

現代的なFortran

総合演習 B

神戸大学

陰山

モダンFortranについて

- ▶ いま Fortranといえば下のどれかを指す
 - Fortran 95 （古い）
 - Fortran 2003 （現在主流）
 - Fortran 2008 （スーパーコンピュータではまだ完全にはサポートされていない）
- ▶ 注意:

Fortranは "FORTRAN" という「古代言語」の子孫だが、今となっては別の言語と考えたほうがよい

参考サイト

- ▶ この講義ではCやJavaとの違いに重点をおき、簡潔に説明する。Fortranの文法について詳しくは例えばNAG社の以下のページをみよ

- Fortran入門

<http://www.nag-j.co.jp/fortran/index.html>

- Fortran 2003入門

<http://www.nag-j.co.jp/fortran/fortran2003/index.html>

モダンなFortran

- ▶ C/C++やJavaとそれほど変わらない
 - 静的型付け言語。コンパイルしてから実行。オブジェクト指向
- ▶ 少し変わった特徴
 - 大文字と小文字の区別をしない（文字列変数は除く）
 - 予約語がない（例えばifという名前の変数を使ってもよい）
- ▶ HPC用言語として優れている点
 - 多次元配列の扱いが得意
 - 数式の取り扱いが得意
 - モジュールと内部副プログラムによる階層的名前空間が容易
 - 組み込み関数が豊富
 - 数学ライブラリを含めて科学技術関係のコード資産が豊富

整数型変数の宣言

```
integer :: i  
integer :: j, k  
integer :: m = 10
```

定数

```
integer, parameter :: nn = 100
```

```
integer, parameter :: NN = 100
```

- Fortranでは大文字と小文字は区別しない（文字列変数は除く）
- したがって上の二つはコンパイラにとっては同じ
- しかし、C言語やJavaの命名習慣と同様に定数は大文字にするとわかりやすい

C言語

```
const int nn = 10;
```

実数型

```
real :: a  
real :: b = 0.0  
real :: c = 1.23e-5
```

デフォルト精度の実数（浮動小数点数）

```
real(SR) :: a  
real(SR) :: b = 0.0_SR  
real(SR) :: c = 1.23e-5_SR
```

「種別番号」がSRで指定される精度をもつ実数（浮動小数点数）

種別番号（単精度実数）

```
integer :: SR = selected_real_kind(6)
```

有効桁数が6桁以上の実数を指定するときに使う種別番号をSRという定数に設定

← いわゆる単精度浮動小数点数（float）

```
real(SR) :: a  
real(SR) :: b = 0.0_SR  
real(SR) :: c = 1.23e-5_SR
```

C言語

```
float a;  
float b = 0.0;  
float c = 1.23e-5;
```


種別番号（倍精度実数）

```
integer :: DR = selected_real_kind(12)
```

有効桁数が12桁以上の実数を指定するときに使う種別番号をDRという定数に設定

← いわゆる倍精度浮動小数点数（double）

```
real(DR) :: a  
real(DR) :: b = 0.0_DR  
real(DR) :: c = 1.23456789e-5_DR
```

C言語

```
double a;  
double b = 0.0;  
double c = 1.23456789e-5;
```

定数 (実数)

```
integer :: SR = selected_real_kind(6)
integer :: DR = selected_real_kind(12)

real(SR), parameter :: PI_SINGLE = 3.1415_SR
real(DR), parameter :: PI_DOUBLE = 3.1415926535897932_DR
```

種別番号（単精度整数）

```
integer :: SI = selected_int_kind(8)
```

有効桁数が8桁以上の整数を使うときの種別番号をSIに設定

参考：4Bの整数 --> 符号1bit × $2^{31} = (2^{10})^{3.1} \sim (10^3)^{3.1} \sim 10^9$

```
integer(SI) :: i = 10000
```

```
integer :: DI = selected_int_kind(16)
```

有効桁数が16桁以上の整数を使うときの種別番号をDIに設定

参考：8Bの整数 --> 符号1bit × $2^{63} = (2^{10})^{6.3} \sim (10^3)^{6.3} \sim 10^{19}$

```
integer(DI) :: long_loop = 10000000000000000
```

コメント

! これはコメント

```
integer :: i = 10    ! これもコメント
```

標準出力への出力

```
print *, "i=", i
```

C言語

```
#include <stdio.h>
...
...
printf("i=%d\n", i);
```

行の分割

```
real(DR) :: variable_with_very_long_named &  
          = 2.00_DR
```

文字列

```
character(len=3) :: str3
character(len=4) :: str4
character(len=7) :: str7
```

`str3 = "abc"` ! ダブルクォーテーションでも

`str4 = 'defg'` ! シングルクォーテーションでもOK

`str7 = str3 // str4` ! 文字列の連結

`print *, "First two letters of str7 = ", str7(1:2)`

`str7(3:7) = " "` ! 5文字の空白

`print *, str7 // str3`

! => "ab abc"

`print *, trim(str7) // str3`

! => "ababc" trim関数は末尾の空白をとる

main プログラム

```
program name  
    implicit none  
  
end program name
```

- implicit noneを書かないと「暗黙の型宣言」が有効になる
- そうなるとややこしいので常に書くこと

C言語

```
int main(void)  
{  
  
}
```


関数

```
function func(i)
  integer(SI), intent(in) :: i
  real(DR) :: func
  ...
  func = i*2.0_DR
end function func
```

自動的な型変換:
整数 × 倍精度実数 ⇒ 倍精度実数

この i は入力専用の引数であることを明示
(func内でiを変更するとコンパイルエラー)

関数の返り値はその関数名
(resultを使い別の名前にすることもできる)

サブルーチン

```
subroutine sub(a, b)
  real(DR), intent(in) :: a
  real(DR), intent(out) :: b
  ...
  b = a*2
end subroutine sub
```

C言語の void func() に相当

関数/サブルーチン引数は参照渡し

```
subroutine sub(i)
  integer, intent(inout) :: i

  i = i*2
end subroutine sub
```

```
j = 10
print *, j      ! => 10
call sub(j)
print *, j      ! => 20
```

値が変更される

値渡しにしたい場合は
integer, intent(in), value :: i
とする

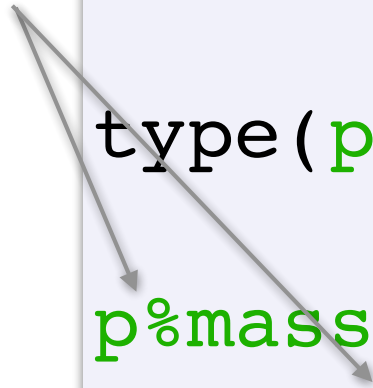
構造体 / クラス

```
type particle_t
  real(DR) :: mass
  real(DR) :: position_x
  real(DR) :: position_y
contains
  procedure :: move
end type particle_t

type(particle_t) :: p

p%mass = 1.0_DR
call p%move()
```

メンバーアクセス演算子は%



do ループ

```
do i = 1, 10
```

```
...
```

```
end do
```

```
do i = 1, 10, 2
```

```
... ! 奇数のみ実行
```

```
end do
```

```
do i = 1, 10
```

```
  do j = 1, 20
```

```
    ...
```

```
  end do
```

```
end do
```

do ループ

```
do
  ...
  if (条件) exit
  ...
end do
```

```
do while (条件)
  ...
end do
```

if文

```
if (i==0) ...
```

```
if (i>=0) then  
    ...  
else  
    ...  
end if
```

```
if (i>=0) then  
    ...  
end if
```

```
if (i>1) then  
    ...  
else if (i==0) then  
    ...  
else  
    ...  
end if
```

select case文

```
character(len=10) :: animal

select case (animal)
  case ('bird')
    print *, "tweet"
  case ('dog')
    print *, "bow"
  case ('cat')
    print *, "mew"
  case default
    print *, "..."
end select
```

各caseの最後の break は不要

配列

```
real(DR), dimension(4) :: array1d  
real(DR), dimension(4,2) :: array2d  
real(DR), dimension(4,2,8) :: array3d
```

1から始まる

a(1)	a(2)	a(3)	a(4)
------	------	------	------

多次元配列は左が優先 (C言語と逆)

a(1,1)	a(2,1)	a(3,1)	a(4,1)	a(1,2)	a(2,2)	a(3,2)	a(4,2)
--------	--------	--------	--------	--------	--------	--------	--------



メモリ空間中で連続している

計算機シミュレーションでは (速度面で) 重要な性質

(C言語での"擬似的多次元配列" a[2][4]の場合は必ずしもそうではないことに注意)

部分配列

```
real(DR), dimension(3:6) :: a  
real(DR), dimension(1:7:2) :: b
```

a(3)	a(4)	a(5)	a(6)
------	------	------	------

b(1)	b(3)	b(5)	b(7)
------	------	------	------

配列演算

```
real(DR), dimension(100) :: a, sin_a  
real(DR) :: sum_of_sin_a
```

```
do i = 1, 100  
    a(i) = ...  
end do
```

```
sin_a = sin(a)    ! 一行で全ての要素のsinをとる
```

```
sum_of_sin_a = sum(sin_a)  
! sum(): 配列の全要素の和をとる組み込み関数
```

配列演算

```
real(DR), dimension(100, 200) :: array_a, array_b
```

```
array_b(:, :) = 1.2_DR * array_a(:, :)
```

! 全ての要素を1.2倍する

! array_b = 1.2_DR * array_a とも書いても良い

```
array_b(:, :) = array_a(:, :) * array_b(:, :)
```

! array_aとarray_bの全要素を掛ける

! 行列の掛け算には組み込み関数のmatmulを使う

```
array_b = matmul(array_a, array_b)
```

モジュール

Moduleとは、

- 変数・関数・サブルーチン等をまとめて（カプセル化して）管理することができる便利な機能
- C / C++ や Javaには対応するものはない

メインプログラム同様、冒頭で
`implicit none`と書く

```
module constants_m  
  implicit none
```

変数
定数
型（構造体定義）など

```
contains
```

関数、サブルーチン

```
end module constants_m
```

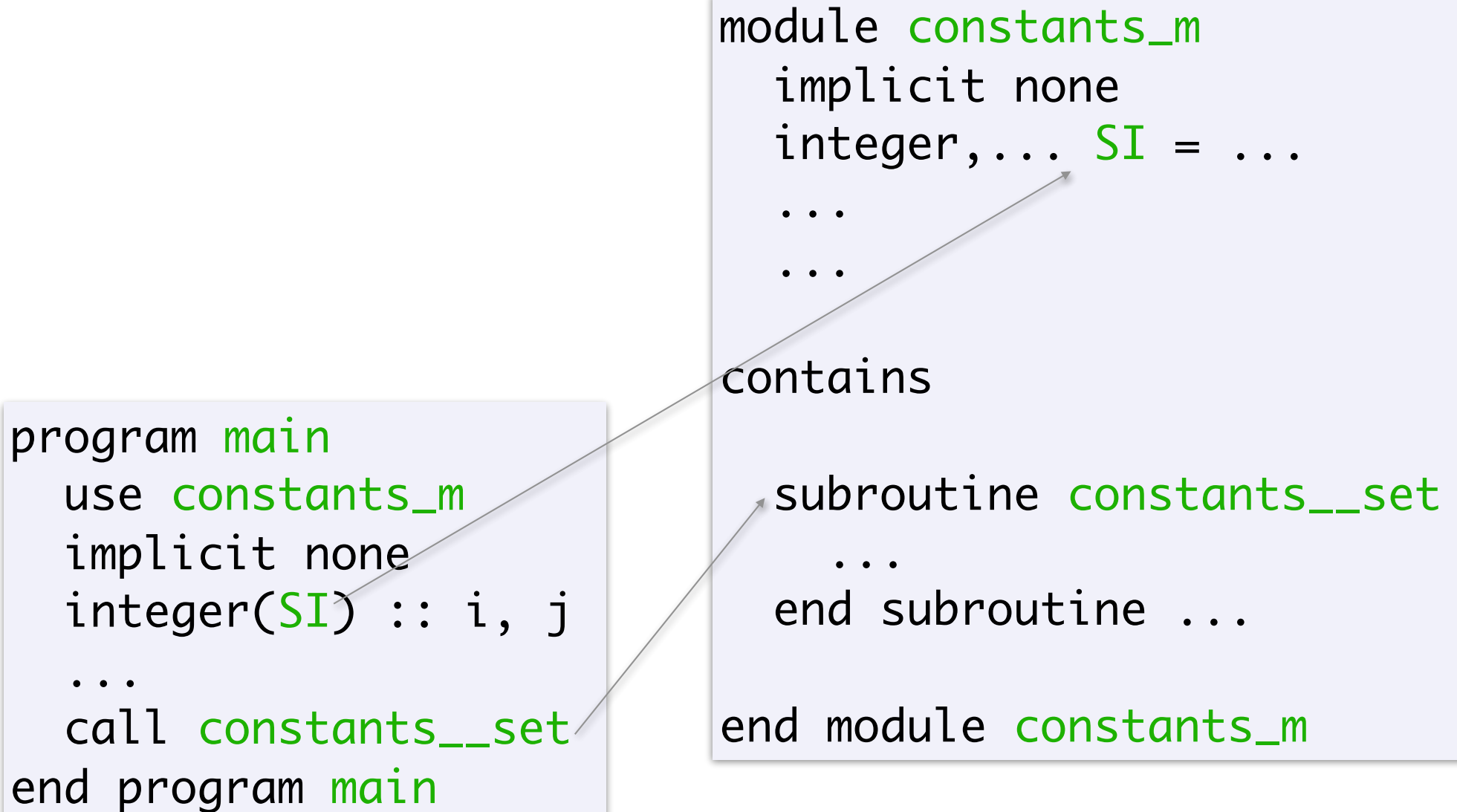
モジュール

```
program main
  use constants_m
  implicit none
  integer(SI) :: i, j
  ...
  call constants__set
end program main
```

```
module constants_m
  implicit none
  integer,... SI = ...
  ...
  ...
contains

  subroutine constants__set
    ...
  end subroutine ...

end module constants_m
```



モジュール

```
module constants_m
```

```
...
```

```
contains
```

```
subroutine constants__set
```

```
...
```

```
...
```

```
contains
```

```
subroutine test
```

```
end subroutine test
```

```
end subroutine ...
```

```
end module constants_m
```

モジュール内のサブルーチンの内部
にさらにサブルーチンを持つことが
できる

⇒ 名前空間の階層的な管理が容易！

モジュール

Moduleは、

- 変数や関数などのカプセル化だけでなくデータ隠蔽にも使える
- 冒頭でデフォルトprivate宣言をしておけばよい

```
program main
  use constants_m
  use ut_m
  implicit none
  integer(SI) :: i, j
  ...
  call ut__assert(...)
end program
```

```
module ut_m
  use constants_m
  implicit none
  private
  public :: ut__assert
  ...
  ...
```

デフォルト private

このサブルーチンは公開

```
contains

  subroutine ut__assert
    ...
  end subroutine ...
```

publicな変数や関数などはモジュール名（から_mを除いたもの）の後にアンダースコア二つをつけると便利。呼び出し側で宣言部を見つけやすくなる。アンダースコア二つがオブジェクトへのアクセス演算子として解釈できる。

ファイル

- ▶ Fortranファイルの拡張子について
 - Fortran 90, Fortran 95, Fortran 2003, Fortran 2008のコードのファイル拡張子は全て .f90
 - Fortran 2003 ファイルの拡張子として .f03 が使われることもあるが一般的ではない
- ▶ 一つのファイルに複数のモジュールを入れることも可能だが、以下のようなルールに従う方が便利
 - メインプログラムは一つのファイル (`main.f90` など)
 - モジュール一つは一つのファイル

例えば、`abc_m` モジュールは `abc.f90`

組み込み関数: dot_product

- ▶ v1とv2を1次元配列（ベクトル）として、内積は

```
value = dot_product(v1,v2)
```

- ▶ と書ける。これは以下と同じ

```
sum = 0.0_DR
```

```
do i = 1 , NX
```

```
    sum = sum + v1(i)*v2(i)
```

```
end do
```

組み込み関数: matmul

- ▶ a, bを行列（2次元配列）、v1, v2をベクトル（1次元配列）として、以下のように書ける

`c = matmul(a,b)` ! 行列aとbの積

`v2 = matmul(c,v1)` ! 行列とベクトルの積

組み込み関数: 行列の転置

- ▶ 行列（2次元配列） a の転置

`transpose(a)`

組み込み関数: 配列の最大・最小値

- ▶ 配列の全要素中の最大値と最小値

`maxval(a)`

`minval(a)`

組み込み関数: 配列の全要素の和

▶ 配列の全要素の和

`sum(a)`

- 応用：配列の全要素の2乗和の平方根

`sqrt(sum(a*a))`