Bonnes pratiques en Fortran moderne

Du plus évident au plus discutable

Baptiste MARY https://www.idref.fr/169675203 [1]
Vincent LAFAGE [2]

[1]CY Cergy Paris Université Centre de Calcul / DirNum Cergy, France

[2]IJCLab, Laboratoire de Physique des 2 Infinis Irène Joliot-Curie
CNRS/IN2P3 & Université Paris-Saclay, Orsay, France

Journée Fortran 2026

Décrire des propriétés, pas des optimisations



Plan

1. Philosophie
2. Style et pièges
3. Le socle consensuel
4. Attributs : les garde-fous
5. Structures adaptées
6. Les points discutables
7. Le compilateur comme garde-fou
8. Outils et coarrays
9. Conclusion



Plan

1. Philosophie
2. Style et pièges
3. Le socle consensuel
4. Attributs : les garde-fous
5. Structures adaptées
6. Les points discutables
7. Le compilateur comme garde-fou
8. Outils et coarrays
9. Conclusion



Décrire des propriétés, pas des optimisations

- Déléguer le plus possible les optimisations au compilateur
- Fortran est simple et dispose de garde-fous pour nous empêcher de faire des bêtises (comme Ada ou Rust)
- Utiliser le plus possible :
  - les intrinsèques
  - les attributs de types et de procédures : tout ce qui aide le compilateur à vérifier ce que l'on veut faire
  - les structures adaptées

Idée directrice

Plus on dit au compilateur *ce que* l'on veut (et non *comment*), plus il peut vérifier, optimiser et même déporter le calcul (vectorisation, GPU).

CC BY SA



Une tension entre syntaxe Fortran et usages HPC

- Les outils externes (OpenMP…) poussent à des syntaxes plus complexes, notamment des boucles imbriquées, là où Fortran offre des syntaxes plus expressives (opérations vectorielles)
- Pas de barrière technique infranchissable : le langage a été conçu pour la performance. Plutôt une affaire de « parts de marché » en faveur de C et C++
- Cela peut changer si les utilisateurs manifestent un besoin auprès des constructeurs
  - NVIDIA l'a compris : déport automatique sur GPU à partir de la syntaxe Fortran standard (`do concurrent`, avec `nvfortran -stdpar=gpu`)
  - la directive OpenMP `workshare` (parallélisation d'une affectation vectorielle, compatible avec `where`) a longtemps été peu ou pas parallélisée par certains compilateurs

CC BY SA



Feuille de route : du plus évident au plus discutable

Consensuel

- `implicit none`
- `use, only:`
- `private` par défaut
- interfaces explicites
- *kinds*, pas de `double precision`
- opérateurs et littéraux modernes

Recommandé

- intrinsèques
- attributs : `intent`, `pure`, `elemental`
- structures adaptées
- `allocatable` avant `pointer`
- analyse statique et dynamique

Discutable

- expressions tableau *vs* `do concurrent`
- constructeurs et objets immuables
- *submodules*
- *thin* / *thick* binding
- `-Werror`

CC BY SA



Plan

1 Philosophie

2 Style et pièges

3 Le socle consensuel

4 Attributs : les garde-fous

5 Structures adaptées

6 Les points discutables

7 Le compilateur comme garde-fou

8 Outils et coarrays

9 Conclusion



Style moderne

- Format libre (lignes jusqu'à 10 000 caractères et 1 000 000 par déclaration depuis la norme 2023)
- Casse basse ; majuscules réservées aux constantes, au *camelCase*, aux types ou à toute autre convention cohérente
- Noms courts/abrégés classiques en Fortran : ils représentent souvent des entités mathématiques connues du lecteur (31 caractères max avant 2003, 63 depuis)
- Comparaisons et tableaux littéraux modernes :

pas bien

```
if (x .lt. 0.0) x = 0.0
if (i .ne. j) call f()
v = (/ 1, 2, 3 /)
```

bien

```
if (x < 0.0) x = 0.0
if (i /= j) call f()
v = [1, 2, 3]
```


CC BY SA



Quatre pièges classiques (*gotchas*)

1. Typage implicite : une faute de frappe crée une nouvelle variable
2. *Save* implicite : une variable initialisée à la déclaration conserve sa valeur d'un appel à l'autre
3. Littéraux flottants : `0.1` est un réel simple précision, même affecté à un réel double
4. Indice de tableau ou appel de fonction ? Même syntaxe : `f(x)`

Bonne nouvelle

Les modules, `implicit none` et quelques options de compilation écartent l'essentiel de ces pièges.

CC BY SA



Piège 1 : le typage implicite

Par défaut, toute variable non déclarée est typée selon sa première lettre (`i–n` : entier, sinon réel).

```
program surprise
    real :: tolerance
    tolerance = 1.0e-6
    if (abs(x) < tolerence) print *, "converge"   ! faute de frappe !
end program surprise
```

- `tolerence` et `x` sont créés silencieusement : réels, non initialisés
- Source d'erreurs parfois très difficiles à déboguer
- Sauf pour écrire son code sur carte perforée ou utiliser un interpréteur : le désactiver partout

CC BY SA



Piège 2 : le *save* implicite

```
subroutine compte(n)
    implicit none
    integer, intent(out) :: n
    integer :: k = 0        ! équivaut à  integer, save :: k = 0
    k = k + 1
    n = k
end subroutine compte
```

- Deux appels successifs retournent 1 puis 2 : l'initialisation n'a lieu qu'une fois
- Aucun avertissement avec `-Wall -Wextra` (observé avec gfortran 13)
- À faire : séparer déclaration et affectation, et écrire `save` quand c'est voulu

```
    integer :: k
    k = 0                   ! réinitialisé à chaque appel
```


CC BY SA



Piège 3 : les littéraux flottants

```
real(wp), parameter :: a = 0.1        ! 0.1 simple précision, converti
real(wp), parameter :: b = 0.1_wp     ! 0.1 dans le kind voulu
print '(2f22.18)', a, b
```

Résultat (`wp` = `real64`) :

```
  0.100000001490116119
  0.100000000000000006
```

- Le suffixe `_wp` est indispensable pour chaque littéral
- `-Wconversion` (inclus dans `-Wall`) signale les conversions implicites, mais pas celles d'un littéral constant : rester vigilant

CC BY SA



Piège 4 : indice de tableau ou appel de fonction ?

```
x = v(2)          ! élément de tableau
y = sqrt(2.0)     ! appel de fonction : même syntaxe
z = f(3)          ! tableau ? fonction externe ?
```

- Sans déclaration, `f(3)` est interprété comme un appel de fonction externe : l'erreur n'apparaît qu'à l'édition de liens, voire à l'exécution
- Remèdes :
  - `implicit none` : l'identifiant doit être déclaré
  - interfaces explicites : modules, ou blocs `interface` pour les bibliothèques anciennes
  - `-Wimplicit-interface` et `-Wimplicit-procedure`

CC BY SA



Plan

1 Philosophie

2 Style et pièges

3 Le socle consensuel

4 Attributs : les garde-fous

5 Structures adaptées

6 Les points discutables

7 Le compilateur comme garde-fou

8 Outils et coarrays

9 Conclusion



implicit none : partout

- À répéter dans chaque « partie » (program, module, procédures isolées) ; sa portée s'étend aux sous-éléments
- Position : après les use, avant tout le reste

```
program demo
    use iso_fortran_env, only: real64
    implicit none
    real(real64) :: x, y
    x = 1.0_real64
    y = 2.0_real64
    print *, add(x, y)
contains
    function add(a, b) result(c)
        real(real64), intent(in) :: a, b
        real(real64) :: c
        c = a + b
    end function add
end program demo
```


CC BY SA



implicit none (type, external)

- Fortran 2018 : `implicit none (type, external)`
  - `type` : plus de typage implicite
  - `external` : plus d'appel de procédure externe sans interface explicite
- Sans modifier le source, options de compilation :

compilateur	option
gfortran	`-fimplicit-none`
nvfortran	`-Mdclchk`
ifx	`-implicitnone`

```
program test
    implicit none (type, external)
    real :: y
    call legacy(y)     ! erreur : pas d'interface explicite
end program test
```


CC BY SA



Modules : organiser, partager, vérifier

Les modules (Fortran 90) permettent de :

- organiser son code
- diffuser variables (en remplacement des `common`) et procédures
- laisser le compilateur vérifier les appels, sans interfaces à écrire

```
module my_module
    implicit none
    real, parameter :: pi = 4*atan(1.0)
contains
    function add(a, b) result(c)
        real, intent(in) :: a, b
        real :: c
        c = a + b
    end function add
end module my_module
```


CC BY SA



use, only: systématique

```
program truc
    use my_module, only: pi, add
    implicit none
    real :: x, y
    x = 1.0
    y = 2.0
    print *, pi * add(x, y)
end program truc
```

- Sans only, use importe tout le contenu public du module
- Avec only : on retrouve facilement l'origine de chaque entité, surtout avec plusieurs modules
- En cas de collision de noms, le problème est plus simple à identifier ; le renommage règle le conflit :

```
use my_module, only: add_real => add
```


CC BY SA



`private` par défaut dans les modules

```
module stats
    use working_precision, only: wp
    implicit none
    private                              ! tout est privé...
    public :: mean                       ! ...sauf l'interface publique
contains
    pure function mean(x) result(m)
        real(wp), intent(in) :: x(:)
        real(wp) :: m
        m = sum(x) / size(x)
    end function mean
end module stats
```

- Interface publique réduite et explicite ; détails d'implémentation modifiables sans casser les utilisateurs
- Option gfortran : `-fmodule-private`

CC BY SA



Interfaces explicites pour les bibliothèques anciennes

Une bibliothèque externe (BLAS, LAPACK…) n'est pas un module : on écrit l'interface une fois.

```
module blas_interfaces
    use working_precision, only: wp
    implicit none
    interface
        subroutine daxpy(n, alpha, x, incx, y, incy)
            use working_precision, only: wp
            integer,  intent(in)    :: n, incx, incy
            real(wp), intent(in)    :: alpha, x(*)
            real(wp), intent(inout) :: y(*)
        end subroutine daxpy
    end interface
end module blas_interfaces
```

- Le compilateur vérifie nombre, types et `intent` des arguments ; options `-Wimplicit-interface`, `-Wimplicit-procedure`

CC BY SA



Contrôler la précision : un module pour tout le programme

```
module working_precision
    use, intrinsic :: iso_fortran_env, only: wp => real64
    implicit none
    private
    public :: wp
end module working_precision
```

```
use working_precision, only: wp        ! dans chaque unité
real(kind=wp) :: x
x = 1.0_wp / 3.0_wp
```

- Changer `real64` en `real32` (ou `real128`) adapte tout le programme
- Variante : `integer, parameter :: wp = selected_real_kind(15, 50);` même principe pour les entiers (`int64`)

CC BY SA



Syntaxes obsolètes à proscrire

pas bien

```
double precision x
real*8 y
x = dabs(y) * 1.0d0
```

bien

```
real(kind=real64) :: x, y
x = abs(y) * 1.0_real64
```

Les intrinsèques génériques gèrent automatiquement la précision ; un argument optionnel `kind` permet de choisir celle de la sortie.

type	déclarations	intrinsèques typées
integer	`integer*N`	`iabs`, `idim`, `min0`…
real	`real*N`, `double precision`	`alog`, `amax1`, `dabs`, `dsqrt`…
complex	`double complex`	`cabs`, `cexp`, `dcmplx`…

CC BY SA



Le futur : précision par défaut (norme 2028 ?)

Une proposition pour la prochaine norme : fixer le *kind* par défaut entre les `use` et `implicit none`, pour les déclarations et les littéraux.

```
use iso_fortran_env, only: real64
default kind :: real = real64
implicit none
...
real :: x
x = 1.0    ! littéral en double précision, affecté à un réel double
```

À vérifier

Syntaxe illustrative, susceptible d'évoluer avant la publication de la norme ; pas utilisable aujourd'hui.

CC BY SA



Know your intrinsics !

Le compilateur sait les optimiser, d'autant plus avec le déport sur GPU.

Mathématiques

`hypot`, `atan2`, `norm2`, `log_gamma`, `erf`, `bessel_j0`…

2023 : trigonométrie en degrés (`sind`, `cosd`…) et en multiples de π (`sinpi`, `cospi`…)

Tableaux

`sum`, `maxval`, `minloc`, `findloc`, `pack`, `merge`, `reshape`, `spread`, `matmul`, `cshift`, `reduce`…

Texte

`trim`, `adjustl`, `index`, `scan`, `verify`…

2023 : `split`, `tokenize`

Système

`command_argument_count`, `get_command_argument`, `get_environment_variable`, `execute_command_line`, `system_clock`, `cpu_time`, `random_init`…

CC BY SA



Plan

1 Philosophie

2 Style et pièges

3 Le socle consensuel

4 Attributs : les garde-fous

5 Structures adaptées

6 Les points discutables

7 Le compilateur comme garde-fou

8 Outils et coarrays

9 Conclusion



Attributs de variables

Tout attribut déclaré est une propriété que le compilateur vérifie.

norme	attributs
77	`dimension`, `parameter`, `save`
90	`allocatable`, `intent`, `optional`, `pointer`, `private`, `target`
2003	`volatile`, `protected`
2008	`contiguous`, `codimension`

- `intent(in)`, `intent(out)`, `intent(inout)` sur tous les arguments muets
- `parameter` pour les constantes ; `protected` pour une variable de module lisible mais non modifiable de l'extérieur

CC BY SA



Attributs de procédures

norme	attributs
77	`external`
90	`recursive`
95	`pure` (pas d'effet de bord), `elemental` (aussi `pure`)
2003	`deferred`, `non_overridable`
2008	`impure` (fonctions élémentaires impures)
2018	`non_recursive` (la récursivité devient le défaut)
2023	`simple` (`pure` sans héritage de l'environnement)

- Plus une procédure est contrainte (`pure`, `simple`), plus le compilateur peut l'optimiser et la paralléliser

CC BY SA



Routines vectorielles : l'attribut `elemental`

On écrit la procédure pour un scalaire ; elle s'applique à tout tableau.

```
pure elemental function clamp(x, lo, hi) result(y)
    real(wp), intent(in) :: x, lo, hi
    real(wp) :: y
    y = min(max(x, lo), hi)
end function clamp
```

```
real(wp) :: a(1000), m(100, 100)
...
a = clamp(a, 0.0_wp, 1.0_wp)        ! vecteur
m = clamp(m, -1.0_wp, 1.0_wp)       ! matrice
```

- Aucune boucle à écrire, ni à paralléliser : le compilateur choisit
- Les arguments sont conformes entre eux (scalaires ou tableaux de même forme)

CC BY SA



Plan

1 Philosophie

2 Style et pièges

3 Le socle consensuel

4 Attributs : les garde-fous

5 Structures adaptées

6 Les points discutables

7 Le compilateur comme garde-fou

8 Outils et coarrays

9 Conclusion



Alternatives : `if` et `select`

`if`

- en ligne : `if (x < 0.0) x = 0.0`
- bloc `if` / `else if` / `else`
- expression conditionnelle (2023) :

```
b = (a > 0.0 ? a : 0.0)
```

Support compilateur encore partiel : refusée par gfortran 13.

`select`

- `select case` : remplace une chaîne de `if (. == .)` / `else if`
- `select type` (2003) : types dynamiques
- `select rank` (2018) : rang des tableaux à rang supposé

CC BY SA



select rank (2018)

```
subroutine describe(a)
    real(wp), intent(in) :: a(..)        ! rang supposé
    select rank (a)
    rank (0)
        print *, "scalaire"
    rank (1)
        print *, "vecteur de taille", size(a)
    rank default
        print *, "rang", rank(a)
    end select
end subroutine describe
```

- Une seule procédure pour scalaire, vecteur, matrice... : très utile pour l'interopérabilité avec C

CC BY SA



Boucles do

- Tableaux stockés par colonnes : le premier indice varie le plus vite (inverse du C), donc boucle la plus interne

```
do j = 1, n
    do i = 1, m                ! i (premier indice) à l'intérieur
        a(i, j) = b(i, j) + c(i, j)
    end do
end do
```

- do while (condition) ; do sans borne avec exit / cycle
- *Implied do* pour construire des tableaux :

```
integer, parameter :: n = 10
integer :: i, values(n) = [ (i*2, i = 1, n) ]
```


CC BY SA



do concurrent

```
do concurrent (i = 1:n, j = 1:m)
    a(i, j) = b(i, j) + c(i, j)
end do
```

- Promesse faite au compilateur : les itérations sont indépendantes, dans un ordre quelconque
- Boucles imbriquées : un seul `do concurrent` avec plusieurs indices
- Clauses de localité (2018) : `local`, `local_init`, `shared`, `default(none)` ; réduction (2023) : `reduce(+: s)`
- Limites : pas de `exit`, pas de `return` ; appels de procédures `pure` uniquement avant 2018

Support inégal

gfortran 13 refuse les clauses `local` et `reduce` ; à vérifier selon la version de votre compilateur.

CC BY SA



where et block

```
where (a < 0.0_wp)
    a = 0.0_wp
end where
```

- where : affectation masquée de tableaux ; instruction ou bloc
- forall est obsolescent depuis 2018 : lui préférer do concurrent ou where

```
block                ! variables locales au plus près de leur usage
    real(wp) :: tmp(size(a))
    tmp = a
end block
```


CC BY SA



allocatable avant pointer

- allocatable : libération automatique en sortie de portée, pas d'alias, meilleures optimisations
- pointer / target : seulement quand l'alias ou la structure l'exige (listes chaînées, arbres)
- Affectation avec (ré)allocation automatique depuis 2003
- move_alloc : transférer l'allocation sans copie

```
real(wp), allocatable :: v(:), tmp(:)
allocate(tmp(2*size(v)))
tmp(:size(v)) = v
tmp(size(v)+1:) = 0.0_wp
call move_alloc(from=tmp, to=v)    ! v agrandi ; tmp libéré
```


CC BY SA



Constructeurs de tableaux et de types dérivés

```
real(wp) :: v(3) = [real(wp) :: 1, 2.5, 3]       ! type explicite
integer  :: squares(10) = [ (i**2, i = 1, 10) ] ! implied do

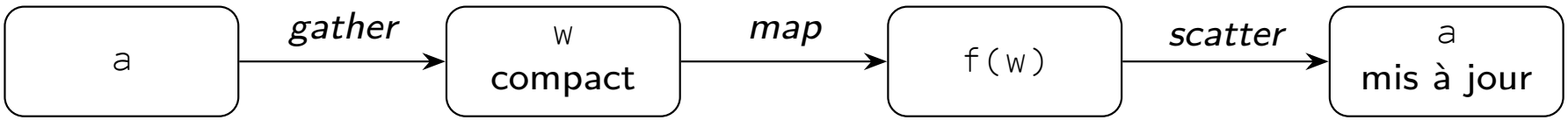
type :: point
    real(wp) :: x, y
end type point
type(point) :: p = point(1.0_wp, 2.0_wp)          ! constructeur de structure
```

- Un constructeur construit l'objet en une seule instruction
- Préciser le type du constructeur de tableau (`[real(wp) :: ...]`) évite les conversions implicites

CC BY SA



La trinité fonctionnelle : *gather* – *map* – *scatter*



étape	rôle	en Fortran moderne
gather	lire selon des indices ou un masque	`a(idx)`, `pack`, `merge`
map	appliquer une fonction à chaque élément	`elemental`, expression tableau
scatter	réécrire aux positions d'origine	`a(idx) = ...`, `unpack`, `where`

- Compagnon du *gather* : la réduction (`sum`, `maxval`, `reduce`)
- On décrit *quoi* faire sur quelles données ; le compilateur choisit la vectorisation ou le déport GPU



Gather : par indices ou par masque

```
integer  :: idx(3) = [4, 1, 7]
real(wp) :: a(8), b(3)
real(wp), allocatable :: c(:)

b = a(idx)                          ! indices vectoriels : b = [a(4), a(1),
 ↪  a(7)]
c = pack(a, mask = a > 0.0_wp)      ! masque : éléments positifs, compactés
n = count(a > 0.0_wp)               ! combien ?
r = merge(a, 0.0_wp, a > 0.0_wp)    ! sans compacter : a ou 0 selon le masque
```

- Les indices peuvent se répéter et venir dans n'importe quel ordre
- `c` est (ré)alloué automatiquement à la taille `count(mask)`
- Indices d'un masque : `pack([(i, i = 1, size(a))], mask)` ou `findloc`

CC BY SA



Map : une procédure `elemental`, sans effet de bord

```
pure elemental function f(x) result(y)
    real(wp), intent(in) :: x
    real(wp) :: y
    y = sqrt(x) / (1.0_wp + x)
end function f
```

```
w = f(w)                              ! expression tableau
do concurrent (i = 1:size(w))         ! équivalent explicite
    w(i) = f(w(i))
end do
```

- `pure` : chaque élément est calculé indépendamment, dans un ordre quelconque, donc parallélisable
- Composable : `sum(f(pack(a, a > 0.0_wp)))` enchaîne *gather*, *map* et réduction sans boucle

CC BY SA



Scatter : écrire aux bons endroits

```
a(idx) = w                                          ! indices vectoriels
a = unpack(w, mask = a > 0.0_wp, field = a)         ! inverse de pack
where (a > 0.0_wp) a = w2                           ! affectation masquée
```

Attention aux indices répétés

Un indice vectoriel à gauche d'une affectation ne doit pas contenir deux fois la même valeur (la norme l'interdit) : le compilateur ne le vérifie pas toujours. Même règle avec `do concurrent` : deux itérations qui écrivent au même endroit rompent la promesse d'indépendance.

- À droite (*gather*), les répétitions sont permises
- `unpack` : `field` donne la valeur des positions non masquées

CC BY SA



Assembler : *gather* → *map* → *scatter*

```
integer,  allocatable :: k(:)
real(wp), allocatable :: w(:)

k = pack([(i, i = 1, size(a))], a > 0.0_wp)   ! indices à traiter
w = a(k)                                      ! gather
w = f(w)                                      ! map
a(k) = w                                      ! scatter
```

Cas simple, même résultat :

```
where (a > 0.0_wp) a = f(a)
```

- `where` suffit quand le masque est évident ; la version explicite sert si les indices `k` se réutilisent, si le traitement est coûteux sur peu d'éléments, ou si l'on travaille sur des données creuses

CC BY SA



Exemple : produit matrice creuse – vecteur (CSR)

```
! val : valeurs ; ja : colonnes ; ia(i):ia(i+1)-1 : éléments de la ligne i
do i = 1, nrows
    y(i) = sum( val(ia(i):ia(i+1)-1) &          ! map : produit
              * x(ja(ia(i):ia(i+1)-1)) )       ! gather de x, puis réduction
end do
```

- `x(ja(...))` : *gather* par indices ; `val(...) * x(...)` : *map* ; `sum` : réduction
- Les lignes sont indépendantes : boucle externe candidate à `do concurrent`
- Accès indirects : limités par la mémoire, pas par le calcul
- `-fcheck=bounds` détecte les indices `ja` hors limites

CC BY SA



Plan

1 Philosophie

2 Style et pièges

3 Le socle consensuel

4 Attributs : les garde-fous

5 Structures adaptées

6 Les points discutables

7 Le compilateur comme garde-fou

8 Outils et coarrays

9 Conclusion



Expression tableau *vs* do concurrent *vs* boucles

```
y = y + alpha * x                    ! expression tableau

do concurrent (i = 1:n)              ! do concurrent
    y(i) = y(i) + alpha * x(i)
end do

!$omp parallel do                    ! boucle + OpenMP
do i = 1, n
    y(i) = y(i) + alpha * x(i)
end do
```

- Point litigieux : en HPC, les performances s'obtiennent souvent avec des boucles écrites à la main (on nous pousse à écrire du C)
- Position défendue : privilégier la syntaxe vectorielle, faire pression sur les constructeurs ; ça s'améliore doucement

CC BY SA



Constructeurs

Notion issue de l'objet :

```
type :: point
    private
    real(wp) :: x_ = 0, y_ = 0
contains
    procedure :: x, y                          ! accesseurs (getters)
end type point

interface point                                ! constructeur générique
    module procedure new_point
end interface

c = new_point(a%x() + b%x(), a%y() + b%y())    ! nouvel objet
```

- …on construit l'objet en une instruction, puis on le lit (*getters*) ou on s'en sert pour en construire d'autres :
  si l'on s'interdit les *setters*… *pipeline,* approche fonctionnelle
- Avantages : invariants garantis, `pure` ; coût : copies

CC BY SA



Objet + fonctionnel : pipeline de constructeurs

```
  integration: do concurrent (I=1:num_events)
     ..
     if (o1Event%cut_on_energy (oCutpar)) cycle integration
     num_passfirstcut_events = num_passfirstcut_events + 1
     call o1Event%sort   !> Sort outgoing photons by energy
     o1Spinor = spinor_typ (o1Event)
     o1Scalar = scalar_typ (o1Spinor)
     o1Angle  = angle_typ (o1Event, o1Scalar)
!>  Compute event total weight including Matrix element
!>  (with cuts if needed)
     if (o1Angle%cut_on_angle (oCutpar)) cycle integration
     o1Result = result_typ (o1Spinor, oParam)
     ..
```

- constructeurs purs

CC BY SA



Submodules (2008)

- répartir un module en plusieurs fichiers, en conservant `private`
- limiter la recompilation : modifier l'implémentation ne recompile que le *submodule*, pas le module et ses utilisateurs

```
module points
    type :: point; real :: x, y; end type point
    interface   ! interface dans le module
        module function point_dist(a, b) result(distance)
            type(point), intent(in) :: a, b
            real :: distance
        end function point_dist
    end interface
end module points

submodule (points) points_impl   ! implémentation
contains
    module procedure point_dist
        distance = hypot(a%x - b%x, a%y - b%y)
    end procedure point_dist
end submodule points_impl
```


CC BY SA



Thin binding et *thick binding*

Deux façons d'appeler une bibliothèque externe (BLAS, LAPACK, C…)

Thin binding

- interface au plus près de la routine : mêmes noms, mêmes arguments
- aucun code en plus
- l'appelant gère dimensions, `lda`…

```
call daxpy(n, alpha, x, 1, y, 1)
```

Thick binding

- enveloppe Fortran : dimensions déduites des tableaux, arguments `optional`
- surcharge générique par *kind*
- un peu de code et un appel supplémentaire

```
call axpy(x, y, alpha)
```


CC BY SA



Thick binding : `optional` et surcharge générique

```
interface axpy                          ! même nom pour tous les kinds
    module procedure saxpy_f, daxpy_f
end interface
...
subroutine daxpy_f(x, y, alpha)
    real(real64), intent(in)    :: x(:)
    real(real64), intent(inout) :: y(:)
    real(real64), intent(in), optional :: alpha
    real(real64) :: a
    a = 1.0_real64
    if (present(alpha)) a = alpha
    call daxpy(size(x), a, x, 1, y, 1)   ! n déduit de size(x)
end subroutine daxpy_f
```

- Résolution de la surcharge à la compilation : pas de coût à l'exécution
- Exemples classiques : LAPACK95, `stdlib`

CC BY SA



Plan

1 Philosophie

2 Style et pièges

3 Le socle consensuel

4 Attributs : les garde-fous

5 Structures adaptées

6 Les points discutables

7 Le compilateur comme garde-fou

8 Outils et coarrays

9 Conclusion



Analyse statique systématique (gfortran)

```
gfortran -std=f2018 -Wall -Wextra -Wpedantic -Wconversion -Werror ...
```

- `-Wall` inclut notamment : `-Waliasing`, `-Wampersand`, `-Wconversion`, `-Wsurprising`, `-Wc-binding-type`, `-Wintrinsics-std`, `-Wtabs`, `-Wintrinsic-shadow`, `-Wline-truncation`, `-Wtarget-lifetime`, `-Winteger-division`, `-Wreal-q-constant`, `-Wunused`, `-Wundefined-do-loop`
- `-Wextra` ajoute : `-Wcompare-reals`, `-Wunused-parameter`, `-Wdo-subscript`
- `-std=f95|f2003|f2008|f2018` : refuser les extensions (gfortran 13 n'accepte pas encore `-std=f2023`)
- `-Werror` : transforme les avertissements en erreurs ; excellent en intégration continue, plus discutable chez l'utilisateur (un nouveau compilateur casse la compilation)

CC BY SA



Options héritées de C, sans effet en Fortran

Testé avec gfortran 13.3 :

- `-Wswitch`, `-Wswitch-enum` : refusées (C/C++ seulement)
- `-Wshadow` : acceptée mais aucun avertissement sur une variable locale qui masque celle de l'hôte
- `-fanalyzer` : acceptée mais sans effet sur du Fortran

En pratique

Ne pas compter sur elles ; préférer `-Wall -Wextra`, les contrôles dynamiques et un linter dédié (`fortitude`).

`-fanalyzer` (GCC) : analyse statique des chemins interprocéduraux, *« outil de recherche de bogues, non de preuve de correction »*, très coûteuse. Elle s'appuie sur les référentiels CWE (*Common Weakness Enumeration*) et CVE (*Common Vulnerabilities and Exposures*).

CC BY SA



Analyse dynamique, au moins en mode debug

```
gfortran -g -Og -fcheck=all -fbacktrace -finit-real=snan \
         -ffpe-trap=invalid,zero,overflow ...
```

- `-g -Og` : informations de débogage, optimisations compatibles
- `-fcheck=all` : `array-temps`, `bits`, `bounds`, `do`, `mem`, `pointer`, `recursion`
- `-fbacktrace` : pile d'appels en cas d'erreur
- `-finit-real=snan` : les réels non initialisés deviennent des NaN signalants ; compléments `-finit-integer`, `-finit-derived`
- `-ffpe-trap=`*`liste`* : interrompre sur `invalid` (`sqrt(-1.0)`), `zero`, `overflow` ; aussi `underflow`, `inexact`, `denormal`
- `-ffpe-summary=`*`liste`* : état des exceptions affiché au `stop` (défaut : toutes sauf `inexact`)

CC BY SA



Sanitizers

- `-fsanitize=undefined` : détecteur rapide de comportements indéfinis
- `-fsanitize=address` : erreurs mémoire
- `-fsanitize=leak` : fuites mémoire
- `-fsanitize=thread` : *data races* (exclusif avec `address`)
- Aussi : `unreachable`, `pointer-compare`, `pointer-subtract`, `shadow-call-stack` (protection contre l'écrasement d'adresses de retour)

À retenir

Un *build* de développement avec toutes ces vérifications, un *build* de production sans : les mêmes tests doivent passer dans les deux.

CC BY SA



Autres compilateurs : équivalences

fonction	gfortran	ifx	nvfortran
`implicit none` **global**	`-fimplicit-none`	`-implicitnone`	`-Mdclchk`
avertissements	`-Wall -Wextra`	`-warn all`	`-Minform=warn`
bornes, etc.	`-fcheck=all`	`-check all`	`-C`
pile d'appels	`-fbacktrace`	`-traceback`	`-traceback`
NaN signalants	`-finit-real=snan`	`-init=snan`	—
pièges FPE	`-ffpe-trap=...`	`-fpe0`	`-Ktrap=fp`
standard strict	`-std=f2018`	`-stand f18`	`-Mstandard`

À vérifier selon la version du compilateur et la documentation du constructeur.

CC BY SA



Plan

1 Philosophie

2 Style et pièges

3 Le socle consensuel

4 Attributs : les garde-fous

5 Structures adaptées

6 Les points discutables

7 Le compilateur comme garde-fou

8 Outils et coarrays

9 Conclusion



Outils modernes

- fortitude : linter / analyseur statique pour Fortran
- camfort : linter / analyseur statique pour Fortran
- fpm : *Fortran Package Manager* (construction, tests, dépendances)
- stdlib : bibliothèque standard communautaire (disponible via fpm)
- pFUnit : tests unitaires

```
fpm build --profile debug
fpm test
fortitude check src/
```

```
# fpm.toml : dépendances
[dependencies]
stdlib = "*"
```


CC BY SA



Coarray Fortran (CAF)

```
real(wp) :: a(100)[*]                    ! tableau présent dans chaque image
integer  :: me
me = this_image()
a = real(me, wp)
sync all
if (me == 1) print *, a(1)[num_images()]   ! lecture à distance
```

- Parallélisme natif du langage ; en pratique, une enveloppe autour d'une partie de MPI
- Questions ouvertes pour la discussion :
  - la gestion de la topologie (communicateurs, grilles) ?
  - et les entrées-sorties parallèles (MPI-IO) ?

CC BY SA



Plan

1 Philosophie

2 Style et pièges

3 Le socle consensuel

4 Attributs : les garde-fous

5 Structures adaptées

6 Les points discutables

7 Le compilateur comme garde-fou

8 Outils et coarrays

9 Conclusion



Checklist minimale

1. `implicit none (type, external)` partout
2. `private` par défaut, `use, only:` systématique
3. un module de précision (`wp`), des littéraux `_wp`, plus de `double precision`
4. `intent`, `pure`, `elemental` dès que possible
5. intrinsèques et syntaxe vectorielle avant les boucles à la main
6. `allocatable` avant `pointer`
7. `-Wall -Wextra -Wpedantic` en continu ; `-fcheck=all -finit-real=snan` en debug
8. un linter (`fortitude`), `fpm`, des tests unitaires

CC BY SA

Questions et discussion

CC BY SA