Fortran スマートプログラミング

— 第２回　サブルーチンと入出力 —

田口　俊弘
摂南大学理工学部電気電子工学科

前回はFortranの基本的な文法とエラーの出にくいプログラムの書き方について説明しました。ここまでの知識を使うだけで，原理的にはどんなプログラムを作ることも可能です．プログラムとは計算機に仕事をさせるための手順ですから，それほどバラエティはありません．動作の繰り返しを書くためのdo文や，条件分岐のif文が使いこなせれば，計算機の能力のほとんどを使いこなすことができます．

しかし長いプログラムを書くときは，メンテナンスのしやすさを考慮して書かなければなりません．プログラムが短ければ全体を見ながらチェックや修正ができますが，長くなるとチェックに時間がかかり，見落としによる修正ミスが起こる可能性も高くなります．見落としを防ぐには何度も見直す必要があり，チェック時間はさらに増大します．

このメンテナンスを容易にする手法の一つが“サブルーチン”です．サブルーチンとはメインプログラムと同じレベルの閉じたプログラムのことで，必要に応じて他のプログラムから呼び出してその機能を使います．長いプログラムをサブルーチンに分割し，それらをビルディングブロックとして全体を構成すれば，プログラム作成や修正の作業が楽になります．今回はこのサブルーチンの作り方と使い方を説明します．

プログラムは計算をさせるだけでは意味がありません．結果を出力して初めて完結します．これまでは，もっとも単純なprint文でとりあえず画面に表示する方法を使ってきましたが，今回は好みに応じて数値の表示形式を変える方法や，ファイルに保存する方法について説明します．さらにデータを入力することでプログラムを変更せずに動作を変える方法についても説明します．

1.　サブルーチン

1.1　サブルーチンの利用目的

“ルーチン”とは，動作開始点と終了点が定義されている閉じたプログラム手順を示す用語です．プログラムを起動したときに最初に動作するのはメインプログラムですが，これを“メインルーチン”ともいいます．“サブルーチン”は，メインプログラムと同様に動作開始点と終了点を持っていますが，メインプログラムと異なり，単独で動作することはできません．必ず，他のルーチンに動作開始命令を記述して，必要に応じて実行させる必要があります．一つのプログラムにメインプログラムは一つですが，サブルーチンは名前が異なれば複数存在してもかまいません．また，いくつかのルーチンごとにファイルを作成して別々にコンパイルし，必要に応じて結合（リンク）させることも可能です．

サブルーチンを利用する目的は主として３つあります．

(1) 複数の場所で同じ計算手順を使用するため
(2) 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ため
(3) 長いプログラムを分割してメンテナンスを容易にするため

(1)がサブルーチン本来の使い方です．ある一連の手順を複数の場所で使うとき，それぞれの場所に同じプログラムを書くと，プログラムが長くなるし，修正の際に何カ所も同じ修正をしなければなりません．このとき，その手順をサブルーチンに置き換えれば，プログラムは短くなるし，チェック作業も短縮されます．

(2)は(1)を発展させたものです.複雑な数学公式や汎用性のある数値計算アルゴリズムをプログラム中に直接記述するときは，プログラム中で宣言された変数や配列を使って書きます．このため，同じ計算手順を別のプログラムで使いたくてもそのまま使えるとは限りません．そこで計算手順をサブルーチンにして，計算に必要な数値を与える部分と計算結果の数値を受け取る部分だけが外部から見えるようにしておきます．そうすれば，受け渡し部を書き換えるだけで，複雑な計算手順を色々なプログラムから利用することができます．このようにブラックボックス化したサブルーチンは，他の人に提供したり，他の人からもらって利用することも可能で，そのような汎用性のあるサブルーチンを集めたものが，“ライブラリ”です.

さて,計算機シミュレーションのように多数の解析手順を複合したプログラムを書く場合には,(3)の目的が重要になります．様々な物理過程の計算や，入出力に関する作業など，ある程度まとまった内容ごとにサブルーチンにするのです.文章でいえば,章や節に分割するようなものです.このため，メインプログラムには，それぞれのサブルーチンを呼び出す文と，必要ならそれらを繰り返すための do 文だけ書いておきます．シミュレーションプログラムは，サブルーチンという部品を使って組み立てるものだと考えればいいでしょう.

サブルーチンに分割しておけば，プログラムのメンテナンスが楽になります．これは，それぞれのルーチンが独立しているので，プログラムを修正したときの影響や，エラーが発生する原因がルーチン内に限定されるからです.

1.2　サブルーチンの宣言と呼び出し

サブルーチンは subroutine 文で開始を宣言し，end subroutine 文で終了します．すなわち，次のような構造にします.

```
      subroutine subr1
        implicit none
        real a,b
        integer i
          .......
          .......
      end subroutine subr1
```

先頭の subroutine 文で，subroutine の後に指定した文字列（この例では subr1）を“サブルーチン名”といいます．また，最後の end subroutine文にもサブルーチン名を指定します．見てわかるように，メインプログラムと同じ構造です．サブルーチン内部のプログラムの書き方も基本的にメインプログラムと同じで，subroutine 文の次に implicit noneを書き，非実行文を上方に集約して，その後に実行文を書きます．サブルーチンはそれ自体で閉じているので，実行文中で使う変数や配列は，基本的にその内部で宣言しなければなりません．ただし，異なるルーチン間で共用可能な変数を別途用意することは可能です．これについては，1.7 節で説明します.

上の例では，subroutine 文にサブルーチン名しかありませんが，これを“引数なしサブルーチン”といいます．これに対して，サブルーチン名の後に，かっこで囲んだ変数リストを付加することができ，これを“引数ありサブルーチン”といいます．以下に例を示します.

```
      subroutine subr2(x,m,y,n)
        implicit none
        real x,y,a,b,z(10)
        integer m,n,i,k
          .......
          .......
      end subroutine subr2
```

リスト中の変数を“引数”といいます．この例では，x,m,y,n が引数です．引数は，サブルーチンとそのサブルーチンを使うルーチンの間で数値を受け渡すのに使います．引数もサブルーチン内部の変数なので，必要な型に応じた宣言をしなければなりません.

サブルーチンを動作させてその機能を使うときは，call 文を使ってそのサブルーチンを指名します．このため，サブルーチンを指名することを，“サブルーチンを呼び出す”とか“コールする”といいます．call 文は以下のような形式です．

```
call サブルーチン名                                    ！引数なしサブルーチン用
call サブルーチン名(数値または変数のリスト)    ！引数ありサブルーチン用
```

例えば，先ほど出てきた，引数なしと引数ありの二つのサブルーチンを使うときは，それぞれ，次の(1)と(2)のようにコールします．

```
  real z
  integer m
  call subr1          .....(1)
  m = 21
  call subr2(10.0,100,z,m*5+1)          ..... (2)
```

前回説明したように，計算式は計算結果の数値を動作命令に与えるので，call 文の引数には，(2)の一番右のように計算式を与えることもできます．

サブルーチンは単独では動作できないので，メインプログラムが別途必要です．例えば，次のようなセットを構成して初めて一つのプログラムが完成します．

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

ここでは，メインプログラムを先に，サブルーチンを後に書きましたが，逆でもかまいません．サブルーチンが複数存在する場合も，ルーチンを記述する順番は実行結果とは無関係です．サブルーチンの中から別のサブルーチンをコールすることも可能です．

上記のプログラム実行の流れを図１に示します．

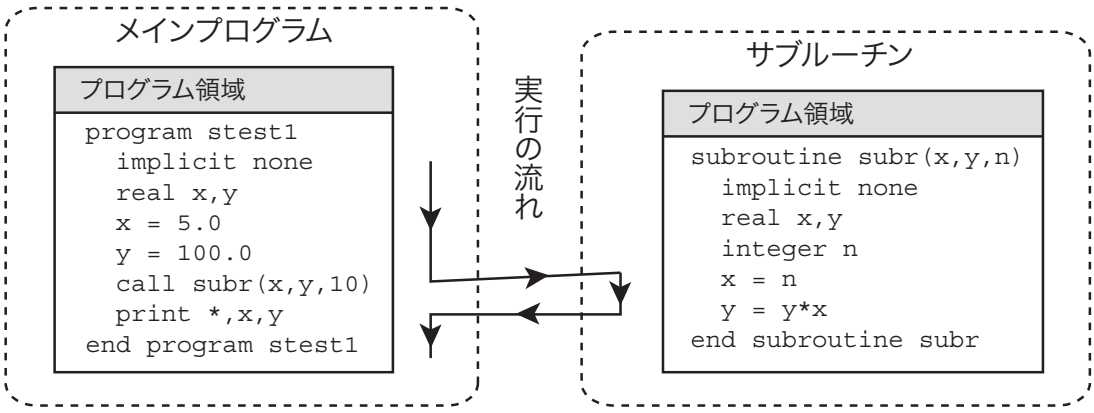

図１．サブルーチンの呼び出しと実行の流れ

図のように，メインプログラムの call 文でサブルーチンを指名すると，サブルーチンの一番最初の実行文に動作が移り，サブルーチン内部の動作が完了すると，コールしたメインプログラムに戻って，その call 文の次の文から実行を継続します.

条件に応じて，途中でサブルーチンの処理を打ち切って戻るときには return 文を用います.

```
subroutine subr(x,y,m,n)
  implicit none
  real x,y
  integer m,n
    .......
  if (m<0) return
    .......
end subroutine subr
```

この例では，m＜0 のときにサブルーチンの処理が終了し，コールしたルーチンに戻ります．ここで return 文の代わりに stop 文を用いれば，プログラム自体が終了します.

1.3　サブルーチン内部の変数と引数

サブルーチン内部で宣言した変数や配列は，メインプログラムや他のサブルーチンから独立しています．すなわち，同じ名前を使っても全く別の変数です．例えば，

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

subroutine subr
  implicit none
  real x,y
  print *,x,y
end subroutine subr
```

と書いても，サブルーチン中の print 文によって出力される x と y はメインプログラムで代入した 10.0 と 30.0 ではなく，全く無関係な数字です．逆に言えば，他のルーチンでの宣言を気にせずに変数や配列の名前を決めることができます．このルーチン内部でのみ有効な変数を“ローカル変数”といいます.

上記のプログラムを期待通り働かせるために，コール側ルーチンの数値をサブルーチンに伝えるのが引数です．例えば，上記のプログラムを以下のように書き直せば，サブルーチン中の print 文で出力される x と y は，メインプログラムと同じ 10.0 と 30.0 になります.

```
program stest2
  implicit none
  real x,y
  x = 10.0
  y = 30.0
  call subr(x,y)
end program stest2

subroutine subr(x,y)
  implicit none
  real x,y
  print *,x,y
end subroutine subr
```

引数は，数値の受け渡しをするための窓口に過ぎないので， call 側の引数と subroutine 側の

引数の変数名を同じにする必要はありません．次のサブルーチンに置きかえても全く同じ動作をします．

```
      subroutine subr(a,b)
        implicit none
        real a,b
        print *,a,b
      end subroutine subr
```

外部からの影響を受けたり，外部に影響を与える，という性質を持つことを除けば，引数もローカル変数です．すなわち，コールしたルーチンからその詳細は見えません．見えるのは，引数という窓口の並びだけです．このため，引数ありサブルーチンを使うときに重要なポイントは，

(1) 引数の数
(2) 対応する引数の型

が call 文と subroutine 文とで一致していなければならないことです．例えば，

```
      program stest3
        implicit none
        real z
        integer n
        z = 200.0
        n = 21
        call subr(10.0,z**2,100,n*5+1)
      end program stest3

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

というプログラムでは，表１のような対応になっています．

表１．サブルーチンの引数対応

call 文	subroutine 文	数値型
10.0	x	実数
z**2	y	実数
100	m	整数
n*5+1	n	整数

ここで注意すべきなのは，第 1 引数の x は実数型なので実定数 10.0 を与え，第 3 引数の m は整数型なので整定数 100 を与えていることです．もし，第 1 引数に同じ意味だろうと思って 10 という整定数を与えると，実行時エラーになります．

サブルーチン内部の実行文で，引数に数値を代入すると，call 文の対応する引数に与えた変数にその数値が代入されます．例えば，

```
      program stest4
        implicit none
        real x,y,p
        x = 10.0
        y = 30.0
        call subr(x+y,20.0,p)
        print *,x,y,p
      end program stest4
```

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

と書くと，サブルーチン中の x はコール側の x+y，すなわち 40.0 であり，サブルーチン中の y はコール側の 20.0 なので，サブルーチン中の z には x*y の計算結果である 800.0 が代入されます．このとき，z が引数なので，call 文の対応する位置にある変数 p に 800.0 が代入され，call 文の次の print 文では x，y，p として，10.0,30.0,800.0 が出力されます．

このようにサブルーチンの引数に数値を代入することで，call 文の引数変数に代入される数値を“戻り値”といいます．戻り値を使えば，サブルーチンの動作で得られた結果をコール側で受け取ることができます．ただし，call 文の引数には定数を与えたり計算式を書いてもよい，と述べましたが，戻り値を指定する引数は別で，必ず変数か配列にしなければなりません．理由を次節で説明します．

1.4　ルーチン間におけるデータの受け渡しと間接アドレス

ルーチン間のつながりをもう少し詳しく考えてみましょう．各ルーチンは基本的にプログラム領域とデータ領域から構成されています．プログラム領域とは文字通りプログラム命令が記録されている領域のことであり，データ領域とは変数や配列のために用意されたメモリ領域のことです．プログラム的には，変数や配列の宣言文がデータ領域の指定を意味しています．

変数や配列が各ルーチンごとに宣言されなければならないのは，データ領域が各ルーチンそれぞれに付属しているからです．このため，図２のように異なるルーチンで同じ名前の変数を宣言しても，それらは別のデータ領域に所属しています．これがローカル変数です．ローカル変数は，そのルーチン内部からしか参照することができません．

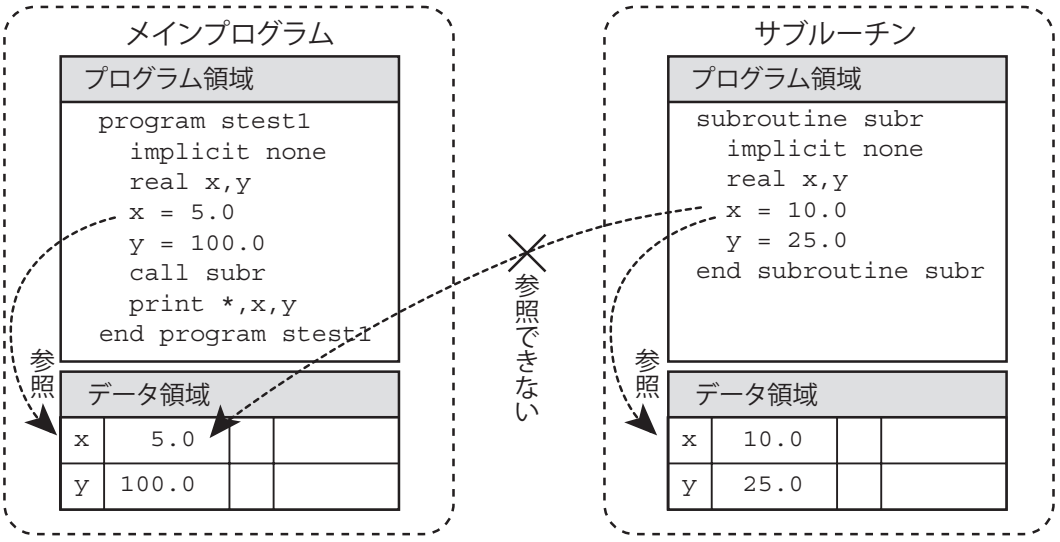

図２．プログラム領域とデータ領域

このため，あるルーチンで計算した数値を別のルーチンで使うには特別な手続きが必要になります．これが引数です．引数を使えば，コール側のルーチンとサブルーチンの間でデータをやりとりすることができます．では，具体的にどうやってデータの受け渡しをしているのでしょう．

データをサブルーチンに渡す手段として，もっとも単純に考えられるのは，引数の数値を直接サブルーチンの変数に代入することです．図３を見て下さい．図のように，コール側の引数 x と y の内容（この例では 5.0 と 100.0）が，サブルーチンの引数として宣言された変数，a と b に代入されています．この方式は，引数の値を直接渡して呼び出す，という意味で“call by value”

と呼ばれています．“call by value”は，コール側からサブルーチン側への値の引渡しについては問題ありません．C 言語はこの方式を採用しています．

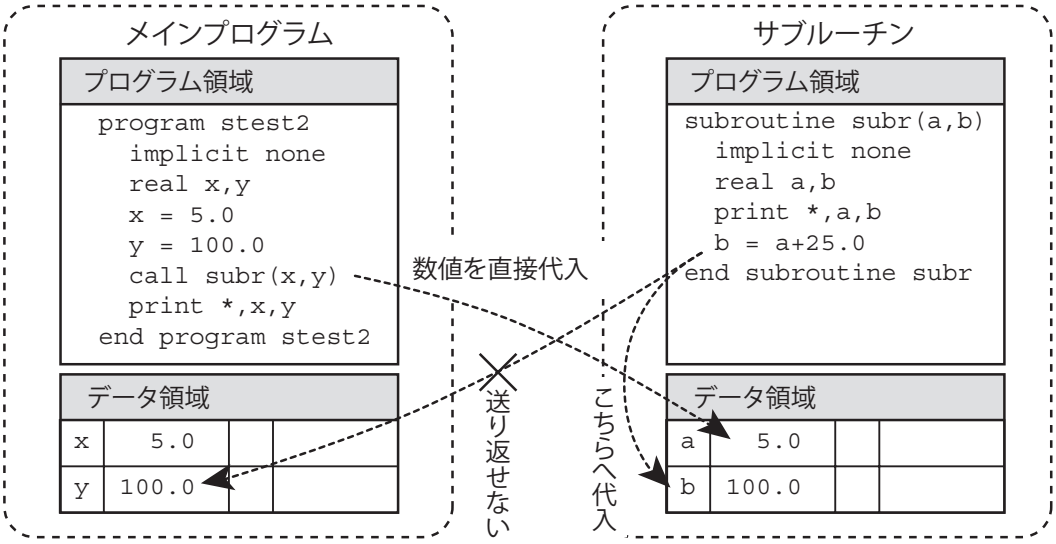

図３．“call by value”を使ったデータ渡しでは，値を送り返せない

しかし，この方式では，サブルーチン側で作成したデータをコール側に戻すことはできません．サブルーチンは，コール側から送られてきた“値”はわかるのですが，それ以上の情報がないので，コール側のどこにデータを代入すればいいかわからないからです．図３のように，サブルーチン中の代入文，b=a+25.0，でサブルーチンの変数 b に 30.0 という値を代入しても，サブルーチンのデータ領域の b に代入されるだけで，メインプログラムのデータ領域には影響がありません．

この問題を解決するために，Fortran は“call by value”ではなく，“call by reference”という方式を採用しています．“call by reference”とは，変数の内容ではなく，変数の存在場所（メモリアドレス）をサブルーチンに伝えて呼び出す方式です．

コンピュータのメモリには，“番地”と呼ばれる通し番号がついています．メモリアドレス（あるいは単にアドレス）とはこの番地のことで，プログラムの実行中に，あるメモリのデータを読み書きするときは，そのメモリのアドレスを指定して行います．例えば，簡単な代入文，

```
a = 1500.0
b = a
```

を考えてみましょう．この結果，b のメモリに 1500.0 という値が書き込まれますが，代入式 b=a において，a（アドレスを[103]とする）という変数からデータを取って来て b に代入するという流れは，

```
a のアドレス指定　→　[103]番地のメモリ内のデータ（1500.0）の呼び出し
　　→　b に代入
```

となります．流れを図に描けば図４のようになります．この方式を“直接アドレス”といいます．

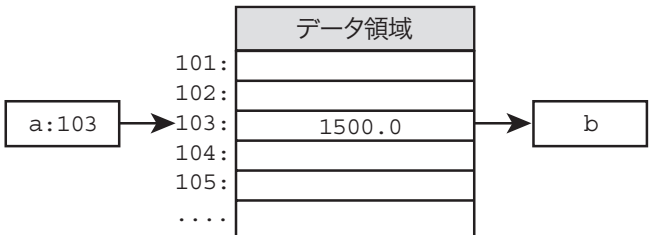

図４．直接アドレスによるメモリ参照

これに対し，あるアドレス（addr1）のメモリに入っているのが別のメモリのアドレス（addr2）で，アドレス addr1 を指定して，アドレス addr2 のメモリに入っているデータを読み書きする方式を“間接アドレス”といいます．Fortran では，“call by reference”でサブルーチンを呼び出すので，引数には call 側ルーチンのメモリアドレスが入っています．このため，サブルーチン内での引数の読み書きは間接アドレスを使って行います[1]．例えば，

```
program stest2
  implicit none
  real a
  a = 1500.0
  call subr(a)
end program stest2

subroutine subr(s)
  implicit none
  real b,s
  b = s
end subroutine subr
```

というプログラムを考えます．サブルーチン subr の中の代入文 b=s によって，変数 b のメモリの値は 1500.0 になりますが，引数変数 s には a の内容である 1500.0 ではなく，メインプログラムの変数 a のアドレスが入っています．このため， b=s という代入文で，変数 s（メモリアドレスを[107]とする）からデータを取ってきて変数 b に代入する流れは，

```
s のアドレス指定　→　[107]番地にあるメモリアドレスデータ（103）の呼び出し
　　→　[103]番地にあるデータ（1500.0）の呼び出し
　　→　b に代入
```

となります．図示すれば図５のようになります．

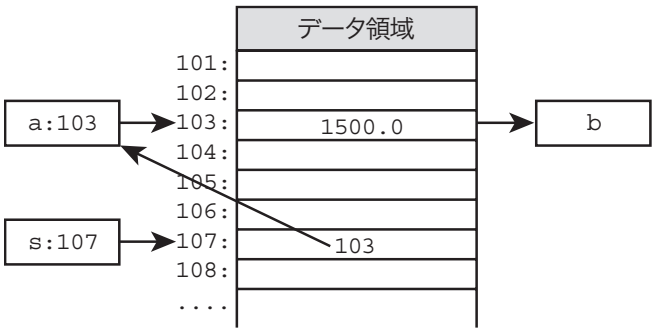

図５．間接アドレスによるメモリ参照

間接アドレスによる読み書きは直接アドレスよりも時間がかかります．よって，b=s の代入文において，代入する値を保持している右辺の変数 s が間接アドレス指定の場合にはメリットがありません．しかし，代入される左辺の変数 b が間接アドレス指定の場合には，この仕組みを利用することで，コール側ルーチンの変数にサブルーチン側のデータを代入することができます．

例えば，図６のようなプログラムを考えます．サブルーチンの引数変数 b に a+25.0 の結果である 30.0 を代入すると，b にはメインプログラムの変数 y のアドレスが入っているので，間接アドレス指定により y に 30.0 が代入されます．

戻り値を利用するときは，対応する引数にコール側変数のアドレスを与えなければなりません．これが変数または配列を指定しなければならない理由です．戻り値指定の引数に定数や計算式を与えると実行時エラーになります．

---

[1] 配列を実現するのにも間接アドレスが使われています．例えば，a(10)は a(1)から数えて 10 番目のメモリです．よって，配列の先頭要素 a(1)のアドレスをメモリに記録し，それに 9（=10-1）を加えて間接アドレスを用いれば，a(10)のメモリを読み書きすることになります．

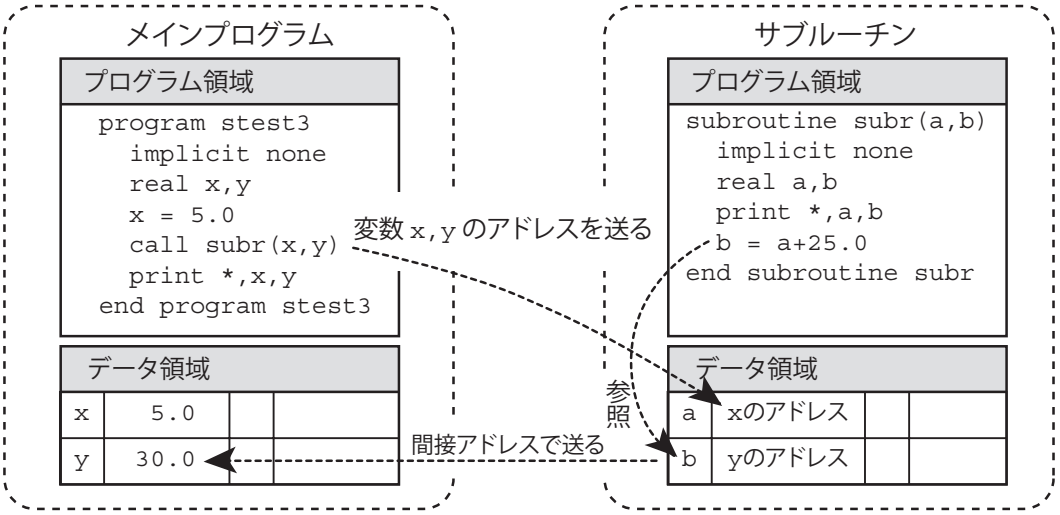

図６．Fortran では“call by reference”により，データを送り返すことができる

1.5　配列を引数にする場合

配列を call 文の引数にするときは，配列名を引数にすることも，配列要素を引数にすることも可能です．配列名は配列の先頭要素アドレスを示すので，配列名を引数にすることと，その配列の第１要素を引数にすることは同じ意味になります．例えば，real a(10)と宣言した１次元配列に対して，

```
call sub(a)　と　call sub(a(1))
```

は同じ動作をします．

これを受けるサブルーチンでの宣言は，サブルーチン内部でそのデータをどう使うかで決めます．一例を示します．

```
subroutine sub(x)
  implicit none
  real x
  x = 10.0
end subroutine sub
```

この例では引数 x が単一変数として扱われています．このため，call 文で指定した a(1)だけがサブルーチン内部で利用でき，他の要素は使えません．これは，call sub(a)のように，配列名を与えた場合でも同じです．

これに対し，

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

のように引数 x を配列宣言すれば，あたかもコール側ルーチンの a という配列がサブルーチン内部の配列 x になったかのように取り扱うことができます．ここで“あたかも”という言葉を使ったのは，サブルーチンでの宣言文の書き方によってはそうならないことがあるからです．例えば，サブルーチンでの配列宣言 x(10)の要素数 10 はコール側と同じ数にする必要はありません．100 のように大きくても，1 のように小さくてもかまわないのです．これは，引数 x が配列の先頭要

素アドレスを保持している１個の変数にすぎないからです. サブルーチンでの引数配列の宣言は, 配列の形状（次元や下限値）を表示するためのダミーにすぎません.

このため，コール側ルーチンで宣言した要素数とサブルーチンで宣言した要素数を一致させる必要はありません[2]. 与えられた配列をどう宣言するかは，サブルーチン内部の計算の都合に合わせて, サブルーチン側で決めることができます. よって, 汎用性のあるサブルーチンにするには, 配列のアドレス以外に，配列の要素数も引数にしてサブルーチンに伝える必要があります.

配列の形状さえ分かればいいのですから，サブルーチンで引数配列を宣言するときは，数字の代わりに“*”を書くことができます．例として，要素数 n の１次元配列 a のデータを，同じ長さの１次元配列 b にコピーするプログラムを考えてみましょう．最も単純な答えは，以下のようなものです.

```
      subroutine copy(a,b,n)
        implicit none
        real a(*),b(*)
        integer n,i
        do i = 1, n
          b(i) = a(i)
        enddo
      end subroutine copy
```

このように，サブルーチンの引数配列 a や b の宣言を“*”にすることで，要素数が不定の１次元配列であることを明示しています.

“*”による宣言は２次元以上の配列でも使うことができますが，制限があります．例えば，a(*,*)という形の宣言はできません．なぜなら，２番目の要素を進めるのは１番目の要素が最大値，すなわち宣言した上限数に達したときなので，１番目の要素が不定では情報不足で進められないからです．このため，“*”が使えるのは，一番右側の要素数のみです．例えば，real a(3,*)と宣言すると，第１要素数が３の２次元配列として計算されます.

しかし, これではどんな２次元配列でも使えるサブルーチンを作ることはできません. そこで, “整合配列”と呼ばれる機能が用意されています. 整合配列とは subroutine 文の引数の中にある整数型変数を利用して宣言した形の引数配列のことです．例えば，以下のように宣言することができます.

```
      subroutine copy2d(a,b,m,n)
        implicit none
        real a(m,n),b(m,n)
        integer m,n,i,j
        do j = 1, n
          do i = 1, m
            b(i,j) = a(i,j)
          enddo
        enddo
      end subroutine copy2d
```

この例では，m と n が引数の中にあるので，これらを使って引数の配列 a と b を宣言することができるのです．a(m,n)のような単純な宣言だけではなく，a(2*m*n,n-1)のように，計算式を使って宣言することも可能です.

このサブルーチンの使用例を次に示します. サブルーチンの引数 a, b にどんな要素数の２次元配列を与えても，コール側ルーチンにおける配列宣言と同じ要素数を m, n に与えれば，正しく動

[2] 要素数だけではなく，次元さえ一致させる必要はありません．コール側が２次元配列で，サブルーチンでの宣言が１次元配列でも動作は可能です．例えば，上記のサブルーチン copy は，２次元配列でも使用できます．これは，２次元配列もメモリ上では１次元的に並んでいるからです．ただし，要素数を与える引数 n には全要素数を指定する必要があります．例えば，a(10,10)と宣言された配列ならば，n=100（=10×10）を与えます.

作します.

```
program stest3
  implicit none
  real a(10,20),b(10,20),c(100,200),d(100,200)
    .......
  call copy2d(a,b,10,20)
  call copy2d(c,d,100,200)
end program stest3
```

なお，配列宣言の際には，コロンを付加して下限を指定することができますが，これをサブルーチンに引き継ぐにはサブルーチン側でも同じ下限指定をする必要があります．例えば，

```
program stest4
  implicit none
  real a(-1:100)
  call sub(a,a,100)
end program stest4

subroutine sub(x,y,n)
  implicit none
  real x(-1:n),y(*)
  integer n
  x(10) = 10.0
  y(10) = 10.0
end subroutine sub
```

と書いた場合，メインプログラムと同じ下限指定をした配列 x では x(10)がメインプログラムの a(10)に対応するので，a(10)に 10.0 が代入されます．しかし，下限指定をしていない配列 y では下限が 1 ですから，a(-1)から数えて 10 番目の a(8)に 10.0 が代入されます．任意の下限を持つ配列に対して動作するプログラムにするには，下限の数値を引数にして整合配列を使います．

1.6　関数副プログラムと色々な組み込み関数

sin(x)のように，引数を使って内部で計算した結果を計算式中で使うことができる“関数”を自作することもできます．これを関数副プログラムといいます．関数副プログラムは，サブルーチンとほとんど同じ構造をしていますが，以下のような違いがあります．

(1) subroutine の代わりに function を書く
(2) 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

これ以外は，引数ありサブルーチンと同じです．引数に関する注意もサブルーチンと同じで，戻り値を与える引数を含めることもできます．関数副プログラムとは，引数以外に戻り値を１個持つサブルーチンの変種だといえます．この戻り値が関数値になります．

関数副プログラムの一例を示します．

```
function square(x)
  implicit none
  real square,x
  square = x*x
end function square
```

このように，function 文で開始し，end function 文で終了します．また，関数名 square を実数型宣言し，引数 x の二乗を代入して終了しています．すなわち，この例は，引数 x に実数型の数値を与えると，x^2 を関数値として与える関数になります．

関数副プログラムを使うことは,“関数名”という変数を使うことであると考えます.このため，作成した関数を使用するルーチンでも関数名を型宣言する必要があります．例えば，上例の square を使うには，

```
program ftest1
   implicit none
   real x,y,square
   x = 5.2
   y = 3.0*square(x+1.0) + 50.5
   print *,x,y
end program ftest1
```

のように，square という関数名を関数副プログラムでの宣言と同じ型で宣言しておかなければなりません．関数副プログラム中で宣言した関数名の型と，その関数を使用するルーチンで宣言した関数名の型が異なる場合には実行時エラーになります．

さて，前回紹介したように，sin や sqrt のような数値計算でよく使う関数はあらかじめ用意されています．これらを“組み込み関数”といいます．組み込み関数は，コンパイラが型を認識しているので，使用に際しての型宣言は不要です．表２に，前回示した関数以外で，よく使うと思われる組み込み関数をいくつか示します．

表２．型変換関数などの組み込み関数

組み込み関数	名　称	引数の数値型	関数値の数値型	関数の意味
real(n)	実数化	整数	実数	実数型へ変換する
int(x)	整数化	実数	整数	整数型に変換する
real(z)	複素数の実部	複素数	実数	z の実部
imag(z)	複素数の虚部	複素数	実数	z の虚部
cmplx(x,y)	複素数化	２個の実数	複素数	x+iy
conjg(z)	共役複素数	複素数	複素数	z の共役複素数
abs(z)	絶対値	複素数	実数	z の絶対値
mod(m,n)	剰余	２個の整数	整数	m を n で割った余り

型変換関数は，サブルーチンの引数に，数値型の異なる変数を与えるときに使います．また，整数化は切り捨て演算を含んでいるので，それを積極的に利用するときにも使います．例えば，正の実数 x の小数点以下を四捨五入した整数は，int(x+0.5)で計算することができます．

最後の剰余を計算する関数 mod は意外によく使います．do 文で繰り返し計算をする場合，“n 回毎に出力する”というように，一定間隔で動作を変えたいことがよくありますが，ここで mod が使えます．例えば，繰り返し計算の中で 10 回に 1 回出力するなら，

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

のように書きます．プログラムというのは，基本的に全て計算であり，流れのコントロールも計算結果を使って行わなければなりません．このようなケースで mod を使うのは，パターンの一つとして覚えておくと良いでしょう．

1.7　モジュールを使ったグローバル変数の利用

ローカル変数のおかげで，各ルーチンの独立性は保たれますが，引数でしかルーチン間のデータ受け渡しができないのは不便です．引数は，かっこを使った並びで指定するので，受け渡す変数が多いと書くのが大変です．しかも並びが重要なので，一カ所順番を間違えただけでもエラーが発生します．そもそも引数による受け渡しはアドレス情報を経由したものなので，コール側とサブルーチン側で完全に同じ変数であるという保証はありません．

計算機シミュレーションでは，内容に応じてサブルーチンを作成すると述べましたが，この用途においては，ルーチン間で共用する変数が多数必要です．しかし，共用変数を全て引数にするのは効率が悪く，エラーも発生しやすくなります．

そこで，ルーチン内部でのみ意味を持つ“ローカル変数”に対して，“グローバル変数”が用意されています．グローバル変数とは，どのルーチンから参照しても共通した値を保持している変数のことで，Fortran では，モジュールと use 文の組み合わせで利用可能です[3]．

モジュールとは，変数やサブルーチンなどを集めて一つのパッケージにしたもので，module 文で開始して end module文で終了します．サブルーチンと同じ構造をしていることからわかるように，モジュールの記述は，メインプログラムやサブルーチンと同レベルです．例えば，

```
module data1
   integer nmin,nmax
   real tinitial,amatrix(20,30)
end module data1
```

のように書きます．先頭の module 文で，module の後に指定した文字列（この例では data1）を“モジュール名”といいます．最後の end module文にもモジュール名を指定します．サブルーチンや関数副プログラムと異なり，モジュールはプログラムに記述しただけでは利用することができません．利用するルーチンの先頭に，use 文を使ってモジュール名を指定し，その利用を宣言する必要があります．use 文は以下のような形式です．

```
use モジュール名
```

use 文は implicit 文よりも前に書かなければなりません．モジュールの使用例を以下に示します．

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

program stest5
   use global
   implicit none
   xaxis = 5.0
   yaxis = 100.0
   call subr
   print *,xaxis,yaxis
end program stest5

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

モジュールの中で宣言された変数や配列は，メインプログラムやサブルーチンとは独立して存在したデータ領域にあり，use 文で利用を宣言すれば，どのルーチンからでも参照することができます．このプログラムのメモリイメージを図示すれば，図７のようになります．

モジュールは，名前が異なれば，一つのプログラムに複数作成することも可能であり，一つのルーチンが複数のモジュールを利用することも可能です．ただし，モジュールは use 文で利用を宣言するより前で（プログラムの上方で）定義されている必要があります．このため，通常はこの例のように全てのルーチンより前に記述しておきます．

---

[3] モジュールや use 文は Fortran90 から採用された比較的新しい仕様です．Fortran77 以前では common 文でグローバル変数を実現していました．しかし，common 文には不便な点が多いので本稿では省略します．

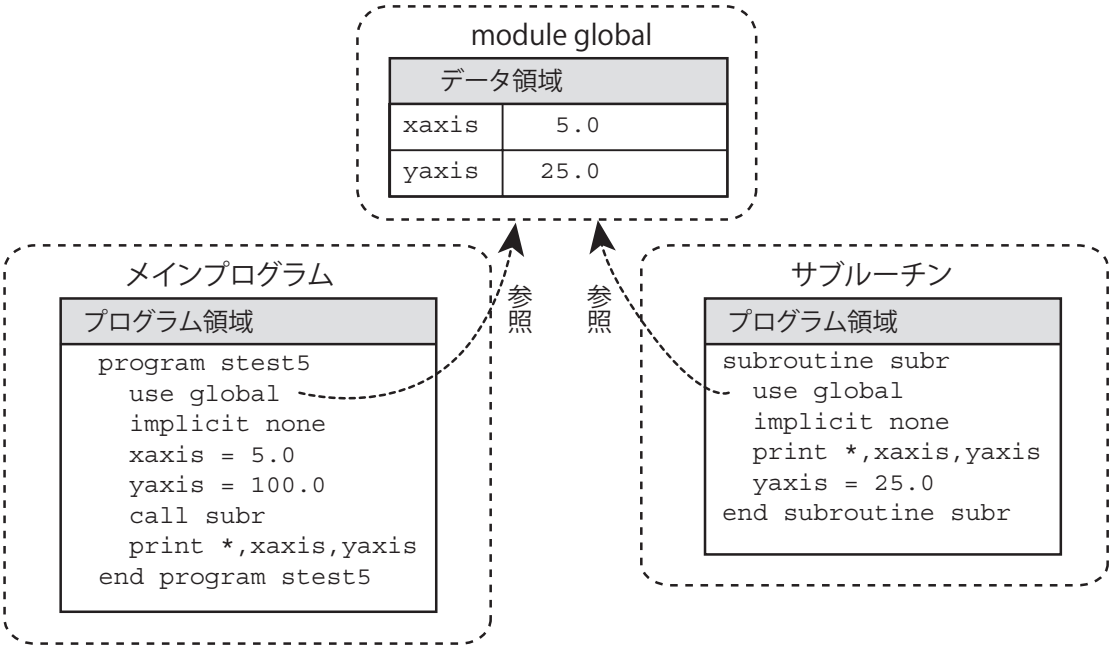

図７．モジュールで宣言されているグローバル変数の参照

モジュールで宣言されている変数は，use 文を書くだけで利用できるという利便性がありますが，ルーチン内で明示的に宣言されていないので，不用意に使ってしまう可能性があります．そこで，次のような only 句を使って，ルーチン内で必要な変数だけを明示することができます．

```
use モジュール名, only : 変数１, 変数２, ...
```

例えば，前の例で，

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

のように書くと，yaxsis 以外の変数（xaxsis）は，このルーチンでは宣言されていないのと同等です．宣言されていない変数は，ローカル変数として別途宣言することも可能です．

グローバル変数は，多用するとルーチンの独立性が失われてしまいます．基本的にはローカル変数でプログラムを作り，ルーチン間で共用する必要がある変数のみ，グローバルにしてください．また，グローバル変数は広い範囲で存在する可能性があるので，できるだけ意味のある長めの名前を付けます．これは，前回出てきた“大域的に有効な変数名の付け方”です．

2.　データ出力の詳細とデータ入力

2.1　データ出力先の指定

これまでもたびたび登場しましたが，もっとも単純な出力命令は，print 文です．

```
print form, データ１, データ２　...
```

print 文で出力すると，画面（正確には標準出力）にデータが表示されます．*form* は出力形式を指定するためのものですが，とりあえず“*”を書いておけば，データの数値型に応じた標準形式で出力します．例えば，

```
print *, x, y(i), x**2+5, 'abc = ', 10
```

のように，データの位置には，変数や配列を与えても良いし，計算式や定数を与えることも可能です．また，文字列を適当に入れて，データの意味を表示することもできます．

画面ではなく，ファイルに出力するときは write 文を用います．print 文との違いは，出力先を指定する整数値 *nd* と出力形式指定の *form* をかっこで記述することです．

```
write(nd,form) データ１，データ２　...
```

nd を装置番号といいます．装置番号は，出力ファイルを識別する数字で，任意に選ぶことができます．ただし，原則として *nd*≧10 にして下さい．これは，Fortran コンパイラの仕様によって，一桁の数値は予約されている可能性があるからです．例えば，*nd*=6 にすれば print 文と同じ意味になって，画面に表示されます．*nd* には，変数や，整数値になる計算式を与えることもできるので，条件に応じて出力先を変更することも可能です．

nd に適当な整数を与えて write 文を実行すると，“fort.*nd*”という名のファイルに出力されます[4]．例えば，

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

と書けば，x,y,z の値が“fort.20”という名前のファイルに出力されます．print 文と同じく，*form* に“*”を与えれば，標準形式で出力します．

2.2　配列の出力，do 型出力

出力文において，データの位置に配列名を書けば，全要素が並んで出力されます．例えば，

```
real a(4)
do i = 1, 4
   a(i) = i
enddo
print *,a
```

というプログラムを実行すると，

```
 1.00000000000000   2.00000000000000   3.00000000000000   4.00000000000000
```

のように，配列要素が先頭から順に並んで出力されます．このとき，配列要素が多いと適当に改行を入れて出力されます．よって，この方法は配列要素が少ないとき以外はあまりお勧めできません．例えば，２次元配列を，

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

のように出力すると，a(1,1),a(2,1),a(3,1),a(1,2),...,a(3,3) という並びで 9 個の要素が出力されます．そこで，次のように do 文で出力する方がいいでしょう．

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

しかし，print 文や write 文は，１文ごとに改行をするので，複数の print 文を使って，横に続けて書くことはできません．よって，この例の出力では全部で 9 行必要です．行数を減らすために，1 行に行列の 1 行を出力するには，次のように書かなければなりません．

```
do i = 1, 3
   print *,a(i,1),a(i,2),a(i,3)
enddo
```

---

[4] “fort”の部分はコンパイラに依存しますが，多くは fort のようです．また，コンパイラによっては環境変数で指示することにより，必要な装置番号に任意の名前のファイルを割り当てることも可能です．

しかし，この例のように列の数がわかっているときは良いのですが，列数が多いときや変数で指定したいときには不便です．そこで do 型出力が用意されています．do 型出力とは，

```
（データ１,データ２,...,整数型変数=初期値,終了値）
```

のような，かっこで囲んだ形式です．これをデータの位置に記述すれば，do 文のように，整数型変数が初期値から終了値まで１づつ増えていき，その変数で指定されるデータが順に出力されます．do 型出力を複数並べたり，通常のデータと並べて書くこともできます．例えば，

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

は，さきほどの“print *,a(i,1),a(i,2),a(i,3)”と書くことと同等です．また，

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

は，“print *,5*x,a(1),b(1),a(2),b(2),a(3),b(3)”と書くことと同等です．

do 型出力は多重にすることもできます．例えば，

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

は，“print *,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)”と書くことと同等です．この例のように，多重のときには内側の繰り返し変数が先に進みます．

なお，do 型出力の整数型変数は，do 文のカウンタ変数に相当します．このため，do 型出力の入っている出力文を do ブロックの中に入れるときには，カウンタ変数が重複しないようにしなければなりません．

2.3　format による出力形式の指定

標準出力形式（“*”指定）で実数を画面に出力すると，有効数字 15 桁の数字を使って表示されます．このため，あまり多くのデータを横に並べることができないし，結果を見るだけなら，それほどの有効数字は必要ありません．また，標準形式の出力には 1 行の出力文字数に制限があるため，出力数が制限を超えると自動的に改行されてしまいます．このため，同じ出力を繰り返すと，行ごとに小数点の位置が違うこともあり，大量に出力するのには向きません．

これらの問題は，出力形式（format）を指定することで解決できます．これまで“*”を書いていた *form* の位置に format を指定すれば，小数点以下の桁数を小さくしたり，必要に応じて数字と数字の間にスペースを空けたり，改行を入れたりすることができます．また，自動改行されることがないので，幅広く出力することも可能です．

format の指定方法は２通りあります．まず，format 文による指定です．一例を示します．

```
      real x,y
      integer n
      x = 1.5
      y = 0.03
      n = 100
      print 600,x,y,n
  600 format('  x = ',f10.5,'    y = ',es12.5,'    n = ',i10)
```

最後の文が format 文です．format 文は，サブルーチンの引数のように，出力形式の指定をリストにしてかっこで囲み，先頭に文番号を付けます．この文番号を print 文や write 文の *form* に指定すれば，その format にしたがって出力データが整形されます．この例では，600 が *form* 指定です．文番号は重複できないので，ルーチン内では format ごとに異なる数字をつけなければなりません[5]．

複数の print 文や write 文が同じ format 文を指定するのは可能です．例えば，次のように共用することができます．

---

[5] 文番号の付け方に決まったルールはありませんが，“format 用である”という意味をどこかに入れておいた方が良いでしょう．この例で 600 を使っているのは，“6”が昔の出力装置番号だったころの習慣です．

```
      real x,y,u,v
      integer n,k
       ......
      print 600,x,y,n
      write(20,600) u,v,k
  600 format('  x = ',f10.5,'   y = ',es12.5,'   n = ',i10)
```

format を指定するもう一つの方法は，文字列を使って *form* の位置に直接 format の内容を記述することです．例えば，上記の format を print 文や write 文に埋め込んで，

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

のように書くことができます．このとき，“format”の文字は不要ですが，両端のかっこは必要です．なお，Fortran での文字列は「'」で囲むのが基本ですが，「"」も使えるので，format 内部に「'」が入っているときには「"」で囲みます．

format を出力文中に書き込む方法は，文番号を必要としない点は良いのですが，出力文が長くなるし，同じ format を何度も使うときには不便です．そこで，文字列変数を利用して書く方法があります．文字列変数の利用については次回説明します．

出力形式の指定方法を上記の format を例にして説明しましょう．まず，format 中の文字列はそのまま出力されるので，必要に応じて適宜挿入します．この例では，' x ='や' y ='は，スペースも含めてそのまま出力されます．

次に，出力文中のデータ値の並びに対し，それぞれの出力形式を指定する“編集記述子”を選んで，前から順に記述します．この例では，f10.5，es12.5，i10，が編集記述子で，print 文の並びに対し，

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

という対応で出力形式を指定しています．文字列と，編集記述子で指定したデータ値は，その並び順に出力されます．よって，この print 文を実行したときの出力は以下のようになります．

```
      x =    1.50000    y =  3.00000e-02    n =         100
```

出力形式を指定する編集記述子の主要なものを表３に示します．データの数値型に応じた編集記述子を使用しないと正しい値が出力されないので注意して下さい．なお，この表で斜体文字（w, m, d, r）は整定数で指定します．

表３．主要な編集記述子

編集記述子	数値型	編集の意味
Iw	整数	幅 w で整数を出力する
I$w.m$	整数	幅 w で整数を出力する 出力整数の桁が m より小さいときには，先頭に 0 を補う（$w \geqq m$）
F$w.d$	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（$w \geqq d$+3）
E$w.d$	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（$w \geqq d$+8） 仮数部の１桁目は 0 になる
G$w.d$	実数	幅 w で実数を出力する 指数部の大きさに応じて F 編集と E 編集を切り替える

表３（続き）．主要な編集記述子

編集記述子	数値型	編集の意味
ES$w.d$	実数	幅 w で実数を浮動小数点形式で出力する 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
EN$w.d$	実数	幅 w で実数を浮動小数点形式で出力する 0 以外の数値を出力すると，仮数部の絶対値は 1 以上 1000 未満になり， 指数部は 3 で割り切れる数になる
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
/	なし	改行する
rX	なし	r 個スペースを挿入する

最後の二つ（改行とスペースの挿入）は，出力動作を記述するものなので，文字列と同様に，出力文中のデータとの対応はありません．また，ここでは編集指定の文字（F や ES など）を指定数（w など）と区別するために大文字で書きましたが，小文字でも同じ意味です．例えば，i10 は整数型値を幅 10 文字で出力することを意味し，f10.5 は実数型値を幅 10 文字，小数点以下 5 桁で出力することを意味しています．このため，

```
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
```

というプログラムの出力は，

```
    1.50000   0.03000       100     00010
+----+----+----+----+----+----+----+----+
```

となります．２行目の目盛りは位置を確認するために書いたものですが，10 文字の中に，右寄りで出力されているのがわかります．なお，出力文字数が指定の幅 w を越えると，「*****」のように「*」が w 個出力されます．

E 編集を使って実数を浮動小数点形式で出力すると，小数点の前が 0 になります．例えば，

```
real x,y
x = 1.5
y = 3.14e10
print "(e15.5,e15.5)",x,y
```

の出力結果は，

```
    0.15000e+01     0.31400e+11
```

となります．これでは感覚的にわかりにくいし，表示字数が 1 個無駄になるので，ES 編集や EN 編集を使う方が良いでしょう．例えば，

```
real x
x = 3.14e10
print "(es15.5,en15.5)",x,x
```

の出力結果は，

```
    3.14000e+10    31.40000e+09
```

となります．

各編集記述子と出力の数値は１対１対応にしなければならないので，配列を出力するときには出力要素数だけ編集記述子を書かなければなりません．このとき，同じ編集記述子を繰り返すならば，編集記述子の前に整数 *r* を付加して，*r* 回反復するという指定ができます．例えば，“3f10.5”は“f10.5,f10.5,f10.5”と書くことと同等です．

さらに，実数，整数，実数，整数のような繰り返しのときには，かっこでくくって反復指定をすることができます．例えば，次のように書くことができます．

```
real x,y
integer m,n
x = 1.5
y = 0.03
m = 5
n = 100
print ″(2(f10.5,3x,i10))″,x,m,y,n
```

この print 文の format は，“f10.5,3x,i10,f10.5,3x,i10”と書くことと同等です．

なお，format 中の編集記述子の数よりも出力文のデータ値の方が多い場合には，編集記述子の数だけ出力した後で改行し，同じ format を再度使って残りのデータ値を出力します．文字列が入っていれば，文字列も再度出力されます．

逆に，format 中の編集記述子の数よりも出力文のデータ値の方が少ない場合には，指定したデータ値を出力した段階で終了し，あまった編集記述子は無視されます．無視された記述子以降は文字列等が入っていても全て無視されます．配列を出力するときなどは，反復指定に大きめの数値を与えておくことができます．

2.4　書式なし write 文

write 文は，次のように *form* を省略して，装置番号のみの指定で出力することができます．これを“書式なし write 文”といいます．

```
write(nd) データ１，データ２　...
```

書式なし write 文で出力すると，データ形式が“バイナリ形式”になります．これに対し，これまで説明した *form* のある出力文で出力したときのデータ形式は“テキスト形式”です．元来，計算機内部の数値は 2 進数データであり，“10”とか，“1.000e20”とかの文字ではありません．このため，画面に表示するときには“2 進数→10 進数文字表現”というデータ変換を行って表示しています．この表示形式がテキスト形式です．

しかし，2 進→10 進変換には時間がかかる上に，文字に変換することでデータ量が増えます．これは，文字データが 1 文字あたり 1byte 必要だからです．例えば，実数を有効数字 15 桁で文字表示するには 15byte 以上必要です．しかし，倍精度実数の 2 進数表現は 8byte ですから，そのままで保存すれば 8 文字分で済みます．この内部の 2 進数データがバイナリ形式です．大量のデータをディスクに保存するときには，バイナリ形式による保存が有効です．

ただし，Fortran の書式なし write 文を使って出力したバイナリ形式ファイルは，2.5.1 節で説明する書式なし read 文で読まなければならないので，あくまでも Fortran プログラムから入力して利用するのが基本です．他の言語で作成したプログラムで使う場合や，データ解析ソフトを使って解析するためのファイルを作成するときは，テキスト形式で出力した方が良いと思います．

2.5　データの入力方法

プログラムが実行しているとき，そのプログラム中の指定した変数に外部からデータを代入することを“入力する”といいます．計算条件を設定するための変数にデータを入力できるようにしておけば，プログラムの実行を開始してから条件を設定して，それに応じた計算をさせることができます．これにより，プログラムはその動作だけを利用する“ブラックボックス”になり，コンパイルしたプログラムを“アプリケーション”として他の人に提供することも可能です．

2.5.1　入力文の一般型

データ入力には read 文を用います.

```
read(nd,*) 変数１，変数２　...
```

nd は装置番号で,*nd* に適当な整数を与えると“fort.*nd*”という名のファイルから入力します[6]. 装置番号に関する条件や注意事項は出力の場合と同じで，原則として *nd*≧10 にして下さい．出力ファイルと違うのは fort.*nd* という名のファイルが存在していなければエラーになるので，あらかじめ用意しておかなければならないことです.なお,“*”の位置には,write 文のように format による書式指定ができますが，あまり使うことがないので説明は省略します.

例えば，fort.30 という名前のファイルに,

```
5.2　　1.5　　3
```

と書いて保存しておき，プログラム中に,

```
read(30,*) x,y,z
```

と書けば，この read 文実行後，x=5.2，y=1.5，z=3.0 となって実行が継続します．入力ファイルに改行が入っていても，read 文の変数入力が完了するまで読み込みを続けるので，fort.30 の入力数値は次のように３行に分けて書くこともできます.

```
5.2
1.5
3
```

read 文実行時に，ファイルが存在しなかったり，データが足らない場合には，エラーになってプログラムが強制終了します．逆に，ファイル中の数値が多い場合には，あまった数値は無視されます.

read 文の変数の位置には，配列名や，do 型出力と同型の do 型入力を書くこともできます．これらは，出力と入力という方向が異なりますが，入力要素数や繰り返しの意味は同じです.

ファイルではなく，キーボード（正確には標準入力）を使って入力したいときには，以下のように書きます.

```
read *, 変数１，変数２　...
```

この文を実行すると，プログラムの実行が一時停止し，キーボードからの数値入力を待つ状態になります．そこで，適切な数値をキーボードから入力すると，その数値を所定の変数に代入した後，実行が再開します．このため，read 文のタイミングを考慮して数値を入力しなければ，いつまでたっても停止したままです.

read 文でも装置番号のみを指定することができます.これを“書式なし read 文”といいます.

```
read(nd) 変数１，変数２　...
```

書式なし read 文で入力する場合は，入力データが“バイナリ形式”だと仮定されるので，書式なし write 文で作ったファイルを使う必要があります．また，出力したときの数値と同じ数値型の変数を同じ順番で並べる必要があります．例えば,

```
real x,y
integer n
x = 10.0
y = 100.0
n = 10
write(20) x,n,y
```

[6] 出力ファイルと同様，fort の部分はコンパイラに依存します．環境変数でファイル名が変更できるコンパイラもあります．ちなみに，標準出力の装置番号は 6 でしたが，標準入力の装置番号は 5 です.

というプログラムで作成された fort.20 というファイルから入力するには，

```
      real x,y
      integer n
      read(20) x,n,y
```

のように書かなければなりません．この場合，x も n も同じ 10 だからと思って，

```
      read(20) n,x,y
```

と入力すると，n と x には，正しい値が代入されません．

書式なし write 文１行で書き込まれたデータ数よりも入力データが少ないのは問題ありません．このとき入力しなかったデータは読み飛ばしたことに相当します．逆に言えば，２回の read 文を使って分けて入力することはできません．例えば，上記のデータを，

```
      read(20) x,n
      read(20) y
```

のように２行の read 文に分けて入力しようとした場合，x と n には正常な値が代入されますが，２行目の read 文実行時に，入力データがないというエラーで強制終了します．

2.5.2　入力時のエラー処理

ファイルからデータを入力するとき，要求したファイルが存在しなかったり，書き込まれたデータ数より多くのデータを入力しようとすれば，実行時エラーになって，プログラムは強制終了します．これを防ぐため，read 文中にエラー処理指定を入れることができます．

```
      read(nd,*,err=num) 変数１，変数２  ...    ！テキスト形式入力
      read(nd,err=num) 変数１，変数２  ...      ！バイナリ形式入力
```

ここで，*num* には文番号を与えます．この read 文を実行したとき，入力エラーが起こると *num* で指定した文番号の行へジャンプします．例えば，

```
      do k = 1, 100
        read(10,*,err=999) x,y,z
        .......
      enddo
  999  x = 100
```

と書けば，エラーが起こると文番号 999 の行にジャンプして，その行から処理を継続します．

もし“ファイルの終了”，すなわち，データを入力するときに，それ以上入っていなかった，という場合を検知するだけなら，err=*num* の代わりに end=*num* と書くこともできます．データの出力回数が不明のときには，err か end 指定を入れておき，データ終了時点で次の処理に進むようなプログラムにしておくと良いでしょう．

2.5.3　namelist を用いた入力

便利なテキスト形式入力として，namelist を用いる方法があります．例えば，

```
      read(10,*) x,y,n
```

という入力文では，入力ファイル fort.10 を，

```
      10.0    1.e10    100
```

のように作成しますが，作成するためには入力変数の対応を常に憶えておかなければなりません．必要なデータを全部書き込まなければならないし，順番を間違えることもできません．

これに対し，namelist 入力文では，入力データを“変数＝データ”という代入形で記述するので，どの変数に代入するかを入力ファイルの中で明示することができます．

namelist 入力文を使うときは，まず入力する可能性のある変数や配列名を namelist 文で登録します．namelist 文は次のような形式です．

```
namelist /ネームリスト名/ 変数１，変数２　...
```

namelist 文は非実行文なので，全ての実行文より前に書かなければなりません．また，変数や配列名の登録だけなので，型宣言は別途必要です．例えば，

```
real x,y,a(10)
integer n
namelist /option/ x,y,n,a
```

と書きます．ローカル変数だけではなく，use 文で指定されたグローバル変数も登録可能です．

この namelist に登録された変数に対し，入力文は，

```
read(nd,ネームリスト名)
```

だけです．変数の指定は必要ありません．必要ならば，文番号 *num* を使って，

```
read(nd,ネームリスト名,err=num)
```

のように，err=*num* や end=*num* を追加することで，エラー発生や終了時の処理をすることも可能です．

namelist 入力に対する入力ファイルは次の形式で用意します．変数の順番は任意です．

```
&ネームリスト名
  変数１＝データ１，変数２＝データ２，...
/
```

“&ネームリスト名”から“/”までが namelist 入力文１回で入力されるデータです．例えば上例のようにネームリスト名が option のときには，

```
&option
  x=10.0, y=1.e10, n=100
/
```

のようにファイルに書いておきます．次のように１行づつ書くこともできます．

```
&option
  x=10.0
  y=1.e10
  n=100
/
```

namelist 入力にはもう一つ利点があります．それは，必ずしも登録された変数全部を入力ファイルに記述する必要がないことです．記述しなかった変数には，namelist 入力文の実行前までに代入されていた値がそのまま残ります．このため，あらかじめ全ての登録変数にデフォルト値を代入しておけば，変更したい変数だけ入力ファイルに記述することができます．例えば，

```
real x,y,a(10)
integer n,i
namelist /option/ x,y,n,a
x = 100.0
y = 100.e10
n = 0
do i = 1, 10
   a(i) = i
enddo
read(10,option)
```

のようにプログラムを書いたとします．入力ファイルとして fort.10 という名のファイルに，

```
&option  x=10.0, a(3)=5.0 /
```

と書き込んでおけば，read 文実行後，x は変更されますが，y や n はそのままです．配列 a の場

合には，代入された要素 a(3)のみが変更されます.

なお, namelist に登録された変数の内容は, 次の namelist 出力文で出力することもできます.

```
write(nd,ネームリスト名)
```

この場合，全登録変数が“変数＝データ値”という形で出力されます．もっとも，namelist 出力文を実行すると，登録された全変数のデータが標準形式で出力されるので，変数が多いと煩雑です．取りあえず値を確認したいとき以外は，あまり使わない方が良いと思います.

2.6　ファイルのオープンとクローズ

write 文や read 文を使ってファイルから入出力をする場合, 何も指定がなければ, 装置番号 *nd* を付加した“fort.*nd*”という名のファイルを使用します．これに対し，任意のファイルを使いたいときには，open 文を使って入出力文の実行前にファイル名を指定しておきます．これを“ファイルをオープンする”といいます．open 文は以下の形式です.

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

[ ]の部分は省略可能です．それぞれの記述（制御指定子）の意味を表４に示します.

表４．open 文の制御指定子の意味

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

例えば，装置番号 10 のテキスト形式ファイルを“test.out”という名にするときは,

```
open(10,file='test.out')
```

と書きます．また，装置番号 30 のバイナリ形式ファイルを“test.dat”という名にするときは,

```
open(30,file='test.dat',form='unformatted')
```

と書きます．なお，存在するファイルを指定して write 文で書き込むと，それまで書き込まれていたデータが上書きされて消えるので注意が必要です. これを回避したいときには, status='new' と err=*num* を指定して，ファイルが存在する場合の処理を用意しておく必要があります.

status=*stat* を省略して open 文を実行すると，指定したファイルが存在しなければ，その名前のファイルが新たに作成されます．このため，read 文で入力するためのファイルをオープンする際にファイルが存在しないと，不必要な空のファイルが作成されてしまいます．これを防ぐために，ファイルが入力用のときは，status='old'を指定して，その存在をチェックした方が良いでしょう.

例えば, バイナリ形式ファイル“test.inp”を入力ファイルとして装置番号 20 に指定するには,

```
open(20,file='test.inp',form='unformatted',status='old',err=999)
```

のように書きます．この open 文には，err=999 というエラー処理が書かれているので，オープンした時点でファイルが存在しなければ，文番号 999 の行へジャンプします．

ファイルへ出力する場合，実行プログラムでの出力命令のタイミングと実際にディスクに書き込まれるタイミングは必ずしも一致していません．これは入出力ハードウェアを効率よく運用するために，一時記憶領域への読み書きが介在するためです．このため，確実に書き込みを完了させたいときにはファイルをクローズします．クローズは，次の close 文で行います．

```
close(nd)
```

nd はクローズするファイルの装置番号です．close 文を実行すると，その時点までに装置番号 *nd* に出力した全てのデータがディスクに書き込まれます．もっとも，プログラムが正常に終了すれば，全てのファイルが自動的にクローズされるので，通常は close 文を書かなくても問題ありません．

ファイルをクローズすると，装置番号 *nd* と open 文で指定したファイル名の関係は途切れます．このため，同じ装置番号に，別のファイルを新たに指定してオープンすることも可能です．また，同じ名前のファイルを再度オープンすることもできます．ただし，再オープンしてもそれ以前の読み書きの継続にはならず，ファイルの先頭に戻って読み書きをします．すなわち，“巻き戻し”をすることになります．ファイルを巻き戻すと，read 文による入力は一からやり直すことになり，write 文による出力はファイルの先頭から書き直します．このため，巻き戻してから write 文で出力すると，それ以前に書き込んだデータは消去されます．

なお，巻き戻すのが目的であれば，rewind 文を使うことで，クローズせずともファイルを巻き戻すことができます．rewind 文とは，次のような文です．

```
rewind(nd)
```

nd は再度最初から読み書きするファイルの装置番号です．巻き戻しを利用すれば，まず write 文を使ってファイルにデータを出力しておき，次にそれを巻き戻して，read 文を使ってそのデータを入力して使うことも可能です．

まとめと次回の予告

今回は，プログラムの拡大に伴って必要となるサブルーチンの使い方と入出力の詳細についてお話ししました．数値計算や計算機シミュレーションに必要な Fortran の基本的文法の説明は，これでほぼ完了です．これまで説明した文法を使って，どんどんプログラムを書いて下さい．たくさん書いて経験を積めば，どういう手順で計算機を動作させれば効率が上がるかがわかってくるので，さらにスマートなプログラムが書けるようになります．

さて，ここまでのプログラミングでは数値を個別に計算するのが基本でした．配列は複数のデータを一括して取り扱うことができますが，それを使って計算するときは，あくまでも要素ごとの計算を繰り返す形で記述しました．これに対し，最近の Fortran には，数値の集合を一つのデータ型としてとらえ，その集合間の演算を記述することで，複雑な処理をシンプルに表現する仕組みが導入されています．その一つは，“文字列”です．文字の集合を“文字列”というデータ型で取り扱い，文字列をつないだり，比較したりするという演算ができます．もう一つは“配列計算”です．実数や整数の配列を一つのデータとして取り扱い，do 文を書かなくても，配列間の演算を記述できます．

最終回である次回は，文字列や配列といった集合データ型とその演算について説明します．また，比較的新しい文法の中から，筆者が使って便利だと思われるものをいくつか紹介し，それらを利用してプログラムのメンテナンスを楽にするテクニックを説明したいと思います．

参考文献

[1] 入門 Fortran90 実践プログラミング，東田幸樹・山本芳人・熊沢友信，ソフトバンク，1994 年