APPRENEZ

Fortran

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#fortran

Table des matières

À propos ... 1

Chapitre 1: Démarrer avec Fortran ... 2

Remarques ... 2

Versions ... 2

Examples ... 2

Installation ou configuration ... 2

Bonjour le monde ... 3

Équation quadratique ... 4

Insensibilité à la casse ... 5

Chapitre 2: C interopérabilité ... 6

Examples ... 6

Appeler C depuis Fortran ... 6

C structs dans Fortran ... 7

Chapitre 3: Contrôle de l'exécution ... 8

Examples ... 8

Si construire ... 8

SELECT CASE construit ... 9

Bloc DO construire ... 10

O construct construire ... 12

Chapitre 4: Des alternatives modernes aux caractéristiques historiques ... 14

Examples ... 14

Types de variables implicites ... 14

Arithmétique si déclaration ... 15

Constructions DO non bloquantes ... 16

Retour alternatif ... 16

Formulaire source fixe ... 18

Blocs communs ... 19

GOTO attribué ... 21

GOTO calculé ... 22

Spécificateurs de format assignés ... 22

Fonctions de déclaration ... 23

Chapitre 5: Extensions du fichier source (.f, .f90, .f95, ...) et leur relation avec le co 25
Introduction 25
Examples 25
Extensions et Signification 25
Chapitre 6: I / O 27
Syntaxe 27
Examples 27
Simple I / O 27
Lire avec quelques vérifications d'erreur 27
Passer des arguments de ligne de commande 28
Chapitre 7: Interfaces explicites et implicites 31
Examples 31
Sous-programmes internes / modules et interfaces explicites 31
Sous-programmes externes et interfaces implicites 32
Chapitre 8: Procédures - Fonctions et sous-programmes 34
Remarques 34
Examples 34
Syntaxe de la fonction 34
Déclaration de retour 35
Procédures récursives 35
L'intention des arguments factices 36
Référencement d'une procédure 37
Chapitre 9: Procédures intrinsèques 40
Remarques 40
Examples 40
Utilisation de PACK pour sélectionner des éléments répondant à une condition 40
Chapitre 10: Programmation orientée objet 42
Examples 42
Définition de type dérivée 42
Procédures de type 42
Types dérivés abstraits 43
Extension de type 44

Type constructeur 45
Chapitre 11: Tableaux 47
Examples 47
Notation de base 47
Tableaux pouvant être alloués 48
Constructeurs de tableaux 48
Spécification de la nature du tableau: rang et forme 51
Forme explicite 51
Forme supposée 51
Taille supposée 52
Forme différée 52
Forme implicite 52
Tableaux entiers, éléments de tableau et sections de tableau 53
Tableaux entiers 53
Éléments de tableau 53
Sections de tableau 53
Composants de tableaux de tableaux 54
Opérations de tableau 55
Addition et soustraction 55
Fonction 55
Multiplication et division 56
Opérations matricielles 56
Sections de tableau avancées: triplets en indice et indices de vecteur 56
Triplets d'indices 57
Indices de vecteur 57
Sections de tableau de rang supérieur 58
Chapitre 12: Types de données 59
Examples 59
Types intrinsèques 59
Types de données dérivés 60
Précision des nombres à virgule flottante 61
Paramètres de type de longueur supposés et différés 63

Constantes littérales 64
Accès aux sous-chaînes de caractères 66
Accéder à des composants complexes 67
Déclaration et attributs 67
Chapitre 13: Unités de programme et disposition des fichiers 69
Examples 69
Programmes Fortran 69
Modules et sous-modules 70
Procédures externes 70
Bloquer les unités de programme de données 71
Sous-programmes internes 71
Fichiers de code source 72
Chapitre 14: Utilisation des modules 74
Examples 74
La syntaxe du module 74
Utilisation de modules d'autres unités de programme 74
Modules intrinsèques 76
Contrôle d'accès 76
Entités de module protégées 78
Crédits 80

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: fortran

It is an unofficial and free Fortran ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Fortran.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec Fortran

Remarques

Fortran est un langage largement utilisé dans la communauté scientifique en raison de son aptitude au calcul numérique. La notation intuitive, qui facilite l'écriture de calculs vectorisés rapides, est particulièrement intéressante.

Malgré son âge, Fortran est toujours activement développé, avec de nombreuses implémentations, notamment GNU, Intel, PGI et Cray.

Versions

Version	Remarque	Libération
FORTRAN 66	Première normalisation par ASA (maintenant ANSI)	1966-03-07
FORTRAN 77	Forme fixe, historique	1978-04-15
Fortran 90	Formulaire libre, norme ISO, opérations sur les tableaux	1991-06-15
Fortran 95	Procédures pures et élémentaires	1997-06-15
Fortran 2003	Programmation orientée objet	2004-04-04
Fortran 2008	Co-tableaux	2010-09-10

Examples

Installation ou configuration

Fortran est un langage qui peut être compilé à l'aide de compilateurs fournis par de nombreux fournisseurs. Différents compilateurs sont disponibles pour différentes plates-formes matérielles et différents systèmes d'exploitation. Certains compilateurs sont des logiciels gratuits, d'autres peuvent être utilisés gratuitement et d'autres nécessitent l'achat d'une licence.

Le compilateur Fortran gratuit le plus commun est GNU Fortran ou gfortran. Le code source est disponible auprès de GNU dans le cadre de GCC, la collection de compilateurs GNU. Des fichiers binaires pour de nombreux systèmes d'exploitation sont disponibles à l' adresse https://gcc.gnu.org/wiki/GFortranBinaries . Les distributions Linux contiennent souvent gfortran dans leur gestionnaire de paquets.

D'autres compilateurs sont disponibles, par exemple:

- EKOPath par PathScale

- LLVM (backend via DragonEgg)
- Oracle Developer Studio

- Absoft Fortran Compiler
- Intel Fortran Compiler
- NAG Fortran Compiler
- PGI Compilers

Sur les systèmes HPC, le fournisseur du système propose souvent des compilateurs spécialisés, tels que les compilateurs IBM ou Cray .

Tous ces compilateurs prennent en charge la norme Fortran 95. Un aperçu de l'état de Fortran 2003 et du statut de Fortran 2008 par différents compilateurs est proposé par le forum ACM Fortran et disponible dans le wiki Fortran.

Bonjour le monde

Tout programme Fortran doit inclure `end` comme dernière déclaration. Par conséquent, le programme Fortran le plus simple ressemble à ceci:

```
end
```

Voici quelques exemples de programmes "salut, monde":

```
print *, "Hello, world"
end
```

Avec déclaration d' `write` :

```
write(*,*) "Hello, world"
end
```

Pour plus de clarté, il est maintenant courant d'utiliser l'instruction de `program` pour démarrer un programme et lui donner un nom. L'énoncé de `end` peut alors faire référence à ce nom pour rendre évident ce à quoi il fait référence, et laisser le compilateur vérifier l'exactitude du code. De plus, tous les programmes Fortran doivent inclure une instruction `implicit none` . Ainsi, un programme Fortran minimal devrait ressembler à ceci:

```
program hello
  implicit none
  write(*,*) 'Hello world!'
end program hello
```

La prochaine étape logique à partir de ce point est de savoir comment voir le résultat du programme hello world. Cette section montre comment y parvenir dans un environnement similaire à Linux. Nous supposons que vous avez quelques notions de base sur les commandes shell , principalement vous savez comment accéder au terminal shell. Nous supposons également que vous avez déjà configuré votre environnement `fortran` . En utilisant votre éditeur de texte

préféré (notepad, notepad ++, vi, vim, emacs, gedit, kate, etc.), sauvegardez le programme hello ci-dessus (copier-coller) dans un fichier nommé `hello.f90` dans votre répertoire personnel. `hello.f90` est votre fichier source. Ensuite, accédez à la ligne de commande et accédez au répertoire (répertoire de base?) Où vous avez enregistré votre fichier source, puis tapez la commande suivante:

```
>gfortran -o hello hello.f90
```

Vous venez de créer votre programme exécutable hello world. En termes techniques, vous venez de compiler votre programme. Pour l'exécuter, tapez la commande suivante:

```
>./hello
```

Vous devriez voir la ligne suivante imprimée sur votre terminal shell.

```
> Hello world!
```

Félicitations, vous venez d'écrire, de compiler et de lancer le programme "Hello World".

Équation quadratique

Aujourd'hui, Fortran est principalement utilisé pour le calcul numérique. Cet exemple très simple illustre la structure de base du programme pour résoudre des équations quadratiques:

```
program quadratic
  !a comment

  !should be present in every separate program unit
  implicit none

  real :: a, b, c
  real :: discriminant
  real :: x1, x2

  print *, "Enter the quadratic equation coefficients a, b and c:"
  read *, a, b, c

  discriminant = b**2 - 4*a*c

  if ( discriminant>0 ) then

    x1 = ( -b + sqrt(discriminant)) / (2 * a)
    x2 = ( -b - sqrt(discriminant)) / (2 * a)
    print *, "Real roots:"
    print *, x1, x2

    ! Comparison of floating point numbers for equality is often not recommended.
    ! Here, it serves the purpose of illustrating the "else if" construct.
  else if ( discriminant==0 ) then

    x1 = - b / (2 * a)
    print *, "Real root:"
    print *, x1
```

```
  else

    print *, "No real roots."
  end if
end program quadratic
```

Insensibilité à la casse

Les lettres majuscules et minuscules de l'alphabet sont équivalentes dans le jeu de caractères Fortran. En d'autres termes, Fortran est *insensible à la casse* . Ce comportement contraste avec les langages sensibles à la casse, tels que C ++ et bien d'autres.

En conséquence, les variables `a` et `A` sont la même variable. En principe, on pourrait écrire un programme comme suit

```
pROgrAm MYproGRaM
..
enD mYPrOgrAM
```

C’est au bon programmeur d’éviter de tels choix.

Lire Démarrer avec Fortran en ligne: https://riptutorial.com/fr/fortran/topic/904/demarrer-avec-fortran

Chapitre 2: C interopérabilité

Examples

Appeler C depuis Fortran

Fortran 2003 a introduit des fonctionnalités de langage qui peuvent garantir l'interopérabilité entre C et Fortran (et vers davantage de langues en utilisant C comme intermédiaire). Ces fonctionnalités sont principalement accessibles via le module intrinsèque `iso_c_binding` :

```
use, intrinsic :: iso_c_binding
```

Le mot-clé `intrinsic` garantit que le bon module est utilisé, et non un module créé par l'utilisateur du même nom.

`iso_c_binding` donne accès à des paramètres de type type *interopérables* :

```
integer(c_int) :: foo    ! equivalent of 'int foo' in C
real(c_float) :: bar     ! equivalent of 'float bar' in C
```

L'utilisation de paramètres de type C kind garantit que les données peuvent être transférées entre les programmes C et Fortran.

L'interopérabilité des caractères C char et Fortran est probablement un sujet pour lui-même et donc pas discuté ici

---

Pour appeler une fonction C à partir de Fortran, l'interface doit d'abord être déclarée. Ceci est essentiellement équivalent au prototype de la fonction C et permet au compilateur de connaître le nombre et le type des arguments, etc. L'attribut `bind` est utilisé pour indiquer au compilateur le nom de la fonction dans C, qui peut être différente de celle du Fortran. prénom.

oies.h

```
// Count how many geese are in a given flock
int howManyGeese(int flock);
```

geese.f90

```
! Interface to C routine
interface
  integer(c_int) function how_many_geese(flock_num) bind(C, 'howManyGeese')
    ! Interface blocks don't know about their context,
    ! so we need to use iso_c_binding to get c_int definition
    use, intrinsic :: iso_c_binding, only : c_int
    integer(c_int) :: flock_num
  end function how_many_geese
end interface
```

Le programme Fortran doit être lié à la bibliothèque C (*dépend du compilateur, inclure ici?*) `howManyGeese()` inclut l'implémentation de `howManyGeese()` , puis `how_many_geese()` peut être appelé à partir de Fortran.

C structs dans Fortran

L'attribut `bind` peut également être appliqué aux types dérivés:

oies.h

```
struct Goose {
   int flock;
   float buoyancy;
}

struct Goose goose_c;
```

geese.f90

```
use, intrinsic :: iso_c_binding, only : c_int, c_float

type, bind(C) :: goose_t
  integer(c_int) :: flock
  real(c_float) :: buoyancy
end type goose_t

type(goose_t) :: goose_f
```

Les données peuvent maintenant être transférées entre `goose_c` et `goose_f` . C routines qui prennent des arguments de type `Goose` peuvent être appelées de Fortran avec le `type(goose_t)` .

Lire C interopérabilité en ligne: https://riptutorial.com/fr/fortran/topic/2184/c-interoperabilite

Chapitre 3: Contrôle de l'exécution

Examples

Si construire

La construction `if` (appelée instruction IF de bloc dans FORTRAN 77) est commune à de nombreux langages de programmation. Il exécute conditionnellement un bloc de code lorsqu'une expression logique est évaluée à true.

```
[name:] IF (expr) THEN
    block
[ELSE IF (expr) THEN [name]
    block]
[ELSE [name]
   block]
END IF [name]
```

où,

- **name** - le nom de la construction if (facultatif)
- **expr** - une expression logique scalaire entre parenthèses
- **block** - une séquence de zéro ou plusieurs instructions ou constructions

Un nom de construction au début d'une instruction `if then` doit avoir la même valeur que le nom de la construction à la `end if` instruction `end if` et doit être unique pour l'unité de portée actuelle.

Dans les instructions `if` , les égalités (in) et les expressions logiques évaluant une instruction peuvent être utilisées avec les opérateurs suivants:

```
.LT.  which is <   ! less than
.LE.           <=  ! less than or equal
.GT.           >   ! greater than
.GE.           >=  ! greater than or equal
.EQ.           =   ! equal
.NE.           /=  ! not equal
.AND.              ! logical and
.OR.               ! logical or
.NOT.              ! negation
```

Exemples:

```
! simplest form of if construct
if (a > b) then
    c =  b / 2
end if
!equivalent example with alternate syntax
if(a.gt.b)then
   c=b/2
endif
```

```
! named if construct
circle: if (r >= 0) then
    l = 2 * pi * r
end if circle

! complex example with nested if construct
block: if (a < e) then
    if (abs(c - e) <= d) then
        a = a * c
    else
        a = a * d
    end if
else
    a = a * e
end if block
```

Une utilisation historique de la construction `if` trouve dans une instruction appelée "arithmétique if". Étant donné que cela peut être remplacé par des constructions plus modernes, il n'est cependant pas couvert ici. Plus de détails peuvent être trouvés ici .

SELECT CASE construit

Une construction de `select case` exécute conditionnellement un bloc de constructions ou d'instructions en fonction de la valeur d'une expression scalaire dans une instruction de `select case` . Cette construction de contrôle peut être considérée comme un remplacement du `goto` calculé.

```
[name:] SELECT CASE (expr)
[CASE (case-value [, case-value] ...) [name]
   block]...
[CASE DEFAULT [name]
   block]
END SELECT [name]
```

où,

- **name** - le nom de la construction de `select case` (facultatif)
- **expr** - une expression scalaire de type entier, logique ou caractère (entre parenthèses)
- **valeur de casse** - une ou plusieurs expressions d'initialisation scalaires entières, logiques ou de caractères entre parenthèses
- **block** - une séquence de zéro ou plusieurs instructions ou constructions

Exemples:

```
! simplest form of select case construct
select case(i)
case(:-1)
    s = -1
case(0)
    s = 0
case(1:)
    s = 1
case default
```

```
    print "Something strange is happened"
end select
```

Dans cet exemple, la valeur de cas `(:-1)` correspond à une plage de valeurs correspondant à toutes les valeurs inférieures à zéro, `(0)` correspond à des zéros et `(1:)` correspond à toutes les valeurs supérieures à zéro, la section `default` implique Non exécuté.

Bloc DO construire

Une construction `do` est une construction en boucle qui a un nombre d'itérations régi par un contrôle de boucle

```
integer i
do i=1, 5
  print *, i
end do
print *, i
```

Dans la forme ci-dessus, la variable de boucle `i` traverse la boucle 5 fois, en prenant les valeurs 1 à 5 tour à tour. Une fois que la construction a terminé, la variable de boucle a la valeur 6, c'est-à-dire que **la variable de boucle est incrémentée une fois de plus après l'achèvement de la boucle** .

De manière plus générale, le `do` construction de la boucle peut être comprise comme suit

```
integer i, first, last, step
do i=first, last, step
end do
```

La boucle commence par `i` avec la valeur `first` , incrémenter chaque itération par `step` jusqu'à ce que `i` est supérieur à la `last` (ou moins de la `last` si la taille de pas est négatif).

Il est important de noter que depuis Fortran 95, la variable de boucle et les expressions de contrôle de boucle doivent être des nombres entiers.

Une itération peut se terminer prématurément avec l'instruction de `cycle`

```
do i=1, 5
  if (i==4) cycle
end do
```

et la construction entière peut cesser l'exécution avec l'instruction de `exit`

```
do i=1, 5
  if (i==4) exit
end do
print *, i
```

`do` constructions peuvent être nommées:

```
do_name: do i=1, 5
end do do_name
```

ce qui est particulièrement utile lorsque sont imbriqués `do` des constructions

```
do1: do i=1, 5
  do j=1,6
    if (j==3) cycle        ! This cycles the j construct
    if (j==4) cycle        ! This cycles the j construct
    if (i+j==7) cycle do1  ! This cycles the i construct
    if (i*j==15) exit do1  ! This exits the i construct
  end do
end do1
```

Est- `do` constructions peuvent aussi avoir un contrôle de boucle indéterminé, soit "pour toujours" ou jusqu'à ce qu'une condition donnée soit remplie

```
integer :: i=0
do
  i=i+1
  if (i==5) exit
end do
```

ou

```
integer :: i=0
do while (i<6)
  i=i+1
end do
```

Cela permet également de `do` boucle infinie via une `.true.` déclaration

```
print *,'forever'
do while(.true.)
  print *,'and ever'
end do
```

Un `do` construct peut aussi laisser l'ordre des itérations indéterminé

```
do concurrent (i=1:5)
end do
```

notant que la forme du contrôle de boucle est la même que dans un contrôle `forall` .

Il existe diverses restrictions sur les instructions qui peuvent être exécutées dans le cadre d'une structure `do concurrent` conçues pour garantir qu'il n'y a pas de dépendance de données entre les itérations de la construction. Cette indication explicite par le programmeur peut permettre une meilleure optimisation (y compris la parallélisation) par le compilateur, ce qui peut être difficile à déterminer autrement.

Les variables "privées" dans une interaction peuvent être réalisées en utilisant une construction de `block` dans le `do concurrent` :

```
do concurrent (i=1:5, j=2:7)
  block
    real tempval  ! This is independent across iterations
  end block
end do
```

Une autre forme du bloc `do` construire utilise une étiquette `continue` déclaration au lieu d'une `end do` :

```
    do 100, i=1, 5
100 continue
```

Il est même possible d'imbriquer de telles constructions avec une instruction de terminaison partagée

```
    do 100, i=1,5
    do 100, j=1,5
100 continue
```

Ces deux formes, et en particulier la seconde (qui est obsolète), doivent généralement être évitées pour des raisons de clarté.

Enfin, il y a aussi un non-bloc `do` construire. Ceci est également considéré comme obsolète et est décrit ailleurs , ainsi que des méthodes de restructuration en bloc `do` construire.

O construct construire

La `where` construction, disponible en Fortran90 représente partir d' un masqué `do` construire. L'instruction de masquage suit les mêmes règles que l'instruction `if` , mais s'applique à tous les éléments du tableau donné. Utilisation de `where` permet d'effectuer des opérations sur un tableau (ou plusieurs tableaux de même taille), dont les éléments satisfont à une certaine règle. Cela peut être utilisé pour simplifier les opérations simultanées sur plusieurs variables.

Syntaxe:

```
[name]: where (mask)
    block
[elsewhere (mask)
    block]
[elsewhere
    block]
end where [name]
```

Ici,

- **name** - est le nom donné au bloc (s'il est nommé)

- **mask** - est une expression logique appliquée à tous les éléments
- **block** - série de commandes à exécuter

Exemples:

```
! Example variables
real:: A(5),B(5),C(5)
A = 0.0
B = 1.0
C = [0.0, 4.0, 5.0, 10.0, 0.0]

! Simple where construct use
where (C/=0)
    A=B/C
elsewhere
    A=0.0
end

! Named where construct
Block: where (C/=0)
    A=B/C
elsewhere
    A=0.0
end where Block
```

Lire Contrôle de l'exécution en ligne: https://riptutorial.com/fr/fortran/topic/1657/controle-de-l-execution

Chapitre 4: Des alternatives modernes aux caractéristiques historiques

Examples

Types de variables implicites

Lorsque Fortran a été développé à l'origine, la mémoire était très coûteuse. Les variables et les noms de procédure peuvent comporter 6 caractères au maximum et les variables sont souvent *implicitement saisies* . Cela signifie que la première lettre du nom de la variable détermine son type.

- les variables commençant par i, j, ..., n sont des `integer`
- tout le reste (a, b, ..., h et o, p, ..., z) est `real`

Des programmes comme le suivant sont acceptables Fortran:

```
program badbadnotgood
  j = 4
  key = 5 ! only the first letter determines the type
  x = 3.142
  print*, "j = ", j, "key = ", key, "x = ", x
end program badbadnotgood
```

Vous pouvez même définir vos propres règles implicites avec la déclaration `implicit` :

```
! all variables are real by default
implicit real (a-z)
```

ou

```
! variables starting with x, y, z are complex
! variables starting with c, s are character with length of 4 bytes
! and all other letters have their default implicit type
implicit complex (x,y,z), character*4 (c,s)
```

Le typage implicite n'est plus considéré comme la meilleure pratique. Il est très facile de faire une erreur en utilisant le typage implicite, car les fautes de frappe peuvent passer inaperçues, par exemple

```
program oops
  real :: somelongandcomplicatedname

  ...

  call expensive_subroutine(somelongandcomplEcatedname)
end program oops
```

Ce programme se déroulera avec plaisir et fera la mauvaise chose.

Pour désactiver le typage implicite, l'instruction `implicit none` peut être utilisée.

```
program much_better
  implicit none
  integer :: j = 4
  real :: x = 3.142
  print*, "j = ", j, "x = ", x
end program much_better
```

Si nous avions utilisé `implicit none` dans le programme `oops` ci-dessus, le compilateur aurait remarqué immédiatement et produit une erreur.

Arithmétique si déclaration

Arithmétique `if` instruction permet d'utiliser trois branches en fonction du résultat d'une expression arithmétique

```
if (arith_expr) label1, label2, label3
```

Cette instruction `if` transfère le flux de contrôle à l'une des étiquettes d'un code. Si le résultat de `arith_expr` est négatif, `label1` est impliqué, si le résultat est nul, `label2` est utilisé et si le résultat est positif, la dernière `label3` est appliquée. `if` arithmétique requiert les trois étiquettes mais permet de réutiliser les étiquettes, cette instruction peut être simplifiée en deux branches `if` .

Exemples:

```
if (N * N - N / 2) 130, 140, 130

if (X) 100, 110, 120
```

Maintenant, cette fonctionnalité est obsolète avec la même fonctionnalité offerte par la construction `if` et `if-else` . Par exemple, le fragment

```
    if (X) 100, 110, 120
100 print*, "Negative"
    goto 200
110 print*, "Zero"
    goto 200
120 print*, "Positive"
200 continue
```

peut être écrit comme la construction `if-else`

```
if (X<0) then
  print*, "Negative"
else if (X==0) then
  print*, "Zero"
else
```

```
  print*, "Positive"
end if
```

Un remplacement de déclaration `if` pour

```
    if (X) 100, 100, 200
100 print *, "Negative or zero"
200 continue
```

peut être

```
if (X<=0) print*, "Negative or zero"
```

Constructions DO non bloquantes

Le non-bloc `do` construire ressemble

```
    integer i
    do 100, i=1, 5
100 print *, i
```

C'est-à-dire que l'instruction de terminaison étiquetée n'est pas une instruction `continue` . Il y a diverses restrictions sur l'instruction qui peut être utilisée comme instruction de terminaison et le tout est généralement très déroutant.

Une telle construction non-bloc peut être réécrite sous forme de bloc comme

```
    integer i
    do 100 i=1,5
      print *, i
100 continue
```

ou mieux, en utilisant une déclaration de `end do` terminaison,

```
integer i
do i=1,5
  print *, i
end do
```

Retour alternatif

Le retour alternatif est une fonction permettant de contrôler le flux d'exécution à la sortie d'un sous-programme. Il est souvent utilisé comme une forme de gestion des erreurs:

```
real x

call sub(x, 1, *100, *200)
print*, "Success:", x
stop
```

```
100 print*, "Negative input value"
stop

200 print*, "Input value too large"
stop

end

subroutine sub(x, i, *, *)
  real, intent(out) :: x
  integer, intent(in) :: i
  if (i<0) return 1
  if (i>10) return 2
  x = i
end subroutine
```

Le retour alternatif est marqué par les arguments `*` dans la liste d'arguments factices de la sous-routine.

Dans la déclaration d' `call` ci-dessus `*100` et `*200` réfèrent aux déclarations étiquetées `100` et `200` respectivement.

Dans la sous-routine elle-même, les instructions de `return` correspondant à une déclaration alternative ont un numéro. Ce nombre n'est pas une valeur de retour, mais indique l'étiquette fournie à laquelle l'exécution est transmise lors du retour. Dans ce cas, `return 1` transmet l'exécution à l'instruction libellée `100` et `return 2` passe l'exécution à l'instruction intitulée `200` . Une instruction de `return` sans ornement, ou l'exécution d'une exécution de sous-routine sans instruction de `return` , passe l'exécution immédiatement après l'instruction d'appel.

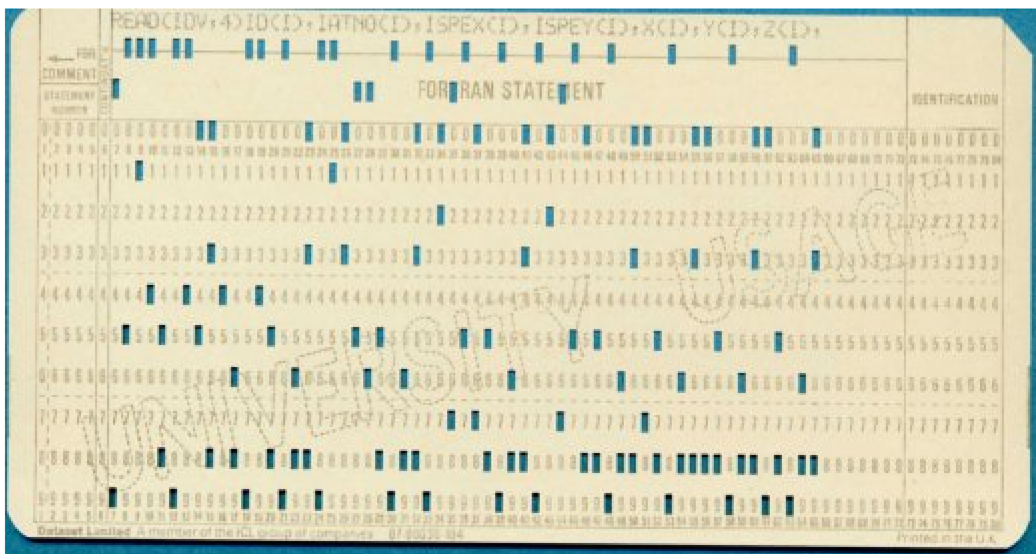
La syntaxe de retour alternative est très différente des autres formes d'association d'arguments et la fonctionnalité introduit le contrôle de flux contrairement aux goûts modernes. Un contrôle de flux plus agréable peut être géré avec le retour d'un code entier "status".

```
real x
integer status

call sub(x, 1, status)
select case (status)
case (0)
  print*, "Success:", x
case (1)
  print*, "Negative input value"
case (2)
  print*, "Input value too large"
end select

end

subroutine sub(x, i, status)
  real, intent(out) :: x
  integer, intent(in) :: i
  integer, intent(out) :: status

  status = 0

  if (i<0) then
```

```
    status = 1
  else if (i>10)
    status = 2
  else
    x = i
  end if

end subroutine
```

Formulaire source fixe

Fortran était à l'origine conçu pour un format de format fixe basé sur une carte perforée de 80 colonnes:



Oui: il s'agit d'une ligne du code de l'auteur

Celles-ci ont été créées sur une machine à perforer les cartes, comme ceci:

Les images sont des photographies originales de l'auteur

Le format, comme indiqué sur la carte illustrée illustrée, comportait les cinq premières colonnes réservées aux étiquettes de relevé. La première colonne a été utilisée pour désigner les commentaires par une lettre **C.** La sixième colonne a été utilisée pour désigner une suite d'instruction (en insérant un caractère autre qu'un zéro 0). Les 8 dernières colonnes ont été

utilisées pour l'identification et le séquencement des cartes, ce qui était très utile si vous jetiez votre jeu de cartes par terre! Le codage des caractères pour les cartes perforées ne comportait qu'un ensemble limité de caractères et était uniquement en majuscules. En conséquence, les programmes Fortran ressemblaient à ceci:

```
       DIMENSION A(10)                                                    00000001
C THIS IS A COMMENT STATEMENT TO EXPLAIN THIS EXAMPLE PROGRAM            00000002
       WRITE (6,100)                                                      00000003
 100   FORMAT(169HTHIS IS A RATHER LONG STRING BEING OUTPUT WHICH GOES OVE00000004
      1R MORE THAN ONE LINE, AND USES THE STATEMENT CONTINUATION MARKER IN00000005
      2COLUMN 6, AND ALSO USES HOLLERITH STRING FORMAT)                   00000006
       STOP                                                               00000007
       END                                                                00000008
```

Le caractère d'espace a également été ignoré partout, sauf dans une constante de caractère *Hollerith* (comme indiqué ci-dessus). Cela signifiait que des espaces pouvaient apparaître à l'intérieur des mots réservés et des constantes, ou manquer complètement. Cela a eu l'effet secondaire de certaines déclarations plutôt trompeuses telles que:

```
       DO 1 I = 1.0
```

est une affectation à la variable `DO1I` alors que:

```
       DO1I = 1,0
```

est en fait une boucle `DO` sur la variable `I`

Fortran moderne n'a pas besoin maintenant de cette forme fixe d'entrée et permet la forme libre en utilisant n'importe quelle colonne. Les commentaires sont maintenant indiqués par un `!` qui peut également être ajouté à une ligne de relevé. Les espaces ne sont plus autorisés nulle part et doivent être utilisés comme séparateurs, comme dans la plupart des autres langues. Le programme ci-dessus pourrait être écrit dans Fortran moderne comme:

```
! This is a comment statement to explain this example program
Print *,"THIS IS A RATHER LONG STRING BEING OUTPUT WHICH no longer GOES OVER MORE THAN ONE
LINE, AND does not need to USE THE STATEMENT CONTINUATION MARKER IN COLUMN 6, or the HOLLERITH
STRING FORMAT"
```

Bien que l’ancienne suite ne soit plus utilisée, l’exemple ci-dessus montre que des déclarations très longues se produiront toujours. Le Fortran moderne utilise un symbole `&` à la fin et au début de la suite. Par exemple, nous pourrions écrire ce qui précède sous une forme plus lisible:

```
! This is a comment statement to explain this example program
Print *,"THIS IS A RATHER LONG STRING BEING OUTPUT WHICH still &
         &GOES OVER MORE THAN ONE LINE, AND does need to USE THE STATEMENT &
         &CONTINUATION notation"
```

Blocs communs

Dans les premières formes de Fortran, le seul mécanisme permettant de créer un stockage de variables global visible à partir de sous-programmes et de fonctions consiste à utiliser le mécanisme de bloc `COMMON` . Cela permettait aux séquences de variables d'être des noms et partagées en commun.

En plus des blocs communs nommés, il peut également y avoir un bloc commun vide (sans nom).

Un bloc commun vide pourrait être déclaré comme

```
common i, j
```

alors que les `variables` bloc nommées pourraient être déclarées comme

```
common /variables/ i, j
```

Comme exemple complet, nous pourrions imaginer un magasin de tas qui est utilisé par les routines qui peuvent ajouter et supprimer des valeurs:

```
      PROGRAM STACKING
      COMMON /HEAP/ ICOUNT, ISTACK(1023)
      ICOUNT = 0
      READ *, IVAL
      CALL PUSH(IVAL)
      CALL POP(IVAL)
      END

      SUBROUTINE PUSH(IVAL)
      COMMON /HEAP/ ICOUNT, ISTACK(1023)
      ICOUNT = ICOUNT + 1
      ISTACK(ICOUNT) = IVAL
      RETURN
      END

      SUBROUTINE POP(IVAL)
      COMMON /HEAP/ ICOUNT, ISTACK(1023)
      IVAL = ISTACK(ICOUNT)
      ICOUNT = ICOUNT - 1
      RETURN
      END
```

Les instructions communes peuvent être utilisées pour déclarer implicitement le type d'une variable et spécifier l'attribut de `dimension` . Ce comportement seul est souvent une source de confusion suffisante. De plus, l'association de stockage implicite et les exigences pour des définitions répétées entre les unités de programme rendent l'utilisation des blocs communs sujettes aux erreurs.

Enfin, les blocs communs sont très restreints dans les objets qu’ils contiennent. Par exemple, un tableau dans un bloc commun doit être de taille explicite; les objets attribuables peuvent ne pas se produire; les types dérivés ne doivent pas avoir d'initialisation par défaut.

Dans Fortran moderne, ce partage de variables peut être géré par l'utilisation de modules . L'exemple ci-dessus peut être écrit comme suit:

```
module heap
  implicit none
  ! In Fortran 2008 all module variables are implicitly saved
  integer, save :: count = 0
  integer, save :: stack(1023)
end module heap

program stacking
  implicit none
  integer val
  read *, val
  call push(val)
  call pop(val)

contains
  subroutine push(val)
    use heap, only : count, stack
    integer val
    count = count + 1
    stack(count) = val
  end subroutine push

  subroutine pop(val)
    use heap, only : count, stack
    integer val
    val = stack(count)
    count = count - 1
  end subroutine pop
end program stacking
```

Les blocs communs nommés et vides ont des comportements légèrement différents. À noter:

- les objets des blocs communs nommés peuvent être définis initialement; les objets en blanc communs ne doivent pas être
- les objets dans les blocs communs vides se comportent comme si le bloc commun avait l'attribut `save` ; les objets dans des blocs communs nommés sans l'attribut `save` peuvent ne plus être définis lorsque le bloc ne fait pas partie d'une unité de programme active

Ce dernier point peut être opposé au comportement des variables de module dans le code moderne. Toutes les variables de module dans Fortran 2008 sont implicitement enregistrées et ne deviennent pas indéfinies lorsque le module est hors de portée. Avant Fortran 2008, les variables de module, comme les variables dans les blocs communs nommés, deviendraient également indéfinies lorsque le module serait hors de portée.

GOTO attribué

GOTO assigné utilise une variable entière à laquelle une étiquette est affectée à l'aide de l'instruction ASSIGN.

```
100 CONTINUE

...

ASSIGN 100 TO ILABEL
```

```
...

GOTO ILABEL
```

Assigné GOTO est obsolète dans Fortran 90 et supprimé dans Fortran 95 et versions ultérieures. Il peut être évité dans le code moderne en utilisant des procédures, des procédures internes, des pointeurs de procédure et d’autres fonctionnalités.

GOTO calculé

GOTO calculé permet le branchement du programme en fonction de la valeur d'une expression entière.

```
GOTO (label_1, label_2,... label_n) scalar-integer-expression
```

Si `scalar-integer-expression` est égal à 1, le programme continue à l'étiquette d'étiquette `label_1` , s'il est égal à 2, il passe à `label_2` et ainsi de suite. Si elle est inférieure à `1` ou plus grand que `n` programme se poursuit sur la ligne suivante.

Exemple:

```
ivar = 2

...

GOTO (10, 20, 30, 40) ivar
```

sautera à l'étiquette d'étiquette 20.

Cette forme de `goto` est obsolète dans Fortran 95 et les versions ultérieures, étant remplacée par la construction de `select case` sélectifs.

Spécificateurs de format assignés

Avant Fortran 95, il était possible d'utiliser des formats assignés pour l'entrée ou la sortie. Considérer

```
integer i, fmt
read *, i

assign 100 to fmt
if (i<100000) assign 200 to fmt

print fmt, i

100 format ("This is a big number", I10)
200 format ("This is a small number", I6)

end
```

L'instruction `assign` assigne une étiquette à une variable entière. Cette variable entière est utilisée ultérieurement comme spécificateur de format dans l'instruction `print` .

Une telle attribution de spécificateur de format a été supprimée dans Fortran 95. Au lieu de cela, un code plus moderne peut utiliser une autre forme de contrôle de flux d'exécution.

```
integer i
read *, i

if (i<100000) then
  print 100, i
else
  print 200, i
end if

100 format ("This is a big number", I10)
200 format ("This is a small number", I6)

end
```

ou une variable de caractère peut être utilisée comme spécificateur de format

```
character(29), target :: big_fmt='("This is a big number", I10)'
character(30), target :: small_fmt='("This is a small number", I6)'
character(:), pointer :: fmt

integer i
read *, i

fmt=>big_fmt
if (i<100000) fmt=>small_fmt

print fmt, i

end
```

Fonctions de déclaration

Considérez le programme

```
implicit none
integer f, i
f(i)=i

print *, f(1)
end
```

Ici `f` est une fonction de déclaration. Il a un type de résultat entier, prenant un argument factice entier. [1]

Une telle fonction d'instruction existe dans le cadre dans lequel elle est définie. En particulier, il a accès aux variables et aux constantes nommées accessibles dans cette étendue.

Cependant, les fonctions d'instruction sont soumises à de nombreuses restrictions et peuvent être

source de confusion (en regardant un regard décontracté comme une instruction d'affectation d'éléments de tableau). Les restrictions importantes sont les suivantes:

- le résultat de la fonction et les arguments factices doivent être scalaires
- les arguments factices ont la même portée que la fonction
- les fonctions d'instruction n'ont pas de variables locales
- les fonctions d'instruction ne peuvent pas être passées comme arguments réels

Les principaux avantages des fonctions de relevé sont répétés par les fonctions internes

```
implicit none

print *, f(1)

contains

  integer function f(i)
    integer i
    f = i
  end function

end
```

Les fonctions internes ne sont pas soumises aux restrictions mentionnées ci-dessus, mais il est peut-être utile de noter qu’un sous-programme interne ne peut contenir d’autres sous-programmes internes (mais il peut contenir une fonction de relevé).

Les fonctions internes ont leur propre portée mais disposent également d'une association hôte.

---

[1] Dans les vrais anciens exemples de code, il ne serait pas inhabituel de voir implicitement les arguments factices d'une fonction d'instruction, même si le résultat a un type explicite.

Lire Des alternatives modernes aux caractéristiques historiques en ligne: https://riptutorial.com/fr/fortran/topic/2103/des-alternatives-modernes-aux-caracteristiques-historiques

Chapitre 5: Extensions du fichier source (.f, .f90, .f95, ...) et leur relation avec le compilateur.

Introduction

Les fichiers Fortran sont soumis à une variété d'extensions et chacun d'eux a une signification distincte. Ils spécifient la version de Fortran, le style de formatage du code et l'utilisation de directives de préprocesseur similaires au langage de programmation C.

Examples

Extensions et Signification

Voici quelques-unes des extensions communes utilisées dans les fichiers sources Fortran et les fonctionnalités sur lesquelles elles peuvent fonctionner.

F minuscule dans l'extension

Ces fichiers n'ont pas les caractéristiques des directives de préprocesseur similaires au langage de programmation en C. Ils peuvent être directement compilés pour créer des fichiers objets. par exemple: .f, .for, .f95

Majuscule F dans l'extension

Ces fichiers ont les caractéristiques des directives de préprocesseur similaires au langage de programmation en C. Les préprocesseurs sont soit définis dans les fichiers, soit en utilisant des fichiers d'entête comme C / C ++ ou les deux. Ces fichiers doivent être pré-traités pour obtenir les fichiers d'extension minuscules qui peuvent être utilisés pour la compilation. par exemple: .F, .FOR, .F95

.f, .for, .f77, .ftn

Celles-ci sont utilisées pour les fichiers Fortran utilisant le **format de style Fixe** et utilisent donc la version **Fortran 77** . Comme ils sont des extensions en minuscule, ils ne peuvent pas avoir de directives de préprocesseur.

.F, .FOR, .F77, .FTN

Celles-ci sont utilisées pour les fichiers Fortran utilisant le **format de style Fixe** et utilisent donc la version **Fortran 77** . Comme ils sont en majuscules, ils peuvent avoir des directives de préprocesseur et doivent donc être prétraités pour obtenir les fichiers d'extension minuscules.

.f90, .f95, .f03, .f08 Ceux-ci sont utilisés pour les fichiers Fortran qui utilisent le **format Free style**

et utilisent donc des versions ultérieures de Fortran. Les versions sont dans le nom.

- f90 - Fortran 90
- f95 - Fortran 95
- f03 - Fortran 2003
- f08 - Fortran 2008

Comme ils sont des extensions en minuscule, ils ne peuvent pas avoir de directives de préprocesseur.

.F90, .F95, .F03, .F08 Ceux-ci sont utilisés pour les fichiers Fortran qui utilisent le **format Free style** et utilisent donc des versions ultérieures de Fortran. Les versions sont dans le nom.

- F90 - Fortran 90
- F95 - Fortran 95
- F03 - Fortran 2003
- F08 - Fortran 2008

Comme ils sont en majuscules, ils ont des directives de préprocesseur et doivent donc être prétraités pour obtenir les fichiers d'extension minuscules.

Lire Extensions du fichier source (.f, .f90, .f95, ...) et leur relation avec le compilateur. en ligne: https://riptutorial.com/fr/fortran/topic/10265/extensions-du-fichier-source---f---f90---f95-------et-leur-relation-avec-le-compilateur-

Chapitre 6: I / O

Syntaxe

- `WRITE(unit num, format num)` affiche les données après les crochets dans une nouvelle ligne.
- `READ(unit num, format num)` saisit la variable après les crochets.
- `OPEN(unit num, FILE=file)` ouvre un fichier. (Il y a plus d'options pour ouvrir des fichiers, mais elles ne sont pas importantes pour les E / S.
- `CLOSE(unit num)` ferme un fichier.

Examples

Simple I / O

À titre d'exemple d'écriture des entrées et des sorties, nous allons prendre une valeur réelle et renvoyer la valeur et son carré jusqu'à ce que l'utilisateur entre un nombre négatif.

Comme spécifié ci-dessous, la commande `read` prend deux arguments: le numéro d'unité et le spécificateur de format. Dans l'exemple ci-dessous, nous utilisons `*` pour le numéro d'unité (qui indique stdin) et `*` pour le format (qui indique la valeur par défaut pour les réels, dans ce cas). Nous spécifions également le format de l'instruction d' `print` . On peut aussi utiliser `write(*,"The value....")` ou simplement ignorer le formatage et l'avoir comme

```
print *,"The entered value was ", x," and its square is ",x*x
```

ce qui entraînera probablement des chaînes et des valeurs étrangement espacées.

```
program SimpleIO
   implicit none
   integer, parameter :: wp = selected_real_kind(15,307)
   real(kind=wp) :: x

   ! we'll loop over until user enters a negative number
   print '("Enter a number >= 0 to see its square. Enter a number < 0 to exit.")'
   do
      ! this reads the input as a double-pricision value
      read(*,*) x
      if (x < 0d0) exit
      ! print the entered value and it's square
      print '("The entered value was ",f12.6,", its square is ",f12.6,".")',x,x*x
   end do
   print '("Thank you!")'

end program SimpleIO
```

Lire avec quelques vérifications d'erreur

Un exemple Fortran moderne qui inclut la vérification des erreurs et une fonction permettant

d'obtenir un nouveau numéro d'unité pour le fichier.

```
module functions

contains

    function get_new_fileunit() result (f)
        implicit none

        logical    :: op
        integer    :: f

        f = 1
        do
            inquire(f,opened=op)
            if (op .eqv. .false.) exit
            f = f + 1
        enddo

    end function

end module

program file_read
    use functions, only : get_new_fileunit
    implicit none

    integer            :: unitno, ierr, readerr
    logical            :: exists
    real(kind(0.d0))   :: somevalue
    character(len=128) :: filename

    filename = "somefile.txt"

    inquire(file=trim(filename), exist=exists)
    if (exists) then
        unitno = get_new_fileunit()
        open(unitno, file=trim(filename), action="read", iostat=ierr)
        if (ierr .eq. 0) then
            read(unitno, *, iostat=readerr) somevalue
            if (readerr .eq. 0) then
                print*, "Value in file ", trim(filename), " is ", somevalue
            else
                print*, "Error ", readerr, &
                        " attempting to read file ", &
                        trim(filename)
            endif
        else
            print*, "Error ", ierr ," attempting to open file ", trim(filename)
            stop
        endif
    else
        print*, "Error -- cannot find file: ", trim(filename)
        stop
    endif

end program file_read
```

Passer des arguments de ligne de commande

Lorsque les arguments de ligne de commande sont pris en charge, ils peuvent être lus via la `get_command_argument` intrinsèque de `get_command_argument` (introduite dans la norme Fortran 2003). La commande `command_argument_count` permet de connaître le nombre d'arguments fournis sur la ligne de commande.

Tous les arguments de ligne de commande sont lus en tant que chaînes, donc une conversion de type interne doit être effectuée pour les données numériques. Par exemple, ce code simple additionne les deux chiffres fournis sur la ligne de commande:

```
PROGRAM cmdlnsum
IMPLICIT NONE
CHARACTER(100) :: num1char
CHARACTER(100) :: num2char
REAL :: num1
REAL :: num2
REAL :: numsum

!First, make sure the right number of inputs have been provided
IF(COMMAND_ARGUMENT_COUNT().NE.2)THEN
  WRITE(*,*)'ERROR, TWO COMMAND-LINE ARGUMENTS REQUIRED, STOPPING'
  STOP
ENDIF

CALL GET_COMMAND_ARGUMENT(1,num1char)   !first, read in the two values
CALL GET_COMMAND_ARGUMENT(2,num2char)

READ(num1char,*)num1                    !then, convert them to REALs
READ(num2char,*)num2

numsum=num1+num2                        !sum numbers
WRITE(*,*)numsum                        !write out value

END PROGRAM
```

L'argument number de `get_command_argument` situe utilement entre `0` et le résultat de `command_argument_count` . Si la valeur est `0` le nom de la commande est fourni (si pris en charge).

De nombreux compilateurs offrent également des éléments intrinsèques non standard (tels que `getarg`) pour accéder aux arguments de la ligne de commande. Comme ils ne sont pas standard, la documentation du compilateur correspondante doit être consultée.

L'utilisation de `get_command_argument` peut être étendue au-delà de l'exemple ci-dessus avec les arguments `length` et `status` . Par exemple, avec

```
character(5) arg
integer stat
call get_command_argument(number=1, value=arg, status=stat)
```

la valeur de `stat` sera `-1` si le premier argument existe et a une longueur supérieure à 5. S'il y a une autre difficulté à récupérer l'argument, la valeur de `stat` sera un nombre positif (et `arg` sera entièrement composé de blancs). Sinon, sa valeur sera `0` .

L'argument `length` peut être combiné avec une variable de longueur de longueur différée, comme

dans l'exemple suivant.

```
character(:), allocatable :: arg
integer arglen, stat
call get_command_argument(number=1, length=arglen)  ! Assume for simplicity success
allocate (character(arglen) :: arg)
call get_command_argument(number=1, value=arg, status=stat)
```

Lire I / O en ligne: https://riptutorial.com/fr/fortran/topic/6778/i---o

Chapitre 7: Interfaces explicites et implicites

Examples

Sous-programmes internes / modules et interfaces explicites

Un *sous-programme* (qui définit une *procédure*) peut être soit un `subroutine` - `subroutine` soit une `function` . il est dit être un *sous* - *programme interne* s'il est appelé ou appelé depuis le même `program` ou *sous* - *programme* qui le `contains` , comme suit

```
program my_program

  ! declarations
  ! executable statements,
  ! among which an invocation to
  ! internal procedure(s),
  call my_sub(arg1,arg2,...)
  fx = my_fun(xx1,xx2,...)

contains

  subroutine my_sub(a1,a2,...)
    ! declarations
    ! executable statements
  end subroutine my_sub

  function my_fun(x1,x2,...) result(f)
    ! declarations
    ! executable statements
  end function my_fun

end program my_program
```

Dans ce cas, le compilateur sera au courant de toute procédure interne, car il traite l'unité de programme dans son ensemble. En particulier, il "verra" l' `interface` la procédure, c'est-à-dire

- que ce soit une `function` ou un `subroutine` - `subroutine` ,
- qui sont les noms et propriétés des arguments `a1` , `a2` , `x1` , `x2` , ...,
- quelles sont les propriétés du *résultat* `f` (dans le cas d'une `function`).

Étant donné que l'interface est connue, le compilateur peut vérifier si les arguments réels (`arg1` , `arg2` , `xx1` , `xx2` , `fx` , ...) transmis à la procédure correspondent aux arguments factices (`a1` , `a2` , `x1` , `x2` , `f` , .. .).

Dans ce cas, on dit que l'interface est *explicite* .

On dit qu'un sous-programme est un sous-programme de *module* lorsqu'il est appelé par une instruction dans le module contenant lui-même,

```
module my_mod
```

```
  ! declarations

contains

  subroutine my_mod_sub(b1,b2,...)
    ! declarations
    ! executable statements
    r = my_mod_fun(b1,b2,...)
  end subroutine my_sub

  function my_mod_fun(y1,y2,...) result(g)
    ! declarations
    ! executable statements
  end function my_fun

end module my_mod
```

ou par une déclaration dans une autre unité de programme `use` ce module,

```
program my_prog

  use my_mod

  call my_mod_sub(...)

end program my_prog
```

Comme dans la situation précédente, le compilateur saura tout sur le sous-programme et, par conséquent, nous disons que l'interface est *explicite* .

Sous-programmes externes et interfaces implicites

Un sous-programme est dit *externe* lorsqu'il n'est pas contenu dans le programme principal, ni dans un module, ni dans un autre sous-programme. En particulier, il peut être défini au moyen d'un langage de programmation autre que Fortran.

Lorsqu'un sous-programme externe est invoqué, le compilateur ne peut pas accéder à son code, de sorte que toutes les informations admissibles au compilateur sont implicitement contenues dans l'instruction appelante du programme appelant et dans le type a les propriétés des arguments, dont la déclaration est inconnue du compilateur). Dans ce cas, nous disons que l'interface est *implicite* .

Une instruction `external` peut être utilisée pour spécifier que le nom d'une procédure est relatif à une procédure externe,

```
external external_name_list
```

mais malgré tout, l'interface reste implicite.

Un bloc d' `interface` peut être utilisé pour spécifier l'interface d'une procédure externe,

```
interface
```

```
  interface_body
end interface
```

où l' `interface_body` est normalement une copie exacte de l'en-tête de procédure suivie de la déclaration de tous ses arguments et, s'il s'agit d'une fonction, du résultat.

Par exemple, pour la fonction `WindSpeed`

```
real function WindSpeed(u, v)
  real, intent(in) :: u, v
  WindSpeed = sqrt(u*u + v*v)
end function WindSpeed
```

Vous pouvez écrire l'interface suivante

```
interface
  real function WindSpeed(u, v)
    real, intent(in) :: u, v
  end function WindSpeed
end interface
```

Lire Interfaces explicites et implicites en ligne:
https://riptutorial.com/fr/fortran/topic/2882/interfaces-explicites-et-implicites

Chapitre 8: Procédures - Fonctions et sous-programmes

Remarques

Les fonctions et les *sous - programmes* , associés aux *modules* , sont les outils permettant de décomposer un *programme* en unités. Cela rend le programme plus lisible et gérable. Chacune de ces unités peut être considérée comme faisant partie du code qui, idéalement, pourrait être compilé et testé isolément. Le ou les programmes principaux peuvent appeler (ou appeler) de tels sous-programmes (fonctions ou sous-programmes) pour accomplir une tâche.

Les fonctions et les sous-programmes sont différents dans le sens suivant:

- **Les fonctions** renvoient un seul objet et - généralement - ne modifient pas les valeurs de ses arguments (c.-à-d. Qu'elles agissent comme une fonction mathématique!);
- **Les sous-routines** effectuent généralement une tâche plus compliquée et modifient généralement leurs arguments (le cas échéant), ainsi que d'autres variables (par exemple, celles déclarées dans le module contenant la sous-routine).

Les fonctions et les sous-programmes sont regroupés sous le nom de *procédures* . (Dans ce qui suit, nous utiliserons le verbe "call" comme synonyme de "invoke" même si techniquement les procédures à `call` ed sont des `subroutine` - `subroutine` s, alors que les `function` s apparaissent à droite de l'affectation ou dans des expressions.)

Examples

Syntaxe de la fonction

Les fonctions peuvent être écrites en utilisant plusieurs types de syntaxe

```
function name()
  integer name
  name = 42
end function
```

```
integer function name()
  name = 42
end function
```

```
function name() result(res)
  integer res
  res = 42
end function
```

Les fonctions renvoient des valeurs via un *résultat de fonction* . À moins que l'instruction de la fonction n'ait une clause `result` , le `result` la fonction porte le même nom que la fonction. Avec

`result` le `result` la fonction est celui donné par le `result` . Dans chacun des deux premiers exemples ci-dessus, le résultat de la fonction est donné par `name` ; dans le troisième par `res` .

Le résultat de la fonction doit être défini lors de l'exécution de la fonction.

Les fonctions permettent d'utiliser des préfixes spéciaux.

La fonction *pure* signifie que cette fonction n'a aucun effet secondaire:

```
pure real function square(x)
  real, intent(in) :: x
  square = x * x
end function
```

La fonction *élémentaire* est définie en tant qu'opérateur scalaire, mais elle peut être invoquée avec le tableau comme argument réel, auquel cas la fonction sera appliquée par élément. À moins que le préfixe `impure` (introduit dans Fortran 2008) ne soit spécifié, une fonction *élémentaire* est également une fonction *pure* .

```
elemental real function square(x)
  real, intent(in) :: x
  square = x * x
end function
```

Déclaration de retour

L'instruction `return` peut être utilisée pour quitter la fonction et la sous-routine. Contrairement à beaucoup d'autres langages de programmation, il n'est pas utilisé pour définir la valeur de retour.

```
real function f(x)
  real, intent(in) :: x
  integer :: i

  f = x

  do i = 1, 10

    f = sqrt(f) - 1.0

    if (f < 0) then
      f = -1000.
      return
    end if

  end do
end function
```

Cette fonction effectue un calcul itératif. Si la valeur de `f` devient négative, la fonction renvoie la valeur -1000.

Procédures récursives

Dans Fortran, les fonctions et les sous-routines doivent être déclarées explicitement comme *récursives* , si elles doivent se rappeler, directement ou indirectement. Ainsi, une implémentation récursive de la série Fibonacci pourrait ressembler à ceci:

```
recursive function fibonacci(term) result(fibo)
  integer, intent(in) :: term
  integer :: fibo

  if (term <= 1) then
    fibo = 1
  else
    fibo = fibonacci(term-1) + fibonacci(term-2)
  end if

end function fibonacci
```

Un autre exemple est autorisé à calculer factoriel:

```
recursive function factorial(n)  result(f)
  integer :: f
  integer, intent(in) :: n

  if(n == 0) then
    f = 1
  else
    f = n * f(n-1)
  end if
end function factorial
```

Pour qu'une fonction se réfère directement de manière récursive, sa définition doit utiliser le suffixe de `result` . Il n'est pas possible qu'une fonction soit à la fois `recursive` et `elemental` .

L'intention des arguments factices

L'attribut d' `intent` d'un argument factice dans une sous-routine ou une fonction déclare son utilisation prévue. La syntaxe est soit l'un des

```
intent(IN)
intent(OUT)
intent(INOUT)
```

Par exemple, considérez cette fonction:

```
real function f(x)
  real, intent(IN) :: x

  f = x*x
end function
```

L' `intent(IN)` spécifie que l'argument factice (non pointeur) `x` peut ne jamais être défini ou devenir indéfini pendant toute la fonction ou son initialisation. Si un argument factice de pointeur a l'attribut `intent(IN)` , cela s'applique à son association.

`intent(OUT)` pour un argument factice non-pointeur signifie que l'argument factice devient indéfini lors de l'appel du sous-programme (sauf pour les composants d'un type dérivé avec initialisation par défaut) et doit être défini lors de l'exécution. L'argument réel transmis en tant qu'argument factice doit être définissable: le passage d'une constante nommée ou littérale, ou d'une expression, n'est pas autorisé.

De même que précédemment, si un argument factice de pointeur est `intent(OUT)` le statut d'association du pointeur devient indéfini. L'argument actuel doit être une variable de pointeur.

`intent(INOUT)` spécifie que l'argument réel est définissable et convient à la fois pour transmettre et renvoyer des données de la procédure.

Enfin, un argument factice peut être sans l'attribut `intent` . Un tel argument factice est limité par l'argument proprement dit.

Par exemple, considérez

```
integer :: i = 0
call sub(i, .TRUE.)
call sub(1, .FALSE.)

end

subroutine sub(i, update)
  integer i
  logical, intent(in) :: update
  if (update) i = i+1
end subroutine
```

L'argument que `i` ne peux avoir aucun attribut d' `intent` qui permet les deux appels de sous-programme du programme principal.

Référencement d'une procédure

Pour qu'une fonction ou un sous-programme soit utile, il doit être référencé. Un sous-programme est référencé dans une déclaration d' `call`

```
call sub(...)
```

et une fonction dans une expression. Contrairement à de nombreux autres langages, une expression ne constitue pas une instruction complète, de sorte qu'une référence de fonction est souvent vue dans une instruction d'affectation ou utilisée d'une autre manière:

```
x = func(...)
y = 1 + 2*func(...)
```

Il existe trois façons de désigner une procédure référencée:

- comme nom d'un pointeur de procédure ou de procédure
- un composant de procédure d'un objet de type dérivé

- un nom de liaison de procédure liée au type

Le premier peut être vu comme

```
procedure(), pointer :: sub_ptr=>sub
call sub()   ! With no argument list the parentheses are optional
call sub_ptr()
end

subroutine sub()
end subroutine
```

et les deux derniers comme

```
module mod
  type t
    procedure(sub), pointer, nopass :: sub_ptr=>sub
  contains
    procedure, nopass :: sub
  end type

contains

  subroutine sub()
  end subroutine

end module

use mod
type(t) x
call x%sub_ptr()   ! Procedure component
call x%sub()       ! Binding name

end
```

Pour une procédure avec des arguments factices, la référence nécessite *des* arguments *réels* correspondants, bien que des arguments factices facultatifs puissent ne pas être donnés.

Considérons le sous-programme

```
subroutine sub(a, b, c)
  integer a, b
  integer, optional :: c
end subroutine
```

Cela peut être référencé des deux manières suivantes

```
call sub(1, 2, 3)   ! Passing to the optional dummy c
call sub(1, 2)      ! Not passing to the optional dummy c
```

C'est ce qu'on appelle le référencement de *position* : les arguments réels sont associés en fonction de la position dans les listes d'arguments. Ici, le mannequin `a` est associé à `1` , `b` avec `2` et `c` (lorsque spécifié) avec `3` .

Alternativement, le référencement par *mot* - *clé* peut être utilisé lorsque la procédure dispose d'une interface explicite disponible

```
call sub(a=1, b=2, c=3)
call sub(a=1, b=2)
```

qui est le même que ci-dessus.

Cependant, avec des mots-clés, les arguments réels peuvent être proposés dans n'importe quel ordre.

```
call sub(b=2, c=3, a=1)
call sub(b=2, a=1)
```

Le référencement par position et par mot clé peut être utilisé

```
call sub(1, c=3, b=2)
```

tant qu'un mot-clé est donné pour chaque argument suite à la première apparition d'un mot-clé

```
call sub(b=2, 1, 3)  ! Not valid: all keywords must be specified
```

La valeur du référencement par mot-clé est particulièrement prononcée lorsqu'il existe plusieurs arguments factices facultatifs, comme indiqué ci-dessous si, dans la définition de la sous-routine ci-dessus, `b` étaient également facultatifs.

```
call sub(1, c=3)  ! Optional b is not passed
```

Les listes d'arguments pour les procédures liées aux types ou les pointeurs de procédures de composants avec un argument passé sont considérées séparément.

Lire Procédures - Fonctions et sous-programmes en ligne: https://riptutorial.com/fr/fortran/topic/1106/procedures---fonctions-et-sous-programmes

Chapitre 9: Procédures intrinsèques

Remarques

Un grand nombre des procédures intrinsèques disponibles ont des types d'argument communs. Par exemple:

- un argument logique `MASK` qui sélectionne les éléments des tableaux d'entrée à traiter
- un argument scalaire entier `KIND` qui détermine le type de résultat de la fonction
- un argument entier `DIM` pour une fonction de réduction qui contrôle la dimension sur laquelle la réduction est effectuée

Examples

Utilisation de PACK pour sélectionner des éléments répondant à une condition

La fonction `pack` intrinsèque emballe un tableau dans un vecteur, en sélectionnant des éléments basés sur un masque donné. La fonction a deux formes

```
PACK(array, mask)
PACK(array, mask, vector)
```

(c'est-à-dire que l'argument `vector` est facultatif).

Dans les deux cas, `array` est un tableau et un `mask` de type logique conforme à `array` (un scalaire ou un tableau de même forme).

Dans le premier cas, le résultat est rang-1 tableau de paramètres de type et de type de `array` le nombre d'éléments étant le nombre d'éléments vrais dans le masque.

```
integer, allocatable :: positive_values(:)
integer :: values(5) = [2, -1, 3, -2, 5]
positive_values = PACK(values, values>0)
```

les valeurs `positive_values` sont le tableau `[2, 3, 5]` .

Avec l'argument `vector` rank-1, le résultat est maintenant la taille du `vector` (qui doit avoir au moins autant d'éléments que de valeurs vraies dans le `mask` .

L'effet avec `vector` est de renvoyer ce tableau avec les éléments initiaux de ce tableau écrasés par les éléments masqués du `array` . Par exemple

```
integer, allocatable :: positive_values(:)
integer :: values(5) = [2, -1, 3, -2, 5]
positive_values = PACK(values, values>0, [10,20,30,40,50])
```

les `[2,3,5,40,50]` `positive_values` sont le tableau `[2,3,5,40,50]` .

Il convient de noter que, quelle que soit la forme du `array` arguments `array` le résultat est toujours un tableau de rang 1.

En plus de sélectionner les éléments d'un tableau répondant à une condition de masquage, il est souvent utile de déterminer les indices pour lesquels la condition de masquage est remplie. Cet idiome commun peut être exprimé comme

```
integer, allocatable :: indices(:)
integer i
indices = PACK([(i, i=1,5)], [2, -1, 3, -2, 5]>0)
```

résultant en des `indices` étant le tableau `[1,3,5]` .

Lire Procédures intrinsèques en ligne: https://riptutorial.com/fr/fortran/topic/2643/procedures-intrinseques

Chapitre 10: Programmation orientée objet

Examples

Définition de type dérivée

Fortran 2003 a introduit la prise en charge de la programmation orientée objet. Cette fonctionnalité permet de tirer parti des techniques de programmation modernes. Les types dérivés sont définis sous la forme suivante:

```
TYPE [[, attr-list] :: ] name [(name-list)]
   [def-stmts]
   [PRIVATE statement or SEQUENCE statement]. . .
   [component-definition]. . .
   [procedure-part]
END TYPE [name]
```

où,

- **attr-list** - une liste de spécificateurs d'attribut
- **name** - le nom du type de données dérivé
- **name-list** - une liste de noms de paramètres de type séparés par des virgules
- **def-stmts** - une ou plusieurs déclarations INTEGER des paramètres de type nommés dans la liste de noms
- **component-definition** - une ou plusieurs déclarations de type ou instructions de pointeur de procédure définissant le composant de type dérivé
- **procedure-part** - une instruction CONTAINS, éventuellement suivie d'une instruction PRIVATE et d'une ou plusieurs instructions de liaison de procédure

Exemple:

```
type shape
    integer :: color
end type shape
```

Procédures de type

Pour obtenir un comportement de type classe, le type et les procédures associées (sous-routine et fonctions) doivent être placés dans un module:

Exemple:

```
module MShape
    implicit none
    private

    type, public :: Shape
    private
```

```
        integer :: radius
    contains
        procedure :: set   => shape_set_radius
        procedure :: print => shape_print
    end type Shape

contains
    subroutine shape_set_radius(this, value)
        class(Shape), intent(in out) :: self
        integer, intent(in)          :: value

        self%radius = value
    end subroutine shape_set_radius

    subroutine shape_print(this)
        class(Shape), intent(in) :: self

        print *, 'Shape: r = ', self%radius
    end subroutine shape_print
end module MShape
```

Plus tard, dans un code, nous pouvons utiliser cette classe de forme comme suit:

```
! declare a variable of type Shape
type(Shape) :: shape

! call the type-bound subroutine
call shape%set(10)
call shape%print
```

Types dérivés abstraits

Un type dérivé extensible peut être *abstrait*

```
type, abstract :: base_type
end type
```

Un tel type dérivé ne peut jamais être instancié, par exemple en

```
type(base_type) t1
allocate(type(base_type) :: t2)
```

mais un objet polymorphe peut avoir ceci comme type déclaré

```
class(base_type), allocatable :: t1
```

ou

```
function f(t1)
  class(base_type) t1
end function
```

Les types abstraits peuvent avoir des composants et des procédures liées aux types

```
type, abstract :: base_type
  integer i
contains
  procedure func
  procedure(func_iface), deferred :: def_func
end type
```

La procédure `def_func` est une procédure de type *différé* avec interface `func_iface` . Une telle procédure liée à un type différé doit être implémentée par chaque type d'extension.

Extension de type

Un type dérivé est *extensible* s'il ne possède ni l'attribut `bind` ni l'attribut `sequence` . Un tel type peut être étendu par un autre type.

```
module mod

  type base_type
    integer i
  end type base_type

  type, extends(base_type) :: higher_type
    integer j
  end type higher_type

end module mod
```

Une variable polymorphe avec le type déclaré `base_type` est de type compatible avec le type `higher_type` et peut avoir cela comme type dynamique

```
class(base_type), allocatable :: obj
allocate(obj, source=higher_type(1,2))
```

La compatibilité de type descend à travers une chaîne d'enfants, mais un type ne peut étendre qu'un autre type.

Un type dérivé étendu hérite des procédures liées au type du parent, mais cela peut être remplacé

```
module mod

  type base_type
  contains
    procedure :: sub => sub_base
  end type base_type

  type, extends(base_type) :: higher_type
  contains
    procedure :: sub => sub_higher
  end type higher_type

contains

  subroutine sub_base(this)
    class(base_type) this
```

```
  end subroutine sub_base

  subroutine sub_higher(this)
    class(higher_type) this
  end subroutine sub_higher

end module mod

program prog
  use mod

  class(base_type), allocatable :: obj

  obj = base_type()
  call obj%sub

  obj = higher_type()
  call obj%sub

end program
```

Type constructeur

Des constructeurs personnalisés peuvent être créés pour les types dérivés en utilisant une interface pour surcharger le nom du type. De cette façon, les arguments de mots-clés qui ne correspondent pas aux composants peuvent être utilisés lors de la construction d'un objet de ce type.

```
module ball_mod
  implicit none

  ! only export the derived type, and not any of the
  ! constructors themselves
  private
  public :: ball

  type :: ball_t
     real :: mass
  end type ball_t

  ! Writing an interface overloading 'ball_t' allows us to
  ! overload the type constructor
  interface ball_t
     procedure :: new_ball
  end interface ball_t

contains

  type(ball_t) function new_ball(heavy)
    logical, intent(in) :: heavy

    if (heavy) then
       new_ball%mass = 100
    else
       new_ball%mass = 1
    end if

  end function new_ball
```

```
end module ball_mod

program test
  use ball_mod
  implicit none

  type(ball_t) :: football
  type(ball_t) :: boulder

  ! sets football%mass to 4.5
  football = ball_t(4.5)
  ! calls 'ball_mod::new_ball'
  boulder = ball_t(heavy=.true.)
end program test
```

Cela peut être utilisé pour créer une API plus efficace que l'utilisation de routines d'initialisation distinctes:

```
subroutine make_heavy_ball(ball)
  type(ball_t), intent(inout) :: ball
  ball%mass = 100
end subroutine make_heavy_ball

...

call make_heavy_ball(boulder)
```

Lire Programmation orientée objet en ligne: https://riptutorial.com/fr/fortran/topic/2374/programmation-orientee-objet

Chapitre 11: Tableaux

Examples

Notation de base

Tout type peut être déclaré en tant que tableau en utilisant soit l'attribut *dimension,* soit en indiquant directement la ou les `dimension` du tableau:

```
! One dimensional array with 4 elements
integer, dimension(4) :: foo

! Two dimensional array with 4 rows and 2 columns
real, dimension(4, 2) :: bar

! Three dimensional array
type(mytype), dimension(6, 7, 8) :: myarray

! Same as above without using the dimension keyword
integer :: foo2(4)
real :: bar2(4, 2)
type(mytype) :: myarray2(6, 7, 8)
```

La dernière manière de déclarer un tableau multidimensionnel permet la déclaration de tableaux de même rang / dimensions de même type dans une seule ligne, comme suit

```
real :: pencil(5), plate(3,-2:4), cuboid(0:3,-10:5,6)
```

Le rang maximum (nombre de dimensions) autorisé est de 15 dans le standard Fortran 2008 et était de 7 auparavant.

Fortran stocke les tableaux dans l'ordre des *colonnes* . Autrement dit, les éléments de `bar` sont stockés dans la mémoire comme suit:

```
bar(1, 1), bar(2, 1), bar(3, 1), bar(4, 1), bar(1, 2), bar(2, 2), ...
```

Dans Fortran, la numérotation des tableaux commence à **1** par défaut, contrairement à C qui commence à **0** . En fait, dans Fortran, vous pouvez spécifier explicitement les limites supérieure et inférieure de chaque dimension:

```
integer, dimension(7:12, -3:-1) :: geese
```

Cela déclare un tableau de forme `(6, 3)` , dont le premier élément est `geese(7, -3)` .

Les limites inférieures et supérieures le long des 2 dimensions (ou plus) sont accessibles par les fonctions intrinsèques `ubound` et `lbound` . En effet, `lbound(geese,2)` retournerait `-3` , tandis que `ubound(geese,1)` retournerait `12` .

La taille d'un tableau est accessible par la `size` fonction intrinsèque. Par exemple, `size(geese, dim = 1)` renvoie la taille de la première dimension qui est 6.

Tableaux pouvant être alloués

Les tableaux peuvent avoir l'attribut *allocatable* :

```
! One dimensional allocatable array
integer, dimension(:), allocatable :: foo
! Two dimensional allocatable array
real, dimension(:,:), allocatable :: bar
```

Cela déclare la variable mais ne lui alloue aucun espace.

```
! We can specify the bounds as usual
allocate(foo(3:5))

! It is an error to allocate an array twice
! so check it has not been allocated first
if (.not. allocated(foo)) then
  allocate(bar(10, 2))
end if
```

Une fois qu'une variable n'est plus nécessaire, elle peut être *libérée* :

```
deallocate(foo)
```

Si, pour une raison quelconque, une instruction d' `allocate` échoue, le programme s'arrêtera. Cela peut être évité si le statut est vérifié via le mot-clé `stat` :

```
real, dimension(:), allocatable :: geese
integer :: status

allocate(geese(17), stat=status)
if (stat /= 0) then
  print*, "Something went wrong trying to allocate 'geese'"
  stop 1
end if
```

L'instruction `deallocate` a également le mot clé `stat` :

```
deallocate (geese, stat=status)
```

`status` est une variable entière dont la valeur est 0 si l'allocation ou la désallocation a réussi.

Constructeurs de tableaux

Une valeur de tableau rank-1 peut être créée en utilisant un *constructeur de tableau* , avec la syntaxe

```
(/ ... /)
```

```
[ ... ]
```

La forme `[...]` été introduite dans Fortran 2003 et est généralement considérée comme plus lisible, en particulier dans les expressions complexes. Ce formulaire est utilisé exclusivement dans cet exemple.

Les valeurs figurant dans un constructeur de tableaux peuvent être des valeurs scalaires, des valeurs de tableau ou des boucles implicites.

Les paramètres type et type du tableau construit correspondent à ceux des valeurs du constructeur de tableau

```
[1, 2, 3]      ! A rank-1 length-3 array of default integer type
[1., 2., 3.]   ! A rank-1 length-3 array of default real type
["A", "B"]     ! A rank-1 length-2 array of default character type

integer, parameter :: A = [2, 4]
[1, A, 3]      ! A rank-1 length-4 array of default integer type, with A's elements

integer i
[1, (i, i=2, 5), 6]  ! A rank-1 length-6 array of default integer type with an implied-do
```

Dans les formulaires ci-dessus, toutes les valeurs données doivent être du même type et du même type. Les types de mélange ou les paramètres de type ne sont pas autorisés. Les exemples suivants **ne sont pas valides**

```
[1, 2.]      ! INVALID: Mixing integer and default real
[1e0, 2d0]   ! INVALID: Mixing default real and double precision
[1., 2._dp]  ! INVALID: Allowed only if kind `dp` corresponds to default real
["Hello", "Frederick"]  ! INVALID: Different length parameters
```

Pour construire un tableau utilisant différents types, une spécification de type pour le tableau doit être donnée

```
[integer :: 1, 2., 3d0]    ! A default integer array
[real(dp) :: 1, 2, 3._sp]  ! A real(dp) array
[character(len=9) :: "Hello", "Frederick"]  ! A length-2 array of length-9 characters
```

Cette dernière forme pour les tableaux de caractères est particulièrement pratique pour éviter le remplissage d'espace, comme l'alternative

```
["Hello    ", "Frederick"]  ! A length-2 array of length-9 characters
```

La taille d'un tableau nommé constante peut être impliquée par le constructeur de tableaux utilisé pour définir sa valeur

```
integer, parameter :: ids(*) = [1, 2, 3, 4]
```

et pour les types paramétrés en longueur, le paramètre de longueur peut être supposé

```
character(len=*), parameter :: names(*) = [character(3) :: "Me", "You", "Her"]
```

La spécification de type est également requise dans la construction de tableaux de longueur nulle. De

```
[ ] ! Not a valid array constructor
```

les paramètres de type et de type ne peuvent pas être déterminés à partir du jeu de valeurs non existant. Pour créer un tableau d'entiers par défaut de longueur nulle:

```
[integer :: ]
```

Les constructeurs de tableaux ne construisent que des tableaux de rang 1. Parfois, par exemple pour définir la valeur d'une constante nommée, des tableaux de rang supérieur sont également requis dans une expression. Les tableaux de rang supérieur peuvent être pris à partir du résultat de `reshape` avec un tableau de rang 1 construit

```
integer, parameter :: multi_rank_ids(2,2) = RESHAPE([1,2,3,4], shape=[2,2])
```

Dans un constructeur de tableaux, les valeurs du tableau dans l'ordre des éléments avec les tableaux de la liste de valeurs sont comme si les éléments individuels étaient eux-mêmes attribués dans l'ordre des éléments du tableau. Ainsi, l'exemple précédent

```
integer, parameter :: A = [2, 4]
[1, A, 3]      ! A rank-1 length-4 array of default integer type, with A's elements
```

est équivalent à

```
[1, 2, 4, 3]   ! With the array written out in array element order
```

En général, les valeurs du constructeur peuvent être des expressions arbitraires, y compris des constructeurs de tableaux imbriqués. Pour qu'un tel constructeur de tableau remplisse certaines conditions, comme être une expression de constante ou de spécification, des restrictions s'appliquent aux valeurs constitutives.

Bien que ce ne soit pas un constructeur de tableau, certaines valeurs de tableau peuvent également être créées de manière pratique en utilisant la fonction intrinsèque de `spread` . Par exemple

```
[(0, i=1,10)]  ! An array with 10 default integers each of value 0
```

est aussi le résultat de la référence de la fonction

```
SPREAD(0, 1, 10)
```

Spécification de la nature du tableau: rang et forme

L'attribut de `dimension` sur un objet spécifie que cet objet est un tableau. Il y a, dans Fortran 2008, cinq natures de tableau: [1]

- forme explicite
- forme supposée
- taille supposée
- forme différée
- forme implicite

Prenez les trois rangées [1]

```
integer a, b, c
dimension(5) a    ! Explicit shape (default lower bound 1), extent 5
dimension(:) b    ! Assumed or deferred shape
dimension(*) c    ! Assumed size or implied shape array
```

Avec ceux-ci, on peut voir qu'un contexte supplémentaire est nécessaire pour déterminer entièrement la nature d'un tableau.

Forme explicite

Un tableau de forme explicite est toujours la forme de sa déclaration. À moins que le tableau ne soit déclaré comme local à un sous-programme ou à une construction de `block` , les limites définissant la forme doivent être des expressions constantes. Dans d'autres cas, un tableau de forme explicite peut être un objet automatique, en utilisant des extensions qui peuvent varier à chaque appel d'un sous-programme ou d'un `block` .

```
subroutine sub(n)
  integer, intent(in) :: n
  integer a(5)   ! A local explicit shape array with constant bound
  integer b(n)   ! A local explicit shape array, automatic object
end subroutine
```

Forme supposée

Un tableau de forme supposé est un argument factice sans l' `allocatable` ou `pointer` . Un tel tableau prend sa forme à partir de l'argument réel auquel il est associé.

```
integer a(5), b(10)
call sub(a)   ! In this call the dummy argument is like x(5)
call sub(b)   ! In this call the dummy argument is like x(10)

contains

  subroutine sub(x)
    integer x(:)    ! Assumed shape dummy argument
  end subroutine sub
```

```
end
```

Lorsqu'un argument factice suppose que la portée faisant référence à la procédure doit avoir une interface explicite disponible pour cette procédure.

Taille supposée

Un tableau de taille supposé est un argument factice dont la taille est prise en compte dans son argument réel.

```
subroutine sub(x)
  integer x(*)    ! Assumed size array
end subroutine
```

Les tableaux de taille supposés se comportent très différemment des tableaux de formes supposés et ces différences sont documentées ailleurs.

Forme différée

Un tableau de forme différé est un tableau qui possède l' `allocatable` ou le `pointer` . La forme d'un tel tableau est déterminée par son affectation ou son affectation de pointeur.

```
integer, allocatable :: a(:)
integer, pointer :: b(:)
```

Forme implicite

Un tableau de forme implicite est une constante nommée qui prend sa forme à partir de l'expression utilisée pour établir sa valeur

```
integer, parameter :: a(*) = [1,2,3,4]
```

Les implications de ces déclarations de tableaux sur les arguments factices doivent être documentées ailleurs.

---

[1] Une spécification technique étendant Fortran 2008 ajoute une sixième nature de tableau: rang supposé. Ceci n'est pas couvert ici.

[2] Celles-ci peuvent être écrites de manière équivalente

```
integer, dimension(5) :: a
integer, dimension(:) :: b
integer, dimension(*) :: c
```

ou

```
integer a(5)
integer b(:)
integer c(*)
```

Tableaux entiers, éléments de tableau et sections de tableau

Considérons le tableau déclaré comme

```
real x(10)
```

Ensuite, nous avons trois aspects d'intérêt:

1. Le tableau entier `x` ;
2. Éléments de tableau, comme `x(1)` ;
3. Sections de tableau, comme `x(2:6)` .

Tableaux entiers

Dans la plupart des cas, le tableau entier `x` désigne tous les éléments du tableau en tant qu'entité unique. Il peut apparaître dans des instructions exécutables telles que `print *, SUM(x)` , `print *, SIZE(x)` ou `x=1` .

Un tableau entier peut faire référence à des tableaux qui ne sont pas explicitement formés (comme `x` ci-dessus):

```
function f(y)
  real, intent(out) :: y(:)
  real, allocatable :: z(:)

  y = 1.          ! Intrinsic assignment for the whole array
  z = [1., 2.,]   ! Intrinsic assignment for the whole array, invoking allocation
end function
```

Un tableau de taille supposée peut également apparaître sous la forme d'un tableau complet, mais uniquement dans des circonstances limitées (à documenter ailleurs).

Éléments de tableau

On appelle un élément tableau des index entiers, un pour chaque rang du tableau, indiquant l'emplacement dans l'ensemble du tableau:

```
real x(5,2)
x(1,1) = 0.2
x(2,4) = 0.3
```

Un élément de tableau est un scalaire.

Sections de tableau

Une section de tableau est une référence à un certain nombre d'éléments (peut-être un seul) d'un tableau entier, en utilisant une syntaxe impliquant les deux points:

```
real x(5,2)
x(:,1) = 0.          ! Referring to x(1,1), x(2,1), x(3,1), x(4,1) and x(5,1)
x(2,:) = 0.          ! Referring to x(2,1), x(2,2)
x(2:4,1) = 0.        ! Referring to x(2,1), x(3,1) and x(4,1)
x(2:3,1:2) = 0.      ! Referring to x(2,1), x(3,1), x(2,2) and x(3,2)
x(1:1,1) = 0.        ! Referring to x(1,1)
x([1,3,5],2) = 0.    ! Referring to x(1,2), x(3,2) and x(5,2)
```

La forme finale ci-dessus utilise un *indice vectoriel* . Ceci est soumis à un certain nombre de restrictions au-delà des autres sections de la baie.

Chaque section de tableau est elle-même un tableau, même lorsqu'un seul élément est référencé. C'est-à-dire que `x(1:1,1)` est un tableau de rang 1 et que `x(1:1,1:1)` est un tableau de rang 2.

Les sections de tableau n'ont généralement pas d'attribut de l'ensemble du tableau. En particulier, où

```
real, allocatable :: x(:)
x = [1,2,3]     ! x is allocated as part of the assignment
x = [1,2,3,4]   ! x is dealloacted then allocated to a new shape in the assignment
```

la tâche

```
x(:) = [1,2,3,4,5]   ! This is bad when x isn't the same shape as the right-hand side
```

n'est pas autorisé: `x(:)` , bien qu'une section de tableau avec tous les éléments de `x` , ne soit pas un tableau pouvant être alloué.

```
x(:) = [5,6,7,8]
```

c'est bien quand `x` la forme du côté droit.

Composants de tableaux de tableaux

```
type t
   real y(5)
end type t

type(t) x(2)
```

Nous pouvons également faire référence à des tableaux entiers, à des éléments de tableau et à des sections de tableau dans des paramètres plus complexes.

De ce qui précède, `x` est un tableau entier. Nous avons aussi

```
x(1)%y          ! A whole array
x(1)%y(1)       ! An array element
x%y(1)          ! An array section
x(1)%y(:)       ! An array section
x([1,2]%y(1)    ! An array section
x(1)%y(1:1)     ! An array section
```

Dans de tels cas, nous ne sommes pas autorisés à avoir plus d'une partie de la référence constituée d'un tableau de rang 1. Les éléments suivants, par exemple, ne sont pas autorisés

```
x%y               ! Both the x and y parts are arrays
x(1:1)%y(1:1)     ! Recall that each part is still an array section
```

Opérations de tableau

En raison de ses objectifs de calcul, les opérations mathématiques sur les tableaux sont simples dans Fortran.

Addition et soustraction

Les opérations sur des tableaux de même forme et de même taille sont très similaires à l'algèbre matricielle. Au lieu de parcourir tous les indices avec des boucles, on peut écrire l'addition (et la soustraction):

```
real, dimension(2,3) :: A, B, C
real, dimension(5,6,3) :: D
A    = 3.    ! Assigning single value to the whole array
B    = 5.    ! Equivalent writing for assignment
C    = A + B ! All elements of C now have value 8.
D    = A + B ! Compiler will raise an error. The shapes and dimensions are not the same
```

Les tableaux de découpage sont également valides:

```
integer :: i, j
real, dimension(3,2) :: Mat = 0.
real, dimension(3)   :: Vec1 = 0., Vec2 = 0., Vec3 = 0.
i = 0
j = 0
do i = 1,3
  do j = 1,2
    Mat(i,j) = i+j
  enddo
enddo
Vec1 = Mat(:,1)
Vec2 = Mat(:,2)
Vec3 = Mat(1:2,1) + Mat(2:3,2)
```

Fonction

De la même manière, la plupart des fonctions intrinsèques peuvent être implicitement utilisées en supposant un fonctionnement par composants (bien que cela ne soit pas systématique):

```
real, dimension(2) :: A, B
A(1) = 6
A(2) = 44 ! Random values
B    = sin(A) ! Identical to B(1) = sin(6), B(2) = sin(44).
```

Multiplication et division

Des précautions doivent être prises avec le produit et la division: les opérations intrinsèques utilisant des symboles `*` et `/` sont des éléments:

```
real, dimension(2) :: A, B, C
A(1) = 2
A(2) = 4
B(1) = 1
B(2) = 3
C = A*B ! Returns C(1) = 2*1 and C(2) = 4*3
```

Cela ne doit pas être confondu avec les opérations matricielles (voir ci-dessous).

Opérations matricielles

Les opérations matricielles sont des procédures intrinsèques. Par exemple, le produit matriciel des tableaux de la section précédente est écrit comme suit:

```
real, dimension(2,1) :: A, B
real, dimension(1,1) :: C
A(1) = 2
A(2) = 4
B(1) = 1
B(2) = 3
C = matmul(transpose(A),B) ! Returns the scalar product of vectors A and B
```

Les opérations complexes permettent d'encapsuler les fonctions en créant des tableaux temporaires. Bien que autorisé par certains compilateurs et options de compilation, ceci n'est pas recommandé. Par exemple, un produit comprenant une transposition de matrice peut être écrit:

```
real, dimension(3,3) :: A, B, C
A(:) = 4
B(:) = 5
C = matmul(transpose(A),matmul(B,matmul(A,transpose(B)))) ! Equivalent to A^t.B.A.B^T
```

Sections de tableau avancées: triplets en indice et indices de vecteur

Comme mentionné dans un autre exemple, un sous-ensemble des éléments d'un tableau, appelé section de tableau, peut être référencé. De cet exemple, nous pouvons avoir

```
real x(10)
x(:)   = 0.
x(2:6) = 1.
x(3:4) = [3., 5.]
```

Les sections de tableau peuvent être plus générales que cela, cependant. Ils peuvent prendre la forme de triplets d'indice ou d'indices vectoriels.

Triplets d'indices

Un indice triple prend la forme `[bound1]:[bound2][:stride]` . Par exemple

```
real x(10)
x(1:10) = ...   ! Elements x(1), x(2), ..., x(10)
x(1:) = ...     ! The omitted second bound is equivalent to the upper, same as above
x(:10) = ...    ! The omitted first bound is equivalent to the lower, same as above
x(1:6:2) = ...  ! Elements x(1), x(3), x(5)
x(5:1) = ...    ! No elements: the lower bound is greater than the upper
x(5:1:-1) = ... ! Elements x(5), x(4), x(3), x(2), x(1)
x(::3) = ...    ! Elements x(1), x(4), x(7), x(10), assuming omitted bounds
x(::-3) = ...   ! No elements: the bounds are assumed with the first the lower, negative
stride
```

Lorsqu'un stride (qui ne doit pas être zéro) est spécifié, la séquence d'éléments commence par la première limite spécifiée. Si la foulée est positive (resp. Négative), les éléments sélectionnés suivant une séquence incrémentée (resp. Décrémentée) par la foulée jusqu'à ce que le dernier élément non plus grand (resp. Plus petit) que la seconde soit pris. Si la foulée est omise, elle est considérée comme une seule.

Si la première limite est supérieure à la deuxième limite et que la foulée est positive, aucun élément n'est spécifié. Si la première limite est inférieure à la deuxième limite et que la foulée est négative, aucun élément n'est spécifié.

Il convient de noter que `x(10:1:-1)` est différent de `x(1:10:1)` même si chaque élément de `x` apparaît dans les deux cas.

Indices de vecteur

Un indice vectoriel est un tableau entier de rang 1. Cela désigne une séquence d'éléments correspondant aux valeurs du tableau.

```
real x(10)
integer i
x([1,6,4]) = ...     ! Elements x(1), x(6), x(4)
x([(i,i=2,4)]) = ... ! Elements x(2), x(3) and x(4)
print*, x([2,5,2])   ! Elements x(2), x(5) and x(2)
```

Une section de tableau avec un indice vectoriel est restreinte dans la façon dont elle peut être utilisée:

- il peut ne pas s'agir d'un argument associé à un argument factice défini dans la procédure;
- il peut ne pas être la cible dans une déclaration d'affectation de pointeur;
- il ne s'agit peut-être pas d'un fichier interne dans une déclaration de transfert de données.

De plus, une telle section de tableau peut ne pas apparaître dans une instruction qui implique sa

définition lorsque le même élément est sélectionné deux fois. D'en haut:

```
print*, x([2,5,2])   ! Elements x(2), x(5) and x(2) are printed
x([2,5,2]) = 1.      ! Not permitted: x(2) appears twice in this definition
```

Sections de tableau de rang supérieur

```
real x(5,2)
print*, x(::2,2:1:-1)  ! Elements x(1,2), x(3,2), x(5,2), x(1,1), x(3,1), x(5,1)
```

Lire Tableaux en ligne: https://riptutorial.com/fr/fortran/topic/996/tableaux

Chapitre 12: Types de données

Examples

Types intrinsèques

Les types de données suivants sont *intrinsèques* à Fortran:

```
integer
real
character
complex
logical
```

`integer` , `real` et `complex` sont des types numériques.

`character` est un type utilisé pour stocker des chaînes de caractères.

`logical` est utilisé pour stocker les valeurs binaires `.true.` ou `.false.` .

Tous les types intrinsèques numériques et logiques sont paramétrés à l'aide de types.

```
integer(kind=specific_kind)
```

ou juste

```
integer(specific_kind)
```

Où `specific_kind` est un entier nommé constant.

Les variables de caractère, tout comme le paramètre kind, ont également un paramètre de longueur:

```
character char
```

déclare `char` comme étant une variable de type longueur-1 de type par défaut, tandis que

```
character(len=len) name
```

déclare que le `name` est une variable de caractère de type par défaut et de longueur `len` . Le genre peut aussi être spécifié

```
character(len=len, kind=specific_kind) name
character(kind=specific_kind) char
```

déclare que le `name` est un personnage de genre `kind` et de longueur `len` . `char` est un caractère de longueur 1 de type `kind` .

Sinon, le formulaire obsolète pour la déclaration de caractères

```
character*len  name
```

peut être vu dans l'ancien code, déclarant que le `name` est de longueur `len` et le type de caractère par défaut.

La déclaration d'une variable de type intrinsèque peut être de la forme ci-dessus, mais peut également utiliser le `type(...)` :

```
integer i
real x
double precision y
```

est équivalent à (mais grandement préféré)

```
type(integer) i
type(real) x
type(double precision) y
```

Types de données dérivés

Définir un nouveau type, `mytype` :

```
type :: mytype
  integer :: int
  real    :: float
end type mytype
```

Déclarez une variable de type *mytype* :

```
type(mytype) :: foo
```

Les composants d'un type dérivé sont accessibles avec l'opérateur `%` [1] :

```
foo%int = 4
foo%float = 3.142
```

Une fonctionnalité de Fortran 2003 (non encore implémentée par tous les compilateurs) permet de définir des types de données paramétrés:

```
type, public :: matrix(rows, cols, k)
  integer, len :: rows, cols
  integer, kind :: k = kind(0.0)
  real(kind = k), dimension(rows, cols) :: values
end type matrix
```

Le type dérivé `matrix` a trois paramètres de type qui sont énumérés ci - après entre parenthèses le

nom du type (elles sont `rows` , `cols` , et `k`). Dans la déclaration de chaque paramètre de type, il faut indiquer s'il s'agit de paramètres de type type (`kind`) ou de longueur (`len`).

Les paramètres de type type, comme ceux des types intrinsèques, doivent être des expressions constantes, tandis que les paramètres de type longueur, comme la longueur d'une variable de caractère intrinsèque, peuvent varier au cours de l'exécution.

Notez que le paramètre `k` a une valeur par défaut, il peut donc être fourni ou omis quand une variable de type `matrix` est déclarée, comme suit

```
type (matrix (55, 65, kind=double)) :: b, c ! default parameter provided
type (matrix (rows=40, cols=50)     :: m    ! default parameter omitted
```

Le nom d'un type dérivé peut ne pas être `doubleprecision` ou identique à l'un des types intrinsèques.

---

1. Beaucoup de gens se demandent pourquoi Fortran utilise `%` comme opérateur d'accès aux composants, au lieu de le plus commun `.` . En effet `.` est déjà pris par la syntaxe de l'opérateur, c.-à-d `.not.` , `.and.` , `.my_own_operator.` .

Précision des nombres à virgule flottante

Les nombres à virgule flottante de type `real` ne peuvent avoir aucune valeur réelle. Ils peuvent représenter des nombres réels allant jusqu’à un certain nombre de chiffres décimaux.

FORTRAN 77 garanti deux types de virgule flottante et des normes plus récentes garantissent au moins deux types réels. Les variables réelles peuvent être déclarées comme

```
real x
double precision y
```

`x` ici est un vrai type par défaut et `y` est un vrai genre avec une plus grande précision décimale que `x` . Dans Fortran 2008, la précision décimale de `y` est d'au moins 10 et son exposant décimal est d'au moins 37.

```
 real, parameter             :: single = 1.12345678901234567890
 double precision, parameter :: double = 1.12345678901234567890d0

 print *, single
 print *, double
```

estampes

```
   1.12345684
   1.1234567890123457
```

dans des compilateurs communs utilisant la configuration par défaut.

Remarquez le `d0` dans la constante de double précision. Un littéral réel contenant `d` au lieu de `e` pour désigner l'exposant est utilisé pour indiquer une double précision.

```
! Default single precision constant
1.23e45
! Double precision constant
1.23d45
```

Fortran 90 a introduit `real` types `real` paramétrés à `real` aide de types. Le type d'un type réel est un entier nommé constant ou littéral constant:

```
real(kind=real_kind) :: x
```

ou juste

```
real(real_kind) :: x
```

Cette déclaration déclare `x` de type `real` avec une certaine précision en fonction de la valeur de `real_kind` .

Les littéraux à virgule flottante peuvent être déclarés avec un type spécifique en utilisant un suffixe

```
1.23456e78_real_kind
```

La valeur exacte de `real_kind` n'est pas normalisée et diffère d'un compilateur à l'autre. Pour connaître le type de toute variable ou constante réelle, la fonction `kind()` peut être utilisée:

```
print *, kind(1.0), kind(1.d0)
```

imprimera généralement

```
4 8
```

ou

```
1 2
```

en fonction du compilateur.

Les nombres types peuvent être définis de plusieurs manières:

1. Simple (par défaut) et double précision:

   ```
   integer, parameter :: single_kind = kind(1.)
   integer, parameter :: double_kind = kind(1.d0)
   ```

2. Utiliser la fonction intrinsèque `selected_real_kind([p, r])` pour spécifier la précision décimale requise. Le type retourné a une précision d'au moins `p` chiffres et permet d'exposer au moins `r` .

```
integer, parameter :: single_kind = selected_real_kind( p=6, r=37 )
integer, parameter :: double_kind = selected_real_kind( p=15, r=200 )
```

3. À partir de Fortran 2003, des constantes prédéfinies sont disponibles via le module intrinsèque `ISO_C_Binding` pour garantir l' `ISO_C_Binding` des types réels avec les types `float` , `double` ou `long_double` du compilateur C `long_double` :

```
use ISO_C_Binding

integer, parameter :: single_kind = c_float
integer, parameter :: double_kind = c_double
integer, parameter :: long_kind = c_long_double
```

4. À partir de Fortran 2008, des constantes prédéfinies sont disponibles via le module intrinsèque `ISO_Fortran_env` . Ces constantes fournissent des types réels avec une certaine taille de stockage en bits

```
use ISO_Fortran_env

integer, parameter :: single_kind = real32
integer, parameter :: double_kind = real64
integer, parameter :: quadruple_kind = real128
```

Si certains types ne sont pas disponibles dans le compilateur, la valeur renvoyée par `selected_real_kind()` ou la valeur de la constante entière est `-1` .

Paramètres de type de longueur supposés et différés

Les variables de type caractère ou de type dérivé avec paramètre de longueur peuvent avoir le paramètre de longueur *pris* ou *différé* . Le `name` variable de caractère

```
character(len=len) name
```

est de longueur `len` tout au long de l'exécution. Inversement, le spécificateur de longueur peut être soit

```
character(len=*) ...  ! Assumed length
```

ou

```
character(len=:) ...  ! Deferred length
```

Les variables de longueur supposées prennent leur longueur à partir d'une autre entité.

Dans la fonction

```
function f(dummy_name)
  character(len=*) dummy_name
```

```
end function f
```

l'argument factice `dummy_name` a la longueur de l'argument réel.

La constante nommée `const_name` dans

```
character(len=*), parameter :: const_name = 'Name from which length is assumed'
```

a la longueur donnée par l'expression constante du côté droit.

Les paramètres de type longueur différée peuvent varier au cours de l'exécution. Une variable avec une longueur différée doit avoir l' `allocatable` ou le `pointer` attribuable

```
character(len=:), allocatable :: alloc_name
character(len=:), pointer :: ptr_name
```

La longueur d'une telle variable peut être définie de l'une des manières suivantes:

```
allocate(character(len=5) :: alloc_name, ptr_name)
alloc_name = 'Name'         ! Using allocation on intrinsic assignment
ptr_name => another_name    ! For given target
```

Pour les types dérivés avec paramétrage de longueur, la syntaxe est similaire

```
  type t(len)
    integer, len :: len
    integer i(len)
  end type t

  type(t(:)), allocatable :: t1
  type(t(5)) t2

  call sub(t2)
  allocate(type(t(5)) :: t1)

contains

  subroutine sub(t2)
    type(t(*)), intent(out) :: t2
  end subroutine sub

end
```

Constantes littérales

Les unités de programme utilisent souvent des constantes littérales. Celles-ci couvrent les cas évidents comme

```
print *, "Hello", 1, 1.0
```

Sauf dans un cas, chaque constante littérale est un scalaire qui a le type, les paramètres de type et la valeur donnée par la syntaxe.

Les constantes littérales entières sont de la forme

```
1
-1
-1_1   ! For valid kind parameter 1
1_ik   ! For the named constant ik being a valid kind paramter
```

Les constantes littérales réelles sont de la forme

```
1.0    ! Default real
1e0    ! Default real using exponent format
1._1   ! Real with kind parameter 1 (if valid)
1.0_sp ! Real with kind paramter named constant sp
1d0    ! Double precision real using exponent format
1e0_dp ! Real with kind named constant dp using exponent format
```

Les constantes littérales complexes sont de la forme

```
(1, 1.)       ! Complex with integer and real components, literal constants
(real, imag)  ! Complex with named constants as components
```

Si les composants réels et imaginaires sont tous deux des nombres entiers, la constante littérale complexe est complexe par défaut et les composants entiers sont convertis en réels par défaut. Si un composant est réel, le paramètre kind de la constante littérale complexe est celui du réel (et le composant entier est converti en ce type réel). Si les deux composants sont réels, la constante littérale complexe est du type réel avec la plus grande précision.

Les constantes littérales logiques sont

```
.TRUE.     ! Default kind, with true value
.FALSE.    ! Default kind, with false value
.TRUE._1   ! Of kind 1 (if valid), with true value
.TRUE._lk  ! Of kind named constant lk (if valid), with true value
```

Les valeurs littérales des caractères diffèrent légèrement dans le concept, en ce sens que le spécificateur de type précède la valeur

```
"Hello"       ! Character value of default kind
'Hello'       ! Character value of default kind
ck_"Hello"    ! Character value of kind ck
"'Bye"        ! Default kind character with a '
'''Bye'       ! Default kind character with a '
""            ! A zero-length character of default kind
```

Comme suggéré ci-dessus, les constantes littérales de caractères doivent être délimitées par des apostrophes ou des guillemets, et les marqueurs de début et de fin doivent correspondre. Les apostrophes littérales peuvent être incluses en étant dans les délimiteurs de guillemets ou en apparaissant doublées. La même chose pour les guillemets.

Les constantes BOZ sont distinctes de celles ci-dessus, car elles ne spécifient qu'une valeur: elles n'ont pas de paramètre de type ou de type. Une constante BOZ est un modèle de bit et est spécifiée comme

```
B'00000'    ! A binary bit pattern
B"01010001" ! A binary bit pattern
O'012517'   ! An octal bit pattern
O"1267671"  ! An octal bit pattern
Z'0A4F'     ! A hexadecimal bit pattern
Z"FFFFFF"   ! A hexadecimal bit pattern
```

Les constantes littérales BOZ se limitent à leur apparition: constantes dans `data` instructions de `data` et sélection de procédures intrinsèques.

Accès aux sous-chaînes de caractères

Pour l'entité de caractère

```
character(len=5), parameter :: greeting = "Hello"
```

une sous-chaîne peut être référencée avec la syntaxe

```
greeting(2:4)  ! "ell"
```

Pour accéder à une seule lettre, il ne suffit pas d’écrire

```
greeting(1)    ! This isn't the letter "H"
```

mais

```
greeting(1:1)  ! This is "H"
```

Pour un tableau de caractères

```
character(len=5), parameter :: greeting(2) = ["Hello", "Yo!  "]
```

nous avons accès à la sous-chaîne comme

```
greeting(1)(2:4)  ! "ell"
```

mais nous ne pouvons pas référencer les caractères non contigus

```
greeting(:)(2:4)  ! The parent string here is an array
```

Nous pouvons même accéder à des sous-chaînes de constantes littérales

```
"Hello"(2:4)
```

Une partie d'une variable de caractère peut également être définie en utilisant une sous-chaîne en tant que variable. Par exemple

```
integer :: i=1
character :: filename = 'file000.txt'

filename(9:11) = 'dat'
write(filename(5:7), '(I3.3)') i
```

Accéder à des composants complexes

L'entité complexe

```
complex, parameter :: x = (1., 4.)
```

a la partie réelle `1.` et la partie complexe `4.` .. Nous pouvons accéder à ces composants individuels comme

```
real(x)  ! The real component
aimag(x) ! The complex component
x%re     ! The real component
y%im     ! The complex component
```

La forme `x%..` est nouvelle dans Fortran 2008 et n'est pas largement prise en charge dans les compilateurs. Cette forme, cependant, peut être utilisée pour définir directement les composants individuels d'une variable complexe

```
complex y
y%re = 0.
y%im = 1.
```

Déclaration et attributs

Tout au long des sujets et des exemples, nous verrons de nombreuses déclarations de variables, de fonctions, etc.

En plus de leur nom, les objets de données peuvent avoir des *attributs* . Couverts dans cette rubrique sont des déclarations de déclaration comme

```
integer, parameter :: single_kind = kind(1.)
```

ce qui donne à l'objet `single_kind` l'attribut du `parameter` (ce qui en fait une constante nommée).

Il y a beaucoup d'autres attributs, comme

- `target`
- `pointer`
- `optional`
- `save`

Les attributs peuvent être spécifiés avec *des instructions de spécification d'attribut*

```
integer i     ! i is an integer (of default kind)...
pointer i     ! ... with the POINTER attribute...
optional i    ! ... and the OPTIONAL attribute
```

Cependant, il est généralement considéré comme préférable d'éviter d'utiliser ces instructions de spécification d'attribut. Pour plus de clarté, les attributs peuvent être spécifiés dans le cadre d'une déclaration unique

```
integer, pointer, optional :: i
```

Cela réduit également la tentation d'utiliser un typage implicite.

Dans la plupart des cas, dans cette documentation Fortran, cette déclaration de déclaration unique est préférable.

Lire Types de données en ligne: https://riptutorial.com/fr/fortran/topic/939/types-de-donnees

Chapitre 13: Unités de programme et disposition des fichiers

Examples

Programmes Fortran

Un programme complet de Fortran est composé d'un certain nombre d'unités de programme distinctes. Les unités de programme sont:

- programme principal
- sous-programme fonction ou sous-programme
- module ou sous-module
- bloc de programme de données

Le programme principal et certains sous-programmes de procédure (fonction ou sous-programme) peuvent être fournis par une langue autre que Fortran. Par exemple, un programme principal C peut appeler une fonction définie par un sous-programme de la fonction Fortran, ou un programme principal Fortran peut appeler une procédure définie par C.

Ces unités de programme Fortran peuvent être des fichiers distincts ou dans un seul fichier.

Par exemple, nous pouvons voir les deux fichiers:

prog.f90

```
program main
  use mod
end program main
```

mod.f90

```
module mod
end module mod
```

Et le compilateur (invoqué correctement) pourra associer le programme principal au module.

Le fichier unique peut contenir plusieurs unités de programme

tout.f90

```
module mod
end module mod

program prog
  use mod
end program prog
```

```
function f()
end function f()
```

Dans ce cas, cependant, il faut noter que la fonction `f` est toujours une *fonction externe* en ce qui concerne le programme principal et le module. Le module sera cependant accessible par le programme principal.

Les règles de portée de saisie s'appliquent à chaque unité de programme individuelle et non au fichier dans lequel elles sont contenues. Par exemple, si nous voulons que chaque unité de portée n'ait pas de saisie implicite, le fichier ci-dessus doit être écrit en tant que

```
module mod
  implicit none
end module mod

program prog
  use mod
  implicit none
end program prog

function f()
  implicit none
  <type> f
end function f
```

Modules et sous-modules

Les modules sont documentés ailleurs .

Les compilateurs génèrent souvent des *fichiers* appelés *modules* : généralement le fichier contenant

```
module my_module
end module
```

se traduira par un fichier nommé quelque chose comme `my_module.mod` par le compilateur. Dans de tels cas, pour qu'un module soit accessible par une unité de programme, ce fichier de module doit être visible avant le traitement de cette dernière unité de programme.

Procédures externes

Une procédure externe est une procédure définie à l'extérieur d'une autre unité de programme ou par un autre moyen que Fortran.

La fonction contenue dans un fichier comme

```
integer function f()
  implicit none
end function f
```

est une fonction externe.

Pour les procédures externes, leur existence peut être déclarée en utilisant un bloc d'interface (pour donner une interface explicite)

```
program prog
  implicit none
  interface
    integer function f()
  end interface
end program prog
```

ou par une déclaration pour donner une interface implicite

```
program prog
  implicit none
  integer, external :: f
end program prog
```

ou même

```
program prog
  implicit none
  integer f
  external f
end program prog
```

L'attribut `external` n'est pas nécessaire:

```
program prog
  implicit none
  integer i
  integer f
  i = f()   ! f is now an external function
end program prog
```

Bloquer les unités de programme de données

Les unités de programme de données en bloc sont des unités de programme qui fournissent des valeurs initiales pour les objets dans les blocs communs. Ceux-ci sont délibérément laissés sans papiers ici, et figureront dans la documentation des fonctionnalités historiques de Fortran.

Sous-programmes internes

Une unité de programme qui n’est pas un sous-programme interne peut contenir d’autres unités de programme, appelées *sous-programmes internes* .

```
program prog
  implicit none
contains
  function f()
  end function f
```

```
  subroutine g()
  end subroutine g
end program
```

Un tel sous-programme interne présente plusieurs caractéristiques:

- il y a association d'hôte entre les entités dans le sous-programme et le programme externe
- les règles de typage implicites sont héritées (`implicit none` dans `f` ci-dessus)
- les sous-programmes internes ont une interface explicite disponible dans l'hôte

Les sous-programmes de module et les sous-programmes externes peuvent avoir des sous-programmes internes, tels que

```
module mod
  implicit none
contains
  function f()
  contains
    subroutine s()
    end subroutine s
  end function f
end module mod
```

Fichiers de code source

Un fichier de code source est un fichier texte (généralement) à traiter par le compilateur. Un fichier de code source peut contenir jusqu'à un programme principal et un nombre quelconque de modules et de sous-programmes externes. Par exemple, un fichier de code source peut contenir les éléments suivants:

```
module mod1
end module mod1

module mod2
end module mod2

function func1()    ! An external function
end function func1

subroutine sub1()   ! An external subroutine
end subroutine sub1

program prog        ! The main program starts here...
end program prog    ! ... and ends here

function func2()    ! An external function
end function func2
```

Rappelons ici que, même si les sous-programmes externes sont donnés dans le même fichier que les modules et le programme principal, les sous-programmes externes ne sont connus explicitement par aucun autre composant.

Les composants individuels peuvent également être répartis sur plusieurs fichiers et même

compilés à des moments différents. La documentation du compilateur doit être lue sur la façon de combiner plusieurs fichiers en un seul programme.

Un seul fichier de code source peut contenir un code source de forme fixe ou libre: ils ne peuvent pas être mélangés, bien que plusieurs fichiers combinés au moment de la compilation puissent avoir des styles différents.

Pour indiquer au compilateur la forme source, il existe généralement deux options:

- choix du suffixe du nom de fichier
- utilisation des drapeaux du compilateur

L'indicateur de compilation pour indiquer une source de forme fixe ou libre peut être trouvé dans la documentation du compilateur.

Les suffixes de nom de fichier significatifs se trouvent également dans la documentation du compilateur, mais en règle générale, un fichier nommé `file.f90` est considéré comme contenant une source de forme libre, tandis que le fichier `file.f` contient une source de forme fixe.

L'utilisation du suffixe `.f90` pour indiquer la source de forme libre (introduite dans la norme Fortran 90) incite souvent le programmeur à utiliser le suffixe pour indiquer la norme de langue à laquelle le code source est conforme. Par exemple, nous pouvons voir des fichiers avec des suffixes `.f03` ou `.f08` . Ceci est généralement à éviter: la plupart des sources de Fortran 2003 sont également compatibles avec Fortran 77, Fortran 90/5 et Fortran 2008. De plus, de nombreux comilers ne considèrent pas automatiquement ces suffixes.

Les compilateurs proposent également souvent un préprocesseur de code intégré (généralement basé sur cpp). De nouveau, un indicateur de compilation peut être utilisé pour indiquer que le préprocesseur doit être exécuté avant la compilation, mais le suffixe du fichier de code source peut également indiquer une telle exigence de prétraitement.

Pour les systèmes de fichiers sensibles à la casse, le fichier `file.F` est souvent considéré comme un fichier source à `file.F90` à `file.F90` et le `file.F90` est un fichier source de forme libre à traiter préalablement. Comme précédemment, la documentation du compilateur doit être consultée pour ces indicateurs et suffixes de fichiers.

Lire Unités de programme et disposition des fichiers en ligne: https://riptutorial.com/fr/fortran/topic/2203/unites-de-programme-et-disposition-des-fichiers

Chapitre 14: Utilisation des modules

Examples

La syntaxe du module

Le module est un ensemble de déclarations de type, de déclarations de données et de procédures. La syntaxe de base est la suivante:

```
module module_name
  use other_module_being_used

  ! The use of implicit none here will set it for the scope of the module.
  ! Therefore, it is not required (although considered good practice) to repeat
  ! it in the contained subprograms.
  implicit none

  ! Parameters declaration
  real, parameter, public :: pi = 3.14159
  ! The keyword private limits access to e parameter only for this module
  real, parameter, private :: e = 2.71828

  ! Type declaration
  type my_type
    integer :: my_int_var
  end type

  ! Variable declaration
  integer :: my_integer_variable

! Subroutines and functions belong to the contains section
contains

  subroutine my_subroutine
    !module variables are accessible
    print *, my_integer_variable
  end subroutine

  real function my_func(x)
    real, intent(in) :: x
    my_func = x * x
  end function my_func
end module
```

Utilisation de modules d'autres unités de programme

Pour accéder aux entités déclarées dans un module à partir d'une autre unité de programme (module, procédure ou programme), le module doit être *utilisé* avec l'instruction `use` .

```
module shared_data
  implicit none

  integer :: iarray(4) = [1, 2, 3, 4]
```

```
  real :: rarray(4) = [1., 2., 3., 4.]
end module


program test

  !use statements most come before implicit none
  use shared_data

  implicit none

  print *, iarray
  print *, rarray
end program
```

L'instruction `use` prend en charge l'importation uniquement des noms sélectionnés

```
program test

  !only iarray is accessible
  use shared_data, only: iarray

  implicit none

  print *, iarray

end program
```

Les entités peuvent également être consultées sous un nom différent en utilisant une *liste de renommage* :

```
program test

  !only iarray is locally renamed to local_name, rarray is still acessible
  use shared_data, local_name => iarray

  implicit none

  print *, local_name

  print *, rarray

end program
```

En outre, renommer peut être combiné avec la `only` option

```
program test
  use shared_data, only : local_name => iarray
end program
```

de sorte que seule l’entité de module `iarray` est accessible, mais qu’elle porte le nom local `local_name` .

Si elle est sélectionnée pour les noms que l' importation *privée* marque vous ne pouvez pas les importer dans votre programme.

Modules intrinsèques

Fortran 2003 a introduit des modules intrinsèques qui donnent accès à des constantes nommées spéciales, à des types dérivés et à des procédures de module. Il existe maintenant cinq modules intrinsèques standard:

- `ISO_C_Binding` ; soutenir l'interopérabilité C;
- `ISO_Fortran_env` ; détaillant l'environnement Fortran;
- `IEEE_Exceptions` , `IEEE_Arithmetic` et `IEEE_Features` ; supportant l'installation arithmétique dite IEEE.

Ces modules intrinsèques font partie de la bibliothèque Fortran et sont accessibles comme d'autres modules, sauf que la déclaration d' `use` peut avoir la nature intrinsèque explicitement énoncée:

```
use, intrinsic :: ISO_C_Binding
```

Cela garantit que le module intrinsèque est utilisé lorsqu'un module fourni par l'utilisateur du même nom est disponible. inversement

```
use, non_intrinsic :: ISO_C_Binding
```

s'assure que ce même module fourni par l'utilisateur (qui doit être accessible) est accessible à la place du module intrinsèque. Sans la nature de module spécifiée comme dans

```
use ISO_C_Binding
```

un module non intrinsèque disponible sera préféré au module intrinsèque.

Les modules IEEE intrinsèques sont différents des autres modules en ce sens que leur accessibilité dans une unité de portée peut modifier le comportement du code même sans référence à aucune des entités qui y sont définies.

Contrôle d'accès

L'accessibilité des symboles déclarés dans un module peut être contrôlée à l'aide `public` attributs et d'énoncés `private` et `public` .

Syntaxe du formulaire de déclaration:

```
!all symbols declared in the module are private by default
private

!all symbols declared in the module are public by default
public

!symbols in the list will be private
private :: name1, name2
```

```
!symbols in the list will be public
public :: name3, name4
```

Syntaxe de la forme d'attribut:

```
integer, parameter, public :: maxn = 1000

real, parameter, private :: local_constant = 42.24
```

Les symboles publics sont accessibles depuis les unités de programme utilisant le module, mais pas les symboles privés.

Si aucune spécification n'est utilisée, la valeur par défaut est `public` .

La spécification d'accès par défaut en utilisant

```
private
```

ou

```
public
```

peut être modifié en spécifiant un accès différent avec l' *entité-déclaration-liste*

```
public :: name1, name2
```

ou en utilisant des attributs.

Ce contrôle d'accès affecte également les symboles importés d'un autre module:

```
module mod1
  integer :: var1
end module

module mod2
  use mod1, only: var1

  public
end module

program test
  use mod2, only: var1
end program
```

est possible, mais

```
module mod1
  integer :: var1
end module

module mod2
```

```
  use mod1, only: var1

  public
  private :: var1
end module

program test
  use mod2, only: var1
end program
```

n'est pas possible car `var` est privé dans `mod2` .

Entités de module protégées

En plus de permettre aux entités de module d'avoir un contrôle d'accès (étant `public` ou `private`), les entités de modules peuvent également avoir l'attribut `protect` . Une entité publique protégée peut être associée, mais l'entité utilisée est soumise à des restrictions d'utilisation.

```
module mod
  integer, public, protected :: i=1
end module

program test
  use mod, only : i
  print *, i   ! We are allowed to get the value of i
  i = 2        ! But we can't change the value
end program test
```

Une cible publique protégée n'est pas autorisée à pointer en dehors de son module

```
module mod
  integer, public, target, protected :: i
end module mod

program test
  use mod, only : i
  integer, pointer :: j
  j => i   ! Not allowed, even though we aren't changing the value of i
end program test
```

Pour un pointeur protégé dans un module, les restrictions sont différentes. Ce qui est protégé est le statut d'association du pointeur

```
module mod
  integer, public, target :: j
  integer, public, protected, pointer :: i => j
end module mod

program test
  use mod, only : i
  i = 2   ! We may change the value of the target, just not the association status
end program test
```

Comme pour les pointeurs variables, les pointeurs de procédure peuvent également être

protégés, empêchant à nouveau le changement d'association cible.

Lire Utilisation des modules en ligne: https://riptutorial.com/fr/fortran/topic/1139/utilisation-des-modules

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec Fortran	Alexander Vogt, Community, Enrico Maria De Angelis, Gilles, haraldkl, High Performance Mark, Ingve, innoSPG, milancurcic, packet0, RamenChef, Serenity, Vladimir F, Yossarian
2	C interopérabilité	Serenity, Yossarian
3	Contrôle de l'exécution	Enrico Maria De Angelis, francescalus, haraldkl, ptev, Serenity, syscreat, TTT, Vladimir F
4	Des alternatives modernes aux caractéristiques historiques	Brian Tompsett - , d_1999, Enrico Maria De Angelis, francescalus, Serenity, TTT, Vladimir F, Yossarian
5	Extensions du fichier source (.f, .f90, .f95, ...) et leur relation avec le compilateur.	Arun
6	I / O	AL-P, Ed Smith, francescalus, Kyle Kanos, TTT
7	Interfaces explicites et implicites	Enrico Maria De Angelis, Serenity, Vladimir F
8	Procédures - Fonctions et sous-programmes	Alexander Vogt, Enrico Maria De Angelis, francescalus, haraldkl , Serenity, Vladimir F, Yossarian
9	Procédures intrinsèques	francescalus
10	Programmation orientée objet	Enrico Maria De Angelis, francescalus, syscreat, Yossarian
11	Tableaux	Enrico Maria De Angelis, francescalus, G.Clavier, Gilles, Serenity, TTT, Vladimir F, Yossarian
12	Types de données	Alexander Vogt, Enrico Maria De Angelis, francescalus, Vladimir F, Yossarian
13	Unités de programme et	agentp, francescalus, haraldkl, trblnc

	disposition des fichiers	
14	Utilisation des modules	Alexander Vogt, Enrico Maria De Angelis, francescalus, Serenity, Vladimir F