Université Pierre et Marie Curie
Paris VI

Master de Sciences et Technologies :
– Mention Physique et Applications
– Mention Sciences de l'Univers, Environnement, Écologie

Méthodes numériques et informatiques : programmation

Introduction au fortran 90/95/2003/2008

Avertissement

Ce document est diffusé dans une version en évolution, qui reste encore très inachevée, notamment concernant les apports des normes 2003 et 2008 du fortran. Plusieurs chapitres ne sont que des ébauches, en particulier ceux sur les structures, les pointeurs et l'interopérabilité avec le C ; d'autres questions sont à peine abordées. L'objectif reste de compléter progressivement le document.
Dans ce contexte, je vous remercie par avance de me signaler les erreurs aussi bien que les formulations obscures. Vos commentaires et suggestions seront les bienvenus.

Jacques.Lefrere@upmc.fr

27 novembre 2017

Remerciements

Je tiens à remercier mes collègues enseignants de la Maîtrise de Physique et Applications de l'Université Pierre et Marie Curie, Paris VI, Marie-Claude et Jean-Claude Justice, Michel Karatchenzeff, Maï Pham, ainsi que mes collègues du M1 du Master de l'UPMC, Albert Hertzog, Philippe Sindzingre et François Ravetta, pour leurs encouragements et leurs conseils au cours de la rédaction de ce document.

Je remercie aussi Marie-Alice Foujols de l'Institut Pierre Simon Laplace pour ses séminaires enthousiastes qui nous ont sensibilisés au fortran 90.

Merci aussi aux collègues du Service d'aéronomie pour les nombreuses discussions au sujet du fortran au sein du laboratoire, et en particulier à Françoise Pinsard, Francis Dalaudier et Stéphane Marchand pour leurs patientes relectures critiques.

Je remercie aussi Frédéric Bernardo sans lequel le chapitre sur l'inter-opérabilité avec le langage C n'aurait pas vu le jour.

Merci enfin à tous ceux qui ont contribué de près ou de loin à l'élaboration de ce document, et surtout aux étudiants de la Maîtrise de Physique et Applications de Paris VI, puis à ceux du M1 du Master (mentions P&A et SDUEE) de l'UPMC, car c'est avant tout pour eux mais aussi grâce à eux qu'il a été rédigé.

Version 3242
Ce document a été mis en page grâce au traitement de texte LaTeX.

Notations

- La police « machine à écrire », à espacement fixe, est utilisée pour indiquer les éléments du code source du fortran.
- Les crochets `[...]` délimitent les éléments optionnels de la syntaxe ou les arguments optionnels des procédures ; ces symboles ne font pas partie de la syntaxe.
- Lors de la description de la syntaxe, les symboles < et > indiquent où l'utilisateur doit substituer les identificateurs, expressions, ou valeurs qu'il a choisis ; ces symboles ne font pas partie de la syntaxe.

Exemple

La syntaxe décrite comme suit :

```
[<nom> :] IF (<expression logique>) THEN
  <bloc d'instructions>
END IF [<nom>]
```

peut être utilisée comme dans l'exemple suivant, où l'identificateur `chiffre` est optionnel :

```
chiffre: IF (i < 10) THEN
  WRITE(*,*) 'i est inférieur à 10'
END IF chiffre
```

Indications de lecture

♠ 0.0.0 Sous-section facultative

Les sections (ou sous-sections) qui peuvent être omises en première lecture sont indiquées par le symbole ♠ placé à gauche du titre de la section comme ci-dessus.

f2003 0.0.0 Sous-section nouveauté f2003

Les sections (ou sous-sections) qui font intervenir des nouveautés des standards 95, 2003 ou 2008 du fortran sont indiquées par le symbole f95, f2003 ou f2008 placé à gauche du titre de la section comme ci-dessus.

S'il s'agit de quelques points spécifiques d'un standard, par exemple le standard 2008, le paragraphe est identifié par le symbole f2008 placé dans la marge extérieure du texte, comme pour ce paragraphe. ⇐ f2008

Conseils pratiques

Les règles de « bon usage » du langage, qui, au delà de la norme, sont motivées par des objectifs de lisibilité, de portabilité ou de robustesse du code source, sont repérées par le symbole ♥ dans la marge extérieure du texte, comme pour ce paragraphe. ⇐ ♥

Difficultés

Les points présentant des difficultés particulières ou des risques d'erreur sont indiqués par le symbole ⚠ dans la marge extérieure du texte, comme pour ce paragraphe. ⇐⚠

Avertissement

Ce document n'est pas un manuel de référence du fortran 90/95/2003/2008 décrivant exhaustivement la norme du langage. Il s'agit d'un document largement inspiré des références citées dans la bibliographie, références auxquelles il est vivement conseillé de se reporter pour une étude plus approfondie ou plus systématique. Incomplet, notamment sur certains aspects les plus récents du fortran, il s'apparente plus à un guide de l'utilisateur du fortran pour le calcul scientifique. Il s'adresse essentiellement à des étudiants ayant déjà une expérience de la programmation dans un langage structuré (fortran ou C) souhaitant s'initier aux nouvelles fonctionnalités du langage à partir du fortran 90, notamment :

- la notation tableaux, les opérateurs et fonctions manipulant globalement des tableaux multidimensionnels,
- l'allocation dynamique, les types dérivés et les pointeurs,
- le contrôle des passages d'arguments entre procédures via les modules.

Présentation

Après un chapitre 1 introductif (p. 1) présentant l'évolution du fortran, les étapes de mise au point des programmes et les éléments du langage, le chapitre 2 (p. 10) décrit les types d'objets du langage et leurs attributs, en particulier numériques, avant d'aborder au chapitre 3 (p. 22) les opérateurs qui permettent de les manipuler.

Le chapitre 4 (p. 27) est entièrement consacré aux structures de contrôle du fortran : c'est un chapitre essentiel et facile qui doit être étudié dès la première lecture.

Le chapitre 5 entrées-sorties (p. 36) aborde les instructions de lecture et d'écriture et les formats de conversion des données : il est conseillé de commencer par les sections d'introduction et de terminologie, quitte à revenir ensuite sur la description détaillée des instructions et surtout des formats, pour laquelle des allers et retours avec les sections d'exemples ainsi qu'avec le chapitre 8 (p. 96) sur les chaînes de caractères sont nécessaires.

Le chapitre 6 intitulé procédures (p. 58) constitue le deuxième thème majeur du document : il présente les fonctions et les sous-programmes en partant de la notion de procédure interne attachée à un programme pour aboutir aux procédures de module réutilisables grâce à la compilation séparée. La dernière section (6.5) abordant les fonctionnalités avancées peut être passée en première lecture.

Le troisième thème majeur du fortran, les tableaux, fait l'objet du chapitre 7 (p. 80) : il constitue l'atout principal du fortran 90 pour le calcul scientifique avec l'introduction des notations matricielles et des fonctions intrinsèques manipulant les tableaux multidimensionnels. Ce chapitre essentiel peut être abordé en deux étapes : la première lecture doit inclure l'allocation dynamique ainsi que le lien entre procédures et tableaux, mais on peut garder pour une deuxième lecture les notions de sections non-régulières, les masques et la structure `FORALL`.

Le chapitre 8 (p. 96) présente la manipulation des chaînes de caractères et les fonctions associées : c'est un chapitre facile, même s'il n'est pas central pour les applications au calcul scientifique.

Les chapitres 9 sur les types dérivés (p. 104), 10 sur la généricité (p. 113) et 11 sur les pointeurs (p. 120) présentent des fonctionnalités orientées objet du fortran, et peuvent être réservés à une deuxième lecture. Quant au chapitre 12 (p. 131), qui présente les techniques d'appel de fonctions écrites en C depuis le fortran et réciproquement, il peut intéresser des non-spécialistes du fortran pour l'interfacer avec le langage C.

Enfin, les annexes ont plus vocation à servir de référence qu'à être lues de façon linéaire, même s'il est utile de les parcourir pour connaître leur existence. Cela vaut par exemple pour la liste des fonctions intrinsèques du fortran (A, p. 140) quitte à revenir au document ponctuellement pour vérifier une syntaxe d'appel. De la même façon, parcourir l'annexe C, p. 156 sur la norme IEEE de représentation des nombres réels permet de visualiser concrètement les approximations des calculs en machine. Par ailleurs, les utilisateurs du langage C pourront trouver dans l'annexe D, p. 162, un aide-mémoire présentant une comparaison approximative des syntaxes et fonctions des deux langages. Une correspondance entre les fonctions de `numpy` sous `python` et les outils pour les tableaux en fortran est esquissée en annexe E, p. 167. Pour terminer, l'annexe F, p. 168 fournit de brèves indications sur l'utilisation de quelques compilateurs.

Table des matières

Chapitre 1

Introduction

1.1 Historique du fortran

Le fortran (**FOR**MULA **TRAN**SLATION) est le premier langage informatique de haut niveau. Né à la fin des années 1950 sous l'impulsion de John Backus [1], il a été standardisé en 1972 sous la forme du fortran 66 et son efficacité dans le calcul scientifique en a fait le langage le plus utilisé dans les applications non commerciales. La mise à jour du standard à la fin des années 1970 a apporté d'énormes améliorations en particulier dans le traitement des chaînes de caractères avec le fortran 77.

Mais c'est avec fortran 90, dont la norme fut particulièrement longue à négocier, qu'est intervenue une véritable modernisation du langage fortran. Cette nouvelle version a permis un nettoyage des éléments les plus obsolètes du fortran (format fixe par exemple, lié à l'utilisation des cartes perforées). Elle a aussi introduit des fonctionnalités nouvelles parfois présentes dans des langages plus récents, parmi lesquelles nous ne soulignerons que les plus attendues dans notre domaine d'applications :

- langage de programmation structuré ;
- outils de manipulation des tableaux (multidimensionnels) puissants, concis et adaptés au calcul vectoriel et parallèle ;
- gestion dynamique, pointeurs ;
- création de types dérivés (structures), surcharge d'opérateurs, généricité, ...
- fiabilisation des passages d'arguments entre procédures.

Enfin, l'évolution du langage fortran a continué avec le fortran 95, qui constitue une révision mineure, mais surtout le fortran 2003, dont le standard a été publié en novembre 2004 [2]. Il apporte notamment :

- une interopérabilité normalisée avec le langage C ;
- de nouvelles possibilités concernant les tableaux dynamiques et les types dérivés ;
- des fonctionnalités de programmation orientée objet, notamment l'extension des types dérivés et l'héritage, la notion de procédure attachée (bound) à un type ;
- une meilleure intégration dans le système d'exploitation.

Une partie de ces fonctionnalités étaient déjà intégrées dans certains compilateurs fortran 95 sous forme d'extensions, mais la publication de la norme a accéléré leur intégration [3] et garantit leur portabilité. Dans ce contexte, nous ne documenterons pas en général les aspects qualifiés d'obsolescents en fortran 90/95, donc destinés à disparaître des prochaines versions.

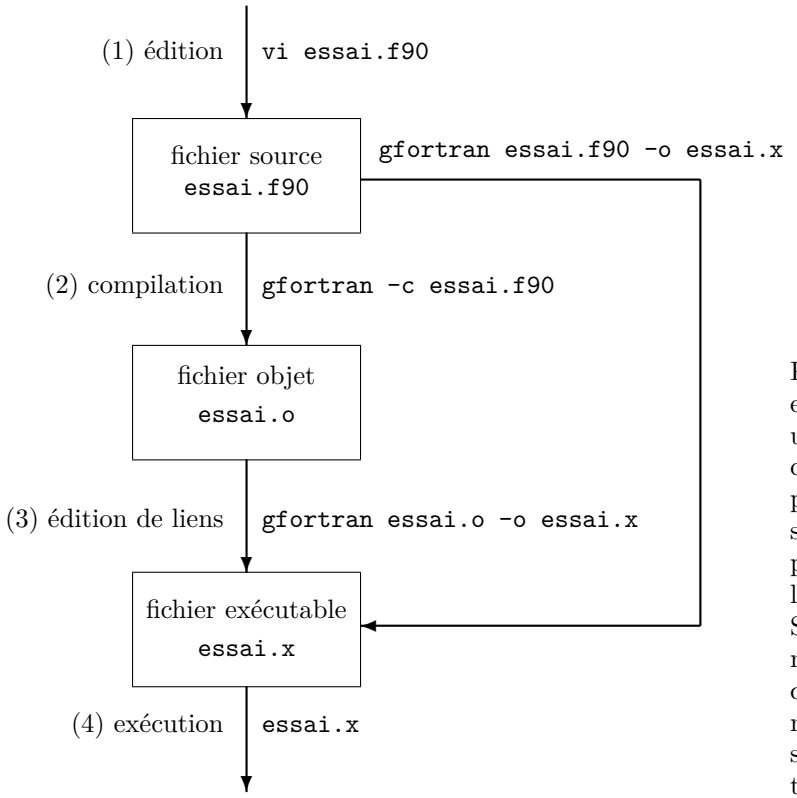
La nouvelle norme fortran 2008 constitue une évolution mineure [4] par rapport au fortran 2003 avec notamment la notion de sous-module, des outils de programmation parallèle (en particulier les

1. John Backus, mort en 2007, a reçu le prix Turing en 1977, notamment pour ses travaux sur Fortran.
2. Consulter https://wg5-fortran.org/f2003.html.
3. Voir la disponibilité sur les différents compilateurs : http://fortranwiki.org/fortran/show/Fortran+2003+status
4. Voir la présentation des apports de fortran 2008 ftp://ftp.nag.co.uk/sc22wg5/N1851-N1900/N1891.pdf par John Reid.

tableaux distribués, ou coarrays [5]), la notion de bloc avec ses variables locales, de nouvelles fonctions mathématiques intrinsèques... Le document final du standard fortran 2008 (Metcalf *et al.* (2011)) a été approuvé [6]. Les discussions sur le prochain standard (initialement nommé fortran 2015, puis renommé fortran 2018) sont en cours [7] pour une publication en août 2018.

1.2 Les étapes de mise en œuvre d'un programme

Outre l'analyse de l'algorithme, la mise en œuvre d'un programme en fortran comporte essentiellement quatre phases, schématisées ci-contre dans le cas d'un seul fichier source [8] :



1. rédaction des unités de programme source à l'aide d'un éditeur de texte
2. compilation (proprement dite) des fichiers source pour produire des fichiers objets
3. lien entre les objets et les bibliothèques produisant un exécutable (grâce à l'éditeur de liens `ld`, lui-même appelé par le compilateur)
4. exécution

Par défaut, les phases de compilation (2) et d'édition de liens (3) sont initiées par un seul appel au compilateur et le fichier objet (créé dans un répertoire temporaire) n'est pas conservé. Il est nécessaire de passer l'option `-c` au compilateur pour ne pas enchaîner automatiquement l'édition de liens.
Sans cette option, le compilateur détermine la nature du fichier (source ou objet) qui lui est fourni au vu du suffixe de son nom et lance la compilation des fichiers sources éventuels (suffixe `.f90`) puis l'édition de liens des objets (suffixe `.o`).

Seul le fichier source est portable d'une machine à une autre, à condition qu'il respecte un standard du langage. Le fichier objet est une traduction en langage machine de bas niveau des instructions du fichier source. Pour constituer le fichier exécutable, il faut résoudre les appels à des fonctions intrinsèques du langage et éventuellement à d'autres bibliothèques : c'est le rôle de l'éditeur de liens, `ld`, d'aller extraire des bibliothèques les codes objets des procédures appelées pour les agréger au fichier objet compilé, de façon à constituer le fichier exécutable, en principe autonome. Les fichiers objets et le fichier exécutable sont des fichiers binaires spécifiques de la machine et du compilateur utilisé.

Les erreurs...

Si certaines erreurs peuvent être détectées lors de la compilation (essentiellement les erreurs de syntaxe), voire lors de l'édition de liens, d'autres peuvent ne se manifester que dans la phase

5. Les coarrays sont déjà implémentés sous le compilateur `g95`.
6. La dernière version en discussion du standard 2008 http://j3-fortran.org/doc/year/10/10-007r1.pdf date de novembre 2010.
7. Voir le projet de norme fortran 2018 (ex-2015) publié en juillet 2017 : https://wg5-fortran.org/f2018.html.
8. Le cas plus réaliste de plusieurs fichiers sources est abordé en 1.2.4, p. 5 et détaillé dans le contexte des modules en 6.4, p 65 avec notamment la figure 6.1, p. 66.

d'exécution (run-time errors) : elles devront être corrigées en reprenant le processus à la phase (1), l'édition des programmes source.

N.-B. : il est vivement conseillé de corriger les erreurs diagnostiquées par le compilateur dans l'ordre d'apparition, car une première erreur [9] peut en induire une quantité d'autres [10] dans la suite du programme (par exemple quand elle porte sur une déclaration de variable). ⇐ ♥

1.2.1 Édition du fichier source

N'importe quel éditeur de texte peut permettre la saisie des programmes sources, qui sont des fichiers texte. Mais certaines fonctions de l'éditeur seront particulièrement utiles pour éditer un langage structuré comme le fortran. Par exemple, sous l'éditeur `vi`, la cohérence des parenthèses éventuellement emboîtées peut être vérifiée par la simple frappe de la touche `%` qui, si le curseur est placé sur un délimiteur ouvrant, le déplace sur le délimiteur fermant associé, et réciproquement.

Certains éditeurs de texte sont dits « sensibles au langage », dont la syntaxe est décrite par l'intermédiaire de fichiers spécifiques. Parmi ces éditeurs « intelligents », citons `emacs` et `xemacs`, `gedit`, `kedit` et la version `vim` [11] (**vi** i**m**proved) de l'éditeur `vi` disponible sous LINUX.

Ils sont alors capables de repérer les commentaires, les chaînes de caractères, de reconnaître les mots-clefs du langage, ses structures de contrôle, ses fonctions intrinsèques, et d'autoriser leur saisie abrégée. La structure syntaxique du fichier source peut alors être mise en évidence par l'usage de couleurs spécifiques, ce qui constitue une aide précieuse pour contrôler la syntaxe dès la phase d'édition.

Les éditeurs `emacs` et `xemacs` possèdent un *mode* fortran 90 [12] activé automatiquement au vu du suffixe (extension) du fichier édité. Dans ce mode, il est possible de lancer une compilation depuis l'éditeur, d'indenter automatiquement les blocs des structures de contrôle... La présentation syntaxique du fichier source est elle-même configurable : par exemple, le degré de marquage syntaxique est ajustable sur quatre niveaux et l'affichage des mots-clefs peut être basculé de minuscules en majuscules, ou en minuscules avec l'initiale en capitale.

Un fichier source élémentaire ne comportant qu'un programme principal débute par l'instruction `PROGRAM`, suivie du nom du programme et se termine par `END PROGRAM`, suivi du même nom. Par exemple, le programme suivant nommé `bienvenue` permet d'afficher le message `Bonjour`.

```
PROGRAM bienvenue
  WRITE(*,*) "Bonjour"
END PROGRAM bienvenue
```

1.2.2 Compilation et édition de liens

De nombreux compilateurs fortran sont disponibles, propriétaires ou libres : on pourra consulter à ce propos l'annexe F, p 168. Dans ce document, nous décrirons essentiellement les possibilités des deux compilateurs libres `g95` (*cf.* F.6, p. 175) et surtout `gfortran` (*cf.* F.5, p. 172) avec lesquels les codes proposés sont testés.

La syntaxe des commandes de compilation est spécifique de chaque compilateur, mais, au moins sous unix, certaines options importantes respectent une syntaxe générale :

- l'option `-c` qui permet de créer un fichier objet sans lancer l'édition de liens.
- l'option `-o` suivie d'un nom de fichier permet de spécifier le nom du fichier produit par le compilateur [13], que ce soit un fichier objet (dans le cas où l'on utilise aussi l'option `-c`) ou un fichier exécutable si l'édition de liens est lancée.

9. Certains compilateurs possèdent une option (`-fone-error` pour `g95`, *cf.* F.6.1, `-fmax-errors=1` pour `gfortran` *cf.* F.5.2, `-diag-error-limit1` pour `ifort`, *cf.* F.4.1) permettant d'arrêter la compilation dès la première erreur.

10. Sous UNIX, les messages d'erreur sont envoyés sur la sortie d'erreur standard. C'est donc celle-ci qu'il faut rediriger pour maîtriser le défilement des messages soit dans un fichier par `2> fichier_erreurs`, soit dans un tube après fusion avec la sortie standard par `2>&1 | less` par exemple.

11. Pour `vim`, la syntaxe du fortran 90 est décrite dans le fichier `/usr/share/vim/syntax/fortran.vim`.

12. Le mode fortran de `emacs` et `xemacs` est décrit dans le fichier lisp `f90.el`, dont la version compilée est `f90.elc`.

13. En l'absence de l'option `-o`, le fichier objet est nommé en substituant le suffixe `.o` au suffixe du fichier source et le fichier exécutable est nommé `a.out`

– l'option `-l` suivie du nom (dépouillé de `lib` en tête et de `.a` en fin) d'une bibliothèque à ⚠⇒ laquelle on fait appel lors de l'édition de liens. Cette option doit être placée *après* les fichiers qui appellent des procédures de la bibliothèque. Pour utiliser par exemple la bibliothèque `libnrecip.a` avec `gfortran`, on lancera [14] :

```
gfortran  <essai>.f90  -lnrecip
```

– les options de type `-O<n>`, où `<n>` est un entier, contrôlent l'optimisation du code. Plus `<n>` est élevé, plus le code peut s'exécuter rapidement, mais les optimisations fortes ne sont pas forcément sûres. Il est conseillé de commencer la mise au point d'un code avec l'option `-O0`.

♥ ⇒ Dans la mise au point des programmes, il est recommandé d'utiliser toute la puissance du compilateur pour détecter dès la compilation les erreurs ou les fragilités d'un code. En choisissant les options adéquates, on lui demandera la plus grande sévérité et le maximum de diagnostics. On exploitera en particulier les avertissements (warnings) qui bien souvent détectent des sources d'erreur potentielles qui ne se manifestent qu'au moment de l'édition de liens ou, pire, de l'exécution (les messages d'erreur produits sont alors plus difficiles à interpréter).

Bibliothèques et exécutables statiques ou dynamiques

On doit aujourd'hui distinguer deux sortes de bibliothèques et deux sortes d'exécutables :

– Les bibliothèques statiques, de suffixe `.a`, sont des archives de fichiers objets, créées avec la commande `ar` [15].

  Elles permettent d'effectuer une édition de liens statique, avec l'option `-static` du compilateur et de l'éditeur de liens, afin de créer des exécutables autonomes [16].

– Les bibliothèques dynamiques partageables, de plus en plus souvent adoptées, de suffixe `.so` (shared object).

  Elles permettent d'effectuer (par défaut) une édition de liens dynamique, et de créer des exécutables beaucoup moins volumineux, pour lesquels l'agrégation des objets est différée au moment de l'exécution. Les exécutables dynamiques ne sont donc plus autonomes et nécessitent la présence des bibliothèques dynamiques [17] lors de l'exécution.

En pratique, fortran dispose d'une bibliothèque intrinsèque, dont l'emplacement est connu du compilateur : c'est pourquoi on charge généralement le compilateur d'appeler l'éditeur de liens `ld` en lui indiquant le chemin de la bibliothèque fortran [18].

1.2.3 Exécution

Le fichier exécutable [19] s'appelle par défaut `a.out`. On peut aussi spécifier son nom lors de la compilation par l'option `-o <fichier_exécutable>`. Ce fichier exécutable se comporte comme une commande du système d'exploitation, lancée en tapant `a.out` [20] par exemple. En particulier, sous

14. On insérera si nécessaire, avant l'option `-l`, le chemin d'accès à la bibliothèque, `-L<rep>` si elle n'est pas placée dans un des répertoires scrutés par défaut lors de sa recherche.

15. La commande `ar` est similaire à `tar` : elle permet de créer des archives de codes objets, de les mettre à jour, d'extraire les objets des bibliothèques, d'en afficher la liste.... Par exemple, on peut afficher la liste des objets dans la bibliothèque intrinsèque de `gfortran` par la commande :
`ar -t /usr/lib/gcc/x86_64-redhat-linux/4.4.4/libgfortran.a` dans laquelle on va trouver par exemple les différentes fonctions mathématiques spécifiques des réels sur 4, 8, 10 et 16 octets de la fonction générique `sin` : `_sin_r4.o`,`_sin_r8.o`, `_sin_r10.o`,`_sin_r16.o`.

16. L'édition de liens statique est en particulier nécessaire s'il on souhaite générer un exécutable destiné à une autre machine (de même architecture) où fortran n'est pas installé.

17. On peut afficher la liste des bibliothèques dynamiques dont dépend un exécutable via la commande `ldd`.

18. De plus, comme les fonctions mathématiques usuelles sont dans la bibliothèque intrinsèque du fortran, il n'est pas nécessaire de la mentionner avec l'option `-l` si l'édition de liens est lancée par le compilateur fortran. Il faudrait au contraire la référencer explicitement si on faisait l'édition de liens avec un compilateur C, par exemple dans le cas des codes mixtes en C et fortran (*cf.* chap. 12, p. 131).

19. Sous UNIX, l'attribut exécutable est automatiquement positionné par l'éditeur de liens lors de la création de ce fichier. L'appel à la commande `chmod` n'est donc pas nécessaire.

20. Cela suppose que le répertoire où se situe l'exécutable fasse partie des chemins contenus dans la variable d'environnement `PATH` sous UNIX. Sinon, il faut préciser le chemin d'accès, en lançant par exemple `./a.out` si l'exécutable est dans le répertoire courant.

UNIX, l'interruption en cours d'exécution est possible en frappant `^C`. De même, ses flux d'entrée, de sortie et d'erreur standard (*cf.* 5.2.1, p. 39) peuvent être redirigés vers des fichiers ou d'autres commandes. Ces redirections sont indispensables si l'exécutable doit être lancé en arrière plan (avec `&`) ou en différé sur un serveur.

Débogueur

Dans la phase de mise au point et de recherche d'erreurs, on peut être amené à exécuter le programme sous le contrôle d'un débogueur (debugger) pour analyser en détail son comportement, notamment le contenu des mémoires où sont stockées les variables et le cheminement dans les instructions. Pour cela, il est nécessaire de compiler le programme avec l'option `-g` pour produire un exécutable spécifique que pourra lancer le débogueur.

Par exemple, sur les systèmes où le compilateur `gcc` est installé, le débogueur `gdb` est disponible et peut analyser des codes compilés avec `gfortran` ou `g95`. Après avoir lancé le débogueur, on peut placer des points d'arrêt (breakpoints) dans le programme source avant de le lancer sous son contrôle. Il est alors possible d'examiner des variables et même de les modifier avant de relancer l'exécution pas à pas ou jusqu'au prochain point d'arrêt... Des interfaces graphiques de `gdb` sont disponibles, notamment `ddd`.

1.2.4 Cas des fichiers sources multiples

Dès qu'une application est un peu importante, elle fait appel à plusieurs procédures (sous-programmes ou fonctions) qu'il est souvent préférable de stocker dans des fichiers distincts, à condition qu'il s'agisse de procédures externes (*cf.* 6.3.2, p. 64). Dans ce cas, on peut procéder à la *compilation séparée* des sous-programmes et fonctions, produisant ainsi des fichiers objets (que l'on pourra éventuellement rassembler dans une bibliothèque personnelle).

```
gfortran -c <subpr1>.f90  <subpr2>.f90
```

Puis on lance la compilation du programme principal et l'édition de liens en fournissant au compilateur le fichier source du programme principal et la liste des fichiers objets produits par la compilation précédente.

```
gfortran <principal>.f90  <subpr1>.o  <subpr2>.o
```

Mais on verra (*cf.* 6.4, p. 65) qu'il est préférable d'intégrer les procédures dans des *modules* : la compilation d'un module produit, en plus du fichier objet, un « fichier de module » d'extension `.mod` permettant de stocker des informations (notamment les interfaces des procédures compilées [21]) utiles pour compiler ultérieurement d'autres procédures qui leur font appel : grâce à l'instruction `USE` portant sur ce module, elles pourront acquérir la visibilité sur l'interface des procédures du module (*cf.* Figure 6.1, p. 66). On notera que ces fichiers de module ne sont pas portables : ils dépendent du compilateur et éventuellement de sa version.

Enfin, dans le cas où on fait appel à une bibliothèque constituée de modules pré-compilés, il faut éventuellement indiquer au compilateur dans quel répertoire trouver les « fichiers de module » associés à la bibliothèque grâce à l'option `-I<mod_rep>` [22].

L'outil `make`

La maintenance (génération et mise à jour) d'applications s'appuyant sur plusieurs fichiers source et éventuellement des bibliothèques peut être automatisée par des outils comme l'utilitaire `make` sous UNIX.

La commande `make` permet d'automatiser la génération et la mise à jour de *cibles* (target), en général un fichier exécutable, qui dépendent d'autres fichiers, par exemple les fichiers sources, en mettant en œuvre certaines *règles* (rules) de construction, constituées de commandes unix.

21. Ces fichiers jouent un rôle comparable aux fichiers d'entête du langage C.
22. Comme pour les fichiers à inclure par le préprocesseur.

Elle s'appuie sur un fichier `makefile` qui liste les cibles, décrit l'arbre des *dépendances* et les règles de production des cibles. Pour plus détails, on pourra consulter la documentation en ligne de `gnumake` : http://www.gnu.org/software/make/manual/make.html.

Les compilateurs `g95` et `gfortran`[23] possèdent une option permettant d'aider à l'écriture des dépendances après une compilation effectuée à la main :
`gfortran -M <fichier.f90>` affiche les dépendances du fichier `<fichier.o>`.

1.3 Les éléments du langage

1.3.1 Les caractères du fortran

Les caractères du fortran 90

L'ensemble des caractères du fortran 90 est constitué des caractères alphanumériques, de signes de ponctuation et de quelques symboles spéciaux. Il comprend :

- les minuscules (lettres de `a` à `z`),
- les majuscules (lettres de `A` à `Z`),
- les chiffres arabes de `0` à `9`,
- le caractère souligné : « `_` »,
- et les symboles suivants[24], parmi lesquels `$` et `?` n'ont pas de signification particulière dans le langage :

	+	-	=	*	(	)
,	.	:	;	'	"	/
<	>	%	!	&	$	?

Noter que le caractère de tabulation (qui pourrait faciliter l'indentation du code) n'est pas autorisé dans les fichiers source fortran, même si plusieurs compilateurs se contentent de produire un avertissement en rencontrant une tabulation. Il est donc prudent de configurer les éditeurs de façon à traduire chaque tabulation en un nombre d'espaces adéquat ; si nécessaire, on peut utiliser le filtre unix `expand` pour effectuer cette conversion a posteriori.

f2003 ⇒

Les caractères du fortran 2003

La norme 2003 a complété le jeu de caractères du fortran par les caractères spéciaux suivants :

~	^	\	{	}	`	[	]	|	#	@

Les crochets carrés `[` et `]` peuvent être utilisés comme délimiteurs dans les constructeurs de tableaux monodimensionnels (*cf.* 7.1.3, p. 82).

1.3.2 Les identificateurs des objets

Les objets (variables, procédures et unités de programme) sont identifiés à l'aide de mots commençant par une lettre et constitués en fortran 95 d'au plus 31 caractères alphanumériques[25] (plus
f2003 ⇒ le souligné « `_` » qui permet un pseudo-découpage des noms en mots). Le nombre de caractères des identificateurs est porté à 63 en fortran 2003. Noter qu'il n'existe qu'un seul espace de noms en fortran : il est en particulier interdit d'utiliser le même identifiant pour une variable et un programme ou une procédure et un module. Contrairement à d'autres langages (notamment le C et unix), le fortran ne distingue pas majuscules et minuscules en dehors des chaînes de caractères.

♥ ⇒ N.-B. : pour des raisons de lisibilité, il est évidemment déconseillé d'en profiter pour introduire

23. Sous `gfortran`, l'option `-M` n'est disponible qu'à partir de la version 4.6.

24. Le langage C comporte plus de caractères avec les délimiteurs (accolades `{}`, crochets `[]`), la contre-oblique `\`, les opérateurs `%`, `~`, `^` et `|`. Le caractère `#` est en fait interprété par le préprocesseur qui peut agir aussi bien sur un programme en C que sur un programme en fortran.

25. En fortran 66 standard, la norme ne différenciait que les six premiers caractères. Heureusement, la plupart des compilateurs sont plus tolérants ce qui permet de choisir des identificateurs plus évocateurs.

des variantes dans l'orthographe des identificateurs, nommant successivement par exemple une même variable `TOTAL`, puis `total` quelques lignes plus loin. On préconise à l'inverse de réserver les capitales pour les mots clefs du langage [26] et d'écrire les identificateurs définis par l'utilisateur en minuscules, ou éventuellement avec les initiales en majuscules pour mettre en évidence les mots dans les noms des objets, comme par exemple dans `TotalPartiel`.

D'autres conventions sont possibles, (consulter par exemple Clerman et Spector (2011) pour les questions de style) sachant que les mots-clefs du langage seront mis en évidence par colorisation lors de l'édition. L'essentiel est de se tenir à une règle précise tout au long du programme.

1.4 Les formats du code source fortran

Pour des raisons de compatiblité ascendante, les compilateurs fortran 90 acceptent deux formats pour les fichiers sources :

- *le format fixe* (fixed format) qui est celui du fortran 77 (lui-même hérité des contraintes imposées par les cartes perforées), avec deux extensions ;
- *le format libre* (free format), qui permet une écriture plus souple du code source, et est fortement préconisé [27] pour les programmes nouveaux. ⇐♥

Ces deux formats sont incompatibles et le compilateur doit être informé (soit au vu du suffixe du nom de fichier, soit grâce à une option, *cf.* annexe F) du format choisi dans les fichiers sources qui lui sont soumis. L'utilisation conjointe de sources aux deux formats est possible en compilant séparément (*cf.* 1.2.4, p. 5) les fichiers sources au format fixe, puis ceux au format libre, et en assemblant ensuite les objets.

1.4.1 Format fixe

- En format fixe, les instructions peuvent s'étendre de la colonne 7 jusqu'à la colonne 72.
- Au-delà de la colonne 72, les caractères ne sont pas pris en compte comme instruction par un compilateur standard [28] ; ⇐⚠
- Une ligne de commentaire peut être introduite grâce au caractère `C` *placé en première colonne.*
- Une ligne ne peut contenir qu'une instruction, mais une instruction peut s'étendre sur plusieurs lignes en plaçant un caractère quelconque, mais différent de `0` et d'un blanc en colonne 6 des lignes de continuation. Toutefois, la coupure de ligne ne peut pas se faire à l'intérieur d'une chaîne de caractères délimitée par des `"..."` ou des `'...'`.
- En dehors d'une ligne de commentaires, les colonnes 2 à 5 sont destinées à des étiquettes numériques permettant de repérer certaines instructions (pour les formats, les branchements ou les boucles).
- Dans la partie instruction de la ligne et hors chaîne de caractères, les blancs ne sont pas significatifs : d'une part, ils ne sont pas nécessaires pour séparer les mots-clefs entre eux ou des identifiants, mais on peut aussi insérer des blancs à l'intérieur des mots-clefs, des noms de variables et des constantes numériques. Utiliser ces facilités est fortement déconseillé, d'autant que les blancs sont significatifs en format libre ; mais il est bon de les connaître pour ⇐⚠

26. Rappelons que certains éditeurs (Language Sensitive Editors), tels qu'`emacs` (*cf.* 1.2.1, p. 3), permettent de basculer globalement la présentation des mots-clefs du fortran entre minuscules, majuscules ou capitalisation des initiales.

27. Le format fixe est déjà obsolescent en fortran 95.

28. Sur les cartes perforées, où chaque carte représentait une ligne de code, ces colonnes permettaient d'indiquer des commentaires, voire de numéroter les cartes pour pouvoir les reclasser en cas de chute du paquet de cartes qui constitue maintenant un fichier. Cependant, en utilisant certaines options (parfois activées par défaut) des compilateurs (voir par exemple F.5.1, p. 172 pour `gfortran`), les colonnes de 73 à 80 (ou 132) peuvent être interprétées comme des instructions.

décrypter certains codes en format fixe qui les exploitaient [29].

Les deux extensions suivantes au format du fortran 77 définissent le format fixe du fortran 90 :

- en plus de `C`, les caractères `c`, `*` et `!` placés en première colonne sont considérés comme introducteurs de commentaire.
- une ligne peut contenir plusieurs instructions, à condition qu'elles soient séparées par des « `;` » (points-virgules).

```
C     exemple d'extrait de fichier source fortran au format fixe
C
C23456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
C        1         2         3         4         5         6         7         8
C                                                                       ||||||||
      SOMME = TERME_1 +
     1        TERME_2 +
     1        TERME_3
C     début de la boucle
      DO 100 I = 1, N                                                    DEBUT
         K = K + I
 100  CONTINUE                                                           FIN
C     fin de la boucle
```

1.4.2 Format libre du fortran 90

- En format libre, les instructions peuvent s'étendre de la colonne 1 à la colonne 132. Les caractères situés au-delà de cette limite ne sont pas interprétés par le compilateur [30].
- Le caractère « `!` » [31], hors chaîne de caractère, permet d'introduire un *commentaire* n'importe où dans la ligne ; ce commentaire se termine à la fin de la ligne [32].
- Une ligne peut contenir plusieurs instructions, séparées par un point-virgule « `;` » [33].

♥ ⇒ Pour des raisons de lisibilité, il est cependant déconseillé de grouper plusieurs instructions sur une même ligne.

- Terminer une ligne d'instruction par le *caractère de suite* `&` indique que l'instruction se continue sur la ligne suivante [34]. Dans le cas où la coupure de ligne intervient au sein d'une chaîne de caractères, il faut insérer un `&` comme premier caractère non-blanc de la ligne de continuation pour indiquer où reprend la chaîne.

29. Les blancs étaient utilisés par exemple afin de séparer les groupes de 3 chiffres, ou les mots constitutifs d'un identifiant. À l'inverse, on pouvait les supprimer entre type et nom de variable pour obtenir un code plus concis.

à éviter

```
C format fixe: à partir de la colonne 7
      IMPLICITNONE
      REALx,sx
      INTEGER :: old j
      old j = 2 000
      x = . 314 159 e 1
      s x = S I N (x)
      W R I T E (*,*) old j, x, s x
```

préférer ⟹

préférable

```
C format fixe: à partir de la colonne 7
      IMPLICIT NONE
      REAL x, sx
      INTEGER :: oldj
      oldj = 2000
      x = .314159e1
      sx = SIN(x)
      WRITE(*,*)  oldj, x, sx
```

30. Certains compilateurs permettent de signaler l'éventuel dépassement de la longueur de ligne, par exemple `gfortran` avec l'option `-Wline-truncation` (*cf.* F.5.3, p. 173).

31. En fortran 90, c'est le seul introducteur de commentaire ; en particulier, « `C` », « `c` » et « `*` » ne permettent plus d'introduire un commentaire dans le source.

32. Le caractère « `!` », introducteur de commentaire n'est pas un délimiteur, contrairement à ce qui se passe en C avec le couple `/*` ... `*/`. Il est à comparer avec le couple `//` introducteur de commentaires du C99 et du C++.

33. Mais, contrairement à ce qui se passe en C, une instruction ne se termine pas nécessairement par un « `;` ».

34. Cette technique ne peut pas être employée pour des lignes de commentaire. À l'inverse, et bien que cette pratique soit déconseillée pour manque de lisibilité, on peut insérer une ou plusieurs lignes de commentaires (chacune introduite par un `!`) entre les différentes lignes (terminées, sauf la dernière, par un `&`) constituant une instruction.

– Les caractères `!` et `&` perdent leur interprétation spéciale lorsqu'ils sont situés *à l'intérieur* d'une chaîne de caractères (entre `"..."` ou entre `'...'`). Mais, dans le cas où la chaîne n'est pas fermée en fin de ligne, un `&` en *dernier caractère* non blanc est interprété comme indiquant que la fin de ligne ne termine pas l'instruction. Le nombre maximum de lignes sur lesquelles peut s'étendre une instruction est limité à 40 en fortran 95 (39 lignes de suite), et 256 en fortran 2003 (255 lignes de suite). ⇐ f2003

```
1  PROGRAM essai
2  !            exemple de source fortran 90 au format libre
3  IMPLICIT NONE
4  REAL :: somme, terme_1 = 1., terme_2 = 2., terme_3 = 3.
5  !
6  ! une instruction longue scindée en plusieurs lignes
7  somme = terme_1 + &                        ! description du premier terme
8          terme_2 + &                        ! description du deuxième terme
9          terme_3                            ! description du troisième terme
10 ! cas où la coupure se fait au sein d'une chaîne de caractères
11 WRITE (*,*) ' chaîne de caractères comportant  plusieurs lignes dans &
12             &le programme source'
13 !           ce & est obligatoire pour marquer la continuation de la chaîne
14 END PROGRAM essai
```

L'instruction d'écriture suivante

```
WRITE (*,*) 'chaîne avec des & et des ! normaux et deux &&
               & spéciaux' ! suivie d'un commentaire
```

affichera sur une seule ligne :
`chaîne avec des & et des ! normaux et deux & spéciaux`

Les blancs Noter que les espaces sont significatifs en format libre. En particulier, en l'absence de séparateur (virgule, parenthèse...), un blanc est obligatoire pour délimiter deux mots-clefs. Mais l'espace est facultatif dans certains cas, comme `ELSEIF`, `DOUBLEPRECISION`, `INOUT`, `SELECTCASE`, ainsi que pour toutes les fins de structures introduites par `END` : `ENDIF`, `ENDDO`, ...

Interposer des espaces permet d'améliorer la lisibilité du code, avec par exemple les règles suivantes (*cf.* Clerman et Spector (2011)), en veillant à maintenir une présentation homogène dans tout le code : ⇐ ♥

– mettre en retrait (indent) les blocs (structures de contrôle, déclaration de types dérivés, interfaces, ...), sans utiliser de tabulation (*cf.* 1.3.1, p. 6) et avec toujours le même nombre d'espaces (2 ou 4 suffisent, mais choisir 8 espaces provoquerait des retraits trop importants pour les structures imbriquées) ;

– entourer d'une espace le signe `=` d'affectation dans les expressions ;

– entourer d'une espace les opérateurs de l'opération principale des expressions ;

– ajouter une espace après la virgule, et seulement après, dans les listes (attributs, arguments des procédures, objets lors de la déclaration, ...).

Chapitre 2

Types, constantes et variables

2.1 La représentation des types numériques

Les entiers sont représentés en mémoire de façon exacte, donc sans perte de précision, mais leur domaine est limité par le nombre de bits choisi pour les représenter.

Au contraire, les réels (et les complexes) ne peuvent pas, en général, être représentés exactement en mémoire : c'est pourquoi on évitera les tests d'égalité sur les réels ; c'est aussi ce qui explique l'impossibilité d'appliquer à des expressions réelles des tests de type énumération de valeurs [1].

♥ ⇒ On verra en particulier que l'écart entre deux réels successifs assez grands ($\approx 10^7$) peut dépasser 1 : il est donc extrêmement déconseillé de compter en utilisant des variables de type réel.

Pour plus de détail sur la représentation des nombres et l'implémentation des opérations arithmétiques, on consultera le chapitre 9 « Arithmétique des ordinateurs » de Stallings (2003).

2.1.1 Représentation des entiers

Si la représentation la plus connue des entiers positifs est celle de la base 10 (qui utilise 10 symboles de `0` à `9`), la plupart des ordinateurs utilisent une représentation en base 2 ne nécessitant que 2 symboles (`0` et `1`) ; en représentation binaire, l'équivalent du chiffre de la représentation décimale est le *bit*. La représentation en *octal* (base 8) utilise les 8 symboles de 0 à 7 alors que la représentation *hexadécimale* [2] (base 16) utilise 16 symboles : 0,1, ..., 9, A, B, ...F.

Représentation des entiers positifs

La représentation des entiers positifs utilise des codes à poids, dont les poids sont les puissances successives de la base b ; soit, avec q coefficients :

$$n = \sum_{i=0}^{q-1} p_i \, b^i \quad \text{où} \quad 0 \leqslant p_i < b \tag{2.1}$$

Réciproquement, les coefficients p_i peuvent être obtenus par des divisions entières successives par b :

- p_0, coefficient de plus faible poids, est le reste de la division de n par b ;
- p_1 le reste dans la division par b^2 ; ...

Dans le cas où $b = 2$, chaque coefficient p_i peut être représenté par un bit. Par exemple, pour l'entier `13` :

$$13 = 1 \times 10^1 + 3 \times 10^0 \quad \text{noté } 13_{10} \text{ en base 10}$$
$$13 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \quad \text{noté } 1101_2 \text{ en base 2}$$

Si on consacre q bits à la représentation d'un entier positif, on peut couvrir le domaine $[0, 2^q - 1]$.

1. `select case` en fortran ou `switch` ... `case` en C.
2. La base 16 a été utilisée sur certains calculateurs IBM.

n	signe	2^6	2^5	2^4	2^3	2^2	2^1	2^0
-128	1	0	0	0	0	0	0	0
-127	1	0	0	0	0	0	0	1
...	...	...	...	...	...	...	...	...
-1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0
$+1$	0	0	0	0	0	0	0	1
...	...	...	...	...	...	...	...	...
$+127$	0	1	1	1	1	1	1	1

TABLE 2.1 – Représentation des entiers relatifs sur 8 bits.

Représentation des entiers négatifs

Les entiers négatifs sont représentés habituellement avec le symbole complémentaire « $-$ » qu'il va falloir traduire en codage binaire par un bit complémentaire appelé *bit de signe*.

En binaire, un entier relatif est donc représenté sur $q+1$ bits. On choisit de coder le signe sur le bit de plus fort poids avec 0 pour les entiers positifs et 1 pour les négatifs. Mais si on codait simplement la valeur absolue en binaire de n sur les q bits restants, une opération arithmétique élémentaire comme l'addition entre entiers de signes différents ne s'effectuerait pas comme celle de deux entiers de même signe.

On choisit au contraire de représenter l'entier n négatif par un bit de signe à 1 suivi des bits du complément à 2^{q+1} de $|n|$, c'est-à-dire ceux de l'entier positif $2^{q+1}-|n|$. On parle alors (abusivement) de complément à deux, obtenu en inversant tous les bits puis en ajoutant 1 avec perte de la retenue. L'arithmétique entière ainsi mise en œuvre fonctionne modulo 2^{q+1}. Le domaine couvert va donc de -2^q à 2^q-1 (voir par exemple 2.3.1, p. 18). Par exemple, les entiers codés sur 8 bits, soit 1 octet ($q=7$) couvrent le domaine $-2^7=-128$ à $2^7-1=+127$ (*cf.* Fig 2.1 et Table 2.1) :

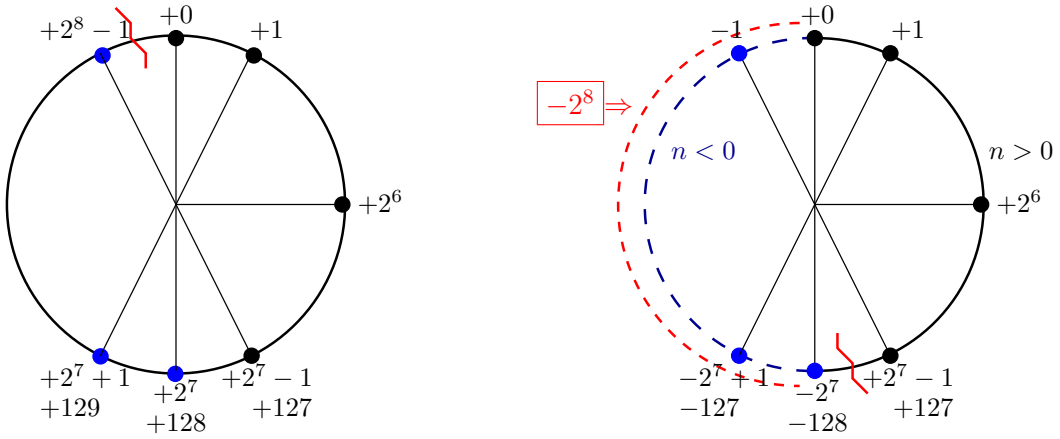


FIGURE 2.1 – Entiers positifs (à gauche) et entiers relatifs (à droite) sur 8 bits : pour passer des positifs aux relatifs, on soustrait 2^8 dans la partie gauche (pointillée) du cercle. La discontinuité initialement en haut à gauche de 0 pour les positifs, se retrouve en bas entre $+127$ et -128.

♠ **Affichage des entiers en binaire** En utilisant le format binaire de sortie noté `B` (*cf.* 5.4.1, p. 49) comme convertisseur binaire-décimal, on peut vérifier simplement la représentation des entiers, par exemple sur 32 bits et sur 8 bits.

```
PROGRAM entier_binaire
IMPLICIT NONE
INTEGER :: i0 ! entier par défaut (32 bits)
! entiers pour stocker au moins 100 => 8 bits
INTEGER, PARAMETER :: k1=SELECTED_INT_KIND(2)
INTEGER(kind=k1) :: i1
WRITE(*, *) "entiers sur",DIGITS(i0),"bits + signe"
WRITE(*, *) "soit ",BIT_SIZE(i0), "bits avec signe"
WRITE(*, "(i11)") HUGE(i0)
WRITE(*, "(b32.32)") HUGE(i0)
WRITE(*, "(i11)") -HUGE(i0)-1
WRITE(*, "(b32.32, /)") -HUGE(i0)-1
WRITE(*, *) "entiers sur",DIGITS(i1),"bits + signe"
WRITE(*, *) "soit ",BIT_SIZE(i1), "bits avec signe"
WRITE(*, "(i4, 1x, b8.8)") HUGE(i1), HUGE(i1)
WRITE(*, "(i4, 1x, b8.8)") -HUGE(i1)-1_k1, &
                           -HUGE(i1)-1_k1
END PROGRAM entier_binaire
```

```
 entiers sur 31 bits + signe
 soit  32 bits avec signe
 2147483647
01111111111111111111111111111111
-2147483648
10000000000000000000000000000000

 entiers sur 7 bits + signe
 soit  8 bits avec signe
 127 01111111
-128 10000000
```

Traditionnellement, comme en décimal, on range les bits par poids décroissant, le plus fort poids à gauche. Mais des variantes dans l'ordre de stockage [3] des octets ou des groupes de deux octets existent : on distingue les conventions little-endian où les octets de poids fort sont stockés en fin et big-endian où les octets de poids fort sont stockés en tête.

2.1.2 Représentation des réels en virgule flottante

Si on codait les réels en virgule fixe [4], la précision absolue de la représentation serait fixe ; mais la précision relative dépendrait de l'ordre de grandeur. Pour conserver une précision **relative** plus indépendante de l'ordre de grandeur, c'est-à-dire pour travailler avec un nombre fixe de chiffres significatifs, on préfère une représentation en *virgule flottante* [5], avec une *mantisse*, représentant la partie fractionnaire et un exposant qui fixe l'ordre de grandeur (*cf* équ. 2.2, p. 12).

Par exemple en base 10, avec 4 chiffres après la virgule, on peut comparer dans la table 2.2, p. 13, les deux représentations approchées (obtenues, pour simplifier, par troncature et non par arrondi) : on constate qu'en virgule fixe, plus les valeurs sont petites, moins on conserve de chiffres.

Représenter un réel (non nul) en virgule flottante, c'est en donner une approximation finie en le développant dans une base b sous la forme :

$$r = s\, b^e m = s\, b^e \sum_{i=1}^{q} p_i\, b^{-i} \tag{2.2}$$

où

- $s = \pm 1$ est le signe ;
- e est un entier qualifié d'*exposant* ;
- la partie fractionnaire m est la *mantisse*.

Mais cette représentation n'est pas unique. Afin de choisir parmi les combinaisons de mantisse et d'exposant, on impose que la mantisse soit comprise entre $1/b$ inclus [6] et 1 exclus soit $p_1 \neq 0$. Par

3. On peut faire l'analogie avec les modes d'écriture de la date suivant les pays qui diffèrent notamment par l'ordre entre le jour, le mois et l'année.

4. On peut, pour simplifier, envisager le cas de la base décimale, et faire l'analogie avec le format `F` en fortran ou `%f` en C, avec un nombre fixe de chiffres après la virgule.

5. Pour poursuivre la comparaison, cela correspond à la notation exponentielle en format `E` en fortran ou `%e` en C

6. Si la mantisse est inférieure à $1/b$, on perd des chiffres significatifs, mais, à exposant fixé, cela permet de représenter des nombres plus petits qualifiés de dénormalisés (*cf.* C.2.2, p. 157).

nombre exact	virgule fixe	virgule flottante
	par troncature	
0.0000123456789	.0000	0.1234×10^{-4}
0.000123456789	.0001	0.1234×10^{-3}
0.00123456789	.0012	0.1234×10^{-2}
0.0123456789	.0123	0.1234×10^{-1}
0.123456789	.1234	0.1234×10^{0}
1.23456789	1.2345	0.1234×10^{1}
12.3456789	12.3456	0.1234×10^{2}
123.456789	123.4567	0.1234×10^{3}
1234.56789	1234.5678	0.1234×10^{4}

TABLE 2.2 – Représentation en virgule fixe et virgule flottante en base 10

exemple en décimal, $1{,}234 \times 10^{-1}$ peut s'écrire $0{,}1234 \times 10^{0}$ ou $0{,}01234 \times 10^{1}$: on exclut donc la deuxième combinaison.

Chaque variante du type réel (déterminée via le paramètre `KIND`) est caractérisée par :

- un nombre q de symboles (chiffres décimaux en base 10 et bits en base 2) de la mantisse qui fixe la précision relative des réels ;
- un nombre de symboles pour l'exposant qui fixe le domaine de valeurs couvert.

Dans cette représentation, en décimal, les réels en virgule flottante sont répartis en progression arithmétique dans chaque *décade* avec un nombre fixe de valeurs par décade et un pas multiplié par 10 à chaque changement de décade. En binaire, c'est dans chaque *octave* (intervalle de type $[2^p, 2^{p+1}[$) qu'ils sont en progression arithmétique avec un nombre $n = 2^{q-1}$ de valeurs par octave ; le pas double à chaque changement d'octave (*cf.* Annexe C, p. 156 sur le codage IEEE).

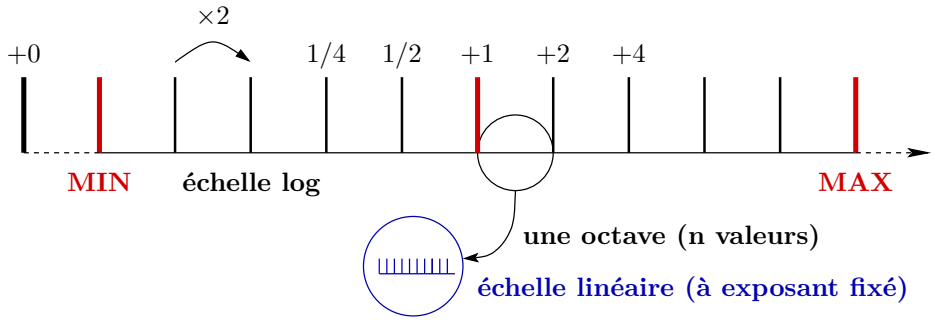


FIGURE 2.2 – Représentation des réels positifs en virgule flottante à base 2 : domaine en échelle log. Seules sont représentées ici les puissances exactes de 2 lorsque l'exposant est incrémenté à mantisse fixe. Le nombre de bits de l'exposant détermine le domaine couvert. Le zoom sur une octave, à exposant fixé, est, lui, en échelle linéaire (*cf.* Fig. 2.3, p. 13).

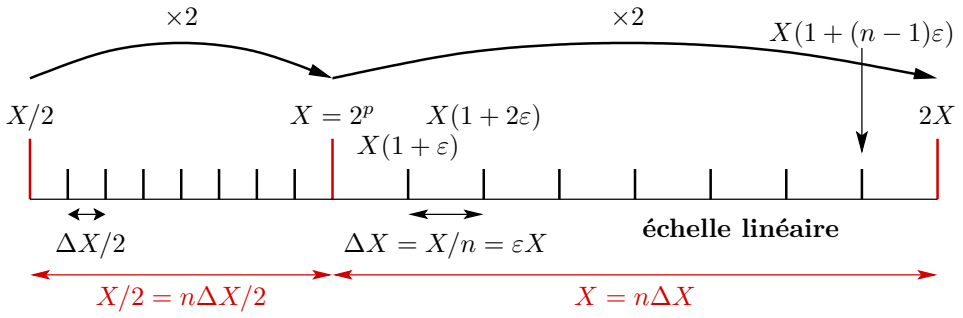


FIGURE 2.3 – Représentation en virgule flottante à base 2 : deux octaves en échelle linéaire. $\Delta X = \varepsilon X$ est le pas dans l'octave $[X, 2X[$ qui comporte $n = 2^{q-1}$ intervalles, où q est le nombre de bits de la mantisse (avec le 1er bit à 1), qui détermine la précision relative de la représentation.

2.2 Les types intrinsèques

Le langage fortran 90 possède les *types intrinsèques* ou prédéfinis suivants :

- `CHARACTER` : chaîne de caractères [7]
- `LOGICAL` : booléen
- types numériques :
  - type numérique représenté exactement :
    - `INTEGER` : entier
  - types numériques représentés approximativement :
    - `REAL` : réel en virgule flottante
    - `DOUBLE PRECISION` : flottant en précision étendue (à éviter : lui préférer une variante de type déterminée de façon plus portable, grâce à l'attribut `KIND`, *cf.* 2.3.2, p. 19)
    - `COMPLEX` : nombre complexe flottant

Notons que les types intrinsèques sont déclinés en différentes variantes ou sous-types (*cf.* 2.3.2, p. 19), dont la liste précédente présente les sous-types par défaut. Enfin, on verra au chapitre 9, p. 104, qu'il est possible d'étendre la liste des types disponibles en définissant des types dérivés pour représenter plus efficacement certaines structures de données.

2.2.1 Déclaration

Les objets doivent être déclarés en tout début [8] de programme, sous-programme, fonction ou module, précisément avant les instructions exécutables, selon la syntaxe générale suivante :

```
<type> [,<liste_d_attributs>::] <liste_d_objets>
```

Le séparateur « `::` » n'est pas obligatoire si la déclaration ne comporte pas d'attributs ni d'initia-
♥ ⇒ lisation, mais il est fortement conseillé. Pour chaque objet déclaré, on peut préciser des attributs qui seront décrits en détail plus loin :

- pour les variables :
  - `PARAMETER` constante nommée (*cf.* 2.2.4, p. 16)
  - `DIMENSION` profil d'un tableau (*cf.* 7.1.1, p. 80)
  - `ALLOCATABLE` objet alloué dynamiquement (*cf.* 7.5.3, p. 92)
  - `SAVE` objet statique (*cf.* 6.1.2, p. 59)
  - `POINTER` objet défini comme un pointeur (*cf.* 11.1, p. 120)
  - `TARGET` objet cible potentielle d'un pointeur (*cf.* 11.1, p. 120)
- pour les procédures :
  - `EXTERNAL` caractère externe d'une procédure (*cf.* 6.5.3, p. 72)
  - `INTRINSIC` caractère intrinsèque d'une procédure (*cf.* 6.5.3, p. 72)
- pour les arguments des procédures :
  - `INTENT` vocation (`IN`, `OUT` ou `INOUT`) d'un argument muet d'une procédure (*cf.* 6.1.4, p. 60)
  - `OPTIONAL` caractère optionnel d'un argument muet d'une procédure (*cf.* 6.5.1, p. 70)

f2003 ⇒
  - `VALUE` pour le passage d'argument par copie de valeur (*cf.* 12.4.1, p. 133)

7. En fortran, un caractère est stocké dans une chaîne de type `CHARACTER` et de longueur 1, contrairement à ce qui se passe en langage C, où seul le type caractère est natif et les chaînes ne sont que des tableaux de caractères.

8. En fortran 2008, il est possible de déclarer tardivement des objets (comme en C89) au début d'une structure de bloc (`BLOCK` ... `END BLOCK`), elle même placée après des instructions exécutables (*cf.* 4.4, p. 32). La portée et la durée de vie de ces objets sont alors limitées au bloc.

- pour les objets encapsulés dans des modules (*cf.* 6.4.3 p. 69) :
  - `PUBLIC` rend accessible un objet à l'extérieur du module courant : c'est l'attribut par défaut
  - `PRIVATE` limite la visibilité d'un objet au module courant
  - `PROTECTED` (fortran 2003 seulement) interdit la modification de l'objet en dehors du module sans restreindre sa visibilité (impose ainsi de passer par une procédure du module pour le modifier) ⇐ f2003

Par défaut, les variables numériques non déclarées suivent un *typage implicite* déterminé par leur initiale : les variables dont l'initiale est `I`, `J`, `K`, `L`, `M` ou `N` sont des entiers et les autres des flottants. L'ordre `IMPLICIT` permet de modifier éventuellement cette convention. La déclaration d'une variable impose explicitement son type et prévaut sur ces conventions.

Mais, il est très fortement conseillé de s'obliger à déclarer toutes les variables, sans mettre en œuvre le typage implicite [9], grâce à l'ordre `IMPLICIT NONE`, placé avant toutes les déclarations. ⇐ ♥ Cette contrainte permet en autres de confier au compilateur le repérage des erreurs typographiques dans les noms de variables : si on a déclaré la variable `somme` et que l'on utilise ensuite `some` sans déclaration, le compilateur va diagnostiquer une erreur. Elle oblige aussi à constituer en tête de programme une sorte de dictionnaire des variables qu'il est conseillé de commenter et qui facilite la lecture et la maintenance du code.

```
IMPLICIT NONE
CHARACTER(LEN=5) :: nom_a_5_lettres ! chaîne de 5 caractères
LOGICAL          :: test, test1
REAL             :: a, longueur ! on ignore le typage implicite pour longueur
COMPLEX          :: nombre_complexe
```

On peut spécifier une variante de type en indiquant la valeur du paramètre `KIND` entre parenthèses, mais cette valeur doit être une constante :

```
<type>(KIND=<kind>) :: <liste_d_objets>
```

2.2.2 Les constantes

Les *constantes* sont des objets dont la valeur ne peut pas être modifiée lors de l'exécution du programme ; elles ne peuvent constituer le membre de gauche d'une instruction d'affectation. On distingue les vraies constantes (literal constants) ou constantes non nommées d'une part et les constantes nommées d'autre part (*cf.* 2.2.4, p. 16), qui sont évaluées lors de la compilation. Il est ⇐ ♥ fortement conseillé de nommer les constantes utilisées dans un programme, pour s'assurer que la même valeur est utilisée dans tout le programme et pour simplifier un éventuel changement de précision d'une constante flottante, voire la transformer en une variable lors d'une généralisation du programme.

Exemples de constantes vraies dans les types prédéfinis

```
-250            ! entier négatif
17              ! entier
B'101'          ! entier en binaire : 5
O'17'           ! entier en octal : 15
Z'1F'           ! entier en hexadécimal : 31
3.14            ! réel en notation décimale
-.33            ! réel en notation décimale
2.1e3           ! réel en notation mantisse et exposant
```

9. Ainsi, le comportement du fortran 90 se rapproche de celui du langage C, dans lequel toutes les variables doivent être déclarées.

```
-2e-3           ! réel en notation mantisse et exposant
2.1d3           ! double précision
(1.5, -2.)      ! complexe 1,5 - 2i
.true.          ! booléen: vrai
.false.         ! booléen: faux
'bonjour !'     ! chaîne de caractères délimitée par des apostrophes (')
"bonjour !"     ! chaîne de caractères délimitée par guillemets (")
'M1 "P & A"'    ! chaîne de caractères (entre les ', seul ' est interprété)
"aujourd'hui"   ! chaîne de caractères (l'apostrophe est autorisée entre les ")
'aujourd''hui'  ! chaîne de caractères (l'apostrophe doublée permet
                !        d'insérer une seule apostrophe dans la chaîne)
''              ! chaîne de caractères vide
```

⚠⇒ Noter en particulier que la notation `1.5e-3` représente la valeur $1{,}5 \times 10^{-3}$ et qu'il est inutile, voire risqué d'appeler explicitement l'opérateur `**` d'élévation à la puissance pour calculer cette constante (*cf.* 3.1.3, p. 24).

2.2.3 Initialisation

Il est possible d'initialiser des objets dès leur déclaration [10], en leur affectant comme valeur des constantes vraies du même type [11], par exemple

```
IMPLICIT NONE
CHARACTER(LEN=5) :: nom_a_5_lettres = 'début'
LOGICAL          :: test = .TRUE. , test1 = .FALSE.
REAL             :: a = 0. , longueur = 1.e3
COMPLEX          :: nombre_complexe = (1.,-1.)
DOUBLE PRECISION :: r = 1.23456789012345d0
```

Mais des expressions simples, dites expressions d'initialisation peuvent aussi être utilisées pour initialiser les variables : elles doivent pouvoir être évaluées par le compilateur.

```
IMPLICIT NONE
REAL :: longueur = 10.,  largeur = 2.
REAL :: surface = longueur * largeur
REAL :: pi = 4.*ATAN(1.)
REAL :: enorme = HUGE(1.e0)
```

f2003 ⇒ Sont notamment autorisés dans les expressions d'initialisation les fonctions intrinsèques élémentaires (*cf.* annexe A, p. 140), les fonctions intrinsèques de transformation (portant sur les tableaux), et les constructeurs de tableaux [12] (*cf.* 7.1.3, p. 82) et de structures (*cf.* 9.2.1, p. 105).

2.2.4 Les constantes nommées (ou symboliques),

Pour déclarer une *constante nommée*, c'est-à-dire attribuer une valeur fixée à la compilation à un identificateur, il faut lui adjoindre l'attribut `PARAMETER`, et l'initialiser.

10. Une variable locale à une procédure devient statique si on l'initialise au moment de la déclaration, elle se comporte donc comme si elle possédait l'attribut `SAVE` (*cf.* 6.1.2, p. 59). Ce n'est pas le cas en C, où l'initialisation se fait à chaque appel de la fonction, sauf si on spécifie l'attribut `static` pour la variable.

11. Veiller à initialiser par exemple les variables de type `DOUBLE PRECISION` avec des constantes du même type, sous peine de perte de précision : dans l'exemple qui suit, les derniers chiffres significatifs de `r` n'auraient pas de sens en l'absence de `d0`

12. En fortran 95, les fonctions intrinsèques n'étaient généralement pas autorisées dans les expressions d'initialisation, à quelques exceptions près, parmi lesquelles les fonctions de précision numérique et certaines fonctions liées au « profil » des tableaux, dont `REPEAT`, `RESHAPE`, `SELECTED_INT_KIND`, `SELECTED_REAL_KIND`, `TRANSFER`, `TRIM`, `LBOUND`, `UBOUND`, `SHAPE`, `SIZE`, `KIND`, `LEN`, `BIT_SIZE`, `HUGE`, `EPSILON`, `TINY`...

Ainsi, `REAL :: enorme = HUGE(1.e0)` était autorisé, mais `REAL :: pi = 4.*ATAN(1.)` n'était pas admis en fortran 95. Ces restrictions ont été levées avec le fortran 2003.

```
IMPLICIT NONE
INTEGER, PARAMETER :: n_points = 100
REAL, PARAMETER    :: pi = 3.1416
```

Il n'est alors plus possible de modifier la valeur de ces constantes nommées, par exemple en écrivant `pi=3.15`.

Les constantes entières nommées peuvent aussi être utilisées pour paramétrer les dimensions de certains tableaux (*cf.* 7.5.1, p. 90), qui seront ainsi fixées au moment de la compilation [13].

2.2.5 Fonctions de conversion de type

Fortran possède de nombreuses fonctions de conversion (explicite) entre les types numériques et leurs variantes, fonctions décrites dans l'annexe A.2, p. 142. En particulier, la conversion implicite d'entier en flottant est assurée par `REAL` et celle de flottant en entier par `INT` qui tronque vers zéro, mais d'autres choix sont possibles pour les conversions explicites : au plus proche avec `NINT`, par excès avec `CEILING`, par défaut avec `FLOOR`.

Ces fonctions de conversion numérique admettent un paramètre optionnel de `KIND` qui permet de préciser la variante de type du résultat.

2.3 Caractéristiques et choix des types numériques

2.3.1 Domaine et précision des types numériques

Les nombres étant stockés sur un nombre de bits fixe, ne sont représentés que dans un domaine fini. De plus, seul le type entier permet une représentation exacte des valeurs numériques, alors que les nombres réels ne sont en général représentés en virgule flottante qu'approximativement [14].

Fonctions d'interrogation sur le codage des types numériques

Plusieurs fonctions intrinsèques permettent de caractériser globalement [15] la représentation du type numérique de leur argument du point de vue :

- de la base b (en général 2) employée dans la représentation (2.1) pour les entiers ou (2.2) pour les réels du type de `x` : `RADIX(x)` ;
- du nombre de digits utilisés pour le représenter : `DIGITS(x)` donne le nombre de symboles (c'est-à-dire de bits dans le cas usuel de la base 2) consacrés au stockage de la mantisse de `x` dans le type de `x`. Plus précisément :
  - si `x` est un entier, `DIGITS(x)` est le nombre q de bits sur lesquels l'entier est stocké hors bit de signe. Il détermine donc le domaine couvert par ce type. Dans le cas entier, `BIT_SIZE(x)` rend le nombre total de bits utilisés pour le stocker (bit de signe inclus).
  - si `x` est un réel en virgule flottante, `DIGITS(x)` est le nombre de bits (ici q) [16] sur lesquels est stockée sa mantisse. Il détermine alors la précision relative de la représentation de `x`.
- du domaine couvert par un type donné :
  - `HUGE(x)`, la plus grande valeur (entière ou réelle) représentable dans le type de `x`.

13. En C, déclarer une variable avec l'attribut `const` permet de diagnostiquer une éventuelle tentative de modification (erreur détectée par le compilateur `gcc` depuis la version 4.0), mais ne suffit pas pour en faire une constante nommée, susceptible par exemple de définir la dimension d'un tableau. Pour ce faire, il faut recourir à la directive `#define` du préprocesseur.

14. Plus précisément, ne peuvent éventuellement être exactement représentés que les rationnels dont le dénominateur est une puissance de la base, donc en pratique du type $n/2^p$ avec des contraintes sur n et p. Ainsi les entiers de valeur absolue assez faible, et certains nombres rationnels de dénominateur 2^p sont représentés exactement en virgule flottante. En revanche la majorité des nombres décimaux, `0.1` par exemple, ne sont pas représentables exactement.

15. Seul le type et non la valeur de l'argument de ces fonctions d'interrogation a une importance.

16. Dans le cas d'une mantisse normalisée, son premier bit, toujours égal à 1, n'est pas stocké (*cf.* C.2.1, p. 157). Cela permet de « gagner » un bit, mais ce bit caché est pris en compte dans la fonction `DIGITS`.

- `TINY(x)`, la plus petite valeur absolue flottante représentable [17] dans le type de `x`.
- `RANGE(x)` qui donne :
  - la puissance de 10 du plus grand entier défini dans le type de `x` s'il s'agit d'un entier : ainsi tout entier `x` tel que `|x|` $\leqslant 10^r$ où r est la valeur de `RANGE` est représentable dans ce type et `RANGE(x) = INT(LOG10(HUGE(x))`
  - la plus petite des valeurs absolues des puissances de 10 du plus petit et du plus grand des réels dans le type de `x`, si `x` est un flottant :
    `RANGE(x) = INT(MIN(LOG10(HUGE(x)), -LOG10(TINY(x))))`
    ainsi tout réel `x` s'écrivant sous la forme `x`= $\pm$ `0.?????` 10^k avec k entier et $|k| \leqslant r$ où r est la valeur de `RANGE`, est représentable dans ce type ;
- de la précision de la représentation en virgule flottante :
  - `PRECISION(x)` donne le nombre de chiffres décimaux *significatifs* d'un réel du type de `x`.
  - `EPSILON(x)` donne la plus petite valeur réelle positive qui n'est pas négligeable devant `1.` dans le type de `x`. C'est la plus petite valeur positive qui, ajoutée à `1.`, donne le flottant successeur de `1.` (le plus proche de `1.` par excès, dans la représentation de `x`).

Si un entier `i` est stocké sur `q+1` bits, `BIT_SIZE(i)` vaut `q+1`, `DIGITS(i)` vaut `q` et `HUGE(i)` vaut $2^q - 1$.

D'autres fonctions intrinsèques d'interrogation permettent de préciser localement la représentation des types numériques ; leur résultat dépend à la fois du type et de la valeur de `x` :

- `EXPONENT(x)` rend un entier donnant l'exposant e de `x` de la représentation (2.2) ;
- `FRACTION(x)` rend un réel donnant la mantisse m de `x` de la représentation (2.2) ;
- `NEAREST(x, s)` rend le réel le plus proche de `x` mais distinct de `x` dans la représentation (2.2) selon la direction fixée par le signe du réel `s` (successeur ou prédécesseur de `x`) ; successeur ou prédécesseur sont en général à égale distance qui est le pas de la progression arithmétique des réels dans une octave, sauf si `x` est une puissance entière de 2 ; par exemple, `NEAREST(1.,+1.)` vaut `1. + EPSILON(1.)` alors que `NEAREST(1.,-1.)` vaut `1. - EPSILON(1.)/2.` ;
- `SPACING(x)` rend la distance entre `x` et son voisin le plus proche en s'éloignant de `0`, soit la distance entre $|x|$ et son successeur ; un multiple de cette distance peut être utilisé comme critère de convergence dans un algorithme itératif. `EPSILON(x)` est donc égal à `SPACING(1.)`.

Caractéristiques des types numériques prédéfinis

Les caractéristiques des *types numériques prédéfinis* dépendent à la fois du processeur et des compilateurs (souvent au travers d'options spécifiques).

Types entiers par défaut Sur un processeur 32 bits, les entiers par défaut sont stockés sur 4 octets donc 31 bits pour la valeur absolue plus un bit de signe. La variante de type (`KIND`) des entiers représente généralement le nombre d'octets sur lesquels est codé cet entier [18].

Sur un processeur de type PC 64 bits, les entiers par défaut peuvent être stockés sur 8 octets, (même si ce n'est pas forcément la règle), donc 63 bits pour la valeur absolue plus un bit de signe. Des options de compilation permettent d'imposer les entiers sur 64 bits (*cf.* Annexe F.7.1 p. 176).

nombre d'octets	BIT_SIZE	DIGITS	HUGE	RANGE
4	32	31	$2147483647 = 2^{31} - 1$	9
8	64	63	$9223372036854775807 = 2^{63} - 1$	18

Le comportement en cas de dépassement de capacité entière dépend du compilateur et des options choisies. Avec les compilateurs `g95` et `gfortran`, l'option `-ftrapv` provoque un arrêt du programme en cas de dépassement de capacité lors d'un calcul en entier.

17. On se limite ici aux représentations normalisées, *cf.* annexe C, p. 156, sachant qu'il est possible de représenter des valeurs absolues plus petites au prix d'une perte de précision.

18. Avec le compilateur `nagfor` de NAG, par défaut, les variantes de type sont numérotées en séquence (option `-kind=sequential` du compilateur). Mais si on précise l'option de compilation `-kind=byte`, les valeurs de `KIND` représentent le nombre d'octets occupés par la variante de type (*cf.* Annexe F.2, p. 169).

Types réels par défaut Les réels sont, sur la plupart des machines, stockés sur 4 octets, dont 24 bits (dont 1 pour le signe) pour la mantisse et 8 bits pour l'exposant. Avec des options de compilation (*cf.* Annexe F.7.2 p. 177), on peut passer à 8 octets dont 53 bits (dont 1 pour le signe) pour la mantisse et 11 bits pour l'exposant (variante de réels aussi disponible sous le type `DOUBLE PRECISION`). On se reportera à l'annexe C p. 156 pour une description de la norme IEEE 754 qui définit une méthode de représentation portable des réels.

nb d'octets	DIGITS	PRECISION	EPSILON	TINY	HUGE
4	24	6	1.192093E-7	1.175494E-38	3.402823E+38
8	53	15	2.220446049250313E-016	2.225073858507201E-308	1.797693134862316E+308

2.3.2 Variantes des types prédéfinis

Les types prédéfinis comportent des *variantes (sous-types)*, notamment selon le nombre d'octets choisis pour stocker l'objet, variantes que l'on peut sélectionner à l'aide du paramètre `KIND`.

En fortran 2008, la liste des paramètres de variantes de type a été introduite dans le standard. ⇐ f2008
Ils sont définis dans le module intrinsèque `ISO_FORTRAN_ENV` [19] (*cf.* 6.4.5, p. 70), en particulier :

- les 4 variantes d'entiers obligatoires nommés en fonction de leur nombre de bits `INT8`, `INT16`, `INT32`, `INT64` [20], ainsi que le tableau des paramètres des variantes d'entiers `INTEGER_KINDS` qui comporte au moins 4 éléments.
- les 3 variantes de réels obligatoires nommés en fonction de leur nombre d'octets `REAL32`, `REAL64`, `REAL128`, ainsi que le tableau des paramètres des variantes de réels `REAL_KINDS`, qui comporte au moins 3 éléments, mais parfois plus.

La norme du fortran 2008 impose donc la présence d'entiers sur 64 bits et de réels sur 128 bits [21]. ⇐ f2008

À chaque type est associé un sous-type par défaut, qui peut dépendre de la machine ; ainsi, les types numériques prédéfinis ne garantissent pas une précision indépendante de la machine. Si l'on veut s'assurer de la portabilité numérique d'une déclaration, en termes de *domaine* (`range`) ⇐ ♥
couvert ou de précision, il faut faire appel à des fonctions intrinsèques afin de choisir les sous-types en fonction du domaine et de la précision.

- pour les entiers, la fonction `SELECTED_INT_KIND(r)` où `r` est un entier positif, renvoie un entier donnant le numéro de la variante du type entier qui permet de couvrir le domaine $[-10^r, 10^r]$. Si aucune variante entière ne convient, cette fonction renvoie la valeur `-1` ;
- pour les réels, la fonction `SELECTED_REAL_KIND([p][,r])` renvoie un entier donnant le numéro de la variante du type réel dont la précision est au moins `p` (au sens donné par la fonction `PRECISION`) ou une étendue couvrant le domaine fixé par `r` (au sens donné par la fonction `RANGE`, c'est-à-dire dont la valeur absolue reste dans l'intervalle $[10^{-r}, 10^{+r}]$). Si aucune variante flottante ne convient, cette fonction renvoie une valeur négative.

Enfin la notion de portabilité numérique reste limitée par la disponibilité des variantes dans un environnement (processeur et compilateur) donné de programmation. Un programme qui requiert une précision ou un domaine exigeants pourra, dans certains environnements, ne pas trouver les variantes nécessaires pour s'exécuter. Mais il existe des bibliothèques qui permettent des calculs dans des précisions non limitées par le compilateur, par exemple avec des types dérivés, notamment FMLIB en fortran.

2.3.3 Constantes nommées spécifiant les variantes des types prédéfinis

Il faut en premier lieu demander au compilateur le paramètre (`KIND`) de la variante de type voulue avec `SELECTED_INT_KIND`, `SELECTED_CHAR_KIND` ou `SELECTED_REAL_KIND` : le résultat est une constante utilisable pour les déclarations de variables qui suivent et aussi pour désigner la

19. Avant la norme 2008, le compilateur NAG (*cf.* Annexe F.2, p. 169) fournissait un module `f90_kind.f90` qui définissant l'ensemble des variantes de type.
20. Ces types entiers dont le nombre de bits est imposé sont à rapprocher des entiers «de taille exacte» `int8_t`, `int16_t` et `int32_t` du C99 définis via `stdint.h`.
21. Sur les processeurs ne disposant pas d'unité flottante suffisante, les réels sur 128 bits, qualifiés de «quad», sont simulés de façon logicielle, au détriment des performances.

variante associée des constantes. Le caractère souligné « _ » permet de préciser la variante du type choisie : la variante[22] est indiquée **après** la valeur pour une constante numérique et **avant** pour une constante chaîne de caractères (*cf.* 8.1.3, p. 97).

```
! recherche des variantes
INTEGER, PARAMETER :: ki = SELECTED_INT_KIND(10)     ! ki=8 sous g95 32/64bits
! choix d'un type entier permettant de représenter 10**10
INTEGER, PARAMETER :: kr = SELECTED_REAL_KIND(9,100) ! kr=8 sous g95 32/64bits
! => type réel permettant de représenter 10**100 avec 9 chiffres significatifs
! déclaration des variables dans ces variantes
INTEGER(KIND = ki) :: i
REAL(KIND = kr)    :: r
WRITE(*,*)  "ki = ", ki, " kr = ", kr ! dépendent du compilateur
i = 1000000000_ki          ! 100000000_8 aurait été moins portable
r = 1.123456789e99_kr      ! 1.123456789e99_8 aurait été moins portable
! r = 1.123456789e99        ! donnerait un dépassement de capacité
```

2.3.4 Une méthode modulaire pour fixer la précision des réels

Dans le cadre d'un programme sensible à la précision des réels, il est important de disposer d'un moyen rapide et portable de modifier le sous-type de tous les réels utilisés (constantes et variables), sans s'appuyer sur des options de compilation (*cf.* F.7.2). Il suffit pour cela de définir la variante ♥⇒ de travail dans un module `my_precision` sous la forme d'une constante nommée (`wp`), puis de déclarer toutes les variables réelles dans ce sous-type. Noter qu'il faut aussi exprimer toutes les ⚠⇒ constantes réelles dans ce sous-type et spécifier le paramètre optionnel `KIND=wp` dans les fonctions où la généricité ne suffit pas à assurer un calcul dans le sous-type voulu. Sachant que les sous-types réels courants utilisent 32, 64 ou 80 bits, on pourra définir les paramètres de type associés (`sp`, `dp`, `edp`) dans un module préliminaire `real_precisions`. Il suffira ensuite de faire le choix :

- soit par renommage via `=>` dans le `USE` : par exemple `USE real_precisions, only: wp=>dp` pour la double précision (*cf.* 6.4.3). C'est une méthode largement utilisée[23] par les bibliothèques, par exemple `Blas` et `Lapack`.
- soit par affectation (ligne 12), méthode préférable, car elle permet d'utiliser aussi directement les précisions `sp` ou `dp` si nécessaire pour certaines variables.

```
1  MODULE real_precisions
2    IMPLICIT NONE
3    ! les differentes variantes de reels disponibles (g95/gfortran)
4    INTEGER, PARAMETER :: sp=SELECTED_REAL_KIND(p=6)   ! simple precision 32 bits
5    INTEGER, PARAMETER :: dp=SELECTED_REAL_KIND(p=15)  ! double precision 64 bits
6    INTEGER, PARAMETER :: edp=SELECTED_REAL_KIND(p=18) ! extra double precision 80
7  END MODULE real_precisions
8
9  MODULE my_precision ! ne modifier que ce module pour choisir la precision wp
10   USE real_precisions
11   ! INTEGER, PARAMETER :: wp = sp ! par exemple simple precision
12   INTEGER, PARAMETER :: wp = dp ! ou sinon double precision
13   ! INTEGER, PARAMETER :: wp = edp ! ou extra double precision
14 END MODULE my_precision
```

22. La variante elle-même peut être une constante vraie ou une constante nommée.
23. Mais cette technique présente l'inconvénient de masquer la variable pointée : elle interdit donc d'utiliser conjointement la précision ajustable `wp` et une précision fixe comme `dp`.

```
16 PROGRAM test_precision
17 USE my_precision
18 IMPLICIT NONE
19 ! toujours utiliser le parametre (kind) de sous-type reel wp = working precision
20 ! dans les declarations, dans les constantes et comme argument optionnel
21 REAL(kind=wp) :: r1 = .1_wp ! constante en accord avec le type
22 REAL(kind=wp) :: r2 = .1   ! constante du type reel par defaut, puis conversion
23 REAL(kind=wp) :: r3, r4
24 r3 = REAL(HUGE(1)-100, kind=wp) ! conversion dans le sous-type voulu
25 r4 = REAL(HUGE(1)-100)  ! sans préciser le sous-type => sous-type par défaut
26 WRITE(*,*) "digits=", DIGITS(r1), "epsilon=", EPSILON(r1)
27 WRITE(*,'(f23.21,a)') r1, " = 0.1 représenté à la précision de travail"
28 WRITE(*,'(f23.21,a)') r2, " = 0.1 représenté à la précision par défaut"
29 WRITE(*,'(i10.10, a)') HUGE(1)-100, "   = huge(1) - 100 en entier"
30 WRITE(*,'(f12.1,a)') r3, " = huge(1) - 100 en réel à la précision de travail"
31 WRITE(*,'(f12.1,a)') r4, " = huge(1) - 100 en réel à la précision par défaut"
32 END PROGRAM test_precision
```

```
 digits=          53 epsilon=  2.22044604925031308E-016
0.10000000000000005551 = 0.1 représenté à la précision de travail
0.10000001490116119385 = 0.1 représenté à la précision par défaut
2147483547   = huge(1) - 100 en entier
2147483547.0 = huge(1) - 100 en réel à la précision de travail
2147483520.0 = huge(1) - 100 en réel à la précision par défaut
```

Chapitre 3

Opérateurs et expressions

3.1 Les opérateurs numériques

Outre l'opérateur unaire `-` qui calcule l'opposé d'un nombre [1], le fortran dispose de quatre opérateurs binaires : addition (`+`), soustraction (`-`), multiplication (`*`) et division (`/`). Ces quatre opérateurs binaires sont prévus pour travailler avec des opérandes de même type (et même sous-type).

Mais le fortran [2] possède aussi un opérateur d'élévation à la puissance, noté `**` et qui est défini différemment suivant la nature de l'exposant :

- si n est entier positif, `a**n` est défini pour tout a réel ou entier par le produit de n facteurs $a \times a \times \cdots a$ (l'exposant n'est pas converti en réel) ;
- si n est entier négatif, `a**n` est défini comme `1/a**(-n)` ;
- si x est réel, `a**x` n'est défini que si a est strictement positif et vaut alors $\exp(x \ln(a))$

♥ ⇒ Ainsi, pour élever un nombre à une puissance entière positive, il faudra éviter d'utiliser un exposant réel, comme par exemple dans `a**4.`, qui forcerait l'utilisation de la deuxième méthode, moins précise numériquement et plus coûteuse.

3.1.1 Règles de priorité entre les opérateurs numériques binaires

Quand l'ordre d'évaluation n'est pas imposé par l'utilisation de parenthèses, les expressions numériques sont évaluées en respectant l'ordre de priorité suivant : (1) `**` (2) `*` et `/` (3) `+` et `-`. À niveau de priorité égal et en l'absence de parenthèses, les évaluations s'effectuent en principe de gauche à droite, sauf pour l'opérateur `**`, pour lequel l'évaluation se fait de droite à gauche pour respecter l'interprétation classique de la notation mathématique $a^{b^c} = a^{(b^c)}$.

expression	évaluée comme	c'est-à-dire
`a + b ** c / d / e`	`a + ( ( (b**c) / d ) / e )`	$a + \dfrac{b^c/d}{e}$
`a ** b ** c * d + e`	`( ( a ** (b**c) ) * d ) + e`	$a^{b^c} d + e$

Mais un compilateur peut s'autoriser de modifier cet ordre, par exemple pour des raisons de rapidité de calcul. Ainsi `a/b/c` est a priori évalué comme `(a/b)/c` , mais si le compilateur optimise la vitesse de calcul et considère que le processeur est plus rapide pour effectuer une multiplication qu'une division, il pourra demander d'évaluer dans l'ordre `a/(b*c)` .

3.1.2 Imperfections numériques des opérations liées aux types

⚠⇒ Les opérations associatives algébriquement peuvent s'avérer non associatives sur ordinateur à cause :

1. `+` peut aussi être utilisé comme opérateur unaire.
2. Contrairement au langage C, qui doit faire appel à la fonction `pow`.

– des erreurs d'arrondi sur les réels en précision limitée,

– ou des dépassements de capacité sur les réels comme sur les entiers.

Il est donc parfois prudent d'imposer l'ordre d'évaluation par des parenthèses, voire de reformuler les expressions. ⇐♥

Exemple de dépassement de capacité

C'est le cas du calcul naïf de la constante de Stefan $\sigma = \frac{2\pi^5}{15}\frac{k^4}{c^2h^3}$ avec des réels sur 32 bits. Les calculs de k^4 et de h^3 vont provoquer des dépassements de capacité[3] par valeur inférieure (underflow) : numérateur et dénominateur étant alors considérés comme nuls, leur rapport 0/0 est indéterminé, et σ est affiché sous la forme `NaN` soit « Not-A-Number » (*cf* C.1.2, 157). La factorisation de $(k/h)^3$ dans l'expression de σ limite le domaine des résultats intermédiaires et permet d'éviter le dépassement.

Comment limiter les erreurs d'arrondi

Si par exemple `a` et `b` sont de signes opposés et de valeur absolue très proches et très supérieure à `c`, par prudence, on évitera d'écrire `a + b + c` qui pourrait laisser le choix au compilateur. On écrira `(a+b) + c` pour minimiser l'erreur d'arrondi, plus faible qu'avec `a + (b+c)` .

Le choix de l'ordre des opérations peut s'avérer crucial pour minimiser les erreurs d'arrondi, par exemple dans l'évaluation des sommes de séries. Dans ce contexte, on aura toujours intérêt à commencer la sommation en accumulant les termes les plus petits entre eux, c'est-à-dire en commençant par le terme d'indice le plus élevé.

De même, quand on doit parcourir un intervalle à pas constant avec une variable réelle `x` dans une boucle `DO` avec compteur[4], le calcul par addition qui accumule les erreurs d'arrondi doit être évité au profit du calcul par multiplication. ⇐♥

à éviter

```
x = xmin - dx
DO i=1, n
   x = x + dx
END DO
```

⟹

préférer

```
DO i=1, n
   x = xmin + REAL(i-1) * dx
END DO
```

3.1.3 Conversions implicites

Lors de l'affectation d'une expression numérique d'un type donné à une variable d'un type numérique différent, il y a conversion implicite, avec éventuellement perte de précision (par exemple dans le cas d'une expression entière affectée à une variable réelle), voire dépassement de capacité (comme dans le cas d'une expression réelle affectée à un entier). Noter que le compilateur `gfortran` est capable de signaler les conversions implicites, grâce à l'option `-Wconversion` (*cf.* F.5.2, p. 173).

```
REAL          :: r = 3.14, s
INTEGER, PARAMETER :: ki=SELECTED_INT_KIND(14) ! variante pour stocker 10**14
INTEGER :: j
INTEGER(KIND=ki) :: i = 1000000000000_ki ! = 10**12
j = r         ! donnera j = 3, comme j = int(r)
s = i         ! donnera s très proche de 1000000000000 à environ 1,2 10**5 près
              ! mais stocké approximativement comme s = real(i)
```

Il en est de même lors de l'évaluation d'expressions dites mixtes impliquant des opérandes de types numériques différents : les opérandes sont alors promus au type le plus riche avant évaluation. La

3. Noter cependant que l'unité de calcul flottant (Floating Processor Unit) effectue bien souvent les calculs dans des registres de 80, voire 128 bits permettant une précision et un domaine étendus pour représenter les résultats intermédiaires. Le dépassement éventuel peut être mis en évidence en activant une option de compilation (`-ffloat-store` *cf.* F.6.1 pour `g95` et `gfortran`, `-float-store` *cf.* F.2.1 pour `nagfor`) interdisant l'usage des registres étendus.

4. On rappelle que le compteur lui-même est forcément entier pour éviter les erreurs d'arrondi.

hiérarchie des types numériques comporte, par ordre de richesse croissante, les entiers, suivis des flottants et au plus haut niveau les complexes. Au sein d'un type donné, les variantes de type sont classées selon le nombre d'octets utilisés : par exemple le type `DOUBLE PRECISION` est plus riche que le type prédéfini `REAL`.

```
COMPLEX      :: u, v
REAL         :: r, s
INTEGER      :: i, j
v = u + r       ! est calculé comme v = u + cmplx(r)
v = u + i       ! est calculé comme v = u + cmplx(i)
s = r + i       ! est calculé comme s = r + real(i)
!               combinaison des deux conversions implicites
j = u + r       ! est calculé comme j = int(u + cmplx(r))
```

♥⇒ Pour une bonne lisibilité, on s'efforcera de rendre explicites de telles conversions, en utilisant les fonctions intrinsèques dites de conversion de type [5] (*cf.* 2.2.5, p. 17 et Annexe A.2, p. 142).

Cas de la division entière

⚠⇒ Noter que l'opérateur `/` appliqué à des entiers permet de calculer le quotient au sens de la division euclidienne [6] : par exemple avec des constantes, `5/3` donne `1`, alors que `5./3.` donne bien `1.666667`. Ne pas croire qu'il suffit de stocker le résultat de la division de deux entiers dans une variable flottante pour imposer une division flottante : c'est seulement le résultat de la division euclidienne qui est ensuite converti en flottant.

```
INTEGER :: i, j
REAL    :: x, y
DOUBLE PRECISION :: t1, t2
i = 3
j = 2
x = i / j
y = REAL(i) / REAL(j)
t1 = 1./3.
t2 = 1./3.d0
```

Dans l'exemple ci-contre, `x` vaut `1.` alors qu'avec la conversion forcée d'au moins un des opérandes en flottant, `y` vaut `1.5`. De même, dans `t1`, la division est évaluée en simple précision et le résultat converti en double, alors que dans `t2`, elle est évaluée en double précision.

⚠⇒ Noter enfin que l'élévation d'un entier à une puissance entière négative va donner lieu à une division entière : par exemple `10**(-3)` sera interprété [7] comme `1/10**3` ou encore `1/1000` qui est nul. Mais `10.**(-3)` donnera bien `.001`.

3.2 Les opérateurs de comparaison

Fortran possède six opérateurs de comparaison dont le résultat est une variable booléenne. L'ancienne notation du fortran 77 est acceptée (*cf.* table 3.1, p. 25).

5. Ces fonctions sont les équivalents de l'opérateur de conversion explicite du langage C ou `cast`, noté en préfixant l'expression par le type d'arrivée entre parenthèses.

6. Ce comportement est commun à beaucoup d'autres langages. Mais il est source de tellement d'erreurs que par exemple le langage `python` a changé de convention au passage de la version 2 à la version 3 en ajoutant un opérateur spécifique, noté `//` pour la division entière !

7. En particulier, si la constante réelle `1.5e-3` vaut $1,5 \times 10^{-3}$, l'expression constante réelle `1.5*10.**(-3)` a la même valeur alors que l'expression `1.5*10**(-3)` est nulle.

fortran	fortran77	signification	remarques à propos des notations
<	.LT.	inférieur à	
<=	.LE.	inférieur ou égal à	
==	.EQ.	égal à	Ne pas confondre avec l'opérateur d'affectation =
>=	.GE.	supérieur ou égal à	
>	.GT.	supérieur à	
/=	.NE.	différent de	Ne pas confondre avec != du langage C

Table 3.1 – Opérateurs de comparaison

```
REAL                                  :: a, b, r
INTEGER                               :: i
IF( a > 0 )  b = LOG(a)               ! a > 0 est évalué à .true. si a est positif
IF( i /= 0 ) r = 1. / REAL(i)
```

Étant donné que les nombres flottants ne sont représentés qu'approximativement en machine, les tests d'égalité entre flottants sont déconseillés ; on préférera comparer les différences à un seuil, en utilisant la fonction intrinsèque `EPSILON` (*cf.* annexe A.4, p. 144). ⇐♥

On remplacera par exemple : `IF ( a == b ) THEN` par :
`IF ( ABS(a-b) <= c *ABS(a) * EPSILON(a) ) THEN`
où `c` est une constante supérieure à 1.

3.3 Les opérateurs booléens

Fortran dispose de quatre opérateurs logiques binaires permettant de combiner des expressions logiques entre elles, ainsi que de l'opérateur unaire de négation (*cf.* table 3.2, p. 25).

.AND.	ET
.OR.	OU (inclusif)
.NOT.	Négation unaire
.EQV.	Équivalence
.NEQV.	OU exclusif

Table 3.2 – Opérateurs booléens

```
REAL                                  :: r
LOGICAL                               :: grand, petit, moyen
grand = r > 1000.
petit = r < .0001
moyen = .NOT. (grand .OR. petit)
```

L'évaluation d'une expression logique ne nécessite pas forcément celle de toutes ses sous-expressions : en particulier, si le premier opérande d'un `AND` est faux ou si le premier opérande d'un `OR` est vrai, le résultat ne dépend pas du second, mais la norme [8] ne garantit pas qu'il ne soit pas évalué [9]. Afin d'écrire du code portable, on ne s'appuiera donc pas sur cette propriété pour conditionner un calcul (*cf.* par exemple 6.5.1, p. 70). ⇐⚠

8. Le compilateur `g95` fournit cependant une option `-fshort-circuit` qui assure que le second membre d'un `.AND.` ou d'un `.OR.` ne soit évalué que si nécessaire.
9. Le langage C, à l'inverse, assure une évaluation minimale des opérandes du `&&` et du `||`.

3.4 Opérateur de concaténation des chaînes de caractères

L'opérateur `//` permet de concaténer [10] deux chaînes de caractères.

```
CHARACTER(len=5)                    :: jour='mardi'
CHARACTER(len=4)                    :: mois='juin'
CHARACTER(len=13)                   :: date
date = jour//'-04-'//mois           ! contiendra 'mardi-04-juin'
```

3.5 Priorités entre les opérateurs

En fortran, les opérateurs intrinsèques numériques d'une part, l'opérateur de concaténation de chaînes d'autre part ne manipulent que des types respectivement numériques ou chaînes. La présence du type booléen [11] impose une première hiérarchie parmi les opérateurs : les comparaisons donnant un résultat booléen, elles ont une priorité inférieure aux opérateurs numériques et à l'opérateur `//`. Les opérateurs portant sur des booléens viennent donc en dernière priorité.

À l'intérieur de cette première hiérarchie, on place en tête l'opération d'élévation à la puissance (avec évaluation commençant par la droite), multiplication et division avant addition et soustraction, mais les opérateurs unaires (`+`, `-`) avant leurs correspondants binaires.

Concernant les opérateurs logiques, `.NOT.` unaire est prioritaire devant les opérateurs binaires, et `.AND.`, assimilé à la multiplication, est prioritaire devant `.OR.`, assimilé à l'addition.

Les opérateurs intrinsèques du fortran sont présentés par ordre de priorité décroissante dans le tableau 3.3, p. 26.

type	opérateurs

numériques	`**`
	`*` et `/`
	`+` et `-` unaires
	`+` et `-` binaires

chaînes	`//`
comparaisons	`==`, `/=`, `<`, `<=`, `>=`, `>`

booléens	`.NOT.`
	`.AND.`
	`.OR.`
	`.EQV.` et `.NEQV.`

TABLE 3.3 – Hiérarchie des opérateurs intrinsèques, classés par ordre de priorité décroissante ; les opérateurs figurant sur une même ligne ont la même priorité.

Par exemple, si `x` et `y` sont des réels, et `ch1` et `ch2` des chaînes de longueur 2, l'expression :
`x-y > -2. .or. x+y < 1. .and. ch1//ch2=='oui!'`
est évaluée comme :
`((x-y) > -2.) .or. (((x+y) < 1.) .and. ((ch1//ch2) == 'oui!'))`

♥ ⇒ Mais, par prudence et pour la lisibilité du code, on n'hésitera pas à expliciter les priorités par des parenthèses.

Noter que les opérateurs surchargés (*cf.* 10.2.1, p. 115) héritent de leur position intrinsèque dans la hiérarchie. En revanche, les opérateurs nouvellement définis (*cf.* 10.2.2, p. 116) acquièrent la plus forte priorité s'ils ont unaires et la plus faible s'ils sont binaires.

10. En langage C, la concaténation de chaînes de caractères est obtenue par simple juxtaposition, sans opérateur explicite.

11. En C, la situation est nettement plus complexe : d'une part, les opérateurs de comparaison donnent des résultats de type entier, susceptibles d'intervenir dans des opérations numériques. D'autre part, l'opérateur `=` d'affectation peut être employé plusieurs fois dans une expression. L'usage des parenthèses est donc encore plus conseillé pour faciliter l'interprétation d'une expression en C.

Chapitre 4

Structures de contrôle

Le nommage des structures de contrôle est facultatif, mais se révèle précieux pour la lisibilité des codes et leur maintenance ; il s'avère cependant nécessaire dans le cas d'une rupture de séquence (*cf.* 4.3.3, p. 30) qui ne concerne pas la boucle intérieure. Plus généralement, il est fortement conseillé de nommer systématiquement les structures comportant des ruptures de séquence. ⇐ ♥

4.1 Structures `IF`

4.1.1 L'instruction `IF`

La forme la plus simple de l'exécution conditionnelle est réservée au cas d'une instruction unique.

```
IF (<expression logique>) <instruction>
```

Si l'expression logique est évaluée à `.TRUE.`, l'instruction est exécutée ; sinon, le contrôle est passé à l'instruction suivante.

```
INTEGER :: i, j, k
IF(j /= 0) k = i/j  ! k n'est calculé que si j est différent de 0
```

4.1.2 La structure `IF ... END IF`

Pour conditionner l'exécution d'un bloc d'instructions à une condition, on emploie la structure :

```
[<nom> :] IF (<expression logique>) THEN
  <bloc d'instructions> ! exécuté si la condition est vraie
END IF [<nom>]
```

où *<nom>* est un identificateur optionnel permettant d'étiqueter le test.

```
INTEGER :: i
...
chiffre: IF (i < 10) THEN
  WRITE(*,*) 'i est inférieur à 10'
END IF chiffre
```

Afin d'exprimer une alternative, cette structure peut comporter un bloc `ELSE` qui n'est exécuté que si la condition est évaluée à `.FALSE.` [1].

1. Si l'identificateur est spécifié en tête de la structure `IF`, il devient nécessaire à la fin du `END IF`, mais pas forcément après le `ELSE`.

```
[<nom> :] IF (<expression logique>) THEN
  <bloc d'instructions> ! exécuté si la condition est vraie
ELSE [<nom>]
  <bloc d'instructions> ! exécuté si la condition est fausse
END IF [<nom>]
```

Si un branchement est possible de l'intérieur de la structure vers l'extérieur, aucun accès vers cette structure n'est autorisé sans passage par le test initial. Ces structures peuvent être imbriquées de façon complexe, bien que, parfois, le recours à un `SELECT CASE` s'avère plus efficace et plus lisible.

```
INTEGER :: i
...
chiffre: IF (i < 10) THEN
  WRITE(*,*) 'i est inférieur à 10'
ELSE chiffre
  WRITE(*,*) 'i est supérieur ou égal à 10'
END IF chiffre
```

4.1.3 Utilisation du `ELSE IF`

Dans le cas où plusieurs tests imbriqués sont effectués, le test `ELSE IF` permet de « raccourcir » l'écriture en « aplatissant » la structure : noter qu'alors un seul `END IF` ferme la structure globale.

Structures imbriquées avec deux `END IF`

```
IF (i < -10) THEN ! externe
  WRITE(*,*) "i < -10"
ELSE
  IF (i < 10) THEN ! interne
    WRITE(*,*) "-10 <= i < 10"
  ELSE
    WRITE(*,*) "i >= 10 "
  END IF ! endif interne
END IF ! endif externe
```

Structure aplatie avec un seul `END IF`

```
IF (i < -10) THEN
  WRITE(*,*) "i < -10"
! ELSE IF sur une même ligne
ELSE IF (i < 10) THEN
  WRITE(*,*) "-10 <=i <10"
ELSE
  WRITE(*,*) "i >= 10 "
END IF    ! un seul endif
```

Dans la forme « aplatie », il est possible d'ajouter autant de clauses `ELSE IF(...) THEN` que nécessaire et la dernière clause `ELSE` n'est pas obligatoire.

4.2 Structure `SELECT CASE`

Lorsque les choix possibles sont nombreux, et portent sur une même expression, la structure `SELECT CASE` est souvent plus efficace que son équivalent à l'aide de `IF` imbriqués. Elle permet, au vu de la valeur d'une expression, de choisir un traitement parmi un ensemble de cas qui doivent être énumérés : l'expression à tester doit faire l'objet d'un test d'égalité ; c'est pourquoi cette structure ne peut s'appliquer qu'à des types possédant une représentation exacte, à l'exclusion des réels.

```
[<nom> :] SELECT CASE (<expression>)
  CASE (<sélecteur_1>) [<nom>]
      <bloc d'instructions>
  CASE (<sélecteur_2>) [<nom>]
      <bloc d'instructions>
```

```
  ...
  CASE (<sélecteur_n>) [<nom>]
      <bloc d'instructions>
  CASE DEFAULT [<nom>]
      <bloc d'instructions>
END SELECT [<nom>]
```

où *<expression>* est une expression de type entier, booléen ou chaîne de caractères, et *<sélecteur_i>* une constante ou une liste de constantes séparées par des virgules, un élément de la liste pouvant être constitué d'un intervalle de valeurs spécifié selon la syntaxe suivante :
<borne_inférieure>:*<borne_supérieure>*,
l'une ou l'autre des bornes pouvant être omise [2], ce qui permet de représenter un nombre infini de valeurs. Le cas `DEFAULT` concerne les expressions qui n'ont coïncidé avec aucun des sélecteurs.

```
INTEGER  :: i
...
tri: SELECT CASE (i)
   CASE(0) tri
      WRITE(*,*) ' i = 0'
   CASE(1,-1) tri
      WRITE(*,*) ' i = 1 ou i = -1'
   CASE(2:10) tri
      WRITE(*,*) ' 2 <= i <= 10'
   CASE(11:) tri
      WRITE(*,*) ' i >= 11'
   CASE DEFAULT tri
      WRITE(*,*) 'i < -1'
END SELECT tri
```

```
CHARACTER(len=3) :: rep
...
valid: SELECT CASE (rep)
   CASE('oui','OUI') valid
      WRITE(*,*) ' oui'
   CASE('non','NON') valid
      WRITE(*,*) ' non'
   CASE DEFAULT valid
      WRITE(*,*) 'réponse invalide'
END SELECT valid
```

4.3 Structures de boucles

Pour effectuer des itérations, le fortran dispose des boucles `DO` avec ou sans compteur, et de la boucle `DO WHILE`. Mais, lorsqu'il s'agit de conditionner le traitement des éléments d'un tableau par un test portant sur un tableau conformant (*cf.* 7.1.1), l'utilisation de l'instruction `WHERE`, ou de la structure `WHERE ... END WHERE` (*cf.* 7.3.2, p. 85) s'avère plus efficace.

4.3.1 Boucles `DO` avec compteur

```
[<nom> :] DO <entier>=<debut>,<fin> [,<pas>]
 <bloc d'instructions>
END DO [<nom>]
```

où *<debut>*, *<fin>* et *<pas>* sont des expressions *entières* [3]. La valeur du pas d'incrémentation du compteur est 1 par défaut, mais peut être négative. Pour certaines combinaisons des paramètres, il est possible que le bloc ne soit exécuté aucune fois. Il est interdit de modifier la valeur du compteur au sein de la boucle ; sa valeur à la fin de la boucle est la première valeur qui ne respecte pas la condition de fin de boucle, sauf en cas de sortie par `EXIT`.

2. Comme dans les shells de type `sh` d'UNIX, mais contrairement au comportement du `switch` en langage C et des shells de type `csh`, dès qu'un choix a été sélectionné (et donc un bloc d'instructions exécuté), le contrôle est passé à la fin de la structure ; cela équivaudrait en C à faire suivre chaque bloc par l'instruction `break` (ou `breaksw` en `csh`).
3. La possibilité d'utiliser un indice de boucle réel a disparu de la norme lors du passage au fortran 90.

```
INTEGER :: i, imax = 10, n
n = 0
impairs: DO i = 1, imax, 2     !  donc  i = 1, 3, 5, 7, 9
   n = n + i
END DO impairs
WRITE (*,*) 'somme des entiers impairs inférieurs à 10', n
```

4.3.2 Boucles DO WHILE

Si le nombre de passages n'est pas connu avant l'entrée dans la boucle, elle doit être contrôlée non par un compteur mais par un test logique : c'est l'objet de la boucle `WHILE` qui intervient de façon classique dans les méthodes itératives. Par prudence, on comptera malgré tout le nombre de ♥ ⇒ passages pour déclencher une sortie par `EXIT` (*cf.* 4.3.3) si le nombre de passages dépasse une limite raisonnable.

```
[<nom> :] DO WHILE (<expression_logique>)
      <bloc d'instructions>
END DO [<nom>]
```

Tant que l'expression logique est vraie, le bloc d'instruction est exécuté. Pour que la boucle ait un intérêt, il faut que le bloc d'instruction modifie la valeur de l'expression logique à un moment donné.

```
INTEGER :: i, imax = 10, n
n = 0
i = 1
impairs: DO WHILE ( i <= imax )
   n = n + i
   i = i + 2  ! modification de l'expression qui intervient dans le test
END DO impairs
WRITE (*,*) 'somme des entiers impairs inférieurs à 10', n
```

4.3.3 Ruptures de séquence dans les boucles : EXIT et CYCLE

Les ruptures de séquence dans les boucles sont possibles grâce aux branchements `EXIT` et `CYCLE`[4].

Branchement à l'indice suivant (éventuel) : CYCLE

L'instruction `CYCLE` à l'intérieur d'une structure `DO ... END DO` permet de court-circuiter la fin du bloc et de passer le contrôle à l'indice de boucle suivant.

```
INTEGER :: i, imax = 10, n
n = 0
impairs : DO i = 1, imax                    ! a priori 1,2,3,4, ... 10
   IF ( modulo(i,2) == 0 ) CYCLE  ! ne pas considérer les entiers pairs
   n = n + i
END DO impairs
WRITE (*,*) 'somme des entiers impairs inférieurs à 10', n
```

4. Ces instructions correspondent respectivement à `break` et `continue` en langage C.

En cas de boucles imbriquées, l'instruction `CYCLE` concerne a priori la boucle *la plus interne* dans laquelle elle est située. Pour qu'elle puisse s'adresser à une boucle externe, il est nécessaire de nommer la boucle et d'utiliser son nom comme argument de `CYCLE` [5].

```
INTEGER :: i, imax, j, jmax
externe: DO i = 1, imax
   ...
   interne: DO j = 1, jmax
      ...
      IF (<expr_log>) CYCLE externe ! fait passer au i suivant
      ...
   END DO interne
   ...
END DO externe
```

Sortie de boucle : `EXIT`

L'instruction `EXIT` à l'intérieur d'une structure `DO ... END DO` permet de sortir immédiatement de la boucle et de passer le contrôle à l'instruction qui suit le `END DO`.

```
INTEGER :: i, n
n = 0
DO i = 1, 1000, 2          ! a priori de 1 à 999
   IF ( i > 10 ) EXIT      ! limitation à 10 (un seul EXIT exécuté)
   n = n + i
END DO
WRITE (*,*) 'somme des entiers impairs inférieurs à 10', n
```

Dans le cas de boucles imbriquées, comme pour l'instruction `CYCLE`, l'instruction `EXIT` ne sort que de la boucle *la plus interne* dans laquelle elle est située. Pour pouvoir sortir directement d'une boucle externe, il faut nommer la-dite boucle et utiliser la syntaxe `EXIT <nom>` [6].

4.3.4 Boucles `DO` sans compteur

```
[<nom> :] DO
      <bloc d'instructions>
END DO [<nom>]
```

Cette structure de boucle, a priori infinie, n'a d'intérêt que si le bloc comporte une instruction `EXIT` permettant de sortir de la boucle au vu d'un test `IF` [7].

```
INTEGER :: i
positifs: DO
   WRITE(*,*) 'entrer un entier positif, sinon 0 pour terminer'
   READ(*,*) i
   IF ( i == 0 ) EXIT positifs
   ...              ! bloc d'instructions
END DO positifs
```

5. Ces possibilités de branchement s'adressant à une boucle externe nommée suivant la même syntaxe qu'en fortran existent aussi en `java` pour les branchements `break` et `continue` concernant des structures de contrôle itératives comme `while` ou `for`.

6. En fortran 2008, cette syntaxe s'applique à d'autres structures que les blocs (*cf.* 4.6, p. 33).

7. À condition de placer le test pour la sortie de la boucle en tête de bloc, cette structure permet de remplacer la structure `DO WHILE ... END DO`, déconseillée car mal adaptée au calcul parallèle.

f2008 4.4 La structure BLOCK

Fortran 2008 a introduit une structure de bloc, en tête duquel il est possible de déclarer des entités *locales* : types, variables, constantes nommées. Les blocs peuvent être placés après [8] des instructions exécutables. Ils peuvent être imbriqués et nommés.

```
[<nom> :] BLOCK
    <déclarations>
    <instructions exécutables>
END BLOCK [<nom>]
```

La portée des déclarations d'un bloc est limitée au bloc et à ses sous-entités. En particulier, les tableaux déclarés et alloués dans un bloc sont *implicitement* désalloués en sortie de bloc. L'instruction `IMPLICIT` est interdite dans un bloc.

Les entités déclarées localement dans le bloc masquent celles de même nom déclarées dans le contexte englobant. La structure de bloc permet donc d'éviter les conflits de nommage, en particulier lors de l'inclusion de code via `INCLUDE` ou l'instruction `#include` pour le préprocesseur. En informant le compilateur du caractère local de certaines entités, elle peut aussi permettre d'aider à l'optimisation du code.

```
PROGRAM bloc
! ok gfortran 4.9 le 16/10/2015
! ok nagfor 5.3(854) le 21/10/2015
IMPLICIT NONE
INTEGER :: i
INTEGER :: tmp ! globale
tmp = 5
WRITE(*,*) "boucle sur", tmp,"cas"
DO i=1, tmp
  BLOCK
    INTEGER :: tmp  ! locale au bloc
    ! tmp locale masque tmp globale
    tmp = 2*i
    WRITE(*,*) "locale", tmp
  END BLOCK
END DO
WRITE(*,*) "globale", tmp
END PROGRAM bloc
```

f2003 4.5 La structure ASSOCIATE

Dans l'objectif de simplifier l'écriture d'expressions complexes, fortran 2003 a prévu la possibilité de définir un raccourci (alias) pour une variable ou une expression : la portée du raccourci est limitée à la structure `ASSOCIATE`. Les blocs `ASSOCIATE` peuvent être imbriqués et nommés.

```
[<nom> :] ASSOCIATE(<liste d'associations>)
    <instructions exécutables>
END ASSOCIATE [<nom>]
```

où chaque association s'écrit à l'aide de l'opérateur `=>` sous la forme
`alias => <expression>` ou `<variable>`.

8. La structure `BLOCK` est comparable au bloc délimité par des accolades en C89, Noter que, à la différence du C99 ou du C++, fortran exige toujours qu'à l'intérieur d'un bloc, les déclarations soient placées avant les instructions exécutables.

Cette association allège notamment l'écriture d'instructions impliquant de façon répétitive [9] des sections de tableaux, des composantes de types dérivés ou des sous-chaînes de caractères. Noter que le raccourci n'a pas à être déclaré. Ne pas le confondre, malgré la notation, avec l'association d'un pointeur à une cible.

Exemple de raccourci d'expression

```
REAL :: x, a, b, z
...
ASSOCIATE(y => 1 - exp(-x)) ! y  est un raccourci d'expression
    z = a*y + b/y
END ASSOCIATE
```

Exemple de raccourcis de variables

```
REAL, DIMENSION(n*p) :: vect
REAL, DIMENSION(n) :: u1, u2
...
ASSOCIATE(v1 => vect(1:n*p-p+1:p), v2 => vect(p:n*p:p))
    v1 = v1 + u1   ! ajout de u1 dans vect avec un pas p à partir du premier
    v2 = v2 + u2   ! ajout de u2 dans vect avec un pas p jusqu'au dernier
END ASSOCIATE
```

f2008

4.6 Sortie de structure quelconque avec l'instruction EXIT nommé

En fortran 2008, l'instruction de sortie `EXIT` peut s'appliquer à d'autres structures que les boucles, à condition qu'elles soient nommées. En plus des boucles, `EXIT` suivi d'un nom permet notamment de sortir des structures nommées `BLOCK`, `IF`, `SELECT CASE`. Il faut donc distinguer deux syntaxes :

- `EXIT` suivi d'un nom de structure transfère le contrôle à la fin de la structure ainsi nommée, quelle qu'elle soit ;
- `EXIT` non suivi d'un nom de structure transfère le contrôle à la fin de la *boucle* englobante la plus interne. ⇐⚠

Pour éviter toute ambiguïté en fortran 2008, on suivra le conseil de Metcalf *et al.* (2011) en utilisant systématiquement la syntaxe `EXIT <nom>`, y compris pour indiquer la sortie de la boucle la plus interne. ⇐ ♥

L'exemple suivant montre trois possibilités pour `EXIT`, dont deux en commentaire. On évitera ici la première forme non nommée.

```
externe : DO i=1,n
  interne: BLOCK
    IF(i >2) THEN
      WRITE(*,*) "sortie"
      ! exit ! sortie non nommée de la boucle do englobante (à éviter)
      ! exit externe ! idem = sortie nommée de la boucle do
      EXIT interne ! sortie du bloc nommé
    END IF
    WRITE(*,*) "dans le bloc", 2*i
  END BLOCK interne
  WRITE(*,*) "dans la boucle", -2*i
END DO externe
```

9. C'est en particulier le cas des instructions qui, en C, feraient intervenir les opérateurs d'affectation composée, par exemple `+=` pour des additions cumulées.

4.7 Autres instructions de contrôle

4.7.1 Instruction CONTINUE

L'instruction *vide* CONTINUE, très utilisée pour délimiter les fins de boucles dans les anciennes versions du fortran, voit, avec l'apparition du END DO, son emploi réduit aux rares branchements spécifiés par des étiquettes numériques (*cf.* 4.7.2, p. 34 et 4.7.6, p. 35).

4.7.2 Branchement par GO TO

L'instruction GO TO <*étiquette_numérique*> permet le branchement vers l'instruction repérée par l'étiquette numérique. L'étiquette numérique est un nombre entier positif qui doit être placé en début de ligne (premier caractère non blanc) et séparée d'au moins un espace de l'instruction sur laquelle se fait le branchement. Inévitable dans les versions anciennes du fortran, l'usage du GO TO est fortement déconseillé pour la lisibilité des programmes, et doit être réservé au traitement des circonstances exceptionnelles, quand elles ne peuvent être gérées via CYCLE ou EXIT.

4.7.3 Instruction STOP

L'instruction STOP [<*code d'arrêt*>] impose l'arrêt immédiat du programme et l'affichage éventuel du code d'arrêt (une chaîne de caractères ou un entier d'au plus 5 chiffres) qui peut être lui-même repris par certains systèmes d'exploitation. Elle est utilisée pour interrompre l'exécution en fin de programme principal ou à un endroit quelconque depuis une procédure quelconque, généralement dans des circonstances exceptionnelles ne permettant plus de poursuivre le traitement.

```
PROGRAM trait
  ...
  CALL sub_prog
  ...
  IF ( ... ) STOP 10    ! interruption exceptionnelle
  ...
  STOP                  ! fin normale
END PROGRAM trait
SUBROUTINE sub_prog
  ...
  IF ( ... ) STOP 12    ! arret exceptionnel
  ...
  RETURN                ! retour normal au programme principal
END SUBROUTINE sub_prog
```

4.7.4 Instruction RETURN

L'instruction RETURN transfère le contrôle de la procédure (sous-programme ou fonction) où elle est située à la procédure l'ayant appelée [10]. Elle n'est pas autorisée dans le programme principal. Elle permet de ménager plusieurs retours de contrôle vers le programme appelant, dans le cas de traitements conditionnels.

L'exemple élémentaire qui suit n'est cependant pas un modèle de programmation structurée et de lisibilité.

```
PROGRAM trait
  ...
  y1 = rac_cub(x1)
  y2 = rac_cub(x2)
  ...
```

10. Ne pas confondre RETURN, qui rend le contrôle à l'appelant, et STOP, qui interrompt l'exécution.

```
END PROGRAM trait
! une fonction racine cubique étendue à tous les réels
FUNCTION rac_cub(x)
  IMPLICIT NONE
  REAL, INTENT(IN) :: x
  REAL             :: rac_cub
  rac_cub = 0.                ! cas où x=0
  IF ( x == 0. ) RETURN       ! retour immédiat au programme principal
  ! la suite n'est vue que pour x non nul
  rac_cub = ABS(x) ** (1./3.) ! calcul de la valeur absolue
  IF ( x > 0 ) RETURN         ! retour immédiat si x>0
  rac_cub = - rac_cub         ! cas x<0 : il faut encore changer le signe
  RETURN                      ! retour au programme principal
END FUNCTION rac_cub
```

4.7.5 Remarques sur les instructions STOP et RETURN

En fortran 77, l'instruction STOP dans le programme principal, de même que l'instruction RETURN dans les sous-programmes et fonctions, étaient obligatoires juste avant END, quand END n'était qu'un délimiteur. En fortran 90, END est aussi un ordre exécutable ; STOP en fin de programme principal ainsi que RETURN en fin de sous-programme ou de fonction sont donc devenus facultatifs.

4.7.6 Branchements dans les entrées-sorties

Certains ordres d'entrée-sorties (*cf.* 5.3, p. 42), notamment OPEN, READ, WRITE[11] possèdent des arguments optionnels (accessibles par mot-clef) de branchement à des instructions étiquetées permettant de traiter des circonstances particulières qu'il est toujours prudent de prévoir :

- en cas d'erreur : ERR=<*étiquette_numérique*> (OPEN, READ, WRITE) ;
- en cas de fin de fichier : END=<*étiquette_numérique*> (READ) ;
- en cas de fin d'enregistrement : EOR=<*étiquette_numérique*> (READ).

```
...
OPEN(...)
READ(..., ERR=900, ...)
...                             ! traitement normal
STOP                            ! arrêt normal
900 CONTINUE                    ! branchement en cas d'erreur de lecture
WRITE(*,*) 'erreur de lecture'
STOP 12                         ! arrêt exceptionnel après erreur de lecture
END PROGRAM ...
```

Pour éviter les branchements sur des étiquettes numériques, on préférera tester si le code de retour IOSTAT des opérations d'entrées-sorties est non nul pour traiter ces cas particuliers à l'aide des structures de contrôle classiques. Avec fortran 2003, les constantes IOSTAT_END et IOSTAT_EOR du module ISO_FORTRAN_ENV (*cf.* 5.3.1 p. 42) ou l'appel des fonctions booléennes IS_IOSTAT_END ou IS_IOSTAT_EOR permettent de distinguer dans ces cas les fins de fichier des fins d'enregistrement.

⇐ ♥

⇐ f2003

11. BACKSPACE, CLOSE, ENDFILE, INQUIRE et REWIND possèdent aussi un argument optionnel de mot-clef ERR.

Chapitre 5

Entrées–sorties, fichiers

Fortran dispose d'outils sophistiqués pour réaliser des transferts d'informations avec des fichiers, mais il n'est pas nécessaire de connaître toutes leurs possibilités pour mettre en œuvre des programmes élémentaires. En sortie comme en entrée, la mise en forme des données peut en effet être effectuée avec d'autres outils (filtres sous unix par exemple) plus adaptés à la manipulation des chaînes de caractères dans ces étapes de pré- ou post-traitement qui n'impliquent pas de calculs lourds. C'est pourquoi, nous aborderons d'abord (5.1) les entrées-sorties standard (interactions avec le terminal, voire des fichiers par redirection sous UNIX) avant de présenter les concepts généraux (5.2) et la syntaxe détaillée (5.3) des instructions d'entrées-sorties. Nous présentons enfin les descripteurs de formats et les règles d'exploration associées dans la section 5.4. L'étude des nombreux exemples présentés dans les dernières sections 5.5 et 5.6 du chapitre devrait éclairer les notions introduites dans les sections 5.2, 5.3 et 5.4.

5.1 Introduction : entrées et sorties standard

Nous avons déjà utilisé, sans les présenter explicitement, les instructions d'affichage (`PRINT`) et de lecture (`READ`) au *format libre*. Ils respectent la syntaxe suivante :

```
PRINT *, <liste d'expressions>
READ *, <liste de variables>
```

où `*` représente le format libre et où les éléments de la liste d'entrée-sortie sont séparés par des virgules.

```
INTEGER :: i, j
REAL :: a
PRINT *, 'donner deux entiers et un réel'      ! affichage au format libre
READ *, i, j, a                                ! lecture au format libre
PRINT *, 'vous avez choisi i = ', i, ' j = ', j, ' a = ', a
```

Dans l'exemple précédent, la liste qui suit `PRINT` comporte seulement des chaînes de caractères constantes et des variables. Mais elle peut aussi comporter des expressions :

```
PRINT *, 'a = ', a, ' a*a = ', a**2, ' sin(a) = ', sin(a)
```

5.1.1 Une instruction d'E/S = un enregistrement

La principale règle régissant la progression par défaut des lignes dans les entrées-sorties veut que (sans option `ADVANCE='no'`, *cf.* 5.3.9, p. 48) :

Chaque instruction de lecture ou d'écriture provoque le passage à la ligne suivante.

En particulier, si la liste est vide, en écriture l'instruction `PRINT *` produit un saut de ligne, de même qu'en lecture l'instruction `READ *` saute une ligne d'entrées qui reste inexploitée.

Cette règle, essentielle pour comprendre en particulier écriture et lecture des tableaux implique que, si on utilise une boucle explicite pour afficher les éléments d'un tableau, on affiche un élément par ligne (affichage en colonne). Pour afficher le tableau en une seule ligne, il faut (*cf.* 5.4.4, p. 52) :

- soit, comme en fortran 77, utiliser une boucle implicite, qui produit « au vol » la liste des éléments pour l'instruction d'écriture,
- soit, de façon plus concise, utiliser la notation globale pour l'ensemble des éléments du tableau à afficher.

5.1.2 Saisie de données au clavier

Les données doivent être saisies au clavier sous une des formes admissibles pour les constantes (*cf.* 2.2.2, p. 15) du type de la variable à affecter. À la conversion implicite près des entiers en réels, une différence de type provoque une erreur d'exécution lors de la lecture.

Lors de la lecture d'un enregistrement, une combinaison quelconque des saisies suivantes au clavier :

- un ou plusieurs espaces ;
- une virgule [1] ;
- un ou plusieurs changements de ligne,

est considérée comme un séparateur de données, permettant de passer à la lecture de la variable suivante.

Au lieu de répéter `<n>` fois une valeur `<val>`, on peut utiliser la notation `<n>*<val>`. Tant que la liste des variables [2] n'est pas pas épuisée, la lecture des données peut se poursuivre sur plusieurs lignes, donc la lecture d'un enregistrement peut correspondre à la saisie de plusieurs lignes.

Mais on peut arrêter la lecture de l'enregistrement avant d'avoir épuisé la liste par la saisie du caractère `/` de fin d'enregistrement. Il est aussi possible de « sauter » une variable au sein de la liste en saisissant deux virgules successivement. Dans ces deux cas, les variables non lues gardent leur valeur antérieure.

Par exemple, pour les jeux d'entrées indiqués à gauche de la flèche ($\Longrightarrow$), le programme suivant affichera ce qui est indiqué à droite.

```
PROGRAM lecture
IMPLICIT NONE
INTEGER :: i=10, j=20, k=30          ! 3 entiers initialisés
WRITE(*,*) 'donner 3 entiers'
READ *, i, j, k
WRITE(*,*) ' i=', i, ' j=', j , ' k=', k
END PROGRAM lecture
```

```
1 2 3          1,2,3          1,   2,        1
                              3              2
                                             3        ⟹   i= 1  j= 2  k= 3

1,,3                         1
                             ,
                             ,
                             3                ⟹   i= 1  j= 20  k= 3

1   2  /       1 2 ,,                         ⟹   i= 1  j= 2  k= 30
```

1. Le séparateur de données virgule « , » est changé en point-virgule « ; » dans le cas où on a choisi la virgule comme séparateur décimal (*cf.* 5.3.2, p. 44).
2. Au sens scalaire du terme, donc par exemple chacun des éléments d'un tableau de réels.

```
3*1
```

```
6*1
```

$\Longrightarrow$

```
 i= 1  j= 1  k= 1
```

Si on fournit plus de données que nécessaire, les données superflues sont perdues : ne pas croire qu'elles sont disponibles pour l'instruction de lecture suivante.

Le programme suivant comporte deux instructions de lecture, et ne peut lire l'entier `m` qu'après un changement d'enregistrement.

```
PROGRAM lecture2
IMPLICIT NONE
INTEGER :: i=10, j=20, k=30, m=40        ! 4 entiers initialisés
WRITE(*,*) 'donner 3 entiers puis 1 entier'
READ *, i, j, k
READ *, m
WRITE(*,*) ' i=', i, ' j=', j, ' k=', k, ' m=', m
END PROGRAM lecture2
```

```
1 2 3
4
```

```
1 2 3 -4 -5
4 -4
```

```
1
2
3
4
```

$\Longrightarrow$

```
 i= 1  j= 2  k= 3 m=4
```

```
1 2 /
4
```

```
1 2 / -3
4 -4
```

```
1 2 ,, ,,
4
```

$\Longrightarrow$

```
 i= 1  j= 2  k= 30 m=4
```

Dans les applications élémentaires et en phase de mise au point, on se contente souvent du format libre pour les entrées et les sorties avec le terminal. S'il est parfois nécessaire de contraindre le format en sortie pour améliorer la lisibilité, même à l'écran, à l'inverse, pour la lecture au clavier, ♥ ⇒ il est conseillé d'utiliser le format libre, sauf à informer explicitement l'utilisateur des contraintes du format de saisie. En revanche, lorsque la quantité des données d'entrée devient importante, il est souvent plus efficace de les saisir dans un fichier avec toutes les possibilités d'un éditeur, puis de faire lire[3] ce fichier par le programme.

5.2 Généralités et terminologie

Dans les échanges de données plus massifs qu'avec le terminal, ou avec des fichiers dont le format peut être imposé par d'autres applications, il est souvent nécessaire de préciser les règles de conversion en entrée ou en sortie grâce à une spécification de format.

```
INTEGER :: k
INTEGER, DIMENSION(100) :: ti  ! tableau de 100 entiers
REAL, DIMENSION(100) :: tr     ! tableau de 100 réels
OPEN(9, FILE='donnees.dat')    ! ouverture du fichier => connecté à l'unité 9
DO k = 1, 100
   READ(9, '(i4, f9.5)') ti(k), tr(k) ! lecture d'un enregistrement
END DO
CLOSE(9)                       ! fermeture du fichier
```

Dans cet exemple, chaque instruction `READ` lit dans le fichier `donnees.dat` un enregistrement constitué d'un entier codé sur 4 chiffres, suivi, sans séparateur, d'un réel sur 9 caractères dont 5 après le point décimal. Chaque ligne comporte donc 13 caractères : par exemple `0096009.37500` pour l'entier `96` suivi du réel `9.375`. La chaîne de caractères `i4, f9.5` constitue le format de lecture, indispensable ici pour séparer l'entier du réel.

3. Sous UNIX, cette lecture peut se faire par simple redirection d'entrée.

L'objectif de ce chapitre est de décrire les différentes instructions permettant les échanges de données et d'introduire la syntaxe des spécificateurs de format.

5.2.1 Fichiers externes et fichiers internes

Fichiers externes et unités logiques

Outre les entrées–sorties standard qui s'opèrent via le terminal, fortran peut contrôler plusieurs flux de données en lecture ou en écriture associés à des *fichiers externes* (external files), hébergés par des périphériques (disques, bandes magnétiques, CD, clef USB, ...).

Ces fichiers externes doivent être connectés à des *unités logiques* (logical unit) à l'aide de l'instruction `OPEN`. L'instruction `CLOSE` permet de fermer le fichier et donc de libérer l'unité logique préalablement connectée. Une unité logique choisie par l'utilisateur est désignée par un entier positif ou nul [4, 5, 6], généralement entre 1 et 100, en évitant 0, 5 et 6, habituellement dédiées aux unités standard.

Noter que les unités logiques ont une portée globale : il est possible de connecter un fichier à une unité logique dans une procédure, d'écrire et de lire dans ce fichier via cette unité dans une autre procédure et de la fermer dans une troisième.

Unités logiques standard

Mais les unités logiques standard d'entrée et de sortie sont préconnectées par défaut, et peuvent être désignées par le symbole `*`.

Sous UNIX, l'entrée standard (clavier) et la sortie standard (écran) sont traditionnellement préconnectées respectivement aux unités logiques 5 et 6. La sortie d'erreur standard est préconnectée par défaut à l'unité logique 0 ; pour écrire sur la sortie d'erreur, il suffirait donc de préciser le numéro d'unité logique 0.

Mais depuis fortran 2003, le module intrinsèque `ISO_FORTRAN_ENV` (*cf.* 5.3.1, p. 42) définit plusieurs constantes nommées caractérisant l'environnement, dont les numéros des unités logiques standard : `input_unit`, `output_unit` et `error_unit`, qui permettent donc de désigner ces unités logiques de façon portable. ⇐ f2003

Fichiers internes

À l'opposé des fichiers externes stockés sur un dispositif périphérique, on parle aussi d'écriture et de lecture sur des *fichiers internes* (internal files) *cf.* 8.4.3 p. 101 lorsque l'on code des données numériques sous forme de chaînes de caractères (écriture) ou inversement que l'on décode des chaînes de caractères représentant des données (lecture). Les fichiers internes ne sont que de simples variables de type chaîne de caractères, explicitement déclarées dans le programme et stockées en mémoire vive. Ils sont notamment utilisés pour générer des formats variables pour des opérations d'entrées-sorties (*cf.* 5.4.5, p. 53).

5.2.2 Notion d'enregistrement

Un *enregistrement* (record) est la généralisation à un fichier de la notion de ligne sur un terminal. Un fichier est une suite d'enregistrements.

Lors d'une opération d'entrée-sortie sur un fichier en accès séquentiel, chaque instruction de lecture (`READ`) ou d'écriture (`WRITE`) provoque par défaut [7] le passage à l'enregistrement suivant.

4. La limite supérieure théorique est bien souvent celle des entiers par défaut : soit $2^{31}-1$ sur processeur 32 bits et $2^{63}-1$ sur processeur 64 bits avec le compilateur `g95` ; mais elle reste de $2^{31}-1$ avec le compilateur XLF d'IBM.

5. Certaines bibliothèques se réservent l'usage d'une plage *fixe* de numéros d'unités logiques, qui ne sont plus, en conséquence, disponibles pour les programmes de l'utilisateur de ces bibliothèques.

6. La lecture ou l'écriture sur une unité logique `n` non connectée force la connexion implicite à un fichier externe dont le nom par défaut est `fort.n` sous UNIX.

7. Il est possible, en fortran 90, de faire exception à cette règle grâce à l'option `ADVANCE=no`.

5.2.3 Fichiers formatés et fichiers non formatés

Les données numériques peuvent être stockées dans des fichiers :

- soit sous la forme de chaînes de caractères (qui seront bien sûr eux-mêmes codés en binaire) : les fichiers sont dits alors *formatés* ou codés (formatted). Les entrées-sorties standard s'effectuent sous forme codée.
- soit directement sous la forme binaire utilisée pour le stockage dans la mémoire de la machine : les fichiers sont dits alors *non-formatés* (unformatted).

Ainsi, lors des entrées-sorties formatées de données numériques sur des fichiers externes, les instructions d'écriture ou de lecture déclenchent en réalité successivement deux opérations : une conversion [8] selon les règles définies par un *format* et un transfert de données vers un périphérique. Plus précisément, suivant qu'il s'agisse d'entrée ou de sortie formatée, elle signifiera :

- soit d'abord une lecture de chaînes de caractères, puis une conversion en représentation interne ;
- soit d'abord une conversion de la représentation interne en chaînes de caractères, puis une écriture de chaînes de caractères.

Si la conversion provoque une erreur à l'exécution, c'est l'ensemble de l'instruction qui échoue, ce qui s'avère particulièrement gênant par exemple pour les entrées interactives, en cas de faute de frappe. Pour fiabiliser les saisies au clavier, on est amené à exploiter les codes de retour de `READ` [9]. La lecture s'effectue alors dans une boucle qui ne s'arrête que lorsqu'il n'y a plus d'erreur.

```
1  PROGRAM robust_lect
2  IMPLICIT NONE
3  INTEGER :: i=-100, ok
4  WRITE(*,*) 'entrer un entier'
5  DO
6      READ(*, *, iostat=ok)  i
7      IF(ok==0)  EXIT
8      WRITE(*,*) 'erreur: recommencez'
9  END DO
10 WRITE(*,*) 'i=', i
11 END PROGRAM robust_lect
```

Choix entre fichier formaté ou non-formaté

Le choix entre les deux types de fichiers pour stocker des données numériques est dicté par les critères suivants :

non-formatés : pas de conversion, donc transfert plus rapide et sans perte de précision, fichiers plus compacts ; mais fichiers non portables [10] et difficiles à lire par d'autres outils [11] ; en

8. Dans le cas d'un fichier interne, c'est la seule opération déclenchée par l'instruction d'entrée-sortie.
9. Cela impose d'utiliser la deuxième syntaxe, plus complète, de `READ`, *cf.* 5.3.4, p. 45.
10. Les données numériques sur plusieurs octets peuvent être enregistrées en mémoire de deux façons, selon l'ordre choisi entre les octets qui représentent la valeur en binaire :
    - dans l'ordre des poids décroissants (octets de poids fort au début) : on parle d'orientation big-endian ou gros-boutiste.
    - dans l'ordre des poids croissants (octets de poids faible au début) : on parle d'orientation little-endian ou petit-boutiste.

Si la majorité les processeurs de PC, en particulier les x86 ont adopté l'orientation litte-endian, les processeurs de serveurs IBM, les PowerPC et les IA64 notamment sont d'orientation big-endian. On notera que ces différences peuvent concerner les entiers, mais aussi les flottants, car la norme IEEE 754 (*cf.* annexe C, p. 156) ne spécifie pas l'orientation des octets, et les caractères multi-octets d'unicode (UTF-16 et UTF-32).

11. Sous UNIX, la commande `od` (octal dump) permet d'afficher sous diverses formes le contenu binaire d'un fichier. Elle permet entre autres d'afficher un aperçu du contenu de fichiers binaires grâce à une ou plusieurs options `-t` qui spécifient comment interpréter les octets (nombre d'octets à grouper et conversion suivant un type), par exemple :
    `od -t d4` pour des entiers en décimal sur 4 octets (32 bits) ;
    `od -t f4` ou `od -t fF` pour des réels sur 4 octets (32 bits) ;
    `od -t f8` ou `od -t fD` pour des réels en double précision sur 8 octets (64 bits).

particulier, dans les fichiers binaires en accès séquentiel, fortran écrit et s'attend à lire, en tête et en fin (c'est moins gênant) de chaque enregistrement, une balise d'enregistrement (record marker), sous forme d'un entier [12] indiquant le nombre d'octets de l'enregistrement : il faut en tenir compte si on accède au fichier au travers de programmes utilisant d'autres langages ;

formatés : lisibilité par d'autres outils (filtres ou simples éditeurs de texte), facilité d'impression, portabilité entre applications et entre machines ; mais fichiers plus volumineux et accès plus lent ; perte de précision possible.

5.2.4 Méthodes d'accès aux données

On distingue deux méthodes d'accès aux enregistrements stockés dans des fichiers :

- l'accès *séquentiel* (sequential), où les enregistrements sont stockés les uns à la suite des autres [13] et où, pour accéder à un enregistrement particulier, il est nécessaire de *lire* tous les enregistrements précédents ; dans ce mode, il n'est pas possible de modifier un enregistrement sans compromettre l'intégrité de tous les suivants ;
- l'accès *direct* (direct), où les enregistrements sont repérés par un numéro qui permet (moyennant la connaissance de la taille, nécessairement fixe, des enregistrements) de calculer la position de l'enregistrement dans le fichier et d'accéder ainsi à un enregistrement particulier (y compris en écriture) sans lire explicitement ceux qui le précèdent, ni affecter les suivants (*cf.* 5.6, p. 56).

Le fortran 2003 a introduit une troisième méthode inspirée du langage C, l'accès stream ou *flot*, combinant des avantages des deux précédents (positionnement possible dans le fichier et taille d'enregistrement non fixe). Le positionnement dans le fichier peut se faire en spécifiant le numéro d'unités de stockage (en général des octets) en commençant à 1 en début de fichier. En dehors de 1, il est prudent d'utiliser `INQUIRE` pour obtenir une position valide dans le fichier. Les fichiers formatés en accès stream ont aussi une structure d'enregistrement, mais les fins d'enregistrement peuvent être spécifiées en écrivant le caractère fourni par la fonction intrinsèque `NEW_LINE` [14]. ⇐ f2003

Choix entre accès direct et accès séquentiel

Le choix entre les méthodes d'accès aux données dépend de leur nature (qui peut déterminer la taille des enregistrements), de leur volume (stockable facilement ou non dans des tableaux en mémoire) et de l'usage qu'on leur prévoit. Ce choix est parfois imposé par les contraintes suivantes :

- les enregistrements d'un fichier à accès direct sont nécessairement de taille fixe ;
- seul l'accès direct permet d'écrire un enregistrement sans avoir à réécrire la totalité du fichier.

Ainsi, les fichiers de bases de données, dont les enregistrements doivent pouvoir être mis à jour de façon individuelle, et dont le volume ne permet pas un stockage facile en mémoire vive, sont stockés dans des fichiers à accès direct.

Parfois, alors que les deux méthodes sont envisageables, des considérations de performances (même en lecture seule) peuvent amener à préférer l'accès direct, si par exemple on doit très souvent accéder à des données non contigües, de façon aléatoire. Supposons par exemple qu'une fonction soit échantillonnée dans le plan au sein d'un fichier selon une grille en coordonnées rectangulaires et que l'on souhaite la rééchantillonner selon des coordonnées polaires ; lorsque l'on calculera les valeurs de la fonction en suivant un cercle, on devra accéder à des données certainement non contigües dans le fichier initial, qu'il sera donc préférable de lire en accès direct ; il sera donc préférable de créer le fichier initial en accès direct.

12. Cet entier peut être codé sur 64 bits sur les processeurs 64 bits, afin de permettre des enregistrements binaires de plus de 2 Go. Suite à des protestations d'utilisateurs, le compilateur `gfortran` est revenu à un codage sur 32 bits par défaut, mais l'option `-frecord-marker=<`*`length`*`>` permet de choisir 32 ou 64 bits (*cf.* F.5.6, p. 174).
13. Comme sur une bande magnétique qui est un support physique à accès séquentiel.
14. La fonction intrinsèque `NEW_LINE` prend pour argument un caractère quelconque et rend le caractère de fin de ligne de même sous-type. C'est l'équivalent du `\n` du langage C.

5.2.5 Notion de liste d'entrée-sortie

Dans une instruction d'entrée-sortie, les informations échangées sont spécifiées par une *liste d'entrée-sortie* (i/o-list), qui est une liste d'expressions pour une instruction d'écriture, mais doit être une liste de variables pour une instruction de lecture. Si un élément de la liste est un tableau, il est équivalent à la liste des éléments du tableau [15] ; s'il s'agit d'une structure, elle doit être interprétée comme la liste des champs de la structure. Enfin, la liste d'entrée-sortie peut comporter des *boucles implicites* (aussi utilisées comme constructeurs de tableau, *cf.* 7.1.3, p. 82), éventuellement imbriquées.

Dans l'exemple suivant, chacune des trois instructions affiche le tableau `t` de la même façon sur une ligne.

```
INTEGER, DIMENSION(3) :: t = (/ 1, 2, 3 /)     ! tableau initialisé
INTEGER :: i
WRITE(*, '(3i4)') t                            ! affichage global
WRITE(*, '(3i4)') t(1), t(2), t(3)             ! liste exhaustive
WRITE(*, '(3i4)') (t(i), i=1, 3)               ! boucle implicite
```

5.3 Instructions d'entrées-sorties

Une opération de transfert de données requiert en général une instruction `OPEN` de connexion entre unité logique et fichier, des instructions d'échange `READ` ou `WRITE`, et enfin une instruction `CLOSE` de libération de l'unité logique. Les instructions d'entrées-sorties `OPEN`, `READ`, `WRITE`, `PRINT` et `INQUIRE` se partagent des arguments obligatoires ou optionnels accessibles par mot-clef dont on va décrire les plus usités.

f2003 5.3.1 Le module intrinsèque `ISO_FORTRAN_ENV`

Depuis fortran 2003, le module intrinsèque `ISO_FORTRAN_ENV` définit des constantes nommées qui permettent de désigner de façon portable des paramètres concernant les entrées sorties :

- Les numéros des unités logiques préconnectées aux flux standard :
  - `INPUT_UNIT`, qui désigne l'entrée standard (descripteur 0 sous unix permettant des redirections d'entrée via `< fichier`) ;
  - `OUTPUT_UNIT`, qui désigne la sortie standard (descripteur 1 sous unix permettant des redirections de sortie via `> fichier`) ;
  - `ERROR_UNIT`, qui désigne la sortie d'erreur standard (descripteur 2 sous unix permettant des redirections d'erreur standard via `2> fichier`).

  Sous UNIX, ces numéros valent en général 5, 6 et 0 respectivement. Si les unités d'entrée et de sortie standard peuvent être désignées par `*` de façon plus rapide que `INPUT_UNIT` et `OUTPUT_UNIT`, l'utilisation de `ERROR_UNIT` est nécessaire pour désigner la sortie d'erreur de façon indépendante du compilateur.

```
1  PROGRAM unites_standard !  emploi du module ISO_FORTRAN_ENV (fortran 2003)
2  USE,INTRINSIC :: ISO_FORTRAN_ENV, ONLY: error_unit, input_unit, output_unit
3  IMPLICIT NONE
4  INTEGER :: i
5  WRITE(*,*) "entrer un entier"
6  ! lecture sur l'entrée standard: redirection par < fichier
7  READ(input_unit, *) i
8  WRITE(*,*) "i vaut ", i
9  ! écriture sur la sortie standard: redirection par > fichier
10 WRITE(output_unit,*) "message sur la sortie standard"
11 ! écriture sur la sortie standard d'erreur : redirection par 2> fichier
12 WRITE(error_unit,*) "message d'erreur"
13 END PROGRAM unites_standard
```

15. Le tableau est transformé en liste en faisant varier d'abord le premier indice, puis le second, ... (*cf.* 7.1.2, p. 81).

- `FILE_STORAGE_UNIT` qui désigne l'unité de mesure en bits (en général 8) des tailles d'enregistrement (`RECL`) dans les opérations `OPEN` (*cf.* 5.3.2 p. 43) et `INQUIRE` (*cf.* 5.3.7 p. 47).
- Les entiers suivants rendus par `IOSTAT=` dans les cas d'échec d'opérations d'entrées/sorties (*cf.* 5.3.2, p. 43 pour `OPEN`) :
  - `IOSTAT_END` qui désigne la fin de fichier ;
  - `IOSTAT_EOR` qui désigne la fin d'enregistrement (End Of Record) ;

  La valeur du code de retour `IOSTAT` peut être comparée à ces paramètres pour effectuer des traitements spécifiques au lieu d'avoir recours aux arguments optionnels `END=` ou `EOR=` (*cf.* 4.7.6 p. 35), qui, eux, nécessitent le branchement sur une étiquette numérique.

Noter que fortran 2003 fournit aussi les deux fonctions `IS_IOSTAT_END` et `IS_IOSTAT_EOR` qui attendent pour argument la valeur du code de retour `IOSTAT` et rendent un booléen vrai en cas de fin de fichier ou de fin d'enregistrement. Ces fonctions booléennes testent donc l'égalité à `IOSTAT_END` ou `IOSTAT_EOR`.

5.3.2 OPEN

L'instruction `OPEN` permet de connecter un fichier à une unité logique.

```
OPEN ([UNIT=]<entier>, FILE=<nom_fichier>, ERR=<étiquette>, &
  IOSTAT=<iostat>, ...)
```

Dans cette syntaxe, seul le premier argument (numéro d'unité logique) est positionnel et tous les paramètres sont accessibles par mot-clef. Seuls les paramètres désignés par les mots-clefs `UNIT` et `FILE` sont presque toujours obligatoires, la plupart des autres paramètres sont optionnels avec des valeurs par défaut indiquées ci-dessous.

- `[UNIT=]<entier>` désigne le numéro de l'unité logique ; cette unité logique ne doit pas être déjà connectée [16] ;
- `NEWUNIT=<nu>` renvoie un numéro d'unité logique non utilisé dans la variable entière `<nu>`. Cet argument, introduit par fortran 2008, permet d'éviter d'avoir à prendre en charge soi-même (*cf.* 5.3.7, p. 47) la recherche d'un numéro libre. Pour éviter toute collision avec des numéros (positifs) attribués avec `UNIT=`, le numéro rendu est négatif (et différent de `-1`). ⇐ f2008
- `FILE=<nom_fichier>` désigne le nom du fichier à connecter, sous forme d'une chaîne de caractères (constante ou variable) ;
- `FORM=<fmt>` où `<fmt>` est une chaîne de caractères égale à `'FORMATTED'` (valeur par défaut) ou `'UNFORMATTED'` pour un fichier non formaté ;
- `ERR=<étiquette>` désigne une étiquette numérique permettant un branchement [17] en cas d'erreur ;
- `IOSTAT=<iostat>` est un paramètre de sortie qui rend une variable entière nulle si l'opération s'est déroulée sans erreur, ou sinon une valeur précisant le type d'erreur rencontrée : le module `ISO_FORTRAN_ENV` définit les constantes nommées pour les fins de fichiers `IOSTAT_END` et d'enregistrement `IOSTAT-EOR` (*cf.* 5.3.1, p.42) ;
- `STATUS=<status>` désigne l'état du fichier à ouvrir et peut valoir :
  - `'OLD'` dans le cas d'un fichier préexistant
  - `'NEW'` dans le cas où le fichier n'existe pas
  - `'REPLACE'` qui permet le remplacement ou la création suivant que le fichier existe ou non
  - `'UNKNOWN'` dans le cas où on ne sait pas a priori si le fichier existe
  - `'SCRATCH'` pour un fichier temporaire qui sera détruit lors de l'instruction `CLOSE`
- `ACCESS=<méthode>` spécifie si le fichier est en accès séquentiel `'SEQUENTIAL'` (par défaut), en accès direct `'DIRECT'` ou (en fortran 2003 seulement) en accès stream `'STREAM'` ; ⇐ f2003

16. On peut vérifier qu'un numéro d'unité logique est disponible en utilisant l'instruction `INQUIRE` (*cf.* 5.3.7, p. 47).
17. On évitera ce branchement en testant si le code de retour `IOSTAT` est non nul.

- `ACTION=<mode>` indique les opérations possibles sur le fichier (elles supposent que l'utilisateur possède des droits d'accès suffisants sur le fichier)
  - `'READ'` si ouvert en lecture seule
  - `'WRITE'` si ouvert en écriture seule
  - `'READWRITE'` (défaut) si ouvert en lecture et écriture
- `POSITION=<pos>` précise la position à l'ouverture d'un fichier en accès séquentiel :
  - `'APPEND'` positionne à la fin du dernier enregistrement avant la marque de fin de fichier et permet d'ajouter des informations à la fin d'un fichier existant
  - `'REWIND'` positionne en début de fichier
  - `'ASIS'` ouvre le fichier à la position courante s'il est déjà connecté, mais dépend du processeur sinon
- `RECL=<entier>` indique la longueur des enregistrements pour un fichier à accès direct (paramètre obligatoire dans ce cas) ou la longueur maximale des enregistrements pour un fichier séquentiel (paramètre optionnel dans ce cas, la valeur par défaut dépendant du compilateur) ; `RECL` s'exprime en nombre de caractères pour les fichiers formatés, mais dans une unité qui peut être l'octet ou le mot de 32 bits, selon le compilateur et ses options pour les fichiers non-formatés ;
- `BLANK` est une chaîne de caractères spécifiant l'interprétation des blancs [18] dans les lectures de données numériques : si elle vaut `'null'` (par défaut), les blancs sont ignorés, si elle vaut `'zero'`, ils sont interprétés comme des zéros ;
- `DELIM` spécifie le délimiteur des chaînes de caractères en écriture, qui peut être `'none'` (aucun, par défaut), `'quote'` «"», ou `'apostrophe'` «'».
- f2003 ⇒ `DECIMAL` spécifie le séparateur entre la partie entière et la partie fractionnaire des nombres flottants. Par défaut, c'est le point, mais la virgule peut lui être substituée par `DECIMAL='comma'`. Pour rétablir le point, il faut préciser `DECIMAL='point'`.

  Noter que ces conventions peuvent aussi être modifiées dans les ordres de lecture (*cf.* 5.3.4, p. 45) et d'écriture (*cf.* 5.3.5, p. 46), ainsi qu'au travers des formats `DC` et `DP` (*cf.* 5.4.2, p. 51).
- f2003 ⇒ `ENCODING` permet de préciser le codage des chaînes de caractères dans le fichier formaté à ouvrir. Les fichiers texte au codage `UTF-8` d'Unicode sont pris en charge par le standard 2003 grâce au paramètre optionnel `ENCODING='UTF-8'` de `OPEN`. Il est prudent de lire de tels fichiers dans des variables chaîne de caractères dont la variante de type, par exemple `'ISO_10646'` (*cf.* 8.1.3, p. 97), permet de représenter les caractères d'Unicode.

Exemple de lecture d'un fichier de 6 caractères codés en `UTF-8` et d'écriture en code `ISO-8859-1`

```
1  PROGRAM io_utf8_latin1
2  IMPLICIT NONE
3  ! 9 oct 2012  fichier source en iso-latin1
4  ! avec NAG Fortran Compiler Release 5.3(854)
5  ! les kind des chaînes (4 en utf-32 et 1 en default)
6  INTEGER, PARAMETER :: utf32=SELECTED_CHAR_KIND('ISO_10646') ! UCS_4 = UTF-32
7  INTEGER, PARAMETER :: def_char=SELECTED_CHAR_KIND('DEFAULT') ! Default=latin1
8  CHARACTER(LEN=6, KIND=def_char) :: latin
9  CHARACTER(LEN=6, KIND=utf32) :: lu_utf32 ! chaîne UTF-32
10 ! lecture sur fichier externe codé UTF-8
11 OPEN(UNIT=10, FILE='fichier-utf8.txt', ENCODING='utf-8')
12 READ(10, *) lu_utf32(1:)
13 CLOSE(10)
14 latin = lu_utf32 ! transcodage UTF32-> default =latin1
15 WRITE(*,*) "latin1:", latin ! affichage iso-latin1
16 ! écriture sur fichier externe en latin1
17 OPEN(UNIT=10, FILE='fichier-latin1.txt', ENCODING='DEFAULT') ! latin1
```

18. Il est aussi possible de faire ces choix avec les formats `BN` et `BZ` dans les instructions de lecture (*cf.* 5.4.2, p. 50).

```
18 WRITE(10,'(a)') latin
19 CLOSE(10)
20 END PROGRAM io_utf8_latin1
```

Le texte du fichier codé `UTF-8` est lu dans des variables de type chaînes `UTF-32`, puis converti [19] dans des variables de type `ISO-8859-1` et enfin écrit dans un fichier au codage par défaut.

Exemples

Pour connecter le fichier formaté `lect.dat` à l'unité `10` en accès séquentiel, on peut se contenter de l'instruction :

```
OPEN(UNIT=10, FILE='lect.dat')
```

Il est aussi possible de préciser :

```
OPEN(UNIT=10, FILE='lect.dat', FORM='formatted', ACCESS='sequential')
```

Mais il est prudent de prévoir une gestion des erreurs avec par exemple :

```
INTEGER :: ok ! status de retour
OPEN(UNIT=10, FILE='lect.dat', IOSTAT=ok)
IF( ok /= 0 ) THEN
   PRINT *, 'erreur fichier lect.dat'
   STOP       ! arrêt à défaut de meilleure solution
END IF
```

5.3.3 CLOSE

L'instruction `CLOSE` permet de déconnecter une unité logique d'un fichier [20]. Elle libère l'unité logique pour une nouvelle connexion.

```
CLOSE ([UNIT=]<entier>, ERR=<étiquette>,  &
   IOSTAT=<iostat>, STATUS=<stat>)
```

Les arguments de `CLOSE` sont tous optionnels sauf l'unité logique et ont la même signification que dans l'instruction `OPEN`, sauf `STATUS=<status>` qui désigne ici le devenir du fichier après déconnexion et peut valoir :

- `'KEEP'` (par défaut [21]) pour conserver un fichier préexistant ;
- `'DELETE'` pour détruire le fichier à la déconnexion.

5.3.4 READ

L'instruction `READ` possède deux variantes :

- dans la première, l'unité logique n'est pas précisée et il s'agit de l'entrée standard ; elle est le symétrique en entrée de `PRINT` (*cf.* 5.1, p. 36) ;
- la seconde respecte la syntaxe suivante, où seul le premier argument (l'unité logique) est positionnel :

19. Noter que dans la norme fortran 2003, seul le transcodage depuis et vers le code par défaut est requis : ici, le défaut étant `ISO-8859-1`, il permet de transcoder des chaînes avec des caractères non-ascii en `UTF-32` vers du latin1.
20. L'arrêt d'un programme provoque implicitement un `CLOSE` sur toutes les unités connectées par le programme
21. Sauf pour les fichiers ouverts avec `STATUS='scratch'`, pour lesquels la valeur par défaut est `DELETE`.

```
READ ([UNIT=]<entier>, ERR=<étiquette>, END=<étiquette>, &
        IOSTAT=<iostat>, ...) liste d'entrées
```

Selon le type de fichier et d'accès, d'autres arguments deviennent obligatoires ou optionnels [22] :

- aucun autre argument pour les fichiers séquentiels non formatés ;
- l'argument `FMT=<format>` spécifiant le format est obligatoire et les arguments `ADVANCE='no'` (*cf.* 5.3.9, p. 48), `SIZE=<taille>` (qui donne le nombre de caractères effectivement lus par l'instruction `READ`), `EOR=<étiquette>` (branchement si on rencontre la fin de l'enregistrement en cas de lecture avec `ADVANCE='no'`) sont optionnels pour les fichiers séquentiels formatés ;
- `END=<étiquette>` branchement si on rencontre la fin de fichier ;
- l'argument `REC=<numéro_d_enregistrement>` est obligatoire pour les fichiers en accès direct non formatés ; les arguments `REC=<numéro_d_enregistrement>` et `FMT=<format>` sont obligatoires pour les fichiers en accès direct formatés ;
- f2003 ⇒ l'argument optionnel d'entrée `POS=<position>` permet de choisir la position en unités de stokage dans les fichiers en accès stream.

f2003 ⇒ Les branchements conditionnels `EOR=` et `END=` sont déconseillés en fortran 2003. On préfère utiliser le module intrinsèque `ISO_FORTRAN_ENV` (*cf.* 5.3.1, p. 42) avec les fonctions booléennes `IS_IOSTAT_END` ou `IS_IOSTAT_EOR` pour tester si le code de retour `IOSTAT` vaut `IOSTAT_EOR` ou `IOSTAT_END`.

5.3.5 WRITE

L'instruction `WRITE` suit la syntaxe générale

```
WRITE ([UNIT=]<entier>, ERR=<étiquette>, &
        IOSTAT=<iostat>, ...) liste_de_sortie
```

où seul le premier paramètre est positionnel et à laquelle, il faut ajouter, suivant le type de fichier et d'accès, d'autres paramètres obligatoires ou optionnels (*cf.* 5.4.4, p. 52) :

- aucun autre argument pour les fichiers séquentiels non formatés ;
- l'argument `FMT=<format>` spécifiant le format est obligatoire et l'argument `ADVANCE='no'`, est optionnel pour les fichiers séquentiels formatés ;
- l'argument `REC=<numéro_d_enregistrement>` est obligatoire pour les fichiers en accès direct non formatés ;
- les arguments `REC=<numéro_d_enregistrement>` et `FMT=<format>` sont obligatoires pour les fichiers en accès direct formatés ;
- f2003 ⇒ l'argument optionnel `POS=<position>` permet de choisir la position en unités de stokage dans les fichiers en accès stream.

5.3.6 PRINT

L'instruction `PRINT <fmt> [, <liste>]` permet d'écrire en mode formaté sur la sortie standard la liste optionnelle des variables ou constantes indiquées après la virgule ; elle est équivalente à `WRITE(*, <fmt>) [, <liste>]`.

22. avec un sens identique à celui éventuellement donné pour l'instruction `OPEN`

5.3.7 INQUIRE

L'instruction INQUIRE permet de s'enquérir de nombreux paramètres caractérisant un flux d'entrée–sortie, notamment : si une unité logique est libre OPENED, si un fichier existe EXIST, quelle est la taille d'un enregistrement binaire IOLENTGH, la position dans un fichier ouvert en accès stream, usages illustrés par les exemples qui suivent.

Elle permet aussi de restituer les paramètres avec lesquels s'est effectuée une ouverture de fichier : ACCESS, ACTION, DECIMAL, DIRECT, FORM, FORMATTED, NUMBER, READ, READWRITE, STREAM, SEQUENTIAL, UNFORMATTED, WRITE ; ces mots-clefs désignent des variables de sortie de type entier ou chaîne de caractère ('YES' ou 'NO' où on aurait pu attendre des booléens).

Le flux interrogé peut être défini :

- soit par un nom de fichier (seule solution si le fichier n'a pas été ouvert) ;
- soit par une unité logique ;
- soit par une liste de sortie.

Par exemple, avant de connecter un fichier à une unité logique via l'instruction OPEN, on peut rechercher un numéro d'unité logique disponible [23] et s'assurer que le fichier existe.

```
LOGICAL :: pris, present
INTEGER :: n = 1, nmax = 99
DO n = 1, nmax
  INQUIRE(UNIT=n, OPENED=pris)
  IF (.not. pris) EXIT                    ! l'unité n est libre
END DO
INQUIRE(FILE='fichier', EXIST=present)
IF(present) THEN                          ! le fichier existe
  OPEN(UNIT=n, FILE='fichier', ...)
  ...
ELSE
  ...
END IF
```

Le paramètre de sortie ENCODING du standard 2003 permet d'obtenir le codage des chaînes de caractères dans un fichier formaté connecté à une unité logique. On peut ainsi détecter le codage 'UTF-8' d'Unicode mais 'UNKNOWN' sera rendu si la détection n'est pas possible. ⇐ f2003

D'autre part, l'instruction INQUIRE permet d'évaluer la taille de l'enregistrement que créerait l'écriture non-formatée d'une liste [24]. Elle permet ainsi de connaître la longueur d'un enregistrement d'un fichier non-formaté en accès direct avant de le connecter à une unité logique (*cf.* 5.6.2, p. 56), de façon à rendre l'instruction d'ouverture du fichier indépendante de la machine.

```
INTEGER :: long, iunit
INQUIRE(IOLENGTH=long) <liste d'expressions>
...
OPEN(iunit, FILE=..., ACCESS='direct', FORM='unformatted', RECL=long)
...
```

Enfin, pour les fichiers en accès stream, INQUIRE permet de s'enquérir de la position courante dans le fichier exprimée en unités de stockage qui sont souvent des octets. Dans le cas d'un fichier non formaté, les décalages peuvent être calculés avec INQUIRE(IOLENGTH=variable).

```
INTEGER :: iunit, positionx
REAL :: x, y
```

23. En fortran 2008, on préfèrera utiliser l'argument NEWUNIT de OPEN (*cf.* 5.3.2, p. 43).
24. Cette taille dépend en général de la machine.

```
OPEN(iunit, FILE=..., ACCESS='stream', FORM='unformatted')
...
INQUIRE(iunit, POS = positionx) ! POS = argument de sortie (prochaine I/O)
READ(iunit) x  ! lecture de x
...
WRITE(iunit, POS = positionx) y ! POS = argument d'entrée
! écriture de y à la place de x
...
```

♠ 5.3.8 Instructions de positionnement dans les fichiers

Il existe enfin des instructions permettant, sans effectuer de transfert de données, de modifier la position courante dans un fichier d'accès séquentiel :

- `REWIND ([UNIT=]<`*entier*`> [, ERR=<`*étiquette*`>] [, IOSTAT=<`*iostat*`>])` positionne en début de fichier ;
- `BACKSPACE ([UNIT=]<`*entier*`> [, ERR=<`*étiquette*`>] [, IOSTAT=<`*iostat*`>])` positionne avant l'enregistrement courant si on est dans un enregistrement ou avant le précédent enregistrement si on est entre deux enregistrements (cette instruction permet la relecture d'un enregistrement) ;
- `ENDFILE ([UNIT=]<`*entier*`> [, ERR=<`*étiquette*`>] [, IOSTAT=<`*iostat*`>])` écrit [25] une marque de fin de fichier à la position courante et positionne juste après cette marque de fin de fichier.

5.3.9 Entrées-sorties sans avancement automatique

Le paramètre `ADVANCE='no'` permet de ne pas avancer automatiquement à l'enregistrement suivant à chaque instruction d'entrée-sortie sur un fichier en accès séquentiel et donc de transférer des parties d'enregistrement, c'est-à-dire de lignes pour les interactions avec le terminal [26].

Par exemple, pour maintenir le curseur juste après une invite à saisir une donnée, on écrira :

```
WRITE(*, '(a)', ADVANCE='no') 'entrer un entier : '
READ(*, *) i
```

De même, pour traiter une saisie incomplète au clavier, on pourra tester si `IOSTAT` vaut `IOSTAT_EOR` et utiliser le paramètre de retour `SIZE` :

```
USE iso_fortran_env, only: IOSTAT_EOR
IMPLICIT NONE
INTEGER :: nlus=4, ok
CHARACTER(len=4) :: chaine='++++'
WRITE(*, '(a)') 'entrer quatre caractères'
READ(*, '(a4)', advance ='no', size=nlus, iostat=ok) chaine(1:4)
WRITE(*, *) nlus, ' caractères lus'
IF (ok == IOSTAT_EOR) THEN
  WRITE(*,*) 'saisie tronquée :', chaine(1:nlus)
ELSE IF(ok /=0) THEN
  WRITE(*,*) "autre erreur de lecture"
ELSE
  WRITE(*,*) "OK, chaine lue :", chaine
END IF
```

25. Ne pas croire que `ENDFILE` est une simple instruction de positionnement en fin de fichier. Ce positionnement est possible à l'ouverture du fichier via l'option `POSITION='APPEND'` de l'instruction `OPEN` (*cf.* 5.3.2).

26. C'est le comportement par défaut en langage C, où les changements de ligne doivent être explicitement spécifiés par des `\n`. Il peut être combiné avec l'accès `STREAM` du fortran 2003.

5.4 Descripteurs de format

Les règles de conversion entre représentation interne des données et chaîne de caractères des fichiers formatés sont déterminées par le format de lecture ou d'écriture. Le format peut être spécifié sous une des formes suivantes :

- le format libre, désigné par `*` ;
- une chaîne de caractères constante ou variable, décrivant le format entre parenthèses ;
- une étiquette numérique (entier positif) faisant référence à une instruction dite de format.

Le format libre est utilisé pour les échanges avec le terminal, en phase de test ou pour accéder à des fichiers manipulés toujours sur la même machine, car il n'est pas portable. Mais, pour des échanges de données plus massifs avec des fichiers ou de données dont le format peut être imposé par des applications externes, il est souvent nécessaire de préciser les règles de conversion en entrée ou en sortie grâce à une spécification de format. La dernière forme, avec étiquette numérique d'instruction est préférée lorsqu'une même spécification de format est utilisée plusieurs fois dans une unité de programme ; il est alors d'usage de regrouper les instructions de format en fin de programme et de leur réserver une plage d'étiquettes numériques particulière.

```
CHARACTER(LEN=4) :: fmt1, fmt2
INTEGER :: j = 12
WRITE(*, *) 'bonjour'             ! format libre
WRITE(*, '("bonjour")')
WRITE(*, '(a7)') 'bonjour'        ! chaîne de caractères constante
fmt1='(a7)'
WRITE(*, fmt1) 'bonjour'          ! chaîne de caractères variable
WRITE(*, 1000) 'bonjour'          ! étiquette numérique
1000 FORMAT(a7)
!
WRITE(*, *) j                     ! format libre
WRITE(*, '(i)') j                 ! chaîne de caractères constante
WRITE(*, '(i2)') j
fmt2='(i2)'
WRITE(*, fmt2) j                  ! chaîne de caractères variable
fmt2='(i4)'
WRITE(*, fmt2) j                  ! chaîne de caractères variable
WRITE(*, 1100) j                  ! étiquette numérique
1100 FORMAT(i2)
```

Comme le format est une expression de type chaîne de caractères, il peut être défini par une variable évaluée lors de l'exécution : cette méthode sera présentée à la sous-section 5.4.5, p. 53.

On distingue les *descripteurs actifs*, qui spécifient le mode de conversion d'une donnée en chaîne de caractères ou réciproquement (a priori un descripteur actif par élément de la liste d'entrée-sortie), les chaînes de caractères et les *descripteurs de contrôle*, qui, entre autres, règlent le positionnement, gèrent les espaces et les changements d'enregistrement.

5.4.1 Descripteurs actifs

D'une façon générale, le premier paramètre (entier), n, détermine le nombre *total* de caractères occupés par la donnée codée, c'est-à-dire avant conversion en lecture ou après conversion en écriture. La signification de l'éventuel deuxième paramètre dépend du type de donnée à convertir.

Si le champ est plus large que nécessaire, la donnée codée est justifiée à droite ; en écriture, le champ est complété par des blancs à gauche. Si la largeur n du champ de sortie est insuffisante compte tenu de tous les caractères requis (signe, exposant éventuel avec son signe, ...), la conversion en sortie est impossible et provoque l'écriture de n signes * [27]. ⇐⚠

27. Ce comportement est contraire à celui du langage C, qui étendrait alors le champ occupé pour convertir la donnée.

- entiers (en sortie, *p* précise le nombre minimal de caractères de chiffres (hors signe et blancs), quitte à compléter par des zéros à gauche) :
  - I*n*[.*p*] en base 10
- réels (*p* spécifie le nombre de chiffres après la virgule) :
  - F*n*.*p* en notation décimale
  - en virgule flottante :
    - mantisse (<1) plus exposant : E*n*.*p*
    - mantisse (<1) plus **e**xposant avec *q* chiffres : E*n*.*p*[e*q*]
    - notation **s**cientifique (mantisse entre 1 et 10) : ES*n*.*p*
    - notation **i**ngénieur (mantisse entre 1 et 1000, exposant multiple de 3) : EN*n*.*p*
- les nombres complexes sont traités comme un couple de nombres réels : ils nécessitent donc deux descripteurs de format réels [28]
- booléens : L*n*
- chaîne de caractères :
  - A*n*, où un mauvais choix de *n* peut provoquer en écriture une troncature à droite si la chaîne est plus longue ou l'adjonction de blancs à gauche si elle est trop courte
  - A où la largeur s'adapte au nombre effectif de caractères à lire ou écrire

Pour une analyse plus bas niveau, les descripteurs B, O et Z (voir les notations associées pour les
f2008 ⇒ constantes, 2.2.2, p. 15) sont utilisables pour les entiers et, en fortran 2008, les réels :

- B*n*[.*p*] en **b**inaire
- O*n*[.*p*] en **o**ctal
- Z*n*[.*p*] en hexadécimal

Enfin, un descripteur qualifié de général permet la conversion de tous les types de données : G*n*.*p*. Dans le cas des réels, il assure une conversion en virgule fixe (format F*n*.*p*) si possible (si la valeur absolue de l'exposant n'est pas trop grande) ou sinon en virgule flottante (format E*n*.*p*).

f95 ⇒ Enfin, on peut demander que la largeur du champ s'ajuste à l'exécution de façon à n'utiliser que le minimum de caractères nécessaires en spécifiant une largeur nulle pour les spécificateurs I, B, O, Z et F, en sortie seulement [29]. Cette possibilité a été étendue au format G en fortran 2008,
f2008 ⇒ avec G0 pour tous les types intrinsèques et G0.*p* pour les réels.

5.4.2 Descripteurs de contrôle

- X provoque l'écriture d'un blanc en sortie et le saut d'un caractère en entrée
- / provoque le changement d'enregistrement (c'est-à-dire le changement de ligne)
- Tabulations (noter que T et TL permettent de changer l'ordre de présentation des données
⚠⇒ en sortie, quitte à écraser alors que TR conserve l'ordre de la liste ; en entrée T et TL peuvent provoquer des doubles lectures !)
  - T*n* positionne le « pointeur » sur le n^{e} caractère de l'enregistrement
  - TL*n* (**t**ab **l**eft) déplace le « pointeur » de *n* caractères vers la gauche
  - TR*n* (**t**ab **r**ight) déplace le « pointeur » de *n* caractères vers la droite ; TR*n* est équivalent à *n*X.
- Interprétation des blancs en lecture [30]

28. Noter que si on souhaite présenter les nombres complexes entre parenthèses et avec parties réelle et imaginaire séparées par une virgule, comme dans les constantes complexes, il faut prendre en charge explicitement cette présentation en adjoignant les caractères attendus dans le format.

29. Ce format de largeur minimale spécifiée par n=0 est comparable au format par défaut du langage C, où l'on ne précise pas la largeur du champ.

30. Bien noter que les formats BN et BZ ne contrôlent pas blancs et zéros en écriture, qui dépendent du compilateur. L'interprétation des blancs en lecture peut aussi être spécifiée de façon plus générale via le paramètre optionnel BLANK= de l'instruction OPEN (*cf.* 5.3.2, p. 44).

  - `BN` ig**n**ore les **b**lancs (internes ou finaux) en lecture de champs numériques
  - `BZ` considère les **b**lancs (internes ou finaux) comme des **z**éros en lecture de champs numériques
- Signe plus optionnel : les trois options qui suivent contrôlent l'écriture éventuelle d'un signe `+` devant les quantités positives ; si une option est spécifiée, elle reste en vigueur tant qu'une autre option ne vient pas la contredire, pour toutes les écritures qui suivent dans la même instruction.
  - `SP` (**s**ign **p**rint) force l'écriture du signe `+` devant les quantités positives
  - `SS` (**s**ign **s**uppress) supprime l'écriture du signe `+` devant les quantités positives
  - `S` rétablit l'option par défaut dépendant du processeur[31] pour l'écriture du signe `+` devant les quantités positives
- `:` interrompt la prise en compte des descripteurs passifs lorsque la liste est épuisée par les descripteurs actifs du format ♠
- *k*`P` descripteur de changement d'échelle, applicable seulement aux réels et complexes, et qui *modifie les valeurs des réels lus ou écrits sans exposant*[32], mais simplement la forme de ceux écrits avec exposant. Il s'applique à toutes les conversions via les descripteurs qui le suivent dans la même instruction, et peut être modifié par un descripteur *n*`P` ou annulé par `0P`. ♠
  - en entrée, *k*`P` divise par le facteur 10^k les valeurs lues *sans exposant*[33], permettant par exemple une saisie plus rapide d'une série de valeurs très faibles ;
  - en sortie, c'est le format qui impose la présence ou l'absence d'un exposant (sauf pour le format `G`) et l'effet de *k*`P` appliqué à un réel dépend du format de conversion :
    - en format `F`, il multiplie par 10^k ;
    - en format `E`, la mantisse est multipliée par 10^k alors que l'exposant est diminué de *k*, la valeur restant inchangée ;
    - en format `G`, l'effet dépend du choix entre `E` et `F` suivant la valeur ;
    - en format `EN` et `ES`, le facteur d'échelle est sans effet.
- `DC` (**d**ecimal **c**omma) ou `DP` (**d**ecimal **p**oint) spécifient le séparateur décimal : par défaut c'est un point, mais ces spécificateurs permettent de le changer et la modification perdure jusqu'au prochain descripteur `dp` ou `dc`. Ce choix[34] peut aussi être fait au niveau des instructions `OPEN` (*cf.* 5.3.2, p. 44), `READ` (*cf.* 5.3.4, p. 45), et `WRITE` (*cf.* 5.3.5, p. 46), selon la syntaxe `decimal='point'` ou `decimal='comma'`. Dans le cas où le séparateur décimal est la virgule, en entrée, le séparateur entre valeurs passe de la virgule au point-virgule « ; » (*cf.* 5.1.2, p. 37). ⇐ f2003

Exemple : à la saisie `1,23;5,67e8`, le programme suivant

```
PROGRAM decimal_comma
IMPLICIT NONE
REAL :: r,t
WRITE(*,*) 'entrer 2 réels séparés par ; (notation décimale avec virgule)'
READ(*, *, decimal='comma') r, t
WRITE(*, '("avec virgule", dc, 2e12.4)') r, t
WRITE(*, '("avec point  ", dp, 2e12.4)') r, t
END PROGRAM decimal_comma
```

affiche :

```
avec virgule  0,1230E+01  0,5670E+09
avec point    0.1230E+01  0.5670E+09
```

31. Pour écrire des fichiers portables, on ne laissera donc pas le choix au processeur, et on spécifiera systématiquement soit `SP`, soit `SS`.
32. On évitera l'emploi du descripteur *k*`P` à cause de cette modification de valeur peu explicite lors de la conversion.
33. En particulier, malgré un format de lecture `E` ou `D`, on peut lire un réel avec ou sans exposant et la conversion dépendra alors de la forme lors de la lecture, rendant le résultat imprévisible pour une saisie au clavier : cela constitue une raison supplémentaire pour éviter l'emploi du format `P` en lecture.
34. Avec le compilateur `g95`, il est possible de choisir la virgule comme séparateur décimal au moment de l'exécution, grâce à la variable d'environnement `G95_COMMA` booléenne, par défaut à `FALSE` (*cf.* F.6, p. 175).

5.4.3 Syntaxe des formats et règles d'exploration

- Une liste de descripteurs, séparée par des virgules, peut être érigée en groupe de descripteurs, délimité par des parenthèses.
- Un facteur de répétition (entier positif préfixant le descripteur) peut être appliqué à un descripteur de format ou à un groupe de descripteurs.
- **Chaque instruction d'entrée-sortie provoque le passage à l'enregistrement suivant**, sauf si l'argument optionnel `ADVANCE='no'` a été spécifié.

En principe la liste d'entrée-sortie comporte autant d'éléments qu'il y a de descripteurs actifs dans le format. Dans le cas contraire, on applique les principes suivants :

⚠⇒
- **Si la liste épuise les descripteurs actifs du format, il y a passage à l'enregistrement suivant** [35] et ré-exploration du format en partant de la parenthèse ouvrante correspondant à l'avant-dernière parenthèse fermante du format, et en tenant compte de son éventuel facteur de répétition.
- Si la liste contient moins d'éléments qu'il y a de descripteurs actifs dans le format, il est honoré jusqu'au premier descripteur actif redondant exclus, sauf si on rencontre le descripteur de contrôle « `:` », qui interrompt l'exploration du format lorsque la liste est épuisée.

5.4.4 Boucles et changements d'enregistrement

Comme, en accès séquentiel, chaque instruction d'entrée-sortie fait par défaut progresser d'un enregistrement dans le fichier, une boucle explicite sur le nombre d'éléments d'un tableau avec une instruction `READ/WRITE` dans la boucle ne permet pas de lire ou écrire tous les éléments du tableau dans un enregistrement unique. On peut alors utiliser soit une instruction agissant sur le tableau global, soit une boucle implicite. Mais, pour éviter des changements d'enregistrements intempestifs, il faut aussi s'assurer (*cf.* 5.4.3, p. 52) que le format spécifié n'est pas épuisé par la liste, ce qui provoquerait une réexploration du format.

```
1  PROGRAM format_tab
2  IMPLICIT NONE
3  INTEGER, DIMENSION(3) :: t = (/ 1, 2, 3 /)    ! tableau initialisé
4  INTEGER :: i
5  WRITE(*, '(3i4)') t(:)             ! tableau global => affichage en ligne
6  DO i = 1, 3                        ! boucle explicite
7     WRITE(*, '(i4)') -t(i)          ! => un changement d'enregistrement par ordre
8  END DO                             ! => affichage en colonne
9  WRITE(*, '(3i4)') (2* t(i), i=1, 3)! boucle implicite => affichage en ligne
10 WRITE(*, '(i4)') (-2*t(i), i=1, 3) ! format réexploré => en colonne
11 DO i = 1, 3                        ! boucle explicite
12    WRITE(*, '(i4)', ADVANCE='no') 3*t(i)  ! mais option advance='no'
13 END DO                             ! => affichage en ligne
14 WRITE(*, *)                        !passage à l'enregistrement suivant à la fin
15 END PROGRAM format_tab
```

Le programme précédent affiche successivement :

- `t` en ligne avec le tableau global (ligne 5) ;
- `-t` en colonne avec la boucle explicite (lignes 6 à 8) ;
- `2*t` en ligne avec la boucle implicite (ligne 9) ;
- `-2*t` en colonne malgré la boucle implicite (ligne 10), car le format ne satisfait qu'un élément de la liste et est donc réexploré ;
- `3*t` en ligne malgré la boucle explicite (lignes 11 à 13), grâce à `ADVANCE='no'`.

```
   1   2   3
  -1
  -2
  -3
   2   4   6
  -2
  -4
  -6
   3   6   9
```

35. Sauf pour les fichiers à accès `stream` ou dans le cas où `ADVANCE=no` a été préalablement spécifié.

♠ 5.4.5 Format variable

Il n'existe pas à proprement parler [36] de format variable en fortran. Mais il est possible de déterminer le format à l'exécution, car il peut être défini par une variable chaîne de caractères. L'opération s'effectue donc en deux étapes, avec par exemple :

- une instruction d'écriture sur un fichier interne qui construit la variable de type chaîne contenant le format ;
- l'instruction d'entrée-sortie utilisant ce format.

À titre d'exemple, le programme suivant montre comment choisir le nombre `n` de chiffres (entre 1 et 9) pour afficher un entier `i`.

```
1  PROGRAM var_fmt
2  ! format variable à l'exécution
3  IMPLICIT NONE
4  INTEGER :: i = 123 ! le nombre (au maximum 9 chiffres) à afficher
5  INTEGER :: n       ! le nombre de chiffres (<=9) pour l'afficher"
6  CHARACTER(LEN=1) :: chiffres ! n codé sur un caractère
7  CHARACTER(LEN=6) :: fmt="(i?.?)" ! la chaîne de format à modifier
8  n = 4    ! premier cas
9  ! écriture sur fichier interne (en format fixe !)
10 WRITE(fmt(3:3), '(i1)') n
11 WRITE(fmt(5:5), '(i1)') n
12 WRITE(*, fmt) i
13 n = 7    ! deuxième cas
14 ! autre méthode : écriture sur une sous-chaîne
15 WRITE(chiffres, '(i1)') n
16 ! concaténation pour construire le format
17 fmt = "(i" // chiffres // "." // chiffres // ")"
18 WRITE(*, fmt) i
19 END PROGRAM var_fmt
```

Avec `n=4` puis `n=7`, il produit les affichages successifs :

```
0123
0000123
```

5.5 Exemples d'entrées-sorties formatées

5.5.1 Descripteurs de données numériques en écriture

Écriture de données entières

Noter que les lignes trop longues ont été repliées : la suite est introduite par le symbole →.

```
1   écriture des entiers 1, 20, 300, -1, -20, -300 et 0
2   écriture de ces entiers en format libre
3  1 20 300 -1 -20 -300 0
4   écriture de ces entiers en format In[.p] (décimal)
5 |i1|.....|1|*|*|*|*|*|0|
6 |i1.0|...|1|*|*|*|*|*|␣|
```

36. Certaines extensions propriétaires du fortran, notamment le compilateur d'Intel (*cf.* chap. F.4, p. 171) , celui d'Ibm, `xlf`, (*cf.* chap. F.1, p. 168) et celui de Portland (*cf.* chap. F.3, p. 170), permettent le format variable : des expressions numériques entières encadrées par les délimiteurs `<` et `>` peuvent figurer dans la chaîne de caractères définissant le format ; elles sont interprétées à chaque opération d'entrée-sortie. Par exemple, `WRITE(*,"(I<j>.<j>)") j` permet d'écrire l'entier `j` avec exactement `j` chiffres. Le format variable est aussi disponible dans le langage C avec le spécificateur de format `*` : `printf("%*d\n", n, i);` imprime l'entier `i` avec `n` chiffres.

```
|i2|.....|␣1|20|**|-1|**|**|␣0|
|i2.2|...|01|20|**|**|**|**|00|
|i3|.....|␣␣1|␣20|300|␣-1|-20|***|␣␣0|
|i4|.....|␣␣␣1|␣␣20|␣300|␣␣-1|␣-20|-300|␣␣␣0|
|i4.2|...|␣␣01|␣␣20|␣300|␣-01|␣-20|-300|␣␣00|
|i5|.....|␣␣␣␣1|␣␣␣20|␣␣300|␣␣␣-1|␣␣-20|␣-300|␣␣␣␣0|
|sp,i5|..|␣␣␣+1|␣␣+20|␣+300|␣␣␣-1|␣␣-20|␣-300|␣␣␣+0|
|i5.0|...|␣␣␣␣1|␣␣␣20|␣␣300|␣␣␣-1|␣␣-20|␣-300|␣␣␣␣␣|
|i5.4|...|␣0001|␣0020|␣0300|-0001|-0020|-0300|␣0000|
|i5.5|...|00001|00020|00300|*****|*****|*****|00000|
   format I0 largeur ajustable (f95/2003 seulement )
|i0|.....|1|20|300|-1|-20|-300|0|
|i0.2|...|01|20|300|-01|-20|-300|00|
  écriture de ces entiers en format Zn[.p] (hexadécimal)
|z8|.....|␣␣␣␣␣␣␣1|␣␣␣␣␣␣14|␣␣␣␣␣12C|FFFFFFFF|FFFFFFEC|FFFFFED4|␣␣␣␣␣␣␣0|
|z8.8|...|00000001|00000014|0000012C|FFFFFFFF|FFFFFFEC|FFFFFED4|00000000|
   format Z0 largeur ajustable (f95/2003 seulement)
|z0|.....|1|14|12C|FFFFFFFF|FFFFFFEC|FFFFFED4|0|
|z0.4|...|0001|0014|012C|FFFFFFFF|FFFFFFEC|FFFFFED4|0000|
  écriture de ces entiers en format On[.p] (octal)
|o8|.....|␣␣␣␣␣␣␣1|␣␣␣␣␣␣24|␣␣␣␣␣454|********|********|********|␣␣␣␣␣␣␣0|
|o8.8|...|00000001|00000024|00000454|********|********|********|00000000|
   format O0 largeur ajustable (f95/2003 seulement)
|o0|.....|1|24|454|37777777777|37777777754|37777777324|0|
|o0.4|...|0001|0024|0454|37777777777|37777777754|37777777324|0000|
  écriture de ces entiers en format Bn[.p] (binaire)
|b8|.....|␣␣␣␣␣␣␣1|␣␣␣10100|********|********|********|********|␣␣␣␣␣␣␣0|
|b8.8|...|00000001|00010100|********|********|********|********|00000000|
   format B0 largeur ajustable (f95/2003 seulement)
|b0|.....|1|10100|100101100|11111111111111111111111111111111|11111111111111111111
  →      11111101100|11111111111111111111111011010100|0|
|b0.4|...|0001|10100|100101100|11111111111111111111111111111111|11111111111111111111
  →      11111111101100|11111111111111111111111011010100|0000|
```

Écriture de données réelles

```
  écriture des réels 1.23456789, 1.23456789e4, 1.23456789e-2
  puis de leurs opposés
  écriture de ces réels en format libre
 1.234568 12345.68 0.01234568
-1.234568 -12345.68 -0.01234568
  écriture de ces réels en format Fn.p
   format Fn.p avec n >0 (fortran 90)
|f9.1|...|␣␣␣␣␣␣1.2|␣␣12345.7|␣␣␣␣␣␣0.0|
|f9.1|...|␣␣␣␣␣-1.2|␣-12345.7|␣␣␣␣␣␣0.0|
|f9.3|...|␣␣␣␣1.235|12345.680|␣␣␣␣0.012|
|f9.3|...|␣␣␣-1.235|*********|␣␣␣-0.012|
   format F0.p largeur ajustable (fortran 95/2003 seulement)
|f0.0|...|1.|12346.|0.|
|f0.0|...|-1.|-12346.|0.|
|f0.3|...|1.235|12345.680|0.012|
|f0.3|...|-1.235|-12345.680|-0.012|
  écriture de ces réels en format En.p[eq]
|e12.4|..|␣␣0.1235E+01|␣␣0.1235E+05|␣␣0.1235E-01|
|e12.4|..|␣-0.1235E+01|␣-0.1235E+05|␣-0.1235E-01|
```

```
|e12.4e1||␣␣␣0.1235E+1|␣␣␣0.1235E+5|␣␣␣0.1235E-1|
|e12.4e1||␣␣-0.1235E+1|␣␣-0.1235E+5|␣␣-0.1235E-1|
  écriture de ces réels en format ESn.p (scientifique)
|es12.4|.|␣␣1.2346E+00|␣␣1.2346E+04|␣␣1.2346E-02|
|es12.4|.|␣-1.2346E+00|␣-1.2346E+04|␣-1.2346E-02|
  écriture de ces réels en format ENn.p (ingénieur)
|en12.4|.|␣␣1.2346E+00|␣12.3457E+03|␣12.3457E-03|
|en12.4|.|␣-1.2346E+00|-12.3457E+03|-12.3457E-03|
  écriture de ces réels en format Gn.p (général)
|g9.3|...|␣1.23␣␣␣␣|0.123E+05|0.123E-01|
|g9.3|...|-1.23␣␣␣␣|-.123E+05|-.123E-01|
```

5.5.2 Descripteurs de contrôle en écriture

```
  Écriture des six entiers 1, 20, 300, -1, 20, -300
  Espace avec "x" (i4, x, i4, x, i4, x, i4, x, i4, x, i4)
␣␣␣1␣␣␣20␣␣300␣␣␣-1␣␣-20␣-300
  Répétition: (6i4)
␣␣␣1␣␣20␣300␣␣-1␣-20-300
  Groupement avant répétition : (6(x,i4))
␣␣␣␣1␣␣␣20␣␣300␣␣␣-1␣␣-20␣-300
  Changement d'enregistrement forcé par "/" : (3(x,i4),/,3(x,i4))
␣␣␣␣1␣␣␣20␣␣300
␣␣␣-1␣␣-20␣-300

  Liste plus longue => réexploration avec changement d'enregistrement
  Réexploration du format (x, i4, x, i4)
␣␣␣␣1␣␣␣20
␣␣300␣␣␣-1
␣␣-20␣-300
  Réexploration du format (3(x, i4))
␣␣␣␣1␣␣␣20␣␣300
␣␣␣-1␣␣-20␣-300
  Réexploration à partir de 3(x, i4) : (9x, 3(x, i4))
␣␣␣␣␣␣␣␣␣␣␣␣␣1␣␣␣20␣␣300
␣␣␣-1␣␣-20␣-300

  Liste plus courte => arrêt avant le 1er descripteur actif inutilisé
  arrêt après "[<" dans le format (8("[","<",i4,">","]"))
[<␣␣␣1>][<␣␣20>][<␣300>][<␣␣-1>][<␣-20>][<-300>][<
  Liste plus courte avec ":" => arrêt juste après le ":"
  suivant le dernier descripteur actif utile
  arrêt via ":" (8("[","<",i4,">",:,"]"))
[<␣␣␣1>][<␣␣20>][<␣300>][<␣␣-1>][<␣-20>][<-300>

  Usage des tabulations
  Référence (6(6x, i4))
␣␣␣␣␣␣␣␣␣1␣␣␣␣␣␣␣␣20␣␣␣␣␣␣␣␣300␣␣␣␣␣␣␣␣-1␣␣␣␣␣␣␣␣-20␣␣␣␣␣␣␣-300
12345678901234567890123456789012345678901234567890123456789012345678901234567890
  Tabulation à droite format (6(tr6, i4))
␣␣␣␣␣␣␣␣␣1␣␣␣␣␣␣␣20␣␣␣␣␣␣␣␣300␣␣␣␣␣␣␣␣-1␣␣␣␣␣␣␣␣-20␣␣␣␣␣␣␣-300
  Tabulations absolues et à gauche  => changement d'ordre possible !
  format (t7,i4, t17,i4, t27,i4, t57,i4, tl14,i4, tl14,i4)
␣␣␣␣␣␣␣␣␣1␣␣␣␣␣␣␣20␣␣␣␣␣␣␣␣300␣␣␣␣␣␣␣-300␣␣␣␣␣␣␣␣-20␣␣␣␣␣␣␣␣␣-1
```

5.6 Exemples de fichiers en accès direct

Les deux exemples élémentaires suivants présentent l'écriture d'un fichier à accès direct, sa lecture avec affichage des données et la mise à jour d'un des enregistrements.

5.6.1 Exemple de fichier formaté en accès direct

```
1  PROGRAM direct
2  ! fichier formaté en accès direct
3  IMPLICIT NONE
4  INTEGER :: ok=0, i, ir, nr
5  REAL :: a
6  INTEGER, PARAMETER :: iunit=11, n=10, recl=12
7  ! création du fichier en accès direct
8  OPEN(FILE='base.dat', UNIT=iunit, ACCESS='direct', RECL=recl, &
9       FORM='formatted', STATUS='replace')
10 ecriture: DO i=1, n
11    WRITE(UNIT=iunit, REC=i, FMT='(i4, 2x, f6.2)') 1000+i, 10*i + float(i)/10.
12 END DO ecriture
13 ! lecture du fichier en accès direct et affichage
14 ir=1
15 lecture: DO  ! on ne sait pas a priori où s'arrêter
16    READ(UNIT=iunit, REC=ir, FMT='(i4, 2x, f6.2)', IOSTAT=ok) i, a
17    IF (ok /= 0) EXIT ! fin de fichier
18    WRITE(*, '(i4, 2x, f6.2)') i, a
19    ir = ir + 1
20 END DO lecture
21 WRITE(*,*) ir-1, ' enregistrements lus'
22 ok=0
23 ! modification d'un enregistrement du fichier en accès direct
24 WRITE(*,*) 'choisir un numéro d''enregistrement à modifier'
25 READ *, nr
26 READ(UNIT=iunit, REC=nr, FMT='(i4, 2x, f6.2)') i, a
27 WRITE(*,*) 'ancienne valeur ', a, ' entrer la nouvelle valeur réelle'
28 READ *, a
29 WRITE(UNIT=iunit, REC=nr, FMT='(i4, 2x, f6.2)') i, a
30 ir=1
31 relecture: DO  ! on ne sait pas a priori où s'arrêter
32    READ(UNIT=iunit, REC=ir, FMT='(i4, 2x, f6.2)', IOSTAT=ok) i, a
33    IF (ok /= 0) EXIT ! fin de fichier
34    WRITE(*, '(i4, 2x, f6.2)') i, a
35    ir = ir + 1
36 END DO relecture
37 WRITE(*,*) ir-1, ' enregistrements lus'
38 ok=0
39 CLOSE(iunit)
40 END PROGRAM direct
```

5.6.2 Exemple de fichier non-formaté en accès direct

Dans le cas non-formaté, c'est l'instruction `INQUIRE(IOLENGTH=recl)` (ligne 8) qui permet de déterminer la taille de l'enregistrement, qui dépend du processeur.

```
1  PROGRAM direct
2  ! fichier non formaté en accès direct
3  IMPLICIT NONE
```

```
4  INTEGER :: ok=0, i, ir, nr, recl
5  REAL :: a
6  INTEGER, PARAMETER :: iunit=11, n=10
7  ! calcul de la longueur d'un enregistrement via INQUIRE
8  INQUIRE(IOLENGTH=recl) i, a !
9  WRITE(*,*) 'longueur d''un enregistrement ', recl
10 ! création du fichier en accès direct
11 OPEN(FILE='base.dat', UNIT=iunit, ACCESS='direct', RECL=recl, &
12      FORM='unformatted', STATUS='replace')
13 ecriture: DO i=1, n
14    WRITE(UNIT=iunit, REC=i) 1000+i, float(10*i) + float(i)/10.
15 END DO ecriture
16 ! lecture du fichier en accès direct et affichage
17 ir=1
18 lecture: DO  ! on ne sait pas a priori où s'arrêter
19    READ(UNIT=iunit, REC=ir, IOSTAT=ok) i, a
20    IF (ok /= 0) EXIT ! fin de fichier
21    WRITE(*, '(i4, 2x, f6.2)') i, a
22    ir = ir + 1
23 END DO lecture
24 WRITE(*,*) ir-1, ' enregistrements lus'
25 ok=0
26 ! modification d'un enregistrement du fichier en accès direct
27 WRITE(*,*) 'choisir un numéro d''enregistrement à modifier'
28 READ *, nr
29 READ(UNIT=iunit, REC=nr) i, a
30 WRITE(*,*) 'ancienne valeur ', a, ' entrer la nouvelle valeur réelle'
31 READ *, a
32 WRITE(UNIT=iunit, REC=nr) i, a
33 ! re-lecture du fichier en accès direct et affichage
34 ir=1
35 relecture: DO  ! on ne sait pas a priori où s'arrêter
36    READ(UNIT=iunit, REC=ir, IOSTAT=ok) i, a
37    IF (ok /= 0) EXIT ! fin de fichier
38    WRITE(*, '(i4, 2x, f6.2)') i, a
39    ir = ir + 1
40 END DO relecture
41 WRITE(*,*) ir-1, ' enregistrements lus'
42 ok=0
43 CLOSE(iunit)
44 END PROGRAM direct
```

Chapitre 6

Procédures

6.1 Introduction

6.1.1 Intérêt des procédures

Dès qu'un programme commence à dépasser une page ou comporte des duplications d'instructions, il s'avère avantageux de le scinder en sous-ensembles plus élémentaires, nommés procédures, qui seront appelées par le programme principal. En itérant ce processus, on structure le code source en une hiérarchie de procédures plus concises et modulaires, qui permet :

- d'améliorer la lisibilité et de faciliter la vérification et la maintenance ;
- de réutiliser des outils déjà mis au point par d'autres applications, au prix d'une généralisation et d'un paramétrage des procédures.

Par exemple, au lieu de dupliquer à chaque usage toutes les instructions de calcul de la somme des `n` premiers entiers (le seul paramétrage de cet exemple élémentaire), il est préférable de les écrire une fois pour toutes dans un sous-programme noté `cumul` qui sera appelé chaque fois que nécessaire. Ce sous-programme devra communiquer avec l'appelant via deux arguments :

- le nombre des entiers à sommer fourni par l'appelant (argument d'entrée) : `n`
- la valeur de la somme renvoyée à l'appelant (argument de sortie) : `somme`.

```
SUBROUTINE cumul(n, somme)
 IMPLICIT NONE
 INTEGER, INTENT(in)  :: n              ! argument d'entrée
 INTEGER, INTENT(out) :: somme          ! argument de sortie
 INTEGER              :: i              ! variable locale
 somme = 0
 DO i = 1, n
   somme = somme + i
 END DO
END SUBROUTINE cumul
```

Les appels successifs de ce sous-programme se feront alors de la manière suivante :

```
INTEGER :: p, s
...
CALL cumul(10, s)     ! appel de cumul pour sommer les 10 premiers entiers
PRINT *, ' somme des 10 premiers entiers = ', s
...
p = 5
CALL cumul(p, s)      ! appel de cumul pour sommer les 5 premiers entiers
PRINT *, ' somme des ', p, ' premiers entiers = ', s
...
```

6.1.2 Variables locales, automatiques et statiques

Les variables déclarées au sein d'une procédure sont, a priori, locales. On dit que leur *portée* (scope) est limitée à l'unité de programme. Par défaut, les variables locales d'une procédure n'ont d'existence garantie que pendant son exécution : elles sont qualifiées d'*automatiques* ; elles ne sont pas nécessairement mémorisées [1] entre deux appels. C'est le cas de la variable `i` dans la procédure `cumul` (*cf.* 6.1.1, p. 58).

On peut cependant forcer cette mémorisation, pour en faire des variables *statiques*, grâce à l'attribut `SAVE`. Ne pas croire qu'alors la portée de la variable soit étendue : elle reste locale mais est alors permanente. De plus, si une variable locale est initialisée lors de sa déclaration, elle devient immédiatement une variable statique [2]. Ne pas oublier que l'initialisation est effectuée par le compilateur, une fois pour toute et diffère donc d'une instruction exécutable d'affectation effectuée à chaque appel [3]. ⇐⚠

Par exemple, le sous-programme `compte` suivant affiche le nombre de fois qu'il a été appelé. Si la variable locale n'était pas *statique*, il faudrait passer `n` en argument de `compte` pour le mémoriser.

```
SUBROUTINE compte
   IMPLICIT NONE
   INTEGER, SAVE :: n = 0 !  n est locale, mais statique
   n = n + 1
   print *, n
END SUBROUTINE compte
```

Si l'on souhaite rendre statiques *toutes* les variables locales d'une procédure, ce qui est fortement déconseillé [4], il suffit d'insérer l'ordre `SAVE` [5] avant toutes les déclarations. On peut aussi passer une option au compilateur (*cf.* annexe F, p. 168) : `-fnoautomatic` pour le compilateur `gfortran`, `-fstatic` pour le compilateur `g95`, `-qsave` pour `xlf` d'IBM, `-save` pour le compilateur NAG, `-save` pour le compilateur `ifort` d'INTEL, `-static` pour celui `f90` de DEC.

6.1.3 Arguments des procédures

La communication entre une procédure et le programme appelant peut se faire par le passage de paramètres appelés arguments de la procédure. Ces arguments sont :

- déclarés comme *arguments muets* (dummy arguments) formels ou symboliques lors de la définition de la procédure ;
- spécifiés comme *arguments effectifs* (actual arguments) lors des appels de la procédure, pouvant prendre des valeurs différentes à chaque appel.

En fortran, les passages d'arguments se font par référence [6], ce qui permet de partager les zones mémoires des variables stockant les arguments entre appelé et appelant. Lors de l'appel de la procédure, la liste des arguments effectifs doit respecter :

- le nombre (hormis dans le cas d'arguments optionnels, *cf.* 6.5.1, p. 70),
- l'ordre (hormis en cas de passage par mot-clef, *cf.* 6.5.2, p. 71),

1. Cela dépend des options de compilation qui peuvent forcer ou non un stockage statique ;
2. Par souci de lisibilité, on précisera explicitement l'attribut `SAVE` dans la déclaration de toute variable locale initialisée, au lieu de s'appuyer sur le caractère statique implicite.
3. Au contraire, en C, une variable locale initialisée est initialisée à l'exécution, à chaque appel à la fonction hôte ; pour la rendre permanente (statique), il faut lui adjoindre l'attribut `static`.
4. Sauf lors de certaines opérations de déboggage.
5. L'instruction `SAVE` doit alors être seule sur la ligne. Elle peut aussi être utilisée sous la forme `SAVE :: liste_de variables` pour déclarer statiques une liste de variables.
6. Au contraire, dans le langage C, les passages d'arguments se font par copie : cela permet d'effectuer des conversions si le type de l'argument effectif diffère de celui de l'argument formel. Mais, si l'on souhaite modifier des variables de l'appelant, cela impose de transmettre des copies de leurs adresses à la fonction appelée : les arguments de la fonction appelée seront donc des pointeurs vers le type de la variable à modifier ; on accédera alors aux zones mémoire de ces variables par indirection depuis la fonction appelée.

– et le type [7] et la variante de type éventuelle

des arguments muets déclarés dans la procédure.

6.1.4 L'attribut `INTENT` de vocation des arguments

Le fortran 90 a introduit, dans un but de fiabilisation des communications entre procédures, un attribut facultatif de vocation, `INTENT`, permettant de spécifier si un argument est :

– `INTENT(in)` : un argument d'entrée, qui ne doit pas être modifié par la procédure [8] (c'est le seul cas où l'argument effectif peut être une expression, en particulier une constante) ;

– `INTENT(out)` : un argument de sortie, qui doit affecté au sein de la procédure (il est indéterminé au début de la procédure) ;

– `INTENT(inout)` : un argument d'entrée-sortie, fourni à la procédure, mais éventuellement modifié par cette procédure.

♥ ⇒ En précisant ces attributs, on permet au compilateur un meilleur contrôle, lui donnant les moyens de détecter par exemple une modification involontaire d'un argument d'entrée (`INTENT(IN)`) dans la procédure, ou l'absence d'affectation d'un argument de sortie (`INTENT(OUT)`) au sein de la procédure.

⚠⇒ Par exemple, le calcul de l'aire d'un triangle effectué maladroitement comme suit, sans préciser la vocation des arguments, produit un effet de bord involontaire qui est de diviser `base` par deux.

```
SUBROUTINE aire(base, hauteur, surface) ! version sans vocation des arguments
   IMPLICIT NONE
   REAL  :: base, hauteur, surface      ! déclaration des arguments
   base = base / 2.                     ! modification de base autorisée
   surface = base * hauteur             ! forme maladroite du calcul de surface
END SUBROUTINE aire
```

Si on avait précisé que l'argument `base` est un argument d'entrée, le compilateur n'aurait pas accepté la modification de la variable `base` dans le sous-programme `aire`.

```
SUBROUTINE aire(base, hauteur, surface) ! version avec vocation des arguments
   IMPLICIT NONE
   REAL, INTENT(IN)   :: base, hauteur  ! déclaration des arguments d'entrée
   REAL, INTENT(OUT)  :: surface        ! déclaration des arguments de sortie
   base = base / 2.                     ! affectation de base interdite ici
   surface = base * hauteur             ! forme maladroite du calcul de surface
END SUBROUTINE aire
```

La vocation `INTENT(INOUT)` pour un argument modifiable, n'impose pas de contrainte dans la procédure, mais permet au compilateur d'imposer l'emploi d'une variable comme argument effectif alors qu'une expression est acceptée dans le cas d'un argument à vocation `INTENT(IN)` non modifiable. Le tableau suivant résume les vocations possibles et les contraintes associées.

INTENT	IN	OUT	INOUT
argument	d'entrée	de sortie	modifiable
affectation	*interdite*	*nécessaire*	possible
argument effectif	expression	*variable*	*variable*
contrainte sur	appelé	appelé + appelant	appelant

7. Dans le cas d'une procédure générique (*cf.* 10.1, p. 113), c'est le type des arguments effectifs passés qui permettra d'aiguiller vers la procédure spéficique adéquate.

8. En langage C, la transmission d'arguments se faisant par copie de valeur, la modification des variables de la fonction appelante est par principe impossible.

6.2 Sous-programmes et fonctions

On distingue deux sortes de procédures :

- Les *sous-programmes* (subroutine) sont constitués d'une suite d'instructions qui effectuent une tâche déterminée quand on les appelle depuis une autre unité de programme via l'instruction `CALL` et rendent le contrôle à l'unité appelante après exécution. Des paramètres ou arguments peuvent être échangés entre un sous-programme et l'unité appelante.
- Les *fonctions* (function) sont des procédures qui renvoient un résultat sous leur nom, résultat qui est ensuite utilisé dans une expression au sein de l'unité de programme appelante : la simple mention de la fonction (avec ses arguments) dans l'expression provoque l'exécution des instructions que comprend la procédure. Cette notion s'apparente ainsi à celle de fonction au sens mathématique du terme.

Remarques :

- Une fonction pourrait être considérée comme un cas particulier d'un sous-programme, qui renverrait un résultat accessible sous le nom de la fonction. On peut donc en général transformer une fonction en sous-programme, quitte à utiliser un argument supplémentaire pour le résultat et une ou éventuellement plusieurs variables supplémentaires dans l'appelant, si plusieurs appels à la fonction coexistent dans une même expression.
- À l'inverse, s'il s'agit d'effectuer une action qui n'est pas un simple calcul et par exemple modifier les arguments d'appel, on préfère le sous-programme. Parallèlement, il est d'usage de ne pas modifier les paramètres d'appel d'une fonction [9], considérés comme arguments d'entrée (*cf.* 6.1.4, p. 60). ⇐♥

6.2.1 Sous-programmes

Un sous-programme est introduit par l'instruction `SUBROUTINE`, suivie du nom du sous-programme et de la liste de ses (éventuels) arguments muets entre parenthèses et séparés par des virgules ; les types de ces arguments ainsi que leur vocation doivent ensuite être déclarés. Le sous-programme est délimité par l'instruction `END` [10] éventuellement suivie de `SUBROUTINE`, éventuellement suivi du nom du sous-programme [11].

```
SUBROUTINE <nom_sous_programme>[(<arg_1>, <arg_2>, ...)]
...
[RETURN]
END [SUBROUTINE [<nom_sous_programme>] ]
```

Par exemple, le sous-programme `aire`,

```
SUBROUTINE aire(base, hauteur, surface) ! arguments muets
   IMPLICIT NONE
   REAL, INTENT(IN)   :: base, hauteur  ! déclaration des arguments d'entrée
   REAL, INTENT(OUT)  :: surface        ! déclaration des arguments de sortie
   surface = base * hauteur / 2.        ! calcul de surface
END SUBROUTINE aire
```

peut être appelé par exemple via

```
REAL :: b1, h1, s1, b2, h2, s2
b1 = 30.
```

9. Plus précisément une fonction qualifiée de `PURE` (*cf.* 6.5.7, p. 79), ne doit pas modifier ses arguments.
10. Avant `END`, on peut insérer une ou plusieurs instructions `RETURN`, (*cf.* 4.7.4, p. 34) qui renvoient le contrôle à l'appelant, mais ce n'est pas obligatoire s'il n'y a qu'un point de retour.
11. Pour améliorer la lisiblité des sources, on utilisera systématiquement cette possibilité de délimiter clairement les procédures au lieu de se contenter de les terminer toutes par `END`.

```
h1 = 2.
CALL aire(b1, h1, s1)            ! appel avec arguments effectifs de même type
PRINT *, ' Surface = ', s1       ! s1 = 30.
b2 = 2.
h2 = 5.
CALL aire(b2, h2, s2)            ! appel avec arguments effectifs de même type
PRINT *, ' Surface = ', s2       ! s2 = 5.
```

6.2.2 Fonctions

Une fonction est encadrée par la déclaration `FUNCTION`, suivie du nom de la fonction et de la liste éventuelle de ses arguments muets et l'instruction `END` [12], éventuellement suivie de `FUNCTION`, éventuellement suivie du nom de la fonction [13]. Le type du résultat peut être déclaré :

- soit dans l'entête [14], comme préfixe de `FUNCTION`,
- soit déclaré au début du corps de la fonction, en même temps que les arguments, mais il ne faut pas lui attribuer une vocation `INTENT(OUT)`.

La fonction doit comporter une ou plusieurs instructions qui affectent une valeur au résultat, sous la forme *<nom_de_la_fonction>=<expression>*, ou par appel à d'autres procédures.

```
[<type>] FUNCTION <nom_de_la_fonction>[(<arg_1>, <arg_2>, ...)]
...
<nom_de_la_fonction> = <expression>
[RETURN]
END [FUNCTION [<nom_de_la_fonction>] ]
```

Par exemple, le sous-programme `aire`, n'ayant qu'un argument de sortie, peut être traduit en fonction [15] qui renvoie la valeur de la surface :

```
FUNCTION surf(base, hauteur)          ! arguments muets
   IMPLICIT NONE
   REAL             :: surf           ! déclaration du résultat
   REAL, INTENT(IN) :: base, hauteur  ! déclaration des arguments muets
   surf = base * hauteur / 2.         ! affectation du résultat
END FUNCTION surf
```

qui peut être utilisée de la manière suivante :

```
...
REAL :: b1, h1, somme
...
b1 = 30.
h1 = 2.
PRINT *, ' Surface= ', surf(b1, h1) ! appel avec arguments effectifs
somme = surf(b1, h1) + surf(b1/2., h1/2.) ! deux appels dans la même expression
...                                 ! somme = 30. + 15. = 45.
```

12. On peut insérer `RETURN` avant `END`, *cf.* 4.7.4, p. 34.
13. Pour améliorer la lisiblité des sources, on utilisera systématiquement cette possibilité de délimiter clairement les procédures au lieu de se contenter de les terminer toutes par `END`.
14. C'est ainsi que procède le langage C, mais sans le mot-clef `FUNCTION`.
15. On aurait aussi pu écrire `REAL FUNCTION surf(base, hauteur)` au lieu de déclarer le type du résultat dans le corps de la fonction.

6.3 Procédures internes et procédures externes

Les procédures appelées (sous-programmes et fonctions) peuvent être :

- soit incluses dans le programme appelant, nommé *hôte* (host) de la procédure et qualifiées de *procédures internes* (internal procedure) ; elles sont alors introduites par la déclaration `CONTAINS` ;
- soit extérieures au programme appelant (comme en fortran 77) et qualifiées alors de *procédures externes* (external procedure) ; on verra (*cf.* 6.4, p. 65) qu'il est recommandé de les insérer dans des modules.

6.3.1 Procédures internes : `CONTAINS`

Les procédures internes sont compilées en même temps que la procédure qui les appelle, ce qui permet un contrôle de la cohérence des arguments. Mais elles se prêtent peu à une réutilisation : on les réservera à des procédures très dépendantes de leur procédure hôte. Un seul niveau d'imbrication est possible : une seule instruction `CONTAINS` est possible dans une procédure [16].

```
PROGRAM principal
...
CALL sub1(...)                          ! appel du sous-programme interne sub1
...
x = fonc1(...)                          ! appel de la fonction interne fonc1
...
CONTAINS                                ! introduit les procédures internes
    FUNCTION fonc1(...)
    ...
    END FUNCTION fonc1
    SUBROUTINE sub1(...)
    ...
    END SUBROUTINE sub1
END PROGRAM principal
```

Dans une procédure interne, toutes les entités déclarées au sein de la procédure hôte sont a priori accessibles, sauf si on les redéclare localement [17], donc la procédure interne ne nécessite pas forcément d'argument pour communiquer avec l'hôte. ⇐⚠

Voici un exemple mettant à profit la portée des variables de l'hôte (ici le programme principal), au détriment de la lisibilité :

```
1  PROGRAM principal                    ! version sans passage d'arguments
2  IMPLICIT NONE
3  REAL :: base, hauteur, surface       ! variables globales
4  base = 30.
5  hauteur = 2.
6  CALL aire
7  PRINT *, ' Surface = ', surface      ! surface = 30.
8  base = 2.
9  hauteur = 5.
10 CALL aire
11 PRINT *, ' Surface = ', surface      ! surface = 5.
12 CONTAINS
13    SUBROUTINE aire                   ! version déconseillée
14       ! ne pas redéclarer base, hauteur ou surface (variables globales)
```

16. Sauf en ce qui concerne les procédures de module, qui peuvent inclure une procédure interne.
17. Si on déclare une variable *locale* dans une procédure interne avec le même nom qu'une variable de l'hôte, la déclaration locale masque la variable de l'hôte.

```
15      ! sous peine de créer des variables locales distinctes !!!
16      surface = base * hauteur / 2.      ! calcul de surface
17    END SUBROUTINE aire
18  END PROGRAM principal
```

Le simple ajout d'une déclaration du type `REAL :: base` dans le sous-programme `aire` créerait ⚠⇒ une variable locale `base`, empêchant toute visibilité de la variable `base` du programme principal et rendant le sous-programme inutilisable : on dit que la variable locale *masque* la variable globale.

Dans un souci de clarté et de portabilité, on évitera de tirer parti de la visibilité des variables ♥ ⇒ globales [18] ; au contraire, on transmettra explicitement en arguments les variables partagées et on donnera aux variables locales à la procédure interne des noms non utilisés par la procédure hôte.

6.3.2 Procédures externes

Seules les procédures externes se prêtent à la réutilisation du code, mais nous verrons que le contexte le plus fiable de leur utilisation est celui des modules (*cf.* 6.4, p. 65). Qu'elles soient placées dans des fichiers séparés, ou dans le même fichier que l'appelant, leur compilation, hors contexte des modules, se déroule de façon indépendante. Les communications entre procédure externe et les procédures appelantes se font au travers des arguments transmis.

Les procédures écrites dans d'autres langages sont évidemment des procédures externes.

6.3.3 La notion d'interface

Dans le cas d'une procédure externe, la compilation séparée de la procédure appelée et de la procédure appelante ne permet pas au compilateur de connaître, sans information complémentaire, le type des arguments échangés et de vérifier la cohérence entre les arguments effectifs lors de l'appel et les arguments muets déclarés. Il est cependant possible d'assurer une visibilité de ces déclarations d'arguments à l'aide d'un *bloc d'interface explicite* (explicit interface) inséré dans la ou les procédures appelantes [19]. Les déclarations des arguments sont ainsi communiquées aux appelants à l'intérieur d'un bloc d'interface [20], délimité par les instructions `INTERFACE` et `END INTERFACE` [21] et inséré après les déclarations de variables de l'appelant.

```
INTERFACE                              ! début de bloc d'interface
  SUBROUTINE sub1(...)
    ... déclaration des arguments de sub1
  END SUBROUTINE sub1
  SUBROUTINE sub2(...)
    ... déclaration des arguments de sub2
  END SUBROUTINE sub2
END INTERFACE                          ! fin de bloc d'interface
```

Par exemple, avec la fonction externe `surf`,

```
FUNCTION surf(base, hauteur)
   IMPLICIT NONE
   REAL              :: surf            ! déclaration du résultat
   REAL, INTENT(IN)  :: base, hauteur   ! déclaration des arguments muets
   surf = base * hauteur / 2.           ! calcul de surface
END FUNCTION surf
```

18. En particulier, si une procédure interne a été écrite en s'appuyant sur cette lisibilité, elle ne sera pas aisément transformable en procédure externe par la suite.

19. Ces déclarations d'interface du fortran 90 sont à rapprocher des prototypes des fonctions en langage C.

20. Un seul bloc d'interface est possible dans une procédure, sauf si on déclare les interfaces via des modules (*cf.* 6.4, p. 65).

21. Il n'est possible de nommer un bloc d'interface que dans le contexte des interfaces génériques (*cf.* 10.1, p. 113).

on doit préciser dans la procédure appelante :

```
...
REAL :: b1, h1
...
INTERFACE
   FUNCTION surf(base, hauteur)
     IMPLICIT NONE
     REAL               :: surf          ! déclaration du résultat
     REAL, INTENT(IN)   :: base, hauteur ! déclaration des arguments muets
   END FUNCTION surf
END INTERFACE
...
b1 = 30.
h1 = 2.
PRINT *, ' Surface= ', surf(b1, h1)     ! arguments effectifs de même type
...
```

L'inconvénient immédiat du bloc d'interface explicite est que la déclaration des arguments doit être fidèlement dupliquée entre la procédure appelée et la ou les blocs d'interface dans la ou les procédures appelantes. Lors des mises à jour du code, il est possible de créer des incohérences entre ces déclarations et cette technique est donc déconseillée. Seule l'insertion des procédures externes dans des modules (*cf.* 6.4, p. 65) fournit une solution satisfaisante à cette question.

6.4 Les modules

Les *modules* (modules) sont des entités compilables séparément (mais non exécutables) permettant d'héberger des éléments de code ou de données qui ont vocation à être utilisés par plusieurs procédures sans nécessité de dupliquer l'information. Un module est délimité par les instructions : `MODULE <nom_de_module>` et `END MODULE <nom_de_module>`. Il peut comporter :

- des déclarations de types dérivés, (*cf.* 9.6, p. 108) et des déclarations de variables, susceptibles d'être partagés entre plusieurs unités de programme,
- un bloc d'interface,
- et des procédures dites de module (module procedure) introduites par l'instruction `CONTAINS`.

L'instruction `USE <nom_de_module>` [22], placée *avant* toutes les déclarations, assure à l'unité de programme ou à la procédure où elle est placée la visibilité de toutes les entités *publiques* (*cf.* 6.4.3 p. 69) du module : variables, types dérivés, procédures, blocs d'interface.

La compilation du module produit un fichier de module [23], dont le nom est celui du module et le suffixe généralement `.mod` [24]. Ce fichier sera exploité par l'instruction `USE` lors de la compilation des modules appelants afin d'effectuer notamment les contrôles inter-procéduraux de respect d'interface (type, vocation des arguments, ...) : il est donc nécessaire de compiler le ou les modules *avant* les procédures qui les utilisent.

Il est fortement recommandé d'inclure les procédures dans des modules afin que les entités qui les utilisent disposent d'une interface explicite extraite automatiquement par le compilateur. ⇐♥

22. L'instruction `USE` peut elle-même être utilisée dans un module, permettant ainsi l'imbrication des modules.

23. La structure de ces fichiers dépend du compilateur, et, pour `g95` et plus récemment, `gfortran`, par exemple, plusieurs versions de format de ces fichiers incompatibles entre elles ont existé. De plus, si les fichiers de module de `g95` et `gfortran` sont en fait des fichiers texte, ceux de `gfortran` sont compressés avec `gzip` depuis la version 4.9.

24. On peut comparer ce fichier `.mod` aux fichiers d'entête (header) des fonctions en langage C et l'instruction `USE` à la directive `#include`. Mais c'est le compilateur qui produit automatiquement le fichier `.mod` en fortran alors qu'en C, le fichier d'entête doit être écrit par l'utilisateur qui doit aussi le maintenir en conformité avec le code de la procédure à chaque mise à jour de la liste des paramètres. De plus, c'est le compilateur qui interprète l'instruction `USE` alors qu'une directive `#include` est traitée par le préprocesseur. D'ailleurs, il est possible d'utiliser le préprocesseur de la même façon en fortran qu'en C pour inclure des parties de code commun à plusieurs unités de programme.

Noter que cette inclusion de code pourrait aussi se faire via une directive de compilation `INCLUDE`, qui fait partie de la norme du fortran 90, et que certains compilateurs ont intégrée comme instruction fortran (hors norme cependant), mais cette méthode est actuellement déconseillée au profit de l'emploi de l'instruction `USE`.

6.4.1 Exemple de procédure de module

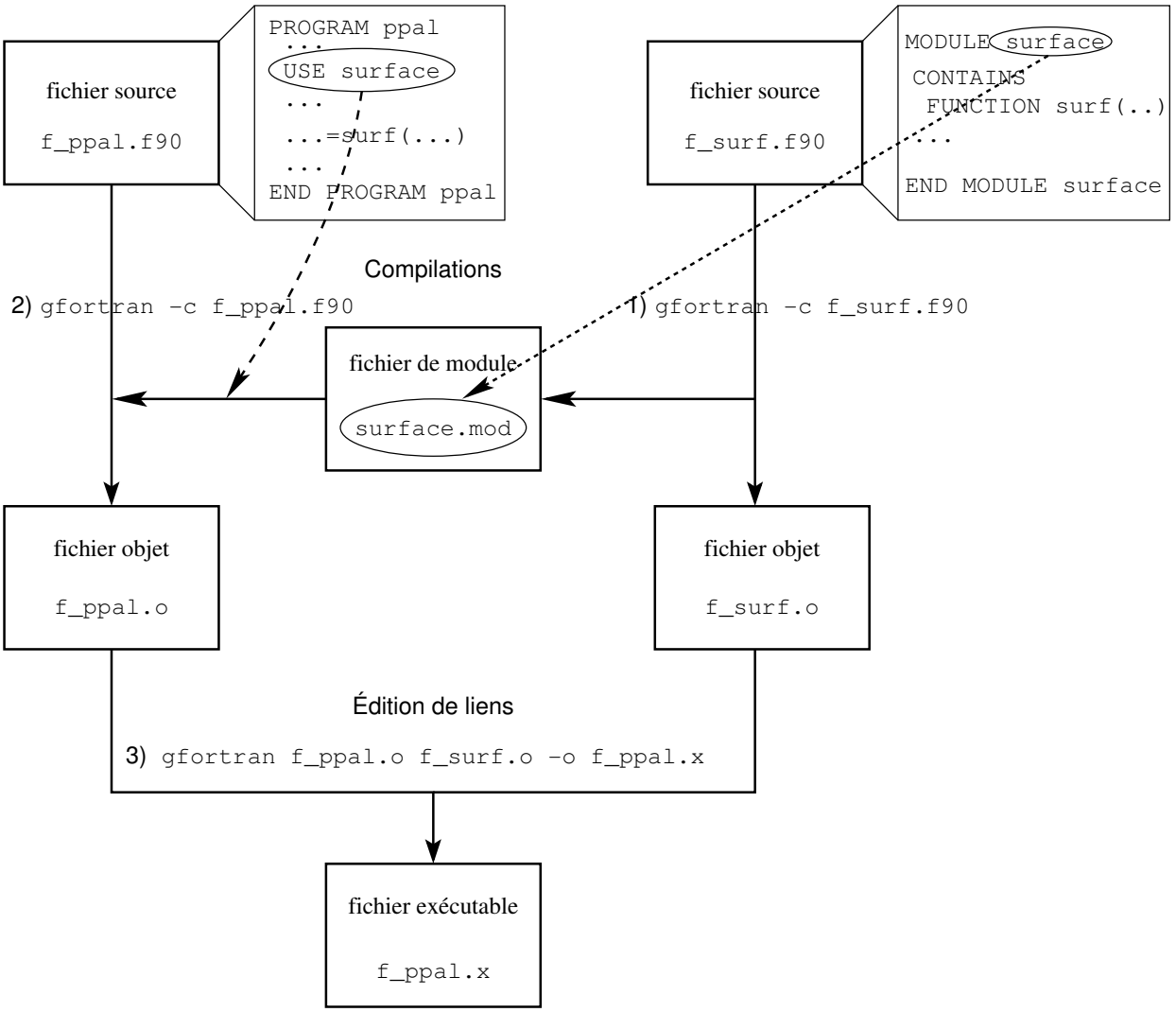


FIGURE 6.1 – Modules et fichiers impliqués en compilation séparée. Respecter l'ordre des trois opérations : 1) compilation du module, 2) compilation de l'appelant, 3) édition de liens.

Par exemple, le module `surface` contient la procédure interne de module `surf`.

```
MODULE surface
IMPLICIT NONE
CONTAINS
   FUNCTION surf(base, hauteur)
      REAL              :: surf            ! déclaration du résultat
      REAL, INTENT(IN)  :: base, hauteur   ! déclaration des arguments muets
      surf = base * hauteur / 2.           ! calcul de surface
   END FUNCTION surf
END MODULE surface
```

La visibilité du module `surface` et donc de sa procédure interne `surf`, est assurée dans une procédure appelante par l'instruction `USE surface` :

```
USE surface   ! assure la visiblité du module sans duplication de code
REAL :: b1, h1
b1 = 30.
h1 = 2.
PRINT *, ' Surface= ', surf(b1, h1)     ! arguments effectifs de même type
```

La compilation du module `surface` crée un fichier de module `surface.mod` (dont le nom ne dépend pas du nom du fichier source) qui sera consulté par le compilateur lorqu'il rencontrera l'instruction `USE surface` et qui assurera aux unités de programmes qui utilisent le module la visibilité de ce qu'il définit (déclarations, interfaces, ...). ⇐⚠

Si on place le module `surface` dans un fichier source séparé `f_surf.f90`, il faut le compiler (*avant* les unités de programme qui lui font appel), via la commande `gfortran -c f_surf.f90`, qui créera deux fichiers (*cf.* Figure 6.1, p. 66) :

- un fichier de module de suffixe `.mod` : `surface.mod` destiné au compilateur ;
- un fichier objet de suffixe `.o` : `f_surf.o` qui sera utilisé par l'éditeur de liens pour construire l'exécutable à partir des objets (via la commande `gfortran f_ppal.o f_surf.o -o f_ppal.x`).

Une autre méthode consisterait à lancer une seule commande :
`gfortran f_surf.f90 f_ppal.f90 -o f_ppal.x`
qui déclencherait la compilation (en compilant le module avant l'appelant), puis l'édition de liens.

6.4.2 Partage de données via des modules

La déclaration de variables partageables par plusieurs unités de programme, équivalent du `COMMON` du fortran 77, est possible à l'intérieur d'un module. Chaque unité de programme qui attachera ce module par l'instruction `USE` partagera alors ces variables, sauf si leur visibilité a été restreinte par l'attribut `PRIVATE` (*cf.* 6.4.3 p. 69).

Les données d'un module ne sont pas a priori permanentes sauf si :

- leur portée s'étend au programme principal via un `USE` ;
- leur portée a été étendue à plus d'une unité de compilation via au moins deux `USE` ;
- l'attribut `SAVE` a été précisé explicitement ou impliqué par une initialisation (*cf.* 2.2.3, p. 16).

Dans l'exemple suivant, si l'instruction `USE constantes` figurait dans le sous-programme `init` au lieu de figurer dans le module hôte `initialisation`, la visibilité de `pi` [25] serait restreinte à `init`. Dans ce cas, l'instruction `USE constantes` serait nécessaire dans le programme principal. ⇐⚠

```
MODULE constantes
  IMPLICIT NONE
  REAL, SAVE :: pi          ! variable statique
END MODULE constantes
MODULE initialisation
  USE constantes            ! permet la visibilité de la variable pi
  CONTAINS
    SUBROUTINE init         ! sans argument
      pi = 4. * atan(1.)    ! mais pi est déclaré dans le module constantes
    END SUBROUTINE init
END MODULE initialisation
PROGRAM principal
  ! USE constantes ! inutile car visibilité via le module initialisation
  USE initialisation ! rend explicite l'interface de la procédure de module init
  REAL :: y, x
  CALL init
  ...
  y = sin ( pi * x )
  ...
END PROGRAM principal
```

25. Noter que `pi` devrait de préférence être déclaré comme une constante nommée (avec l'attribut `PARAMETER`) initialisée lors de la déclaration dans un module sans procédure (*cf.* 2.2.3, p. 16).

Cette technique de partage des informations entre procédures est puissante, mais elle n'est pas aussi explicite que le passage par argument qui reste la méthode à privilégier. Elle reste cependant incontournable dans le cas d'appels de procédures imbriqués où la procédure intermédiaire ne peut pas voir les données à transmettre [26] : un exemple classique est l'application d'une procédure fonctionnelle (intégration, dérivation, ...) à une fonction d'une variable, qui possède des paramètres que la procédure fonctionnelle ignore alors qu'elle effectue des appels à la fonction. Le seul moyen de passer ces paramètres est de les partager via un module.

Dans l'exemple suivant, introduit à propos du passage d'argument de type procédure (*cf.* 6.5.3, p. 72), le paramètre `puissance` de la fonction effective `fonct` à intégrer ne peut pas être passé à la méthode d'intégration `integr`, mais doit être communiqué de l'appelant (ici le programme principal) à la fonction `fonct` à évaluer.

```
1  MODULE mod_fonct                  ! module définissant la fct effective
2   IMPLICIT NONE
3   INTEGER :: puissance             ! masqué à la procédure d'intégration
4   CONTAINS
5    FUNCTION fonct(x)               ! définition de la fonction effective
6      REAL, INTENT(IN)              :: x
7      REAL                          :: fonct
8      fonct = x**puissance          ! paramétré par puissance par exemple
9    END FUNCTION fonct
10 END MODULE mod_fonct
11 !
12 MODULE mod_integr                 ! module définissant le sous-programme
13  IMPLICIT NONE
14  CONTAINS
15   SUBROUTINE integr(f, xmin, xmax, somme, n_points)  ! f = argument muet
16     REAL, INTENT(IN)              :: xmin, xmax
17     REAL, INTENT(OUT)             :: somme
18     INTEGER, INTENT(IN)           :: n_points
19     INTERFACE                     ! interface de la fonction muette
20        FUNCTION f(x)
21          REAL, INTENT(IN)         :: x
22          REAL                     :: f
23        END FUNCTION f
24     END INTERFACE
25     REAL                          :: x, pas
26     INTEGER                       :: i
27     pas = (xmax - xmin) / n_points
28     somme = pas * (f(xmin) + f(xmax)) / 2.
29     DO i = 2, n_points
30        x = xmin + (i - 1) * pas
31        somme = somme + pas * f(x)
32     END DO
33   END SUBROUTINE integr
34 END MODULE mod_integr
35 PROGRAM principal
36   USE mod_fonct           ! accès au module de la fonction effective
37   USE mod_integr          ! accès au module du sous-programme
38   IMPLICIT NONE
39   REAL    :: x0 = 0., x1 = 1., s
40   INTEGER :: n = 100
41   puissance = 1           ! ne pas redéclarer: visibilité assurée par use
42   CALL integr(fonct, x0, x1, s, n)
43   WRITE(*,*) 'somme de f(x) = x : ', s
44   puissance = 2           ! partagé avec fonct, mais inconnu de integr
45   CALL integr(fonct, x0, x1, s, n)
```

26. La transmission des données s'effectue à l'insu de la procédure intermédiaire : celle qui l'appelle et celle qu'elle appelle se partagent les données grâce à un `USE` commun inconnu de la procédure intermédiaire !

```
46   WRITE(*,*) 'somme de f(x) = x*x : ', s
47 END PROGRAM principal
```

♠ 6.4.3 Éléments d'encapsulation

Les modules permettent de faciliter la réutilisation des codes et leur maintenance. Au sein d'une procédure que l'on souhaite partager, on peut distinguer :

- l'interface (qui doit rester parfaitement stable), qui doit être visible de ses procédures appelantes et doit donc être publique ;
- les détails de l'implémentation (qui peuvent évoluer) qu'il est souvent prudent de masquer aux utilisateurs, en les rendant privés.

Par défaut, les entités (types dérivés, variables, procédures) définies dans un module `mon_module` sont publiques et donc rendues accessibles (et, pour les variables, modifiables) par l'instruction `USE mon_module`. À l'intérieur d'un module, on peut utiliser la déclaration `PRIVATE` pour masquer toutes les entités qui suivent cette déclaration en dehors du module où elles sont déclarées. On dispose aussi des attributs `PUBLIC` et `PRIVATE` pour spécifier plus finement la visibilité des objets lors de leur déclaration.

Par ailleurs, on peut aussi, dans la procédure appelante, restreindre la visibilité des objets publics d'un module en spécifiant la liste exhaustive des objets auxquels on souhaite accéder [27] : ⇐ ♥

`USE` *`nom_de_module`*`, ONLY:` *`liste_d'objets`*

Ces deux approches ne sont pas destinées aux mêmes usages : la première méthode permet au concepteur de la procédure de restreindre la visibilité alors que la seconde permet à l'utilisateur de choisir les procédures dont il souhaite faire usage.

De plus, lorsque l'on réutilise des modules rédigés dans un autre contexte, on peut être confronté à des conflits de nommage des identificateurs. L'instruction `USE` permet le renommage des entités publiques du module qu'elle appelle, selon la syntaxe suivante, similaire à celle de l'association d'un pointeur (*cf.* 11.1.1, p. 121) : ⇐ ⚠

`USE` *`nom_de_module`*`,` *`nom_local`* `=>` *`nom_dans_le_module`*

Ce renommage permet aussi une sorte de paramétrage au niveau de la compilation (*cf.* l'exemple 2.3.4 p. 20).

Enfin, sans restreindre la visibilité d'une variable, on souhaite parfois réserver aux procédures du module où elles sont déclarées le droit de modifier leur valeur. La norme fortran 2003 a introduit l'attribut `PROTECTED` dans cet objectif. Le programme suivant : ⇐ f2003

```
1  MODULE m_protege
2  IMPLICIT NONE
3  INTEGER, PRIVATE   :: m ! non visible en dehors du module
4  INTEGER, PUBLIC    :: n ! visible et modifiable en dehors du module
5  INTEGER, PROTECTED :: p ! visible mais non modifiable en dehors du module
6  ! => oblige à passer par une des procédures du module pour le modifier
7  CONTAINS
8    SUBROUTINE sub1 ! pas de paramètres : m, n et p ont une portée suffisante
9      m = 1
10     n = 2
11     p = 3
12     WRITE(*,*) "m, n, p dans sub1 ", m, n, p
13   END SUBROUTINE sub1
14   SUBROUTINE sub2 ! pas de paramètres : m, n et p ont une portée suffisante
15     m = 10
16     n = 20
17     p = 30
18     WRITE(*,*) "m, n, p dans sub2 ", m, n, p
19   END SUBROUTINE sub2
20 END MODULE m_protege
21
```

27. Cette technique s'avère indispensable lorsque l'on utilise une bibliothèque : elle permet d'éviter les trop nombreux avertissements concernant les procédures de la bibliothèque non appelées.

```
22 PROGRAM t_protege
23   USE m_protege
24   CALL sub1
25   WRITE(*,*) "n, p dans l'appelant après appel de sub1 ", n, p
26   CALL sub2
27   WRITE(*,*) "n, p dans l'appelant après appel de sub2 ", n, p
28   ! m = -1 ! serait incorrect car m n'est pas visible ici (privé)
29   n = -2 ! autorisé car n est public dans le module
30   ! p = -3 ! serait incorrect car p est protégé => non modifiable hors du module
31   ! à comparer avec l'attribut intent(in) mais ici dans l'appelant
32   WRITE(*,*) "n, p dans l'appelant après affectation de n ", n, p
33 END PROGRAM t_protege
```

produit l'affichage :

```
m, n, p dans sub1  1 2 3
n, p dans l'appelant après appel de sub1  2 3
m, n, p dans sub2  10 20 30
n, p dans l'appelant après appel de sub2  20 30
n, p dans l'appelant après affectation de n  -2 30
```

f2003 6.4.4 L'instruction IMPORT dans les interfaces

Contrairement aux procédures, en fortran 95, les interfaces encapsulées dans un module n'ont pas de visibilité sur les entités déclarées (ou simplement visibles) dans le module qui les héberge. Le fortran 2003 a introduit l'instruction `IMPORT` qui permet dans une interface, d'accéder aux entités du module soit globalement si on ne spécifie pas de liste, soit seulement à celles spécifiées par la liste (voir par exemple les sous-programmes 12.4.2, p. 133).

f2003 6.4.5 Modules intrinsèques

La norme 2003 du fortran a introduit plusieurs *modules intrinsèques* dont :

- `IEEE_ARITHMETIC`, `IEEE_EXCEPTIONS`, et `IEEE_FEATURES` pour les questions de calcul numérique en flottant ;
- `ISO_FORTRAN_ENV`, pour l'interaction avec l'environnement système en particulier les entrées-sorties (*cf.* 5.3.1, p. 42), puis en fortran 2008, pour définir les paramètres `KIND` des sous-types (*cf.* 2.3.2, p. 19) et accéder aux informations sur le compilateur (*cf.* A.10, p. 148) ; f2008 ⇒
- et `ISO_C_BINDING`, pour l'inter-opérabilité avec le langage C (*cf.* chap. 12).

On peut accéder à ces modules avec l'instruction `USE, INTRINSIC ::` en priorité sur un module utilisateur homonyme.

6.5 Fonctionnalités avancées

6.5.1 Arguments optionnels

Lors de la déclaration d'une procédure, il est possible de préciser que certains arguments pourront être omis lors de certains appels, en leur spécifiant l'attribut `OPTIONAL`. Sauf à utiliser un passage d'arguments par mot-clef (*cf.* 6.5.2, p. 71), les arguments optionnels doivent être placés en fin de liste[28]. Lors de l'écriture de la procédure, il faut prévoir un traitement conditionnel suivant que l'argument a été ou non fourni lors de l'appel ; la fonction intrinsèque d'interrogation `PRESENT(<arg>)`, de résultat booléen, permet de tester si l'argument a été effectivement passé.

Par exemple, le sous-programme `integr` de calcul d'intégrale d'une fonction (pour le moment fixée et non paramétrée) sur un intervalle `[xmin, xmax]` passé en argument, comporte deux paramètres optionnels : le nombre de points d'évaluation (`n_points`) et la précision relative (`precision`) demandée pour le calcul.

28. Si tous les arguments sont omis, l'appel se fera sous la forme `CALL sub()` pour un sous-programme et `fct()` pour une fonction.

```
MODULE util
IMPLICIT NONE
CONTAINS
 SUBROUTINE integr(xmin, xmax, somme, precision, n_points)
  REAL, INTENT(IN)              :: xmin, xmax ! arguments d'entrée requis
  REAL, INTENT(OUT)             :: somme      ! argument de sortie requis
  REAL, INTENT(IN), OPTIONAL    :: precision  ! argument d'entrée optionnel
  INTEGER, INTENT(IN), OPTIONAL :: n_points   ! argument d'entrée optionnel
  !
  INTEGER                       :: points     ! variable locale indispensable
  ...
  IF( PRESENT(n_points) ) THEN
     points = n_points           ! utiliser la valeur transmise
  ELSE
     points = 200                ! valeur par défaut
  END IF
  ...
  somme = ...
 END SUBROUTINE integr
END MODULE util
!
PROGRAM principal
USE util
...
CALL integr(xmin1, xmax1, somme1, 1e-3, 100) ! tous les arguments sont passés
CALL integr(xmin2, xmax2, somme2, 1e-3)      ! n_points est omis
...
END PROGRAM principal
```

Remarques : dans cet exemple, la variable locale `points` est nécessaire, car `n_points`, déclaré comme argument d'entrée (`INTENT(IN)`), ne peut pas être modifié dans la procédure. Noter par ailleurs, qu'il faut conditionner toute référence au paramètre optionnel à sa fourniture effective et que la formulation suivante

```
IF( PRESENT(n_points) .AND. n_points > 1 ) points = n_points
```

est incorrecte car elle suppose que le second opérande du `.AND.` n'est pas évalué (*cf.* 3.3, p. 25) si le paramètre optionnel n'a pas été spécifié.

6.5.2 Transmission d'arguments par mot-clef

Dès le fortran 77, certaines procédures intrinsèques comme l'instruction `OPEN` par exemple autorisaient le passage d'argument par mot-clef, ce qui permettait de s'affranchir de l'ordre des arguments. Cependant le passage d'argument aux procédures définies par l'utilisateur restait nécessairement positionnel.

En fortran 90, on peut passer certains arguments par un mot-clef, en utilisant la syntaxe *<argument_muet> = <argument_effectif>*. Cette possibilité prend tout son sens quand la procédure possède des arguments optionnels. En effet, dans ce cas, le passage par mot-clef est le seul moyen de fournir un argument qui suit un argument optionnel non spécifié lors de l'appel [29]. Plus précisément, dès qu'un argument optionnel a été omis, les arguments qui le suivent ne peuvent plus être passés de façon positionnelle.

En reprenant l'exemple précédant (*cf.* 6.5.2, p. 71), on peut mixer les modes de passage d'arguments :

29. Un appel du type `f(x1, , x3)` n'est pas autorisé en fortran.

```
...
CALL integr(0., 1., precision=.1, n_points=100, somme=resultat)
!                   plus de passage positionnel à partir du troisième
! tous les arguments passés, mais en ordre modifié avec les mots-clefs
CALL integr(xmin2, xmax2, somme2, n_points=500)
! precision est omis, mais on veut passer n_points, donc par mot-clef car après
...
```

♠ 6.5.3 Noms de procédures passés en argument

La paramétrisation des sous-programmes conduit parfois à transmettre le nom d'une procédure en argument d'une autre procédure. Par exemple, un sous-programme d'intégration numérique `integr` doit accepter en argument le nom de la fonction à intégrer. Dans le sous-programe d'intégration, l'identificateur (muet) `f` de la fonction est nécessairement vu comme une fonction dès que le sous-programme invoque cette fonction. Mais, vu du programme appelant `principal`, il est nécessaire d'indiquer explicitement que l'argument *effectif* fournissant le nom de la fonction à intégrer n'est pas une simple variable.

Nous allons présenter successivement plusieurs solutions possibles des plus élémentaires (*cf.* p 72, puis p. 73), déconseillées, applicables aussi dans le cas de procédures non intégrées dans des modules, en passant par une méthode utilisant des modules et une interface (*cf.* p. 74) jusqu'à la solution la plus élaborée faisant appel à une interface abstraite qui sera présentée dans la section 6.5.5, p. 75.

- En fortran 77, on informe le compilateur du caractère particulier de cet argument, grâce à l'instruction `EXTERNAL` *<fonc>* [30], où *<fonc>* est la fonction à intégrer, insérée dans la procédure appelante [31].
- En fortran 90, il vaut mieux assurer la visiblité de l'interface de la fonction effectivement intégrée, à l'appelant, ici le programme `principal`. Cela est possible soit en déclarant dans l'appelant l'interface explicite de la fonction `fonct`, soit en créant un module `mod_fonct` avec cette fonction `fonct` en procédure interne de module et en déclarant `USE mod_fonct` dans le programme `principal`.

Version minimale du passage d'argument fonction

Le premier exemple présente une version minimale du programme `principal` avec des procédures `integr` et `fonct` externes. Seule l'interface explicite de la fonction effective `fonct` (lignes 7 à 11) est nécessaire dans le programme `principal`.

```
1  ! version avec procédures externes et interface minimale
2  PROGRAM principal
3    IMPLICIT NONE
4    REAL :: x0 = 0., x1 = 1., s
5    INTEGER :: n = 100
6    INTERFACE
7      FUNCTION fonct(x)          ! interface obligatoire de la fonction effective
8        IMPLICIT NONE
9        REAL, INTENT(IN)        :: x
10       REAL                    :: fonct
11     END FUNCTION fonct
12   END INTERFACE
13   CALL integr(fonct, x0, x1, s, n)
14   WRITE(*,*) 'somme = ', s
15 END PROGRAM principal
16 ! sous-programme externe
17 SUBROUTINE integr(f, xmin, xmax, somme, n_points)  ! f = argument muet
18   IMPLICIT NONE
19   REAL, INTENT(IN)             :: xmin, xmax
20   REAL, INTENT(OUT)            :: somme
21   INTEGER, INTENT(IN)          :: n_points
22   ! déclaration de f nécessaire, sauf si l'interface de f est connue
```

30. Sauf dans le cas très rare où la fonction utilisée est une fonction intrinsèque du fortran ; alors il faut déclarer `INTRINSIC` *<fonc>*

31. Si `IMPLICIT NONE` a été déclaré dans l'appelant, il faut aussi déclarer le type du résultat si la procédure est une fonction.

```
23    REAL                          :: f
24    REAL                          :: x, pas
25    INTEGER                       :: i
26    pas = (xmax - xmin) / n_points
27    somme = pas * (f(xmin) + f(xmax)) / 2.
28    DO i = 2, n_points
29       x = xmin + (i - 1) * pas
30       somme = somme + pas * f(x)
31    END DO
32  END SUBROUTINE integr
33  ! fonction externe
34  FUNCTION fonct(x)               ! définition de la fonction effective
35    IMPLICIT NONE
36    REAL, INTENT(IN)              :: x
37    REAL                          :: fonct
38    fonct = x                     ! par exemple
39  END FUNCTION fonct
```

Version avec les trois interfaces

L'exemple suivant présente encore une version du programme `principal` avec des procédures externes, mais qui comporte aussi les interfaces explicites du sous-programme `integr` (lignes 12 à 18) dans le programme principal et de la fonction symbolique `f` (lignes 29 à 34) au sein du sous-programme `integr`. Ces deux interfaces sont facultatives ici, mais vivement conseillées.

```
1   ! version avec procédures externes et interfaces
2   PROGRAM principal
3     IMPLICIT NONE
4     REAL :: x0 = 0., x1 = 1., s
5     INTEGER :: n = 100
6     INTERFACE
7       FUNCTION fonct(x)           ! interface obligatoire de la fonction effective
8          IMPLICIT NONE
9          REAL, INTENT(IN)         :: x
10         REAL                     :: fonct
11      END FUNCTION fonct
12      SUBROUTINE integr(f, xmin, xmax, somme, n_points) ! interface facultative
13         IMPLICIT NONE
14         REAL, INTENT(IN)         :: xmin, xmax
15         REAL, INTENT(OUT)        :: somme
16         INTEGER, INTENT(IN)      :: n_points
17         REAL                     :: f
18      END SUBROUTINE integr
19    END INTERFACE
20    CALL integr(fonct, x0, x1, s, n)
21    WRITE(*,*) 'somme = ', s
22  END PROGRAM principal
23  ! sous-programme externe
24  SUBROUTINE integr(f, xmin, xmax, somme, n_points)  ! f = argument muet
25    IMPLICIT NONE
26    REAL, INTENT(IN)              :: xmin, xmax
27    REAL, INTENT(OUT)             :: somme
28    INTEGER, INTENT(IN)           :: n_points
29      INTERFACE                   ! interface facultative de la fonction muette
30         FUNCTION f(x)
31            REAL, INTENT(IN)      :: x
32            REAL                  :: f
33         END FUNCTION f
34      END INTERFACE
35    REAL                          :: x, pas
36    INTEGER                       :: i
37    pas = (xmax - xmin) / n_points
38    somme = pas * (f(xmin) + f(xmax)) / 2.
39    DO i = 2, n_points
40       x = xmin + (i - 1) * pas
41       somme = somme + pas * f(x)
42    END DO
43  ! fonction externe
44  END SUBROUTINE integr
45  ! fonction externe
46  FUNCTION fonct(x)               ! définition de la fonction effective
47    IMPLICIT NONE
48    REAL, INTENT(IN)              :: x
49    REAL                          :: fonct
50    fonct = x                     ! par exmple
51  END FUNCTION fonct
```

Version avec modules et une seule interface explicite

♥ ⇒ Enfin, pour éviter la lourdeur des interfaces explicites, on préfère souvent intégrer les procédures dans des modules. Avec les modules `mod_fonct` (lignes 2 à 11) pour la fonction effective et `mod_integr` (lignes 13 à 38) pour le sous-programme, seule demeure l'interface explicite (lignes 23 à 28) de la fonction symbolique [32] `f`.

```
1  ! version avec modules
2  MODULE mod_fonct                  ! module définissant la fct effective
3   CONTAINS
4    ! fonction interne de module
5    FUNCTION fonct(x)               ! définition de la fonction effective
6      IMPLICIT NONE
7      REAL, INTENT(IN)              :: x
8      REAL                          :: fonct
9      fonct = x                     ! par exemple
10   END FUNCTION fonct
11 END MODULE mod_fonct
12 !
13 MODULE mod_integr                 ! module définissant le sous-programme
14  CONTAINS
15   ! sous-programme interne de module
16   SUBROUTINE integr(f, xmin, xmax, somme, n_points)  ! f = argument muet
17     IMPLICIT NONE
18     REAL, INTENT(IN)              :: xmin, xmax
19     REAL, INTENT(OUT)             :: somme
20     INTEGER, INTENT(IN)           :: n_points
21     ! REAL                        :: f
22     ! cette déclaration peut remplacer l'interface explicite de f qui suit
23     INTERFACE                  ! interface conseillée de la fonction muette
24        FUNCTION f(x)
25          REAL, INTENT(IN)        :: x
26          REAL                    :: f
27        END FUNCTION f
28     END INTERFACE
29     REAL                          :: x, pas
30     INTEGER                       :: i
31     pas = (xmax - xmin) / n_points
32     somme = pas * (f(xmin) + f(xmax)) / 2.
33     DO i = 2, n_points
34        x = xmin + (i - 1) * pas
35        somme = somme + pas * f(x)
36     END DO
37   END SUBROUTINE integr
38 END MODULE mod_integr
39 ! version du programme principal avec modules
40 PROGRAM principal
41   USE mod_fonct           ! chargement du module de la fonction effective
42   USE mod_integr          ! chargement du module du sous-programme
43   IMPLICIT NONE
44   REAL :: x0 = 0., x1 = 1., s
45   INTEGER :: n = 100
46   CALL integr(fonct, x0, x1, s, n)
47   WRITE(*,*) 'somme = ', s
48 END PROGRAM principal
```

32. on pourrait la remplacer par une simple déclaration de `f`, ligne 21, moins précise, car elle ne permet pas de vérifier le type des arguments.

f2003 6.5.4 La déclaration PROCEDURE

En fortran 2003, si l'interface d'une procédure *fmodele* est visible dans une unité de compilation, on peut s'en servir comme modèle pour déclarer l'interface d'autres procédures de même interface grâce à la déclaration PROCEDURE, avec pour argument le nom de la procédure modèle.

```
PROCEDURE(fmodele) :: f1, f2
```

Cette déclaration admet des attributs optionnels, en particulier POINTER, PUBLIC ou PRIVATE. Cette possibilité, utilisée notamment pour les pointeurs de procédures (*cf.* 11.4, p. 128) et les procédures passées en argument est le plus souvent associée à la déclaration d'une interface abstraite de préférence à un modèle. C'est dans ce cadre que sont présentés les exemples (*cf.* 6.5.5, p. 75). Bien entendu, PROCEDURE ne peut pas être utilisé pour des procédures génériques : cette déclaration d'interface est réservée aux procédures spécifiques (*cf.* 10.1, p. 113).

Noter que PROCEDURE permet aussi de déclarer des procédures d'interface implicite avec un argument vide (équivalent d'un EXTERNAL), ou d'interface incomplète avec seulement le type de retour dans le cas d'une fonction (par exemple `PROCEDURE(REAL) :: freel` pour une fonction à valeur de type REAL sans précision sur les arguments).

f2003 6.5.5 Interfaces abstraites

Lorsque l'on doit déclarer l'interface commune de plusieurs procédures, il est préférable de déclarer une seule fois cette interface, dans un bloc dit d'*interface abstraite*, en la nommant tel un type particulier. On peut ensuite faire référence à cette interface nommée via la déclaration PROCEDURE pour déclarer de façon beaucoup plus synthétique les interfaces des procédures effectives.

```
1  MODULE abstr_interf
2  IMPLICIT NONE
3  ! bloc d'interface abstraite avec des interfaces nommées
4  ABSTRACT INTERFACE
5    FUNCTION fr_de_r(x)
6    ! fonction réelle d'un réel
7      REAL :: fr_de_r
8      REAL, INTENT(in) :: x
9    END FUNCTION fr_de_r
10   FUNCTION fr_de_2r(x, y)
11   ! fonction réelle de 2 réels
12     REAL :: fr_de_2r
13     REAL, INTENT(in) :: x, y
14   END FUNCTION fr_de_2r
15 END INTERFACE
16 END MODULE abstr_interf
```

Ces interfaces abstraites, sortes de « patrons » de procédures pourront être invoquées pour rendre explicite l'interface de procédures d'interface a priori implicite respectant ce modèle (par exemples définies hors module ou dans un module non visible du côté appelant). Il suffit de déclarer PROCEDURE avec comme argument le nom de l'interface abstraite pour expliciter l'interface.

Dans l'exemple suivant, le non-respect des interfaces est détecté dès la compilation :

```
! fonctions externes hors module
! => interface implicite
FUNCTION f1(x)
  IMPLICIT NONE
  REAL :: f1
  REAL, INTENT(in)  :: x
  f1 = x
END FUNCTION f1
FUNCTION f2(x, y)
  IMPLICIT NONE
  REAL :: f2
  REAL, INTENT(in)  :: x, y
  f2 = x*y
END FUNCTION f2
```

```
PROGRAM t_abstract
USE abstr_interf
IMPLICIT NONE
PROCEDURE(fr_de_r) :: f1
! mais interface rendue ainsi visible
PROCEDURE(fr_de_2r) :: f2
REAL :: x, y
INTEGER :: i
x = 10
y = 5
WRITE(*,*)  x, f1(x)
WRITE(*,*)  i, f1(i) ! => erreur de compilation
! Error: Type mismatch in parameter 'x'
! Passing INTEGER(4) to REAL(4)
WRITE(*,*)  x, y, f2(x, y)
WRITE(*,*)  f2(x) ! => erreur de compilation
! Error: Missing actual argument for argument 'y'
END PROGRAM t_abstract
```

Les interfaces abstraites prennent tout leur intérêt dans les cas où plusieurs procédures de même interface sont susceptibles d'être utilisées :

- lorsqu'une méthode s'applique à une procédure passée en argument (*cf.* 6.5.3, p. 72) ;
- lorsqu'on utilise des pointeurs de procédures (*cf.* 11.4, p. 128) pour une partie de code pouvant s'appliquer à plusieurs procédures.

L'exemple de la méthode d'intégration `integr` prend alors la forme suivante :

```
! version avec interface abstraite pour l'argument de la méthode
MODULE mod_abstract_fr            ! module pour l'interface abstraite
  ABSTRACT INTERFACE              ! bloc d'interface abstraite
     FUNCTION fr(x)               ! le modèle
       REAL, INTENT(IN)     :: x
       REAL                 :: fr
     END FUNCTION fr
  END INTERFACE
END MODULE mod_abstract_fr
MODULE mod_integr
 USE mod_abstract_fr               ! assure la visibilité du modèle fr
 CONTAINS
  SUBROUTINE integr(f, xmin, xmax, somme, n_points)  ! f = argument muet
    IMPLICIT NONE
    PROCEDURE(fr)                :: f ! f est une procédure d'interface fr
    REAL, INTENT(IN)             :: xmin, xmax
    REAL, INTENT(OUT)            :: somme
    INTEGER, INTENT(IN)          :: n_points
    REAL                         :: x, pas
    INTEGER                      :: i
    pas = (xmax - xmin) / n_points
    somme = pas * (f(xmin) + f(xmax)) / 2.
    DO i = 2, n_points
       x = xmin + (i - 1) * pas
       somme = somme + pas * f(x)
    END DO
  END SUBROUTINE integr
END MODULE mod_integr
!
MODULE mod_fonct                  ! module définissant la fct effective
 CONTAINS
  FUNCTION fonct(x)               ! définition de la fonction effective
    IMPLICIT NONE
    REAL, INTENT(IN)             :: x
    REAL                         :: fonct
    fonct = x                    ! par exemple
  END FUNCTION fonct
END MODULE mod_fonct
```

```
39 PROGRAM principal
40   USE mod_fonct    ! visibilité de l'interface de la fonction effective
41   USE mod_integr   ! visibilité de l'interface de la méthode
42   IMPLICIT NONE
43   REAL :: x0 = 0., x1 = 1., s
44   INTEGER :: n = 100
45   CALL integr(fonct, x0, x1, s, n)
46   WRITE(*,*) 'somme = ', s
47 END PROGRAM principal
```

6.5.6 Procédures récursives

Lorsqu'une procédure s'appelle elle-même, de façon *directe* ou *indirecte* (au travers d'appels d'autres procédures), on la qualifie de *récursive*. Si beaucoup d'algorithmes se codent de façon élégante sous forme récursive, il est souvent plus efficace en termes de ressources de préférer un codage non-récursif. Mais les procédures récursives restent incontournables pour agir sur des structures comme les listes chaînées (*cf.* 11.5, p. 128).

Contrairement au fortran 77, le fortran 90 permet la récursivité directe ou indirecte des procédures, par la déclaration explicite via le mot-clef `RECURSIVE`.

Mais, dans le cas d'une fonction récursive, le nom de la fonction ne peut plus être utilisé pour désigner le résultat : on doit alors introduire, après le mot-clef `RESULT`[33], le nom de la variable qui stockera le résultat. Dans une procédure récursive, une variable avec l'attribut `SAVE` (*cf.* 6.1.2, p. 59) est partagée par toutes les instances de la procédure, alors que ses variables automatiques sont spécifiques de chaque instance. ⇐⚠

L'archétype de la *récursivité directe* est le calcul de la factorielle : $n! = n \times (n-1)!$ avec $0! = 1$, qu'il est imprudent de programmer en type entier sur 32 bits à cause du dépassement de capacité dès n dépasse 12. ⇐⚠

```
INTEGER RECURSIVE FUNCTION fact(n) RESULT(factorielle)
! le type entier s'applique en fait au résultat : factorielle
IMPLICIT NONE
INTEGER, INTENT(IN)     :: n
IF ( n > 0 ) THEN
   factorielle = n * fact(n-1)   ! provoque un nouvel appel de fact
ELSE
   factorielle = 1               ! ne pas oublier d'arrêter la récursion
END IF
END FUNCTION fact
```

On peut aussi programmer la factorielle comme sous-programme récursif :

```
RECURSIVE SUBROUTINE fact(n, factorielle)
IMPLICIT NONE
INTEGER, INTENT(IN)   :: n
INTEGER, INTENT(OUT)  :: factorielle
IF ( n > 0 ) THEN
   CALL fact(n-1, factorielle)   ! appel récursif
   factorielle = n * factorielle
ELSE
   factorielle = 1
END IF
END SUBROUTINE fact
```

33. L'utilisation de la clause `RESULT` est aussi possible pour des fonctions non récursives.

On peut rencontrer la *récursivité indirecte* lors des calculs numériques d'intégrale double, si on utilise la même procédure `integr` (*cf.* 6.5.3, p. 72) pour intégrer selon chacune des variables : la fonction de y, $F(y) = \int_{x_0}^{x_1} f(x,y)\,dx$, fait appel, dans son évaluation numérique pour chaque valeur de y, à la procédure `integr`, et l'intégrale double sera évaluée en appliquant `integr` à $F(y)$. La ⚠⇒ procédure `integr` devra donc être déclarée récursive. De plus, il faudra masquer le paramètre y lors de l'intégration en x pour simuler le passage d'une fonction d'une seule variable à la procédure d'intégration : il sera transmis à F par partage via un module (*cf.* 6.4.2, p. 68).

```
1  MODULE mod_fonct                  ! module définissant la fct effective
2   IMPLICIT NONE
3   REAL :: yy                       ! à y fixé
4   CONTAINS
5    FUNCTION fonct(x)               ! définition de la fonction effective de x
6      REAL, INTENT(IN)              :: x
7      REAL                          :: fonct
8      fonct = x**2 - yy**2
9    END FUNCTION fonct
10 END MODULE mod_fonct
11 !
12 MODULE mod_param_y                ! module des paramètres cachés à fonct
13  IMPLICIT NONE
14  REAL :: x0, x1                   ! bornes en x
15  INTEGER :: nx                    ! nombre de points en x
16 END MODULE mod_param_y
17 !
18 MODULE mod_integr                 ! module définissant le sous-programme
19  IMPLICIT NONE
20  CONTAINS
21   RECURSIVE SUBROUTINE integr(f, xmin, xmax, somme, n_points)
22     ! f = argument muet de type fonction
23     REAL, INTENT(IN)              :: xmin, xmax ! bornes
24     REAL, INTENT(OUT)             :: somme      ! résultat
25     INTEGER, INTENT(IN)           :: n_points   ! nombre de points
26     INTERFACE                     ! interface de la fonction muette
27        FUNCTION f(x)
28          REAL, INTENT(IN)         :: x
29          REAL                     :: f
30        END FUNCTION f
31     END INTERFACE
32     REAL                          :: x, pas
33     INTEGER                       :: i
34     pas = (xmax - xmin) / n_points
35     somme = pas * (f(xmin) + f(xmax)) / 2.
36     DO i = 2, n_points
37        x = xmin + (i - 1) * pas
38        somme = somme + pas * f(x)
39     END DO
40   END SUBROUTINE integr
41 END MODULE mod_integr
42
43 MODULE mod_fy
44  USE mod_param_y                  ! module des paramètres cachés à fonct
45  USE mod_fonct, only: yy, fonct
46  USE mod_integr                   ! module définissant le sous-programme
47  CONTAINS
48   FUNCTION fy(y)                  ! intégrale en x de la fonction effective
49     REAL                          :: fy
50     REAL, INTENT(IN)              :: y
51     REAL                          :: s
52     yy = y                        ! transfert de y à fonct via mod_fonct
```

```
53      CALL integr(fonct, x0, x1, s, nx) ! intégration en x à y fixé
54      fy = s
55    END FUNCTION fy
56  END MODULE mod_fy
57  !
58  PROGRAM principal
59    USE mod_param_y              ! accès aux paramètres utilisés par fy
60    USE mod_fy, only:x0, x1, fy, nx ! accès au module de la fct effective de y
61    USE mod_integr               ! accès au module d'intégration 1D
62    REAL    :: y0 = 0., y1 = 1., s
63    INTEGER :: ny = 100
64    ! paramètres de l'intégration en x : passés via mod_fy (ne pas redéclarer)
65    x0 = 0.
66    x1 = 1.
67    nx = 50
68    CALL integr(fy, y0, y1, s, ny)
69    WRITE(*,*) 'somme de f(x, y) = x**2 -y**2 : ', s
70  END PROGRAM principal
```

f95 6.5.7 Procédures pures

Dans certaines circonstances de parallélisation des codes, comme à l'intérieur d'une structure `FORALL` (*cf.* 7.4.6, p. 90), seules sont admises les procédures sans effet de bord. Un attribut `PURE` permet de certifier qu'une procédure est sans effet de bord, c'est-à-dire :

- qu'elle ne modifie pas de variables non locales : en particulier, s'il s'agit d'une fonction, tous ses arguments doivent posséder l'attribut `INTENT(IN)` ;
- qu'elle ne modifie pas de variables non locales soit visibles car la procédure est interne, soit rendues visibles par un `USE` ;
- qu'elle ne fait pas de lecture ou d'écriture sur un fichier externe : en particulier, elle ne peut rien lire au clavier ou afficher ;
- qu'elle ne contient pas de variable locale avec l'attribut `SAVE` : en particulier, pas de variable initialisée à la déclaration (*cf.* 2.2.3, p. 16) ;
- qu'elle ne comporte pas d'instruction `STOP`.

Ces conditions imposent qu'une procédure pure n'appelle que des procédures pures. Toutes les procédures intrinsèques sont pures. Les procédures élémentaires (*cf.* 6.5.8, p. 79) possèdent automatiquement l'attribut `PURE`.

f95 6.5.8 Procédures élémentaires

Une procédure est qualifiée d'élémentaire (attribut `ELEMENTAL`) si elle peut être appelée avec des arguments tableaux conformes de la même façon qu'avec des arguments scalaires. La majorité des procédures intrinsèques (*cf.* annexe A, p. 140) sont élémentaires.

Une procédure élémentaire est automatiquement pure, mais de plus tous ses arguments muets (ainsi que la valeur renvoyée pour une fonction) doivent être des scalaires sans l'attribut `POINTER`. Une procédure récursive (*cf.* 6.5.6, p. 77) ne peut pas être élémentaire. Une procédure élémentaire non-intrinsèque ne peut pas être passée en argument *effectif* d'une autre procédure.

```
ELEMENTAL REAL FUNCTION my_log(x) ! une fonction log protégée pour les x<=0
  REAL, INTENT(IN) :: x
  IF ( x > 0 ) THEN
    my_log = LOG (x)
  ELSE
    my_log = -1000.
  END IF
END FUNCTION my_log
```

Chapitre 7

Tableaux

7.1 Généralités

Un *tableau* (array) est un ensemble « rectangulaire [1] » d'éléments de même type [2] (plus précisément de même variante de type), repérés au moyen d'indices entiers. Par opposition, on qualifiera de *scalaire* un élément du tableau. Les éléments d'un tableau sont rangés selon un ou plusieurs « axes » appelés *dimensions* du tableau. Dans les tableaux à une dimension (qui permettent de représenter par exemple des vecteurs au sens mathématique du terme), chaque élément est repéré par un seul entier, mais le fortran accepte des tableaux jusqu'à 7 dimensions [3], où chaque élément est désigné par un 7-uplet d'entiers [4].

7.1.1 Terminologie des tableaux

- *rang* (rank) d'un tableau : nombre de ses dimensions
- *étendue* (extent) d'un tableau selon une de ses dimensions : nombre d'éléments dans cette dimension
- *bornes* (bounds) d'un tableau selon une de ses dimensions : limites inférieure et supérieure des indices dans cette dimension. La borne inférieure par défaut vaut 1 [5].
- *profil* (shape) d'un tableau : vecteur dont les composantes sont les étendues du tableau selon ses dimensions ; son étendue (sa taille) est le rang du tableau.
- *taille* (size) d'un tableau : nombre total des éléments qui le constituent, c'est-à-dire le produit des éléments du vecteur que constitue son profil.
- deux tableaux sont dits *conformants* (conformable) s'ils ont le même profil [6].

La déclaration d'un tableau s'effectue grâce à l'attribut `DIMENSION` qui indique le profil du tableau, ou éventuellement les bornes, séparées par le symbole « `:` ».

```
INTEGER, DIMENSION(5)    :: vecteur  ! tableau à une dimension de 5 entiers
                                     ! indicés de 1 à 5
REAL, DIMENSION(-2:2)    :: table    ! tableau à une dimension de 5 réels
                                     ! indicés de -2 à +2
```

1. L'assemblage « non-rectangulaire » d'éléments du même type est possible grâce aux pointeurs, *cf.* 11.3, p. 125.
2. L'assemblage d'éléments de types différents est possible au sein d'un type dérivé ou structure, *cf.* chapitre 9, p. 104. Mais les éléments constitutifs d'un tableau peuvent être des structures, et ... les éléments de ces structures peuvent eux-mêmes comporter des tableaux !
3. Cette limite est portée à 15 en fortran 2008.
4. Certains termes employés en informatique à propos des tableaux peuvent avoir un sens profondément différent de leur sens en mathématiques dans le domaine des espaces vectoriels. C'est le cas, par exemple, de la notion de dimension : un vecteur à trois dimensions de la géométrie dans l'espace peut être représenté par un tableau monodimensionnel, de rang un et de taille trois.
5. Ne pas confondre avec le C où l'indexation des tableaux commence à zéro.
6. Noter que deux tableaux conformants n'ont pas forcément les mêmes bornes.

```
REAL, DIMENSION(2,3)    :: m        ! tableau à 2 dimensions de 2 x 3 = 6 réels
                                    ! son profil a 2 éléments : 2 et 3
REAL, DIMENSION(2,-1:1) :: mat      ! profil 2 x 3 conformant avec m
```

Les tableaux `vecteur` et `table` sont de dimension 1 et ont même profil : ils sont conformants.

Un élément (scalaire) d'un tableau est désigné en précisant entre parenthèses les indices selon toutes ses dimensions, séparés par des virgules : `vecteur(2)`, `table(-1)` et `m(2,2)` sont des éléments des tableaux définis précédemment. Cependant, le fortran distingue un tableau monodimensionnel à un seul élément de l'élément scalaire qu'il contient :

```
INTEGER, DIMENSION(1)  :: t        ! tableau à un élément
INTEGER                :: i        ! entier
i = t(1)                           ! affectation autorisée
i = t                              ! affectation illicite
```

7.1.2 Ordre des éléments dans un tableau multidimensionnel

Le caractère multidimensionnel des tableaux n'est pas respecté dans la mémoire des calculateurs, où les tableaux sont en général stockés dans une mémoire à un seul indice d'adressage. En fortran, l'ordre « classiquement » utilisé pour stocker les éléments des tableaux est celui où *le premier indice varie le plus vite*[7], c'est-à-dire que deux éléments dont seul le premier indice diffère d'une unité sont contigus en mémoire.

Pour des tableaux de rang 2, cela signifie que le rangement s'effectue colonne par colonne[8]. Par exemple, la matrice m de 2 lignes par 3 colonnes (au sens mathématique), est stockée dans le tableau `m` dimensionné par l'attribut `DIMENSION(2,3)` de la manière suivante :

matrice

$$\begin{pmatrix} \boxed{1}\ m_{11} & \boxed{3}\ m_{12} & \boxed{5}\ m_{13} \\ \boxed{2}\ m_{21} & \boxed{4}\ m_{22} & \boxed{6}\ m_{23} \end{pmatrix}$$

tableau associé

1	2	3	4	5	6
m(1,1)	m(2,1)	m(1,2)	m(2,2)	m(1,3)	m(2,3)

Mais la norme fortran 90 ne spécifie pas l'ordre de rangement en mémoire des éléments d'un tableau, laissé à la disposition du compilateur, entre autres pour des raisons d'efficacité de calcul suivant les processeurs utilisés : on ne pourra donc pas s'appuyer sur cet ordre de stockage pour le passage de tableaux multidimensionnels en argument de procédures[9].

Cependant, cet ordre est respecté dans les instructions d'entrée-sortie et, par exemple, par la fonction `RESHAPE` (*cf.* 7.4.3, p. 87) de restructuration des tableaux.

Un exemple de lecture erronée de matrice

En particulier, une lecture globale de matrice dans un fichier par un seul ordre `READ` (sans boucle implicite) stockera les éléments contigus (sur une même ligne) dans le fichier en incrémentant l'indice le plus à gauche du tableau de rang 2, c'est-à-dire... celui des lignes ! Dans le cas d'une matrice carrée, cette lecture remplira le tableau avec la *transposée* de la matrice figurant sous la forme habituelle dans le fichier. ⇐⚠

7. Contrairement au choix fait en langage C, où l'indice qui varie le plus rapidement est celui de la dernière dimension. D'ailleurs, en C, il n'y a pas, à proprement parler, de tableaux multidimensionnels, mais des tableaux de tableaux. Cette représentation implique une hiérarchie qui, par exemple dans le cas des tableaux 2D, permet de désigner une ligne, mais pas une colonne de matrice.

8. On suppose ici que, dans un tableau de rang 2 représentant un objet mathématique de type matrice, on attribue au premier indice le rôle de numéro de ligne et au deuxième celui de numéro de colonne. Cette convention, qui semble naturelle, peut être inversée pour des raisons de performance des codes. Par exemple, dans des boucles imbriquées portant sur les lignes et les colonnes, il est plus efficace d'accéder à des éléments contigus en mémoire dans la boucle interne, qui doit donc porter sur l'indice rapide, c'est-à-dire le premier.

9. En fortran 77, où l'ordre de stockage était garanti, on mettait à profit la disposition des éléments des tableaux pour passer des sections de tableaux multidimensionnels en tant qu'arguments effectifs à des procédures dans lesquelles l'argument muet était déclaré monodimensionnel : il suffisait que la section de tableau s'étende uniquement selon la dernière dimension du tableau.

Par exemple, si le fichier `mat2x2.dat` contient les deux lignes $\begin{array}{|cc|}\hline 11 & 12 \\ 21 & 22 \\ \hline\end{array}$,

```
INTEGER, DIMENSION(2,2)          :: mat          ! matrice 2 lignes x 2 colonnes
...
OPEN(10, file='mat2x2.dat', STATUS='old')    ! ouverture du fichier en lecture
READ(10, *)  mat(:,:)                        ! lecture globale en format libre
CLOSE(10)
WRITE(*, *)  mat(:,:)                        ! affichage global en format libre
```

les ordres précédents rempliront le tableau `mat` avec : $\begin{array}{|cc|}\hline 11 & 21 \\ 12 & 22 \\ \hline\end{array}$; en particulier `mat(1,2) = 21` et `mat(2,1) = 12`. Mais l'affichage global en format libre reproduira (sur une seule ligne) l'ordre du fichier initial, `11 12 21 22` , bien que le premier indice du tableau `mat` soit celui des *colonnes*.

Un exemple de lecture correcte de matrice

Pour lire la matrice en affectant chaque ligne du fichier à une ligne (au sens mathématique) de la matrice, on doit expliciter la boucle sur les lignes, alors que celle sur les colonnes peut être implicite ; on fera de même pour afficher le tableau selon la disposition mathématique de la matrice (*cf.* 5.4.4, p. 52). Par exemple, pour lire une matrice 2x3 d'entiers et l'afficher :

```
INTEGER, PARAMETER               :: nl=2, nc=3 ! constantes nommées
INTEGER, DIMENSION(nl, nc)       :: matrice    ! matrice 2 lignes x 3 colonnes
INTEGER                          :: ligne, colonne
...
OPEN(10, file='mat2x3.dat', STATUS='old')    ! ouverture du fichier en lecture
DO ligne = 1, nl
  READ(10, *)  matrice(ligne,:)              ! lecture ligne par ligne
END DO
CLOSE(10)
DO ligne = 1, nl
  WRITE(*, *)  matrice(ligne,:)              ! affichage ligne par ligne
END DO
```

7.1.3 Constructeurs de tableaux

Lors de sa déclaration, il est possible d'initialiser un tableau *de rang 1* grâce à un *constructeur de tableau* qui est un vecteur encadré par les délimiteurs `(/` et `/)`, ou (en fortran 2003) les délimiteurs
f2003 ⇒ `[` et `]`, et dont les éléments peuvent être des scalaires ou des vecteurs, tous du même type, séparés par des virgules. Le vecteur pourra être produit par une ou des boucles implicites.

```
INTEGER, DIMENSION(3)    :: vecteur = (/ 3, 5, 1 /)
INTEGER, DIMENSION(3)    :: vecteur = [ 3, 5, 1 ] ! en fortran 2003
INTEGER                  :: i
INTEGER, DIMENSION(7)    :: v = (/ (2*i, i=1, 4), vecteur /) ! boucle implicite
```

Le tableau `v` contient alors les entiers suivants : 2, 4, 6, 8, 3, 5 et 1.

⚠⇒ Pour les tableaux de rang supérieur, le constructeur n'est pas autorisé : on pourra utiliser la fonction `RESHAPE` (*cf.* 7.4.3, p. 87) qui restructure le tableau mono-dimensionnel `v` en tableau multi-dimensionnel `matrice` suivant le vecteur de profil `profil`.

7.2 Sections de tableaux

Une *section de tableau* est un sous-tableau, de rang et d'étendues inférieurs ou égaux à ceux du tableau dont il est extrait.

7.2.1 Sections régulières

Une *section régulière* (regular section) est un sous-tableau dont les indices dans le tableau initial sont en progression arithmétique. Une section régulière peut donc être définie, pour chaque dimension, par un triplet `<debut>:<fin>:<pas>`[10]. Chacun des éléments du triplet est optionnel et les valeurs par défaut sont :

- `<debut>` : la borne inférieure selon la dimension, telle qu'elle a été spécifiée lors de la déclaration (donc 1 si elle n'a pas été précisée dans la déclaration)
- `<fin>` : la borne supérieure selon la dimension, telle qu'elle a été spécifiée lors de la déclaration
- `<pas>` : 1

Noter que le pas peut être négatif, ce qui permet d'inverser l'ordre des éléments. Comme dans les boucles `DO`, si le pas est différent de 1, il est possible que la borne finale ne soit pas atteinte.

```
REAL, DIMENSION(3,6)   :: matrice          ! 3 lignes par 6 colonnes (rang 2)
REAL, DIMENSION(3)     :: vecteur          ! 3 éléments ligne ou colonne (rang 1)
REAL, DIMENSION(3,3)   :: sous_matrice     ! 3 lignes par 3 colonnes
vecteur(:) = matrice(:,5)                  ! vecteur est la 5ème colonne
vecteur(:) = matrice(1,1:3)                ! extrait de la 1ère ligne
sous_matrice(:,:) = matrice(:,2:6:2)       ! extraction des colonnes 2, 4 et 6
                                           ! de matrice
```

`matrice(1:2,2:6:2)`

	x		x		x
	x		x		x

`matrice(::2,2:6:2)`

	x		x		x
	x		x		x

`matrice(2:,3::2)`

		x		x	
		x		x	

Remarque : à chaque fois que l'on fixe un indice, on diminue d'une unité le rang du tableau, mais si on restreint l'intervalle d'un des indices de façon à ne sélectionner qu'une valeur, le rang reste inchangé. Par exemple, si on déclare un tableau de rang deux par : ⇐⚠
`REAL, DIMENSION(3,6) :: matrice`
la section de tableau `matrice(2,:)` est un tableau de rang 1[11] à 6 éléments ; mais la section de tableau `matrice(2:2,:)` reste un tableau de rang 2 de profil `(/1, 6/)` contenant les mêmes 6 éléments.

♠ 7.2.2 Sections non-régulières

Les *sections non régulières* sont obtenues par indexation indirecte, grâce à un vecteur d'indices.

```
INTEGER, DIMENSION(3)    :: vecteur_indice = (/ 3, 5, 1 /)
INTEGER, DIMENSION(4,6)  :: m
INTEGER, DIMENSION(2,3)  :: sous_matrice
sous_matrice = m(1:3:2, vecteur_indice)
```

Le sous-tableau `sous_matrice` est alors constitué des éléments suivants de `m` :

`m(1,3)`	`m(1,5)`	`m(1,1)`
`m(3,3)`	`m(3,5)`	`m(3,1)`

10. Ne pas confondre avec `scilab`, `octave` ou `matlab` dans lesquels l'ordre est début:pas:fin.
11. Il est d'ailleurs très rare en fortran de déclarer des tableaux dont l'étendue est limitée à 1 sur un des axes, contrairement à ce qui se fait sous `scilab` ou `matlab` qui distinguent vecteurs lignes et vecteurs colonnes tous les deux représentés par des matrices avec un seul élément selon un des deux axes.

Exemple d'utilisation : Supposons qu'on échantillonne à n instants stockés dans un tableau `t` de rang 1 et de taille `n` l'évolution temporelle d'un vecteur à p composantes représenté par un tableau `v` de rang 2 et de profil `[n, p]`. Les p instants des maxima de chacune des composantes peuvent être obtenus comme section non-régulière du tableau `t` par indexation avec le vecteur de taille `p` des positions des maxima des composantes, donné par la fonction de réduction `MAXLOC` (*cf.* 7.4.1, p. 86) : `t(maxloc(v, dim=1))`.

7.3 Opérations sur les tableaux

7.3.1 Extension des opérations élémentaires aux tableaux

Les opérateurs agissant sur des types « scalaires » peuvent être appliqués à des tableaux, par extension de leur fonction scalaire à chacun des éléments des tableaux. De telles opérations sont qualifiées d'opérations *élémentaires*. Dans le cas des opérateurs binaires, les tableaux opérandes doivent être conformants pour que cette opération puisse s'effectuer terme à terme. Un scalaire est considéré comme conformant avec un tableau quelconque. Dans le cas où les types des éléments de deux tableaux diffèrent, il y a conversion implicite avant application de l'opérateur comme pour des opérations scalaires.

♥ ⇒ Pour des raisons de lisibilité, on pourra indiquer le rang des tableaux manipulés par une notation de section de tableau, par exemple `tab(:,:)` pour un tableau à deux dimensions, au lieu de la notation plus concise `tab` qui occulte le caractère multidimensionnel des variables manipulées. Mais

⚠ ⇒ désigner ainsi un tableau allouable comme section de tableau n'est pas équivalent : cela inhibe les possibilités de (ré-)allocation au vol par affectation (*cf.* 7.5.4, p. 94).

```
REAL, DIMENSION(2,3)         :: tab, tab1  ! profil [2,3]
REAL, DIMENSION(0:1,-1:1)    :: tab2       ! profil [2,3] identique
tab(:,:) = tab(:,:) + 1    ! le scalaire 1 est ajouté à chaque terme du tableau
tab(:,:) = - tab(:,:)
tab(:,:) = tab(:,:) * tab(:,:)    ! attention: produit terme à terme
                                  ! (et non produit matriciel)
tab(:,:) = cos(tab(:,:))          ! cosinus de chacun des termes du tableau
tab(:,:) = tab1(:,:) + tab2(:,:)  ! tab(1,1) = tab1(1,1) + tab2(0,-1) ...
                                  ! seul compte le profil
```

La multiplication matricielle de deux tableaux peut par exemple s'effectuer des deux façons suivantes (en pratique, on utilisera la fonction intrinsèque `MATMUL`, *cf* 7.4.2, p. 87) :

```
REAL, DIMENSION(2,3) :: a      ! matrice 2 lignes x 3 colonnes
REAL, DIMENSION(3,4) :: b      ! matrice 3 lignes x 4 colonnes
REAL, DIMENSION(2,4) :: c      ! matrice 2 lignes x 4 colonnes
INTEGER              :: i, j, k
c = 0                          ! initialisation du tableau c à 0
DO i = 1, 2                    ! boucle sur les lignes de c
  DO k = 1, 4                  ! boucle sur les colonnes de c
    somme: DO j = 1, 3         ! calcul de la somme : produit scalaire
      c(i,k) = c(i,k) + a(i,j) * b(j,k)          !    de a(i,:) par b(:,k)
    END DO somme
  END DO
END DO
c(:,:) = 0.                    ! réinitialisation de c
DO k = 1, 4                    ! boucle sur les colonnes de c
  somme_v: DO j = 1, 3           ! calcul vectoriel des sommes
    c(:,k) = c(:,k) + a(:,j) * b(j,k)
  END DO somme_v
END DO
```

♠ Affectation de sections de tableau avec recouvrement

Dans une instruction « vectorielle », il est possible d'affecter une section de tableau à une section de tableau avec recouvrement entre membre de gauche et membre de droite de l'opération d'affectation : le résultat peut être différent de celui qu'aurait donné une boucle explicite avec affectation scalaire. En effet,

Le second membre d'une instruction d'affectation dans un tableau est complètement évalué avant l'affectation elle-même.

```
INTEGER                          :: i
INTEGER, DIMENSION(5)            :: tab = (/ (i, i = 1, 5) /)
tab = tab(5:1:-1)                ! permet d'inverser l'ordre des éléments de tab
```

Une version scalaire aurait nécessité l'emploi d'une variable tampon supplémentaire pour éviter d'écraser des valeurs lors de l'échange entre deux coefficients placés symétriquement :

```
INTEGER                          :: i
INTEGER, DIMENSION(5)            :: tab = (/ (i, i = 1, 5) /)
INTEGER                          :: tampon
DO i = 1, 5/2              !  i=1, 2 ici : on s'arrête à la moitié du tableau
   tampon = tab(i)
   tab(i) = tab(6-i)
   tab(6-i) = tampon
END DO
```

f95 ### 7.3.2 Instruction et structure WHERE

Lorsque les opérations à effectuer sur les éléments d'un tableau dépendent d'une condition portant sur ce tableau ou un autre tableau conforme, on peut utiliser l'instruction `WHERE`, ou la structure `WHERE ... END WHERE`, bien adaptée au calcul parallèle [12].

```
[<nom>:] WHERE (<expression_logique_tableau>)
   <bloc d'affectation de tableau(x) conforme(s)>
   ! exécuté pour les éléments où l'expression est vraie
   [ELSEWHERE
   <bloc d'affectation de tableau(x) conforme(s)>]
   ! exécuté pour les éléments où l'expression est fausse
END WHERE [<nom>]
```

```
REAL, DIMENSION(10) :: a, b, c
WHERE ( a > 0. ) b = SQRT(a)  ! simple instruction WHERE
WHERE ( a > 0. )       ! bloc WHERE
   c = log(a)          ! pour tous les éléments positifs du tableau a
ELSEWHERE
   c = -1.e20          ! pour tous les éléments négatifs ou nuls du tableau a
END WHERE
```

♠ 7.3.3 Affectation d'une section non-régulière

Dans le cas où l'opération provoque l'affectation d'une section non-régulière de tableau, il est interdit d'affecter plusieurs fois le même élément, c'est-à-dire qu'*il ne peut y avoir de recouvrement à l'intérieur du membre de gauche* de l'opération d'affectation [13].

12. Depuis le fortran 95, il est possible d'imbriquer des structures `WHERE`.

13. Comme l'ordre d'affectation n'est pas garanti à l'exécution, on conçoit qu'autoriser l'affectation multiple de certains éléments introduirait une ambiguïté dans le résultat. Il est donc logique d'interdire cette possibilité, mais

```
INTEGER, DIMENSION(3) :: i = (/ 1, 3, 1 /), v = (/ -1, -2, -3 /), w
w = v(i)              ! opération licite qui  donne w = (/ -1, -3, -1 /)
w(i) = v              ! opération illicite car double affectation pour w(1)
```

7.4 Fonctions intrinsèques particulières aux tableaux

La majorité des procédures intrinsèques sont élémentaires (*cf.* 6.5.8, p. 79) et peuvent donc être appelées avec des arguments tableaux : elles s'appliquent alors à chacun des éléments scalaires constituant les tableaux. Mais il existe des fonctions intrinsèques spécifiquement consacrées à la manipulation des tableaux (*cf.* A.5, p. 145).

7.4.1 Fonctions d'interrogation

- `SIZE(array [, dim])` donne l'étendue d'un tableau selon la dimension `dim` si cet argument optionnel est fourni, et le nombre total d'éléments sinon ;
- `SHAPE` donne le vecteur profil d'un tableau (son étendue est le rang du tableau) ;
- `LBOUND` et `UBOUND` fournissent un *vecteur* constitué des bornes respectivement inférieures et supérieures des indices selon chacune des dimensions du tableau. Si une dimension est précisée comme argument optionnel, elles fournissent un scalaire indiquant les bornes des indices selon cette dimension.
- ⚠⇒ `MINLOC` et `MAXLOC` fournissent le *vecteur* des indices du *premier* élément respectivement minimum ou maximum du tableau, mais en indexant le tableau *à partir d'une borne inférieure égale à 1* selon chaque dimension, quelles que soient les bornes déclarées.

⚠⇒ Attention : même si on appelle `LBOUND`, `UBOUND`, `MINLOC` ou `MAXLOC` d'un tableau de rang 1, le résultat reste un tableau de rang 1 à un seul élément ; pour obtenir un résultat scalaire, il faut utiliser l'argument optionnel `DIM=1`.

```
INTEGER, DIMENSION(0:3)       :: v = (/ 3, 2, 2, 3 /)
INTEGER, DIMENSION(-2:2,3)    :: tab
INTEGER                       :: i
tab(:,1) = (/ (i, i=-2, 2) /)      !   | -2  -1  0  1  2 |
tab(:,2) = (/ (2*i, i=-2, 2) /)    !   | -4  -2  0  2  4 | (transposée de tab)
tab(:,3) = (/ (3*i, i=-2, 2) /)    !   | -6  -3  0  3  6 |
```

La matrice associée à `tab` est alors :

−2	−4	−6
−1	−2	−3
0	0	0
1	2	3
2	4	6

.

`SIZE(v)`	`4`
`SHAPE(v)`	`(/ 4 /)`
`LBOUND(v)`	`(/ 0 /)`
`LBOUND(v, DIM=1)`	`0`
`MINLOC(v)`	`(/ 2 /)`
`MAXLOC(v)`	`(/ 1 /)`
`MAXLOC(v, DIM=1)`	`1`

`SIZE(tab)`	`15`
`SIZE(tab,1)`	`5`
`SIZE(tab,2)`	`3`
`SHAPE(tab)`	`(/ 5, 3 /)`
`LBOUND(tab)`	`(/ -2, 1 /)`
`UBOUND(tab)`	`(/ 2, 3 /)`
`LBOUND(tab,1)`	`-2`
`MINLOC(tab)`	`(/ 1, 3 /)`
`MAXLOC(tab)`	`(/ 5, 3 /)`

... parfois difficile de prévoir quand elle pourrait intervenir dans le cas où les indices déterminant les sections sont variables.

7.4.2 Fonctions de réduction

Les fonctions de réduction renvoient un tableau de rang inférieur (éventuellement un scalaire) à celui passé en argument. Si on spécifie une dimension, seule cette dimension sera affectée par l'opération de réduction et le tableau rendu sera de rang $r-1$, où r est le rang du tableau initial.

- `SUM` et `PRODUCT` calculent respectivement la somme et le produit des éléments du tableau.
- `MAXVAL` et `MINVAL` fournissent respectivement le maximum et le minimum d'un tableau.
- `DOT_PRODUCT` donne le produit scalaire de deux vecteurs :
  - si `u` et `v` sont des vecteurs de réels (ou d'entiers), `DOT_PRODUCT(u, v) = SUM(u*v)`.
  - si `u` et `v` sont des vecteurs de complexes, `DOT_PRODUCT(u, v) = SUM(CONJG(u)*v)`.
- `NORM2` calcule la norme euclidienne d'un tableau de *réels* de rang quelconque, soit `NORM2(t) = SQRT(SUM(t)**2))` ⇐ f2008
- `MATMUL` [14] effectue le *produit matriciel* de deux tableaux [15]. Elle suppose que les profils de ses arguments permettent l'opération :
  - produit d'une matrice de n lignes et p colonnes par une matrice de p lignes et q colonnes donnant une matrice de n lignes et q colonnes
  - produit d'une matrice de n lignes et p colonnes par un vecteur de taille p donnant un vecteur de taille n
  - produit d'un vecteur de taille n par une matrice de n lignes et p colonnes donnant un vecteur de taille p.

`PRODUCT(tab, DIM=2)`	`(/ -48, -6, 0, 6, 48 )/`
`SUM(tab, DIM=2)`	`(/ -12, -6, 0, 6, 12 )/`
`SUM(tab)`	`0`
`MAXVAL(tab, DIM=2)`	`(/ -2, -1, 0, 3, 6 )/`
`MAXVAL(tab, DIM=1)`	`(/ 2, 4, 6 /)`
`MAXVAL(tab)`	`6`
`MINVAL(tab, DIM=2)`	`(/ -6, -3, 0, 1, 2 )/`
`DOT_PRODUCT(tab(:,1), tab(:,3))`	`30`

Par exemple, les calculs de moments d'une série statistique stockée dans un tableau monodimensionnel peuvent s'exprimer de façon très concise à l'aide de ces fonctions.

```
INTEGER, PARAMETER              :: n = 1000
REAL, DIMENSION(n)              :: x, y
REAL                            :: moyenne_x, moyenne_y, variance_x, covariance
moyenne_x = SUM( x(:) ) / REAL( SIZE( x(:) ) )
moyenne_y = SUM( y(:) ) / REAL( SIZE( y(:) ) )
variance_x = SUM( (x(:) - moyenne_x) ** 2 ) / REAL(SIZE(x(:)) - 1)
covariance = SUM((x(:)-moyenne_x) * (y(:)-moyenne_y)) / REAL(SIZE(x(:)) - 1)
```

7.4.3 Fonctions de transformation

- `RESHAPE(source,shape[,pad][,order])` restructure les éléments du tableau `source` suivant le profil `shape`. Si le tableau source est de taille inférieure au tableau cible, les éléments manquants sont pris dans le tableau `pad`. L'ordre « traditionnel » de rangement (premier indice le plus rapide, donc par colonnes pour les matrices) dans le tableau cible peut être modifié via l'argument (optionnel) `order`, qui est un tableau de rang 1 : ces éléments sont les numéros des axes du tableau cible, de celui de l'indice le plus rapide à celui du plus lent.

14. Avec certains compilateurs, il est possible de faire appel à une version optimisée de la bibliothèque BLAS pour effectuer les calculs de `MATMUL` sur des matrices de grande taille (*cf.* F.5.2 par exemple avec l'option `-fexternal-blas` pour `gfortran`).

15. Rappel : le produit de deux matrices via l'opérateur `*` agissant sur deux tableaux n'est applicable que sur deux tableaux conformes et effectue le produit *terme à terme* et non le produit matriciel.

- `PACK(array, mask [, vector])` réarrange les éléments du tableau `array` sélectionnés selon le masque `mask` (tableau de booléen conforme à `array`) dans un tableau de rang 1. Si l'argument `vector` (tableau de rang 1) est fourni, il prescrit l'étendue du tableau résultant et doit comporter au moins autant d'éléments que `mask` comporte de valeurs `.true.` : les éléments de `vector` sont utilisés pour compléter le résultat si le nombre d'éléments vrais du masque est inférieur à l'étendue de `vector`. Les éléments du tableau source `array` sont traités dans l'ordre habituel, par colonnes pour une matrice.
- `UNPACK(vector, mask, field)` recopie les éléments du tableau de rang 1 `vector` dans un tableau sous le contrôle du masque `mask` conformant avec le tableau résultat. C'est donc `mask` qui détermine le profil du tableau résultat. Si l'élément du masque est vrai, l'élément correspondant du tableau résultat est pris dans `vector`, sinon, on lui donne la valeur `field`.
- `SPREAD(source, dim, ncopies)` crée, par duplication (`ncopies` fois selon la dimension `dim`) du scalaire ou tableau `source`, un tableau de rang immédiatement supérieur à `source`.
- `MERGE(tsource, fsource, mask)` fusionne les tableaux `tsource` et `fsource` selon le masque `mask` : les trois arguments et le résultat sont des tableaux conformants.
- `CSHIFT(array, shift [, dim])` et `EOSHIFT(array, shift [, boundary][, dim])` effectuent des décalages de `shift` des éléments du tableau selon la dimension éventuellement spécifiée (si le paramètre `dim` n'est pas précisé, cela implique `dim=1` [16]). Si `shift` est positif, les décalages se font vers les indices décroissants et réciproquement. `CSHIFT` effectue des décalages circulaires (sans perte d'éléments) alors que `EOSHIFT` remplace les éléments perdus par des « zéros » du type des éléments de `array` ou par `boundary` si cet argument optionnel est précisé.
- `TRANSPOSE` calcule la transposée d'une matrice.

```
INTEGER                    :: i
INTEGER, DIMENSION(8)      :: v = (/ (i, i = 1, 8) /), w
INTEGER, DIMENSION(3,4)    :: tab1, tab2, tab3
tab1 = RESHAPE(v, (/ 3, 4 /), pad = (/ 0 /) )
w = CSHIFT(v, 2)
tab2 = CSHIFT(tab1, SHIFT=-1 )  ! décalage des lignes vers le bas
tab3 = CSHIFT(tab1, (/ 1, -2, 2 /), DIM=2) ! un décalage différent par ligne
```

1	2	3	4	5	6	7	8

RESHAPE(v,(/3,4/),pad=(/0/)) ⇒

1	4	7	0
2	5	8	0
3	6	0	0

1	4	7	0
2	5	8	0
3	6	0	0

PACK(tab1,mod(tab1,2)==1) ⇒

1	3	5	7

1	2	3	4

UNPACK(v(1:4),mod(tab1,2)==1, -1) ⇒

1	-1	4	-1
-1	3	-1	-1
2	-1	-1	-1

1	2	3

SPREAD((/1, 2, 3/),dim=1,ncopies=2) ⇒

1	2	3
1	2	3

1	2	3

SPREAD((/1, 2, 3/),dim=2,ncopies=2) ⇒

1	1
2	2
3	3

16. Les décalages sont appliqués à chacun des vecteurs constituant le tableau et non globalement.

1	4	7
2	5	8
3	6	9

-1	-4	-7
-2	-5	-8
-3	-6	-9

F	T	F
T	F	T
F	T	F

MERGE ⇒

-1	4	-7
2	-5	8
-3	6	-9

1	2	3	4	5	6	7	8

CSHIFT(v,SHIFT=2) ⇒

3	4	5	6	7	8	1	2

1	2	3	4	5	6	7	8

EOSHIFT(v,SHIFT=2) ⇒

3	4	5	6	7	8	0	0

1	4	7	0
2	5	8	0
3	6	0	0

CSHIFT(tab1,SHIFT=-1)
ou ⇒
CSHIFT(tab1,SHIFT=+2)

3	6	0	0
1	4	7	0
2	5	8	0

1	4	7	0
2	5	8	0
3	6	0	0

CSHIFT(tab1, (/ 1, -2, 2 /), DIM=2) ⇒

+1	4	7	0	1
−2	8	0	2	5
+2	0	0	3	6

♠ 7.4.4 Notion de masque

Parfois, un traitement a priori vectoriel via une fonction intrinsèque ne doit pas être appliqué à tous les éléments d'un tableau, mais une expression logique tableau peut servir à décider quels éléments seront concernés par l'instruction tableau. Certaines procédures intrinsèques portant sur des tableaux acceptent un tel tableau booléen conforme à l'argument traité, appelé *masque* (mask), en argument optionnel (de mot-clef `MASK=`).

```
INTEGER                  :: i, produit, somme
INTEGER, DIMENSION(5)    :: v = (/ (i, i = -2, 2) /), w = 2 * v
produit = PRODUCT(v, MASK = v /= 0)   ! produit des éléments non nuls de v
somme = SUM(w, MASK = v /=0) ! somme des éléments de w(i) tels que v(i) non nul
```

7.4.5 ALL, ANY, COUNT

Trois fonctions intrinsèques opèrent sur des tableaux de booléens pour vérifier la validité d'une expression logique sur les éléments d'un tableau :

- `ALL(<mask>)` rend `.TRUE.` si tous les éléments du tableau booléen `<mask>` sont vrais, `.FALSE.` sinon ; elle réalise un ET logique entre les éléments du tableau
- `ANY(<mask>)` rend `.TRUE.` si un au moins des éléments du tableau booléen `<mask>` est vrai, `.FALSE.` si tous sont à `.FALSE.` ; elle réalise un OU logique entre les éléments du tableau
- `COUNT(<mask>)` rend le nombre (entier) des éléments du tableau `<mask>` qui valent `.TRUE.`.

En pratique, `<mask>` est produit par application d'un test à un tableau (ou des tableaux conformes).

```
INTEGER                  :: i, nb_positifs
INTEGER, DIMENSION(5)    :: v = (/ (i, i=-2, 2) /)
REAL, DIMENSION(5)       :: u, u_log
...
nb_positifs = COUNT (v > 0)     ! nombre d'éléments positifs (2 ici)
IF (ALL(u > 0)) u_log = log(u)  ! calculé si tous les éléments de u sont > 0
```

Une autre application de `ALL` est de tester si un tableau `v` de réels de rang un et de bornes `1` et `n` est trié, par exemple par ordre croissant, en travaillant sur deux sections de taille `n-1` décalées : `ALL( v(:n-1) < v(2:) )`

f95

7.4.6 Instruction et structure FORALL

Certaines affectations de tableaux, par exemple combinant deux indices, ne peuvent pas s'exprimer de façon globale sur le tableau, malgré leur facilité à être traitées en parallèle. Par exemple, une affectation entre tableaux de rang différent :

```
DO i = 1, n
   tab(i, i) = v(i) ! copie du vecteur v sur la diagonale de tab
END DO
```

peut s'exprimer à l'aide de l'instruction FORALL :

```
FORALL (i = 1:n) tab(i, i) = v(i)
```

Les expressions de droite sont alors calculées en parallèle pour toutes les valeurs de l'indice, comme si les résultats intermédiaires étaient stockés dans des variables temporaires jusqu'à la fin des calculs en attendant l'affectation finale du membre de gauche. En particulier, au contraire de ce qui se passe dans une boucle classique, l'ordre de parcours des indices est indifférent ; plus précisément il peut être choisi par le processeur pour optimiser (vectoriser) le calcul. Cette construction s'avère donc très adaptée aux calculs de filtrage numérique linéaire. Par exemple, l'instruction suivante calcule bien une différence finie pour estimer une dérivée, alors qu'une boucle dans le sens croissant des indices ne donnerait pas le même résultat.

```
FORALL (i = 1: size(v)-1 )  v(i+1) = v(i+1) - v(i)
```

La structure FORALL ... END FORALL permet d'exprimer une série d'affectations successives portant sur les mêmes indices [17].

```
FORALL (i=1:n, j=1:p)
    a(i, j) = i + j                ! tous les a(i,j) sont calculés
    b(i, j) = a(i, j) ** i         ! avant de calculer b(i,j)
END FORALL
```

L'instruction et la structure FORALL admettent de plus un masque (*cf.* 7.4.4, p. 89), qui à l'inverse de celui de WHERE, doit être scalaire, avec une condition logique permettant de sélectionner les indices.

```
FORALL (i = 1:n, v(i) /=0 ) tab(i, i) = 1./v(i)
```

Une structure FORALL peut contenir aussi des constructions FORALL et WHERE (*cf.* 7.3.2, p. 85), mais ne peut pas appeler des fonctions de transformation de tableaux.

7.5 Tableaux, procédures et allocation dynamique

7.5.1 Les trois méthodes d'allocation des tableaux

L'allocation d'espace mémoire pour des tableaux peut s'effectuer, en excluant provisoirement les arguments muets de procédures, de trois façons différentes :

– Pour les tableaux de profil fixe, la réservation d'espace mémoire se fait une fois pour toutes à la compilation. Portée et durée de vie dépendent alors de la procédure hôte.

♥ ⇒ Noter qu'il est possible, et conseillé de paramétrer les profils de ces tableaux par des expressions entières constantes, en utilisant des constantes symboliques (grâce à l'attribut

17. Chaque instruction est alors exécutée sur tous les indices avant de passer à la suivante.

PARAMETER). Cette précaution permet de simplifier un changement de taille de ces tableaux qui se réduit au changement de constantes symboliques une seule fois dans le code, suivi d'une recompilation.

- Au début d'une procédure, il est possible de déclarer des tableaux *locaux* de rang fixe dont le profil dépend d'un argument d'entrée de la procédure ou d'un paramètre partagé via un module. L'espace mémoire leur est attribué automatiquement sur la pile (stack), au moment de l'appel de la procédure, et il est bien sûr libéré à la fin de son exécution. Cela limite leur durée de vie à l'appel de la procédure, et restreint leur portée (scope) à cette procédure et éventuellement celles qu'elle appelle. Ces tableaux sont qualifiés de *tableaux automatiques*.
- Enfin, il est possible de déclarer des tableaux de rang fixé à la compilation, mais dont le profil est défini en cours d'exécution : portée et durée de vie de ces tableaux sont décidées par l'utilisateur qui doit prendre en charge lui-même l'allocation et la libération de la mémoire, prise cette fois sur le tas (heap). Ces tableaux à profil différé sont aussi qualifiés de tableaux dynamiques (*cf.* 7.5.3, p. 92).

7.5.2 Tableaux en arguments muets des procédures

Bien entendu, si on travaille avec des tableaux de profil fixe, on peut déclarer leur profil à la fois dans le programme principal et les procédures appelées : on dit que ces tableaux sont *à profil explicite*. Rappelons que, dans ce cas, seul compte le profil du tableau et non ses bornes. ⇐⚠

```
MODULE util
IMPLICIT NONE
CONTAINS
  SUBROUTINE sub(tab)
    REAL, DIMENSION(5), INTENT(INOUT) :: tab
    ...                     ! tab(1) contient tab0(-2) au premier appel
    ...                     ! tab(1) contient tab1(1) au deuxième appel
  END SUBROUTINE SUB
END MODULE util
...
USE UTIL
REAL, DIMENSION(-2:2)       :: tab0
REAL, DIMENSION(5)          :: tab1
CALL sub(tab0)
CALL sub(tab1)
...
```

Mais, il est plus fréquent que des procédures nécessitent l'usage de tableaux de rang connu, mais de profil variable d'un appel à l'autre. On peut passer les étendues en argument de la procédure pour déclarer ensuite les arguments tableaux, mais cela constitue un risque d'incohérence.

La méthode la plus fiable consiste à attribuer aux tableaux arguments muets de la procédure un *profil implicite* (assumed-shape, profil « supposé »), symbolisé par des deux points à la place des étendues qui seront déterminées à chaque appel par le tableau argument effectif. Pour pouvoir appeler une procédure utilisant comme argument muet des tableaux de profil implicite, la procédure appelante doit connaître l'interface de la procédure appelée. Trois possibilités sont envisageables, dont la dernière est préférable : ⇐♥

- une procédure interne (*cf.* 6.3.1, p. 63) ;
- une procédure externe complétée par son interface explicite (*cf.* 6.3.3, p. 64) ;
- de préférence, une procédure de module (*cf.* 6.4, p. 65), dont l'interface est rendue explicite via l'instruction USE.

```
MODULE util
IMPLICIT NONE
```

```
CONTAINS
   SUBROUTINE sub(tab)
     REAL, DIMENSION(:,:)                      :: tab ! profil implicite
     ...
     RETURN
   END SUBROUTINE sub

   SUBROUTINE sub2(tab)
     REAL, DIMENSION(:,:)                      :: tab ! profil implicite
     REAL, DIMENSION(SIZE(tab,1),SIZE(tab,2)) :: aux ! tableau local automatique
     ...
     RETURN
   END SUBROUTINE sub2
...
END MODULE util

PROGRAM principal
USE util
INTEGER, PARAMETER                             :: lignes = 3, colonnes = 4
REAL, DIMENSION(lignes,colonnes)               :: tab
REAL, DIMENSION(2*colonnes,2*lignes)           :: tab2
...
CALL sub(tab)
CALL sub2(tab)
CALL sub(tab2)
CALL sub2(tab2)
...
END PROGRAM principal
```

♠ 7.5.3 Tableaux dynamiques allouables

Lorsque, par exemple, le profil d'un tableau n'est connu qu'à l'exécution, on peut lui allouer dynamiquement la mémoire nécessaire. Il faut alors déclarer ce tableau en précisant son rang [18] et l'attribut `ALLOCATABLE`, qui en fait un tableau à *profil différé* (deferred-shape array). Une fois le profil déterminé, l'instruction `ALLOCATE` permettra d'allouer effectivement la mémoire requise, avant toute opération d'affectation sur ce tableau. Après avoir utilisé le tableau, il est recommandé de libérer la mémoire allouée par l'instruction `DEALLOCATE` [19], en particulier si on souhaite réutiliser le même identificateur de tableau avec une autre taille.

♥ ⇒ Dans un souci de fiabilité des programmes, on prendra soin de tester le statut de retour de la procédure `ALLOCATE` et, avant toute opération d'allocation ou de libération de mémoire, de tester l'état d'allocation des tableaux grâce à la fonction d'interrogation `ALLOCATED` qui fournit un résultat booléen.

```
1 PROGRAM alloc1 ! fichier alloc-tab1d.f90
2 IMPLICIT NONE  ! allocation dynamique sans pointeur
3 INTEGER :: i, n, ok=0
4 INTEGER, DIMENSION(:), ALLOCATABLE :: ti
5    !    rang seulement ^^^^^^^^^^^ attribut obligatoire
6 DO ! boucle sur la taille du tableau jusqu'à n<=0
7    WRITE(*, *) "entrer le nb d'éléments du tableau (0 si fin)"
8    READ(*,*) n
```

18. Mais sans indiquer sa taille.

19. En fin de programme principal, il est prudent de désallouer explicitement tous les tableaux dynamiques alloués. En effet, cette opération implique la relecture d'informations sur la taille de la zone mémoire allouée. La désallocation peut provoquer une erreur de segmentation si cette relecture est erronée, par exemple à cause d'un dépassement de mémoire dans un autre tableau stocké au voisinage. Ainsi la désallocation peut révéler utilement une corruption de la mémoire causée par d'autres opérations.

```
9     IF(n <=0 ) EXIT          ! sortie de la boucle
10    ALLOCATE(ti(n), stat=ok) ! allocation de la mémoire
11    IF (ok /= 0) THEN
12      WRITE(*, *) "erreur d'allocation"
13      CYCLE ! on passe à une autre valeur de n
14    END IF
15    DO i=1, n       ! affectation du tableau
16      ti(i) = i
17    END DO
18    DO i=1, n       ! affichage du tableau en colonne
19      WRITE(*,*)  ti(i)
20    END DO
21    IF(ALLOCATED(ti)) THEN
22      DEALLOCATE(ti) ! libération de la mémoire
23    END IF
24  END DO
25  END PROGRAM alloc1 ! fichier alloc-tab1d.f90
```

Un tableau dynamique *local à une procédure* et alloué dans cette procédure est automatiquement libéré à la sortie de la procédure (via `END` ou `RETURN`), sauf s'il est rendu permanent par l'attribut `SAVE`.

f2003 **Tableau dynamique en paramètre** Un tableau dynamique peut aussi être passé en argument d'une procédure, mais, en fortran 95, il doit être déjà alloué avant l'appel de la procédure et n'est pas considéré comme allouable dans la procédure. Ces restrictions sont levées en fortran 2003, ainsi qu'avec les extensions spécifiques de la plupart des compilateurs fortran 95 [20]. Le tableau doit alors posséder l'attribut `ALLOCATABLE` dans la procédure, qui peut alors modifier son allocation. S'il s'agit d'un argument de sortie (`INTENT(OUT)`), il est automatiquement désalloué si nécessaire à l'entrée dans la procédure.

```
1   MODULE m_alloc_sub
2   IMPLICIT NONE
3   CONTAINS
4     SUBROUTINE init_tab(tab, m, n)
5       INTEGER, INTENT(in) :: m, n
6       INTEGER, ALLOCATABLE, INTENT(out) :: tab(:,:)
7       INTEGER :: ok, i, j
8       IF(.NOT.ALLOCATED(tab)) THEN
9         ALLOCATE(tab(m, n), stat = ok)
10        IF (ok /= 0) THEN
11          WRITE(*, *) 'erreur allocation'
12          STOP
13        END IF
14        DO i = 1, m
15          DO j = 1, n
16            tab(i, j) = 1000 * i + j
17          END DO
18        END DO
19      ELSE ! ne devrait pas se produire en fortran 2003 car tab est INTENT(OUT)
20        WRITE(*, *)  'tableau déjà alloué'
21        STOP
22      END IF
23    END SUBROUTINE init_tab
24  END MODULE m_alloc_sub
25  PROGRAM alloc_proc
26    USE m_alloc_sub
27    IMPLICIT NONE
```

20. En particulier `xlf` d'IBM, le compilateur NAG et `g95` avec l'option `-std=f2003` ou `-std=f95` assortie de `-ftr15581` (*cf.* F.6.1, p. 175) autorisent l'allocation de l'argument tableau dans la procédure.

```
  INTEGER, ALLOCATABLE :: t(:,:)
  INTEGER :: m, n, i
  WRITE (*, *)  'entrer les dimensions du tableau: m et n'
  READ (*, *) m, n
  ! appel du sous programme qui alloue et initialise le tableau
  CALL init_tab(t, m, n)
  DO i = 1 ,m
    WRITE (*, *)  t(i, :)
  END DO
  m = 2 * m
  n = 2 * n
  ! deuxième appel du sous programme avec tab déjà alloué, mais
  ! nécessitant une réallocation car les dimensions ont changé
  CALL init_tab(t, m, n)
  DO i = 1 ,m
    WRITE (*, *)  t(i, :)
  END DO
  IF(ALLOCATED(t)) THEN
    DEALLOCATE(t)
  END IF
END PROGRAM alloc_proc
```

f2003 ### 7.5.4 Allocation et réallocation au vol de tableaux dynamiques par affectation

En fortran 2003, un tableau allouable peut être (ré-)alloué automatiquement si on lui affecte une expression tableau :

- s'il n'était pas alloué auparavant, il est alloué avec le profil de l'expression qui lui est affectée ;
- s'il était déjà alloué avec un profil différent, il est réalloué pour ajuster son profil à celui du membre de droite de l'affectation.

⚠⇒ Noter que si le membre de gauche de l'affectation est une section de tableau, aucune allocation n'est effectuée. La réallocation automatique peut donc être inhibée pour un tableau complet si on le désigne comme une section de tableau en précisant dans le membre de gauche le symbole « : » pour chaque dimension. Dans ce cas, les éléments de l'expression de droite en dehors du profil du tableau de gauche sont ignorés.

Par exemple, pour des tableaux de rang 1, le programme suivant :

```
! allocation automatique par affectation
! fortran 2003 seulement : gfortran 4.6.3 (nov 2013 mageia 2)
! ok avec NAG Fortran Compiler Release 5.2(643) le 12 fev 2009
! ok avec ifort (IFORT) 12.1.3 20120212 + option -assume realloc_lhs
PROGRAM t_alloc_affect
INTEGER, DIMENSION(:), ALLOCATABLE :: v1, v2, v3
v1 = [1, 2]  ! allocation de v1 par affectation => 2 éléments
WRITE(*,*) "taille de v1=", SIZE(v1), " v1=", v1
v2 = [-3, -2, -1 ] ! allocation de v2 par affectation => 3 éléments
WRITE(*,*) "taille de v2=", SIZE(v2), " v2=", v2
v3 = v1   ! allocation implicite de v3 => 2 éléments
WRITE(*,*) "taille de v3=", SIZE(v3), " v3=", v3
v1 = v2   ! réallocation implicite de v1 => 3 éléments
WRITE(*,*) "taille de v1=", SIZE(v1), " v1=", v1
v3(:) = v2   ! pas de réallocation de v3 => v2(3) inutilisé
! provoque une erreur sous gfortran : tableaux non conformants
WRITE(*,*) "taille de v3=", SIZE(v3), " v3=", v3
DEALLOCATE(v1, v2, v3)
END PROGRAM t_alloc_affect
```

affiche :

```
taille de v1= 2  v1= 1 2
taille de v2= 3  v2= -3 -2 -1
taille de v3= 2  v3= 1 2
taille de v1= 3  v1= -3 -2 -1
taille de v3= 2  v3= -3 -2
```

La (ré-)allocation automatique de tableaux par affectation peut entrer en conflit avec certaines optimisations effectuées par le compilateur. C'est pourquoi certains compilateurs disposent d'une option [21] pour choisir si on l'autorise, indépendamment du choix du standard fortran 2003 ou postérieur.

21. `-fno-realloc-lhs` ou `-f-realloc-lhs` pour `gfortran` (*cf.* F.5.2, p. 173), `-Mallocatable=95` ou `-Mallocatable=03` pour `pgf95` (*cf.* F.3.1, p. 171), `-assume realloc_lhs` ou `-assume no-realloc_lhs` pour `ifort` (*cf.* F.4.1, p. 171), `-qxlf2003=autorealloc` pour `xlf` (*cf.* F.1.2, p. 169).

Chapitre 8

Chaînes de caractères

Ce court chapitre regroupe les outils, dont certains déjà introduits dans les chapitres précédents, permettant la manipulation des chaînes de caractères en fortran : le type chaîne (*cf.* chapitre 2), les expressions de type chaîne (*cf.* chapitre 3) , les sous-chaînes, les fonctions manipulant des chaînes, les entrées/sorties de chaînes (*cf.* chapitre 5), les tableaux de chaînes et le passage d'arguments chaînes dans les procédures.

8.1 Le type chaîne de caractères

Le langage fortran comporte le type intrinsèque `CHARACTER` pour stocker les chaînes de caractères. À chaque objet de type `CHARACTER` est associée une longueur `LEN` qui est le nombre des caractères de la chaîne.

8.1.1 Les constantes chaînes de caractères

Les constantes chaînes de caractères sont délimitées par des apostrophes simples « ' » ou [1] des guillemets « " ». À l'intérieur d'un des types de délimiteur, l'autre est considéré comme un caractère quelconque, ce qui permet d'écrire par exemple `"aujourd'hui"`. Toutefois, on peut aussi introduire le délimiteur à l'intérieur de la chaîne à condition de le dupliquer, comme dans `'aujourd''hui'`.

Pour écrire une constante chaîne sur plusieurs lignes, il faut terminer toutes les lignes sauf la dernière par le symbole `&` de continuation de ligne. Par défaut tous les espaces en début de ligne de suite sont alors significatifs (*cf.* 1.4.2 p. 8) ; mais on peut adopter une présentation plus agréable en spécifiant par un symbole `&` supplémentaire le début de la suite de la chaîne, ignorant ainsi tous les espaces situés avant.

```
WRITE (*,*) 'chaîne de caractères comportant plusieurs lignes dans &
            &le programme source sans espace superflu'
```

8.1.2 Les déclarations de chaînes de caractères

```
CHARACTER(LEN=<longueur>) :: <chaine>
```

permet de déclarer une chaîne comportant *longueur* caractères. La longueur de la chaîne est bien le nombre de caractères [2], espaces inclus, qui la constituent. Un caractère est simplement une chaîne de longueur 1. La longueur d'un scalaire chaîne de caractères *constant nommé* (c'est-à-dire avec l'attribut `PARAMETER`) peut être calculée par le compilateur si on spécifie [3] `LEN=*` dans la déclaration.

1. En langage C, ces délimiteurs sont spécialisés : le délimiteur des constantes chaînes de caractères est le guillemet « " » et l'apostrophe « ' » sert à délimiter les constantes de type caractère individuel.
2. En langage C au contraire, le tableau de caractères doit comporter un élément supplémentaire pour indiquer la fin de la chaîne.
3. On retrouve la même propriété qu'en langage C où la taille d'un tableau de caractères constant peut être calculée par le compilateur lors de l'initialisation.

```
CHARACTER(LEN=*), PARAMETER :: chaine_fixe="mot"
```

f2003

8.1.3 Les variantes du type chaînes de caractères

Fortran 90 doit donner accès à un type de caractères par défaut, et peut aussi fournir d'autres jeux de caractères dépendant du compilateur de la machine.

Mais fortran 2003 a introduit la possibilité d'accéder à des variantes du type chaîne associées à différents codages des caractères, notamment pour les caractères UNICODE. La fonction `SELECTED_CHAR_KIND` rend un entier indiquant le numéro du sous-type chaîne demandé en paramètre : parmi les sous-types possibles, on trouve `DEFAULT`, `ASCII` et `ISO_10646` pour les caractères Unicode (encore noté `UCS_4` assimilable à `UTF-32`) [4] , mais seul `DEFAULT` est requis par la norme. Si la variante demandée n'est pas disponible, la fonction `SELECTED_CHAR_KIND` rend `-1`. La variante `ISO_10646` sera nécessaire pour lire des fichiers texte codés en `UTF-8` (*cf.* 5.3.2, p. 44).

Les constantes chaînes de variante de type autre que le défaut sont désignées en les *préfixant* par le numéro du sous-type donné par `SELECTED_CHAR_KIND` suivi du symbole souligné [5]. Par exemple, si le codage par défaut est `ISO-8859-1` :

```
INTEGER, PARAMETER :: defchar = SELECTED_CHAR_KIND('DEFAULT')
CHARACTER(LEN=3, KIND=defchar) :: mot=defchar_"été"
```

Pour désigner un caractère unicode en `UTF-32`, on peut utiliser la fonction `CHAR` en lui fournissant les codes-points unicode comme des entiers en hexadécimal (*cf.* 2.2.2, p. 15), par exemple :

```
INTEGER, PARAMETER :: utf32=SELECTED_CHAR_KIND('ISO_10646') ! UCS_4 = UTF-32
CHARACTER(LEN=1, KIND=utf32) :: car1_utf32, car2_utf32
car1_utf32 = CHAR(INT(Z'00E6'), KIND=utf32) ! ligature ae æ
car2_utf32 = CHAR(INT(Z'FB03'), KIND=utf32) ! ligature ffi (3 octets en UTF-8)
```

8.2 Expressions de type chaîne de caractères

Une expression de type chaîne de caractères peut construite par concaténation de variables ou de constantes chaînes, par extraction de sous-chaîne ou enfin à l'aide de fonctions à résultat chaîne de caractères.

8.2.1 Concaténation de chaînes

L'opérateur `//` permet de concaténer [6] deux chaînes de caractères. Par exemple, l'expression `"bonjour" // " au revoir"` est évaluée comme `"bonjour au revoir"`.

8.2.2 Sous-chaînes

De façon assez semblable aux sous-sections des tableaux, le langage fortran permet de manipuler des sous-chaînes de caractères, selon la syntaxe suivante [7] :

4. Par exemple, le compilateur NAG possède quatre variantes du type chaîne, de paramètres `KIND` :
   - 1 pour l'`ASCII` sur 1 octet qui comporte en fait le codage `ISO-8859-1` ;
   - 2 sur deux octets pour le codage `UCS_2`, qui coïncide avec `UTF-16` ;
   - 3 pour le codage `JIS X 0213` permettant d'accéder aux caractères japonais ;
   - 4 sur quatre octets pour `UCS_4`, assimilable à `UTF-32`.

5. Ainsi, pour les chaînes, on préfixe la variante de type, alors que pour les autres types (entiers, réels), on suffixe la variante de type.

6. En langage C, la concaténation de chaînes de caractères est obtenue par simple juxtaposition, sans opérateur explicite.

7. On notera que le symbole « `:` » n'est pas facultatif, même si on veut désigner une sous-chaîne réduite à un caractère.

```
chaine1([<deb>]:[<fin>])
```

désigne la sous-chaîne extraite de `chaine1` commençant au caractère numéro ***deb*** et terminant au caractère numéro ***fin*** sachant que par défaut ***deb*** vaut 1 et ***fin*** est la longueur de `chaine1`, soit `LEN(chaine1)`. Si ***deb*** est supérieur à ***fin***, la sous-chaîne est vide. Avec les déclarations suivantes :

`CHARACTER(LEN=*), PARAMETER :: mot1 = "bonjour", mot2 = "␣bonjour␣␣␣"`

`mot1(1:3)` vaut `"bon"` ainsi que `mot1(:3)` et `mot1(4:7)` vaut `"jour"` de même que `mot1(4:)` mais `mot2(:4)` vaut `"␣bon"` et `mot2(5:)` vaut `"jour␣␣␣"`.

8.2.3 Affectation

L'affectation globale d'expressions chaînes de caractères est possible en fortran [8], sachant que le membre de gauche peut être une chaîne ou une sous-chaîne et le membre de droite une expression de type chaîne. Au cas où la longueur des chaînes de part et d'autre du signe `=` diffère,

- si l'expression chaîne du membre de droite est plus longue que la variable à laquelle on doit l'affecter, la chaîne de droite est tronquée [9] côté droit ;
- si l'expression chaîne du membre de droite est plus courte que la variable à laquelle on doit l'affecter, la chaîne de droite est complétée par des espaces pour aboutir à la longueur de la chaîne de gauche.

Avec les déclarations et les affectations suivantes :

```
CHARACTER(LEN=7) :: mot1 = "bonjour"
CHARACTER(LEN=9) :: mot2 = "au revoir"
CHARACTER(LEN=17) :: mot3
CHARACTER(LEN=7) :: mot4
CHARACTER(LEN=5) :: mot5
mot3 = mot1 // " " // mot2
mot4 = mot2(1:3) // mot1(4:)
mot5 = mot1(4:7)
mot2(1:2) = "A "
```

on obtient `mot3="bonjour au revoir"`, `mot4="au jour"`, `mot5="jour "` et `mot2="A  revoir"`.

8.3 Les fonctions opérant sur les chaînes de caractères

8.3.1 Suppression des espaces terminaux avec `TRIM`

La fonction `TRIM(STRING=`***ch***`)` d'argument et de résultat de type chaîne de caractère supprime les blancs à droite de la chaîne passée en argument. Elle rend donc une chaîne de longueur `LEN_TRIM(`***ch***`)` inférieure au égale à celle `LEN(`***ch***`)` de ***ch***.
`TRIM("  ab cd  ")` donne `"  ab cd"`

8.3.2 Justification à gauche avec `ADJUSTL`

La fonction élémentaire `ADJUSTL(STRING=`***ch***`)` supprime tous les blancs en début de la chaîne ***ch*** et en ajoute autant à droite de façon à rendre une chaîne *de même longueur* que son argument, mais justifiée à gauche.
`ADJUSTL("  ab cd  ")` donne `"ab cd    "`
`TRIM(ADJUSTL(`***chaine***`))` enlève donc les espaces à la fois à gauche et à droite.

8. Les chaînes de caractères en C étant des tableaux de caractères, il n'est pas possible de les affecter globalement et on doit recourir à des fonctions de copie de chaînes à cette fin.
9. Le compilateur `gfortran` peut signaler cette troncature grâce à l'option `-Wcharacter-truncation` (*cf.* F.5.3, p. 173).

8.3.3 Justification à droite avec ADJUSTR

De façon symétrique, la fonction élémentaire `ADJUSTR(STRING=ch)` supprime tous les blancs en fin de la chaîne *ch* et en ajoute autant à gauche de façon à rendre une chaîne *de même longueur* que son argument, mais justifiée à droite.
`ADJUSTR("  ab cd  ")` donne `"    ab cd"`

8.3.4 Les fonctions LEN et LEN_TRIM

La fonction `LEN`, d'argument chaîne et de résultat entier, rend le nombre de caractères de la chaîne telle qu'elle a été déclarée à la compilation, indépendamment du contenu de la dite chaîne. La variante `LEN_TRIM` ne tiend pas compte des espaces terminaux, mais elle compte ceux en début de chaîne : `LEN_TRIM(chaine) = LEN(TRIM(chaine))`. Ainsi, avec les constantes chaînes suivantes,

```
CHARACTER(LEN=*), PARAMETER :: mot1 = "bonjour", mot2 = " bonjour   "
WRITE(*,*) "longueurs totales de mot1 et mot2 ", LEN(mot1), LEN(mot2)
WRITE(*,*) "longueurs sans espaces terminaux ", LEN_TRIM(mot1), LEN_TRIM(mot2)
```

on affichera des longueurs respectives de 7 et 11, mais des longueurs après suppression des espaces finaux de 7 et 8.

Contrairement à `LEN`, `LEN_TRIM` est une fonction élémentaire (*cf.* 8.5, p. 101).

8.3.5 Recherche de sous-chaîne avec INDEX

La fonction élémentaire `INDEX(STRING=ch1, SUBSTRING=ch2)` rend un entier donnant la position du début de la sous-chaîne *ch2* dans la chaîne *ch1*. Seule la première occurrence est recherchée et la fonction `INDEX` rend `0` si la sous-chaîne cherchée n'est pas présente dans la chaîne *ch1*.

Par défaut, la recherche se fait de gauche à droite dans *ch1*. Mais cette fonction admet un paramètre optionnel `BACK` de type booléen qui, lorsqu'il est fourni avec la valeur `.true.` précise que la recherche se fait en commençant par la droite : dans ce cas, si la sous-chaîne *ch2* n'est pas présente dans *ch1*, `INDEX` rend `LEN(ch1) + 1`.
`INDEX("attente", "te")` donne `3`
`INDEX("attente", "te", BACK=.true.)` donne `6`

8.3.6 Recherche des caractères d'un ensemble avec SCAN

La fonction élémentaire `SCAN(STRING=ch1, SET=ch2)` recherche dans la chaîne *ch1* l'un des caractères de l'ensemble *ch2* (peu importe l'ordre) et rend un entier donnant la position du premier trouvé. Si aucun des caractères de *ch2* n'est présent dans *ch1*, elle rend `0`.

Par défaut, la recherche se fait de gauche à droite dans *ch1*. Mais cette fonction admet un paramètre optionnel `BACK` de type booléen qui, lorsqu'il est fourni avec la valeur `.true.`, précise que la recherche se fait en commençant par la droite : dans ce cas, l'entier rendu est donc la position dans *ch1* du dernier caractère de l'ensemble *ch2*.
`SCAN("aujourd'hui", "ru")` donne `2` pour le premier `u`
`SCAN("aujourd'hui", "ru", BACK=.true.)` donne `10` pour le dernier `u`

8.3.7 Recherche des caractères hors d'un ensemble avec VERIFY

La fonction élémentaire `VERIFY(STRING=ch1, SET=ch2)` recherche dans la chaîne *ch1* le premier caractère *hors de* l'ensemble *ch2* (peu importe l'ordre) et rend un entier donnant la position du premier trouvé. Si tous les caractères de *ch1* appartiennent à *ch2*, elle rend `0`.

Par défaut, la recherche se fait de gauche à droite dans *ch1*. La fonction `VERIFY` admet, comme `SCAN`, un argument optionnel booléen `BACK`, qui, s'il est positionné à `.true.`, précise que la recherche se fait de droite à gauche.
`VERIFY("aujourd'hui", "aeiouy")` donne `3` pour le `j`
`VERIFY("aujourd'hui", "aeiouy", BACK=.true.)` donne `9` pour le `h`

Les fonctions `VERIFY` et `SCAN` jouent des rôles identiques, à ceci près que leurs arguments `SET=`*`ch2`* devraient être des ensembles complémentaires (relativement à l'ensemble des caractères présents dans *`ch1`*) pour qu'elles donnent le même résultat.
`SCAN("aujourd'hui", "dhjr'")` donne 3 pour le `j`
`VERIFY("aujourd'hui", "aiou")` donne 3 pour le `j`

8.3.8 Duplication de chaînes avec `REPEAT`

La fonction `REPEAT(STRING=`*`ch`*`, NCOPIES=`*`n`*`)` duplique *`n`* fois la chaîne *`ch`* et concatène les copies, rendant une chaîne de longueur *`n`* fois celle de *`ch`*.
`REPEAT(" ab ", 3)` donne `" ab  ab  ab "`

8.3.9 Conversion de caractère en entier avec `ICHAR` et `IACHAR`

Les fonctions élémentaires `ICHAR(c)` et `IACHAR(c)` renvoient l'entier donnant le numéro du caractère `c` respectivement dans la variante de type de `c` ou dans le code ASCII. Le résultat dépend de la machine et du compilateur sauf avec `IACHAR` pour les 128 premiers caractères du code ASCII. Par exemple,
`IF ((IACHAR(c) >= IACHAR('0')) .AND. (IACHAR(c) <= IACHAR('9')))` permet de tester si `c` est un caractère numérique.

8.3.10 Conversion d'entier en caractère avec `CHAR` et `ACHAR`

Les fonctions élémentaires `CHAR(`*`i`*` [, KIND=`*`var_char`*`])` et `ACHAR(`*`i`*`)` renvoient le caractère correspondant à l'entier *`i`* respectivement dans la variante de type indiquée par *`var_char`* (la variante `DEFAULT` si le paramètre de variante de type n'est pas précisé) ou dans le code ASCII. Le résultat dépend de la machine et du compilateur sauf avec `ACHAR` en ASCII pour $1 \leqslant i \leqslant 127$.

8.3.11 Comparaison de chaînes de caractères avec `LLT`, `LLE`, `LGE` et `LGT`

Les fonctions élémentaires de comparaison selon l'ordre lexicographique `LLT`, `LLE`, `LGE` et `LGT` admettent deux arguments chaînes de caractères et donnent un résultat booléen. La comparaison s'appuie sur l'ordre lexicographique de l'ASCII alors que les opérateurs logiques associés effectuent la comparaison selon l'ordre lexicographique du jeu de caractères par défaut, qui dépend de la
♥ ⇒ machine. Ces fonctions sont donc plus portables que les opérateurs associés. Si les deux chaînes à comparer n'ont pas la même longueur, la plus courte est complétée par des espaces à droites avant comparaison.

LLT	lexically lower than	<
LLE	lexically lower or equal	<=
LGE	lexically greater or equal	>=
LGT	lexically greater than	>

`IF (LGE(c, '0') .AND. LLE(c, '9'))` permet aussi de tester si le caractère `c` est numérique.

f2003 ### 8.3.12 Le caractère de fin de ligne `NEW_LINE`

Pour permettre l'écriture de fichiers en mode stream (*cf.* 5.2.4, p.41), fortran 2003 a introduit la fonction `NEW_LINE(CHAR=`*`ch`*`)` qui rend le caractère de fin de ligne (new-line associé à `\n` du C) dans la variante de type caractère de son argument.

8.4 Les entrées-sorties de chaînes de caractères

8.4.1 Les entrées-sorties de chaînes formatées

Le descripteur de conversion des chaînes de caractères est soit (*cf.* 5.4.1, p.49) :

– `A` : dans ce cas, le nombre de caractères est celui de la variable associée.

– `An` où ***n*** est un entier spécifiant le nombre de caractères. Si ***n*** est plus grand que la longueur de la chaîne à écrire, on complète par des blancs *à gauche*. Si à l'inverse, ***n*** est plus petit que la longueur de la chaîne à écrire, celle-ci est tronquée [10] aux ***n*** premiers caractères.

Par exemple, si on écrit `"salut"` avec différentes largeurs, on obtient :

format	résultat
`A`	`[salut]`
`A5`	`[salut]`
`A8`	`[ salut]`
`A2`	`[sa]`

8.4.2 Les entrées de chaînes en format libre

En format libre, la lecture des chaînes de caractères peut se faire :

– avec des délimiteurs « `"` » ou « `'` », ce qui permet de lire des chaînes comportant des séparateurs comme l'espace, la virgule ou le « `/` ».

– sans délimiteur et alors la lecture s'arrête au premier séparateur (espace, virgule ou « `/` »).

8.4.3 Les fichiers internes : codage en chaîne de caractères et décodage

L'écriture de données numériques sur des chaînes de caractères ou la lecture de données dans des chaînes de caractères [11] mettent en œuvre les méthodes de conversion évoquées plus haut, où les chaînes dans lesquelles on écrit ou on lit sont qualifiées de fichiers internes (*cf.* 5.2.1, p. 39).

Par exemple, le programme suivant convertit `i` et `j` en chacun 3 caractères qu'il écrit dans la variable `f_int` ; puis il lit dans cette même variable (ou fichier interne) l'entier `k` sur 6 chiffres obtenus par concaténation de ceux de `i` et `j`.

```
1  PROGRAM fich_int
2  IMPLICIT NONE
3  CHARACTER(len=6) :: f_int
4  INTEGER :: i=100 , j=200 , k
5  WRITE(*,*) 'i=', i, 'j=', j
6  WRITE(f_int(1:6), '(2i3.0)') i, j  ! écriture dans le fichier interne f_int
7  ! les chiffres de i et j sont juxtaposés dans la chaîne f_int
8  READ(f_int(1:6), '(i6.0)') k       ! lecture de k sur le fichier interne f_int
9  WRITE(*,*) 'i=', i, 'j=', j, 'k=', k
10 END PROGRAM fich_int
```

Il affiche :

```
 i=  100 j=  200
 i=  100 j=  200 k=  100200
```

On utilise notamment la conversion d'entiers en chaînes de caractères pour composer des formats variables pour effectuer des opérations d'entrée-sortie (*cf.* 5.4.5, p. 53).

8.5 Les tableaux de chaînes de caractères

L'assemblage de plusieurs chaînes de caractères sous forme de tableau de chaînes n'est possible que si toutes les chaînes ont la même longueur, conformément à la structure « rectangulaire » des tableaux. ⇐⚠

10. Ce comportement diffère du cas de l'écriture de données numériques, où la conversion ne se fait pas si la largeur du champ est insuffisante : on écrit alors ***n*** symboles « `*` » .

11. En langage C, ces opérations se font grâce aux fonctions `sprintf` et `sscanf`.

```
CHARACTER(LEN=<longueur>), DIMENSION(<dim>) :: <tab_ch>
```

Si la syntaxe d'accès aux éléments du tableau respecte la syntaxe générale des tableaux, il est aussi possible de référencer directement des sous-chaînes d'un élément du tableau. On remarquera que la fonction `LEN` appliquée à un tableau de chaînes renvoie *un scalaire* entier, qui est la longueur ⚠⇒ *commune* de chacun de ses éléments, alors que la fonction *élémentaire* `LEN_TRIM` renvoie un tableau d'entiers conforme au tableau de chaînes.

```
1  PROGRAM tab_chaines
2  IMPLICIT NONE
3  CHARACTER(LEN=4), DIMENSION(3) :: tab_ch
4  CHARACTER(LEN=4) :: ch
5  INTEGER :: i
6  tab_ch(1:2) = (/ "abcd", "ABCD" /) ! avec constructeur
7  tab_ch(3) = "1234"
8  WRITE(*,*) "lgueur commune des chaines ", LEN(tab_ch)
9  WRITE(*,*) "affichage en colonnes "
10 DO i=1, SIZE(tab_ch)
11   WRITE(*,*)  tab_ch(i)
12 END DO
13 WRITE(*,*) "... en ligne: ", tab_ch
14 ! extraction de sous-chaînes pour construire une chaîne
15 ch = tab_ch(1)(1:2) // tab_ch(2)(3:4)
16 WRITE(*,*) "déduite par extraction ", ch
17 ! autre formulation avec des sous-chaînes
18 ch(1:2) = tab_ch(1)(1:2)
19 ch(3:4) = tab_ch(2)(3:4)
20 WRITE(*,*) "déduite par extraction ", ch
21 ! remplacer les fins de chaîne par 1, 2 ou 3 blancs
22 DO i=1, SIZE(tab_ch)
23   tab_ch(i)(5-i:) = repeat(" ", i)
24 END DO
25 WRITE(*,*) "des blancs en fin "
26 WRITE(*,*) "affichage en colonnes entre [] "
27 DO i=1, SIZE(tab_ch)
28   WRITE(*,*)  "[", tab_ch(i), "]"
29 END DO
30 WRITE(*,*) "LEN(tab_ch) ",  LEN(tab_ch)
31 WRITE(*,*) "LEN_TRIM(tab_ch) ",  LEN_TRIM(tab_ch)
32 END PROGRAM tab_chaines
```

affichage associé

```
lgueur commune des chaines  4
affichage en colonnes
abcd
ABCD
1234
... en ligne: abcdABCD1234
déduite par extraction abCD
déduite par extraction abCD
des blancs en fin
affichage en colonnes entre []
[abc ]
[AB  ]
[1   ]
LEN(tab_ch)  4
LEN_TRIM(tab_ch)  3 2 1
```

8.6 Chaînes et procédures

Les chaînes de caractères peuvent être passées en argument de procédures sans que leur longueur soit indiquée explicitement dans l'argument formel de type chaîne, comme on peut le faire pour le profil des tableaux (si le rang est connu). Au moment de l'appel, la longueur de la chaîne passée en argument effectif est déterminée dans l'appelant et transmise de façon implicite à l'appelé.

Il est aussi possible de déclarer des chaînes de caractères *automatiques* dans les procédures, chaînes dont la longueur dépend d'un argument passé à la procédure.

L'exemple suivant montre une fonction `create_ch` qui rend une chaîne dont la longueur est passée en argument, ainsi qu'un sous-programme `imprime_tab` qui admet en argument un tableau 1D de taille quelconque constitué de chaînes de caractères de longueur quelconque (mais identique).

```
1  MODULE m_proc_ch
2  IMPLICIT NONE
3  CONTAINS
4   FUNCTION create_ch(deb, n)
5    CHARACTER(LEN=n) :: create_ch
6    INTEGER, INTENT(IN) :: deb, n
7    CHARACTER(LEN=n) :: ch ! automatique
8    INTEGER :: k
9    DO k=1, n ! pris dans l'ordre ascii
10     ch(k:k) = ACHAR(k + deb -1 )
11   END DO
12   create_ch = ch
13  END FUNCTION create_ch
14  SUBROUTINE imprime(ch)
15   CHARACTER(LEN=*), INTENT(IN) :: ch
16   WRITE(*,*) "ch de lg ", LEN(ch), "[", ch, "]"
17   RETURN
18  END SUBROUTINE imprime
19  SUBROUTINE imprime_tab(tab_ch)
20   CHARACTER(LEN=*), DIMENSION(:), INTENT(IN) :: tab_ch
21   CHARACTER(LEN=LEN(tab_ch)+2) :: ch ! automatique
22   INTEGER :: i
23   WRITE(*,*) "tableau de ", SIZE(tab_ch), " chaines"
24   WRITE(*,*) " de longueur ", LEN(tab_ch)
25   DO i=1, SIZE(tab_ch)
26     ch = "["// tab_ch(i) // "]"
27     WRITE(*,*) ch
28   END DO
29   RETURN
30  END SUBROUTINE imprime_tab
31 END MODULE m_proc_ch
32
33 PROGRAM proc_chaines
34 USE m_proc_ch
35 CHARACTER(LEN=4) :: ch4
36 CHARACTER(LEN=6) :: ch6
37 CHARACTER(LEN=4), DIMENSION(3) :: tab_ch
38 ch4 = create_ch(33, 4)
39 ch6 = create_ch(33, 6)
40 CALL imprime(ch4)
41 CALL imprime(ch6)
42 tab_ch = (/ "abcd", "ABCD", "1234" /)
43 CALL imprime_tab(tab_ch)
44 END PROGRAM proc_chaines
```

affichage associé

```
ch de lg  4 [!"#$]
ch de lg  6 [!"#$%&]
tableau de  3  chaines
 de longueur  4
[abcd]
[ABCD]
[1234]
```

Chapitre 9

Types dérivés ou structures

Pour représenter des objets plus sophistiqués que ceux représentables avec les types prédéfinis, même avec les tableaux (dont les éléments sont tous de même type), on peut définir des *types dérivés* (derived types) ou *structures*[1], constitués de plusieurs éléments ou *composantes* encore appelées *champs* dont les types peuvent être différents. Si un tableau ne permet de regrouper que des éléments homogènes désignés par leur indice, un type dérivé permet de regrouper des éléments hétérogènes désignés par leur identificateur.

Les types dérivés permettent de représenter des données composites (à l'image par exemple des enregistrements d'un fichier de données) et de les encapsuler afin de simplifier leur manipulation globale par des procédures (*cf.* 9.6, p.108) et des opérateurs dédiés (*cf.* 10.2, p.114). Ces possibilités d'extension du langage, définition de types avec ses opérateurs spécifiques, surdéfinition d'opérateurs (*cf.* 10.2.1, p.115) et la notion de *procédure attachée à un type* (bounded procedure) constituent une des richesses du fortran en tant que langage de haut niveau, permettant d'aborder la programmation orientée objet.

Citons quelques exemples de données qui peuvent être représentées par des types dérivés avec les composantes associées :

- une fiche d'étudiant : nom, prénom, numéro, date de naissance, unités d'enseignement et notes associées ;
- un échantillon de mesures atmosphériques sous ballon sonde à un instant donné : altitude, pression, température, humidité, vitesse, orientation du vent (certains paramètres en entier, d'autres en réel) ;
- un point d'une courbe : coordonnées, texte associé, couleur, poids...

Le nombre de champs d'un type dérivé est généralement plus faible que le nombre d'éléments d'un tableau, mais il est possible d'imbriquer des types dérivés, voire de les étendre (*cf.* 9.3, p. 106), alors qu'on ne peut pas construire des tableaux de tableaux. Il est cependant possible de construire des tableaux de types dérivés et d'insérer des tableaux dans des types dérivés (*cf.* 9.4, p. 107).

9.1 Définition d'un type dérivé

On peut, par exemple, définir un type dérivé `point` dont les trois composantes sont : une lettre d'identification `lettre`, suivie des coordonnées (réelles) `abscisse` et `ordonnee` du point.

```
TYPE :: point
  CHARACTER(LEN=1) :: lettre
  REAL :: abscisse
  REAL :: ordonnee
END TYPE point
```

1. La terminologie du fortran et celle du langage C diffèrent, mais on emploiera parfois les termes du C, en particulier structure pour désigner une *variable* de type dérivé.

langage fortran	type dérivé	composante
langage C	structure	champ

Plus généralement, la syntaxe de définition d'un type dérivé, bien souvent encapsulée dans un module (avant l'instruction `CONTAINS`), est :

```
TYPE [, <accès> ::] <nom_du_type>
  [SEQUENCE]
  <définition d'un champ>
  [<définition d'un champ>] ...
END TYPE [<nom_du_type>]
```

où *accès* permet de préciser sa visibilité `PRIVATE` ou `PUBLIC`. Noter qu'un type dérivé public peut comporter des composantes privées.

Depuis fortran 95, une composante d'un type dérivé peut être initialisée, lors de la définition du type, à une valeur qui sera considérée comme sa valeur par défaut. Noter que cela ne confère pas l'attribut statique aux objets de ce type. ⇐ f95/2003

♠ 9.1.1 L'attribut `SEQUENCE`

La définition d'un type dérivé n'implique pas en général un ordre ni une taille pour le stockage en mémoire des composantes, laissés libres pour le compilateur. Ne pas croire en particulier que la taille d'une variable de type dérivé [2] soit toujours la somme des tailles de ses composantes. En général, pour des raisons d'efficacité, le compilateur procède à un alignement des composantes sur des mots par exemple de 32 bits par adjonction de zéros (zero padding) [3]. On a donc intérêt à déclarer les composantes les plus volumineuses en tête de structure pour limiter ce remplissage. ⇐⚠

Dans certains cas, notamment si on souhaite transmettre des données de type dérivé à une procédure écrite dans un autre langage (hormis le C, pour lequel d'autres techniques d'interopérabilité existent, *cf.* chap. 12), on peut imposer, par l'instruction `SEQUENCE`, le stockage des composantes par concaténation en respectant l'ordre de définition. S'il s'agit de structures imbriquées, il est évidemment nécessaire que toutes les sous-structures aient été définies avec l'attribut `SEQUENCE`.

9.2 Référence et affectation des structures

9.2.1 Constructeur de type dérivé

Une fois le type dérivé `point` défini, on peut déclarer les variables `a` et `b` du type `point`, et leur affecter une valeur en utilisant le *constructeur de structure*, qui est une fonction (automatiquement créée) nommée comme le type dérivé. Appeler le constructeur est par exemple nécessaire pour initialiser une structure lors de sa déclaration (comme pour `b` ci-après) :

```
TYPE(point) :: a, b = point('B', 1., -1.) ! initialisation => statique
a = point('A', 0., 1.)                    ! affectation
```

En fortran 95, les composantes doivent être passées au constructeur en respectant leur position dans le type dérivé, mais depuis fortran 2003, elles peuvent aussi être désignées par mot-clef. Les composantes initialisées sont alors considérées comme des arguments optionnels du constructeur. ⇐ f2003

```
TYPE(point) :: a
a = point(abscisse = 0., lettre ='A', ordonnee = 1.) ! fortran 2003
```

9.2.2 Sélecteur de champ %

Références et affectations à une variable de type dérivé peuvent concerner la structure globale. Mais seul l'opérateur d'affectation est défini par défaut pour les types dérivés ; les autres opérateurs ⇐⚠

2. On peut le vérifier en affichant la taille en bits de cette variable, grâce à la fonction `STORAGE_SIZE`.
3. Mais il existe des options de compilation comme `-fpack-derived` sous `g95` ou `gfortran` (*cf* F.5.6, p. 174), `-noalign records` de `ifort`, pour éviter les octets de remplissage, au détriment des performances.

(y compris le test d'égalité ==) doivent être étendus par l'utilisateur (*cf.* chap 10).

Par exemple :

```
a = b          ! suppose a et b de même type dérivé
IF(a==b) WRITE(*,*) "a=b" ! interdit car opérateur de comparaison pas défini
```

affecte au `point a` les mêmes composantes que celles du `point b`.

Les entrées/sorties globales de la structure sont équivalentes à celles de la liste des composantes ultimes, dans l'ordre de la définition du type. Elles peuvent se faire au format libre ou en spécifiant autant de descripteurs de format actifs que de composantes ultimes.

```
WRITE(*, *)  a ! affiche (dans l'ordre) les champs qui constituent la structure a
WRITE(*,'("a= ", A, " x=", F6.2, " y=", F6.2)')  a
```

affiche successivement :

```
 A   0.00000000       1.00000000
a= A x=  0.00 y=  1.00
```

Mais on peut aussi accéder individuellement à chacun des champs qui composent la structure grâce au sélecteur de champ % pour les affecter :

```
a%lettre = 'A'
a%abscisse = 0.
a%ordonnee = 1.
```

ou pour les référencer, y compris dans des expressions :

```
CHARACTER(LEN=1) :: c
c = a%lettre
WRITE(*, *) a%abscisse, a%ordonnee
WRITE(*, *) sqrt(a%abscisse**2 + a%ordonnee**2)
```

9.3 Imbrication et extension de structures

9.3.1 Imbrication de structures

Les structures peuvent comporter des composantes qui sont elles-mêmes des structures à condition que les types des composantes aient été définis préalablement. On peut par exemple définir une structure `cercle` dont un champ est une structure `point` :

```
TYPE :: cercle
  TYPE(point) :: centre ! de type défini au préalable
  REAL :: rayon
END TYPE cercle
```

et déclarer ensuite une variable de type `cercle` et lui affecter une valeur, par exemple grâce à son constructeur. Le sélecteur % permet d'accéder aux composantes ultimes du type dérivé imbriqué.

```
TYPE(cercle) :: rond
rond = cercle(b, 2.) ! où b est de type point
WRITE(*, *) 'rond = ', rond
WRITE(*, *) 'rond = ', rond%centre, rond%rayon
WRITE(*, *) 'abscisse du centre :', rond%centre%abscisse
WRITE(*, *) 'périmètre :', 2.* 3.14159 * rond%rayon
```

f2003

9.3.2 Extension d'un type dérivé

La norme fortran 2003 prévoit aussi que l'on puisse étendre un type dérivé en lui adjoignant des composantes, grâce à l'attribut `EXTENDS`[4]. Ce mécanisme permet d'hériter des outils conçus pour le type parent (*cf.* 9.7.1, 111). Le constructeur du type étendu admet deux syntaxes (voir exemple ci-après) :

- la syntaxe classique (pour `circle1`) où l'on passe toutes les composantes ;
- le passage d'un objet du type parent (`a` de type `point`) et des seules composantes nécessaires (ici le rayon) pour l'étendre.

```
TYPE, EXTENDS(point) :: pt_ext_cerc
  REAL :: rayon
END TYPE pt_ext_cerc
TYPE(pt_ext_cerc) :: circle1, circle2
circle1 = pt_ext_cerc('B', 2., -0., 2.) ! constructeur du type étendu
circle2 = pt_ext_cerc(a, 2.) ! constructeur à partir du type parent
```

N.-B. : un type dérivé portant l'attribut `SEQUENCE` (*cf.* 9.1.1, p. 105) ou `BIND` (*cf.* 12.2 p. 132) n'est pas extensible. ⇐⚠

L'accès aux composantes du type parent peut se faire soit directement, soit via le type parent :

```
WRITE(*,*) "circle1%lettre = ", circle1%lettre ! accès direct
WRITE(*,*) "circle1%point%lettre = ", circle1%point%lettre ! via type parent
```

9.4 Structures et tableaux

On peut définir des tableaux de structures :

```
TYPE(point), DIMENSION(10) :: courbe
```

définit un tableau de 10 éléments de type `point`. Alors `courbe%lettre` est un tableau de 10 caractères, alors que `courbe(2)` est une structure de type `point`.

On peut aussi définir des types dérivés dont certains champs sont des tableaux. En fortran 95, le profil de ces tableaux doit être connu au moment de la compilation.

```
TYPE :: point3d
  CHARACTER(LEN=1) :: lettre
  REAL, DIMENSION(3) :: coordonnees (réels)
END TYPE point3d
TYPE(point3d) :: m
m = point3d('M', (/1., 2., 3./) ) ! coordonnées entières interdites
WRITE(*, *)  m
WRITE(*,*) 'coordonnée 2 de m ', m%coordonnees(2)
```

On peut aussi définir des tableaux de structures de ce type :

```
TYPE(point3d), DIMENSION(10) :: courbe3d
```

Alors `courbe3d%coordonnees(1)` est le tableau d'étendue 10 des coordonnées x des points de la courbe en trois dimensions. De façon similaire, `courbe3d(4)%coordonnees` désigne le tableau 1D des 3 coordonnées du 4e point de la courbe.

Bien noter qu'en revanche, la notation `courbe3d%coordonnees` n'est pas autorisée car elle ⇐⚠

4. Mais cette possibilité n'est pas encore implémentée dans tous les compilateurs, notamment dans `g95`.

supposerait de composer un tableau 2D à partir des indices des objets dérivés et de ceux de leurs coordonnées. La norme précise que dans une telle expression, au plus une des références peut être de rang non nul (c'est-à-dire non scalaire).

f2003 9.5 Types dérivés à composantes allouables dynamiquement

Pour manipuler des structures comportant des tableaux de dimensions allouables dynamiquement en fortran 95, il était nécessaire (comme en C) de faire appel aux pointeurs (*cf.* chap. 11). Mais ces limitations ont disparu avec la norme fortran 2003 [5].

Dans ce cas, l'appel du constructeur ainsi que l'opérateur d'affectation se chargent de l'allocation des composantes allouables (par un mécanisme déjà évoqué pour l'allocation «implicite» des tableaux par affectation, *cf.* 7.5.4, p. 94). L'opération de copie d'une variable de type dérivé comportant des composantes de taille variable (tableau, chaîne...) réalise donc une *copie profonde* (deep copy), où les données elles-mêmes sont dupliquées. À l'inverse, en fortran 95 ou en langage C, la structure ne comporte que des pointeurs vers les données dynamiques et la recopie simple par affectation ne duplique que les pointeurs, réalisant ainsi une *copie superficielle* (shallow copy). Il est nécessaire de créer une fonction de recopie profonde qui prend en charge l'allocation des données dynamiques que l'on souhaite dupliquer.

```
TYPE :: point
  CHARACTER(LEN=1) :: lettre
  REAL, DIMENSION(:), ALLOCATABLE :: coord
END TYPE point
 ...
TYPE(point) :: a,  b    ! déclaration
a = point('A', (/1. ,-1./) ) ! construction de a => allocation de a%coord
WRITE(*,*) 'a ', a%lettre, a%coord   ! affichage de a composante par composante
! WRITE(*,*) 'a ', a ! pas autorisé en fortran 2003
b = a ! affectation globale => allocation de b%coord
```

⚠⇒ Noter qu'il n'est pas possible de lire ou d'écrire globalement une variable de type dérivé à composante allouable ou d'attribut pointeur via `READ` ou `WRITE`, à moins de définir soi-même des procédures d'entrée et de sortie spécifiques du type dérivé, en précisant éventuellement le format de conversion `DT`.

9.6 Procédures agissant sur les structures

La transmission de structures en argument de procédures externes suppose que les types dérivés soient connus de l'ensemble des procédures qui les manipulent. La méthode la plus fiable [6] consiste à insérer la définition du type dérivé dans un module (*cf.* 6.4, p. 65) qui sera appelé par toutes les procédures utilisant ce type.

```
1 MODULE m_point ! module de définition du type point
2   IMPLICIT NONE
3   TYPE point
4     CHARACTER(len=1) :: lettre
5     REAL :: abscisse
6     REAL :: ordonnee
7   END TYPE point
```

5. Des options spécifiques de certains compilateurs l'autorisent aussi comme extension de la norme fortran 95. En particulier `xlf` d'IBM, le compilateur NAG et `g95` avec l'option `-std=f2003` ou `-std=f95` assortie de `-ftr15581` (*cf.* F.6.1, p. 175) permettent les composantes allouables dans les structures.

6. On pourrait aussi employer des interfaces explicites, mais il faudrait alors préciser l'attribut `SEQUENCE` lors de la définition du type dérivé. De plus, la duplication des définitions des structures risque d'aboutir à des définitions incohérentes lors de la mise au point du code

```
8  END MODULE m_point
9  MODULE m_sym ! module de la procédure de calcul du symétrique
10   USE m_point
11   IMPLICIT NONE
12   CONTAINS
13   SUBROUTINE sym(m, n)  ! sous programme de calcul du symétrique
14     TYPE(point), INTENT(in) :: m
15     TYPE(point), INTENT(out) :: n
16     n = point (m%lettre, m%ordonnee, m%abscisse) ! échange des coordonnées
17   END SUBROUTINE sym
18 END MODULE m_sym
19 PROGRAM sym_point       ! programme principal
20   USE m_point ! pour connaître le type point, mais redondant à cause du suivant
21   USE m_sym   ! pour connaître l'interface de la procédure sym
22   IMPLICIT NONE
23   TYPE(point) :: a,  b    ! déclaration
24   a = point('A', 1. ,-1.) ! construction de a
25   WRITE(*,*) 'a ', a      ! affichage de a
26   CALL sym(a, b)          ! appel de sym
27   WRITE(*,*) 'b ', b      ! affichage de b
28 END PROGRAM sym_point
```

9.7 Procédures attachées à un type dérivé

Dans le contexte de la programmation orientée objet, il est possible d'attacher à un type dérivé un ensemble de procédures (bound procedures), qui apparaîtront comme des composantes du type dérivé : on les qualifie alors de *méthodes*. L'ensemble, données et méthodes constitue alors un *objet*.

Elles sont introduites après le mot-clef `CONTAINS`, à l'intérieur de la définition du type, par la déclaration `PROCEDURE`, suivie du nom de la méthode. Le sélecteur de composantes `%` permet ensuite de les référencer pour les appliquer à une structure. Par défaut, cette structure sera passée *implicitement* comme premier argument lors de l'appel à la procédure ; on peut le préciser par l'attribut `PASS`. Au contraire, il faut préciser `NOPASS` pour éviter cette transmission implicite.

Les procédures attachées au type dérivé sont définies en dehors du type, comme des procédures classiques, après le mot-clef `CONTAINS`. Leur définition (sauf en cas de déclaration avec `NOPASS`) doit comporter explicitement la structure (souvent nommée `this` ou `self`) comme premier argument.

L'exemple suivant présente plusieurs procédures permettant de déterminer le symétrique d'un point du plan par rapport à la première bissectrice des axes (échange entre abscisse et ordonnée) :

- le sous-programme `bound_sym_sub` attaché au type `point`, que l'on peut appeler avec l'instruction `CALL a%bound_sym_sub(b)`, où la structure `a` est transmise implicitement ;
- la fonction `bound_sym_fct` attachée au type `point` que l'on peut appeler avec l'expression `a%bound_sym_sub()`, où la structure `a` est transmise implicitement ;
- leurs variantes `bound_nopass_sym_sub` et `bound_nopass_sym_fct` avec l'attribut `NOPASS`, où la structure n'est pas transmise implicitement ;
- le sous-programme `sym` non attaché au type `point`, qui n'est donc pas accessible avec le sélecteur `%`.

Seules celles attachées au type `point` sont déclarées dans le type (lignes 14 à 17).

```
6  MODULE m_point ! module de définition du type point
7    IMPLICIT NONE
8    TYPE :: point
9      CHARACTER(len=1) :: lettre
10     REAL :: abscisse
11     REAL :: ordonnee
```

```
12      CONTAINS ! procédures attachées au type point
13        ! symétries par rapport à y=x :  échange entre abscisse et ordonnée
14        PROCEDURE, PASS :: bound_sym_fct ! PASS = objet passé en 1er arg
15        PROCEDURE, PASS :: bound_sym_sub
16        PROCEDURE, NOPASS :: bound_nopass_sym_fct ! NOPASS
17        PROCEDURE, NOPASS :: bound_nopass_sym_sub !
18    END TYPE point
19    CONTAINS
20    FUNCTION bound_sym_fct(this)
21      CLASS(point), INTENT(in) :: this ! extension polymorphe du type
22      TYPE(point) :: bound_sym_fct
23      bound_sym_fct = point(this%lettre, this%ordonnee, this%abscisse)
24    END FUNCTION bound_sym_fct
25    SUBROUTINE bound_sym_sub(this, p_sym)
26      CLASS(point), INTENT(in) :: this ! extension polymorphe du type
27      TYPE(point), INTENT(out) :: p_sym
28      p_sym = point(this%lettre, this%ordonnee, this%abscisse)
29    END SUBROUTINE bound_sym_sub
30    FUNCTION bound_nopass_sym_fct(p)
31      TYPE(point) :: bound_nopass_sym_fct
32      TYPE(point), INTENT(in) :: p ! CLASS(point), INTENT(in) :: p ! possible aussi
33      bound_nopass_sym_fct = point(p%lettre, p%ordonnee, p%abscisse)
34    END FUNCTION bound_nopass_sym_fct
35    SUBROUTINE bound_nopass_sym_sub(p_sym, p)
36      TYPE(point), INTENT(out) :: p_sym
37      CLASS(point), INTENT(in) :: p
38      p_sym = point(p%lettre, p%ordonnee, p%abscisse)
39    END SUBROUTINE bound_nopass_sym_sub
40  END MODULE m_point
41  MODULE m_sym ! module de la procédure de calcul du symétrique
42    USE m_point
43    IMPLICIT NONE
44    CONTAINS
45    SUBROUTINE sym(m, n)  ! sous programme (non attaché) de calcul du symétrique
46      TYPE(point), INTENT(in) :: m
47      TYPE(point), INTENT(out) :: n
48      n = point(m%lettre, m%ordonnee, m%abscisse)
49    END SUBROUTINE sym
50  END MODULE m_sym
51  PROGRAM sym_point       ! programme principal
52    USE m_point  ! pour connaître le type point (pas nécessaire à cause du suivant)
53    USE m_sym    ! pour connaître l'interface de la procédure sym
54    IMPLICIT NONE
55    TYPE(point) :: a, b, c, d, e, f, g  ! déclarations
56    a = point('A', 1. ,-2.) ! construction de a
57    WRITE(*,*) 'a ', a      ! affichage de a
58    CALL sym(a, b)          ! appel de sym non attachée au type
59    WRITE(*,*) 'b ', b      ! affichage de b
60    ! procedures PASS => a passé implicitement
61    c = a%bound_sym_fct()   ! avec fct attachée au type (a passé implicitement)
62    WRITE(*,*) 'c ', c      ! affichage de c
63    CALL a%bound_sym_sub(d) ! avec subroutine attachée au type
64    WRITE(*,*) 'd ', d      ! affichage de d
65    ! procedures NOPASS => a passé explicitement
66    e = a%bound_nopass_sym_fct(a)  ! avec fct attachée au type
67    WRITE(*,*) 'e ', e      ! affichage de e
```

```
68   CALL bound_nopass_sym_sub(f, a) ! avec subroutine attachée au type
69   WRITE(*,*) 'f ', f      ! affichage de f
70   CALL a%bound_nopass_sym_sub(g, a) ! avec subroutine attachée au type
71   WRITE(*,*) 'g ', g      ! affichage de g
72 END PROGRAM sym_point
```

Noter que, afin d'autoriser l'application des procédures attachées aux types étendus déduits du type `point` (mécanisme d'héritage), l'argument `this` est déclaré avec le mot-clef `CLASS` (*cf.* ligne 21 par exemple) au lieu de `TYPE`.

Dans le cas où l'attribut `NOPASS` a été précisé, même si l'appel se fait comme composante du type, il faut passer explicitement la structure comme argument si on veut l'utiliser dans la procédure attachée (*cf.* lignes 66 et 70).

9.7.1 Un exemple élémentaire d'héritage

Si l'on étend le type `point` au type `point_num` par ajout d'une composante, on peut lui appliquer les procédures attachées au type parent. C'est l'intérêt de la déclaration `CLASS(point)` au lieu de `TYPE(point)` de leur argument implicite. Le mot-clef `CLASS` est une extension de `TYPE` permettant de désigner des objets *polymorphes* à *typage dynamique* : il s'applique aux arguments muets, aux entités allouables et aux pointeurs.

Dans l'exemple suivant, on ajoute un sous-programme `bound_sym_sub_ext` dont l'argument de sortie est de *classe* `point` :

- dans le cas particulier où l'argument effectif est plus précisément de *type* `point_num`, la procédure construit une structure de type `point_num` ;
- sinon, la procédure construit simplement une structure de type `point`.

```
5  MODULE m_point ! module de définition du type point et de ses extensions
6    IMPLICIT NONE
7    TYPE point
8      CHARACTER(len=1) :: lettre
9      REAL :: abscisse
10     REAL :: ordonnee
11     CONTAINS ! procédures attachées au type point
12       ! symétries par rapport à y=x :  échange entre abscisse et ordonnée
13       PROCEDURE, PASS :: bound_sym_fct ! PASS = l'objet est passé en 1er arg
14       PROCEDURE, PASS :: bound_sym_sub
15       PROCEDURE, PASS :: bound_sym_sub_ext ! rend un objet polymorphe
16   END TYPE point
17   TYPE, EXTENDS (point) :: point_num ! ajoute une composante à point
18     INTEGER :: num = -100 ! avec valeur par défaut
19   END TYPE point_num
20   CONTAINS
21   FUNCTION bound_sym_fct(this)
22     CLASS(point), INTENT(in) :: this ! extension polymorphe du type
23     TYPE(point) :: bound_sym_fct
24     bound_sym_fct = point(this%lettre, this%ordonnee, this%abscisse)
25   END FUNCTION bound_sym_fct
26   SUBROUTINE bound_sym_sub(this, p_sym)
27     CLASS(point), INTENT(in) :: this ! extension polymorphe du type
28     TYPE(point), INTENT(out) :: p_sym
29     p_sym = point(this%lettre, this%ordonnee, this%abscisse)
30   END SUBROUTINE bound_sym_sub
31   SUBROUTINE bound_sym_sub_ext(this, p_sym)
32     CLASS(point), INTENT(in) :: this ! extension polymorphe du type
33     CLASS(point), INTENT(out) :: p_sym ! idem
34     ! on ne peut pas faire de copie car pas forcement de même type
35     ! p_sym%point = this%point ! impossible
36     p_sym%lettre = this%lettre
```

```
37      p_sym%abscisse = this%ordonnee
38      p_sym%ordonnee = this%abscisse
39      ! les autres composantes éventuelles ont les valeurs par défaut
40      ! sauf avec les combinaisons de types qui suivent
41      SELECT TYPE(p_sym) ! choix en fonction du type
42      TYPE IS (point_num) ! enrichir p_sym s'il est de type point_num
43        SELECT TYPE (this) !
44        TYPE IS (point_num) ! si les 2 sont des point_num, la copie serait possible
45              p_sym%num = -this%num
46        CLASS DEFAULT ! si p_sym de type point_num, mais this ne l'est pas
47              p_sym%num = 10000 ! sinon on prendrait la valeur par défaut
48        END SELECT
49      END SELECT
50    END SUBROUTINE bound_sym_sub_ext
51  END MODULE m_point
52  PROGRAM sym_point       ! programme principal
53    USE m_point  ! pour connaître le type point (pas nécessaire à cause du suivant)
54    IMPLICIT NONE
55    TYPE(point) :: a, b, c, d ! déclarations
56    TYPE(point_num) :: an, bn, cn, dn, en, fn
57    a = point('A', 1. ,-1.) ! construction de a
58    WRITE(*,*) 'a ', a      ! affichage de a
59    b = a%bound_sym_fct()   ! avec fct attachée au type
60    WRITE(*,*) 'b ', b
61    call a%bound_sym_sub(c) ! avec subroutine attachée au type
62    WRITE(*,*) 'c ', c
63    an%point = a
64    an%num = 11
65    WRITE(*,*) 'an ', an    ! affichage d'un point numéroté
66    bn = point_num('B', 2., -2., 20) ! constructeur explicite
67    WRITE(*,*) 'bn ', bn    ! affichage d'un point numéroté
68    d = an%bound_sym_fct()  ! méthode héritée qui donne un point simple
69    WRITE(*,*) 'd ', d    ! affichage d'un point simple
70    dn%point = d
71    dn%num = an%num
72    WRITE(*,*) 'dn ', dn    ! affichage d'un point numéroté
73    call an%bound_sym_sub_ext(cn)
74    WRITE(*,*) 'cn ', cn    ! affichage d'un point numéroté
75    en = point_num('E', 20., -20.) ! constructeur explicite sauf arg par défaut
76    WRITE(*,*) 'en ', en    ! affichage d'un point numéroté
77    call a%bound_sym_sub_ext(fn)
78    WRITE(*,*) 'fn ', fn    ! affichage d'un point numéroté
79  END PROGRAM sym_point
```

Cette procédure utilise la structure de contrôle `SELECT TYPE` (*cf.* lignes 41 et 43) qui s'applique aux variables dont le type peut varier à l'exécution (typage dynamique) selon la syntaxe générale suivante :

```
[<nom>:] SELECT TYPE(<class_var>)
  [TYPE IS <type_dérivé>
       <bloc d'instructions>]
  [TYPE IS <type_intrinsèque>
       <bloc d'instructions>]
  [CLASS IS <type_dérivé>
       <bloc d'instructions>]
  [CLASS DEFAULT
       <bloc d'instructions>]
END SELECT [<nom>]
```

Chapitre 10

Généricité, surcharge d'opérateurs

Dans ce chapitre, on introduit, à partir de quelques exemples, certains aspects novateurs du fortran, comme la généricité et la surcharge d'opérateurs, qui, associés aux modules, au contrôle de visibilité et aux pointeurs, permettent d'aborder les fonctionnalités d'un langage orienté objet.

10.1 Procédures génériques et spécifiques

Le fortran 77 comportait déjà la notion de procédure générique, mais elle était réservée aux procédures intrinsèques. Avec les versions 90 et au delà, cette notion devient accessible pour les procédures définies par l'utilisateur. En fortran [1], de nombreuses fonctions intrinsèques numériques (*cf.* annexe A, p. 140) invoquables sous leur nom *générique* font en réalité appel, suivant le type de leurs paramètres effectifs, à une version *spécifique* particulière. Par exemple, la fonction générique intrinsèque `MAX` référence les fonctions spécifiques :

- `MAXO` si les arguments sont entiers ;
- `AMAX1` si les arguments sont réels ;
- `DMAX1` si les arguments sont double précision.

Pour regrouper sous une même procédure générique plusieurs procédures spécifiques, il faut déclarer une interface commune *obligatoirement nommée* et ensuite dresser la liste des procédures spécifiques qui seront effectivement appelées, avec chacune leur interface. Au moment de l'appel de la procédure générique le choix de la procédure spécifique sera fait en fonction :

- du nombre des arguments effectifs d'appel ;
- et du type de ces arguments.

Par exemple dans le cas de fonctions [2] :

```
INTERFACE <nom_générique>
   FUNCTION <nom_spécifique_1(...)>
     ...
   END FUNCTION <nom_spécifique_1>
   ...
   FUNCTION <nom_spécifique_n(...)>
     ...
   END FUNCTION <nom_spécifique_n>
END INTERFACE <nom_générique>
```

1. En langage C, seule la norme C99 permet grâce à `#include <tgmath.h>`, d'utiliser un nom générique pour les fonctions **mathématiques**. Par exemple suivant le type de son argument, la fonction générique `exp` appellera la fonction spécifique `expf` pour un `float`, `exp` pour un `double` et `expl` pour un `long double`. La généricité est en revanche parfaitement disponible en C++.
2. Depuis fortran 95, il est possible de rappeler le nom générique à la fin de l'instruction `END INTERFACE`.

Dans l'interface générique, au lieu de recopier les interfaces des procédures spécifiques, on peut
f2003 ⇒ se contenter d'en donner la liste, chacune précédée par `MODULE`[3] `PROCEDURE`, à condition que chaque procédure spécifique possède une interface explicite visible dans le module hébergeant l'interface générique.

Par exemple, pour définir une fonction générique `moy` qui calcule la moyenne (en flottant) d'un tableau de réels ou d'entiers, il faut définir deux fonctions spécifiques[4] `moy_reel` et `moy_int` que l'on peut placer dans un module :

```
1  MODULE m_moyenne
2  IMPLICIT NONE
3  INTERFACE moy ! interface générique pour une famille de procédures spécifiques
4      PROCEDURE :: moy_reel ! version avec tableau de réels  (:: en f2008)
5      PROCEDURE :: moy_int  ! version avec tableau d'entiers
6  ! le choix se fera sur le type de l'argument
7  END INTERFACE moy ! nommage possible en fortran 95 seulement
8  CONTAINS
9    FUNCTION moy_reel(tab_r)  !  version réels
10     REAL :: moy_reel
11     REAL, DIMENSION(:), INTENT(in) :: tab_r
12     moy_reel = SUM(tab_r(:)) / REAL(SIZE(tab_r(:)))
13   END FUNCTION moy_reel
14   FUNCTION moy_int(tab_i)   ! version entiers
15     REAL :: moy_int
16     INTEGER, DIMENSION(:), INTENT(in) :: tab_i
17     moy_int = REAL(SUM(tab_i(:))) / REAL(SIZE(tab_i(:)))
18   END FUNCTION moy_int
19 END MODULE m_moyenne
20
21 PROGRAM t_moyenne
22    USE m_moyenne
23    IMPLICIT NONE
24    INTEGER :: i
25    INTEGER, DIMENSION(10) :: ti =(/ (i,  i=1, 10) /)
26    REAL, DIMENSION(10) :: tr
27    tr(:) = REAL(ti(:)) / 10.
28    ! appel de la procédure avec le nom générique (moy)
29    WRITE(*,*) moy( tr(:) ) ! argument tableau de réels => appelle moy_reel
30    WRITE(*,*) moy( ti(:) ) ! argument tableau d'entiers => appelle moy_int
31 END PROGRAM t_moyenne
```

Dans le cas de procédures externes dont le code source est inaccessible ou ne peut pas être modifié, il suffit de fournir explicitement l'interface des procédures spécifiques pour pouvoir les intégrer dans une interface générique. C'est ce qui est pratiqué pour les bibliothèques.

10.2 L'interface-opérateur

En fortran, les opérateurs intrinsèques sont soit monadiques, comme `+`, `-` et `.not.` suivis de leur opérande, soit plus souvent dyadiques et alors entourés par leurs deux opérandes. À partir de fortran 90, il est possible :

- de définir de nouveaux opérateurs ;
- et d'étendre la portée des opérateurs intrinsèques du fortran (*surcharge*, overloading en anglais) à des types pour lesquels leur action n'est pas définie par le langage.

3. En fortran 2003, le mot-clef `MODULE` est devenu facultatif. En fortran 2008, on peut utiliser `PROCEDURE ::` et le faire suivre d'une liste.

4. Ces procédures spécifiques pourraient être qualifiées de `PRIVATE`, accessibles seulement via l'interface générique.

Les procédures spécifiques qui permettent de définir l'action d'un opérateur rendent un résultat qui peut être utilisé dans une expression ; elles doivent donc être des *fonctions* :

- à un argument (opération monadique) ;
- ou deux arguments (opération dyadique).

```
INTERFACE OPERATOR (<opérateur>)
   FUNCTION nom_spécifique_1(...)
     ...
   END FUNCTION nom_spécifique_1
   ...
   FUNCTION nom_spécifique_n(...)
     ...
   END FUNCTION nom_spécifique_n
END INTERFACE
```

10.2.1 Surcharge d'opérateurs

Pour surcharger un opérateur, seules doivent être fournies les fonctions qui définissent le rôle de l'opérateur sur les types pour lesquels il n'est pas intrinsèquement défini. Le choix entre les fonctions spécifiques est fait en fonction du nombre des arguments (1 ou 2) et de leur type. La priorité de l'opérateur surchargé est celle de l'opérateur intrinsèque (*cf.* 3.5, p. 26).

À titre d'exemple, l'opérateur `//` réalise la concaténation entre deux chaînes de caractères et on peut l'étendre, en tant qu'opérateur dyadique [5] sur des réels, au calcul de la résistance équivalente à une association de deux résistances en parallèle.

```
1  MODULE m_resist
2  ! l'opérateur // est défini entre deux chaînes de caractères seulement
3  ! on le surcharge pour les réels (représentant ici des résistances par ex)
4  INTERFACE OPERATOR(//)
5     MODULE PROCEDURE parallele ! liste des fonctions qui étendent l'opérateur
6  END INTERFACE
7  CONTAINS
8     FUNCTION parallele(r1,r2)
9        IMPLICIT NONE
10       REAL :: parallele
11       REAL, INTENT(in) :: r1, r2
12       parallele = 1. / (1./r1 + 1./r2)
13       RETURN
14    END FUNCTION parallele
15 END MODULE m_resist
16 !
17 PROGRAM elec
18    USE m_resist
19    IMPLICIT NONE
20    REAL :: r1, r2
21    WRITE(*,*) ' entrer deux valeurs de résistances'
22    READ *, r1, r2
23    WRITE(*,*) 'r1 = ', r1, ' r2 = ', r2
24    WRITE(*,*) 'r1 '//'série r2 = ' , r1+r2  ! utilisation de // natif
25    WRITE(*,*) 'r1 // r2 = ' , r1//r2   ! utilisation de // surchargé
26 END PROGRAM elec
```

5. L'opérateur `//` ne pourrait pas être défini comme monadique.

10.2.2 Création d'opérateurs

Un opérateur nouveau est introduit (identificateur entouré par des points, comme les opérateurs de comparaison) suivant une syntaxe similaire à celle de la surcharge des opérateurs existants. Le nouvel opérateur a la priorité maximale s'il est unaire et la plus faible s'il est binaire (*cf.* 3.5, p. 26).

Par exemple, on peut définir l'opérateur `.para.` d'association en parallèle :

- pour les réels, en particulier pour les résistances ;
- pour un type dérivé `capacite` représentant les capacités.

```
MODULE m_parallele
IMPLICIT NONE
TYPE :: capacite
   REAL :: capa
END TYPE capacite
INTERFACE OPERATOR(.para.)
! liste des fonctions qui définissent l'opérateur
   MODULE PROCEDURE parallele_res    ! pour les réels (dont les résistances)
   MODULE PROCEDURE parallele_capa   ! pour les capacités (type dérivé)
END INTERFACE
CONTAINS
   FUNCTION parallele_res(r1,r2)
      REAL :: parallele_res
      REAL, INTENT(in) :: r1, r2
      parallele_res = 1. / (1./r1 + 1./r2)
      RETURN
   END FUNCTION parallele_res
   FUNCTION parallele_capa(c1,c2)
      TYPE(capacite) :: parallele_capa
      TYPE(capacite), INTENT(in) :: c1, c2
      parallele_capa%capa = c1%capa + c2%capa
      RETURN
   END FUNCTION parallele_capa
END MODULE m_parallele
!
PROGRAM elec
   USE m_parallele
   IMPLICIT NONE
   REAL :: r1, r2
   TYPE(capacite) :: c1, c2, c3
   WRITE(*,*) ' entrer deux valeurs de résistances'
   READ(*,*) r1, r2
   WRITE(*,*) 'r1 = ', r1, ' r2 = ', r2
   WRITE(*,*) 'r1 // r2 = ' , r1.para.r2   ! .para. appelle parallele_res
   WRITE(*,*) ' entrer deux valeurs de capacités'
   READ(*, *) c1%capa, c2%capa
   WRITE(*,*) 'c1 = ', c1%capa, ' c2 = ', c2%capa
   c3 = c1.para.c2                         ! .para. appelle parallele_capa
   WRITE(*,*) 'c1 // c2 = ' , c3%capa
END PROGRAM elec
```

10.3 L'interface-affectation

En fortran le symbole = n'est pas un opérateur de comparaison, mais le symbole de l'affectation d'une expression à une variable. Si le type de la variable est différent du type de l'expression, cette affectation se fait grâce à une conversion implicite de type.

Il est possible d'étendre la portée du symbole d'affectation à des types pour lesquels son sens n'est pas défini intrinsèquement. On parle alors de *surcharge de l'affectation* et on utilise une syntaxe particulière qui regroupe les interfaces des *sous-programmes* à deux arguments (le premier est la variable à affecter ou membre de gauche, le second l'expression qui lui sera affectée ou membre de droite) qui définissent le sens de cette affectation. Le choix du sous-programme spécifique à appeler se fait en fonction du type des deux arguments.

Dans l'exemple suivant, on définit un type `hms`, heures, minutes, secondes. En surchargeant l'affectation (*cf.* ligne 10 et suivantes), on construit les conversions en secondes (type entier) et heures fractionnaires et les conversions réciproques. Via ces conversions, il est possible de surcharger les opérateurs `+`, `-` (monadique `opp_hms`, ligne 84 et dyadique `diff_hms`, ligne 75), `*` et `>`.

```
1  ! module de définition du type hms (heures, minutes,secondes)
2  MODULE m_hms
3  IMPLICIT NONE
4  TYPE hms
5    INTEGER :: heures
6    INTEGER :: minutes
7    INTEGER :: secondes
8  END TYPE hms
9  ! interfaces des opérateurs pour le type hms
10 ! surcharge de l'affectation pour conversion
11 INTERFACE ASSIGNMENT (=)
12   MODULE PROCEDURE hms2hfrac ! hms -> heures fractionnaires (type réel)
13   MODULE PROCEDURE hms2s     ! hms -> secondes (type entier)
14   MODULE PROCEDURE hfrac2hms ! heures fractionnaires (type réel) -> hms
15   MODULE PROCEDURE s2hms     ! secondes (type entier) -> hms
16 END INTERFACE
17 INTERFACE OPERATOR (+) ! surcharge de l'addition
18   MODULE PROCEDURE som_hms
19 END INTERFACE
20 INTERFACE OPERATOR (-) ! surcharge  dyadique et monoadique de "-"
21   MODULE PROCEDURE diff_hms
22   MODULE PROCEDURE opp_hms
23 END INTERFACE
24 INTERFACE OPERATOR (*) ! surcharge de la multiplication: entier par hms
25   MODULE PROCEDURE mult_hms
26 END INTERFACE
27 INTERFACE OPERATOR (>) ! surcharge de l'opérateur > (supérieur)
28   MODULE PROCEDURE sup_hms
29 END INTERFACE
30 CONTAINS
31   ! surcharges de l'affectation => par des sous-programmes
32   SUBROUTINE hfrac2hms(t_hms, h) ! t_hms = h
33   ! conversion des heures fractionnaires en heures, minutes, secondes
34      TYPE(hms), INTENT(out) :: t_hms
35      REAL, INTENT(in) :: h ! attention: tous les réels sont des heures !
36      REAL :: tmp
37      t_hms%heures = INT(h) ! et non floor (cas des négatifs)
38      tmp = 60 * (h - t_hms%heures)
39      t_hms%minutes = INT(tmp)
40      tmp = 60 * (tmp - t_hms%minutes)
41      t_hms%secondes = INT(tmp)
42   END SUBROUTINE hfrac2hms
43   SUBROUTINE hms2hfrac(h, t_hms) ! h = t_hms
44   ! conversion des heures, minutes, secondes en heures fractionnaires
45      REAL, INTENT(out) :: h
```

```
46      TYPE(hms), INTENT(in) :: t_hms
47      h = REAL(t_hms%heures)+REAL(t_hms%minutes)/60.+REAL(t_hms%secondes)/3600.
48    END SUBROUTINE hms2hfrac
49    SUBROUTINE hms2s(s, t_hms) ! s = t_hms
50    ! conversion des heures, minutes, secondes en secondes
51      INTEGER, INTENT(out):: s
52      TYPE(hms), INTENT(in) :: t_hms
53      s = t_hms%secondes + 60*(t_hms%minutes + 60*t_hms%heures)
54    END SUBROUTINE hms2s
55    SUBROUTINE s2hms(t_hms, s)
56    ! conversion des secondes en heures, minutes, secondes
57       TYPE(hms), INTENT(out) :: t_hms
58       INTEGER, INTENT(in):: s ! attention: tous les entiers sont des secondes !
59       INTEGER :: tmp
60       t_hms%heures = s/3600 ! division entière
61       tmp = s - 3600*t_hms%heures
62       t_hms%minutes = tmp/60 ! division entière
63       t_hms%secondes = tmp - 60*t_hms%minutes
64    END SUBROUTINE s2hms
65    ! surcharges des opérateurs => par des fonctions
66    FUNCTION som_hms(x, y)
67      TYPE(hms) :: som_hms
68      TYPE(hms), INTENT(in) :: x, y
69      INTEGER :: sx, sy, ss
70      sx = x  ! conversion en secondes par hms2s
71      sy = y  ! conversion en secondes par hms2s
72      ss = sx + sy
73      som_hms = ss  ! conversion en hms par s2hms
74    END FUNCTION som_hms
75    FUNCTION diff_hms(x,y) ! "-" dyadique
76      TYPE(hms) :: diff_hms
77      TYPE(hms), INTENT(in) :: x, y
78      INTEGER :: sx, sy, ss
79      sx = x  ! conversion en secondes par hms2s
80      sy = y  ! conversion en secondes par hms2s
81      ss = sx - sy
82      diff_hms = ss ! conversion en hms par s2hms
83    END FUNCTION diff_hms
84    FUNCTION opp_hms(x) ! "-" monadique
85      TYPE(hms) :: opp_hms
86      TYPE(hms), INTENT(in) :: x
87      INTEGER :: sx
88      sx = x  ! conversion en secondes par hms2s
89      opp_hms = -sx ! opposé en entier, puis conversion en hms par s2hms
90    END FUNCTION opp_hms
91    FUNCTION mult_hms(n, x)
92      TYPE(hms) :: mult_hms
93      INTEGER, INTENT(in) :: n
94      TYPE(hms), INTENT(in) :: x
95      INTEGER :: sx
96      sx = x  ! conversion en secondes par hms2s
97      mult_hms = n * sx  ! multiplication entière puis conversion en hms par s2hms
98    END FUNCTION mult_hms
99    FUNCTION sup_hms(x, y)
100     LOGICAL :: sup_hms
101     TYPE(hms), INTENT(in) :: x, y
```

```
102     INTEGER :: sx, sy
103     sx = x  ! conversion en secondes par hms2s
104     sy = y  ! conversion en secondes par hms2s
105     sup_hms = sx > sy
106   END FUNCTION sup_hms
107 END MODULE m_hms
108 !
109 PROGRAM t_hms
110 ! traitement des temps en heures, minutes, secondes
111 USE m_hms
112 IMPLICIT NONE
113 TYPE(hms) :: hms1, hms2, hms3, hms4, hms5
114 REAL :: h1, h2
115 INTEGER :: s1, s2
116 WRITE(*,*) 'entrer deux instants en heures minutes secondes'
117 READ(*,*) hms1, hms2
118 h1 = hms1  ! conversion implicite en fraction d'heures
119 h2 = hms2  ! conversion implicite en fraction d'heures
120 s1 = hms1  ! conversion implicite en secondes
121 s2 = hms2  ! conversion implicite en secondes
122 hms3 = hms1 + hms2 ! addition de deux types hms
123 hms4 = hms1 - hms2 ! soustraction de deux types hms
124 hms5 = - 2*hms1    ! opposé d'un multiple entier
125 WRITE(*,*) ' en heures minutes secondes'
126 WRITE(*,*) ' hms1 = ', hms1, ' hms2 = ', hms2
127 WRITE(*,*) ' en heures fractionnaires'
128 WRITE(*,*) ' h1 = ', h1, ' h2 = ', h2
129 WRITE(*,*) ' en secondes '
130 WRITE(*,*) ' s1 = ', s1, ' s2 = ', s2
131 WRITE(*,*) ' somme en h m s ', hms3
132 WRITE(*,*) ' différence en h m s ', hms4
133 WRITE(*,*) ' -2 fois h1 en h m s ', hms5
134 IF (hms1 > hms2) THEN ! comparaison
135    WRITE(*,*) 'hms1 > hms2'
136 ELSE
137    WRITE(*,*) 'hms1 <= hms2'
138 END IF
139 END PROGRAM t_hms
```

L'exemple précédent reste une ébauche très élémentaire. En particulier, la surcharge de l'affectation suppose que tous les entiers sont des secondes et tous les réels des heures décimales : il faudrait restreindre ces conversions à des types spécialisés à une seule composante stockant des temps, mais alors aussi définir les opérateurs numériques associés pour ces types.

Pour fiabiliser l'usage de type dérivé `hms`, il faudrait contraindre les composantes heures, minutes, secondes à posséder un signe commun, et limiter minutes et secondes à l'intervalle ouvert $]-60,+60[$. On pourrait par exemple interdire l'accès direct aux composantes (attribut `PRIVATE`, *cf.* 9.1, p. 104) du type `hms`, quitte à définir des procédures publiques de saisie et de communication des données `hms` précisément chargées de cet accès restreint.

Chapitre 11

Pointeurs

Historiquement, la notion de pointeur a été introduite dans d'autres langages, en particulier le C [1], pour manipuler les données via les adresses de la mémoire qu'elles occupent : dans cette vision, un pointeur est une variable typée ; le type permet de connaître la taille de la zone mémoire nécessaire et le mécanisme de codage et de décodage [2] pour stocker l'objet (variable, structure, ...) pointé [3] ; le pointeur est destiné à stocker des valeurs qui sont les adresses des objets pointés, mais ces adresses seules seraient inutilisables sans la connaissance du type de l'objet pointé.

Mais la notion de pointeur en fortran s'apparente plus à un descripteur comportant non seulement une adresse, mais aussi d'autres informations sur la nature et la structure de la cible que le pointeur permet de désigner, notamment afin d'en assurer une gestion dynamique. La cible peut être une donnée de type quelconque (scalaire, tableau, structure, voire pointeur), ou une procédure, mais doit avoir été déclarée comme cible potentielle. D'autre part, fortran ne permet pas d'accéder explicitement à l'adresse que stocke le pointeur. D'ailleurs, il n'existe en fortran ni opérateur d'adresse [4], ni d'opérateur d'indirection [5] et la cible est désignée par le pointeur lui-même.

Dans la norme fortran 95, les pointeurs étaient nécessaires pour gérer les tableaux dynamiques dans certains contextes en tant que résultats de fonctions, arguments formels de procédures ou composantes de types dérivés. La norme 2003 apporte une gestion plus performante des tableaux dynamiques sans faire appel explicitement aux pointeurs. On réservera donc les pointeurs à des usages plus spécifiques comme la manipulation de structures dynamiques telles que les arbres ou les listes chaînées, ou, dans le cadre de la programmation objet, l'incorporation de méthodes dans les types dérivés grâce aux pointeurs de procédures.

11.1 Pointeurs, cibles et association

Une variable est considérée comme un *pointeur* si on précise l'attribut `POINTER` lors de sa déclaration ; noter que cette déclaration ne réserve alors pas d'espace pour stocker les valeurs qu'elle est susceptible de pointer.

1. Le tableau annexe (*cf.* D.7, p. 164) résume approximativement les syntaxes employées dans les deux langages pour manipuler des pointeurs.
2. Le type ne renseigne pas seulement sur la taille, mais aussi sur le codage de la valeur : par exemple si, sur une machine 32 bits, 4 octets permettent de stocker un entier par défaut et aussi un réel simple précision, le motif binaire qui représente par exemple la valeur numérique 10 ne sera pas le même en entier et en réel.
3. C'est cette information qui permet en C de faire de l'arithmétique sur les pointeurs : incrémentation, différence...
4. Si `var` est une variable, son adresse est désignée en C par `&var`. D'ailleurs, dans le cadre de l'interopérabilité entre fortran et C (*cf.* 12.3, p. 132), il faut faire appel en fortran, à une fonction spécifique du module `ISO_C_BINDING`, `c_loc` pour accéder à l'adresse d'une variable au sens du C.
5. Si `ptr` est un pointeur, la cible pointée est désignée par `*ptr` en C alors qu'en fortran, `ptr` désigne la cible.

11.1.1 Association à une cible nommée

Un pointeur peut être *associé à une variable* de même type (et éventuellement sous-type) qualifiée de cible si celle-ci a été déclarée avec l'attribut `TARGET`[6]. L'instruction

```
ptr => trgt
```

associe le pointeur *ptr* à la cible *trgt*. Tant que cette association restera en vigueur, toute mention du pointeur *ptr* dans une expression fera en fait référence à la cible *trgt*.

Une variable pointeur peut aussi être associée à une autre variable pointeur de même type : dans ce dernier cas, par transitivité, l'association est faite avec la cible ultime, d'attribut `TARGET`, c'est-à-dire non pointeur.

```
1  PROGRAM pointeurs
2  IMPLICIT NONE
3  INTEGER, TARGET  :: i=1 , j=2  ! target obligatoire pour pointer vers i ou j
4  INTEGER, POINTER :: pi, pj, pk
5  ! association entre pointeur et cible : nécessite le même type
6  pi => i      ! pi pointe sur i
7  pj => j      ! pj pointe sur j
8  pk => i      ! pk pointe aussi sur i
9  WRITE(*, *) "associations : i, j = ", i, j, " pi, pj, pk = ", pi, pj, pk
10 ! affectation : conversions implicites possibles
11 pi = 3.5      ! affecte int(3.5)  à la variable i, pointée par pi
12 pj = pi + 10  ! ajoute 10 à la variable pointée (i) et place le résultat dans j
13 pk => j       ! maintenant pk pointe sur j et plus sur i
14 WRITE(*, *) "affectations : i, j = ", i, j, " pi, pj, pk = ", pi, pj, pk
15 ! association d'un pointeur à un autre (en fait sa cible)
16 pi => pj      ! pi pointe maintenant sur la cible de pj, soit j
17 WRITE(*, *) "association d'un pointeur à un autre : pi, pj, pk = ", pi, pj, pk
18 END PROGRAM pointeurs
```

affichera

```
associations : i, j =  1 2  pi, pj, pk =  1 2 1
affectations : i, j =  3 13  pi, pj, pk =  3 13 13
association d'un pointeur à un autre : pi, pj, pk =  13 13 13
```

En fortran 2008, il est possible d'associer un pointeur à une cible dès la déclaration du pointeur, à condition que la cible ait l'attribut `SAVE` et ne soit pas allouable. Cette initialisation de pointeur est aussi possible vers une cible constituant une partie constante d'une variable statique. ⇐ f2008

Fonctions concernant l'association des pointeurs

La fonction intrinsèque `ASSOCIATED` qui rend un booléen permet de s'informer sur l'état d'association d'un pointeur. Elle admet deux formes d'appel :

- `ASSOCIATED(<ptr>)` rend `.true.` si le pointeur *<ptr>* est associé ;
- `ASSOCIATED(<ptr>, <target>)` rend `.true.` si le pointeur *<ptr>* est associé à la cible *<target>*.

La déclaration d'un pointeur lui donne par défaut un statut d'association indéterminé. Il est ⇐ ♥

6. Cette restriction permet d'éviter de manipuler accidentellement des variables via des pointeurs d'une part et de laisser le compilateur optimiser plus efficacement les instructions manipulant des variables non accessibles via des pointeurs d'autre part. À l'inverse, le langage C n'impose pas cette précaution, mais la notion de pointeur restreint introduite en C99 a un objectif similaire.

donc recommandé de le désassocier explicitement dès la déclaration [7] par `<ptr> => NULL()` ou, en début de programme par l'instruction `NULLIFY(<ptr>)`.

```
REAL, POINTER :: ptr1 => NULL()
REAL, TARGET  :: r1, r2
WRITE(*, *) ASSOCIATED(ptr1)     ! affichera .false.
ptr1 => r1
WRITE(*, *) ASSOCIATED(ptr1)     ! affichera .true.
WRITE(*, *) ASSOCIATED(ptr1, r1) ! affichera .true.
WRITE(*, *) ASSOCIATED(ptr1, r2) ! affichera .false.
```

11.1.2 Association à une cible dynamique anonyme

Une autre façon d'associer un pointeur est de lui allouer une zone mémoire, par l'instruction `ALLOCATE` (*cf.* ligne 3 du programme suivant), qui, dans le cas d'un pointeur, réserve l'espace mémoire et associe le pointeur à la mémoire réservée. Comme cette zone mémoire n'est pas désignée par une variable cible, la cible dynamique est qualifiée d'anonyme. Elle n'est accessible qu'au travers des pointeurs dont elle est la cible.

L'association cesse dès que l'on libère la zone mémoire par une instruction `DEALLOCATE` (*cf.* ligne 8). L'instruction `DEALLOCATE(<ptr>)` appliquée à un pointeur a donc deux effets :

- elle libère la mémoire allouée pour la cible anonyme ;
- elle désassocie le pointeur `<ptr>`.

⚠⇒ Si on a, entre temps, associé d'autres pointeurs à cette même cible (*cf.* ligne 6), seul celui qui a permis la libération de mémoire est désassocié par l'instruction `DEALLOCATE` : les autres continuent de pointer vers une zone mémoire dont rien de garantit qu'elle ne soit occupée par d'autres données...

♥⇒ Il est donc fortement conseillé de désassocier (*cf.* ligne 12), *tous* les pointeurs qui se partagent cette cible immédiatement après la désallocation.

```
1  PROGRAM alloc_assoc_pointeurs
2  REAL, POINTER :: ptr => NULL(), ptr2 => NULL()
3  ALLOCATE(ptr) ! reservation de mémoire et association de ptr
4  WRITE(*, *) "ptr associé ? ", ASSOCIATED(ptr) ! affichera .true.
5  ptr = .25 ! affectation d'une valeur à la cible
6  ptr2 => ptr ! ptr2 pointe aussi vers la cible allouée
7  WRITE(*, *) "ptr2 associé à ptr ?", ASSOCIATED(ptr2, ptr) ! affichera .true.
8  DEALLOCATE(ptr) ! libére la mémoire et desassocie ptr
9  WRITE(*, *) "ptr associé ? ", ASSOCIATED(ptr)  ! affichera .false.
10 WRITE(*, *) "ptr2 associé ?", ASSOCIATED(ptr2) ! affichera .true.
11 WRITE(*, *) "ptr2 associé à ptr ?", ASSOCIATED(ptr2, ptr) ! affichera .false.
12 NULLIFY(ptr2) ! fortement conseillé
13 ! si ptr2 est désassocié, l'instruction suivante provoquerait une erreur
14 ! si ptr2 n'était pas désassocié, elle donnerait un résultat aléatoire
15 ! WRITE(*, *) "cible de ptr2 ", ptr2
16 END PROGRAM alloc_assoc_pointeurs
```

Mais si on désassocie *tous* les pointeurs associés à une cible anonyme allouée sans instruction ⚠⇒ `DEALLOCATE`, la zone mémoire allouée reste réservée, mais n'est définitivement plus accessible : on parle alors de fuite de mémoire (memory leak). Si cette fuite de mémoire se produit de façon répétitive dans une boucle par exemple, elle peut provoquer un dépassement de capacité de la mémoire. On veillera donc à éviter de telles pertes de mémoire dans les procédures.

7. L'option `-fpointer=null` du compilateur `g95` (*cf.* F.6.1, p. 175) permet aussi d'initialiser à `null` les pointeurs non initialisés dans le code, mais ce n'est qu'une prudence supplémentaire.

```
PROGRAM ALLOC_PTR
IMPLICIT NONE
REAL, POINTER :: ptr => NULL()
ALLOCATE(ptr) ! allocation du pointeur
WRITE(*, *) ASSOCIATED(ptr) ! affichera .true.
ptr = 2 ! utilisation de la mémoire allouée
NULLIFY(ptr) ! déassociation avant déallocation ! => memory leak
WRITE(*, *) ASSOCIATED(ptr) ! affichera .false.
END PROGRAM ALLOC_PTR
```

Lors de l'exécution, le programme précédent compilé avec `g95` affichera `T` (true) puis `F` (false) et signalera cette fuite de mémoire : `Remaining memory: 4 bytes allocated at line 4`

Noter qu'affecter une valeur à un pointeur non associé n'est pas autorisé, car rien ne spécifie alors la zone mémoire où stocker la valeur.

11.2 Pointeurs sur des tableaux

L'attribut pointeur peut être donné à une variable tableau à condition que seul son rang et non son profil (*cf.* 7.1.1, 80) soit spécifié.

```
INTEGER, DIMENSION(10), POINTEUR :: ptab ! est interdit
INTEGER, DIMENSION(:), POINTEUR  :: ptab ! est autorisé
```

Dans le cas de tableaux dont l'indice ne commence pas à 1, on distinguera les deux formes d'association :

```
INTEGER, DIMENSION(-2:2), TARGET :: tab
INTEGER, DIMENSION(:), POINTER   :: pt1, pt2
pt1 => tab    ! qui conserve l'indexation initiale
pt2 => tab(:) ! qui renumérote à partir de 1 car tab(:) est une section de tab
```

Plus généralement, il est enfin possible de pointer vers une section régulière de tableau, de rang inférieur ou égal à celui du tableau dont la section est issue.

```
PROGRAM section_ptr
INTEGER, DIMENSION(3,5), TARGET  :: mat
INTEGER, DIMENSION(:), POINTER   :: pt1 => NULL() ! rang 1
INTEGER, DIMENSION(:,:), POINTER :: pt2 => NULL() ! rang 2
INTEGER :: i, j
WRITE(*,*) "matrice initiale"
DO i=1, 3
   mat(i,:) = i* (/ (j, j=1, 5) /)
   WRITE(*,*) mat(i,:)
END DO
pt1 => mat(2,:)
pt2 => mat(1:3:2, 2:4)
WRITE(*,*) "deuxième ligne de mat"
WRITE(*,*) pt1
WRITE(*,*) "sous-matrice de mat (lignes 1 et 3 , colonnes 2 à 4)"
DO i=1, 2
   WRITE(*,*) pt2(i,:)
END DO
END PROGRAM section_ptr
```

affiche

```
matrice initiale
1 2 3 4 5
2 4 6 8 10
3 6 9 12 15
deuxième ligne de mat
2 4 6 8 10
sous-matrice de mat (lignes 1 et 3 , colonnes 2 à 4)
2 3 4
6 9 12
```

f2003 ⇒ Il y aura alors réindexation à partir de 1 de la section pointée sauf si on désigne l'indice de départ du pointeur [8].

```
pt2(1:, 2:) => mat(1:3:2, 2:4)
```

Exemple de fonction à valeur pointeur

Une fonction peut produire un résultat tableau avec l'attribut pointeur [9] : cela lui permet de rendre un tableau dont le profil dépend de calculs effectués dans la fonction. Dans ce cas, il ne faut pas oublier (après usage) de désallouer dans l'appelant le tableau alloué par la fonction.

```
1  MODULE m_fct_ptr
2  IMPLICIT NONE
3  CONTAINS
4  FUNCTION positifs(x) ! fonction à valeur pointeur sur un tableau de rang 1
5  INTEGER, DIMENSION(:), POINTER :: positifs
6  INTEGER, DIMENSION(:), INTENT(in) :: x
7  INTEGER :: p, ok, i, j
8  p = COUNT( x>0 ) ! comptage du nombre de positifs
9  ALLOCATE (positifs(p), stat=ok) ! allocation du pointeur
10 IF(ok /= 0) STOP 'erreur allocation'
11 j = 0
12 DO i=1, SIZE(x)
13   IF (x(i) <= 0 ) CYCLE
14      j = j +1
15      positifs(j) = x(i) ! affectation du pointeur
16 END DO
17 END FUNCTION positifs
18 END MODULE m_fct_ptr
19
20 PROGRAM ppal
21 USE m_fct_ptr
22 IMPLICIT NONE
23 INTEGER :: i
24 INTEGER, DIMENSION(10) :: t=(/(i, i=-5, 4) /)
25 INTEGER, DIMENSION(:), POINTER :: tp => NULL()
26 WRITE(*, *) "tableau initial ",  t(:)
27 tp => positifs(t) ! appel de la fonction qui alloue le pointeur
28 WRITE(*, *) size(tp), " éléments positifs :", tp(:)
29 DEALLOCATE(tp) ! pour libérer la mémoire
30 END PROGRAM ppal
```

8. Cette possibilité non implémentée sous `g95` ni sous `gfortran v.4.4.3`, est disponible avec `gfortran v.4.9.3`.

9. C'était la seule méthode en fortran 95 standard qui n'autorisait pas les fonctions à résultat tableau allouable. Mais ces restrictions n'existent plus en fortran 2003, ou grâce à certaines extensions propres des compilateurs, en particulier `-ftr15581` pour le compilateur `g95` (*cf.* F.6.1, p. 175).

qui affiche

```
tableau initial  -5 -4 -3 -2 -1 0 1 2 3 4
4  positifs : 1 2 3 4
```

Les pointeurs permettent de nombreuses autres applications dans le domaine des données dynamiques, notamment pour gérer les listes chaînées (*cf.* 11.5, p. 128).

11.3 Tableaux de types dérivés contenant des pointeurs

Lorsqu'un pointeur possède un attribut de dimension, son profil qui détermine la taille de la cible ne peut pas être fixé à la déclaration (on parle de profil différé). Il n'est donc pas possible de définir directement des tableaux de pointeurs. Mais il suffit d'encapsuler le pointeur dans un type dérivé pour pouvoir simuler un tableau de pointeurs avec un tableau d'éléments du type dérivé qui ne contient que le pointeur. Il est ainsi possible de représenter des assemblages « non-rectangulaires » d'éléments du même type, qui ne sont pas représentables sous forme de tableaux. Cela vaut par exemple pour les parties triangulaires inférieure et supérieure d'une matrice. ⇐⚠

```
1  PROGRAM lower_upper_alloc
2  IMPLICIT NONE
3  ! pas de tableau de pointeurs => les encapsuler dans un type dérivé
4  TYPE iptr
5    INTEGER, DIMENSION(:), POINTER :: pt => null()
6    ! pointeur de tableau 1D d'entiers
7  END TYPE iptr
8  TYPE(iptr), DIMENSION(:), POINTER :: lower=> null(), upper => null()
9  ! tableaux d'éléments de type iptr pour stocker les parties triang. inf et sup
10 INTEGER, ALLOCATABLE, DIMENSION(:,:) :: mat
11 INTEGER :: ligne, colonne, n
12 DO
13   WRITE(*,*) "entrer la dimension de la matrice carrée"
14   READ(*,* ) n
15   IF (n <=0) EXIT
16   ALLOCATE(mat(n,n))
17   DO ligne = 1, n
18      DO colonne = 1, n
19         mat(ligne, colonne) = 10*ligne + colonne
20      END DO
21   END DO
22   WRITE(*,*) "matrice complète"
23   DO ligne = 1, n
24      WRITE(*,*) mat(ligne, :)
25   END DO
26   ALLOCATE(lower(n)) ! allocation du tableau-pointeur de n lignes
27   ALLOCATE(upper(n)) ! allocation du tableau-pointeur de n lignes
28   DO ligne = 1, n
29      ALLOCATE(lower(ligne)%pt(ligne)) ! une allocation par ligne
30      ALLOCATE(upper(ligne)%pt(n-ligne +1)) ! une allocation par ligne
31      lower(ligne)%pt = mat(ligne, 1:ligne) ! affectation ligne à ligne
32      upper(ligne)%pt = mat(ligne, ligne:n) ! affectation ligne à ligne
33   END DO
34   DEALLOCATE(mat) ! libération de la matrice
35   WRITE(*,*) "partie triangulaire inférieure"
36   DO ligne = 1, n
37      WRITE(*,*) lower(ligne)%pt(:) ! impression de la partie triang. inf.
38   END DO
39   WRITE(*,*) "partie triangulaire supérieure (non alignée)"
```

```
40   DO ligne = 1, n
41      WRITE(*,*) upper(ligne)%pt(:) ! impression de la partie triang. sup.
42   END DO
43   DO ligne = 1, n
44      DEALLOCATE(lower(ligne)%pt, upper(ligne)%pt)
45   END DO
46   DEALLOCATE(lower, upper)
47 END DO
48 END PROGRAM lower_upper_alloc
```

```
 entrer la dimension de la matrice carrée
4
 matrice complète
 11 12 13 14
 21 22 23 24
 31 32 33 34
 41 42 43 44
 partie triangulaire inférieure
 11
 21 22
 31 32 33
 41 42 43 44
 partie triangulaire supérieure (non alignée)
 11 12 13 14
 22 23 24
 33 34
 44
```

Dans l'exemple précédent (programme `lower_upper_alloc`), les tableaux de pointeurs `lower` et `upper` sont alloués (ligne 26, p. 125), après le choix de la matrice `mat` ; puis, les lignes des parties triangulaires inférieure et supérieure sont allouées (ligne 29, p. 125), et enfin les éléments correspondants sont recopiés (ligne 31, p. 125). Il y a donc duplication de l'espace mémoire de la matrice. On peut alors libérer l'espace alloué pour la matrice (ligne 34, p. 125) et continuer à utiliser les parties inférieure et supérieure, `lower` et `upper`.

Mais on peut aussi, comme dans la version `lower_upper_ptr` qui suit, se contenter de désigner les éléments correspondants de la matrice comme cibles (ligne 29, p. 127) des pointeurs des tableaux `lower` et `upper` : dans ce cas, ils suivent les modifications (ligne 40, p. 127) des coefficients de la matrice. Il est alors prudent de les désassocier à la libération de la matrice (ligne 50, p. 127).

```
1  PROGRAM lower_upper_ptr
2  IMPLICIT NONE
3  ! pas de tableau de pointeurs => les encapsuler dans un type dérivé
4  TYPE iptr
5    INTEGER, DIMENSION(:), POINTER :: pt => null()
6    ! pointeur de tableau 1D d'entiers
7  END TYPE iptr
8  TYPE(iptr), DIMENSION(:), POINTER :: lower=> null(), upper => null()
9  ! tableaux d'éléments de type iptr pour stocker les parties triang. inf et sup
10 INTEGER, ALLOCATABLE, DIMENSION(:,:), TARGET :: mat
11 INTEGER :: ligne, colonne, n
12 DO
13   WRITE(*,*) "entrer la dimension de la matrice carrée"
14   READ(*,* ) n
15   IF (n <=0) EXIT
16   ALLOCATE(mat(n,n))
17   DO ligne = 1, n
18      DO colonne = 1, n
19         mat(ligne, colonne) = 10*ligne + colonne
```

```
      END DO
   END DO
   WRITE(*,*) "matrice complète"
   DO ligne = 1, n
      WRITE(*,*) mat(ligne, :)
   END DO
   ALLOCATE(lower(n)) ! allocation du tableau-pointeur de n lignes
   ALLOCATE(upper(n)) ! allocation du tableau-pointeur de n lignes
   DO ligne = 1, n    ! désignation de la cible des pointeurs
      lower(ligne)%pt => mat(ligne, 1:ligne) !
      upper(ligne)%pt => mat(ligne, ligne:n)
   END DO
   WRITE(*,*) "partie triangulaire inférieure"
   DO ligne = 1, n
      WRITE(*,*) lower(ligne)%pt(:) ! impression de la partie triang. inf.
   END DO
   WRITE(*,*) "partie triangulaire supérieure (non alignée)"
   DO ligne = 1, n
      WRITE(*,*) upper(ligne)%pt(:) ! impression de la partie triang. sup.
   END DO
   mat = - mat ! modification de la matrice
   WRITE(*,*) "mat change de signe"
   WRITE(*,*) "partie triangulaire inférieure"
   DO ligne = 1, n
      WRITE(*,*) lower(ligne)%pt(:) ! impression de la partie triang. inf.
   END DO
   WRITE(*,*) "partie triangulaire supérieure (non alignée)"
   DO ligne = 1, n
      WRITE(*,*) upper(ligne)%pt(:) ! impression de la partie triang. sup.
   END DO
   DEALLOCATE(mat) ! libération de la matrice après usage des parties sup/inf
   DO ligne = 1, n ! par prudence (ils pointent alors vers une zone désallouée)
      NULLIFY(lower(ligne)%pt, upper(ligne)%pt)
   END DO
   DEALLOCATE(lower, upper)
END DO
END PROGRAM lower_upper_ptr
```

```
 entrer la dimension de la matrice carrée
3
 matrice complète
 11 12 13
 21 22 23
 31 32 33
 partie triangulaire inférieure
 11
 21 22
 31 32 33
 partie triangulaire supérieure (non alignée)
 11 12 13
 22 23
 33
 mat change de signe
 partie triangulaire inférieure
 -11
 -21 -22
 -31 -32 -33
 partie triangulaire supérieure (non alignée)
 -11 -12 -13
 -22 -23
 -33
```

11.4 Pointeurs de procédures

Un pointeur de procédure est un pointeur destiné à être associé à une procédure. Il est généralement déclaré via la spécification `PROCEDURE` (*cf.* 6.5.4, p. 75) avec l'attribut `POINTER`. Comme pour les autres objets, le pointeur, une fois associé, désigne sa cible.

♠ 11.5 Manipulation de listes chaînées

Les composantes des types dérivés peuvent posséder l'attribut pointeur, ce qui permet la constitution de listes chaînées : les listes chaînées permettent de représenter des ensembles ordonnés d'éléments (appelés nœuds de la liste) dont non seulement le contenu, mais aussi le nombre peut évoluer en cours de l'exécution du programme. On peut en particulier insérer ou supprimer des éléments dans la liste tout en conservant l'ordre : l'allocation se fait donc élément par élément, contrairement à ce qui passe pour les tableaux où l'allocation, si elle peut être dynamique, reste globale. Mais, l'accès à un élément donné de la liste nécessite de parcourir tous les précédents alors qu'on peut accéder directement à un élément d'indice donné d'un tableau.

Les listes chaînées sont souvent associées à des procédures récursives (*cf.* 6.5.6, p. 77) qui parcourent la chaîne tout en déclenchant des actions. Si l'action est déclenchée avant de passer à l'élément suivant, elle porte sur les éléments pris dans l'ordre de la liste. Si elle est déclenchée après, elle agira sur les éléments pris dans l'ordre inverse.

Dans une liste simplement chaînée, chaque élément comporte un pointeur vers l'élément suivant sauf le pointeur du dernier élément de la liste qui est nul.

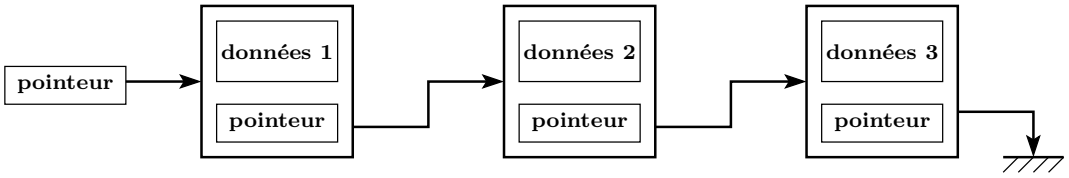


FIGURE 11.1 – Liste simplement chaînée à trois éléments : le dernier élément pointe vers `NULL`.

L'exemple suivant montre une liste chaînée permettant de représenter une courbe dans le plan : le type dérivé `point` comporte donc un pointeur vers une donnée de type `point` pour chaîner la liste. Pour simplifier, la courbe initialement créée est la deuxième bissectrice des axes. Les procédures d'impression `imprime_chaine` (ligne 10, p. 128) et de libération `libere_chaine` (ligne 26, p.129) de la liste sont ici implémentées sous forme récursive. Pour libérer la mémoire, il est nécessaire de commencer par la fin de la liste afin d'éviter de perdre l'accès aux éléments suivants. En revanche, l'impression de la liste se fait dans l'ordre si on imprime avant l'appel récursif (en « descendant » la récurrence) ou dans l'ordre inverse (procédure `imprime_chaine_inverse` (ligne 18, p. 129) si on imprime après (en « remontant » la récurrence). Enfin, les opérations d'insertion, `insere_point` (ligne 57, p. 129) ou de suppression, `supprime_point` (ligne 35, p. 129) d'un point intérieur se font dans une boucle qui parcourt l'ensemble de la liste jusqu'au pointeur nul signifiant l'accès au dernier point.

```
1  MODULE m_chaine
2  IMPLICIT NONE
3  TYPE point
4  ! définition du type dérivé point avec un pointeur vers le suivant
5    REAL :: x
6    REAL :: y
7    TYPE(point), POINTER :: next => null() ! f95 seulement (par prudence)
8  END TYPE point
9  CONTAINS
10   RECURSIVE SUBROUTINE imprime_chaine(courant) !
11      TYPE(point), POINTER :: courant
12      ! affichage en commençant par le début : imprimer avant la récursion
```

```
13      WRITE(*,*)  courant%x, courant%y
14      IF(ASSOCIATED(courant%next)) THEN
15        CALL imprime_chaine(courant%next)
16      END IF
17    END SUBROUTINE imprime_chaine
18    RECURSIVE SUBROUTINE imprime_chaine_inverse(courant) !
19      TYPE(point), POINTER :: courant
20      ! affichage en commençant par la fin : dérouler la récursion avant
21      IF(ASSOCIATED(courant%next)) THEN
22        CALL imprime_chaine_inverse(courant%next)
23      END IF
24      WRITE(*,*)  courant%x, courant%y
25    END SUBROUTINE imprime_chaine_inverse
26    RECURSIVE SUBROUTINE libere_chaine(courant)!
27      TYPE(point), POINTER :: courant
28      ! attention libérer en commençant par la fin
29      IF(ASSOCIATED(courant%next)) THEN
30        CALL libere_chaine(courant%next)
31      END IF
32      WRITE(*,*) "libération du point de coordonnées", courant%x, courant%y
33      DEALLOCATE(courant)
34    END SUBROUTINE libere_chaine
35    SUBROUTINE supprime_point(debut) !
36      ! suppression d'un point (intérieur seulement)
37      TYPE(point), POINTER :: debut
38      TYPE(point), POINTER :: courant=>null(), suivant=>null()
39      TYPE(point), POINTER :: precedent=>null()
40      INTEGER :: suppr
41      precedent => debut
42      courant => precedent%next
43      DO
44        suivant => courant%next
45        WRITE(*,*) courant%x, courant%y, "supprimer ? 1 si oui"
46        READ(*,*) suppr
47        IF (suppr == 1) THEN
48          precedent%next => courant%next ! court-circuiter le point courant
49          DEALLOCATE(courant) ! libération du point courant
50        ELSE
51          precedent => courant
52        ENDIF
53        courant => suivant
54        IF (.NOT.ASSOCIATED(suivant%next)) EXIT
55      END DO
56    END SUBROUTINE  supprime_point
57    SUBROUTINE insere_point(debut) !
58      ! ajout d'un point (intérieur seulement)
59      TYPE(point), POINTER :: debut
60      TYPE(point), POINTER :: nouveau=>null(), suivant=>null()
61      TYPE(point), POINTER :: precedent=>null()
62      INTEGER :: ajout
63      REAL :: x, y
64      precedent => debut
65      DO
66        suivant => precedent%next
67        WRITE(*,*) precedent%x, precedent%y, "insérer ? 1 si oui"
68        READ(*,*) ajout
69        IF(ajout == 1) THEN
70          ALLOCATE(nouveau)
71          nouveau%next => precedent%next
72          precedent%next => nouveau
73          WRITE(*,*) "entrer x et y"
```

```
74            READ(*,*) x, y
75            nouveau%x = x
76            nouveau%y = y
77          ENDIF
78          precedent => suivant
79          IF (.NOT.ASSOCIATED(suivant%next)) EXIT
80        END DO
81    END SUBROUTINE insere_point
82  END MODULE m_chaine
```

```
84  PROGRAM linked_list
85  USE m_chaine
86  INTEGER :: i, n
87  TYPE(point), POINTER :: courant => null(), suivant=>null()
88  TYPE(point), POINTER :: debut => null()
89  ! construction d'une liste chaînée de n points
90  WRITE(*,*) 'entrer le nb de points'
91  READ(*,* ) n
92  ! premier point qui permettra l'accès à toute la chaîne
93  ALLOCATE(debut)
94  courant => debut
95  DO i = 1, n-1
96    courant%x = REAL(i)
97    courant%y = -REAL(i)
98    ALLOCATE(suivant)
99    courant%next => suivant
100   courant => suivant
101 END DO
102 ! dernier point sans successeur
103 courant%x = REAL(n)
104 courant%y = -REAL(n)
105 courant%next => null() ! fin de la liste
106 WRITE(*,*) "impression dans le sens direct"
107 CALL imprime_chaine(debut) ! impression dans le sens direct
108 WRITE(*,*) "impression dans le sens inverse"
109 CALL imprime_chaine_inverse(debut) ! impression en sens inverse
110 WRITE(*,*) "suppression de points"
111 CALL supprime_point(debut) ! suppression de points intérieurs
112 WRITE(*,*) "impression dans le sens direct"
113 CALL imprime_chaine(debut) ! impression dans le sens direct
114 WRITE(*,*) "insertion de points"
115 CALL insere_point(debut)   ! insertion de points intérieurs
116 WRITE(*,*) "impression dans le sens direct"
117 CALL imprime_chaine(debut) ! impression dans le sens direct
118 WRITE(*,*) "libération de la chaîne"
119 CALL libere_chaine(debut)  ! libération de la chaîne
120 END PROGRAM linked_list
```

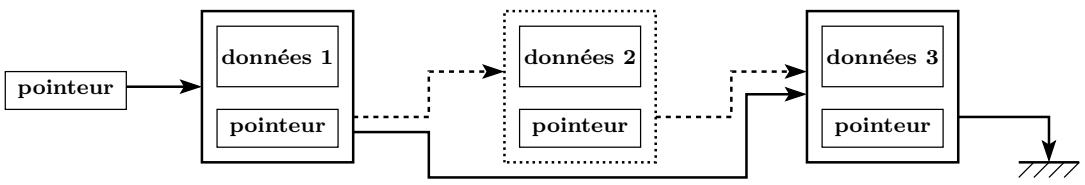


FIGURE 11.2 – Suppression d'un élément intérieur d'une liste simplement chaînée à trois éléments

Chapitre 12

Interopérabilité avec le C

La standardisation de l'interfaçage entre les langages C et fortran est intervenue seulement avec la norme fortran 2003. Elle permet de rendre portables les appels de fonctions en C depuis le fortran et réciproquement. Elle s'appuie sur la définition en fortran d'entités (types, variables et procédures) dites interopérables car équivalentes à celles du langage C.

La construction d'un exécutable à partir de code C et de code fortran nécessite une compilation séparée avec des compilateurs C et fortran compatibles, puis une édition de lien qui fournisse toutes les bibliothèques nécessaires. En particulier dans le cas où le programme principal est en C, il faut spécifier explicitement la bibliothèque du fortran lors de l'édition de liens si elle est lancée par le compilateur C [1]. Une autre solution consiste à confier l'édition de liens au compilateur fortran associé, quitte à lui indiquer par une option [2] que le programme principal n'est pas en fortran.

12.1 Interopérabilité des types intrinsèques

Le module intrinsèque `iso_c_binding` définit les paramètres de codage (kind) sous forme de constantes symboliques pour les sous-types intrinsèques interopérables avec le langage C. Le tableau suivant décrit les équivalences entre les principaux types interopérables du fortran 2003 et ceux de la norme C89 du langage C :

C89	fortran 2005
`char`	`CHARACTER(kind=c_char)`
`short_int`	`INTEGER(kind=c_short)`
`int`	`INTEGER(kind=c_int)`
`long int`	`INTEGER(kind=c_long)`
`long long int`	`INTEGER(kind=c_lon_long)`
`size_t`	`INTEGER(kind=c_size_t)`
`float`	`REAL(kind=c_float)`
`double`	`REAL(kind=c_double)`
`long double`	`REAL(kind=c_long_double)`

Ce module définit de même des types complexes et un type booléen interopérables respectivement avec les types complexes et le type booléen du C99.

C99	fortran 2005
`float _Complex`	`COMPLEX(kind=c_float_complex)`
`double _Complex`	`COMPLEX(kind=c_double_complex)`
`long double _Complex`	`COMPLEX(kind=c_long_double_complex)`
`_bool`	`LOGICAL(kind=c_bool)`

1. Avec le compilateur `g95` sous linux et un processeur Intel 32 bits, on lancera par exemple, sous réserve que la bibliothèque fortran soit installée dans ce répertoire,
`gcc partie_c.o partie_fortran.o -L/usr/lib/g95/lib/gcc-lib/i686-pc-linux-gnu/4.0.3/ -lf95` suivi éventuellement de `-lm`
De plus on s'assurera de la compatibilité des versions des compilateurs utilisés.

2. Par exemple `gfortran` ou `g95` avec `gcc`, ou `ifort` et l'option `-nofor-main` (*cf.* F.4.1, p. 171) avec `icc`.

Il définit aussi des types interopérables équivalents aux variantes de type entier définies en C99, `int8_t`, `int16_t`, `int32_t` et `int64_t` à nombre de bits fixé ainsi que les versions à nombre minimal de bits `int_least8_t` et suivants ou les plus rapides avec un nombre de bits fixé `int_fast8_t` et suivants : le nommage en fortran est obtenu par adjonction du préfixe `c_`, par exemple `c_int8_t`.

Le module `iso_c_binding` définit enfin des constantes caractères de paramètre de codage (`kind`) `c_char` pour les caractères spéciaux du C, comme `c_null_char` ou `c_new_line`.

f2008 ⇒ Le fortran 2008 introduit la fonction intrinsèque `c_sizeof` qui s'applique aux objets interopérables et renvoie (comme l'opérateur `sizeof` du langage C) la taille en octets de son argument.

12.2 Interopérabilité des types dérivés

Les types dérivés du fortran peuvent être rendus interopérables avec les structures du C à condition que [3] :

- les composantes du type dérivé fortran soient interopérables, publiques, qu'elles ne soient pas allouables et ne comportent pas de pointeur (au sens de l'attribut `pointer` en fortran [4]) ;
- Le type dérivé fortran ne soit pas une extension de type (*cf.* 9.3.2, p. 107), ne soit pas de type `SEQUENCE` (*cf.* 9.1.1, p. 105), ni un type paramétré et ne possède pas de procédures attachées ;
- l'attribut `bind(c)` soit spécifié en fortran dans la définition du type dérivé ;
- la structure du C ne comporte pas de champ de bits.

12.3 Interopérabilité avec les pointeurs du C

Le module `iso_c_binding` définit des types dérivés (à composantes privées) `c_ptr` et `c_funptr` interopérables respectivement avec les pointeurs d'objets et de fonctions du C. Les constantes nommées `c_null_ptr` et `c_null_funptr` sont les équivalents en fortran du pointeur nul du C.

Ce module définit aussi des procédures de manipulation des pointeurs du C depuis le fortran.

- `c_loc(x)` renvoie une valeur du type `c_ptr` contenant l'adresse au sens du C de son argument. Cette fonction joue le rôle de l'opérateur `&` du langage C.
- `c_funloc(f)` renvoie l'adresse d'une procédure interopérable.
- `c_f_pointer(cptr, fptr, shape)` est un sous-programme qui permet de traduire un pointeur du C et un profil éventuel en un pointeur au sens du fortran. L'argument d'entrée `cptr` est du type `c_ptr`. L'argument de sortie `fptr` est un pointeur vers un type interopérable. En sortie, `fptr` est associé à la cible de `cptr`. L'argument d'entrée `shape` en général optionnel, est requis si `fptr` est un tableau.
- `c_f_procpointer(cptr, fptr)` est un sous-programme qui effectue la même conversion que `c_f_pointer` mais pour des procédures.

12.4 Interopérabilité des procédures

L'interopérabilité des procédures entre fortran et C requiert que les paramètres échangés soient interopérables, que l'interface de la procédure fortran soit explicite et déclarée avec l'attribut `BIND(C)`. L'attribut `BIND` permet aussi de spécifier le nom de la fonction C associée grâce au mot clef `NAME` selon la syntaxe : `BIND(C, NAME=`*`nom_de_la_fonction_en_C`*`)`. À défaut du paramètre `NAME` dans l'attribut `BIND`, le nom en C est le même qu'en fortran, mais en minuscule [5].

Elle exclut notamment :

- les procédures internes du fortran ;
- les arguments optionnels en fortran et le nombre variable d'arguments en C ;

3. L'espace occupé par les structures est soumis à des contraintes d'alignement en mémoire qui conduisent parfois à compléter une structure par des octets de remplissage. Ces contraintes diffèrent suivant les langages, les processeurs et les compilateurs et peuvent ainsi affecter l'interopérabilité des structures.
4. Mais un type dérivé interopérable peut comporter une composante de type `c_ptr` ou `c_funptr`.
5. On rappelle qu'en fortran, la casse n'est pas distinguée, *cf* 1.3.2, p. 6.

– les résultats de fonction non scalaires en fortran .

Les sous-programmes du fortran correspondent aux fonctions C sans valeur de retour (type `void`).

12.4.1 Le passage par copie de valeur (`value`)

Comme le passage de paramètre en C se fait par copie de valeur, alors qu'il se fait par référence en fortran, le fortran 2003 a introduit l'attribut `VALUE` pour modifier le mécanisme de passage d'argument du fortran dans le sens du C. Cette méthode permet de travailler sur la copie passée en argument sans que cela affecte la valeur côté appelant en retour. L'attribut `VALUE` peut être utilisé en dehors de l'interopérabilité avec le C mais n'est pas applicable aux paramètres pointeurs ou allouables et est évidemment incompatible avec les vocations `INTENT(OUT)` et `INTENT(INOUT)`.

À l'inverse, un argument sans l'attribut `VALUE` doit correspondre dans la fonction en C à un pointeur vers le type de l'argument (interopérable) du fortran.

12.4.2 Exemple : appel de fonctions C depuis le fortran

L'exemple suivant montre l'appel en fortran de fonctions C sans valeur de retour pour saisir des réels et les afficher. Elles sont vues du fortran comme des sous-programmes de même nom en l'absence du paramètre `NAME` dans l'attribut `BIND` (*cf.* lignes 6 et 12). L'accès au type `c_float` dans l'interface fortran (*cf.* lignes 8 et 14) nécessite soit un `USE`, soit un `IMPORT` (*cf.* 6.4.4, p. 70).

Le sous-programme d'affichage `c_write_float` peut se contenter d'un passage par copie de valeur, d'où l'attribut `VALUE` en fortran (*cf.* ligne 15), car il ne modifie pas la valeur du paramètre. En revanche, la fonction de lecture, `c_read_float`, nécessite un passage par pointeur en C, directement compatible avec le passage d'argument utilisé par défaut en fortran.

Partie fortran 2003

```
1  MODULE m_interf
2    USE, INTRINSIC :: iso_c_binding, ONLY:c_float
3    IMPLICIT NONE
4    INTERFACE
5      ! void c_read_float(float *pf);
6      SUBROUTINE c_read_float(f) BIND(c) !
7        ! USE, INTRINSIC :: iso_c_binding, ONLY:c_float
8        IMPORT :: c_float ! préférable au USE !
9        REAL(kind=c_float), INTENT(out) :: f
10     END SUBROUTINE c_read_float
11     ! void c_write_float(float f);
12     SUBROUTINE c_write_float(f) BIND(c) !
13       ! USE, INTRINSIC :: iso_c_binding, ONLY:c_float
14       IMPORT :: c_float ! préférable au USE !
15       REAL(kind=c_float), intent(in), value :: f ! copie de valeur
16     END SUBROUTINE c_write_float
17   END INTERFACE
18 END MODULE m_interf
19
20 PROGRAM f_appel_c_sub_io
21 USE m_interf
22 REAL(kind=c_float) :: r
23 CALL c_read_float(r)
24 WRITE (*,*) "float en fortran après lecture en C ", r
25 CALL c_write_float(r)
26 END PROGRAM f_appel_c_sub_io
```

Partie langage C

```
1  #include <stdio.h>
2  #include <stdlib.h>
```

```
3
4  void c_read_float(float *pf);
5  void c_write_float(float f);
6
7  void c_read_float(float *pf){
8    /*  *pf est saisi en C */
9    printf("saisie d'un float en C\n");
10   scanf("%g", pf);
11   printf("float tel que saisi en C %g\n", *pf);
12   return;
13 }
14
15 void c_write_float(float f){
16   printf("valeur affichée en C %g\n", f);
17   return;
18 }
```

12.4.3 Exemple : appel de sous-programmes fortran depuis le C

L'exemple suivant montre l'appel en C de sous-programmes fortran pour saisir des réels et les afficher. Ils sont vus depuis le C comme des fonctions sans valeur de retour dont le nom est donné par le paramètre `NAME` dans le `BIND` (*cf.* lignes 6 et 13). Le sous-programme d'affichage `write_float` peut se contenter d'un passage par copie de valeur, d'où l'attribut `VALUE` en fortran (*cf.* ligne 14), car il ne modifie pas la valeur du paramètre. En revanche, la fonction de lecture, `f_read_float`, nécessite un passage par pointeur en C, directement compatible avec le passage d'argument utilisé par le fortran.

Partie langage C

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void f_read_float(float *px);
5  void f_write_float(float x);
6
7  int main(void){
8    float f;
9    /* f est saisi en fortran */
10   f_read_float(&f);
11   printf("valeur du float affichée en C %g\n", f);
12   f_write_float(f);
13   exit(EXIT_SUCCESS);
14 }
```

Partie fortran 2003

```
1  MODULE m_io
2    USE, INTRINSIC :: iso_c_binding, ONLY:c_float
3    IMPLICIT NONE
4    CONTAINS
5      ! void f_read_float(float *px);
6      SUBROUTINE read_float(x) BIND(c, name="f_read_float") !
7        REAL(kind=c_float), INTENT(out) :: x
8        WRITE (*, *) "saisie en fortran: entrer un float"
9        READ (*,*) x
10       WRITE (*,*) "float lu en fortran avant passage en C", x
11     END SUBROUTINE read_float
```

```
12      ! void f_write_float(float x);
13      SUBROUTINE write_float(x) BIND(c, name="f_write_float") !
14        REAL(kind=c_float), INTENT(in), value :: x ! passage par copie
15        WRITE (*,*) "affichage en fortran après passage en C "
16        WRITE (*,*) x
17      END SUBROUTINE write_float
18  END MODULE m_io
```

12.5 Interopérabilité des tableaux

Les tableaux ne peuvent être interopérables que si leurs éléments sont de type et de paramètres interopérables. De plus, ils doivent soit être de profil explicite, soit ne comporter qu'une dimension indéterminée :

- la dernière spécifiée par `*` en fortran ;
- la première spécifiée par `[]` en C.

Dans le cas d'un tableau de rang 1, si la taille est explicite, elle doit être précisée et identique dans les deux langages. Si elle est indéterminée en fortran, elle ne doit pas être précisée en C.

L'interopérabilité des tableaux de rang supérieur à 1 s'appuie sur un ordre de rangement en mémoire qui veut qu'en fortran, ce soit le premier indice qui défile le plus vite [6] alors qu'en C, où il s'agit alors de tableaux de tableaux, c'est le dernier indice qui défile le plus vite. Il faudra donc systématiquement échanger l'ordre des tailles pour passer d'un langage à l'autre.

Par exemple le tableau fortran déclaré : `INTEGER(kind=c_int) :: tab(3,5,6)`
pourra être inter-opérable avec le tableau C : `int tab[6][5][3];`
S'il reste une dimension indéterminée, ce sera la dernière en fortran, `tab(3,5,*)` et la première en C, `tab[][5][3]`.

12.5.1 Exemple : fonctions C manipulant des tableaux définis en fortran

Les tableaux sont ici de rang deux et de taille fixe pour le compilateur dans le programme principal en fortran afin d'éviter l'allocation dynamique. Mais les fonctions en C supposent seulement qu'une étendue est connue : celle dont la connaissance permet de calculer les décalages à effectuer dans les adresses mémoire pour accéder aux éléments du tableau, c'est-à-dire celle de droite en C, notée `C_N2` (*cf.* ligne 8, partie C), qui correspond à celle de gauche en fortran, notée `F_LIG` (*cf.* ligne 12, partie fortran). Cette dimension commune, fixée ici grâce au préprocesseur, devrait l'être sous un seul nom dans un fichier unique inclus par le source C et par le source fortran. Avec les conventions de nommage classiques des compilateurs (*cf.* annexe F), il est nécessaire de nommer le source fortran avec comme suffixe `.F90` au lieu de `.f90` pour le voir traité par le préprocesseur.

Partie fortran 2003

```
1   ! toutes les étendues du tableau transmis sont fixes
2   ! sauf la plus à droite en fortran (qui varie le plus lentement)
3   ! donc déclarées dans l'interface, sauf la dernière  *
4   ! attention:  stockage inversé en C
5
6   ! instruction pré-processeur à répéter en C
7   ! ou mieux : inclure un fichier commun dans les deux codes
8
9   ! pour imposer le traitement par le pré-processeur
10  ! choisir un fichier de suffixe .F90 au lieu de .f90 pour le source
11
12  #define F_LIG 3 !
13
14  MODULE m_interf
15    USE, INTRINSIC :: iso_c_binding, ONLY:c_int
```

6. Les éléments qui ne diffèrent que d'une unité de cet indice sont supposés être contigus en mémoire.

```
  IMPLICIT NONE
  INTERFACE
    SUBROUTINE c_print_tab2d(mat, f_col) BIND(c)
      IMPORT ::c_int !
      INTEGER(kind=c_int), INTENT(in) :: mat(F_LIG, *)
      INTEGER(kind=c_int), value :: f_col
    END SUBROUTINE c_print_tab2d
    SUBROUTINE c_mult_tab2d(mat, f_col) BIND(c)
      IMPORT ::c_int !
      INTEGER(kind=c_int), INTENT(inout) :: mat(F_LIG, *)
      INTEGER(kind=c_int), value :: f_col
    END SUBROUTINE c_mult_tab2d
  END INTERFACE
END MODULE m_interf

PROGRAM f_appel_sub_c_tab2d
USE m_interf
IMPLICIT NONE
! f_col est connu du compilateur en fortran
! mais est un paramètre pour la fonction définie en C
INTEGER(kind=c_int), PARAMETER :: f_col = 4, f_col2 = 2
INTEGER(kind=c_int) :: mat(F_LIG, f_col), mat2(F_LIG, f_col2)
INTEGER :: i, j
! remplissage des tableaux
DO i = 1, F_LIG
  DO j = 1, f_col
    mat(i,j) = 10 * i + j
  END DO
END DO
mat2(:,:) = - mat(:,1:f_col2)
! affichage en fortran
WRITE (*,*) "tableau 2d affiché en fortran: 1er indice = ligne"
DO i = 1, F_LIG
   WRITE(*,*) mat(i,:)
END DO
WRITE (*,*) "2ème tableau 2d affiché en fortran: 1er indice = ligne"
DO i = 1, F_LIG
   WRITE(*,*) mat2(i,:)
END DO
! appel pour l'affichage en C
CALL c_print_tab2d(mat, f_col)
WRITE(*,*) "deuxième tableau"
CALL c_print_tab2d(mat2, f_col2)
! on  multiplie chaque ligne par son numéro
WRITE (*,*) "tableau 2d multiplié par l'indice de ligne fortran"
CALL c_mult_tab2d(mat, f_col)
! appel pour l'affichage en C
CALL c_print_tab2d(mat, f_col)
END PROGRAM f_appel_sub_c_tab2d
```

Partie langage C

```
#include <stdio.h>
#include <stdlib.h>

/* doit être cohérent avec le 1er indice du tableau fortran */
```

```
5  /* C_N2 = F_LIG  ce qui pourrait être assuré par un fichier */
6  /* commun inclus dans les deux sources fortran et C */
7
8  #define C_N2 3
9
10 /* transposition des dimensions entre fortran et C */
11 /* en C toutes les dimensions sauf  la première doivent être connues */
12 /* du compilateur => fixées ici par le pré-processeur */
13 /* en fortran mat(F_LIG, *) = en C mat[][F_LIG] */
14 /* et passe la dimension variable en argument  */
15 /* mat(F_LIG, f_col) = mat[c_n1][C_N2]  avec c_n1=f_col C_N2=f_LIG */
16
17 /* integer(kind=c_int), intent(inout) :: mat(F_LIG, *) */
18 void c_print_tab2d(int mat[][C_N2], const int c_n1);
19 void c_mult_tab2d(int mat[][C_N2], const int c_n1);
20
21 void c_print_tab2d(int mat[][C_N2], const int c_n1){
22   int i, j;
23   printf("affichage en C\n");
24   for (i=0; i<C_N2; i++){   /* 2e indice C = 1er fortran */
25     for (j=0; j<c_n1; j++){ /* 1er indice C = 2e fortran */
26       printf("%d ", mat[j][i]);
27     }
28     printf("\n");
29   }
30   return;
31 }
32
33 void c_mult_tab2d(int mat[][C_N2], const int c_n1){
34   int i, j;
35   for (i=0; i<C_N2; i++){   /* 2e indice C = 1er fortran */
36     for (j=0; j<c_n1; j++){ /* 1er indice C = 2e fortran */
37       mat[j][i] *= (i+1); /* mult par 1er indice fortran */
38     }
39   }
40   return;
41 }
```

12.5.2 Exemple : fortran manipulant des tableaux dynamiques alloués en C

On peut utiliser les fonctions avancées du fortran portant sur les tableaux à partir du langage C, y compris dans le cas de tableaux dynamiques alloués et désalloués en C. Dans le cas de tableaux de rang 1, on définit en C une structure `vecteur` (lignes 11 à 14) comportant un pointeur C sur le début du tableau (de float ici) et un entier spécifiant le nombre de ses éléments. On définit un type dérivé `vecteur` interopérable en fortran (module `m_vecteur`) constitué des mêmes composantes en utilisant les types interopérables `c_ptr` et `c_int` associés en fortran.

Une fois l'argument `vecteur` passé au fortran par copie [7] (ligne 8), il s'agit de lui associer un tableau de rang 1 de la taille précisée et dont les éléments sont ceux pointés par le pointeur C. Les tableaux allouables ou les tableaux à attribut pointeur du fortran sont plus riches que les pointeurs du C, car ils comportent les informations sur le profil du tableau.

On utilise donc le sous-programme `c_f_pointer` (ligne 16) pour effectuer le transfert des informations de la structure C vers le pointeur fortran. À ce propos, noter que la conversion du

7. Avec le passage par valeur, on s'interdit de modifier le pointeur C et la taille du tableau, donc de libérer ou de réallouer le tableau en fortran. Bien sûr les valeurs des éléments du tableau sont modifiables en fortran.

⚠⇒ paramètre de taille de `INTEGER(kind=c_int)` en entier par défaut (ligne 13) est nécessaire pour spécifier le profil du tableau de façon portable en fortran. Une fois le tableau connu du fortran, on peut lui appliquer toutes les transformations possibles avec les fonctions-tableau du fortran, tant que l'on ne modifie pas sa taille.

Définition du type dérivé en fortran 2003

```
1  MODULE m_vect
2  USE, INTRINSIC :: iso_c_binding, ONLY : c_int, c_ptr
3  IMPLICIT NONE
4  ! définition du type dérivé vecteur tel que vu en C
5  ! ses composantes sont des types fortran inter-opérables C
6  TYPE, bind(c) :: vecteur
7    INTEGER(c_int) :: n  ! taille du tableau
8    TYPE(c_ptr) :: ptc_v ! pointeur C
9  END TYPE vecteur
10 END MODULE m_vect
```

Sous-programme fortran 2003

```
1  ! les vecteurs sont alloués dynamiquement et libérés par l'appelant (en C)
2  MODULE m_modif_vect
3  USE m_vect
4  IMPLICIT NONE
5  CONTAINS
6    SUBROUTINE f_modif_vect(v1) bind(c, name="f_modif_vect")
7    USE, INTRINSIC :: iso_c_binding, ONLY : c_f_pointer,  c_float, c_null_ptr
8      TYPE(vecteur), INTENT(in), value :: v1 ! passage de la structure par copie
9      REAL(c_float), DIMENSION(:), POINTER :: ptf_v ! pointeur fortran
10     ! affectation du pointeur de tableau fortran
11     ! avec l'adresse au sens pointeur C et le profil
12     INTEGER, DIMENSION(1) :: loc_v1_n ! vecteur profil fortran du tableau
13     loc_v1_n (1)= v1%n ! conversion du type integer(c_int)
14     ! vers le type entier par défaut (32 ou 64 bits)
15     ! pour spécifier le profil du tableau en fortran
16     CALL c_f_pointer(cptr=v1%ptc_v, fptr=ptf_v, shape=loc_v1_n)
17     WRITE(*,*) "on double les valeurs du tableau"
18     ptf_v(:) = 2* ptf_v(:)
19     WRITE(*,*) ptf_v(:)
20     WRITE(*,*) "décalage circulaire de 2 cases dans le tableau"
21     ptf_v(:) = CSHIFT(ptf_v(:) , shift=-2)
22     WRITE(*,*) "résultat en fortran"
23     WRITE(*,*) ptf_v(:)
24   END SUBROUTINE f_modif_vect
25 END MODULE m_modif_vect
```

Programme principal en langage C

```
1  /* version avec passage du pointeur vers a structure
2     => permet de désallouer et réallouer en fortran */
3  /* version avec allocation dynamique et type dérivé */
4  /* attention compiler avec gcc -lm pour inclure libm.a */
5  /*                  et inclure l'entête associé math.h */
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <math.h>
9
10 /* type dérivé vecteur */
```

```
11 typedef struct {
12   int n ;
13   float *v;
14 } vecteur;
15
16 void f_modif_vect(vecteur v) ; /* déclaration de la procédure en fortran */
17 /* passage d'un pointeur vers la structure C => elle est modifiable */
18 /* attention : aucune vérification avec l'interface fortran bien sûr !  */
19
20 int main(void) {
21   vecteur v1 = {0, NULL};
22   int i;
23   /* Lecture du nombre de dimensions */
24   while(
25   printf("Entrez le nombre de composantes : (0 pour terminer) \n"),
26   scanf("%d", &(v1.n)),
27   v1.n > 0){
28   /* Allocation mémoire en C */
29   v1.v = (float *) calloc((size_t) v1.n, sizeof (float));
30   /*                size_t: conversion nécessaire en 64 bits  */
31   /* Affectation des composantes */
32   for (i=0; i<v1.n; i++) {
33     v1.v[i] = (float) (i+1) ;
34   }
35   /* Affichage des composantes */
36   printf("affichage des %d composantes\n", v1.n);
37   for (i=0; i<v1.n; i++) {
38     printf("%f ", v1.v[i]);
39   }
40   printf("\n");
41   /* appel de la subroutine fortran */
42   printf("appel fortran \n"),
43   f_modif_vect(v1); /* pointeur vers la structure de vecteur*/
44   /*  Impression du résultat  */
45   printf("affichage des composantes après appel fortran\n");
46   for (i=0; i<v1.n; i++) {
47     printf("%f ", v1.v[i]);
48   }
49   printf("\n");
50   /* Libération de la memoire dans le même langage */
51   /* on suppose que libération <=> v1.n =0 */
52   if (v1.n > 0  || v1.v != NULL ) {
53     free(v1.v);
54     v1.n = 0;
55     v1.v = NULL ;
56     printf("mémoire libérée en C\n");
57   }
58  }
59  exit (EXIT_SUCCESS);
60 }
```

Si on passe la structure `vecteur` par pointeur en C, le pointeur du tableau et le nombre de composantes seront modifiables : il est ainsi possible de libérer le tableau et de le réallouer avec éventuellement une taille différente. Cette opération peut être déclenchée par le fortran, mais en appelant une fonction C à laquelle on repasse la structure vecteur. En effet, plus globalement, un espace mémoire alloué dans un langage doit être désalloué dans ce même langage.

Annexe A

Procédures intrinsèques

Les procédures intrinsèques sont des procédures intégrées au langage, en ce sens que leur interface est connue (aucune déclaration explicite via `USE` par exemple n'est nécessaire) et que leur code objet est inclus dans la bibliothèque du fortran (aucune autre bibliothèque n'est requise lors de l'édition de liens). La plupart des procédures intrinsèques font partie du standard du fortran 90, d'autres ont été ajoutées en fortran 95 puis en fortran 2003, et en fortran 2008 (les fonctions inverses de trigonométrie hyperbolique, les fonctions erreur et gamma ainsi que certaines fonctions de Bessel, voir Site gfortran, de la collection de compilateurs gcc par exemple). Par ailleurs, des options de compilation (*cf.* F.6.1, p. 175 pour `g95`, et F.5.2, p. 173 pour `gfortran`) permettent d'autoriser l'emploi d'autres fonctions intrinsèques comme extension au standard.

Cette annexe rassemble les principales procédures intrinsèques standard du fortran 90, 2003 et 2008. Les procédures *élémentaires* (elemental) (*cf.* 6.5.8, p. 79) sont précédées du signe $*$.

A.1 Fonctions numériques de base

`*ABS(a)`	valeur absolue de l'entier ou du réel a ou module du complexe a	
`*ACOS(x)`	$\arccos(x)$ en radians	
`*ACOSH(x)`	$\operatorname{argch}(x)$	f2008
`*ASIN(x)`	$\arcsin(x)$ en radians	
`*ASINH(x)`	$\operatorname{argsh}(x)$	f2008
`*ATAN(x)`	$\arctan(x)$ en radians dans l'intervalle $[-\pi/2, \pi/2]$	
`*ATAN2(y,x)`	argument du nombre complexe $x + iy \neq 0$, en radians dans l'intervalle $[-\pi, \pi]$	
`*ATANH(x)`	$\operatorname{argth}(x)$	f2008
`*BESSEL_J0(x)`	$\mathrm{J}_0(x)$ fonction de Bessel de première espèce d'ordre 0	f2008
`*BESSEL_J1(x)`	$\mathrm{J}_1(x)$ fonction de Bessel de première espèce d'ordre 1	f2008
`*BESSEL_JN(n,x)`	$\mathrm{J}_n(x)$ fonction de Bessel de première espèce d'ordre n	f2008
`*BESSEL_Y0(x)`	$\mathrm{Y}_0(x)$ fonction de Bessel de deuxième espèce d'ordre 0	f2008
`*BESSEL_Y1(x)`	$\mathrm{Y}_1(x)$ fonction de Bessel de deuxième espèce d'ordre 1	f2008
`*BESSEL_YN(n,x)`	$\mathrm{Y}_n(x)$ fonction de Bessel de deuxième espèce d'ordre n	f2008
`*CONJG(x)`	$\bar{x}$ complexe conjugué	
`*COS(x)`	$\cos(x)$	
`*COSH(x)`	$\cosh(x)$	
`*DIM(a,b)`	différence positive : $\max(a-b, 0)$, donc $a-b$ si $a \geqslant b$ et 0 sinon a et b doivent être de même type, entier ou réel	
`*ERF(x)`	$\operatorname{erf}(x)$ fonction d'erreur $\frac{2}{\sqrt{\pi}} \int_0^x \exp(-t^2)\,\mathrm{d}t$	f2008
`*ERFC(x)`	$\operatorname{erfc}(x)$ fonction d'erreur complémentaire $\frac{2}{\sqrt{\pi}} \int_x^\infty \exp(-t^2)\,\mathrm{d}t$	f2008
`*ERFC_SCALED(x)`	$\exp(x^2)\operatorname{erfc}(x)$ pour éviter l'underflow de erfc pour x grand	f2008
`*EXP(x)`	$\exp(x)$	

`*GAMMA(x)`	$\Gamma(x)$ fonction gamma $\int_0^\infty t^{x-1}\exp(-t)\,\mathrm{d}t$ $\Gamma(n+1)=n!$ si n entier positif	f2008
`*HYPOT(x,y)`	$\sqrt{x^2+y^2}$ sans dépassement de capacité évitable [a]	f2008
`*LOG(x)`	$\ln(x)$ logarithme népérien	
`*LOG10(x)`	$\log(x)$ logarithme décimal	
`*LOG_GAMMA(x)`	$\ln(\lvert\Gamma(x)\rvert)$ logarithme de la valeur absolue de la fonction gamma	f2008
`*MAX(a1,a2,...)`	valeur maximum	
`*MIN(a1,a2,...)`	valeur minimum	
`*MOD(a,p)`	reste de la division entière de a par p par troncature vers 0, soit `a - int(a/p)*p`, fonction impaire de a et paire de p (*cf.* figure A.1, p. 141) [b]	
`*MODULO(a,p)`	reste de la division entière de a par p, soit `a - floor(a/p)*p`, fonction de période $\lvert p\rvert$ en a et son signe est celui de p (*cf.* figures A.2 et A.3, p. 141)	
`*SIGN(a,b)`	$\lvert a\rvert \times \mathrm{sign}(b)$	
`*SIN(x)`	$\sin(x)$	
`*SINH(x)`	$\sinh(x)$	
`*SQRT(x)`	$\sqrt{x}$	
`*TAN(x)`	$\tan(x)$	
`*TANH(x)`	$\tanh(x)$	

a. Pour éviter un dépassement avec le calcul du carré, on factorise le plus grand : $\sqrt{x^2+y^2}=|x|\sqrt{1+(y/x)^2}$.

b. `MOD` est l'équivalent de l'opérateur `%` du C dans la norme C99, qui spécifie que la divison entière se fait par troncature vers 0.

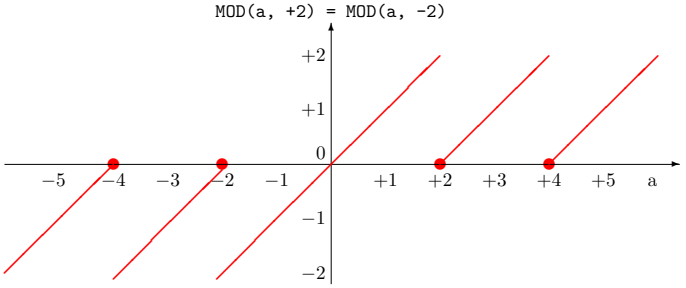


FIGURE A.1 – Graphique de la fonction `MOD(a, p)` pour $p=\pm 2$

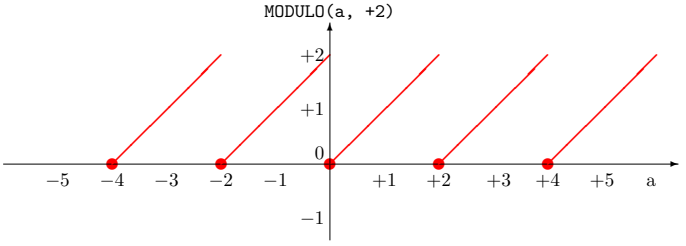


FIGURE A.2 – Graphique de la fonction `MODULO(a, p)` pour $p>0$, par exemple $p=+2$

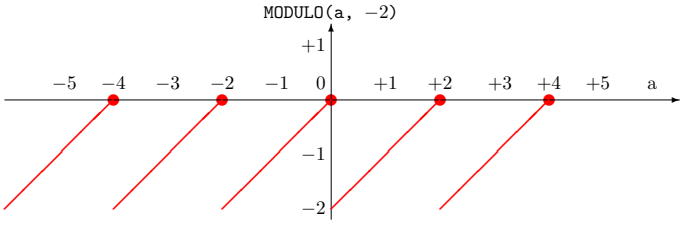


FIGURE A.3 – Graphique de la fonction `MODULO(a, p)` pour $p<0$, par exemple $p=-2$

A.2 Conversion, arrondi et troncature (*cf.* 2.2.5)

`*AIMAG(z)`	partie imaginaire du complexe `z`
`*AINT(a[,kind])`	valeur entière par troncature d'un réel, convertie en réel, c'est-à-dire `REAL(INT(a))`
`*ANINT(a[,kind])`	valeur entière par arrondi d'un réel, convertie en réel, c'est-à-dire `REAL(NINT(a))`
`*CEILING(a[,kind])`	conversion en type entier par excès (*cf.* figure A.4, p.142)
`*CMPLX(x[,y][,kind])`	conversion en type complexe
`*FLOOR(a[,kind])`	conversion en type entier par défaut (*cf.* figure A.5, p. 142)
`*INT(a[,kind])`	conversion en type entier par troncature (*cf.* figure A.6, p. 142)
`*LOGICAL(L[,kind])`	conversion entre variantes de booléens
`*NINT(a[,kind])`	conversion en type entier par arrondi (*cf.* figure A.7, p. 142)
`*REAL(a[,kind])`	conversion en type réel ou partie réelle d'un complexe

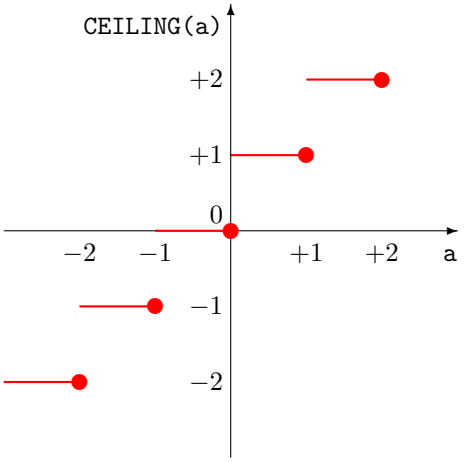


FIGURE A.4 – Graphique de la fonction `CEILING`, entier par excès

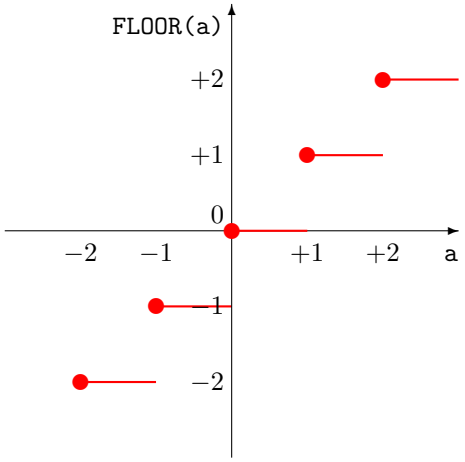


FIGURE A.5 – Graphique de la fonction `FLOOR`, entier par défaut

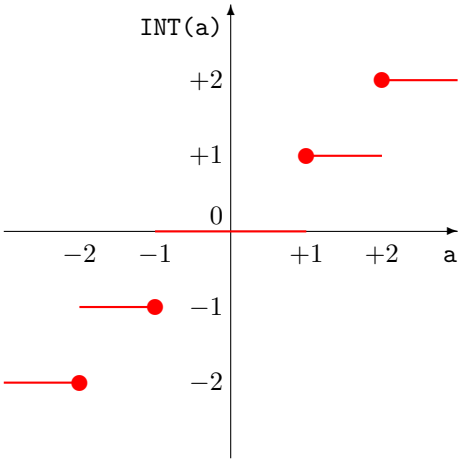


FIGURE A.6 – Graphique de la fonction `INT`, entier vers zéro

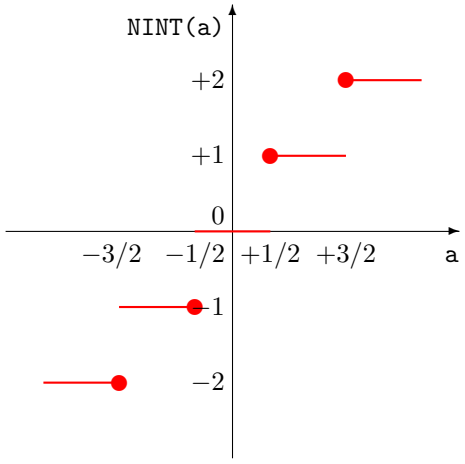


FIGURE A.7 – Graphique de la fonction `NINT`, entier le plus proche

A.3 Générateur pseudo-aléatoire

Les générateurs pseudo-aléatoires fournissent des tirages successifs statistiquement indépendants pris dans une suite *finie* de très longue période qui dépend du processeur et du compilateur,

mais dont l'interface d'appel `RANDOM_NUMBER` est, elle, portable. Le générateur possède une mémoire, ou tableau d'état, qui « progresse » naturellement à chaque invocation, mais qui peut aussi être manipulée grâce au sous-programme `RANDOM_SEED`. En mémorisant l'état du générateur à un instant donné, et en lui imposant [1] plus tard la valeur mémorisée, on peut forcer le générateur à reprendre les tirages à partir d'un état déterminé et à reproduire ainsi plusieurs fois la même séquence.

`*RANDOM_NUMBER(harvest)` générateur de nombres pseudo-aléatoires distribués uniformément sur $[0, 1[$; `harvest` est un scalaire ou un tableau de réels.

`RANDOM_SEED(SIZE|PUT|GET)` initialisation du générateur de nombres pseudo-aléatoires [a], quatre usages possibles en fonction du mot-clef choisi :

`()` initialisation à des valeurs dépendant du processeur

```
call RANDOM_SEED()
```

`(SIZE)` lecture de la dimension du tableau d'état `INTENT(out)`

```
INTEGER :: n
call RANDOM_SEED(SIZE = n)
PRINT *, ' taille ', n
```

`(GET)` lecture des valeurs du tableau d'état `INTENT(out)`

```
INTEGER :: n
INTEGER, DIMENSION(n) :: tab
call RANDOM_SEED(GET = tab(1:n))
PRINT *, ' tableau d''état ', tab
```

`(PUT)` réinitialisation du tableau d'état `INTENT(in)`

```
INTEGER :: n
INTEGER, DIMENSION(n) :: tab
tab(1:n) = ... ! au moins une valeur non nulle
call RANDOM_SEED(PUT = tab(1:n))
```

Avec le compilateur `xlf` d'IBM, un autre mot-clef, `GENERATOR` est disponible, qui permet de choisir entre le générateur de base (`GENERATOR=1`, valeur par défaut) ou un générateur optimisé (`GENERATOR=2`), plus rapide et de période plus grande.

a. La taille du tableau d'état du générateur aléatoire n'est pas spécifiée par le standard. Elle dépend du compilateur utilisé. Sous LINUX et processeur INTEL ou AMD 32 bits, elle est de 1 avec le compilateur `f95` de NAG, de 2 avec le traducteur `f90` (de fortran 90 vers fortran 77) Vast (Pacific Sierra) et le compilateur `ifort` d'INTEL, de 4 avec `g95` et de 34 avec `pgf90` de Portland. Avec un processeur AMD 64 bits, cette taille passe à 2 avec `g95`, qui utilise des entiers par défaut sur 64 bits, mais cela correspond au même nombre d'octets. Avec `gfortran` et un processeur 64 bits, la taille du tableau d'état dépend de la version du compilateur (8 en v4.4.7 et 12 en v4.9.2).

Avec le compilateur de Portland sous SOLARIS de SUN, elle est de 4. Sur la machine NEC SX5 de l'IDRIS, elle est de 128. Sous DEC-OSF1 et processeur ALPHA de 64 bits, elle est de 2, comme sous AIX avec le compilateur `xlf` d'IBM.

Exemple d'usages du générateur aléatoire

```
PROGRAM random_test
IMPLICIT NONE
REAL                  :: x
REAL, DIMENSION(3,4) :: alea
INTEGER :: n_alea  ! taille de la mémoire du générateur aléatoire de fortran90
INTEGER, DIMENSION(:), ALLOCATABLE :: mem_alea, mem_alea2 ! tableaux d'état
```

1. Dès que l'on force le tableau d'état, on risque de perdre l'indépendance entre des tirages successifs.

```
INTEGER  :: i, erreur_alloc
! tirage d'un nombre pseudo-aléatoire entre 0 et 1
CALL RANDOM_NUMBER(x)
WRITE (*, *) ' valeur tirée: x = ',x
! tirage d'un tableau de nombres pseudo-aléatoires indépendants entre 0 et 1
CALL RANDOM_NUMBER(alea)
WRITE (*, *) ' tableau des valeurs tirées:'
DO i = 1, SIZE(alea, 1)
  WRITE (*, *)  alea(i, :)
END DO
! recherche de la taille du tableau d'état
CALL RANDOM_SEED(SIZE = n_alea)
WRITE(*, *) ' taille de la mémoire du générateur aléatoire : ', n_alea
! réservation de mémoire pour deux tableaux d'état
ALLOCATE(mem_alea(n_alea), stat = erreur_alloc)
IF(erreur_alloc /= 0 ) STOP
ALLOCATE(mem_alea2(n_alea), stat = erreur_alloc)
IF(erreur_alloc /= 0 ) STOP
! lecture et mémorisation du tableau d'état courant avant la boucle
CALL RANDOM_SEED(GET = mem_alea)
WRITE(*, *) '- tableau d''état du générateur avant la boucle:',  mem_alea(:)
DO i = 1 , 3
  ! lecture et affichage du tableau d'état courant puis tirage
  CALL RANDOM_SEED(GET = mem_alea2)
  WRITE(*, *) ' tableau d''état du générateur :',  mem_alea2(:)
  CALL RANDOM_NUMBER(x)
  WRITE (*, *) ' valeur tirée: x = ',x
END DO
! réinitialisation avec le tableau d'état mémorisé avant la boucle
! pour repartir du même état et donc obtenir la même série
CALL RANDOM_SEED(PUT = mem_alea)
WRITE (*, *) '- puis réinitialisation à l''état avant la boucle '
DO i = 1 , 3
  ! lecture et affichage du tableau d'état courant puis tirage
  CALL RANDOM_SEED(GET = mem_alea2)
  WRITE(*, *) ' tableau d''état du générateur :',  mem_alea2(:)
  CALL RANDOM_NUMBER(x)
  WRITE (*, *) ' valeur tirée: x = ',x
END DO
END PROGRAM random_test
```

A.4 Représentation des nombres (*cf.* 2.1)

`DIGITS(x)`	nombre de digits (bits en base 2) utilisés pour représenter la mantisse x dans le type de x
`EPSILON(x)`	plus petite valeur dont la somme avec 1 diffère de 1 dans le type de x
∗`EXPONENT(x)`	exposant (entier) de x dans la représentation de x sous la forme `FRACTION(x) * REAL(RADIX(x))**EXPONENT(x)`
∗`FRACTION(x)`	partie fractionnaire de x dans la représentation de x sous la forme `FRACTION(x) * REAL(RADIX(x))**EXPONENT(x)`
`HUGE(x)`	plus grande valeur absolue représentable dans le type de x
`KIND(x)`	type numérique dans lequel est représenté x
`MAXEXPONENT(x)`	valeur maximale de l'exposant (entier) de `RADIX(x)` dans le type de x
`MINEXPONENT(x)`	valeur minimale de l'exposant (entier) de `RADIX(x)` dans le type de x
∗`NEAREST(x,s)`	nombre le plus proche de x *exactement* représentable dans le type de x (par excès ou défaut suivant le signe de s)

`PRECISION(x)`	nombre de chiffres décimaux significatifs dans le type de x
`RADIX(x)`	base de la représentation des nombres dans le type de x (en général 2)
`RANGE(x)`	domaine de x, exprimé en puissance entière de 10, soit `INT(MIN(LOG10(HUGE(x)) , -LOG10(TINY(x))))`
∗`RRSPACING(x)`	inverse (**R**eciprocal) de la distance relative (**R**elative) entre deux nombres de même type que x autour de x (quantité presque indépendante de x pour un type fixé)
∗`SCALE(x, i)`	`x * REAL(RADIX(x))**i`
`SELECTED_INT_KIND(r)`	sous-type entier permettant de représenter les entiers de l'intervalle $]-10^r, 10^r[$
`SELECTED_REAL_KIND(p,r)`	sous-type réel permettant de représenter les réels x tels que $10^{-r} < \mid x \mid < 10^r$ avec p chiffres significatifs
∗`SET_EXPONENT(x, i)`	`FRACTION(x) * REAL(RADIX(x))**i`
∗`SPACING(a)`	plus petite distance entre a et la valeur de même type la plus proche
`TINY(x)`	plus petite valeur absolue non nulle représentable dans le type de x
`BIT_SIZE(i)`	nombre de bits des *entiers* du même type que l'entier [a] i

a. L'argument `i` peut aussi être un tableau d'entiers, mais la taille est celle du scalaire.

A.5 Fonctions opérant sur des tableaux (*cf.* 7.4)

`ALL(array[,dim])`	teste si tous les éléments d'un tableau booléen sont vrais
`ALLOCATED(array)`	teste si un tableau est alloué (résultat booléen)
`ANY(array[,dim])`	teste si au moins un des éléments d'un tableau booléen est vrai
`COUNT(array[,dim])`	compte les éléments vrais d'un tableau booléen
`CSHIFT(array, shift [, dim])`	décalage circulaire de `shift` des éléments d'un tableau, vers les indices croissants si `shift<0`, selon la dimension 1, ou `dim` s'il est spécifié.
`DOT_PRODUCT(vector_a, vector_b)`	produit scalaire de deux vecteurs
`EOSHIFT(array,shift[,boundary][,dim])`	décalage de `shift` des éléments d'un tableau avec perte aux bords (remplacement par 0 ou `boundary` si l'argument est spécifié)
`LBOUND(array[,dim])`	vecteur des bornes inférieures des indices d'un tableau (scalaire si la dimension concernée est précisée)
`MATMUL(matrix_a, matrix_b)`	multiplication matricielle
`MAXLOC(array[,mask])`	tableau de rang 1 donnant la position de la première occurrence de la valeur maximum d'un tableau
`MAXVAL(array[,dim][,mask])`	valeur maximum d'un tableau
`MERGE(tsource,fsource,mask)`	fusion des tableaux `tsource` (si vrai) et `fsource` (si faux) selon le masque `mask`
`MINLOC(array[,mask])`	tableau de rang 1 donnant la position de la première occurrence de la valeur minimum d'un tableau
`MINVAL(array[,dim][,mask])`	valeur minimum d'un tableau
`NORM2(array[,dim][,mask])`	norme euclidienne d'un tableau f2008
`PACK(array, mask [,vector])`	range dans un tableau de rang 1 les éléments de `array` sélectionnés par le masque `mask` (conformant avec `tab`). Si `vector` est fourni, le tableau résultat est si nécessaire complété par les éléments terminaux de `vector` pour obtenir un vecteur de même étendue. `vector` doit donc posséder au moins autant d'éléments que le masque en sélectionne.

`PRODUCT(array[,dim][,mask])`	produit des valeurs d'un tableau
`RESHAPE(source,shape[,pad][,order])`	restructuration du tableau `source` selon le profil `shape`
`SHAPE(array)`	profil d'un tableau
`SIZE(array[,dim])`	taille (nombre d'éléments) d'un tableau [suivant la dimension `dim`]
`SPREAD(source,dim,ncopies)`	création d'un tableau de dimension supérieure par duplication (`ncopies` fois) du scalaire ou tableau `source` selon la dimension `dim`
`SUM(array[,dim][,mask])`	somme des valeurs d'un tableau
`TRANSPOSE(matrix)`	transposition
`UBOUND(array[,dim])`	vecteur des bornes supérieures des indices d'un tableau (scalaire si la dimension concernée est précisée)
`UNPACK(vector, mask, field)`	déploie les éléments du tableau `vector` de rang 1 dans un tableau initialement rempli avec `field` selon le masque `mask` conformant à `field`

A.6 Manipulation de bits

Les fonctions intrinsèques suivantes permettent la manipulation directe de bits sur les entiers :

`*BTEST(i, pos)`	vrai si le bit `pos` de l'entier `i` vaut 1
`*IAND(i, j)`	vrai si tous les bits de `i` et `j` sont égaux
`*IBCLR(i, pos)`	rend un entier du type de `i` dont le bit `pos` a été mis à zéro.
`*IBITS(i, pos, len)`	rend un entier du type de `i` dont les `len` bits en commençant au bit `pos` sont ceux de `i` complétés par des zéros à gauche.
`*IBSET(i, pos)`	rend un entier du type de `i` dont le bit `pso` a été mis à un.
`*IEOR(i, j)`	rend un entier du type de `i` et `j` dont les bits sont obtenus par ou exclusif entre ceux de `i` et `j`.
`*IOR(i, j)`	rend un entier du type de `i` et `j` dont les bits sont obtenus par ou inclusif entre ceux de `i` et `j`.
`*ISHFT(i, shift)`	rend un entier du type de `i` dont les bits sont obtenus par décalage de `shift` vers la gauche (la droite si `shift` négatif) le remplissage étant assuré par des zéros.
`*ISHFTC(i, shift)`	rend un entier du type de `i` dont les bits sont obtenus par décalage de `shift` vers la gauche (la droite si `shift` négatif) le remplissage étant assuré par recyclage circulaire.
`*NOT(i)`	rend un entier du type de `i` dont les bits sont obtenus par inversion de ceux de `i`.

A.7 Interopérabilité avec le langage C

Les procédures intrinsèques suivantes sont fournies par le module `ISO_C_BINDING` pour l'interopérabilité (*cf.* 12.3, p. 132).

`C_LOC(x)`	renvoie un `c_ptr` donnant l'adresse C de `x`
`C_FUNLOC(x)`	renvoie un `c_funptr` donnant l'adresse C de la fonction `x`
`C_F_POINTER(cptr, fptr[, shape])`	sous-programme qui traduit un pointeur du C et un profil éventuel en un pointeur au sens du fortran.
`C_F_PROCPOINTER(cptr, fptr)`	sous-programme qui effectue la même conversion que `c_f_pointer` mais pour des procédures.
`C_SIZEOF(x)`	taille en octets de l'expression interopérable `x`

A.8 Divers

`PRESENT(a)`	fonction à valeur booléenne : vraie si l'argument optionnel `a` de la procédure a été fourni, fausse sinon (*cf.* 6.5.1).
`STORAGE_SIZE(a [, kind=])`	fonction donnant la taille en bits d'un élément du tableau `a` dans le type dynamique de `a` f2008
`TRANSFER(source, mold [, size])`	interprète la représentation binaire de `source` selon le type spécifié par `mold`

L'usage de la fonction `TRANSFER` est susceptible de nuire à la portabilité des codes [2], car cette fonction s'appuie sur la représentation binaire des variables qui peut dépendre de l'ordinateur. Noter que cette opération peut aussi être réalisée par écriture puis relecture sur fichier texte externe au format binaire (format `B`, *cf.* 5.4.1). Elle permet par exemple de considérer les 32 bits d'une zone de la mémoire codant un entier comme les 32 bits codant un réel ; la valeur numérique de ce réel est évidemment différente de celle de l'entier ; les bits considérés peuvent même représenter un réel invalide ou une valeur spéciale telle que `Inf` ou `Nan`.

```
1  PROGRAM t_transfer_int ! test de la fonction transfer entre entiers
2  IMPLICIT NONE
3  ! pour représenter 100 => un octet suffit
4  INTEGER, PARAMETER :: ki1 = SELECTED_INT_KIND(2)
5  ! pour représenter 10**4 => 2 octets suffisent
6  INTEGER, PARAMETER :: ki2 = SELECTED_INT_KIND(4)
7  INTEGER(kind=ki1), DIMENSION(2) :: ti
8  INTEGER(kind=ki2) :: j
9  INTEGER(kind=ki1), DIMENSION(:), ALLOCATABLE :: tk
10 INTEGER :: n
11 ti(1) = 3
12 ti(2) = 1
13 ! recopie binaire du tableau de 2 entiers 8 bits sur un entier 16 bits
14 j = TRANSFER(ti,j)
15 WRITE(*,*) "ti(1) sur ", BIT_SIZE(ti(1)), "bits"
16 WRITE(*,*) "j sur ", BIT_SIZE(j), "bits"
17 WRITE(*,*) "affichage en binaire des 2 entiers ti(2) et ti(1) sur 8 bits chacun"
18 WRITE(*,'(A,B8.8,A,B8.8)') "ti(2)=",ti(2), " ti(1)= ",ti(1) ! ordre inversé
19 WRITE(*,'(B8.8,B8.8,A)') ti(2:1:-1), " concaténés (ordre inverse)"
20 WRITE(*,*) "affichage en binaire de l'entier j sur 16 bits "
21 WRITE(*,'(B16.16)') j
22 WRITE(*,*) "j =", j, " est égal à ti(1) + ti(2)*2**8 =", ti(1) + ti(2)*2**8
23 ! même opération  avec calcul du nombre d'éléments de tk pour stocker j
24 n =SIZE(TRANSFER(j, tk)) ! en principe n = bit_size(j)/bit_size(tk(1))
25 ALLOCATE(tk(n)) ! l'allocation au vol ne fctne pas
26 tk = TRANSFER(j, tk) ! recopie bit à bit de j dans la tableau tk
27 WRITE(*,*) "nb d'éléments de tk ", SIZE(tk), "valeurs des éléments ", tk(:)
28 DEALLOCATE(tk)
29 END PROGRAM t_transfer_int
```

produit l'affichage suivant :

```
 ti(1) sur  8 bits
 j sur  16 bits
 affichage en binaire des 2 entiers ti(2) et ti(1) sur 8 bits chacun
ti(2)=00000001 ti(1)= 00000011
0000000100000011 concaténés (ordre inverse)
 affichage en binaire de l'entier j sur 16 bits
0000000100000011
```

2. Cette réinterprétation d'un motif binaire selon un autre type constitue une opération de bas niveau, facile à coder en C avec des pointeurs par transtypage : `float a; int i; int * p; p =&i; a = *((float *) p);`. En fortran, l'association entre pointeurs de types différents est au contraire interdite.

```
j = 259  est égal à ti(1) + ti(2)*2**8 = 259
nb d'éléments de tk  2 valeurs des éléments  3 1
```

A.9 Pointeurs (*cf.* 11.1.1, p. 121)

`ASSOCIATED(pointer [, target])` rend une valeur booléenne vraie si le pointeur `pointer` est associé à une cible (à la cible `target` si l'argument optionnel est fourni).

`NULL([mold])` rend un pointeur désassocié du type du pointeur `mold` si cet argument est présent.

A.10 Accès à l'environnement fortran

À partir de la norme fortran 2008, le module intrinsèque `ISO_FORTRAN_ENV` (*cf.* 5.3.1, p. 42) fournit deux fonctions d'interrogation sans argument qui renvoient une chaîne de caractères sur l'environnement de compilation :

`COMPILER_OPTIONS()` indique la liste des options du compilateur activées. f2008

`COMPILER_VERSION()` indique la version du compilateur. f2008

Exemple d'utilisation :

```
1 PROGRAM compiler_info
2 ! standard fortran 2008 : emploi du module ISO_FORTRAN_ENV
3 ! juin 2013: pas OK en gfortran 4.4, mais OK en gfortran 4.6
4 USE, INTRINSIC :: ISO_FORTRAN_ENV, ONLY : compiler_version, compiler_options
5 IMPLICIT NONE
6 WRITE(*,*) "version du compilateur : ", compiler_version()
7 WRITE(*,*) "options du compilateur : ",  compiler_options()
8 END PROGRAM compiler_info
```

Affichage obtenu :

```
1 version du compilateur : GCC version 4.6.3
2 options du compilateur : -mtune=generic -march=x86-64 -std=f2008
```

f2003 A.11 Accès à l'environnement système

`GET_ENVIRONMENT_VARIABLE(name [,value] [,length] [,status] [,trim_name])`
sous-programme qui récupère la valeur de la variable d'environnement `name`[3] dans la chaîne de caractères `value`, sa longueur dans l'entier `length`, éventuellement avec les blancs à gauche si `trim_name` vaut `.false.` Le statut de retour `status` vaut 1 si la variable demandée n'existe pas, -1 si la valeur est trop longue pour la chaîne `value` et 0 en l'absence d'erreur ou d'avertissement.

`GET_COMMAND_([command] [,length] [,status])`
sous-programme qui rend dans la chaîne `command` la commande ayant lancé le programme et éventuellement le nombre de caractères de cette chaîne dans `length`. L'argument optionnel `status` vaut `0` quand l'appel s'est déroulé sans problème, `-1` quand la chaîne passée est trop courte pour héberger la ligne de commande, et un nombre positif en cas d'erreur.

`COMMAND_ARGUMENT_COUNT()`
fonction qui rend un entier contenant le nombre de paramètres[4] de la commande ayant lancé le programme.

`GET_COMMAND_ARGUMENT(number [,value] [,length] [,status])`
sous-programme qui rend dans la chaîne `value` l'argument numéro `number`[5] de la commande ayant lancé le programme et éventuellement sa taille dans `length`.

3. C'est l'équivalent de la fonction `getenv` du langage C.
4. C'est à dire `argc -1` du langage C.
5. C'est le paramètre donné par `argv[number]` du langage C.

Exemple d'accès à l'environnement système

On peut ainsi passer des paramètres sous forme de chaînes de caractères au programme principal, de la même façon qu'en C. Après avoir compilé le programme suivant :

```
1  PROGRAM t_environ
2  IMPLICIT NONE
3  CHARACTER(len=20) :: nom, valeur, arg0, arg
4  CHARACTER(len=80) :: commande
5  INTEGER :: longueur, statut, n_param, i
6  WRITE(*,*) "récupération d'une variable d'environnement"
7  nom(1:) = "SHELL"
8  CALL get_environment_variable(nom, valeur, longueur, statut)
9  IF (statut /= 0) STOP "erreur get_environment_variable"
10 WRITE(*,*) nom(1:5), "=", valeur(1:longueur)
11 CALL get_command(COMMAND=commande, LENGTH=longueur, STATUS=statut)
12 IF (statut /= 0) STOP "erreur get_command"
13 WRITE(*,*) "longueur de la ligne de commande", longueur
14 WRITE(*,*) "ligne de commande = |", commande(1:longueur), "|"
15 CALL get_command_argument(0, arg0, longueur, statut)
16 IF (statut /= 0) STOP "erreur get_command_argument"
17 WRITE(*,*) "commande ayant lancé ce programme=", arg0(1:longueur)
18 n_param = command_argument_count()
19 WRITE(*,*) "nb de paramètres de la commande = ", n_param
20 DO i=1, n_param
21     CALL get_command_argument(i, arg, longueur, statut)
22     IF (statut /= 0) STOP "erreur get_command_argument"
23     WRITE(*,*) "paramètre  ", i, " de la commande=", arg(1:longueur)
24 END DO
25 END PROGRAM t_environ
```

L'exécution de la commande `a.out test 12 "5 6"` affiche :

```
1  récupération d'une variable d'environnement
2  SHELL=/bin/bash
3  longueur de la ligne de commande          17
4  ligne de commande = |a.out test 12 5 6|
5  commande ayant lancé ce programme=a.out
6  nb de paramètres de la commande =            3
7  paramètre             1  de la commande=test
8  paramètre             2  de la commande=12
9  paramètre             3  de la commande=5 6
```

f2008

A.12 Exécution d'une commande système

`EXECUTE_COMMAND_LINE(command [,wait] [,exitstat] [,cmdstat], [,cmdmsg])` est un sous-programme qui permet de passer la commande `command` (argument d'entrée de type chaîne de caractères) au système d'exploitation. Tous ses autres arguments sont optionnels :

`wait` est un argument d'entrée de type booléen, vrai par défaut, qui permet d'autoriser une exécution asynchrone de la commande.

`exitstat` est un argument entier de vocation `inout`, qui, sauf si la commande est lancée en asynchrone, récupère le statut de retour de la commande système.

`cmdstat` est un argument de sortie entier qui vaut `0` si la commande s'est exécutée sans erreur, `-1` si le processeur ne peut pas l'exécuter et `-2` si le processeur ne peut honorer l'exécution synchrone requise (`wait=.true.`) et une valeur positive pour les autres erreurs.

`cmdmsg` est un argument de type chaîne de caractères de vocation `inout`, qui, si `cmdstat` est positif, contient un message d'explication.

f95

A.13 Gestion du temps

`DATE_AND_TIME([DATE,] [TIME,] [ZONE,] [VALUES])`
sous-programme permettant d'accéder à la date et à l'heure fournis par le système d'exploitation. Les quatre arguments sont des arguments de sortie optionnels :

- `DATE` : chaîne de 8 caractères recevant la date sous forme `aaaammjj`
- `TIME` : chaîne de 10 caractères recevant le temps sous forme `hhmmss.iii` où `iii` représentent les millisecondes
- `ZONE` : chaîne de 5 caractères recevant le décalage par rapport au temps universel sous la forme `+/-hhmm`
- `VALUE` : tableau de 8 entiers représentant successivement l'année, le mois, le jour, le décalage par rapport au temps universel exprimé en minutes, l'heure, les minutes, les secondes et les millisecondes.

Exemple d'utilisation de `DATE_AND_TIME`

```
1  MODULE util_time
2  IMPLICIT NONE
3  CONTAINS
4  REAL FUNCTION deltat(td, tf)
5    ! calcul de la duree en secondes
6    ! attention : incorrect si changement de date
7    INTEGER, DIMENSION(8), INTENT(in) :: td, tf
8    deltat = (tf(8) -td(8))/1000. + (tf(7) -td(7)) + &
9             60 * ( (tf(6) -td(6)) + 60 *(tf(5) -td(5)))
10 END FUNCTION deltat
11 SUBROUTINE affiche(t)
12   ! affichage de la date suivie du temps
13   ! en heures, minutes, secondes et millisecondes
14   INTEGER, DIMENSION(8), INTENT(in) :: t
15   WRITE(*,'(i2.2,a,i2.2,a,i4.4, a, i2.2,a,i2.2,a,i2.2,a,i3.3, a)') &
16    t(3),"/",t(2),"/",t(1), " ", &
17    t(5), "h", t(6), "'", t(7), '"', t(8), "ms "
18 END SUBROUTINE
19 END MODULE util_time
```

```
21 PROGRAM datetime ! test du sous-programme date_and_time
22 USE util_time
23 INTEGER, DIMENSION(8) :: debut, fin
24 CHARACTER(len=8) :: jour
25 CHARACTER(len=10) :: instant
26 INTEGER:: i, j
27 CALL DATE_AND_TIME(date=jour, time=instant)
28 WRITE(*,*) "date (AAAAMMJJ) : ", jour
29 WRITE(*,*) "heure (hhmmss.*): ", instant
30 CALL DATE_AND_TIME(values=debut)
31 WRITE(*,'(a)', advance="no") "debut "
32 CALL affiche(debut)
33 DO i=1, 1000000 ! ajuster < huge(i)
34   j = 1 - i
35 END DO
36 CALL DATE_AND_TIME(values=fin)
37 WRITE(*,'(a)', advance="no") "fin   "
38 CALL affiche(fin)
39 WRITE(*,*) "duree de la boucle", deltat(debut, fin), 's'
40 END PROGRAM datetime
```

L'exécution de ce code donne par exemple :

```
 date (AAAAMMJJ) : 20110824
 heure (hhmmss.*): 135703.472
debut 24/08/2011 13h57'03"472ms
fin   24/08/2011 13h57'03"482ms
 duree de la boucle 0.01 s
```

`CPU_TIME(time)`
sous-programme permettant d'accéder au temps du processeur exprimé en secondes avec une résolution de la microseconde : l'origine est arbitraire, seules les différences sont pertinentes. Ce sous-programme permet notamment d'évaluer des durées d'exécution de code. Le paramètre est un argument de sortie réel.

`SYSTEM_CLOCK([count] [,count_rate] [,count_max])`
sous-programme permettant d'accéder au temps exprimé en nombre de tops d'horloge de fréquence `COUNT_RATE` modulo une valeur maximale `COUNT_MAX`. Les trois arguments sont des arguments de sortie entiers optionnels.

A.14 Caractères et chaînes de caractères

*`ACHAR(i)`	`i`^{e} caractère (où `i` est compris entre 0 et 127) dans l'ordre de l'ascii
*`CHAR(i [, kind])`	`i`^{e} caractère dans l'ordre du compilateur (qui inclut souvent l'ordre ascii, mais peut aussi fournir des caractères nationaux, comme nos caractères accentués)
*`IACHAR(c)`	position du caractère (entre 0 et 127) `c` dans l'ordre ascii
*`ICHAR(c)`	position du caractère `c` dans l'ordre du compilateur (qui inclut souvent l'ordre ascii, mais peut aussi fournir des caractères nationaux, comme nos caractères accentués)
`NEW_LINE(a)`	caractère de nouvelle ligne dans la variante de caractères de `a`
`SELECTED_CHAR_KIND(name)`	rend le numéro de la variante de caractères dans laquelle la chaîne `name` est donnée (`'DEFAULT'`, `'ASCII'` ou `'ISO_10646'`)

*`ADJUSTL(string)`	justification à gauche de la chaîne `string`
*`ADJUSTR(string)`	justification à droite de la chaîne `string`
`LEN(string)`	longueur de la chaîne `string`
*`LEN_TRIM(string)`	longueur de la chaîne `string` sans les blancs à droite
`REPEAT(string,ncopies)`	duplique `ncopies` fois la chaîne `string`
`TRIM(string)`	suppression des blancs à droite

*`LGE(string_a,string_b)`	vrai si `string_a` est placé après `string_b` dans l'ordre lexicographique de l'ascii ou coïncide avec `string_b`
*`LGT(string_a,string_b)`	vrai si `string_a` est placé strictement après `string_b` dans l'ordre lexicographique de l'ascii
*`LLE(string_a,string_b)`	vrai si `string_a` est placé avant `string_b` dans l'ordre lexicographique de l'ascii ou coïncide avec `string_b`
*`LLT(string_a,string_b)`	vrai si `string_a` est placé strictement avant `string_b` dans l'ordre lexicographique de l'ascii

*`INDEX(string,substring)`	position de la sous-chaîne `substring` dans la chaîne `string`
*`SCAN(string,set)`	repère un des caractères de la chaîne `set` dans la chaîne `string`
*`VERIFY(string,set)`	vérifie que la chaîne `string` ne contient que des caractères de la chaîne `set` (renvoie l'entier 0), sinon donne la position du premier caractère de `string` qui ne fait pas partie de l'ensemble `set`

Codes ascii et iso-latin1

Les fonctions réciproques `achar` et `iachar` traduisent le codage ASCII des caractères explicité dans le tableau ci-après, qui indique le numéro en octal, décimal et hexadécimal de chaque caractère, soit sa représentation s'il est imprimable soit son abréviation et éventuellement la séquence d'échappement (commençant par une contre-oblique) qui le représente dans le langage C.

Oct	Dec	Hex	Char	Oct	Dec	Hex	Char
000	0	00	NUL \0	100	64	40	@
001	1	01	SOH	101	65	41	A
002	2	02	STX	102	66	42	B
003	3	03	ETX	103	67	43	C
004	4	04	EOT	104	68	44	D
005	5	05	ENQ	105	69	45	E
006	6	06	ACK	106	70	46	F
007	7	07	BEL \a	107	71	47	G
010	8	08	BS \b	110	72	48	H
011	9	09	HT \t	111	73	49	I
012	10	0A	LF \n	112	74	4A	J
013	11	0B	VT \v	113	75	4B	K
014	12	0C	FF \f	114	76	4C	L
015	13	0D	CR \r	115	77	4D	M
016	14	0E	SO	116	78	4E	N
017	15	0F	SI	117	79	4F	O
020	16	10	DLE	120	80	50	P
021	17	11	DC1	121	81	51	Q
022	18	12	DC2	122	82	52	R
023	19	13	DC3	123	83	53	S
024	20	14	DC4	124	84	54	T
025	21	15	NAK	125	85	55	U
026	22	16	SYN	126	86	56	V
027	23	17	ETB	127	87	57	W
030	24	18	CAN	130	88	58	X
031	25	19	EM	131	89	59	Y
032	26	1A	SUB	132	90	5A	Z
033	27	1B	ESC	133	91	5B	[
034	28	1C	FS	134	92	5C	\ \\
035	29	1D	GS	135	93	5D	]
036	30	1E	RS	136	94	5E	^
037	31	1F	US	137	95	5F	&
040	32	20	SPACE	140	96	60	`
041	33	21	!	141	97	61	a
042	34	22	"	142	98	62	b
043	35	23	#	143	99	63	c
044	36	24	$	144	100	64	d
045	37	25	%	145	101	65	e
046	38	26	&	146	102	66	f
047	39	27	'	147	103	67	g
050	40	28	(	150	104	68	h
051	41	29	)	151	105	69	i
052	42	2A	*	152	106	6A	j
053	43	2B	+	153	107	6B	k
054	44	2C	,	154	108	6C	l
055	45	2D	-	155	109	6D	m
056	46	2E	.	156	110	6E	n
057	47	2F	/	157	111	6F	o
060	48	30	0	160	112	70	p
061	49	31	1	161	113	71	q
062	50	32	2	162	114	72	r
063	51	33	3	163	115	73	s

064	52	34	4	164	116	74	t
065	53	35	5	165	117	75	u
066	54	36	6	166	118	76	v
067	55	37	7	167	119	77	w
070	56	38	8	170	120	78	x
071	57	39	9	171	121	79	y
072	58	3A	:	172	122	7A	z
073	59	3B	;	173	123	7B	{
074	60	3C	<	174	124	7C	|
075	61	3D	=	175	125	7D	}
076	62	3E	>	176	126	7E	~
077	63	3F	?	177	127	7F	DEL

Plusieurs codages des caractères sur 8 bits permettent d'étendre le code ASCII en représentant aussi les caractères dotés de signes diacritiques (accents, cédille, ...) présents dans les langages européens. Le tableau suivant indique les caractères du codage ISO-8859-1 (alphabet latin 1, pour les langues de l'Europe occidentale) qui sont imprimables et ne figurent pas dans le codage ASCII.

Oct	Dec	Hex	Char	Description
240	160	A0		NO-BREAK SPACE
241	161	A1	¡	INVERTED EXCLAMATION MARK
242	162	A2	¢	CENT SIGN
243	163	A3	£	POUND SIGN
244	164	A4	¤	CURRENCY SIGN
245	165	A5	¥	YEN SIGN
246	166	A6	¦	BROKEN BAR
247	167	A7	§	SECTION SIGN
250	168	A8	¨	DIAERESIS
251	169	A9	©	COPYRIGHT SIGN
252	170	AA	ª	FEMININE ORDINAL INDICATOR
253	171	AB	«	LEFT-POINTING DOUBLE ANGLE QUOTATION MARK
254	172	AC	¬	NOT SIGN
255	173	AD	-	SOFT HYPHEN
256	174	AE	®	REGISTERED SIGN
257	175	AF	¯	MACRON
260	176	B0	°	DEGREE SIGN
261	177	B1	±	PLUS-MINUS SIGN
262	178	B2	²	SUPERSCRIPT TWO
263	179	B3	³	SUPERSCRIPT THREE
264	180	B4	´	ACUTE ACCENT
265	181	B5	µ	MICRO SIGN
266	182	B6	¶	PILCROW SIGN
267	183	B7	·	MIDDLE DOT
270	184	B8	¸	CEDILLA
271	185	B9	¹	SUPERSCRIPT ONE
272	186	BA	º	MASCULINE ORDINAL INDICATOR
273	187	BB	»	RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
274	188	BC	¼	VULGAR FRACTION ONE QUARTER
275	189	BD	½	VULGAR FRACTION ONE HALF
276	190	BE	¾	VULGAR FRACTION THREE QUARTERS
277	191	BF	¿	INVERTED QUESTION MARK
300	192	C0	À	LATIN CAPITAL LETTER A WITH GRAVE
301	193	C1	Á	LATIN CAPITAL LETTER A WITH ACUTE
302	194	C2	Â	LATIN CAPITAL LETTER A WITH CIRCUMFLEX
303	195	C3	Ã	LATIN CAPITAL LETTER A WITH TILDE
304	196	C4	Ä	LATIN CAPITAL LETTER A WITH DIAERESIS
305	197	C5	Å	LATIN CAPITAL LETTER A WITH RING ABOVE
306	198	C6	Æ	LATIN CAPITAL LETTER AE
307	199	C7	Ç	LATIN CAPITAL LETTER C WITH CEDILLA

310	200	C8	È	LATIN CAPITAL LETTER E WITH GRAVE
311	201	C9	É	LATIN CAPITAL LETTER E WITH ACUTE
312	202	CA	Ê	LATIN CAPITAL LETTER E WITH CIRCUMFLEX
313	203	CB	Ë	LATIN CAPITAL LETTER E WITH DIAERESIS
314	204	CC	Ì	LATIN CAPITAL LETTER I WITH GRAVE
315	205	CD	Í	LATIN CAPITAL LETTER I WITH ACUTE
316	206	CE	Î	LATIN CAPITAL LETTER I WITH CIRCUMFLEX
317	207	CF	Ï	LATIN CAPITAL LETTER I WITH DIAERESIS
320	208	D0	Ð	LATIN CAPITAL LETTER ETH
321	209	D1	Ñ	LATIN CAPITAL LETTER N WITH TILDE
322	210	D2	Ò	LATIN CAPITAL LETTER O WITH GRAVE
323	211	D3	Ó	LATIN CAPITAL LETTER O WITH ACUTE
324	212	D4	Ô	LATIN CAPITAL LETTER O WITH CIRCUMFLEX
325	213	D5	Õ	LATIN CAPITAL LETTER O WITH TILDE
326	214	D6	Ö	LATIN CAPITAL LETTER O WITH DIAERESIS
327	215	D7	×	MULTIPLICATION SIGN
330	216	D8	Ø	LATIN CAPITAL LETTER O WITH STROKE
331	217	D9	Ù	LATIN CAPITAL LETTER U WITH GRAVE
332	218	DA	Ú	LATIN CAPITAL LETTER U WITH ACUTE
333	219	DB	Û	LATIN CAPITAL LETTER U WITH CIRCUMFLEX
334	220	DC	Ü	LATIN CAPITAL LETTER U WITH DIAERESIS
335	221	DD	Ý	LATIN CAPITAL LETTER Y WITH ACUTE
336	222	DE	Þ	LATIN CAPITAL LETTER THORN
337	223	DF	ß	LATIN SMALL LETTER SHARP S
340	224	E0	à	LATIN SMALL LETTER A WITH GRAVE
341	225	E1	á	LATIN SMALL LETTER A WITH ACUTE
342	226	E2	â	LATIN SMALL LETTER A WITH CIRCUMFLEX
343	227	E3	ã	LATIN SMALL LETTER A WITH TILDE
344	228	E4	ä	LATIN SMALL LETTER A WITH DIAERESIS
345	229	E5	å	LATIN SMALL LETTER A WITH RING ABOVE
346	230	E6	æ	LATIN SMALL LETTER AE
347	231	E7	ç	LATIN SMALL LETTER C WITH CEDILLA
350	232	E8	è	LATIN SMALL LETTER E WITH GRAVE
351	233	E9	é	LATIN SMALL LETTER E WITH ACUTE
352	234	EA	ê	LATIN SMALL LETTER E WITH CIRCUMFLEX
353	235	EB	ë	LATIN SMALL LETTER E WITH DIAERESIS
354	236	EC	ì	LATIN SMALL LETTER I WITH GRAVE
355	237	ED	í	LATIN SMALL LETTER I WITH ACUTE
356	238	EE	î	LATIN SMALL LETTER I WITH CIRCUMFLEX
357	239	EF	ï	LATIN SMALL LETTER I WITH DIAERESIS
360	240	F0	ð	LATIN SMALL LETTER ETH
361	241	F1	ñ	LATIN SMALL LETTER N WITH TILDE
362	242	F2	ò	LATIN SMALL LETTER O WITH GRAVE
363	243	F3	ó	LATIN SMALL LETTER O WITH ACUTE
364	244	F4	ô	LATIN SMALL LETTER O WITH CIRCUMFLEX
365	245	F5	õ	LATIN SMALL LETTER O WITH TILDE
366	246	F6	ö	LATIN SMALL LETTER O WITH DIAERESIS
367	247	F7	÷	DIVISION SIGN
370	248	F8	ø	LATIN SMALL LETTER O WITH STROKE
371	249	F9	ù	LATIN SMALL LETTER U WITH GRAVE
372	250	FA	ú	LATIN SMALL LETTER U WITH ACUTE
373	251	FB	û	LATIN SMALL LETTER U WITH CIRCUMFLEX
374	252	FC	ü	LATIN SMALL LETTER U WITH DIAERESIS
375	253	FD	ý	LATIN SMALL LETTER Y WITH ACUTE
376	254	FE	þ	LATIN SMALL LETTER THORN
377	255	FF	ÿ	LATIN SMALL LETTER Y WITH DIAERESIS

Annexe B

Ordre des instructions

Ce diagramme indique l'ordre à respecter entre les différentes instructions dans une unité de programme en fortran. Les instructions présentées dans des colonnes différentes à l'intérieur d'une même ligne du tableau peuvent être insérées dans un ordre quelconque.

<table>
<tr><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE, MODULE, BLOCK DATA</td></tr>
<tr><td colspan="3">USE</td></tr>
<tr><td colspan="3">IMPORT</td></tr>
<tr><td rowspan="4">FORMAT, ENTRY</td><td colspan="2">IMPLICIT NONE</td></tr>
<tr><td>PARAMETER</td><td>IMPLICIT</td></tr>
<tr><td>PARAMETER et DATA</td><td>définition de types, blocs d'interfaces, déclarations</td></tr>
<tr><td>DATA</td><td>instructions exécutables</td></tr>
<tr><td colspan="3">CONTAINS</td></tr>
<tr><td colspan="3">procédures internes ou de module</td></tr>
<tr><td colspan="3">END</td></tr>
</table>

Noter, en particulier, que l'on place dans l'ordre, d'abord `USE`, puis `IMPLICIT`, ensuite les déclarations de types ou de variables, puis les instructions exécutables. Enfin, l'instruction `CONTAINS` qui introduit les procédures internes se place juste avant la fin de l'unité de programme qui l'héberge.

En pratique, les instructions de formats sont soit placées juste après les ordres d'entrées–sortie qui les appellent, soit groupées en fin d'unité de programme, surtout dans le cas où elles sont utilisées par plusieurs ordres d'entrées–sortie. ⇐ ♥

Concernant les déclarations de variables dans les procédures, une pratique saine consiste à déclarer d'abord les arguments de la procédure, en précisant leur vocation (`INTENT`), puis les variables locales. ⇐ ♥

Annexe C

La norme IEEE 754 de représentation des flottants

C.1 Introduction

C.1.1 Représentation des réels en virgules flottante

Représenter un réel (non nul) en virgule flottante dans une base b consiste à en donner une approximation finie sous la forme :

$$r = s\, b^{e'} m' = s\, b^{e'} \sum_{i=0}^{q'} p_i\, b^{-i} \tag{C.1}$$

où

- $s = \pm 1$ est le signe ;
- e' est un entier qualifié d'*exposant* ;
- m' est le *significande*, appelé abusivement la *mantisse* que l'on peut décomposer sur la base b selon les $q' + 1$ poids entiers p_i tels que $0 \leqslant p_i < b$

Plusieurs décompositions mantisse–exposant étant possibles (de la même façon qu'en décimal, on peut écrire $1{,}2 \times 10^0$ ou $0{,}12 \times 10^1$), on choisit ici l'exposant pour que $1 \leqslant m' < b$ (qui correspondrait à l'intervalle $[1., 10.[$ si on travaillait en décimal). Cette représentation diffère donc de celle introduite plus haut (*cf.* 2.2, p. 12), avec une mantisse m entre $1/b$ et 1 (qui correspondrait à l'intervalle $[.1, 1.[$ si on travaillait en décimal), choix qui était plus en accord avec les fonctions intrinsèques `FRACTION` et `EXPONENT`. Ces deux représentations sont liées par $e' = e - 1$ et $q' = q - 1$.

Dans le cas de la base 2 et en prenant en compte le choix d'un premier bit p_0 de m' à 1, l'expression (C.1) prend la forme :

$$r = (-1)^s\, 2^{e'} \left(1 + \sum_{i=1}^{q-1} \frac{p_i}{2^i}\right) = (-1)^s\, 2^{e'} (1 + f) = (-1)^s\, 2^e \left(\frac{1}{2} + \sum_{i=1}^{q-1} \frac{p_i}{2^{i+1}}\right) \tag{C.2}$$

où $f = m' - 1$ est la partie fractionnaire du significande sur $q' = q - 1$ bits.

Chaque variante du type réel (déterminée via le paramètre de codage `KIND`) est caractérisée par :

- un nombre M de bits dédiés au stockage de la mantisse qui fixe la précision relative des réels ;
- un nombre E de bits dédiés au stockage de l'exposant qui fixe le domaine de valeurs couvert.

signe	exposant	mantisse
1 bit	E bits	M bits

C.1.2 Arithmétique étendue

Mais la norme IEEE-754-2008 permet aussi de représenter des résultats d'opérations arithmétiques qui ne s'évaluent pas comme des réels, tels que :

- ◇ `1./0.` soit $+\infty$ affiché `+Inf`
- ◇ `-1./0.` soit $-\infty$ affiché `-Inf`
- ◇ `0./0.` qui est indéterminé, ou `sqrt(-1.)` qui est une opération invalide : il est codé comme `NaN` soit « Not-A-Number ». `NaN` possède aussi une variante négative. Enfin, les résultats $\pm$`NaN` peuvent donner lieu à des messages (signaling NaN) ou être propagés sans avertissement dans les calculs (quiet NaN) : un bit particulier permet de déterminer ce comportement.

L'arithmétique traditionnelle s'étend à ces valeurs spéciales de façon naturelle si on précise que dès qu'un `NaN` intervient dans une expression, le résultat est aussi représenté par un `NaN`. Par exemple, `Inf + Inf` donne `+Inf`, mais `Inf - Inf` donne `NaN`. Enfin, le codage IEEE permet de représenter un « zéro positif » et un « zéro négatif » qui interviennent naturellement dans des opérations arithmétiques étendues telles que `1./+Inf` ou `1./-Inf`.

C.2 Les codes normaux

L'exposant e' codé sur E bits qui détermine l'intervalle couvert définissant ainsi le domaine, est de signe *a priori* quelconque entre $-2^{E-1}+1$ et 2^{E-1} : on lui ajoute $2^{E-1}-1$ de façon à stocker le nombre sans signe $e''=e'+2^{E-1}-1$, appelé *exposant biaisé*, qui est un entier compris entre 0 et 2^E-1. Mais on exclut les deux valeurs extrêmes de l'exposant pour réserver ces *codes spéciaux* (où les bits de l'exposant e'' sont tous à 0 ou tous à 1) aux deux zéros signés et à l'arithmétique étendue de la norme IEEE. Ainsi, pour les *codes normaux*, $1 \leqslant e'' \leqslant 2^E-2$, donc $-2^{E-1}+2 \leqslant e' \leqslant 2^{E-1}-1$.

C.2.1 Les nombres normalisés

En binaire, pour les nombres *normalisés*, le bit de plus fort poids de la mantisse m' vaut donc 1. On convient alors de ne pas le stocker pour gagner un bit sur la précision de la représentation ; on stocke donc la partie fractionnaire f de la mantisse sur $M=q'=q-1$ bits seulement. Mais le nombre de bits de la mantisse donné par la fonction intrinsèque `DIGITS` est $M+1=q$ car il prend en compte ce bit caché.

Ainsi pour les flottants normalisés, $r=(-1)^s\ 2^{e''-2^{E-1}+1}\times 1.f$ où f est la partie fractionnaire de la mantisse. Dans chaque octave $[2^n, 2^{n+1}[$, il y a 2^M nombres représentés exactement, en progression arithmétique de raison 2^{n-M} ; cette raison double à chaque changement d'octave. Si le successeur de 2^n est $2^n(1+2^{-M})$, son prédécesseur est $2^n(1-2^{-M-1})$.

Valeurs extrêmes

Le plus grand réel normalisé positif ainsi représentable, donné par la fonction `HUGE`, est donc obtenu pour $e''=2^E-2$, soit $e'=2^{E-1}-1$ avec $p_0=1$ et $p_i=1$ pour $i\geqslant 0$, soit $m'=\sum_{i=0}^{i=M}2^{-i}=2(1-2^{-M+1})$. Il vaut donc $2^{2^{E-1}}(1-2^{-M-1})$. Si un calcul donne une valeur absolue supérieure à `HUGE(x)`, il y a théoriquement dépassement de capacité par valeur supérieure (floating overflow).

Le plus petit réel positif normalisé ainsi représentable, donné par la fonction `TINY`, est donc obtenu pour $e''=1$, soit $e'=-2^{E-1}+2$ avec $p_0=1$ et $p_i=0$ pour $i\geqslant 1$, soit $f=0$. Il vaut donc $2^{-2^{E-1}+2}$. Si un calcul donne une valeur absolue non nulle mais inférieure à `TINY(x)`, il y a théoriquement dépassement de capacité par valeur inférieure (floating underflow).

Le nombre d'octaves de réels positifs est $e''_{\max}-e''_{\min}+1$, soit 2^E-2.

C.2.2 Les nombres dénormalisés

Mais, quitte à perdre en précision, on peut aussi représenter des nombres de valeur absolue inférieure à la plus petite valeur ayant un exposant décalé e'' nul : on travaille alors en virgule fixe. Si l'exposant décalé e'' est nul, les M bits suivants représentent exactement la mantisse sans

bit caché à 1 : on définit ainsi les nombres *dénormalisés*. Le plus petit nombre dénormalisé positif (tous les bits à `0` sauf le dernier) vaut donc $2^{-2^{E-1}+2-M}$.

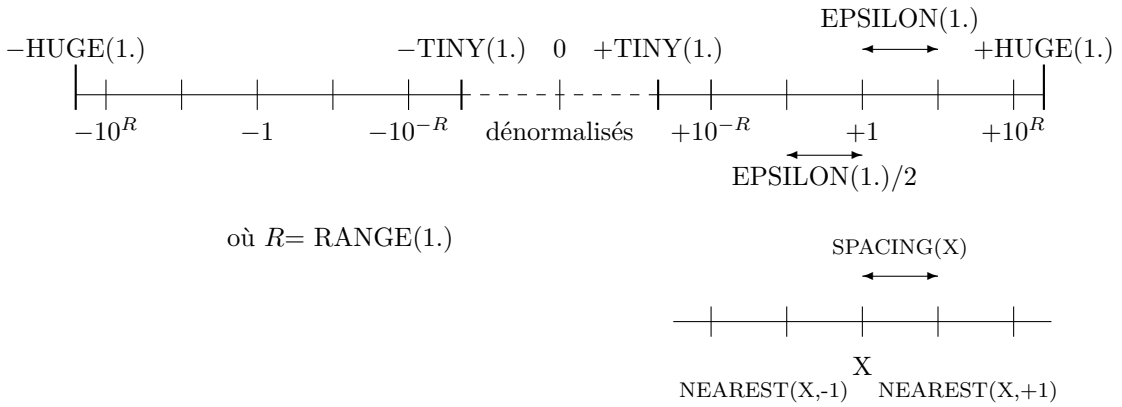


FIGURE C.1 – Représentation des réels. Même si l'échelle s'apparente à une échelle log, celle-ci n'est pas respectée afin de figurer le zéro. De plus, successeur (`NEAREST(X,1.)`)et prédécesseur de `X` (`NEAREST(X,-1.)`) sont en général à égale distance de `X` (progression arithmétique de raison `SPACING(X)` dans chaque octave) sauf si `X` est à la limite d'une octave, c'est-à-dire une puissance entière de 2 (dans ce cas le pas est deux fois plus faible du côté de zéro).

C.3 Les codes spéciaux

Les codes employés pour les valeurs spéciales de l'arithmétique étendue sont caractérisés par le fait que tous leurs bits d'exposant sont à 1. On note aussi les deux zéros signés avec tous les bits d'exposant à 0.

	signe	exposant	mantisse
`+0`	0	E bits à 0	M bits à zéro
`-0`	1	E bits à 0	M bits à zéro
quiet `NaN` > 0	0	E bits à 1	1 suivi de (M-1) bits quelconques
signaling `NaN` > 0	0	E bits à 1	0 suivi d'au moins 1 bit à 1
quiet `NaN` < 0	1	E bits à 1	1 suivi de (M-1) bits quelconques
signaling `NaN` < 0	1	E bits à 1	0 suivi d'au moins 1 bit à 1
`+Inf`	0	E bits à 1	M bits à 0
`-Inf`	1	E bits à 1	M bits à 0

C.3.1 Arithmétique IEEE et options de compilation

Il est cependant nécessaire de choisir les options de compilation adéquates pour imposer le respect de la norme IEEE (voir par exemple F.4.1 p. 171 pour `ifort` et F.3.1 p. 171 pour `pgf95`) dans la propagation des drapeaux indiquant la sortie de l'arithmétique classique. On peut en effet constater une grande diversité de comportement par défaut entre plusieurs compilateurs sur le simple cas d'un calcul dont le résultat devrait être indéterminé.

```
PROGRAM nan_indeterm
IMPLICIT NONE
REAL :: zero, infini, indet
zero = 0.
infini = 1./zero
indet = infini * zero ! en principe indéterminé
```

```
WRITE(*,*) " 0. ,            1./0.,            (1./0.)*0."
WRITE(*,*) zero, infini, indet
END PROGRAM nan_indeterm
```

```
différences de comportement suivant les compilateurs
et sensiblité aux options IEEE notamment
pgf90 et ifort sans option sont à éviter
=> "pgf90 -Kieee" et "ifort -fltconsistency"
pgf90 nan_indeterm.f90
 0. ,            1./0.,            (1./0.)*0.
   0.000000                Inf    0.000000
pgf90 -Kieee nan_indeterm.f90
 0. ,            1./0.,            (1./0.)*0.
  0.000000                Inf             NaN
ifort nan_indeterm.f90
 0. ,            1./0.,            (1./0.)*0.
 0.0000000E+00 Infinity        0.0000000E+00
ifort -fltconsistency nan_pgf.f90
                    (ou "-mp" mais "-mp1" est insuffisnt))
 0. ,            1./0.,            (1./0.)*0.
 0.0000000E+00 Infinity       NaN
 g95 nan_indeterm.f90
 0. ,            1./0.,            (1./0.)*0.
 0. +Inf NaN
 gfortran nan_indeterm.f90
 0. ,            1./0.,            (1./0.)*0.
  0.000000          +Infinity            NaN
 nagfor nan_indeterm.f90
NAG Fortran Compiler Release 5.2(643)
Runtime Error: *** Arithmetic exception: Floating divide by zero - aborting
Abandon
```

C.4 Le codage IEEE des flottants sur 32 bits et 64 bits

C.4.1 Le codage IEEE des flottants sur 32 bits

Les réels par défaut sont généralement stockés sur 32 bits avec 8 bits d'exposant et 23 bits plus un bit à 1 non stocké pour la mantisse. Le décalage de l'exposant est de 127.

signe	exposant biaisé	partie fractionnaire de la mantisse
s	e''	$f = m' - 1$
1 bit	8 bits	23 bits

$$r = (-1)^s \; 2^{e''-127} \times 1.f \text{ où } f = m' - 1 \tag{C.3}$$

Pour des flottants sur 32 bits (*cf.* table C.1, p. 160), la plus petite valeur positive en normalisé, c'est-à-dire le résultat de la fonction `TINY`, vaut 2^{-126} soit approximativement `1.175494E-38`. La plus grande valeur, c'est-à-dire le résultat de la fonction `HUGE`, est obtenue avec les 23 bits de la mantisse à 1 et le plus grand exposant (7 bits à 1 et le dernier à 0). Elle vaut $2^{+127} \times (2 - 2^{-23}) \approx 2^{128}$, soit approximativement `3.402823E+38`. Quant à la précision relative, elle est caractérisée par la fonction `EPSILON`, l'écart relatif *maximal* entre deux flottants successifs, soit $2^{-23} \approx$ `1.192093E-7`. Dans chaque octave, de 2^n à 2^{n+1} par exemple, il y a 2^{23}, soit environ 8 millions de réels en progression arithmétique de raison 2^{n-23}. Pour conserver approximativement la précision relative, le pas double quand on progresse d'une octave en s'éloignant de zéro. Noter que le voisin gauche de +1., soit `NEAREST(+1.,-1.)` est deux fois plus proche de +1. que son voisin de droite, soit `NEAREST(+1.,+1.)`. Enfin, le nombre d'octaves [1] de réels positifs normalisés est de $2^8 - 2 = 254$.

1. Le nombre total de réels normalisés non nuls sur 32 bits est donc $2 \times 2^{23}(2^8 - 2)$, soit $2^{32} - 2^{25}$. Si on y ajoute

TABLE C.1 – Codage IEEE de quelques valeurs remarquables sur 32 bits

0	00000000	00000000000000000000000	+0
1	00000000	00000000000000000000000	-0
0	00000001	00000000000000000000000	TINY(1.) $2^{1-127} \times 1. = 2^{-126}$
0	11111110	11111111111111111111111	HUGE(1.) $\approx 2^{254-127} \times 2$
0	01111111	00000000000000000000000	$1 = 2^{127-127}$
0	01111111	10000000000000000000000	$1.5 = 2^{127-127}(1 + 2^{-1})$
0	01111111	11000000000000000000000	$1.75 = 2^{127-127}(1 + 2^{-1} + 2^{-2})$
0	10000000	11111111111111111111111	$2 - 2^{-23}$ = NEAREST(2.,-1.)
0	10000000	00000000000000000000000	2
0	01111111	00000000000000000000001	NEAREST(1.,+1.)=1+EPSILON(1.) $= 1 + 2^{-23}$
0	01111110	11111111111111111111111	NEAREST(1.,-1.) $= 1 - 2^{-24}$
0	00000000	00000000000000000000001	$2^{-127} \times 2^{-22} = 2^{-149}$ (plus petit dénormalisé > 0)
0	11111111	00000000000000000000000	+Inf

f2008 ⇒ Pour vérifier le codage binaire, on peut utiliser le format binaire `b` (*cf.* 5.4.1, p. 49) des réels : en écriture, il permet de convertir un réel décimal en binaire et en lecture de convertir du binaire en réel affiché en décimal.

```
PROGRAM bin_real
! format B avec des réels = norme 2008
! acceptée par gfortran et ifort, pas par g95
IMPLICIT NONE
REAL :: r0 = 1. ! réel par défaut (32 bits)
CHARACTER(len=32) :: binreal
WRITE(*, *) "mantisse sur",DIGITS(r0),"bits (1 caché)"
! écriture d'un réel en binaire
WRITE(*, *) r0
WRITE(*, "(a)") "seeeeeeeemmmmmmmmmmmmmmmmmmmmmmm"
WRITE(*, "(b32.32)") r0
! lecture d'un réel saisi en binaire
!       "seeeeeeeemmmmmmmmmmmmmmmmmmmmmmm"
binreal="00111111100000000000000000000000" ! 1.
READ(binreal, "(b32.32)") r0
WRITE(*, "(es16.9)") r0
END PROGRAM bin_real
```

Par exemple pour des réels sur 32 bits, le programme `bin_real` ci-contre affiche :

```
 mantisse sur 24 bits (1 caché)
 1.
seeeeeeeemmmmmmmmmmmmmmmmmmmmmmm
00111111100000000000000000000000
 1.000000000E+00
```

C.4.2 Le codage IEEE des flottants sur 64 bits

Les réels en double précision sont généralement stockés sur 64 bits avec 11 bits d'exposant et 52 bits plus un bit à 1 non stocké pour la mantisse. Le décalage de l'exposant est de 1023.

signe	exposant biaisé	partie fractionnaire de la mantisse
s	e''	$f = m' - 1$
1 bit	11 bits	52 bits

$$r = (-1)^s \; 2^{e''-1023} \times 1.f \text{ où } f = m' - 1 \tag{C.4}$$

2×2^{23} réels dénormalisés et 2 zéros, cela donne $2^{32} - 2^{24} + 2$ valeurs normales, parmi les 2^{32} codes disponibles. Au contraire, tous les codes sur 32 bits représentent des entiers valides, qui sont donc très légèrement plus nombreux que les réels sur 32 bits !

Avec 64 bits, le domaine s'étend : `TINY` vaut $2^{-2^{10}+2} = 2^{-1022} \approx 2.2250738585072014 \times 10^{-308}$ alors que `HUGE` vaut $2^{2^{10}-1}(2 - 2^{-52}) = 2^{1024}(1 - 2^{-53}) \approx 1.7976931348623157 \times 10^{+308}$. Mais la précision est aussi améliorée avec `EPSILON`$=2^{-52} \approx 2.220446049250313 \times 10^{-16}$. Chaque octave comporte 2^{52} soit environ $4,510^{15}$ réels en progression arithmétique et il y a $2^{11} - 2$, soit 2046 octaves de réels positifs normalisés.

Pour plus d'informations sur la norme IEEE-754, on pourra consulter le site WIKIPEDIA (2010) et le chapitre 9 « Arithmétique des ordinateurs » de l'ouvrage de STALLINGS (2003). On pourra aussi s'exercer sur les convertisseurs hexadécimal-IEEE en ligne sur le site VICKERY (2009).

Annexe D

Correspondance entre la syntaxe du fortran 90 et celle du langage C

Avertissement

Les correspondances de syntaxe mises en évidence ici ne signifient absolument pas des fonctionnalités équivalentes dans les deux langages. Par exemple, les chaînes de caractères sont traitées de manière différente en fortran, où le type intrinsèque existe et en C, où elles sont représentées par des tableaux de caractères. Sans chercher l'exhaustivité dans l'analogie, elles ont simplement pour but d'aider le lecteur familier du langage C à éviter les erreurs de syntaxe en fortran.

D.1 Déclarations des types de base (*cf.* 2.2)

fortran 90	C89	compléments C99
`INTEGER :: var`	`int var;`	
`REAL :: var`	`float var;`	
`COMPLEX :: var`		`complex var ;`
`CHARACTER :: var`	`char var;`	
`LOGICAL :: var`		`bool var ;`

D.2 Opérateurs algébriques (*cf.* 3.1)

	fortran 90	langage C
addition	`+`	`+`
soustraction	`-`	`-`
multiplication	`*`	`*`
division	`/`	`/`
élévation à la puissance	`**`	≈ `pow(x,y)`
reste modulo `j`	≈ `MOD(i,j)` ou `MODULO(i,j)`	`%`

D.3 Opérateurs de comparaison (*cf.* 3.2)

	fortran 90	C
inférieur à	`<`	`<`
inférieur ou égal à	`<=`	`<=`
égal à	`==`	`==`
supérieur ou égal à	`>=`	`>=`
supérieur à	`>`	`>`
différent de	`/=`	`!=`

D.4 Opérateurs logiques (*cf.* 3.3)

	fortran 90	C	
		C89	C95
ET	`.AND.`	`&&` (évaluation minimale)	`and`
OU	`.OR.`	`||` (évaluation minimale)	`or`
NON	`.NOT.`	`!`	`not`
équivalence	`.EQV.`		
OU exclusif	`.NEQV.`		

D.5 Opérations sur les bits (*cf.* A.6)

Les arguments des fonctions (en fortran) et les opérandes des opérateurs (en C) agissant bit à bit sont des entiers.

fortran 90	signification	langage C
`NOT(i)`	négation bit à bit	`~i`
`IAND(i, j)`	et	`i & j`
`IOR(i, j)`	ou (inclusif)	`i | j`
`IEOR(i, j)`	ou exclusif	`i ^ j`
`ISHFT(i, j)`	décalage à gauche de `j` bits	`i << j`
`ISHFT(i, -j)`	décalage à droite de `j` bits	`i >> j`

D.6 Structures de contrôle (*cf.* chap. 4)

fortran 90	C

`IF` (expr. log.) `THEN`
bloc
`ELSE`
bloc
`END IF`
	`if` (expr. log.)
	bloc
	`else`
	bloc

`SELECT CASE` (expr.)
`CASE` ('sélecteur')
bloc
 	
`CASE (DEFAULT)`
bloc
`END SELECT`
	`switch` (expr. entière) {
	`case` sélecteur :
	bloc
	`break;`
	`default` :
	bloc
	}

`DO` entier = début, fin`[`, pas`]`
bloc
`END DO`
	`for` (expr$_1$; expr$_2$; expr$_3$)
	bloc

`DO WHILE` (expr. log.) bloc `END DO`	`while` (expr. log.) bloc
`CYCLE`	`continue;`
`EXIT`	`break;`
`GO TO` étiquette-numér.	`goto` étiquette;
`RETURN`	`return;`
`STOP`	`exit;`

D.7 Pointeurs (*cf.* chap. 11)

	fortran 90	C
déclarer des pointeurs	`REAL, POINTER:: ptr, pts`	`float *ptr, *pts;`
déclarer une cible	`REAL, TARGET:: r`	`float r;`
pointer vers	`ptr => r`	`ptr = &r;`
pointer vers	`pts => ptr`	`pts = ptr;`
cible pointée	`ptr`	`*ptr`
dissocier un pointeur	`ptr => null()`	`ptr = NULL;`
tester l'association	`IF(ASSOCIATED(ptr))`	`if(ptr != NULL);`
tester l'association	`IF(ASSOCIATED(ptr, pts))`	`if(ptr == pts);`
allouer un pointeur	`ALLOCATE(ptr[...])` de plus, associe `ptr`	`ptr=malloc(...);` `ptr=calloc(...);`
désallouer un pointeur	`DEALLOCATE(ptr)` de plus, désassocie `ptr`	`free(ptr);` ajouter `ptr = NULL;`
conversion implicite	`pt_real = pt_int`	`*pt_real = *pt_int`
interdit/déconseillé	~~`pt_real => entier`~~	`pt_real = &entier`
interdit/déconseillé	~~`pt_real => pt_entier`~~	`pt_real = pt_entier`

D.8 Fonctions mathématiques (*cf.* A.1, A.2 et A.4)

Les fonctions mathématiques intrinsèques du fortran sont utilisables sans faire appel à un quelconque `USE`, ni lier explicitement une bibliothèque. À l'inverse, pour utiliser les fonctions de la bibliothèque mathématique en langage C, il est nécessaire d'une part d'inclure le fichier d'entête `math.h` et d'autre part de lier la bibliothèque `libm.a`. De plus, ces fonctions sont génériques en fortran, alors qu'en C, chaque type utilise une fonction spécifique [1], d'où les variantes pour argument entier (i), double (r), voire complexe (c) en C99, et les sous-variantes pour un argument de type réel (`double` par défaut) suffixées `f` pour les `float`, et `l` pour les `long double`.

fortran	C89	C99	remarques
`ABS(a)`	`abs(a)` (i) `fabs/f/l(a)` (r)	`cabs(a)` (z)	`abs` pour les entiers en C
`ACOS(a)`	`acos/f/l(a)` (r)	`cacos/f/l(a)` (z)	
`AIMAG(a)`		`cimag/f/l(a)` (z)	norme C99
`ASIN(a)`	`asin/f/l(a)` (r)	`casin/f/l(a)` (z)	
`ATAN(a)`	`atan/f/l(a)` (r)	`catan/f/l(a)` (z)	
`ATAN2(a,b)`	`atan2/f/l(a,b)`		
`CEILING(a)`	`ceil/f/l(a)`		résultat flottant en C
`CONJG(a)`		`conj/f/l(a)` (z)	norme C99
`COS(a)`	`cos/f/l(a)` (r)	`ccos/f/l(a)` (z)	
`COSH(a)`	`cosh/f/l(a)` (r)	`ccosh(a)` (z)	
`EXP(a)`	`exp/f/l(a)` (r)	`cexp/f/l(a)` (z)	
`FLOOR(a)`	`floor/f/l(a)`		résultat flottant en C
`LOG(a)`	`log/f/l(a)` (r)	`clog/f/l(a)` (z)	
`LOG10(a)`	`log10/f/l(a)`		
`MOD(a,p)`	`a%p` si positifs	`a%p`	opérateur en C
`MODULO(a,p)`	`a%p` si positifs		opérateur en C
`NINT(a)`	`lrint/f/l(a)`		résultat entier long en C
`a**b`	`pow/f/l(a,b)` (r)	`cpow/f/l(a,b)` (z)	opérateur en fortran
`REAL(a)`		`creal/f/l(a,b)`	
`SIGN(a,b)`		`copysign/f/l(a,b)`	
`SIN(a)`	`sin/f/l(a)` (r)	`csin/f/l(a)` (z)	
`SINH(a)`	`sinh/f/l(a)` (r)	`csinh/f/l(a)` (z)	
`SQRT(a)`	`sqrt/f/l(a)` (r)	`csqrt(a)` (z)	
`TAN(a)`	`tan/f/l(a)` (r)	`ctan/f/l(a)` (z)	
`TANH(a)`	`tanh/f/l(a)` (r)	`ctanh/f/l(a)` (z)	
`NEAREST(a,s)`		`nextafter/f/l(a,s)`	

fortran 2008	C89 `(r)`	C99 `(z)` avec `tgmath.h`	remarques
`ACOSH(a)`	`acosh/f/l(a)`	`cacosh(a) acosh`	C99 + `tgmath.h`
`ASINH(a)`	`asinh/f/l(a)`	`casinh/f/l(a) asin`	C99 + `tgmath.h`
`ATANH(a)`	`atanh/f/l(a)`	`catanh/f/l(a) atanh`	C99 + `tgmath.h`
`BESSEL_J0(a)`	`j0/f/l(a)`	les fonctions de Bessel `j0`, `j1`, `jn`, `y0`, `y1` et `yn` ne sont pas standard en C, y compris dans les normes C99 ou C2011	$J_0(a)$
`BESSEL_J1(a)`	`j1/f/l(a)`		$J_1(a)$
`BESSEL_JN(n,a)`	`jn/f/l(n,a)`		$J_n(n,a)$
`BESSEL_Y0(a)`	`y0/f/l(a)`		$Y_0(a)$
`BESSEL_Y1(a)`	`y1/f/l(a)`		$Y_1(a)$
`BESSEL_YN(n,a)`	`yn/f/l(n,a)`		$Y_n(n,a)$
`ERF(a)`	`erf/f/l(a)`		$\frac{2}{\sqrt{\pi}}\int_0^a e^{-t^2}\,\mathrm{d}t$
`ERFC(a)`	`erfc/f/l(a)`		$\frac{2}{\sqrt{\pi}}\int_a^\infty e^{-t^2}\,\mathrm{d}t$
`GAMMA(a)`	`tgamma/f/l(a)`		$\Gamma(a)$
`LOG_GAMMA(a)`	`lgamma/f/l(a)`		$\ln(\lvert\Gamma(a)\rvert)$
`HYPOT(a, b)`	`hypot/f/l(a,b)`		$\sqrt{a^2+b^2}$

⇐ f2008

1. En C99, il est cependant possible d'implémenter des fonctions mathématiques génériques pour les nombres réels flottants grâce au préprocesseur en incluant le fichier `tgmath.h`.

D.9 Formats (*cf.* 5.4.1)

La correspondance entre les spécifications de format en fortran et en C reste très approximative.

⚠⇒ Les comportements des deux langages pour les entrées-sorties sont assez différents : en particulier en écriture, lorsque la taille du champ de sortie précisée dans la spécification de format s'avère insuffisante pour permettre l'écriture, le C prend la liberté d'étendre cette taille (quitte à ne pas respecter la taille prévue) alors que le fortran refuse de déborder et affiche des caractères * en lieu et place de la valeur à écrire.

⚠⇒ D'autre part, le fortran donne la priorité à la liste d'entrées/sorties sur le nombre de descripteurs actifs du format, alors que le langage C cherche à satisfaire en priorité les descripteurs actifs. En particulier, si le nombre de descripteurs actifs est inférieur au nombre d'entités à coder ou décoder, le fortran ré-explore le format, alors que dans le cas contraire, il ignore les descripteurs **actifs** superflus. À l'inverse, en C, si le nombre de descripteurs actifs est inférieur au nombre d'éléments de la liste, seuls ceux qui correspondent à un descripteur sont codés ou décodés, alors que dans le cas contraire, les descripteurs actifs superflus provoquent des accès à des zones mémoires non réservées avec des conséquences imprévisibles mais souvent plus graves en entrée qu'en sortie.

	fortran	C
entier en décimal	`In`	`%nd`
	`I0`	`%d`
entier en octal	`On`	`%nO`
	`O0`	`%O`
entier en hexadécimal	`Zn`	`%nX`
	`Z0`	`%X`
flottant en virgule fixe	`Fn.p`	`%n.pf`
	`F0.p`	`%f`
flottant en mantisse et exponentielle	`En.p`	`%n.pe`
général	`Gn.p`	`%n.pg`
	`G0`	`%g`
chaîne de caractères	`An`	`%n.ps`

D.10 Codage des valeurs numériques : domaine et précision

D.10.1 Domaine des entiers (*cf.* 2.3.1)

fortran		langage C
Entiers sur 32 bits		
`HUGE(1)`	$2147483647 = 2^{31} - 1 \approx 2 \times 10^9$	`INT32_MAX`
Entiers sur 64 bits		
`HUGE(1_8)`	$9223372036854775807 = 2^{63} - 1 \approx 9 \times 10^{18}$	`INT64_MAX`

D.10.2 Domaine et précision des flottants (*cf.* 2.3.1)

fortran		langage C
Flottants sur 32 bits		
`HUGE(1.)`	$3.402823 \times 10^{+38}$	`FLT_MAX`
`TINY(1.)`	1.175494×10^{-38}	`FLT_MIN`
`EPSILON(1.)`	1.192093×10^{-7}	`FLT_EPSILON`
Flottants sur 64 bits		
`HUGE(1.D0)`	$1.7976931348623157 \times 10^{+308}$	`DBL_MAX`
`TINY(1.D0)`	$2.2250738585072014 \times 10^{-308}$	`DBL_MIN`
`EPSILON(1.D0)`	$2.220446049250313 \times 10^{-16}$	`DBL_EPSILON`

Annexe E

Quelques éléments de correspondances entre fortran et python (module `numpy`)

E.1 Fonctions opérant sur les tableaux en fortran et en python avec le module `numpy`

Le module `numpy` de python possède des types tableaux de rang n (`ndarray`) dont la structure est très différente des tableaux du fortran : en particulier l'ordre de stockage est par défaut celui du langage C (dernier indice le plus rapide). Mais on peut lui préférer celui du fortran (premier indice le plus rapide), avec l'argument optionnel `order='F'` de `array`.

La correspondance des fonctions opérant sur les tableaux en fortran avec celles du module `numpy` de python est très approximative. Sous python, aux fonctions traditionnelles, on peut en particulier préférer les méthodes de la classe `ndarray`, par exemple pour transposer une matrice. Mais les opérations mathématiques effectuées par les fonctions de `numpy` et du fortran sont très similaires. Noter que, dans les fonctions de réduction, à l'argument optionnel `DIM` du fortran qui permet par exemple de préciser quelle dimension est «réduite» correspond l'argument optionnel `axis` de `numpy` (avec le décalage dû à l'indexation à partir de 0).

	fortran	**numpy**
taille	`size`	`size`
profil	`shape`	`shape`
rang	`size(shape)`	`ndim`
redimensionnement	`reshape`	`reshape`
somme	`sum`	`sum`
produit	`product`	`product`, `prod`
transposition	`transpose`	`transpose`
valeur minimale	`minval`	`min`
valeur maximale	`maxval`	`max`
position du min	`minloc`	`argmin`
position du max	`maxloc`	`argmax`
et logique	`all`	`all`
ou logique	`any`	`any`
fusion de 2 tableaux	`merge`	`where`

Annexe F

Compilateurs et options de compilation

Certains compilateurs fortran sont disponibles sur plusieurs plates-formes comme `pgf95` de PORTLAND ou `nagfor` (ex-`f95`) de NAG (The Numerical Algorithms Group). D'autres sont plus spécifiques de certains systèmes propriétaires, comme `xlf` sur AIX d'IBM, ou `f90`, celui de HP issu de COMPAQ... lui-même issu de DIGITAL EQUIPMENT CORPORATION, dont la version pour LINUX sur processeur ALPHA est libre dans le cadre d'une licence particulière. Le compilateur `ifort` d'INTEL sous LINUX (http://www.intel.com/cd/software/products/asmo-na/eng/compilers/flin/index.htm) est aussi utilisable sans frais pour un usage non commercial.

Enfin, deux compilateurs fortran libres (sur le modèle de `g77`) de la collection de compilateurs `gcc` ont été développés :

- `gfortran` (http://gcc.gnu.org/onlinedocs/gcc-4.4.3/gfortran/),
- `g95` (http://www.g95.org) au développement initial plus rapide, mais figé en 2010,

disponibles pour plusieurs plates-formes dont LINUX (sur processeurs 32 bits et 64 bits), WINDOWS et MACINTOSH OS X.

F.1 Compilateur `xlf` sous ibm-aix

Le compilateur fortran des machines IBM fonctionnant sous AIX [1] ou sur processeur POWER-PC sous linux est `xlf` [2] (voir Manuels Fortran IBM). Le compilateur `xlf`, invoqué tel quel, attend du format respectivement fixe ou libre en fonction du suffixe (extension) du nom du fichier source :

- fixe, à la norme fortran 77 avec `.f`
- libre, à la norme déterminée par le suffixe avec `.f90`, `.f95`, `.f03` ou `.f08` (*cf.* 1.4.2, p. 8).

On peut aussi spécifier immédiatement le standard de langage invoqué en choisissant de lancer `xlf90`, `xlf95`, `xlf2003`, tous attendant du format libre par défaut.

```
xlf90 <essai>.f90
```

Des options permettent aussi de spécifier :

- les suffixes des fichiers sources avec par exemple `-qsuffix=f=f90`
- le format du source avec `-qfree` (format libre)

```
xlf -qfree=f90 -qsuffix=f=f90 <essai>.f90
```

F.1.1 Options du compilateur `xlf` conseillées

♥ ⇒ Il est conseillé de demander au compilateur de respecter la norme choisie et de signaler les

1. AIX est l'UNIX d'IBM.
2. En 2017, la version 15 de `xlf` est disponible.

constructions obsolescentes [3] pour cette norme par l'option `-qlanglvl=<niveau>` où `<niveau>` peut valoir : `90pure`, `95pure` ou `2003pure`.

```
xlf -qlanglvl=90pure <essai>.f90
```

De plus, au moins dans la phase de mise au point, on peut lui demander de vérifier notamment l'affectation des variables avant leur utilisation, le non-dépassement des bornes des tableaux par l'option `-qcheck=all` (version longue de l'option `-C`)r.,

Enfin, il est prudent de provoquer une interruption de l'exécution dans les cas où les calculs en virgule flottante aboutissent à des résultats non garantis [4] :

`ENable :` nécessaire pour permettre l'interruption à la suite d'un des problèmes suivants

`INValid :` opération invalide

`OVerflow :` dépassement de capacité vers le haut (quantité trop grande en valeur absolue pour être représentée sur la machine, car l'exposant serait trop grand et positif)

`ZEROdivide :` division par zéro

Cette précaution est assurée en compilant avec l'option `-qflttrap`, suivie des sous-options nécessaires : `xlf90 -qflttrap=en:inv:ov:zero`

Enfin, l'option `-qinit=f90ptr` permet d'imposer aux pointeurs un état non associé à leur création, au lieu de l'état indéterminé (obtenu par défaut).

F.1.2 Autres options du compilateur `xlf`

L'option `-F fichier.cfg` permet de spécifier un fichier de configuration personnel (inspiré de `/etc/xlf.cfg` qui représente la configuration par défaut) qui rassemble les options choisies.

Noter enfin l'option `-qxlf2003=autorealloc` qui permet d'inhiber l'allocation automatique (en fortran 2003 et au-delà) des tableaux allouables lors de l'affectation (cf. 7.5.4, p. 94).

F.2 Compilateur `f95` ou `nagfor` de NAG

Le compilateur du Numerical Algorithms Group s'appelait `f95` mais comme le compilateur `gfortran` est aussi appelé `f95`, à partir de la version 5.2, il s'appelle maintenant `nagfor` (voir Manuels Fortran NAG). La version 5.3 met en œuvre l'intégralité du standard 95, presque tout le fortran 2003 et quelques éléments du standard 2008.

Par défaut, `nagfor` considère les fichiers sources de suffixes `.f90`, `.f95` et leurs variantes majuscules comme écrits au format libre (*cf.* 1.4.2, p. 8), alors que ceux de suffixes `.f`, `.for` `.ftn` et leurs variantes majuscules sont supposés au format fixe (*cf.* 1.4.1, p. 7). Les options `-free` et `-fixed` permettent de spécifier le type de format indépendamment de ces conventions. Dans le cas du format fixe, on peut augmenter la longueur maximale des lignes de 72 à 132 caractères avec l'option `-132`.

Les fichiers sources de suffixe en majuscules sont par défaut traités par le préprocesseur `fpp`, alors que ceux dont le suffixe est en minuscules ne le sont pas par défaut.

Les options `-f95`, `-f2003` et `-f2008` définissent le standard de référence (fortran 2008 par défaut). Les fonctionnalités de niveau supérieur au standard choisi sont simplement signalées comme extensions par des avertissements du compilateur.

L'option `-kind=` permet de spécifier la numérotation des variantes de type pour les entiers, les booléens et les flottants :

`-kind=sequential` (par défaut) les numérote en séquence à partir de 1 par taille croissante ;

`-kind=byte` les numérote avec le nombre d'octets utilisés pour le stockage.

3. Les options de type `-qlanglvl=90std` seraient suffisantes pour exiger la norme en acceptant les instructions obsolescentes.

4. Les noms des sous-options peuvent être abrégés en ne conservant que la partie du mot-clef écrite ici en majuscules.

Un module `f90_kind.f90` fournit un ensemble de variantes de type (`KIND`) prédéfinis.

L'option `-nomod` empêche la production du fichier de module de suffixe `.mod`.

L'option `-convert=`, suivie du format choisi, permet de préciser la conversion à effectuer sur les fichiers binaires (*cf.* note 10, p. 40) ; mais ces choix peuvent se faire plus finement, fichier par fichier, avec les options de `OPEN`, ou, à l'exécution, via des variables d'environnement du type `FORT_CONVERT`*n*, où *n* est le numéro de l'unité logique, comme pour `gfortran` (*cf.* F.5.2, p. 173) et `ifort`.

Avec la version 5.3 est apparue l'option `-encoding=` qui permet de préciser le codage du fichier source, parmi `ISO_Latin_1` (par défaut), `Shift_JIS` et `UTF_8`. Ne pas confondre avec l'option `encoding` de `OPEN` pour les entrées-sorties sur fichiers texte externes (*cf.* 5.3.2, p. 44) ni avec les variantes de type pour les variables de type chaîne de caractères (*cf.* 8.1.3, p. 97).

F.2.1 Options du compilateur `nagfor` conseillées

♥ ⇒ Les options suivantes sont conseillées pour forcer des vérifications complémentaires, au moins dans la phase de mise au point des programmes :

`-C=array` signale les dépassements de bornes des tableaux et des chaînes de caractères ;

`-C=calls` vérifie les références aux procédures ;

`-C=dangling` signale les pointeurs indéfinis ;

`-C=do` signale les boucles `do` dont le pas est nul ;

`-C=recursion` signale les récursivités invalides ;

`-nan` initialise toutes les variables (locales y compris allouables, de module, arguments muets à vocation de sortie) de type `REAL` ou `COMPLEX` à `NaN` (Not A Number), ce qui provoque un arrêt si elles sont utilisées avant d'être définies.

Cet ensemble d'options peut être sélectionné globalement avec la syntaxe `-C=all`, à laquelle il est prudent d'ajouter `-C=undefined` [5] pour rechercher les variables non définies.

L'option `-float-store` interdit l'utilisation de registres de capacité supérieure à 64 bits pour les calculs flottants : en effet l'unité de calcul flottant (Floating Processor Unit) effectue bien souvent les calculs dans des registres de 80 voire 128 bits permettant une précision et un domaine étendus pour représenter les résultats intermédiaires. Cette option ne doit être activée qu'à titre de test car elle dégrade les performances des programmes même si elle contribue à leur portabilité !

Enfin l'option `-gline` permet de retracer l'origine d'une erreur à l'exécution en indiquant les lignes de code [6] où elle s'est produite. Comme cette option est consommatrice de temps et de place, on ne l'activera qu'en cas d'erreur à l'exécution et on relancera le programme ainsi recompilé pour analyser l'erreur.

F.3 Compilateur `pgf95` de Portland

Le groupe Portland commercialise un ensemble de compilateurs C, C++ et fortran ainsi que des outils de développement et de calcul parallèle associés pour diverses plates-formes et systèmes d'exploitation. Sous LINUX, avec des processeurs AMD64 ou IA32/EM64T, on peut utiliser le compilateur `pgf95` (voir Manuels Fortran PGI).

Par défaut, les fichiers sources de suffixes `.f90`, `.f95` et `.f03` sont considérés comme écrits au format libre (*cf.* 1.4.2, p. 8), alors que ceux de suffixes `.f`, `.for` et `.ftn` sont supposés au format fixe (*cf.* 1.4.1, p. 7). Les options `-Mfreeform` et `-Mnofreeform` permettent de spécifier le type de format indépendamment de ces conventions. Enfin, l'option `-Mnomain` spécifie lors de l'édition de lien que le programme principal n'est pas en fortran (*cf.* chapitre 12, p. 131).

5. Une procédure compilée avec l'option `-C=undefined` peut ne pas être compatible avec une procédure compilée sans cette option. De plus, cette option est incompatible avec les fonctions à résultat allouable.
6. En remontant dans la hiérarchie des appels de procédures.

F.3.1 Options du compilateur pgf95 conseillées

Pour compiler des procédures récursives, il est nécessaire de préciser l'option `-Mrecursive`, car l'option par défaut est `-Mnorecursive`. Par défaut, les variables locales ne sont pas sauvegardées d'un appel à l'autre (option `-Mnosave`). Le stockage statique des variables locales peut être évité avec l'option `-Mrecursive`.

Les options suivantes sont conseillées pour signaler aider à écrire un code robuste, au moins dans la phase de mise au point des programmes : ⇐♥

`-Mdclchk` requiert que toutes les variables soient déclarées (l'option par défaut est `-Mnodclchk`) ;

`-Mstandard` demande au compilateur de signaler les écarts à la syntaxe du fortran standard selon la norme choisie ;

`-Minform=inform` choisit le niveau maximum d'information (erreurs, avertissements et informations)

`-Mbounds` signale les dépassements de bornes des tableaux et des chaînes de caractères.

De plus, l'option `-Mallocatable=95` ou `-Mallocatable=03` permet de choisir le comportement des affectations de variables allouables : allocation préalable nécessaire en fortran 95 ou (ré-)allocation implicite en fortran 2003 (*cf.* 7.5.4, p. 94).

Par ailleurs, l'option `-Mbyteswapio` permet d'intervertir l'ordre des octets dans les entrées-sorties sur des fichiers binaires (conversion litte endian vers big-endian et réciproquement) (*cf.* note 10, p. 40).

Enfin l'option `-Ktrap=fp` force l'interruption en cas de problème d'opérations en flottant (division par zéro, opération invalide, dépassement de capacité) à l'exécution. On peut aussi activer l'option `-Kieee` pour respecter la norme IEEE 754 (*cf.* annexe C, p. 156) lors des calculs en flottant, au prix éventuel d'un moins bon temps de calcul.

F.4 Compilateur ifort d'Intel

Sous LINUX, avec des processeurs INTEL ou compatibles, on peut utiliser le compilateur `ifort` (voir Manuels Fortran Intel), disponible gratuitement sous conditions pour un usage non commercial.

Par défaut, les fichiers sources de suffixes[7] `.f90` sont considérés comme écrits au format libre (*cf.* 1.4.2, p. 8), alors que ceux de suffixes `.f`, `.for` et `.ftn` sont supposés au format fixe (*cf.* 1.4.1, p. 7). Les options `-free` et `-fixed` permettent de spécifier le type de format indépendamment de ces conventions. Les options `-i8` et `-r8` permettent de promouvoir respectivement les variables entières par défaut en 64 bits et les variables réelles par défaut en double précision (64 bits). De plus l'option `-fpconstant` permet de promouvoir les constantes réelles par défaut en double précision si elles sont affectées à des variables double précision. Enfin, l'option `-nofor-main` spécifie lors de l'édition de lien que le programme principal n'est pas en fortran (*cf.* chapitre 12, p. 131).

F.4.1 Options du compilateur ifort conseillées

Le stockage statique des variables locales peut être évité avec l'option `-automatic`, `-nosave` ou `-auto`. Pour compiler des procédures récursives, il est nécessaire de préciser l'option `-recursive` (qui implique `-automatic`), car l'option par défaut est `-norecursive`.

Les options suivantes sont conseillées pour signaler des erreurs à l'exécution, au moins dans la phase de mise au point des programmes : ⇐♥

`-stand f90` ou `-stand f95` ou `-stand f03` ou `-stand f08` demande au compilateur de signaler tous les écarts à la syntaxe du fortran standard selon la norme choisie ;

Noter à ce propos que malgré le choix de la norme 2003, le compilateur `ifort` ne met pas en œuvre l'allocation automatique par affectation (*cf.* 7.5.4, p. 94) : il faut ajouter l'option `-assume realloc_lhs`. ⇐⚠

7. Contrairement à d'autres compilateurs, `ifort` n'admet ni le suffixe `.f95` ni le suffixe `.f03` pour des sources au standard fortran 95 ou 2003.

`-warn declarations` ou `-implicitnone` signale les identifiants utilisés sans être déclarés (typage implicite) ;

`-warn uncalled` signale les fonctions qui ne sont jamais appelées ;

`-warn unused` signale les variables déclarées mais jamais utilisées ;

`-warn all` demande d'émettre tous les avertissements que le compilateur peut diagnostiquer (cela implique notamment `-warn declarations`, `-warn uncalled`, `-warn unused`, ...)

`-check bounds` ou `-CB` signale les dépassements de bornes des tableaux et des chaînes de caractères ;

`-check format` détecte une inconsistance entre une valeur et son format de sortie ;

`-check all` ou `-C` permet de spécifier globalement toutes les vérifications du type `-check`

`-diag-error-limit1` permet d'arrêter la compilation à la première erreur.

`-traceback` ajoute dans le code objet des informations complémentaires permettant, en cas d'erreur grave, de retrouver la ligne de code, la procédure et le fichier source associés afin de localiser l'origine de l'erreur.

`-assume byterecl` permet d'exprimer les longueurs des enregistrements (argument `RECL` de `OPEN` et `INQUIRE`) des fichiers non-formatés en octets, au lieu des mots longs de 4 octets par défaut.

Enfin l'option `-ftrapuv` qui initialise les variables allouées sur la pile à des valeurs invalides peut permettre de détecter des défauts d'initialisation.

D'autre part, l'option `-fltconsistency`, ou `-mieee-fp` requiert des calculs en virgule flottante plus portables et conformes à la norme IEEE (*cf.* exemple C.3.1, p. 158), au prix d'une perte de performances, de la même façon que l'option `-ffloat-store` de `gfortran`. Elle limite par exemple certaines optimisations du compilateur affectant l'ordre des opérations et respecte la précision des calculs associé au type des variables.

Par ailleurs, l'alignement des types dérivés (sans attribut `SEQUENCE` ou `BIND`, *cf.* 9.1.1, p. 105) par des octets de remplissage est conditionné par les options :

`-noalign records` pour éviter le remplissage ;

`-align records` pour forcer l'alignement sur des frontières de mots ;

`-align rec`***n***`byte` pour aligner sur un nombre ***n*** d'octets, où ***n*** peut valoir 1, 2, 4, 8 ou 16.

Des variables d'environnement permettent de spécifier le contexte d'exécution des programmes compilés avec `ifort`. Parmi celles-ci, on notera celle qui permet de désigner l'ordre des octets (*cf.* note 10, p. 40) dans la représentation des données binaires : `F_UFMTENDIAN`. Elle permet d'indiquer (sous diverses formes) la liste des numéros d'unités logiques pour lesquels les entrées/sorties se feront avec conversion entre `little` et `big`, sachant que le format par défaut est `little` sur les processeurs où ce compilateur fonctionne.

F.5 Compilateur `gfortran`

Le compilateur `gfortran` fait partie de `gcc`[8], la collection de compilateurs du `GNU`, capable de compiler les langages C, C++, Objective-C, Fortran, Java, et Ada. Plus précisément, il s'appuie sur `gcc`, suit sa syntaxe et admet donc ses options générales, mais propose aussi quelques options spécifiques pour le langage fortran.

Les deux compilateurs `gfortran` et `g95` proposent des fonctionnalités similaires. que l'on décrira essentiellement pour `gfortran`. Le lecteur pourra se reporter à la documentation en ligne (Site gfortran, de la collection de compilateurs gcc) pour plus de détails.

F.5.1 Nommage des fichiers sources pour `gfortran`

Par défaut, les fichiers sources de suffixes `.f90`, `.f95`, `.f03`, `.f08` et leurs variantes majuscules sont considérés comme écrits au format libre (*cf.* 1.4.2, p. 8), alors que ceux de suffixes `.f`, `.for`,

8. En fait, c'est le compilateur fortran 90 officiel de `gcc` et son développement continue, contrairement à celui de `g95` (*cf.* F.6, p. 175). Il est maintenant distribué avec les outils de développement dans la plupart des distributions linux (`http://gcc.gnu.org/wiki/GFortranBinaries`). Noter qu'il est parfois accessible sous le nom `f95`.

`.ftn` et leurs variantes majuscules sont supposés au format fixe (*cf.* 1.4.1, p. 7). Les fichiers sources de suffixe en majuscules sont par défaut traités par le préprocesseur du C, alors que ceux dont le suffixe est en minuscules ne le sont pas par défaut.

Les options `-ffree-form` et `-ffixed-form` permettent de spécifier le type de format indépendamment de ces conventions. On peut préciser la longueur maximale ***n*** des lignes d'instructions avec les options `-ffixed-line-length-`***n*** (72 par défaut (*cf.* 1.4.1, p. 7), 0 signifiant non limitée) en format fixe, ou `-ffree-line-length-`***n*** (132 par défaut (*cf.* 1.4.2, p. 8), 0 signifiant non limitée) en format libre.

F.5.2 Options du compilateur gfortran conseillées

- Par défaut, les tableaux locaux sans l'attribut `SAVE` des procédures ne sont pas stockés en statique mais sur la pile, sauf s'ils dépassent une taille spécifiée en octets par l'option `-fmax-stack-var-size=`***n*** (défaut 32768). L'option `-frecursive` empêche leur stockage en mémoire statique.
- L'option `-fimplicit-none` interdit le typage implicite des variables comme le fait l'instruction `IMPLICIT NONE`.
- L'option `-std=f95`, `-std=f2003` ou `-std=f2008` permet d'exiger un code conforme à la norme fortran 95, fortran 2003 ou fortran 2008. Les écarts à la norme requise génèrent des erreurs et les instructions obsolescentes sont signalées par des avertissements. Cette option exclut l'emploi des procédures intrinsèques non standard, sauf si on ajoute l'option `-fall-intrinsics` [9] qui autorise toutes les procédures intrinsèques qui sont des extensions à la norme (comme `GAMMA`, `ERF`, `ERFC`... qui font partie du standard fortran 2008). Par ailleurs, à partir de la version 4.6, `gfortran` permet l'allocation dynamique implicite par affectation (*cf.* 7.5.4, p. 94) si le standard demandé est `2003` ou `2008`, mais il est possible de l'inhiber avec l'option `-fno-realloc-lhs` ou de l'autoriser en standard `95` avec `-frealloc-lhs`.

  ⇐⚠ ⇐ f2003
- L'option `-fmax-errors=`***n*** peut être activée pour arrêter la compilation dès la n^{e} erreur, évitant ainsi l'avalanche de messages concernant les nombreuses erreurs provoquées par exemple par une déclaration inexacte.
- L'option `-fbacktrace` permet d'afficher la hiérarchie des appels avec les numéros de ligne en cas d'erreur à l'exécution.
- Les options suivantes initialisent les variables locales non allouables [10] : respectivement les entiers `-finit-integer=`***n***, les réels et les complexes `-finit-real=nan`, par exemple à Not a Number de signalisation (*cf.* C.1.2, p. 157).

L'option `-ffloat-store` de `gcc` interdit l'utilisation de registres de capacité supérieure pour les calculs flottants (*cf.* exemple C.3.1, p. 158) : en effet l'unité de calcul flottant (Floating Processor Unit) effectue bien souvent les calculs dans des registres de 80 voire 128 bits permettant une précision et un domaine étendus pour représenter les résultats intermédiaires. Cette option ne doit être activée que si nécessaire car elle dégrade les performances des programmes même si elle contribue à leur portabilité ! Elle a le même effet qu'avec `g95` (*cf.* F.6.1).

Pour la mise au point, `-fbounds-check` [11] ou (à partir de la version 4.5) `-fcheck=bounds` active la vérification des bornes des tableaux à l'exécution.

F.5.3 Avertissements de `gfortran` à la compilation

On conseille, pour la compilation, les options `-Wall` qui requièrent des avertissements classiques et `-Wextra` pour des avertissements complémentaires.

L'option `-Wline-truncation` avertit quand une ligne de code source dépasse la longueur autorisée qui correspond au standard (par défaut 132 caractères en format libre). Elle est activée par `-Wall` et, en format libre, déclenche même une erreur.

9. Avec `g95`, il est possible de spécifier la liste des procédures non standard autorisées, *cf.* F.6.1, p. 175.
10. Pour le moment, elles n'initialisent pas les composantes de types dérivés.
11. À partir de la version 4.5 `-fbounds-check` est un alias obsolescent de `-fcheck=bounds`.

L'option `-Wcharacter-truncation` permet de détecter la troncature d'une expression de type chaîne de caractères trop longue pour être affectée à une variable.

Noter que l'option `-Wconversion`, qui signale les conversions *implicites* induites par les opérateurs (*cf.* 3.1.3, p. 23), peut s'avérer trop bavarde pour être utilisée systématiquement [12], si on ne s'impose pas de rendre toutes ces conversions explicites pour éviter d'alourdir les codes. On peut l'ajouter ponctuellement pour localiser les conversions implicites dans des codes sensibles aux erreurs numériques de conversion.

L'option `-Wintrinsic-shadow` permet d'avertir si une procédure définie par l'utilisateur peut être masquée par une procédure intrinsèque de même nom.

F.5.4 Options de `gfortran` concernant l'édition de liens

Si l'on dispose d'une version optimisée de la bibliothèque BLAS (Basic Linear Algebra Subroutines), comme par exemple OpenBlas (*cf.* http://xianyi.github.com/OpenBLAS/), il est possible de déléguer les calculs de `MATMUL` à cette bibliothèque dans le cas des tableaux de taille importante, avec l'option `-fexternal-blas`. Le compilateur génère alors un appel à BLAS si la taille des matrices est supérieure ou égale à 30 par défaut, seuil qui peut être modifié via l'option `-fblas-matmul-limit=`*n*.

L'option `-static` impose une édition de liens statique, pour produire un exécutable autonome ; en effet, de nombreux environnements fonctionnent aujourd'hui par défaut avec des bibliothèques dynamiques partageables. S'il s'agit simplement de produire un exécutable «embarquant» la version statique des objets issus de la bibliothèque du fortran, on peut se contenter de l'option `-static-libgfortran`, qui produit cependant un exécutable lié dynamiquement, mais ne requiert plus la bibliothèque du fortran au chargement, comme on peut le vérifier avec `ldd`.

F.5.5 Options de `gfortran` concernant les modules

Avec l'option `-J`, suivie du chemin d'un répertoire, il est possible de fixer le répertoire où seront créés les fichiers `.mod` de module lors de la compilation. On pourra l'associer avec l'option `-I` qui permet d'ajouter un répertoire [13] à la liste des chemins de recherche des fichiers de module requis par l'instruction `USE`. C'est une option nécessaire pour utiliser les fichiers de module d'une bibliothèque fortran installée en dehors des répertoires standard.

Noter que les fichiers de module de `gfortran` comportent un numéro de version lié à (mais différent de) la version du compilateur. De plus, à partir de la version 4.9, les fichiers de module de `gfortran` sont compressés avec `gzip`.

F.5.6 Autres options de gfortran

Il est possible de limiter l'adjonction d'octets de remplissage destinés à respecter des contraintes d'alignement dans les types dérivés (*cf.* 9.1.1, p 105) avec l'option `-fpack-derived`.

L'option `-frecord-marker=<`*length*`>` permet de choisir la taille en octets de l'entier qui permet de stocker la balise d'enregistrement (record marker) indiquant la taille des enregistrements dans les fichiers non-formattés (*cf.* 5.2.3, p. 40). La valeur par défaut pour `gfortran` est de 4 signifiant 32 bits. On peut la passer à 8 pour gérer des enregistrements de plus de 2 Go.

F.5.7 Variables d'environnement associées à `gfortran`

On notera que les variables d'environnement qui permettent d'ajuster le comportement de l'exécutable sont ici préfixées par `GFORTRAN_` au lieu de `G95_`. De plus, le choix de l'ordre des octets dans les fichiers binaires (*cf.* note 10, p. 40) peut se faire unité logique par unité logique grâce à la variable `GFORTRAN_CONVERT_UNIT`, comme avec `ifort` (*cf.* F.4.1, p. 171).

12. C'est pourquoi les versions récentes de `gfortran` ont restreint la portée de `-Wconversion` aux conversions potentiellement dégradantes ; une option supplémentaire, `-Wconversion-extra` a été introduite pour signaler les conversions moins suspectes.

13. L'option `-I` permet aussi de compléter la liste des chemins de recherche des fichiers d'entête pour le C.

F.6 Compilateur g95

Le compilateur g95 (voir Vaught, 2009) est aussi issu de gcc [14], la collection de compilateurs du GNU, capable de compiler les langages C, C++, Objective-C, Fortran, Java, et Ada. Plus précisément, il s'appuie sur gcc, suit sa syntaxe et admet donc ses options générales, mais propose aussi des options spécifiques pour le langage fortran.

Par défaut, les fichiers sources de suffixes .f90, .f95, .f03 et leurs variantes majuscules sont considérés comme écrits au format libre (*cf.* 1.4.2, p. 8), alors que ceux de suffixes .f, .for et leurs variantes majuscules sont supposés au format fixe (*cf.* 1.4.1, p. 7). Les options -ffree-form et -ffixed-form permettent de spécifier le type de format indépendamment de ces conventions. Dans le cas du format fixe, on peut préciser la longueur maximale des lignes avec les options -ffixed-line-length-80 ou -ffixed-line-length-132.

Les fichiers sources de suffixe en majuscules sont par défaut traités par le préprocesseur du C, alors que ceux dont le suffixe est en minuscules ne le sont pas par défaut.

Avec l'option -fmod=, il est possible de fixer le répertoire où seront créés les fichiers .mod de module lors de la compilation. On pourra l'associer avec l'option -I qui permet d'ajouter un répertoire au chemin de recherche des fichiers de module.

L'option -M permet l'affichage des dépendances des fichiers objets et fichiers de module, en vue de la rédaction du fichier makefile :
g95 -M <*fichier.f90*> affiche les dépendances du fichier <*fichier.o*>.

Enfin, une liste de variables d'environnement (préfixées par G95_) permet d'adapter l'environnement d'exécution des programmes compilés par g95. L'avantage de tels réglages par rapport aux options de compilation est qu'ils peuvent être modifiés une fois l'exécutable créé et ajustés *a posteriori* lors de l'exploitation. On peut connaître les valeurs de ces variables d'environnement en lançant l'exécutable produit par g95 avec l'option --g95. Ces variables concernent notamment les entrées/sorties, avec, entre autres (*cf.* 5.2.1, p. 39) :

- G95_STDIN_UNIT de type entier qui permet de choisir le numéro de l'unité logique préconnectée à l'entrée standard ;
- G95_STDOUT_UNIT de type entier qui permet de choisir le numéro de l'unité logique préconnectée à la sortie standard ;
- G95_STDERR_UNIT de type entier qui permet de choisir le numéro de l'unité logique préconnectée à la sortie d'erreur standard ;
- G95_COMMA de type booléen qui permet de choisir la virgule comme séparateur décimal (*cf.* 5.3.2, p. 44) ;
- G95_ENDIAN de type chaîne de caractères et de valeurs possibles BIG, "LITTLE", "SWAP" ou "NATIVE" (par défaut), qui permet de choisir l'ordre des octets (*cf.* note 10, p. 40) dans la représentation des données pour les lectures et écritures de fichiers binaires.

F.6.1 Options du compilateur g95 conseillées

- Le stockage statique des variables locales peut être évité avec l'option -fno-static.
- Il est possible de demander au compilateur d'interdire tous les écarts à la syntaxe du fortran standard avec l'option -std=f95 [15] ou -std=f2003. Cette option exclut l'emploi des procédures intrinsèques non standard, telles que la fonction system qui permet de lancer des commandes du système d'exploitation depuis le fortran. Mais des options [16] sont prévues pour autoriser toutes ou partie seulement des extensions intrinsèques tout en respectant le standard par ailleurs. ⇐⚠

14. En fait, le développement d'un compilateur fortran 90 au sein de gcc continue sous le projet gfortran (*cf.* F.5, p. 172). Développé en parallèle par un des anciens membres de l'équipe de gfortran, le compilateur g95 avait abouti plus rapidement, mais semble maintenant figé depuis août 2010.

15. On peut cependant dans ce cas autoriser certaines extensions relevant du standard 2003. En particulier l'allocation de tableaux dynamiques arguments de procédures (*cf.* 7.5.3, p. 93) et les tableaux allouables dans les structures (*cf.* 9.5, p. 108) en ajoutant l'option -ftr15581.

16. L'option -fintrinsic-extensions permet d'accéder à toutes les fonctions intrinsèques de g95 alors que -fintrinsic-extensions= permet de spécifier la liste explicite des fonctions intrinsèques non-standard (par exemple les fonctions GAMMA, ERF, ERFC... et aussi la fonction system) à accepter en exigeant par ailleurs le respect d'un standard.

– L'option `-Wimplicit-none` permet de signaler toutes les variables non déclarées. L'option `-fimplicit-none` interdit l'usage de variables non déclarées.
– Si on souhaite être informé qu'une variable a été déclarée, mais n'est pas utilisée [17], il faut préciser l'option `-Wunused`.
– À l'inverse, l'option `-Wunset` avertit quand on n'attribue pas de valeur à une variable, ce qui est suspect si elle est utilisée.

Afin de bénéficier au mieux de l'analyse du code faite par le compilateur, il est utile de lui demander la plupart des avertissements disponibles : on utilise à cet effet l'option `-Wall` qui regroupe les avertissements plus importants et on peut ajouter `-Wextra` pour activer notamment `-Wmissing-intent` et `-Wobsolescent`.

♥ ⇒ Les options suivantes sont conseillées pour signaler des erreurs à l'exécution, au moins dans la phase de mise au point des programmes :

`-fbounds-check` signale les dépassements de bornes des tableaux et des chaînes de caractères ;

`-ftrapv` provoque une erreur lors d'un dépassement de capacité dans les opérations d'addition, de soustraction et de multiplication.

`-freal=nan` initialise les variables scalaires réelles et complexes (non explicitement initialisées) à **n**ot **a** **n**umber, produisant des erreurs de calcul si on les utilise sans leur attribuer d'autre valeur.

`-fpointer=null` initialise les pointeurs scalaires (non explicitement initialisés) à `null()`

`-ftrace=full` permet retrouver le numéro de ligne d'une éventuelle erreur arithmétique, mais ralentit l'exécution.

`-ffloat-store` interdit l'utilisation de registres de capacité supérieure pour les calculs flottants : elle a le même effet qu'avec `gfortran` (*cf.* F.5.2).

`-fone-error` peut être activée pour arrêter la compilation dès la première erreur, évitant ainsi l'avalanche de messages concernant les erreurs provoquées par exemple par une déclaration inexacte.

`G95_MEM_INIT="NAN"` cette variable d'environnement, par défaut à `"NONE"`, permet de choisir d'initialiser la mémoire allouée de façon dynamique, par exemple à `Not A Number`, quitte à ralentir l'allocation.

Pour plus de détails, consulter le manuel du compilateur, maintenant disponible en ligne (Vaught, 2006), y compris traduit en français (`http://ftp.g95.org/G95Manual.fr.pdf`).

F.7 Options de compilation contrôlant le nombre d'octets des types numériques intrinsèques

F.7.1 Options passant sur 64 bits les entiers par défaut

processeur	compilateur	option
32 bits	`g95`	`-i8`/`-d8`
64 bits	`g95`	par défaut
32/64 bits	`gfortran`	`-fdefault-integer-8`
32/64 bits	`nagfor` (NAG)	`-double`
32/64 bits	`pgf95` (PORTLAND)	`-i8`
32/64 bits	`ifort` (INTEL)	`-integer_size 64` ou `-i8`
64 bits (alpha)	`f90` (HP)	`-integer_size 64`
64 bits (power PC)	`xlf` (IBM)	`-qintsize=8`

17. Ces avertissements peuvent paraître au premier abord superflus, en particulier dans la phase de mise au point du code. Mais ils peuvent permettre de découvrir des erreurs de codage de façon assez subtile, comme dans cet exemple, vécu par un étudiant : en signalant que, dans l'expression `x**(1/3)`, la variable `x` est en fait inutilisée, le compilateur `g95` a permis de détecter une grossière erreur de codage dans le calcul de la racine cubique. La division `1/3` étant ici programmée en entier, elle donne un exposant nul, rendant l'expression indépendante de `x` car $x^0 = 1$.

F.7.2 Options passant sur 64 bits les réels par défaut

processeur	compilateur	option
32/64 bits	`g95`	`-r8/-d8`
32/64 bits	`gfortran`	`-fdefault-real-8`
32/64 bits	`nagfor` (NAG)	`-r8/-double`
32/64 bits	`pgf95` (PORTLAND)	`-r8`
32/64 bits	`ifort` (INTEL)	`-real_size 64` ou `-r8`
64 bits (alpha)	`f90` (HP)	`-real_size 64`
64 bits (power PC)	`xlf` (IBM)	`-qrealsize=8`

F.8 Équivalences approximatives des options entre les compilateurs

Ce tableau récapitule les principales options dont les noms diffèrent selon les compilateurs. Il ne s'agit que d'un aide-mémoire grossièrement simplifié pouvant aider à migrer des applications d'un compilateur à un autre : les correspondances sont parfois assez approximatives et on se référera aux manuels respectifs pour s'assurer dans le détail de l'effet de chaque option.

gfortran	g95	ifort	nagfor	pgfortran
`-ffloat-store`	`-ffloat-store`	`-fltconsistency`		`-Kieee`
`-fmax-errors=1`	`-fone-error`	`-diag-error-limit1`		
`-std=f95`	`-std=f95`	`-std f95`	`-f95`	
`-std=f2003`	`-std=f2003`	`-std f03`	`-f2003`	
`-realloc-lhs`		`-assume realloc_lhs`		`-Mallocatable=03`
`-fckeck=bounds`	`-fbound-checks`	`-check bounds`	`-C=array`	`-Mbounds`
`-fimplicit-none`	`-fimplicit-none`	`-implicitnone`	`-u`	`-Mdclchk`
		`-nofor-main`		`-Mnomain`
`-static`		`-Bstatic`	`-Bstatic`	`-Bstatic`

Bibliographie

Adams, Jeanne, Walter Brainerd, Richard Hendrickson, Richard Maine, Jeanne Martin et Brain Smith, *The Fortran 2003 Handbook : The Complete Syntax, Features and Procedures*, 712 pages (Springer, 2009), ISBN 978-1846283789. F.8

Akin, Ed, *Object-oriented programming via fortran 90/95*, 360 pages (Cambridge University Press, 2003), ISBN 0-521-52408-3.

Cet ouvrage présente les aspects programmation orientée objet du fortran 90/95 au travers de nombreux exemples commentés. Il comporte des tableaux synthétiques mettant en parallèle les syntaxes du fortran, du C++ et de matlab ainsi qu'une annexe résumant succintement le langage fortran 90.

Brainerd, Walter, *Guide to Fortran 2008 Programming*, 408 pages (Springer, 2015), ISBN 978-1-4471-6758-7.

Rédigée par un des co-auteurs du Handbook de Adams *et al.* (2009), cet ouvrage constitue un tutoriel très pédagogique du fortran, qui fait le choix (encore rare) de présenter directement la norme 2008 du fortran. Riche en exemples, études de cas et exercices, il comporte notamment un chapitre sur les co-tableaux (coarrays) et une annexe sur les procédures intrinsèques..

Chapman, Stephen J., *Fortran 95/2003 for Scientists and Engineers*, 974 pages (Mc Graw-Hill, 2007), 3e édition, ISBN 978-0-07-319157-7.

La troisième édition de ce livre est un des rares ouvrages qui présente délibérément le fortran du standard 2003, tout en distinguant explicitement par la présentation les éléments non disponibles dans le standard fortran 95. Toutefois, l'interopérabilité avec le C n'est pas abordée. Contrairement à Metcalf *et al.* (2004), il peut être abordé par des débutants car l'approche est progressive et assortie de nombreux conseils, exemples et exercices.

Chivers, Ian et Jane Sleightholme, *Introduction to Programming with Fortran With Coverage of Fortran 90, 95, 2003, 2008 and 77*, 619 pages (Springer, 2012), 2e édition, ISBN 978-0-85729-232-2.

Clerman, Norman S. et Walter Spector, *Modern fortran : Style and Usage*, 360 pages (Cambridge University Press, 2011), ISBN 978-0521514538.

Comme son titre l'indique, cet ouvrage présente des règles de «bon usage» du fortran 2003 aussi bien dans l'écriture du code (nommage, mise en page, ...) que dans sa documentation. Il s'adresse donc à un public possédant déjà une certaine expérience du langage.

1.3.2, 1.4.2

Corde, Patrick et Hervé Delouis, *Langage Fortran (F2003)*, Institut du Développement et des Ressources en Informatique Scientifique (IDRIS) – CNRS, 2017a, URL : http://www.idris.fr/media/formations/fortran/idris_fortran_2003_cours.pdf.

Corde, Patrick et Hervé Delouis, *Langage Fortran (F95-2)*, Institut du Développement et des Ressources en Informatique Scientifique (IDRIS) – CNRS, 2017b, URL : http://www.idris.fr/media/formations/fortran/idris_fortran_avance_cours.pdf.

DELANNOY, CLAUDE, *Programmer en Fortran 90 Guide complet*, 413 pages (Eyrolles, 1997), ISBN 2-212-08982-1.

Écrit par un auteur bien connu pour le succès de ses nombreux ouvrages sur les langages de programmation, ce livre présente le fortran 90 de façon progressive et concise. Il est accompagné d'exemples et d'exercices corrigés, et peut constituer un très bon manuel d'apprentissage. Depuis 2015, on lui préférera la deuxième édition (DELANNOY (2015)) qui couvre une partie des standards 2003 et 2008.

F.8

DELANNOY, CLAUDE, *Programmer en Fortran : Fortran 90 et ses évolutions – Fortran 95, 2003 et 2008*, 480 pages (Eyrolles, 2015), deuxième édition, ISBN 978-2-212-14020-0.

Deuxième édition de l'ouvrage de 1997 (DELANNOY (1997)), qui s'arrêtait à fortran 95, cet ouvrage aborde maintenant les standards 2003 et 2008 dans son annexe I , ainsi que la programmation orientée objet avec fortran 2003 dans son annexe H.

F.8

DUBESSET, CLAUDE et JEAN VIGNES, *Les spécificités du Fortran 90*, 367 pages (Éditions Technip, 1993), ISBN 2-7108-0652-5.

Cet ouvrage en français s'adresse avant tout à ceux qui connaissent déjà le fortran 77. Il présente avec beaucoup de précision et de clarté les apports de la norme fortran 90. Il est doté de précieuses annexes et notamment d'un lexique et d'un glossaire.

FOUILLOUX, ANNE et PATRICK CORDE, *Langage Fortran (F95-1)*, Institut du Développement et des Ressources en Informatique Scientifique (IDRIS) – CNRS, 2016, URL : http://www.idris.fr/media/formations/fortran/idris_fortran_base_cours.pdf.

LIGNELET, PATRICE, *Manuel complet du langage Fortran 90 et Fortran 95 : calcul intensif et génie logiciel*, 314 pages (Masson, 1996), ISBN 2-225-85229-4.

Au delà du langage fortran, cet ouvrage en français d'un auteur de nombreux ouvrages dans ce domaine, aborde les questions de calcul numérique, de vectorisation et de généricité.

Manuel Fortran DEC, *Digital Fortran, Langage Reference Manual*, Digital Equipment Corporation, Maynard, Massachusetts, USA, 1997.

Le manuel papier du compilateur `f90` de DIGITAL EQUIPMENT CORPORATION, repris successivement par COMPAQ, puis par HP. Très bien présenté, il distingue clairement les extensions propriétaires de la norme du fortran 90.

Manuels Fortran IBM, *XL Fortran for AIX, Langage Reference and User's Guide*, IBM, 8200 Warden Avenue, Markham, Ontario, Canada, 2005, URL : http://publib.boulder.ibm.com/infocenter/comphelp/. F.1

Manuels Fortran Intel, *Intel Fortran Compiler Documentation*, Intel Corporation, 2010, URL : http://software.intel.com/sites/products/documentation/hpc/compilerpro/en-us/fortran/lin/compiler_f/index.htm.

Site d'INTEL qui présente un manuel en ligne du compilateur `ifort`.

F.4

Manuels Fortran NAG, *NAGWare Fortran 95 Compiler*, Numerical Algorithms Group, The Numerical Algorithms Group Ltd, Wilkinson House, Jordan Hill Road, Oxford, OX2 8DR, UK, 2011, URL : http://www.nag.co.uk/nagware/np.asp.

Site du NUMERICAL ALGORITHMS GROUP permettant d'accéder aux pages de manuel en ligne du compilateur `nagfor` (ex `f95`) et des modules associés.

F.2

Manuels Fortran PGI, *PGI Fortran Compiler*, The Portland Group, The Portland Group, STMicroelectronics, Two Centerpointe Drive, Suite 320, Lake Oswego, OR 97035, 2011, URL : http://www.pgroup.com/resources/docs.htm.

Site de The Portland Group, permettant d'accéder au manuel de référence du compilateur fortran PGI au format pdf.

F.3

Markus, Arjen, *Modern fortran in Practice*, 253 pages (Cambridge University Press, 2012), ISBN 978-1-107-60347-9.

Marshall, A.C., J.S. Morgan et J. L. Schofelder, *Fortran 90 Course Notes*, The University of Liverpool, 1997, URL : http://www.liv.ac.uk/HPC/F90page.html.

Le site de l'Université de Liverpool propose une série de cours de différents niveaux sur le fortran 90 : les documents associés (transparents, notes, exercices, codes ...) sont accessibles via `ftp` à l'url ftp://ftp.liv.ac.uk/pub/F90Course/.

Metcalf, Michael, «Fortran 90/95/HPF information file», 2004, URL : http://www.fortran.com/metcalf.htm.

Site dressant un inventaire des ressources sur fortran 90/95 (compilateurs, livres, cours, logiciels, ...). M. Metcalf a maintenu ce site jusqu'en septembre 2004, date à laquelle il a été figé, dans la perspective de la norme fortran 2003.

Metcalf, Michael, John Reid et Malcolm Cohen, *Fortran 95/2003 explained*, 434 pages (Oxford University Press, 2004), 3e édition, ISBN 0-19-852693-8.

Rédigée par des promoteurs des nouveaux standards du fortran, la troisième édition de ce livre est un des rares ouvrages qui aborde les apports du fortran 2003. Il constitue une référence majeure dans ce domaine. Les apports de la norme fortran 2003 y sont présentés dans les 7 derniers chapitres, permettant ainsi de les séparer de la norme fortran 95. Il aborde notamment l'interopérabilité avec le C. Cette référence n'est cependant pas conseillée pour débuter.

F.8

Metcalf, Michael, John Reid et Malcolm Cohen, *Modern Fortran explained*, Numerical Mathematics and Scientific Computation, 488 pages (Oxford University Press, 2011), 4e édition, ISBN 978-019-960142-4.

La quatrième édition de ce classique est le premier ouvrage à aborder la norme 2008 du fortran. Succédant à l'édition Metcalf *et al.* (2004) sur le fortran 2003, cette version révisée comporte notamment un chapitre sur les co-tableaux (coarrays) et un sur les nouveautés du fortran 2008. Cette référence n'est cependant pas conseillée pour débuter et rares sont encore les compilateurs à honorer la norme 2008.

1.1, 4.6

Nyhoff, Larry R. et Sandfird C. Leestma, *Fortran 90 for Engineers and Scientists*, 1070 pages (Prentice-Hall, 1997), ISBN 0-13-6571209-2.

Un ouvrage volumineux mais très bien illustré avec des applications concrètes dans le domaine du calcul scientifique. Sa présentation très progressive permet de l'utiliser pour découvrir le fortran 90 y compris comme premier langage de programmation.

Olagnon, Michel, *Traitement des données numériques avec Fortran 90*, 244 pages (Masson, 1996), ISBN 2-225-85259-6.

Cet ouvrage ne présente pas le langage fortran en tant que tel, mais des méthodes de traitement des données (en particulier statistiques) mises en œuvre en fortran 90. Cette approche pragmatique s'appuyant sur de nombreux exemples permet d'aborder rapidement les applications tout en respectant les règles de bon usage du langage.

Press, William H., Saul A. Teukolsky, William T. Vetterling et Brain P. Flannery, *Numerical Recipes in Fortran 90*, 551 pages (Cambridge University Press, 1996), 2e édition, ISBN 0-521-57439-0.

Une référence très classique en analyse numérique, aussi disponible dans d'autres langages (pascal, C et C++). Bien la distinguer du manuel *Numerical Recipes in Fortran 77* qui comporte la discussion des algorithmes et leur mise en œuvre en fortran 77 et constitue le volume 1 du traité. Le volume 2 présente les codes associés en fortran 90, mais aussi dans son premier chapitre (21) une très précieuse introduction aux fonctionnalités du fortran 90. Noter enfin que les chapitres de cet ouvrage sont consultables en ligne : http://www.library.cornell.edu/nr/

Site gfortran, de la collection de compilateurs gcc, *Documentation de gfortran*, Free Software Foundation, 2011, URL : http://gcc.gnu.org/onlinedocs/gfortran/. A, F.5

Stallings, W., *Organisation et architecture de l'ordinateur*, 829 pages (Pearson Education, 2003), sixième édition, ISBN 9782744070075. 2.1, C.4.2

Vaught, Andy, *Manuel de g95*, 2006, URL : http://ftp.g95.org/G95Manual.pdf. F.6.1

Vaught, Andy, *Documentation de g95*, 2009, URL : http://www.g95.org/docs.html.

Le site du compilateur libre `g95` d'où il est possible de télécharger les binaires pour les différentes architectures. Ce site comporte des liens vers la documentation. Il liste les bibliothèques qui ont été compilées avec succès avec `g95`.

F.6

Vickery, Christopher, «IEEE-754 calculators», 2009, URL : http://babbage.cs.qc.edu/IEEE-754/. C.4.2

wikipedia, «The IEEE-754-2008 standard», 2010, URL : http://en.wikipedia.org/wiki/IEEE_754. C.4.2

Index

– B –

– C –

– D –

– E –

– F –

– J –

– K –

– L –

– M –

– S –

– T –