
Fortran : historique rapide

- Fortran I (1954-1957) : 1er langage de haut niveau
 - Fortran II (1958) : fonctions, sous-routines (procédures)
 - Fortran IV (1961) : nettoyage du langage
 - Fortran 66 : standard international
 - Fortran 77 : structures de contrôles
 - Fortran 90 : influence C/C++, types de données utilisateurs, structures de contrôles
 - Fortran 95 : modifications mineures
 - Fortran 2003 : interface avec C, orientation objet
 - Fortran 2008 : extensions parallèles (concurrent do, co-array)
-

Les compilateurs fortran récents sont au niveau Fortran 95, et *partiellement*, aux niveaux Fortran 2003/2008.

Fortran 2003 status in Fortran Wiki

<http://fortranwiki.org/fortran/show/Fortran+2003+status>

Fortran 2003 status in Fortran Wiki

<http://fortranwiki.org/fortran/show/Fortran+2003+status>



Fortran Wiki

Fortran 2003 status

[Home Page](#) | [All Pages](#) | [Recently Revised](#) | [Authors](#) | [Feeds](#) | [Export](#) |

Compiler Support for the Fortran 2003 Standard

Fortran 2003 features	Absoft	Cray	GNU	g95	HP	IBM	Intel	NAG	Oracle	PathScale	PGI
ISO TR 15580	Y	Y	N	P	Y	Y	Y	Y	Y	Y	Y
IEEE Arithmetic											
ISO TR 15581											
Allocatable Enhancements	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Data enhancements and object orientation											
Parameterized derived types	N	Y	N	N	N	Y	N	N	N	N	Y
Procedure pointers	N	Y	Y	Y	Y	Y	Y	Y	N	N	Y
Finalization	N	Y	N	N	N	Y	Y	Y	N	N	Y
Procedures bound by name to a type	N	Y	Y	N	N	Y	Y	Y	N	N	Y
The PASS attribute	N	Y	Y	N	Y	Y	Y	Y	N	N	Y
Procedures bound to a type as operators	N	Y	Y	N	N	Y	Y	Y	N	N	Y
Type extension	N	Y	Y	N	N	Y	Y	Y	N	N	Y
Overriding a type-bound procedure	N	Y	Y	N	N	Y	Y	Y	N	N	Y
Enumerations	N	Y	Y	Y	N	Y	Y	Y	N	N	Y
ASSOCIATE construct	N	Y	P	N	N	Y	Y	Y	N	N	Y
Polymorphic entities	N	Y	P (1)	N	N	Y	Y	Y	N	N	Y
SELECT TYPE construct	N	Y	P	N	N	Y	Y	Y	N	N	Y

Fortran 2003 features	Absoft	Cray	GNU	g95	HP	IBM	Intel	NAG	Oracle	PathScale	PGI
Deferred bindings and abstract types	N	Y	Y	N	N	Y	Y	Y	N	N	Y
Allocatable scalars	N	Y	Y	?	N	Y	Y	Y	?	N	Y
Allocatable character length	N	Y	P	?	N	Y	Y	Y	?	N	Y
Miscellaneous enhancements											
Structure constructors	N	Y	Y	Y	N	Y	Y	Y	N	N	Y
The allocate statement	N	Y	Y	P	N	Y	Y	Y	N	N	Y
Assignment to an allocatable array	N	Y (2)	Y	N	Y	Y	Y (2)	Y	N	N	Y (2)
Transferring an allocation	N	Y	Y	N	N	Y	Y	Y	N	N	Y
More control of access from a module	N	Y	Y	N	N	Y	Y	Y	Y	N	Y
Renaming operators on the USE statement	Y	Y	P	Y	N	Y	Y	Y	Y	N	Y
Pointer assignment	N	Y	Y	Y	N	Y	Y	Y	N	N	Y
Pointer INTENT	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	Y
The VOLATILE attribute	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
The IMPORT statement	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Intrinsic modules	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Access to the computing environment	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Support for international character sets	N	P (19)	Y	Y	N	P	P (19)	Y	N	N	N
Lengths of names and statements	N	Y	Y	?	Y	Y	Y	Y	Y	Y	Y
Binary, octal and hex constants	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

Fortran features 2003	Absoft	Cray	GNU	g95	HP	IBM	Intel	NAG	Oracle	PathScale	PGI
Array constructor syntax	N	Y	Y	Y	Y	Y	Y	Y	N	Y	Y
Specification and initialization expressions	N	Y	P	Y	Y	Y	P	Y	N	N	Y
Complex constants	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Changes to intrinsic functions	N	Y	P (9)	Y	Y	Y	Y	Y	N	?	Y
Controlling IEEE underflow	Y	Y	N	N	Y	Y	Y	P	Y	Y	Y
Another IEEE class value	Y	Y	N	N	Y	Y	Y	Y	Y	Y	Y
Input/output enhancements	Absoft	Cray	GNU	g95	HP	IBM	Intel	NAG	Oracle	PathScale	PGI
Derived type input/output	N	Y	N	N	N	Y	N	N	N	N	N
Asynchronous input/output	N	Y	Y (10)	Y	N	Y	Y	Y	Y	N	Y
FLUSH statement	N	Y	Y	Y	N	Y	Y	Y	Y	Y	Y
IOMSG= specifier	N	Y	Y	Y	Y	Y	Y	Y	Y	N	Y
Stream access input/output	N	Y	Y	Y	N	Y	Y	Y	Y	N	Y
ROUND= specifier	Y	Y	P (30)	P	Y	Y	Y	Y	Y	N	Y (20, 30)
DECIMAL= specifier	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	Y (21)
SIGN= specifier	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	Y (22)
Kind type parameters of integer specifiers	N	Y	N	?	N	Y	Y	Y	N	N	Y
Recursive input/output	N	Y	Y	Y	N	Y	Y	Y	Y	Y	N
Intrinsic function for newline character	N	Y	Y	Y	Y	Y	Y	Y	N	Y	Y

Fortran features 2003	Absoft	Cray	GNU	g95	HP	IBM	Intel	NAG	Oracle	PathScale	PGI
Input and output of IEEE exceptional values	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Comma after a P edit descriptor	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Interoperability with C	Absoft	Cray	GNU	g95	HP	IBM	Intel	NAG	Oracle	PathScale	PGI
Interoperability of intrinsic types	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Interoperability with C pointers	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Interoperability of derived types	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Interoperability of variables	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Interoperability of procedures	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Interoperability of global data	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	Y

Legend: Y = Yes, N = No, P = Partial, U = Unconfirmed

Footnotes: (1) No unlimited polymorphic; (2) Optional under flag; (9) `kind=` of `maxloc`, `minloc`, `shape` missing; (10) implemented as synchronous I/O; (18) `move_alloc`; (19) `selected_char_kind` only; (20) plus RC, RD, RN, RP, RU, RZ; (21) plus BLANK=, DELIM=, PAD=, SIZE=; (22) plus DC,DP; (30) only for output.

Changes

Changes between December 2010 and April 2011:

- Cray supports ISO TR 15580 IEEE arithmetic, renaming operators on the USE statement, controlling IEEE underflow, another IEEE class value, and the ROUND= specifier.
- GNU supports allocatable character length and partially supports assignment to an allocatable array.
- HP supports ISO TR 15580 IEEE arithmetic and lengths of names and statements.
- IBM supports more control of access from a module, renaming operators on the USE

Fortran 90/95 : Notions de base

Exemple de programme fortran

Fichier ex1.f90 :

```
1 program exemple
2 include 'sub.h'
3 real , dimension(6) :: x
4 call sub4(x)
5 write (*,*) x
6 end program exemple
```

Points à remarquer :

- le programme principal commence par :

`program nom_du_programme`

et se termine par :

`end program nom_du_programme`

(équivalent du `main` C ou C++),

- il y a 2 parties : une première partie avec les déclarations de variables, une seconde avec les instructions du programme,
 - pas de différence majuscule/minuscule :
 - `I` et `i` désignent la même variable,
 - `write` et `WriTe`, la même instruction
-

-
- on n'est pas obligé de déclarer les variables utilisées, par défaut une variable qui commence par `I`, `J`, ..., `N` est de type entier, sinon réel.

Il est très fortement conseillé d'utiliser l'instruction `implicit none` qui désactive cette règle.

- `read/write` sont les instructions d'entrées/sorties
-

Déclaration de variables (types simples)

Forme générale :

`type[, liste d'attributs ::] liste de variables`

où les types possibles sont :

- `integer`
 - `real`
 - `double precision`
 - `complex`
 - `character`
 - `logical`
 - `type` (types utilisateurs)
-

quelques attributs possibles :

- `parameter` (pour les constantes)
- `dimension` (pour les vecteurs/matrices)

(autres attributs possibles pour la gestion de la mémoire dynamique, le type de passage d'arguments dans les fonctions, etc)

Exemples :

```
1 integer :: n
2 real , dimension(100) :: x, y
3 complex , dimension(-2:4, 0:5) :: c
4 integer , parameter :: m = 4, p = 10
5 character(len=5) , dimension(10) :: s
```

Exemples de constantes de différents types :

```
1 1          ! entier
2 1.0        ! reel
3 5.0e4      ! reel (notation scientifique)
4 1.0d0      ! reel (double precision)
5 (3.0 , 4.0) ! complexe
6 'abcd'
7 .T.        ! booleen (.T.: vrai ou .F.:faux)
```

Étendue et précision des nombres

On peut spécifier plus précisément le type de nombres entiers, réels ou complexes à utiliser :

```
1 integer , parameter :: prec = &  
2     selected_real_kind(p=9, r=50)  
3 integer , parameter :: iprec = &  
4     selected_int_kind(r=3)  
5  
6 integer(kind=iprec) :: n1, n2  
7 real(kind=prec) :: a, b  
8 complex(kind=prec), dimension(5) :: comp
```

`selected_real_kind(p, r)` désigne le type de réels capables de représenter des valeurs x telles que

$$10^{-r} < |x| < 10^r$$

avec p chiffres significatifs,

`selected_integer_kind(r)` désigne le type d'entiers capables de représenter des valeurs n telles que

$$|n| < 10^r$$

Ces fonctions retournent un entier (négatif si le type demandé n'existe pas).

`real(kind=<entier>)` et `integer(kind=<entier>)` précisent les types de réel et d'entier désirés.

Boucles

Boucle numérique

```
do variable = expr1, expr2 [, expr3]  
  bloc d'instructions  
end do
```

où `expr1`, `expr2`, `expr3` sont des expressions à valeur entière.

`expr1` valeur initiale de la variable d'indice

`expr2` valeur finale ...

`expr3` incrément (positif ou négatif) ...

Exemple :

```
1 do i = 1, 10  
2   write (*,*) i, i*i  
3 end do
```

Boucle conditionnelle :

```
1 do while (condition)
2     bloc d'instructions
3 end do
```

Boucle "infinie"

```
1 do
2     bloc d'instructions 1
3     if (condition) exit
4     bloc d'instructions 2
5 end do
```

Exécution conditionnelle

```
1 if (condition1) then  
2     bloc d'instructions _1  
3 else if (condition2) then  
4     _ _ _ bloc d'instructions 2  
5 else  
6     bloc d'instructions _3  
7 end if
```

```
1 select case (variable)
2 case(1, 3, 7)
3     bloc d'instructions
4 case(12:17)
5     _ _ bloc _d'instructions
6 case(2)
7     bloc d'instructions
8 case _ default
9     _ _ bloc _d'instructions
10 end select
```

Entrées/sorties en fortran90

Les entrées/sorties sont repérées des unités (entiers de 1 à 99).

Numéros réservés :

- 5: entrée standard (clavier)
- 6: sortie standard (écran)
- 7: sortie d'erreur

Les autres numéros sont utilisables pour les entrées/sorties sur fichiers.

Pour écrire une information sur un fichier ou sur une sortie standard :

```
1 write ([unit=] <entier> | variable chaine de caracteres | *, &  
2         [fmt=] <entier> | <chaine de caracteres> | *, &  
3         [iostat=variable entiere]) liste d'expressions
```

Exemples :

```
1 integer :: i , j  
2 real , dimension(17) :: X  
3 character(len=12) :: s  
4 write (* , *) ' i _ = _ ' , i  
5 write(unit = 34, fmt = "(6E12.7)") X  
6 write(s , '(A2,I2,A1)') , '(I', 6, ')'  
7 write (9 ,*) s  
8 write (* ,s) j
```

A la fin du `write`, on passe à la ligne suivante.

Pour lire une information dans un fichier ou de l'entrée standard :

```
1 read([ unit=] <entier> | variable chaine de caracteres | *, &  
2      [ fmt=] <entier> | <chaine de caracteres> | *, &  
3      [ iostat=variable entiere]) liste de variables
```

Exemples :

```
1  integer :: i , j  
2  real , dimension(3) :: X  
3  character( len=12) :: s  
4  read( *, *) i  
5  read( unit = 35, fmt = *) X  
6  write( s , '(A2,I2,A1)') , '(I', 6, ')'  
7  read (* , fmt=s) j
```

Le paramètre de format `fmt=` :

- `*` : format d'entrée ou de sortie par défaut,
- une constante entière qui se réfère à une instruction de format (liste de spécifications d'entrées/sorties, voir, par exemple, <http://www.cs.mtu.edu/~shene/COURSES/cs201/NOTES/format.html>)
- une chaîne de caractères (variable ou constante) analogue à la liste ci-dessus (y compris les parenthèses)

Exemples :

```
1  integer i , j
2  double precision x
3  write (* , 12) i , j , x
4  12 format(2I3 , E12.4)
5  write (* , '(I3 , _I5 , _F14.7) ') i , j , x
```

Pour ouvrir un fichier

```
1 open ([ unit=] <entier> , &  
2       file= <chaine de caracteres> , &  
3       status= 'old' | 'new' | 'replace' , &  
4       [iostat=variable entiere])
```

Le paramètre `status=` permet de spécifier si le fichier à ouvrir existe ou s'il faut créer un fichier vide.

Pour fermer un fichier

```
1 close ([ unit=] <entier> )
```

Tableaux (vecteurs, matrices)

Ensemble d'éléments du même type. Pour déclarer un tableau, on utilise l'attribut `dimension` :

```
1 integer , dimension(50) :: i
2 real , dimension(10, -3:4, 1:10) :: x
```

Un tableau peut avoir jusqu'à 7 dimensions.

Les caractéristiques d'un tableau sont :

- le rang : nombre de dimensions
 - l'étendue (extent) : nombre d'éléments dans une dimension
 - le profil (shape) : l'ensemble des étendues
 - la taille (size) : nombre total d'éléments
-

Les fonctions `size(a)` et `shape(a)` donnent la taille et le profil du tableau `a`.

Deux tableaux sont dit conformes s'ils ont les mêmes profils.

Exemple :

```
1 real , dimension( -5:4 , 0:2 ) :: x  
2 real , dimension( 1:10 , 1:3 ) :: y
```

Les 2 tableaux ont le même profil : 10×3 éléments. Ils sont donc conformes.

Opérations sur les tableaux

Pour utiliser une composante d'un tableau :

```
1 real , dimension(-5:4, 0:2) :: x, y, z
2 x(3, 1) = 2.0
3
4 do i=-5,4
5     do j=0,2
6         z(i, j) = x(i, j) - y(i, j)
7     end do
8 end do
```

Fortran90 offre des fonctionnalités beaucoup plus efficaces (à la manière de matlab) pour travailler sur des tableaux.

Exemples :

```
1 real , dimension (6 ,7) :: a
2 real , dimension (2:7 , 5:11) :: b
3
4 a = 1.5
5 b = 2.0 + 3.0*a
6 write (* ,*) b
```

Attention, dans la dernière ligne, on ajoute 2 à chaque composante de la matrice (et pas $2 \times$ la matrice identité).

Sections de tableaux

Exemple : on a le tableau T défini par :

1 **real , dimension (9 ,5) :: T**

On peut en extraire des sections régulières :

⌈	⌈			
⌈	⌈			
⌈	⌈			
⌈	⌈			
⌈	⌈			

T(3:7, 1:2)

⌈		⌈		⌈
⌈		⌈		⌈
⌈		⌈		⌈
⌈		⌈		⌈
⌈		⌈		⌈
⌈		⌈		⌈
⌈		⌈		⌈
⌈		⌈		⌈
⌈		⌈		⌈

T(:, 1:5:2)

	⌈	⌈	⌈	

T(4:4, 2:4)

Fonctions utilitaires sur les tableaux

Ces fonctions agissent sur l'ensemble des éléments ou sur l'ensemble des lignes ou colonnes.

- **sum** : somme de coefficients

Exemple :

```
1 real , dimension (3 ,5) :: A
```

```
1 s = sum(A)
```

s est un réel (la somme de tous les coefficients de **A**)

```
1 t = sum(A, 1)
```

t est un vecteur dont chaque composante est la somme des coefficients sur une colonne

```
1 t = sum(A, 2)
```

`t` est un vecteur dont chaque composante est la somme des coefficients sur une ligne

- `product` : produit de coefficients
- `minval` : minimum des coefficients
- `maxval` : maximum des coefficients

Les fonctions qui suivent travaillent sur des tableaux de booléens

- `all` : vrai si tous les coefficients concernés sont vrais
 - `any` : vrai si au moins un des coefficients concernés est vrai
 - `count` : comptage des valeurs vraies
-

`matmul(a, b)` effectue le produit de 2 tableaux de rang 1 ou 2 (au moins un des tableaux est de rang 2), à condition que le produit soit possible,

`dot_product(a, b)` calcule le produit scalaire de 2 tableaux `a` et `b` de même taille et de rang 1

`transpose(a)` construit un nouveau tableau en transposant le tableau `a` de rang 2

`reshape(a, shape)` construit un nouveau tableau avec les coefficients de `a` et le profil `shape`.

where effectue des traitement sur de tableau (coefficient par coefficient) en tenant compte d'un masque (tableau de booléens avec le même profil).

Exemple :

```
1 where (a > 0.0)
2           c = 2*a + b
3 else where
4           c = 3*a + b
5 end where
```

Fonctions et procédures (sous-routines)

Une fonction prend des arguments et renvoie un résultat.

Le code d'une fonction s'écrit

```
1 function f(<liste d'arguments>)  
2 type _f _ _ ! _type _du _resultat _de _la _fonction  
3 types _des _arguments  
4 code _de _la _fonction  
5 f _ = _ ...  
6 end _function _f
```

Pour appeler une fonction :

```
1 variable = f(<liste de variables>)
```

Une procédure (sous-routine) prend des arguments mais ne renvoie pas de résultat.

Le code d'une procédure s'écrit :

```
1 subroutine sub(<liste d'arguments>)  
2   type _des _arguments  
3   code _de _la _sous-routine  
4   end _subroutine _sub
```

Pour appeler une procédure :

```
1 call sub(<liste de variables>)
```

Mode de passage des arguments :

- `intent(in)` : valeur en entrée (la fonction/sous-routine ne peut pas modifier la valeur),
- `intent(out)` : valeur en sortie (la fonction/sous-routine doit d'abord initialiser la valeur, qui est récupérée à la sortie),
- `intent(inout)` : valeur en entrée/sortie (la fonction/sous-routine peut utiliser et modifier la valeur, qui est récupérée à la sortie).

Par défaut, le mode `intent(inout)` est utilisé, mais il est (fortement) conseillé de spécifier explicitement le mode de passage des arguments.

Des arguments peuvent être déclarés **optional** (optionels). Dans ce cas, avant de les utiliser dans la fonction/sous-routine, il faut tester que l'argument est présent lors de l'appel :

```
1 function f(x, y)
2 real, intent(in) :: x
3 real, intent(in), optional :: y
4 real :: f
5 if (present(y)) then
6     f = x+y
7 else
8     f = x
9 end if
10 end function f
```

Lors de l'appel (s'il y a ambiguïté), on peut préciser quels sont les arguments qui sont passés :

₁ `val = f (x = mon_x, y = mon_y)`

₂ `val2 = f (x = mon_x)`

Exemple :

```
1 program test
2 real :: x, y, z, val
3 real, dimension(3) :: grad
4
5 x = 1.0; y = 2.0; z = 3.0
6 val = f(x, y, z)
7 call df(x, y, z, grad)
8 end program test
9
10 real function f(x, y, z)
11 real, intent(in) :: x, y, z
12 f = x*y*sin(z)
13 end function f
```

```
14
15 subroutine df(x, y, z, grad)
16 real, intent(in) :: x, y, z
17 real, intent(out), dimension(3) :: grad
18 grad(1) = y*sin(z)
19 grad(2) = x*sin(z)
20 grad(3) = x*y*cos(z)
21 end subroutine df
```

Notion d'interface

Dans le programme (plus généralement, dans la fonction ou sous-programme) qui appelle une fonction/sous-programme, on conseille de déclarer l'interface.

Une interface est l'analogue d'un prototype C/C++ : description du nom du sous-programme/fonction et de ses paramètres.

En général, on recopie le code de la fonction sans les instructions exécutables et sans la déclaration des variables locales.

Exemple :

```
1 interface  
2   subroutine sub4(x)  
3     implicit none  
4     real , dimension (:) :: x  
5     end subroutine sub4  
6 end interface
```

Ce n'est pas obligatoire sauf dans certains cas, par exemple, si on passe des tableaux en argument avec un profil implicite.

Pratiquement :

- mettre les interfaces dans un ou plusieurs fichiers .h
- utiliser l'instruction `include 'xxx.h'` chaque fois que ces sous-programmes/fonctions sont appelés.

Exemple

- programme principal :

```
1 program exemple
2 include 'sub.h'
3 real , dimension(6) :: x
4 call sub4(x)
5 write (* , *) x
6 end program exemple
```

-
- interface dans le fichier sub.h :

```
1 interface  
2   subroutine sub4(x)  
3   implicit none  
4   real , dimension (:) :: x  
5   end subroutine sub4  
6 end interface
```

- implémentation des sous-routines dans le fichier sub.f90 :

```
1 subroutine sub4(x)  
2 implicit none  
3 real , dimension (:) :: x  
4 x(size(x, 1)/2) = 1.0  
5 end subroutine sub4
```

Passage des tableaux en argument

Dans une fonction (ou une sous-routine), quand on passe un tableau, on a le choix de :

- spécifier le profil explicitement :

```
1 subroutine sub3 ( x )  
2 integer , intent ( out ) , dimension ( 3 ) :: x  
3 do i = 1 , 3  
4     x ( i ) = ...
```

La fonction/sous-routine ne pourra être appelée que pour des tableaux d'une taille précise.

- utiliser le profil de la variable passée en argument (voir plus loin la notion d'interface)
-

```
1 subroutine sub4(x)
2 integer , intent(out) , dimension (: , :) :: x
3 do i=1 , size(x,1)
4     x(i , i) = ...
```

- passer les dimensions en paramètres supplémentaires (moyen utilisé pour les codes fortran77)

```
1 subroutine sub5(x , n)
2 integer , intent(in) :: n
3 integer , intent(out) , dimension(n) :: x
4 do i=1 , n
5     x(i , i) = ...
```

Partage d'informations entre fonctions / sous-routines / programme principal

Pour passer de l'information à une sous-routine/fonction, on peut tout faire passer par les arguments d'appel.

En *fortran 77* : conduit très rapidement à un grand nombre de paramètres dans la définition des sous-routines/fonctions.

Pas de vérification entre l'appel de la définition des fonctions ou sous-programmes en *fortran 77* \implies difficile de gérer des longues listes de paramètres.

\implies utilisation de zones mémoire partagées : les **COMMON**

Exemple :

main.f

```
1      program test
2      implicit none
3      real eta , nu , alpha , beta , rho
4      common/physics/ eta ,nu ,alpha ,beta ,rho
5      real u(5)
6      call init()
7      call stress(u)
8      write (* ,*) u
9      end
```

init.f

```
1      subroutine init ()  
2      real eta , nu , alpha , beta , rho  
3      common / physics / eta , nu , alpha , beta , rho  
4      eta = 1.0  
5      rho = 100.0  
6      return  
7      end
```

stress.f

```
1      subroutine stress(u)
2      real u(5)
3      real eta , nu , alpha , beta , rho
4      common / physics / eta , nu , alpha , beta , rho
5      u(1) = eta * rho*rho
6      return
7      end
```

Mêmes possibilités d'erreur de cohérence avec l'utilisation de commons.

Il est fortement conseillé de mettre les commons dans des fichier .h et de les inclure dans **toutes les fonctions et sous-programmes** qui accèdent aux commons.

Solution proposée par fortran 90 : notion de module.

Modules fortran 90

Module : regroupement de constantes, variables, fonctions et sous-programmes

Exemple de définition de module :

```
1 module m
2 integer :: p
3 real , parameter :: q = 3.5
4 contains
5 function f(x)
6 real :: f , x
7 f = 3*x
8 end function f
9 end module m
```

Utilisation de ce module :

```
1 program main  
2 use m  
3 write (*,*) q  
4 write (*,*) f(4.5)  
5 end program main
```

Quelques références sur le fortran

Sites internet :

- Cours de fortran90 :
Large <http://cch.loria.fr/documentation/documents/F95/F95.html>
- Cours de fortran90 :
http://www.idris.fr/data/cours/lang/fortran/choix_doc.html
- Cours de fortran77 :
<http://perso.enstimac.fr/~gaborit/lang/CoursDeFortran/Fortran.html>

Livres :

- Manuel complet du langage Fortran 90 et Fortran 95 (Lignelet), Masson, 1996
-

-
- Fortran 95/2003 Explained, (Metcalf, Reid, Cohen), Oxford University Press, 2004
 - Fortran 90/95 for Scientists and Engineers (Chapman), McGraw Hill, 2004
-