Rappels de programmation Fortran

Violaine Louvet[1], très inspirée du cours de l'IDRIS réalisé par Anne Fouilloux et Patrick Corde

[1]CNRS/ICJ

Rabat, 9-12/04/2012

1. Généralités
2. Déclarations & variables
3. Opérateurs et expressions
4. Structures de contrôles
5. Tableaux
6. Gestion de la mémoire
7. Types dérivés et modules
8. Procédures
9. Entrées-Sorties
10. Fonctions intrinsèques
11. Pointeurs
12. Travaux pratiques

1 Généralités
- Unité de programme
- Exécution
- Syntaxe
- Exercice

2 Déclarations & variables

3 Opérateurs et expressions

4 Structures de contrôles

5 Tableaux

6 Gestion de la mémoire

7 Types dérivés et modules

8 Procédures

9 Entrées-Sorties

10 Fonctions intrinsèques

11 Pointeurs

12 Travaux pratiques

Historique

- Langage compilé (par opposition à interprété)
- Evolution :
  - 1958 : Fortran II
  - 1966 : Fortran IV
  - 1977 : Fortran 77
  - 1994 : Norme Fortran 90
  - 1997 : Norme Fortran 95
  - 2004 : Norme Fortran 2003
  - 2010 : Norme Fortran 2008

Contenu du cours

Nous verrons ici essentiellement les éléments de la norme 2003. La norme 2008 introduit notamment des notions avancées de programmation objet qui ne seront pas abordées.

Un programme source Fortran est composé de parties indépendantes appelées unités de programme (scoping unit) :

- le programme principal
- les sous-programmes :
  - de type subroutine
  - de type function
- les modules
- les block data

Organisation des unités de programme

- Chaque unité comprend une partie déclarative (déclaration des variables locales, ...) suivie d'une partie comportant des instructions exécutables.
- Idéalement, elles peuvent être dans des fichiers séparés.

Compilation / Exécution

Rappel : compilation/édition de liens

- On compile les fichiers contenant les différentes unités de programme
- Eventuellement, on crée une bibliothèque avec les fichiers objets générés
- L'éditions de liens permet de générer l'exécutable à partir des fichiers objet ou des bibliothèques créés à la compilation

Exécution

- Processus (en anglais, process) : programme en cours d'exécution :
  - Un ensemble d'instructions à exécuter (un programme)
  - Un espace d'adressage en mémoire vive

Processus / Thread

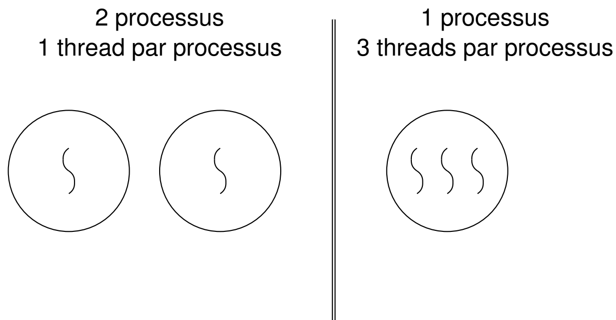
Processus

- Un processus a son propre espace mémoire, sa propre pile qu'il ne partage pas avec les autres
- les processus sont plus lourds que les threads à créer : en général, 100 fois plus rapides à créer qu'un processus
- les processus sont réellement indépendants

Processus léger / Thread

- Les threads partagent le même espace mémoire
- les threads peuvent se partager des informations facilement
- les threads doivent eux-mêmes faire attention à ne pas se marcher sur les pieds : il n'y a pas de protection entre les threads d'un même processus
- Les threads ont peu d'informations propres

Processus / Thread



Contexte

On se place ici dans le cadre d'une programmation séquentielle : un seul processus. Les prochains cours aborderont des concepts différents :

- Plusieurs processus
- Plusieurs threads
- voir les deux à la fois

Eléments syntaxiques

- En format libre, les lignes peuvent être de longueur quelconque, maximale de 132 caractères.
- Depuis Fortran 90, on oublie le format fixe !
- On peut coder plusieurs instructions sur la même ligne en les séparant avec « ; »
- On peut coder une instruction sur plusieurs lignes en utilisant le caractère de continuation « & »
  - Cas particulier de la chaîne de caractères :

```
print *, "Discretization error &
         &for poisson solver : ",err
```

- Le caractère « ! » indique que ce qui suit est un commentaire

Exercice

```
PROGRAM MAIN ;INTEGER : :degreesfahrenheit&
 ,degreescentigrade ;READ∗,&
 degreesfahrenheit ;degreescentigrade&
 =5∗(degreesfahrenheit −32)/9 ;PRINT∗,&
 degreesCENtiGrAde ;END
```

- Reformatter ce code correctement (de façon lisible)
- Compiler, exécuter
- Que fait-il ?

1 Généralités

2 Déclarations & variables
- Identificateurs
- Types du fortran
- Précision des nombres
- Kind
- Chaîne de caractères
- Implicit none
- Constantes et initialisations
- Exercice

3 Opérateurs et expressions

4 Structures de contrôles

5 Tableaux

6 Gestion de la mémoire

7 Types dérivés et modules

8 Procédures

9 Entrées-Sorties

10 Fonctions intrinsèques

11 Pointeurs

12 Travaux pratiques

Identificateurs

- Donne un nom à :
  - une variable
  - une constante
  - une procédure
- Règles à suivre :
  - suite de caractères alphanumériques (lettres non accentuées, chiffres, underscore)
  - une lettre en premier caractère
  - longueur limitée à 31 caractères
  - pas de distinction majuscule/minuscule (case insensitive)

Exemples

```
Matrix
matrix
mesh_size
nx0
```

Types du fortran

Le type d'une variable détermine :

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

Types prédéfinis

INTEGER	entier
CHARACTER	caractère
LOGICAL	deux valeurs, .TRUE. ou .FALSE.
REAL	réel simple précision
DOUBLE PRECISION	réel double précision
COMPLEX	complexe simple précision

Précision des nombres

Simple précision	4 octets	7 chiffres décimaux significatifs
Double précision	8 octets	15 chiffres décimaux significatifs
Quadruple précision	16 octets	34 chiffres significatifs

- Les types prédéfinis en Fortran 90 sont en fait des noms génériques renfermant chacun un certain nombre de variantes ou sous-types que l'on peut sélectionner à l'aide du paramètre KIND lors de la déclaration des objets.
- Ce paramètre est un mot-clé à valeur entière. Cette valeur désigne la variante souhaitée pour un type donné.
- Les différentes valeurs du paramètre KIND sont dépendantes du système utilisé. Elles correspondent, en général, au nombre d'octets désirés pour coder l'objet déclaré.

Rappel d'arithmétique flottante

- Les nombres réels sont représentés de façon approximative en mémoire (représentation en virgule flottante), avec la convention standardisée de la forme $m \times 2^e$, où m est la mantisse, $1 \leq m \leq 2$ et e l'exposant.
- Certains réels sont par définition impossibles à représenter en numération classique : $1/3$, π ...
- La représentation en un nombre fixe d'octets oblige le processeur à faire appel à des approximations afin de représenter les réels.
- Le degré de précision de la représentation par virgule flottante des réels est directement proportionnel au nombre de bits alloué à la mantisse, alors que le nombre de bits alloué à l'exposant conditionnera l'amplitude de l'intervalle des nombres représentables.
- Plusieurs propriétés de l'arithmétique (associativité, distributivité,...) ne sont plus valides en arithmétique flottante !

Utilisation de KIND

Exemples

```
real(kind=8)      :: x  ! reel double precision
integer(kind=2)   :: i  ! entier code sur 2 octets, integer normal sur 4
```

- On peut indiquer le sous-type désiré lors de l'écriture des constantes en les suffixant (pour les constantes numériques) ou de les préfixant (pour les constantes chaînes de caractères) par la valeur du sous-type à l'aide du caractère « _ ».

Exemples

```
n = 23564_4       ! entier sur 4 octets
x = 12.87976543245_8   ! reel sur 8 octets
```

Fonction KIND

- Renvoie la valeur entière qui correspond au sous-type de l'argument spécifié

Exemples

```
kind(1.0)      ! valeur associee au sous-type reel simple precision
kind(1.0d0)  ! valeur associee au sous-type reel double precision
real(kind=kind(1.d0))     :: x   ! declare x comme reel double precision
                                 ! quelle que soit la machine utilisee
```

Remarque

Les types définis via cette fonction KIND sont assurés d'être portables au niveau de la compilation.

Chaîne de caractères

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

Exemples

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

Manipulation de chaînes

- LEN_TRIM(string) : longueur (entier) de la chaîne débarrassée de ses blancs de fin
- TRIM(string) : débarrasse string de ses blancs de fin
- LGE(string_a, string_b) : VRAI si string_a « supérieure ou = » à string_b (Lexically Greater or Equal), et les variantes LGT, LLE, LLT (on peut aussi utiliser >=, >,<=, <).
- ADJUSTL(string) : débarrasse string de ses blancs de tête (cadrage à gauche) et complète à droite par des blancs. Idem avec ADJUSTR à droite.

Instruction IMPLICIT NONE

- Par défaut, les variables dont l'identificateur commence par les caractères I à N sont de type INTEGER.
- Toutes les autres sont de type REAL.
- L'instruction IMPLICIT NONE change cette règle car elle impose à l'utilisateur la déclaration de chaque variable.
- Cette instruction est vivement recommandée car elle permet la détection d'un certain nombre d'erreurs à la compilation.
- IMPLICIT NONE se place avant les déclarations des variables.
- L'instruction ne s'applique qu'à l'unité de programme qui la contient.

Constantes littérales

- Une constante réelle simple précision 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
- Une constante double precision doit obligatoirement être écrite en virgule flottante, le E étant remplacé par un D
- Une constante de type COMPLEX est obtenue en combinant deux constantes réelles entre parenthèses séparées par une virgule
- Une constante chaîne de caractères est une suite de caractères encadrée par le délimiteur « ' » ou « " »

Exemples

```
x = 0.
y = 1.0
z = -1.6e-04
d = 2.7d-3
zero = 0.d0
cplx = (1.32e-3, 4.3)
CHARACTER (LEN=10) :: ch
ch = "Bonjour" ; ch (4:7) = "soir"
```

Initialisations

- Une initialisation pourra s'effectuer au moyen de l'instruction DATA
  - les variables initialisées par ce moyen héritent de l'attribut SAVE : leur emplacement mémoire est permanent
- Fortran permet d'initialiser une variable lors de sa déclaration à l'aide du symbole « = »
  - Dans ce cas, les caractères :: sont obligatoires
  - Ces variables héritent aussi de l'attribut SAVE

Exemples

```
REAL      a,b,c
INTEGER   n
DATA      a,b,n/1.0, 2.0, 17/
LOGICAL   :: flag = .TRUE.
```

Constantes symboliques

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

Exemples

```
INTEGER, PARAMETER      ::  n = 1000
DOUBLE PRECISION        ::  PI, ZERO
PARAMETER (PI = 3.14159265d0, ZERO = 0.d0)
```

Ces déclarations sont-elles ou non correctes ? Pourquoi ?

- ReAl : : x
- CHARACTER : : name
- CHARACTER(LEN=10) : : name
- REAL : : var-1
- INTEGER : : 1a
- BOOLEAN : : loji
- DOUBLE : : X
- CHARACTER(LEN=5) : : town = "Glasgow"
- CHARACTER(LEN=*) : : town = "Glasgow"
- CHARACTER(LEN=*), PARAMETER : : city = "Glasgow"
- INTEGER : : pi = +22/7
- LOGICAL : : wibble = .TRUE.
- CHARACTER(LEN=*), PARAMETER : : "Bognor"
- REAL, PARAMETER : : pye = 22.0/7.0
- REAL : : two_pie = pye*2
- REAL : : a = 1., b = 2
- LOGICAL(LEN=12) : : frisnet
- CHARACTER(LEN=6) : : you_know = 'y'know"
- CHARACTER(LEN=6) : : you_know = "y'know"
- INTEGER : : ia ib ic id
- DOUBLE PRECISION : : pattie = +1.0D0
- DOUBLE PRECISION : : pattie = -1.0E-0
- REAL : : poie = 4.*atan(1.)

Ces déclarations sont-elles ou non correctes ? Pourquoi ?

- ReAl : : x OK
- CHARACTER : : name
- CHARACTER(LEN=10) : : name
- REAL : : var-1
- INTEGER : : 1a
- BOOLEAN : : loji
- DOUBLE : : X
- CHARACTER(LEN=5) : : town = "Glasgow"
- CHARACTER(LEN=*) : : town = "Glasgow"
- CHARACTER(LEN=*), PARAMETER : : city = "Glasgow"
- INTEGER : : pi = +22/7
- LOGICAL : : wibble = .TRUE.
- CHARACTER(LEN=*), PARAMETER : : "Bognor"
- REAL, PARAMETER : : pye = 22.0/7.0
- REAL : : two_pie = pye*2
- REAL : : a = 1., b = 2
- LOGICAL(LEN=12) : : frisnet
- CHARACTER(LEN=6) : : you_know = 'y'know"
- CHARACTER(LEN=6) : : you_know = "y'know"
- INTEGER : : ia ib ic id
- DOUBLE PRECISION : : pattie = +1.0D0
- DOUBLE PRECISION : : pattie = -1.0E-0
- REAL : : poie = 4.*atan(1.)

Ces déclarations sont-elles ou non correctes ? Pourquoi ?

- ReAl : : x OK
- CHARACTER : : name OK
- CHARACTER(LEN=10) : : name
- REAL : : var-1
- INTEGER : : 1a
- BOOLEAN : : loji
- DOUBLE : : X
- CHARACTER(LEN=5) : : town = "Glasgow"
- CHARACTER(LEN=*) : : town = "Glasgow"
- CHARACTER(LEN=*), PARAMETER : : city = "Glasgow"
- INTEGER : : pi = +22/7
- LOGICAL : : wibble = .TRUE.
- CHARACTER(LEN=*), PARAMETER : : "Bognor"
- REAL, PARAMETER : : pye = 22.0/7.0
- REAL : : two_pie = pye*2
- REAL : : a = 1., b = 2
- LOGICAL(LEN=12) : : frisnet
- CHARACTER(LEN=6) : : you_know = 'y'know"
- CHARACTER(LEN=6) : : you_know = "y'know"
- INTEGER : : ia ib ic id
- DOUBLE PRECISION : : pattie = +1.0D0
- DOUBLE PRECISION : : pattie = -1.0E-0
- REAL : : poie = 4.*atan(1.)

Ces déclarations sont-elles ou non correctes ? Pourquoi ?

- ReAl : : x OK
- CHARACTER : : name OK
- CHARACTER(LEN=10) : : name OK
- REAL : : var-1
- INTEGER : : 1a
- BOOLEAN : : loji
- DOUBLE : : X
- CHARACTER(LEN=5) : : town = "Glasgow"
- CHARACTER(LEN=*) : : town = "Glasgow"
- CHARACTER(LEN=*), PARAMETER : : city = "Glasgow"
- INTEGER : : pi = +22/7
- LOGICAL : : wibble = .TRUE.
- CHARACTER(LEN=*), PARAMETER : : "Bognor"
- REAL, PARAMETER : : pye = 22.0/7.0
- REAL : : two_pie = pye*2
- REAL : : a = 1., b = 2
- LOGICAL(LEN=12) : : frisnet
- CHARACTER(LEN=6) : : you_know = 'y'know"
- CHARACTER(LEN=6) : : you_know = "y'know"
- INTEGER : : ia ib ic id
- DOUBLE PRECISION : : pattie = +1.0D0
- DOUBLE PRECISION : : pattie = -1.0E-0
- REAL : : poie = 4.*atan(1.)

Ces déclarations sont-elles ou non correctes ? Pourquoi ?

- ReAl : : x OK
- CHARACTER : : name OK
- CHARACTER(LEN=10) : : name OK
- REAL : : var-1 Non, on ne peut pas avoir « -1 » dans un identificateur
- INTEGER : : 1a
- BOOLEAN : : loji
- DOUBLE : : X
- CHARACTER(LEN=5) : : town = "Glasgow"
- CHARACTER(LEN=*) : : town = "Glasgow"
- CHARACTER(LEN=*), PARAMETER : : city = "Glasgow"
- INTEGER : : pi = +22/7
- LOGICAL : : wibble = .TRUE.
- CHARACTER(LEN=*), PARAMETER : : "Bognor"
- REAL, PARAMETER : : pye = 22.0/7.0
- REAL : : two_pie = pye*2
- REAL : : a = 1., b = 2
- LOGICAL(LEN=12) : : frisnet
- CHARACTER(LEN=6) : : you_know = 'y'know"
- CHARACTER(LEN=6) : : you_know = "y'know"
- INTEGER : : ia ib ic id
- DOUBLE PRECISION : : pattie = +1.0D0
- DOUBLE PRECISION : : pattie = -1.0E-0
- REAL : : poie = 4.*atan(1.)

Ces déclarations sont-elles ou non correctes ? Pourquoi ?

- ReAl : : x OK
- CHARACTER : : name OK
- CHARACTER(LEN=10) : : name OK
- REAL : : var-1 Non, on ne peut pas avoir « -1 » dans un identificateur
- INTEGER : : 1a Non, un identificateur ne peut pas commencer par un nombre
- BOOLEAN : : loji
- DOUBLE : : X
- CHARACTER(LEN=5) : : town = "Glasgow"
- CHARACTER(LEN=*) : : town = "Glasgow"
- CHARACTER(LEN=*), PARAMETER : : city = "Glasgow"
- INTEGER : : pi = +22/7
- LOGICAL : : wibble = .TRUE.
- CHARACTER(LEN=*), PARAMETER : : "Bognor"
- REAL, PARAMETER : : pye = 22.0/7.0
- REAL : : two_pie = pye*2
- REAL : : a = 1., b = 2
- LOGICAL(LEN=12) : : frisnet
- CHARACTER(LEN=6) : : you_know = 'y'know"
- CHARACTER(LEN=6) : : you_know = "y'know"
- INTEGER : : ia ib ic id
- DOUBLE PRECISION : : pattie = +1.0D0
- DOUBLE PRECISION : : pattie = -1.0E-0
- REAL : : poie = 4.*atan(1.)

Ces déclarations sont-elles ou non correctes ? Pourquoi ?

- ReAl : : x OK
- CHARACTER : : name OK
- CHARACTER(LEN=10) : : name OK
- REAL : : var-1 Non, on ne peut pas avoir « -1 » dans un identificateur
- INTEGER : : 1a Non, un identificateur ne peut pas commencer par un nombre
- BOOLEAN : : loji Non, type « boolean » n'existe pas : utiliser « logical »
- DOUBLE : : X
- CHARACTER(LEN=5) : : town = "Glasgow"
- CHARACTER(LEN=*) : : town = "Glasgow"
- CHARACTER(LEN=*), PARAMETER : : city = "Glasgow"
- INTEGER : : pi = +22/7
- LOGICAL : : wibble = .TRUE.
- CHARACTER(LEN=*), PARAMETER : : "Bognor"
- REAL, PARAMETER : : pye = 22.0/7.0
- REAL : : two_pie = pye*2
- REAL : : a = 1., b = 2
- LOGICAL(LEN=12) : : frisnet
- CHARACTER(LEN=6) : : you_know = 'y'know"
- CHARACTER(LEN=6) : : you_know = "y'know"
- INTEGER : : ia ib ic id
- DOUBLE PRECISION : : pattie = +1.0D0
- DOUBLE PRECISION : : pattie = -1.0E-0
- REAL : : poie = 4.*atan(1.)

Ces déclarations sont-elles ou non correctes ? Pourquoi ?

- ReAl :: x OK
- CHARACTER :: name OK
- CHARACTER(LEN=10) :: name OK
- REAL :: var-1 Non, on ne peut pas avoir « -1 » dans un identificateur
- INTEGER :: 1a Non, un identificateur ne peut pas commencer par un nombre
- BOOLEAN :: loji Non, type « boolean » n'existe pas : utiliser « logical »
- DOUBLE :: X Non, le type correct est « double precision »
- CHARACTER(LEN=5) :: town = "Glasgow"
- CHARACTER(LEN=*) :: town = "Glasgow"
- CHARACTER(LEN=*), PARAMETER :: city = "Glasgow"
- INTEGER :: pi = +22/7
- LOGICAL :: wibble = .TRUE.
- CHARACTER(LEN=*), PARAMETER :: "Bognor"
- REAL, PARAMETER :: pye = 22.0/7.0
- REAL :: two_pie = pye*2
- REAL :: a = 1., b = 2
- LOGICAL(LEN=12) :: frisnet
- CHARACTER(LEN=6) :: you_know = 'y'know"
- CHARACTER(LEN=6) :: you_know = "y'know"
- INTEGER :: ia ib ic id
- DOUBLE PRECISION :: pattie = +1.0D0
- DOUBLE PRECISION :: pattie = -1.0E-0
- REAL :: poie = 4.*atan(1.)

Ces déclarations sont-elles ou non correctes ? Pourquoi ?

- ReAl : : x OK
- CHARACTER : : name OK
- CHARACTER(LEN=10) : : name OK
- REAL : : var-1 Non, on ne peut pas avoir « -1 » dans un identificateur
- INTEGER : : 1a Non, un identificateur ne peut pas commencer par un nombre
- BOOLEAN : : loji Non, type « boolean » n'existe pas : utiliser « logical »
- DOUBLE : : X Non, le type correct est « double precision »
- CHARACTER(LEN=5) : : town = "Glasgow" OK, town=« Glasg »
- CHARACTER(LEN=*) : : town = "Glasgow"
- CHARACTER(LEN=*), PARAMETER : : city = "Glasgow"
- INTEGER : : pi = +22/7
- LOGICAL : : wibble = .TRUE.
- CHARACTER(LEN=*), PARAMETER : : "Bognor"
- REAL, PARAMETER : : pye = 22.0/7.0
- REAL : : two_pie = pye*2
- REAL : : a = 1., b = 2
- LOGICAL(LEN=12) : : frisnet
- CHARACTER(LEN=6) : : you_know = 'y'know"
- CHARACTER(LEN=6) : : you_know = "y'know"
- INTEGER : : ia ib ic id
- DOUBLE PRECISION : : pattie = +1.0D0
- DOUBLE PRECISION : : pattie = -1.0E-0
- REAL : : poie = 4.*atan(1.)

Ces déclarations sont-elles ou non correctes ? Pourquoi ?

- ReAl : : x OK
- CHARACTER : : name OK
- CHARACTER(LEN=10) : : name OK
- REAL : : var-1 Non, on ne peut pas avoir « -1 » dans un identificateur
- INTEGER : : 1a Non, un identificateur ne peut pas commencer par un nombre
- BOOLEAN : : loji Non, type « boolean » n'existe pas : utiliser « logical »
- DOUBLE : : X Non, le type correct est « double precision »
- CHARACTER(LEN=5) : : town = "Glasgow" OK, town=« Glasg »
- CHARACTER(LEN=*) : : town = "Glasgow" Non, on ne peut pas avoir « len(*) » pour un non-PARAMETER
- CHARACTER(LEN=*), PARAMETER : : city = "Glasgow"
- INTEGER : : pi = +22/7
- LOGICAL : : wibble = .TRUE.
- CHARACTER(LEN=*), PARAMETER : : "Bognor"
- REAL, PARAMETER : : pye = 22.0/7.0
- REAL : : two_pie = pye*2
- REAL : : a = 1., b = 2
- LOGICAL(LEN=12) : : frisnet
- CHARACTER(LEN=6) : : you_know = 'y'know"
- CHARACTER(LEN=6) : : you_know = "y'know"
- INTEGER : : ia ib ic id
- DOUBLE PRECISION : : pattie = +1.0D0
- DOUBLE PRECISION : : pattie = -1.0E-0
- REAL : : poie = 4.*atan(1.)

Ces déclarations sont-elles ou non correctes ? Pourquoi ?

- ReAl : : x OK
- CHARACTER : : name OK
- CHARACTER(LEN=10) : : name OK
- REAL : : var-1 Non, on ne peut pas avoir « -1 » dans un identificateur
- INTEGER : : 1a Non, un identificateur ne peut pas commencer par un nombre
- BOOLEAN : : loji Non, type « boolean » n'existe pas : utiliser « logical »
- DOUBLE : : X Non, le type correct est « double precision »
- CHARACTER(LEN=5) : : town = "Glasgow" OK, town=« Glasg »
- CHARACTER(LEN=*) : : town = "Glasgow" Non, on ne peut pas avoir « len(*) » pour un non-PARAMETER
- CHARACTER(LEN=*), PARAMETER : : city = "Glasgow" OK
- INTEGER : : pi = +22/7
- LOGICAL : : wibble = .TRUE.
- CHARACTER(LEN=*), PARAMETER : : "Bognor"
- REAL, PARAMETER : : pye = 22.0/7.0
- REAL : : two_pie = pye*2
- REAL : : a = 1., b = 2
- LOGICAL(LEN=12) : : frisnet
- CHARACTER(LEN=6) : : you_know = 'y'know"
- CHARACTER(LEN=6) : : you_know = "y'know"
- INTEGER : : ia ib ic id
- DOUBLE PRECISION : : pattie = +1.0D0
- DOUBLE PRECISION : : pattie = -1.0E-0
- REAL : : poie = 4.*atan(1.)

Ces déclarations sont-elles ou non correctes ? Pourquoi ?

- ReAl : : x — OK
- CHARACTER : : name — OK
- CHARACTER(LEN=10) : : name — OK
- REAL : : var-1 — Non, on ne peut pas avoir « -1 » dans un identificateur
- INTEGER : : 1a — Non, un identificateur ne peut pas commencer par un nombre
- BOOLEAN : : loji — Non, type « boolean » n'existe pas : utiliser « logical »
- DOUBLE : : X — Non, le type correct est « double precision »
- CHARACTER(LEN=5) : : town = "Glasgow" — OK, town=« Glasg »
- CHARACTER(LEN=*) : : town = "Glasgow" — Non, on ne peut pas avoir « len(*) » pour un non-PARAMETER
- CHARACTER(LEN=*), PARAMETER : : city = "Glasgow" — OK
- INTEGER : : pi = +22/7 — OK, la valeur de pi sera 3 car c'est un entier
- LOGICAL : : wibble = .TRUE.
- CHARACTER(LEN=*), PARAMETER : : "Bognor"
- REAL, PARAMETER : : pye = 22.0/7.0
- REAL : : two_pie = pye*2
- REAL : : a = 1., b = 2
- LOGICAL(LEN=12) : : frisnet
- CHARACTER(LEN=6) : : you_know = 'y'know"
- CHARACTER(LEN=6) : : you_know = "y'know"
- INTEGER : : ia ib ic id
- DOUBLE PRECISION : : pattie = +1.0D0
- DOUBLE PRECISION : : pattie = -1.0E-0
- REAL : : poie = 4.*atan(1.)

Ces déclarations sont-elles ou non correctes ? Pourquoi ?

- ReAl :: x OK
- CHARACTER :: name OK
- CHARACTER(LEN=10) :: name OK
- REAL :: var-1 Non, on ne peut pas avoir « -1 » dans un identificateur
- INTEGER :: 1a Non, un identificateur ne peut pas commencer par un nombre
- BOOLEAN :: loji Non, type « boolean » n'existe pas : utiliser « logical »
- DOUBLE :: X Non, le type correct est « double precision »
- CHARACTER(LEN=5) :: town = "Glasgow" OK, town=« Glasg »
- CHARACTER(LEN=*) :: town = "Glasgow" Non, on ne peut pas avoir « len(*) » pour un non-PARAMETER
- CHARACTER(LEN=*), PARAMETER :: city = "Glasgow" OK
- INTEGER :: pi = +22/7 OK, la valeur de pi sera 3 car c'est un entier
- LOGICAL :: wibble = .TRUE. OK
- CHARACTER(LEN=*), PARAMETER :: "Bognor"
- REAL, PARAMETER :: pye = 22.0/7.0
- REAL :: two_pie = pye*2
- REAL :: a = 1., b = 2
- LOGICAL(LEN=12) :: frisnet
- CHARACTER(LEN=6) :: you_know = 'y'know"
- CHARACTER(LEN=6) :: you_know = "y'know"
- INTEGER :: ia ib ic id
- DOUBLE PRECISION :: pattie = +1.0D0
- DOUBLE PRECISION :: pattie = -1.0E-0
- REAL :: poie = 4.*atan(1.)

Ces déclarations sont-elles ou non correctes ? Pourquoi ?

- ReAl : : x OK
- CHARACTER : : name OK
- CHARACTER(LEN=10) : : name OK
- REAL : : var-1 Non, on ne peut pas avoir « -1 » dans un identificateur
- INTEGER : : 1a Non, un identificateur ne peut pas commencer par un nombre
- BOOLEAN : : loji Non, type « boolean » n'existe pas : utiliser « logical »
- DOUBLE : : X Non, le type correct est « double precision »
- CHARACTER(LEN=5) : : town = "Glasgow" OK, town=« Glasg »
- CHARACTER(LEN=*) : : town = "Glasgow" Non, on ne peut pas avoir « len(*) » pour un non-PARAMETER
- CHARACTER(LEN=*), PARAMETER : : city = "Glasgow" OK
- INTEGER : : pi = +22/7 OK, la valeur de pi sera 3 car c'est un entier
- LOGICAL : : wibble = .TRUE. OK
- CHARACTER(LEN=*), PARAMETER : : "Bognor" Non, pas de nom de variable
- REAL, PARAMETER : : pye = 22.0/7.0
- REAL : : two_pie = pye*2
- REAL : : a = 1., b = 2
- LOGICAL(LEN=12) : : frisnet
- CHARACTER(LEN=6) : : you_know = 'y'know"
- CHARACTER(LEN=6) : : you_know = "y'know"
- INTEGER : : ia ib ic id
- DOUBLE PRECISION : : pattie = +1.0D0
- DOUBLE PRECISION : : pattie = -1.0E-0
- REAL : : poie = 4.*atan(1.)

Ces déclarations sont-elles ou non correctes ? Pourquoi ?

- ReAl : : x OK
- CHARACTER : : name OK
- CHARACTER(LEN=10) : : name OK
- REAL : : var-1 Non, on ne peut pas avoir « -1 » dans un identificateur
- INTEGER : : 1a Non, un identificateur ne peut pas commencer par un nombre
- BOOLEAN : : loji Non, type « boolean » n'existe pas : utiliser « logical »
- DOUBLE : : X Non, le type correct est « double precision »
- CHARACTER(LEN=5) : : town = "Glasgow" OK, town=« Glasg »
- CHARACTER(LEN=*) : : town = "Glasgow" Non, on ne peut pas avoir « len(*) » pour un non-PARAMETER
- CHARACTER(LEN=*), PARAMETER : : city = "Glasgow" OK
- INTEGER : : pi = +22/7 OK, la valeur de pi sera 3 car c'est un entier
- LOGICAL : : wibble = .TRUE. OK
- CHARACTER(LEN=*), PARAMETER : : "Bognor" Non, pas de nom de variable
- REAL, PARAMETER : : pye = 22.0/7.0 OK
- REAL : : two_pie = pye*2
- REAL : : a = 1., b = 2
- LOGICAL(LEN=12) : : frisnet
- CHARACTER(LEN=6) : : you_know = 'y'know"
- CHARACTER(LEN=6) : : you_know = "y'know"
- INTEGER : : ia ib ic id
- DOUBLE PRECISION : : pattie = +1.0D0
- DOUBLE PRECISION : : pattie = -1.0E-0
- REAL : : poie = 4.*atan(1.)

Ces déclarations sont-elles ou non correctes ? Pourquoi ?

- ReAl : : x OK
- CHARACTER : : name OK
- CHARACTER(LEN=10) : : name OK
- REAL : : var-1 Non, on ne peut pas avoir « -1 » dans un identificateur
- INTEGER : : 1a Non, un identificateur ne peut pas commencer par un nombre
- BOOLEAN : : loji Non, type « boolean » n'existe pas : utiliser « logical »
- DOUBLE : : X Non, le type correct est « double precision »
- CHARACTER(LEN=5) : : town = "Glasgow" OK, town=« Glasg »
- CHARACTER(LEN=*) : : town = "Glasgow" Non, on ne peut pas avoir « len(*) » pour un non-PARAMETER
- CHARACTER(LEN=*), PARAMETER : : city = "Glasgow" OK
- INTEGER : : pi = +22/7 OK, la valeur de pi sera 3 car c'est un entier
- LOGICAL : : wibble = .TRUE. OK
- CHARACTER(LEN=*), PARAMETER : : "Bognor" Non, pas de nom de variable
- REAL, PARAMETER : : pye = 22.0/7.0 OK
- REAL : : two_pie = pye*2 OK si pye est un PARAMETER
- REAL : : a = 1., b = 2
- LOGICAL(LEN=12) : : frisnet
- CHARACTER(LEN=6) : : you_know = 'y'know"
- CHARACTER(LEN=6) : : you_know = "y'know"
- INTEGER : : ia ib ic id
- DOUBLE PRECISION : : pattie = +1.0D0
- DOUBLE PRECISION : : pattie = -1.0E-0
- REAL : : poie = 4.*atan(1.)

Ces déclarations sont-elles ou non correctes ? Pourquoi ?

- ReAl : : x — OK
- CHARACTER : : name — OK
- CHARACTER(LEN=10) : : name — OK
- REAL : : var-1 — Non, on ne peut pas avoir « -1 » dans un identificateur
- INTEGER : : 1a — Non, un identificateur ne peut pas commencer par un nombre
- BOOLEAN : : loji — Non, type « boolean » n'existe pas : utiliser « logical »
- DOUBLE : : X — Non, le type correct est « double precision »
- CHARACTER(LEN=5) : : town = "Glasgow" — OK, town=« Glasg »
- CHARACTER(LEN=*) : : town = "Glasgow" — Non, on ne peut pas avoir « len(*) » pour un non-PARAMETER
- CHARACTER(LEN=*), PARAMETER : : city = "Glasgow" — OK
- INTEGER : : pi = +22/7 — OK, la valeur de pi sera 3 car c'est un entier
- LOGICAL : : wibble = .TRUE. — OK
- CHARACTER(LEN=*), PARAMETER : : "Bognor" — Non, pas de nom de variable
- REAL, PARAMETER : : pye = 22.0/7.0 — OK
- REAL : : two_pie = pye*2 — OK si pye est un PARAMETER
- REAL : : a = 1., b = 2 — OK, la valeur de b est 2.0
- LOGICAL(LEN=12) : : frisnet
- CHARACTER(LEN=6) : : you_know = 'y'know"
- CHARACTER(LEN=6) : : you_know = "y'know"
- INTEGER : : ia ib ic id
- DOUBLE PRECISION : : pattie = +1.0D0
- DOUBLE PRECISION : : pattie = -1.0E-0
- REAL : : poie = 4.*atan(1.)

Ces déclarations sont-elles ou non correctes ? Pourquoi ?

- ReAl : : x — OK
- CHARACTER : : name — OK
- CHARACTER(LEN=10) : : name — OK
- REAL : : var-1 — Non, on ne peut pas avoir « -1 » dans un identificateur
- INTEGER : : 1a — Non, un identificateur ne peut pas commencer par un nombre
- BOOLEAN : : loji — Non, type « boolean » n'existe pas : utiliser « logical »
- DOUBLE : : X — Non, le type correct est « double precision »
- CHARACTER(LEN=5) : : town = "Glasgow" — OK, town=« Glasg »
- CHARACTER(LEN=*) : : town = "Glasgow" — Non, on ne peut pas avoir « len(*) » pour un non-PARAMETER
- CHARACTER(LEN=*), PARAMETER : : city = "Glasgow" — OK
- INTEGER : : pi = +22/7 — OK, la valeur de pi sera 3 car c'est un entier
- LOGICAL : : wibble = .TRUE. — OK
- CHARACTER(LEN=*), PARAMETER : : "Bognor" — Non, pas de nom de variable
- REAL, PARAMETER : : pye = 22.0/7.0 — OK
- REAL : : two_pie = pye*2 — OK si pye est un PARAMETER
- REAL : : a = 1., b = 2 — OK, la valeur de b est 2.0
- LOGICAL(LEN=12) : : frisnet — Non, pas de LEN dans LOGICAL
- CHARACTER(LEN=6) : : you_know = 'y'know"
- CHARACTER(LEN=6) : : you_know = "y'know"
- INTEGER : : ia ib ic id
- DOUBLE PRECISION : : pattie = +1.0D0
- DOUBLE PRECISION : : pattie = -1.0E-0
- REAL : : poie = 4.*atan(1.)

Ces déclarations sont-elles ou non correctes ? Pourquoi ?

- ReAl : : x OK
- CHARACTER : : name OK
- CHARACTER(LEN=10) : : name OK
- REAL : : var-1 Non, on ne peut pas avoir « -1 » dans un identificateur
- INTEGER : : 1a Non, un identificateur ne peut pas commencer par un nombre
- BOOLEAN : : loji Non, type « boolean » n'existe pas : utiliser « logical »
- DOUBLE : : X Non, le type correct est « double precision »
- CHARACTER(LEN=5) : : town = "Glasgow" OK, town=« Glasg »
- CHARACTER(LEN=*) : : town = "Glasgow" Non, on ne peut pas avoir « len(*) » pour un non-PARAMETER
- CHARACTER(LEN=*), PARAMETER : : city = "Glasgow" OK
- INTEGER : : pi = +22/7 OK, la valeur de pi sera 3 car c'est un entier
- LOGICAL : : wibble = .TRUE. OK
- CHARACTER(LEN=*), PARAMETER : : "Bognor" Non, pas de nom de variable
- REAL, PARAMETER : : pye = 22.0/7.0 OK
- REAL : : two_pie = pye*2 OK si pye est un PARAMETER
- REAL : : a = 1., b = 2 OK, la valeur de b est 2.0
- LOGICAL(LEN=12) : : frisnet Non, pas de LEN dans LOGICAL
- CHARACTER(LEN=6) : : you_know = 'y'know" Non, problèmes dans les délimiteurs
- CHARACTER(LEN=6) : : you_know = "y'know"
- INTEGER : : ia ib ic id
- DOUBLE PRECISION : : pattie = +1.0D0
- DOUBLE PRECISION : : pattie = -1.0E-0
- REAL : : poie = 4.*atan(1.)

Ces déclarations sont-elles ou non correctes ? Pourquoi ?

- ReAl : : x OK
- CHARACTER : : name OK
- CHARACTER(LEN=10) : : name OK
- REAL : : var-1 Non, on ne peut pas avoir « -1 » dans un identificateur
- INTEGER : : 1a Non, un identificateur ne peut pas commencer par un nombre
- BOOLEAN : : loji Non, type « boolean » n'existe pas : utiliser « logical »
- DOUBLE : : X Non, le type correct est « double precision »
- CHARACTER(LEN=5) : : town = "Glasgow" OK, town=« Glasg »
- CHARACTER(LEN=*) : : town = "Glasgow" Non, on ne peut pas avoir « len(*) » pour un non-PARAMETER
- CHARACTER(LEN=*), PARAMETER : : city = "Glasgow" OK
- INTEGER : : pi = +22/7 OK, la valeur de pi sera 3 car c'est un entier
- LOGICAL : : wibble = .TRUE. OK
- CHARACTER(LEN=*), PARAMETER : : "Bognor" Non, pas de nom de variable
- REAL, PARAMETER : : pye = 22.0/7.0 OK
- REAL : : two_pie = pye*2 OK si pye est un PARAMETER
- REAL : : a = 1., b = 2 OK, la valeur de b est 2.0
- LOGICAL(LEN=12) : : frisnet Non, pas de LEN dans LOGICAL
- CHARACTER(LEN=6) : : you_know = 'y'know" Non, problèmes dans les délimiteurs
- CHARACTER(LEN=6) : : you_know = "y'know" OK
- INTEGER : : ia ib ic id
- DOUBLE PRECISION : : pattie = +1.0D0
- DOUBLE PRECISION : : pattie = -1.0E-0
- REAL : : poie = 4.*atan(1.)

Ces déclarations sont-elles ou non correctes ? Pourquoi ?

- ReAl : : x — OK
- CHARACTER : : name — OK
- CHARACTER(LEN=10) : : name — OK
- REAL : : var-1 — Non, on ne peut pas avoir « -1 » dans un identificateur
- INTEGER : : 1a — Non, un identificateur ne peut pas commencer par un nombre
- BOOLEAN : : loji — Non, type « boolean » n'existe pas : utiliser « logical »
- DOUBLE : : X — Non, le type correct est « double precision »
- CHARACTER(LEN=5) : : town = "Glasgow" — OK, town=« Glasg »
- CHARACTER(LEN=*) : : town = "Glasgow" — Non, on ne peut pas avoir « len(*) » pour un non-PARAMETER
- CHARACTER(LEN=*), PARAMETER : : city = "Glasgow" — OK
- INTEGER : : pi = +22/7 — OK, la valeur de pi sera 3 car c'est un entier
- LOGICAL : : wibble = .TRUE. — OK
- CHARACTER(LEN=*), PARAMETER : : "Bognor" — Non, pas de nom de variable
- REAL, PARAMETER : : pye = 22.0/7.0 — OK
- REAL : : two_pie = pye*2 — OK si pye est un PARAMETER
- REAL : : a = 1., b = 2 — OK, la valeur de b est 2.0
- LOGICAL(LEN=12) : : frisnet — Non, pas de LEN dans LOGICAL
- CHARACTER(LEN=6) : : you_know = 'y'know" — Non, problèmes dans les délimiteurs
- CHARACTER(LEN=6) : : you_know = "y'know" — OK
- INTEGER : : ia ib ic id — Non, enlever les espaces ou mettre des virgules
- DOUBLE PRECISION : : pattie = +1.0D0
- DOUBLE PRECISION : : pattie = -1.0E-0
- REAL : : poie = 4.*atan(1.)

Ces déclarations sont-elles ou non correctes ? Pourquoi ?

- ReAl : : x OK
- CHARACTER : : name OK
- CHARACTER(LEN=10) : : name OK
- REAL : : var-1 Non, on ne peut pas avoir « -1 » dans un identificateur
- INTEGER : : 1a Non, un identificateur ne peut pas commencer par un nombre
- BOOLEAN : : loji Non, type « boolean » n'existe pas : utiliser « logical »
- DOUBLE : : X Non, le type correct est « double precision »
- CHARACTER(LEN=5) : : town = "Glasgow" OK, town=« Glasg »
- CHARACTER(LEN=*) : : town = "Glasgow" Non, on ne peut pas avoir « len(*) » pour un non-PARAMETER
- CHARACTER(LEN=*), PARAMETER : : city = "Glasgow" OK
- INTEGER : : pi = +22/7 OK, la valeur de pi sera 3 car c'est un entier
- LOGICAL : : wibble = .TRUE. OK
- CHARACTER(LEN=*), PARAMETER : : "Bognor" Non, pas de nom de variable
- REAL, PARAMETER : : pye = 22.0/7.0 OK
- REAL : : two_pie = pye*2 OK si pye est un PARAMETER
- REAL : : a = 1., b = 2 OK, la valeur de b est 2.0
- LOGICAL(LEN=12) : : frisnet Non, pas de LEN dans LOGICAL
- CHARACTER(LEN=6) : : you_know = 'y'know" Non, problèmes dans les délimiteurs
- CHARACTER(LEN=6) : : you_know = "y'know" OK
- INTEGER : : ia ib ic id Non, enlever les espaces ou mettre des virgules
- DOUBLE PRECISION : : pattie = +1.0D0 OK, les « : : » sont obligatoires à cause de l'initialisation
- DOUBLE PRECISION : : pattie = -1.0E-0
- REAL : : poie = 4.*atan(1.)

Ces déclarations sont-elles ou non correctes ? Pourquoi ?

- ReAl : : x OK
- CHARACTER : : name OK
- CHARACTER(LEN=10) : : name OK
- REAL : : var-1 Non, on ne peut pas avoir « -1 » dans un identificateur
- INTEGER : : 1a Non, un identificateur ne peut pas commencer par un nombre
- BOOLEAN : : loji Non, type « boolean » n'existe pas : utiliser « logical »
- DOUBLE : : X Non, le type correct est « double precision »
- CHARACTER(LEN=5) : : town = "Glasgow" OK, town=« Glasg »
- CHARACTER(LEN=*) : : town = "Glasgow" Non, on ne peut pas avoir « len(*) » pour un non-PARAMETER
- CHARACTER(LEN=*), PARAMETER : : city = "Glasgow" OK
- INTEGER : : pi = +22/7 OK, la valeur de pi sera 3 car c'est un entier
- LOGICAL : : wibble = .TRUE. OK
- CHARACTER(LEN=*), PARAMETER : : "Bognor" Non, pas de nom de variable
- REAL, PARAMETER : : pye = 22.0/7.0 OK
- REAL : : two_pie = pye*2 OK si pye est un PARAMETER
- REAL : : a = 1., b = 2 OK, la valeur de b est 2.0
- LOGICAL(LEN=12) : : frisnet Non, pas de LEN dans LOGICAL
- CHARACTER(LEN=6) : : you_know = 'y'know" Non, problèmes dans les délimiteurs
- CHARACTER(LEN=6) : : you_know = "y'know" OK
- INTEGER : : ia ib ic id Non, enlever les espaces ou mettre des virgules
- DOUBLE PRECISION : : pattie = +1.0D0 OK, les « : : » sont obligatoires à cause de l'initialisation
- DOUBLE PRECISION : : pattie = -1.0E-0 OK
- REAL : : poie = 4.*atan(1.)

Ces déclarations sont-elles ou non correctes ? Pourquoi ?

- ReAl : : x OK
- CHARACTER : : name OK
- CHARACTER(LEN=10) : : name OK
- REAL : : var-1 Non, on ne peut pas avoir « -1 » dans un identificateur
- INTEGER : : 1a Non, un identificateur ne peut pas commencer par un nombre
- BOOLEAN : : loji Non, type « boolean » n'existe pas : utiliser « logical »
- DOUBLE : : X Non, le type correct est « double precision »
- CHARACTER(LEN=5) : : town = "Glasgow" OK, town=« Glasg »
- CHARACTER(LEN=*) : : town = "Glasgow" Non, on ne peut pas avoir « len(*) » pour un non-PARAMETER
- CHARACTER(LEN=*), PARAMETER : : city = "Glasgow" OK
- INTEGER : : pi = +22/7 OK, la valeur de pi sera 3 car c'est un entier
- LOGICAL : : wibble = .TRUE. OK
- CHARACTER(LEN=*), PARAMETER : : "Bognor" Non, pas de nom de variable
- REAL, PARAMETER : : pye = 22.0/7.0 OK
- REAL : : two_pie = pye*2 OK si pye est un PARAMETER
- REAL : : a = 1., b = 2 OK, la valeur de b est 2.0
- LOGICAL(LEN=12) : : frisnet Non, pas de LEN dans LOGICAL
- CHARACTER(LEN=6) : : you_know = 'y'know" Non, problèmes dans les délimiteurs
- CHARACTER(LEN=6) : : you_know = "y'know" OK
- INTEGER : : ia ib ic id Non, enlever les espaces ou mettre des virgules
- DOUBLE PRECISION : : pattie = +1.0D0 OK, les « : : » sont obligatoires à cause de l'initialisation
- DOUBLE PRECISION : : pattie = -1.0E-0 OK
- REAL : : poie = 4.*atan(1.)

Ces déclarations sont-elles ou non correctes ? Pourquoi ?

- ReAl : : x OK
- CHARACTER : : name OK
- CHARACTER(LEN=10) : : name OK
- REAL : : var-1 Non, on ne peut pas avoir « -1 » dans un identificateur
- INTEGER : : 1a Non, un identificateur ne peut pas commencer par un nombre
- BOOLEAN : : loji Non, type « boolean » n'existe pas : utiliser « logical »
- DOUBLE : : X Non, le type correct est « double precision »
- CHARACTER(LEN=5) : : town = "Glasgow" OK, town=« Glasg »
- CHARACTER(LEN=*) : : town = "Glasgow" Non, on ne peut pas avoir « len(*) » pour un non-PARAMETER
- CHARACTER(LEN=*), PARAMETER : : city = "Glasgow" OK
- INTEGER : : pi = +22/7 OK, la valeur de pi sera 3 car c'est un entier
- LOGICAL : : wibble = .TRUE. OK
- CHARACTER(LEN=*), PARAMETER : : "Bognor" Non, pas de nom de variable
- REAL, PARAMETER : : pye = 22.0/7.0 OK
- REAL : : two_pie = pye*2 OK si pye est un PARAMETER
- REAL : : a = 1., b = 2 OK, la valeur de b est 2.0
- LOGICAL(LEN=12) : : frisnet Non, pas de LEN dans LOGICAL
- CHARACTER(LEN=6) : : you_know = 'y'know" Non, problèmes dans les délimiteurs
- CHARACTER(LEN=6) : : you_know = "y'know" OK
- INTEGER : : ia ib ic id Non, enlever les espaces ou mettre des virgules
- DOUBLE PRECISION : : pattie = +1.0D0 OK, les « : : » sont obligatoires à cause de l'initialisation
- DOUBLE PRECISION : : pattie = -1.0E-0 OK
- REAL : : poie = 4.*atan(1.) Non, on ne peut pas utiliser « atan » dans l'initialisation

Déclarer les objets suivants

Nom	Statut	Type	Valeur initiale
feet	variable	integer	-
miles	variable	real	-
town	variable	character ($\leq$ 20 lettres)	-
home_town	constant	character	<votre ville>
in_ma	constant	logical	<est-elle au Maroc ?>
sin_half	constant	real	$sin(0.5) = 0.47942554$

Déclarer les objets suivants

Nom	Statut	Type	Valeur initiale
feet	variable	integer	-
miles	variable	real	-
town	variable	character ($\leq$ 20 lettres)	-
home_town	constant	character	<votre ville>
in_ma	constant	logical	<est-elle au Maroc ?>
sin_half	constant	real	$sin(0.5) = 0.47942554$

- INTEGER feet

Déclarer les objets suivants

Nom	Statut	Type	Valeur initiale
feet	variable	integer	-
miles	variable	real	-
town	variable	character ($\leq$ 20 lettres)	-
home_town	constant	character	<votre ville>
in_ma	constant	logical	<est-elle au Maroc ?>
sin_half	constant	real	$sin(0.5) = 0.47942554$

- INTEGER feet
- REAL : : miles

Déclarer les objets suivants

Nom	Statut	Type	Valeur initiale
feet	variable	integer	-
miles	variable	real	-
town	variable	character ($\leq$ 20 lettres)	-
home_town	constant	character	<votre ville>
in_ma	constant	logical	<est-elle au Maroc ?>
sin_half	constant	real	$sin(0.5) = 0.47942554$

- INTEGER feet
- REAL : : miles
- CHARACTER(LEN=20) : : town

Déclarer les objets suivants

Nom	Statut	Type	Valeur initiale
feet	variable	integer	-
miles	variable	real	-
town	variable	character ($\leq$ 20 lettres)	-
home_town	constant	character	<votre ville>
in_ma	constant	logical	<est-elle au Maroc ?>
sin_half	constant	real	$sin(0.5) = 0.47942554$

- INTEGER feet
- REAL : : miles
- CHARACTER(LEN=20) : : town
- CHARACTER(LEN=*), PARAMETER : : home_town = 'Lyon'

Déclarer les objets suivants

Nom	Statut	Type	Valeur initiale
feet	variable	integer	-
miles	variable	real	-
town	variable	character ($\leq$ 20 lettres)	-
home_town	constant	character	<votre ville>
in_ma	constant	logical	<est-elle au Maroc ?>
sin_half	constant	real	$sin(0.5) = 0.47942554$

- INTEGER feet
- REAL : : miles
- CHARACTER(LEN=20) : : town
- CHARACTER(LEN=*), PARAMETER : : home_town = 'Lyon'
- LOGICAL, PARAMETER : : in_ma = .FALSE.

Déclarer les objets suivants

Nom	Statut	Type	Valeur initiale
feet	variable	integer	-
miles	variable	real	-
town	variable	character ($\leq$ 20 lettres)	-
home_town	constant	character	<votre ville>
in_ma	constant	logical	<est-elle au Maroc ?>
sin_half	constant	real	$sin(0.5) = 0.47942554$

- INTEGER feet
- REAL : : miles
- CHARACTER(LEN=20) : : town
- CHARACTER(LEN=*), PARAMETER : : home_town = 'Lyon'
- LOGICAL, PARAMETER : : in_ma = .FALSE.
- REAL, PARAMETER : : sin_half = 0.47942554

1 Généralités
2 Déclarations & variables
3 Opérateurs et expressions
4 Structures de contrôles
5 Tableaux
6 Gestion de la mémoire
7 Types dérivés et modules
8 Procédures
9 Entrées-Sorties
10 Fonctions intrinsèques
11 Pointeurs
12 Travaux pratiques

Opérateurs arithmétiques

- Le type d'une expression arithmétique dépend des types de ses opérandes. Dans le cas d'opérateurs binaires :
  - si les 2 opérandes sont du même type alors l'expression arithmétique résultante sera de ce type.
  - si les deux opérandes ne sont pas du même type alors l'expression arithmétique sera évaluée dans le type le plus fort relativement à la hiérarchie suivante :

$$INTEGER < REAL < DOUBLE\ PRECISION < COMPLEX$$

Exemples

```
99/100      ! vaut 0, INTEGER
99./100     ! vaut 0.99, REAL
99./100d0   ! vaut 0.99d0, DOUBLE PRECISION
```

Attention

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

Variable d déclarée en DOUBLE PRECISION.

- La sous-expression 5.**0.5 est évaluée dans le type REAL car les opérandes de l'opérateur ** le sont.
- Le reste de l'évaluation s'effectue ensuite dans le type DOUBLE PRECISION
- le résultat étant finalement stocké dans la variable d.

Cette variable d bien que du type DOUBLE PRECISION hérite d'un calcul qui a commencé dans le type REAL, d'où une perte de précision. Cela peut induire par la suite des comportements inattendus lors de l'évaluation d'expressions dans lesquelles figurent cette variable.
En conclusion, lors de l'écriture d'expressions avec présence de constantes réelles que l'on désire évaluer en DOUBLE PRECISION, il est impératif d'écrire ces constantes dans ce type.

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

Opérateurs relationnels, logiques, concaténation

Opérateurs relationnels

- .LT., <, .LE., <=, .EQ., ==, .NE., /=, .GT., >, .GE. >=
- Admettent des opérandes de type INTEGER, REAL ou CHARACTER
- Seuls les opérateurs == et/= peuvent s'appliquer à des COMPLEX

Opérateurs logiques

- .NOT., .AND., .OR., .EQV., .NEQV.
- Les opérandes doivent être des expressions de type LOGICAL

Opérateur de concaténation

- N'admet que des expressions de type CHARACTER
- La chaîne réceptrice est plus grande que celle affectée, elle est complétée à l'aide du caractère espace

```
CHARACTER(len=10)   :: ch
ch = ''BON'' // ''JOUR''   ! ch = ''BONJOUR''
```

1 Généralités

2 Déclarations & variables

3 Opérateurs et expressions

4 Structures de contrôles
- Tests
- Itérations
- Exercice

5 Tableaux

6 Gestion de la mémoire

7 Types dérivés et modules

8 Procédures

9 Entrées-Sorties

10 Fonctions intrinsèques

11 Pointeurs

12 Travaux pratiques

IF

```
[nom_bloc :] IF (expr1) THEN
             ...
             ELSE IF (expr2) THEN
             ...
             ELSE
             ...
             END IF [nom_bloc]
```

- nom_bloc est une étiquette facultative : si elle est présente elle doit figurer au niveau de l'instruction END IF et peut apparaître à la suite des éventuelles instructions ELSE, ELSE IF.

SELECT CASE

```
[nom_bloc :] SELECT CASE (expr)
                CASE (liste1) [nom_bloc]
                  ...
                CASE (liste2) [nom_bloc]
                  ...
                [CASE DEFAULT [nom_bloc] ]
                  ...
             END SELECT [nom_bloc]
```

- nom_bloc est une étiquette,
- expr est une expression de type INTEGER, LOGICAL ou CHARACTER,
- liste1, liste2 sont des listes de constantes du même type que expr.

DO

```
[nom_bloc :] DO (controle_de_boucle)
                ...
             END DO [nom_bloc]
```

- nom_bloc est une étiquette,
- controle_de_boucle définit les conditions d'exécution et d'arrêt de la boucle

```
[nom_bloc :] DO
                ...
               IF (expr) EXIT
             END DO [nom_bloc]
```

- expr expression de type LOGICAL
- La condition IF peut être remplacée par une instruction SELECT CASE

Exemples

```
INTEGER        somme,n,i
DOUBLE PRECISION  x,xlast,tol
...
somme = 0
DO i=1,n,2
   somme = somme + i
END DO

DO
  xlast = x
  x = ...
  IF (ABS(x - xlast)/x < tol) EXIT
END DO
```

DO WHILE

```
[nom_bloc :] DO WHILE (expr)
                 ...
             END DO [nom_bloc]
```

- nom_bloc est une étiquette,
- expr est une expression de type scalaire logique
- Le corps de la boucle est exécuté tant que l'expression est vraie

Exemples

```
...
eps   = 1.d-08
error = 10.d0
DO WHILE (error > eps)
   ...           ! effectue une iteration quelconque
   error = ...   ! calcul de l'erreur
END DO
```

CYCLE

```
[nom_bloc :] DO
            ...
            IF (expr) CYCLE
         END DO [nom_bloc]
```

- CYCLE permet d'abandonner le traitement de l'itération courante et de passer à la suivante
- La condition IF peut être remplacée par une instruction SELECT CASE

Exemples

```
...
somme = 0
n = 0
DO
   n = n + 1
   IF (n/2*2 .NE. n) CYCLE    ! somme des n entiers pairs < 1000
   somme = somme + n
   IF (n > 1000) EXIT
END DO
```

Exercice

Ecrire un programme qui calcule les racines d'un polynôme du second degré donné par 3 coefficients a, b, et c.

$$f(x) = a^2 x + bx + c$$

a pour racine :

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Une donnée de $a = 0, b = 0, c = 0$ doit terminer le programme. Utiliser une variable intermédiaire ϵ pour tester la nullité d'un nombre.

```
program Quadratic
...
    PRINT*, "Type in a, b and c (0,0,0 will terminate)"
    READ*, a, b, c
...
end program Quadratic
```

Solution

```
program Quadratic
  implicit none
  real :: a, b, c, root1, root2
  real :: b_sq, ac4, sqrt_thing
  real, parameter :: eps = 0.0000001
  do
    print*, "type in a, b and c (0,0,0 will terminate)"
    read*, a, b, c
    if (abs(a+b+c) < eps) exit
    b_sq = b**2
    ac4  = 4*a*c
    if (b_sq > ac4) then
      sqrt_thing = sqrt(b_sq-ac4)
      root1 = (-b+sqrt_thing)/(2*a)
      root2 = (-b-sqrt_thing)/(2*a)
      print*,"the real roots are ", root1, root2
    elseif (abs(b_sq - ac4) < eps) then
      root1 = (-b)/(2*a)
      print*,"there is one real root which is", root1
    else
      print*,"there are no real roots"
    end if
  end do
end program Quadratic
```

1 Généralités

2 Déclarations & variables

3 Opérateurs et expressions

4 Structures de contrôles

5 Tableaux
- Déclaration
- Quelques définitions
- Ordre des éléments
- Initialisation
- Manipulation de tableaux
- Sections de tableaux
- Fonctions intrinsèques
- Instruction et bloc WHERE
- Exercice

6 Gestion de la mémoire

7 Types dérivés et modules

8 Procédures

9 Entrées-Sorties

10 Fonctions intrinsèques

11 Pointeurs

12 Travaux pratiques

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

```
TYPE, DIMENSION(expr1,expr2,...)    :: tab
```

- un nombre de dimensions maximal de 7
- expr*i* indique l'étendue du tableau sous la forme :
  - d'une constante entière, dans ce cas la borne inférieure est 1
  - d'une expression de la forme const1 :const2, indiquant les bornes inférieures et supérieures.

Exemples

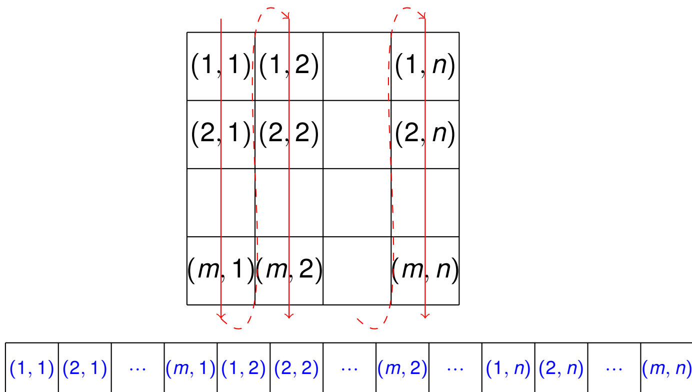
```
REAL, DIMENSION (1:5,1:5,1:5)    :: R
INTEGER, DIMENSION (-10:-1)      :: S
```

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

Exemples

```
! de rang 1, de profil (15), de taille 15
REAL, DIMENSION (15)          :: X
! de rang 2, de profil (5,3), de taille 15
REAL, DIMENSION (1:5,1:3)     :: Y
! de rang 2, de profil (5,3), de taille 15
REAL, DIMENSION (-1:3,0:2)    :: Z
! Y et Z conformants
```

- En mémoire la notion de tableau n'existe pas : les éléments sont rangés les uns à la suite des autres.
- Pour accéder à ces éléments, dans l'ordre mémoire, Fortran fait d'abord varier le premier indice, puis le second et ainsi de suite.



- Fortran permet de manipuler globalement l'ensemble des éléments d'un tableau. On pourra alors utiliser le symbole « = » comme pour l'initialisation d'une variable scalaire.

Exemples

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

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

Exemples

```
REAL, DIMENSION (4) :: heights = (/ 5.10, 5.6, 4.0, 3.6 /)
INTEGER, DIMENSION (10) :: ints = (/ 100, (i, i = 1,8), 100 /)
```

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

Exemples

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

Important

La valeur d'une expression tableau est entièrement évaluée avant d'être affectée. Les deux exemples ci-dessous ne sont pas du tout équivalents :

```
real, dimension(20) :: tab
tab(:) = tab(20:1:-1)
```

```
do i=1,20
   tab(i) = tab(21-i)
end do
```

les expressions tableaux sont en fait des notations vectorielles ce qui facilite leur vectorisation puisque contrairement aux boucles, elles évitent au compilateur le contrôle des dépendances.

- Sections régulières : ensemble d'éléments dont les indices forment une progression arithmétique
- Une section de tableau est aussi un tableau

Exemples

```
A( :)          ! tableau global
A(3 :9)        ! A(3) a A(9) par pas de 1
A(8 :3,−1)     ! A(8) a A(3) par pas de −1
```

- Sections irrégulières : accéder à des éléments quelconques par l'intermédiaire d'un vecteur d'indices. Il s'agit en fait d'une indexation indirecte

Exemples

```
integer, dimension (10,9) :: tab
tab ((/ 3, 5, 8 /) , (/ 1, 5, 7 /)) = 1
```

Attention

La notation dégénérée sous la forme « : » correspond à l'étendue de la dimension considérée.

- SHAPE(source) : retourne le profil du tableau passé en argument
- SIZE(array[,dim]) : retourne la taille ou l'étendue de la dimension dim
- UBOUND(array[,dim]), LBOUND(array[,dim]) : retourne les bornes supérieures/inférieures de chacune des dimensions (ou celle indiquée par dim)
- MINVAL(array,dim[,mask]) ou MINVAL((array[,mask]), MAXVAL(array,dim[,mask]) ou MAXVAL((array[,mask]) : plus petite ou plus grand élément du tableau

Exemples

```
integer ,   &
   dimension(-2:27,0:49) :: t
SHAPE(t)      ! (/ 30,50 /)
UBOUND(t)     ! (/ 27,49 /)
UBOUND(t(:,:))   ! (/ 30,50 /)
LBOUND(t,dim=2)  ! 0
```

Exemples

$$A = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$$

```
MINVAL( (/ 1, 4, 9 /) )    ! 1
MAXVAL( (/ 1, 4, 9 /) )    ! 9
MAXVAL(A,dim=2)  ! (/ 5, 6 /)
MINVAL(A,dim=1)  ! (/ 1, 3, 5 /)
```

- PRODUCT(array,dim[,mask]) ou PRODUCT(array[mask]) : produit de tous les éléments du tableau
- SUM(array,dim[,mask]) ou SUM(array[mask]) : somme de tous les éléments du tableau
- DOT_PRODUCT(vector_a,vector_b) : produit scalaire
- MATMUL(matrix_a,matrix_b) : produit matriciel

$$A = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix} \qquad B = \begin{pmatrix} 3 & -6 & -1 \\ 2 & 3 & 1 \\ -1 & -2 & 4 \end{pmatrix} \qquad V = \begin{pmatrix} 2 \\ -4 \\ 1 \end{pmatrix}$$

Exemples

```
PRODUCT( (/ 2,5,-6 /) )     ! -60
SUM( (/ 2,5,-6 /) )     ! 1
PRODUCT(A,dim=1)  ! (/ 2,12,30 /)
SUM(A,dim=2)  ! (/ 9,12 /)

DOT_PRODUCT( (/ 2,-3,-1 /), (/ 6,3,3 /) ) = 0
MATMUL(B,V) = (/ 29, -7, 10 /)
```

L'instruction WHERE permet d'effectuer des affectations de type tableau par l'intermédiaire d'un filtre (masque logique)

```
[nom_bloc :] WHERE (mask)
               ...
             ELSE WHERE [nom_bloc]
               ...
             END WHERE [nom_bloc]
```

Où mask est une expression logique retournant un tableau de logiques.

Exemples

```
real, dimension (10) :: a
...
WHERE ( a > 0. )
   a = log (a)
ELSE WHERE
   a = 1.
END WHERE
```

Equivalent à

```
do i = 1, 10
   if ( a(i) > 0. ) then
      a(i) = log(a(i))
   else
      a(i) = 1.
   end if
end do
```

Exercice 1

```
INTEGER, DIMENSION(-1:1,3,2) :: A
```

Ligne de code pour :

- récupérer le nombre d'éléments de A
- Le profil de A
- la borne inférieure de la deuxième dimension
- la borne supérieure de la troisième dimension

Exercice 2

```
REAL, DIMENSION(100,100) :: A, B, C
```

- Quelle différence entre C=MATMUL(A,B) et C=A*B ?

Exercice 1

```
INTEGER, DIMENSION(-1:1,3,2) :: A
```

Ligne de code pour :

- récupérer le nombre d'éléments de A SIZE(A)
- Le profil de A
- la borne inférieure de la deuxième dimension
- la borne supérieure de la troisième dimension

Exercice 2

```
REAL, DIMENSION(100,100) :: A, B, C
```

- Quelle différence entre C=MATMUL(A,B) et C=A*B ?

Exercice 1

```
INTEGER, DIMENSION(-1:1,3,2) :: A
```

Ligne de code pour :

- récupérer le nombre d'éléments de A SIZE(A)
- Le profil de A SHAPE(A)
- la borne inférieure de la deuxième dimension
- la borne supérieure de la troisième dimension

Exercice 2

```
REAL, DIMENSION(100,100) :: A, B, C
```

- Quelle différence entre C=MATMUL(A,B) et C=A*B ?

Exercice 1

```
INTEGER, DIMENSION(-1:1,3,2) :: A
```

Ligne de code pour :

- récupérer le nombre d'éléments de A SIZE(A)
- Le profil de A SHAPE(A)
- la borne inférieure de la deuxième dimension LBOUND(A,2) vaut 1
- la borne supérieure de la troisième dimension

Exercice 2

```
REAL, DIMENSION(100,100) :: A, B, C
```

- Quelle différence entre C=MATMUL(A,B) et C=A*B ?

Exercice 1

```
INTEGER, DIMENSION(-1:1,3,2) :: A
```

Ligne de code pour :

- récupérer le nombre d'éléments de A SIZE(A)
- Le profil de A SHAPE(A)
- la borne inférieure de la deuxième dimension LBOUND(A,2) vaut 1
- la borne supérieure de la troisième dimension UBOUND(A,3) vaut 2

Exercice 2

```
REAL, DIMENSION(100,100) :: A, B, C
```

- Quelle différence entre C=MATMUL(A,B) et C=A*B ?

Exercice 1

```
INTEGER, DIMENSION(-1:1,3,2) :: A
```

Ligne de code pour :

- récupérer le nombre d'éléments de A SIZE(A)
- Le profil de A SHAPE(A)
- la borne inférieure de la deuxième dimension LBOUND(A,2) vaut 1
- la borne supérieure de la troisième dimension UBOUND(A,3) vaut 2

Exercice 2

```
REAL, DIMENSION(100,100) :: A, B, C
```

- Quelle différence entre C=MATMUL(A,B) et C=A*B ?
  - le premier effectue une multiplication matricielle, le second une multiplication élément par élément

1 Généralités

2 Déclarations & variables

3 Opérateurs et expressions

4 Structures de contrôles

5 Tableaux

6 Gestion de la mémoire
- Modes d'allocation
- Tableaux automatiques
- Tableaux dynamiques
- Exercice

7 Types dérivés et modules

8 Procédures

9 Entrées-Sorties

10 Fonctions intrinsèques

11 Pointeurs

12 Travaux pratiques

Trois modes d'allocation mémoire :

- Allocation statique :
  L'espace mémoire nécessaire est spécifiée dans le code source. Lors de l'exécution, aucune allocation n'a lieu. Avantage au niveau des performances : on évite les coûts de l'allocation dynamique à l'exécution, la mémoire statique est immédiatement utilisable.
- Allocation sur la pile :
  La pile (stack en anglais) est utilisée pour stocker des variable locales à une fonction et pour passer des paramètres à cette fonction. La mémoire allouée dans la pile est toujours éphèmère et sera libérée à la sortie de la fonction.
- Allocation sur le tas :
  Le tas (heap en anglais) est utilisé lors de l'allocation dynamique de mémoire durant l'exécution d'un programme, à la demande de celui-ci par le biais de fonctions d'allocation (malloc, new en C/C++, allocate en Fortran).

Problèmes liés à la mémoire :

- Débordement de pile (stack overflow) : Bug du programme qui écrit à l'extérieur de l'espace alloué à la pile, écrasant ainsi des informations nécessaires au processus.
  - Récursivité infinie
  - Variables trop grandes : par exemple tableau de grande dimension non alloué dynamiquement
- Débordement de tas (heap overflow) : Bug du programme qui écrit à l'extérieur de l'espace alloué au tas
  - Ecriture en dehors des bornes d'un tableau alloué dynamiquement
  - ...
- En général, provoque un arrêt du programme sur une Segmentation Fault

Prévention / Debuggage

- Eviter les tableaux implicites (sans précision des dimensions) dans les procédures
- Utiliser des debuggers comme valgrind pour déterminer l'endroit du dépassement mémoire

Il est possible de définir au sein d'une procédure des tableaux dont la taille varie d'un appel à l'autre. Ils sont alloués dynamiquement à l'entrée de la procédure et libérés à sa sortie de façon implicite. Pour cela ces tableaux seront dimensionnés à l'aide d'expressions entières non constantes.

Exemple

```
subroutine echange (a , b , taille )
  integer , dimension (: ,:)   :: a,b
  integer                      :: taille
  integer , dimension(size(a,1),size(a,2)) :: C
  real , dimension (taille) :: V

  C = a
  a = b
  b = C
  ...
end subroutine echange
```

Important

- Pour pouvoir exécuter ce sous-programme, l'interface doit être « explicite »
- un tableau automatique ne peut être initialisé

- Apport intéressant de la norme Fortran 90 : allocation dynamique de mémoire.
- On spécifie l'attribut ALLOCATABLE au moment de la déclaration du tableau. Un tel tableau s'appelle tableau à profil différé (deffered-shape-array).
- Son allocation s'effectue grâce à l'instruction ALLOCATE à laquelle on indiquera le profil désiré.
- L'instruction DEALLOCATE permet de libérer l'espace mémoire alloué.
- La fonction intrinsèque ALLOCATED permet d'interroger le système pour savoir si un tableau est alloué ou non.

Exemple

```
real , dimension (: ,:) , ALLOCATABLE :: a
integer                              :: n,m, etat
read * , n , m

if (.not. ALLOCATED(a)) then
  ALLOCATE (a(n,m), stat = etat)
   if (etat /= 0) then
      print *," Erreur allocat . tableau a " ; stop 4
  end if
end if
...
DEALLOCATE (a)
```

- Il n'est pas possible de réallouer un tableau déjà alloué. Il devra être libéré auparavant.
- Un tableau local alloué dynamiquement dans une unité de programme a un état indéterminé à la sortie (RETURN/END) de cette unité sauf dans les cas suivants :
  - l'attribut SAVE a été spécifié pour ce tableau,
  - une autre unité de progr. encore active a visibilité par use association sur ce tableau déclaré dans un module,
  - cette unité de progr. est interne. De ce fait (host association), l'unité hôte peut encore y accéder.

Exercice

Écrire un programme permettant de valoriser une matrice de n lignes et m colonnes (n et m n'étant connus qu'au moment de l'exécution) de la façon suivante :

- les lignes de rang pair seront constituées de l'entier 1,
- les lignes de rang impair seront constituées des entiers successifs 1, 2, 3, ...

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 1 & 1 & 1 \\ 5 & 6 & 7 & 8 \end{pmatrix}$$

Imprimer la matrice obtenue ligne par ligne afin de vérifier son contenu.

```fortran
program matrix
  implicit none
  integer                  :: n,m
  integer                  :: err,i,j,k
  integer, dimension(:,:),allocatable  :: mat

  print *,"Entrez␣le␣nombre␣de␣lignes␣:"
  read(*,*)n
  print *,"Entrez␣le␣nombre␣de␣colonnes␣:"
  read(*,*)m
  !
  allocate(mat(n,m),stat=err)
  if (err /= 0) then
    print *,"Erreur␣d'allocation"
    stop 4
  endif
  ! Remplissage des lignes de rang pair avec l'entier 1
  mat(2:n:2,:) = 1
  ! Remplissage des lignes de rang impair avec les entiers 1,2,...
  k=1
  do i=1,n,2
     do j=1,m
        mat(i,j)=k
        k = k+1
     end do
  end do
  ! On imprime la matrice obtenue apres remplissage
  do i = 1,n
     print *,mat(i,:)
  end do
  deallocate(mat)
```

1 Généralités

2 Déclarations & variables

3 Opérateurs et expressions

4 Structures de contrôles

5 Tableaux

6 Gestion de la mémoire

7 Types dérivés et modules
- Définition et déclaration de structures
- Symbole % d'accès à un champ
- Modules
- Exercice

8 Procédures

9 Entrées-Sorties

10 Fonctions intrinsèques

11 Pointeurs

12 Travaux pratiques

- Tableau : objet regroupant des données de même type repérées par un/des indices numériques.
- Nécessité de définir un objet composite (structure de données) regroupant des données (champs ou composantes) hétérogènes. Chaque champ est identifié par son nom. Sa déclaration nécessite la définition préalable du type dérivé étendant les types prédéfinis.

Exemple : stockage Yale Sparse Matrix

```
type YSM
  integer         :: nn,m
  real, dimension(:), allocatable  :: a
  integer, dimension(:)    :: ia
  integer, dimension(:)    :: ja
end type YSM
...
type(YSM)          :: matr
...
```

- Le symbole « % » permet d'accéder à un champ d'une structure de donnée.
  - matr : structure de données de type dérivé YSM
  - matr%nn : champ nn de la structure matr
  - matr%ia : tableau dynamique d'entiers

Exemple : matrice tridiagonale

```
type(YSM)          :: matr

matr%m = 100   ! nbre de lignes de la matrice
matr%nn = 3*(matr%m-2)+4  ! nbre d'elements non nuls
allocate(matr%ia(matr%m+1),matr%ja(matr%nn))
```

Modules

- Un module est une unité de programme particulière introduite en Fortran 90 pour encapsuler entre autres :
  - des données et des définitions de types dérivés,
  - des procédures (après l'instruction CONTAINS)
- Quel que soit le nombre d'accès (USE) au même module, les entités ainsi définies sont uniques.
- Doit être compilé séparément avant de pouvoir être utilisé. Le compilateur crée pour chaque fichier source :
  - un fichier objet de même nom suffixé par .o,
  - autant de fichiers nom_module.mod qu'il y a de modules
- Si un module fait appel (USE) à d'autres modules, ces derniers doivent avoir été précédemment compilés

Exemple : stockage Yale Sparse Matrix

```
module YSM_mod

  type YSM
    integer      :: nn,m
    real, dimension(:), allocatable  :: a
    integer, dimension(:)    :: ia
    integer, dimension(:)    :: ja
  end type YSM

contains

  subroutine YSMcreate(matr,m)

     type(YSM)        :: matr
     integer          :: m
     ...
  end subroutine YSMcreate

end module YSM_mod
```

Exercice

- Construire un module MyComplexMod contenant un type dérivé MyComplex composé de deux réels
- Créer un programme (dans un autre fichier) initiant 2 nombres complexes de type MyComplex
- Compiler, tester

```
module MyComplexMod

  implicit none

  type MyComplex
    real(kind=kind(1.d0))     :: xr, xi
  end type

end module MyComplexMod

program Complex
  use MyComplexMod
  implicit none

  type(MyComplex)    :: yc1, yc2

  yc1%xr  =  1.  ; yc1%xi  =  0.5
  yc2%xr=-2.6  ; yc2%xi  =  1.3

end program Complex
```

1 Généralités

2 Déclarations & variables

3 Opérateurs et expressions

4 Structures de contrôles

5 Tableaux

6 Gestion de la mémoire

7 Types dérivés et modules

8 Procédures
- Arguments
- Attributs INTENT et OPTIONAL
- Durée de vie et visibilité des identificateurs
- Subroutines et functions
- Interface explicite
- Exercice

9 Entrées-Sorties

10 Fonctions intrinsèques

11 Pointeurs

12 Travaux pratiques

- Les traitements répétitifs sont définis une seule fois à l'aide d'une unité de programme de type procédure (SUBROUTINE ou FUNCTION).
- Les différentes valeurs nécessaires au traitement sont transmises via des variables appelées arguments d'appel (actual-arguments). La procédure appelée récupère les valeurs qu'on lui a transmises via des variables appelées arguments muets (dummy-arguments).
- En Fortran le passage de ces valeurs s'effectue par référence :
  - les adresses des arguments d'appel sont transmises à la procédure appelée,
  - dans la procédure appelée, les arguments muets sont des alias des arguments d'appel

Argument de type chaîne de caractères

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

Argument de type tableau

- Lorsque l'on transmet un tableau en argument il est commode de transmettre également ses dimensions afin de pouvoir déclarer l'argument muet correspondant au sein de la procédure appelée
- Lorsqu'un tableau est passé en argument c'est l'adresse de son premier élément qui est transmise.
- De façon générale, supposons que l'on dispose d'un tableau tab à 2 dimensions constitué de n lignes et m colonnes. L'adresse de l'élément tab(i,j) est :

$$@tab(i,j) = @tab(1,1) + [n \times (j-1) + (i-1)] \times taille(element)$$

  Le nombre de colonnes m n'intervient pas dans ce calcul.

Attention

- D'une façon générale, si le tableau déclaré dans la procédure est de rang r, seules les r-1 premières dimensions sont nécessaires car la dernière n'intervient pas dans le calcul d'adresses
- On peut mettre un « * » à la place de la dernière dimension.
- A l'exécution de la procédure, les étendues de chaque dimension, hormis la dernière, seront connues mais, la taille du tableau ne l'étant pas, c'est au développeur de s'assurer qu'il n'y a pas de débordement.
- Ce type de tableau est dit à « taille implicite » (assumed-size-array).

Section de tableau non contiguë en argument d'une procédure

- Une section de tableau peut être passée en argument d'une procédure.
- Si elle constitue un ensemble de valeurs non contiguës en mémoire, le compilateur peut être amené à copier au préalable cette section dans un tableau d'éléments contigus passé à la procédure, puis en fin de traitement le recopier dans la section initiale ...
- En fait, cette copie (copy in–copy out) n'a pas lieu si les conditions suivantes sont réalisées :
  - la section passée est régulière,
  - l'argument muet correspondant est à profil implicite (ce qui nécessite que l'interface soit explicite).
- Sections non régulières en argument de procédures :
  - c'est une copie contiguë en mémoire qui est passée par le compilateur,
  - l'argument muet correspondant de la procédure ne doit pas avoir la vocation INTENT(inout) ou INTENT(out) ; autrement dit, en retour de la procédure, il n'y a pas mise-à-jour du tableau « père » de la section irrégulière.

Argument de type procédure

- Une procédure peut être transmise à une autre procédure. Il est nécessaire de la déclarer dans la procédure appelante avec l'attribut EXTERNAL ou INTRINSIC si elle est intrinsèque (c'est-à-dire fournie par le compilateur).

Un meilleur contrôle par le compilateur de la cohérence des arguments est possible en Fortran 90 à deux conditions :

1. améliorer la visibilité de la fonction appelée. Par exemple, en la définissant comme interne (CONTAINS). On parle alors d'interface « explicite ».
2. préciser la vocation des arguments muets de façon à pouvoir contrôler plus finement l'usage qui en est fait. Pour ce faire, Fortran 90 a prévu :
   - l'attribut INTENT d'un argument :
     - INTENT(in) : entrée seulement ;
     - INTENT(out) : sortie seulement (dans la procédure, l'argument muet doit être défini avant toute référence à cet argument) ;
     - INTENT(inout) : entrée et sortie.
   - l'attribut OPTIONAL pour déclarer certains arguments comme optionnels et pouvoir tester leur présence éventuelle dans la liste des arguments d'appel (fonction intrinsèque PRESENT).

Exemple

```
real, dimension(:), intent(in) :: vect
integer, optional, intent(out) :: rg_max
.....
if ( present(rg_max) ) then ...
```

- On appelle durée de vie d'un identificateur le temps pendant lequel il existe en mémoire.
- Il est visible s'il existe en mémoire et est accessible.
- Par défaut, une variable a une durée de vie limitée à celle de l'unité de programme dans laquelle elle a été définie.
- l'attribut SAVE permet de prolonger la durée de vie à celle de l'exécutable : on parle alors de variable permanente ou statique.
- Dans une unité de programme l'instruction SAVE sans spécification de liste de variables indique que toutes les variables de cette unité sont permanentes.
- Une compilation effectuée en mode static force la présence de l'instruction SAVE dans toutes les unités de programme, ce qui implique que toutes les variables sont permanentes.
- Par contre si elle est faite en mode stack, les variables permanentes sont :
  - celles pour lesquelles l'attribut SAVE a été précisé,
  - celles initialisées à la déclaration (via l'instruction DATA ou à l'aide du signe « = »).

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

- Une interface de procédure est constituée des informations nécessaires permettant la communication entre plusieurs unités de programmes :
  - type de la procédure : function ou subroutine
  - arguments de la procédure (arguments formels), avec leur type et leurs attributs
  - propriétés du résultat dans le cas d'une fonction.
- Par défaut, interface implicite : les arguments de l'appel de la procédure sont définis dans l'unité de programme appelant, et les arguments muets de la procédure sont définis dans la procédure.
- Dans le cas d'une procédure interne, la compilation de l'unité appelante et de la procédure se fait ensemble : contrôle de cohérence. Interface explicite avec une visibilité limitée de la procédure.
- Dans le cas de procédures externes, la compilation de l'unité appelante et de la procédure se fait séparément, les arguments sont passés par adresse : pas de contrôle de cohérence entre les arguments d'appel et les arguments muets. Nécessité d'une interface explicite.

5 possibilités :

1. procédures intrinsèques (Fortran 77 et Fortran 95),
2. procédures internes (CONTAINS),
3. présence du bloc interface dans la procédure appelante,
4. la procédure appelante accède (USE) au module contenant le bloc interface de la procédure appelée,
5. la procédure appelante accède (USE) au module contenant la procédure appelée.

Exemple

```
program inout
   implicit none
   integer, parameter  :: n =5
   integer             :: rgmax=0, control=0
   real, dimension (n) :: v =(/ 1. ,2. ,40. ,3. ,4. /)
   real                :: vmax, vmin
! ————————— Bloc interface —————————
   interface subroutine maxmin (vect,v_max,v_min,ctl,rg_max)
      real, dimension (:), intent(in)  :: vect
      real, intent(out)                :: v_max, v_min
      integer, optional, intent(out)   :: rg_max
      integer, intent(inout)           :: ctl
    end subroutine maxmin
   end interface
! ————————————————————————————————————————————
.......
call maxmin (v, vmax, vmin, control, rgmax)
.......
end program inout
```

Exemple, suite, dans une autre unité de programme

```
subroutine maxmin(vect, v_max, v_min, ctl, rg_max)
   implicit none

   real, dimension (:), intent(in)        :: vect
   real, intent(out)                      :: v_max, v_min
   integer, optional, intent(out)         :: rg_max
   integer, intent(inout)                 :: ctl

   v_max = MAXVAL(vect)
   v_min = MINVAL(vect)
   ctl = 1
   if (present (rg_max)) then
      rg_max = MAXLOC(vect,DIM =1)
      ctl = 2
   endif
   print *, " Taille vecteur via size : ", SIZE(vect)
   print *, " Profil vecteur via shape : ", SHAPE(vect)
end subroutine maxmin
```

Intérêt des interfaces explicites :

- la transmission du profil et de la taille des tableaux à profil implicite et la possibilité de les récupérer via les fonctions SHAPE et SIZE,
- la possibilité de contrôler la vocation des arguments en fonction des attributs INTENT et OPTIONAL, en particulier l'interdiction de passer en argument d'appel une constante (type PARAMETER ou numérique) si l'argument muet correspondant a la vocation OUT ou INOUT,
- la possibilité de tester l'absence des arguments optionnels (fonction PRESENT),
- la détection des erreurs liées à la non cohérence des arguments d'appel et des arguments muets (type, attributs et nombre) ; conséquence fréquente d'une faute de frappe, de l'oubli d'un argument non optionnel ou de l'interversion de deux arguments.

Exercice

A partir du module MyComplexMod, ajouter les fonctions addition, soustraction, multiplication et division entre 2 nombres complexes :

$$(a + bi) + (x + yi) = (a + x) + (b + y)i$$

$$(a + bi) - (x + yi) = (a - x) + (b - y)i$$

$$(a + bi) \times (x + yi) = (a \times x - b \times y) + (a \times y + b \times x)i$$

$$\frac{(a + bi)}{(x + yi)} = \frac{(a + bi) \times (x - yi)}{x^2 + y^2}$$

```
module MyComplexMod
...
  contains
  function addition(yc1,yc2)
    type(MyComplex),intent(in)      :: yc1,yc2
    type(MyComplex)                 :: addition

    addition%xr = yc1%xr+yc2%xr
    addition%xi = yc1%xi+yc2%xi
  end function addition
  ...
end module MyComplexMod
program Complex
  use MyComplexMod
  implicit none

  type(MyComplex)   :: yc1,yc2,yc3

  yc1%xr = 1.  ; yc1%xi = 0.5
  yc2%xr=-2.6  ; yc2%xi = 1.3

  yc3=addition(yc1,yc2)
  print *,'Complex = (',yc3%xr,',',yc3%xi,')'
  ...
end program Complex
```

1 Généralités

2 Déclarations & variables

3 Opérateurs et expressions

4 Structures de contrôles

5 Tableaux

6 Gestion de la mémoire

7 Types dérivés et modules

8 Procédures

9 Entrées-Sorties
- Introduction
- Accès séquentiel
- Formats
- Accès direct
- Inquire
- Unités standards
- Exercice

10 Fonctions intrinsèques

11 Pointeurs

12 Travaux pratiques

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

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

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

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

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

Fichier binaire

Fichier dans lequel on stocke les informations telles qu'elles sont représentées en mémoire. C'est au moment de l'ouverture du fichier que l'on indique le type de fichier à traiter.

Exemple

```
real , dimension(200)   :: tab
integer                 :: i,ios
real                    :: r
open(UNIT=1, FILE=''data_bin_seq'',FORM=''unformatted'', &
   ACCESS=''sequential'',ACTION=''read'',POSITION=''rewind'', IOSTAT=ios)
if (ios /= 0) stop ''pb a l'ouverture''
read␣(unit=1,iostat=ios)␣tab,i,r
do␣while␣(ios␣==␣0)
␣␣...
␣␣read(unit=1,iostat=ios)␣tab,i,r
end␣do
close(unit=1)
```

- Ouverture du fichier de nom data_bin_seq, fichier binaire séquentiel, qu'on veut lire depuis le début (rewind).
- Fichier connecté à l'unité logique 1.
- En cas d'erreur à l'ouverture, en l'absence du mot clé iostat, le programme s'interrompt. Si le mot-clé est précisé :
  - si la valeur est 0, tout s'est bien passé
  - sinon, une erreur s'est produite
  - le même mot-clé est utilisé pour l'instruction read.

Une opération de conversion est nécessaire au sein des instructions READ/WRITE (paramètre FMT) car dans un fichier texte (non binaire), les données sont stockées sous forme de caractères.

- Descripteur I pour le type INTEGER,
- Descripteurs F, E pour le type REAL,
- Descripteur L pour le type LOGICAL,
- Descripteur A pour le type CHARACTER.

Exemple

```
WRITE (UNIT = 1, FMT = "(10F8␣.4,I3,F6.3)")  tab,i,r
```

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

- Format réel : de forme générale : Fw.d, Ew.d ou bien Dw.d.
- Le nombre réel à lire peut être soit en notation virgule fixe, soit exponentielle avec, dans ce dernier cas, l'exposant préfixé de la lettre E ou D.
- Avec les formats Ew.d ou bien Dw.d, en écriture, on obtiendra les motifs :
  - $\underbrace{S0.\underbrace{XXXXXXXXX}_{d}ESXX}_{w}$
  - $\underbrace{S0.\underbrace{XXXXXXXXX}_{d}DSXX}_{w}$

  avec S : position pour le signe

- En Fortran il existe un format implicite appelé format libre (list-directed formatting). Dans l'instruction READ/WRITE, on spécifie alors le caractère * à la place du format.
- Dans ce contexte, les enregistrements sont interprétés comme une suite de valeurs séparées par des caractères appelés séparateurs. C'est le type des variables auxquelles ces valeurs vont être affectées qui détermine la conversion à effectuer.
- Les caractères interprétés comme des séparateurs sont :
  - la virgule (,)
  - le blanc (espace)
- Une chaîne de caractères contenant un caractère séparateur doit être délimitée soit par des quotes (') soit par des guillemets (").
- En entrée, plusieurs valeurs identiques peuvent être regroupées à l'aide d'un facteur de répétition sous la forme n*valeur.

- A la différence d'un fichier séquentiel, il est possible d'accéder à un enregistrement d'un fichier à accès direct sans avoir traité les précédents.
- Chaque enregistrement est repéré par un numéro qui est son rang dans le fichier.
- Leur taille est fixe.
- Au sein de l'instruction OPEN :
  - le paramètre RECL= est obligatoire, sa valeur indique la taille des enregistrements (en caractères pour les fichiers textes, dépend du processeur pour les fichiers binaires),
  - Si le paramètre FORM n'est pas précisé, c'est la valeur unformatted qui est prise en compte.
- Le rang de l'enregistrement que l'on désire traiter doit être spécifié à l'aide du paramètre REC= de l'instruction READ/WRITE.
- Un enregistrement ne peut pas être détruit mais par contre il peut être réécrit.

Exemple

```
open(unit=1, file=''data_bin_direct'', access=''direct'',  &
   action=''read'', status=''old'', recl=''400'')
n = 100
read(unit=1, rec=n, iostat=ios) tab
```

- Le fichier dont le nom est data_bin_direct est connecté à l'unité logique numéro 1
- C'est un fichier binaire à accès direct (ACCESS="direct" et paramètre FORM absent donc considéré égal à unformatted).
- Chaque enregistrement fait 400 octets (RECL=400)
- On accède à l'enregistrement de rang n (qui vaut 100)
- Le paramètre IOSTAT de l'instruction READ permet de récupérer l'état de la lecture dans l'entier ios : une valeur non nulle positive signale une erreur du type enregistrement inexistant par exemple.

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

Exemple

```
logical existe
character(len=3)    :: form
character(len=10)   :: acces
inquire(file=''data_txt_seq'', exist=existe)
if (existe) then
   open(unit=1, file=''data_txt_seq'', position=''rewind'', &
      action=''read'', iostat=ios)
   if (ios /= 0) then ! erreur ouverture
      ...
   else
      inquire(unit=1, formatted=form, access=acces)
   endif
   close(1)
end if
```

- Les variables form et acces prendront respectivement les valeurs YES et SEQUENTIAL (si le fichier existe)

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

Exercice

- Ecrire un programme initialisant une matrice et la stockant sur un fichier binaire
- Ecrire un programme relisant ce fichier binaire et lui appliquant une transformation à partir d'un choix de l'utilisateur appliqué à chaque élément de la matrice sous la forme $y = f(x)$:
  - identité
  - carré
  - racine
  - logarithme

1. Généralités
2. Déclarations & variables
3. Opérateurs et expressions
4. Structures de contrôles
5. Tableaux
6. Gestion de la mémoire
7. Types dérivés et modules
8. Procédures
9. Entrées-Sorties
10. Fonctions intrinsèques
    - Mesure de temps, nombres aléatoires
    - Exercice
11. Pointeurs
12. Travaux pratiques

- CPU_TIME(time) : sous-progr. retournant dans le réel time le temps CPU en secondes (ou réel < 0 si indisponible). Par différence entre deux appels, il permet d'évaluer la consommation CPU d'une section de code.
- DATE_AND_TIME(date,time,zone,values) : sous-programme retournant dans les variables (de type caractères) date et time, la date et l'heure en temps d'horloge murale. L'écart par rapport au temps universel est retourné optionnellement dans zone. Toutes ces informations sont aussi stockées sous forme d'entiers dans le vecteur values.
- SYSTEM_CLOCK(count,count_rate,count_max) : sous-programme retournant dans des variables entières la valeur du compteur de périodes d'horloge (count), le nombre de périodes/sec. (count_rate) et la valeur maximale de ce compteur (count_max)
- RANDOM_NUMBER(harvest) : sous-progr. retournant un/plusieurs nombres pseudo-aléatoires compris entre 0. et 1. dans un scalaire/tableau réel passé en argument (harvest).
- RANDOM_SEED(size,put,get) : sous-programme permettant de ré-initialiser une série de nombres aléatoires. Tous les arguments sont optionnels.

Exemple : évaluation du temps CPU et du temps d'horloge (elapsed time)

```
INTEGER  :: cpt_init ,& ! Val. init. compteur periodes horloge
            cpt_fin , & ! Val. finale compteur periodes horloge
            cpt_max , & ! Val. maximale du compteur d'horloge
            freq ,    & ! Nb. de periodes d'horloge par seconde
            cpt         ! Nb. de periodes d'horloge du code
REAL     :: temps_elapsed , t1 , t2 , t_cpu
...
! Initialisations
CALL SYSTEM_CLOCK(COUNT_RATE=freq , COUNT_MAX=cpt_max)
CALL SYSTEM_CLOCK(COUNT=cpt_init)
CALL CPU_TIME(TIME=t1)
...  ! << Partie du code a evaluer >>
CALL CPU_TIME (TIME=t2)
CALL SYSTEM_CLOCK(COUNT=cpt_fin)
!
cpt = cpt_fin - cpt_init
IF (cpt_fin < cpt_init) cpt = cpt + cpt_max
temps_elapsed = REAL(cpt)/freq
t_cpu = t2 - t1
!
print * , " Temps elapsed = ", temps_elapsed, " sec."
print * , " Temps CPU = ", t_cpu ," sec ."
```

Calcul de π

Compléter le programme qui calcule le nombre π à partir de l'algorithme suivant :

$$\pi = \int_0^1 f(x)dx \quad \text{with } f(x) = \frac{4}{1+x^2}$$

$$\pi = \frac{1}{n}\sum_{i=1}^{\infty} f(x_i) \quad \text{with } x_i = \frac{i-\frac{1}{2}}{n}$$

- Evaluation du temps de calcul grâce aux fonctions intrinsèques

Remarque

- CPU time : Temps pendant lequel le CPU est actif pour l'exécution. Cela ne prend donc pas en compte notamment les temps d'attente des I/O.
- Elapsed time : Ces temps d'attente sont inclus.

CPU time $<$ elapsed time

1 Généralités

2 Déclarations & variables

3 Opérateurs et expressions

4 Structures de contrôles

5 Tableaux

6 Gestion de la mémoire

7 Types dérivés et modules

8 Procédures

9 Entrées-Sorties

10 Fonctions intrinsèques

11 Pointeurs
- Définitions
- Déclaration
- Symbole =>
- Symbole = appliqué aux pointeurs
- Allocation dynamique de mémoire
- Fonction NULL() et instruction NULLIFY
- Fonction intrinsèque ASSOCIATED
- Passage d'un pointeur en argument de procédure

12 Travaux pratiques

- En C, Pascal : variable contenant l'adresse d'objets
- En Fortran 90 : alias d'objets

Tout pointeur Fortran a un état parmi les suivants :

1. Indéfini : à sa déclaration en tête de programme
2. Nul : alias d'aucun objet
3. Associé : alias d'un objet (cible)

Remarque

Pour ceux connaissant la notion de pointeur (type langage C), le pointeur Fortran 90 peut être vu comme une abstraction de niveau supérieur en ce sens qu'il interdit la manipulation directe d'adresse.
À chaque pointeur Fortran 90 est associé un « descripteur interne » contenant les caractéristiques (type, rang, état, adresse de la cible, et même le pas d'adressage en cas de section régulière, etc.).
Pour toute référence à ce pointeur, l'indirection est faite pour vous, d'où la notion d'« alias ».

Exemple

```
real, pointer                         :: p1
integer, dimension(:), pointer        :: p2
character(len=80), pointer            :: p4(:)
```

Attention

p4 n'est pas un « tableaux de pointeurs » !
C'est en fait un pointeur susceptible d'être ultérieurement associé à un tableau de rang 1 et de type « chaîne de caractères ». Préférer l'écriture :

```
character(len=80), dimension(:), pointer :: p4
```

- Cible : c'est un objet pointé. Cet objet devra avoir l'attribut target lors de sa déclaration.

```
integer, target :: i
```

- Le symbole « => » sert à valoriser un pointeur.

```
integer, target :: n
integer, pointer :: ptr1, ptr2
n = 10
ptr1 => n
ptr2 => ptr1
n = 20
print *, ptr2     ! output : ?
```

Remarque

p1 et p2 étant deux pointeurs, « *p2 => p1* » implique que p1 prend l'état de p2 (indéfini, nul ou associé à la même cible).

Attention

lorsque les opérandes du symbole = sont des pointeurs, l'affectation s'effectue sur les cibles et non sur les pointeurs.

Exemple

```
integer, pointer :: ptr1, ptr2
integer, target  :: i1, i2
i1 = 1 ; i2 = 2
ptr1 => i1 ; ptr2 => i2
ptr2 = ptr1
! equivalent a "i2 = i1"
print *, ptr2
```

- L'instruction ALLOCATE permet d'associer un pointeur et d'allouer dynamiquement de la mémoire.

Exemple

```
implicit none
integer, dimension(:,:), pointer :: p
integer                          :: n
read(*,*) n
ALLOCATE(p(n,n))
print *, p
DEALLOCATE (p)
```

- L'espace alloué n'a pas de nom, on y accède par l'intermédiaire du pointeur
- Pour libérer la place allouée on utilise l'instruction DEALLOCATE
- Après l'exécution de l'instruction DEALLOCATE le pointeur passe à l'état nul.
- L'instruction DEALLOCATE appliquée à un pointeur dont l'état est indéterminé provoque une erreur

- Au début d'un programme un pointeur n'est pas défini : son état est indéterminé.
- La fonction intrinsèque NULL permet de forcer un pointeur à l'état nul (y compris lors de sa déclaration).

```
real , pointer , dimension (:) :: p1 => NULL ()
...
p1 => NULL ()
...
```

- L'instruction NULLIFY permet de forcer un pointeur à l'état nul :

```
real, pointer :: p1 , p2
nullify (p1)
nullify (p2)
```

- Si deux pointeurs p1 et p2 sont alias de la même cible, NULLIFY(p1) force le pointeur p1 à l'état nul, par contre le pointeur p2 reste alias de sa cible.
- Si p1 est à l'état nul, l'instruction « *p2 => p1* » force p2 à l'état nul.

- Il n'est pas possible de comparer des pointeurs, c'est la fonction intrinsèque ASSOCIATED qui remplit ce rôle :
  - ASSOCIATED(p) : vrai si p est associé à une cible, faux si p est à l'état nul
  - ASSOCIATED(p1, p2) : vrai si p1 et p2 sont alias de la même cible, faux sinon
  - ASSOCIATED(p1, c) : vrai si p1 est alias de la cible c faux sinon
- Le deuxième argument optionnel peut être au choix une cible ou un pointeur
- Le pointeur ne doit pas être dans l'état indéterminé
- Si p1 et p2 sont à l'état nul alors ASSOCIATED(p1,p2) renvoie faux.

- L'argument muet n'a pas l'attribut pointer :
  - le pointeur doit être associé avant l'appel
  - c'est l'adresse de la cible associée qui est passée
- L'argument muet a l'attribut pointer :
  - le pointeur n'est pas nécessairement associé avant l'appel
  - c'est l'adresse du descripteur du pointeur qui est passée
  - l'interface doit être explicite (pour que le compilateur sache que l'argument muet a l'attribut pointer)
  - si le pointeur passé est associé à un tableau avant l'appel, les bornes inférieures/supérieures de chacune de ses dimensions sont transmises à la procédure ; elles peuvent alors être récupérées via les fonctions UBOUND/LBOUND

1 Généralités
2 Déclarations & variables
3 Opérateurs et expressions
4 Structures de contrôles
5 Tableaux
6 Gestion de la mémoire
7 Types dérivés et modules
8 Procédures
9 Entrées-Sorties
10 Fonctions intrinsèques
11 Pointeurs
12 Travaux pratiques

Problème étudié

On souhaite résoudre l'équation aux dérivées partielles suivante

$$\begin{cases} -\Delta u = f \quad \text{dans} \quad \Omega = [a,b] \times [c,d] \\ u = g \quad \text{sur} \quad \partial\Omega \end{cases}$$

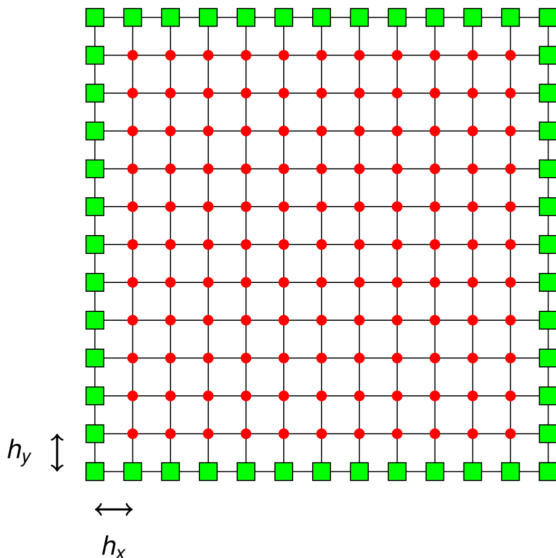
Discrétisation de l'intervalle $[a,b]$ suivant l'axe des x

$$\begin{aligned} h_x &= \frac{b-a}{n_x+1} \quad (n_x + 2 \text{ points}) \\ x(i) &= x_i = a + ih_x, \quad i = 0, \cdots, n_x + 1 \end{aligned}$$

Discrétisation de l'intervalle $[c,d]$ suivant l'axe des y

$$\begin{aligned} h_y &= \frac{d-c}{n_y+1} \quad (n_y + 2 \text{ points}) \\ y(j) &= y_j = c + jh_y, \quad j = 0, \cdots, n_y + 1 \end{aligned}$$

Domaine discrétisé



Méthode des différences finies

$$\frac{\partial^2 u}{\partial x^2}(x_i, y_j) = \frac{u(x_{i+1}, y_j) - 2u(x_i, y_j) + u(x_{i-1}, y_j)}{h_x^2} + O(h_x^2)$$

Méthode des différences finies

$$\frac{\partial^2 u}{\partial x^2}(x_i, y_j) = \frac{u(x_{i+1}, y_j) - 2u(x_i, y_j) + u(x_{i-1}, y_j)}{h_x^2} + O(h_x^2)$$

$$\frac{\partial^2 u}{\partial y^2}(x_i, y_j) = \frac{u(x_i, y_{j+1}) - 2u(x_i, y_j) + u(x_i, y_{j-1})}{h_y^2} + O(h_y^2)$$

Méthode des différences finies

$$\frac{\partial^2 u}{\partial x^2}(x_i, y_j) = \frac{u(x_{i+1}, y_j) - 2u(x_i, y_j) + u(x_{i-1}, y_j)}{h_x^2} + O(h_x^2)$$

$$\frac{\partial^2 u}{\partial y^2}(x_i, y_j) = \frac{u(x_i, y_{j+1}) - 2u(x_i, y_j) + u(x_i, y_{j-1})}{h_y^2} + O(h_y^2)$$

$$-\Delta u(x_i, y_j) = -\frac{\partial^2 u}{\partial x^2}(x_i, y_j) - \frac{\partial^2 u}{\partial y^2}(x_i, y_j)$$

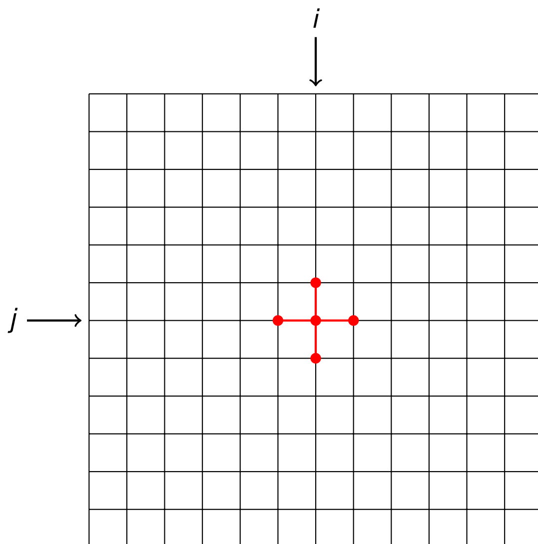
Méthode des différences finies

$$\frac{\partial^2 u}{\partial x^2}(x_i, y_j) = \frac{u(x_{i+1}, y_j) - 2u(x_i, y_j) + u(x_{i-1}, y_j)}{h_x^2} + O(h_x^2)$$

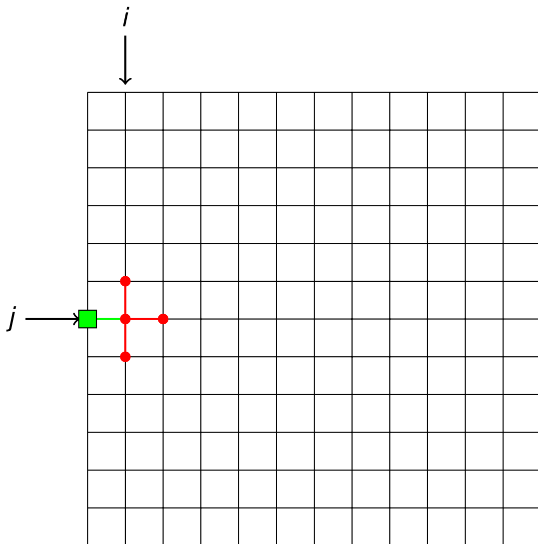
$$\frac{\partial^2 u}{\partial y^2}(x_i, y_j) = \frac{u(x_i, y_{j+1}) - 2u(x_i, y_j) + u(x_i, y_{j-1})}{h_y^2} + O(h_y^2)$$

$$-\Delta u(x_i, y_j) \approx \frac{-u(x_{i+1}, y_j) + 2u(x_i, y_j) - u(x_{i-1}, y_j)}{h_x^2} + \frac{-u(x_i, y_{j+1}) + 2u(x_i, y_j) - u(x_i, y_{j-1})}{h_y^2}$$

Méthode des différences finies



Méthode des différences finies



Système linéaire Ax = b

$$A = \begin{bmatrix} A_1 & A_2 & 0 & \cdots & 0 \\ A_2 & A_1 & A_2 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & A_2 & A_1 & A_2 \\ 0 & \cdots & 0 & A_2 & A_1 \end{bmatrix}$$

avec

$$A_1 = \begin{bmatrix} \frac{2}{h_x^2} + \frac{2}{h_y^2} & -\frac{1}{h_x^2} & 0 & \cdots & 0 \\ -\frac{1}{h_x^2} & \frac{2}{h_x^2} + \frac{2}{h_y^2} & -\frac{1}{h_x^2} & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & -\frac{1}{h_x^2} & \frac{2}{h_x^2} + \frac{2}{h_y^2} & -\frac{1}{h_x^2} \\ 0 & \cdots & 0 & -\frac{1}{h_x^2} & \frac{2}{h_x^2} + \frac{2}{h_y^2} \end{bmatrix}$$

Système linéaire Ax = b

$$A = \begin{bmatrix} A_1 & A_2 & 0 & \cdots & 0 \\ A_2 & A_1 & A_2 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & A_2 & A_1 & A_2 \\ 0 & \cdots & 0 & A_2 & A_1 \end{bmatrix}$$

avec

$$A_2 = \begin{bmatrix} -\frac{1}{h_y^2} & 0 & 0 & \cdots & 0 \\ 0 & -\frac{1}{h_y^2} & 0 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & 0 & -\frac{1}{h_y^2} & 0 \\ 0 & \cdots & 0 & 0 & -\frac{1}{h_y^2} \end{bmatrix}$$

Système linéaire Ax = b

$$b = \begin{bmatrix} f(x_1, y_1) + \frac{g(x_0, y_1)}{h_x^2} + \frac{g(x_1, y_0)}{h_y^2} \\ f(x_2, y_1) + \frac{g(x_1, y_0)}{h_y^2} \\ \vdots \\ f(x_i, y_j) \\ \vdots \\ f(x_{n_x}, y_{n_y}) + \frac{g(x_{n_x+1}, y_{n_y})}{h_x^2} + \frac{g(x_{n_x}, y_{n_y+1})}{h_y^2} \end{bmatrix}$$

Stockage de la matrice A

On utilisera un stockage creux appelé CSR (compressed sparse row). Sa structure de données peut être représentée par

- val : tableau représentant les valeurs non nulles de la matrice.
- col_ind : tableau représentant les indices des colonnes où on trouve les valeurs non nulles.
- row_ptr : liste des indices où commence chacune des lignes .

Exemple CSR

$$A = \begin{pmatrix} 10 & 0 & 0 & 0 \\ 3 & 9 & 0 & 0 \\ 0 & 7 & 8 & 7 \end{pmatrix}$$

$$val = [10, 3, 9, 7, 8, 7]$$

$$col_ind = [0, 0, 1, 1, 2, 3]$$

$$row_ptr = [0, 1, 3, 6]$$

Résolution du système

- méthode itérative : gradient conjugué

Travaux pratiques

1. A partir des sources proposées, structurer le répertoire de façon classique (*src*, *test*) et créer les fichiers *CMakeLists.txt* qui permettront de compiler le programme.
2. Créer le type dérivé *CSR*.
3. Définir les bonnes étendues de tableaux pour la définition des conditions aux limites (routine *setDirichlet*).
4. Calculer le résidu dans la routine du gradient conjugué à partir des fonctions intrinsèques.
5. Compiler et tester le code
6. Ajouter l'appel à la fonction C++ *vtkTools.cxx* pour la sortie graphique de la solution.