

Mémo Fortran

Mathias DELESCLUSE et Lionel GUEZ

31 janvier 2012

Table des matières

1	Pour compiler un programme	2
2	Programme vide de base	2
3	Déclarations de variables	2
4	Affichage et lecture	3
5	Conditions	3
6	Opérations de comparaison	4
7	Opérations sur les booléens	4
8	Boucles	4
9	Les vieux programmes	5
10	Les fichiers textes	5
11	Les formats d'écriture et de lecture	7
11.1	Écriture des réels	8
11.2	Les chaînes de caractères	8
11.3	Les entiers	9
11.4	Composer les formats	9
11.5	Instruction format	9
12	Convertir les variables	10
13	Les tableaux	11
13.1	Déclaration	11
13.2	Manipulation	12
13.3	Le cas particulier des chaînes de caractères	12
14	Les constantes symboliques	12
15	Les fonctions	13
16	Les sous-programmes (subroutine)	14

1 Pour compiler un programme

```
gfortran -ffree-form mon_programme.f -o mon_programme
```

ou, si vous avez un makefile :

```
make mon_programme
```

Le makefile contient le nom du compilateur (gfortran pour les travaux pratiques) et éventuellement des options du compilateur. Notamment, on peut choisir de compiler un programme avec des options pour détecter les erreurs, encore appelées “options de débogage”. L’inconvénient est qu’un programme compilé avec les options de débogage s’exécute plus lentement. On utilise donc en général les options de débogage seulement dans la phase de mise au point d’un programme. Pour exécuter le programme :

```
./mon_programme
```

2 Programme vide de base

```
program vide
! les points d’exclamations servent à commenter
!1 ligne= 1 instruction
end program vide
```

Tout ce qui suit s’écrit entre les deux lignes ci-dessus, en commençant par la déclaration des variables.

Vous pouvez arrêter l’exécution d’un programme avec l’instruction stop. Avec un système d’exploitation de type Unix ou Linux, il est utile de faire suivre le stop d’un entier non nul : cet entier sera le code de retour du programme, indiquant au shell que le programme ne s’est pas terminé normalement. Exemple :

```
stop 1
```

L’instruction stop est utile dans des tests de déroulement normal du programme.

3 Déclarations de variables

```
implicit none
```

C’est une ligne conseillée avant toute déclaration de variable, pour une programmation plus sûre. Interdit l’utilisation de variables non déclarées.

Entiers :

```
integer nom_variable
```

32 bits : -2147483648 à 2147483647

Réels :

```
real nom_variable
```

(simple précision, en général 6 chiffres décimaux significatifs, 32 bits soit 4 octets)

```
double precision nom_variable
```

(double précision, en général 14 chiffres décimaux significatifs, 64 bits soit 8 octets)

Caractères et chaînes de caractères :

```
character nom_variable  
character(len=valeur) nom_variable
```

(la valeur de len est entière et indique le nombre de caractères)

Booléens :

```
logical nom_variable
```

4 Affichage et lecture

```
print *, chaîne de caractères, entier, réel, ...
```

Cette instruction écrit sur la sortie standard (l'écran). L'étoile signifie que le format d'une valeur affichée est choisi automatiquement par le compilateur en fonction du type de cette valeur. C'est-à-dire que vous ne contrôlez pas, par exemple pour un réel, le nombre de chiffres après la virgule affichés. Nous verrons plus loin comment choisir le format.

```
read *, variable entière, variable réelle...
```

Cette instruction lit sur l'entrée standard (le clavier). L'étoile signifie que le format de la valeur lue peut être n'importe quel format compatible avec le type de la variable, occupant un nombre de caractères quelconque. Nous verrons plus loin comment spécifier que telle valeur lue est censée occuper une certaine position sur la ligne et s'étendre sur un certain nombre maximal de caractères.

Remarque. Les entrées-sorties avec le format "*" sont généralement appelées en français entrées-sorties "au format libre". Ne pas confondre avec le "format libre" du fichier source, qui concerne la façon d'écrire le texte du programme Fortran lui-même. Le terme officiel (donc en anglais) pour les entrées-sorties avec le format "*" est "list-directed input-output", qui indique bien que le format est contrôlé simplement par le type des expressions ou variables apparaissant après read ou write.

5 Conditions

```
if (expression logique) then  
    instruction  
else  
    instruction  
end if
```

Le bloc else est facultatif. On peut imbriquer les conditions les unes dans les autres.

```

select case (variable)
case(valeur 1)
    instruction 1
case(valeur 2)
    instruction 2
...
case(valeur n)
    instruction n
case default
    instruction par défaut
end select

```

6 Opérations de comparaison

a == b : compare a et b, renvoie `.TRUE.` si $a=b$, `.FALSE.` sinon

a > b : compare a et b, renvoie `.TRUE.` si $a>b$, `.FALSE.` sinon

a < b : compare a et b, renvoie `.TRUE.` si $a<b$, `.FALSE.` sinon

a >= b : compare a et b, renvoie `.TRUE.` si $a>=b$, `.FALSE.` sinon

a <= b : compare a et b, renvoie `.TRUE.` si $a<=b$, `.FALSE.` sinon

Attention lorsque vous comparez des réels : rappelez-vous que les nombres sont représentés avec une précision finie et que la précision des calculs est finie!

7 Opérations sur les booléens

Si A et B sont de type logique :

A .or. B effectue l'opération OU logique

A .and. B effectue l'opération ET logique

.not. inverse la valeur d'un logique ou d'une expression de type booléenne.

8 Boucles

```

do while(booléen)
    Instruction
    Instruction
...
end do

```

Le booléen de type logique qui sert de condition d'entrée et de ré-entrée dans la boucle doit passer de `.TRUE.` à `.FALSE.` au cours de la boucle sans quoi la boucle est infinie.

```

do
    Instruction
    if (booléen) then
        exit
    endif
    Instruction
end do

```

La condition permet de sortir de la boucle, on peut la positionner où l'on veut.

```
do i=i_start, i_stop[, i_inc]
  Instruction
  Instruction
  ...
end do
```

A chaque itération, la variable entière `i` qui a commencé à `i_start` s'incrémente de `i_inc` et la boucle s'arrêtera quand `i` atteint `i_stop`. La valeur par défaut de `i_inc` est 1.

9 Les vieux programmes

Vous aurez peut-être affaire à de vieux programmes en Fortran. La norme Fortran 77 est incluse dans la norme Fortran 90 et, à peu de choses près, dans la norme Fortran 95. En d'autres termes, tout programme conforme à la norme Fortran 77 est conforme à la norme Fortran 90. Si bien qu'avec un compilateur moderne, vous pouvez compiler et utiliser tel quel un vieux programme en Fortran. Néanmoins, du fait des nouvelles possibilités apportées par Fortran 95, le style optimal de programmation en Fortran 95, du point de vue de la concision, de la clarté, de la sécurité de programmation, est nettement différent du style de Fortran 77. Certains aspects du Fortran 77, bien que valides (donc acceptés par n'importe quel compilateur) sont déclarés officiellement (dans la norme) obsolètes en Fortran 90 et Fortran 95. Entre autres, le format fixe de fichier source est obsolète. (Dans le format fixe, les 6 premières colonnes sont réservées, les lignes sont limitées à 72 caractères, un caractère dans la sixième colonne indique une continuation de ligne, un caractère C ou * dans la première colonne signale une ligne de commentaires.) Vous pourrez trouver aussi dans de vieux programmes des étiquettes (un numéro en début de ligne) marquant une fin de boucle DO et des instructions GOTO. Les boucles DO avec étiquette peuvent être avantageusement remplacées par des blocs DO ...END DO. Les instructions GOTO peuvent toujours être avantageusement supprimées en n'utilisant que des blocs IF... ENDIF, DO... END DO et éventuellement les instructions EXIT ou CYCLE.

10 Les fichiers textes

Comment lire et écrire un fichier texte (en format ASCII) à partir d'un programme Fortran ?

L'instruction OPEN.

```
OPEN(UNIT=..., FILE=..., STATUS=..., ACTION=..., POSITION=...)
```

Cette commande permet d'associer un numéro d'unité (argument "unit") à un fichier (argument "file"). Les cinq arguments de la commande "open" ci-dessus sont des arguments d'entrée : vous devez fournir des valeurs.

unit : vous devez fournir une valeur entière, supérieure ou égale à 0. La norme Fortran ne précise pas de valeur maximale possible, qui dépend donc du compilateur. Certaines valeurs sont en général réservées : 5 pour l'entrée

standard et 6 pour la sortie standard, mais ce n'est pas imposé par la norme, donc dépendant a priori du compilateur. Par ailleurs, si vous ouvrez plusieurs fichiers en même temps, c'est à vous de faire attention à choisir un numéro différent pour chaque fichier. Une méthode sûre et générale consiste à appeler un sous-programme qui fournit un numéro correct et libre (sous-programme général que vous pourrez ré-utiliser dans n'importe quel programme, joint pour les TP).

file : fournissez le nom du fichier. C'est une chaîne de caractères, à renseigner entre guillemets ou apostrophes donc.

status : la chaîne affectée à STATUS peut prendre les valeurs suivantes :

'old' : le fichier doit déjà exister.

'new' : crée un nouveau fichier, le fichier ne doit pas déjà exister

'replace' : crée un nouveau fichier, en écrasant éventuellement un fichier déjà existant

'scratch' : crée un fichier temporaire le temps de l'exécution

action peut recevoir les chaînes de caractères suivantes :

'read' : permet la lecture uniquement

'write' : permet l'écriture uniquement

'readwrite' : permet les deux

position peut recevoir les chaînes de caractères suivantes :

"rewind" : pour se placer au début du fichier

"append" : pour se placer à la fin du fichier

Lecture et écriture du fichier ouvert. Une fois le fichier ouvert avec OPEN, on lit ou écrit dans ce fichier en fournissant le numéro d'unité dans les instructions READ et WRITE :

```
read(unit, fmt=*) ...  
write(unit, fmt=*) ...
```

Exemple :

```
WRITE(unit=11, fmt=*) "l'entier i vaut:", i
```

écrit dans le fichier correspondant à l'unité 11 ce qui serait écrit à l'écran par :

```
print *, "l'entier i vaut:", i
```

Tout est toujours écrit en ASCII, y compris la valeur de i. L'argument `fmt=*` signifie que le format libre est utilisé, donc la chaîne de caractère représentant i sera adaptée automatiquement au type de la variable "i". (Voir le chapitre sur les formats.)

```
READ(unit=11, fmt=*) i
```

lit dans le fichier correspondant à l'unité 11 au format libre (s'attend donc ici à une chaîne de caractère représentant une valeur du type de la variable i). WRITE et READ acceptent également l'option IOSTAT, qui est très utile (cf 9.4)

Fermeture du fichier. Une fois les opérations de lecture/écriture terminées, on peut fermer le fichier par

`CLOSE(unit)`

pour fermer le fichier dont le numéro d'unité est `n`. A partir de ce moment là, le fichier sur le disque dur est définitivement enregistré, et aucun lien ne le lie plus au programme. D'autres processus peuvent y accéder, et on est sûr que rien dans le code qui suit le `CLOSE` ne pourra le modifier. Si on tente un `READ(unit, fmt=*)` ou `WRITE(unit, fmt=*)` après `CLOSE(unit)`, ce sera un échec.

Lire ou écrire dans un fichier à l'intérieur d'une boucle. Dans le cas où, sur chaque ligne du fichier, on doit avoir un même nombre de valeurs, avec la même succession de types, on peut traiter le fichier en plaçant `READ` ou `WRITE` dans une boucle. Ci-dessous un squelette typique pour lire un fichier entier ligne par ligne. Nous supposons ici que le fichier contient une seule valeur par ligne, toujours du même type.

```
OPEN(UNIT=11, FILE='fichier', STATUS='OLD', ACTION='READ', position= "rewind")
DO
  ... instructions ...
  READ (unit=11, fmt=*, IOSTAT=iostat) i
  IF (iostat /= 0) exit
  ... instructions ...
ENDDO
CLOSE(UNIT=11)
```

Attention dans ce cas à ouvrir le fichier (avec `OPEN`) et le fermer (avec `CLOSE`) en DEHORS de la boucle. Si on ferme par exemple le fichier dans la boucle, le `READ` de la prochaine itération n'aboutira pas.

attention : ici, on peut sortir de la boucle pour d'autres erreurs qu'une fin de fichier. Si par exemple la variable Fortran `i` est déclarée de type entier, et que dans le fichier se trouve un réel, l'instruction `read` affecte à `iostat` une valeur non nulle et on sort de la boucle sans lire le reste des lignes. Les significations des valeurs non nulles données à `iostat` dépendent du compilateur. Par contre, le fait qu'une valeur nulle signifie qu'il n'y a pas eu d'erreur est prescrit par la norme (donc indépendant du compilateur).

11 Les formats d'écriture et de lecture

Au lieu de laisser le compilateur, avec `fmt=*`, choisir automatiquement le format d'écriture, ou reconnaître automatiquement les valeurs en lecture, vous pouvez spécifier un format, en écrivant :

```
FMT='(descripteur de format)'
```

Le format peut être explicité dans l'argument `fmt` des instructions `read` et `write`, et dans les instructions simplifiées `read` et `print`. En effet, dans les instructions :

```
read *, liste de variables
print *, liste de valeurs à imprimer
```

l'étoile indique le format libre. On peut choisir un format explicite à la place :

```
read '(descripteur de format)', liste de variables
print '(descripteur de format)', liste de valeurs à imprimer
```

Les sous-parties qui suivent traitent principalement des formats d'écriture, mais le principe est le même en lecture. Pour chacun des principaux types de variables, le format libre et les formats explicites sont détaillés.

En lecture, si vous spécifiez un format explicite, le fichier que vous lisez doit utiliser ce format explicite sans aucune exception. En format libre (`fmt=*`), la lecture est plus souple. Remarquez qu'en lecture au format libre, il faut des séparateurs (espace, virgule ou retour à la ligne) entre les valeurs (les séparateurs ne sont pas nécessaires avec un format explicite). Vous pouvez utiliser un format explicite en lecture si les données sont censées être cadrées d'une façon connue sur les lignes. L'intérêt est que vous pouvez détecter ainsi des problèmes inattendus dans le fichier lu. L'inconvénient est que la programmation est plus lourde.

11.1 Écriture des réels

Par défaut (`FMT=*`), les réels sont affichés avec tous leurs chiffres significatifs, avec ou sans exposant selon la valeur. Si on veut quelque chose de plus régulier, il faut spécifier un format à la place de `*`.

Pour afficher un réel (simple ou double précision) sans exposant, avec un nombre de décimales fixé, on utilisera `'(fw.d)'` où `w` est un entier réservant le nombre total de caractères (le signe, le point décimal, les chiffres avant et après le point décimal), et `d` est un entier indiquant combien de décimales seront affichées.

Pour afficher un réel (simple ou double précision) avec exposant, avec un nombre de décimales fixé, on utilisera `'(esw.d)'` où `w` est un entier réservant le nombre total de caractères (le signe, le point décimal, les chiffres avant et après le point décimal, la lettre "e" avant l'exposant, le signe de l'exposant, les chiffres de l'exposant), et `d` est un entier indiquant combien de décimales seront affichées. Si la valeur affichée est non nulle, la valeur avant l'exposant est supérieure ou égale à 1 et strictement inférieure à 10.

Attention : si, avec le format `fw.d`, la valeur du réel est trop grande pour être affichée sur `w` caractères, ou plus généralement si vous n'avez pas prévu assez de caractères, des étoiles sont affichées sur toute la largeur du champ, c'est-à-dire `w` étoiles. Exemple : Si vous tentez d'afficher 1234.5678 avec `f5.4`, vous obtiendrez `'*****'`.

11.2 Les chaînes de caractères

Le format libre `fmt=*` écrit tout simplement la chaîne (sans les délimiteurs, apostrophes ou guillemets). L'effet est le même avec le format explicite `FMT='(a)'`. Si on veut insérer des blancs entre deux valeurs (de n'importe quel type) sur une ligne, on peut utiliser la commande de positionnement `nx`, où "n" est le nombre de blancs souhaités. La commande de positionnement se place dans le format. Par exemple, pour imprimer un nombre réel, suivi de 10 blancs, suivi d'une chaîne de caractères, on peut utiliser le format :

```
'(f4.2, 10x, a)'
```

Pour écrire une chaîne de caractères non suivie d'un retour à la ligne :

```
write(unit=*, fmt="(a)", advance="no") "chaîne quelconque"
```

C'est utile par exemple pour demander une valeur à l'utilisateur :

```
write(unit=*, fmt="(a)", advance="no") "Nombre de pas de temps ? "  
read *, delta_t
```

Le curseur reste alors sur la ligne de la question, en attente d'une valeur.

11.3 Les entiers

L'écriture d'un entier au format libre `FMT=*` peut être, selon le compilateur, cadrée à droite dans un champ de longueur indépendante de la valeur, ou occuper un nombre de caractères variable adapté à la valeur.

Pour afficher un entier avec un nombre de caractères `w`, et cadrage à droite, utiliser le format explicite `'(iw)'`. Pour afficher des zéros de tête, utiliser `'(iw.m)'`, où `m` est le nombre minimal de chiffres imprimés. Par exemple `FMT='(I10.8)'` réservera 10 caractères, dont au moins 8 chiffres, avec éventuellement des zéros de tête si la valeur écrite contient moins de huit chiffres.

```
print "(i10.7)", -12345
```

donne :

```
-0012345
```

avec un blanc en tête.

11.4 Composer les formats

Pour enchaîner les formats, il faut les séparer par des virgules. Exemple :

```
READ '(a, 1x, I2, 1x, I10.5)', chaine, entier1, entier2
```

On peut aussi répéter des formats ou des blocs de formats. Exemple :

```
READ '(2(a, 1x, I2), 1x, 5I10.5)', chaine1, i1, chaine2, i2, i3, &  
i4, i5, i6, i7
```

11.5 Instruction format

Si vous utilisez un même format dans plusieurs instructions de lecture ou écriture, au lieu de le répéter, vous pouvez placer ce format dans une instruction `format` :

```
étiquette format (flist)
```

où l'étiquette est un entier compris entre 1 et 99999, et `flist` est le contenu du format, que vous auriez pu écrire dans l'instruction de lecture ou écriture. Dans l'instruction de lecture ou écriture, vous faites référence au format par son étiquette. Exemple :

```
read 1000, chaine, entier1, entier2  
1000 format (a, 1x, I2, 1x, I10.5)
```

12 Convertir les variables

Les constantes littérales numériques. Le compilateur considère par exemple les suites de chiffres décimaux, sans point décimal, comme des constantes entières. Les constantes réelles doivent être écrites avec un point décimal ou une lettre, e ou d, introduisant un exposant. Par exemple :

5. : réel simple précision

5e0 : réel simple précision

5d0 : réel double précision

5 : entier

Inconvénients du mélange des types. Dans une expression numérique, si des valeurs de types différents apparaissent (des entiers, des réels simple précision, des réels double précision...), les valeurs sont converties vers le type le plus “riche” : réel double précision plus riche que réel simple précision, réel simple précision plus riche qu’entier. Le risque principal concerne la division des entiers. En effet, l’opération de division entre deux entiers est une division euclidienne, qui a pour résultat un entier :

5 / 2 ! vaut 2

Alors que la division entre deux réels donne un réel :

5. / 2. ! vaut 2.5

Le risque principal, dans une expression mélangeant entiers et réels, est de penser faire une division réelle et de ne pas voir qu’une division entière est faite. Par exemple :

```
real x, a, b
x = 1 / 2 * (a + b)
```

Vous pensez peut-être calculer la moyenne de a et b mais le résultat est toujours $x = 0$ parce que $1 / 2$, division euclidienne, est nul. La moyenne est bien calculée par :

```
x = 1. / 2. * (a + b)
```

En dehors de ce risque d’erreur, une conversion représente une opération supplémentaire, un coût supplémentaire de temps. C’est donc aussi une raison d’éviter de mélanger les types dans une expression ou une affectation.

Subtilité : Quand vous lisez un fichier de texte contenant des nombres réels, vous lisez des chaînes de caractères représentant des nombres. Ces chaînes n’ont pas le type simple précision ou double précision par elles-mêmes. Si vous avez besoin d’un exposant, vous pouvez toujours l’introduire avec la lettre e, mettre la lettre d ne change rien. À la lecture, vous décidez si la valeur va dans une variable simple ou double précision.

Entiers et réels. Transformer un entier en réel double précision :

```
dble(entier)
```

On peut également de cette manière transformer un simple précision en double précision. Transformer un entier en réel simple précision :

```
real(entier)
```

Transformer un réel en entier :

```
int(réel)
```

Partie entière :

```
floor(réel)
```

Chaînes de caractères et nombres. Comment transformer une chaîne de caractères en entier ou réel ?

```
chaine='1234 567.89'  
READ(unit=chaine, fmt=*) i, real_SP
```

Et on récupère `i=1234` et `real_SP=567.8900` en simple précision. Comment transformer un entier ou un réel en chaîne de caractère ?

```
WRITE(unit=chaine, fmt='(I4, F7.2)') i, real_SP
```

On peut ainsi recréer la chaîne de l'exemple précédent.

13 Les tableaux

Un tableau est un espace mémoire contenant plusieurs valeurs d'un même type, celles-ci remplissant l'espace mémoire les unes derrière les autres. On accède aux différentes valeurs à l'aide d'un indice entier.

13.1 Déclaration

```
integer, dimension(1000):: tabi
```

déclare un tableau de 1000 entiers.

```
integer tabi(1000)
```

est une syntaxe équivalente, qui permet d'ajouter sur la même ligne des entiers qui ne sont pas des tableaux :

```
integer tabi(1000), i
```

ce que l'on ne peut faire avec un `integer, dimension(n)` qui ne déclare que des tableaux de dimension `n`. On peut évidemment déclarer un tableau d'un autre type, par exemple :

```
real tabr(1000)  
logical tabl(1000)
```

Attention : dans une telle déclaration, la taille indiquée doit être une constante. Vous ne pouvez pas mettre `dimension(n)`, où `n` est une variable qui va être définie au cours de l'exécution. Si la taille du tableau n'est pas connue au début du programme, il faut déclarer le tableau avec l'attribut `allocatable` :

```
real, allocatable:: a(:)
```

Lorsque le programme a déterminé la taille nécessaire, le tableau doit être alloué avant d'être utilisé :

```
allocate(a(n))
```

Pour déclarer un tableau à plusieurs dimensions, on écrit simplement, par exemple pour un tableau d'entiers à 2 dimensions :

```
integer tabi2(1000, 3)
```

13.2 Manipulation

Si on veut accéder au 3ème élément du tableau, on écrit simplement `tabi(3)` si `tabi` est mon tableau d'entiers déclaré dans le premier paragraphe. Pour accéder au *n*-ième élément, on écrirait `tabi(n)` avec *n* une variable entière, dont on doit s'assurer que sa valeur ne dépasse jamais la taille du tableau déclaré. Ceci entraînerait une erreur à l'exécution ou pire, pas d'erreur détectée mais un résultat faux.

Pour attribuer une valeur au tableau, on écrit par exemple :

```
tab(3) = 500
```

13.3 Le cas particulier des chaînes de caractères

Déclaration. On peut déclarer un tableau de caractères :

```
character tabcar(1000)
```

pour une taille de 1000 caractères. Réfléchissez si un tableau de caractères ou une chaîne de caractère est plus pratique pour votre problème. La déclaration déjà étudiée :

```
character(len=1000) tabcar
```

permet d'utiliser des opérations spécifiques aux chaînes de caractères.

Manipulation. Pour donner une valeur :

```
character(len=10) chaine  
chaine='abcdefghijkl'
```

Pour faire référence à la sous-chaîne comprise entre le 2ème et le 5ème caractère inclus, soit `'bcde'`, il me suffit d'utiliser la syntaxe `chaine(2:5)`. Si on veut accéder au 6ème caractère uniquement, on utilise `chaine(6:6)`. Avec des variables *i* et *j* entières, on peut bien sûr accéder à la chaîne de longueur *j-i* qui commence à *i* grâce à la syntaxe `chaine(i:j)` et aussi `chaine(i:i)` pour le *i*-ème caractère tout seul.

14 Les constantes symboliques

En programmation scientifique, on gagne souvent en clarté en utilisant des symboles pour des valeurs constantes : par exemple `pi` pour 3.14... ou `kB` pour la constante de Boltzmann. Déclarer `pi` comme une variable n'est pas sécurisé car on peut par accident changer sa valeur. La syntaxe suivante est appropriée :

```
double precision, parameter:: pi = 3.14159265d0
```

Quelle différence ? pi n'est alors PAS une variable, mais une référence pour le compilateur. Quand, dans la suite de votre programme, vous utiliserez pi, le compilateur remplacera pi par 3.14159265d0. En d'autres termes, c'est comme si vous aviez écrit 3.14159265d0 partout à chaque fois que vous aviez besoin de pi, mais sans avoir à le faire en réalité. Dans les manuels de Fortran en français, vous trouverez souvent le nom "constante symbolique" pour un objet avec l'attribut parameter, parfois le nom "paramètre". Dans les manuels en anglais, vous trouverez le nom "named constant".

Il est aussi pratique d'utiliser une constante symbolique pour une taille de tableau qui intervient dans plusieurs tableaux. Par exemple :

```
integer, parameter:: N = 2000
integer tabi(n)
real a(n,n)
```

15 Les fonctions

Dans le même ordre d'idée, on peut éviter d'écrire plusieurs fois les mêmes lignes de codes. L'exemple le plus parlant est le cas simple d'une fonction $f(x, y, z) = x^2 + y^2 + z^2$ à utiliser plusieurs fois dans un programme.

Déclaration. Une fonction s'écrit en dehors du programme principal et possède un type (réel, entier ...).

```
module f_m
  implicit none
contains
  double precision function f(x, y, z)
    double precision, intent(in):: x, y, z
    f=x**2 + y**2 + z**2
  end function f
end module f_m
```

Le type de f est ici déclaré dans la première ligne. Attention : les arguments x, y et z à l'intérieur de la fonction doivent avoir le même type que les arguments correspondants à l'appel de la fonction (voir plus bas, appel de la fonction). La valeur renvoyée est celle de la variable qui porte le nom de la fonction, ici f.

Attention : Les variables déclarées dans la fonction ne sont pas visibles du programme, et vice-versa. Il est donc possible de nommer les variables d'un nom identique à celles du programme principal (x,y,z). On peut évidemment mettre plus ou moins de 3 arguments à la fonction.

L'attribut "intent(in)" de x, y, z signifie que ces variables reçoivent une valeur lorsque la fonction est appelée et ne sont pas modifiées à l'intérieur de la fonction. Il est dangereux d'avoir des arguments modifiables et modifiés dans une fonction. (Sans entrer dans les détails, du fait que le résultat d'une fonction peut être utilisé directement dans une expression, des arguments peuvent devenir indéfinis, des optimisations par le compilateur peuvent devenir impossibles.). Il est donc conseillé de déclarer tous les arguments d'une fonction avec "intent(in)". La 'philosophie' de la fonction est d'utiliser des arguments en entrée, nécessaire au calcul d'une seule valeur de sortie, qui est la fonction elle-même. Pour modifier

plusieurs arguments à la fois, c'est à dire utiliser les arguments en entrée/sortie on utilise une subroutine.

Appel. Pour être appelée, une fonction doit être déclarée dans le programme principal :

```
program exemple
  use f_m, only: f
  implicit none
  integer i
  double precision x,y,z
  x=1d0
  y=1d0
  z=1d0
  do i=1, 1000
    print *, f(dble(i),y,z)
    print *, f(x,dbble(i),z)
    print *, f(x,y,dbble(i))
  enddo
end program exemple
```

16 Les sous-programmes (subroutine)

On va reprendre l'exemple précédent, mais cette fois, on va vouloir que x,y et z soient modifiés dans la fonction en plus de renvoyer une valeur f. Il faut toujours écrire la procédure en dehors du programme principal.

Déclaration.

```
module farg_m
  implicit none
contains
  subroutine farg(i,x,y,z,f)
    double precision, intent(inout):: x,y,z
    double precision, intent(out):: f
    integer i
    f=x**2+y**2+z**2
    x=x+dbble(i)*2d0
    y=y+dbble(i)*2.1d0
    z=z+dbble(i)*2.2d0
  end subroutine farg
end module farg_m
```

Les variables déclarées dans la subroutine ne sont toujours pas visibles du programme. Les variables x,y,z et f de la subroutine sont toutefois associées aux variables correspondantes dans l'instruction d'appel du sous-programme. Les arguments du côté de l'appelant sont appelés arguments effectifs, les arguments du côtés de l'appelé sont appelés arguments muet. Les types d'un argument effectif et de l'argument muet associé doivent être identiques.

Appels.

```
program exemple
  use farg_m, only: farg
```

```

implicit none
integer i
double precision x,y,z,f
do i=1,1000
  call farg(i,x,y,z,f)
  print *, i,x,y,z,f
enddo
end program exemple

```

Cette fois, x, y et z sont modifiés, et c'est le but d'une procédure.

Compilation avec plusieurs fichiers. On gagne en clarté à écrire chaque subroutine ou fonction dans un fichier séparé.

```
gfortran main.f90 subr.f90 func.f90 -o programme
```

est un exemple pour compiler tous les fichiers à la fois.

A Mots-clefs du langage Fortran 95

allocatable	format	print
allocate	function	private
assignment	if	procedure
backspace	implicit	program
call	in	public
case	include	pure
character	inout	read
close	inquire	real
complex	integer	recursive
contains	intent	result
cycle	interface	return
data	intrinsic	rewind
deallocate	logical	save
default	module	select
dimension	namelist	sequence
do	none	stop
double precision	nullify	subroutine
elemental	only	target
else	open	then
elsewhere	operator	type
end	optional	use
endfile	out	where
equivalence	parameter	while
exit	pointer	write
forall		