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３部

サブルーチン

日本原子力研究開発機構　田口 俊弘

第３部の講習内容

１．サブルーチンの書き方

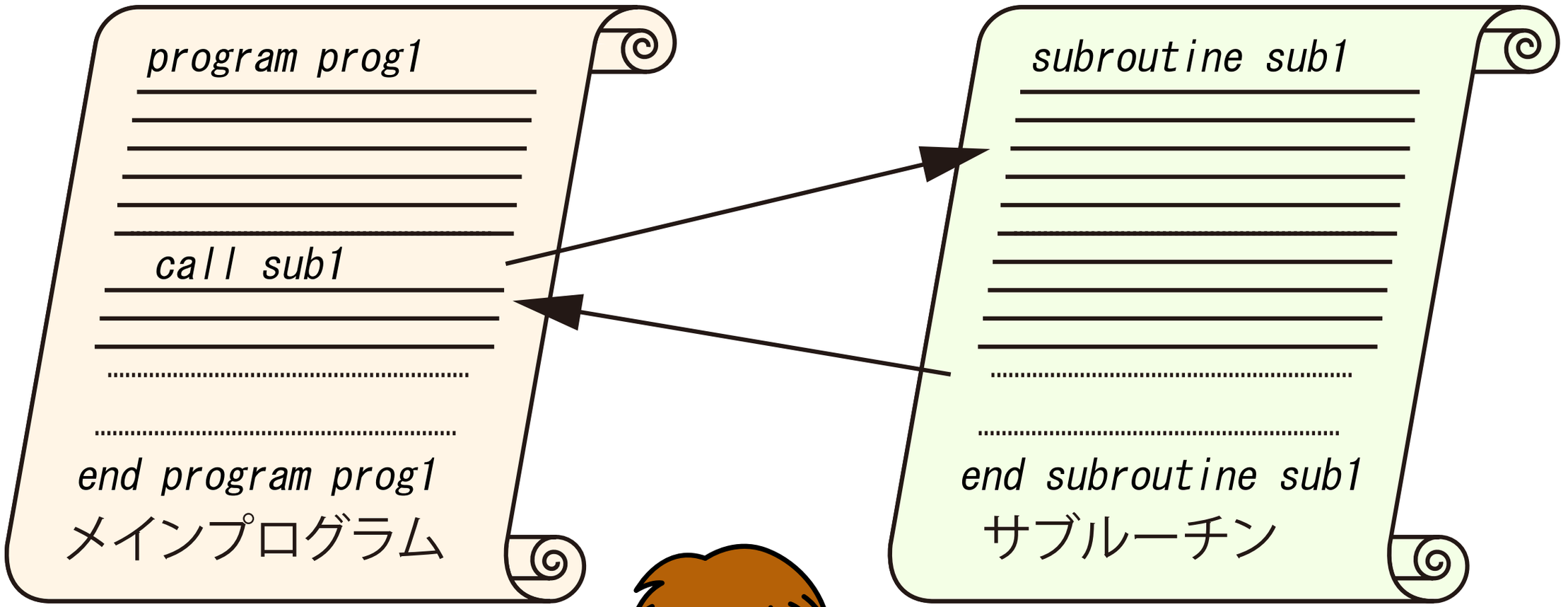
２．引数の使い方

３．関数副プログラム

４．グローバル変数

付録：知っておくと便利な文法３

１．サブルーチンの書き方



プログラムが長くなったら
サブルーチンを使って分割しよう

サブルーチンとは

- 開始点と終了点を持ったメインプログラムとは独立したプログラム
- メインプログラムと異なり，動作中のプログラムから呼び出して利用する

サブルーチンの使い道

- 複数の場所で一連の同じ計算手順を使用するとき
- 方程式の解法や行列式の計算などのような定型処理をするとき
- 長いプログラムを分割してメンテナンスを容易にしたいとき

◎メインプログラムを「メインルーチン」ということもあるので，
サブルーチンとひとまとめにして呼ぶときは「ルーチン」という

サブルーチンの書式

☆サブルーチンは subroutine 文と end subroutine 文の間に書く

```
subroutine サブルーチン名          ! subroutine文
  implicit none
  real a,b
  integer i
   .......
   .......
end subroutine サブルーチン名      ! end subroutine文
```

★ ただし，データ受け渡しのために，subroutine文には
「引数」を付けることができる（後で説明）

★「end subroutine サブルーチン名」は「end」だけでも
エラーではないが，書いた方が良い

☆サブルーチンの内部はメインプログラムとほぼ同じ書式

★ 最初に implicit none を書く

★ 次に宣言文などの非実行文を書く

★ 非実行文の後に実行文を書く

サブルーチンには引数が付けられる

引数なしサブルーチン（サブルーチン名のみ）

```
subroutine サブルーチン名
  implicit none
  real a,b
  integer i
   .......
   .......
end subroutine サブルーチン名
```

引数ありサブルーチン

```
subroutine サブルーチン名(引数1,引数2,...)
  implicit none
  real a,b
  integer i
   .......
   .......
end subroutine サブルーチン名
```

★「引数」は，変数または配列名

★ 引数はサブルーチン内で宣言しなければならない

サブルーチンの利用手段
call 文

☆サブルーチンだけのプログラムは動かない☆

- サブルーチンは，他のルーチンから呼び出して初めて有効
- 呼び出しにはcall文を使う

引数なしサブルーチンの呼び出し

```
call サブルーチン名
```

引数ありサブルーチンの呼び出し

```
call サブルーチン名(数値1, 数値2, 数値3,...)
```

★「数値」には「数式」を書いても良い（ただし注意あり！）
その場合には，数式の計算結果の数値がサブルーチンに渡される

プログラムを書いてみよう

3-A　サブルーチンを使った四則計算

例題3-A

実数 x と y を与えると，その和・差・積・商の４個の値を計算して出力するサブルーチンを作成し，それを使って，いくつかの２個の実数を使ったときの結果を出力せよ

3-A　サブルーチンを使った四則計算 の解答例

```
program answer3a
   implicit none
   real x, y
   x = 0.1;  y = 200
   call arithmetic(1.0, 2.0)
   call arithmetic(x, y)
   call arithmetic(2*x+1, x+y)
end program answer3a

subroutine arithmetic(x, y)
   implicit none
   real x, y
   print *, ' + - * / = ', x+y, x-y, x*y, x/y
end subroutine arithmetic
```

☆書き方のポイント☆

★ call 文で呼び出すときは，サブルーチンの引数として，定数，変数，計算式を書くことができる

★ ただし，数値型には気をつけなければならない（後で説明）

★ メインとサブで同じ名前の変数を宣言しているが，別物（後で説明）

サブルーチン実行の流れ

☆ サブルーチンを使ったときの実行の流れは以下の通り

① call 文を実行するとサブルーチンの実行開始点にジャンプ
② サブルーチンが終了するとcall文の次の行から実行を継続

例えば，次のプログラムの流れは．．．

```
program stest1
   implicit none
   real x,y
   x = 5.0
   y = 100.0
   call subr(x,y,10)          !  サブルーチンの呼び出し
   print *,x,y
end program stest1

subroutine subr(x,y,n)
   implicit none
   real x,y
   integer n
   x = n
   y = y*x
end subroutine subr
```

サブルーチン実行の流れ

．．．このようになる

メインプログラム

プログラム領域

```
program stest1
  implicit none
  real x,y
  x = 5.0
  y = 100.0
  call subr(x,y,10)
  print *,x,y
end program stest1
```

実行の流れ

サブルーチン

プログラム領域

```
subroutine subr(x,y,n)
  implicit none
  real x,y
  integer n
  x = n
  y = y*x
end subroutine subr
```

★ サブルーチンの途中で実行を終了したいときは，return文を書く

★ サブルーチンの中で stop 文を実行すると，プログラムが終了する

```
      return                ! return文
      stop                  ! stop文
```

メインプログラムとサブルーチンの位置関係

- サブルーチンとメインプログラムの順序は動作と無関係

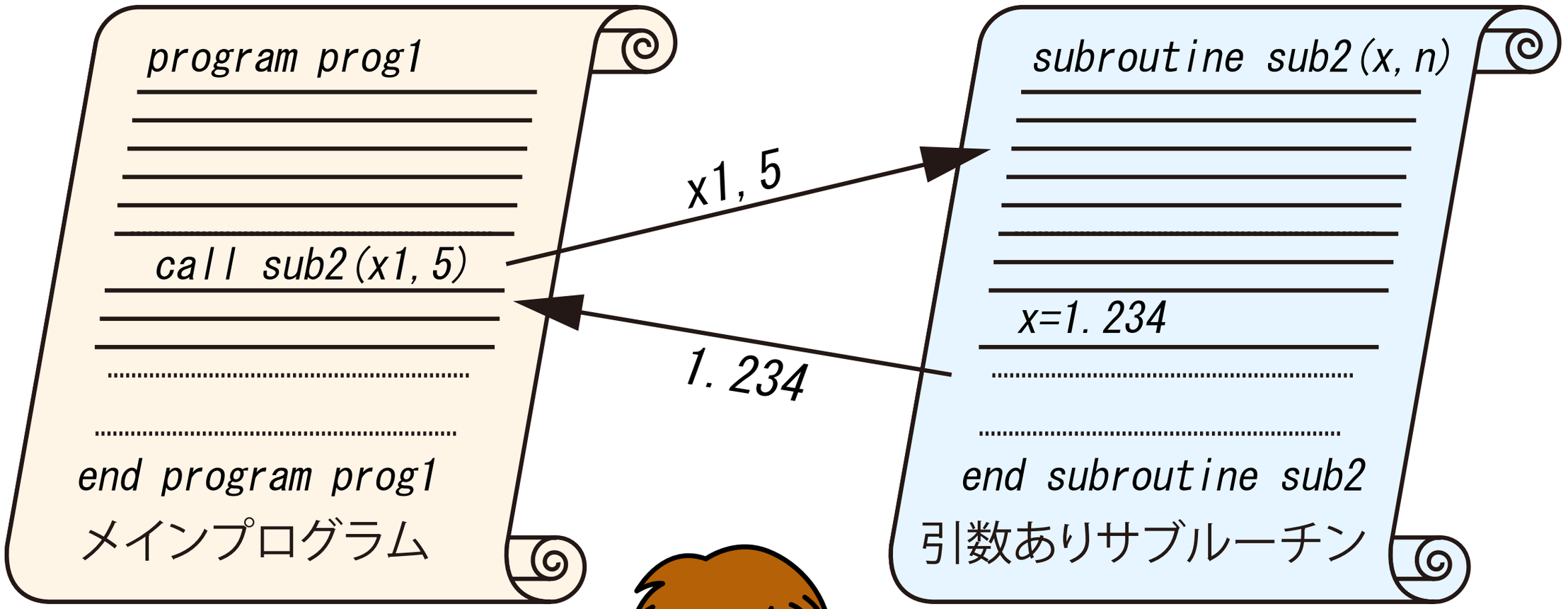
```
program stest1
   ...
   call subr(x,y,10)
   ...
end program stest1
subroutine subr(x,y,n)
   ...
end subroutine subr
```

＝

```
subroutine subr(x,y,n)
   ...
end subroutine subr
program stest1
   ...
   call subr(x,y,10)
   ...
end program stest1
```

- 一つのプログラムの中に入れられるメインプログラムは１個だが，サブルーチンは，名前が異なればいくつでも入れられる
- サブルーチンは記述するだけでもＯＫ．どこからもコールされなくてもエラーにはならない
- メインプログラムとサブルーチンを別のファイルにして結合することも可能　→　これが「リンク」
- 数値計算などに関する既存のサブルーチンをリンクさせることも可能　→　これが「ライブラリ」

2．引数の使い方


program prog1
call sub2(x1,5)
end program prog1
メインプログラム
x1,5
1.234
subroutine sub2(x,n)
x=1.234
end subroutine sub2
引数ありサブルーチン

ローカル変数

- サブルーチン内部で宣言した変数は，他のルーチンでの宣言とは無関係，これを「ローカル変数」という
- このため，ルーチンが異なれば同じ名前の変数を宣言してもエラーにはならないし，全く異なるメモリを指定するので実体も異なる

☆ **あるルーチンから他のルーチンの内部は見えない！**

例えば，次のように書いても問題ないが，数値の受け渡しはできない

```
program stest1
  implicit none
  real x,y
  x = 10.0
  y = 30.0
  call subr1
end program stest1

subroutine subr1
  implicit none
  real x,y
  print *,x,y            ! このxとyは，10や30ではない！
end subroutine subr1
```

数値の受け渡し手段の一つが引数

数値の受け渡しには「引数」を使う

```
program stest2
  implicit none
  real x,y
  x = 10.0
  y = 30.0
  call subr2(x,y)          ! 引数ありサブルーチンの呼び出し
end program stest2

subroutine subr2(x,y)      ! 引数ありの subroutine文
  implicit none
  real x,y                 ! 引数は宣言しなければならない
  print *,x,y
end subroutine subr2
```

★ 引数を使えば，数値の受け渡しができる

★ 「引数」は，データの窓口であるが，あくまでもローカル変数なので，数値型や名前などはサブルーチン側で決める

★ このため，call文とsubroutine文での数値の対応には気をつけなくてはいけない

```
call subr2(10,30)   ! このcall文にすると期待通りの結果にはならない!
                    ! call subr2(10.0,30.0)と書かなければならない
```

プログラムを書いてみよう

3-B　変数の交換用サブルーチン

例題3-B

実変数 x と y を与えると，それに代入されている値を交換して戻るサブルーチンを作成し，実際に交換されているかどうかを確かめよ

3-B　変数の交換用サブルーチン の解答例

```
program answer3b
    implicit none
    real x, y
    x = 10;  y = 50
    print *, ' x, y (before) = ', x, y
    call swap(x, y)
    print *, ' x, y (after)  = ', x, y
end program answer3b

subroutine swap(a, b)
    implicit none
    real a, b, c
    c = a
    a = b
    b = c
end subroutine swap
```

☆書き方のポイント☆

★ サブルーチン内で引数変数に値を代入するとcall文の変数に代入される

★ この機能を利用するとき，call文では必ず「変数」を指定すること

引数を使うときの規則

- 並びの順番や数は，全て一致していなければならない
- call文の数値は，対応するsubroutine文の引数と数値型が一致していなければならない
- サブルーチン内で引数変数に数値を代入すると，call文の対応する変数に値が代入される．これを「戻り値」という
- 戻り値を利用する場合には，call文の対応する引数に，
  必ず「同じ数値型を持つ変数」を指定しなければならない

☆誤った使い方☆

```
program answer3b
    implicit none
    real x, y
    call swap(x,  x+y)    ! このcall文はエラー（数式には代入できない）
    call swap(20.0,y)     ! このcall文もエラー（定数には代入できない）
end program answer3b

subroutine swap(a, b)
    implicit none
    real a, b, c
    c = a
    a = b
    b = c
end subroutine swap
```

配列を引数にする場合

- 引数には，配列を指定することもできる

☆ 配列名 a の配列を使って，call sub(a) と呼び出したとき

subroutine文の引数が単一変数として宣言されているときには，
配列 a の先頭要素を使ってサブルーチンが動作する

```
subroutine sub(x)
  implicit none
  real x                          !  単一変数宣言
  x = 10.0
end subroutine sub
```

サブルーチン側で配列として使いたければ，引数を配列として宣言する

```
subroutine sub(x)
  implicit none
  real x(10)                      !  配列宣言
  integer i
  do i = 1, 10
    x(i) = i
  enddo
end subroutine sub
```

★ なお，コール側と要素数が一致する必要はない（見えない！）

★ 次元でさえ一致させる必要はない

プログラムを書いてみよう

3-C　ベクトルの内積を計算するサブルーチン

例題3-C

整数 n と，長さ n の２個の１次元配列 a と b を引数に与えると，
それぞれを n次元ベクトルと考えたときのベクトルの内積を計算して
その結果を戻り値とするサブルーチンを作成し，結果を確かめよ.

☆n次元ベクトルの内積☆

$\boldsymbol{A} = (A_1, A_2, \cdots, A_n)$,　$\boldsymbol{B} = (B_1, B_2, \cdots, B_n)$　として

$$\boldsymbol{A} \bullet \boldsymbol{B} = A_1 B_1 + A_2 B_2 + \cdots + A_n B_n$$

3-C　ベクトルの内積を計算するサブルーチン の解答例

```
program answer3c
   implicit none
   real a(10), b(10), pab
   integer i, n
   n = 10
   do i = 1, n
      a(i) = i
      b(i) = n-i
   enddo
   call inprod(a, b, n, pab)
   print *, ' A.B = ', pab
end program answer3c

subroutine inprod(x, y, n, p)
   implicit none
   real x(10), y(10), p
   integer n, i
   p = 0
   do i = 1, n
      p = p + x(i)*y(i)
   enddo
end subroutine inprod
```

☆書き方のポイント☆

★ サブルーチン内の配列宣言は「10」である必要はない

★ この例の場合には，10 の代わりに，引数 n でも良い（整合配列）

整合配列

- Fortranでは，サブルーチンの引数配列を他の引数の中にある整数型の変数を使って，あたかもその長さの配列のように宣言することができる

☆これを「整合配列」という☆

例えば，先ほどの解答例のサブルーチンは以下のように書ける

```
subroutine inprod(x, y, n, p)
   implicit none
   real x(n), y(n), p       ! nが引数なので整合配列が可能
   integer n, i
   p = 0
   do i = 1, n
      p = p + x(i)*y(i)
   enddo
end subroutine inprod
```

★ ２次元以上の配列でも可能

整合配列はなぜ可能か？

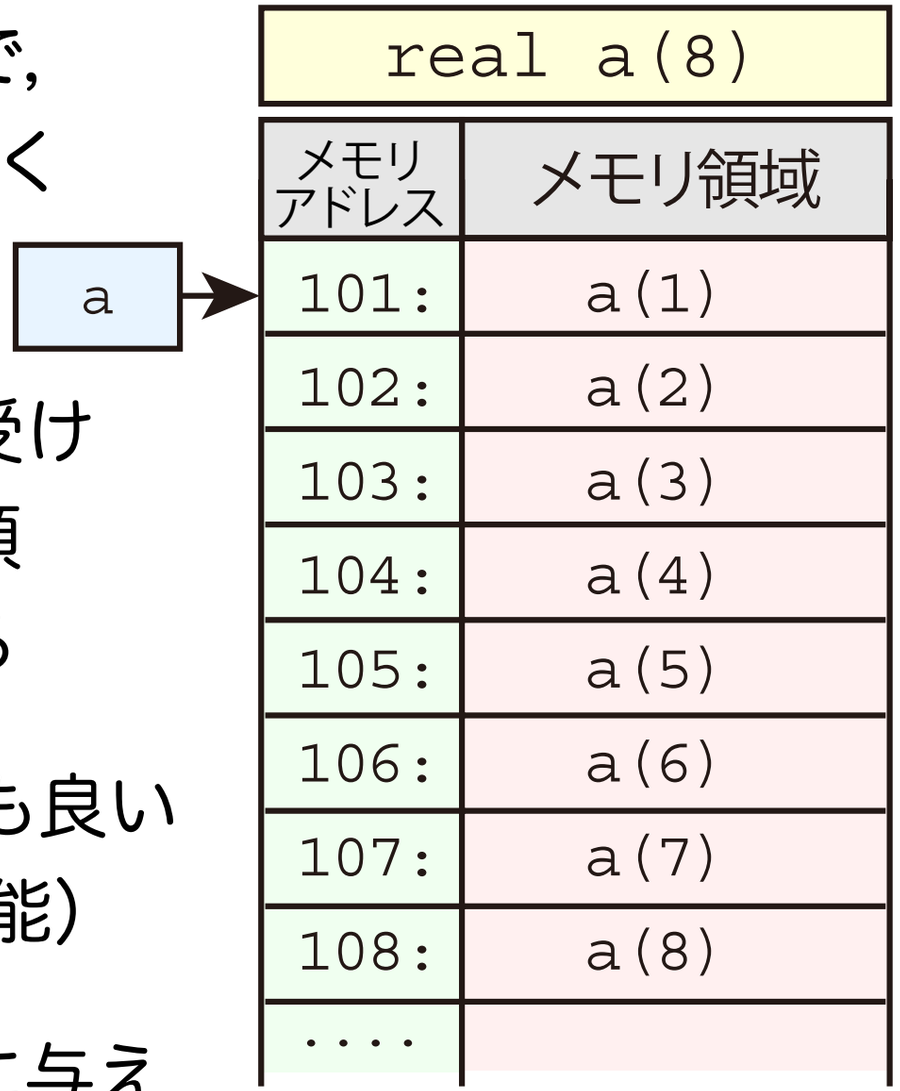
- サブルーチンにおける「引数」は特別な存在で，call文から受け取っているのは「数値」ではなく「アドレス」である

- 引数が配列の場合には，その先頭アドレスを受け取っているだけであり，他の要素は，その先頭アドレスから数えてメモリの位置を決めている

- よって１次元配列の場合は，宣言数 n は何でも良い（ * を使って a(*) のように宣言することも可能）

- 下限を指定した配列は，下限を意識して引数に与えないと処理する位置が変わるので注意！

- ２次元以上の配列では，配列要素の上限がわからないといけないので，整合配列が有用である



配列を引数にするときの注意

例えば，サブルーチン sub の引数を，整合配列を使って右のように宣言した場合を考える

```
subroutine sub(x,n)
  implicit none
  real x(n)
  integer n
```

もし，コール側のプログラムで

```
real a(10)
```

と宣言した配列をcall 文の引数に使う場合，

```
call sub(a,10)　と　call sub(a(1),10)
```

は，同じ意味になる

また

```
call sub(a(3),8)　　！ a(3)からスタートするので，要素数は 8
```

と書けば，a(3)が先頭要素の配列と見なすので，a(3)とx(1)が対応する

もし，下限を指定して，

```
real ac(0:10)
```

と宣言した配列を call 文の引数に使った場合，

```
call sub(ac,11)　　！ ac(0)からac(10)までなので，要素数は 11
```

と書けば，ac(0)が配列の先頭要素なので，ac(0)とx(1)が対応する

☆サブルーチンで，real x(0:n) と宣言すれば，ac(0)とx(0)が対応する

プログラムを書いてみよう
3-D　行列の和を計算するサブルーチン

例題3-D

整数 n と，n×n の２個の２次元配列，a と b を引数に与えると，それぞれを n次正方行列と考えたときの行列の和を計算して，別の２次元配列に代入して戻るサブルーチンを作成し，結果を確かめよ.

3-D　行列の和を計算するサブルーチン の解答例

```
program answer3d
   implicit none
   real a(5,5), b(5,5), c(5,5)
   integer i, j, n
   n = 5
   do j = 1, n
     do i = 1, n
       a(i,j) = i*j
       b(i,j) = 2*i-j
     enddo
   enddo
   call matrix_sum(a, b, n, c)
   do i = 1, n
     print *, i, (c(i,j), j=1, n)
   enddo
end program answer3d
```

```
subroutine matrix_sum(x, y, n, z)
   implicit none
   real x(n,n), y(n,n), z(n,n)
   integer n, i, j
   do j = 1, n
     do i = 1, n
       z(i,j) = x(i,j) + y(i,j)
     enddo
   enddo
end subroutine matrix_sum
```

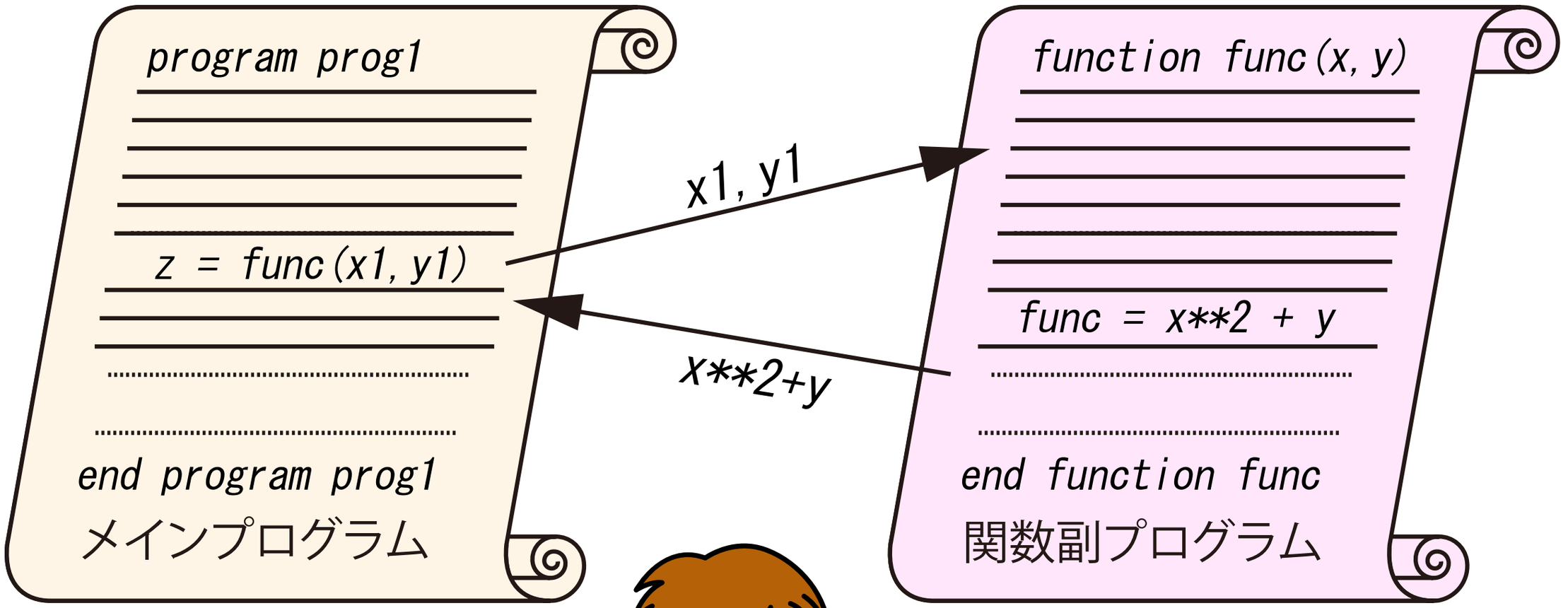
☆書き方のポイント☆

★　２次元配列を受け渡すときには，整合配列を利用すると便利

★　この例では，第１要素と第２要素の宣言がどちらも nであるが，別の引数 m を加えて，real x(m,n) のように宣言することも可能

★　配列の出力に do型出力を使用している（第２部 付録2C参照）

3．関数副プログラム


program prog1
z = func(x1,y1)
end program prog1
メインプログラム
x1,y1
x**2+y
function func(x,y)
func = x**2 + y
end function func
関数副プログラム

関数副プログラムとは

- y=3*fun(x) のように数式中で使える自前の関数を作ることもできる
  これを「関数副プログラム」という
- 関数副プログラムは，サブルーチンとほぼ同じ構造だが「関数名」が
  １個の変数であり，その変数が関数値として数式中で使えるものである
- このため，使うときには，関数名の型宣言をしなければならない

☆関数副プログラムとサブルーチンの相違点☆

★ 関数副プログラムは，function文と end function文の間に書く

★ 関数名を変数として宣言し，それに計算結果（関数値）を代入し
  なければならない

例えば,

```
function square(x)                    !  function文
  implicit none
  real square,x                       !  関数名と引数の型宣言
  square = x*x                        !  関数名に値を代入する
end function square                   !  end function文
```

のように書く

プログラムを書いてみよう
3-E　ベクトルの長さを計算する関数

例題3-E

整数 n と，長さ n の１次元配列 a を引数に与えると，a の要素を
n次元ベクトルの成分と考えたときのベクトルの長さを関数値とする
関数副プログラムを作成し，それを使って適当な成分値を代入した
２個の n 次元ベクトルの長さの合計を計算して出力するプログラム
を作成せよ.

☆n次元ベクトルの長さ☆

$\boldsymbol{A} = (A_1, A_2, \cdots, A_n)$ として

$$|\boldsymbol{A}| = \sqrt{A_1^2 + A_2^2 + \cdots + A_n^2}$$

3-E　ベクトルの長さを計算する関数 の解答例

```
program answer3e
    implicit none
    real a(10), b(10), sum, length
    integer i, imax
    imax = 10
    do i = 1, imax
      a(i) = 2.5*i + 0.5
      b(i) = 1.25*i - 10
    enddo
    sum = length(a, imax) + length(b, imax)
    print *, ' Length(a)+Length(b) = ', sum
end program answer3e
```

```
function length(x, n)
    implicit none
    real length, x(n), xl
    integer n, i
    xl = 0
    do i = 1, n
       xl = xl + x(i)**2
    enddo
    length = sqrt(xl)
end function length
```

☆書き方のポイント☆

★ 関数を利用するメインプログラムでは，関数の名前を型宣言する

★ 関数副プログラム中でも関数名を宣言し，その関数名を変数として数値を代入すれば，その数値が関数値となる

result句を使った関数副プログラム

- result句を利用すれば関数名に代入する形にしなくても良い

例えば,

```
function square(x) result(y)  ! result句を付加
  implicit none
  real x,y                    !  関数名ではなく, result句で指定した変数を型宣言
  y = x*x                     !  result句で指定した変数に関数値を代入する
end function square
```

のように書くことができる

利用するときは「関数名」を型宣言しなければならないのは同じ

```
program ftest1
  implicit none
  real x,y,square                    !  使用する関数名を型宣言する
  x = 5.2
  y = 3.0*square(x+1.0) + 50.5
  print *,x,y
end program ftest1
```

4．グローバル変数

module mod1
real xax, yax
end module mod1
モジュール
program prog1
xax = 10
call sub1
end program prog1
メインプログラム
subroutine sub1
yax = xax**2
end subroutine sub1
サブルーチン

グローバル変数を使って変数を共有する

- 大きなプログラムになると引数だけでデータを受け渡すのは不便
- 複数のルーチンで値を共有したいときには「グローバル変数」を利用すると便利
- Fortranではモジュールを使ってグローバル変数を利用する
- モジュールにはより高度な使い方もある（本講座では説明を省略）

モジュールの構造

```
module モジュール名
   宣言文1
   宣言文2
    ........
end module モジュール名
```

★ モジュールはメインプログラムやサブルーチンとは独立しているので外部に書く

★ 利用する全てのルーチンより前に書く

★ モジュールの中から他のモジュールを利用することも可能

モジュールの利用手段： use 文

- グローバル変数を利用するルーチン内で，use文を使ってモジュールの利用を宣言する
- 逆に use文で宣言しなければ，そのモジュールで宣言されたグローバル変数は使えない（同じ名前をローカル宣言しても良いが，別物）
- use文はimplicit noneよりも前に書く

use文

```
use モジュール名
```

例えば，次のように書くことができる

```
module global          ! モジュールは前に書く
  real xaxis,yaxis
end module global

program stest4
  use global           ! implicitより前でuse宣言
  implicit none
  xaxis = 5.0
  yaxis = 100.0
  call subr4
  print *,xaxis,yaxis
end program stest4

subroutine subr4
  use global           ! implicitより前でuse宣言
  implicit none
  print *, xaxis,yaxis
  yaxis = 25.0
end subroutine subr4
```

モジュール利用のイメージ

module global

データ領域	
xaxis	5.0
yaxis	25.0

メインプログラム

プログラム領域

```
program stest4
  use global
  implicit none
  xaxis = 5.0
  yaxis = 100.0
  call subr4
  print *,xaxis,yaxis
end program stest4
```

参照

参照

サブルーチン

プログラム領域

```
subroutine subr4
  use global
  implicit none
  print *,xaxis,yaxis
  yaxis = 25.0
end subroutine subr4
```

★ 外部の共有メモリ領域を参照するところがポイント

プログラムを書いてみよう
3-F　連成振動の解析

例題3-F

複数個の質量 1 のおもりがバネ定数 k のバネで x 方向に 1 次元的に結合しているとき，
i 番目のおもりの運動方程式は以下のようになる

$$\frac{dx_i}{dt} = v_i \qquad \frac{dv_i}{dt} = -k(x_i - x_{i-1}) - k(x_i - x_{i+1}) \qquad i = 0 \sim i_{max}$$

この微分方程式を以下のサブルーチンを使ったプログラムで解いて，結果を出力せよ

① x_iとv_iの初期値を代入するサブルーチン

② 微分方程式の右辺を計算するサブルーチン

☆メインプログラムとサブルーチンとの共通変数の受け渡しにはグローバル変数を使うこと

初期条件は以下の通りとする，また左右両端のおもりは動かない（i=0, i=i_{max}は固定）

$$x_i = i + A\sin(\pi i / i_{max}) \qquad v_i = 0 \qquad i = 0 \sim i_{max}$$

なお，微分方程式の解法には，もっとも簡単なオイラー法を使う

オイラー法とは，関数 $y_1(t), y_2(t), ..., y_M(t)$ に対するM元連立微分方程式

$$\frac{dy_i}{dt} = f_i(t, y_1, y_2, ..., y_M) \qquad i = 1 \sim M$$

を，１ステップの時間幅をΔtとして，sステップの関数値からs+1ステップの関数値を以下で推定する手法である.

$$y_i^{s+1} = y_i^s + f_i(t, y_1^s, y_2^s, ..., y_M^s)\Delta t \qquad i = 1 \sim M$$

例えば，i_{max}=6，A=0.1，k=1，Δt=0.01 で2000ステップ計算せよ

3-F　連成振動の解析 の解答例

グローバル変数

```
module xandv
  real xp(0:6), vl(0:6), xx(0:6)
  real dxp(5), dvl(5), ks, amp
  integer imax
end module xandv

program answer3f
   use xandv
   implicit none
   real dt
   integer nt, ntmax, i
   imax = 6
   ks   = 1
   amp  = 0.1
   dt   = 0.01
   ntmax = 2000
   call initial
   do nt = 1, ntmax
     call equation
     do i = 1, imax-1
      xp(i) = xp(i) + dxp(i)*dt
      vl(i) = vl(i) + dvl(i)*dt
     enddo
     print *,nt, xp
   enddo
end program answer3f

subroutine initial
   use xandv
   implicit none
   real pi
   integer i
   pi = acos(-1.0)
   do i = 0, imax
     xp(i) = i + amp*sin(pi*i/imax)
     vl(i) = 0
   enddo
end subroutine initial

subroutine equation
   use xandv
   implicit none
   integer i
   do i = 1, imax-1
     dxp(i)= vl(i)
     dvl(i)=-ks*(xp(i)-xp(i-1))-ks*(xp(i)-xp(i+1))
   enddo
end subroutine equation
```

☆書き方のポイント☆

- ★ 係数などの共通変数もグローバル変数にすると使いやすい
- ★ モジュールで宣言するグローバル変数は意味のある長い名前にする
- ★ do 文のカウンタ変数などは，ローカル変数にする
- ★ i の範囲が0からなので配列の下限を０にしている
- ★ オイラー法は精度が良くないので，より精度の高い解法に改良して下さい

グローバル変数を利用するときの注意

- むやみやたらにグローバル変数にしない
- 使途に応じて変数を分類し，それぞれのモジュールを作る
- グローバルであることを意識した意味のある長めの変数名にする

必要とするグローバル変数が少ないときは，宣言を限定することもできる

```
use モジュール名，only ： 変数１，変数２，．．．
```

例えば，xaxsisとyaxsisが宣言されているモジュールを考える

```
module global
  real xaxis,yaxis
end module global
```

このモジュールglobalを使うときに必要なのが yaxsis のみの場合には，

```
use global, only : yaxsis
```

と書けば，　yaxsisのみ使える

xaxsisは使えないし，　別のローカル変数として宣言してもかまわない

第３部　終了

Fortranには便利な書き方がまだまだあります

- 便利なデータ入力方法
- 文字列の処理
- 配列計算
- モジュールの高度な使い方

★ 東北大学より，もう少し詳しい内容が入った
解説書をダウンロードすることができます

★ より高度に使いこなしてみたいと思ったら
他の解説書を読んで勉強して下さい

数値計算法も含めて勉強するなら
拙著もあります！

Fortranハンドブック
田口俊弘 著
技術評論社

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

これで今回の講習会は終了です

お疲れ様でした！

付録：知っておくと便利な文法３

3A. 拡張宣言文

これまでの型宣言文では，数値型の指定と変数名（配列）のみ記述

```
型指定　変数1, 変数2, ...
```

拡張宣言文では，属性や初期数値を代入することができる

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

★ 拡張宣言文には，連続した2個のコロン「::」 が必要である

★ なお，この文で［...］は省略可能という意味で使っているだけなので，
“[” と “]” はプログラムには書かないようにして下さい

例えば，

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

のように宣言と同時に数値代入しておくと，この値でプログラムが開始する

拡張宣言文における数値代入の意味

- 拡張宣言文での数値代入は最初の１回だけである
- 実行文中で変更したら変更後の値が残る

例えば，次の例ではサブルーチン subr1をコールするごとに出力が異なる

```
program stest1
  implicit none
  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
```

★ すなわち，宣言文での代入は，プログラム開始後の１回だけ

ここで，サブルーチンのローカル変数に値が保持されていることに注目！

一時メモリ領域と固定メモリ領域

- 実は，サブルーチン内で宣言した変数はサブルーチン終了後に消滅すると考えなければならない　→　一時メモリ領域

例えば，次のプログラムを考えてみよう

```
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
```

この場合，２回目のコールで10と100が出力されるという保証はない！

☆ 数値代入しておくと，消滅しない　→　固定メモリ領域

save属性を付加すると固定メモリになる

- 数値代入しないで固定メモリ領域に所属させるには，save属性を付ける

```
integer, save :: 変数1, 変数2, ...
```

★ この変数に代入した値は，サブルーチン終了後も保持される

先ほどのプログラムは，

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

subroutine subr2(n)
  implicit none
  integer n
  real, save :: x,y              ! save属性を付加
  if (n == 2) print *,x,y        ! この出力値は10.0と100.0
  x = 10.0
  y = 100.0
end subroutine subr2
```

のように修正すれば，期待通りの結果になる

3B. parameter 変数

- 配列宣言は「定数」で範囲を指定しなければならないが，これは，プログラムの開始時にメモリを確保しなければならないためである
- 配列の数が増えると修正が面倒である

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
```

100を変更したいときは，3カ所修正が必要！

parameter属性を付加したparameter変数を使えば，変数で宣言できる

```
integer, parameter :: imax=100        ! 修正はここだけでよい
real a(imax),b(imax)                  ! imax で宣言ができる
integer i
do i = 1, imax
  a(i) = i*i
  b(i) = a(i)**2
enddo
```

parameter 変数は置き換えるだけ

- parameter変数を使って，別のparameter変数を定義したり，下限指定や計算式を使った指定も可能

例えば，

```
integer, parameter :: imax=100, imax2=imax**2
real a(imax-1),b(0:imax*2),c(-imax2:imax2)
```

は，以下のプログラムと等価である

```
integer, parameter :: imax=100, imax2=10000
real a(99),b(0:200),c(-10000:10000)
```

☆ ただし，parameter変数は，コンパイル時に単語を数字に置き換えているだけなので，実行文で変更することはできない

例えば，次のように書くとコンパイルエラーになる

```
integer, parameter :: imax=100
imax = 5                    ! これはエラー
```

なぜなら，プログラム的には次のように書いたことになるから

```
integer, parameter :: imax=100
100 = 5                     ! このプログラムを書いたことになるので
```

parameter 変数を活用しよう

- parameter変数をモジュールで用意すれば，長いプログラムで便利

```
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)          ! グローバル配列宣言
end module mod_array

program mtest1
  use mod_array                             ! use global_paramは不要
  implicit none
  integer km(imax)                          ! ローカル配列宣言
   .......
```

- 物理定数などを用意しておくのも便利（CODATA 2018推奨値）

```
module physical_constants
  real, parameter :: clight =299792458.0,       hplanck=6.62607015e-34
  real, parameter :: echarge=1.602176634e-19,   kboltz =1.380649e-23
  real, parameter :: emass  =9.1093837015e-31,  pmass  =1.67262192369e-27
end module physical_constants
```

3C. 配列の動的割り付け

動的割り付け：実行時に必要に応じたメモリ領域を確保すること

- Fortranでの動的割り付けには２つの方法がある

動的割り付けその１：引数整数で宣言する

例えば,

```
    subroutine memory1(a,m)
      implicit none
      real a(m),b(m)            ! 引数 m を使ってローカル配列を宣言する
      integer m,i
      do i = 1, m
         b(i) = 2.5*i - 3
         a(i) = b(i)**3
      enddo
    end subroutine memory1
```

と書くことができる

★ このサブルーチンで，a(m)は整合配列

★ b(m)はこのサブルーチン内で割り付けられたメモリ領域

★ あまり大きなサイズを割り付けると，エラーになることもある

配列の動的割り付け

動的割り付けその２：allocatable宣言とallocate文

- 引数による動的割り付けは，サブルーチンでしか使えない
- メインプログラムで使う配列やグローバル配列を割り付けたいときには，allocatable 宣言と allocate文を使って，より明示的に割り付ける

割り付け手順　その１

割り付ける予定の配列は，allocatable属性を付けて宣言する

```
real, allocatable :: ab(:),z2(:,:)     ! ab は１次元配列，z2 は２次元配列
integer, allocatable :: km1(:,:,:)     ! km1 は３次元配列
```

★ その際，配列要素数は不要だが，次元は後から決められないので，コロンの数で指定しておく

割り付け手順　その２

必要に応じて allocate文で割り付けを実行する

```
allocate ( ab(100), z2(0:m,-m:m), km1(k-1,2*k,5) )    ! 両端のかっこは必須！
```

★ このように，変数や数式を使って割り付けることも可能

★ 数値型に関係なく１回の allocate文で複数の配列を割り付けられる

動的割り付け配列の解放と確認

allocatable宣言とallocate文による割り付けの特長

★ メインプログラムでも可能

★ モジュールの配列もallocatable宣言ができる

★ 必要に応じてメモリを解放することができる

割り付けたメモリ領域の解放：deallocate文

```
deallocate ( ab, km1 )                    ! 両端のかっこは必須！
```

★ deallocate文では配列名のみ記述する

★ 数値型に関係なく１文で複数の配列を解放できる

割り付けられているかどうかの確認：allocated関数（論理型）

例えば，配列 a が割り付けられていないときのみ割り付けたいときは，

```
if (.not.allocated(a)) allocate ( a(n) )
```

のように書くことができる（論理型は 第２部 付録2B参照）

例えば例題2-Cの解答は以下のように書き換えられる

```
program answer2c_alloc
    implicit none
    real, allocatable :: a(:)
    real sum
    integer i, imax
    imax = 30
    allocate ( a(imax) )
    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
    deallocate ( a )
    print *,' Average = ', sum/imax
end program answer2c_alloc
```

☆書き方のポイント☆

★ このプログラムなら，原理的には imax がいくつでも動作可能

★ プログラムが終了すればメモリは解放されるので，このプログラムでは deallcate 文が無くても問題はない