Fortran スマートプログラミング

— 第３回　もっとスマートなプログラムにしよう —

田口　俊弘
摂南大学理工学部電気電子工学科

第１回で Fortran プログラムの基本的書式とその考え方を示し，第２回でサブルーチンと入出力について説明しました．数値計算や計算機シミュレーション用のプログラムを書くための知識はここまでで十分です．数値計算とは，コンピュータの基本動作である“高速数値演算”と“大容量メモリの読み書き”を利用するのが主目的で，それに“入出力インターフェース”を加えれば処理は完結するからです．しかし，Fortran にはこの他にも様々な文法が用意されています．その中には，プログラムを書く手間を省いたり，メンテナンスを容易にしたり，コンピュータの動作を細かく制御するための書式が含まれています．最終回である今回は，この中から便利なものをいくつか紹介します．例えば，“配列演算”を使えば，数値の集合を一つの変数で代表し，集合の要素間の演算を，集合と集合の演算の形で記述することができます．また，変数宣言の拡張形や，文字列変数などについても紹介します．プログラム作りに慣れてきたら，このようなちょっと進んだ文法を使って，プログラムをさらにスマートにしてみましょう．

また，数値計算をするときには，高速フーリエ変換や多倍長計算など，自分で一から作成するのは難しいプログラムが必要になることがあります．幸いなことに，これらの多くはインターネットからダウンロード可能なサブルーチンライブラリとして公開されているので，最後に Fortran から利用できる筆者がお勧めのライブラリをいくつか紹介します．

1.　知っておくと便利な文法

本節では，覚えておくと便利な書式や，これまでの基本的文法では記述することができない，コンピュータの細かな動作指定について説明します．

1.1　継続行と複文

計算式が非常に長くて，作成に使っているエディタウィンドウの横幅を越えると読みづらくなります．また，コンパイラによっては１行に書ける文字数に制限があるので，そのままではエラーになることもあります．そこで，１行の文を途中で改行して，複数行に分割して書くことができます．このとき，次の行に継続するという意志を示す印として，行末に“&”の文字を書きます．例えば，

```
      print *,alpha,beta,gamma,delta,epsilon,zeta,eta,iota,kappa,lambda,mu
```

という１行は，

```
      print *,alpha,beta,gamma,delta,epsilon &
             ,zeta,eta,iota,kappa,lambda,mu
```

と分割して書くことができます．継続行は何行にわたってもかまいません．

```
      print *,alpha,beta,gamma &
             ,delta,epsilon &
             ,zeta,eta,iota &
             ,kappa,lambda,mu
```

と書いても同じです．最後の行に“&”を付加するとエラーになるので注意して下さい．

逆に，１行に複数の文を書くことも可能です．これを“複文”といいます．２個以上の文を１行で書くには，文と文を分離する記号として“;”の文字を書きます．例えば，

```
x = 1; y = 2; z = 3
```

のように書くことができます．ただし，複文の使用はこのような短い代入文を書く場合に限定して下さい．プログラムは１行で一つの完結した動作を表すのが基本です．１行に複数の文を書くと，動作の流れが見えにくくなるので，多用しない方が良いと思います．

1.2　拡張された宣言文

これまで変数や配列の宣言に使っていた宣言文は，

```
型指定 変数 1，変数 2，...
```

という形式でした．ここで，“型指定”には“integer”とか，“real”のような数値型を記述し，“変数”には変数名か，配列名とその要素数を記述します．この宣言文は，データ領域におけるメモリの確保という意味もあります．

さて，Fortran90 から宣言文の記述形式が拡張されました[1]．拡張宣言文を使うと，メモリを確保すると同時に初期値を代入したり，メモリの特性を指定することができます．拡張宣言文は以下の形式です．

```
型指定 [,属性１，属性２，...] :: 変数１[=数値１],変数２[=数値１],...
```

ここで，角かっこ，“[”と“]”は，この中が省略可能であるという意味で使っているだけなので，角かっこ自体は書かないで下さい．拡張宣言文を使用するときは，型指定部と宣言する変数や配列の間にコロン２個“::”を書く必要があります．また，“属性”には宣言する変数の特性を指定するための予約語を記述します．属性も数値も省略して，単に型指定と変数の間に“::”を書いただけの宣言文は，書かずに宣言した旧来の書式と同じ意味になります．

属性を書かずに，単に数値を代入した形で宣言すると，その変数に指定した数値を代入して，プログラムの動作が開始することを意味します．例えば，

```
integer :: imax=10, jmax=100
real :: xx=1.0, yy=2.0
```

のように宣言すると，それぞれの変数に指定された値を代入した状態で動作が開始します．ただし，この宣言文中での数値代入は一度だけです．サブルーチン中で数値代入した変数を宣言しても，コールするたびに数値が代入されるのではありません．このため，その変数を実行文で変更すると，その変更した結果が残ります．例えば，

```
program stest1
  implicit none
  call subr1
  call subr1
  call subr1
end program stest1

subroutine subr1
  implicit none
  integer :: n=1
  print *,n              ! コールするたびに n は増加する
  n = n + 1
end subroutine subr1
```

というプログラムでは，メインプログラムでサブルーチン subr1 が３回コールされていますが，サブルーチン中の print 文の出力は，１回目が 1，２回目が 2，３回目が 3，となります．宣言文における代入は，プログラム開始時の初期値だと考えて下さい．

さて，上記のサブルーチンの動作を考えるとき，一つ注意しなければならないことがあります．

---

[1] 本稿では，拡張された宣言文のことを“拡張宣言文”と呼んで，旧来の型指定だけの宣言文と区別します．

前回のサブルーチンの節で，ローカル変数は各ルーチンのデータ領域に所属しているという説明をしましたが，もう少し詳しく言うと，サブルーチンのデータ領域には２種類あります．一つは，サブルーチンがコールされた時点で一時的に生成されるメモリ領域で，もう一つは，サブルーチンの呼び出しに関係なく常に確保されたメモリ領域です．本稿では，前者を“一時メモリ領域”と呼び，後者を“固定メモリ領域”と呼ぶことにします．旧来の宣言文で宣言したローカル変数は，一時メモリ領域に所属します[2]．一時メモリ領域は，サブルーチンを呼び出すたびに場所や内容が変わる可能性があるので，同じサブルーチンを２回コールした場合，１回目のコールでローカル変数に代入した値が，２回目にコールした時点でそのまま残っているとは限りません．

例えば，

```
program stest2
  implicit none
  call subr2(1)
  call subr2(2)
end program stest2

subroutine subr2(n)
  implicit none
  integer n
  real x,y
  if (n == 2) print *,x,y          ! この出力値は不定
  x = 10.0
  y = 100.0
end subroutine subr2
```

のようなプログラムを書いたとします．１回目のコール文，call subr2(1)で，ローカル変数 x と y に，それぞれ 10.0 と 100.0 という値が代入されますが，２回目のコール文，call subr2(2)を実行したときに，print 文で出力した x と y が 10.0 と 100.0 になる保証はないのです[3]．ローカル変数に代入した値を保証するには，固定メモリ領域に所属させなければなりません．拡張宣言文を使って初期値を代入したローカル変数は，一時メモリ領域ではなく，固定メモリ領域に所属します．最初のサブルーチン例，subr1 の変数 n が，コールするごとに１ずつ増加した値を出力するのは，初期値１を代入して宣言することで固定メモリ領域に所属させたためです．

数値を代入せずに，ローカル変数を固定メモリ領域に所属させるには，変数に save 属性を指定します．例えば，上記のサブルーチン subr2 を以下のように書き換えれば，２回目のコールで 10.0 と 100.0 が出力されます．

```
subroutine subr2(n)
  implicit none
  integer n
  real, save :: x,y                ! save 属性を指定
  if (n == 2) print *,x,y
  x = 10.0
  y = 100.0
end subroutine subr2
```

変数の初期値が未定のときや，ローカル配列を固定メモリ領域に所属させたいときは，save 属性を使って下さい．また，初期値を代入する場合でも，固定メモリ領域に所属させることを積極的に利用するのであれば，次のように save 属性を付加して，固定メモリ領域にあることを明示した方が良いと思います．

---

2 これに対し，メインプログラムの変数とモジュール内のグローバル変数は，宣言文の記述にかかわらず，全て固定メモリ領域に所属します．

3 コンパイラによっては，全てのローカル変数を固定メモリ領域に所属させるものがあり，このプログラムでもうまくいく場合があります．しかし，文法的には保証されていないのですから，当てにするべきではありません．

```
integer, save :: n=1
```

その他の便利な属性をいくつか紹介します．まず，同型の配列を多数宣言するときのために，dimension 属性が用意されています．例えば，10×10 の２次元実数型配列を 5 個用意するとき，通常の宣言では，

```
real a(10,10), b(10,10), c(10,10), d(10,10), e(10,10)
```

と書きますが，dimension 属性を使えば，これを次のように書くことができます．

```
real, dimension(10,10) :: a, b, c, d, e
```

すなわち，dimension 属性に配列の形状（次元や要素数）を書いておけば，宣言する変数の位置には，配列名を記述するだけになります．属性は複数付加することができるので，dimension 属性に save 属性を加えて宣言することも可能です．

もう一つ，配列を宣言するときに便利なのが，parameter 属性を指定した変数，parameter 変数です．parameter 変数には必ず値を代入しなければなりません．例えば，

```
integer, parameter :: imax=10, jmax=200
```

のように書くと，imax と jmax が parameter 変数になります．

さて，配列を宣言するとき，その要素数は整数定数で指定しなければなりません．このため，配列の長さを変更するときには，宣言した要素数に関連した数値を全て変更する必要があり，手間がかかるだけでなく，見落としたり，書き間違える可能性があります．例えば，

```
real a(100),b(100)      ! 100 が２カ所
integer i
do i = 1, 100           ! 100 が１カ所
  a(i) = i*i
  b(i) = a(i)**2
enddo
```

というプログラムにおいて，配列要素数の 100 は，宣言文だけでなく，do 文の終了値にも入っています．このため，配列要素数を変更するときは３カ所変更しなければなりません．このとき，配列宣言の要素数を parameter 変数に代入する形で用意しておくと,配列要素数をその parameter 変数で置きかえることができます．例えば，上のプログラムは次のように書き換えることができます．

```
integer, parameter :: imax=100
real a(imax),b(imax)
integer i
do i = 1, imax
  a(i) = i*i
  b(i) = a(i)**2
enddo
```

このプログラムならば，imax に代入する数値を変更するだけで，配列要素数の変更は完了です．

parameter 変数を含んだ計算式を別の parameter 変数への代入値にしたり，配列宣言の要素数に使うこともできます．これらは，その計算式の結果で宣言したことに相当します．例えば，

```
integer, parameter :: imax=100, imax2=imax**2
real a(imax-1),b(imax*2),c(imax2)
```

という宣言は，以下の宣言文と同等です．

```
integer, parameter :: imax=100, imax2=10000
real a(99),b(200),c(10000)
```

ただし，他の parameter 変数に代入するときは，代入される変数（この例では imax2）が，代入

する計算式に使う変数（この例では imax）より後で宣言されなければなりません．

実は，parameter 属性を指定した変数は，“変数＝数値”の形式で変数名と数値の対応関係を示しているだけで，メモリとしての実体はありません．この対応関係を使った“変数”から“数値”への置き換えは，コンパイルの段階で行われるので，非実行文である宣言文でも使えるのです．逆に言えば，parameter 変数に実行文を使って代入することはできません．例えば，

```
integer, parameter :: imax=100
real a(imax),b(imax)
imax = 5                    ! これはエラー
```

と書いた場合，“imax=5”は，コンパイルエラーです．これは，プログラムに“100=5”と書いたことに等しいのですから当然だと言えます．

コンパイル時に数値の置き換えをするのですから，代入という実行動作が不要になり，少しですが計算時間の短縮になります．また，同時にメモリの節約にもなります．そこで，計算で使う定数を parameter 変数にしておくと便利です．例えば，

```
real, parameter :: d13=1.0/3.0, pi=3.141592653589793, pi2=pi*2
real, parameter :: clight=2.99792458e8, hplanck=6.62606896e-34
```

のように，数学定数や物理定数を parameter 変数で用意しておけばいいでしょう．これらをモジュールの中に入れてグローバル変数にしておくと，さらに良いと思います．parameter 変数は，不用意に書き換えられることがないので，このような定数の定義にはうってつけです．

1.3　数値データの精度指定

第１回で，基本的な数値型には整数型と実数型があり，実数型には単精度実数型と倍精度実数型があるという話をしました．Fortran のデフォルト実数型は単精度ですが，コンパイラの自動倍精度機能を使えばデフォルトを倍精度実数型に変更できるので，本稿ではこれまで数値の精度に関して特に意識をしない説明をしてきました．しかし，大量のデータをバイナリ形式で保存するときには保存容量を圧縮するために精度を落とすことが考えられますし，“４倍精度実数”という倍精度よりも有効桁数の多い実数型を使って，高精度計算をすることもできます[4]．このことから，精度を指定する方法を知っておくと便利です．

変数の精度を指定するには，３種類の方法があります．ここでは，基本的な宣言文だけを紹介しますが，属性や数値代入を含む拡張宣言も可能です．

```
型指定*精度数　変数１，変数２，...
型指定(精度数)　変数１，変数２，...
型指定(kind=精度数)　変数１，変数２，...
```

ここで，“精度数”のところには，数値型の byte 数を整数で指定します．実数型の場合，単精度なら 4，倍精度なら 8，４倍精度なら 16 です．例えば，単精度実数型変数を宣言するには，

```
real*4 xs1,ys1,as1(100)
real(4) xs2,ys2,as2(100)
real(kind=4) xs3,ys3,ds3(100,100)
```

などと書きます．また，通常の整数型は 4byte ですが，最近は“倍精度整数”という 8byte の整数型を使うことができるので，これを利用するときには，

```
integer*8 n81,m81,k81(100)
integer(8) n82,m82,k82(100)
integer(kind=8) n83,m83,ka83(100,100)
```

などと書きます．

３種類の書式の中でどれが良いかは一概には言えません．一番目の“*”を使った書式は，古い

[4] ただし，４倍精度実数型は，必ず実装しなければならない規格ではないようで，コンパイラによっては使えない場合があります．gfortran でも，バージョン 4.8 では使えますが，4.2 では使えないようです．

Fortran から使われているので，コンパイラが古い場合や，昔のプログラムを継承する場合のために覚えておいた方が良いと思いますが，これからプログラムを書き始める場合には，二番目か，精度数の意味を明記した三番目の方が良いと思います．これらは，次のように精度数を parameter 変数を使って指定することができます．

```
integer, parameter :: kp=16
real(kp) xq2,yq2,aq2(100)
real(kind=kp) xq3,yq3,dq3(100,100)
```

これに対し，“*”を使った書式では，必ず定数で指定しなければなりません．parameter 変数で精度指定をしておけば，計算機環境に応じて精度を変更する必要があるときに便利です．

ただし，高精度の計算をする場合は，定数も高精度にしなければなりません．例えば，“1.23”と書けば，デフォルトの実数型になるので，本稿では倍精度実数です．よって，このまま４倍精度の計算に使うと精度が落ちてしまいます．$A\times10^B$ で表される実数を４倍精度で表現するときは“*A*q*B*”と書きます．このため，“1.23”は“1.23q0”と書かなければなりません[5]．

しかし，parameter 変数で精度を指定しているときに，e や q のような文字で精度指定をしていると，精度を変更する際に，定数の変更が別途必要です．そこで，数値の後にアンダースコア“_”と精度数を付けて定数の精度を指定することができます．例えば，倍精度の“1.23”は“1.23_8”と書け，４倍精度の“1.23q0”は，“1.23_16”と書くことができます．この精度指定には parameter 変数を使うことができるので，次のように書くことができます．

```
integer, parameter :: kp=16
real(kp) xx,yy
xx = 1.23_kp
yy = 1.2345e-15_kp*xx**3
```

この例の yy に代入している実数の指数指定は e を使っていますが，精度数が 16 なので，４倍精度定数になります．

1.4　include 文

parameter 変数を使えば，配列の要素数だけでなく数値型の精度まで指定することができますが，それでもルーチンごとに宣言をしていると，変更が必要になったときに，使用している全てのルーチンで parameter 変数の値を変更しなければなりません．そこで，

```
module global_param
  integer, parameter :: imax=300, jmax=200
end module global_param
```

のように，parameter 変数だけを記述したモジュールを用意して，共有するルーチンで use 宣言をするのが良いと思います．use 宣言はモジュールの中でも使うことができるので，以下のように記述することが可能です．

```
module global_param
  integer, parameter :: imax=300, jmax=200
end module global_param
module mod_array
  use global_param              ! モジュール内でも use 宣言可能
  real abc(imax,jmax),cd(jmax*2-1)
end module mod_array
program mtest1
  use mod_array                 ! use global_param は不要
  implicit none
  integer km(imax)
```

[5] ちなみに，デフォルト実数が単精度の時，倍精度実定数は“A*d*B”と書きます．例えば，“1.2”は“1.2d0”です．

ここで，あるモジュールの中で別のモジュールを use 宣言した場合，前者のモジュールを use 宣言したルーチンでは，後者のモジュールも合わせて use 宣言したことになります．上記のメインプログラム mtest1 ではモジュール mod_array しか use 宣言をしていませんが,mod_array の中で use 宣言をしているモジュール global_param も同時に宣言したことになり，その中にある変数 imax もメインプログラムで利用可能になります．ただし，次のように，メインプログラムに global_param の use 宣言を加えて，その使用を明示することもできます．

```
program mtest1
  use global_param              ! 書いてもエラーではない
  use mod_array
  implicit none
  integer km(imax)
```

これは，同じモジュールの use 宣言を重複して書いてもエラーにはならないからです．

さて，プログラムが完成して，後は問題に応じて parameter 変数を変更するだけになると，parameter 変数の変更のためだけにプログラム本体を変更するのは煩わしいし，変更するときに誤って本体を書き換えてしまうリスクもあります．プログラムを変更すると，プログラムファイルの日付が変更されるので，いつの時点で完成したプログラムかもわからなくなります．これらは，特に他人にプログラムを提供するときに問題となります．

そこで，parameter 変数の部分だけをプログラム本体から分離して保存し，必要に応じて結合すると便利です．結合する一つの手段に include 文の利用があります．include 文とは，以下のように include の後に，ファイル名を文字列で記述した文です．

```
include 'ファイル名'
```

include 文を書くと，コンパイラはその位置に“ファイル名”で指定したファイルの内容を挿入したプログラムを生成して，それをコンパイルします．例えば，上記のモジュール global_param を“global.inc”というファイルに書き込んで保存しておけば，次のように記述することができます．

```
include 'global.inc'            ! ここに global_param が挿入される
module mod_array
  use global_param
  real abc(imax,jmax),cd(jmax*2-1)
end module mod_array

program mtest1
  use mod_array
  implicit none
  integer km(imax)
  ......
```

include 文が指定するファイルの内容は，それを include 文の位置に挿入したときに，プログラム全体が正しく動作すればいいので，モジュールやサブルーチンのような完結したプログラムである必要はありません．また，同じファイルを一つのプログラムの複数の場所で include 指定することもできます．例えば，

```
  integer, parameter :: imax=300, jmax=200
```

だけをファイルに書いて，サブルーチンごとに include 指定することも可能です．

2. 文字列の活用

これまで print 文や write 文に挿入して，数値の意味などを表示するのに使ってきた文字列ですが，この他にも様々な用途があります．また，固定した文字の並びだけではなく，文字列変数を使って，条件に応じてその内容を変更したり，文字列と文字列を連結したり，一部を取り出し

た文字列を作成するなどの“文字列演算”をすることもできます．本節では文字列をより積極的に活用する手法を紹介します．

2.1　文字列定数と文字列変数

Fortran における文字列とは２個の「'」または「″」で囲んだ文字の並びのことです．例えば，

```
'abc'　'Taguchi␣T.'　″(123.5678+X^2)″　″漢字も書けます″
```

等が文字列です．文字列にはスペースや記号を入れることもできます[6]．「'」と「″」は同等なので，どちらを使うかは自由です．例えば，通常は「'」で囲い，文字列の中に「'」を使いたいときは，「″Teacher's″」のように全体を「″」で囲む，というように決めておけば良いでしょう．

コンピュータの“文字”は文字コードという整数値と対応していて，文字列は文字コードを表す整数の配列で実現されています．このため，整数定数の 1 と文字列の '1' は全く異なるものです．文字コードは半角英数字なら１文字あたり 1byte の整数，全角の漢字やひらがななら１文字あたり 2byte の整数です．半角英数字のコードは，現在ほとんど全てのコンピュータで ASCII コードが使われています．このため，半角英数字だけでプログラムを書けば移植性の問題はありません．しかし，全角文字は OS や利用環境によって文字コードが違う可能性があるので，プログラム中に日本語を記述すると，別のコンピュータに移したときに文字化けすることがあります[7]．Fortran において日本語が使えるのはコメント文とテキスト表示くらいなので，それほど重要ではありません．以下，文字列中の“文字”は 1byte の ASCII コードであると仮定して説明します．

「'」または「″」で囲んだ文字列は，“文字列定数”ですが，文字列を代入して保存する文字列の変数を作ることもできます．文字列変数を作るには character 宣言を使います．character 宣言は，以下のような書式です．

```
character 文字列変数１*文字数１,文字列変数２*文字数２,...
```

文字数とは，文字列変数に代入可能な最大の文字数です．指定できる最小の文字数は 1 です．また，拡張宣言文にして属性などを付加することも可能です．例えば，

```
character c1*10,c2*20,cc*1,chr(20)*30,chs(100,200)*50
```

と宣言すると，文字列変数 c1 には 10 個まで，c2 には 20 個までの文字を代入することができます．これに対し cc には１文字しか入りません[8]．また chr や chs のように文字列の配列を宣言することも可能です．chr は 30 文字まで入る１次元配列で，chs は 50 文字まで入る２次元配列です．

同じ文字数の文字列変数を複数個宣言するときは，以下の書式で宣言することができます．

```
character*文字数　文字列変数１，文字列変数２，...
character(文字数)　文字列変数１，文字列変数２，...
character(len=文字数)　文字列変数１，文字列変数２，...
```

ここで，“文字列変数”には，変数名か，配列名とその要素数を記述します．この３種類の相違点は，1.3 節の精度数の指定と同様で，最初の“*文字数”が旧来の書式，それ以外が新しい書式です．新しい書式の場合には，parameter 変数を利用して文字数を指定することができます．

文字列変数に文字列を代入するには，数値の代入と同じで，イコール「=」を使います．例えば，上記の 10 文字の文字列変数 c1 に文字列定数を代入するには，

```
c1 = 'abc'
```

のように書きます．このとき，代入される文字 'abc' は 3 文字なので，10 文字の変数 c1 の先頭

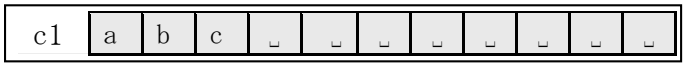
---

6 本節では半角スペース記号を明記するときに“␣”を使います．

7 パソコンで使われているのは Shift JIS コードが多く，Linux などの UNIX 系 OS で使われているのは EUC コードが多いようです．最近は，コード体系をより統一化した Unicode を使う OS が増えてきました．

8 文法的には“*文字数”を省略して変数名だけで宣言すると，文字数が１の文字列になります．しかし，プログラムがわかりにくくなるので，常に“*1”を付加して文字数１を陽に指定した方が良いでしょう．

から順に 1 文字づつ代入され，残りの領域は半角スペースが代入されます．スペースも文字ですから，c1 の文字数は，代入した文字列の文字数にかかわらず，10 のままです．図に描けば，以下のようなイメージです．



逆に，代入する文字列の文字数の方が多い場合には，その文字列の先頭から，代入できる最大文字数までが代入され，残りは無視されます．拡張宣言文を使えば，次のようにあらかじめ文字列定数を代入した文字列変数を用意することも可能です．

```
character :: chr*10='abcde'
```

文字列変数に代入された文字列は部分的に取り出すことができます．これを“部分文字列”といいます．部分文字列は，文字列変数の先頭から数えて何番目から何番目という範囲を「:」を使って指定します．例えば，c1 という文字列変数の n1 番目から n2 番目の文字を取り出すには，

```
c1(n1:n2)
```

と指定します．この文字列は，n2-n1+1 文字の文字列として扱われます．例えば，c1='abcdefg' ならば，c1(3:5)は，'cde'です．n1 と n2 を等しくすることで１文字を取り出すこともできます．例えば，

```
c1 = 'abcdefg'
do i = 1, 7
   print *,c1(i:i)
enddo
```

とすれば，１文字ずつ縦に出力されます．

文字列配列の部分文字列を取り出す場合には，要素指定を先に，部分文字列指定を後に書きます．例えば，１次元の文字列配列 chr に対し，k 番目の要素の部分文字列は，

```
chr(k)(n1:n2)
```

のように指定します．かっこが連続するので，順番に気を付けて下さい．

文字列は連結することもできます．文字列の連結には演算子「//」を用います．例えば，

```
c1 = 'abc'//'xyz'
```

と書くと，c1 には'abcxyz'という文字列が代入されます．文字列の連結は，文字列定数と文字列変数，文字列変数と文字列変数という組み合わせでも可能です．ただし，文字列変数に代入された文字列を連結するときには注意が必要です．例えば，

```
character c1*10,c2*20
c1 = 'abc'
c2 = c1//'xyz'
```

と書いた場合，c2 に'abcxyz'という文字列が代入されると思ったら間違いです．正しくは

```
'abc␣␣␣␣␣␣␣xyz'
```

が代入されます．これは，上記のように 10 文字の文字変数 c1 に 3 文字の 'abc'を代入しても，c1 の文字数は変わらないからです．末尾の不要なスペースを削除するには，部分文字列を使う必要があります．

```
character c1*10,c2*20
c1 = 'abc'
c2 = c1(1:3)//'xyz'
```

このプログラムならば，c2 に'abcxyz'が代入されます．

しかし，この方法は変数に代入されている文字数が不明のときには使えません．そこで，関数

trim が用意されています．trim は末尾のスペースを除去した文字列を返す組み込み関数です．例えば，上記の例は，

```
character c1*10,c2*20
c1 = 'abc'
c2 = trim(c1)//'xyz'
```

と書くことができます．この結果も c2 には'abcxyz'が代入されます．

文字列をサブルーチンの引数にするときは，「(*)」を指定して宣言します．例えば，

```
subroutine subr3(chr)
  implicit none
  character(*) chr          ! 文字数は不要
```

の chr のように宣言します．Fortran の文字列には，文字コードの並びだけではなく，文字数の情報も含まれています．このため，(*)指定のように文字数が明記されていなくても，コール側で指定した文字数の文字列として使用することができます．例えば，

```
program ctest1
  implicit none
  call subr3('abcde')
end program ctest1

subroutine subr3(chr)
  implicit none
  character(*) chr
  print *,chr,len(chr)      ! 文字列 'abced'と文字数 5 が出力される
end subroutine subr3
```

のようなプログラムにおいて，サブルーチン subr3 の print 文の出力は，文字列 'abcde'と文字数 5 になります．ここで，関数 len は，文字列の文字数を取得する関数です．文字列に関する組み込み関数は，2.4 節で説明します．

文字列には大小関係があり，これを利用して if 文で条件分岐をすることもできます．２個の文字列を比較するときは，以下の手順で行います．

(1) 文字数の長さが異なるときは，短い方の文字列の末尾にスペースを追加して文字数を等しくする
(2) ２個の文字列を先頭から１文字づつ比較していって，全て同じならば，“等しい (==)”
(3) 異なる文字があれば，最初に異なる文字の ASCII コードを比較して，コード値の大小が文字列の“大 (>)”または“小 (<)”

ASCII コードは，“スペース”＜“数字”＜“英大文字”＜“英小文字”の順で大きくなり，個々の文字には次のような大小関係があります．

```
' '< '0'< '1'<...< '9' < 'A'< 'B'<...< 'Z' < 'a'< 'b'<...< 'z'
```

例えば，文字列の比較を使って次のようなプログラムを書くことができます．

```
character c1*10,c2*20
c1 = 'abcde'                    ! ３番目が 'c'
c2 = 'abdce'                    ! ３番目が 'd'
if (c1 < c2)  print *,'c1 < c2'
if (c1 == c2) print *,'c1 == c2'
if (c1 > c2)  print *,'c1 > c2'
```

この結果は，'c1＜c2'です．なぜなら，３番目の文字が初めて異なり，c1 は 'c'，c2 は 'd' ですが，'c'＜'d'だからです．なお，c1 は 10 文字，c2 は 20 文字ですが，後ろにスペースを補うので，この例の結果には無関係です．

2.2　出力における文字列の利用

最もよく文字列を使う場面は，print 文や write 文の中に入れて表示の補助に使うことでしょう．例えば，

```
real spd,pres
spd = 10.0
pres = 960.0
print *,' Wind Speed = ',spd,'  Pressure = ',pres
```

と書けば，出力が

```
 Wind Speed =    10.0000000000000        Pressure =    960.000000000000
```

のようになって，どの数値がどの変数の値なのかが一目でわかります．

ここで，文字列はその長さに合わせて出力されています．これは，print 文が文字列に入っている文字数の情報に合わせて出力するからです．この機能は，format を使って文字列の出力指定をするときにも使うことができます．文字列の出力指定には，A 編集を用いますが，A 編集は，「a」だけ書くと，文字列の文字数が出力幅になります．例えば上記の print 文を，

```
print 600,' Wind Speed = ',spd,' m/s'
print 600,' Pressure = ',pres,' hPa'
600 format(a,f8.2,a)
```

と書けば，次のような文字幅に合わせた出力結果が得られます．

```
 Wind Speed =    10.00 m/s
 Pressure =   960.00 hPa
```

前回の format の説明で紹介しましたが，出力形式指定を print 文や write 文の中に埋め込む書式も，文字列の利用方法の一つです．例えば，

```
print 600,x
write(10,600) x
600 format(' x=',es12.5)
```

というプログラムは次のように書き換えることができます．

```
print "(' x=',es12.5)",x
write(10,"(' x=',es12.5)") x
```

すなわち，format 文のかっこ以下を，両端のかっこを含めて文字列にして print 文や write 文の書式指定の位置に書き込みます．このとき，文字列変数を使えば，複数の場所で同じ format を使うときに便利だし，出力内容に応じて，実行時に format を変更することも可能です．例えば，

```
real x
character form*20
if (abs(x) >= 1.e5) then
   form = "('x=',es12.5)"
 else
   form = "('x=',f10.5)"
endif
print form,x
```

とすれば，x の絶対値が 10^5 以上のときには es12.5 編集で，さもなくば f10.5 編集で出力されます．

また，部分文字列を利用して変更することも考えられます．例えば，

```
real x
character form*20
form = "('x=',f10.5)"
if (x >= 100.0) form(8:11)='12.3'    ! 10.5 から 12.3 に変更
print form,x
```

のように書けば, x が 100.0 以上のときは f12.3 編集で, さもなくば f10.5 編集で出力されます.

2.3　数値・文字列変換

ここまでは write 文や read 文の書式指定の位置に文字列を使用していましたが,装置番号の位置に文字列を記述することもできます. これは“文字列”から“数値”への変換, またはその逆変換を行うときに使用します.上記のように 123 という数値と'123'という文字列は計算機内部の表現が異なりますが, 処理の過程で 123 という数値からそれに相当する文字を作ったり, 逆に'123'という文字列を数値として計算に利用したい場合があります. そもそも, 書式あり read 文はファイルなどから“文字で表現された数値”を入力して“計算機内部の数値”に変換する動作であり, 書式あり write 文は“計算機内部の数値”を“文字で表現された数値”に変換して出力する動作です. 装置番号の位置に文字列を利用すると, その変換機能だけを使うことができるのです.

“数値”→“文字列”変換をするには write 文を用います. 例えば,

```
real x
character ch*20
x = 123.5
write(ch,"(f10.5)") x
```

と書けば, 文字列変数 ch に'123.50000'という文字列が代入されます. ただし, 文字列に変換するときは, 出力文字数以上の長さを持つ文字列変数を用意しておく必要があります.

open 文を使ってファイルをオープンするとき, ファイル名は文字列で指定する必要があります. このため, file01.dat, file02.dat,...,file10.dat という通し番号の付いた 10 個のファイルが存在して, それぞれを装置番号 11～20 としてオープンするときは, 次のように書きます.

```
character name*20
integer m
do m = 1, 10
   write(name,"('file',i2.2,'.dat')") m
   open(10+i,file=name)
enddo
```

ここで, 整数 m の出力編集を i2.2 にしているのは, スペースを '0'で埋めて, 整数の出力を 01, 02, ...のようにするためです.

逆に, “文字”→“数値”変換をするには read 文を用います. 例えば,

```
real x
character ch*20
ch = '12345.0'
read(ch,*) x
```

と書けば, 実数型変数 x に 12345.0 という“数値”が代入されます.

2.4　文字列に関する組み込み関数

文字列に関する組み込み関数は, trim の他にも色々あります.代表的なものを表１に示します.

表１. 文字列に関する組み込み関数

文字列関数	引数の型	関数値の型	関数の意味
trim(c)	文字列	文字列	末尾の空白を削除
len(c)	文字列	整数	文字列の文字数
len_trim(c)	文字列	整数	末尾の空白を削除した文字列の文字数
adjustl(c)	文字列	文字列	文字列を左にそろえた文字列
adjustr(c)	文字列	文字列	文字列を右にそろえた文字列

表１（続き）．文字列に関する組み込み関数

文字列関数	引数の型	関数値の型	関数の意味
index(c,cp)	２個の文字列	整数	文字列 c 中で文字列 cp が含まれていれば，その開始位置を返す． 無ければ 0 を返す
repeat(c,n)	文字列と整数	文字列	文字列 c を n 個連結した文字列
char(n)	整数	１文字の文字列	ASCII コード値を与えると，それに対応する半角英数文字を返す
ichar(c)	１文字の文字列	整数	半角英数文字を与えると，それに対応する ASCII コード値を返す

使用例をいくつか示します．2.1 節に出てきた関数 len は，文字列の文字数を取得するための関数ですが，使用の際は注意が必要です．例えば，

```
character c1*10
integer m,n
m  = len('abc')   ! m = 3
c1 = 'abc'
n  = len(c1)      ! n = 10
```

とすると，m の値は 3 ですが，n の値は 10 になります．これは，文字列変数の文字数が宣言時の文字数だからです．末尾のスペースを除去した文字列の長さを取得するには，関数 trim を使って削除してから関数 len で調べるか，関数 len_trim を使います．例えば，上のプログラムで，最後の n への代入文を

```
n=len(trim(c1))  または  n=len_trim(c1)
```

で置き換えれば，n=3 になります．

最後の char と ichar の使用例を示します．この二つの関数を組み合わせれば，read 文や write 文を使わなくても，文字と数値の変換をすることができます．例えば，２桁の整数を３桁の８進数で表示するプログラムは以下のようになります．

```
integer zero,num
character c1*10
num = 12
c1 = '000'
zero = ichar('0')                    ! '0'の文字コードを整数値に変換
c1(3:3) = char(zero+mod(num,8))      ! num の 1 の位の文字
c1(2:2) = char(zero+mod(num/8,8))    ! num の 8 の位の文字
c1(1:1) = char(zero+mod(num/64,8))   ! num の 64 の位の文字
print *,'decimal(',num,') = octal( ',trim(c1),' )'
```

ASCII コードでは，'0','1','2',...,'8','9'の順で値が 1 ずつ増加しています．そこで，先頭の'0'のコード値に１桁の数字を加えれば，その数字のコード値が得られます．同様に，大文字の英数字や，小文字の英数字も 1 ずつ増加して並んでいるので，先頭の文字である'A'や'a'のコード値に，その文字の順番の差の数字を加えて逆変換すれば，対応する文字を得ることができます．整数の 16 進数表示をするプログラムを考えてみてはいかがでしょうか．

3.　配列計算式

文字列の処理は，これまで説明してきた数値の計算と少し毛色が違います．数値計算が１個の数値と１個の数値の間の演算を基本としているのに対し，文字列の処理では，“文字列”という文字コードの集合をひとまとめにして取り扱い，文字列と文字列の連結や文字列変数への代入を一つの式で実行することができます．文字列とは文字コードの配列なのですから，これを数値計算的に処理するなら

ば，本来は１文字ずつ処理しなければなりません．文字列の演算が可能なのは，Fortran コンパイラが，“文字列”という数値集合の処理を，内部で１文字ずつ処理するプログラムに変換してくれるからです．

Fortran では，この概念を一般の配列に拡張した“配列演算”が可能です．配列演算とは，数値の集合を配列名で代表させ，配列要素と配列要素の演算を，配列名と配列名の演算の形で記述するものです．配列演算を使えば，do 文を使わずに配列を使った計算を記述することができます．これを“配列計算式”と呼びます．配列計算式を使う利点は，単にプログラムが短くなるだけではありません．計算が最適になるように，繰り返しの順序やメモリ配置をコンパイラが決めてくれるので，場合によっては，do 文を使って書くより高速に計算させることができます．

3.1　基本的な配列計算式

配列演算は，配列の全要素に対して，全て同じパターンの演算をするのが基本です．このため，１行の配列計算式で用いる配列は，次元と各次元の要素数が全て等しい，“同型の配列”でなければなりません．その点に注意をすれば，配列の全ての要素に同じ定数を代入するときや，２個の配列の対応する要素間で全て同じ四則演算をするときに，あたかも配列名が一つの変数であるかのような記述をすることができます．例えば，次のような記述が可能です．

```
real x(10,10),y(10,10),z(10,10)
x = 1.0
y = 2.0
z = x + y*3.5
```

このプログラムを実行すると，配列 x の全ての要素は 1.0 になり，配列 y の全ての要素は 2.0 になり，配列 z の全ての要素は 8.0 (=1.0+2.0*3.5) になります．このプログラムは do 文を使った以下のプログラムと同じ結果になります[9]．

```
real x(10,10),y(10,10),z(10,10)
integer i,j
do j = 1, 10
  do i = 1, 10
    x(i,j) = 1.0
    y(i,j) = 2.0
    z(i,j) = x(i,j) + y(i,j)*3.5
  enddo
enddo
```

配列計算式中の定数は，全要素に対して共通の値として計算します．

次のような，関数を使った配列計算式も可能です．

```
real a(10,10),b(10,10),c(10,10)
...
a = sqrt(b*sin(c)/(3.2*c + 1.5e-3))
```

この配列計算式を do 文で表すと，次のようになります．

```
do j = 1, 10
  do i = 1, 10
    a(i,j) = sqrt(b(i,j)*sin(c(i,j))/(3.2*c(i,j) + 1.5e-7))
  enddo
enddo
```

配列計算式においては，式中に含まれる全ての配列が同型でなければならないので，異なる次元や要素数の配列が混じっているとエラーになります．ただし，下限は異なってもかまいません．

---

9 ただし，ループの処理はコンパイラが自動的に生成するので，実際にこの通りの順序で計算するかどうかはわかりません．あくまでも，このプログラムと同じ結果になるという意味だと考えてください．

例えば,

```
real a(3,3),r(-1:1,0:2)
...
a = 3.14*r**2
```

のように計算することができます．ただし，この配列計算式を do 文で表せば,

```
do j = 1, 3
  do i = 1, 3
    a(i,j) = 3.14*r(i-2,j-1)**2
  enddo
enddo
```

のように，対応する要素の位置がずれるので注意して下さい.

このように，配列計算式を使うとプログラムがシンプルになります．逆に，単一変数の計算式と配列計算式との区別がなくなるので，両者が入り交じるとプログラムがわかりにくくなり，エラーが見つけにくくなるという欠点もあります.

なお，サブルーチンの引数配列を使って配列計算をするときには注意が必要です．例えば,

```
subroutine subr(a,b)
  implicit none
  real a(*),b(*)
  a = b**2        ! これはエラーになる
  ...
```

のようなプログラムでは，配列計算式はエラーになります．なぜなら，“*”を入れて配列宣言をした場合は，要素数が不定だからです．これに対し,

```
subroutine subr(a,b,n)
  implicit none
  real a(n),b(n)
  integer n
  a = b**2        ! この場合は OK
  ...
```

のように，整合配列を使うと，配列計算が可能になります.

3.2　部分配列

配列名だけを使った配列計算式では全ての要素について計算をしますが，常に全要素の計算が必要とは限りません．そこで，配列の一部を取り出した配列，“部分配列”を指定することができます．部分配列は“:”（コロン）を使って要素番号の範囲を指定します．例えば，１次元配列 a に対して,

```
a(n1:n2)
```

と書けば，a(n1)～a(n2)という n2-n1+1 個の要素から構成された１次元配列として計算に使うことができます．また，２次元配列 b に対して,

```
b(n11:n12,n21:n22)
```

と書けば，b(n11,n21)～b(n12,n21)～b(n11,n22)～b(n12,n22)という (n12-n11+1) ×(n22-n21+1) の２次元配列として計算に使うことができます．また，ある次元の要素番号を固定して,

```
b(n,n21:n22)
```

と書けば，b(n,n21)～b(n,n22)という n22-n21+1 個の要素から構成された１次元配列として計算に使うことができます.

配列計算式で部分配列を使うときは，その部分配列がその他の配列と同型であれば良いので，宣言文での次元や要素数が一致している必要はありません．例えば１次元配列なら,

```
real a(10),b(5)
...
a(3:5) = b(1:3)**2
```

のように書くことができます．また，２次元配列の部分配列を使って，

```
real a(10,10),b(5,4),c(10)
...
a(3:5,6:8) = b(1:3,1:3)**2
```

などと書くことができます．この配列計算式を do 文で表せば，

```
do j = 6, 8
  do i = 3, 5
    a(i,j) = b(i-2,j-5)**2
  enddo
enddo
```

となります．ある次元の要素番号を固定した２次元配列は１次元配列として扱えるので，２次元配列と１次元配列が混在した計算も可能です．例えば，

```
real a(10,10),c(10)
...
c(3:8) = a(3,3:8)**2
```

と書くことができます．この配列計算式を do 文で表せば，

```
do j = 3, 8
  c(j) = a(3,j)**2
enddo
```

となります．

なお，要素のところに“:”だけを記述すると“その次元の全要素”という意味になります．そこで，全配列要素を使った配列計算式を書くときにも“:”を使って配列であることを明示することができます．例えば，前節の２次元配列計算式，

```
a = sqrt(b*sin(c)/(3.2*c + 1.5e-3))
```

は，次のように書くことができます．

```
a(:,:) = sqrt(b(:,:)*sin(c(:,:))/(3.2*c(:,:) + 1.5e-3))
```

この方が，単一変数の計算と区別できるので良いでしょう．本稿でもこれ以降はこの書式を使って記述します．

3.3　where 文による条件分岐

配列演算は便利ですが，全ての要素について同じ形の計算をするので，要素の条件に応じて異なる処理をさせることができません．そこで，配列要素の条件に応じて動作を分岐させるための where 文が用意されています．where 文は，以下のような形式です．

```
where（配列条件）配列計算式
```

これは，“配列条件”に合った要素に対してのみ，“配列計算式”を実行するという意味です．例えば，

```
real a(10,10),b(10,10)
...
where (a(:,:) > 0) b(:,:) = a(:,:)**2
```

のように書きます．この where 文の部分を do 文で表せば，次のようになります．

```
do j = 1, 10
  do i = 1, 10
    if (a(i,j) > 0) b(i,j) = a(i,j)**2
  enddo
enddo
```

where 文はブロック構文にすることもできます. ブロック where 文では, 次のように else where 文を付加して，条件に一致しない場合の記述をすることもできます.

```
where（配列条件１）
  配列計算式１
  .....
else where（配列条件２）
  配列計算式２
  .....
else where
  配列計算式０
  .....
endwhere
```

この場合，配列条件１を満足する要素は配列計算式１のブロックを実行し，配列条件１を満足せず，配列条件２を満足する要素は配列計算式２のブロックを実行し,... という具合に続いて，全ての条件を満足しない要素は配列計算式０のブロックを実行するという動作になります．中間の else where に関するブロックは省略可能です．if 文と違って，“then”が不要なこと，“全ての条件以外”を表す文が“else where”であること，ブロックの最後に“endwhere”文を付加すること，などに注意して下さい．例えば,

```
real a(10,10),b(10,10)
...
where (3.0*b(:,:) > 0)
  a(:,:) = b(:,:)**2
 else where
  a(:,:) = b(:,:)**3
endwhere
```

のように書くことができます．このブロック where 文を do 文で表せば，次のようになります.

```
do j = 1, 10
  do i = 1, 10
    if (3.0*b(i,j) > 0) then
      a(i,j) = b(i,j)**2
     else
      a(i,j) = b(i,j)**3
    endif
  enddo
enddo
```

where ブロック中で使用する配列は，全て同型でなければなりません．このため，where ブロックの中に，単一変数の計算式や，次元や要素数の異なる配列計算式を入れることはできません.

3.4　配列定数

配列演算を使えば，do 文を使わなくても配列要素の計算ができますが，ここまでの書式では配列要素ごとに異なる定数を使用することができません．そこで，１次元配列だけですが，配列定数が用意されています．n 個の要素からなる１次元の配列定数は以下の形式です.

```
(/定数１,定数２,...,定数 n/)
```

このとき，n 個の定数は全て同じ型の数値型でなければなりません．例えば,

```
real a(5)
a(:) = (/1.0,2.0,3.0,4.0,5.0/)
```

と書けば，右辺は実数型の１次元配列定数であり，この配列計算式は，a(1)=1.0，a(2)=2.0，a(3)=3.0，a(4)=4.0，a(5)=5.0 という 5 個の代入文を実行したことに相当します．この配列定数は，次のように拡張宣言文の初期値代入に使うこともできます．

```
real :: a(5)=(/1.0,2.0,3.0,4.0,5.0/)
```

しかし，この形式では数値を一個一個書かねばならず，要素数が多いと大変です．そこで，前回の入出力文の説明に出てきた，“do 型形式”を使って定数設定をすることができます．これを do 型定数といいます．do 型定数とは，以下のような形式です．

（データ１,データ２,...,整数型変数=初期値,終了値）

この形式では，まず整数型変数（カウンタ変数）を初期値にして，データ１，データ２，...と並べ，次に，カウンタ変数を１増加して，再度，データ１，データ２，...と並べ，カウンタ変数が終了値になるまで繰り返して得られる配列定数になります．このため，配列要素数は“データ並びの数×カウンタ変数のカウント回数”になります．例えば，

```
(/(n, n=1,5)/)
```

という do 型定数を使った配列定数は，整数型の配列定数，

```
(/1,2,3,4,5/)
```

と同じです．また，

```
(/(n, n**2, n=1,5)/)
```

という do 型定数を使った配列定数は，整数型の配列定数，

```
(/1,1,2,4,3,9,4,16,5,25/)
```

と同じです．複数の do 型定数を並べたり，通常の定数と混在させることも可能です．例えば，

```
(/(real(n), n=0,2),1.5,1.8,(real(n), n=3,5)/)
```

という実数型の配列定数は，

```
(/0.0,1.0,2.0,1.5,1.8,3.0,4.0,5.0/)
```

と同じです．このとき，型変換関数の real(n)を単に n と書くとエラーになるので注意して下さい．これは，n が整数型なので，n だけを書くと一つの配列定数の中に実数型と整数型が混在するからです．

do 型定数は多重にすることも可能です．例えば，

```
(/((i+j,i=1,3),j=1,4)/)
```

という do 型定数を使った整数型の配列定数は，

```
(/2,3,4,3,4,5,4,5,6,5,6,7/)
```

と同じです．多重にした場合，カウンタ変数は内側が先に進みます．

本節最初の例は，次のように do 型定数を使って書くことができます．

```
real a(5)
a(:) = (/(n, n=1,5)/)
```

ここで，右辺は整数型の配列で，左辺は実数型の配列ですが，この場合は全要素に対して整数型から実数型への変換が行われて代入されるのでエラーにはなりません．なお，do 型定数は拡張された宣言文での初期値代入に使うこともできますが，カウンタ変数はその宣言文より前に宣言されている必要があります．

配列定数は１次元配列しか用意されていません．このため，２次元以上の配列計算で使うときは，１次元ごとに計算する必要があります．例えば，

```
real a(3,4)
integer i,j
do j = 1, 4
  a(:,j) = (/(sin(0.5*(i+j)), i=1,3)/)
enddo
```

のように書きます．これは以下のプログラムと同等です．

```
real a(3,4)
integer i,j
do j = 1, 4
  do i = 1, 3
    a(i,j) = sin(0.5*(i+j))
  enddo
enddo
```

ただ，配列計算で do 文を使うのはあまりスマートではありません．そこで，配列の形状を変換する組み込み関数 reshape が用意されています．reshape を使って１次元の配列定数を多次元配列に変換すれば，そのまま多次元の配列計算式に記述できます．この方法は次節で説明します．

3.5　配列に関する組み込み関数

配列を引数に持つ組み込み関数は色々ありますが，主として配列情報に関する関数を表２に示します．

表２．配列に関する組み込み関数

配列関数	引数の型	関数値の型	関数の意味
sum(a)*	配列	配列要素の型	全ての配列要素の和
product(a)*	配列	配列要素の型	全ての配列要素の積
minval(a)*	配列	配列要素の型	全ての配列要素の最小値
maxval(a)*	配列	配列要素の型	全ての配列要素の最大値
minloc(a)	配列	整数型配列	全ての配列要素の最小値の位置
maxloc(a)	配列	整数型配列	全ての配列要素の最大値の位置
lbound(a)*	配列	整数型配列	配列の各次元の最小要素番号リスト
ubound(a)*	配列	整数型配列	配列の各次元の最大要素番号リスト
shape(a)	配列	整数型配列	配列の各次元の要素数リスト
size(a)*	配列	整数	配列の全要素数
reshape(a,s)	配列と整数型配列	a の型の配列	配列 a を配列 s で指定された型の配列に変換する
allocated(a)	配列	論理	配列 a が割り付けられていれば真，さもなくば偽

ここで，* の付いた関数は，引数配列の後に次元を示す整数型の引数を追加して，その次元に関する値を取り出すこともできます．例えば，

```
real array(3,4)          ! 3×4 の配列
integer m,n1,n2
m  = size(array)         ! 全要素数 3×4=12
n1 = size(array,1)       ! 第１次元要素数 3
n2 = size(array,2)       ! 第２次元要素数 4
```

と書けば，m には，array の全要素数である 12 が代入され，n1 には array の第１次元の要素数で

ある 3 が，n2 には array の第２次元の要素数である 4 が代入されます．

表２で，関数値の型が“整数型配列”になっている関数は，引数に与えた配列の次元を要素数に持つ１次元整数型配列が戻り値です．その際，その１次元配列の各要素には，引数配列の各次元の情報が代入されています．例えば，下限と上限を取得するための関数，lbound と ubound は以下のように使います．

```
real bb(-10:10,4)          ！２次元配列
integer blow(2),bupp(2)    ！２次元なので要素数２の配列
blow = lbound(bb)
bupp = ubound(bb)
print *,blow,bupp
```

ここで，２次元配列 bb は，第１次元の下限が-10，上限が 10，第２次元の下限が 1，上限が 4 ですから，定数で書けば，blow は(/-10,1/)，bupp は(/10,4/)になります．このとき，ubound(bb,2)のように，次元に関する引数を加えると，その結果は一個の整数（この例では 4）になります．

reshape は，配列の形状を変換する関数です．そもそも，どんな次元の配列もメモリ上では１次元的に並んでいるので，２次元配列 a(3,4)と１次元配列 b(12)のように，全要素数が等しい配列は同じように取り扱えるはずですが，配列演算は，次元および各次元の要素数が等しい，同型の配列間でしか許可されていません．そこで，a(3,4)と b(12)の間で演算をするには，形式上，同型になるような変換が必要です．これを実行するのが関数 reshape です．reshape には，最初の引数に変換したい配列を与え，２番目の引数に変換後の形状を表す１次元配列を与えます．ここで形状を表す１次元配列とは，変換後の配列が，k1×k2×...×kn の n 次元配列であれば，

```
(/k1,k2,...,kn/)
```

で与えられる要素数 n の１次元整数型配列のことです．例えば，3.4 節最後の例文は，

```
real a(3,4)
integer i,j
a(:,:) = reshape((/((sin(0.5*(i+j)), i=1,3), j=1,4)/), (/3,4/))
```

と書くことができます．reshape の第１引数は do 型定数なので１次元配列ですが，これを第２引数の(/3,4/)で 3×4 の２次元配列に変換するよう指定しているわけです．

この配列の形状を表す１次元配列は，関数 shape を使って取得することができます．例えば，上記の a(3,4)という宣言をした配列に対し，shape(a)は，１次元整数型配列，(/3,4/)になります．そこで，上記の配列計算式は次のように書くことができます．

```
a(:,:) = reshape((/((sin(0.5*(i+j)),i=1,3),j=1,4)/),shape(a))
```

しかし，これではまだ do 型変数における i や j の範囲指定が定数のままです．そこで，ここは先ほど説明した関数 size で書き直すと良いでしょう．

```
a(:,:) = reshape((/((sin(0.5*(i+j)),i=1,size(a,1)),j=1,size(a,2))/),shape(a))
```

この配列計算式なら，宣言文の要素数を変更するだけで変更は完了です．

3.6　配列の動的割り付け

配列を宣言するときには，整数定数を与えて要素数を明示しなければなりません．これは，その情報に基づいて，プログラムの動作開始時に計算で使用するメモリ量を確定するためです．しかし，最近のコンピュータではプログラムの実行中に必要になったメモリを確保したり，不要になったメモリを解放するという動作が頻繁に行われています．このような，プログラムの実行中に必要に応じてメモリを確保することを“動的割り付け”といいます．Fortran も Fortran90 から動的割り付けが可能になり，プログラムの動作に必要な要素数を持つ配列を実行中に確保できるようになりました．動的割り付けを使うと，メモリ効率の良い汎用性のあるプログラムを作成することができます．ここでその方法を紹介します．

まず，サブルーチンの中で必要に応じた要素数の配列を確保する簡単な方法として，サブルーチンの引数を使った配列宣言があります．例えば，次のサブルーチンで real 宣言されている２次元配列 b がこれに相当します．

```
subroutine memory1(a,m,n)
  implicit none
  real a(m,n),b(m,n),x              ！引数 m と n を使って宣言する
  integer m,n,i,j
  x = 10.0
  b(:,:) = x*reshape((/((i*j,i=1,m),j=1,n)/),shape(b))
  a(:,:) = b(:,:)**3
end subroutine memory1
```

配列 b は，このサブルーチンがコールされたときに，引数 m と n の値を使って確保されます．このとき，b(m,n)のような単純な宣言だけではなく，b(0:2*m,n-1)のような，下限指定や計算式を使った宣言も可能です．なお，配列 a も引数を使って宣言されていますが，a は引数に含まれているので前回説明した整合配列です．整合配列は，引数配列の取り扱いを決める仕組みにすぎず，配列自体はサブルーチン側にありません．

引数に含まれていない配列 b はローカル配列であり，宣言されたサブルーチンの一時メモリ領域に所属します．1.2 節で説明したように，一時メモリ領域に所属するローカル変数は，サブルーチンを呼び出した時点で生成されます．引数の値はサブルーチン開始時に確定しているので，これを使って配列に必要なメモリを確保することが可能なのです．ただし，これはあくまでもサブルーチンで宣言する配列についてのみ使える機能であり，メインプログラムの配列やモジュール中のグローバル配列としては使えません．

そこで，メモリの動的割り付けを明示的に行う仕組みが用意されています．Fortran で動的割り付けを行うには，二つの手続きが必要です．一つは，割り付けたメモリを配列として使用するための配列名の宣言です．これは，割り付けたメモリの先頭アドレスを保持するためのメモリを用意することだと考えられます．もう一つは，その配列名に，実際のメモリを割り付ける動作です．前者は宣言文なので非実行文，後者はプログラム実行中に割り付けるので実行文です．

メモリを割り付ける予定の配列名は，allocatable 属性を付けて型宣言をします．このとき，配列の次元情報をコロン“:”を使って示す必要があります．配列の次元はコロンの数で決まり，実行時に変更することはできません．例えば，

```
real, allocatable :: ab(:),z2(:,:)
integer, allocatable, dimension(:,:) :: km
```

のように宣言します．１行目のように，名前の後ろに次元情報を付加しても良いし，２行目のように，dimension 属性を使うことも可能です．この例の場合，ab は１次元実数型配列，z2 は２次元実数型配列，km は２次元整数型配列として割り付けることができます．

配列の割り付けは allocate 文で行います．allocate 文は，次のように配列の要素指定をサブルーチンのようにかっこで囲んで記述します．

```
allocate ( ab(100), z2(0:m,-m:m), km(k-1,2*k) )
```

かっこ内は，配列宣言と同じ形式です．ab や km のように，要素数に整数だけを与えると，その次元の下限は１になるし，z2 のように“:”を使って下限指定をすることも可能です．割り付けるときは，配列の変数型が混在していてもかまいません．allocate 文は実行文なので，z2 や km のように整数型変数や整数式の計算結果を使って割り付けることも可能です．一旦割り付けられた配列は，通常の宣言文で宣言した配列と同様に使用することができます．

ただし，むやみに割り付けを繰り返すと，コンピュータで利用できる容量を超える“メモリオーバー”になる可能性があります．そこで，不用になった配列のメモリ領域は deallocate 文で解放することができます．deallocate 文は，次のように配列名だけをかっこで囲んで指定します．

```
deallocate ( ab, z2, lm )
```

メモリの割り付けや解放はそれほど時間がかかる処理ではないので,必要なときに割り付けて,不要になったら解放するようにしましょう．例えば，先ほどのサブルーチン memory1 は，以下のように書き換えることができます.

```
subroutine memory1(a,m,n)
  implicit none
  real a(m,n),x
  real, allocatable :: b(:,:)          ！割り付け用２次元実数型配列
  integer m,n,i,j
  allocate (b(m,n))                    ！ｍ×ｎの２次元配列を割り付け
  x = 10.0
  b(:,:) = x*reshape((/((i*j,i=1,m),j=1,n)/),shape(b))
  a(:,:) = b(:,:)**3
  deallocate (b)                       ！メモリの解放
end subroutine memory1
```

なお，引数変数を使って宣言したローカル配列は一時メモリ領域に所属しますが，allocate 文で割り付けた配列は固定メモリ領域に所属します[10]．サブルーチンにおける一時メモリ領域は固定メモリ領域より容量が少ないことが多いので，要素数の大きな配列を割り付けるときは allocate 文を使う方が良いでしょう. allocate 文を使った動的割り付けは，メインプログラムでも利用できるし，モジュールの中で配列宣言をして，グローバル配列として使用することも可能です.

ただし，allocate 文で割り付けられた配列を，解放しないで再度 allocate 文で割り付けようとすると実行時エラーになります．また，割り付ける前にその配列を使用しようとしても実行時エラーになります. そこで,配列がすでに割り付けられているかどうかを確認する関数 allocated が用意されています. allocated は論理型の値を返す関数で，allocatable 属性を持つ配列を引数に与えると，その配列がすでに割り付けられていれば“真”，割り付けられていなければ“偽”を返します. 論理型の関数は，そのまま if 文の条件として使うことができるので，例えば次のように利用することができます.

```
real, allocatable :: a(:)
integer i,n
.....
if (allocated(a)) then
  a(:) = (/(100*i,i=1,n)/)     ！代入１
 else
  allocate ( a(n) )
  a(:) = (/(200*i,i=1,n)/)     ！代入２
endif
```

この場合，配列 a がすでに割り付けられていれば“代入１”を実行し，割り付けられていなければ，allocate 文で割り付けてから“代入２”を実行します.

4.　数値計算を補助するフリーのサブルーチンライブラリ

本稿における Fortran プログラミングの説明は以上です．これまで紹介した文法を使って色々なプログラムを書いてみて下さい．ただ，複雑な計算アルゴリズムを使ったプログラムをすべて一から書くのはかなりの手間がかかります．アルゴリズムによってはインターネット上にサブルーチンライブラリとして公開されているので，これらを利用してプログラムを作ると手間が省け

10 ただし，サブルーチンを複数回コールするために固定メモリ領域に所属させることを利用する場合には，配列名の宣言に save 属性を加えて，配列名も固定メモリ領域に所属させなければなりません.

ます．ここでは筆者が使ったことのあるライブラリの中からお勧めをいくつか紹介します[11]．

4.1　高速フーリエ変換

偏微分方程式の解法やデータ解析で良く使う高速フーリエ変換（Fast Fourier Transform, FFT）は，アルゴリズムが複雑で，効率の良いプログラムを自作するのは大変です．筆者はもっとも簡単なレベルのプログラムを作ったことはありますが，最近はネットで公開されているものを使っています．お勧めなのは，FFTW（Fastest Fourier Transform in the West）です．

```
http://www.fftw.org/
```

このライブラリには，Fortran から使用できるサブルーチンが入っています．

4.2　多倍長計算

Fortran での倍精度実数は 8byte で，有効数字は 15 桁程度です．４倍精度実数は 16byte で有効数字が 30 桁程度に増加しますが，それ以上は無理だし，コンパイラによっては４倍精度実数が使えないこともあります．筆者は多項式の根を計算するときに高精度計算が必要になったことがあります．数十次の多項式を計算するときは，最大次数の項と最小次数の項との数値の差が非常に大きくて，桁落ちが深刻な問題になるからです．

このときに便利なのが多倍長計算ライブラリです．これは，任意の精度で実数計算をすることができるものです．通常の計算より処理に時間がかかりますが，精度優先のときにはやむを得ません．ネット上には何種類か公開されていますが，筆者が使って便利だったのは，FMLIB です．

```
http://myweb.lmu.edu/dmsmith/FMLIB.html
```

これは Fortran のモジュールになっていて，四則演算を“＋－＊／”の記号で書けるし，sin や cos などの関数も充実しています．本稿では説明していない構造体の知識が少しだけ必要ですが，変数宣言程度なのでそれほど難しくはありません．筆者は，倍精度計算用に作った多項式の根を計算するプログラムを，それほど大きな変更をすることなく多倍長で動作させることができました．

4.3　乱数発生

計算機シミュレーションには，時間発展の方程式を解くことで未来を予測する決定論的手法と，サイコロを振って確率的に将来の可能性を探る確率論的手法があります．後者の確率論的手法で，サイコロに相当する働きをするのが乱数です．Fortran には，乱数発生用の組み込みサブルーチン random_number が用意されているので，

```
call random_number(x)
```

のように書けば，乱数 x を得ることができます．しかし，乱数は周期長が問題で，周期の短い乱数を使うと良いシミュレーション結果が得られません．random_number では周期長が不足するときにお勧めなのは，周期が非常に長い乱数として有名な，Mersenne Twister です．これを考案した松本眞さんが以下のサイトでプログラムを公開されています．

```
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/mt.html
```

松本さんが公開されているのはＣ言語で書かれたものですが，リンク集の中に Fortran 用の公開先が入っています．

4.4　数値計算ライブラリ

連立方程式の解法や数値積分など，一般的な数値計算ライブラリがあれば，数値計算のプログラムを作るのが楽になります．ただ，残念ながら筆者はフリーの良いライブラリを知りません．ネットで公開されていて使えそうなものに，GSL（GNU Scientific Library）があります．

```
http://www.gnu.org/software/gsl/
```

---

11 サイバーサイエンスセンターの計算機には商用のサブルーチンライブラリがインストールされています．詳しくは，“http://www.ss.isc.tohoku.ac.jp/super/library.html”をご覧下さい．

ただし，基本的にＣ言語用なので，Fortran で使うには若干テクニックがいるようです．また，ちょっと使ってみましたが，必ずしも速くありません．ベッセル関数などの特殊関数は，その高速バージョンを公開しているサイトからダウンロードした方が良いと思います．筆者は，数値計算の教科書を読んで原理から自作するか,参考文献[2,3]に掲載されているものをコピーして使うことが多いです.

4.5　グラフィックライブラリ

計算機シミュレーションは物理的に起こっている現象を模擬するものですが，出力される数値だけで現象を読み取るのは容易ではありません．そこで必要不可欠なのが可視化ソフトです．フリーの可視化ソフトとして有名なものに gnuplot があります.

```
http://www.gnuplot.info/
```

プログラムの結果をファイルに保存しておけば，gnuplot からそれを読み込んで，様々なグラフに加工して表示することができます.

最近のパソコンは基本操作が全てビジュアルであり,グラフィック出力機能は OS に組み込まれています．Fortran にはグラフィック出力に関する文法はありませんが，他の言語で書かれたグラフィックライブラリとリンクすれば，Fortran プログラムからサブルーチンをコールする形式で図を描くことが可能です．その一つに，筆者が作った Frames があります.

```
http://www.pp.teen.setsunan.ac.jp/frames/
```

Frames は，Fortran で使うためのグラフィックツールとして開発してきたもので，本稿で説明した知識だけで簡単に２次元や３次元のグラフを描くことができます．また，描画した図形をイメージやアニメーション形式で出力して，プレゼンテーションに使うことも可能です．良かったら試してみて下さい.

連載を終えるに当たって

今回は，前回までに説明した書式の拡張と，文字列や配列計算式の使い方を説明しました．Fortran には，この他にも色々な文法が用意されていて，構造体を使ったオブジェクト指向的なプログラミングも可能ですし，最新の Fortran2008 には並列処理の記述まで入っています．しかし，筆者が計算機シミュレーションをするために使っている文法は，この３回の内容でほぼ全てです．数値計算や計算機シミュレーションは，計算機を効率良く動かしさえすればそれで良いのであり，新しい文法は必ずしも必要ではありません．コンピュータの動作とプログラムとの関係が理解できれば，後はいくらでも複雑な計算をさせることができます．Fortran を使ってコンピュータを“スマートに”使いこなし，数値計算や計算機シミュレーションをもっともっと楽しんでもらえればと思います.

本稿は，筆者が摂南大学の卒研生向けに書いた Fortran のテキストがベースになっています．これを，大阪大学レーザーエネルギー学研究センターの福田優子氏が Web で公開できる Fortran テキストを探しているというので提供したところ，回り回ってこちらで連載させていただくことになりました．最後に，このような機会を与えて下さった福田優子氏に深い感謝の意を表して，本稿を終えたいと思います.

参考文献

[1] 入門 Fortran90 実践プログラミング，東田幸樹・山本芳人・熊沢友信，ソフトバンク，1994 年
[2] Numerical Recipes in Fortran 77, Second Edition, W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, Cambridge University Press, 1996.
[3] Numerical Recipes in Fortran 90, W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, Cambridge University Press, 1996.