

Module et Types dérivés en Fortran 95

Vers une programmation objet

Jalel.Chergui@limsi.fr

MCours.com

1 – Historique

3

1 – Historique

- ☞ 1954, naissance de Fortran (John Backus, IBM).
- ☞ 1956, Fortran I (nom des variables jusqu'à 6 caractères et introduction de l'instruction **format**).
- ☞ 1957, Fortran II (premier compilateur, notions de sous-programme et de fonction).
- ☞ 1958, Fortran III est disponible mais ne sort pas de chez IBM (type logique, paramètres et prototypage des procédures).
- ☞ 1962, Fortran IV est le langage des scientifiques mais il faut établir une norme.
- ☞ 1966, Fortran IV est rebaptisé Fortran 66 (norme ANSI : American National Standards Institut).
- ☞ 1978, c'est la fin des cartes perforées. La norme Fortran 77 (ou Fortran V) modernise le langage. On passe d'une programmation symbolique à une programmation structurée.

Modules et types dérivés en Fortran 95 : plan

1 – Historique	2
2 – Introduction	4
3 – Bibliographie	7
4 – Généralités	8
5 – Cas d'exemple	10
5.1 – Sans module	11
5.2 – Avec module	12
5.3 – Quelques avantages	13
6 – Interface générique	15
7 – Vers une programmation objet	17
7.1 – Notion de type dérivé	18
7.2 – Application	19
8 – Quelques apports de Fortran 2003	27
9 – Conclusion	28

1 – Historique

4

Après 30 ans d'existence, Fortran n'est plus le seul langage disponible mais reste très utilisé dans le domaine du calcul scientifique. Pour maintenir ce langage, une nouvelle norme est élaborée.

- ☞ 1991, Fortran 90 apparaît (norme internationale ISO/ANSI). Fortran devient modulaire et plus fiable. Il s'oriente vers une programmation objet (modules, récursivité, surcharge des opérateurs, types dérivés, etc.).
- ☞ 1995, Fortran 95 définit les instructions dépréciées ou obsolètes et introduit la structure **forall** et quelques autres extensions.
- ☞ 2002, Fortran 2000 (fin du processus de normalisation et publication de la norme ISO prévue fin 2004). Fortran supporte désormais la programmation objet.
- ☞ 2005, correction de Fortran 2000 et adoption de la norme dite « Fortran 2003 ».
- ☞ A ce jour (mars 2009), la norme 2003 n'est que partiellement implémentée dans les compilateurs Fortran libres ou du marché.

2 – Introduction

Fortran est utilisé depuis plus d'un demi siècle dans des domaines scientifiques variés :

- ☞ climat,
- ☞ mécanique des fluides,
- ☞ physique,
- ☞ chimie.

- ☞ **Format libre du code source.**
- ☞ Longueur des identificateurs de variables jusqu'à 31 caractères.
- ☞ Commentaire sur une ligne d'instruction.
- ☞ Possibilités d'opérer sur des tableaux (ou section de tableau).
- ☞ Procédures récursives.
- ☞ **Grouper les procédures et les données au sein d'un module.**
- ☞ **Profonde amélioration du mécanisme de passage d'arguments dans une procédure permettant leur vérification à la compilation.**
- ☞ **Interface des procédures génériques.**
- ☞ Surcharge d'opérateur.
- ☞ **Type dérivés.**
- ☞ **Nouvelle syntaxe pour la déclaration des types de variables.**
- ☞ Allocation dynamique et pointeurs.
- ☞ Structure des boucles et sélection conditionnelle **select ... case.**
- ☞ Portabilité et contrôle de la précision numérique.
- ☞ Nouvelles procédures **intrinsic.**

La norme Fortran 90 a introduit des extensions majeures à Fortran 77. Elle apporte des solutions efficaces (parfois encore partielles) concernant :

- ☞ la portabilité des codes sources,
- ☞ la fiabilité (précision numérique, contrôle d'erreurs par le compilateur),
- ☞ la performance du code assembleur généré par le compilateur,
- ☞ certaines contraintes de développement en génie logiciel.

Quelles sont ces extensions ?

3 – Bibliographie

Ouvrages :

- ☞ *Fortran 90/95 Explained* de Michael Metcalf & John K. Reid « *This book is concerned with the Fortran programming language (Fortran 90 and Fortran 95), setting out a reasonably concise description of the whole language ...* ». Ed. Oxford University Press.
- ☞ *Manuel complet du langage Fortran 90 et Fortran 95. Calcul intensif et génie logiciel* de Patrice Lignelet. Ed. Masson.

Liens :

- ☞ http://www.idris.fr/data/cours/lang/fortran/fortran_base/IDRIS_Fortran_cours.pdf
- ☞ <http://www.obs.u-bordeaux1.fr/radio/JMHure/CoursF95CMichaut.pdf>
- ☞ http://www.ocea.eu/pichon/Cours_F95.pdf

Compilateurs :

- ☞ http://www.fortranplus.co.uk/fortran_info.html
- ☞ http://en.wikipedia.org/wiki/Fortran#External_links

4 – Généralités

Un programme source Fortran est composé de parties indépendantes appelées unités de programme (*scoping unit*). Une unité de programme peut être :

- ☞ un programme principal (pouvant contenir des procédures internes),
- ☞ une procédure externe (sous-programme ou fonction),
- ☞ un module.

Chaque unité comprend une partie déclarative suivie d'une partie comportant des instructions exécutables.

MCours.com

5 – Cas d'exemple

Il s'agit de définir une bibliothèque de procédures permettant la résolution d'un système linéaire $A \times x = B$ par différentes méthodes itératives.

Programme principal :

```
program amoi
! Déclarations
! Instructions

contains ! Procé. internes

subroutine A(...)
...
end subroutine A

function B(...)
...
end function B

end program amoi
```

Procédure externe :

```
subroutine aelle(...)
! Déclarations
! Instructions

contains ! Procé. internes

subroutine C(...)
...
end subroutine C

function D(...)
...
end function D

end subroutine aelle
```

Module :

```
module alui
! Déclarations
...

contains

subroutine E(...)
...
end subroutine E

function F(...)
...
end function F

end module alui
```

Regardons de plus près quelques apports importants liés à l'utilisation des **modules**...

5.1 – Sans module

alui.f90 : procédures

```
subroutine gmres(..., N, A, B, x)
implicit none
integer, parameter :: N
real*8, dimension(N,N) :: A
real*8, dimension(N) :: B, x
...
end subroutine gmres

subroutine mgrid(..., N, A, B, x)
implicit none
integer, parameter :: N
real*8, dimension(N,N) :: A
real*8, dimension(N) :: B, x
...
end subroutine mgrid
```

amoi.f90 : utilisation

```
program amoi
implicit none
integer, parameter :: N=11
real*8, dimension(N,N) :: Au, Av
real*8, dimension(N) :: Su, Sv
real*8, dimension(N) :: u, v
...

call gmres(..., N, Au, Su, u)
...
call mgrid(..., N, Av, Sv, v)
...
end program amoi
```

```
> ifort -c amoi.f90
> ifort -c alui.f90
> ifort -o amoi.x amoi.o alui.o
```

5.2 – Avec module

alui_mod.f90 : module

```

module alui
implicit none
public
integer, parameter :: d=selected_real_kind(15)

contains
subroutine gmres(..., A, B, x)
implicit none
real(kind=d), intent(in), dimension(:,): A
real(kind=d), intent(in), dimension(:) :: B
real(kind=d), intent(out), dimension(:) :: x
...
end subroutine gmres

subroutine mgrid(..., A, B, x)
implicit none
real(kind=d), intent(in), dimension(:,): A
real(kind=d), intent(in), dimension(:) :: B
real(kind=d), intent(out), dimension(:) :: x
...
end subroutine mgrid
...
end module alui

```

amoi.f90 : utilisation

```

program amoi
use alui
implicit none
integer, parameter :: N=11
real(kind=d), dimension(N,N) :: Au, Av
real(kind=d), dimension(N) :: Su, Sv
real(kind=d), dimension(N) :: u, v
...

call gmres(..., Au, Su, u)
...
call mgrid(..., Av, Sv, v)
...
end program amoi

> ifort -c alui_mod.f90
> ifort -c -I. amoi.f90
> ifort -o amoi.x amoi.o alui_mod.o

```

5.3 – Quelques avantages

- La possibilité de détecter, à la **compilation**, les erreurs liées à la non-cohérence des arguments d'appels et des arguments muets.
- Plus besoin d'indiquer en argument les dimensions des tableaux. La transmission du profile et de la taille des tableaux est implicite et peuvent être connus grâce aux fonctions respectivement **shape** et **size**.
- La vocation des arguments est contrôlée en fonction de l'attribut **intent** (et **optional**).
- Le contrôle de la visibilité des variables et des procédures en fonction des instructions (ou attributs) **public** et **private**.
- Le passage des arguments d'appels aurait pu se faire par mot-clé (voir exemple suivant).

5 – Cas d'exemple : quelques avantages

Un mot-clé n'est autre que le nom donné à l'argument muet d'une procédure définie dans un **module** ou un bloc **interface**.

```

program amoi
use alui
implicit none
integer, parameter :: N=11
real(kind=d), dimension(N,N) :: Au, Av
real(kind=d), dimension(N) :: Su, Sv
real(kind=d), dimension(N) :: u, v
...

call gmres(..., B=Su, A=Au, X=u)
...
call mgrid(..., X=v, B=Sv, A=Av)
...
end program amoi

```

6 – Interface générique

6 – Interface générique

alui_mod.f90 : module

```

module alui
implicit none
public
integer, parameter :: d=selected_real_kind(15)

interface solve
module procedure gmres, mgrid
end interface solve

private :: gmres, mgrid

contains

subroutine gmres(tol, relax, maxiter, &
                 maxcomp, A, B, x)
...
end subroutine gmres

subroutine mgrid(tol, maxiter, maxlevel, A, B, x)
...
end subroutine mgrid
...
end module alui

```

Les procédures **gmres** et **mgrid**, définies précédemment, peuvent être appelées par un nom générique (ici **solve**).

amoi.f90 : utilisation

```

program amoi
use alui
implicit none
integer, parameter :: N=11
real(kind=d), dimension(N,N) :: Au, Av
real(kind=d), dimension(N) :: Su, Sv
real(kind=d), dimension(N) :: u, v
...

call solve(TOL=1.D-12, RELAX=1.9_d, &
           MAXITER=100, MAXCOMP=5, &
           A=Au, B=Su, X=u)
...

call solve(TOL=1.D-12, MAXITER=25, &
           MAXLEVEL=10, A=Av, B=Sv, X=v)
...
end program amoi

```

Dans l'exemple précédent, nous avons :

- créé une interface générique publique nommée `solve`,
- restreint avec l'attribut `private` la visibilité des procédures `gmres` et `mgrid` qui ne pourront désormais être appelées d'une unité externe que via leur interface générique `solve`.

Ainsi, le concepteur du module peut se réserver le droit de changer le nom des procédures privées sans impacter le programme utilisateur.

Cependant, il peut être amené à modifier le nombre et le type des arguments muets au gré des mises à jour du module ...

7 – Vers une programmation objet

Jusqu'à présent, toute modification des arguments muets relatifs aux procédures (`gmres` et `mgrid`) par le concepteur du module impacterait nécessairement l'unité de programme appelante (utilisateur).

7 – Vers une programmation objet : type dérivé 19

7.1 – Notion de type dérivé

- Un type dérivé permet de définir un nouveau type.
- Un type dérivé est défini dans la partie déclarative d'une unité de programme.
- Les composantes d'un type dérivé sont des identificateurs de types quelconques (prédéfinis ou dérivés).
- L'accès à une composante d'un type dérivé se fait via l'opérateur `%`.

```
module alui
integer,parameter:: d=selected_real_kind(15)
type gmres
real(kind=d) :: tol=1.D-12, relax=1.85_d
integer :: maxiter=50
integer :: maxcomp=10
end type gmres
end module alui
```

```
program amoi
use alui

implicit none
type(gmres) :: gm

print *, "avant : ", gm%maxcomp
gm%maxcomp=20
print *, "apres : ", gm%maxcomp
end program amoi
```

```
> ifort -c alui_mod.f90
> ifort -c -I. amoi.f90
> ifort -o amoi.x amoi.o alui_mod.o
> ./amoi.x
avant :      10
apres :      20
```

7 – Vers une programmation objet : application 20

7.2 – Application

Un contrôle accru sur l'interface générique nécessite :

- ① l'encapsulation des paramètres de contrôle (`tol`, `maxiter`, etc.) de chacune des procédures de résolution dans un type dérivé semi-privé (type public dont les composantes sont privées),
- ② de permettre à l'utilisateur l'accès aux composantes privées via une procédure générique publique (dite « méthode » en langage objet).

MCours.com

Encapsulation

- Les paramètres de contrôle (tol, relax, maxiter, etc.) des méthodes de résolution sont encapsulés dans les types dérivés semi-privés gmres et mgrid.
- Des valeurs par défaut sont assignées aux composantes privés.

```
alui_mod.f90 : module
module alui
public
integer, parameter :: &
    d=selected_real_kind(15)

interface solve
module procedure pgmres, pmgrid
end interface solve
private :: pgmres, pmgrid

type gmres
private
real(kind=d) :: tol=1.D-12, relax=1.85_d
integer :: maxiter=50
integer :: maxcomp=10
...
end type gmres

type mgrid
private
real(kind=d) :: tol=1.D-12
integer :: maxiter=50
integer :: maxlevel=10
...
end type mgrid
```

Encapsulation (suite ...)

- L'interface des procédures privées de résolution est modifiée.
- L'introduction de l'objet solver en argument permet de réduire à 4 le nombre d'arguments muets.
- A noter que les noms des procédures privées, chargées de la résolution des systèmes linéaires, ont changé.

Suite...

```
contains
subroutine pgmres(solver, A, B, x)
implicit none
real(kind=d), intent(in), dimension(:, :) :: A
real(kind=d), intent(in), dimension(:) :: B
real(kind=d), intent(out), dimension(:) :: x
type(gmres), intent(inout) :: solver
integer :: it
...
do it=1, solver%maxiter
...
end do
end subroutine pgmres
subroutine pmgrid(solver, A, B, x)
implicit none
real(kind=d), intent(in), dimension(:, :) :: A
real(kind=d), intent(in), dimension(:) :: B
real(kind=d), intent(out), dimension(:) :: x
type(mgrid), intent(inout) :: solver
integer :: it
...
do it=1, solver%maxlevel
...
end do
end subroutine pmgrid
end module alui
```

Encapsulation (suite ...)

- L'appel des procédures de résolution est réalisé via leur interface générique solve.
- Les objets gm et mg, déclarés respectivement de types gmres et mgrid, permettent au compilateur de choisir l'une ou l'autre procédure de résolution.

Des méthodes (procédures publiques facilitant la manipulation globale d'objets privés ou semi-privés) permettront de modifier les valeurs par défaut des composantes des types gmres et mgrid.

amoi.f90 : utilisation

```
program amoi
use alui
implicit none
integer, parameter :: N=11
real(kind=d), dimension(N,N) :: Au, Av
real(kind=d), dimension(N) :: Su, Sv
real(kind=d), dimension(N) :: u, v
type(gmres) :: gm
type(mgrid) :: mg
...
call solve(SOLVER=gm, A=Au, B=Su, X=u)
...
call solve(SOLVER=mg, A=Av, B=Sv, X=v)
...
end program amoi
```

Procédure « méthode »

- Les procédures privées pgmres.set et pmgrid.set permettent la manipulation des composantes privées des types publiques gmres et mgrid via leur interface générique publique solve.set.

alui_mod.f90 : module

```
module alui
public
integer, parameter :: &
    d=selected_real_kind(15)

interface solve
module procedure pgmres, pmgrid
end interface solve
private :: pgmres, pmgrid

interface solve_set
module procedure pgmres_set, &
    pmgrid_set
end interface solve_set
private :: pgmres_set, pmgrid_set
...
contains

subroutine pgmres(solver, A, B, x)
...
end subroutine pgmres
subroutine pmgrid(solver, A, B, x)
...
end subroutine pmgrid
```

Suite du module...

Procédure « méthode » (suite...)

- ⇒ L'objectif est de pouvoir appeler ces procédures en passant les arguments d'appel par mot clé pour changer une valeur par défaut ou antérieure d'une ou plusieurs composantes des types `gmres` et `mgrid`.
- ⇒ Si l'argument, supposé optionnel, est présent alors affecter sa nouvelle valeur à la composante correspondante du type dérivé.

```

subroutine pgmres_set(solver, tol, relax, &
                     maxiter, maxcomp)
    implicit none
    real(kind=d),intent(in), optional :: tol,relax
    integer, intent(in), optional :: maxiter
    integer, intent(in), optional :: maxcomp
    type(gmres), intent(inout) :: solver
    if(present(tol)) solver%tol=tol
    if(present(relax)) solver%relax=relax
    if(present(maxiter)) solver%maxiter=maxiter
    if(present(maxcomp)) solver%maxcomp=maxcomp
end subroutine pgmres_set

subroutine pmgrid_set(solver, tol, maxiter, &
                     maxlevel)
    implicit none
    real(kind=d),intent(in), optional :: tol
    integer, intent(in), optional :: maxiter
    integer, intent(in), optional :: maxlevel
    type(mgrid), intent(inout) :: solver
    if(present(tol)) solver%tol=tol
    if(present(maxiter)) solver%maxiter=maxiter
    if(present(maxlevel)) solver%maxlevel=maxlevel
end subroutine pmgrid_set
...
end module alui

```

Procédure « méthode » (suite...)

Le reste n'a de limite que celle de notre imagination...

- ⇒ Gérer une composante dynamique privés dans un type dérivé (un compteur, un tableau dynamique, un pointer sur un autre type dérivé, etc.
- ⇒ Définir les « méthodes » associées pour gérer ces nouveaux objets (les allouer, les initialiser, en extraire une valeur, les détruire, etc.)

Procédure « méthode » (suite...)

- ⇒ Par exemple, modifier les paramètres `maxcomp` et `maxlevel` avant résolution par `gmres` et `mgrid`.

amoi.f90 : utilisation

```

program amoi
  use alui
  implicit none
  integer, parameter :: N=11
  real(kind=d), dimension(N,N) :: Au, Av
  real(kind=d), dimension(N) :: Su, Sv
  real(kind=d), dimension(N) :: u, v
  type(gmres) :: gm
  type(mgrid) :: mg
  ...
  call solve_set(SOLVER=gm, MAXCOMP=5)
  call solve(SOLVER=gm, A=Au, B=Su, X=u)
  ...
  call solve_set(SOLVER=mg, MAXLEVEL=15)
  call solve(SOLVER=mg, A=Av, B=Sv, X=v)
  ...
end program amoi

```

8 – Quelques apports de Fortran 2003

Parmi les nombreuses extensions qui font aujourd'hui de Fortran un Langage Orienté Objet :

- ⇒ Type dérivé : paramétrisation, extension, héritage, polymorphisme, etc.
- ⇒ Module : notion de sous-module (expression d'un même module dans des unités de programmes différentes), permet de définir des bibliothèques de grande taille, de préserver le contenu secret pour des raisons commerciales par exemple, etc.

9 – Conclusion

- ☞ Les modules offrent la possibilité de regrouper une famille de procédure sous un nom générique.
- ☞ A l'appel, le choix de la procédure à exécuter est fait automatiquement par le compilateur en fonction du nombre et du type des arguments.
- ☞ Le concepteur d'un module a la possibilité de cacher (rendre non visible) certaines variables et/ou procédures définies à l'intérieur de celui-ci.
- ☞ Les types dérivés est un moyen permettant d'encapsuler des données.
- ☞ La privatisation de certaines données conduit le concepteur à fournir au développeur des « méthodes » (procédures publiques) facilitant la manipulation d'objets privés ou semi-privés.
- ☞ La **documentation des ressources publiques d'un module** est un aspect important de la programmation objet.

