Fortranの文法まとめ

Fortran90基本事項

- **大文字と小文字は区別しない**
  - 昔はすべて大文字で書くのが主流
  - 予約語(PROGRAMなど)だけ大文字、あるいは文頭だけ大文字(Program)など色々なスタイルがある
  - 授業で配布するサンプルはすべて小文字で作成
- **自由形式：1行132文字まで記述**(コンパイルオプションで撤廃も可能)
  - FORTRAN77では固定形式(何文字目から書くかが指定されていた)
- **一行に一文を記述**
  - ;を使うと一行に複数文を書くことも可能
  - 例）　a = 0; b = 0
- **スペースは無視される**。 a=bも a = bも同じ。
- **!(エクスクラメーションマーク)以降の文字はコメント**。コンパイル時には無視される
- **一文が長く複数行に分ける場合は & で次の行に続けて書ける**。次の行頭にも&を置いてよい。

```
a = 1.0d0 + 2.0d0 + 3.0d0 + 4.0d0 + 5.0d0 &
& + 6.0d0 + 7.0d0 + 8.0d0 + &
&    9.0d0 + 10.0d0
```

Fortranプログラムの基本構造

```
program プログラム名

宣言部 (implicit none宣言、変数宣言)


実行部


end program プログラム名
```

プログラム名は英字から始まる適当な名前(FORTRAN77では6文字以内だったが90では31文字以内)

型宣言とimplicit none

- 変数(数値を格納する入れ物)には型がある
- **implicit noneは暗黙の型宣言を使用せず、すべての変数の型を明示的に指定する→必ず使用すること**
- 暗黙の型宣言とは
  - 古いFORTRANのコードではIMPLICIT REAL*8(a-h, o-z)と書き、
a~ h またはo~zから始まる変数→倍精度実数
それ以外のi~nから始まる変数→デフォルトの整数 と常に仮定。
  - 名前を間違えて打った時などに検出できないので非推奨
  - 今でも整数はiから始まるものを慣習的に使うことがある (ierrorなど)
- とりあえずprogram文を書いたら次の行に**implicit none**と書く

変数の型

型	
整数型	**integer**
単精度実数型 (物理の数値計算ではあまり使わない)	real, real*4, real(4)
倍精度実数型	double precision, **real*8**, real(8)
複素数型(物理の数値計算ではあまり使わない)	complex, complex*8
倍精度複素数型	complex(kind(0d0)), **complex*16**, double complex
論理型	**logical**
文字型	**character**

太字はよく使うもの。

変数名：英数字とアンダーラインが使える。英字から始まる31文字までの名前

変数の宣言例

implicit noneの下で変数の宣言をすべて行う。**宣言しない変数はプログラム中では使えない。**

```
integer :: a                                !整数型で変数aを宣言
real*8 :: b, c                              !倍精度実数型のbとcを宣言
double precision :: d = 0.0d0               !倍精度実数型のdを宣言して初期値0.0d0を代入
complex*16 :: dt1                           !倍精度複素数型のdt1を宣言
complex*16, parameter :: iunit = (0.0d0, 1.0d0)
!倍精度実数型のiunitを宣言し、実部が0.0d0, 虚部が1.0 (つまり虚数単位)を代入。parameter属性を
!つけるとプログラム中でこの値が変更できない
character ::  s                             !長さ1文字の文字型変数sを宣言
character(len=4) :: s2                      !長さ4文字の文字型変数s2を宣言
character(4) :: s3                          !長さ4文字の文字列変数s3を宣言(上と同じ)
character*4 :: s4                           !長さ4文字の文字列変数s4を宣言(同じ)
```

初期値を設定しない場合は :: を省略できるが全部につけておけばよい。
1.23d4と書いた場合は 1.23×10^4 の倍精度実数、の意味。0.0d0 は倍精度実数の0を表す。
1.23e4と書いた場合は 1.23×10^4 の単精度実数、の意味。0.0e0 は単精度実数の0を表す。

変数について

- **変数はコンピュータのメモリ上の保持されるため表現できる値に上限がある。**
- 整数型：4バイト (=32ビット)
  (1バイト=8ビット、1ビットは0か1の情報をもつ)
  ⇒2進数で32桁、正負の表現に1ビット使う(これは正確な表現ではなく負の数は補数を使って表す)ので最小値は-2^{31}、最大値は $2^{31}-1$
- 単精度実数：4バイト、10^{-38}～10^{38}程度の範囲を約7桁の精度で表現
  →物理の計算では精度が不十分
- 倍精度実数：8バイト、10^{-308}~10^{308}程度の範囲を約16桁の精度で表現
- 単精度複素数型：8バイト、実部と虚部それぞれが単精度実数型
- 倍精度複素数型：16バイト、実部と虚部それぞれが倍精度実数型
- 論理型：値は真(.true.)か偽(.false.) 。T, F としてもOK
- 文字型：1文字1バイト

倍精度実数のメモリ上での形式(発展)

倍精度実数型：8バイト＝64ビット

実数　＝ $(-1)^{符号} \times (1 + b_1 2^{-1} + b_2 2^{-2} + b_3 2^{-3} + \dots + b_{52} 2^{-52}) \times 2^{(e-1023)}$

この形式で近似されるため全ての実数値が表現できるわけではない

四則演算と代入文

- 数学の=(イコール)は左辺と右辺が等しいことを表すが
  プログラムの=は**右辺の計算結果を左辺の変数に代入する**、の意味
- 四則演算は +, -, *, /, 累乗は ** で表す。
- **同じ型同士の演算は同じ型になる。**
- 整数と実数の演算→実数、実数と複素数の演算→複素数

```
a = 1.0d0          ! 倍精度実数型のaに1.0d0を代入
i  = i  +  1        ! 整数型iの値を1増やす (再帰代入)
a =  4**3           ! aは4の3乗 (64)
a =  5 / 2          ! aは2となる。これは整数型の5を整数型の2で割っているため
! 同じ型同士の演算となり、割り算の商の整数の2が右辺の計算結果となる。
! 2.5にしたい場合は5または2のいずれかを実数にする。
! 5と書くと整数、5.0, 5.0e0 (単精度)または5.0d0 (倍精度)と書くと実数となる
```

実行文

- プログラムの実行文は上から順番に一行ずつ実行される(重要)

```
integer :: a, b, c

a = 2
b = 5
c = a + b
```

変数aに2を代入
変数bに5を代入
変数cにa + bを代入。この時点でaとbに値が入っているので
2+5が計算されて変数cに7が代入される

```
integer :: a, b, c

c = a + b
a = 2
b = 5
```

順番を変えると結果が変わる。
cにa+bを代入。aとbに値がまだ入っていない。システム依存になるが
おそらくaにもbにも初期値の0が入っているのでcもおそらく0となる
aに2を代入
bに5を代入
cの値はその後変更していないので0のまま。

数学的には同じに見えるがプログラムでは結果は異なる。
プログラムを書くときは**右辺に使う変数に値が入っているかを確認**しながら書く

標準入出力

- 標準入力(キーボードからの入力)

```
read (*,*) var        ! 標準入力(キーボード)からの入力を読み込んで変数varに代入
read (5,*) var        ! 標準入力(キーボード)からの入力を読み込んで変数varに代入
```

- 標準出力(画面への出力)
- 標準エラー出力(画面への出力)

```
write (*,*) var            ! 標準出力(画面)にvarの値を出力
write (6,*) var            ! 標準出力(画面)にvarの値を出力
write (0,*) var            ! 標準エラー出力(画面)にvarの値を出力
print *,  var              ! 標準出力(画面)にvarの値を出力
```

read, writeの１つ目の引数で**ファイル装置番号**を指定する。*とすると標準入出力になるが、Fortranでは5が標準入力、6が標準出力、0が標準エラー出力となっている
シェル側でリダイレクトを使うことで標準入出力先をファイルなどに変更することができる
計算した結果はwriteかprintで必ず出力する

doループ

- 実行文は上から一行ずつ実行するのが原則だがよく使う例外は2つだけ
- doループ：do~end doで囲まれた領域をカウンタを変えて繰り返し実行

```
sum = 0                          ！変数sumに0を代入
do i = 1, n                      ！カウンタ(整数型のみ)iに1を代入。end doまで実行したら次は2を代入
                                 ！ 繰り返して i=nまで実行する。
    sum = sum + i                ！変数sumの値をiだけ増やす
end do                           ！ここまで来て iの値がnより小さければ doまで戻る。
                                 ！ i=nならdoループから抜けて次の行へ。
```

$$\sum_{i=1}^{n} i$$ を計算していることになっている

doループ

増分値(ストライド)を1以外にする

```
sum = 0
do i = 1, n, 3            ! 増分が3になる。i=1の次はi=4, i =7, ...となる。
    sum = sum + i
end do
```

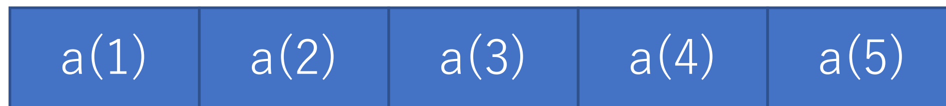
```
sum = 0
do i = 1, -10, -3         ! 増分が-3になる。i=1の次はi=-2, i =-5, ...となる。
    sum = sum + i
end do
```

配列

- 配列とは：変数が複数並んだもの(メモリ上でも並んで配置される)。ベクトルや行列が表現できる。

配列の宣言の基本

```
double precision :: a(1:5)   ! aは5つの要素を持つ倍精度実数変数の配列
```



←メモリ上にこのように5つの倍精度実数変数の領域が連続して確保される

配列の宣言例

```
double precision :: a(5)                                  !としてもよい。
double precision, dimension(1:10) :: p, q, r, s  ! dimension属性を使うと同じサイズの配列を複数まとめて作れる。
! p,q,r,sはすべて要素数が10の配列
double precision :: p(1:10), q(1:10), r(1:10), s(1:10)  !と同じ
double precision :: x(-2:2) !  インデックスは1から始めなくてもよい。
                                       ! この場合は x(-2) x(-1) x(0) x(1) x(2)の5つとなる
double precision :: b(1:3,1:4) ! 3x4の2次元配列(行列)。b(3,4)としてもよい。
```

配列の初期値設定

```
double precision ::  a(1:5) = 0.0d0    ! 配列aのすべての要素に0.0d0を代入
integer :: k(1:5) = (/1, 2, 3, 4, 5/)       !配列kの各要素に初期値(順に1,2,3,4,5)を設定
```

実行文として設定してもよい(こちらのほうが自由度が高い)。**doループを使う**

```
do i = 1, 5
    a(i) = 0.0d0    !  5次元配列のすべての要素に0.0d0を代入
end do
```

異なる配列の違う添字を持つ要素の代入・演算も可能

```
do i = 1, 5
     a(i) = b(i+3)  ! a(1)=b(4), a(2)=b(5) ... のように代入される
end do
```

1次元配列の計算例

ベクトル和

ベクトルの内積

```
real*8, dimension(1:5)  :: a, b, c    ! a, b, cを5要素の配列(5次元ベクトル)として宣言
real*8 :: adotb                        ! adotbとして倍精度実数型変数を宣言

! ここでaとbには値を代入しておく(省略)

do i = 1, 5
    c(i) = a(i) + b(i)                 ! ベクトルcにベクトルaとbの和を代入
end do
do i = 1, 5
    write(*,*) "c(", i, ")=",  c(i)    ! ベクトルcの値を要素ごとに出力
     ! この意味は "c("という文字列、整数型変数i, ")="という文字列、倍精度実数c(i)を順番に出力
     !  結果として出力は c(  1  )=  1.00.... のようになる。
end do
adotb = 0.0d0                          ! 和を取るときは変数の初期化をする
do i = 1, 5
    adotb = adotb + a(i)*b(i)          ! adotbの値をa(i)*b(i)だけ増やす(内積)
end do
print *, "a dot b = ", adotb           ! adotbの値を出力
```

1次元配列の組み込み関数

和

```
c = sum(a(1:10))      ! a(1)からa(10)までの和をcに代入
```

最大値・最小値

```
d = maxval(a(1:n))   ! a(1)からa(n)の中の最大値をdに代入
e = minval(a(1:n))   ! a(1)からa(n)の中の最小値をeに代入
```

2つの1次元配列の内積

```
f = dot_product(a(1:n), b(1:n))  ! n要素の配列aとbの内積をfに代入
```

a, bが複素数型の場合は aの複素共役とbの積が計算される

行列

- 2次元配列で行列が表現できる

```
real*8, dimension(1:5,1:5) :: b

do j = 1, 5
    do i = 1, 5
        b(i,j) = 0.0d0
    end do
end do
```

行列の初期化
二重ループ。この例では計算の順番は
j=1, i=1 → j=1, i=2 → j=1, i=3 → j=1, i=4 → j=1, i=5 →
j=2, i=1 → j=2, i=2 → ・・・
二重doループでは内側のループが先に変化する。

b(1:5,1:5) = 0.0d0 !0を代入するだけなら1行でもかける。

Fortranでの多次元配列は

b(1,1)	b(2,1)	b(3,1)	b(4,1)	b(5,1)	b(1,2)	b(2,2)	b(3,2)	b(4,2)	・・・

の順でメモリ上に配置されるため(C言語では行・列が逆になる)doループは１つ目の
インデックスを先に回すほうがメモリ上で連続的にアクセスできてよい(キャッシュミスが少なくなる)

行列の和・値の出力

```
real*8, dimension(1:5,1:5) :: a, b, c      !5x5の2次元配列(行列)としてa,b,cを宣言

! a, bに値をここで代入(省略)

do j = 1, 5
    do i = 1, 5
        c(i,j) = a(i,j) + b(i,j)                    ! cの各要素にa+bの値を代入
    end do
end do

do i = 1, 5
    write(*,*) (c(i,j), j = 1, 5)                   ! cの値を出力。
                                                     ! (c(i,j),j=1,5)はc(i,1), c(i,2), c(i,3),c(i,4), c(i,5)と展開される
end do
```

行列の積

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

一つの要素を計算するのにdoループが必要

```
real*8, dimension(1:5,1:5) :: amat, bmat, cmat    ! 5x5の行列を宣言

! aとbの値を代入

do j = 1, 5
   do i = 1, 5
      cmat(i,j) = 0.0d0       ! cの値を初期化する
   end do
end do

do j = 1, 5
   do i = 1, 5
      do k = 1, 5
         cmat(i,j) = cmat(i,j) + amat(i,k) * bmat(k,j)    ! amatとbmatの積をcmatに代入。
      end do
   end do
end do
```

2次元配列の組み込み関数

転置行列

```
bmat(1:n,1:n) = transpose(amat(1:n,1:n))    ! bmatにamatの転置行列を代入
```

行列積

```
cmat(1:n,1:m) = matmul( amat(1:n,1:k), bmat(1:k,1:m))
! (n×k行列のamat とk×m行列のbmatの積をcmatに代入(n×m行列)
```

matmulは経験上計算速度が早くないので大行列の計算では他のライブラリを使うほうがよい

配列のテクニック

同じ値を複数の要素に代入する：doループで書くほか、一行でも書ける。

```
a(1:5) = 0.0d0
b(1:5,1:5) = 0.0d0

a = 0.0d0       !とすると配列aのすべての要素に0.0d0が代入される。
a(:) = 0.0d0   ! 配列aの宣言されたすべての要素はこれでも指定できる
a(:3) = 0.0d0 !とすると a(1)から(3)までに0.0d0が代入される
```

同じ形状の配列に同じ代入を行う

```
a(1:5) = b(1:5)     ! a(1)=b(1), a(2)=b(2), ... と同じことになる。
a(1:5) = b(4:8)     ! a(1)=b(4), a(2)=b(5), ... と同じ。要素数が左右で同じであればよい

cmat(1:5,1:5) = amat(1:5,1:5) + bmat(1:5,1:5)   ! 行列の和
```

配列のテクニック

- **宣言時に配列の大きさを変数に使いたい**
  →配列より手前で parameter属性をつけて配列の大きさになる整数型を宣言。サブルーチンの場合はintent(in)属性をつける。

```
integer, parameter  ::  n = 100   ! parameter属性をつけるとプログラム中で値が変更できなくなる
double precision :: a(1:n), b(1:n), c(1:2*n,1:2*n)
```

- **配列の割付**：コンパイル時に配列の大きさを決めない(プログラム実行時に決める)

```
integer :: n
real*8, dimension(:), allocatable :: a  ! サイズを指定せずに配列を宣言

read (*,*) n  ! 例えば実行時に標準入力からnの値を読み込む
allocate (a(1:n)) ! 配列の割り付け。aを要素数nの配列としてメモリに割り当てる
! ここで計算をする(a(1:n)が使用可能)
deallocate (a)     ! aの配列をメモリから解放する
```

配列の割付(発展)

- allocateはすでに割り付けられた配列に行うとエラーとなる
- 割り付けられていない配列をdeallocateするのもエラーとなる
- allocated()関数は配列が割り付けられているかどうかの真偽を返す
- 特に割付配列をモジュール変数にした場合などに割付状況が不明になる。

```
if( .not. allocated(a) ) allocate (a(1:n))  ！配列aが割り付けられていなければ割り付ける

!これだと割り付けられていた場合前の情報が保持されるため、割り付けをリセットする場合は
if(allocated(a)) deallocate(a)      ！割り付けられている場合は解除する
allocate( a(1:n))                    ！割り付ける
```

条件文

- 実行文は上から一行ずつ実行するのが原則だがよく使う例外は2つだけ
- if文：条件を論理式で指定し条件を満たすかどうかで処理内容を変える

```
integer :: i              ! 整数型変数iを定義
read (*,*) i              ! iの値を標準入力から読み込む
if ( i<0) i=-i            ! iの値が負の場合はiに-iの値を代入
print *, "i = ", i        ! iの値を標準出力に出力(入力した値の絶対値)
```

条件文

```
if (論理式) 実行文

if (論理式) then
    実行文1
    実行文2
end if

if (論理式) then
    実行文1
else
    実行文2
end if

if (論理式1) then
    実行文1
else if (論理式2) then
    実行文2
else
    実行文3
end if
```

論理式が真のときのみ実行文が実行される(実行文が1行の場合)

論理式が真のときのみ実行文1と実行文2が実行される。
実行文が複数行ある場合はこのスタイル

論理式が真の場合は実行文1が実行され、偽の場合は実行文2が実行される

論理式1が真のときは実行文1を実行
論理式1が偽かつ論理式2が真の時は実行文2を実行
論理式1も論理式2も偽の場合は 実行文3を実行

else ifは複数あってもよい。

論理式

関係演算子		
a == b	a .eq. b	aがbと等しい時に真 (equal)
a /= b	a .ne. b	aがbと等しくないとき真(not equal)
a >= b	a .ge. b	aがb以上の時に真 (greater equal)
a > b	a .gt. b	aがbより大きい時に真(greater than)
a <= b	a .le. b	aがb以下の時に真(less equal)
a < b	a .lt. b	aがbより小さい時に真(less than)

等しい時は=一つではなく2つなので注意

論理演算子	意味	使用例
.not.	(右)以外	.not. a==b
.and.	かつ	a==b .and. b==c
.or.	もしくは	a==b .or. b==c
.eqv.	論理値が等しい	a<0 .eqv b<0 (aとbの符号が同じ)
.neqv.	論理値が異なる	a<0 .neqv. b<0 (aとbの符号が違う)

論理型変数に論理式を代入できる。

```
logical :: abpositive
integer :: a, b
read(*,*) a, b
abpositive = a>0 .and. b>0

if (abpositive) then
....
end if
```

論理式の値は真の時は .true. (またはT) 、偽の時は .false. (またはF) となる

プログラムの終了

- プログラムは最終行まで到達すれば終了
- 条件文を使うと途中で終了させることも可能
- stop文で終了。 stop “文字列” とすると文字列が標準エラー出力に出力
- プログラムの一番最後にstopと正常終了のメッセージを書く場合もある

```
read(*,*) x                                      ! xを標準出力から読み込む
if( x<0.0d0) stop “x must be zero or positive”   ! xが負であればプログラムを終了
                                                 ! xがゼロか正であれば以下の処理が行われる
print *, “sqrt(x) = “, sqrt(x)
```

条件文の例：クロネッカーのデルタ

```
integer :: delta, i, j

read (*,*) i, j

if (i == j) then
    delta = 1
else
    delta = 0
end if

print *, delta
```

```
integer :: delta, i, j

read(*,*) i,j

delta = 0
if(i==j) delta = 1

print *, delta
```

doループとif文の組み合わせ

無限ループ, exit

```
do                          ! カウンタを指定しないと無限に繰り返すループとなる。
!ここでいろいろ計算
if ( a<0 ) exit             ! ある条件が満たされたらループから抜ける
end do
```

exit文はdoループから抜け、end do文の直後に移動
条件が満たされない場合は計算が終わらないため実行してみて終わらない場合は強制終了(C-c)する

cycle: doループの先頭まで戻り、カウンタを次の値に進める

```
do i = 1, n

   a(i) = ....ここで何かを計算する

    if ( a(i)>0 ) cycle !a(i)が正なら戻ってカウンタを一つ回す

    ! a(i)が0か負の場合だけこの領域で処理が行われる

end do
```

```
do i = 1, n

    if( i == 7) cycle ! iが7のときだけ何もしない

     ! iが7以外のときは以下の計算をする。
     ! ....

end do
```

doループに名前をつける

doループが入れ子になっている場合cycleやexit文でどこに飛ぶのかがわかりにくい
名前をつけない場合は一番内側のループに対して処理が行われる

```
loop1: do i = 1, n
   loop2: do j = 1, n

     ! ここで計算

     if( 論理式 ) exit loop1  ! 抜けるループを名前で指定する

   end do loop2
end do loop1
```

型変換

整数型↔実数型

```
ai = dble(i)   ! 整数型変数iを倍精度実数型に変換
ia = int(a)    ! 倍精度実数型変数aを整数型に変換
```

組み込み関数sqrt(x)などでは引数は倍精度実数であるため、
整数を引数に使いたい場合は先に実数型に変換する（sqrt(dble(i))など）

整数・実数型↔文字型

```
write(c,*) i   ! 整数型変数iを文字型変数cに変換
read (c,*) i   ! 文字型変数cを整数型変数iに変換
```

ファイル装置番号に文字型配列を使うことによって値の変換が可能
整数の値を出力ファイル名に使うのに使うことがある

複素数

値の代入

```
z = (2.0d0, 3.0d0)    ! 倍精度複素数型変数zに 2.0d0+3.0d0*i を代入する。
```

実部と虚部が倍精度実数の変数a, bで与えられる場合

```
iunit = (0.0d0, 1.0d0)  !虚数単位をプログラムの最初のほうで定義
z = a + b * iunit
z = dcmplx(a, b)          !と書いてもよい。
```

実部・虚部の取り出し

```
rez = dble(z)    ! 複素数zの実部を倍精度実数型rezに代入
imz = aimag(z) ! 複素数zの虚部を倍精度実数型imzに代入
```

複素共役

```
zconjg = conjg(z)   !組み込み関数conjgを使う
```

組み込み関数

数学関数

関数	名称	数学式	関数	名称	数学式
log(x)	自然対数		log10(x)	常用対数	$\log_{10}(x)$
exp(x)	指数		sqrt(x)	平方根	$\sqrt{x}$
sin(x)	正弦		cos(x)	余弦	
tan(x)	正接		asin(x)	逆正弦	arcsin(x)
acos(x)	逆余弦	arccos(x)	atan(x)	逆正接	arctan(x)
atan2(y,x)	逆正接2	arctan(y/x)	sinh(x)	双曲線正弦	
cosh(x)	双曲線余弦		tanh(x)	双曲線正接	
rand()	[0,1]の乱数	random()	x**a	累乗	x^a

引数は実数なので整数を使いたい場合は型変換で実数に変換したものをxに使う
累乗は整数でも実数でもOK

角度の単位はラジアン（180° ＝ π rad）

組み込み関数

数値関数

関数	名称	関数	名称
int(x)	整数化	real(x)	単精度実数化 複素数の実部
dble(x)	倍精度実数化 複素数の実部	dcmplx(x,y)	倍精度複素数化 (x+yi)
mod(x,y)	x/yの余り	max(x,y,…)	最大値
min(x,y,…)	最小値	abs(x)	絶対値
aimag(x)	複素数の虚部	conjg(x)	共役複素数
sign(x,y)	xの符号替え y/|y| * |x|	nint(x)	四捨五入後整数化

多くの関数には整数用・単精度実数用・倍精度実数用・単精度複素数用などの別名がある
(abs(x)は iabs (整数→整数), abs(単精度実数→単精度実数), dabs(倍精度実数→倍精度実数)
cabs(単精度複素数→単精度複素数), zabs(倍精度複素数→倍精度複素数)など。
総称名のabsを使えば適切なものを呼び出してくれるため、dabsなどを明示的に使う必要はない。
ただしcmplx は単精度複素数のみとなり倍精度複素数を使うにはdcmplxを使う。
realは単精度実数であるため倍精度実数ではdbleを明示的に使う。

文字列の操作(発展)

- Fortranでは文字列の詳細な操作はしないほうがよい
- 文字列配列は主に入出力ファイル名として使う

文字列の代入

```
character(len=100) :: filename   ! 100文字の文字列型変数filenameを宣言

filename = 'outputfile1.txt'       ! outputfile1.txtという文字列をfilenameに代入

read (*,*) filename                ! または標準入力から読み込んだ文字列をfilenameに代入

open(unit=10, file=filename, ... ) ! filename文字列で指定されるファイルを開く
```

o	u	t	p	u	t	f	i	l	e	1	.	t	x	t		

左詰めで配列に格納される。 filename(1:2)は'ou'となる。
余った部分は空白となる

文字列の操作(発展)

- 例：整数nを標準入力から読み込んでそれを使ったファイル名(outputfile1.txtの1の部分を変数nから読み込んだ値)を作りたい

```
integer :: n
character(100) :: filename

read(*,*) n                                               ！ nの値を標準入力から読み込む
write(unit=filename, fmt=‘(“outputfile”, i3.3, “.dat” )’) n   ！ outputfile(3桁のn).datをfilenameに書き込む
```

```
integer :: n
character(100) :: filename, filename1, filename2, nchar
read(*,*) n
write(nchar,*) n              ！整数型nを文字列ncharに変換する(右詰めで入る)
filename1=‘outputfile’   ！左詰めで代入
filename2=‘.txt’              ！左詰めで代入
filename =  trim(filename1) // trim(adjustl(nchar)) // trim(filename2) ！3つの文字列を結合する
```

// で文字列の結合, trim()は文字列の右側の空白を削除する関数、adjustl()は文字列を左詰めにする関数

関数副プログラム

- 関数：与えられた入力変数に対して操作をして値を返す一連の手続き
- 例えば $4x^3$-5xを何度も様々なxに対して計算したいとする。
  プログラムのあちこちに$4x^3$-5xと繰り返して書くとミスのもとになる。
  また後から関数形に変更が必要になったときにすべて探し出して変更する必要がある。
- 関数副プログラムとしてf(x)=$4x^3$-5xを計算する部分を独立させる。

- 関数副プログラムを置く場所
  1. 主プログラム(program ~end program) の中に置く(内部副プログラム)
  2. 主プログラムの外(end programの後)に置く(外部副プログラム)
  3. モジュールの中に置く(モジュール副プログラム)

- 関数の引数にintent(in)属性を付けると関数内で値が変更できない(xの値は関数内では変更しない)
- return文は関数やサブルーチンの処理を中断してその段階の値が関数やサブルーチンを呼び出したプログラムに戻される。プログラムの末尾の場合は省略できる

関数副プログラム

例１：主プログラムの中に関数副プログラムを置く場合(内部副プログラム)

```
program example2
    implicit none
    double precision :: x
    x = 0.0d0
    do
        print *, x, func(x)    ！ここでfunc(x)が呼び出される。関数はcontainsより下で定義されている。
        x = x + 0.10d0
        if( x > 5.0d0) exit
    end do
contains                       ！ contains以下に副プログラムを列挙する
    function func(x)           ！関数見出し
        double precision, intent(in) :: x   ！xを倍精度実数として宣言。intent(in)がつくと関数内で値を変更できない
        double precision :: func            ！関数の型を宣言する(倍精度実数型)
        func =  4.0d0*x**3 – 5.0d0*x
        return                               ！関数を呼び出された元のプログラムに戻る
    end function func
end program example2
```

関数副プログラム

例2：主プログラムの外に関数副プログラムを置く場合(外部副プログラム)

```
program example2
   implicit none
   double precision :: x
   double precision :: func  ！外部副プログラムは主プログラムから見えないので型宣言が必要
   x = 0.0d0
   do
      print *, x, func(x)    ！ここでfunc(x)が呼び出される。
      x = x + 0.10d0
      if( x > 5.0d0) exit
   end do
end program example2
function func(x)          ！関数見出し
   double precision, intent(in) :: x   ！xを倍精度実数として宣言。intent(in)をつけると関数内で値を変更できない
   double precision :: func            ！関数の型を宣言する(倍精度実数型)
   func =  4.0d0*x**3 – 5.0d0*x
   return                        ！関数を呼び出された元のプログラムに戻る
end function func
```

関数副プログラム

例3: モジュールの中に置く(モジュール副プログラム)

```
module mod_func          ! program よりも手前にmoduleを配置。mod_funcは適当につけたモジュール名
  implicit none          ! module内でもimplicit none宣言
contains                 ! containsより下にモジュール副プログラムを配置
  function func(x)       ! 関数見出し
    double precision, intent(in) :: x   ! xを倍精度実数として宣言。intent(in)をつけると関数内で値を変更できない
    double precision :: func            ! 関数の型を宣言する(倍精度実数型)
    func =  4.0d0*x**3 - 5.0d0*x
    return               ! 関数を呼び出された元のプログラムに戻る
  end function func
end module mod_func
program example2
  use mod_func           ! モジュールを使う場合はimplicit noneより手前でuse モジュール名 としてモジュールを参照
  implicit none
  double precision :: x = 0.0d0
  do
     print *, x, func(x)    ! ここでfunc(x)が呼び出される。
     x = x + 0.10d0
     if( x > 5.0d0) exit
  end do
end program example2
```

関数副プログラムの例：外積を計算

```
program test
    implicit none
    double precision :: a(1:3), b(1:3), c(1:3)
    a(1) = 6.0d0; a(2) = 3.0d0; a(3) = 4.0d0
    b(1) = 3.0d0; b(2) = -2.0d0; b(3) = -4.0d0
    c(1:3) = vectorproduct( a(1:3), b(1:3) )    ！3要素の1次元配列型の関数
    write(*,*) c(1:3)
contains
    function vectorproduct(a, b)
        double precision, intent(in) :: a(1:3),  b(1:3)
        double precision :: vectorproduct(1:3)   ！関数の型を要素数3の1次元配列として宣言
        vectorproduct(1) = a(2)*b(3)-a(3)*b(2)
        vectorproduct(2) = a(3)*b(1)-a(1)*b(3)
        vectorproduct(3) = a(1)*b(2)-a(2)*b(1)
        return
    end function vectorproduct
end program test
```

関数副プログラムの例(再帰関数)

- 再帰(recurrsion)とは：関数副プログラム(やサブルーチン)の中で自分自身(関数副プログラムやサブルーチン)を引用すること。
- 再帰的に関数副プログラム(やサブルーチン)を呼び出す場合はrecursiveをつける。
- 関数の実行結果は関数名ではなくresult句で指定された変数に代入される。

フィボナッチ数列の例

```
recursive fuction fibonacci(n) result(fibo)
    integer, intent(in) :: n
    integer :: fibo                ! 結果を代入する変数
    if (n<0) then
        fibo = -1
        return
    end if
    if (n == 0) then
        fibo = 0
    else if (n == 1) then
        fibo = 1
    else
        fibo = fibonacci(n-1) + fibonacci(n-2)  !  自分自身を呼び出す
    end if
    return
end function fibonacci
```

サブルーチン副プログラム

- 関数副プログラムとほぼ同じ。違う点は
  - 値を返さない(関数には型があるがサブルーチンにはない)
  - ただし引数として(複数の)値を返すことができる。
  - call文で呼び出す
  - サブルーチン副プログラムは1つの実行文となるが、関数服プログラムは実行文ではない

- 長いプログラムを書く時は主プログラムに長々と書かずサブルーチンに分割する

サブルーチン副プログラム

```
program example2
    implicit none
    double precision :: x, y
    x = 0.0d0
    do
        call func(x,y)          ! call文でfuncサブルーチンを呼び出し
        print *, x, y
        x = x + 0.10d0
        if( x > 5.0d0) exit
    end do
contains                        ! contains以下に副プログラムを列挙する
    subroutine func(x, y)           ! サブルーチン見出し。入力も出力も引数に入れる。
        implicit none               ! サブルーチンではimplicit noneを使うことを推奨
        double precision, intent(in) :: x     ! intent(in)がつくとサブルーチン内で値を変更できない
        double precision, intent(out) :: y    ! intent(out)がつくとサブルーチン内で値をセットしなければならない
        y =  4.0d0*x**3 – 5.0d0*x
        return                      ! 関数を呼び出された元のプログラムに戻る
    end subroutine func
end program example2
```

サブルーチン副プログラム

- サブルーチンの引数にはintent属性をつけることを推奨
  - intent(in) : 入力引数、サブルーチンに値が渡され、サブルーチン内で値の変更ができない
  - intent(out) : 出力変数、サブルーチンで値がセットされる変数
  - intent(inout) : 入出力変数、サブルーチンに値が渡され、サブルーチンで値を変更できる変数
  - サブルーチン内だけで使う変数には何もつけない。

サブルーチン例：外積を計算

```
program test
    implicit none
    double precision :: a(1:3), b(1:3), c(1:3)
    a(1) = 6.0d0; a(2) = 3.0d0; a(3) = 4.0d0
    b(1) = 3.0d0; b(2) = -2.0d0; b(3) = -4.0d0
    call vectorproduct( a(1:3), b(1:3), c(1:3) )    ！入力も出力も引数に入れる
    write(*,*) c(1:3)
contains
    subroutine vectorproduct(a, b, c)
        implicit none
        double precision, intent(in) :: a(1:3),  b(1:3)
        double precision, intent(out) :: c(1:3)
        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)
        return
    end subroutine vectorproduct
end program test
```

変数の共有(内部副プログラムの場合)

- 変数を主プログラムと副プログラム(関数やサブルーチン)で共有する
- 基本は引数として渡すのが間違いが少ない

```
program test
  implicit none
  real*8 :: x0, a=1.0d0, b=2.0d0, c=3.0d0
  x0 = 1.0d0
  print *, x0, f(x0)            ! 副プログラムのxとしてx0を渡す
  a = 2.0d0
  print *, x0, f(x0)            ! aの値が変わったのでf(x0)の値が変わる
contains
  function f(x)                 ! 2次関数を計算する関数副プログラム
    real*8, intent(in) :: x     !  xは引数として主プログラムから副プログラムに渡される
    real*8 :: f
    f = a*x**2 + b*x + c        !  a, b, cは主プログラムの値を参照する
    return
  end function f
end program test
```

短いプログラムの場合は内部副プログラムが簡単に書けるが、**内部副プログラムは独立性が低い**
(この副プログラムだけ他のコードにコピーしても使えない(a,b,cが中で未定義なので))

変数の共有(外部副プログラムの場合)

- 基本は引数にすべて書く。引数に書かれていないものは参照できない

```
program test
  implicit none
  real*8 :: x0, a=1.0d0, b=2.0d0, c=3.0d0, f   ! a,b,cの値は外部プログラムからは直接参照できない
  x0 = 1.0d0
  print *, x0, f(x0,a,b,c)          ! x0以外にa,b,cも引数として関数に渡す
  a = 2.0d0
  print *, x0, f(x0,a,b,c)
end program test
function f(x,a,b,c)                 ! 様々なa,b,cの2次関数を計算をしたい場合はa,b,cも引数に入れる
  real*8, intent(in) :: x,a,b,c
  real*8 :: f
  f = a*x**2 + b*x + c
  return
end function f
```

外部プログラムは独立性が高い。計算に必要な変数がすべて引数に入っているので外部プログラムだけを他のコードで使うことができる。ただし引数が煩雑になる。

変数の共有(モジュール副プログラム)

- モジュール内に変数を記述し、モジュールを参照する
- 引数には色々な値で計算することを想定されるものだけを入れる

```
module func_mod !モジュールはprogramより手前に配置
  implicit none
  real*8, save :: a, b, c
!副プログラムで使う変数をsave属性つきで定義
!save属性をつけておくと前に代入された値を保持し続ける
contains
  function f(x) !  モジュール副プログラム
    real*8, intent(in) :: x
    real*8 :: f
    f = a*x**2 + b*x + c
!モジュール副プログラムはモジュール内変数を参照できる
    return
  end function f
end module func_mod
```

```
program test
  use func_mod !implicit noneの手間でモジュールを参照
  implicit none
  real*8 :: x0
  a=1.0d0; b=2.0d0; c=3.0d0 !モジュール変数に値をセットできる
  x0 = 1.0d0
  print *, x0, f(x0) !モジュール関数も呼び出せる
  a = 2.0d0
  print *, x0, f(x0)
end program test
```

モジュールは独立性が高い。モジュールの中で他のモジュールをuse文で参照することも可能
module, programの順番に書く(別のファイルに分ける場合はこの順番にコンパイル)
複数のmoduleがある場合は順番(依存関係)に注意

ファイル入出力

標準入力： read (*,*) var
標準出力： write(*,*) var または print *, var
標準エラー出力： write(0,*) var

シェル側でファイルに入出力

シェルでリダイレクトによって標準入出力をファイルに変更
Linuxのリダイレクトの部分を復習してください

```
./a.out < inputfile > outputfile     # inputfileから標準入力を読み込み、標準出力をoutputfileに出力
```

Fortranでファイルを指定

```
open( 1, file='inputfile.txt', action='read')      ! 装置番号1としてinputfile.txtを開く
open( 2, file='outputfile.txt', action='write')  ! 装置番号2としてoutputfile.txtを開く
read(1,*) var                          ! 装置番号1(inputfile.txt)から読み込んだ値をvarに代入
write(2,*) var                         ! 装置番号2(outputfile.txt)に変数varの値を出力
close(1)                               ! 装置番号1のファイルを閉じる
!(この後装置番号1は他のファイルに当てることができる)
close(2)                               ! 装置番号2のファイルを閉じる
```

装置番号5,6,0は標準入力、出力、エラー出力に予約されているのでこれ以外の番号を指定

ファイル入出力

open関数で指定できるのは

unit指定子:	装置番号を指定。 unit=は省略して数字だけでもOK
file指定子：	ファイル名を指定
status指定子：	‘old’ すでにファイルが存在する場合 ‘new’ ファイルが新しく作られる場合 ‘replace’ ファイルが存在する場合は前のファイルが削除される ‘unknown’ デフォルト値。ファイルがない場合は新しく作成。
action指定子：	‘read’ 読み込み専用でファイルを開く（間違えて書き込むミスを防ぐ） ‘write’ 書き込み専用でファイルを開く ‘readwrite’ デフォルト。読み書き可能
position指定子:	‘asis’ デフォルトの場所、通常ファイルの先頭位置 ‘rewind’ ファイルの先頭位置 ‘append’ ファイルの末尾位置

すでに存在するファイルに追記したい場合は

```
open(unit=1, file=‘ファイル名’, status=‘old’, position=‘append’)
```

などとしてファイルを開く

書式指定(発展)

- write,printで出力の書式指定ができる
- writeの2つ目の引数、またはprintの*を書式に置き換える

```
write(*,'(i5)')   ivar           !  5桁までの整数を表示
write(*,'(3f15.8)')   x, y, z  !  実数を15桁で、小数点以下は8桁として3つ表示
write(*,'(2i5,3f5.8)') i, j, x, y, z  ! 5桁までの整数を2つ表示、その後15桁,小数点以下8桁で実数を3つ表示
```

編集記述子	形式	意味
i	iw	幅wの整数
f	fw.d	幅w,小数点以下d桁の実数
e	ew.d	幅w,小数点以下d桁の実数
a	aw	幅wの文字列
x	x	空白
/	/	改行

```
write(*,'(3x,i5)') 123
write(*,'(f8.2)') 123.4567
write(*,'(e8.2)') 123.4567
write(*,'(a8)') 'abc'
```

```
_ _ _ _ _ _ 1 2 3
_ _ 1 2 3 . 4 5
_ 0 . 1 2 E + 3
_ _ _ _ _ _ A B C
```

書式指定しない(*と書く)場合はデフォルトの書式が適用される(システム依存)

書式指定(発展)

- 書式指定は文字列配列に代入したものを指定してもOK
- writeの指定子はunit(装置番号を指定)、fmt (書式を指定)
- 文字列操作で書式をプログラム実行中に決定したい場合に使える

```
real*8 :: a
character(50) :: format1

format1 = ‘(a, f15.5)’                    ! 文字列配列format1に書式を代入

write(unit=*, fmt=format1)  “a = “, a    ! 書式はformat1で指定する
```

ファイル入出力(発展)

- バイナリ入出力 (バイナリ=2進数)
- 倍精度実数は10進数の形式 (1.00000E+02など)で出力すると情報が落ちる。(メモリ上では64ビット、仮数部と指数部に分けて2進数で保持)
- ファイル入出力するときに2進数(バイナリ)のまま扱うことができる
- ただしテキストとして開いて読むことはできない(0と1が並んでいるだけでありバイナリエディタが必要だが読めたものではない)。

バイナリファイルに出力する

```
open(1, file='output.txt', form='unformatted') !form指定子がunformattedだとバイナリ出力
write(1) var                                   ! unformatted(書式なし)の場合は書式なし出力
close(1)
```

バイナリファイルから読み込む

```
open(2,file='output.txt', form='unformatted')  ! 開くときも同様
read(2) var                                     ! 書式なしのread文
close(2)
```

計算の途中経過出力、途中から再開などに使う
また図にする必要のないデータの保存などはバイナリで行う

モジュール(発展)

- モジュール：変数、サブルーチン、関数などをまとめたもの
- 主プログラムなどからモジュールを参照し、中の変数、サブルーチン、関数などを使うことができる。
- 大規模プログラムでは関連するものをモジュールとしてまとめることが多い
- モジュール変数
  - parameter属性：例えば虚数単位など、主プログラムや関数・サブルーチンの様々な場所から参照する、値の変わらないものはparameter属性をつけてモジュール変数に入れておくとよい。
  - save属性：モジュール変数にはsave属性を付けておくと主プログラム、副プログラムなど様々な場所から呼び出されたときに前回に代入された値が保存される。
  - private属性：モジュールの外から変数の参照ができない
  - public属性：(デフォルト)モジュールの外から変数の参照が可能
- モジュールを参照するにはuse文で参照するモジュール名を指定
  - use モジュール名, only: a, b, func
  - などとするとモジュールに含まれるa, b変数やfunc関数などのみが参照可能となる

配列を副プログラムの引数に渡す方法

- 従来の方法：形状明示配列。副プログラムに値を渡すときに配列の大きさも引数に入れる

```
module vec
  implicit none
contains
  function vectoraddition(n,a,b)
    integer, intent(in) :: n  !配列のサイズをintent(in)属性で引数にいれる
    double precision, dimension(1:n) :: a, b !(intent(in)の値は指定できる)
    double precision, dimension(1:n) :: vectoraddition
    integer :: i
    do i = 1, n
       vectoraddition(i) = a(i) + b(i)
    end do
    return
  end function vectoraddition
end module vec
```

```
program array
  use vec
  implicit none
  real*8 :: a(1:3), b(1:3), c(1:3)
  a = (/ 1.0d0, 2.0d0, 4.0d0/)
  b = (/ -1.0d0, -3.0d0, 2.0d0/)
  c = vectoraddition(3,a,b) ! aとbは3要素の配列
  print *, c(1:3)
end program array
```

配列を副プログラムの引数に渡す方法

- 形状引継ぎ配列：モジュール内の副プログラムで使用可能
- 配列のサイズを明示的に指定する必要がない
- サイズが計算で必要な場合はsize関数で実行時に入手する

```
module vec
  implicit none
contains
  function vectoraddition(a,b)
    double precision, dimension(:), intent(in) :: a, b !形状引き継ぎ配列
    double precision, dimension(:), allocatable :: vectoraddition  !出力は割り付け配列
    integer :: i, n
    n = size(a,1) ! サイズが必要な場合はsize関数(配列aの１つ目のインデックスの要素数)
    allocate(vectoraddition(n)) !出力の配列を割り付ける
    do i = 1, n
       vectoraddition(i) = a(i) + b(i)
    end do
    return
  end function vectoraddition
end module vec
```

```
program array
  use vec
  implicit none
  real*8 :: a(1:3), b(1:3), c(1:3)
  a = (/ 1.0d0, 2.0d0, 4.0d0/)
  b = (/ -1.0d0, -3.0d0, 2.0d0/)
  c = vectoraddition(a,b)
  print *, c(1:3)
end program array
```

gfortranのコンパイルオプション

- -o ファイル名　：作成される実行ファイルの名前を指定
- -O0, -O1, -O2, -O3　：最適化オプション
  -O3が最も最適化されるが計算結果がおかしくなる可能性もある
- -ffree-line-length-none ：一行132文字の制限を撤廃
- -Wall ：全てのコンパイル時の警告メッセージを出力
- -Wuninitialized ：初期化されずに使われた変数を検出
- -pedantic ：標準外の機能利用を警告
- -fbounds-check ：配列の領域外参照を検出
- -ffpe-trap=invalid,zero,overflow 浮動小数点例外発生時に異常終了
- -fbacktrace ：異常終了時にプログラムソースコードの行番号を表示
- -c ：実行ファイルを作らずその手前のオブジェクトファイル(.o)のみ作成

デバッグ

- プログラムは一回で完全なものは書けない。間違い(バグ)が多くの場合ある。
- 数値計算を行う研究もかなりの時間はバグ取り(デバッグ)に費やされる
- コンパイル時にエラーが出て実行ファイルが作成されない
  - 警告が出ても実行ファイルが作成される場合もある(-Wallをつけている場合など)のでls –lで実行ファイルと更新日時をチェック
  - エラーメッセージを見てバグを順番に取り除く
- コンパイルは出来たが実行すると結果が明らかにおかしい
  - こちらは見つけるのが難しい
- **バグが見つけられない場合はどんどん質問してください。**

コンパイルエラーの例

エラーがどのファイルの何行目で出たかが出るので場所を確認する
test.f90:14:5: → test.f90の14行目(5文字目)でエラー

Error: Symbol ‘**x**’ at (1) has no IMPLICIT type

- ”x”が使われているが型が見つからないエラー
- 変数の宣言し忘れ、変数名の打ち間違いはないか？

```
  print *, b(4)
          1
Warning: Array reference at (1) is out of bounds (4 > 3) in dimension 1
```

WarningであってErrorではないので実行ファイルはできるがこの配列の次元1のインデックスは3要素(b(1:3)で宣言)しかないのに4要素目を参照しようとしているという警告。
このまま実行すると配列の領域外参照となる。
コンパイル時に –**fbounds-check**オプションをつけておくとエラーがでて終了になるがつけずに実行すると問題なく実行されることもある(結果がおかしくなる可能性が高い)

コンパイルエラーの例

```
  end function vectoraddition
    1
Error: Expecting END DO statement at (1)
```

```
 end program array
   1
Error: Expecting END IF statement at (1)
```

コンパイラはend doやend ifが来ると思っているがその前に他のもの(end function/end program)が書いてある。
do, if に対して(もっと手前で)end do, end ifを書き忘れていないか？

emacsではdo-end do等でTabキーを押すと字下げをしてくれるのでend do/end if書き忘れを見つけることができる。

コンパイルエラーの例

```
    print *, b(1)
 1
Error: Invalid character in name at (1)
```

printの手前に全角スペースを混ぜた場合
有効ではない文字が混ざっているとエラーがでる
(これは見つけにくい)

```
   print *, b(1,2)
          1
Error: Rank mismatch in array reference at (1) (2/1)
```

１次元配列bを2次元配列のように２つインデックスを指定した場合
配列のランクが合わないとエラーがでる。

コンパイルエラーの例

```
program sin
  implicit none
  real*8 :: a
  a = 1.0d0
  print *, sin(a)
end program sin
```

```
  print *, sin(a)
           1
Error: Symbol at (1) is not appropriate for an expression
```

プログラム名にプログラム中に使う組み込み関数と同じものを使うとコンパイル時にエラーとなる
この例ではsin(x)を使わなければコンパイルできるが、プログラム名にはできるだけ予約されている名前を使うのは避ける

実行しながらのデバッグ

- コンパイルは通ったが結果がおかしい場合、特にどこかで出力が NaN(Not a Number)またはInfinity(無限大)と表示される
- 負の数の平方根やゼロによる割り算などがおきていないか。
- ゼロで割っているつもりはなくても変数に値を入れ忘れると変数の値がゼロになっている場合がある。
- **write文やprint文で割り算が行われる変数などの値を直前で出力して直に確かめる。**

```
program error
  implicit none
  real*8 :: a, b
  a = 0.0d0
  b = 10.0d0
  print *, b/a, sqrt(-b)   ! この実行結果は Infinity NaN となる
end program error
```

実行しながらのデバッグ

- コンパイルは通ってNaNやInfinityは出ないが計算結果がおかしい場合
- そのプログラムで答えが既知の問題は解けないか？
  - 使っている関数を簡単なものにしたときに正しい答えが出せるか？
  - 手計算で答えが求められる場合に正しい答えを出すか？
  - どの場合に結果が正しくてどの場合は結果がおかしくなるのかを調べてバグの場所を特定する。

実行しながらのデバッグ

- 書き直す時は削除せずに!でコメントアウトし、色々試すとよい。
- 宣言された配列の領域外に値を代入すると他の変数の値を破壊する可能性がある。
- 関数、サブルーチンの引数の順番、数、型などが合っているか？外部副プログラムの場合は特に整合性のチェックが入らないので実行時に結果がおかしくなる。
- 色々な所にwrite,print文を入れてプログラムの動きをモニターする。
  - 特にdoループ, if文などが想定したとおりに動いているかをチェックする。doループではカウンタの値を毎回出力して想定通りに動いているか？if文では分岐それぞれにwrite文を入れて想定通りの分岐に入っているか？
  - read文で変数・配列に読み込んだデータは正しく読み込めているか？
  - サブルーチンに渡す前の引数の値、サブルーチンに渡った直後の引数の値、サブルーチン実行後の引数の値など想定通りになっているか？
- コンパイルオプションを増やすと情報が得られる可能性がある。