
FORTRAN

Généralités sur le langage Fortran

Les contenu des lignes en Fortran doit respecter les quelques règles suivantes :

- Une ligne de **commentaire** doit commencer par un `c` ou une `*` en première colonne.
- Tout ce qui suit un `!` dans une ligne est aussi considéré comme un **commentaire**.
- Les **instructions** :
  - doivent commencer à partir de la colonne 7 (ou plus) ;
  - se placent automagiquement avec[X]Emacs en appuyant sur la touche `TAB` ;
  - ne doivent pas dépasser la colonne 72.
- Les **Labels** (ou étiquettes) sont entre les colonnes 2 et 5 et gérés automagiquement avec [X]Emacs.
- Pour **couper** une instruction sur plusieurs lignes, on met un caractère quelconque en colonne 6 des lignes de suite.

La figure 2.1 illustre ces règles.

Figure 2.1: Exemples de contenu de lignes en Fortran

```
1 2 3 4 5 6 7                                                  72 73 74 75 76 77 78 79 80
c     Ligne de commentaire
*     Ici aussi
c     Ci-dessous: instruction
      c = a*b+(c-d)**2.0
c     Ici on coupe en deux
      e = a*b+(c-d)**2.0 +....+( theta*2
     &)**(0.5)
c     Label
c     10 continue
```

Organisation d'un programme FORTRAN

Succession de « *pavés* » élémentaires qu'on appellera *blocs fonctionnels*. Il en existe 3 sortes :

1. *Le programme principal* inclus entre `program` (facultatif) et `end`

```
program nom
...
end
```

2. *Les subroutines* inclus entre `subroutine` et `end`

```
subroutine nom (arguments)
...
end
```

3. *Les fonctions* inclus entre `function` et `end`

```
type function nom (arguments)
...
end
```

Programme principal

Le programme principal est obligatoirement présent. Il n'existe qu'un seul programme principal. Ce programme principal se découpe en deux parties distinctes successives détaillées ci-dessous.

Partie déclaration

C'est dans cette partie qu'on définit les objets (type + nom) qui seront manipuler par le programme.

Partie instructions

L'exécution d'un programme FORTRAN consiste à dérouler dans l'ordre toutes les instructions de la partie exécutable du programme principal.

Certaines instructions déroutent le pointeur de programme vers d'autres blocs fonctionnels (subroutines ou fonctions)

Les données

Les différents types de données

Le tableau 3.1 résume l'ensemble des types de données manipulables en Fortran. De plus, il est possible d'assembler plusieurs grandeurs dans des tableaux. Ce qui permet de créer des vecteurs, des matrices...

Nous allons maintenant détaillé ces types.

Tableau 3.1: Types de données manipulables en Fortran.

Grandeurs numériques	Entiers	`integer`
	Réels	`real`
		`double precision`
	Complexes	`complex`
Caractères		`character`
Grandeurs logiques (vraies ou fausses)		`logical`

Type `integer`

Un `integer` contient un entier et est représenté par son écriture en base 2 signée sur 4 octets (31 bits pour la valeur plus un bit pour le signe). Ses valeurs possibles sont dans l'intervalle $[-2^{31},2^{31}-1]$.

Type `real`

un `real` contient un nombre réel et est codé en virgule flottante (IEEE) sur 4 octets [3.1]. Chaque nombre est représenté sous la forme $x = \pm 0.m \times 2^e$ où *m* est la *mantisse* codée sur 23 bits et *e* est *l'exposant*, codé sur 8 bits ($-127 < e < 128$).

Les valeurs (en valeur absolue) sont comprises dans l'intervalle $[1.401 \times 10^{-45},\ 3.403 \times 10^{38}]$ et il stocke environ 7 chiffres significatifs.

Type `double precision`

Le `double precision` est un `real` plus précis, codé en virgule flottante sur 8 octets dont une mantisse codée sur 52 bits et un exposant codé sur 11 bits ($-1023 < e < 1024$).

Les valeurs (en valeur absolue) sont comprises entre $[4.941 \times 10^{-324}, 1.798 \times 10^{308}]$ avec 15 chiffres significatifs.

Type `complex`

Assemblage de 2 `real` dans un même objet.

Constantes numériques

Question: je veux utiliser dans un programme les nombres 1, 3.14, 2+3*i*.
Comment les écrire ?

Constantes `integer`

> Une constante de type `integer` est écrite sans point décimal.

Exemples :

```
1
123
-28
0
```

Constantes `real`

> Une constante de type `real` doit obligatoirement comporter :
>
> - soit le point décimal, même s'il n'y a pas de chiffres après la virgule ;
> - soit le caractère `e` pour la notation en virgule flottante.
>
> Pour les nombres écrits `0.xxxxx`, on peut omettre le `0` avant le point décimal.

Exemples :

```
0.
1.0
1.
3.1415
31415e-4
1.6e-19
```

```
1e12
.001
-36.
```

Constantes `double precision`

> Une constante **`double precision`** doit obligatoirement être écrite en virgule flottante, le `e` étant remplacé par un `d`.

Exemples :

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

Constantes `complex`

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

Exemples :

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

Définition de constantes symboliques

Elle permettent de référencer une constante à l'aide d'un symbole.

Elles ne peuvent être modifiées au milieu du programme et sont affectées une fois pour toutes avec le mot-clé **`parameter`** dans la section déclarations.

Syntaxe

```
parameter(const1=valeur1,const2=valeur2, ...)
```

Le type de chaque constante doit être déclaré explicitement ou en suivant les mêmes règles de typage automatique que les variables (cf 4.3). Une constante est toujours locale à un bloc fonctionnel.

Exemple

```
      double precision q
      parameter(max=1000, q=1.6d-19)
```

Les variables

> Une variable est un emplacement en mémoire référencé par un nom, dans lequel on peut lire et écrire des valeurs au cours du programme.

Les variables permettent (entre autres) de :

- manipuler des symboles ;
- programmer des formules.

Avant d'utiliser une variable, il faut :

- définir son type ;
- lui donner un nom.

C'est la *déclaration*. Elle doit être écrite dans la première partie (la partie déclaration) d'un bloc fonctionnel (programme principal, subroutine ou fonction) dans lequel intervient la variable.

Dans les langages modernes, *toute variable doit être déclarée*. En FORTRAN, il y a des exceptions obéissant à des règles bien précises.

> Une variable est :
>
> - *locale* si seul le bloc fonctionnel où elle est déclarée peut y accéder. C'est le défaut ;
> - *globale* si tous les blocs fonctionnels peuvent y accéder.

Pour savoir comment rendre une variable globale, voir le paragraphe concernant l'instruction **common** (10.6).

Déclaration de variables

A quel endroit ?

Entre le mot-clé

program
subroutine
function

et la première instruction exécutable.

Syntaxe

```
type var1, var2, var3, .....
```

On peut déclarer plusieurs variables du même type sur une même ligne.

Exemple

```
      integer i,j,k
      real alpha, beta
      double precision x,y
      complex z
```

Noms des variables

Pour nommer une variable, il faut respecter les règles suivantes :

- Caractères autorisés :
  - toutes les lettres,
  - tous les chiffres,
  - le caractère « blanc » (déconseillé),
  - le caractère « _ » (« underscore » $\neq$ « moins ») ;
- Le premier caractère doit être une lettre ;
- *Seuls les 6 premiers caractères sont significatifs* : `epsilon1` et `epsilon2` représentent la même variable ;
- (Rappel) pas de différence minuscules majuscules.

Règles de typage implicite

On peut ne pas déclarer une variable (fortement déconseillé), en utilisant les règles suivantes :

> Une variable dont le nom commence par `i,j,k,l,m,n` est automatiquement de type **integer**. Une variable commençant par toute autre lettre (de `a` à `h` et de `o` à `z`) est automatiquement de type **real**.

Directive implicit

Elle permet modifier ces règles par défaut *de manière locale* à un bloc fonctionnel.

```
implicit type (lettre1-lettre2, lettre3, ....)
```

Toute variable commençant par une lettre comprise entre lettre1 et lettre2 ou par lettre3 sera par défaut du type indiqué.

Cette directive doit être écrite juste après

program
subroutine
function

.

Exemples

```
      implicit real (a-c,e,w-z)
```

Tout ce qui commence par `a,b,c,e,w,x,y,z` sera **real**

```
      implicit double precision (a-h,o-z)
```

Similaire à la règle par défaut : tout ce qui commence par `i,j,k,l,m,n` sera **integer**, tout le reste **double precision** (*Très Utilisé !!*).

```
      implicit complex (z)
```

Tout ce qui commence par z est **complex** par défaut.

```
      implicit none
```

Aucune variable n'est utilisable si elle n'est pas déclarée.

> **Important** : il s'agit de règles applicables aux variables *non déclarées*. La déclaration d'une variable l'emporte sur les règles implicites.

Affectation d'une variable

Syntaxe

`nomvar = constante`	Ex : `x=1.23`
`nomvar = autre variable`	Ex : `x=y`
`nomvar = opération`	Ex : `x=y+3.2*z`

Fonctionnement

- Lit la valeur de toutes les variables à droite du signe `=` ;
- Effectue les opérations demandées ;
- Affecte le résultat à la variable à gauche de `=`.

Exemple

```
      i = i + 1
```

Augmente le contenu de la variable `i` de 1 (on dit aussi « *incrémenter* » une variable).

Opérateurs et fonctions mathématiques

Opérateurs arithmétiques

Tableau 5.1: Les opérateurs arithmétiques.	
Addition	`+`
Soustraction	`-`
Multiplication	`*`
Division	`/`
Puissance	`**`

Le tableau 5.1 donne la liste des opérateurs arithmétiques de Fortran. Ils sont listés par ordre de priorité croissante.

Dans une expression, on évalue donc d'abord `**` puis `/`, puis `*`, et enfin `+` et `-` .

On peut aussi grouper des sous-expressions entre parenthèses.

Exemples

```
x=a+b/c-d
```

évalue $a + \frac{b}{c} - d$.

```
x=(a+b)/(c+d)
```

évalue $\frac{a+b}{c+d}$.

Conversion de type dans une opération

Lorsque deux opérandes de types différents interviennent de chaque coté d'un opérateur :

1. l'opérande de type le plus faible est converti dans le type de l'autre opérande
2. l'opération est effectuée, et le type du résultat est le type le plus fort

Les types sont classés dans l'ordre suivant (du plus fort au plus faible) :

- **`complex`**
- **`double precision`**

- `real`
- `integer`
- `logical` [5.1]

Exemples

`b**2` sera du type de `b`.

`4*a*c` sera du type le plus fort de `a` et de `c`.

Si `a` et `c` sont `real`, et `b` `double precision`, le résultat de `b**2-4*a*c` sera `double precision`, mais attention, le produit `a*c` sera effectué en simple précision, d'où une perte de précision possible.

Pièges classiques

`2/3` sera du type `integer`. Autrement dit la division effectuée sera une division entière, et le résultat sera 0. Pour calculer effectivement deux tiers en réel, écrire : `2./3`, `2/3.` ou `2./3.`

Idem pour `i/j` où `i` et `j` sont deux variables `integer`. Écrire : `real(i)/j`

Conversion de type dans une affectation

> Lorsqu'une variable est affectée avec une expression de type différent, le résultat de l'expression est converti dans le type de la variable.

Exemples

`i=1.3456` affectera la variable `integer` `i` avec 1.

Attention :

`x=2/3` affectera la variable `real` `x` avec 0.0 !!!

Fonctions mathématiques

Une fonction FORTRAN est une boite dans laquelle rentre un ensemble de grandeurs d'un type donné (les *arguments*) et de laquelle sort une grandeur d'un type donné (cf 10.4).

Certaines fonctions mathématiques sont prédéfinies dans le langage (tables 5.2 et 5.3). On n'utilisera pas la même fonction selon le type de l'argument. Par exemple la fonction sinus sera `SIN` pour un argument réel, `DSIN` pour un argument double precision, `CSIN` pour un argument complexe.

Important :

- Le ou les arguments d'une fonction sont *toujours entre parenthèses*.
- Chaque fonction a son domaine de définition. Son non-respect entraîne une erreur d'exécution, c'est-à-dire que le programme s'arrête.

Exemples

```
      z=(a*sin(x)+b*cos(y)) / (a*sinh(x)+b*cosh(y))
```

Pour définir π en double precision :

```
      pi=4d0*datan(1d0)
```

Tableau 5.2: Fonctions arithmétiques prédéfinies.

`real`	`double precision`	`complex`	Fonction
SIN	DSIN	CSIN	$sin(x)$
COS	DCOS	CCOS	$cos(x)$
TAN	DTAN		$tg(x)$
ASIN	DASIN		$arcsin(x)$
ACOS	DACOS		$arccos(x)$
ATAN	DATAN		$arctg(x)$
SINH	DSINH		$sh(x)$
...	...		...
LOG10	DLOG10		$log_{10}(x)$
LOG	DLOG	CLOG	$ln(x)$
EXP	DEXP	CEXP	$exp(x)$

SQRT	DSQRT		$\sqrt{x}$

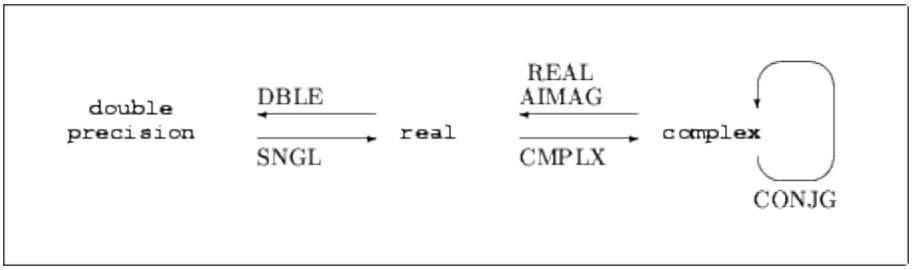
Tableau 5.3: Fonctions diverses prédéfinies.

`integer`	`real`	`double precision`	`complex`	Fonction
MAX0	AMAX1	DMAX1		*max*(*x*,*y*)
MIN0	AMIN1	DMIN1		*min*(*x*,*y*)
IABS	ABS	DABS	CABS	|*x*| ou |*z*|
INT	AINT	DINT		Partie entière
MOD	AMOD	DMOD		Reste dans la division entière

Fonctions de conversion

Ces fonctions permettent de convertir explicitement des données d'un type en un autre type. La figure 5.1 donne les noms des des différentes fonctions permettant à partir d'un type initial d'obtenir un autre type de valeur.

Figure 5.1: Fonctions de conversion de types.



Calcul en virgule flottante

La représentation des réels en virgule flottante entraîne fatalement des erreurs[5.2] :

- de représentation lors d'une affectation Elles viennent du fait qu'un nombre réel quelconque n'admet pas de représentation exacte en virgule flottante.
- d'arrondi dans les calculs. À la suite d'une opération arithmétique, le résultat est tronqué (c'est-à-dire que des chiffres significatifs sont effacés) pour pouvoir être codés en virgule flottante.

Ces erreurs peuvent être dramatiques, dans des algorithmes sensibles aux erreurs. Lors d'une opération simple, l'erreur effectuée est petite, mais si l'algorithme amplifie ces erreurs, le résultat peut être complètement faux.

Exemples

`1.+1e-8` sera codé comme 1.

`1d0+1d-16` sera codé comme 1d0

De nombreuses suites récurrentes exhibent les conséquences dramatiques des erreurs d'arrondi (voir TD).

Par exemple, la suite :

$$\begin{aligned} u_0 &= e-1 \\ u_n &= n.u_{n-1}-1 \end{aligned}$$

converge mathématiquement vers 0 mais selon la machine utilisée et la précision choisie, elle pourra converger vers $^{-\infty}$, ∞ ou 0 ou même ne pas converger du tout !

Manipulation de textes

Les textes sont stockés dans des chaînes de caractères. Dans ce chapitre, nous donnons quelques moyens de les manipuler.

Constantes chaînes

Elles sont constituées par une série de caractères encadrés par des apostrophes (ou « simple quotes » en anglais). Exemples :

```
'Ceci est une chaîne'
'/home/louisnar'
'L''apostrophe doit être doublé'
```

Variables chaînes

Syntaxe de déclaration :

```
character*n var
```

où *n* représente la longueur de la chaîne. Cette déclaration réserve *n* octets en mémoire pour y stocker *n* caractères.

Exemples :

```
      character*15 nom
      character*100 nomfichier
```

On peut ensuite affecter ces variables avec l'opérateur `=` comme pour toute autre variable :

```
      nomfichier='/usr/local/public/louisnar/th.mai'
      nom='Louisnard'
```

Comme la chaîne `'Louisnard'` ne contient que 9 caractères, les 6 derniers caractères de `nom` sont affectés avec le caractère blanc. Si on affecte une variable chaîne de 15 caractères avec une chaîne de 16 caractères, il y aura une erreur d'exécution.

Fonctions sur les chaînes

Longueur d'une chaîne

`LEN(chaine)` renvoie la longueur en mémoire de `chaîne`.

Attention : avec l'exemple précédent, `len(nom)` renvoie 15 (et pas 9 !) puisque la variable `nom` a été définie comme une chaîne de 15 caractères.

Recherche dans une chaîne

`INDEX(chaine1, chaine2)` renvoie la position de la chaine `chaine2` dans la chaine `chaine1`.

Par exemple (toujours à partir de l'exemple précédent), `index(nom,'nar')` renvoie 6.

Sous-chaînes

`chaine(m:n)` est la sous-chaîne allant du $m^{ième}$ ou $n^{ième}$ caractère de `chaine`. `m` et `n` peuvent être des constantes ou des variables entières.

Concaténation

La concaténation permet de coller deux chaînes bout à bout. En FORTRAN, cet opérateur se note `//`.

Par exemple :

```
'Bon'//'jour'
```

représente la chaîne suivante :

```
'Bonjour'
```

Attention (toujours avec l'exemple précédent) :

```
nom //'est mon nom'
```

représente la chaîne suivante :

```
'Louisnard ␣␣␣␣␣␣ est ␣ mon ␣ nom'
```

Entrées / Sorties

On appelle « *Entrées / Sorties* », tout ce qui permet à un programme de dialoguer avec l'extérieur :

- l'utilisateur via le clavier, l'écran, une imprimante, etc. ;

- les disques via des fichiers ;
- d'autres machines via le réseau ;
- ...

Le langage FORTRAN permet d'écrire ou de lire des données sur différentes choses.

Pour écrire, on utilise l'instruction **write** :

- à l'écran ,
- sur un fichier,
- dans une chaîne de caractères.

Pour lire, on utilise l'instruction **read** :

- sur le clavier,
- dans un fichier.
- dans une chaîne de caractères.

On distingue les lectures/écritures :

formatées
: c'est à dire organisée en lignes de caractères. C'est la cas des lectures clavier et des écritures écran, des lectures/écritures de fichiers texte et de chaînes.

non-formatées
: qui signifie transfert octet par octet. C'est le cas des lectures/écritures sur fichiers binaires.

Écriture formatée

Syntaxe

```
write(unité d'écriture, formatage) liste de données
```

- L'*unité d'écriture* est un entier (voir « fichiers » 11). Pour l'écran, on utilise `*`.
- Le formatage indique *sous quelle forme* on va écrire les données. Il existe une formatage par défaut qui laisse FORTRAN écrire comme il veut : le format `*`.

Exemples

```
      write(*,*) i,j,x,y
```

écrit les valeurs des variables `i,j,x,y` sur une ligne, séparées par des blancs.

```
      write(*,*) 'z vaut',z
```

écrit la chaine « `z vaut` », suivie de la valeur de la variable `z`.

Important : se souvenir qu'une instruction **write** écrit une ligne puis revient à la ligne. Donc un **write** est égal à une ligne d'affichage.

Formats d'écriture

Un format est une série de codes, chaque code définissant le format d'écriture d'un élément d'une donnée.

Définition du format

Deux solutions :

- Directement dans l'instruction **write** avec une chaîne de caractères :

```
write(*,'format') liste
```

- Dans une ligne labellée contenant l'instruction **format** :

```
nn format(définition du format)
write(*,nn)
```

- Cette solution permet d'utiliser le même format dans plusieurs instructions **write**.

Format entier

Dans un format, la notation `in` permet d'afficher un entier. L'entier est écrit sur *n* caractères en tout :

- S'il y a besoin de plus, l'écriture échoue.
- Si cela suffit, on ajoute éventuellement des blancs à gauche.

Exemples :

```
      i=11
      j=20312
      write(*,'(i6,i6)') i,j
```

donne

Variante avec l'instruction **format** :

```
10    format(i6,i6)
      i=11
      j=20312
      write(*,10) i,j
```

Format réel virgule flottante

Dans un format, la notation `en.m` permet d'afficher un réel en virgule flottante sur *n* caractères en tout avec *m* chiffres significatifs, c'est à dire :

$$\underbrace{\pm 0.\underbrace{\sqcup\sqcup\sqcup\ldots\sqcup}_{m\ caract.}E\pm\sqcup\sqcup}_{n\ caract.}$$

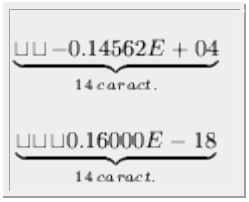
- S'il y a besoin de plus, l'écriture échoue.
- Si cela suffit, on ajoute éventuellement des blancs à gauche.

Pour que le format soit cohérent, il faut $n \geq m+7$.

Exemples :

```
      x=-1456.2
      y=1.6e-19
      write(*,'(e14.5,e14.5)') x,y
```

affiche



Format chaîne

Dans un format, la notation `an` permet d'afficher une chaîne de caractères.

- Si *n* est spécifié, la chaine est écrite sur *n* caractères en tout, en ajoutant des blancs à gauche pour compléter.
- Si *n* n'est pas spécifié, la chaine est écrite avec son nombre de caractères total (tel que déclaré !).

On peut ajouter un `$` après le `a` pour éviter de revenir à la ligne.

Exemples de formats mixtes

```
      i=36
      px=-1456.2
      write(*,'(a,i4,a,e12.5)')
     &  'i vaut', i, ' et x vaut', x
```

affichera

```
i vaut 36 et x vaut -0.14562E+04
```

En conclusion

Les formatages doivent être utilisés si c'est *absolument nécessaire*. Dans la plupart des cas le format par défaut (`*`)1 suffit largement, et il est inutile de perdre du temps a formater les sorties écran.

Lecture formatée

La lecture formatée s'applique

au clavier
aux fichiers texte

Principe

On lit une ligne de caractères d'un seul coup, la lecture étant validée par :

- la frappe de la touche RETURN pour une lecture clavier,
- une fin de ligne pour une lecture de fichier texte.

Les données sur une même ligne doivent être séparées par des blancs.

Syntaxe

```
read(unité de lecture, formatage) liste de variables
```

- L'unité de lecture est un entier (voir « fichiers » 11). Pour le clavier, on utilise `*`.
- Le formatage indique *sous quelle forme* on va lire les données (voir **write**).

Conseil

> Le plus simple est d'utiliser tout le temps le format libre `*`. **Exception**: pour lire des variables caractères ou chaînes de caractères, le format libre ne fonctionne pas. Utiliser le format chaîne `a`.

Exemple

```
      real a,b,c
      ...
      read(*,*) a,b,c
```

attend de l'utilisateur qu'il frappe trois réels au clavier séparés par des espaces puis la touche RETURN (↩). On peut entrer les nombres en format virgule flottante. Pour entrer (1,2) dans `a`,(1,6.10^{-19}) dans `b` et (32) dans `c`, l'utilisateur pourra taper :

```
1.2 1.6e-19 32
```

Un exemple classique d'écriture suivie d'une lecture sur la même ligne :

```
      write(*,'(a,$)') 'Entrez x :'
      read(*,*) x
```

Le message sera affiché, mais le curseur ne reviendra à la ligne que lorsque l'utilisateur aura entré x suivi de RETURN.

Contrôle de l'exécution

Un programme enchaîne les instructions qu'on lui donne une à une dans l'ordre. Pour réaliser un *vrai* programme, il faut tout de même disposer de moyens pour faire des tests et des boucles : on appelle cela le contrôle d'exécution.

Instructions conditionnelles

Objectif

Exécuter une séquence d'instructions si une condition logique est vérifiée, sinon en exécuter une autre.

Syntaxes

Pour exécuter une série d'instructions uniquement si une *condition logique* est vraie :

```
if (condition logique) then
    ...
    ...
endif
```

On peut aussi spécifier une autre série d'instructions à exécuter si la *condition logique* est fausse :

```
if (condition logique) then
    ...
    ...
else
    ...
    ...
endif
```

On peut même enchaîner plusieurs conditions logiques :

```
if (condition logique 1) then
    ...
    ...
else if (condition logique 2) then
    ...
    ...
else if (condition logique 3) then
    ...
    ...
else
    ...
    ...
endif
```

Le programme exécute le bloc d'instructions suivant la première condition logique vraie puis reprend après `endif`. Si aucune condition logique n'est vraie, le bloc `else` est exécuté.

Expressions logiques

Ce sont des objets de type `logical`.

Ils ne peuvent prendre que deux valeurs

`.true.`
`.false.`

Une expression logique est en général le résultat d'une comparaison entre deux objets :

En maths	En FORTRAN	En Anglais
$x = y$	`x.eq.y`	« equal »
$x \neq y$	`x.ne.y`	« not equal »
$x > y$	`x.gt.y`	« greater than »
$x < y$	`x.lt.y`	« less than »
$x \geq y$	`x.ge.y`	« greater or equal »
$x \leq y$	`x.le.y`	« less or equal »

On peut combiner ces expressions entre elles avec les opérateurs logiques usuels :

Ou	`.or.`
Et	`.and.`
Ou exclusif	`.xor.`
Négation	`.not.`

Exemples

Pour lire un caractère au clavier et agir en conséquence en tenant compte du fait que l'utilisateur peut répondre « Oui » en tapant `O` majuscule ou `o` minuscule, et idem pour non (avec `N` ou `n`) :

```
      character rep
      ...
      write(*,'(a,$)')
     &  'Répondez par (o)ui ou (n)on :'
      read(*,a) rep

      if (rep.eq.'0'.or.rep.eq.'o') then
        write(*,*) 'Vous répondîtes oui'
        ...
      else if (rep.eq.'N'.or.rep.eq.'n') then
        write(*,*) 'Vous répondîtes non'
        ...

      else
        write(*,*)
     &    'Vous répondîtes n''importe quoi'
        ...
      endif
```

Pour tester le signe du discriminant d'une équation du second degré :

```
      double precision a,b,c,delta,x1,x2
      ...
      delta=b**2-4*a*c
      x1=(-b-dsqrt(delta))/2/a
      x2=(-b+dsqrt(delta))/2/a

      if (delta.gt.0d0) then
        write(*,*) x1,x2

      else if (delta.eq.0d0) then
        write(*,*) x1

      else
        write(*,*) 'Pas de racines'
      endif
```

Variables `logical`

On peut déclarer des variables de ce type et leur affecter comme valeur :

- soit `.true.` ou `.false.`,
- soit le résultat d'une expression logique.

Cela sert parfois à améliorer la lisibilité des programmes.

Par exemple, pour l'équation du second degré :

```
      double precision a,b,c,delta,x1,x2
      logical une_racine
      logical deux_racines
      ...
      delta=b**2-4*a*c
      une_racine=(delta.eq.0d0)
```

```
      deux_racines=(delta.gt.0d0)

      if (deux_racines) then
        x1=(-b-dsqrt(delta))/2/a
        x2=(-b+dsqrt(delta))/2/a
        write(*,*) x1,x2
      else if (une_racine) then
        x1=-b/2/a
        write(*,*) x1
      else
        write(*,*) 'Pas de racines'
      endif
```

Boucles

Objectif

Une boucle permet d'exécuter une séquence d'instructions plusieurs fois d'affilée.

Le nombre de boucles peut être déterminé :

- à l'avance ,
- par le basculement d'une condition logique.

Boucles `do` ... `enddo`

On effectue la boucle un nombre de fois prédéterminé.

Syntaxes

```
do var = deb, fin
    ...
    ...
enddo
```

```
do var = deb, fin, pas
    ...
    ...
enddo
```

Fonctionnement

var est une variable de type **integer** et *deb*, *fin* et *pas* sont des objets de type **integer** (constantes ou variables).

La variable *var* prend d'abord la valeur de *deb* et est augmenté de *pas* à chaque boucle. Dès que *var* > *fin*, la boucle s'arrête et l'exécution continue après le **enddo**.

L'entier *pas* peut être omis et vaut 1 par défaut.

Si *fin*<*deb* et *pas*>0, la boucle n'est jamais exécutée.

Si *fin*>*deb* et *pas*<0, la boucle n'est jamais exécutée.

Exemples

Somme des premiers nombres entiers jusqu'à *n* :

```
... ! affectation de n
somme=0

do i=1,n
  somme=somme+i
enddo
```

ou bien encore :

```
... ! affectation de n
somme=0

do i=n,1,-1
  somme=somme+i
enddo
```

Somme des nombres impairs inférieurs à *n* :

```
... ! affectation de n
somme=0

do i=1,n,2
  somme=somme+i
enddo
```

Boucles do ... while

On effectue la boucle tant qu'une condition logique est vérifiée.

Syntaxe

```
do while (condition logique)
    ...
    ...
```

```
    ...
enddo
```

Fonctionnement

On rentre dans la boucle seulement si la condition logique vaut `.true.` et on exécute la boucle tant qu'elle reste à `.true.`.

Dès qu'à la fin d'une boucle, la condition est `.false.`, l'exécution reprend après `enddo`.

Remarque importante

Pour pouvoir sortir de la boucle, il faut que la condition logique puisse devenir `.true.` à l'intérieur. Si ce n'est pas le cas, le programme ne s'arrêtera jamais (Pensez-y !).

Exemples

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

```
integer n
double precision somme, epsilon
... ! affectation de epsilon
n=1
somme=0
do while (1d0/n**2 .ge. epsilon*somme)
  somme=somme + 1d0/n**2
  n=n+1
enddo
```

Plus élégant, en utilisant une variable `logical` :

```
integer n
double precision somme, epsilon
logical fini

... ! affectation de epsilon
n=1
somme=0
fini=.false
do while (.not. fini)
  somme=somme + 1d0/n**2
  n=n+1
  fini=(1d0/n**2 .lt. epsilon*somme)
enddo
```

Instructions `goto` et `continue`

> Il s'agit d'un archaïsme. À proscrire absolument, sauf si on ne peut pas faire autrement. Tout abus sera puni !

Les instructions de branchement conduisent à des programmes illisibles et difficiles à corriger. Certaines instructions du FORTRAN (voir « fichiers » 11) utilisent des branchements de manière implicite pour gérer des erreurs. Dans ce cas, et seulement dans celui-ci, le branchement est obligatoire.

Syntaxe

```
goto N° de label
```

En arrivant sur une telle ligne, le programme est branché directement sur la ligne comportant le label mentionné. En général, pour faire beau (il faut le dire vite...), cette ligne contient seulement l'instruction qui ne fait rien `continue`.

Exemple

Pour vous donner des mauvaises idées :

```
      character rep
      ...
10    continue
      write(*,*) 'Repondez oui ou non'
      read(*,*) rep

      if (rep.ne.'o'.and.rep.ne.'n') then
        goto 10
      endif
      ...
```

Les tableaux

A partir des types simples du FORTRAN, on peut former des vecteurs, des matrices, et même des tableaux à plusieurs indices.

Déclaration

```
type var(m1, m2, ...)
```

`m1`, *m2*,... déterminent la taille du tableau. Elles doivent être des *des constantes entières*. Il est interdit de mettre des variables entières. Autrement dit : la taille d'un tableau FORTRAN est fixée une fois pour toutes.

Exemples

```
      real v(100)
      double precision a(100,100)
      integer i(20)
```

Premier conseil

Pour créer des tableaux, il est conseillé de déclarer les tailles dans des constantes symboliques :

```
      parameter (max=100)
      double precision a(max,max)
      real v(max)
```

Second conseil

Avant de déclarer un tableau, pensez à la taille mémoire qu'il va occuper. Le tableau `a` ci-dessus occupe par exemple $100 \times 100 \times 8 = 80000$ octets.

Utilisation des tableaux

On y accède élément par élément en indiquant le ou les indices entre parenthèses séparés par des virgules. Les indices peuvent être des constantes ou des variables.

Exemples

Somme de deux matrices :

```
      double precision a(max,max)
      double precision b(max,max)
      double precision c(max,max)
      ...
      do i=1,max
        do j=1,max
          c(i,j)=a(i,j)+b(i,j)
```

```
      enddo
    enddo
```

Produit scalaire de deux vecteurs :

```
    double precision u(max), v(max)
    double precision prodsca
    ...
    prodsca=0
    do i=1,max
      prodsca=prodsca + u(i)*v(i)
    enddo
```

Instructions `read` et `write` avec boucles implicites

C'est une extension très pratique pour lire des matrices au clavier ou les écrire à l'écran. Il s'agit en quelque sorte d'une boucle `do` ... `enddo` combinée a un `read` ou un `write`.

Syntaxe

```
read(*,*) (var(i), i = i1, i2, i3)
write(*,*) (var(i), i = i1, i2, i3)
```

i représente une variable entière, *var(i)* une expression ou un tableau dépendant de cette variable *i*. *i1*, *i2* et *i3* ont le même sens que pour les boucles `do`.

Exemples

Lire les *n* premières composantes d'un vecteur sur une même ligne au clavier :

```
    read(*,*) (v(i), i=1,n)
```

Écrire les *n* premiers termes de la $j^{\text{ème}}$ ligne d'une matrice à l'écran, séparés par le caractère « `!` » :

```
    write(*,*) (a(i,j), ' ! ', j=1,n)
```

Utilisation optimale des tableaux

Problématique

La taille de déclaration des tableaux définit la taille mémoire réservée pour stocker le tableau. En général, on choisit cette taille comme étant la taille maximale du problème que l'on veut traiter, mais il se peut que pour un problème particulier, on utilise seulement une partie du tableau.

Dans l'exemple de l'addition, supposons que nous utilisions les tableaux FORTRAN `a(100,100)` et `b(100,100)` pour stocker des matrices $^{3\times 3}$. La structure de `a` sera la suivante :

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 & \cdots & 0 \\ a_{21} & a_{22} & a_{23} & 0 & \cdots & 0 \\ a_{31} & a_{32} & a_{33} & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 0 \end{bmatrix}$$

et de même pour `b`. Pour faire la somme des deux matrices, il est inutile d'additionner les `0`, et les boucles doivent être effectuées de 1 à 3 plutôt que de 1 à 100.

> Lorsqu'on utilise un tableau FORTRAN, il faut stocker ses dimensions réelles dans des variables en plus de sa taille de déclaration.

Dans l'exemple de l'addition précédent, on utilisera une variable `n` si on somme des matrices carrées, ou deux variables `nligne` et `ncolon` si on somme des matrices rectangulaires.

Exemple

Écrire un programme complet qui lit au clavier deux matrices de même taille $^{m\times n}$ et effectue leur somme. On part du principe que ces matrices ont au plus 50 lignes et 80 colonnes.

On déclare donc 3 tableaux 50×80, et on utilise deux variables `nligne` et `ncolon` pour stocker les tailles réelles des matrices que l'on traite. Ces tailles réelles sont bien sûr demandées à l'utilisateur.

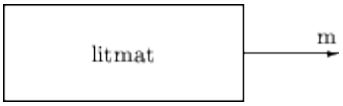
```
      parameter (mligne=50, mcolon=80)

      double precision a(mligne, mcolon)
      double precision b(mligne, mcolon)
      double precision c(mligne, mcolon)

      integer nligne, ncolon

      write(*,*)
     &  'Nombre de lignes des matrices'
      read(*,*) nligne
      write(*,*)
     &  'Nombre de colonnes des matrices'
      read(*,*) ncolon

      write(*,*) 'Matrice a '
      do i=1,nligne
        read(*,*) (a(i,j), j=1,ncolon)
      enddo

      write(*,*) 'Matrice b '
      do i=1,nligne
        read(*,*) (b(i,j), j=1,ncolon)
      enddo

      do i=1,nligne
        do j=1,ncolon
          (i,j)=a(i,j)+b(i,j)
        enddo
      enddo

      end
```

Fonctions et subroutines

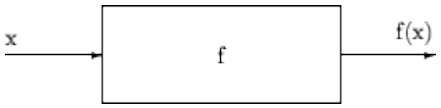
Premier objectif

Il arrive fréquemment que l'on doive faire plusieurs fois la même chose au sein d'un même programme, mais dans un contexte différent. Par exemple :

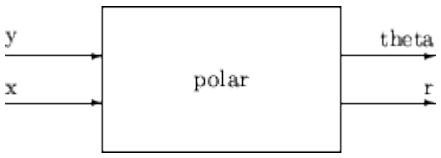
- Saisir des matrices au clavier :



- Calculer la fonction $f(x) = \arcsin(\log x / x)$ pour plusieurs valeurs de *x* :



- Calculer les coordonnées polaires (r, θ) d'un point défini par un couple de réels (*x*,*y*) :

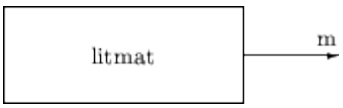


Second objectif

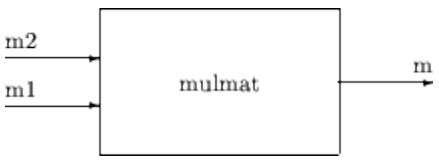
On a un problème décomposable en plusieurs sous problèmes. On cherche à « *enfermer* » chaque sous problème dans un bloc et à faire communiquer ces blocs.

Exemple : écrire un programme qui lit trois matrices au clavier, en fait le produit, puis affiche le résultat. On écrira trois blocs de base :

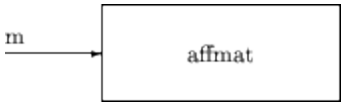
- un qui permet de lire une matrice au clavier :



- un qui effectue la somme de deux matrices :



- un qui affiche une matrice à l'écran :



Les objets FORTRAN correspondants à ces blocs sont les *subroutines* ou les *fonctions*.

On voit que chacun de ces blocs peut être écrit séparément, et qu'il est relié à l'extérieur par des « portes » d'entrée/sortie repérées par un nom. Persuadons-nous que ce nom n'a de sens que pour le bloc, et qu'il est là pour identifier une entrée ou une sortie.

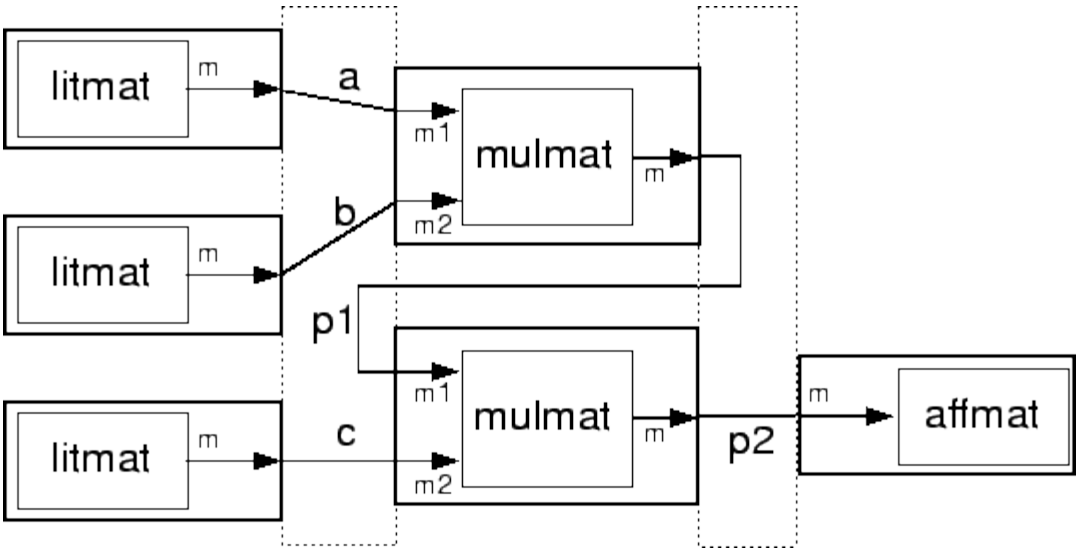
Les connexions des boites avec l'extérieur sont appelés en FORTRAN « *paramètres formels* », et seront traités comme des variables dans les instructions exécutables.

Avant d'aller plus loin, montrons comment utiliser les trois blocs définis ci-dessus pour résoudre le problème proposé.

- On va utiliser le bloc LITMAT 3 fois, et lui faire cracher 3 matrices *a*, *b*, *c*.
- On va utiliser MULMAT pour faire le produit de *a* et *b*, et mettre le résultat dans une variable *p*1
- on va réutiliser MULMAT pour faire le produit de *p*1 par *c* et mettre le résultat dans *p*2.
- on va utiliser AFFMAT pour afficher *p*2.

Symboliquement cela revient à connecter les blocs comme le montre la figure 10.1.

Figure 10.1: Connexion de blocs fonctionnels pour réaliser un programme.



Les subroutines

C'est une séquence d'instructions appelable d'un point quelconque du programme. Elle peut être appelée depuis le programme principal, ou depuis une autre subroutine ou fonction.

Une subroutine est définie par :

- un nom
- des paramètres formels, qui ont comme les variables :
  - un nom
  - un type

Écriture d'une subroutine

```
subroutine nomsub(pf1, pf2, pf3, ...)
type pf1
type pf2
type pf2
...
Déclaration des variables locales
...
Instructions exécutables
...
return
end
```

Les instructions de la subroutine peuvent manipuler :

- les paramètres formels (comme des variables normales),

- les variables locales,
- les variables d'un common.

Les instructions de la subroutine ne peuvent pas manipuler les variables locales du programme principal ou d'une autre subroutine ou fonction.

Appel d'une subroutine

L'appel d'une subroutine se fait depuis un bloc fonctionnel quelconque (programme principal, subroutine, fonction) avec l'instruction `call`.

```
call nomsub (v1, v2, v3, ...)
```

Les arguments `v1`, `v2`, `v3` peuvent être :

- des variables du bloc fonctionnel,
- des constantes (déconseillé !).

Très important

> Les types des arguments `v1`, `v2`, `v3`,... doivent correspondre exactement à ceux des paramètres formels `pf1`, `pf2`,`pf3`,...

Exemple

Écrire une subroutine qui calcule les coordonnées polaires associées à des coordonnées cartésiennes (x,y).

$$
\begin{array}{ll}
r = \sqrt{x^2 + y^2} & \\
\theta = \arctan|y/x| & x > 0 \\
\theta = -\arctan|y/x| & x < 0 \\
\theta = \pi/2 & x = 0 \text{ et } y > 0 \\
\theta = -\pi/2 & x = 0 \text{ et } y < 0
\end{array}
$$

Figure 10.2: Code source d'une subroutine de calcul de coordonnées polaires à partir de coordonnées cartésiennes.

```
      subroutine polar(x, y, r, theta)

      double precision x, y, r, theta

      double precision pi
      double precision temp

      pi=4*datan(1d0)

      r=dsqrt(x**2+y**2)

      temp=datan(y/x)
      if (x.gt.0d0) then
        theta=temp
      else if (x.lt.0d0) then
        theta=-temp
      else if (x.eq.0d0) then
        if (y.ge.0d0) then
          theta=pi/2
        else
          theta=-pi/2
        endif
      endif

      return
      end
```

Le figure 10.2 donne un code source possible de cette subroutine.

Un exemple d'appel de cette subroutine dans le programme principal :

```
      programm
      ...
      double precision a, b, rho, phi
      ...
      call polar (a, b, rho, phi)
      ...
      end
```

Remarques très importantes

- Les variables `a`, `b`, `rho` et `phi` sont des variables locales du programme principal : elles n'ont aucun sens pour la subroutine `polar`.
- Les variables `pi` et `temp` sont des variables locales de la subroutine `polar` : elles n'ont aucun sens pour le programme principal.
- `x`, `y`, `r` et `theta` sont les paramètres formels de la subroutine `polar`. Ce sont les portes de communication de la subroutine et leurs noms n'ont aucun sens à l'extérieur de la subroutine.

En particulier, s'il y a dans le programme principal des variables locales appelées `x`, `y`, `r` ou `theta`, *elles n'ont rien à voir avec les paramètres formels de la subroutine*.

En revanche, il est possible de passer ces variables locales en tant qu'arguments d'appel à `polar` :

```
      call polar (x, y, rho, phi)
```

La variable locale `x` du programme principal est alors passée à la subroutine via son premier paramètre formel, qui incidemment s'appelle aussi `x`, mais les deux objets sont bien distincts.

Exercice 1

Réécrire le programme de résolution des équations du second degré avec des subroutines.

Exercice 2

Dans le programme de la figure 10.3, il y a trois erreurs ô combien classiques. Corrigez-les, puis répondez ensuite aux questions suivantes :

- Quelles sont les variables locales
  - du programme principal ?
  - de la subroutine ?
- Pourquoi n'a-t-on pas utilisé une variable `integer` pour coder $n!$?
- Pourquoi ne pas utiliser la subroutine factor pour calculer les C_n^p ?

Complétez ensuite les subroutines pour qu'elles calculent effectivement $n!$ et C_n^p.

Figure 10.3: Programme de calcul de $n!$ et de C_n^p (à compléter).

```
      program factcnp

      do i=1,100
        call factor(i, facti, ifaux)
        write(*,*) i, facti
      enddo

      write(*,*)
     &  'Entrez deux entiers i et j'
      read(*,*) i, j
```

```
      call calcnp (i, j, cij)
      end

      subroutine factor(n, fact, ierr)

      integer n, ierr
      double precision fact
      ...
      return
      end

      subroutine calcnp (n, p, cnp, ierr)
      implicit double precision (a-h,o-z)
      integer n, p, cnp
      ...
      return
      end
```

Les fonctions

Une fonction est en tout point identique à une subroutine, mais son nom contient en plus une valeur. C'est-à-dire qu'au lieu de l'appeler par `call`, on l'utilise à droite du signe `=` dans une instruction d'affectation.

On avait déjà vu les fonctions prédéfinies du FORTRAN, par exemple `datan`, qui n'a qu'un paramètre formel :

```
      pi=4*datan(1d0)
```

Valeur de retour

Puisque le nom de la fonction contient une valeur, cette valeur doit être typée. Par exemple `datan` renvoie une valeur **`double precision`**. En plus du type de ses paramètres formels, la définition d'une fonction doit donc décrire le type de la valeur qu'elle retourne.

La valeur de retour de la fonction sera affectée dans le corps de la fonction, comme si le nom de la fonction était une variable ordinaire.

Écriture d'une fonction

```
type function nomfonc (pf1, pf2, ...)
type pdf1
type pdf2
```

```
...
déclarations des variables locale
...
instructions exécutables
! devrait contenir quelque chose comme : nomfonc = ...
...
return
end
```

Le type peut être omis auquel cas le type de la fonction est fixé par les règles par défaut.

Utilisation d'une fonction

Une fonction est utilisable dans tout autre bloc fonctionnel, comme une variable qui aurait des arguments. Comme pour les subroutines, *il est indispensable de respecter la correspondance entre les arguments passés à la fonction et ses paramètres formels*.

Attention : il faut non seulement déclarer le type de la fonction lors de sa définition, mais aussi dans tous les blocs fonctionnels où on l'utilise. Si cette déclaration est absente, les règles de typage automatiques du bloc fonctionnel courant s'appliquent.

Exemple 1

Fonction qui calcule le rayon-vecteur $r = \sqrt{x^2 + y^2}$ associé à un couple (x,y) : figure 10.4.

Figure 10.4: Calcul de $r = \sqrt{x^2 + y^2}$ par une fonction.

```
      program test

      double precision abscis, ordonn, r
      double precision rayon

      read(*,*) abscis, ordonn
      r=rayon(abscis, ordonn)
      write(*,*) r
      end

      double precision function rayon (x, y)

      double precision x, y

      rayon=dsqrt(x**2+y**2)
```

```
      return
      end
```

Exemple 2

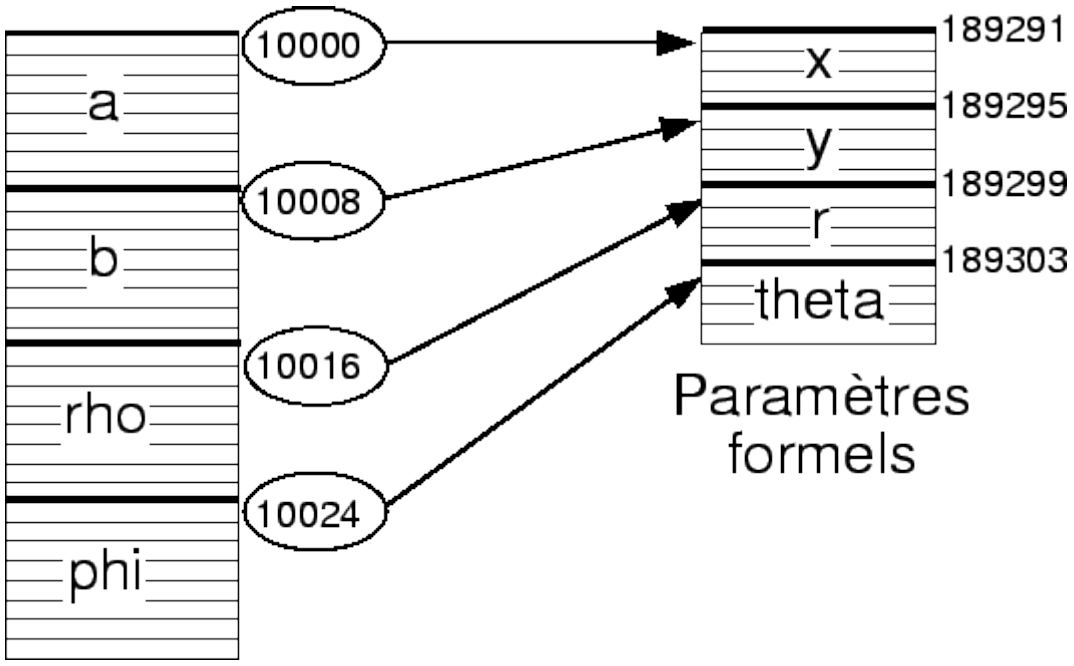
Fonction qui saisit un caractère au clavier : figure 10.5.

Figure 10.5: Fonction de lecture d'un caractère au clavier.

```
      program test
      character c, litcar

      do while (litcar() .ne. 'q')
        write(*,*) 'On continue'
      enddo
      end

      character function litcar()
      read(*,*) litcar
      return
      end
```

Mécanisme de passage des arguments à une subroutine ou une fonction

Figure 10.6: Passage des paramètres dans l'exemple de la fonction `polar`.



Les variables locales des divers blocs fonctionnels sont stockées à une adresse mémoire fixée par le compilateur. Cette adresse mémoire est un grand entier, qui est « le numéro de bureau » de la variable.

Certains bureaux sont plus grands que d'autres. Une variable `integer` ou `real` occupera un bureau à 4 cases, une variable `double precision` un bureau à 8 cases, un vecteur de 100 `real`, un bureau de 4×100 cases, une matrice de 50×50 `double precision` un bureau de $50 \times 50 \times 8$ cases.

Mais dans tous les cas, l'adresse de la variable est l'adresse de la première de ces cases.

A un paramètre formel de subroutine ou de fonction est associé en mémoire un nombre de cases suffisant pour stocker une adresse (4 octets sous UNIX). Lors d'un appel à une subroutine ou fonction, *l'adresse du premier argument* est écrite à l'emplacement réservé au premier paramètre formel, idem pour le second, le troisième, etc.

La figure 10.6 illustre ceci dans l'exemple de la subroutine `polar`.

Lorsque dans le corps de la subroutine, le programme rencontrera le paramètre formel `x` de type double precision à droite de `=`, il ira lire 8 octets à partir de l'adresse contenue dans `x` (de 10000 à 10007), c'est-à-dire la valeur de `a`.

Lorsqu'il rencontrera une affectation du paramètre formel `theta`, il ira écrire les 8 octets à partir de l'adresse contenue dans `theta` (de 10024 à 10031), c'est-à-dire `phi`.

D'où l'importance de respecter le nombre et le type d'arguments.

L'instruction `common`

On a vu que par défaut les variables d'un bloc fonctionnel lui étaient locales, donc inconnues des autres blocs. Il existe un moyen d'étendre la portée d'une variable à plusieurs blocs fonctionnels : le `common`.

Un `common` comporte un nom et une liste de variables. Les variables de cette liste seront connues dans tous les blocs fonctionnels où l'on écrit le `common`.

Syntaxe

```
common/nomcom/v1, v2, v3...
```

Remarques

- Le `common` ne dispense pas des déclarations.
- On ne peut mettre des constantes déclarées par `parameter` dans les `common`s (en particulier les tailles de tableaux)
- On ne peut mettre la même variable dans deux `common`s différents.
- On ne peut mettre un paramètre formel dans un `common`.
- On peut passer des tableaux en `common`, à condition de les déclarer de la même longueur partout.
- Tout ce que vérifie le compilateur, c'est que la longueur totale en octets de chaque `common` est la même dans tous les blocs fonctionnels où il apparaît. On a donc le droit de changer le nom des variables entre deux utilisations d'un même `common`, mais cela est déconseillé.

Exemples

Figure 10.7: Exemple d'utilisation de l'instruction `common`.

```
      program test

      double precision pi
```

```
      real a, b

      common /trig/ pi
      common /bidon/ a, b

      pi=4*datan(1d0)
      ...
      end

      subroutine truc (x,y)

      common /trig/ pi
      common /bidon/ u, v

      double precision pi
      real u, v
      ...
      y=x*tan(pi*u/v)
      ...
      return
      end
```

Dans l'exemple de la figure 10.7, on voit que les noms des variables du **common** bidon sont différentes dans le programme principal et dans truc.

Mais cela est correct puisqu'il y a 2 **real**s de chaque coté. u et a représentent exactement le même objet car ils correspondent à la même zone mémoire. Cela dit, il vaut mieux garder les mêmes noms partout.

Paramètres formels de type tableau

On peut passer des tableaux à des subroutines ou fonctions si celles-ci sont conçues pour les recevoir. Comment déclare-t-on des paramètres formels de type tableaux ? Cela dépend du nombre d'indices.

vecteur v (1 indice)
: on n'a pas besoin de la taille de déclaration du vecteur qui arrivera par le paramètre v :

```
subroutine sub (v, ...)
type v(*)
```

matrice m (2 indices)
: il faut la première taille de déclaration de la matrice et donc il faut prévoir un paramètre formel pour cette constante :

```
subroutine sub (a, mligna, ...)
type a (mligna, *)
```

Ces déclarations s'appliquent uniquement aux paramètres formels, qui rappelons-le ne sont que

des portes de communication pour les subroutines et les fonctions. En aucun cas une variable ne pourra être déclarée avec une `*`. Une variable de type tableau (rappel) doit toujours être déclarée avec des constantes entières

Exercice

Au vu du mécanisme de passage des arguments aux subroutines, expliquez pourquoi les paramètres formels de type vecteurs et matrices sont déclarés de cette manière.

Exemple

Subroutine qui lit une matrice au clavier. La subroutine devra sortir la matrice, son nombre de lignes réelles, son nombre de colonnes réelles (figure 10.8).

Il faut bien comprendre que la seule matrice ayant une existence réelle est la matrice `mat` du programme principal, et que pour lui réserver de la mémoire, il faut la déclarer avec un nombre de lignes et un nombres de colonnes explicites.

Figure 10.8: Lecture d'une matrice au clavier.

```
      program test
      parameter(mligne=10, mcolon=20)
      double precision mat (mligne, mcolon)

      call litmat (mat, mligne, nligne, ncolon)

      end

      subroutine litmat (a, ma, nl, nc)

      double precision a (ma, *)

      write(*,*)
     &  'Entrez nombre de lignes-colonnes'
      read(*,*) nl, nc

      do i=1, nl
        write(*,*) 'Ligne ', i
        read(*,*) (a(i,j), j=1,nc)
      enddo

      return
      end
```

Le paramètre formel `a` de la subroutine va recevoir l'adresse de `mat` au moment de l'appel, et connaissant sa première taille de déclaration (10) via le paramètre formel `ma`, elle sera à même de lire et écrire un élément quelconque de `mat`.

Exercice 1

Que se passe-t-il si l'utilisateur tape un nombre de lignes supérieur à 10 et/ou un nombre de colonnes supérieur à 20 ?

Exercice 2

Écrire une subroutine qui affiche une matrice à l'écran, et combinez-la avec `litmat` pour vérifier votre réponse à la question précédente.

Déclaration `external`

Objectif

Utiliser le nom d'une fonction ou d'une subroutine comme argument d'une autre fonction ou subroutine.

Quelle drôle d'idée ?

Examinons le problème suivant : Écrire une subroutine qui calcule l'intégrale $\int_a^b f(x)\,dx$ pour une fonction *f quelconque*. Que faudra-t-il faire entrer et sortir de la subroutine ?

- **la fonction *f*** ;
- les bornes d'intégration *a* et *b* ;
- la valeur de l'intégrale.

Or quel est le moyen en FORTRAN de programmer une fonction *f*(*x*) ? C'est d'utiliser une fonction FORTRAN, qui renverra par exemple une valeur `real`.

On peut donc prédire la forme de la subroutine d'intégration :

```
      subroutine integ (a, b, f, valint)
```

Les paramètres formels `a`, `b` et `valint` seront **`double precision`**, mais de quel type est `f` ?

C'est le nom d'une fonction FORTRAN, et pour déclarer un paramètre formel aussi bizarre, on écrira :

```
      external f
```

et aussi

```
      real f
```

car `f` renvoie une valeur **`real`**. On peut aussi déclarer des paramètres formels de type subroutine, en utilisant simplement **`external`**. Dans le corps de la subroutine, on fera bien sûr appel à cette fonction `f` pour calculer l'intégrale. Pas de difficulté ! On fait comme si elle existait et on écrira des choses du style :

```
      valint = valint + h/2 *(f(x1+h) + f(x1))
```

Maintenant ma subroutine d'intégration est écrite. Je souhaite l'appliquer à une fonction que j'ai écrite en FORTRAN, du style :

```
      real function truc(x)
      real x
      ...
      truc=...
      return
      end
```

Je veux appeler la subroutine `integ` pour calculer l'intégrale de cette fonction entre disons 1 et 2. J'écrirai :

```
      call integ (1.0, 2.0, truc, somtruc)
```

`1.0` et `2.0` sont des constantes **`real`**, `somtruc` une variable **`real`** qui me renverra la valeur de l'intégrale...

Et `truc` ? C'est le nom d'une fonction FORTRAN. Ai-je le droit de passer ce genre de chose en argument à une subroutine ? La réponse est oui, si je la déclare :

```
      external truc
```

dans le bloc fonctionnel d'où je fais le **`call`**. De plus, comme cette fonction renvoie une valeur **`real`**, j'ajouterai :

```
      real truc
```

Récapitulation

La figure 10.9 récapitule tout ce que l'on vient d'expliquer.

Figure 10.9: Structure générale d'un programme d'intégration simple.

```
      program test

      real somtruc

      external truc
      real truc

      call integ (1.0, 2.0, truc, somtruc)

      end


      subroutine integ (a, b, f, valint)

      real a, b, valint
      external f
      real f
      ...
      valint=valint + ( f(x1+h) + f(x1) )*h/2
      ...
      return
      end


      real function truc(x)
      real x
      ...
      truc=...
      return
      end
```

Remarquons que la structure de `truc` est imposée par la subroutine qui demande que la fonction représentée par son paramètre formel `f` ait un argument réel et renvoie une valeur réelle. Nous ne pourrions donc pas par exemple déclarer :

```
      real function truc(x,y)
```

En général, les subroutines du type `integ` sont des boites noires toutes faites (par des spécialistes), et on vous indique juste le type de ses paramètres

formels. Lorsque l'un d'entre eux est une fonction ou une subroutine, on vous indique en plus la liste des paramètres formels que doit avoir cette fonction ou subroutine.

Exercice

Écrire la structure d'un programme (programme principal / subroutine / fonctions) pour trouver les zéros d'une fonction $f(x)$ par la méthode de Newton. On rappelle que cette méthode nécessite la connaissance de la fonction $f(x)$ et de sa dérivée $f'(x)$.