot:
      y = x*x
      z = t*t*t
```

De même, si l'on a besoin de la valeur `x**i` dans une boucle d'indice `i`, il faut mieux la calculer par récurrence :

```
      ...
      xi=1.0
      do i=1, n
c ..... xi vaut x**i, calcul par recurrence
         xi=xi*x
         ...
      enddo
      ...
```

2.2 Opérateurs relationnels

Ces opérateurs s'appliquent aux types numériques (entier, réels, complexes) et aux chaînes de caractères.

Opérateurs de comparaisons	**`.lt.` `.le.` `.ge.` `.gt.`**	associatifs à gauche
Opérateurs égalité, inégalité	**`.eq.` `.ne.`**	associatifs à gauche

Les significations sont les suivantes :
- `.lt.` est l'opérateur `inférieur` (*Less Than*) ;
- `.le.` est l'opérateur `inférieur ou égal` (*Less than or Equal*) ;
- `.eq.` est l'opérateur `égalité` (*EQual*) ;
- `.ne.` est l'opérateur `inégalité` (*Not Equal*) ;
- `.ge.` est l'opérateur `supérieur ou égal` (*Greater than or Equal*) ;
- `.gt.` est l'opérateur `supérieur` (*Greater Than*).

Ces opérateurs ne peuvent être combinés pour former des expressions. Leur résultat est de type `logical` et prend donc l'une des valeurs `.true.` ou `.false.` .

Remarque :

⇒ les opérateurs arithmétiques sont prioritaires sur les opérateurs relationnels. Il est inutile de parenthéser l'expression `x+y.gt.z**t` sauf pour la rendre plus lisible... `(x+y).gt.(z**t)` ce qui est souhaitable !.

Attention :

- l'application de ces opérateurs à des chaînes de caractères **nécessite des précautions** ! ! Si l'on applique l'opérateur égalité ou inégalité (`.eq.` ou `.ne.`) à des chaînes de longueur différentes on procède de la façon suivante :

  1. On cherche la longueur «utile» de la chaîne la plus longue, c'est à dire en enlevant les blancs à droite.
  2. Si cette longueur «utile» est supérieure strictement à la longueur de la chaîne la plus courte alors les deux chaînes sont forcement différentes.
  3. Si cette longueur «utile» est inférieure ou égale à la longueur de la chaîne la plus courte la comparaison s'effectue sur la longueur «commune» des deux chaînes.

```
...
logical      ok1, ok2, non3
character*2  ch
character*10 ch1, ch2, ch3
...
ch   = 'zz'
ch1  = 'zz'
ch2  = 'zz    '
ch3  = 'zzz'
ok1  = ch.eq.ch1
ok2  = ch.eq.ch2
non3 = ch.eq.ch3
...
```

Dans cet exemple `ok1` et `ok2`, ont la valeur `.true.` , les longueurs utiles de `ch1` et `ch2` sont égales à la longueur de `ch`.
`non3` a la valeur `.false.` , la longueur «utile» de `ch3` est supérieure à celle de `ch`.
Le résultat des opérateurs de comparaison (`.lt.` , `.le.` , `.ge.` , `.gt.`) dépend en plus du codage interne des caractères donc de la machine utilisée. Pour écrire un code portable, il faut utiliser les fonctions de la bibliothèque standard `Fortran` : `llt, lle, lge` et `lgt` qui garantissent un résultat conforme au codage `ASCII`.
Un opérateur de comparaison (`.lt.` , `.le.` , `.ge.` , `.gt.`) appliqué à des chaînes de longueur différentes ne travaille que sur la partie «commune» des deux chaînes, il «tronque» donc la chaîne la plus longue. Si les parties

«communes» sont identiques, la chaîne qui a la plus grande partie «utile» est la plus grande.

```
...
logical      ok1, ok2, ok3, non4
character*2  ch
character*10 ch1, ch2, ch3, ch4
...
ch   =’bb’
ch1  =’zz’
ch2  =’zzzz    ’
ch3  =’bbzzzzz’
ch4  =’aaa’
ok1  = ch.lt.ch1
ok2  = ch.lt.ch2
ok3  = ch.lt.ch3
non4 = ch.lt.ch4
...
```

Dans cet exemple, `ok1`, `ok2` et `ok3` ont la même valeur (**`.true.`**), dans les deux premiers cas, la comparaison s’effectue sur la longueur «commune», dans le troisième cas, `ch3` a une longueur «utile» supérieure.
`non4` à la valeur `.false.` car la comparaison s’effectue sur la longueur «commune».

Conseils méthodologiques :

- attention, comparer à zéro des valeurs réelles ou complexes (stockées en mode *flottant*) n’a pas forcément de sens ;
- attention aux chaînes de caractères, **ne pas utiliser** **`.lt.`** , **`.le.`** , **`.ge.`** et **`.gt.`** mais les fonctions de la bibliothèque standard `Fortran` `llt, lle, lge` et `lgt` ;
- **ne pas utiliser** ces opérateurs avec des variables de type `logical`. Il existe d’autres opérateurs pour tester l’égalité ou l’inégalité de variables booléennes.

2.3 Opérateurs logiques

Les opérateurs logiques permettent de manipuler des variables booléennes de type `logical`.

Opérateurs unaire (négation)	**.not.**	associatif à droite.
Opérateurs binaires (et, ou équivalent, non-équivalent)	**.and.** **.or.** **.eqv.** **.neqv.**	associatifs à gauche.

Les priorités sont les suivantes on applique :

1. **.not.** *puis*
2. **.and.** *puis*
3. **.or** *et enfin*
4. **.eqv.** et **.neqv.**

Ouf ! pour une fois l'opérateur unaire a bien la plus forte priorité. Mais Les noms des opérateurs rendent les expressions logiques parfois difficiles à déchiffrer…

```
      ...
c ... pas clair:
c ...       l_1.or.l_2.and..not.x.lt.eps.or.i.gt.imax
c ... un peu mieux si on parenthese:
c ...
       if ((l_1.or.l_2).and.
     &     (.not((.x.lt.eps).or.(i.gt.imax)))) then
         ...
```

Attention :

- ▷ la norme ne prévoit pas d'effet de «court-circuit» lors de l'évaluation d'expressions impliquant les opérateurs .and. et .or.. Ainsi dans :

```
      l_1.and.l_2
```

si l_1 est faux, et bien que le résultat de l'expression soit alors déjà connu, l_2 sera aussi évalué et doit être défini. De même dans

```
      l_1.or.l_2
```

si `l_1` est vrai, et bien que le résultat de l'expression soit aussi déjà connu, `l_2` sera aussi évalué et de même doit être défini. Un exemple :

```
      ...
      integer n, p, q
      logical l
      ...
c ..... RISQUE D'ERREUR !!!
      l = (n.ne.0).and.(p/n.ne.q)
      ...
```

ici, si `n` vaut zero (`n.eq.0`), l'expression `p/n` sera quand même évaluée avec sûrement des conséquences néfastes...

Remarque :

⇒ dans l'exemple ci-dessus, les parenthèses ne sont pas obligatoires car les opérateurs relationnels sont prioritaires sur les opérateurs logiques, elles augmentent néanmoins la lisibilité du code.

Conseil méthodologique :

- ► toujours utiliser `.eqv.` et `.neqv.` pour comparer des variables de type `logical` (et non `.eq.` et `.ne.`) ;
- ► éviter toutes les variantes exotiques et aussi l'opérateur «*ou exclusif*» (`.xor.`) que l'on rencontre parfois.

2.4 Opérateurs sur les chaînes de caractères

Le langage `Fortran` est surtout ciblé «calcul scientifique» aussi ses possibilités en matière de manipulation de chaînes de caractères, apport relativement récent, restent assez limitées. Il est possible de les affecter, d'accéder à des sous chaînes et le seul véritable opérateur est la concaténation.

L'affectation ne modifie pas la longueur des chaînes de caractères, fixée une fois pour toutes à la déclaration.

```
     chaine_1 = chaine_2
```

Si `chaine_2` est plus longue que `chaine_1`, on ne copie dans `chaine_1` que le «début» de `chaine_2`, tronqué à la taille déclarée de `chaine_1`. En revanche, si `chaine_2` est plus courte que `chaine_1`, elle est copiée dans

`chaine_1` puis on remplit «à droite» avec des blancs. Cela généralise ce qui a été exposé pour les constantes.

Opérateur binaire (concaténation)	`//`	associatif à gauche.

L'opérateur concaténation `//` sert à «mettre bout à bout» deux chaînes de caractères. Pour la première, il tient compte de la longueur déclarée. Ainsi la chaîne résultat a pour longueur «utile» la somme de la longueur déclarée de la première chaîne et de la longueur «utile» de la seconde. Si le résultat est trop long pour être stocké dans la variable qui doit le recueillir, il est tronqué à la longueur déclarée de cette variable sinon, on «remplit» à droite avec des blancs.

```
      ...
      character*5 a, b
      character*6  c_petit
      character*10 c_grand
      a='aa'
      b='bbb'
      c_petit=a//b
      c_grand=a//b
c ... c_petit contient maintenant 'aa   b'
c ... c_grand contient maintenant 'aa   bbb'
      ...
```

Attention :

- ▷ cet opérateur ne met donc pas bout à bout les parties «utiles» des chaînes ;
- ▷ les troncations éventuelles des résultats ne sont pas des erreurs et donc passent inaperçues. Il convient de bien s'assurer que les chaînes utilisées ont des longueurs suffisantes.

2.4.1 Sous-chaîne

L'accès à une partie (ou sous-chaîne) d'une chaîne de caractères se fait par une variante de la notation indicée utilisant le symbole `:` . Les indices sont des entiers ou des expressions entières. Soient par exemple les déclarations :

```
      ...
      character*20 var_ch
      integer      i,j
```

On a alors quatre cas :

- `var_ch(i :j)` désigne la sous-chaîne commençant au `i`^{eme} caractère (inclus) et se terminant au `j`^{eme} (inclus) ;
- `var_ch(i :)` désigne la sous-chaîne commençant au `i`^{eme} caractère (inclus) et se terminant au dernier caractère de `var_ch`. Cette notation est équivalente à `var_ch(i :len(var_ch))` ;
- `var_ch( :j)` désigne la sous-chaîne commençant au premier caractère de `var_ch` et se terminant au `j`^{eme} caractère (inclus). Cette notation est équivalente à `var_ch(1 :j)` ;
- `var_ch( :)` cette notation est permise mais **inutile**, elle est équivalente à `var_ch(1 :len(var_ch))` et désigne simplement la chaîne `var_ch` en entier.

```
      ...
      character*20 var_ch
      .....
c ..... la premiere moitie de var_ch (debut a 10)
      ...  var_ch (:10)
c ..... la deuxieme moitie (11 a la fin)
      ...  var_ch(11:)
c.......... les 4 caracteres du "milieu" (de 9 a 12)
      ...  var_ch(9:12)
      ...
```

Attention :

- ▷ une sous-chaîne est une chaîne jusqu'a un certain point…Il n'est pas possible d'écrire directement une sous-chaîne de sous-chaîne ;
- ▷ il faut prendre quelques précautions sous peine de comportement imprédictible :
  - le premier indice pour une sous-chaîne doit être inférieur au second ;
  - le second indice d'une sous-chaîne doit être inférieur ou égal à la dimension déclarée de la chaîne ;
  - enfin deux sous-chaînes d'une même variable utilisées à droite et à gauche d'une affectation **ne doivent pas se chevaucher**.

Conseils méthodologiques :

- ne pas utiliser la forme (inutile) **(:)** sans indices ;
- d'une façon générale, se méfier des sous-chaînes, `Fortran` n'est pas un langage fait pour manipuler des chaînes de caractères.

Chapitre 3

Affectation et évaluation

Les règles de précédence et d'associativité associées aux opérateurs permettent de connaître exactement l'ordre d'évaluation d'une expression comportant plusieurs opérateurs. Un exemple :

```
...
logical l, ok
integer i,j
real*8  x,y,z
...
l= .not.ok.and.i.lt.j.or.x+y.ge.z
...
```

Dans cette exemple l'expression est évaluée comme :

```
...
l= ((.not.ok) .and. (i.lt.j)) .or. ((x+y).ge.z)
...
```

Un autre exemple avec uniquement des expressions arithmétiques : :

```
...
integer i,j,k
real*8  x,y,z
...
j= x*y+i/k
...
```

Dans cette exemple l'expression est évaluée comme :

```
...
j= ((x*y)+(i/k))
...
```

Mais un problème supplémentaire se pose ici. En effet, une expression arithmétique, en `Fortran` peut contenir des variables de plusieurs types numériques : entiers, réels ou complexes. Pour chacun de ces opérateur, si les deux opérandes sont de types différents, une conversion sera nécessaire. De même, si le résultat d'une expression doit être affecté à une variable de type différent une conversion s'impose.

Conseils méthodologiques :

- **parenthéser** complètement les expressions n'est jamais pénalisant et garantit un ordre d'évaluation conforme à ce que l'on croit écrire ;
- éviter de mélanger les types dans les expressions numériques. Les conversions pouvant éventuellement entraîner une perte d'information et/ou de précision.

3.1 Les conversions implicites

On considère que les entiers sont les types les plus «faibles», suivis des réels simple précision «machine», des réels double précision «machine», puis des complexes simple précision «machine» et enfin, si ils existent des complexes double précision «machine». Les règles suivantes s'appliquent :

cas des expressions : si les deux opérandes sont de types différents, l'opérande de type «type le plus faible» est converti en type «type le plus fort». L'opération est effectuée et fournit un résultat de type «type le plus fort» ;

cas de l'affectation : si les deux opérandes sont de types différents, l'opérande de gauche est d'abord converti dans le type de l'opérande de droite.

Reprenons l'exemple ci-dessus : **`j= x*y+i/k`**

- on effectue **`(i/k)`**. Les deux opérandes sont de type `integer`, il s'agit d'une **division euclidienne**, le résultat est de type `integer` ;
- on effectue **`(x*y)`**. Les deux opérandes sont de type `real*8`, le résultat est de type `real*8` ;

– on effectue **`(x*y)+(i/k)`**. Le premier opérande est de type `real*8` ; le second est de type `integer`, il sera converti en `real*8` avant d'effectuer l'addition, Le résultat est de type `real*8` ;
– On affecte le résultat **`j= ((x*y)+(i/k))`**. L'opérande de droite est de type `integer` ; l'opérande de gauche est de type `real*8`, il sera converti en `integer` puis affecté.

Remarque :

⇒ si l'on convertit un entier ou un réel en complexe, on obtient un nombre dont la partie imaginaire est nulle. Si l'on convertit un complexe en entier ou en réel, la partie imaginaire est perdue.

Conseil méthodologique :

► ne pas laisser le compilateur effectuer des conversions implicites (donc lire le paragraphe suivant !) et, rappel, d'une façon générale, ne pas mélanger les types dans les expressions.

3.2 Les conversions explicites

La bibliothèque standard `Fortran` contient des fonctions qui permettent de convertir explicitement un type numérique en un autre. Ces fonctions ont des noms génériques, indépendants du type précis de leurs arguments mais qui dépendent du type résultat en terme de «précision machine» et non hélas de «précision utilisateur». Toutes ces conversions ne sont donc pas portables.

– types réels et type entier :

```
c ... real ou double precision  => integer
      j = int (x)
c ... integer => real
      b = real(k)
c ... integer => double precision
      y = dble(l)
      ...
```

La conversion standard réel vers entier se fait en tronquant simplement la partie décimale du réel. Pour des nombres négatifs ce n'est donc pas la partie entière. La conversion entier vers réels se fait soit en simple précision «machine», c'est la fonction `real`, soit en double précision «machine» c'est la fonction `dble` ;

– types complexes et type entier :

```
c ... complex ou double complex => integer
      i = int (c)
c ....integer => complex
      cb = cmplx(k)
      cc = cmplx(k, l)
c ....integer => double complex
      cy = dcmplx(l)
      cz = dcmplx(k, l)
      ...
```

La conversion standard complexe vers entier se fait en convertissant en entier la partie réelle. Même remarque que pour les réels, ce n'est donc pas forcément la partie entière de la partie réelle. Dans l'autre sens, il existe généralement une fonction `dcmplx` associée au type `double complex`. Si l'on ne donne qu'un argument à la fonction `cmplx` (ou `dcmplx`), la partie imaginaire du résultat sera nulle ;

– types complexes et types réels :

```
c ... complex => reel   ou
c ... double complex => double precision
      x  = real (c)
c ... reel ou double precision => complex
      c  = cmplx(x   )
      d  = cmplx(y, z)
c ... reel ou double precision => double complex
      cc = dcmplx(z   )
      dd = dcmplx(u, t)
      ...
```

Convertir un complexe en réel c'est simplement prendre sa partie réelle ! cela résout les problèmes de conversion lors de la récupération des parties réelles. La fonction `imag` fonctionne de la même façon. Dans l'autre sens, comme pour les entiers, si la fonction `cmplx` (ou `dcmplx`) n'a qu'un seul argument, la partie imaginaire du résultat sera nulle.

Attention :

▷ le type `double complex` et la fonction associée : `dcmplx`, ne font pas partie de la norme ;

▷ si les fonctions `cmplx` et `dcmplx` sont appelées avec deux arguments ceux-ci doivent être **exactement** du même type (`integer`, `real` ou bien `double precision`);

Il existe aussi , dans la bibliothèque standard `Fortran` une fonction spécifique pour chaque conversion possible. Là encore, elles se réfèrent à la précision «machine» des variables ; un exemple : `idint` conversion de `double precision` en `integer`...

Conseil méthodologique :

► **toujours** utiliser les noms génériques pour les fonctions de conversion.

3.2.1 Changement de précision

Changer de précision, par exemple de réel simple précision à réel double précision c'est simplement effectuer une conversion. Les mêmes fonctions s'utilisent donc. . .avec les mêmes restrictions : elles sont conçues en fonction de la précision «machine» et non de la précision «utilisateur». Par exemple :

```
      ...
      real             a,  b
      double precision x,  y
      complex          ca, cb
c ... attention double complex n'existe pas forcement
      double complex   cx, cy
      ...
c ... real              => double precision
      x  = dble   (a)
c ... double precision  => real
      b  = real   (y)
c ... complex           => double complex
c ... dcmplx -si le type double complex existe-
      cx = dcmplx (ca)
c ... double complex    => complex
      cb = cmplx  (cy)
```

Conseil méthodologique :

► encore une fois, ne pas mélanger plusieurs «précisions utilisateur» et/ou «machine» à l'intérieur d'un même code.

Chapitre 4

Les programmes

4.1 Formattage

Les instructions écrites en langage `Fortran` doivent être «formattées» d'une façon très rigide, héritée de l'époque des cartes perforées. De plus le langage ne pouvant être décrit simplement par une grammaire formelle, cette syntaxe imposée est restée.

Une instruction `Fortran` doit être écrite sur une ligne entre les colonnes 7 à 72. Si l'instruction comprend plusieurs lignes, les suivantes comprennent alors un caractère *suite* positionné en colonne 6. La colonne 1 sert à indiquer les commentaires, les colonnes 2 à 5 (inclus) sont utilisées pour donner un numéro (étiquette ou encore *"label"*) permettant d'identifier l'instruction. Enfin, tout ce qui suit la colonne 72 est ignoré. Ainsi :

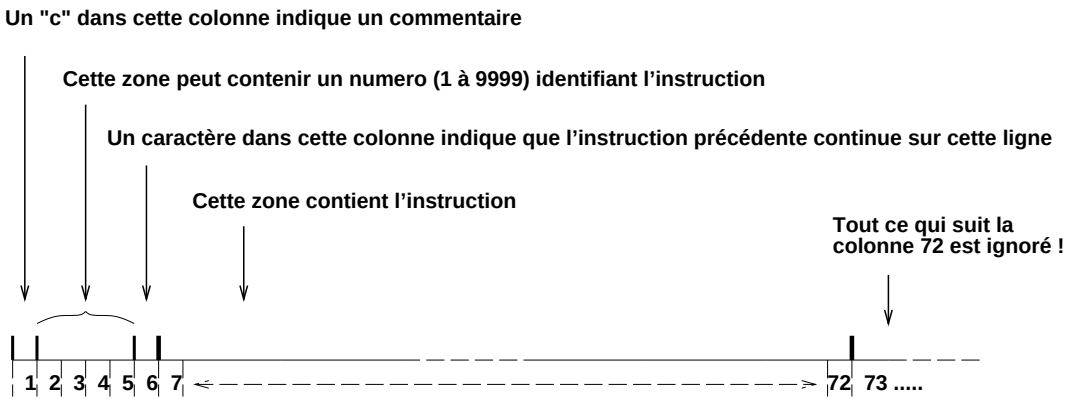

FIG. 4.1 – Une ligne de code `Fortran`

Note historique : les cartes perforées permettaient d'écrire 80 caractères. Les colonnes 73 à 80 servaient à numéroter les cartes au cas ou le paquet tomberait par terre (ce qui arrivait assez souvent...).

En `Fortran`, les «blancs» ne sont pas significatifs, on peut donc, à l'intérieur de la zone 7 à 72, indenter à sa guise les instructions pour rendre le programme plus lisible.

Conseil méthodologique :

- si une instruction s'étend sur plusieurs lignes, toujours utiliser le même caractère de continuation, par exemple **&** :

```
     ...
     integer i, k, entier_1, entier_2, entier_4,
    &        ix, iy, iz, encore_un, et_un_autre
     ...
```

- une instruction doit être codée au maximum sur 20 lignes. Il est prudent de s'interroger sur la façon dont on programme avant d'atteindre cette limite...

4.1.1 Commentaires

Toute ligne qui comporte un caractère **c** en colonne 1 est un commentaire.

```
      ...
c ces lignes sont  ...
c...  des commentaires
      ...
```

En fait n'importe quel caractère en colonne 1 fait généralement l'affaire mais il faut mieux utiliser uniquement les caractères **c** (norme `Fortran 77`) ou **!** (norme `Fortran 90`) pour indiquer les commentaires.

Conseils méthodologiques :

- bien sûr il ne faut pas hésiter à abuser (raisonnablement) des commentaires et pour améliorer la lisibilité du programme, on peut suivre l'indentation des instructions ;
- ne pas utiliser -dans ce contexte «77»- l'extension `Fortran 90` qui permet de mettre, à la fin d'une ligne, un commentaire précédé du caractère **!** ;

- ne pas insérer de lignes blanches qui perturbent certains compilateurs. Utiliser de préférence des «commentaires blancs» ;
- si il y a plusieurs unités de programmation (fonctions, sous programmes) dans le même fichier, il ne doit pas y avoir de commentaires entre ces derniers. En effet si le fichier doit subir un traitement automatique, (gestionnaire de bibliothèque qui découpe le source etc...) on ne saura pas à quelle unité rattacher les commentaires.

4.2 Fichiers à inclure

La gestion du source des programme est grandement facilitée si l'on peut utiliser, comme en langage **C** des «fichiers à inclures» qui contiennent des définitions ou déclarations destinées à être reproduites un grand nombre de fois. Cette possibilité n'existe pas en `Fortran 77` mais il s'agit d'une extension `Fortran 90` assez répandue. Comme en langage **C**, on peut convenir de suffixer par `".h"` les noms de ces fichiers.

```
      program essai
c
      include 'definition.h'
c
c ..........  variables locales
      real    x, y, z
c
      ...
```

Attention :

- ▷ cette extension a ses limites. Le fichier à inclure est le plus souvent recherché **uniquement dans le répertoire courant** contrairement à ce qui se passe en langage **C** ou l'on peut préciser à l'aide de l'option `-I` du compilateur un chemin d'accès pour ces fichiers.

Conseil méthodologique :

- ▶ ne pas hésiter néanmoins à utiliser cette extension très courante. Un cas intéressant est de regrouper des constantes déclarées avec l'instruction `parameter` et/ou des zones communes (voir plus loin).

4.3 Quelques instructions

4.3.1 L'instruction `program`

Un programme `Fortran` peut (mais on devrait dire **doit**) débuter par l'instruction `program`. Le mot-clef `program` est suivi d'un identificateur : le nom de l'application.

```
      program simple
c
c ... pas de declarations implicites
c ... voir ce qui precede!!
c
      implicit none
      integer i,j
      ...
c ... instructions du programme
      ...
      ...
      ...
      end
```

Conseil méthodologique :

► toujours commencer le programme principal par l'instruction `program`.

4.3.2 L'instruction `end`

Un programme `Fortran` doit se terminer par l'instruction `end`. En `Fortran 77` `end` est une instruction *exécutable*, non seulement elle indique la fin du programme (son seul rôle, en `Fortran 66`), mais encore elle en assure la terminaison correcte.

4.3.3 L'instruction `continue`

`continue` n'est pas une instruction exécutable. Son intérêt sera de délimiter des constructions syntaxiques (boucles etc...) et surtout de recevoir une «étiquette». Une étiquette est un numéro de `1` à `9999` obligatoirement codé dans les colonnes 2 à 5 incluses.

```
      ...
  100 continue
      ...
      ...
  200 continue
      ...
      end
```

Conseils méthodologiques :

- dans un programme n'étiqueter que des instructions `continue`. Les étiquettes pourront servir aussi à repérer les `formats` utilisés pour les entrées sorties ;
- en fait, on espère bien pouvoir se passer des instructions `continue`.

4.3.4 L'instruction `stop`

Cette instruction permet d'interrompre définitivement l'exécution du programme. Elle était nécessaire avant la norme 77 pour assurer que le programme se terminait correctement ainsi il était indispensable alors de coder :

```
      ...
      integer i,j
c ... norme fortran 66, l'instruction program n'existe
c ... pas encore, le programme se termine par stop end
      ...
      stop
      end
```

L'instruction `stop` peut être insérée n'importe où dans le code. Le mot-clef `stop` peut être suivi d'une **constante** entière ou d'une chaîne de caractères. Si l'instruction `stop` est exécutée, cette constante sera affichée sur la «sortie standard». La constante entière peut être aussi récupérée par l'environnement d'où l'application a été lancée et servir de «code retour».

```
...
...
stop 4
...
...
stop 'fin anormale'
...
...
end
```

Conseil méthodologique :

- un programme bien construit ne se termine que d'une seule manière : à l'instruction `end` du «programme principal». L'idéal serait de **ne pas utiliser** l'instruction `stop`. Néanmoins, ce conseil n'est pas vraiment réaliste, il est raisonnable d'envisager un sous programme spécial : `subroutine fin_anormale (message, ...)`, qui imprimé un message d'erreur passé en argument et termine le programme correctement avec un `stop` (i.e. après avoir fermé les fichiers etc...).

4.3.5 L'instruction `pause`

Survivance des temps héroique de l'informatique, l'instruction `pause` interrompt l'exécution du programme qui reprendra *après intervention de l'opérateur*. Comme l'instruction `stop`, `pause` peut être suivie d'une constante entière ou chaîne de caractères. Concrètement si l'instruction `pause` est exécutée, la constante éventuelle sera affichée sur la «sortie standard» et l'exécution recommencera lorsque l'utilisateur aura frappé la touche «entrée».

Conseil méthodologique :

- ne pas utiliser l'instruction `pause` dans les programmes. Tout cela avait un sens lorsque des interventions manuelles étaient nécessaires pour assurer le bon déroulement des programmes.

4.4 Les structures de contrôle

4.4.1 La répétition

Le langage `Fortran 77` n'offre qu'une seule structure de contrôle pour exprimer la répétition : la *boucle pour*.

La forme standard comprend une «étiquette» ou *"label"* qui permet de repérer la dernière instruction du corps de boucle. Le compteur de boucle est suivi de sa valeur initiale, de sa valeur finale et d'un incrément optionnel. La valeur par défaut de cet incrément est **1**.

```
      ...
      do 10 i=1, 100
c ..... i va de 1 a 100 par pas de 1
c ..... valeur par defaut de l'increment
c .......la boucle s'arrete sur l'etiquette 10
         ...
         t(i)= ...
         ...
   10 continue
      ...
```

```
      ...
      do 20 i=1, 100, 2
c ..... ici de 1 a 100 par pas de 2
c .......la boucle s'arrete sur l'etiquette 20
         ...
         v(j)= ...
         ...
   20 continue
      ...
```

Conseils méthodologiques :

- n'utiliser que des variables entières (`integer`) comme compteurs de boucle ;
- toujours terminer le corps de boucle par une instruction `continue` ; en cas de boucles imbriquées utiliser **plusieurs** instructions `continue` :

```
    ...
     integer i, j
     real*8  a(100,100)
     ...
     do 200 i=1, 100
        do 100 j=1, 100
           ...
           ...
           a(i,j)= ...
 100     continue
 200 continue
     ...
```

Un exemple de ce qu'il ne faut **pas faire** : pas de `continue`, la même instruction délimite plusieurs boucles :

```
    ...
c ... tres mauvais style
c ... (mais ca marche quand meme)
c
     do 111 i=1, 100
     do 111 j=1, 100
     ...
     ...
 111  a(i,j)= ...
     ...
```

4.4.2 Les formes `do ...enddo` et `dowhile ...enddo`

Il est préférable d' utiliser l'extension `Fortran 90` `do ...enddo` qui permet d'éviter le `continue` muni de son étiquette et de structurer plus clairement le programme. A chaque mot-clef `do` doit correspondre un mot-clef `enddo`. Il est impératif de passer à la ligne après `enddo`. On peut réecrire les exemples précédents avec cette nouvelle forme :

```
      ...
      integer i, j, k
c
      integer t(100), v(100), a(100,100)
c
      ...
      do i=1, 100
c ..... de 1 a 100 par pas de 1 (valeur par defaut)
         ...
         t(i)= ...
         ...
         ...
      enddo
      ...
      do  i=1, 100, 2
c ..... ici de 1 a 100 par pas de 2
         ...
         v(j)= ...
         ...
      enddo
      ...
      ...
c ..... boucles imbriquees
      do i=1, 100
         do j=1, 100
            ...
            ...
            a(i,j)= ...
            ...
         enddo
      enddo
      ...
```

L'extension `do while ...`, réalisant une boucle *tant que*, semble aussi répandue et peut être utilisée sans problèmes surtout qu'elle n'a pas d'équivalent en `Fortran 77` standard. Le mot-clef `do while` est suivi d'une expression logique. La construction se termine bien sûr par `enddo` :

```
      ...
      real*8     eps
      parameter (eps=1.0d-6)
      logical    ok
      real*8     residu
      ...
      ok = .true.
      do while (ok)
         ...
         ...
      enddo
      ...
      ...
      do while (residu.gt.eps)
         ...
         ...
      enddo
      ...
```

4.4.3 Conditionnelle

Le langage `Fortran 77` a été doté, en son temps d'une forme conditionnelle relativement elaborée qui est l'équivalent du *si . . .alors . . .sinon . . .* des langages les plus évolués. La syntaxe utilise les mots-clef `if`, `then`, `else`, `endif`. la partie *sinon* est optionnelle. Il est impératif de passer à la ligne après les mots-clef `then`, `else` et `endif`.

```
      ...
      integer i, j, k
      real*8  x
      ...
      if (i.gt.0) then
c ..... exemple avec if seul
         ...
      endif
      ...
```

```
      ...
      if (x.le.1.0d0) then
c ..... exemple avec partie else
        ...
        ...
      else
        ...
        ...
      endif
      ...
```

Il est bien sûr possible d'imbriquer des constructions `if` :

```
      ...
      if (i.lt.0) then
         ...
      else
         if (j.gt.1000) then
            ...
            ...
         else
            ...
            ...
         endif
      endif
      ...
```

Une variante utile

Pour simplifier la programmation dans le cas de nombreux `if` imbriqués, il faut programmer la variante utilisant le nouveau mot-clef `elseif`. La construction se termine par `endif` et comprend une partie `else` optionnelle. Cette dernière est prise en compte uniquement si tous les tests précédents se sont révélés négatifs :

```
      if (i.le.0) then
         ...
      elseif (k.lt.100) then
c ..... instructions executees si i>0 et k<100
         ...
      elseif (m.lt.1000) then
c ..... instructions executees si i>0, k>=100 et m<1000
         ...
      else
c ..... partie else optionnelle,
c ..... executee ici si i>0, k>=100 et m>= 1000
        ...
      endif
      ...
```

Cette construction peut servir, bien sûr, pour simuler élégament une instruction *choix*.

```
c ... simule "choix i entre":
      if     (i.eq.0) then
c ..... instructions executees si i=0
         ...
      elseif (i.eq.1) then
c ..... instructions executees si i=1
         ...
      elseif (i.eq.2) then
c ..... instructions executees si i=2
         ...
      ...
      else
c ..... partie else optionnelle,
        ...
      endif
      ...
```

Des variantes inutiles

Il existe deux autres formes de conditionnelles que nous citons pour mémoire car on les rencontre encore hélas dans des applications. Ce sont :

Le `if` dit *logique* : il s'écrit sur une seule ligne avec seulement le mot-clef `if`. L'instruction est exécutée si la condition est vraie.

```
      ...
      ...
      if (i.le.0) t(i)=x(i)+y(i)
      ...
```

Le `if` dit *arithmétique* : réellement archaïque ! Si l'expression entière qui suit le `if` est négative, l'exécution du programme reprendra à l'instruction qui a pour étiquette la première valeur donnée après le `if`, si cette expression est nulle on reprendra à la deuxième étiquette, si elle est positive à la troisième.

```
      ...
      ...
      if (k) 11,12,13
      ...
c l'execution reprend ici si k < 0
   11 x = ...
      ...
c l'execution reprend ici si k = 0
   12 x = ...
      ...
c l'execution reprend ici si k > 0
   13 x = ...
      ...
```

Pour corser le tout, les étiquettes peuvent figurer dans le programme dans un ordre quelconque et même se trouver avant l'instruction `if` considérée.

Conseils méthodologiques :

- ne **pas** utiliser le *if logique*, obsolète ; utiliser systématiquement `if ...then ...endif`;
- **ne jamais utiliser** le *if arithmétique*, en contradiction totale avec une structuration correcte des programmes.

4.4.4 Branchement inconditionnel

Lors de l'exécution d'un programme les instructions sont exécutées dans l'ordre où elles sont écrites. Mais l'on peut écrire la forme :

```
      goto numero
      ...
```

L' instruction `goto` permet de provoquer une `rupture de séquence`, l'exécution reprenant à l'instruction étiquetée par `numero`. Cette dernière peut se trouver n'importe où dans l'unité de programmation (programme, sous programme) considérée. Le `goto` doit être évité au maximum car son utilisation est en contradiction totale avec une bonne structuration des programmes. Néanmoins il peut encore être utile pour simuler des structures de contrôle qui n'existent pas en `Fortran 77`, notamment une boucle *«tant que»* si l'on utilise pas l'extension `dowhile`. Voici un exemple :

```
      ...
c ....simulation de  "tant que x>=epsilon faire"
  100 continue
         if (x.lt.epsilon) then
            goto 200
         else
            .....
            .....
            .....
            goto 100
         endif
  200 continue
c ... fin de la boucle "tant que"
      ...
```

Conseils méthodologiques :

- le `goto` doit toujours renvoyer à une instruction `continue`, la seule qu'il soit raisonnable d'«étiqueter» mais...
- ...**ne pas utiliser** l'instruction `goto` sauf, éventuellement et **en le signalant**, pour simuler une boucle *tant que* mais...
- ...dans ce cas **utiliser** plutôt l'extension `do while`.

Des variantes inutiles

Inutiles car encore une fois elles correspondent à une façon de programmer qui n'est plus de mise aujourd'hui. Donnons juste des exemples. Le (dé)branchement «calculé» :

```
      goto (100, 200, 300)  i
      ...
      ...
100   ...
      ...
200   ...
      ...
300   ...
      ...
```

Dans l'exemple ci-dessus si `i` vaut `1` on se débranchera sur l'instruction étiquetée `100`, si `i` vaut `2` celle étiquetée par `200`, etc.... Si `i` est négatif ou supérieur à `3` on ne fera rien.

Il y a aussi le débranchement «assigné». Une instruction `assign`, permet de stoker une étiquette dans une variable entière (qui ne doit être utilisée que pour cela). Cette variable peut, par la suite être référencée par une instruction `goto` :

```
      ...
      integer etiq
      ...
      assign 100 to etiq
      ...
      ...
      goto etiq
      ...
100   ...
      ...
```

Il est possible de préciser, entre parenthèses, la liste des étiquettes «valides». Dans l'exemple ci-dessus cela donnerait :

```
...
goto etiq (100, 200)
...
```

Ce qui se passe quand l'étiquette contenu dans la variable n'est pas dans la liste des possibles n'est pas très clair…

Conseils méthodologiques :

- **ne jamais utiliser** de débranchement calculé. Pour programmer une instruction choix, il faut utiliser `if ...elsif...`.
- …**ne jamais utiliser** de débranchement assigné.

Chapitre 5

sous programmes

Toute application doit être découpée en unités de programmation. En `Fortran`, langage «procédural», les tâches (élémentaires ou non...) effectuant des traitement sur les données s'écrivent sous forme de sous programmes. Le langage `Fortran 77` distingue en fait deux types de sous programmes :

la `subroutine` est un sous programme qui ne retourne pas de valeur ;

la `function` est un sous programme qui retourne une valeur.

5.1 sous programme `subroutine`

Ce type de sous programme se déclare à l'aide du mot-clef `subroutine` et se termine par l'instruction `end` selon le schéma suivant :

```
subroutine nom_sous_pgm (arguments_formels)
...
...
...
...
end
```

Il peut ne pas y avoir d' arguments mais les parenthèses sont obligatoires. Ainsi :

```
      subroutine ssp_1 (x, y)
      ...
      ...
      end
      subroutine ssp_2 ()
      ...
      ...
      end
```

Attention :

- si l'on ne précise pas à l'intérieur du sous programme, le type des arguments, ceux-ci subissent le typage implicite exposé précédemment. Si l'on utilise `implicit none` -et il faut le faire- il est alors obligatoire de typer les arguments :

```
      subroutine ssp_1 (x, y)
      implicit none
c ... arguments (donnees)
      real*8  x, y
c ... variables locales
      integer i,j
      real*8  z
      ...
      ...
      end
```

- un sous programme peut modifier ses arguments (cela sera détaillé par le suite) mais `Fortran` n'offre pas de mécanisme pour déclarer le statut d'un argument («donnée», «donnée modifiée» ou «résultat»).

Conseils méthodologiques :

- utiliser toujours `implicit none` et donc déclarer explicitement le type des arguments ;
- dans les déclarations, bien mettre en évidence, comme dans l'exemple ci-dessus :
  - les arguments ;
  - les variables locales.

  et préciser à l'aide de commentaires le statut des arguments :

– donnée (donc non modifiée), l'argument est accédé seulement en lecture ;
– donnée modifiée, l'argument est accédé en lecture et en écriture ;
– résultat, l'argument n'est accédé qu'en écriture.

Pour appeler un sous programme de type `subroutine`, il faut utiliser le mot-clef `call`. Le nom de la `subroutine` doit être suivi de la liste des arguments réels. S' il n'y a pas d'arguments, les parenthèses sont néanmoins nécessaires :

```
program appli
implicit none
real*8 p1, p2
...
...
call ssp_1 (p1, p2)
...
call ssp_2 ()
...
end
```

Attention :

▷ à l'appel les listes d'arguments formels et réels doivent se correspondre exactement (nombre et types). Le programme appelant «ne connait pas» la séquence d'appel correcte. Il n'y a pas, en `Fortran 77`, de notion de *prototypes* de fonctions comme (par exemple) en langage **C**. Les compilateurs qui traitent indépendament chaque unité de programmation ne peuvent pas tester la validité de l'appel et (pire !) un appel incorrect ne provoquera pas forcement une erreur à l'exécution.

5.1.1 L'instruction `return`

La norme `Fortran 66` imposait de coder l'instruction `return` avant le `end` pour «sortir» du sous programme. Ce n'est plus nécessaire depuis la norme `Fortran 77` : `end` est devenu une instruction «exécutable» et fait le travail du `return`. Il est aussi possible de coder `return` n'importe où dans le sous programme et de provoquer ainsi une sortie prématurée :

```
      subroutine ssp_3 (x, y)
      ...
      ...
      if (...) then
         ...
         ...
c ..... sortie anticipe: pas conseille !!
         return
      endif
      ...
      ...
      ...
c ... return avant end: inutile en fortran 77!!
      return
      end
```

Conseil méthodologique :

- un sous programme bien écrit ne possède qu'un seul point d'entrée et un seul point de sortie. Il n'y a pas lieu d'utiliser l'instruction `return`.

5.1.2 L'instruction `entry`

Cette instruction permet de définir plusieurs sous programmes en une seule unité de programmation, elle n'est **heureusement** plus guère utilisée de nos jours. Nous donnerons juste un exemple :

```
      subroutine ff_1 (x, y, z)
      ...
      ..
      entry ff_2 (x, y)
      ..
      ..
      end
```

L'unité de programmation ci-dessus possède «deux points d'entrée» `ff_1` ou `ff_2`. On peut maintenant utiliser :

```
      program appli
      implicit none
      real*8 a, b, c
      ..
      call ff_1 (a, b, c)
      ...
      call ff_2 (a, b)
      ...
      end
```

Tout cela causant généralement beaucoup d'ennuis ! ! !

Conseil méthodologique :

- un sous programme bien écrit ne possède qu'un seul point d'entrée et un seul point de sortie. **Ne jamais utiliser** l'instruction `entry` dans un sous programme de type `subroutine`.

5.2 sous programme `function`

Ces sous programmes se déclarent à l'aide du mot-clef `function` et se terminent par l'instruction `end`. La `function` retourne une valeur, elle est donc typée :

```
      type function nom_function (arguments_formels)
      ...
      ...
      end
```

Il peut ne pas y avoir d' arguments mais les parenthèses, elles, sont obligatoires comme pour une `subroutine`. Si la fonction ne reçoit pas de type alors les règles de typage implicite s'appliquent au nom de la fonction (aïe !) :

```
      function  sans_type ()
c ... donc real (ici): a eviter!!!
      ...
      ...
      end
```

Il est possibler aussi de déclarer le type dans le corps de la fonction :

```
      function  gen_index (i, j)
      implicit  none
c ... pas clair: a eviter !!!
      integer   gen_index
c     integer   i, j
      ...
      ...
      end
```

Conseils méthodologiques :

- ► toujours typer explicitement une fonction ;
- ► comme pour une `subroutine`, utiliser `implicit none` pour déclarer le type des arguments formels ;
- ► un sous programme bien écrit ne possède qu'un seul point d'entrée et un seul point de sortie. **Ne jamais utiliser** l'instruction `entry` dans un sous programme de type `function`, **ne pas utiliser** non-plus l'instruction `return`.

La fonction `nom_de_function` retourne la dernière valeur affectée à l'identificateur `nom_de_function` :

```
      real*8 function calcul (x, y, z)
      implicit none
c ... arguments
      real*8     x, y, z
c ... variables  locales
      real*8     deux
      parameter (deux=2.od0)
      real*8     local
c ...
      local = ...
      ...
      calcul = local+((x*x+y*y+z*z)/deux)
      end
```

Conseil méthodologique :

- pour être sûr que tout fonctionne bien, il est préférable d'utiliser des variables locales et d'affecter **une fois seulement** `nom_de_function`, juste avant l'instruction `end`.

La fonction peut ensuite être utilisée comme une variable de même type dans une expression, son type doit être déclaré dans le programme appelant :

```
      program util_calcul
      implicit none
c ... fonctions
      real*8  calcul
c ... variables
      real x0, y0, z0, x1, y1, z1, ss
      ...
      ...
      ss = calcul(x0, y0, z0) + calcul (x1, y1, z1)
      ...
      end
```

Conseils méthodologiques :

- de par leur nature, les fonctions sont destinées à être appelées dans des expressions. Si une `function` modifie ses arguments cela peut induire des effets de bords non désirés. Ainsi, un sous programme de type `function` ne doit **jamais** modifier ses arguments ;
- toujours déclarer le type des fonctions utilisées dans une unité de programme. C'est obligatoire si l'on utilise `implicit none` ce qui est bien sûr conseillé ;
- enfin de même que pour les `subroutines`, il faut faire attention au nombre et aux types des arguments réels lors de l'appel d'une `function`, le compilateur ne faisant aucune vérification et une erreur éventuelle n'étant pas forcément détectée à l'exécution.

Dernier problème, il existe en `Fortran` un grand nombre de fonctions prédéfinies (qualifiées d'«intrinsèques») par exemple `cos` pour *cosinus*, etc.... De plus, ces fonctions ont souvent plusieurs noms selon le type des arguments ou du résultat. Il peut arriver qu'un sous programme `function` «utilisateur» porte le même nom qu'une de ces fonctions prédéfinie. Pour garantir que l'on appellera effectivement la fonction «utilisateur», il faut que cette dernière soit déclarée à l'aide du mot-clef `external` dans le programme appelant. Reprenons l'exemple du programme utilisant la fonction `calcul` :

```
      program util_calcul
      implicit none
c ... fonctions
c ... evite les <<collisions>> de noms
      external calcul
      real*8   calcul
c ... variables
      real x0, y0, z0, x1, y1, z1, ss
      ...
      ...
      ss = calcul(x0, y0, z0) + calcul (x1, y1, z1)
      ...
      end
```

Conseil méthodologique :

- dans les nouvelles normes `Fortran 90` etc..., le nombre de fonctions «intrinsèques» a tendance à augmenter de façon inquiétante. Il est sûrement prudent, dès maintenant, de déclarer comme étant `external` toutes les fonctions «utilisateur» ;
- on voit bien que ce mécanisme peut permettre aussi de redéfinir des fonctions «intrinsèques». Cette possibilité ne doit pas être utilisée sciement !

5.3 Les arguments

Rappelons que les *arguments formels* d'un sous programme sont ceux qui sont utilisés dans sa définition. Les variables utilisées lors de l'appel du sous programme sont appelées *arguments réels*. Le lien entre arguments réels et formels est défini par le *mode de passage* des arguments. En langage `Fortran`, **tous les arguments sont passés par adresse**. Dans ce mode, le sous programme reçoit les adresses en mémoire des arguments réels et travaille directement sur eux. Toute modification de l'argument dans le corps du sous programme sera «visible» à l'extérieur.

5.3.1 Simple, Tableau

Le langage `Fortran` ne fait pas de distinction entre arguments simples et tableaux, puisqu'il transmet simplement des adresses en mémoire. Le nom d'un

tableau, correspond à son adresse en mémoire, c'est aussi celle du premier élément. Les suivants sont supposés être rangés de manière contigüe à partir de cette adresse. Un élément de tableau peut bien sûr être passé comme argument simple :

```
      subroutine ff_1 (x, y,z)
      implicit none
      real*8   x, y, z
      ...
      ..
      end
      program appli
      implicit none
      integer  i
      real*8   a
      real*8   tab(100), xloc(100,3)
      ..
      call ff_1 (t(i), xloc(i,2), a)
      ...
      end
```

Examinons maintenant précisement ce qui se passe si l'argument formel est un tableau. Ce qui importe ici est d'indiquer au sous programme que l'argument est un tableau. Cette indication, contrairement à la déclaration de l'argument réel dans le programme appelant ne réservera pas de place dans la mémoire.

Tableau à une dimension

1. Le tableau peut être dimensionné explicitement dans le sous programme. Mais le tableau passé en argument réel a bien sûr déjà été dimensionné dans le programme appelant et préciser la dimension de l'argument formel est inutile, voire limite la généralité du sous programme :

```
      subroutine f_tab_1  (x)
      implicit none
c ... dimensionnnement explicite de x: NON
      real*8   x(100)
      ...
      ..
      end
```

2. On peut penser alors qu'il suffit de mettre n'importe quel nombre. En effet ce qui importe c'est de préciser que l'argument formel est un tableau. On rencontre souvent dans les programmes :

```
      real*8 function f_tab_2  (x)
      implicit none
c ... dimensionnnement a 1 juste pour
c ... indiquer que x est un tableau: NON PLUS
      real*8   x(1)
      ...
      f_tab_2= ...
      end
```

   Ce n'est pas plus satisfaisant que le cas précédent.

3. La norme `Fortran` donne aussi la possibilité d'utiliser le caractère `*`, pour simplement indiquer qu'il s'agit d'un tableau dimensionné dans une autre unité de programme. C'est déjà mieux :

```
      subroutine f_tab_3  (x)
      implicit none
c ... l'etoile indique que x est un tableau
c ... a une dimension: OUI
      real*8   x(*)
      ...
      ..
      end
```

4. Pour éviter tout ennuis, il est prudent de passer aussi au sous programme, la dimension réelle du tableau et de s'en servir pour le dimensionner. La dimension réelle peut être testée dans le sous programme ce qui permet d'éviter bien des mauvaises surprises.

```
      subroutine f_tab_4  (n, x)
      implicit none
c ... Encore mieux !!
c ... le tableau et sa dimension: OUI !!
      integer  n
      real*8   x(n)
      ...
      ..
      end
      programme de_test
      implicit none
      integer    dim
      parameter (dim=100)
      real*8     xx (dim)
      ..
      call f_tab_4  (dim, x)
      ...
      end
```

Conseil méthodologique :

- passer toujours la dimension en même temps que le tableau et dimensionner correctement le tableau dans le sous programme. À l'appel, on passera soit la vrai dimension du tableau, soit sa dimension «utile» dans le contexte du programme appelant. Par exemple, en reprenant l'exemple précédent :

```
      programme de_test
c ... bon exemple !!
      implicit none
      integer    dim, dim_util
      parameter (dim=100)
      real*8     xx (dim)
      ...
      dim_util = 50
      ...
      call f_tab_4  (dim_util, x)
      ...
      end
```

Cette notion de dimension «utile» est souvent utilisée car il n'y a pas d'allocation dynamique en `Fortran 77` et les dimensions déclarées des tableaux sont souvent les dimensions maximales pour traiter une certaine famille de problèmes.

Tableau à plusieurs dimensions

Pour le stockage des tableaux multi-dimensionnels en `Fortran`, c'est l'indice le plus à gauche qui varie en premier. Nous prendrons le cas d'un tableau à deux dimensions, le cas général s'en déduisant facilement. Un tableau à deux dimensions est rangé *par colonnes* et celles-ci sont stockées bout à bout dans le mémoire. Pour accéder à l'élément `(i, j)` d'un tableau, il faut donc connaitre la première dimension du tableau (i.e. la longueur des colonnes) pour calculer le décalage à effectuer dans la mémoire à partir de l'adresse du tableau (donc du premier élément). Plus généralement, pour un tableau à *N* dimensions, le sous programme doit connaitre les *N-1* premières dimensions. La dernière dimension, elle, peut être notée ***** car elle n'est pas utile pour localiser les éléments.

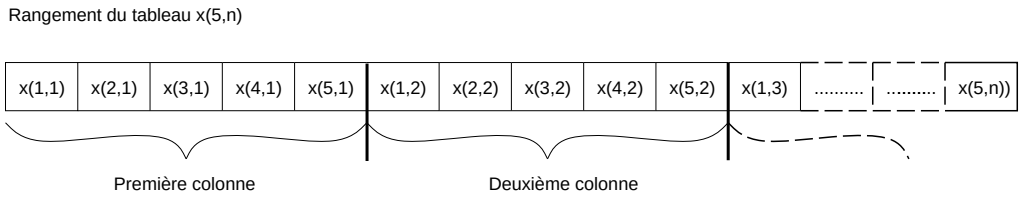

FIG. 5.1 – Rangement d'un tableau à deux dimensions en mémoire

Attention :

- ▷ il faut passer au sous programme les *N-1* premières dimensions telles qu'elles sont déclarées dans le programme appelant et pas seulement des «dimensions utiles». Il n'y a pas d'allocation dynamique en `Fortran 77` et une pratique courante consiste à déclarer une taille maximale pour les tableaux et donc à n'en utiliser qu'une partie. Cela ne pose aucun problème pour les tableaux à une dimension mais risque de causer des ennuis avec des tableaux à plusieurs dimensions.

1. Il est indispensable de passer la première dimension du tableau :

```
      subroutine f_tab_5  (n, tx)
      implicit none
c ... le tableau et sa premiere dimension: oui
      integer  n
      real*8   tx (n,*)
      ...
      ..
      end
      programme de_test
      implicit none
      integer    dim_1,   dim_2
      parameter (dim=100, dim_2=50)
      real*8     txx (dim_1, dim_2)
      ..
      call f_tab_5   (dim_1, txx)
      ...
      end
```

2. Mieux, on peut aussi passer explicitement toutes les dimensions :

```
      subroutine f_tab_6  (n1, n2, tx)
      implicit none
c ... le tableau et ses dimensions: oui
      integer  n1, n2
      real*8   tx (n1, n2)
      ...
      end
      programme de_test
      implicit none
      integer    dim_1,   dim_2
      parameter (dim=100, dim_2=50)
      real*8     txx (dim_1, dim_2)
      ..
      call f_tab_6   (dim_1, dim_2, txx)
      ...
      end
```

3. Et si l'on veut aussi passer les dimensions utiles ? Prenons un exemple, un tableau est déclaré avec des tailles maximales `dim_1` et `dim_2`, mais on utilise réellement `dim_util_1` et `dim_util_2` en fonction du problème traité. Puis on appelle «naïvement» les sous programmes `f_tab_5` et `f_tab_6` des exemples précédents :

```
      programme pb_tableau
      implicit none
c ...
      integer    dim_1,      dim_2,
     &           dim_util_1, dim_util_2
c ...
      parameter (dim=5,   dim_2=3)
      real*8     h (dim_1, dim_2)
c ...
      dim_util_1=3
      dim_util_2=2
c
c ... ATTENTION ERREUR !!!!!!
c
      call f_tab_5  (dim_util_1, h)
c
c ... ATTENTION ERREUR !!!!!!
c
      call f_tab_6  (dim_util_1, dimutil_2, h)
      ...
      end
```

Bien sûr, ça ne marche pas, que se passe-t-il ? Les dimensions passées au sous programme servent à repérer les éléments dans la mémoire, si l'on passe uniquement les dimensions utiles, ce repérage ne sera pas correctement effectué :

Idéalement, le sous programme doit connaître les dimensions **déclarées** du tableau et aussi les dimensions utiles :

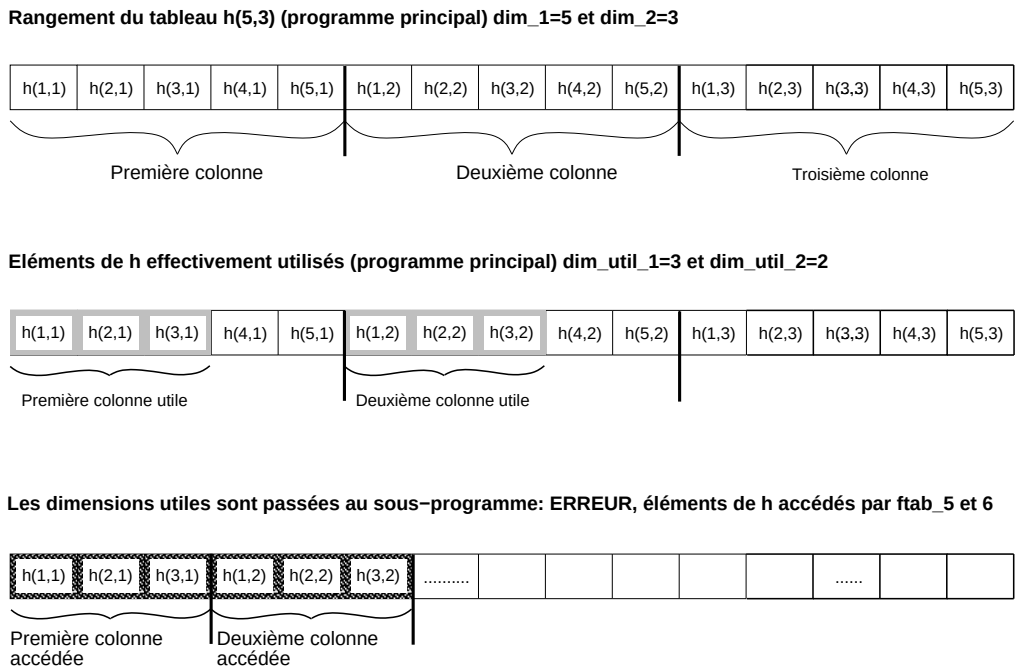

FIG. 5.2 – Tableau à deux dimensions, éléments effectivement utilisés

```
      subroutine f_tab_7
     &           (dim_1, dim_2, util_1, util_2, tx)
      implicit none
c
c ... les dimensions reelles ET
c ... les   dimensions utiles
      integer  dim_1, dim_2, util_1, util_2
c ... le dimensionnement doit utiliser les
c ... "n-1" premieres valeurs reelles
c
      real*8   tx (dim_1, util_2)
      ...
      ..
      end
```

et cette fois-ci à l'appel il n'y aura aucun problème :

```
      programme OK_tableau
      implicit none
      integer     dim_1,      dim_2,
     &            dim_util_1, dim_util_2
      parameter  (dim_1=5, dim_2=3)
      real*8   h (dim_1,   dim_2  )
      ..
      dim_util_1=3
      dim_util_2=2
      ...
      ...
c
c ... cette fois ca marche
c
      call f_tab_7
     &   (dim_1,dim_2,dim_util_1,dim_util_2, h)
      ...
      end
```

Mais en multipliant les arguments on risque de nuire à la lisibilité ce n'est pas toujours souhaitable ! D'autres techniques peuvent être utilisées pour passer les dimensions aux sous programmes :

– dimensions réelles définies dans des ordres `parameter` mis dans des fichiers à inclure ;
– dimensions utiles rangées dans des zones communes (voir plus loin).

5.4 Argument sous programme

Un nom de sous programme (`function` ou `subroutine`) peut être utilisé comme argument d'un autre sous programme. Attention, la notion de «prototype» de fonction n'existe pas en langage `Fortran` et rien n'indique, à l'extérieur d'un sous programme quelle doit être se séquence d'appel (nombre et types des d'arguments, `function` ou `subroutine`). Un exemple classique, ce sous programme qui intègre la fonction définie par le sous programme `f` :

```
      subroutine integration (x, f, a, b)
c .........  but: integrer la fonction f
c .........  entre a et b
c .........  resultat dans x
      implicit none
      real*8 x, a, b
      real*8 f
c ... ......
      do i=1, n
         x=x+ f(a+i*h)
      enddo
      end
```

Attention :

- notons qu'ici pour pouvoir utiliser aussi des fonctions prédéfinies (intrinsèques) on ne doit pas déclarer `f` comme étant *a priori* `external`;
- en revanche, le programme qui appelle ce sous programme `integration` doit **obligatoirement** déclarer le nom du sous programme passé en argument : si il s'agit d'un sous programme «utilisateur», il doit être déclaré `external`. Cette déclaration est ici **obligatoire** :

```
      programme util_integration
      implicit none
      real*8 xmin, xmax, result
c .........  fonction
      external fff
      real*8   fff
      ...
      call integration (result, fff, xmin, xmax)
      ...
      end
      real*8 function fff (x)
      implicit none
      real*8 x
      ...
      fff= ...
      end
```

si il s'agit d'une fonction `Fortran` prédéfinie, par exemple la fonction *cosinus*, elle doit être obligatoirement déclarée avec le mot-clef `intrinsic` :

```
      programme util_integration
      implicit none
      real*8 xmin, xmax, result
c .........  fonction
      intrinsic dcos
      real*8    dcos
      ...
      ...
      call integration (result, dcos, xmin, xmax)
      ...
      ...
      ...
      end
```

DANGER ! ! En `Fortran`, les fonctions prédéfinies telles que *cosinus* ont plusieurs noms selon le type des arguments ou du résultat (typiquement `acos` avec des réels simple précision, `dcos` avec des réels double précision etc...) mais aussi un nom «générique» que l'on peut utiliser quelque soit le type des arguments ici `cos`. Dans le cas d'un argument sous programme, il n'est pas permis d'utiliser le nom générique (ici par exemple, si nos «`real*8`» sont bien des « `double precision`» il faut utiliser le nom exact `dcos`). Si l'on utilise `cos` par exemple, le résultat risque d'être tout simplement faux sans erreur à la compilation ni à l'exécution...

5.5 Fonction «en ligne»

Il est possible de définir dans le texte du programme de petites fonctions, valables uniquement dans la suite de l'unité de compilation considérée (programme principal ou sous programme). Par exemple :

```
      program ex_de_fonction
      implicit none
      real*8 a, b, c, d
      ...
c ... definition d'une fonction sum2
      sum2(x,y,z)=x*x+y*y+z*z
      ...
      d = sum2 (a, b, c)
      ...
      end
```

Pratique en apparence, mais la définition étant noyée dans le reste du code tout cela n'est pas très clair d'autant plus que cette définition à l'apparence d'une affectation. Une justification possible **était** le coût trop élevé de l'appel à une «vrai» fonction. Maintenant la plupart des compilateurs proposent des options permettant de considérer certains sous programmes comme «ouverts» ou *"inline"*. À l'appel d'une fonction «ouverte» le compilateur recopie le code exécutable correspondant au lieu de générer un débranchement. Il n'y a pas de surcoût.

Conseil méthodologique :

- l'utilisation des fonction «en ligne» (aussi appelées «instructions fonctions») ne se justifie plus de nos jours !

Chapitre 6

Plus sur les Variables

Une application `Fortran` est divisée en unités de programmation ; nous avons déjà rencontré :

- le programme «principal» `program ...`;
- les sous programmes `subroutines` ou `functions`.

Jusqu'à présent l'échange d'informations entre ces unités se faisait en passant des arguments à des sous programmes. Il existe en `Fortran` un autre moyen par le biais de variables partagées.

6.1 Variables partagées : le `common`

Les zones communes (`common`) permettent à plusieurs unités de programmation `Fortran` (i.e. le programme «principal», les `subroutines`, les `functions`) de partager des données.

Cette zone porte un nom (donné entre caractères `/ /`) et contient une liste de variables («simples» ou tableaux) qu'il faut bien sûr déclarer dans l'unité considérée. Concrètement ces données ne sont plus «locales» à une unité de programmation mais stockées dans une zone spéciale repérée par le nom du `common`. Il est très important que les descriptions de cette zone soient strictement identiques dans toutes les unités où elle est impliquée et en pratique, on **doit utiliser** un fichier à inclure. Un premier exemple, très simple, plusieurs unités de programmation utilisent la variable `iii` qui doit être initialisée par un sous programme spécifique :

```
      program avec_common
      implicit none
      integer iii
      common /ok/ iii
      call init()
      ...
      call calcul(...)
      ...
      end
      subroutine init()
      implicit none
      integer iii
      common /ok/ iii
      ...
      iii=...
      end
      subroutine calcul (...)
      implicit none
      integer iii
      common /ok/ iii
      ...
      ...
      end
      ...
```

Attention :

- ▷ les anciennes versions de `Fortran` ne permettaient d'utiliser qu'une seule zone commune sans nom. Il est très important en `Fortran 77` de nommer les zones communes. Le «common blanc» (ou sans nom) fait toujours l'objet de la part des compilateurs actuels d'un traitement spécial qui peut causer bien des soucis...
- ▷ pour que deux unités de programmation partagent effectivement une zone commune, celle-ci doit aussi être déclaré «au dessus» dans l'arbre des appels. Les deux sous programmes `calc_1` et `calc_2` utilisent la zone commune `util`. Si celle-ci n'apparaît pas «au dessus» les deux zones ne sont pas *a priori* identiques. Ce qui se passe réellement dépend de la façon dont le compilateur traite ces instructions ;

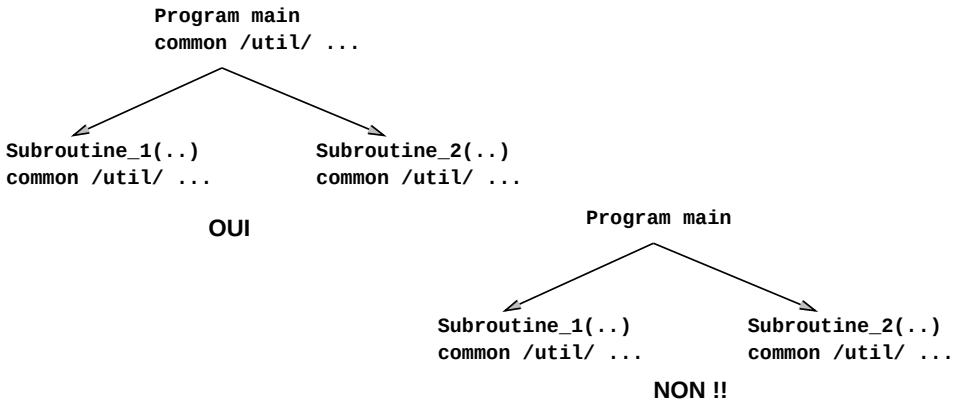

FIG. 6.1 – L'arbre des appels

- pour des raisons liées à l'optimisation de l'accès aux données dans la mémoire (problème d'alignement), si une zone commune contient des données de plusieurs types, il faut mettre les plus longues en premier i.e. d'abord les `complex*16`, `complex*8` puis `real*8`, `real*4` et enfin les `integer` et les `logical` ;

```
      ...
      complex *16 a(1000,1000),c(2000)
      real*8      x(1000),y(1000),z(1000)
      integer     nb1, nb2
c .........  donnees bien rangees
      common /data_1/ a, c, x, y, z, nb1, nb2
      ...
```

- il n'est pas possible de mélanger dans une zone commune des variable de type `character` avec des variables d'autres types ;
- si deux unités de programmation partagent une zone commune, les variables qu'elle contient ne doivent pas être passées de l'une à l'autre comme argument sous peine de provoquer de dangereux «effets de bord» (modifications non désirées de variables).

```
      program common_et_arg
      implicit none
      real*8eps
      common /param/ eps
      ...
c ... argument et commom NON  !!!!!!!!
      call cal(eps)
      ...
      end
      subroutine cal (x)
      implicit none
      real*8 x
      real*8 eps
      common /param/ eps
      ...
      ...
      end
```

Conseils méthodologiques :

- rappel, toujours nommer les zones communes ;
- on se servira des zones communes pour stocker des données dites **d'environnement**, constantes physiques, un maillage etc...ces données sont nécessaires à la plupart des unités de programmation d'une application mais elles ne sont modifiées que par un **très petit nombre** d'entre-elles, dûment identifiées, typiquement les sous programmes de lecture des données ou d'initialisations. Ainsi :
  - les arguments d'une `subroutine` sont donc ceux sur lesquels elle travaille, en les modifiant éventuellement. Les variables «en commun» ne doivent pas être modifiées sauf cas exceptionnel (sous programme d'initialisation, de lecture de données etc...) ;
  - une `function` **ne doit pas** modifier ses arguments et pour la même raison, **jamais** des variables «en commun».
- l'idéal est de placer dans un fichier à inclure, la déclaration des variables et de la zone commune. Celui-ci sera recopié dans toutes les unités de programme concernées avec une instruction `include` qui garantira que toutes les déclarations sont bien identiques. Ce qui donne, en reprenant notre premier exemple de `common` :

```
c ..........  contenu du fichier: comm_ok.h
      integer iii
      common /ok/ iii
```

```
      program include_common
      implicit none
      include 'comm_ok.h'
      ...
      call init()
      ...
      call calcul(...)
      ...
      end
      subroutine init()
      implicit none
      include 'comm_ok.h'
      ...
      iii=...
      end
      subroutine calcul (...)
      implicit none
      include 'comm_ok.h'
      ...
      ...
      end
      ...
```

6.2 (pas de) block data

Les variables situées dans des zones communes ne peuvent pas être initialisées à l'aide de l'instruction `data` (de toutes façons déconseillée). Il existe une unité de programmation spéciale appelée `block data` qui permet d'initialiser, à la compilation, ces variables. Un exemple :

```
      program avec_common
      implicit none
      integer n
      real*8  p, q
      common /ok/ n, pi, q
      ...
      ...
      ...
      end
      block data pour_ok
      common /ok/ n, pi, q
      n =1000
      pi=3.14159
      q =6.0d-3
      end
```

Remarquons que le `block data` est une unité de programmation au même titre qu'un sous programme ou le programme principal. On peut initialiser plusieurs zones communes dans le même `block data`. Notons enfin que si il n'y a qu'un seul `block data` dans un programme, il n'est même pas utile de lui donner un nom.

Conseil méthodologique :

- ne pas utiliser la construction `block data`. Des sous programme spéciaux doivent effectuer les initialisations et les lectures de données (rappellons aussi qu'il faut utiliser `parameter` pour les constantes). Certains compilateurs récents ne savent même pas traiter correctement les constructions `block data`.

6.3 Rémanence : `save`

La rémanence est la possibilité de conserver la valeur d'une variable interne à un sous programme d'un appel à l'autre de ce sous programme. Cette possibilité **n'existe pas par défaut** même si elle semble être offerte par certains compilateurs comme un effet secondaire lié à la façon dont sont traduits, en langage machine, les appels de sous programmes.

6.3.1 Dans un seul sous programme

La variable rémanente doit être déclarée avec le mot-clef `save`. Pour donner une valeur initiale à la variable on est obligé (hélas !) d'utiliser l'instruction «non-exécutable» `data` qui prend effet à la compilation donc avant le premier appel du sous programme. Un exemple classique, mais rarement utile (…), compter le nombre d'appels à un sous programme :

```
      subroutine util (..., ...)
      implicit none
c .........  arguments
      ...
c .........  variables locales
      ...
      integer compte_util
      save    compte_util
      data compte_util/0/
      ...
      ...
      compte_util=compte_util+1
      ...
      end
```

Attention :

- ▷ la déclaration `save` doit figurer **avant** l'ordre `data` ;
- ▷ ne **jamais** compter sur la «rémanence par défaut».

Conseil méthodologique :

- ► ce mécanisme ne doit être utilisé qu'exceptionnellement ! Il y a bien sûr, le cas (d'école) ci-dessus. Une autre utilisation est de pouvoir vérifier que certaine initialisations ont été correctement effectuées et de garantir qu'elles ne le seront qu'une seule fois.

6.3.2 Entre sous programmes

Deux unités de programmation partagent effectivement une zone commune si celle-ci est déclarée «au dessus» dans l'arbre des appels. Une autre possibilité est de déclarer dans les deux unités, la zone commune avec le mot-clef `save`. Comparons les deux versions :

le cas «sans `save`» : déjà vu

```
      subroutine init ()
      implicit none
c ..........  communs
      real*8  eps, sigma
      common /param/ eps, sigma
      ...
      end
      subroutine util (...)
      implicit none
c ..........  communs
      real*8  eps, sigma
      common /param/ eps, sigma
      ...
      ...
      end
      programme test
      implicit none
c ..........  communs
      real*8  eps, sigma
      common /param/ eps, sigma
      call init ()
      ...
      ...
      call util (...)
      ...
      end
```

Tout se passe bien car la zone commune aux sous programmes `init` et à `util` est aussi déclarée dans le programme principal «au dessus» ;

le cas «avec `save`» :

```
      subroutine init ()
      implicit none
c .........  communs
      real*8  eps, sigma
      common /param/ eps, sigma
      save    param
      ...
      end
      subroutine util (...)
      implicit none
c .........  communs
      real*8  eps, sigma
      common /param/ eps, sigma
      save    param
      ...
      ...
      end
      programme test_save
      implicit none
c .........  communs
      call init ()
      ...
      ...
      call util (...)
      ...
      end
```

Ça marche ! la zone commune à `util` et à `init` est déclarée avec le mot-clef `save` dans **les deux sous programmes** qui l'utilisent. Notons qu'elle n'est pas accessible depuis l'unité `test_save`.

Conseil méthodologique :

- ce mécanisme ne doit être utilisé qu'exceptionnellement ! Il faut mieux dupliquer partout la déclaration des zones communes et si possible avec un fichier à inclure. L'utilisation peut être justifiée par le fait que l'on a pas accès au code source du programme appelant.

6.4 Le super-tableau

Il s'agit de gérer la mémoire «à la main» pour pallier l'absence d'allocation dynamique. Juste un exemple, le programmeur déclare dans le programme principal, un super-tableau dans lequel il rangera tous les tableaux de `reel*8` nécessaires. Les tableaux utiles sont repérés par la position de leur premier élément dans le super-tableau :

```
      program test
      implicit none
      parameter   (max_dim=10000)
c ... <<super-tableau>>
      real*8 super(max_dim)
c ... pointeurs pour les tableaux
c ... des coordonnees x, y et z
      integer ix, iy, iz
c ... n est la taille du probleme
      integer n
      ...
c ... calcul des pointeurs dans super
      ix=1
      iy=ix+n
      iz=iy+n
c ... ...
      ...
      end
```

Si un sous programme utilise par exemple les tableaux de coordonnées `x`, `y` et `z`, il suffit de lui passer les adresses des premiers éléments. Ainsi :

```
      subroutine util (dim, x, y, z,  ...)
      implicit   none
      integer    dim
      real*8     x(dim), y(dim), z(dim)
      ...
      ...
      end
```

puis dans le programme principal :

```
     ...
     call util (n, super(ix), super(iy),
    &              super(iz), ...     )
     ...
     ...
```

Conseil méthodologique :

- pratique car il n'y a qu'un paramètre à mettre à jour lorsque l'on change la taille des problèmes visés. Pour que tout cela reste clair, il est préférable que le super-tableau n'apparaîsse que dans le programme principal ;
- ne **jamais mélanger** les types et donc utiliser un super-tableau par type de données (entier, réel(s), et complexe(s)).

6.5 Synonymes : (pas d') `equivalence`

La déclaration `equivalence` permet d'exprimer que plusieurs variables d'une unité de programmation vont partager le même espace mémoire. Cela peut faire penser, en plus rudimentaire aux `union` du langage **C**.

Ces données peuvent être de type quelconque et il est bien évident que seule la dernière variable affectée, possède une valeur cohérente.

Rappelons nous que les noms de variable, en `Fortran`, correspondant de fait à des adresses en mémoires. Ainsi :

- si le nom d'un tableau apparait dans une déclaration `equivalence`, tout le tableau est concerné ;
- si un élément de tableau apparait dans une déclaration `equivalence`, le tableau est concerné **à partir** de cet élément. Dans le cas de tableaux multi-dimensionnels, il faut veiller à prendre en compte le rangement des éléments en mémoire tel qu'il est défini par la norme.

Enfin si les variables sont de tailles différentes, l'«équivalence» est effective, bien sûr, sur la partie commune. Tout cela n'est pas très sain...donnons néanmoins quelques exemples :

- une variable, un élément de tableau :

```
      ...
      real*8 x_0, x, y, z, val_1
      real*8 val(100)
      equivalence (x_0, val(1))
c ...
c ...    x_0 <=> val(1)
      ...
```

– un tableau, une «colonne» d'un autre :

```
      ...
      real*8 a(100,100), v(100)
      equivalence (v, a(1,10))
c ...
c ...    v(i) <=> a(i,10)
      ...
```

L'on peut avoir plus de deux variables en «équivalence» et l'on peut aussi regrouper les déclarations en les séparant par des virgules :

```
      ...
      real*8 x_0, x, y, z. val_1
      real*8 val(100), a(100,100), v(100)
      equivalence ((x_0, val_1, val(1)), (v, a(1,10)))
c ... deux equivalences:
c ...    x_0  <=> val_1 <=> val(1)
c ...    v(i) <=> a(i,10)
      ...
```

Conseil méthodologique :

- ne **jamais** utiliser l'instruction `equivalence` source d'innombrables erreurs (de logique ou plus simplement d'inattention) ;
- de plus l'instruction `equivalence` inhibe généralement les optimisations effectuées normalement par le compilateurs : on ne peut plus garantir que deux variables occupent en mémoire des emplacements distincts.

Remarque :

⇒ cette instruction était souvent utilisée pour réaliser des système de gestion de la mémoire palliant dans une certaine mesure l'absence d'allocation dynamique en `Fortran`. Un tel système gére nous l'avons vu des «pointeurs» dans un «super-tableau». On peut déclarer un tableau pour chaque type de données et les mettre en équivalence pour ne gérer qu'une seule zone. Tout cela revient à faire *à la main* le travail normalement dévolu à un compilateur. . ..

Chapitre 7

Les entrées/Sorties

Les ordres de lecture et d'écriture sont des instructions du langage dont la description fait partie intégrante de la norme `Fortran 77`..

7.1 Unités logiques, physiques

Un programme `Fortran` lit et écrit sur des *unités logiques* repérées par des numéros appelés donc *numéros logiques*. Du point de vue de la machine, ces numéros logiques correspondent aux *unités physiques* sur lesquelles sont effectivement réalisées les opérations d'entrées-sorties. Il peut s'agir du clavier, de l'écran, d'une imprimante ou d'un fichier repéré par son nom et son chemin d'accès dans une arborescence (typiquement dans un système de fichier `UNIX`).

À l'origine de nombreux numéros étaient attribués par défaut à divers dispositifs matériels tels que le *lecteur de cartes perforées* (l'unité `5`), l'*imprimante* (l' unité `6`), le *perforateur de cartes*, la *trieuse de cartes*, etc. . .De tout cela il est resté que :

- l'unité `5` correspond à l'entrée «standard» (au sens `UNIX`) du programme en cours d'exécution ;
- l'unité `6` correspond à la sortie «standard» (au sens `UNIX`) du programme en cours d'exécution.

Ainsi, si l'on se doute bien que les fichiers peuvent (et doivent) être *ouverts* puis *fermés* comme cela se fait dans tous les langages, il est possible de lire sur l'unité logique `5` et d'écrire sur l'unité logique `6` sans autre(s) précision(s).

7.2 Écran-clavier

Nous allons utiliser dans un premier temps le clavier et l'écran. C'est le minimum pour communiquer et cela permet d'introduire simplement les instructions

`read` et `write`.

7.2.1 l'instruction `read`

Dans l'exemple suivant, on lit au clavier (entrée «standard») un entier puis deux réels :

```
...
integer n
real*8  x, y
...
read (5,*) n
...
read (5,*) x, y
...
...
```

Cette forme minimale d' instruction `read` possède deux arguments suivis d'une liste d'identificateurs :

- le premier est le numéro logique de l'unité sur laquelle s'effectue la lecture ;
- le second est un descripteur du *flot d'entrée* : le *format*. Le plus souvent, comme ici, nous utiliserons `*` qui signifie que l'on ne s'intéresse pas à la forme exacte sous laquelle vont arriver les données désirées (y a t-il des blancs ? combien de chiffres ? y a t-il un exposant ? -cas des réels-, etc...).

Comment cela fonctionne-t-il ? Chaque instruction `read` lit une suite de caractères dans ce que l'on appelle un *enregistrement*. Si l'on lit au clavier, il s'agit de la suite des caractères tapés avant d'appuyer sur la touche `<entrée>`. Dans l'exemple ci-dessus, il faut taper un entier, appuyer sur la touche `<entrée>`, taper les deux réels et appuyer de nouveau sur la touche `<entrée>`.

D'autres arguments (optionnels) peuvent être donnés sous forme de mots-clef, nous verrons cela par la suite mais nous pouvons déjà utiliser ici une forme «par mots-clef» de cette instruction. Il est nécessaire de «renseigner» au moins les deux mots-clef (donnés dans un ordre quelconque) `unit` l'unité logique et `fmt` le «format». Nous verrons qu'il peut y avoir d'autres mots-clef.

```
...
integer n
real*8  x, y
...
read (unit=5,fmt=*) n
...
read (unit=5,fmt=*) x, y
...
```

Remarques :

⇒ la description «fine» des données à lire peut devenir nécessaire si l'on veut lire par exemple des chaînes de caractères contenant des blancs. Le comportement «normal» étant de retenir à partir du premier caractère non blanc, une suite de caractères non blancs de longueur au moins égale à la taille déclarée de la chaîne de caractère ;

⇒ pour lire des valeurs complexes, il faut taper au clavier deux réels (les parties réelle et imaginaire) entre parenthèses et séparées par une virgule ;

⇒ les variables de type logique peuvent être entrées soit sous forme de constantes logiques `.true.` (ou `true`) et `.false.` (ou `false`) soit sous forme «condensée» `T` (ou `t`) pour *"true"* et `F` (ou `f`) *"false"* ;

Attention :

▷ tout cela ne fonctionne pas toujours aussi bien que l'on pourrait l'espérer ! Cela dépend des compilateurs, certains cherchent semble-t-il à interpréter les entrées en ignorant *a priori* le type des données ce qui provoque des erreurs avec les chaînes de caractères. Que faire ? et bien `Fortran` n'est pas un langage fait pour manipuler des chaînes de caractères !

Conseil méthodologique :

► utiliser de préférence la forme sans mots-clef : `read(5,*) ...`, plus «usuelle».

7.2.2 L'instruction `write`

Elle est «symétrique» de l'instruction `read`. Dans l'exemple suivant, on écrit à l'écran (sortie «standard») l'entier et les deux réels lus au paragraphe précédent :

```
...
integer n
real*8  x, y
...
write (6,*) n
...
write (6,*) x, y
...
...
```

Comme pour la lecture, cette forme minimale d' instruction `write` possède deux arguments suivi d'une liste d'identificateurs ou de constantes :

- le premier est le numéro logique de l'unité sur laquelle s'effectue l'écriture ;
- le second est un descripteur du *flot de sortie* : le *format*. Le plus souvent, comme ici, nous utiliserons `*` qui signifie que l'on ne s'intéresse pas à la forme exacte sous laquelle vont être écrite les valeurs mais que l'on laisse toute latitude au système pour imprimer le maximum d'informations (i.e. gérer le nombre de chiffres significatifs etc...).

Chaque instruction `write` écrit une suite de caractère (un *enregistrement*) c'est à dire, ici, une ligne sur l'écran.

Là aussi, d'autres arguments (optionnels) peuvent être donnés sous forme de mots-clef et il existe comme pour l'instruction `read` une forme «par mots-clef». Il est obligatoire de «renseigner» au moins les deux mots-clef (donnés dans un ordre quelconque) `unit` l'unité logique et `fmt` le «format».

```
...
integer n
real*8  x, y
...
write (unit=6,fmt=*) n
...
write (unit=6,fmt=*) x, y
...
```

Remarques :

⇒ si l'on écrit une variable chaîne de caractère, c'est la longueur «utile» de la chaîne qui est utilisée (i.e. ce qui y est stocké) ;

⇒ il est facile, pour améliorer la présentation d'écrire des constantes chaînes de caractères :

```
...
integer n
real*8  x, y
...
...
write (6,*) 'Voici un entier: ', n
write (6,*) 'un reel:', x,' et un autre:' y
write (6,*) 'c'' est tout! au revoir'
write(6,*)  ' '
...
```

⇒ les variables de type complexes sont écrites entre parenthèses, parties réelle et imaginaire séparés par une virgule ;
⇒ Les variables de type logique sont écrites le plus souvent sous la forme `T` et `F` pour (*"true"* et *"false"*) ;
⇒ normallement des instructions `read (..,*)` peuvent relire ce qui est écrit avec `write (..,*)` mais on peut avoir des ennuis avec des chaînes de caractères contenant des blancs.

Conseil méthodologique :

► utiliser de préférence la forme sans mots-clef : `write (6,*) ...`, plus «usuelle».

7.2.3 Redirections

Il est possible (notamment sur les systèmes `Unix`) de «rediriger» l'entrée standard et la sortie standard d'un programme. Cela permet d'utiliser des fichiers sans lire la suite...Si `fpgm` est le nom d'un exécutable, on peut taper les commandes suivantes :

données lues au clavier ; sorties à l'écran :

==> `fpgm`

données lues dans le fichier `fpgm.don` ; sorties à l'écran :

==> `fpgm < fpgm.don`

données lues dans le fichier `fpgm.don` ; sorties dans le fichier `fpgm.res` (attention : écrase le contenu de `fpgm.res` !) :

$==>$ `fpgm < fpgm.don > fpgm.res`

données lues dans le fichier `fpgm.don` ; sorties dans le fichier `fpgm.res` (à la suite de ce qui existe déjà dans `fpgm.res`) :

$==>$ `fpgm < fpgm.don >> fpgm.res`

Remarque :
$\Rightarrow$ dans les deux derniers cas le fichier `fpgm.res` est crée si il n'existe pas.

7.2.4 Des variantes inutiles

On peut utiliser l'instruction `read` sans indiquer d'unité logique. Il s'agit alors d'une lecture sur l'entrée standard :

```
c ...........  NON
      read *, x, y, n
      ...
```

cette forme est à éviter, il faut mieux utiliser :

```
c ...........  mieux !!
      read (5,*)  x, y, n
      ...
```

Il existe enfin l'instruction `print` pour effectuer une écriture sur la sortie standard :

```
c ...........  NON
      print *, x, y, n
      ...
```

cette forme est à éviter, il faut mieux utiliser :

```
c ...........  mieux !!
      write (6,*)  x, y, n
      ...
```

Conseil méthodologique :

► ne **pas** utiliser les formes inutiles `read *, ...` et `print *, ...`.

7.3 Entrées-Sorties sur fichiers

Pour utiliser un fichier en `Fortran`, il suffit de le «relier» à une unité logique, c'est à dire un numéro. Cette opération s'appelle «ouvrir» le fichier, elle est réalisée par un appel à l'instruction `open`. Après usage, le fichier doit être «fermé» par un appel à l'instruction `close`. Rappelons qu'il n'y a pas lieu d'«ouvrir» et de «fermer» les unités `5` et `6`, gérées par l'environnement dans lequel s'exécute le programme.

7.3.1 Organisation et contenu des fichiers

Nous distinguerons, en `Fortran` deux organisations possibles pour les fichiers selon la façon dont on peut accéder aux informations qui y sont contenues. Si le fichier est «séquentiel» les enregistrements sont écrits les uns à la suite des autres. La lecture commence par le début et pour accéder à la n^{e}information, il faut avoir lu les `n-1` premières. En revanche, dans un fichier en «accès direct», un «pointeur mobile» permet d'accéder aux informations dans un ordre quelconque. On peut considérer un fichier en accès direct comme un tableau d'enregistrements.

Le contenu d'un fichier, lui peut être de plusieurs nature. Un fichier «formatté» contient les valeurs des données sous forme de chaînes de caractères, c'est l'image de ce que l'on tape au clavier ou de ce que l'on lit à l'écran. Un fichier «non formatté» lui contient les valeurs des données sous forme «binaire» c'est à dire l'image de leurs représentations dans la mémoire de l'ordinateur. Il va de soi qu'un fichier «formatté» est «portable» c'est à dire lisible d'une machine à l'autre tandis qu'un fichier «non-formatté» ne peut être relu que sur une machine où la représentatioon des nombres et des caractères en mémoire est compatible.

L'intérêt du fichier «binaire» est bien sûr le gain de place (un réel codé en mémoire occupe 4 ou 8 octets, sa représentation «lisible» au moins autant d'octets qu'il faut de caractères pour l'écrire sans compter les blancs etc...) et la rapidité (pas de conversion/déconversion lors des entrées sorties).

7.3.2 L'instruction `open`

Comme nous l'avons vu cette instruction associe un fichier -unité physique- à une unité logique repérée par un numéro. Le seul argument obligatoire est le numéro codé seul ou à l'aide du mot-clef `unit` comme dans les instructions `read` ou `write` :

```
      ...
      open (2, ...)
      ...
```

ou

```
      ...
c ..........  utiliser plutot
      open (unit=3, ...)
      ...
```

Viennent ensuite un grand nombre de paramètres, sous forme de mots-clef à renseigner qui pour être optionnels n'en sont pas moins nécessaires. Nous allons les examiner en détail puis donner quelques exemples.

Le nom du fichier : se code à l'aide du mot-clef `file`, la valeur est une chaîne de caractères. Elle doit contenir un nom de fichier. Dans l'environnement `Unix`, le fichier est recherché dans ou à partir du répertoire courant si le nom ne contient pas un chemin d'accès «absolu» :

```
      ...
c ..........  dans le repertoire "courant"
      open (unit=3, file='fpgm.don', ...)
```

```
      ...
c ..........  a partir du  repertoire "courant"
      open (unit=3, file='data/fpgm.don', ...)
```

```
      ...
c ..........  chemin d'acces  "absolu"
      open (unit=3,
     &      file='/home/anfray/data/fpgm.don',...)
```

Ce paramètre n'est pas obligatoire ! mais il est en fait indispensable de le coder (sauf comme nous le verrons dans le cas de fichiers temporaires où il possède donc une valeur par défaut qui dépend du système utilisé).

Le «statut» du fichier : le mot-clef `status` permet de préciser si le fichier existe déjà, doit être crée, etc.... Il peut prendre l'une des valeurs suivantes qui sont des constantes de type chaîne de caractère :
- `status='old'`, le fichier existe déjà ;
- `status='new'`, le fichier n'existe pas et doit être crée au moment de l'ouverture ;
- `status='unknown'`, le programmeur ne sait pas et ...la norme ne fixe pas non plus le comportement dans ce cas. Le plus standard est que le fichier est crée si il n'existe pas.
- `status='scratch'` cette valeur est réservée à des fichiers **temporaires**, qui ne survivront pas à l'exécution du programme.

Ce paramètre possède une valeur par défaut qui est `'unknown'`.

En cas d'erreur : l'«ouverture» du fichier peut échouer pour diverses raisons (il n'existe pas, ne peut être créé, etc...) et il est préférable d'agir en conséquence ! Pour cela deux mots-clef peuvent être utiles :
- `iostat` a pour valeur le nom d'une variable entière qui contiendra, après exécution de l'instruction, un code d'erreur : zéro, bien sûr si tout c'est bien passé, une autre valeur qui dépend de l'environnement d'exécution si l'ouverture a échouée ;
- `err` a pour valeur une étiquette qui référence l'instruction du programme à exécuter en cas d'erreur. Il s'agit d'un débranchement inconditionnel déguisé et son utilisation n'est pas recommandée dans des programmes bien structurés !.

```
      ...
      integer io_erreur
      ...
      open (unit=3, file='fpgm.don',
     &      iostat=io_err, ...)
      if (io_err.ne.0) then
c .........  echec !!
         ...
      else
c .........  tout va bien
c .........  on peut continuer
         ...
         ...
         ...
      endif
      ...
```

L'organisation et le contenu du fichier : l'organisation du fichier est précisée avec le mot-clef `access` qui peut prendre les deux valeurs (de type chaîne de caractère) :

– `access='sequential'`, pour un fichier séquentiel ;
– `access='direct'`, pour un fichier en accès direct.

Ce paramètre possède une valeur par défaut qui est `'sequential'`, le fichier «classique».

Le contenu du fichier est précisée avec le mot-clef `form` qui peut prendre les deux valeurs (de type chaîne de caractère) :

– `form='formatted'`, pour un fichier «formatté» ;
– `form='unformatted'`, pour un fichier «non-formatté».

La valeur par défaut est `'formatted'`, si `access='sequential'` (le fichier «classique») et `'unformatted'`, si `access='direct'`

Un fichier contient des «enregistrements» qui correspondent à ce qui est écrit ou lu lors d'un appel à une instruction `write` ou `read`. Il est possible de préciser la taille maximale de ces enregistrements et si cela est inutile pour des fichiers «séquentiels» c'est obligatoire dans le cas des fichiers en «accès direct». On utilise alors le mot-clef `recl` (pour *"record length"*), par exemple :

– `recl=1000`

Pour les fichiers «formattés» l'unité est le caractère, pour les fichiers «formattés» la valeur doit être donnée en «mots» mémoire (c'est à dire selon les machines 4 ou 8 octets). Ce paramètre ne possède pas de valeur par défaut.

Enfin un dernier mot-clef permet de préciser l'interprétation des caractères blanc éventuellement présent dans une constante numérique (tout cela est exotique et rarement utile).

Si `blank='null'`, ils sont ignorés et si `blank='zero'`, ils sont considérés comme des zéros. Ce paramètre possède une valeur par défaut que la norme ne fixe pas...

Conseils méthodologiques :

- les unités «stantards» (`5`, et `6`) sont gérées par le système et il faut mieux éviter d'effectuer des `open` dessus.

Récapitulation :

mot-clef	valeurs possibles	...par défaut	commentaire(s)
`unit`	entier positif		obligatoire
`file`	ch. de caractères	?	systématiquement
`status`	`'old'` `'new'` ou `'unknown'`	`'unknown'`	à préciser de préférence
`iostat`	nom de variable		code d'erreur
`err`	etiquette		NON ! utiliser `iostat`
`access`	`'sequential'` ou `'direct'`	`'sequential'`	utiliser seulement si `'direct'`
`form`	`'formatted'` ou `'unformatted'`	-`'formatted'` si `'sequential'` -`'unformatted'` si `'direct'`	n'utiliser que si le défaut ne convient pas (rare ! !)
`recl`	entier positif		obligatoire si `'direct'`
`blank`	`'null'` ou `'zero'`	?	à oublier

7.3.3 L'instruction `close`

Pour être sûr que tout se passe correctement, un fichier doit être fermé après usage. Pourquoi ? les dispositifs d'entrées-sorties installés sur un calculateur peuvent utiliser des mémoires tampon ("*buffers*"). Une requête d'écriture sera alors réalisée (rapidement) sur la mémoire tampon en attendant d'être effectuée définitivement sur le disque ou à l'écran. L'instruction `close` dans ce cas garantiera que toutes les écritures sont effectives.

Cette instruction agit sur une unité logique et le seul argument obligatoire est le numéro codé seul ou à l'aide du mot-clef `unit` comme dans l'instruction `open` :

```
      ...
      close (2, ...)
      ...
```

ou

```
      ...
c .........  utiliser de preference
      close (unit=3, ...)
      ...
```

Comme pour l'ouverture, viennent ensuite des paramètres optionnels sous forme de mots-clef à renseigner.

Le statut du fichier : au moment de l'ouverture, on peut se demander si le fichier existe déjà ; la question ici est que faire du fichier : le garder ou le détruire. Le mot-clef `status` peut donc prendre deux valeurs (de type chaîne de caractères) :

- `status='keep'`, le fichier est conservé ;
- `status='delete'`, le fichier est détruit après usage !

Ce paramètre possède une valeur par défaut qui est `'keep'` sauf si le fichier a été ouvert avec le statut `scratch`. Le défaut est alors la seule valeur possible dans ce cas c'est à dire `'delete'`.

En cas d'erreur : nous retrouvons ici les paramètres de l'instruction `open` avec les mêmes significations. La «fermeture» du fichier peut échouer elle aussi : (il ne peut être conservé ou détruit, etc...) et de même il est préférable d'agir en conséquence avant la fin du programme. Comme au paragraphe précédent :

- `iostat` a pour valeur le nom d'une variable entière qui contiendra, après exécution de l'instruction, un code d'erreur : zéro, bien sûr si tout c'est bien passé ;
- `err` a pour valeur une étiquette qui référence l'instruction du programme à exécuter en cas d'erreur. Son utilisation n'est pas recommandée.

```
      ...
      integer io_erreur
      ...
      close (unit=3,  iostat=io_erreur)
      if (io_erreur.eq.0) then
c .........  tout va bien
         ...
         ...
      else
c .........  echec !!
         ...
         ...
      endif
      ...
```

Conseil méthodologique :

► toujours refermer un fichier après usage.

Récapitulation :

mot-clef	**valeurs possibles**	**…par défaut**	**commentaire(s)**
`unit`	entier positif		obligatoire
`status`	`'keep' 'delete'`	-`'keep'` si `'old'`, `'new'` ou `'unknown'` -`'delete'` si `'scratch'`	à préciser de préférence
`iostat`	nom de variable		code d'erreur
`err`	etiquette		NON ! utiliser `iostat`

7.4 Informations, l'instruction `inquire`

L'instruction `inquire` permet dans un programme `Fortran` d'obtenir des informations :

– sur une «unité physique», donc un fichier, existe-t-il ?, quel est son «format», est-il relié à une unité logique ouverte etc. . . ;
– sur une «unité logique», est-elle fermée, ouverte et dans ce cas quel type de fichier y est connecté etc. . ..

Le premier paramètre sera donc **soit** une unité logique **soit** un fichier. Les autres paramètres possibles se donnent sous forme de mots-clef auquels on fait correspondre des variables qui contiendront les informations désirées. On distinguera trois «familles» d'arguments. Ceux dont la valeur de retour à toujours un sens, ceux qui n'ont de sens que si le fichier ou l'unité existe ou est ouvert et enfin ceux qui ne concernent que certaine organisations de fichiers. Les deux requêtes (unité logique ou fichier) sont similaires :

`inquire` pour un fichier : le premier paramètre obligatoire se code à l'aide du motclef `file`, la valeur est une chaîne de caractères qui contient le nom de fichier. Comme pour l'instruction `open`, le nom peut contenir un chemin d'accès relatif ou absolu :

```
      ...
c ..... dans le repertoire "courant"
      inquire (file='fpgm.don1', ...)
c ..... a partir du  repertoire "courant"
      inquire (file='data/fpgm.don2', ...)
c ..... chemin d'acces  "absolu"
      inquire (file='/home/anfray/fpgm.don3', ...)
```

`inquire` pour une unité logique : le premier paramètre obligatoire se code alors à l'aide du motclef (optionnel) `unit`.

```
      ...
      inquire (3, ...)
```

```
      ...
c ..... utiliser de preference
      inquire (unit=3, ...)
```

En cas d'erreur : nous retrouvons ici les même paramètres que dans `open` ou `close` :

- `iostat` a pour valeur le nom d'une variable entière qui contiendra, après exécution de l'instruction, un code d'erreur : zéro, bien sûr si tout c'est bien passé ;
- `err` a pour valeur une étiquette qui référence l'instruction du programme à exécuter en cas d'erreur. Son utilisation n'est pas recommandée.

Le fichier existe-t-il ? : le mot-clef `exist` associé à une variable de type `logical` permet de savoir si un fichier existe :
- `exist=fic_ok` (avec la déclaration `logical fic_ok`) si `fic_ok` vaut `.true.` le fichier existe sinon `fic_ok` vaut `.false.`.

L'organisation et le contenu du fichier : **si le fichier existe**, il est possible d'accéder aux informations nécessaires pour l'ouvrir, les valeurs que l'on récupère alors peuvent servir à l'instruction `open`. L'organisation du fichier peut être connue à l'aide des deux mots-clef `sequential` et `direct` (qui renvoient au mot-clef `access` de l'instruction `open`). Si le fichier n'existe pas, les valeurs retournées sont sans significations :
- `sequential=fic_seq` avec la déclaration `character*7 fic_seq`, le fichier a-t-il une organisation séquentielle les valeurs possibles sont `'yes'`, `'no'` ou `'unknown'` ;
- `direct=fic_dir` (avec la déclaration `character*7 fic_seq`) est-ce un fichier en accès direct les valeurs possibles sont `'yes'`, `'no'` ou `'unknown'`.

La nature du contenu du fichier peut être connue à l'aide des deux mots-clef `formatted` et `unformatted` (qui renvoient au mot-clef `form` de l'instruction `open`) :
- `formatted=fic_form`, avec la déclaration `character*7 fic_form`, le fichier est-il «formatté» les valeurs possibles sont `'yes'`, `'no'` ou `'unknown'` ;
- `unformatted=fic_unform` (avec `character*7 fic_form`), le fichier est-il «non-formatté» les valeurs possibles sont `'yes'`, `'no'` ou `'unknown'`.

Le fichier ou l'unité logique est-il ouverte ? : le mot-clef `opened` associé à une variable de type `logical` permet de savoir si une unité logique est ouverte ou si un fichier est relié à une unité ouverte :
- `opened=ouvert` (avec la déclaration `logical ouvert`) les réponses possibles sont `.true.` ou `.false.`.

À quelle unité logique est relié un fichier ? si le fichier est ouvert, i.e. relié à une unité logique on peut alors utiliser :
- `number=fic_num` (avec la déclaration `integer fic_num`), qui retourne le numéro de l'unité logique reliée au fichier par une instruction `open` ;

À quel fichier est reliée une unité logique ? si l'unité est ouverte, elle peut être reliée à un fichier dont on a précisé le nom (ce n'est pas obligatoire : cas d'un fichier temporaire ! !) ;
- `name=nom_fic` avec par exemple `character*80 nom_fic` (il faut prévoir assez de place), qui retourne le nom du fichier relié à l'unité logique par une instruction `open` ;

Comment l'unité logique a-t-elle été ouverte ? si une unité logique est ouverte, il est possible de connaître les valeurs de des arguments passés à l'instruction `open` (ou les valeurs par défaut si ils ont été omis) :
- `form=unit_cont` (avec la déclaration `character*11 unit_cont`), permet de connaître la nature de ce qui sera lu ou écrit, les valeurs possibles sont `'formatted'` ou `'unformatted'`.
- `form=unit_org` (avec la déclaration `character*10 unit_org`), permet de connaître l'organisation du fichier associé, Les valeurs possibles sont `'sequential'` ou `'direct'`.

Si l'unité a été ouverte en mode «formatté» , on peut s'intéresser à l'interprétation des caractères blanc éventuellement présent dans une constante numérique (exotique et rarement utile) :
- `blank=unit_bl` (avec la déclaration `character*4 unit_bl`), la valeur `blank='null'` indique qu'ils seront ignorés en revanche, la valeur `blank='zero'` indique qu'ils seront considérés comme des zéros.

Si l'unité a été ouverte en accès direct , le fichier correspondant est constitué d' «enregistrements» dont la taille maximale a été précisée lors de l'ouverture :
- `recl=unit_rec` (avec la déclaration `integer unit_rec`), retourne la taille maximale des enregistrements ;
- `nextrec= i` (`integer i`) retourne la position du «pointeur mobile» dans le fichier c'est à dire la valeur `1` si l'on a encore rien fait ou le numéro du dernier enregistrement accédé plus `1` sinon.

7.5 `read` et `write` : entrées/Sorties «formattées»

Dans la terminologie `Fortran`, un fichier «formatté» contient des informations lisibles, on peut le consulter à l'écran, l'éditer, l'imprimer. Comme pour l'écran-clavier ci dessus, les instructions lire et écrire contiennent la description des enregistrements. Chaque enregistrement correspondant à une ligne du fichier.

7.5.1 Lectures formattées

Examinons d'abord la forme générale de l'instruction `read` décrite plus haut. Le premier argument est l'unité logique sur laquelle on effectue la lecture, le

deuxième est un descripteur, le «format» de l'enregistrement qui sera lu.

L' unité logique : le premier paramètre obligatoire se code éventuellement à l'aide du motclef `unit`.

```
      ...
      read (unit=2, ...)
```

```
      ...
c ..........  le plus frequent
      read (2, ...)
```

Le format : le deuxième paramètre correspond au «format», décrivant l'enregistrement qui sera écrit. Il peut prendre plusieurs formes, le mot-clef `fmt` est optionnel. On peut rencontrer :

– une étoile, le système se débrouille, c'est le plus simple!!!,

```
c ..........  le plus courant
      read (2, *) x, y, z
```

```
c ...........  ou mieux
      read (2, fmt=*) x, y, z
```

– un entier i.e. une étiquette renvoyant à une déclaration `format` :

```
      read   (2, 1000) x, y, z
 1000 format (....)
```

```
c ...........  ou mieux
      read (2, fmt=1000) x, y, z
```

– une chaîne de caractère qui contient le format :

```
      read   (2, '(...)') x, y, z
```

```
c ...........  ou mieux
      read (2, fmt='(...)') x, y, z
```

En cas d'erreur : nous retrouvons ici les paramètres des instructions `open` et `close` avec la même signification et un nouveau venu pour gérer la fin de fichier :

- `iostat` a pour valeur le nom d'une variable entière qui contiendra, après exécution de l'instruction, un code d'erreur : zéro, bien sûr si tout c'est bien passé, sinon une autre valeur qui dépend de l'environnement d'exécution. Cette valeur sera **négative** si l'on essaie de lire après la fin du fichier ;
- `err` a pour valeur une étiquette qui référence l'instruction du programme à exécuter en cas d'erreur. Son utilisation n'est pas recommandée.
- `end` a pour valeur une étiquette qui référence l'instruction du programme à exécuter si l'on essaie de lire après la fin du fichier. Il s'agit encore une fois d'un `goto` déguisé et son utilisation n'est pas recommandée d'autant plus que `iostat` permet d'avoir cette information.

Conseil méthodologique :

- ► utiliser seulement `iostat` et non `err` ou `end` qui sont des débranchements déguisés !

7.5.2 Écritures formattées

C'est la forme générale de l'instruction `write` décrite plus haut. Le premier argument est l'unité logique sur laquelle on effectue l'écriture, le deuxième est un descripteur le «format» de l'enregistrement qui sera lu.

L' unité logique : le premier paramètre obligatoire se code éventuellement à l'aide du motclef `unit`.

```
      ...
      write (unit=10, ...)
```

```
      ...
c ...........  le plus frequent
      write (10, ...)
```

le format : le deuxième paramètre correspond au «format», décrivant l'enregistrement qui sera écrit. Il peut prendre plusieurs formes, le mot-clef `fmt` est optionnel. On peut rencontrer, comme pour la lecture :

– une étoile, le système se débrouille, c'est le plus simple!!!,

```
c ..........  le plus courant
      write (10, *) x, y, z
```

```
c ...........  ou mieux
      write (10, fmt=*) x, y, z
```

– un entier i.e. une étiquette renvoyant à une déclaration `format` :

```
      write (10, 1000) x, y, z
 1000 format (....)
```

```
c ...........  ou mieux
      write (10, fmt=1000) x, y, z
```

– une chaîne de caractère qui contient le format :

```
      write (10, '(...)') x, y, z
```

```
c ...........  ou mieux
      write (10, fmt='(...)') x, y, z
```

En cas d'erreur : nous retrouvons ici les paramètres des instructions `open` et `close` avec la même signification :

– `iostat` a pour valeur le nom d'une variable entière qui contiendra, après exécution de l'instruction, un code d'erreur : zéro, bien sûr si tout c'est bien passé ;

– `err` a pour valeur une étiquette qui référence l'instruction du programme à exécuter en cas d'erreur. Son utilisation n'est pas recommandée.

Conseil méthodologique :

► utiliser seulement `iostat` et non `err` qui est débranchement déguisé !

7.5.3 Les formats

Les formats servent à décrire de façon précise les enregistrements qui seront lus ou écrits lors d'entrées-sorties «formattées». Complexes et causant plutôt beaucoup de soucis il est préférable d'essayer de s'en passer. Nous donnerons juste un exemple, tous les manuels `Fortran` étant extrêmement prolixes à ce sujet. Le format contient un certain nombre de descripteur séparés par des virgules, qui indiquent la taille et le contenu des champs de l'enregistrement. On peut utiliser l'instruction `format`, il est nécessaire de la munir d'une étiquette permettant, par la suite d'y faire référence. Le format de l'exemple ci dessous contient :

- un champ chaîne de caractères (ouf ! la longueur est quelconque), descripteur `a` ;
- un champ entier, descripteur `i`, de taille 2 ;
- un champ chaîne de caractères ;
- un champ réel «flottant», descripteur `f` (un des descripteurs possibles pour les réels), de taille 8 avec 2 positions après la virgule ;
- un champ chaîne de caractères de taille 5.

```
     ...
1001 format (a,i2,a,f8.2,a5)
```

On peut maintenamt l'utiliser pour écrire, par exemple :

```
     integer ii
     real*8 x
     ...
     write (6, 10001) 'je tente ma chance ii=', ii,
    &                 ' et x=', x , ' ....'
     ...
```

Tout se passe bien si les valeurs des variables `ii` et `x` «rentrent» bien dans les «trous» qui leurs sont assignés sinon on récupère généralement des étoiles (exemple `ii` vaut 456 et on veut l'écrire dans un champ de taille 2) :

```
  je tente ma chance ii=** et x******** ....
```

Catastrophe surtout si les valeurs sont les résultats d'un long calcul. Plus difficile à déceler, si le format pour les réels est mal choisi on peut perdre de l'information : ne pas afficher suffisament de chiffres significatifs, etc…

Conseil méthodologique :

- ► obsolète et dangereux. À éviter à moins d'être fanatique d'impressions bien alignées. Utiliser systématiquement le «format `*`».
- ► si l'on ne peut pas s'en passer, on peut aussi «doubler» subtilement certaines impressions dans un fichier qui fera foi.

Attention :

- ▷ avec le «format `*`», l'**aspect** des sorties peut varier d'un système à l'autre. Il n'est pas possible de comparer deux résultats d'un même programme avec la fonction `Unix diff` qui teste l'égalité de deux fichier au niveau binaire.

7.6 `read` et `write` : entrées/Sorties «binaires»

Un fichier «binaire» contient l'image de la représentation en mémoire des variables. On ne peut donc le consulter à l'écran, l'éditer ou l'imprimer. Il est possible de le relire uniquement sur des systèmes où la représentation des données en mémoire est identique. Chaque instruction lire ou écrire traite comme précédement un enregistrement mais il n'y a pas de «format».

7.6.1 Lectures non-formattées (binaires)

C'est la même instruction `read` mais sans le paramètre format. On peut lire une variable ou un tableau. Le nombre d'éléments est alors fixé par les dimensions déclarées du tableau **dans l'unité de programmation qui effectue la lecture**.

```
integer n
real*8 x,y
real*8 a(1000)
...
read (22, ...) n, x, y
read (22, ...) a
...
```

7.6.2 Écritures non-formattées

De même, c'est l'instruction `write` sans le paramètre format. On peut écrire une variable ou un tableau de variables. Dans ce dernier cas, le nombre d'éléments

est fixé par les dimensions déclarées du tableau **dans l'unité de programmation qui effectue l'écriture**.

```
      integer n
      real*8 x,y
      real*8 a(1000)
      ...
      write (88, ...) n, x, y
      write (88, ...) a
      ...
```

7.7 `read` et `write` : l'accès direct

Un fichier en «accès direct» peut être considéré comme un tableau d'enregistrement (formattés mais plus généralement non formattés). Ainsi tout ordre de lecture ou d'écriture sur ce fichier doit comporter un paramètre supplémentaire `rec` qui est le numéro de l'enregistrement accédé (i.e. son «indice» dans le tableau). Par exemple :

```
      ...
      real*8 xx(1000), yy(1000)
      integer num_rec
      ...
      num_rec= ...
      ...
c .........  lecture sur un fichier en acces direct
      read  (23, rec=num_rec, ...) xx, yy
      ...
```

et de même pour l'écriture :

```
      ...
      real*8 a(1000)
      ...
      ...
c .........  ecriture sur un fichier en acces direct
      write  (22, rec=7)  a
      ...
```

7.8 gestion du «pointeur de fichier»

Si le fichier est en accès direct, la lecture ou l'écriture se fait à l'endroit où est se situe le «pointeur mobile» et ce pointeur peut être positionné sur n'importe quel enregistrement du fichier. En revanche, pour un fichier dont l'organisation est séquentielle, le sens de parcours est imposé. Il est néanmoins possible d'agir de façon limitée à l'aide des deux instructions `rewind` et `backspace`.

7.8.1 Rembobiner

Ce non est bien sur hérité des bandes magnétiques. Cette instruction permet de revenir au début du fichier connecté à une unité logique. Les arguments sont :

L' unité logique : le premier paramètre obligatoire se code de préférence à l'aide du mot-clef `unit`.

```
      ...
      rewind (3, ...)
```

```
      ...
c .........  utiliser de preference
      rewind (unit=3, ...)
```

En cas d'erreur : nous retrouvons ici les paramètres des instructions `open` et `close` avec la même signification :
– `iostat` (recommandé) et `err` (à éviter).

Attention :

- ▷ ne pas utiliser avec les unités «standards» 5 et 6 qui peuvent être reliées au clavier ou à l'écran ;
- ▷ ne pas utiliser avec les fichiers en accès direct.

Conseil méthodologique :

- ► peut être utile pour lire plusieurs fois le même fichier ou pour gérer des fichier temporaires, mais son utilisation doit rester exceptionnelle.

7.8.2 Un pas en arrière

Cette instruction permet de repositionner le «pointeur de fichier» avant le dernier enregistrement écrit ou lu i.e. à revenir dans l'état où l'on se trouvait avant d'avoir exécuté la dernière instruction `write` oui `read` sur l'unité logique correspondant à ce fichier. Les arguments sont :

L' unité logique : le premier paramètre obligatoire se code de préférence à l'aide du motclef `unit`.

```
      ...
      backspace (25, ...)
```

```
      ...
c .........  utiliser de preference
      backspace (unit=25, ...)
```

En cas d'erreur : nous retrouvons ici les paramètres des instructions `open` et `close` avec la même signification :
– `iostat` (recommandé) et `err` (à éviter).

Attention :

- ▷ ne pas utiliser avec les unités «standards» 5 et 6 qui peuvent être reliées au clavier ou à l'écran ;
- ▷ ne pas utiliser avec les fichiers en accès direct.

Conseil méthodologique :

- ► cette instruction est réputée peu efficace mais surtout ne semble pas fonctionner de façon «homogène» sur tous les systèmes. À proscrire absolument !

7.8.3 Gérer la fi n de fi chier

L'instruction `endfile` permet d'écrire explicitement un marqueur de «fin de fichier» à la suite de ce que l'on a déjà écrit sur une unité logique. Notons que l'instruction `close` fait cela très bien. Une utilisation possible serait dans le cas d'un fichier temporaire sans nom quand la partie de programme qui relie le fichier après un `rewind` n'a pas connaissance du nombre d'enregistrements déjà écrits. Les arguments sont :

L' unité logique : le premier paramètre obligatoire se code de préférence l'aide du mot-clef `unit`.

```
      ...
      endfile (25, ...)
```

```
      ...
c .........  utiliser de preference
      endfile (unit=25, ...)
```

En cas d'erreur : nous retrouvons ici les paramètres des instructions `open` et `close` avec la même signification :
– `iostat` (recommandé) et `err` (à éviter).

Attention :
- ▷ ne pas utiliser avec les unités «standards» 5 et 6 qui peuvent être reliées au clavier ou à l'écran ;
- ▷ ne pas utiliser avec les fichiers en accès direct.

Conseil méthodologique :
- ► cette instruction ne semble pas fonctionner de façon «homogène» sur tous les systèmes. À proscrire ! sauf éventuellement et exceptionnellement dans le cas de fichiers temporaires.

7.9 Des exemples

7.9.1 Fichiers «classiques»

Pour manipuler des fichiers «classiques», c'est à dire séquentiels et formattés, un minimum d'information suffit. Le paramètre `status` peut être utile, par

exemple pour des fichiers de sorties si l'on veut être sûr de ne pas écraser des résultats déjà obtenus :

```
      ...
      integer  io_erreur
c ..... fichier de donnees, doit exister
      open (unit=3, file='fpgm.don3', status='old',
     &      iostat=io_erreur)
      ...
c ..... le fichier 3 est a detruire apres lecture
      close (unit=3, status='delete', iostat=io_erreur)
      ...
```

```
      ...
      integer  io_erreur
c ..... fichier resultat ecrase si il existe deja
      open (unit=7, file='fpgm.res7', status='unknown',
     &      iostat=io_erreur)
      ...
c ..... le fichier 7 est conserve
      close (unit=7, iostat=io_erreur)
      ...
```

```
      ...
      integer  io_erreur
      char*80  nom_file_res
      ...
      nom_file_res= ...
      ...
c ..... ce fichier de resultat est a creer
c ..... obligatoirement
      open (unit=8, file=nom_file_res, status='new',
     &      form='unformatted', iostat=io_erreur)
      ...
c ..... le fichier 8 est conserve
      close (unit=8, iostat=io_erreur)
      ...
```

7.9.2 Fichier «binaire»

Il est souvent intéressant d'utiliser des fichiers «binaires» c'est à dire non formattés pour stocker de grosses quantités d'informations (typiquement des tableaux de nombres).

Ces fichiers sont moins volumineux mais surtout il n'y a pas de pertes d'information dues, par exemple, aux arrondis (contrairement à ce qui peut se passer lors d'une écriture et d'une relecture d'un nombre représenté sous forme de chaîne de caractères).

```
      ...
      integer  io_erreur
      ...
      ...
c
c ..........  creation fichier de donnees binaire
c
      open (unit=10, file='fpgm.bin1', status='old',
     &      form='unformatted', iostat=io_erreur)
      ...
      write (10) ...
      ...
      ...
      close (unit=10, iostat=io_erreur)
      ...
```

7.9.3 Fichier en «accès direct»

Cette possibilité n'existe pas toujours sur les calculateurs...on l'utilise le plus souvent pour optimiser l'accès à de grandes quantités de données et donc avec des fichiers non formattés. Le paramètre `recl` indique une taille **maximale** pour les enregistrements lu ou écrits par le programme (bien sûr, chaque système impose sa propre limitation «physique») :

```
      ...
      real*8 a(100), b(100), c(50)
      integer  io_erreur, num_rec
      ...
      ...
c
c ..... fichier de donnees (binaires)
c .....  a acces direct
c
      open (unit=33, file='newdata/fpgm.dir3',
     &      status='old', access='direct',
     &       recl=5000,   iostat=io_erreur)
      ...
      ...
c
c .....  lecture de l'enregistrement num_rec
c
      read (33, rec=num_rec) a, b, c
      ...
      ...
      ...
      close (unit=33, iostat=io_erreur)
      ...
```

7.9.4 Fichier «temporaire»

Le statut d'ouverture `'scratch'` doit être reservé à des fichiers temporaires écrits et lus au cours d'une même exécution. Ils seront le plus souvent «binaires». Notons qu'il n'est pas toujours possible (et de toutes façons inutile) de nommer ces fichiers :

```
      ...
      integer  io_erreur
      ...      ...
c ..... fichier temporaire
      open (unit=50, status='scratch',
     &      form='unformatted', iostat=io_erreur)
      ...
c .....  sauvegarde de donnees
      write(50) ...
      ...
c .....  <<rembobinage>>
      rewind(50)
      ...
c .....  relecture des donnees sauvegardees
      read(50)
      ...
c ..... ici le statut par defaut est 'delete'
      close (unit=50, iostat=io_erreur)
      ...
```

Chapitre 8

Les fonctions standards

Le langage `Fortran` comporte un certain nombre de fonctions prédéfinies, dites «intrinsèques» qui constituent pour l'essentiel une bibliothèque mathématique. Quelques fonctions sont utiles pour manipuler des chaînes de caractères. Tout cela a été déjà rencontré ou évoqué dans les chapitres précédents.

Souvent ces fonctions ont un nom générique et des noms spécifiques dépendant du type des arguments. Il est bien sûr préférable d'utiliser le nom générique ! Néanmoins, rappellons que si l'on utilise une ce ces fonctions comme argument l'usage du nom spécifique s'impose. Il n'est pas obligatoire mais préférable d'utiliser la déclaration `intrinsic` pour exprimer que l'on utilise une fonction standard `Fortran` ce qui permet d'éviter les confusions avec des fonctions «utilisateur».

Rappellons que le type `double complex` est une extension à la norme. Il en est de même pour toutes les fonctions qui le manipule. Ces extensions sont par la suite signalées par un astérisque $^{(*)}$. Enfin les noms ne suivant pas vraiment de convention très précise, il y a souvent plusieurs formes, selon les dialectes, pour les noms qui ne sont pas définis par la norme.

8.1 Conversions

opération	fonction	type résultat	exemple
xxx→integer	int	integer	i=int(x)
xxx→real	real	real	x=real(i)
xxx→double precision	dble	double precision	dx=dble(i)
xxx→complex	cmplx	complex	cx=cmplx(x) ou cx=cmplx(x,y)
xxx→double complex(*)	dcmplx(*)	double complex(*)	cd=dcmplx(x) ou cd=dcmplx(x,y)

8.2 Manipulations variées

fonction	type operande(s)	type résultat	exemple
min et max (au moins deux arguments)			
min	integer real double precision	integer real double precision	i=min(j,k)
max	integer real double precision	integer real double precision	i=max(j,k,l)

fonction	type operande(s)	type résultat	exemple
«troncation» d'un réel			
aint	real	real	x=aint(y)
	double precision	double precision	
Entier le plus proche d'un réel (`int(x+0.5)` si x>=0, `int(x-0.5)` sinon)			
nint	real	integer	i=nint(x)
	double precision	integer	
anint	real	real	x=anint(y)
	double precision	double precision	
modulo			
mod	integer	integer	i=mod(j,k)
	real	real	
	double precision	double precision	
transfert de signe (|`y`| avec le signe de `z`)			
sign	integer	integer	x=sign(y,z)
	real	real	
	double precision	double precision	
différence positive (`y-z` si y>z, `0` sinon)			
dim	integer	integer	x=dim(y,z)
	real	real	
	double precision	double precision	
produit en double précision			
dprod	real	double precision	x=dprod(a, b)

8.3 Fonctions mathématiques

fonction	type operande(s)	type résultat	exemple
valeur absolue			
abs	integer	integer	x=abs(y)
	real	real	
	double precision	double precision	
	complex	real	
	double complex(*)	double precision	
racine carrée			
sqrt	real	real	x=sqrt(y)
	double precision	double precision	
	complex	complex	
	double complex(*)	double complex(*)	
exponentielle			
exp	real	real	x=exp(y)
	double precision	double precision	
	complex	complex	
	double complex(*)	double complex(*)	
logarithme			
log	real	real	x=log(y)
	double precision	double precision	
	complex	complex	
	double complex(*)	double complex(*)	
logarithme en base 10			
log10	real	real	x=log10(y)
	double precision	double precision	

fonction	type operande(s)	type résultat	exemple
sinus			
sin	real double precision complex double complex(*)	real double precision complex double complex(*)	x=sin(y)
cosinus			
cos	real double precision complex double complex(*)	real double precision complex double complex(*)	x=cos(y)
tangente			
tan	real double precision	real double precision	x=tan(y)
arc sinus			
asin	real double precision	real double precision	x=asin(y)
arc cosinus			
acos	real double precision	real double precision	x=acos(y)
arc tangente (atan2(y,z)<=>atan(y/z))			
atan	real double precision	real double precision	x=atan(y) x=atan2(y,z)

fonction	type operande(s)	type résultat	exemple
sinus hyperbolique			
sinh	real double precision	real double precision	x=sinh(y)
cosinus hyperbolique			
cosh	real double precision	real double precision	x=cosh(y)
tangente hyperbolique			
tanh	real double precision	real double precision	x=tanh(y)

fonction	type operande(s)	type résultat	exemple
conjugué d'un nombre complexe			
conjg	complex double complex(*)	complex double complex(*)	cy=conjg(cx)
partie réelle			
real	complex double complex(*)	real double precision	x=real(c)
partie imaginaire			
imag	complex double complex(*)	real double precision	x=imag(c)

8.4 chaînes de caractères

fonction	type operande(s)	type résultat	exemple
Représentation interne d'un caractère			
ichar	character*1	integer	i=ichar(c)
char	integer	character*1	c=ichar(i)
longueur d'une chaîne			
len	character*	integer	i=len(cc)
Position d'une sous chaîne dans une chaîne			
index	character*	integer	i=index(cc, sc)
Comparaison, ordre lexical (`llt` : <, `lle` : <=, `lge` : >=, `lgt` : >)			
llt	character*	logical	l=llt(c1, c2)
lle	character*	logical	l=lle(c1, c2)
lge	character*	logical	l=lge(c1, c2)
lgt	character*	logical	l=lgt(c1, c2)

8.5 Les noms spécifiques

Les tableaux précédents utilisent toujours le nom générique indépendant donc du type des données manipulées. Le nom spécifique, lui, n'existe pas toujours, il peut être identique au nom générique, il peut y en avoir plusieurs (...) Notons qu'il est préférable de toujours utiliser le nom générique, le nom spécifique n'est indispensable que dans le cas rare d'une fonction passée en argument.

Conversions

générique	type argument(s)	type résultat	nom spécifique
int	real	int	int ou ifix
	double precision	int	idint
	complex	int	
	double complex(*)	int	
real	integer	real	real ou float
	double precision	real	sngl
	complex	real	
	double complex(*)	real	
dble	integer	double precision	dble ou dfloat
	real	double precision	dreal(*)
	complex	double precision	dbleq
	double complex(*)	double precision	

Manipulations variées

générique	type argument(s)	type résultat	nom spécifique
min	integer	integer	min0
	integer	real	amin0
	real	real	amin1
	real	integer	min1
	double precision	double precision	dmin1
max	integer	integer	max0
	integer	real	amax0
	real	real	amax1
	real	integer	max1
	double precision	double precision	dmax1

générique	type argument(s)	type résultat	nom spécifique
aint	real	real	aint
	double precision	double precision	dint
nint	real	integer	nint
	double precision	integer	idnint
anint	real	real	anint
	double precision	double precision	dnint
mod	integer	integer	mod
	real	real	amod
	double precision	double precision	dmod
sign	integer	integer	isign
	real	real	sign
	double precision	double precision	dsign
dim	integer	integer	idim
	real	real	dim
	double precision	double precision	ddim

Fonctions mathématiques

générique	type argument(s)	type résultat	nom spécifique
abs	integer	integer	iabs
	real	real	abs
	double precision	double precision	dabs
	complex	real	cabs
	double complex(*)	double precision	zabs(*)
sqrt	real	real	sqrt
	double precision	double precision	dsqrt
	complex	complex	csqrt
	double complex(*)	double complex(*)	zsqrt(*) oucdsqrt(*)
exp	real	real	exp
	double precision	double precision	dexp
	complex	complex	cexp
	double complex(*)	double complex(*)	zexp(*) ou cdexp(*)
log	real	real	alog
	double precision	double precision	dlog
	complex	complex	clog
	double complex(*)	double complex(*)	zlog(*) ou cdlog(*)
log10	real	real	alog10
	double precision	double precision	dlog10
sin	real	real	sin
	double precision	double precision	dsin
	complex	complex	csin
	double complex(*)	double complex(*)	zsin(*) ou cdsin(*)
cos	real	real	cos
	double precision	double precision	dcos
	complex	complex	ccos
	double complex(*)	double complex(*)	zcos(*) ou cdcos(*)
tan	real	real	tan
	double precision	double precision	dtan

générique	**type argument(s)**	**type résultat**	**nom spécifique**
asin	real double precision	real double precision	asin dasin
acos	real double precision	real double precision	acos dacos
atan	real double precision	real double precision	atan datan
atan2	real double precision	real double precision	atan2 datan2
sinh	real double precision	real double precision	sinh dsinh
cosh	real double precision	real double precision	cosh dcosh
tanh	real double precision	real double precision	tanh dtanh
conjg	complex double complex(*)	complex double complex(*)	conjg dconjg(*)
imag	complex double complex(*)	real double precision	aimag dimag(*)

Annexe A

Précédence et associativité

Cette table donne pour tous les opérateurs du langage **FORTRAN**

- la précédence (par ordre décroissant) et
- l'associativité, c'est à dire l'ordre d'évaluation des expressions.

`Opérateurs :`	`Associativité :`

`**`
`*` `/`
`+` (unaire et binaire) `-` (unaire et binaire)
	droite
	gauche
	gauche

`//` (concaténation)	gauche
`.lt.` `.le.` `.ge.` `.gt.` `.eq.` `.ne.`	

`.not.`
`.and.`
`.or.`
`.eqv.` `.neqv.`
	droite
	gauche
	gauche
	gauche

Il n'y a pas d'associativité associée aux opérateurs relationnels (`.lt.`, `.le.`, `.ge.`, `.gt.`, `.eq.` et `.ne.`). Ils ne permettent pas à eux-seuls de former des expressions (par exemple `i.eq.j.eq.k` n'est pas valide).

En cas de doute(s), l'utilisation de parenthèses redondantes n'est jamais pénalisante !

Annexe B

Liste des mots-clef

Ce ne sont pas des «mots reservés» mais il est préférable d'éviter de les utiliser pour nommer des variables.

- `program, subroutine, function, block data, entry`
- `integer, real, logical, character, double precision, complex`
- `implicit`
- `double complex, implicit none` (extensions à la norme)
- `common, dimension, equivalence, parameter, external, intrinsic`
- `save, data`
- `assign, to, call, go to, do, if, then, else, elseif, endif`
- `continue, pause, return, end, stop`
- `open, close, inquire, read, write, print, format, endfile`
- `rewind, backspace`

D'autres mots-clef sont utilisés uniquement dans les ordres d'entrées/sorties (`open`, `close`, `inquire`, `read` et `rewind`) :

- `access, blank, direct, end, err, exit, file, fmt, form`
- `iostat, name, named, nextrec, number, opened, rec, recl`
- `sequential, status, unformatted, unit`

Il faudrait rajouter à cette liste les noms de toutes les fonctions prédéfinies.