東北大学サイバーサイエンスセンター講習会
第8回 Fortran入門（2021年9月10日）

第２部

手順のくり返しと条件分岐

日本原子力研究開発機構　田口 俊弘

第２部の講習内容

１．配列の使い方

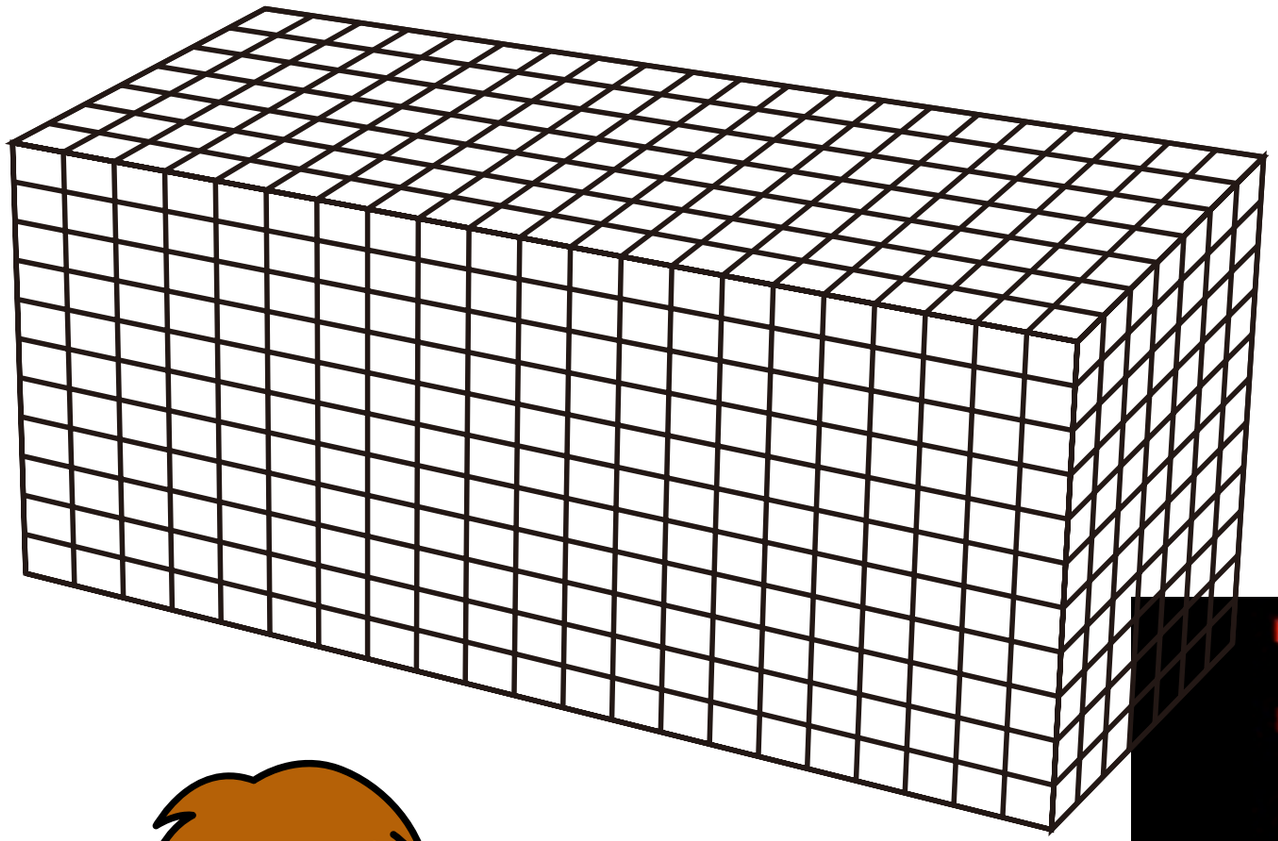
２．手順のくり返し

３．条件分岐

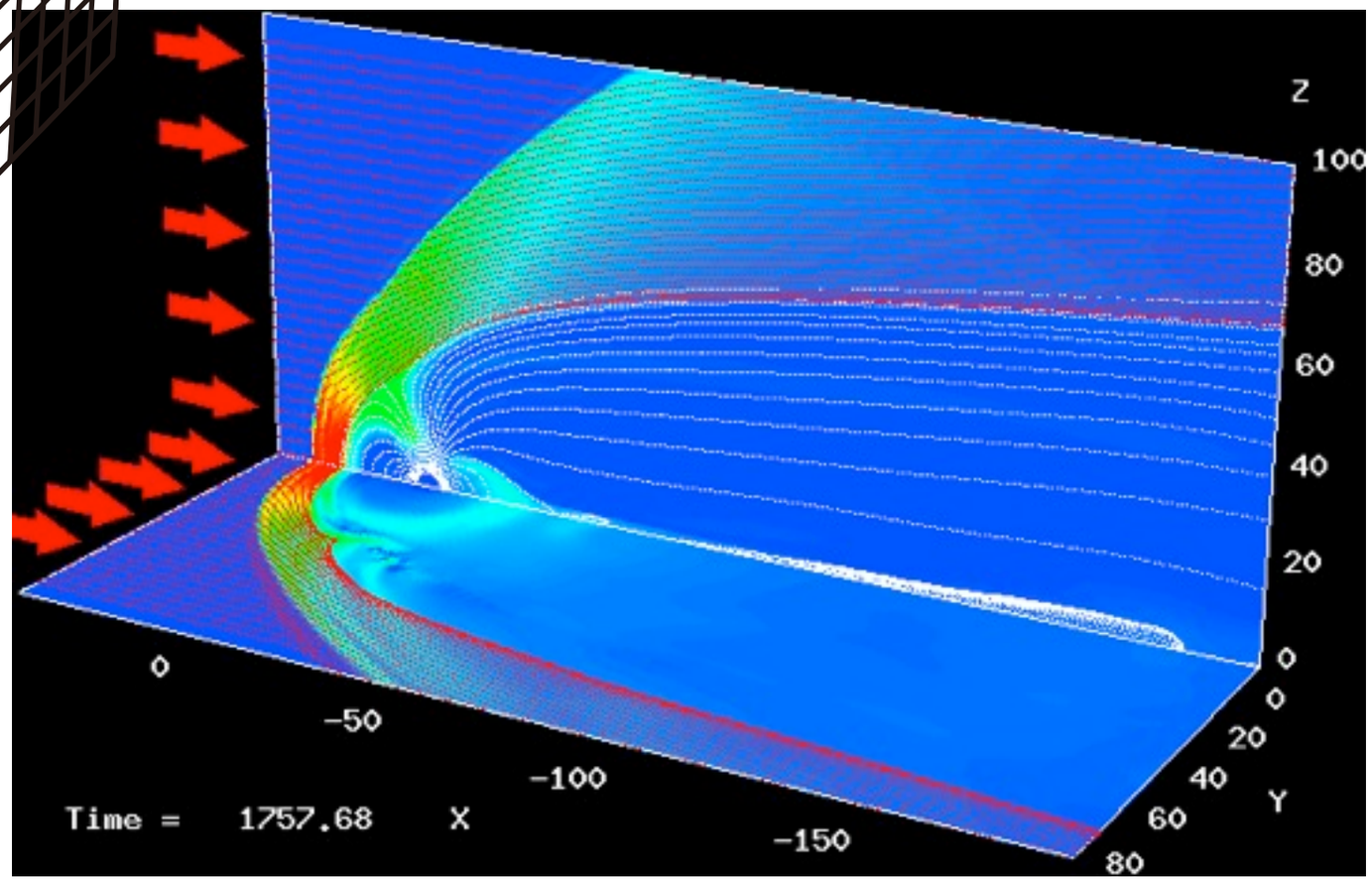
４．無条件ジャンプ

付録：知っておくと便利な文法２

１．　配列の使い方



```
real p(20,6,10)
```



配列とは複数のメモリを番号指定で取り扱う変数

- 数列 $a_1, a_2, \ldots, a_{10}$ のように，複数の同じ数値型のデータをひとまとめにして扱う「変数」の一種
- 配列名に「要素番号（添字）」をかっこで付加してデータを指定する

```
１次元配列　a(1)，b1(15)，cde(100) 等
```

◎ 指定要素数が２個以上の多次元配列もある（後で説明）

- 配列は，宣言文で型宣言をする時に，「要素番号の最大値」を指定する

```
real a(10),b(20)          実数型配列
integer node(100)         整数型配列
```

- 要素番号で指定された変数（配列要素）は，宣言時の数値型を持った１個の変数として利用できる

```
a(15) = b(2)*10 - cos(2*a(3))
node(i*2) = a(15)**3 - a(i-3)
```

★ 要素番号には変数や数式を使っても良い（左辺でも可能）

★ ただし，整数型の変数や数式でなければならない

配列の要素番号の使用可能範囲

基本的には，１～宣言数まで

例えば，real a(5) の宣言の場合は
　a(1),a(2),a(3),a(4),a(5)　の5個が使用可能

◎ Fortranでは，コロン（：）で数値をはさんだ

配列名(下限:上限)

の形式を使えば，下限を指定した宣言ができる

例えば，real ac(-2:2) の宣言の場合は
　ac(-2),ac(-1),ac(0),ac(1),ac(2)　の5個が使用可能

プログラムを書いてみよう

2-A　配列を使った３次元ベクトル計算

例題2-A

配列 a(3) と b(3) を３次元ベクトル $\boldsymbol{A}=(A_x, A_y, A_z)$ と
$\boldsymbol{B}=(B_x, B_y, B_z)$ の要素を保存する配列と考えて以下の計算をせよ

① $\boldsymbol{A}$と$\boldsymbol{B}$ の要素として適当な値を設定する

② $\boldsymbol{A}$の長さ $|\boldsymbol{A}|$ と$\boldsymbol{A}$と$\boldsymbol{B}$ の内積 $\boldsymbol{A}\bullet\boldsymbol{B}$ を計算して出力する

③ $\boldsymbol{A}$と$\boldsymbol{B}$ の外積ベクトル $\boldsymbol{A}\times\boldsymbol{B}$ を計算し，各要素を配列c(3)に代入して出力する

参考

$$|\boldsymbol{A}|=\sqrt{A_x^2+A_y^2+A_z^2}$$

$$\boldsymbol{A}\bullet\boldsymbol{B}=A_xB_x+A_yB_y+A_zB_z$$

$$\boldsymbol{A}\times\boldsymbol{B}=(A_yB_z-A_zB_y, A_zB_x-A_xB_z, A_xB_y-A_yB_x)$$

2-A　配列を使った３次元ベクトル計算 の解答例

```
program answer2a
    implicit none
    real a(3), b(3), c(3), al, ip
    a(1) = 3;  a(2) = 1; a(3) = -2
    b(1) = -1; b(2) = 5; b(3) = 3
    al = sqrt(a(1)**2+a(2)**2+a(3)**2)
    ip = a(1)*b(1)+a(2)*b(2)+a(3)*b(3)
    print *,'|A| = ', al, '   A.B = ', ip
    c(1) = a(2)*b(3)-a(3)*b(2)
    c(2) = a(3)*b(1)-a(1)*b(3)
    c(3) = a(1)*b(2)-a(2)*b(1)
    print *,'AxB = ', c
end program answer2a
```

☆書き方のポイント☆

★ 短い同レベルの代入文はセミコロン「；」でつないで書いてもよい

★ 配列を出力するときは「配列名」だけを書くと，宣言した数の要素全てを横に並べて出力する

配列とは，連続したメモリのこと

real a(8)の宣言では

real a(8)

メモリアドレス	メモリ領域
101:	a(1)
102:	a(2)
103:	a(3)
104:	a(4)
105:	a(5)
106:	a(6)
107:	a(7)
108:	a(8)
....	

a →

real ac(-3:4)の宣言では

real ac(-3:4)

メモリアドレス	メモリ領域
201:	ac(-3)
202:	ac(-2)
203:	ac(-1)
204:	ac(0)
205:	ac(1)
206:	ac(2)
207:	ac(3)
208:	ac(4)
....	

ac →

◎「配列名」は，メモリの先頭アドレスを示す

a は a(1)，ac は ac(-3) を示す

◎ 配列要素番号は先頭から何番目かを指定する数字

多次元配列

- 行列要素 $a_{11}, a_{21}, a_{12}, a_{22}$ のように，複数の要素番号で変数を指定することもできる．これを「多次元配列」という
- 識別する要素番号の数で「次元」が異なる

```
２次元配列　a(1, 1)，b(3, 2), c(10, 25) など
３次元配列　f(1, 1, 1), g(15, 3, 2), h(2, 5, 6) など
```

- 多次元配列は，宣言文で型宣言をする時に，それぞれの次元の「要素番号の最大値」を指定する

```
real a(10, 10), b(20, 30), f(0:5, -2:3, 5)　　実数型配列
```

◎ f のように，それぞれの次元で別の下限を指定することも可能

◎ 利用可能な要素数は，各次元の要素数の積になる
例えば，real a(10, 10) という宣言なら100個の配列要素が使える

- 利用方法は１次元配列と同じで，あくまでも１個の変数である

```
a(1, 2) = b(2, 3)*10 - cos(2*a(3, 4))
g(i, j) = f(i+2*j, 0, 2)*a(j, k)
```

多次元配列は，左の要素が先に進むように１次元的に並んでいるメモリ

real b(3,2)の宣言では

real b(3,2)	
メモリアドレス	メモリ領域
301:	b(1,1)
302:	b(2,1)
303:	b(3,1)
304:	b(1,2)
305:	b(2,2)
306:	b(3,2)
....	

b →

「配列名」が，メモリの先頭アドレスを示すのは同じ

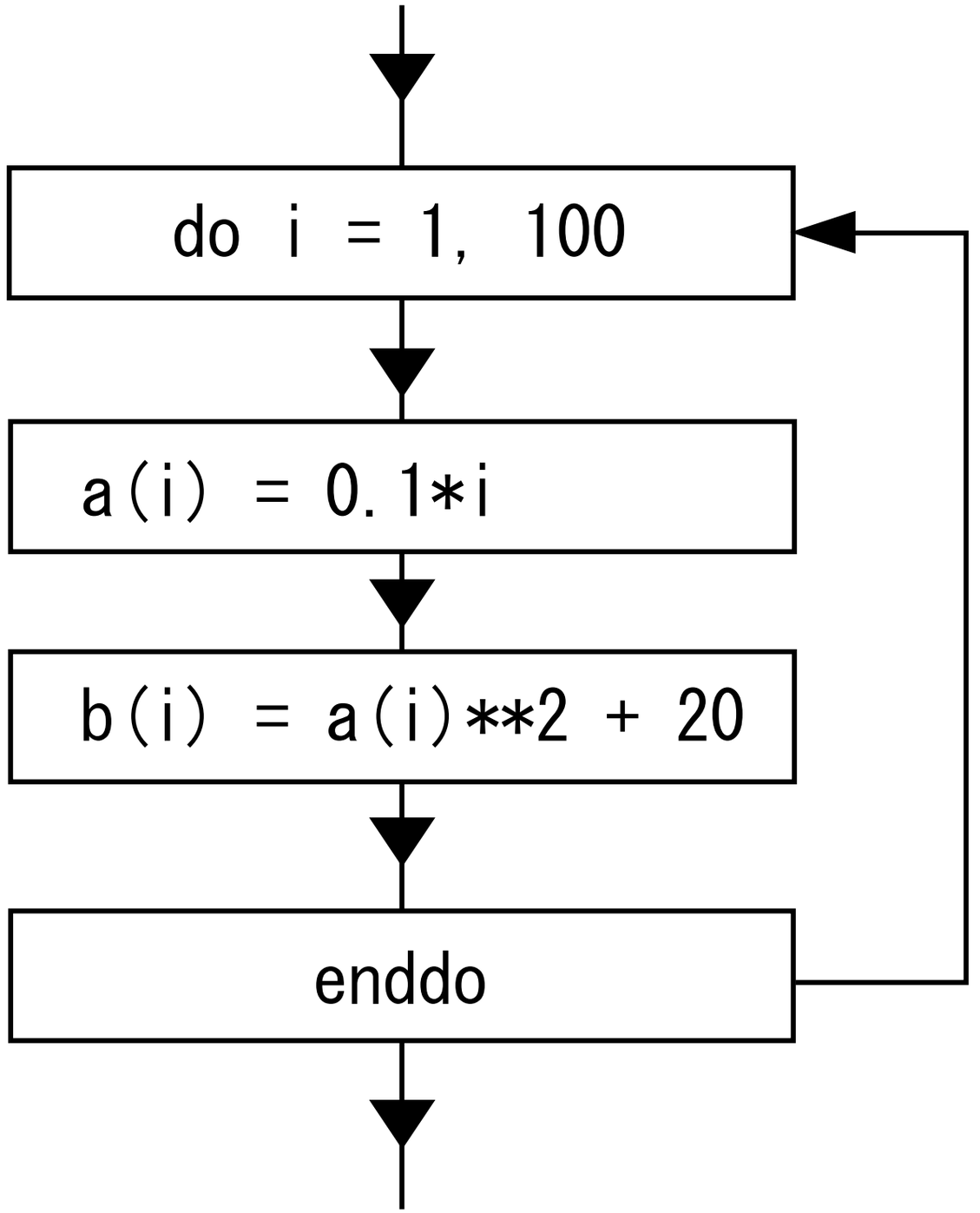
コンピュータはメモリの順番に計算すると効率がよいので，並びは覚えておいた方が良い

◎１次元的な順番を数式で表せば，次のようになる

real b(n1,n2)の配列宣言で，b(i,j)は先頭から数えて i+n1*(j-1)番目

real b(n1,n2,n3)の配列宣言で，b(i,j,k)は先頭から数えて i+n1*(j-1+n2*(k-1))番目

2．手順のくり返し



doブロックによる手順のくり返し

- 同じパターンの動作を所定の回数くり返す時にはdoブロックを使う
- 基本的なdoブロックは，do文とenddo文でくり返したい手順をはさむ

```
do 整数型変数 = 初期値，終了値      ... do文
  .......
  .......          この間をくり返す（doループともいう）
  .......
enddo                              ... enddo文
```

◎ 初期値，終了値は，整数型変数や整数型の計算式でも良い

doブロックの動作は，以下の通り

① 整数型変数（カウンタ変数）に初期値を代入する

② カウンタ変数と終了値を比較して，終了値以下ならブロック内部を実行

③ enddo文に到達したら，カウンタ変数に1を加えて②に戻る

④ ②に戻ったときに，カウンタ変数が終了値より大きかったら
　くり返しは終了し，enddo文の次の文から実行を続ける

☆ 初期値が終了値より大きかったら，doブロックは１回も計算されない

プログラムを書いてみよう

2-B　doブロックを使ったくり返し計算

例題2-B

２個の整数型１次元配列 m(15) と n(15) を用意して，doブロックを使ったくり返しにより以下の計算をせよ

① m(1)～m(15)に，1～15の数字を順番に代入する

② n(1)～n(15) に，それぞれ m(1)～m(15) の２乗を代入する

③ 配列 n の要素を3個ごとに横並びで出力する

☆プログラム作成時のコツ☆

★ doブロック内部は，ブロックの範囲が一目でわかるように字下げする

```
do i = 1, n
  b(i) = a(i)
  c(i) = a(i)*sin(b(i))
enddo
```

↕ この間を右にずらす

2-B　doブロックを使ったくり返し計算 の解答例

```
program answer2b
    implicit none
    integer m(15), n(15)
    integer i
    do i = 1, 15
      m(i) = i
    enddo
    do i = 1, 15
      n(i) = m(i)**2
    enddo
    do i = 0, 4
      print *,n(i*3+1),n(i*3+2),n(i*3+3)
    enddo
end program answer2b
```

☆書き方のポイント☆

★ 最初のdoブロックと２番目のdoブロックは次のように一つにできる

```
    do i = 1, 15
      m(i) = i
      n(i) = m(i)**2
    enddo
```

★ ３個ずつ出力する方法は，わかりにくいのであまり良いとはいえない

増分値付きのくり返しも可能

```
do 整数型変数 = 初期値，終了値，増分値
  .......
  .......
enddo
```

◎ 増分値は，整数型変数や整数型の計算式でも良い

この場合，do文の動作は，以下の通り

① 整数型変数（カウンタ変数）に初期値を代入する
② カウンタ変数と終了値を比較して，終了値以下ならブロック内部を実行
③ enddo文に到達したら，カウンタ変数に「増分値」を加えて②に戻る
④ ②に戻ったときに，カウンタ変数が終了値より大きかったら
　くり返しは終了し，enddo文の次の文から実行を続ける

例えば，例題2-Bの解答例の最後のdoブロックは以下のように書ける

```
do i = 1, 15, 3
  print *,n(i),n(i+1),n(i+2)
enddo
```

増分値はマイナスでも良い

```
do 整数型変数 = 初期値，終了値，負の増分値
  ．．．．．．．
  ．．．．．．．
enddo
```

この場合，do文の動作は以下の通り（**赤色**の部分が異なる）

① 整数型変数（カウンタ変数）に初期値を代入する

② カウンタ変数と終了値を比較して，**終了値以上**ならブロック内部を実行

③ enddo文に到達したら，カウンタ変数に「**負の増分値**」を加えて②に戻る

④ ②に戻ったときに，カウンタ変数が終了値より**小さかったら**
　くり返しは終了し，enddo文の次の文から実行を続ける

☆ 初期値が終了値より小さかったら，doブロックは１回も計算されない

例えば，

```
do m = 10, 1, -1
  a(m) = m
enddo
```

は，a(10)=10，a(9)=9，…，a(1)=1という動作になる

プログラムを書いてみよう

2-C　doブロックを使った合計計算

例題2-C

次の漸化式で与えられる数列（フィボナッチ数列）をi = 1～30まで計算し，それらの平均を計算して出力せよ

$$a_1 = 1,\ a_2 = 1$$

$$a_i = a_{i-1} + a_{i-2} \qquad i > 2$$

2-C　doブロックを使った合計計算 の解答例

```
program answer2c
    implicit none
    real a(100), sum
    integer i, imax
    imax = 30
    a(1) = 1;    a(2) = 1
    do i = 3, imax
      a(i) = a(i-1) + a(i-2)
    enddo
    sum = 0
    do i = 1, imax
      sum = sum + a(i)
    enddo
    print *,' Average = ', sum/imax
end program answer2c
```

☆書き方のポイント☆

★ 要素番号の最大値「30」は変数（この例ではimax）にして共有すれば，プログラムが修正しやすくなる

★ この例では，配列 a の宣言要素数を 100 にしているので，imax を100までの数字に変更しても計算することが可能

合計を計算するdoブロックを覚えておこう

例えば，a(1)+a(2)+a(3)+...+a(10)を計算したいときは，

```
sum = 0
do m = 1, 10
   sum = sum + a(m)
enddo
```

のように書けばよい，これは，次のような動作だから

```
sum = 0
m = 1
sum = sum + a(m)
m = m + 1
sum = sum + a(m)
m = m + 1
sum = sum + a(m)
m = m + 1
 .....
```

☆ なお，最初の sum=0 を忘れないように！（ゼロリセット）

◎ 加算を乗算にすれば，階乗計算もできる（初期値は１です）

doブロックの中にdoブロックを入れても良い
～多重doブロック～

例えば，次のようにdoブロック中にdoブロックを入れることができる

```
do k = 1, 100
  a(k) = k**2
  do m = 1, 10
    b(m,k) = m*a(k)**3
    c(m,k) = b(m,k) + m*k
  enddo
  d(k) = a(k) + c(10,k)
enddo
```

☆ただし，内側のカウンタ変数と外側のカウンタ変数は
　異なるものを使わなければならない

★Fortranではカウンタ変数をブロック内部で変更するのはエラー！

◎ 内側のブロックが先にくり返すことになる

プログラムを書いてみよう
2-D　行列積の計算

例題2-D

３個の実数型２次元配列 a(5,5) ， b(5,5) ， c(5,5) を用意し，
配列の全ての要素に対して以下の計算をせよ

① a(i,j) に $i \times j$ の計算結果を代入する

② b(i,j) に $2i-3j$ の計算結果を代入する

③ a(i,j) と b(i,j) を正方行列 AとB の行列要素 a_{ij}, b_{ij} と考えて，
AとB の積の行列要素 c_{ij} を配列要素 c(i,j) に代入する

④ 配列 c の要素を5個ごと行列のように並べて出力する

☆n行n列の行列 A と B の積 C の要素☆

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj}$$

2-D　行列積の計算 の解答例


```
program answer2d
    implicit none
    real a(5,5), b(5,5), c(5,5), sum
    integer i, j, k, imax
    imax = 5
    do j = 1, imax
      do i = 1, imax
        a(i,j) = i*j
        b(i,j) = 2*i - 3*j
      enddo
    enddo
    do j = 1, imax
      do i = 1, imax
        sum = 0
        do k = 1, imax
          sum = sum + a(i,k)*b(k,j)
        enddo
        c(i,j) = sum
      enddo
    enddo
    do i = 1, imax
      print *, c(i,1),c(i,2),c(i,3),c(i,4),c(i,5)
    enddo
end program answer2d
```

☆書き方のポイント☆

★ 多次元配列の計算では，左側の要素が先に進むようにする

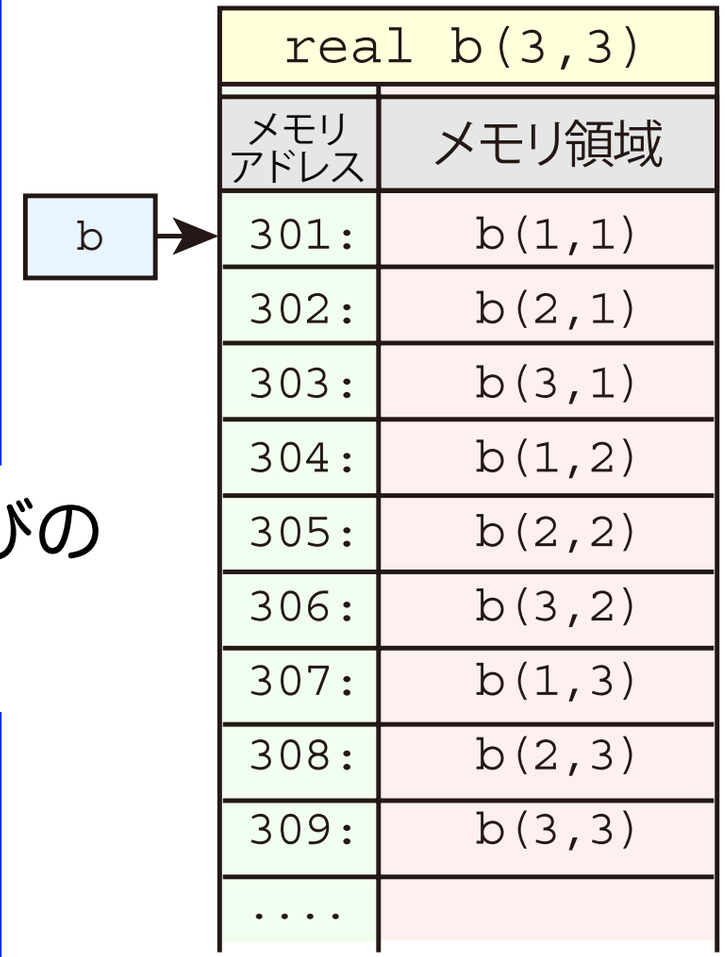
★ 最後のprint文は，以下のように書くこともできる（付録2C参照）

```
    print *, (c(i,j), j=1,imax)
```

多次元配列の計算は左側の要素を先に進める方が良い

多次元配列は，メモリの並び順で処理をした方が速い

```
real b(3,3)
integer m,n
do n = 1, 3
  do m = 1, 3          ! 左の要素を先に進めると速い
    b(m,n) = m*n
  enddo
enddo
```



この場合は，b(1,1),b(2,1),b(3,1),...というメモリ並びの順で計算するので速い

```
real b(3,3)
integer m,n
do m = 1, 3
  do n = 1, 3          ! 右の要素を先に進めると遅い
    b(m,n) = m*n
  enddo
enddo
```

この場合は，b(1,1),b(1,2),b(1,3),... のようにメモリを飛び飛びに行ったり来たりしながら計算するので遅い

doブロックを使うときの注意

- カウンタ変数は，doブロック内で変更してはいけない
- カウンタ変数は，enddo文に到達した段階で増分値を加えるので，doブロックを終了したときのカウンタ変数の値は「終了値と異なる」
- ジャンプ命令（後で説明）を使って，doブロックから外へ出ても良いが，外から中に入ることはできない
- 初期値，終了値，増分値はdoブロックの開始時に決定されるので，
  doブロック内部で変更しても無関係

例えば，

```
j = 10
k = 1
do i = 1, j, k
   print *,i,j,k
   j = 20
   k = 2
enddo
```

のように書いても良いが，終了値や増分値は最初決めた数値でくり返す

doブロックのテクニック

定数は，doブロックの外で計算しておくと速くなる．例えば，

```
do i = 1, 10000
  a(i) = b(i)*c*f**2
  b(i) = sin(x)*a(i)
  y(i) = b(i)/d
enddo
```

は，以下のように書いた方が速い

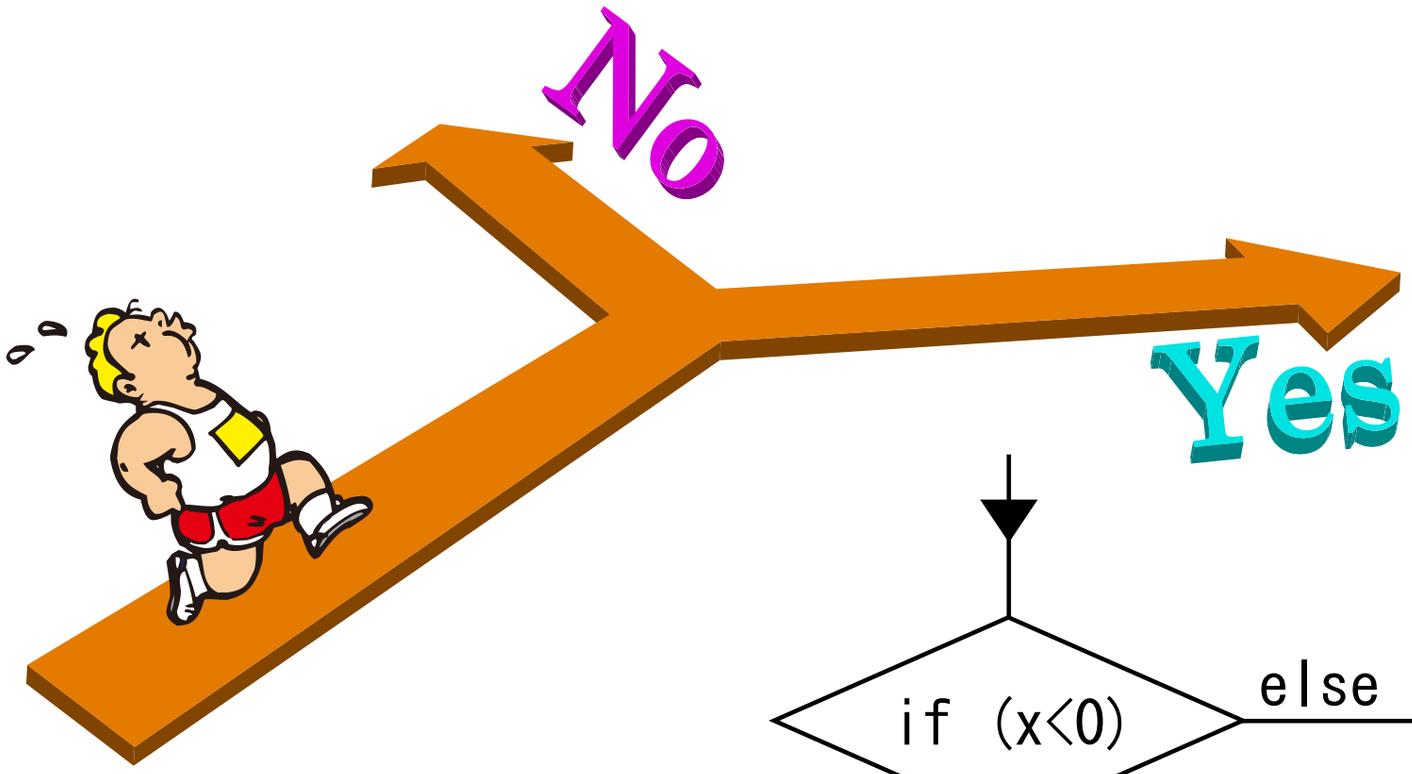
```
cf = c*f**2
sx = sin(x)
di = 1/d
do i = 1, 10000
  a(i) = b(i)*cf
  b(i) = sx*a(i)
  y(i) = b(i)*di
enddo
```

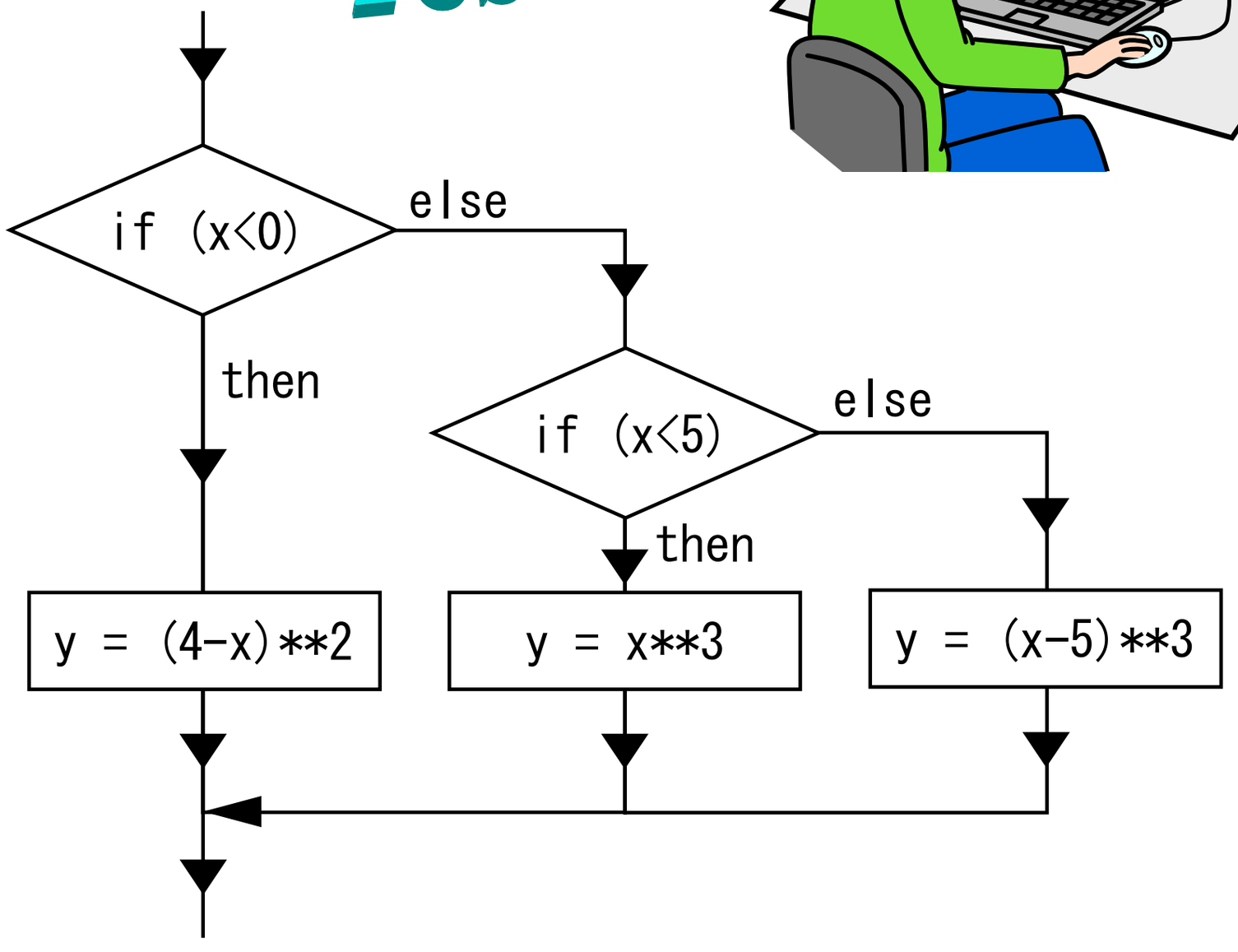
★ c*f**2やsin(x)の計算部分はdoブロック内部では変化しないので，
あらかじめ変数に代入しておいたのを使えば，無駄な計算をしない

★ d による割り算は逆数を掛ける方が速い

★ もっとも最近はあまり神経質になる必要はないかもしれない

3．条件分岐





if 文による条件分岐

条件に応じて異なる動作をさせるときは if文を使う

単純if文

```
if（条件）実行文
```

★ 条件が「真」の時，右の実行文を実行する
★「偽」の時は，何もせず次の文に移る
★ １個の実行文しか実行できない

ブロックif文

```
if（条件）then                    ... ブロックif文
  ........
  ........                  条件が真のときのみ，この間を実行する
  ........
endif                             ... endif文
```

★「真」の時，ブロック内の実行文を実行する
★「偽」の時は，何もせず endif文の次の文に移る
★ 複数の実行文を実行できる

ブロックif 文による条件分岐

- ブロックif文は「偽」の時に別の動作をさせることもできる

else付きのブロックif文

```
if（条件）then
  .......          条件が「真」のとき実行
 else                                  ... else文
  .......          条件が「偽」のとき実行
endif
```

```
if（条件1）then
  .......          条件1が「真」のとき実行
 else if（条件2）then                    ... else if文
  .......          条件1が「偽」で条件2が「真」のとき実行
 else
  .......          条件1も条件2も「偽」のとき実行
endif
```

★ else文は最後の１回だけだが，else if文は複数指定することができる
★ if ブロックの中に別の if文や ifブロックを入れることもできる
★ if ブロックの中にdoブロックを入れても良いし，その逆も可能

比較条件の書き方

- 比較条件の書き方は以下の通り

比較条件記号	記号の意味	使用例
==	左辺と右辺が等しいとき	x == 10
/=	左辺と右辺が等しくないとき	x+10 /= y-5
>	左辺が右辺より大きいとき	2*x > 1000
>=	左辺が右辺以上のとき	3*x+1 >= a(10)**2
<	左辺が右辺より小さいとき	sin(x+10) < 0.5
<=	左辺が右辺以下のとき	tan(x)+5 <= log(y)

★ 比較条件は，両辺に数式を書いても良い

★ 両辺の数値型が合っていないときは，演算と同じ方法で数値型を合わせてから比較する

★ 実数を比較するとき，特にイコールか否かの判定には注意が必要

```
real x
x = 0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1
if (x == 1.0) print *,x   ! これは出力されない可能性がある
```

プログラムを書いてみよう

2-E　判別式に応じた２次方程式の解計算

例題2-E

変数 a, b, c に適当な数値を代入して以下の２次方程式の２解を計算せよ

$$ax^2 + bx + c = 0$$

① まず，変数 a, b, c に適当な数値を代入する

② 次に，判別式 $D = b^2 - 4ac$ を計算する

③ D が正なら２解を計算して出力する

④ D が０なら１解だけ計算して出力する

⑤ D が負なら複素解なので，その実部と虚部を計算して出力する

2-E　判別式に応じた２次方程式の解計算 の解答例

```
program answer2e
    implicit none
    real a, b, c, d, x1, x2
    a = 1;  b = 1;  c = 1
    d = b*b-4*a*c
    if (d > 0) then
      x1 = (-b+sqrt(d))/(2*a)
      x2 = (-b-sqrt(d))/(2*a)
      print *, 'x1, x2 = ', x1, x2
     else if (d == 0) then
      x1 = -b/(2*a)
      print *, 'x1 only = ', x1
     else
      x1 = -b/(2*a)
      x2 = sqrt(-d)/(2*a)
      print *, 'x = ', x1, '+- i*', x2
    endif
end program answer2e
```

☆書き方のポイント☆

★ if ブロック内部も字下げをする

★ 複素解のときは，出力を複素数的に表示している

```
 x =  -0.500000000000000 +- i*  0.866025403784439
```

論理式の書き方

- 比較条件は，論理式でつなぐこともできる

論理演算記号	演算の意味	使用例
“条件１”.or.“条件２”	“条件１”または“条件２”のとき	x ＞ 0 .or. y ＞ 0
“条件１”.and.“条件２”	“条件１”かつ“条件２”のとき	x ＞ 10 .and. 2*x <= 50
.not.“条件”	“条件”を満足しないとき	.not.(x ＜ 0 .and. y ＞ 0)

例えば，

```
if (i ＞ 0 .and. i <= 5) then
   a = 10.0
 else
   a = 0.0
endif
```

のように書くことができる

なお，この条件を

```
if (0 ＜ i <= 5) then
```

とは書けません！

プログラムを書いてみよう

2-F　倍数の個数計算

例題2-F

1~10000 の整数の中で，次の２条件を同時に満足する整数の個数を計算し，その個数，および全体の何パーセントがその条件を満足するかの割合を出力せよ

① 3 で割ったときの余りが 0 になる

② 7 で割ったときの余りが 1 になる

☆プログラム作成時のヒント☆

★ 整数 m を整数 n で割ったときの余りは，関数 mod(m,n) で計算する

```
n = mod(10,3)
```

◎ この結果は，10を3で割ったときの余りなので，n=1

★ 整数 m と整数 n の比を実数で計算したいときには，整数 n を実数に変換する関数 real(n)を使う

2-F　倍数の個数計算 の解答例

```
program answer2f
    implicit none
    integer i, imax, n
    imax = 10000
    n = 0
    do i = 1, imax
      if (mod(i,3)==0 .and. mod(i,7)==1) n = n + 1
    enddo
    print *,' n = ', n, '  rate = ',real(n)/imax*100,'%'
end program answer2f
```

☆書き方のポイント☆

★ １行の短い実行文の場合には，単純 if 文で良い

★ ２個の条件を同時に満足するときだけ実行したいときは，
.and. で条件をつなぐ

★ 整数 n を整数 imax で割ると切り捨てになるので，どちらかを
実数化しなければならない

★ 横着をして 100.0*n/imax と書く手もある（順番が重要）

論理式の優先順位，その他

論理演算の優先順位は　.not. ＞　.and.　＞　.or.

例えば，

```
if (i > 100 .or. i > 0 .and. i <= 5) then
```

は，

```
if (i > 100 .or. (i > 0 .and. i <= 5)) then
```

と同じである

★ もっとも，四則演算のように明白ではないので，優先したい方を常にかっこで囲った方がわかりやすい

★ 余りを計算する関数modは，くり返しの中で何回かに１回の一定間隔で出力させたいときなどにも使えるので覚えておこう！

```
do m = 1, 10000
   ......
  if (mod(m,10) == 0) print *,m,x,y
   ......
enddo
```

４．無条件ジャンプ

goto文による無条件ジャンプ

- 動作の流れを強制的に変える
- goto 文と「文番号」で指定する

goto文

```
    goto 文番号
```

文番号

```
文番号　実行文
```

★ 文番号は重複してはならない
★ 順番は無関係だが，下方に行くほど増加する方がわかりやすい
★ 途中で挿入する可能性を考慮して，最初は10ごととか，
　適当に間隔を空けた番号にした方が良い

◎ 動作のない continue文を使うと位置の指定だけ行うことができる

```
    文番号 continue
```

goto文による無条件ジャンプの使用例

例えば,

```
    cd = 10
    goto 11                          ! 文番号11の行へジャンプ
    cd = 50
    ab = 20                          ! この間は無視される
    ij = 1
11   ab = 1000                       ! この行へジャンプ
```

★ このプログラムには無駄があるが，エラーを見つけるときなど
一時的に使うことはある

```
22   ab = 200                        ! この行へジャンプ
   cd = cd + ab - ef
   ef = 10
   goto 22                           ! 文番号22の行へジャンプ
   ij = 1                            ! いつまでたってもこの行は実行されない
```

★ これを「無限ループ」という

★ 無限ループはエラーの一つ，必ず条件を入れて抜けるようにする

★ くり返しはできる限りdoブロックで書くようにする

プログラムを書いてみよう

2-G　非線形方程式の解

例題2-G

$$x = \cos x$$

以下のような手順（逐次代入法）に基づいて，上の方程式の解の近似値を，goto文を利用したプログラムで計算せよ

① まず，変数 x に 0 を代入する

② 次に，変数 y に cos(x) の計算結果を代入する

③ 変数 x と y の差が 10^{-7}より小さかったら，その x を近似値として出力して終了する

④ もし大きい場合には，y の値を x に代入して②に戻る

☆逐次代入法の原理☆

x_0=0を初期値として，漸化式 $x_i = \cos(x_{i-1})$の極限値は，x=cos(x)の解となる

2-G　非線形方程式の解 の解答例

```
program answer2g
    implicit none
    real x, y
    x = 0
 10 continue
    y = cos(x)
    if (abs(x-y) < 1e-7) goto 20
    x = y
    goto 10
 20 continue
    print *,' x = ', x,'  err = ', abs(x-cos(x))
end program answer2g
```

☆書き方のポイント☆

★ 差を計算するときは，絶対値関数abs(x)を使う

★ print 文で出力するときに，誤差も一緒に出力して収束を確認

★ goto の行き先をcontinue文にしておくと，プログラムを修正するときに便利

☆実をいえば，あまりこういうプログラムは書かない方が良い

doブロック専用のジャンプ ：exit文

☆解答例の問題点☆

★ この問題は収束することがわかっているから良いが,
　収束しない可能性がある問題なら無限ループになることがある
★ 収束回数の最大値を決め，do ブロックで書いた方が良い
★ そのとき goto 文の代わりに exit 文を使うと文番号が不要になる

exit文：強制的にenddo文の次の行へジャンプ

★ 先ほどの解答例における実行文の部分は，goto 文なしで，以下のように書き直すことができる

```
x = 0
do i = 1, 10000      ! 最大10000回まで収束を確認する
  y = cos(x)
  if (abs(x-y) < 1e-7) exit
  x = y
enddo
print *,' x = ', x,'  err = ', abs(x-cos(x))
```

doブロック専用のジャンプ その他

多重do文の場合，外側のdoの外に出たいときは「ラベル」を使う

```
    sum = 0
out: do m = 1, mmax                    ! 外側のdoにラベル「out」
       do l = 1, lmax
         sum = sum + a(l,m)
         if (sum > 100) exit out       ! ラベル「out」の付いた外側のdoを出る
       enddo
     enddo out                         ! 外側のenddoにラベル「out」
     ave = sum/(mmax*lmax)
```

★ do文のラベルには「:」が必要だが，enddo文のラベルには不要

cycle文：強制的にenddo文へジャンプ（くり返しは継続）

```
    do m = 1, 10000
      sum = sum + a(m)
      if (sum > 100) cycle
      sum = sum*2              ! sumが100を越えるとこの行は実行されない
    enddo
```

★ cycle文は，enddo文にジャンプする動作になる

第２部　終了

付録：知っておくと便利な文法２

2A. do while文と無条件do文

do while文：「条件」を満足する限りくり返す

```
do while (条件)
  ........
  ........
enddo
```

★ 「条件」の書き方は，if 文と同じ

★ ブロック内部で条件が変わるようなプログラムにしなければならない

無条件do文：無条件でくり返す

```
do
  ........
  ........
enddo
```

★ ブロック内部に外に出るジャンプがなければ止まらない

★ 無限ループにならないように注意して使うこと

2B. 論理型

☆「真」または「偽」の２値のみ取り得る数値型の一種

論理型定数
.true.　（真）
.false.　（偽）

◎変数宣言はlogicalで行う

```
logical 変数１,変数２,...
```

◎比較や論理演算の判定結果を直接代入できる

```
logical k1, k2
integer m
m  = 6
k1 = m == 8
k2 = m > 5 .and. m < 10
```

◎論理型変数はif文の条件にそのまま入れられる

```
if (k1) print *,m
```

2C. do型出力

print文の「数値」の位置には，次のようなdo型出力が利用できる

```
（データ1, データ2, . . . , 整数型変数=初期値, 終了値）
```

★ do型出力は，それを記述した位置に，データの並びを展開して並べたのと等価

例えば，

```
print *, 5*x, (i, a(i), b(i), i=1, 3)
```

は，次の文と等価

```
print *, 5*x, 1, a(1), b(1), 2, a(2), b(2), 3, a(3), b(3)
```

★ 増分値を指定することもできる

```
（データ1, データ2, . . . , 整数型変数=初期値, 終了値, 増分値）
```

例えば，

```
print *, (a(i), i=10, 1, -2)
```

は，次の文と等価

```
print *, a(10), a(8), a(6), a(4), a(2)
```

★ do型出力は多重にすることもできる（内側が先に進む）

```
print *, ((a(i, j), j=1, 3), i=1, 3)
```

2D. ファイルへのデータ出力方法

- データをファイルに出力するときは write 文で行う

write文

```
write(nd,form) データ1, データ2,  ...
```

★ ndは整数（装置番号と呼ばれる，整数型ならば変数や式も可能）

★ ndは10以上が良い

★ form は出力形式指定（後で説明，とりあえずは＊で標準形式）

◎ 装置番号は適当に決めてよいが，一旦決めたらその番号への出力は，同じファイルへの出力となる

例えば，

```
write(20,*) x,y,z
```

と書けば，「fort.20」という名のファイルに出力される

☆ ただし「fort.」の部分はコンパイラに依存する

2E. ファイルからのデータ入力方法

- ファイルからのデータ入力は read 文で行う

read文

```
read(nd,*) データ1, データ2, ...
```

★ ndは整数（装置番号，整数型ならば変数や式も可能）

★ ndは10以上が良い

★ 入力形式を指定することもできるが，あまり利用しないので省略

例えば，

```
read(20,*) x,y,z
```

と書けば，「fort.20」という名のファイルから入力する

☆ **write文と同様に「fort.」の部分はコンパイラに依存する**

open文

open文を使えば，入出力のファイル名を指定できる

```
open(20,file='text.out')      ! 入出力ファイル名が text.out になる
```

open文の書式

open文

```
open(nd,file=name [, form=format] [, status=stat] [, err=num] )
```

- fileやformは，制御指定子と呼ばれ，以下のような意味を持つ

指定子	指定情報	指定子の意味と注意
nd	装置番号	整数を与える（整数型変数や整数式を与えることも可能） オープンした後，read 文や write 文の装置番号として使う
name	ファイル名	文字列で指定する（文字変数も可能） ファイル名は大文字・小文字を正しく指定する必要がある
format	ファイル形式	文字列で指定する 省略するとテキスト形式の入出力が仮定される バイナリ形式の入出力を使うときは「'unformatted'」を指定する
stat	ファイル情報	文字列で指定する 既存のファイルを使うときは「'old'」を，存在しないファイルを使うときは「'new'」を指定する 条件に合わないとエラーが発生する
num	文番号	エラーのときにジャンプする行の文番号を指定する 省略すると，エラーが起きたときにはプログラムが強制終了する

★ 詳細説明は省略します．解説書で説明していますのでそちらを参照して下さい．

2F. formatによる出力形式指定

標準形式の問題点

★ 有効数字一杯で出るので無駄が多い

★ 横に並べた数字が多いと強制的に改行される

★ 同じ並びの数字を複数出力した場合，位置がずれることがある

◎ 出力形式を変えたいときには format を指定する

★ print文やwrite文の「*」のところにformatを指定する

formatの指定方法１：文番号＋format文

```
      print 文番号, データ1, データ2,  ...
文番号 format(指定したいformat)
```

formatの指定方法２：文字列で直接書く

```
      print "(指定したいformat)", データ1, データ2,  ...
```

★ Fortranでは，２個の「"」ではさんだ文字も「文字列」である

★ format中には文字列を入れることが多いため，全体は" "で囲むと良い

formatの書き方

- 数値などの並びに対応して編集記述子をコンマで区切って並べる
- 編集記述子は，数値型に対応したものを書かなければならない
- format中の文字列には対応する数値はなく，そのまま出力される

例えば，文番号指定方式なら，

```
   print 600,x,y,n                          !  600はformat文の文番号
600 format('   x = ',f10.5,'    y = ',es12.5,'    n = ',i10)
```

文字列指定方式なら，

```
   print "('  x = ',f10.5,'    y = ',es12.5,'    n = ',i10) ",x,y,n
   write(20, "('  x = ',f10.5,'    y = ',es12.5,'    n = ',i10) ") u,v,k
```

のように記述する．なお，文字列の時にも両端の「かっこ」は必要

上の例における数値と編集記述子の対応は以下の通り

```
 print 600,          x        ,       y        ,        n
                     ↓                ↓                 ↓
600 format('   x = ',f10.5,'    y = ',es12.5,'    n = ',i10)
```

編集記述子一覧

編集指定	数値の型	編集の意味
Iw	整数	幅 w で整数を出力する
Iw.m	整数	幅 w で整数を出力する 出力整数の桁が m より小さい時には，先頭に 0 を補う（w ≧ m ）
Fw.d	実数	幅 w で実数を固定小数点形式で出力する d は小数点以下の桁数（w ≧ d + 3）
Ew.d	実数	幅 w で実数を浮動小数点形式で出力する d は小数点以下の桁数（w ≧ d + 8） 仮数部の 1 桁目は 0 になる
ESw.d	実数	幅 w で実数を浮動小数点形式で出力する（d は E 編集と同じ） 0 以外の数値を出力すると，仮数部の 1 桁目は 1 から 9 になる
ENw.d	実数	幅 w で実数を浮動小数点形式で出力する（d は E 編集と同じ） 0 以外の数値を出力すると，仮数部の整数は 1 以上 1000 未満となり，指数部は 3 で割り切れる数になる
Gw.d	整数／実数	整数の場合は I 編集を使って出力する 実数の場合は F 編集を使った固定小数点形式で出力するが， 指数部の絶対値が大きくて，幅 w では出力できないときには， E 編集に切り替えて浮動小数点形式で出力する
A	文字列	文字列をそれ自身の長さの幅で出力する
Aw	文字列	幅 w で文字列を出力する
Lw	論理	幅 w で論理型値を出力する（T または F）

とりあえずは，I編集，F編集，ES編集を覚えればよい

formatの書き方例

例えば,

```
real x,y
integer m,n
x = 1.5
y = 0.03
m = 100
n = 10
print "(f10.5,f10.5,i10,i10.5)",x,y,m,n
```

の出力は，次のようになる

```
      1.50000   0.03000         100       00010
+----+----+----+----+----+----+----+----+    ...  この行は出力されない
```

★ 数値が右寄せになっていることに注意！

★「文字列」は左寄せになる

★ 非常に大きな実数や非常に小さな実数はES編集で出力した方が良い

```
real x
x = 3.14e10
print "(es15.5)",x
```

の出力結果は，以下のようになる

```
    3.14000e+10
```

反復数の指定

- 編集記述子の前に定数を書くと，その編集記述子をその定数回くり返し書いたことに相当する

例えば,

```
print "(f10.5,f10.5,f10.5)",x,y,z
```

は次のように書ける

```
print "(3f10.5)",x,y,z
```

かっこ付けで反復指定することもできる

```
print "(f10.5,i10,f10.5,i10)",x,m,y,n
```

のように，実数と整数を交互に並べたいときは，次のように書ける

```
print "(2(f10.5,i10))",x,m,y,n
```

反復数は，多めに書いても良い

```
print "(10f10.5)",x,y,z
```

と書いておいても，3個しか出力されない