Fortran90/95入門と演習
後半

担当： 臼井英之、三宅洋平
（神戸大学大学院システム情報学研究科）

目標

- 本スクールで用いる数値計算用プログラム言語「Fortran90/95」の基礎を習得する。

参考資料：

TECS-KOBE第二回シミュレーションスクール(神戸大学）2010/12/6 :Fortran 講義ノート（平尾　一）

「Fortran90/95入門」2010年度計算科学演習I　講義資料、神戸大院システム情報学専攻・陰山聡
（http://exp.cs.kobe-u.ac.jp/wiki/comp_practice/2010/index.php）

予定

1. イントロダクション
2. 入出力
3. 変数の型
4. 演算の基礎
5. 条件の扱い
6. 繰り返し処理
7. 配列
8. 副プログラム
9. 数値計算に向けて
10. 付録

前半

後半

繰り返し処理

Doループ

例1

1~10まで出力したい

```
program sample_do
!-------------------------------
  implicit none
  integer :: i
!-------------------------------
  do i=1,10
    write(6,*) i
  end do
!-------------------------------
end program sample_do
```

制御変数iの増分値はデフォルトで1

例2

```
program sample_do2
!-------------------------------
  implicit none
!-------------------------------
  write(6,*) 1
  write(6,*) 2
  write(6,*) 3
  write(6,*) 4
  write(6,*) 5
  write(6,*) 6
  write(6,*) 7
  write(6,*) 8
  write(6,*) 9
  write(6,*) 10
!-------------------------------
end program sample_do2
```

面倒、非効率的

×

1以外の増分値を使う場合

```
do i=1,100,2
 …
end do
```

結果

```
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
```

増分値が規則的な時に便利

多重ループ

例　　i, j の値を出力している

```
program sample_domulti
!--------------------------------
  implicit none
  integer :: i, j
!--------------------------------
  write(6,'(a)') "   i   j"
  write(6,'(a)') "----------"
  do i=1,3
    do j=1,3
      write(6,'(2i4)') i, j
    end do
  end do
!--------------------------------
end program sample_domulti
```

結果

```
   i   j
----------
   1   1
   1   2
   1   3
   2   1
   2   2
   2   3
   3   1
   3   2
   3   3
```

内側のループが先に回る

Doループにおけるcycle文とexit文

例1

```
program sample_docycle
!-------------------------------
  implicit none
  integer :: i
!-------------------------------
  do i=1,10
    if(i == 3) cycle
    write(6,*) i
  end do
!-------------------------------
end program sample_docycle
```

次の制御変数の処理へ飛ぶ

結果

```
 1
 2
 4
 5
 6
 7
 8
 9
10
```

例1

```
program sample_doexit
!-------------------------------
  implicit none
  integer :: i
!-------------------------------
  do i=1,10
    if(i == 3) exit
    write(6,*) i
  end do
!-------------------------------
end program sample_doexit
```

Doループから抜け出る

結果

```
 1
 2
```

Doループの活用：和の計算

演習b1

例　1~10の和をとる

```
program sample dosum
!--------------------------------
  implicit none
  integer :: i, isum
!--------------------------------
  isum = 0  初期値として0を設定
  do i=1,10
    isum = isum + i
  end do
  write(6,*) isum
!--------------------------------
end program sample_dosum
```

isumに、順にiを足し込んで行く

結果

```
./a.out
          55
```

和を求めるのに非常によく使うパターン （重要）

```
isum = isum + i
```

もともとのisum の値（右辺）にiを加えて、isumに新しい値を入れる（左辺）
→“＝”は「代入」の意味

演習: sample_dosum.f95を作成および実行し、プログラムの意味を理解せよ。

配列

一次元配列

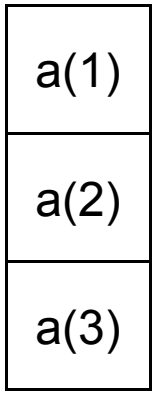
例

```
program sample_array
!-------------------------------
  implicit none
  #JISSU#
  integer :: i
  real(SP), dimension(3) :: a
!-------------------------------
  do i=1,3
    a(i) = real(i,SP)
    write(6,*) a(i)
  enddo
  write(6,*)
  write(6,*) (a(i),i=1,3)
  write(6,*) a
!-------------------------------
end program sample_array
```

値の代入

空行出力

要素を全部出力



1.0

2.0

3.0

ベクトルを定義できる

結果

```
   1.000000
   2.000000
   3.000000

   1.000000        2.000000        3.000000
   1.000000        2.000000        3.000000
```

二次元配列

例

```
program sample_array2
!-------------------------------
  implicit none
  integer :: i, j
  integer, dimension(3,3) :: a
!-------------------------------
  do i=1,3
    do j=1,3
      a(i,j) = 10*i + j
    enddo
    write(6,'(3i4)') (a(i,j),j=1,3)
  enddo
!-------------------------------
end program sample_array2
```

3×3の配列を宣言

例えば(2,1)要素が21となるようにしている

出力

配列のイメージ

a(1,1)	a(1,2)	a(1,3)
a(2,1)	a(2,2)	a(2,3)
a(3,1)	a(3,2)	a(3,3)

結果

```
  11  12  13
  21  22  23
  31  32  33
```

行列を定義できる

3次元以上の配列も定義可能

課題2

① 前ページのプログラムを参考、もしくはひな形にして次のプログラムを作成せよ。

“3x3配列a(i,j)に「単精度実数型」の10*i+j値を入れ、出力する。”
（ヒント：配列の宣言、write文における書式指定子）

② ①のプログラムとその出力結果をテキストファイル
（result_130509_1.txt）にまとめ、
臼井（usui）までメイルで送ってください。

mail –s “メールアドレス_130509_1” usui < result_130509_1.txt

配列演算のための組込み関数

例

```
program sample_array3
!-------------------------------
  implicit none
  integer :: i, j
  integer, dimension(3,3) :: a, b, c
!-------------------------------
  do i=1,3
    do j=1,3
      a(i,j) = 10*i + j
    enddo
  enddo

  write(6,'(3i6)')
 ((a(i,j),j=1,3),i=1,3)
  write(6,*) sum(a)
  write(6,*) size(a,1), size(a,2)

  b = matmul(a,a)
  write(6,'(3i6)')
 ((b(i,j),j=1,3),i=1,3)

  c = transpose(a)
  write(6,'(3i6)')
 ((c(i,j),j=1,3),i=1,3)
!-------------------------------
end program sample_array3
```

値の代入

行列の積

aの転置行列

結果

```
a
     11    12    13
     21    22    23
     31    32    33

b
    776   812   848
   1406  1472  1538
   2036  2132  2228

c
     11    21    31
     12    22    32
     13    23    33

            198
              3                3
```

Σa(i,j)

size(a,1)

a(1,1)	a(1,2)	a(1,3)
a(2,1)	a(2,2)	a(2,3)
a(3,1)	a(3,2)	a(3,3)

size(a,2)

便利な配列演算法

例

```
do i=1,10
  do j=1,10
    C(i,j) = A(i,j) + B(i,j)
  enddo
enddo
```

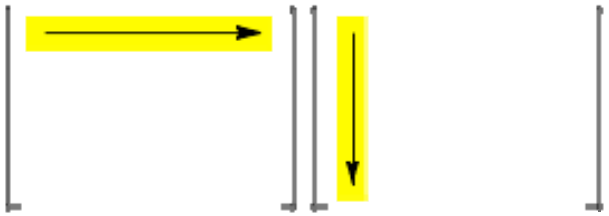
要素を全部書いて和(A + B)を
計算するための古いやり方

```
C = A + B
```

同じ計算するための簡潔な書き方

```
do i=1,10
  do j=1,10
    AT(i,j) = A(j,i)
  enddo
enddo
do i=1,10
  do j=1,10
    s = 0.0
    do k=1,10
      s = s + AT(i,k) * B(k,j)
    enddo
    C(i,j) = s
  enddo
enddo
```

$C = {}^tAB$ の計算をするための古い書き方



```
C = matmul(transpose(A),B)
```

簡潔な書き方

内積は dot_product(a,b)

プログラムがシンプルになる例(1) 演習b2

$$\sum_{i=1}^{\infty} \frac{1}{i} \cdot \frac{1}{i+1} \cdot \frac{1}{i+2} = \frac{1}{1} \cdot \frac{1}{2} \cdot \frac{1}{3} + \frac{1}{2} \cdot \frac{1}{3} \cdot \frac{1}{4} + \frac{1}{3} \cdot \frac{1}{4} \cdot \frac{1}{5} + \cdots = \frac{1}{4}$$

例

```
program sample_series
  implicit none
  #JISSU#
  integer, parameter :: nterms = 1000
  real(SP), dimension(nterms) :: x, y, z
  integer :: i

  do i = 1, nterms
     x(i) = 1.0 / i
     y(i) = 1.0 / (i+1)
     z(i) = 1.0 / (i+2)
  end do

  print *, 'ans = ', sum(x*y*z)

end program sample_series
```

配列の積→要素の和を計算

課題３

① 前ページのプログラムを作成し、動作を確認せよ。
ただし、そのときループ数ntermsを標準入力から指定できるように
プログラムを修正すること。
数種類のntermsに対する計算結果を調べてみよ。

② ①のプログラムとその出力結果をテキストファイル
（result_130509_2.txt）にまとめ、
臼井（usui)までメイルで送ってください。

mail –s “メールアドレス_130509_2” usui < result_130509_2.txt

配列の動的な割り付け

例

```
program sample_arrayallocate
!-------------------------------
  implicit none
  integer :: nmax
  integer, dimension(1000,1000) :: a
  integer, dimension(:,:), allocatable :: b
!-------------------------------
  write(6,'(a)',advance='no') "nmax: "
  read(5,*) nmax
  allocate(b(nmax,nmax))
  write(6,*) size(a,1), size(a,2)
  write(6,*) size(b,1), size(b,2)
  deallocate(b)
!-------------------------------
end program sample_arrayallocate
```

サイズを固定

サイズは後で決める

advance='no'は「改行しない」

サイズを決めた（1~nmaxに割り付けた）
i.e., b(1:nmax,1:nmax)

メモリを解放した

結果

```
nmax: 50
        1000         1000
          50           50
```

必要な分だけメモリを確保する

配列要素の初期値の設定

例

```
program sample_array5
!-------------------------------
  implicit none
  #JISSU#
  integer :: i, j
  integer, dimension(3,3) :: a = 1
  integer, dimension(3,3) :: b
  integer, dimension(3) :: c = (/1, 2, 3/)
  real(DP), dimension(3,3) :: d
!-------------------------------
  b = 2
  d = sqrt(real(b,DP))
  write(6,'(3i3)') ((a(i,j),j=1,3),i=1,3)
  write(6,'(3i3)') ((b(i,j),j=1,3),i=1,3)
  write(6,'(3i3)') (c(i),i=1,3)
  write(6,'(3f10.6)') ((d(i,j),j=1,3),i=1,3)
!-------------------------------
end program sample_array5
```

結果

```
  1  1  1
  1  1  1
  1  1  1
  2  2  2
  2  2  2
  2  2  2
  1  2  3
 1.414214  1.414214  1.414214
 1.414214  1.414214  1.414214
 1.414214  1.414214  1.414214
```

便利な配列処理

例1

```
real(DP), dimension(NX,NY)    :: array02d
real(DP), dimension(NX,NY,NZ) :: array03d
do j = 1 , NY
  do i = 1 , NX
    array03d(i,j,1) = arrya02d(i,j)
  end do
end do
```

これまではこう書いていたが、
こう書ける。

```
array03d(:,:,1) = arrya02d(:,:)
```

例2

```
real(DP), dimension(10) :: A
real(DP), dimension(15) :: B
do i = 1 , 10
  A(i) = B(i+5)
end do
```

```
A(:) = B(6:15)
```

シンプルに記述できる

副プログラム

サブルーチン、関数、モジュール

サブルーチン　　演習b3

例

```
program sample_subroutine
  implicit none
  #JISSU#
  real(SP) :: x, y
  write(6,*) "x?"
  read(5,*) x
  call nijo(x,y)
  write(6,'(a,f8.4)') 'x   = ', x
  write(6,'(a,f8.4)') 'x^2 = ', y
end program sample_subroutine

!================================
subroutine nijo(x,y)
  implicit none
  #JISSU#
  real(SP), intent(in) :: x
  real(SP), intent(out) :: y
  y = x**2
!  x = y
end subroutine nijo
!================================
```

x,y: 引数

Subroutineの呼び出し

x,y: 仮引数

入力用変数

出力用変数

値を二乗するサブルーチン

結果

```
 x?
2.5
x   =   2.5000
x^2 =   6.2500
```

特定の作業をsubroutineとしてまとめておき、callで呼び出す

演習: subroutine内のx=yを実行してみよ。xの属性をinoutとしたときの結果も確認せよ。

引数／仮引数と入出力属性

引数／仮引数

①主プログラム

```
…
call subtest(x,y)
…
```

引数

②サブルーチン

```
subroutine subtest(a,b)
…
```

仮引数

- 引数と仮引数の名前は違っていてもよいが、順番と型を揃えなければならない
- サブルーチンから別のサブルーチンを呼んでもよい

入出力属性

Intent(in): ①→②へと渡される変数（変更不可）
Intent(out): ②の処理の結果として①へと戻る値
Intent(inout): 両方の性質を持つ仮引数

入出力属性の指定はFortranのメリット

関数

例

```
program sample_function
  implicit none
  #JISSU#
  real(SP) :: nijo
  real(SP) :: x
  write(6,*) "x?"
  read(5,*) x
  write(6,'(a,f8.4)') 'x^2 = ', nijo(x)
end program sample_function

!================================
function nijo(x)
  implicit none
  #JISSU#
  real(SP) :: nijo
  real(SP), intent(in) :: x
  nijo = x**2
end function nijo
!================================
```

結果

```
 x?
2.5
x^2 =   6.2500
```

関数名そのものが戻り値のようになっている

sin(x)のような組込み関数を思い出してみましょう

モジュール：定数をまとめる

例1

```
module module_constants
  implicit none
  integer, parameter :: SP = kind(1.0)
  integer, parameter :: DP = selected_real_kind(2*precision(1.0_SP))
  real(DP), parameter :: pi = 3.14159265358793238_DP
  real(DP), parameter :: planck = 6.62606896e-34_DP
end module module_constants

program sample_module1
  use module_constants
  implicit none
  write(6,*) pi
  write(6,*) planck
end program sample_module1
```

useによってモジュールの使用を宣言

結果

```
   3.141592653589793
  6.626068599999996E-034
```

- 同じ定数を何度も定義する必要がなくなる
- ModuleはmainプログラムLの前に置く

データ、型などをひとまとめにできる。必要なモジュールだけ使う

モジュール：変数の共有

例2

```
module module_com_var
  implicit none
  integer, parameter :: SP = kind(1.0)
  integer, parameter :: DP = selected_real_kind(2*precision(1.0_SP))
  real(DP) :: shared_var = 1.0_DP
end module module_com_var
```

モジュール内で変数宣言（および初期化）

```
program sample_module1
  use module_com_var
  implicit none
  write(6,*) shared_var
  call substitute
  write(6,*) shared_var
end program sample_module1

subroutine substitute
  use module_com_var
  implicit none
  shared_var = 10.0_DP
end subroutine substitute
```

結果

```
 1.000000000000000
 10.00000000000000
```

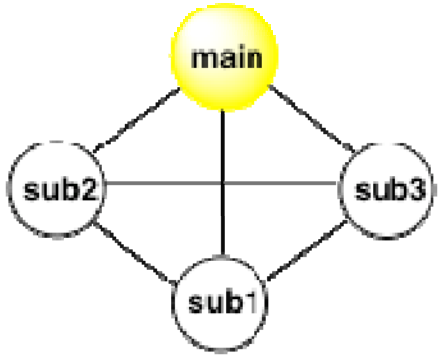
- 引数・仮引数が不要
- ただしサブルーチンの再利用性を低めてしまうことに注意

（例ではshared_var専用のサブルーチンになっている）

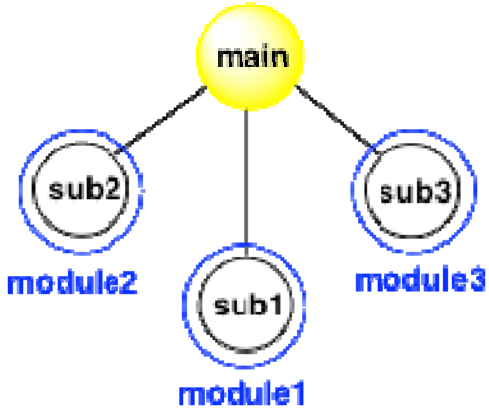
複数のプログラム単位間で変数を共有

モジュール：カプセル化

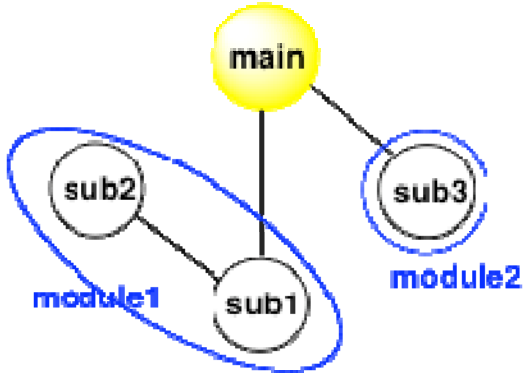
1



2



3



1: あちこちと相互作用でき、例えばsub1を改良すると、いろんなところに影響が及ぶ可能性あり。
2: 1モジュールの中に1サブルーチンを入れ、サブルーチン間の相互作用を断ち切った（**情報の隠蔽**）。
3: 一つの「機能」をなすsub1, sub2をひとまとめにし、sub1の下請け的ルーチンであるsub2はsub1とのみ相互作用するようにした。

Moduleを機能単位でまとめ、相互作用を減らし、独立性を高める→プログラムの保守に有利

つなぎ方は操作できる

モジュール：モジュール内副プログラム

演習b4

例

```
module module_kinou
  implicit none
  private
  public :: pihello
  #JISSU#

  contains

  subroutine pihello()
    call shitauke()
  end subroutine pihello

  subroutine shitauke()
    write(6,*) 4.0*atan(1.0_DP)
    write(6,'(a)') "hello"
  end subroutine shitauke

end module module_kinou
```

型宣言はmoduleの最初でのみ行えばよい

デフォルトではprivate（module内でしか参照できない）。public宣言で外部からも参照できる

モジュール内のサブルーチンや関数の直前に置く

Publicの、「πを計算・出力しhelloと言う」機能の窓口的業務を担うルーチン

Privateのサブルーチン（実際に作業を行うルーチン。

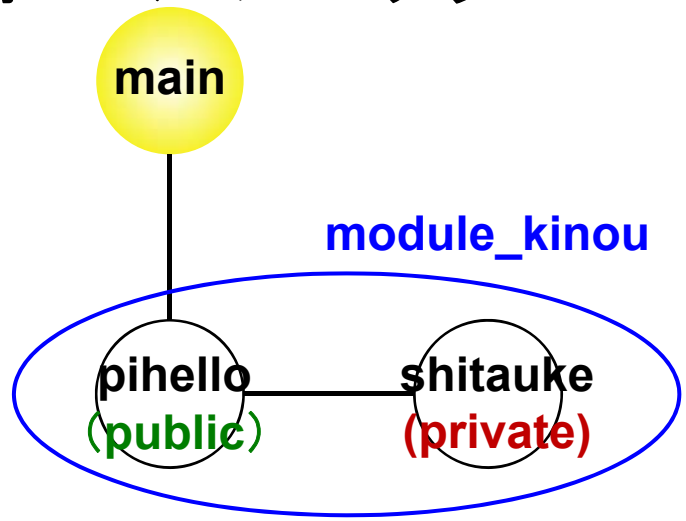
```
3.1415926535897931
hello
```

```
program sample_module6
  use module_kinou
  implicit none
  call pihello()
! call shitauke()
end program sample_module6
```

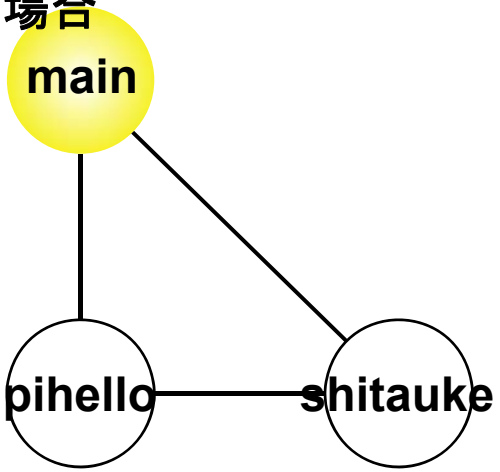
演習: /home/cs/usui/pub/sample_module6.f95を自分の作業ディレクトリにコピーし、動作確認。
また、shitaukeをmainからcallして動作確認

モジュール：モジュール内副プログラム

前ページのプログラム



参考：すべて外部副プログラムとして定義した場合



右の例: あちこちと相互作用でき、一つのプログラム単位を改良すると、いろんなところに影響が及ぶ可能性あり。
左の例: モジュールの中にサブルーチンを入れることにより、モジュールの使用を宣言した場合（use module_kinou）のみアクセス可能。
さらにpihelloの下請け的ルーチンであるshitaukeにはprivate属性を指定し、モジュール内のpihelloのみ外から呼び出せるようにした。

数値計算に向けて

配列処理の効率

例1

```
program sample_time1
!------------------------------
  implicit none
  integer, parameter :: nmax=10000
  integer :: i, j
  integer, dimension(nmax,nmax) :: a
!------------------------------
  do i=1,nmax
    do j=1,nmax
      a(i,j) = i + j
    enddo
  enddo
!------------------------------
end program sample_time1
```

0:02.60 (s)　遅い

例2

```
program sample_time2
!------------------------------
  implicit none
  integer, parameter :: nmax=10000
  integer :: i, j
  integer, dimension(nmax,nmax) :: a
!------------------------------
  do j=1,nmax
    do i=1,nmax
      a(i,j) = i + j
    enddo
  enddo
!------------------------------
end program sample_time2
```

0:00.41 (s)　速い

timeコマンドによる計測

```
% time ./a.out
real    0m2.597s
user    0m2.523s
sys     0m0.074s
```

経過時間

同等の作業なのに処理時間に大きな差が出ることがある

配列要素のメインメモリ上での配置

Fortran: A(2,2)のとき A(1,1), A(2,1), A(1,2), A(2,2)

C: a[2,2]のとき a[0][0], a[0][1], a[1][0], a[1][1]　　Fortranとは逆

非効率的なプログラム：0:02.60 (s)

```
do i=1,nmax
  do j=1,nmax
    a(i,j) = i + j
  enddo
enddo
```

アクセス順：
a(1,1), a(1,2),a(1,3),…
→不連続メモリアクセス

効率的なプログラム：0:00.41 (s)

```
do j=1,nmax
  do i=1,nmax
    a(i,j) = i + j
  enddo
enddo
```

アクセス順：
a(1,1), a(2,1),a(3,1),…
→連続メモリアクセス

左側の添字を先に動かした方が効率的。
「参照局所性」の向上により、「キャッシュメモリ」を有効利用。

おつかれさまでした

コメント、質問等は

h-usui@port.kobe-u.ac.jp

までお願いします

付録

基礎事項

文字列の処理

例

```
program sample_character3
  implicit none
  character(len=*), parameter :: moji1 = "hyogo"
  character(len=*), parameter :: moji2 = "kobe"
  character(len=10) :: moji3 = "nada"
  character(len=10) :: moji4 = "ku"
  write(6,'(a)') moji1(2:4)
  write(6,'(a)') moji1//"-"//moji2
  write(6,'(a)') moji3//"-"//moji4
  write(6,'(a)') trim(moji3)//"-"//trim(moji4)
  write(6,*) len(moji3), len_trim(moji3)
end program sample_character3
```

2-4文字目まで
//で文字列の連結
余白を削る
文字列の長さ

実行

```
yog
hyogo-kobe
nada      -ku
nada-ku
           10           4
```

整数型から文字型への変換

例

```
program sample_transformSeMo
!-------------------------------
  implicit none
  integer :: i
  character(len=*), parameter :: base="file."
  character(len=4) :: serial_num
!-------------------------------
  do i=1,10
    write(serial_num,'(i4.4)') i
    open(i,file=base//serial_num)
    write(i,'(a,i4)') "File number = ", i
    close(i)
  enddo
!-------------------------------
end program sample_transformSeMo
```

文字型定数の場合、len=*とできる

文字型変数serial_numに、整数iが文字列として入る

0	0	0	1

0	0	0	2

演習：本プログラムをコンパイル・実行し、何が起こるか観察せよ。

ファイル入力(2)：rewind文

例

```
program sample_input4
  implicit none
  integer :: n1, n2
  open(10,file="input")
  read(10,*) n1, n2
  write(6,*) n1, n2
!  rewind(10)
  read(10,*) n1, n2
  write(6,*) n1, n2
  close(10)
end program sample_input4
```

“input”

```
100 200      ①
300 400      ②
```

実行（rewind無効の時）

```
 100  200
 300  400
```

実行（rewind有効の時）

```
 100  200
 100  200
```

開いているファイルの、先頭に戻る

Do While文

例1

```
program sample_dowhile
!-------------------------------
  implicit none
  integer :: i
!-------------------------------
  i = 1                    初期値
  do while (i <= 10)       条件
    write(6,*) i
    i = i + 1              増分
  end do
!-------------------------------
end program sample_dowhile
```

結果

```
           1
           2
           3
           4
           5
           6
           7
           8
           9
          10
```

「do i=…」でできる作業

例2

```
program sample_dowhile2
!-------------------------------
  implicit none
  #JISSU#
  real(DP) :: a = 1.0_DP
!-------------------------------
  do while (a > 0.1_DP)
    a = a/2.0_DP
    write(6,'(f12.6)') a
  end do
!-------------------------------
end program sample_dowhile2
```

結果

```
    0.500000
    0.250000
    0.125000
    0.062500
```

a>0.1である限り、a/2を
繰り返し行っている

増分値が規則的でない時に便利

Open文とClose文

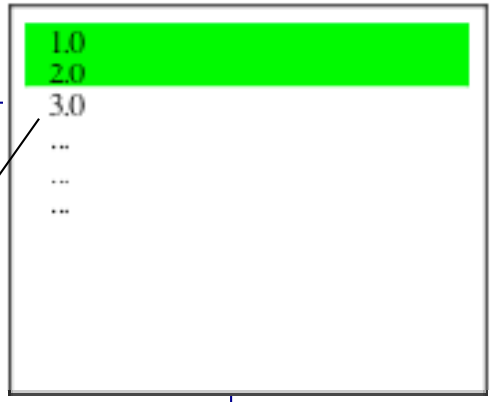
```
open(番号,file=filename)
close(番号)
```

ファイルを開ける
閉じる

例1

```
open(1,file="input")
read(1,*) …
read(1,*) …


プログラムの後半
read(1,*) …
```



closeしなかったら、前回の続きになる

例2

```
open(1,file="input")
read(1,*) …
read(1,*) …
close(1)

プログラムの後半
open(1,file="input")
read(1,*) …
close(1)
```

Closeした場合、再定義が必要

必要な処理が終わったらファイルをcloseするように心がける

その他の入出力操作：リダイレクション

例

```
program hello_world
  implicit none
  print *,   "hello, world."
end program sample_output
```

標準出力

実行例

```
% ./hello_world > output

% ./hello_world >> output

% ./hello_world >& output
```

① すでにoutputに何か書かれていた場合、今回の出力で上書きされる

② 古い内容の下に追加する形で出力

③ エラー出力（コンパイルのエラーメッセージ等）をoutputへ

標準出力内容がファイルに書き出される

演習: すでに作成したhello_worldを使い、
1. ①→①の処理
2. ①→②の処理

を行ってそれぞれの場合のoutputの中身を確認せよ。

その他の入出力操作：リダイレクション(2)

例

```
program sample_input3
  implicit none
  integer :: n1, n2
  read(5,*)  n1, n2
  write(6,*) n1, n2
end program sample_input3
```

標準入力
標準出力

“input”

```
100 200
```

実行例

```
% ./sample_input < input > output
```

標準出力内容をoutputへ
標準入力内容をinputから

配列用組込み関数のまとめ

組込み関数	機能
dot_product(a,b)	ベクトルの内積
matmul(a,b)	行列a,bの積
transpose(a)	行列aの転置行列
maxval(a)	配列要素の最大値
minval(a)	配列要素の最小値
sum(a)	配列要素の和
lbound(a,dim=N)	配列の下限の大きさ
ubound(a,dim=N)	配列の上限の大きさ

等価な計算でも所要時間が異なる例

例

```
program sample_kumikomi
!-------------------------------
  implicit none
  #JISSU#
  integer, parameter :: nmax=10000000
  integer :: i, j
  integer, dimension(nmax,nmax) :: a
  real(DP) :: t1, t2, t3, x, y
!-------------------------------
  call cpu_time(t1)
  do i=1,nmax                      ┐
    x = 1.0_DP                     │
    y = exp(x)*exp(x)              ├  処理A
  end do                           ┘
  call cpu_time(t2)
  do i=1,nmax                      ┐
    x = 1.0_DP                     │
    y = exp(x+x)                   ├  処理B
  end do                           ┘
  call cpu_time(t3)
  write(6,'(2f12.6)') t2-t1, t3-t2
!-------------------------------
end program sample_kumikomi
```

実行

```
  0.473538     0.232642
```

例

遅い	速い
exp(x)*exp(y)	exp(x+y)
log(m)+log(n)	log(m*n)
sin(θ)cos(θ)	0.5*sin(2*θ)
x**3	x*x*x
x**3 + x**2 + 1	x*x*(x+1)+1

組み込み関数やべき乗は、命令は単純だが
実際の演算量は少なくない（テーラー展開等）
ので時間がかかる

グラフ用入力ファイルの作成

例

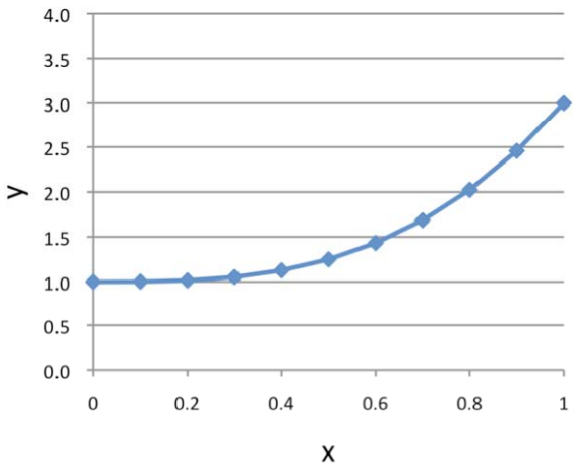
```
program sample_graph
!-------------------------------
  implicit none
  #JISSU#
  integer :: i
  real(DP) :: x, y
!-------------------------------
  do i=1,10
    x = i*0.1_DP
    y = 2.0*x**3 + 1.0
    write(6,’(2f12.6)’) x, y
  end do
end program sample_graph
```

$$y = 2x^3 + 1 \ (0 \leq x \leq 1)$$

実行

```
    0.100000    1.002000
    0.200000    1.016000
    0.300000    1.054000
    0.400000    1.128000
    0.500000    1.250000
    0.600000    1.432000
    0.700000    1.686000
    0.800000    2.024000
    0.900000    2.458000
    1.000000    3.000000
```

グラフソフトへ



リンク

複数のソースコードへ分割

例 二つのファイルに分けた

module_constants.f95

```
module module_constants
  implicit none
  #JISSU#
  double precision, parameter :: pi = 3.14159265358979323 8_DP
  double precision, parameter :: planck = 6.62606896e-34_DP
end module module_constants
```

sample_module.f95

```
program sample_module
  use module_constants
  implicit none
  write(6,*) pi
  write(6,'(e20.15)') planck
  write(6,*) 1.0_DP
end program sample_module
```

moduleの利用

ディレクトリの様子

```
% ls
module_constants.f95  sample_module.f95
```

機能毎にファイルを分けて整理整頓

オブジェクトファイルのリンク

（不完全な）各ソースコードをコンパイルする

```
% frtpx -c module_constant.f95
% frtpx -c sample_module.f95
```

module_constant.oができる
sample_module.oができる

リンク

```
% frtpx -o sample_module.exe module_constant.o sample_module.o
```

sample_module.exeができる

各ソースコードをコンパイル→オブジェクトファイルを作成→リンク

Make

例（Makefile）

```
F95  = frtpx
.SUFFIXES:
.SUFFIXES: .f95 .o
OBJS = module_constants.o ¥
       sample_module.o
sample_module.exe: ${OBJS}
        ${F95} -o sample_module.exe ${OBJS}
.f95.o:
        ${F95} -c $<
clean:
        rm -f *.o *.mod *.exe
```

SpaceではなくTABにすること

依存関係

.f95から.oの作成方法を記述

cleanの方法

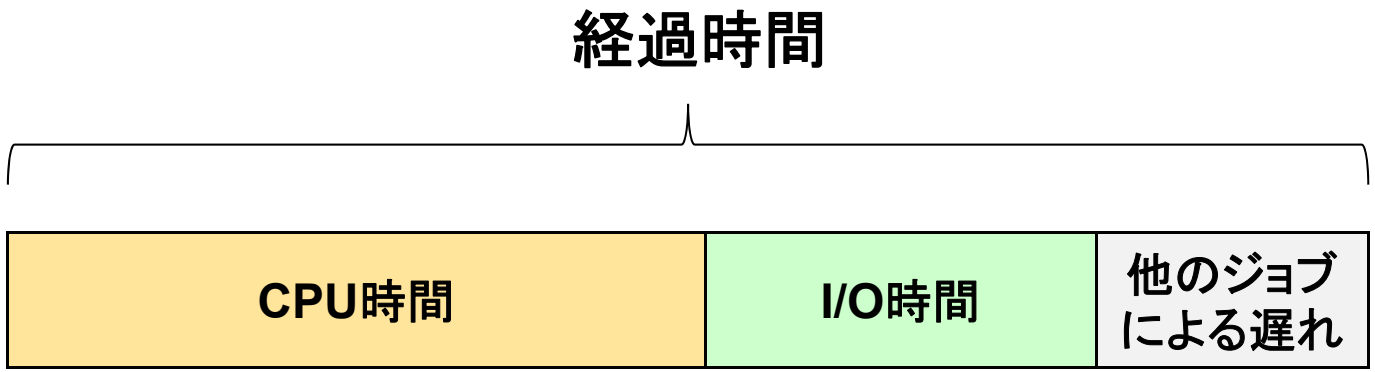
実行

```
% ls
Makefile  module_constants.f95  sample_module.f95
% make
frtpx -c module_constants.f95
frtpx -c sample_module.f95
frtpx -o sample_module.exe module_constants.o sample_module.o
```

基本的には新たに編集したファイルだけを再コンパイル。多数のソースファイルがからなるプログラムのコンパイルを効率化

数値計算の効率化

処理にかかる時間



CPU時間
-プログラムが消費する時間
-OSが消費する時間

I/O時間
- データの読み書きに使う時間

反復回数の少ないdoループの展開

例

```
program sample_doexpand
!-------------------------------
  implicit none
  #JISSU#
  integer, parameter :: nmax=100000000
  integer :: i, j
  integer, dimension(3,nmax) :: a
  real(DP) :: t1, t2, t3, x, y
!-------------------------------
  call cpu_time(t1)
  do i=1,nmax
    do j=1,3
      a(j,i) = a(j,i) + 1
    enddo
  enddo
  call cpu_time(t2)
  do i=1,nmax
    a(1,i) = a(1,i) + 1
    a(2,i) = a(2,i) + 1
    a(3,i) = a(3,i) + 1
  enddo
  call cpu_time(t3)
  write(6,'(2f12.6)') t2-t1, t3-t2
!-------------------------------
end program sample_doexpand
```

何度もこのループに到達し、初期設定が行われて時間をロスする

展開してあらわに書いた → 速くなる

実行

```
    1.872249    0.706740
```

ループに到達すると、「ループカウンタの初期設定」が行われ、時間を費やす

インライン展開による高速化

例

```
program sample_inline
!------------------------------
  implicit none
  #JISSU#
  integer, parameter :: nmax=100000000
  integer :: i
  real(DP) :: t1, t2, t3, y
!------------------------------
  call cpu_time(t1)
  do i=1,nmax
    call oneone(y)
  end do
  call cpu_time(t2)
  do i=1,nmax
    y = 1.0_DP + 1.0_DP
  end do
  call cpu_time(t3)
  write(6,'(2f12.6)') t2-t1, t3-t2
!------------------------------
end program sample_inline

!******************************
subroutine oneone(y)
  implicit none
  #JISSU#
  real(DP), intent(out) :: y
  y = 1.0_DP + 1.0_DP
end subroutine oneone
!******************************
```

単純作業をサブルーチンのコールで処理

同じことを直接行う

実行

```
    0.444058     0.126826
```

やりすぎるとプログラムが見にくくなるので注意