東北大学サイバーサイエンスセンター講習会

Fortran 入門　解説書

(2021 年度版)

日本原子力研究開発機構　田口 俊弘

2021 年 9 月 10 日

目　　次

第 1 章　基本的な Fortran プログラムの書き方

コンピュータとは，プログラムに記述された指示に従って，一連の計算を自動的に実行する機械のことです．最近では，コンピュータの性能がアップして，“並列コンピュータ”という複数の計算を同時に実行できるコンピュータもあり，これらの性能を引き出すためのプログラムの書き方など，プログラミングも多様化していますが，命令を 1 個ずつ順番に実行するというコンピュータの基本的な動作原理が大きく変わったわけではありません．本章では，プログラムを書く際に知っておいた方が良いと思われる簡単なコンピュータのしくみの説明から始めて，電卓レベルの計算を Fortran で書いて実行できるようなプログラムの書式を説明します．

1.1　コンピュータのしくみとコンパイラ

コンピュータの大まかなしくみを説明しましょう．コンピュータの中心部を大きく分ければ，図 1.1 に示すように，計算を実行する **CPU** (Central Processing Unit，中央処理装置) と，データを保存する **主記憶装置** (メモリ) から構成されています．コンピュータの動作を指示するプログラムもメモリに保存されていますが，これらは **プログラム領域** と **データ領域** という別々の場所に保存されています．CPU は，基本的には 1 回あたり一つの動作を行うことしかできないので，その動作が終了するまで次の動作には移れません．この一つ一つの動作指令が **命令** です．命令には，四則演算を行う算術命令，メモリを指定して数値データを読み込んだり書き込んだりするデータ転送，周辺機器の動作制御などがあります．

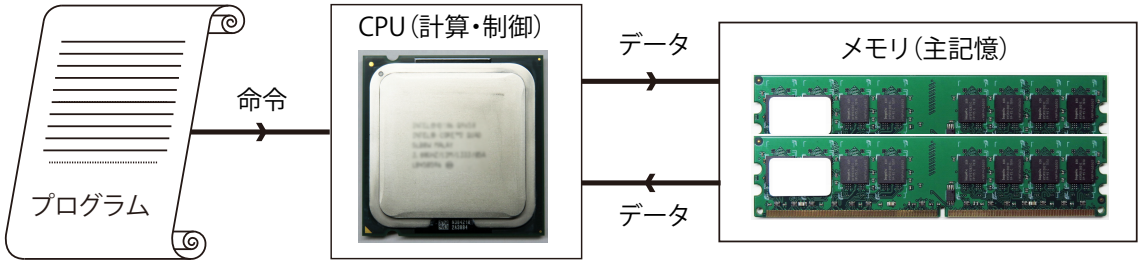


図 1.1　コンピュータのしくみ

計算機プログラムとはコンピュータへの命令を記述した文の集まりですが，CPU を動作させる本来の命令は，加算 1 回とかメモリ書き込み 1 回といった単純なものである上に，“機械語”と呼ばれる数値で表されているので，これで直接プログラムを書くことはまず不可能です．そこで，人間が理解しやすいように，単語や記号を使った数式を使って複数の計算機動作を 1 行で書けるようにしたものが，Fortran や C 言語のような **プログラミング言語** です．プログラミング言語を使ってプログラムを書いておけば，これを **コンパイラ** という機械語への翻訳ソフトを使って，コンピュータで実行可能な機械語 (実行形式) に変換することができます．しかし，コンパイラは翻訳機に過ぎません．プログラミング言語という，我々には理解しやすい書式で書けるとはいっても，あくまでもコンピュータを動作させる手順として書かねばならないのです．

さて，コンピュータの主記憶装置であるメモリは **RAM** (Random Access Memory) と呼ばれる半導体チップでできた記憶装置で，電源を切るとデータが消えてしまいます．このメモリ上のデータはプログラムでは **変数** という形で記述します．これに対し，一般的に RAM より記憶容量が大きくて，電源を切ってもデータを保持できる補助記憶装置としてハードディスクや SSD などがありますが，プログラム上ではこれらは「読む（入力する)」「書く（出力する)」という **入出力動作** の指定でデータのやりとり

を行います．また，パソコンなどを使うときにはディスプレイを見ながらキーボードでプログラムを入力したり実行したりしますが，これら機器とのデータのやりとりも「読む」「書く」という動作としてプログラムに記述します．このため，プログラミングの基本は，“変数”という形で表現されているメモリ上のデータを使って様々な計算を行い，必要があれば周辺機器との読み書きによってデータ保存や表示を行う，という形式になります．

このように，メモリ上のデータは変数で表現するため，ハードウェアを意識する必要はあまりないのですが，プログラミングをする上で知っておいたほうがいい概念に**メモリアドレス**があります．データを保存するメモリには“アドレス”または“番地”と呼ばれる通し番号が付いていて，CPU はこの番号を使って読み書きするメモリを指定します．Fortran プログラムではメモリアドレスに相当する数値を直接操作することはありませんが，プログラムの書式の中にはメモリに保存されている数値ではなく，メモリアドレスで指定している場合がいくつかあります．例えば次の文を考えてみましょう．

```
a = x
```

これは“代入文”といって，「右辺の変数 x の値を左辺の変数 a に代入する」という動作を表しています．計算機的には，x のメモリに保存されているデータを読み込んで，a のメモリに書き込むという動作を示しています．このとき注意して欲しいのは，右辺の変数と左辺の変数では意味が違うことです．右辺の x は，x という名で指定したメモリに格納されている数値を示しているのですが，左辺の a は保存先のメモリアドレスを指定しています．このため，

```
10 = 2
```

のような数値に代入する代入文や

```
x+b = 5
```

のような数式に代入する代入文はありえないわけです．もっとも，“代入文”という意味を考えれば，この程度のことは意識せずとも間違うことはないと思います．しかし，プログラムの書式の中には，変数を与えた場合に数値ではなくアドレスを指定していることを意識した方が良いものがいくつかあります．そのあたりを理解した上でプログラムを作成すれば，思わぬエラーが発生した時の原因を探る手がかりになります．“プログラムエラー”というのは，単に計算式を書き間違えて実行結果が予想値と異なるという場合だけではありません．動作中に突然実行が停止するといった予想外の動作で現れるエラーもあるのです[1]．

1.2 コンピュータにおける基本的動作

コンピュータはプログラムに書かれた一連の計算手順に従って処理を行います．このため，プログラムに記述された動作を上から順に処理していくのが基本です．例えば，図 1.2(a) のように書かれたプログラムでは，まず，“x=100”という動作をし，それが終わったら“y=x-50”という動作をし，それが終わったら“z=x*y*20”という動作をする，というように，一つの動作が完了したら次の動作を開始する，という手順になります．

しかし，順番に動作するだけではありません．ジャンプ命令を使えば，プログラムの格納場所を指定して，動作開始位置を一気に移動することもできます．ジャンプは先に進むこともできるし，前に戻ることもできます．前に戻れば，図 1.2(b) のように同じ手順を何度もくり返すことになります．大量のデータを処理するプログラムでは，このようなくり返し動作 (ループ) を適宜利用して，効率良く計算手順を記述することが要求されます．

コンピュータの動作で，もう一つ重要なのが条件分岐です．条件分岐とは，指定した条件に応じて実行する処理を変えることですが，コンピュータの動作としては，図 1.2(c) のように条件に応じて次に実

[1] プログラムを作成するときに発生するエラーの対処法については付録 B を参考にして下さい．

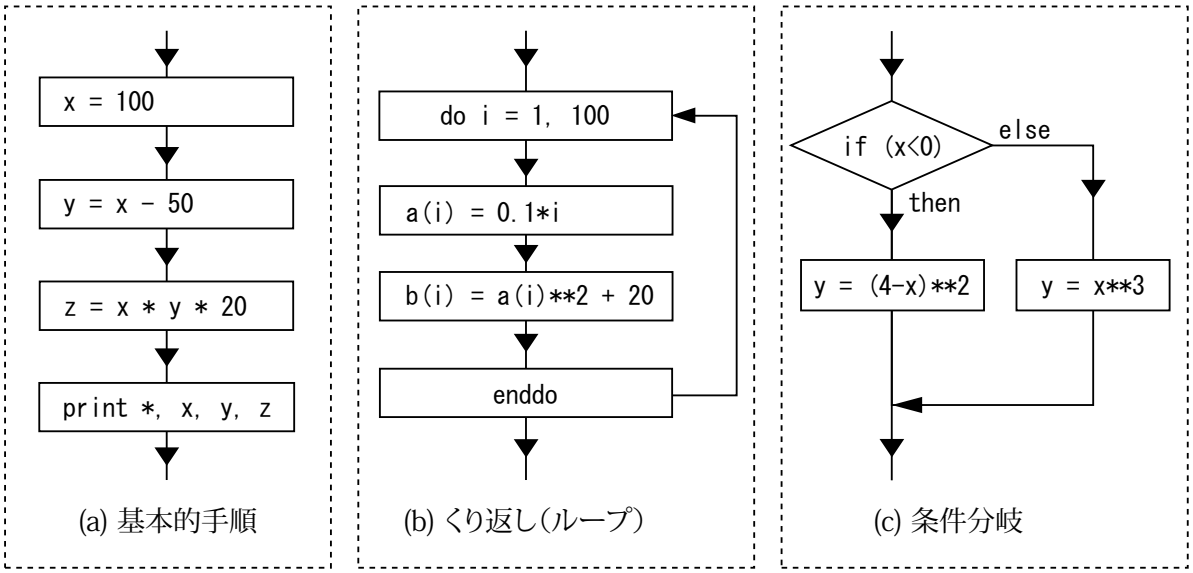


図 1.2　プログラムで記述する様々な動作

行するジャンプ先を変えて異なる処理をするようになっています．

プログラミング言語には様々な書式が用意されていますが，基本的動作はこの程度です．色々用意されているのは，プログラマにとってわかりやすく簡便に書けるように工夫されているだけです．Fortran でのくり返しの記述は，主として 2.2 節で説明する `do` ブロックを利用して書きます．また，条件分岐は 3.1 節で説明する `if` 文や `if` ブロックで書きます．

1.3　Fortran プログラムの書式

Fortran プログラムは，基本的にコンピュータに与える命令を，次の形をした“文”で記述します．

```
動作指示語　動作制御パラメータ
```

すなわち，先頭に **動作指示語** と呼ばれるコンピュータの動作を指定する単語を書き，それに続けて，動作を制御するための数値や予約語を **動作制御パラメータ** として記述します．ただし動作制御パラメータを記述する必要のない，動作指示語だけの命令も存在します．

実行形式に変換されたプログラムを起動すると，コンピュータは，

```
実行開始処理　→　プログラムに記述された命令を順に実行　→　実行終了処理
```

という流れで動作します．実行を開始した時に最初に実行する部分を **メインプログラム**，または **メインルーチン** といいます．言わばプログラムの本体です．本章から第 3 章までは，メインプログラムだけを使った Fortran の基本的プログラムの書き方について説明します．メインプログラムと同レベルの完結したプログラムには，他のプログラムの中から実行開始を指示してその機能を利用する **サブルーチン** がありますが，これについては第 4 章で説明します．

1.4　メインプログラムの開始と終了

Fortran では特別な手続きをせずにプログラムを書くと，それがメインプログラムと仮定されます．しかし，それではサブルーチンと区別しにくいので，**program 文** を用いて最初にプログラムの名前を書きます．`program` 文は以下の形式です．

```
program プログラム名
```

これに対し，メインプログラムの終了は **end program 文** で指定します[2].

```
end program プログラム名
```

ここで，program 文と end program 文で指定する “プログラム名”は同じでなければなりません．このため，メインプログラムは次のような構造になります．

```
program code_name
  .......
  .......
  .......
end program code_name
```

プログラム名は，頭文字が a〜z のどれかであれば，後はアルファベットの a〜z や数字の 0〜9 をどの様な順序で並べたものでも良いので，プログラムの内容を代表するような名前を指定して下さい．また，この例のようにアンダースコア (_) も使えるので，これをスペースの代わりに使えば単語をつないだ長いプログラム名を付けることもできます．

program 文と end program 文の間に Fortran の文法に従った文を並べて，動作させたい計算手順を記述します．動作手順を記述する文を **実行文** といいます．しかし，プログラムに記述するのは実行文だけではありません．計算途中で必要となる変数 (メモリ) 領域を確保するための文 (宣言文) も書かねばなりません．このような計算動作に直接携わらない文を **非実行文** といいます．Fortran では，非実行文をプログラムの最初に集約して，実行文はその後に書きます．このため，動作の開始は program 文の次の文ではなく，最初の実行文になります．

完結したメインプログラムの 1 例を示します[3].

```
program test_code
  implicit none
  real x,y,z
  x = 5
  y = 100
  z = x + y*100
  print *,x,y
  print *,'z =',z
end program test_code
```

このプログラムにおける実行文・非実行文の区分と，動作開始から終了までの実行の流れを図 1.3 に示します．実行形式のプログラムを起動すると，最初の実行文から動作が開始し，上から順に 1 行ずつ実行して，end program 文に到達した段階で動作終了となります．1.2 節で述べたように，do 文を使ったくり返しや，if 文による条件分岐など，動作指定によっては所定の位置にバックしたり，条件に応じて実行するかどうかの選択ができますが，それでもそれぞれの文が終了した後に次の文が実行されるという基本的動作は変わりません．

これはコンピュータが 1 回に一つの動作しかできないためです．プログラム全体を見渡して実行するのではなく，1 行ずつ順に実行していくので，バックするような動作指定をしない限り，下方に書いた実行文は上方に影響を及ぼしません．プログラムはこのことを常に念頭に置いて書かなければなりません．

なお，if 文を使って，条件によって途中でプログラムを終了するときや，サブルーチン内でプログラムを終了するときには **stop 文** を用います．stop 文を実行すると，その時点でプログラムが終了します．

[2] end program 文は end だけの end 文でも良いのですが，メインプログラムの名前を入れておくと終了点を検索するときに便利です．将来長いプログラムを書くことを考えて，end program 文で終了する習慣をつけておきましょう．

[3] この例のように，プログラム内部の文は，先頭にスペースを数個入れて，program 文と end program 文よりも少し右にずらして書きます．そうすれば，メインプログラムの範囲が明確になります．

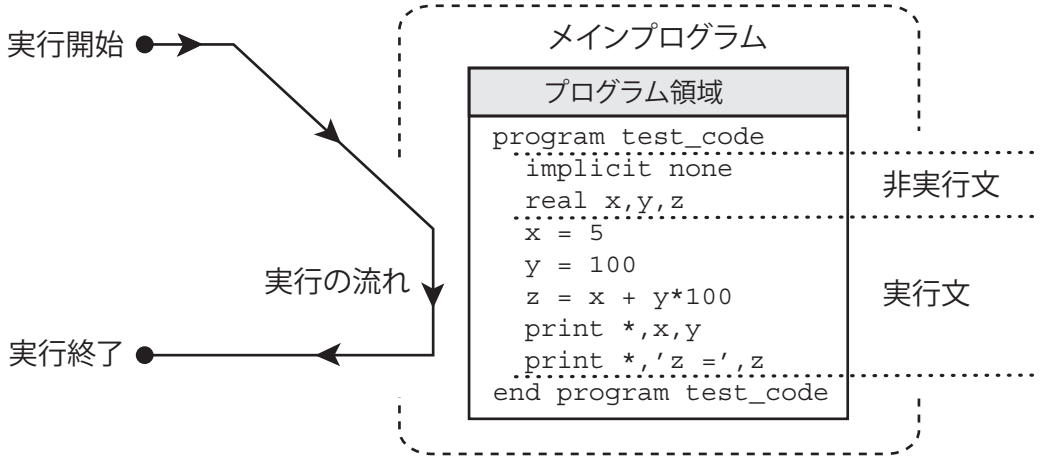


図 1.3　プログラム開始から終了までの流れ

例えば,

```
program fluid_code
  implicit none
  real x,y
  integer m,n
   .......
  if (m < 0) stop
   .......
end program fluid_code
```

と書けば，m＜0 の時にプログラムは終了し，それ以降は実行しません[4].

1.5　代入文と演算の書式

実行文には，stop 文のように常に同じ動作を指示する文と，いくつかの“数値”を伴って，その数値に応じた動作を指示する文とがあります．プログラム中で“数値”を与える方法は 3 種類あります．

(1) −500 とか 3.14 のような数字を直接書く **定数**
(2) x とか yrz のような単語で示した **変数**
(3) それらを使って x+3 や sin(y-5) のような計算手順を表した **計算式**

です．(3) の“計算式”に書かれている手順もコンピュータの動作なのですが，プログラム中の計算式は，その計算結果を動作命令に与える“数値”として位置づけられています．このため，計算式を書いただけでは実行文になりません．

プログラムで最も良く使う実行文は **代入文** です．これは,

```
        変数 = 計算式
```

という形をしています．計算式の代わりに，定数や変数を書くこともできます．1.1 節で説明したように，プログラムにおける“変数”とはコンピュータのメモリ領域のことで，数値を入れる箱だと考えればいいでしょう．代入文は右辺の“数値”を左辺で示した変数メモリに格納する動作を表します．よって,

```
        計算式 = 計算式       !  これはエラー
```

という形は使えません．Fortran において“＝”という記号は，“左辺と右辺が等しい”という意味ではないからです．例えば,

[4] if 文の使い方は，3.1 節で説明します．

```
x = 4 + 2
y = 9 - x
y = y + 1
```

のようなプログラムを考えてみましょう．プログラムは上から順に実行されるので，まず 4 + 2 の結果である 6 が変数 x に代入されます．次に，9 から変数 x に保存されている数値を引いた値が変数 y に代入されるので，y には 3 が代入されます．最後の文は数学の等式と見れば変ですが，イコールが “代入する動作”だと理解できれば特に不思議な文ではありません．代入という動作は，右辺の全ての演算が終了した後で実行されるため，最後の文では，まず y+1 を計算し，その結果が y に代入されます．すなわち y には 4 が代入されます．このとき，それまで y のメモリに入っていた “3”という数値は上書きされて消えてしまいます．

Fortran における基本演算の書き方と使い方を表 1.1 に示します．

表 1.1　演算記号の書き方と使い方

演算記号	演算の意味	使用例	使用例の意味
+	足し算	x+y	$x + y$
-	引き算	x-y	$x - y$
*	掛け算	x*y	$x \times y$
/	割り算	x/y	$x \div y$
**	べき乗	x**y	x^y
-	マイナス	-x	$-x$

べき乗までの二項演算には以下のような優先順位があり，優先順位の高い方が低い方より先に計算されます．これは数学における演算順序と同じです．

べき乗 ＞ 掛け算または割り算 ＞ 足し算または引き算

掛け算と割り算のような同レベルの演算は左から順に計算されます．さらに，かっこを使えばかっこの中が優先的に計算されます．例えば，

```
f = x + (y-3)/z**2
```

という文では，一番最初に y-3 を計算し，次に z の 2 乗を計算し，次に y-3 の結果を z の 2 乗の結果で割り，その結果を x に加えて，最後に f に代入する，という動作になります．

なお，時々見かける間違いに，

$$f = x + y/ab$$

という数式を次のようなプログラムにするものがあります．

```
f = x + y/a*b
```

数学では，y/ab の記述を $\dfrac{y}{ab}$ と解釈することが多いのですが，プログラムでは掛け算と割り算が同じレベルなので，y/a*b は “y を a で割って，その結果に b を掛ける”です．よって，数学的解釈になるようにするには，

```
f = x + y/(a*b)
```

と書かなければなりません．

1.6 数値の型

コンピュータという機械が取り扱う数値は2進数で表されていて，基本的には整数です．例えば，1byteの数は，8bit，つまり0か1のどちらかを選択できる数が8個並んだもので，10進数では0〜255 ($=2^8-1$)までの整数を表すことができます[5]．Fortranでは小数点のない数字，56とか-1112などの数字を**整数型**といいます．Fortranの整数型数は4byte (32bit)で表現されていて，-2^{31}〜$2^{31}-1$の整数を扱うことができます．

しかし，数値計算や計算機シミュレーションをする場合には，3.14のように小数点以下を含んだり，3×10^{19}とか，1.67×10^{-27}のような様々なスケールの数値を取り扱うため，整数型だけでは不便です．特に，整数型数は小数点以下の表現ができないので，割り算をすると全て“切り捨て”になります．これを忘れてプログラムを書くと，よく間違いを起こします．例えば，

```
x = (5/2)**2
y = x**(3/2)
t = 2/3*y*x
```

と書くと，x，y，zはいくつになると思いますか？　答えは$x=4$，$y=4$，$t=0$です．これは整数の割り算が切り捨てになるため，5/2 $\rightarrow 2$，3/2 $\rightarrow 1$，2/3 $\rightarrow 0$になるからです．

そこで，整数型の他に，3.14のような少数点以下を含んだり，1.67×10^{-27}のような，$A\times10^B$という形式の数値を取り扱うことができます．これを**実数型**といい，Aの部分を**仮数部**，Bの部分を**指数部**といいます．通常，数値計算は実数型を使って計算しなければなりません．

実数型には有効数字の違う2種類が用意されています．**単精度実数型**と**倍精度実数型**です．単精度実数とは4byteで表される実数のことで，仮数部の有効数字は7桁程度です．これに対し，倍精度実数とは8byte (64bit)で表される実数のことで，仮数部の有効数字は15桁程度です．7桁あれば問題ないと思われるかもしれませんが，何百万個もの数字の合計を計算したり，次数の高い多項式の計算をするときなどでは，計算回数の増加につれて有効桁数が減少することを考慮しなければなりません．このため，通常は倍精度実数で計算をします．

Fortranにおける基本的な実数型は単精度であり，倍精度実数型を使用する時にはそれを意識した書式で書かなければなりません．しかし，最近のコンパイラはオプションを指定すればデフォルトの実数型を倍精度にする**自動倍精度化機能**を持っているので，本書では，単精度実数型と倍精度実数型を使い分けることはせず，単に“実数型”と表現します[6]．よって，特に単精度で計算する必要がなければ，コンパイル時に自動倍精度化オプションを付加して倍精度で計算して下さい．多くのコンピュータは倍精度実数計算を高速で行えるハードウェアを内蔵しているので，倍精度計算をしてもさほど実行速度は下がりません．

プログラム上で，整数型の定数と実数型の定数は小数点の有無で区別します．例えば，100とか，-12345と書けば整数型ですが，100.0とか，-12345.とか，-0.0314とか書けば実数型になります．また，6.2×10^{23}を入力したい時には，6.2e23と書きます．すなわち，$A\times10^B$はAeBと書きます．Bが負の場合でも1.6e-19のようにeの後に続けて書きます．また，6e20のように整数にeBを付加した形の数値も実数型になります．例えば，

$$a = 3.141592\,r^2 + 3x^5 + 6.5\times10^{-5}\,x - 10^5$$

という式をプログラムで書けば，以下のようになります[7]．

```
a = 3.141592*r**2 + 3.0*x**5 + 6.5e-5*x - 1e5
```

[5] bitやbyteの詳細と実数型の表現については1.11.4節参照．
[6] 自動倍精度化については付録C参照．精度の変更については付録Dで説明しています．
[7] 間違えて，10^5を10e5と書かないようにして下さい．

この例のように，r**2 とか x**5 のように整数べき乗の指数 (2 とか 5) には整数型を使います．これを r**2.0 とか x**5.0 とか書いても良いのですが，処理速度が遅くなる可能性があります．

先ほど整数型を使ったら期待どおりの結果が出ない例を挙げましたが，以下のように実数型を使えば正しい結果が得られます．

```
x = (5.0/2.0)**2
y = x**(3.0/2.0)
t = 2.0/3.0*y*x
```

なお，ここまでの説明だけだと整数型は必要ないと思われるかもしれませんが，そうではありません．2.1 節で説明する配列の要素を指定する数や，2.2 節で説明する do 文のカウンタ変数は整数型でなければなりません．また，オン・オフを表すだけの変数を作るときにも整数型を使います．すなわち，整数型数は“順番”や“区別”を表すときに不可欠です．また実数の整数部を取り出したいときや，剰余 (余り) を計算するときも整数型を利用します．数値計算プログラムの中でも整数型の使い道は多いのです．

計算式中に異なる数値型が混在する時は，情報量の多い方の型に合わせて計算し，その型の値が結果になります．このため，実数型と整数型の計算は整数型の数値を実数型に変換して実数型と実数型の計算を行い，実数型の結果になります．ただし，掛け算と割り算の計算は左から順に行うので注意が必要です．最初の整数型を使った計算例で，

```
t = 2/3*y*x
```

が 0 になるのは，最初に計算されるのが 2/3 という整数型÷整数型だからです．y や x が実数型であれば，計算順序を次のように変えることで切り捨てを避けることができます．

```
t = y*x*2/3
```

Fortran の便利な機能の一つは，複素数が使えることです．複素数にも単精度複素数型と倍精度複素数型がありますが，自動倍精度化機能を使えばデフォルトが倍精度複素数型になるので，本書では単に **複素数型** と表現します．複素数型の定数は，

```
(0.0,1.0)
(1e-5,-5.2e3)
(-3200.0,0.005)
```

のように，2 個の実数をコンマでつないで，かっこで囲んだものです．左の数が実部，右の数が虚部です．すなわち，この例は，それぞれ，i，$10^{-5}-5.2\times10^{3}\,i$，$-3200+0.005\,i$ を表した複素数型定数です．複素数型と実数型の計算の場合にはその実数型の数値を実部とした複素数型にして複素数計算をし，結果は複素数型になります．

1.7 変数の宣言

次に，数式の計算結果を保存する **変数** の使い方を説明します．変数の名前は，プログラム名と同様に，頭文字が a〜z のどれかであれば，後は a〜z，0〜9 をどの様な順序で並べたものでも使えます．例えば，abc とか k10xy 等です．Fortran では大文字と小文字を区別しないため，abc と ABC は同じ変数になります．変数にも型があり，計算結果に応じた型の変数を使わなければ正確に保存することができません．この変数の型を決める文を **宣言文** (型宣言文) といいます．型を決めると同時に，変数のメモリ領域を確保します．このため，計算に用いる変数は全て宣言しなければなりません．宣言は一度しかできず，プログラムの実行時に変更することはできません．宣言文は“非実行文”です．

ところが Fortran には“暗黙の宣言”があり，通常は宣言しなくても文法的な間違いにならないので，タイプミスなどで思わぬエラーが発生する可能性があります．これを防ぐため，プログラムの 2 行目，すなわち program 文の次の行に必ず次の **implicit 文** を書いておきます．

```
implicit none
```

この文を入れておけば，宣言せずに使用した変数があるとコンパイルエラーになり，タイプミスのチェックができます．

数値計算は基本的に実数で行いますが，実数型変数は次のように宣言します．

```
real 変数1，変数2，...
```

これに対し，整数型の変数は，次のように宣言します．

```
integer 変数1，変数2，...
```

宣言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．例えば，メインプログラムの始まりは以下のようになります．

```
program test1
  implicit none
  real x,y,z,omega,wave,area
  integer i,k,n,imin,imax,kmax
   .......
```

real や integer 等の数値型宣言文は何行書いても良いし，順番も無関係です．

Fortran の暗黙宣言では，名前の頭文字が a〜h か o〜z の変数は実数型で，i〜n の変数は整数型でした．このため，整数型変数の頭文字を i〜n にする習慣があります．全てを宣言する以上，基本的に変数名の付け方に制限はないのですが，整数型は用途が限られているので，頭文字を限定する方が良いと思います．特に，くり返し処理で良く使う i,j,k,l,m,n の 1 文字は実数型変数として使わない方が無難です．なお，本書のプログラム例において，宣言文を記述しないで変数を使うときは，この暗黙宣言に従った数値型を仮定しています．

複素数型変数の宣言文は，

```
complex 変数1，変数2，...
```

です．複素数型も用途が限定されているので名前の付け方に規則をつける方が良いでしょう．複素数型変数名は，頭文字を c か z にすることが多いようです．

プログラムにおいて，数値を変数に保存する意義は大別して二つあります．一つはプログラム全域にわたって共通してその値を利用するためであり，もう一つは狭い範囲で計算結果を一時的に保存するためです．この二つは意識して使い分けるようにします．特に，前者の大域的に利用する変数には長めで意味のある名前を付けるべきです．これは，1 文字のような短い変数名を使うと，プログラムが長くなるにつれて，どこでその変数を使っているかを検索するのが難しくなるからです．

変数に数値を代入する時は，右辺の計算結果を左辺の変数の型に変換して代入します．このため，実数の計算結果を整数型の変数に代入すると，小数点以下は切り捨てられます．例えば，

```
real x
integer n
x = 3.14
n = x + 6
```

の結果，n に代入される値は 9 です．

また，複素数型の計算結果を実数型の変数に代入すると，その複素数の実部が代入されます．例えば，

```
real x
complex c
c = (1.0,-2.0)
x = -c
```

とすると，x＝−1.0 になります．逆に，複素数型の変数に実数型の数値を代入すると，実部に結果が代入され，虚部は 0 になります．例えば，

```
real x
complex c
x = 5
c = x**2
```

とすると，c＝(25.0,0.0) になります．

1.8 組み込み関数

数値計算上よく使う数学関数はあらかじめ用意されています．この用意された関数を **組み込み関数** といいます．表 1.2 に代表的な組み込み数学関数を示します．

表 1.2 数学関数の書き方と使い方

組み込み関数	名　称	数学的表現	必要条件	関数値 f の範囲
sqrt(x)	平方根*	$\sqrt{x}$	$x \geqq 0$	
sin(x)	正弦関数*	$\sin x$		
cos(x)	余弦関数*	$\cos x$		
tan(x)	正接関数	$\tan x$		
asin(x)	逆正弦関数	$\sin^{-1} x$	$-1 \leqq x \leqq 1$	$-\frac{\pi}{2} \leqq f \leqq \frac{\pi}{2}$
acos(x)	逆余弦関数	$\cos^{-1} x$	$-1 \leqq x \leqq 1$	$0 \leqq f \leqq \pi$
atan(x)	逆正接関数	$\tan^{-1} x$		$-\frac{\pi}{2} < f < \frac{\pi}{2}$
atan2(y,x)	逆正接関数[8]	$\tan^{-1}(y/x)$		$-\pi < f \leqq \pi$
exp(x)	指数関数*	e^x		
log(x)	自然対数*	$\log_e x$	$x > 0$	
log10(x)	常用対数	$\log_{10} x$	$x > 0$	
sinh(x)	双曲線正弦関数	$\sinh x$		
cosh(x)	双曲線余弦関数	$\cosh x$		
tanh(x)	双曲線正接関数	$\tanh x$		

関数名の後のかっこは必ず必要です．かっこの中の x や y を **引数** (ひきすう) といいます．関数を計算式中に記述すると，引数に対する関数値が結果となってその計算式を実行します．例えば，

```
c = exp(-x**2) + sin(10*x+3) - 2*tan(-2*log(x))**3
```

のように書くことができます．この例のように関数の引数に関数を使った計算式を与えることも可能です．

表 1.2 の数学関数は，引数に実数型数を与えると実数型の結果になる“実数型関数”ですが，名称に“*”の付いている関数は，引数が複素数型でも使えます[9]．Fortran の組み込み関数には，引数の型や精

[8] 逆正接関数 atan2(y,x) は，座標点 (x, y) の偏角を計算する関数です (x と y の順序に注意)．よって，表の数学的表現には“$\tan^{-1}(y/x)$”と記述してありますが，実際には x と y の値に応じて $-\pi$ から π の間の角度が結果になります．

[9] 表 1.2 の必要条件と関数値の範囲は引数が実数型の場合です．この必要条件を満たさない実数型数を与えると実行時エラーになります．しかし複素数型の数値を与えた場合にはエラーになるとは限りません．

度に応じて計算をする **総称名機能** があるので，複素数を引数に与えた場合の関数値は複素数型の結果になります．

ただし，総称名機能では引数の数値型に対応した関数が用意されていないとコンパイルエラーになります．例えば，$\sqrt{2}$ を計算したくて，sqrt(2) と書くとエラーです．“2”は整数型の定数であり，整数型数の平方根は用意されていないからです．sqrt(2.0) のように実数型を引数に書かなければなりません．

この他，絶対値や余りを計算するのにも組み込み関数を使うし，実数型を複素数型に変換するなどの型変換も組み込み関数を使って処理します．その中の代表的なものを表 1.3 に示します．この表の最初の 3 個は引数が整数型の関数であり，それ以外は引数が実数または複素数型の関数です．

表 1.3　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	実数型に変換
abs(n)	絶対値	整数	整数	n の絶対値
mod(m,n)	剰余[10]	2 個の整数	整数	m を n で割った余り
int(x)	整数化	実数	整数	整数型に変換 (切り捨て)
nint(x)	整数化	実数	整数	整数型に変換 (四捨五入)
sign(x,s)	符号の変更	実数	実数	s$\geqq 0$ なら |x|, さもなくば $-$|x|
abs(x)	絶対値	実数または複素数	実数	x の絶対値
mod(x,y)	剰余	2 個の実数	実数	x を y で割った余り
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

型変換関数は，数値型が限定されている場所に，その数値型以外の値を指定したいときなどに使います．例えば，整数型の変数 n の平方根を計算したいときには，上記のように sqrt(n) と書くことはできません．このときは，sqrt(real(n)) と書きます．また，整数化は切り捨て演算を含んでいるので，こ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．

複素数型の定数は，(1.0,-2.0) のような表現で指定することができますが，変数や計算式で実部と虚部を指定した複素数を作るときには関数 cmplx を使います．例えば，

```
complex z,z1
real x,y
x = 10.0
y = -3.5
z = cmplx(x,y)
z1 = cmplx(x**2,y+2.0)
```

のように書きます．この場合，z = x + iy であり，z1 = $x^2 + i(y + 2)$ になります．

[10]剰余を計算する関数 mod の利用例としてくり返しの制御があります．3.1 節を参照して下さい．

1.9 print 文による簡易出力

数値計算を目的としたプログラムでは，得られた計算結果を表示したりファイルに保存する必要があります．入出力の詳細については第 5 章で説明しますが，とりあえず **print 文** を使った標準形式による数値出力を覚えておきましょう．print 文の 1 例を以下に示します．

```
integer n
n = 3
print *, 4+5, n, n*2, 2*n-11
```

このように，“print *,”に続いて変数や計算式をコンマで区切って並べると，それらの計算結果が横に並んで出力されます．上例の場合には，4+5 の結果である 9 から 2*n-11 の結果である −5 までが以下のように出力されます．

```
           9           3           6          -5
```

なお，“print *,”の“*”は，標準形式で出力することを意味しているのですが，とりあえずは形式的に書くものだと覚えて下さい[11]．

複数の数値を出力するときに数字だけを出力すると，どれがどの数値かわからなくなる可能性があります．このような場合は，文字列を併用して変数の意味を同時に出力します．Fortran における“文字列”とは，2 個のアポストロフィ「’」または 2 個のダブルクォーテーション「"」ではさんだ文字の並びのことで，print 文中で数値といっしょに並べると，その文字の並びがそのまま出力されます．例えば，

```
real x
x = 3
print *,'x = ',x,'  x**3 = ',x**3
```

というプログラムの出力は，

```
x =    3.00000000000000        x**3 =    27.0000000000000
```

となります．このように，実数型の数値を出力すると実数型数の有効数字で出力されます．

1.10 コメント文，継続行，複文

Fortran では“!”の後に続く文字は全て無視されます．すなわち何を書いても実行とは無関係です．これを **コメント文** といいます．コメント文を機会あるごとにプログラム中に入れて，書かれている内容を表示するように心がけましょう．さもなくば，プログラムが長くなるにつれて，自分でも何を書いたのか忘れてしまいます．例えば，

```
!  area of circles
   s = pi*r*r
```

のように，1 列目に“!”を書けば，その行はコメント行になります．また，

```
s = pi*r*r          !  円の面積
v = 4*pi*r*r*r/3    !  球の体積
```

のように，実行文の末尾に書き込むこともできるし，日本語を書くこともできます．ただし，日本語環境によっては正しく認識できずに文字化けすることがあります．可般性を考慮するならローマ字つづりでもよいから半角英数字だけでコメントを書いた方が無難です．

計算式が非常に長くて，作成に使っているエディタウィンドウの横幅を越えると読みづらくなります．また，コンパイラによっては 1 行に書ける文字数に制限があるので，そのままではエラーになることもあ

[11] 出力形式の指定方法は 5.3 節参照．

ります．そこで，1 行の文を途中で改行して，複数行に分割して書くことができます．これを **継続行** といいます．分割したときは，次の行に継続するという意志を示す印として，行末に “&”の文字を書きます．例えば，

```
print *,alpha,beta,gamma,delta,epsilon,zeta,eta,iota,kappa,lambda,mu
```

という 1 行は，

```
print *,alpha,beta,gamma,delta,epsilon &
       ,zeta,eta,iota,kappa,lambda,mu
```

と分割して書くことができます．継続行は何行にわたってもかまいません．

```
print *,alpha,beta,gamma &
       ,delta,epsilon &
       ,zeta,eta,iota &
       ,kappa,lambda,mu
```

と書いても同じです．ただし，最後の行に “&”を付加するとエラーになるので注意して下さい．

長い文字列を書くときは，文字列の最後に “&”を入れて継続させることができます．その際，次の行の開始文字の前にも “&”を書いておきます．例えば，

```
print *,'ABCDEFG&
       &hijklmn'
```

と書けば，ABCDEFGhijklmn と出力されます．

逆に，1 行に複数の文を書くことも可能です．これを **複文** といいます．2 個以上の文を 1 行で書くには，文と文を分離する記号として “;”の文字を書きます．例えば，

```
x = 1; y = 2; z = 3
```

のように書くことができます．ただし，複文の使用はこのような短い代入文を書く場合に限定して下さい．プログラムは 1 行で一つの完結した動作を表すのが基本です．1 行に複数の文を書くと，動作の流れが見えにくくなるので，多用しない方がよいと思います．

1.11 プログラミング スマートテクニック (その 1)

計算手順が複雑になったり大量のデータを処理するようになると，数式を単純にプログラムに置きかえるのではなく，コンピュータの特性を考慮したプログラムにすることで計算効率や精度を高めることができます．ここでは，計算速度を向上させるテクニックや，計算精度を高める書き方について説明します．また，コンピュータで利用できる数値の大きさについても説明します．

1.11.1 演算の速度を考える

コンピュータの計算動作は a+b のような加算が基本です．a-b のような減算は b を負数に変換して加算するだけなので加算とそれほど実行時間は変わりませんが，a*b のような乗算は加算のくり返し動作ですから，加減算よりかなり遅いです．a/b のような除算にいたっては，減算のくり返しを条件付きで行うので，さらに時間がかかります．この演算速度の比較を書けば次のようになります．

```
加減算 ≫ 乗算 ≫ 除算
```

このため，割り算ができるだけ少なくなるような計算手順にします．例えば，

```
x = a/b/c
```

と書くより，

```
x = a/(b*c)
```

と書く方が速くなります．最近のコンピュータはかなり高速に計算することができるので，数回の計算だけならこのような書き換えは必要ありませんが，大量のデータを使ったくり返し処理をする時には価値があります．

べき乗算はもっと遅いので，2 乗や 3 乗程度のときは掛け算にする方が速くなります．例えば，

```
x = a**2 + b**3
```

と書くより，

```
x = a*a + b*b*b
```

と書く方が速くなります．ただし，指数が大きいときには意味がないし，プログラムもわかりにくくなるので 3 乗程度まででよいと思います．

実数のべき乗 (1.2 乗とか，3.7 乗など) は，対数関数と指数関数を使って計算するので時間がかかります．このため，$\frac{1}{2}$ 乗に関しては，組み込み関数 `sqrt` を利用する方が速いです．例えば，

```
y = x**0.5
z = y**1.5
```

と書くより，

```
y = sqrt(x)
z = y*sqrt(y)
```

と書く方が速くなります．

1.11.2 桁落ちに気を付ける

実数型数の有効数字は倍精度でも 15 桁程度です．よって，値の接近した 2 個の実数の引き算をするときは気を付けなければなりません．例えば，

$$2000.06 - 2000.00 = 0.06$$

ですが，計算結果 0.06 は元の 2000.06 と比べると有効数字が 5 桁も少なくなっています．これを **桁落ち** といいます．桁落ちするような引き算はできるだけ避けねばなりません．

例えば，次のように変数 `x1` に小さい正の数 `h1` を加えて `x2` を計算し，`x2` に小さい正の数 `h2` を加えて `x3` を計算したとします．

```
real x1,x2,h1,h2
x1 = 1.0
h1 = 1e-7
h2 = 2e-7
x2 = x1 + h1
x3 = x2 + h2
print *,x3-x1,h1+h2
```

このプログラムをパソコンで計算したところ，出力結果は以下の様になりました．

```
 2.999999999531155E-007 3.000000000000000E-007
```

すなわち，`x3-x1` は小数点以下 10 桁目以降が正しくありません．このことから，`x3` と `x1` の差が必要なときには，引き算で直接計算するのではなく `h1+h2` で計算するべきであることがわかります．

2 次方程式 $ax^2 + bx + c = 0$ の解を求めるときにも注意が必要です．この方程式の解の一つは，

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

ですが，$b > 0$ かつ $b^2 \gg |4ac|$ の時には，b と $\sqrt{b^2 - 4ac}$ がほぼ等しくなるので，$-b + \sqrt{b^2 - 4ac}$ の計算をすると桁落ちする可能性があります．そこでこの解を計算する時は，分子の有理化をした公式を使います．すなわち，分母分子に $-b - \sqrt{b^2 - 4ac}$ を掛けると，

$$x_1 = \frac{\left(-b + \sqrt{b^2 - 4ac}\right)\left(-b - \sqrt{b^2 - 4ac}\right)}{2a\left(-b - \sqrt{b^2 - 4ac}\right)} = \frac{2c}{-b - \sqrt{b^2 - 4ac}}$$

となりますが，最後の公式を使えば桁落ちする心配がありません．2 解とも計算する時には，2 解の積が c/a であることを利用して，まず桁落ちしない方の解 x_1 を計算した後で，もう一方の解を $c/(ax_1)$ で計算すると良いでしょう．

1.11.3 π の作り方

物理や数学の計算をするときには円周率 π を使うことが多いです．π の値を倍精度計算用に有効数字 16 桁で表せば，

```
real pi
pi = 3.141592653589793
```

となります．これをこのまま使えば良いのですが，コンピュータによっては有効数字が異なるし，数値を覚えるのも面倒です．そこで逆三角関数を使って計算で求める方法がよく使われています．例えば，

```
real pi
pi = acos(-1.0)
pi = 2.0*asin(1.0)
pi = 4.0*atan(1.0)
```

などのように書くことができます．関数計算には時間がかかりますが，π の値は変化しないので計算するのは 1 回だけです．問題になるほどではありません．

1.11.4 コンピュータで表現可能な数値の大きさ

我々が日常的に使っている 10 進数は，10 のべき乗が基本です．すなわち，数字を 10 の多項式で表したときの係数を次数の大きい方から並べたものです．ただし，係数は 10 より小さい整数 (0〜9) でなければなりません．例えば，1203 という数は，10 の多項式で表せば，

$$1203 = \underline{1} \times 10^3 + \underline{2} \times 10^2 + \underline{0} \times 10^1 + \underline{3} \times 10^0$$

となるので，表現が “1203” なのです．

これに対し，コンピュータ内部で使われている 2 進数は，2 のべき乗を基本として表された数です．この場合，多項式の係数は 2 より小さい整数でなければならないので，0 か 1 です．例えば，10 進数で 123 と表示される数を 2 の多項式で表せば，

$$123 = \underline{1} \times 2^6 + \underline{1} \times 2^5 + \underline{1} \times 2^4 + \underline{1} \times 2^3 + \underline{0} \times 2^2 + \underline{1} \times 2^1 + \underline{1} \times 2^0$$

となるので，2 進数で表すと，1111011 となります．2 進数は 0 または 1 だけで表すことができるので，電子回路の ON/OFF 回路を使って実現することができます．これがコンピュータで 2 進数が使われている理由です．2 進数の 1 桁を **ビット** (bit) といいます．現在のコンピュータは 2 進数 8 桁 (8bit) を基本

単位として作られていて，これを **バイト** (byte) といいます．1byte は 8 桁の 2 進数ですから $2^8 = 256$ 個の数字を取り扱うことができます．これを 0 以上の整数と考える場合には，0〜255 となります．

さて，Fortran の整数型は 4byte なので，32 桁 (32bit) の 2 進数です．すなわち，2^{32} 個の数字が取り扱えます．しかし負の整数を含ませるために，整数型の範囲は -2^{31}〜$2^{31}-1$ になります．2^{31} は 2,147,483,648 ですから，取り扱える数の上限はおよそ 21 億です．整数型の計算をするときは，絶対値がこれを超えないようにしなければなりません．

これに対し，実数型数は数値のビットを仮数部 f と指数部 e，および正負を決める符号ビットに分割して，$\pm f \times 2^e$ のような形式で表現します．指数部 e は負数も含んでいるので，10^{-19} のような絶対値の小さい数も表現できます．実数型におけるビット分割とそれによる数の表現には規格がいくつかありますが，現在最もよく使われているのは IEEE 754 と呼ばれている規格です．IEEE 754 における実数型のビット分割数と，それによる表現可能な絶対値の上限を表 1.4 に示します[12]．

表 1.4　実数型のビット分割数と表現可能な絶対値の上限 (IEEE 754)

数値型 (bit)	符号ビット	指数部ビット	指数部 e の範囲	仮数部ビット	絶対値の上限
単精度 (32)	1	8	−126〜127	23	3.4×10^{38} 程度
倍精度 (64)	1	11	−1022〜1023	52	1.7×10^{308} 程度
4 倍精度 (128)[13]	1	15	−16382〜16383	112	1.1×10^{4932} 程度

例えば，倍精度実数型は仮数部が 52bit なので，$2^{52} \fallingdotseq 4.5\times10^{15}$ 個の区別ができ，これが “有効数字 15 桁程度”の根拠です．また，指数部の最大が 1023 なので，最大 10^{308} 程度まで表現することができます．これに対し，単精度は指数部の最大が 127 なので，最大は 10^{38} 程度です．10^{38} のような大きな数字を扱うことはあまりないかもしれませんが，計算の途中で現れる可能性は考慮する必要があります．例えば，プランク定数は $h \fallingdotseq 6.6\times10^{-34}$ Js なので，h の逆 2 乗が公式の中に入っていると，途中計算で 10^{38} を越えてしまいます．倍精度実数を使う意義はここにもあります．

なお，IEEE 754 での指数部には 2^e という以外にも意味があり，数字の組み合わせによっては，**無限大** や **非数** を表します．もし，`print` 文で数値を出力したときに，“Infinity”や “INF”という文字が出力された場合には無限大です．つまり，計算した結果が表現できる上限を超えたという意味です．これを **オーバーフロー** といいます．例えば，1 を 0 で割ったときに発生します．

これに対し，“NaN”と出力された場合は非数です．非数 (Not a Number) というのは，「数字じゃない」という意味ですが，具体的には，0 を 0 で割ったときや，−1 の平方根を計算したときの結果がこれに相当します．ただし，結果が非数になっても計算が継続することもあるので，どの時点で発生したのかを特定するのは結構面倒です．

[12] IEEE 754 に関しては「http://ja.wikipedia.org/wiki/浮動小数点数」を参考にしました．詳しくは，この Web ページを見て下さい．なお，表現できる正数の下限は，およそ「1/上限数」程度，という見当で良いと思いますが，IEEE 754 の規格ではもう少し小さい桁の数まで表せます．もっとも，実際にパソコンで上限と下限をチェックしてみたところ，下限は動作環境やコンパイラによって異なりました．また，4 倍精度に関しては上限が倍精度と同じ 10^{308} 程度しかないものもありました．すなわち，必ずしも IEEE 754 の規格通りに実装されているとは限らないようです．

[13] 4 倍精度実数型に関しては，付録 D 参照．

演 習 問 題 1

(1-1) 2 次方程式の解

3 個の実変数 a，b，c に適当な値を代入し，それから，

$$ax^2 + bx + c = 0$$

の 2 解を計算して出力するプログラムを作成せよ．さらに，その解を $ax^2 + bx + c$ に代入して 0 に近い値になることを確かめよ．また，a，b，c の値が不適切だと，エラーが出ることも確認せよ．(例えば，$a = b = c = 1$ にしてみよ)

(1-2) ヘロンの公式

三角形の 3 辺の長さを a，b，c とすると，その三角形の面積 S は次式 (ヘロンの公式) で表される．

$$S = \sqrt{p\,(p-a)(p-b)(p-c)}$$

ただし，$p = \dfrac{a+b+c}{2}$ である．

a，b，c に適当な数値を与え，この公式を使って三角形の面積を計算するプログラムを作成せよ．例えば，$a = 3$, $b = 4$, $c = 5$ の直角三角形の面積がヘロンの公式から求めた面積と一致するかどうか確かめよ．

(1-3) 面積，体積

1 個の実変数 a と整変数 n に適当な数値を与え，これから，半径 a の円の面積 S，半径 a の球の体積 V，1 辺の長さ a の正 n 角形の面積 D を計算するプログラムを作成せよ．ここで，S，V，D は以下の公式で与えられる．

$$S = \pi a^2$$
$$V = \frac{4}{3}\pi a^3$$
$$D = \frac{na^2}{4\tan(\pi/n)}$$

ただし，π の値はできるだけ正確な値 (有効数字 15 桁程度) を使うこと．

(1-4) 複素数の n 乗根

複素数 z の n 乗根は，次のようにして計算することができる．まず，z の絶対値を r，偏角を θ とすると，

$$z = r(\cos\theta + i\sin\theta)$$

と表すことができるので，z の 1 個の n 乗根は，

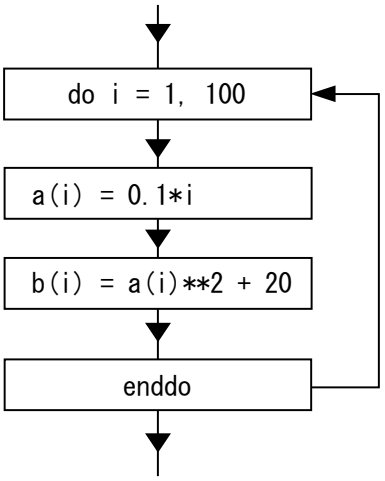
$$z^{1/n} = r^{1/n}(\cos\frac{\theta}{n} + i\sin\frac{\theta}{n}) = r^{1/n}w_1, \qquad \text{ただし，} w_1 = \cos\frac{\theta}{n} + i\sin\frac{\theta}{n}$$

である．$r^{1/n}$ と w_1 の計算結果を使えば，$r^{1/n}w_1$，$r^{1/n}w_1^2$，$r^{1/n}w_1^3$，$\cdots$，$r^{1/n}w_1^n$，という n 個の複素数が z の n 乗根となる．なお，z が正の実数のときには $\theta = 2\pi$ とする．

例えば，2 の 5 乗根を全て計算せよ．

第 2 章　配列と手順のくり返し

前章で説明した基本的プログラミングは，単純な計算動作を並べたものであり，言ってみれば関数電卓の操作をプログラムという書式で表しただけに過ぎません．コンピュータの能力を使いこなすには，同じような動作を何度もくり返したり，条件に応じて異なる動作をさせる手法の知識が必要です．本章では複数の数値をまとめて表現するための“配列”と，それを有効に利用するためのくり返し動作の書き方について説明します．



2.1　配列

これまで利用してきた“変数”は，型に応じた 1 個の数値を記憶することしかできませんでした．しかし，数値計算やシミュレーションでは，数万個のデータを保存して，それぞれを条件に応じて変化させていく，なんていうのが当たり前のように行われます．そこで，数学で $a_1, a_2, \cdots, a_n$ のように変数に添字を付けて区別するように，数字で区別した変数を作ることができます．これを **配列** といいます．本節では配列の使い方について説明します．

2.1.1　配列宣言

配列は変数の一種なので，型宣言文を使って宣言しておかねばなりません．単一変数と異なるのは，宣言時に添字の範囲を示す整数値をかっこを使って付加することです．例えば，次のように宣言します．

```
real a(10),b(20,30)
complex cint(10,10)
integer node(100)
```

ここで，a や node のように数字が 1 個の配列を 1 次元配列，b や cint のように数字が 2 個の配列を 2 次元配列といいます．3 次元以上の配列を作ることもできます．配列の名前の付け方，頭文字の選択などの規則は単一変数名の付け方と同じにします．宣言した配列の各部の名称を表 2.1 に示します．

表 2.1　配列の各部の名称

real a(10) と宣言した配列について	
記　号	名　称
a	配列名
a(3) など	配列要素
かっこ中の数字	要素番号，添字

配列宣言で指定した数値は要素番号の最大値を表し，要素番号の使用可能範囲は 1 から指定した最大値までです．例えば a(10) の宣言では a(1) から a(10) までの 10 個の配列要素が使用可能になります．また，2 次元以上の配列の場合には各次元ごとの最大値を指定しているので，b(20,30) の宣言では全部で $20 \times 30 = 600$ 個の配列要素が使用可能になります．

問題によっては，要素番号として 0 や負数を使いたい時があります．このような時には“:”を間に入れて，使用可能な要素番号の最小値と最大値を同時に指定します．例えば，

```
real ac(-3:5),bc(-20:20,0:100)
```

と宣言すると，1 次元配列 ac は，ac(-3) から ac(5) までの 9 個が使用可能であり，2 次元配列 bc は，bc(-20,0) から bc(20,100) までの $(20 \times 2 + 1) \times (100 + 1) = 4141$ 個が使用可能です．

2.1.2 メモリ上での配列の並び

配列はコンピュータ内部において連続したメモリ領域で実現されています．例えば，“real a(10)”と宣言された配列は，図 2.1 左のように，a(1),a(2),...,a(10) の順で並んだ実数型のメモリです．

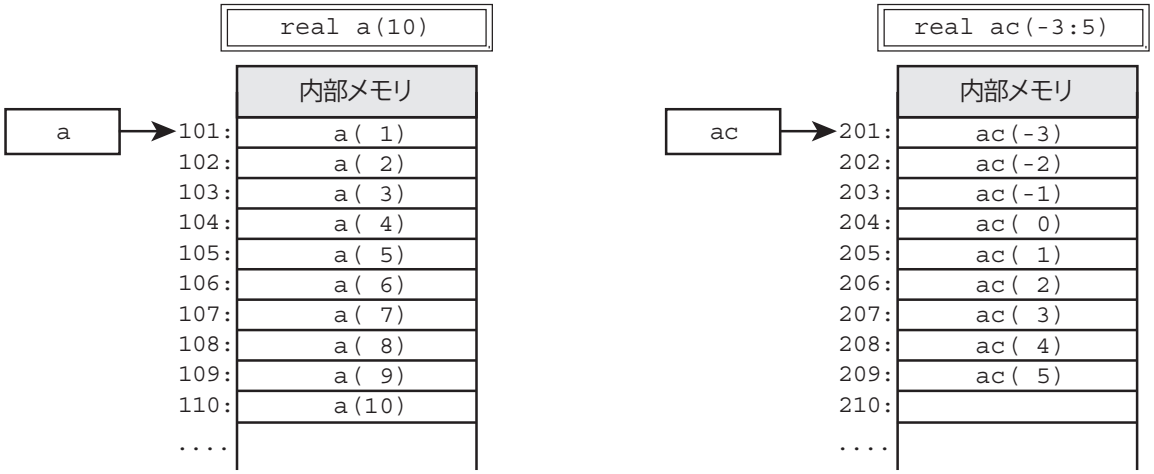


図 2.1　1 次元配列のメモリ並び

図からわかるように，a(3) とは a(1) から数えて 3 番目のメモリのことです．また，“real a(10)”の宣言では 10 個しかメモリを確保していないのですから，a(10000) のように範囲外の要素番号を指定すると問題が起こります．10000 番目の要素 a(10000) がどのメモリを示すのか不明だし，そもそも存在するという保証さえありません．

Fortran での配列名は配列を代表する名称であると同時に，配列の先頭メモリを示します．例えば，配列名 a は図 2.1 左のように a(1) を示します．また，ac(-3:5) のように最小値も指定して宣言した場合には，図 2.1 右のように並んでいて，配列名 ac は ac(-3) を示します．配列名が先頭要素を示すことは配列をサブルーチンの引数として与えるときに意識する必要があります[14]．

2 次元以上の配列の場合は，左の方の要素番号から先に進むようにメモリ上で並んでいます[15]．例えば，

```
real b(3,2)
```

と宣言した場合，メモリ上での並びは図 2.2 のようになります．2 次元以上の配列も配列名は先頭要素を示しています．配列 b の場合，配列名 b は b(1,1) を示します．

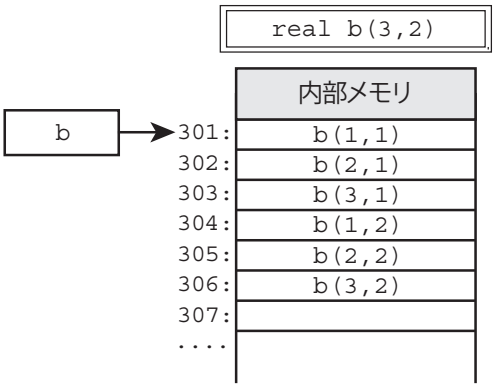


図 2.2　2 次元配列のメモリ並び

2 次元と 3 次元の配列の並び方を式で表せば，

[14]配列をサブルーチンの引数にする方法は 4.5 節で説明します．
[15]C 言語をご存じの方は注意して下さい．C 言語では右側の要素番号が先に進むので，進む順番が逆なのです．

```
real b(n1,n2) の配列宣言で，b(i,j) は先頭から数えて i+n1*(j-1) 番目
real b(n1,n2,n3) の配列宣言で，b(i,j,k) は先頭から数えて i+n1*(j-1+n2*(k-1)) 番目
```

となります．どちらの公式も一番右側の要素数に依存していません．一般的に，どんな次元の配列の順番を与える公式も一番右側の要素数には依存しません．

2.2 手順のくり返し — do 文

実行文は基本的に上から下へ順に実行されますが，それだけでは類似した手順をくり返すときに，必要な回数だけ同じような文を書かねばなりません．そこで，ある範囲の手順を必要な回数だけくり返し行わせる手段として **do 文** があります．配列を使って大量のデータを取り扱う場合，a(1)=...，a(2)=...，のように要素ごとの実行文をすべて書くのは大変です．そこで，計算のパターンがわかっている場合には，do 文を使ってそのパターンをくり返す形で配列の計算を行うことができます．

do 文を使うときの基本形は，

```
do 整数型変数 = 初期値，終了値
  .......
  .......
enddo
```

です．最初の do の行が do 文で，do 文と最後の **enddo 文** で範囲を指定した一連の実行文がくり返し実行されます[16]．この範囲を **do ブロック** といいます．また，プログラムの流れが循環するという意味で **do ループ** ともいいます．do 文の整数型変数を，本書では **カウンタ変数** と呼びます．do ブロックのくり返しは，カウンタ変数に“初期値”を代入することから開始し，do ブロック内の手順を 1 回実行するごとにカウンタ変数を 1 増加します．そして，カウンタ変数が“終了値”より大きくなった時点でくり返しは終了し，enddo 文の次の文に実行が移ります．例えば，

```
do m = 1, 10
  a(m) = m
enddo
```

と書けば，a(1)=1，a(2)=2，...，a(10)=10，という 10 個の代入文が順番に実行されます．do 文における，“カウンタ変数＞終了値”の判定は do ブロックの開始時にも行うため，初期値が終了値より大きい場合にはブロック内部が一度も実行されずに enddo 文の次の文に実行が移ります．

次の do 文のように，整数値をもう一つ追加することで増加量を指定することもできます．

```
do 整数型変数 = 初期値，終了値，増分値
  .......
  .......
enddo
```

このときカウンタ変数は初期値から開始して，“増分値”ずつ増加しながら do ブロック内の手順をくり返し，“カウンタ変数＞終了値”の時点で終了します．

例えば，m が 10 以下の奇数のときのみ計算をしたい場合は，

```
do m = 1, 10, 2
  .......
  .......
enddo
```

と書きます．この do 文の終了値は 10 ですが，10 は奇数ではないので計算しません．

[16] enddo は“end do”と離して書いても OK です．

増分値は負数を指定することもできます．負数のときにはカウンタ変数が減少していくので，“カウンタ変数＜終了値”になった時点で終了します．例えば，100 から順に下って 1 までくり返すときには以下のように書きます．

```
do m = 100, 1, -1
  .......
  .......
enddo
```

増分値が負数のときには，“初期値＜終了値”ならば do ブロック内部は一度も実行されません．

do 文のカウンタ変数に増分値を加えるタイミングは，enddo 文の実行時です．例えば，

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

という文は，

```
 m = 1
   [m＞3 の判定をする．満足しないので，do ブロックを実行]
 a(m) = m**2
 m = m + 1
   [m＞3 の判定をする．満足しないので，do ブロックを実行]
 a(m) = m**2
 m = m + 1
   [m＞3 の判定をする．満足しないので，do ブロックを実行]
 a(m) = m**2
 m = m + 1
   [m＞3 の判定をする．満足するので，do ブロックを終了]
```

という動作になります．この展開からわかるように，do ブロックを終了した時点で，カウンタ変数には終了値より大きい値が代入されています．上例では，do ブロック終了後，m＝4 です．do ブロック終了時の m の値を利用するときは，このことを考慮しなければなりません．

次のように，do ブロックの中に別の do ブロックを入れて多重にすることもできます．ただし，カウンタ変数は異なるものを使わなければなりません．これは，do ブロック内でカウンタ変数を変更することが禁止されているからです．

```
do k = 1, 100
  a(k) = k**2
  do m = 1, 10
    b(m,k) = m*a(k)**3
    c(m,k) = b(m,k) + m*k
  enddo
  d(k) = a(k) + c(10,k)
enddo
```

よく使うので覚えておくと便利なのが，合計を計算する do 文です．例えば，n 要素の 1 次元配列，a(1)，a(2)，...，a(n) の中に数値データが入っている場合，これらの合計を計算して変数 sum に代入するには，do 文を使って以下のように書きます．

```
sum = 0
do m = 1, n
  sum = sum + a(m)
enddo
```

この文が合計を計算していることを理解するには，やはり以下のように手動展開してみると良いでしょう．

```
sum = 0
m = 1
sum = sum + a(m)
m = m + 1
sum = sum + a(m)
m = m + 1
sum = sum + a(m)
m = m + 1
 ......
```

ここで判定文は省略しました．このプログラムで重要なことは，do 文の前で変数 sum に 0 を代入していることです．これがないと正しい結果が得られないことがあります．このようなプログラムならではの書き方はパターンとして覚えておくと良いと思います．

なお，カウンタ変数の初期値，終了値，増分値は do ブロックの開始時に決定されるので，do ブロック内部で変更しても無関係です．例えば，

```
j = 10
k = 1
do i = 1, j, k
   print *,i,j,k
   j = 20
   k = 2
enddo
```

のように書いてもエラーではありませんが，終了値 j や増分値 k は最初の do 文を実行した時点の数値でくり返します．

2.3 プログラミング スマートテクニック (その 2)

本節では do ブロックを使うときのテクニックや，プログラムを読みやすくするためのヒントなどを紹介します．配列を使った本格的なプログラムが書けるようになると，「文法的に間違いがなければ良い」という考えで書くのではなく，読みやすいプログラムになるよう心がける必要があります．ここで，“読みやすいプログラム”というのは，チェックがしやすいプログラムのことです．読みやすく書いておけば，書き間違いに気付きやすいし，実行時にエラーが出ても早期に誤りを発見することができます．

2.3.1 do ブロックのテクニック

do ブロック内部に同じ計算をくり返す記述があるときには，あらかじめ計算をしておきます．例えば，

```
do i = 1, 10000
  a(i) = b(i)*c*f**2
  b(i) = sin(x)*a(i)
enddo
```

と書くと，くり返しごとに c*f**2 や sin(x) を計算することになりますが，これらは do ブロック内部では変化しないのですから，あらかじめ計算しておくと速くなります．例えば，

```
cf = c*f**2
sx = sin(x)
do i = 1, 10000
  a(i) = b(i)*cf
  b(i) = sx*a(i)
enddo
```

のように書き換えると速くなります．

また，do ブロック内で何度も同じ割り算をするときには，逆数を掛けるように書き換えると速くなります．これは，割り算が遅いからです．例えば，

```
do i = 1, 10000
  a(i) = b(i)/c
  x(i) = a(i)/10.0
enddo
```

と書くより，

```
d = 1.0/c
do i = 1, 10000
  a(i) = b(i)*d
  x(i) = a(i)*0.1
enddo
```

と書く方が速くなります．もっとも，プログラムがわかりにくくなるという欠点もあるので，割り算の箇所が少なく，くり返し回数がさほど多くない時にはそれほど神経質に変形する必要はありません．

配列を使ったくり返し計算をするときは，メモリをできるだけ連続的に読み書きする方が速くなります．このため，多重 do ブロックを使って 2 次元以上の配列計算をするときは，左の要素から先に進めるようにします．例えば，

```
real b(10,100)
integer m,n
do n = 1, 100
  do m = 1, 10              !  左の要素を先に進めると速い
    b(m,n) = m*n
  enddo
enddo
```

のように，2 次元配列 b(m,n) に計算結果を代入するときは，左側の要素 m に関する do ブロックを右側の要素 n の do ブロックの内側に持ってくる方が高速です．これは，内側の do ブロックのカウンタ変数が先に進むので，b(1,1)=1*1，b(2,1)=2*1，b(3,1)=3*1…のように，メモリの並んでいる順に格納していくからです．

これを，

```
real b(10,100)
integer m,n
do m = 1, 10
  do n = 1, 100             !  右の要素を先に進めると遅い
    b(m,n) = m*n
  enddo
enddo
```

のように書くと，b(1,1)=1*1，b(1,2)=1*2，b(1,3)=1*3…のようにとびとびに格納したり，戻って格納したりするので効率が悪くなり，スピードダウンします．

2.3.2 多項式を計算する手法

多項式を計算するときは，掛け算の回数ができるだけ少なくなるように変形します．一般的に良い計算手法は horner 法です．例えば，

$$y = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4$$

を，乗算を明示してプログラムにすると，

```
y = a0 + a1*x + a2*x*x + a3*x*x*x + a4*x*x*x*x
```

のようになり，10 回の乗算が必要です．ところが，これを変形して，

```
y = a0 + (a1 + (a2 + (a3 + a4*x)*x)*x)*x
```

と書けば，4 回の乗算で計算できます．これが horner 法です．

一般的に，n 次の多項式，

$$y = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

における係数 $a_0, a_1, a_2, ..., a_n$ が，下限が 0 の 1 次元配列，a(0),a(1),a(2),...,a(n) にそれぞれ代入されている場合には，次のような do 文を使って計算することができます．

```
y = a(n)
do i = n-1, 0, -1
  y = a(i) + x*y
enddo
```

なお，$x^8 + 1$ は $((x^2)^2)^2 + 1$ のように変形すれば 3 回の乗算で計算できます．すなわち，多項式によっては horner 法より速く計算できる手順が存在することもあります．

2.3.3 ブロックを明確にするために字下げする

これまでのプログラム例でも示していますが，do や第 3 章で説明する if などのブロック内では **字下げ** (インデント) をしましょう．これは，ブロックの範囲が一目でわかるからです．例えば，

```
do i = 1, n
  b(i) = a(i)
  c(i) = a(i)*sin(b(i))
enddo

do i = 1, n
  do k = 1, 10
    d(k,i) = k*b(i)
  enddo
enddo
```

のように，do ブロック内部の文をスペースを入れて右にずらせておくとブロックの範囲が一目で判断できます．通常，2〜4 個のスペース程度でよいと思います．

最近のエディタには，**オートインデント** といって，改行するとその前の行と先頭がそろうようにカーソルが移動する機能があるので，インデントをするのはそれほどの手間ではありません．

2.3.4 文字間や行間は適当に空ける

文中の文字間は適当に空けたほうが読みやすくなります．筆者は “=” と “+” の両側に必ずスペースを入れることにしています．また，代入文が何行も続くときには “=” を上下にそろえると読みやすくなります．特に，同じパターンで少しずつ内容が違う文を並べるときには，パターンが並ぶようにスペースを入れることをお勧めします．以下は，筆者のシミュレーションプログラムの一部です．

```
wm( 1,m1) = (1-dx)*(1-dy)*(1-dz)
wm( 2,m1) =    dx *(1-dy)*(1-dz)
wm( 3,m1) = (1-dx)*   dy *(1-dz)
wm( 4,m1) =    dx *   dy *(1-dz)
wm(11,m1) = (1-dx)*(1-dy)*   dz
wm(12,m1) =    dx *(1-dy)*   dz
wm(13,m1) = (1-dx)*   dy *   dz
wm(14,m1) =    dx *   dy *   dz
```

これをスペース抜きで書くと，以下のようになります．

```
wm(1,m1)=(1-dx)*(1-dy)*(1-dz)
wm(2,m1)=dx*(1-dy)*(1-dz)
wm(3,m1)=(1-dx)*dy*(1-dz)
wm(4,m1)=dx*dy*(1-dz)
wm(11,m1)=(1-dx)*(1-dy)*dz
wm(12,m1)=dx*(1-dy)*dz
wm(13,m1)=(1-dx)*dy*dz
wm(14,m1)=dx*dy*dz
```

この二つは全く同じことが書いてあるのですから全く同じ結果を出します．しかし，パターンをそろえた前者は単なる美的趣味でスペースを入れたのではありません．この例では，どこか 1 カ所，例えば dx を dy に書き間違えただけでも正しい結果が得られません．しかも，dx と dy を書き間違えても文法的なミスにはならず，出力結果を見ても間違いに気づかない可能性があります．よって，できるだけプログラムを書いている段階でミスを取り除いておかねばなりません．

この時，前者のようにパターンをそろえておけば，横方向に 1 行 1 行チェックするだけでなく縦方向にも比較しながらチェックをすることができます．色々な角度からチェックすることで，ミスのないプログラムにすることができるのです．

この他，文と文の間に，適当なコメント文や何も書いていない行を挿入すれば，プログラムの流れに区切りができて読みやすくなります．これは文章を段落に分けるように，プログラムを区分けすると考えればいいでしょう．

2.3.5 広い範囲で共有する定数は意味のある名前をつけた変数に代入して使う

do ブロックを 100 回くり返す，とか，10 回ごとに出力する，とかいう制御数は，できるだけ変数にしましょう．変数にしておけば数値の意味が明示されるし，変更もしやすくなります．例えば，

```
do i = 1, 100                    !  100 が 1 個
  a(i) = b(i)**2
enddo
sum = 0
do i = 1, 100                    !  100 が 1 個
  sum = sum + a(i)
enddo
ave = sum/100                    !  100 が 1 個
```

というプログラムにおいて，100 という数は配列要素数に依存する共通の数値ですから，整数変数 imax を用意して，以下のように書き換えます．

```
imax = 100                       !  100 を変更するときはここだけで OK
do i = 1, imax
  a(i) = b(i)**2
enddo
sum = 0
do i = 1, imax
  sum = sum + a(i)
enddo
ave = sum/imax
```

imax は “ i の最大値 ” という意味を込めた名前です．こうしておけば，総数 100 を変更したい時には imax=100 の数値を変えるだけで完了です．このような変数化はプログラムが長くなるほど価値が増します．ただし，大域的に利用する変数ですから，1.7 節で説明したように長めで意味のある名前を付けるようにしましょう．

演 習 問 題 2

(2-1) 3 次元ベクトル演算

3 次元ベクトル $\boldsymbol{A} = (A_1, A_2, A_3)$ を要素数 3 の 1 次元配列 `a(3)` で表すとする．まずベクトル $\boldsymbol{A}$ とベクトル $\boldsymbol{B}$ を配列で表してそれぞれに適当な数値を代入し，2 個の配列から，$\boldsymbol{A}$ と $\boldsymbol{B}$ の内積 $\boldsymbol{A}\bullet\boldsymbol{B}$，および外積ベクトル $\boldsymbol{A}\times\boldsymbol{B}$ を計算するプログラムを作成せよ．外積ベクトルも配列にすること．ただし，ベクトル $\boldsymbol{A}$ と $\boldsymbol{B}$ の内積，および外積ベクトルは以下の公式で与えられる．

$$\boldsymbol{A}\bullet\boldsymbol{B} = A_1B_1 + A_2B_2 + A_3B_3$$
$$\boldsymbol{A}\times\boldsymbol{B} = (A_2B_3 - A_3B_2, A_3B_1 - A_1B_3, A_1B_2 - A_2B_1)$$

(2-2) 2 行 2 列の行列計算

2 行 2 列の行列，$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$ を，2 次元配列 `a(2,2)` で表すとする．2 個の 2 行 2 列の行列 A と B を表現する配列，`a(2,2)` と `b(2,2)` の各要素に適当な数値を代入して，A と B の足し算の行列 $C = A + B$，掛け算の行列 $M = AB$，および，それぞれの行列式 D_A，D_B を計算するプログラムを作成せよ．ここで，$D_A = a_{11}a_{22} - a_{12}a_{21}$ である．

(2-3) くり返し出力

i=1,2,3,...,n の整数に対し，i，i^2，$1/i$，$\sqrt{i}$ の値を並べて出力するプログラムを作成せよ．なお，整数と実数の変換に注意すること．

(2-4) 漸化式と統計計算

次の漸化式で与えられる数列を，それぞれ，配列 `a(n)`，`b(n)` に代入するプログラムを作成せよ．

$$(1)\ \ a_{i+1} = 3a_i + 2, \qquad a_1 = -1$$
$$(2)\ \ b_{i+1} = \frac{2}{3}b_i - 4, \qquad b_1 = 3$$

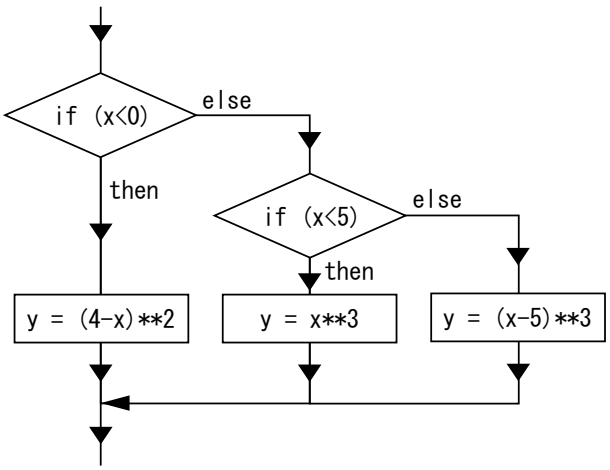
ここで，最大値 n は適当に決めよ．

次に，それぞれの配列に代入された数値を使って，それらの平均と標準偏差を計算せよ．ここで，n 個のデータ，A_1，A_2，...，A_n の平均と標準偏差の定義は以下の通りである．

$$\text{平　均}\ \bar{A} = \frac{1}{n}\sum_{i=1}^{n} A_i$$
$$\text{標準偏差}\ \sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(A_i - \bar{A})^2}$$

標準偏差の定義の中に平均が入っているため，まず平均を計算し，その後で標準偏差を計算するという手順にすること．

第 3 章　条件分岐とジャンプ



手順のくり返しを記述する do 文と並んで重要なのが，条件に応じて異なる動作をさせるための if 文です．do 文と if 文の使い方を習得すれば，どんなプログラムでも書けるといっても過言ではありません．本章では，if 文による条件分岐の書き方と，必要に応じて実行の流れを強制的に変えるジャンプについて説明します．

3.1　条件分岐 — if 文

条件に応じて異なる手順を行わせることを **条件分岐** といい，**if 文** を使って指定します．最も単純に，一つの条件に応じて一つの文を実行するかしないかを決めるだけの時は **単純 if 文** を使います．単純 if 文は以下の形式です．

```
if (条件) 実行文
```

単純 if 文はかっこ内の “条件”が満足されれば，その右の “実行文”を実行し，条件が満足されなければ何もしないで次の文に実行が移ります．例えば，

```
a = 5
if (i < 0) a = 10
b = a**2
```

と書くと，i < 0 の場合には a = 10 となり，それ以外は a = 5 のままなので，それに応じて b に代入される値が異なります．

しかし単純 if 文には実行文が 1 個しか書けないので，実行したい文が複数あるときには使えません．また，条件に合った時の動作指定しかできないので，合わなかった時の動作を別に指定したい時には不便です．そこで，通常は単純 if 文ではなく，**ブロック if 文** を使います．ブロック if 文とは，if 文の実行文のところを **then** にした文のことで，以下のようにブロック if 文と **endif 文** で一連の実行文の範囲を指定します[17]．

```
if (条件) then
  .......
  .......
endif
```

この指定された範囲を **if ブロック** といい，ブロック if 文に書かれた条件を満足したときのみ，if ブロック内の実行文が実行されます．例えば，

```
a = 5
b = 2
if (i < 0) then
   a = 10
   b = 6
endif
c = a*b
```

と書くと，i < 0 の場合には a=10，b=6 の代入文を実行してから c=a*b を計算しますが，それ以外，すなわち i ≧ 0 の場合には if ブロック内を実行しないので，a も b も変化せず，a=5，b=2 のままで c=a*b

[17] endif は “end if”と離して書いても OK です．

を計算します．

さて，この例においては，a=5，b=2 という代入は i < 0 という条件を満足するときには不要で，満足しないときのみ実行すれば十分です．このように“条件を満足しない場合”に別の動作をさせたいときには if ブロック内に **else 文** を挿入します．else 文を挿入すると，ブロック if 文で指定した条件を満足しない場合に，else 文から endif 文までの実行文が実行されます．例えば上記のプログラムは，

```
if (i < 0) then
   a = 10
   b = 6
 else
   a = 5
   b = 2
endif
c = a*b
```

と書くことができます．この if ブロックでは，i < 0 の場合には a=10，b=6 を実行，それ以外の場合には a=5，b=2 を実行します．

また，条件を満足しない場合に，さらに別の条件を指定したいときには **else if 文** を使います．else if 文も条件はかっこで指定し，その後に then を書きます．

```
if (i < 0) then
   a = 10
   b = 6
 else if (i < 5) then
   a = 4
   b = 7
 else
   a = 5
   b = 2
endif
c = a*b
```

この場合，i < 0 の場合には a=10，b=6 を実行，0 ≦ i < 5 の場合には a=4，b=7 を実行，それ以外 (i ≧ 5) の場合は a=5，b=2 を実行，となります．else if 文による新たな条件は if ブロック内にいくつ入れてもかまいません．その場合は，“その else if 文より以前の条件を全て満足しない場合に，その条件を満足すれば”という意味になります．これに対し，else 文は最後の 1 回しか使えません．

数値計算でよく使う代表的な比較条件の書き方を表 3.1 に示します．

表 3.1　比較条件の書き方

比較条件記号	記号の意味	使用例
==	左辺と右辺が等しいとき	x == 10
/=	左辺と右辺が等しくないとき	x+10 /= y-5
>	左辺が右辺より大きいとき	2*x > 1000
>=	左辺が右辺以上のとき	3*x+1 >= a(10)**2
<	左辺が右辺より小さいとき	sin(x+10) < 0.5
<=	左辺が右辺以下のとき	tan(x)+5 <= log(y)

使用例のように，比較条件を指定する場合には両辺どちらにも計算式を書くことが可能です．

さらに表 3.2 の論理演算記号を使えば，これらの条件を論理的につないだ条件や，否定した条件を与えることもできます．

表 3.2　論理式の書き方

論理演算記号	演算の意味	使用例
“条件 1”.or.“条件 2”	“条件 1”または“条件 2”のとき	x > 0 .or. y > 0
“条件 1”.and.“条件 2”	“条件 1”かつ“条件 2”のとき	x > 10 .and. 2*x <= 50
.not.“条件”	“条件”を満足しないとき	.not.(x < 0 .and. y > 0)

例えば,

```
if (i > 0 .and. i <= 5) then
   a = 10.0
 else
   a = 0.0
endif
```

と書くと，i が 0 より大きく “かつ” 5 以下の時，すなわち，$0 < \mathtt{i} \leqq 5$ のときに $\mathtt{a} = 10$，それ以外は $\mathtt{a} = 0$ となります．なお，横着してこのブロック if 文を,

```
if (0 < i <= 5) then      !  これはエラー
```

と書くことはできないので注意しましょう．

論理演算は，数値演算のように連ねて書くことも可能ですが，その際は表 3.2 の下に行くほど優先順位が高くなります．

```
      .not.  ＞  .and.  ＞  .or.
```

例えば,

```
if (i > 100 .or. i > 0 .and. i <= 5) then
```

という if 文は,

```
if (i > 100 .or. (i > 0 .and. i <= 5)) then
```

と同等です．もっとも，論理演算の優先順位は数値演算の優先順位ほどなじみがないので，後者のようにかっこを利用して明示する方が良いと思います．

do 文でくり返し計算をする場合，“n 回ごとに出力する”というように，一定間隔で動作を変更したいことがあります．この場合には，表 1.3 に示した剰余を計算する組み込み関数 mod を利用して条件分岐をします．例えば，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．

3.2　無条件ジャンプ — goto 文，exit 文，cycle 文

1.1 節で説明したように，プログラムというのは基本的に上から順に実行していくものです．do 文を使えば，do ブロックで指定した範囲の実行文を所定の回数くり返すことができますが，くり返す範囲は

固定されているし，くり返しを終了する条件はカウンタ変数の増加で決まります．

これに対し，より一般的にプログラムの流れを変えたい時，例えば途中で計算を中断してプログラムの最初からやり直す，とか，最後の文に一気に移動して終了する，とかいう時には **goto 文** を使います．goto 文を使えば，指定した行へ強制的に移動することができます．計算機的には，この移動のことを **無条件ジャンプ**，あるいは単に **ジャンプ** といいます．

goto 文とは，以下のように goto の後に **文番号** と呼ばれる整数を指定した文です．

```
   goto 文番号
```

これに対し，この goto 文でジャンプしたい先の行には，以下のように実行文の前にスペースを 1 個以上空けて文番号を書きます．

```
   文番号  実行文
```

goto 文を使ったプログラム例を以下に示します．

```
   cd = 10
   goto 11                ！  文番号 11 の行へジャンプ
   cd = 50
   ab = 20
   ij = 1
11  ab = 1000             ！  この行へジャンプ
```

最後の行で ab=1000 の前に書かれた 11 が文番号です．この例では，最初の cd=10 の実行後，cd=50 から ij=1 までの文は実行されず，直ちに ab=1000 が実行されます．すなわち，cd=50，ab=20，ij=1 の文は書いていないのと等価です．

goto 文でバックすることも可能です．例えば，次のように書けば，指定した文番号 22 の行と goto 文の間の動作をくり返し実行します．

```
   cd = 50
22  ab = 200              ！  この行へジャンプ
   cd = cd + ab - ef
   ef = 10
   goto 22                ！  文番号 22 の行へジャンプ
   ij = 1
```

もっとも，この例ではいつまでたっても goto 文の次の ij=1 は実行されません．こういうのは **無限ループ** と呼ばれ，プログラムエラーの一つです．計算結果に応じて条件分岐をし，goto 文より下の行へジャンプする別の goto 文や，プログラム自体を終了させる stop 文を挿入しなければ，プログラムは永遠に終了しません．

文番号は行の指定に使うだけなので，重複さえしなければどの行にどんな数値を付けても良いのですが，なるべく下に行くほど大きくなるように数値を選びます．また，文番号を特定の実行文に付けると，変更するときに付け替えが面倒だと思う時には **continue 文** に付けて挿入します．continue 文に動作は無く，文番号で位置を指定するためだけに用います．例えば，上記のプログラムは，次のプログラムと等価です．

```
   cd = 50
22  continue              ！  この行は動作がない
   ab = 200
   cd = cd + ab - ef
   ef = 10
   goto 22
   ij = 1
```

goto 文は，多用するとプログラムの流れがわかりにくくなるし，無限ループに陥る可能性もあるので使用はできるだけ避けるべきです．基本的なくり返しや，条件に応じたジャンプは do ブロックと if ブロックでほとんどすべて書くことができます．

do ブロックを使ってくり返し計算をするとき，条件によって途中でくり返しを終了したい場合があります．この場合，原理的には goto 文を使わなければなりませんが，goto 文の使用を極力避けるという方針から，**exit 文**が用意されています．do ブロック中で exit 文を実行すると，その do ブロックの外に飛び出して enddo 文の直後から実行を継続します．

例えば，要素数 n の配列 a(n) に代入されている値を条件付きで平均するため，

```
  sum = 0
  do m = 1, n
    sum = sum + a(m)
    if (sum > 100) exit
  enddo
  ave = sum/n
```

のように書いたプログラムは，次の goto 文を使ったプログラムと等価です[18]．

```
  sum = 0
  do m = 1, n
    sum = sum + a(m)
    if (sum > 100) goto 10
  enddo
10 ave = sum/n
```

なお，do ブロックの中から外へのジャンプはできますが，外から中に入るジャンプは禁止されています．

do ブロックの途中で実行を中断し，残りの部分をスキップしてカウンタ変数を進める時には**cycle 文**を使います．例えば，

```
  do m = 1, n
    sum = sum + a(m)
    if (sum > 100) cycle
    sum = sum*2
  enddo
```

のように書いたプログラムは，次の goto 文を使ったものと同じです．

```
  do m = 1, n
    sum = sum + a(m)
    if (sum > 100) goto 10
    sum = sum*2
10 enddo
```

すなわち，cycle 文の実行は enddo 文にジャンプするのと等価です．enddo 文にジャンプすれば，カウンタ変数 m を増加して，m＞n かどうかをチェックした後，条件を満足しなければ再び do ブロックを最初から実行することになります．ただし，cycle 文は do ブロック内部の制御なので次のように if ブロックを使って同じ動作をさせることができます．

```
  do m = 1, n
    sum = sum + a(m)
    if (sum <= 100) then
      sum = sum*2
    endif
  enddo
```

[18] このプログラムを修正して，合計した要素 a(1)〜a(m) の平均値を計算したい場合には，単に n の代わりに m で割るように書き換えるだけでは不完全です．なぜだか考えてみましょう．

この方が，条件に応じた動作を明示している点でよいでしょう．goto 文を多用しない方がよい，という方針で考えれば，cycle 文も多用しない方がよいと思います．

なお，do ブロックが多重の場合，exit 文や cycle 文はその文を含む最も内側の do ブロックに対しての動作になります．より外側の do ブロックの外に抜け出たい場合には，do 文に **ラベル** を付けて exit 文や cycle 文のジャンプ先を指定します．

ラベルを付けるには，do 文と enddo 文の 2 箇所に **ラベル名** を付加します．例えば，

```
   sum = 0
out: do m = 1, n
     do l = 1, n
       sum = sum + a(l,m)
       if (sum > 100) exit out
     enddo
   enddo out
   ave = sum/n
```

のように書くことができます．“out”がラベル名です．do 文に付加するラベルは，ラベル名の最後にコロン (:) を付けて，do の前に書きます．これに対し，enddo 文に付加するラベルは，対応する do 文と同じラベル名を enddo の後ろにスペースを入れて書きます．コロンは不要です．このプログラムは，

```
   sum = 0
   do m = 1, n
     do l = 1, n
       sum = sum + a(l,m)
       if (sum > 100) goto 10
     enddo
   enddo
 10 ave = sum/n
```

と等価です．なお，複数の do 文にラベルを付けるときは，ラベル名が重複しないようにして下さい．

3.3 プログラミング スマートテクニック (その 3)

本節では条件判断によりくり返しを制御する do while 文や，if 文を使うときの注意点について説明します．また，条件分岐をより多様に記述するための論理型データやビット操作関数についても説明します．

3.3.1 do while 文と無条件 do 文

do 文には，2.2 節で述べた基本形の他に，条件でループの動作を続けるか終了するかを決める形式も用意されています．これが **do while 文** です．do while 文を用いた do ブロックは以下のような書式です．

```
   do while (条件)
     .......
     .......
   enddo
```

この場合，最初の do while 文の“条件”を満足しなくなるまで，その do ブロック内部をくり返し実行します．もし条件を満足しなければ，do ブロックは終了して enddo 文の次に実行が移ります．

例えば，

```
   integer n
   n = 100
   do while (n > 0)
     n = n/2
     print *,n
   enddo
```

と書けば，n を 2 で割っていって，0 になった時点でループが終了します．ここで，n を整数型にしているところがポイントです．実数型だとループがいつ終了するかわかりません．

また，“do のみ”の do 文，**無条件 do 文**もあります．この場合，do ループを終了させる条件がないので，do 文と enddo 文で指定したブロックをいつまでもくり返します．例えば，

```
do
  sum = sum + x**2
  if (sum > 100) exit
  x = 1.2*x + 0.5
enddo
```

と書くと，sum が 100 を超えるまで計算を続けます．無条件 do 文を使うときは，ブロック内部に適切な条件分岐によって外に出る記述が必要です．さもなくば無限ループになるので注意して下さい．

3.3.2 0.1 を 10 回足しても 1 に等しくならない

実数型を使って条件分岐をするときに注意しなければならないことがあります．次のプログラムを考えてみましょう．

```
real x
integer m
x = 0.0
do m = 1, 20
  x = x + 0.1                !  0.1 をくり返し加える
  if (x == 1.0) exit
enddo
print *,'x = ',x
```

この do ループは，実数型変数 x に 0.1 をくり返し加えていきますが，ループの 10 回目で if 文の条件「x が 1.0 に等しい」を満足し，exit 文によりループの外に出て，print 文の出力は 1.0 になると予想されます．しかし，実際にやってみるとわかりますが，print 文の出力は 2.0 です．つまり，if 文の条件は満足しない，すなわち，x が 1.0 に等しくなることはないのです．なぜでしょうか．

答えは，コンピュータが 2 進数で計算しているためです．1.11.4 節で述べたように，10 進数で 123 と表現される数を 2 の多項式で表せば，

$$123 = 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

となり，これが 2 進数で表したときに 1111011 となる理由でした．コンピュータの数値は全て 2 進数なのですから，小数点以下の数値も 2 進数で表さねばなりません．例えば，10 進数の 0.123 は

$$0.123 = 1 \times 10^{-1} + 2 \times 10^{-2} + 3 \times 10^{-2}$$

のように，10 の負べき乗を基本として表された数のことですが，これと同様に，2 進数で小数点以下の数 x を表すには，

$$x = b_1 \times 2^{-1} + b_2 \times 2^{-2} + b_3 \times 2^{-2} + \cdots$$

となるような 0 または 1 の係数 $b_1, b_2, b_3, \cdots$ を選んで，

$$0.b_1 b_2 b_3 \cdots$$

と並べます．コンピュータはこの係数 $b_1, b_2, b_3, \cdots$ で小数点以下を表現しています．

さて，コンピュータのメモリで表現できる 2 進数の桁 (ビット数) には限りがあるので，小数はどこかで打ち切る必要があります．ところが，2 進数の場合，有限小数で表せるのは，2^{-n} の和だけです．例えば，0.5 ($=2^{-1}$) や 0.125 ($=2^{-3}$) やその和である 0.625 は有限の 2 進小数で表すことができますが，2^{-n}

の和で表せない数値は無限小数になってしまいます．例えば，10 進数の 0.1 は，

$$0.1 = 0\times2^{-1} + 0\times2^{-2} + 0\times2^{-3} + 1\times2^{-4} + 1\times2^{-5} + 0\times2^{-6} + 0\times2^{-7} + 1\times2^{-8} + 1\times2^{-9}\cdots$$

となり，2 進数で表せば，0.0001100110011... という循環小数になります．よって，コンピュータにおける 0.1 は近似値でしかなく，10 回加えても 1 に等しくなりません．

このように，実数計算をするときは 10 進数と 2 進数の違いを考慮してプログラムを書かないと，予想外の結果になることがあります[19]．また，同じ少数点以下の数をくり返し加えるときには，回数を掛け算した方が精度が上がります．例えば，

```
real x
integer m
x = 0.0
do m = 1, 100
  x = x + 0.1
   .......
enddo
```

と書くよりも，

```
real x
integer m
do m = 1, 100
  x = 0.1*m
   .......
enddo
```

と書く方が，x の精度は上がります．

3.3.3 非数（NaN）のチェック

1.11.4 節で，負数の平方根を計算したり，0 を 0 で割ったときには **非数** (Not a Number，NaN) になる，という話をしました．非数が出たらそれ以上計算することは無意味なので，どこで発生したかをつきとめる必要がありますが，計算結果が非数になった時点で計算が終了するとは限らないので，発生場所を突きとめるのは結構やっかいです．非数は普通の数値と異なるので，「NaN に等しい」という条件を書くことはできませんが，面白いことに，「どんな数値とも等しくならない」という性質があるため，次のような if 文でチェックをすることができます．

```
x = -1
x = sqrt(x)
if (x /= x) print *,'x = NaN'
```

すなわち，「どんな数値とも等しくならない」ため，自分自身とも等しくならないのです．通常のプログラムで非数チェックを挿入することはあまりないと思いますが，プログラムエラーが発生したときに使ってみて下さい[20]．

[19] ちなみに，この理屈で言えば，0.1×10 という掛け算結果も 1.0 と等しくならないはずですが，手元のパソコンでやってみると if 文を満足しました．これは，四捨五入のように計算結果を丸めるためです．しかし，丸め方はハードウェアやコンパイラの仕様に依存するので，それを期待してプログラムを書くのは避けた方が無難だと思います．

[20] コンパイラによっては，非数をチェックする関数 isnan(x) が用意されています．isnan は 3.3.4 節で説明する論理型の値が結果となる関数で，x が NaN であれば “真”，さもなくば “偽” です．

3.3.4 論理型

Fortran で取り扱える数値には，整数型，実数型，複素数型があります．また，print 文で出力するときに使用する文字列もデータ型の一種で，これについては第 6 章で詳しく説明しますが，Fortran には，もう一つ，**論理型** と呼ばれるデータ型が用意されています．論理型は，“真”と“偽”という 2 種類の値しか取らないデータ型のことで，大小関係の判定結果を変数に保存したり，if 文に記述して条件分岐に使うことができます．また，組み込み関数の中には論理型の結果を返すものがあるので，使い方を知っておくと便利です．

論理型の定数は 2 つしかありません．“.true.”と“.false.”です．これらは以下の意味を持ちます．

```
.true.      真
.false.     偽
```

これに対し，論理型の変数は logical を使って宣言します．

```
logical 変数 1，変数 2，...
```

ただし，論理型数が 2 値の情報しかないのにもかかわらず，この基本的な logical 宣言文により確保した論理型変数は整数型と同じ 4byte のメモリ領域を使用します．これではメモリが無駄なので，要素数の大きい論理型配列を使用する場合には，次のように 1byte の精度指定をした方が良いでしょう[21]．

```
logical(1) 変数 1，変数 2，...
```

この場合には，1 変数当たり 1byte のメモリ領域を使用するので，メモリの使用量は 1/4 になります．

さて，表 3.1 の比較条件や表 3.2 の論理式は，if 文の“条件”の中に記述するものとして説明してきましたが，論理型変数には，この“条件”の判定結果である“真偽”を直接代入することができます．例えば，次のようなプログラムを書くことができます．

```
logical l1,l2,l3
integer m
m  = 10
l1 = .true.                !  定数の代入
l2 = m == 5                !  比較条件の判定結果の代入
l3 = l1 .and. l2           !  論理演算結果の代入
print *,l1,l2,l3           !  論理型も出力可能
```

ここで，l1 には定数.true. が代入され，l2 には“m==5”の判定結果，すなわち「m は 5 に等しいか」という判定の真偽が代入されます．m は 10 なので，l2 は.false. です．l3 には「l1 かつ l2」の真偽が代入されるので.false. になります．この結果，最後の print 文の出力は，

```
T F F
```

のようになります[22]．

論理型変数は，if 文や do while 文の“条件”に記述して判定に使うこともできます．例えば，

```
logical l1,l2
integer m
do m = 1, 10
   l1 = m > 3
   l2 = m > 8
   if (l2) exit                        ! m > 8 なら do ブロック終了
   if (l1.and.(.not.l2)) print *,m     ! m > 3 かつ m <= 8 なら m を出力
enddo
```

[21] 精度指定の詳細は付録 D で説明しています．

[22] ここで，T が“真”，F が“偽”であることを示しています．この出力表示はコンパイラによって異なることがあります．

のように書くことができます．

論理型変数は，if 文を使わなくても大小関係の判定結果を保存することができるのですから，使い方によってはスマートなプログラムを作るのに役立ちます．

3.3.5 ビット操作関数

3.3.4 節で紹介した論理型は，真か偽かの 2 値しか取らないのですから情報量は 1bit ですが，保存する変数は logical(1) で宣言しても 1byte 必要です．一方，1.11.4 節で説明したように，整数型の数値は 4byte，すなわち 32bit ですから，整数変数 1 個あれば，32 個の真偽判定情報を保存することができます．そこで，この 2 進数情報をダイレクトに利用するための組み込み関数がいくつか用意されています．これを **ビット操作関数** といいます．代表的なビット操作関数を表 3.3 に示します．

表 3.3 ビット操作関数

ビット操作関数	引数の型	関数の型	関数の意味
not(i)	整数型	整数型	否定 (ビット演算)
iand(i,j)	整数型	整数型	論理積 (ビット演算)
ior(i,j)	整数型	整数型	論理和 (ビット演算)
ieor(i,j)	整数型	整数型	排他的論理和 (ビット演算)
ishft(i,n)	整数型	整数型	シフト
ishftc(i,n)	整数型	整数型	循環シフト

否定 (not) は “ビット反転”ともいいます．表 3.3 の上から 4 個の “ビット演算”というのは，整数型の数値を 2 進数で表したときに，各桁のビットごとに表 3.4 の演算を施した結果で表される整数を関数値として返す関数です．

表 3.4 ビット演算の一覧表

i	j	not(i)	iand(i,j)	ior(i,j)	ieor(i,j)
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

例えば，

```
i = 13
j = 11
print *,iand(i,j),ior(i,j),ieor(i,j),not(i)
```

というプログラムの出力結果は，

```
 9  15  6  -14
```

となります．11 と 13 を 2 進数で表すと “1011”と “1101”なので，各桁の論理積は “1001”となり，10 進数では 9．論理和は “1111”なので 10 進数では 15．排他的論理和は “0110”なので，10 進数では 6 になるというわけです．

最後の 13 の否定が −14 になるのは少し説明が必要です．ほとんどの計算機では負の整数を **2 の補数** で表現します．2 の補数というのは，正の整数 n に対して，n に加えると 2^M になる整数です[23]．ここで，M は整数を表現する最大ビット数で，整数型数では 32 です．さて，正の整数 n と n の否定を加えれば，全てのビットが 1 の整数，$2^M - 1$ になります．よって，n の否定に 1 を加えた数が 2 の補数，$-n$ になります．すなわち，n の否定は，$-n$ から 1 を引いた数になるわけです．

これに対し，表 3.3 のシフト関数は整数 i をビットシフトする関数です．数学的に言えば，i に 2^n を掛けることに相当します[24]．n が正の時は上位シフト，負の時は下位シフトになります．ただし，関数 ishft はあふれた桁を切り捨てるのに対し，ishftc は循環させます．すなわち，上位桁にあふれた場合は下位桁に回し，下位桁からあふれた場合には上位桁に回します．

例えば，

```
i = 13
print *,ishft(i,2),ishft(i,-2),ishftc(i,3),ishftc(i,-3)
```

というプログラムの出力結果は，

```
52  3  104 -1610612735
```

となります．ishft(i,2) は 13 の 2^2 倍なので 52，ishft(i,-2) は 13 を 2^2 で割って切り捨てるので 3 になります．これに対し，ishftc(i,3) は 13 の 2^3 倍なので，104 になりますが，ishftc(i,-3) は下位の 3 桁を上位に回すので，−1610612735 になります．上記の 2 の補数を考慮してなぜだか考えてみて下さい．

ビット操作は，コンピュータの基本的動作なので高速に処理できるという利点もあります．例えば，2^n で掛けたり割ったりするときにはシフトを使うと高速化されるし，2^n で割ったときの余りは，$2^n - 1$ との論理積を使えば高速化されます．

[23] n と，n の 2 の補数を加えた 2^M は，M ビットが全て 0 で，$M+1$ ビット目だけが 1 です．よって，M ビットだけ取り出せば 0 になることから，計算機的には 2 の補数が $-n$ になるというわけです．

[24] 厳密に言えば，2^n を掛けたときにオーバーフローした場合には結果が異なります．

演 習 問 題 3

(3-1) 2 次方程式の解 (解の判別付き)

3 個の実変数 a，b，c に適当な値を代入し，それから，

$$ax^2 + bx + c = 0$$

の解を計算して出力するプログラムを作成せよ．まず，解の判別式，$D = b^2 - 4ac$，を計算して，2 実解になった時，重解になった時，2 虚解になった時でそれぞれ正しい結果を出力するようにせよ．さらに，その解を $ax^2 + bx + c$ に代入して 0 に近い値になることを，判別式の値と数値型に応じて計算手順を変えて確かめよ．

(3-2) 非線形方程式の解法：逐次代入法

$x_0 = 0$ として，次のように次々に代入していくと，x_n は徐々に $\cos x = x$ の解に近づくことがわかっている．

$$\begin{aligned} x_1 &= \cos x_0 \\ x_2 &= \cos x_1 \\ x_3 &= \cos x_2 \\ &\vdots \\ x_{n+1} &= \cos x_n \end{aligned}$$

最初に，この代入を 100 回くり返し，その後で 101 回目の値 x_{101} と 100 回目の値 x_{100} の差を計算するプログラムを作成せよ．

それができたら，収束条件を $|x_{n+1} - x_n| < \varepsilon$ とし，ε を適当に小さい値に定めて (例えば，10^{-7})，収束したら x_n を出力して終了するプログラムにせよ．このとき，収束するまでの回数も出力せよ．

(3-3) 倍数の抽出

1〜10000 の整数の中で，「3 で割ったときの余りが 0 になる」，「7 で割ったときの余りが 1 になる」，「13 で割ったときの余りが 3 になる」，のどれかの条件を満足する数値の個数を計算せよ．また，それらの数値の合計と平均も計算せよ．

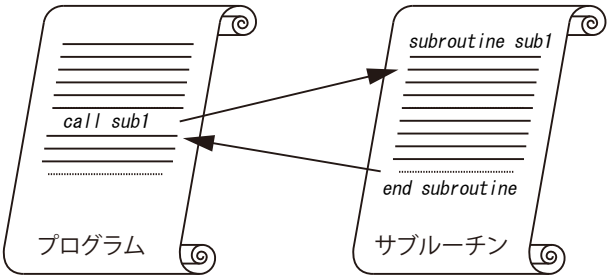
(3-4) バブルソートによる並べ替え

まず 10 要素の整数型 1 次元配列 `a(10)` を用意して，`a(1)`〜`a(10)` に $71, 71\times2, 71\times3, ..., 71\times10$ のように 71 の倍数を代入する．次に，それらの数字の下 2 桁 (100 で割ったときの余り) が小さいものから順になるように `a(1)`〜`a(10)` の要素を並べ替えよ．なお，小さいものから順になるような並べ替えには次のバブルソートアルゴリズムを利用する．

(1) n 個のデータを a_1，a_2，$\cdots$，a_n とする．最初に a_1 を $j = 2$〜n の a_j と比較して，もし $a_1 > a_j$ ならば，a_1 と a_j を交換し，さもなくばそのままとする．この結果，a_1 が最も小さい要素になる．
(2) 次に，a_2 と $j = 3$〜n の a_j を比較して，$a_2 > a_j$ ならば，a_2 と a_j を交換し，さもなくばそのままとする．この結果，a_2 が下から 2 番目に小さな要素になる．
(3) 以下同様に a_i と $j = i+1$〜n の a_j を比較して，$a_i > a_j$ ならば交換するという手順を $i = n-1$ になるまでくり返せば，全ての要素が小さい方から順に並ぶ．

これがバブルソートアルゴリズムである．なお，この問題では，配列要素を直接比較するのではなく，要素を 100 で割ったときの余りの大小比較を行えばよい．

第 4 章　サブルーチン



ここまで，Fortran の基本的な文法と高速化のテクニック，エラーの出にくいプログラムの書き方などを説明してきました．これ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．プログラムとはコンピュータ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．動作のく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，コンピュータの能力のほとんどを引き出すことができます．

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を起こす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．本章ではこのサブルーチンの作り方と使い方を説明します．

4.1　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しか書けませんが，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，必要ならそれらをくり返すための do 文だけを書いておきます．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．

4.2 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[25]．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の並び (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 文には subroutine 文と同じサブルーチン名を指定します．サブルーチン名は，メインプログラムと同じ規則で付けて下さい．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[26]．

上の例では，subroutine 文にサブルーチン名しかありませんが，サブルーチン名の後に，かっこで囲んだ変数のリストを付加することもできます．リスト中の変数を **引数** (ひきすう) といいます．以下に引数ありサブルーチンの 1 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
```

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ために使います．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，サブルーチンの動作を指定することを，“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．

---

[25] end subroutine 文は end だけの end 文でも良いのですが，program 文に対応した end program 文を書くのと同様に，end subroutine 文を使ってサブルーチンの終了点に名前を付記するようにしましょう．

[26] ただし，異なるルーチン間で共有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，4.7 節で説明します．

```
    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)
```

1.5 節で説明したように，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)              !  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
```

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 4.1 に示します．

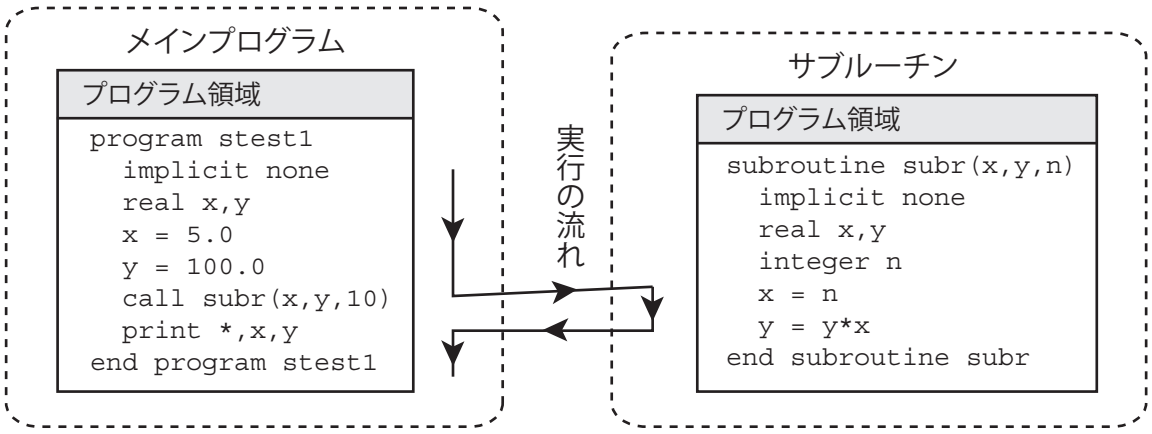


図 4.1　サブルーチンの呼び出しと実行の流れ

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．

4.3 ローカル変数と引数

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 subr1
end program stest1

subroutine subr1
  implicit none
  real x,y
  print *,x,y
end subroutine subr1
```

と書いても，サブルーチン subr1 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次の subr2 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 subr2(x,y)
end program stest2

subroutine subr2(x,y)
  implicit none
  real x,y
  print *,x,y
end subroutine subr2
```

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，コール側の引数とサブルーチン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 subr2(a,b)
  implicit none
  real a,b
  print *,a,b
end subroutine subr2
```

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．このため，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 subr3(10.0,z**2,100,n*5+1)
end program stest3

subroutine subr3(x,y,m,n)
  implicit none
  real x,y
  integer m,n
  print *,x,y,m,n
end subroutine subr3
```

というプログラムでは，表 4.1 のような対応になっています．

表 4.1　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正常に動作しません．これは，サブルーチンから見れば，どの様な型の数値が来るのかはわからず，あくまでもサブルーチン側で期待する数値型として解釈するからです．

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 subr4(x+y,20.0,p)
  print *,x,y,p
end program stest4
```

```
subroutine subr4(x,y,z)
  implicit none
  real x,y,z
  z = x*y
end subroutine subr4
```

と書くと，サブルーチン subr4 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，戻り値を代入する引数は別で，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．

4.4 間接アドレスを用いたルーチン間におけるデータの受け渡し

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．

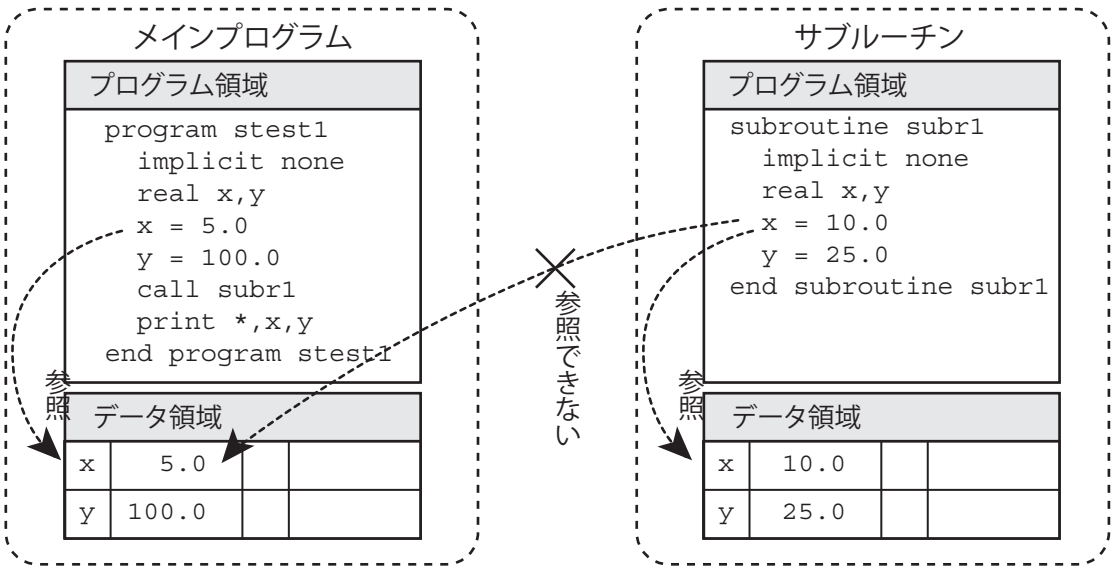


図 4.2　プログラム領域とデータ領域

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 4.2 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．

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，最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ローカル変数に代入することです．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．しかし，“call by value”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のかわからないからです．

そこで，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．

1.1 節で述べたように，CPU が変数 (メモリ) とデータのやりとりをするときにはメモリアドレスを利用しています．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，代入文 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 4.3 のようになります．このような指定したアドレスのメモリに入っているデータを読み書きする方式を **直接アドレス** といいます．

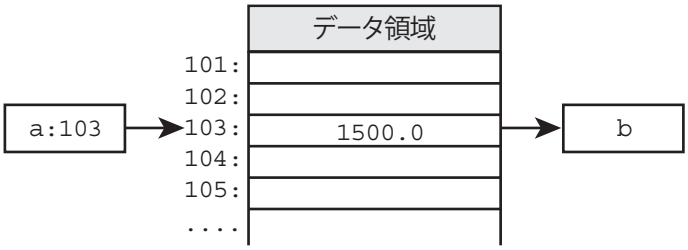


図 4.3　直接アドレスによるメモリ参照

この方式に加えて，コンピュータには，あるアドレス (AD_1) のメモリに入っているのが別のメモリのアドレス (AD_2) で，アドレス AD_1 を指定して，アドレス AD_2 のメモリに入っているデータを読み書きする方式も用意されています．これを **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，引数にはコール側ルーチンのメモリアドレスが入っています．このため，サブルーチン内で引数の値を読み書きするときは間接アドレスを使います[27]．例えば，

```
program stest3
  implicit none
  real a
  a = 1500.0
  call subr(a)
end program stest3

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

[27]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．

というプログラムを考えてみましょう．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 [103] 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 [106] 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  →  [106] 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 4.4 のようになります．

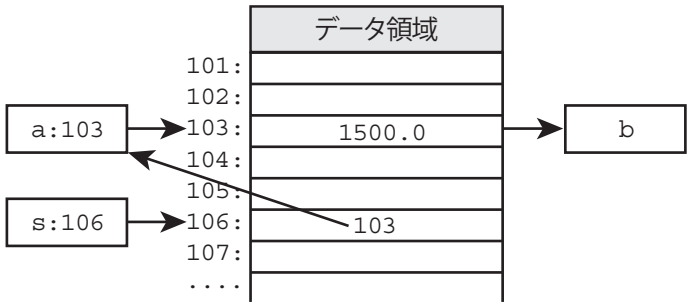


図 4.4　間接アドレスによるメモリ参照

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[28]．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特にメリットはありません．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，このしく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．

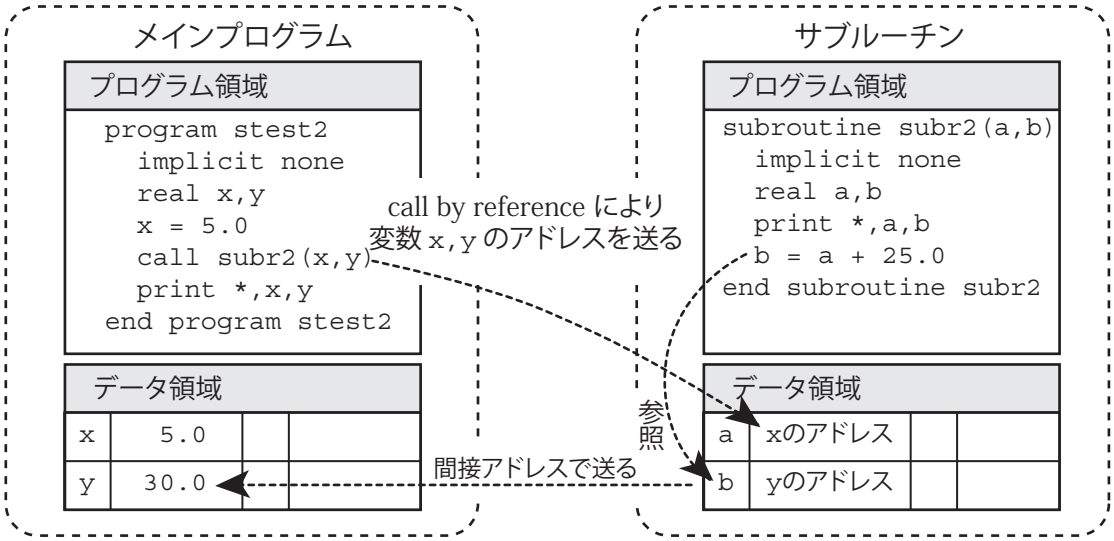


図 4.5　“call by reference”を利用してデータを送り返す

例えば，図 4.5 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正常に動作しません．

[28] このため，引数変数の値を頻繁に使う場合には，その値を別に用意したローカル変数に代入して使う方が速くなります．

4.5 配列を引数にする場合

配列を call 文の引数に使うときは，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．2.1.2 節で述べたように，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．1 例を示します．

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

この例では引数 x が単一変数として宣言されています．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)                 !  配列宣言
  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．x(100) のように大きくても，x(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ためです．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) を明示するためのダミーにすぎません．

このため，

```
call sub(a(3))
```

と書くと，a(3) を先頭とする並びの配列として処理します．すなわち，10 個の要素を持つ配列 x(1)，x(2)，...，x(10) は，a(3)，a(4)，...，a(12) に対応します．このため，“real b(10,10)”と宣言した 2 次元配列の j 列目の 10 個の要素，b(1,j)，b(2,j)，...，b(10,j) を処理したいときには，

```
call sub(b(1,j))
```

と書けばいいことになります[29]．なぜなら，2 次元配列の要素がこの順番で並んでいるからです．

[29] これに対し，i 行目の要素，b(i,1)，b(i,2)，...，b(i,10) を処理したい場合には，順番に並んでいないのでこの方法は使えません．この場合には部分配列を利用します．詳しくは 7.2 節で説明します．

このように，サブルーチンの引数が受けるのは，配列の先頭アドレスの情報だけなので，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[30]．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，配列だけではなく，配列の要素数も引数に加えてサブルーチンに伝える必要があります．

配列の形状さえわ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[31]．

“*”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[32]．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) のような宣言だけではなく，整数変数 1 個で，a(n,n) のように宣言することもできるし，a(0:2*m,n-1) のように，下限指定や計算式を使った宣言も可能です．

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，正しく動作します．

---

[30] 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，2.1.2 節で述べたように 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 に 100 (=10×10) を与えます．

[31] なお，“*”だけ書くと配列の下限は 1 です．もし下限を変更したいときには下限を別途指定します．例えば，配列 a の下限を 0 にする場合は，“real a(0:*)”と宣言します．

[32] 2.1.2 節の配列の順番を与える公式が一番右側の要素数に依存しないことを思い出して下さい．

```
program stest1
  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 stest1
```

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 stest2
  implicit none
  real a(-1:100)
  call sub(a,a,100)
end program stest2

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，下限の数値も引数変数にした整合配列を用います．

4.6 関数副プログラム

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型宣言し，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 1 例を示します．

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

このように，**function 文** で開始し，**end function 文** で終了します[33]．また，関数名 square を実数型宣言し，引数 x の 2 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副プログラムになります．

[33] end function 文は end だけの end 文でも良いのですが，subroutine 文に対応した end subroutine 文を書くのと同様に，end function 文を使って関数副プログラムの終了点に名前を付記するようにしましょう．

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                ! 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
```

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正常に動作しません．

関数名に値を代入するという形式は，Fortran の古くからある文法の一つですが，関数名が長いときなど，計算式の中に関数名を入れたくないときのために，function 文に付加する **result 句** が用意されています．result 句を使えば，関数値を代入する変数名を任意に指定することができます．例えば，上記の関数副プログラム square は，以下のように書くことができます．

```
function square(x) result(y)  ! result 句付きの function 文
  implicit none
  real x,y                    ! result 句で指定した変数の型宣言
  y = x*x                     ! result 句で指定した変数に値を代入すると関数値になる
end function square
```

このように，function 文の末尾に “result(変数名)”を加えるのが result 句です．result 句で指定した変数に値を代入すれば，その値が関数値になります．このため，result 句の変数は関数値の型と同じ型宣言をしておく必要があります．

4.7 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，引数を介してのみ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，数値型や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．

4.1 節で，計算機シミュレーションでは計算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，ルーチン間で共有する変数が多数必要です．しかし，共有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[34]．

モジュールは，次のようにモジュール名を指定した **module 文** で開始して **end module 文** で終了します[35]．モジュールの中で変数を宣言すれば，宣言された変数がグローバル変数として利用できます．

[34] モジュールや use 文は Fortran90 から採用された仕様です．それ以前の Fortran 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書では説明を省略します．

[35] end module 文は end だけの end 文でも OK です．グローバル変数を宣言するためのモジュールは短いので，end 文でもいいと思います．ただし，モジュールにはサブルーチンや関数副プログラムを含めてプログラムパッケージにする機能があるので，そのような使い方をする場合には end module 文を使って終了点に名前を付記した方がいいと思います．なお，より高度な使い方は参考文献などで調べて下さい．

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

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．

モジュールはサブルーチンや関数副プログラムと異なり，プログラムに記述しただけではその内容を利用することができません．利用するルーチンの先頭に，モジュール名を指定した **use 文** を使って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．モジュールを使ったプログラムの 1 例を以下に示します．

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

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

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

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 4.6 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．また，モジュールの中に use 文を書いて別のモジュールの変数を利用することもできます[36]．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．

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に宣言を限定することができます．

[36] 一つの応用例として parameter 変数の利用があります．4.8.2 節を参照して下さい．

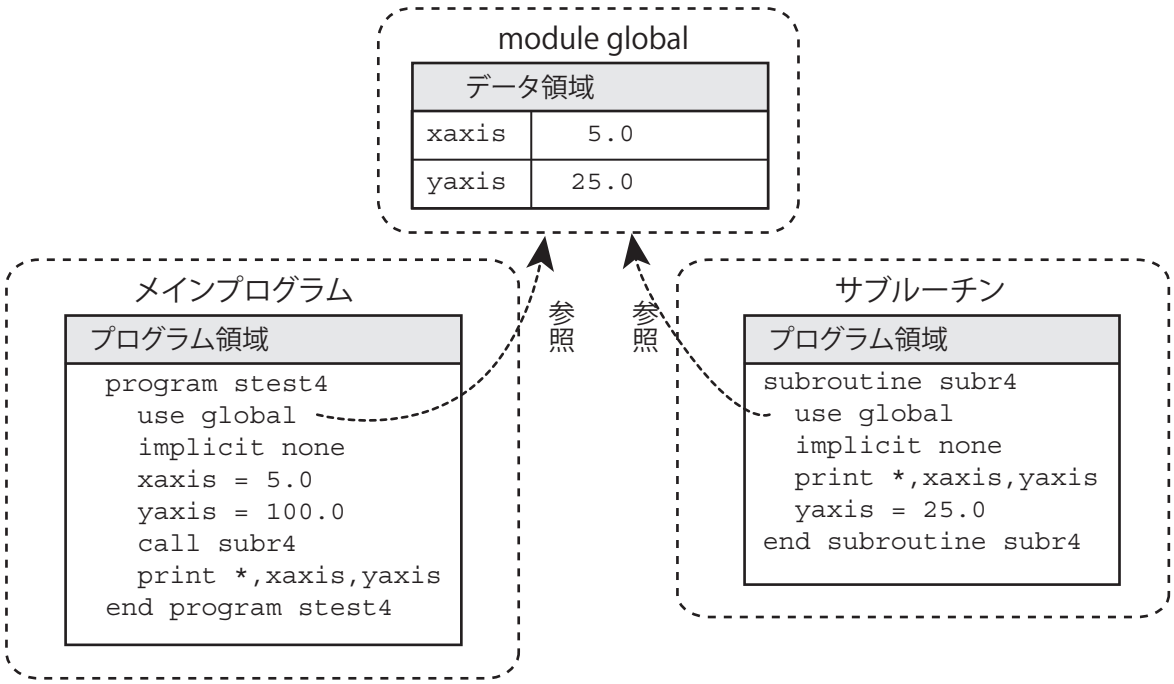


図 4.6　モジュールで宣言されているグローバル変数の参照

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

配列の場合には配列名だけを変数の位置に記述します.

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，ルーチン間で共有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利用する可能性があるので，意味のある長めの名前を付けます．これは，1.7 節で述べた“大域的に有効な変数名の付け方”です.

4.8　プログラミング スマートテクニック (その 4)

サブルーチンを利用するときは，変数の使い分けが一つのポイントです．4.7 節でローカル変数とグローバル変数の使い分けについて説明しましたが，ローカル変数にも 2 種類あり，これを使い分ければサブルーチンの動作をより細かく制御することができます．ここでは，プログラムのメンテナンスをさらに容易にしたり汎用性を持たせたりするための文法として，ローカル変数の使い分け方や，プログラムの実行中に必要に応じたサイズの配列を生成する配列の動的割り付けの方法などについて説明します. 加えて，乱数発生用サブルーチンも紹介します.

4.8.1　拡張宣言文とそれを用いたメモリ領域の使い分け

これまで変数や配列の宣言に使っていた宣言文は,

```
型指定 変数 1, 変数 2, ...
```

という形式でした．ここで，“型指定”には“integer”とか，“real”のような数値型を記述し，“変数”

には変数名か，配列名とその要素数を記述します．この宣言文は，データ領域におけるメモリの確保という意味もあります．

さて，Fortran90 から宣言文の記述形式が拡張されました[37]．この **拡張宣言文** を使うと，メモリを確保すると同時に初期値を代入したり，メモリの特性を指定することができます．拡張宣言文は以下の形式です．

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

ここで，角かっこ，“［”と“］”は，この中が省略可能であるという意味で使っているだけなので，角かっこ自体は書かないで下さい．拡張宣言文を使用するときは，型指定部と宣言する変数や配列の間にコロン 2 個“::”を書く必要があります．また，属性には宣言する変数の特性を指定するための予約語を記述します．属性も数値も省略して，単に型指定と変数の間に“::”を書いただけの宣言文は，書かずに宣言した旧来の書式と同じ意味になります．

属性を書かずに，単に数値を代入した形で宣言すると，その変数に指定した数値を代入して，プログラムの動作が開始することを意味します．例えば，

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

のように宣言すると，それぞれの変数に指定された値を代入した状態で動作が開始します．ただし，この宣言文中での数値代入は一度だけです．サブルーチン中で数値代入した変数を宣言しても，コールするたびに数値が代入されるのではありません．このため，その変数を実行文で変更すると，その変更した結果が残ります．例えば，

```
program stest1
  implicit none
  call subr1
  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
```

というプログラムでは，メインプログラムでサブルーチン subr1 が 3 回コールされていますが，サブルーチン中の print 文の出力は，1 回目が 1，2 回目が 2，3 回目が 3，となります．宣言文における代入は，プログラム開始時の初期値だと考えて下さい．

さて，上記のサブルーチンの動作を考えるとき，一つ注意しなければならないことがあります．4.4 節で，ローカル変数は各ルーチンのデータ領域に所属しているという説明をしましたが，もう少し詳しく言うと，サブルーチンのデータ領域には 2 種類あります．一つはサブルーチンがコールされた時点で一時的に生成されるメモリ領域で，もう一つはサブルーチンの呼び出しに関係なく常に確保されたメモリ領域です．本書では，前者を **一時メモリ領域** と呼び，後者を **固定メモリ領域** と呼ぶことにします．旧来の宣言文で宣言したローカル変数は，一時メモリ領域に所属します[38]．一時メモリ領域は，サブルーチンを呼び出すたびに場所や内容が変わる可能性があるので，同じサブルーチンを 2 回コールした場合，1 回目のコールでローカル変数に代入した値が，2 回目にコールした時点でそのまま残っているとは限り

[37] 本書では，拡張された宣言文のことを“拡張宣言文”と呼んで，旧来の型指定だけの宣言文と区別します．

[38] これに対し，メインプログラムの変数とモジュール内のグローバル変数は，宣言文の記述にかかわらず，全て固定メモリ領域に所属します．

ません．例えば，

```
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
```

のようなプログラムを書いたとします．1 回目の call 文，call subr2(1) で，ローカル変数 x と y に，それぞれ 10.0 と 100.0 という値が代入されますが，2 回目の call 文，call subr2(2) を実行したときに，print 文で出力した x と y が 10.0 と 100.0 になる保証はないのです[39]．ローカル変数に代入した値を保証するには，固定メモリ領域に所属させなければなりません．拡張宣言文を使って初期値を代入したローカル変数は，一時メモリ領域ではなく，固定メモリ領域に所属します．最初のサブルーチン例，subr1 の変数 n が，コールするごとに 1 ずつ増加した値を出力するのは，初期値 1 を代入して宣言することで固定メモリ領域に所属させたためです．

数値を代入せずに，ローカル変数を固定メモリ領域に所属させるには，変数に **save 属性** を指定します．例えば，上記のサブルーチン subr2 を以下のように書き換えれば，2 回目のコールで 10.0 と 100.0 が出力されます．

```
subroutine subr2(n)
  implicit none
  integer n
  real, save :: x,y                ! save 属性を指定
  if (n == 2) print *,x,y
  x = 10.0
  y = 100.0
end subroutine subr2
```

変数の初期値が未定のときや，ローカル配列を固定メモリ領域に所属させたいときには save 属性を使って下さい．また，初期値を代入する場合でも，固定メモリ領域に所属させることを積極的に利用するのであれば，次のように save 属性を付加して，固定メモリ領域にあることを明示した方がよいでしょう．

```
integer, save :: n=1
```

拡張宣言文の属性には save の他にも色々ありますが, ここでは配列を宣言するときに便利な **dimension 属性** を紹介しましょう．例えば，10×10 の 2 次元実数型配列を 5 個用意するとき，通常の宣言では，

```
real a(10,10), b(10,10), c(10,10), d(10,10), e(10,10)
```

のように書きますが，dimension 属性を使えば，これを次のように書くことができます．

```
real, dimension(10,10) :: a, b, c, d, e
```

すなわち，dimension 属性に配列の形状 (次元や要素数) を書いておけば，宣言する変数の位置には，配列名を記述するだけになります．属性は複数付加することができるので，dimension 属性に save 属性を加えて宣言することも可能です．

[39] コンパイラによっては，全てのローカル変数を固定メモリ領域に所属させるものがあり，このプログラムでもうまくいく場合があります．しかし，文法的には保証されていないのですから，当てにするべきではありません．

4.8.2 parameter 変数の利用

配列を宣言するときに便利なのが，**parameter 属性** を指定した変数，**parameter 変数** です．parameter 変数には必ず値を代入しなければなりません．例えば，

```
integer, parameter :: imax=10, jmax=200
```

のように書くと，imax と jmax が parameter 変数になります．

さて，配列を宣言するとき，その要素数は整数定数で指定しなければなりません．このため，配列の長さを変更するときには，宣言した要素数に関連した数値を全て変更する必要があり，手間がかかるだけでなく，見落としたり，書き間違える可能性があります．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
```

というプログラムにおいて，配列要素数の 100 は，宣言文だけでなく，do 文の終了値にも入っています．このため，配列要素数を変更するときは 3 カ所変更しなければなりません．このとき，配列宣言の要素数を parameter 変数に代入する形で用意しておくと，配列要素数をその parameter 変数で置きかえることができます．例えば，上のプログラムは次のように書き換えることができます．

```
integer, parameter :: imax=100
real a(imax),b(imax)
integer i
do i = 1, imax
  a(i) = i*i
  b(i) = a(i)**2
enddo
```

このプログラムならば，imax に代入する数値を変更するだけで，配列要素数の変更は完了です．

parameter 変数を含んだ計算式を別の parameter 変数への代入値にしたり，配列宣言の要素数に使うこともできます．これらは，その計算式の結果で宣言したことに相当します．例えば，

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

という宣言は，以下の宣言文と同等です．

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

ただし，他の parameter 変数に代入するときは，代入される変数 (この例では imax2) が，代入する計算式に使う変数 (この例では imax) より後で宣言されなければなりません．

実は，parameter 属性を指定した変数は，“変数＝数値”の形式で変数名と数値の対応関係を示しているだけで，メモリとしての実体はありません．この対応関係を使った“変数”から“数値”への置き換えは，コンパイルの段階で行われるので，非実行文である宣言文でも使えるのです．逆に言えば，parameter 変数に実行文を使って数値を代入することはできません．例えば，

```
integer, parameter :: imax=100
real a(imax),b(imax)
imax = 5                  ! これはエラー
```

と書いた場合，“imax=5”はコンパイルエラーです．これは，プログラムに“100=5”と書いたことに等しいのですから当然だと言えます．

このように，parameter 変数を使えば配列要素数の変更が楽になりますが，それでもルーチンごとのローカルな宣言では，変更が必要になったときに使用している全てのルーチンで parameter 変数の設定値を変更しなければなりません．そこで，

```
module global_param
  integer, parameter :: imax=300, jmax=200
end module global_param
```

のように，parameter 変数を記述したモジュールを用意して，共有するルーチンで use 宣言をすると便利です．use 宣言はモジュールの中でも使うことができるので，以下のように記述することが可能です．

```
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
   .......
```

ここで，あるモジュールの中で別のモジュールを use 宣言した場合，前者のモジュールを use 宣言したルーチンでは，後者のモジュールも合わせて use 宣言したことになります．上記のメインプログラム mtest1 ではモジュール mod_array しか use 宣言をしていませんが，mod_array の中で use 宣言をしているモジュール global_param も同時に宣言したことになり，その中にある変数 imax もメインプログラムで利用可能になります．

なお，次のようにメインプログラムに global_param の use 宣言を加えてその使用を明示してもエラーではありません．

```
program mtest1
  use global_param                          ! 書いてもエラーではない
  use mod_array
  implicit none
  integer km(imax)
   .......
```

これは，同じモジュールの use 宣言を重複して書いてもエラーにはならないからです．

parameter 変数は，数学定数や物理定数を用意するのにも便利です．筆者は次のような定数の入ったモジュールを作成して，必要なサブルーチンから利用するようにしています[40]．

```
module physical_constants
  real, parameter :: pi=3.141592653589793,     pi2=pi*2
  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
```

parameter 変数は不用意に書き換えられることがないので，このような定数の定義にはうってつけです．また，コンパイル時に数値の置き換えをするのですから，代入という実行動作が不要になり，わずかですが計算時間の節約になります．

[40] 2018CODATA 利用．

4.8.3 サブルーチンや関数副プログラムを引数にする方法

本章で説明したサブルーチンや関数副プログラムは，変数や配列などの数値を引数に持つものでした．これに対し，サブルーチンや関数副プログラムといった **外部副プログラム** を引数に持つサブルーチンや関数副プログラムを作ることも可能です．例えば，方程式 $f(x) = 0$ を解くサブルーチンにおいて，関数 $f(x)$ を計算する関数副プログラムの名前も引数に加えておけば，コールするときにユーザーが関数を指定することができる，より汎用的なサブルーチンになります．

サブルーチン名を引数にしたサブルーチンを作るのは簡単で，引数並びの中にサブルーチン名を入れるだけです．例えば，

```
subroutine subrout(subr,xmin,xmax,n)
   implicit none
   real xmin,xmax,dx,y
   integer n,i
   dx = (xmax-xmin)/n
   do i = 0, n
      call subr(dx*i+xmin,y)          !  引数 subr を使った call 文
      print *,i,y
   enddo
end subroutine subrout
```

のように書くことができます．この例では，subr が引数としてのサブルーチン名で，これをコールする文を書くことができます．関数副プログラムを引数にする場合も同様ですが，関数名の型宣言は必要です．例えば，次のように書きます．

```
subroutine funcout(func,xmin,xmax,n)
   implicit none
   real func,xmin,xmax,x,dx,y         !  引数 func が関数なので型宣言をする
   integer n,i
   dx = (xmax-xmin)/n
   do i = 0, n
      x = dx*i + xmin
      y = func(x)**3                  !  引数 func を関数として使う
      print *,x,y
   enddo
end subroutine funcout
```

このように，外部副プログラムを引数に持つサブルーチンを作るのは簡単ですが，このサブルーチンを使用するときは，「引数が変数ではなく，外部副プログラムである」という宣言が必要です．この宣言は **external 文** で行います．例えば，

```
subroutine sub(x,y)
   implicit none
   real y,x
   y = 2*sin(x) + cos(x**2)
end subroutine sub

function fun(x)
   implicit none
   real fun,x
   fun = sin(x)**3
end function fun
```

というサブルーチンや関数副プログラムを引数にして，上記のサブルーチン (subrout や funcout) をコールするときは，

```
program test_func
   implicit none
   external fun,sub                          !  external 文を使って宣言する
   call subrout(sub,0.0,3.0,10)
   call funcout(fun,0.0,3.0,10)
end program test_func
```

のように書きます．external 文による宣言がないと，sub や fun という単語が“変数”と見なされて，型宣言されていないというエラーになります[41]．external 文は非実行文なので，変数の型宣言と同様に，全ての実行文より前に書く必要があります．

なお，関数を引数に持つサブルーチンに sin や exp のような組み込み関数名を与えてコールするときは，external 文の代わりに intrinsic 文で宣言しなければなりません．例えば，

```
program test_sin
   implicit none
   intrinsic sin
   call funcout(sin,0.0,3.0,10)
end program test_sin
```

のように書きます．これは，Fortran における組み込み関数が宣言をしなくても使える特別なものだからです．しかし，それなら宣言せずとも使えるようにしてくれればいいと思うのですが，intrinsic 宣言をしておかないと「sin という変数が宣言されていない」というエラーになります．このあたりは Fortran コンパイラの仕様の問題だと思います．

4.8.4 配列の動的割り付け

配列を宣言するときには，整数定数を与えて要素数を明示しなければなりません．これは，その情報に基づいて，プログラムの動作開始時に計算で使用するメモリ量を確定するためです．しかし，最近のコンピュータではプログラムの実行中に必要になったメモリを確保したり，不要になったメモリを解放するという動作が頻繁に行われています．このような，プログラムの実行中に必要に応じてメモリを確保することを **動的割り付け** といいます．動的割り付けを使うと，メモリ効率の良い汎用性のあるプログラムを作成することができます．ここでは Fortran における動的割り付けの方法を説明します．

まず，サブルーチンの中で必要に応じた要素数の配列を確保する簡単な方法として，サブルーチンの整数型引数を使った配列宣言があります．例えば，次のサブルーチンで real 宣言されている 1 次元配列 b と c がこれに相当します．

```
   subroutine memory1(a,m,n)
     implicit none
     real a(m,n),b(m),c(n)              ! 引数 m と n を使って宣言する
     integer m,n,i,j
     do i = 1, m
       b(i) = i**2
     enddo
     do j = 1, n
       c(j) = 10*j
     enddo
     do j = 1, n
       do i = 1, m
         a(i,j) = b(i)*c(j)**3
       enddo
     enddo
   end subroutine memory1
```

[41] ちなみに，サブルーチン funcout に与えている関数名 fun は型宣言をしていませんが，これはエラーになりません．ただし，このルーチン内で fun を関数として計算に使う場合には型宣言をする必要があります．

配列 b と c は，このサブルーチンがコールされた時点で，引数 m と n の値を使ってそれぞれ確保されます．このとき，b(m) のような単純な宣言だけではなく，b(0:2*m) のような，下限指定や計算式を使った宣言も可能だし，d(m,n) のような 2 次元以上の配列も宣言できます．ここで，配列 a も引数を使って宣言されていますが，a は引数に含まれているので 4.5 節で説明した整合配列です．整合配列は，引数配列の取り扱いを決めるしくみにすぎず，配列自体はサブルーチン側にありません．

これに対し，引数に含まれていない配列 b や c はローカル配列であり，宣言されたサブルーチンの一時メモリ領域に所属します．4.8.1 節で説明したように，一時メモリ領域に所属するローカル変数は，サブルーチンを呼び出した時点で生成されます．引数の値はサブルーチン開始時に確定しているので，これを使って配列に必要なメモリを確保することが可能なのです．ただし，これはあくまでもサブルーチンで宣言する配列についてのみ使える機能であり，メインプログラムの配列やモジュール中のグローバル配列としては使えません．

そこで，メモリの動的割り付けを明示的に行うしくみが用意されています．Fortran で動的割り付けを行うには，二つの手続きが必要です．一つは割り付けたメモリを配列として使用するための配列名の宣言です．これは割り付けたメモリの先頭アドレスを保持するためのメモリを用意することだと考えられます．もう一つは，その配列名に実際のメモリを割り付ける動作です．前者は宣言文なので非実行文，後者はプログラム実行中に割り付けるので実行文です．

メモリを割り付ける予定の配列名は，**allocatable 属性** を付けて型宣言をします．このとき，配列の次元情報をコロン “:” を使って示す必要があります．配列の次元はコロンの数で決まり，実行時に変更することはできません．例えば，

```
real, allocatable :: ab(:),z2(:,:)
integer, allocatable, dimension(:,:) :: km1, km2
```

のように宣言します．1 行目のように，名前の後ろに次元情報を付加しても良いし，2 行目のように，dimension 属性 (4.8.1 節) を使うことも可能です．この例の場合，ab は 1 次元実数型配列，z2 は 2 次元実数型配列，km1 と km2 は 2 次元整数型配列として割り付けることができます．

配列の割り付けは **allocate 文** で行います．allocate 文は，次のように配列の要素指定をサブルーチンのようにかっこで囲んで記述します．

```
allocate ( ab(100), z2(0:m,-m:m), km1(k-1,2*k) )
```

かっこ内は，配列宣言と同じ形式です．ab や km1 のように，要素数に整数だけを与えると，その次元の下限は 1 になるし，z2 のように “:” を使って下限指定をすることも可能です．配列の数値型は型宣言の際に確定しているので，この例のように，一つの allocate 文で異なる数値型の配列を同時に割り付けることも可能です．allocate 文は実行文なので，z2 や km1 のように整数型変数や整数式の計算結果を使って割り付けることも可能です．一旦割り付けられた配列は，通常の宣言文で宣言した配列と同様に使用することができます．

ただし，むやみに割り付けをくり返すと，コンピュータで利用できる容量を超える “メモリオーバー” になる可能性があります．そこで，不用になった配列のメモリ領域は **deallocate 文** で解放することができます．deallocate 文は，次のように配列名だけをかっこで囲んで指定します．

```
deallocate ( ab, km1 )
```

deallocate 文も異なる数値型の配列を同時に解放することが可能です．

例えば，先ほどのサブルーチン memory1 を allocate 文を利用する形に書き換えると，以下のようになります．

```
subroutine memory1(a,m,n)
  implicit none
  real a(m,n)
  real, allocatable :: b(:), c(:)          ! 割り付け用 1 次元実数型配列
  integer m,n,i,j
  allocate ( b(m), c(n) )                  ! 要素数 m と n の 1 次元配列を割り付け
  do i = 1, m
    b(i) = i**2
  enddo
  do j = 1, n
    c(j) = 10*j
  enddo
  do j = 1, n
    do i = 1, m
      a(i,j) = b(i)*c(j)**3
    enddo
  enddo
  deallocate ( b, c )                      ! メモリの解放
end subroutine memory1
```

なお，引数変数を使って宣言したローカル配列は一時メモリ領域に所属しますが，allocate 文で割り付けた配列は固定メモリ領域に所属します．サブルーチンにおける一時メモリ領域は固定メモリ領域より使用可能容量が少ない可能性があるので，要素数の大きな配列を割り付けるときは allocate 文を使う方が良いでしょう．ただし，サブルーチン終了後，再度そのサブルーチンを呼び出したときに配列に代入しておいた数値を再利用するには，配列名の宣言に save 属性 (4.8.1 節) を加えて，“配列名”も固定メモリ領域に所属させなければなりません．

allocate 文を使った動的割り付けは，メインプログラムでも利用できるし，モジュールの中で配列名宣言をして，グローバル配列として使用することも可能です．ただし，allocate 文で割り付けられた配列を，解放しないで再度 allocate 文で割り付けようとすると実行時エラーになります．また，割り付ける前にその配列を使用しようとしても実行時エラーになります．そこで，配列が割り付けられているか否かを確認する関数 allocated が用意されています．allocated は論理型 (3.3.4 節) の値を返す関数で，allocatable 属性を持つ配列名を引数に与えると，その配列がすでに割り付けられていれば“真”，割り付けられていなければ“偽”を返します．例えば，次のように利用することができます．

```
real, allocatable :: a(:)
integer i,n
 .......
if (allocated(a)) then
   do i = 1, n
     a(i) = 100*i                  ! do ブロック 1
   enddo
 else
   allocate ( a(n) )
   do i = 1, n
     a(i) = 200*i                  ! do ブロック 2
   enddo
endif
```

この場合，配列 a がすでに割り付けられていれば“do ブロック 1”を実行し，割り付けられていなければ，allocate 文で割り付けてから“do ブロック 2”を実行します．

メモリの動的割り付けや解放は，それほど時間がかかる処理ではありません．計算の途中で一時的に利用する配列は，必要なときに割り付けて，不要になったら解放するようにしておけば，メモリの利用効率が高くて汎用性のあるプログラムになります．

4.8.5 乱数発生用サブルーチン

世の中の現象は全てが定まった法則によって動いているとは限りません．知らない道を歩いていて分岐点に出くわした時，どっちに進むかをコイントスで決めることもあれば，すごろくのようにサイコロを振って次の一手を決めることもあります．

このような，事象の発生が確率的にしかわからない現象をコンピュータで調べるとき，さいころの代わりをするのが**乱数**です．Fortran には，乱数を発生するサブルーチン random_number が用意されているので，これを用いれば乱数を使った計算ができます．例えば，1 個の乱数を使うときには

```
call random_number(x)
```

と書けば，変数 x に $0 \leqq x < 1$ の範囲の実数が代入されます．このサブルーチンの戻り値 x には規則性が無く，同じ呼び出しをくり返しても，それ以前に発生させた値との相関はありません．

乱数を 1 回で複数用意したいときには配列を使います．例えば，実数配列に乱数を代入するときは次のように書きます[42]．

```
real a(20)
call random_number(a)          ! 配列 a の全ての要素に乱数を代入
call random_number(a(1:10))    ! a(1)〜a(10) に乱数を代入
```

random_number は，実際に雑音のような規則性のないものを利用して値を決めているのではなく，ある一定の公式を使ってできるだけ乱数に近い数字列を計算しているものです．これを**擬似乱数**といいます．擬似乱数には初期値が必要で，初期値が同じ場合には同じ乱数列を発生します．このため，プログラムを実行するごとに異なる乱数列になるとは限りません．そこで，乱数の初期値を調べたり，自分で決めるためのサブルーチン random_seed が用意されています．random_seed の使い方は次の 4 通りです．

```
call random_seed               ! 初期値をシステムが設定する
call random_seed(size=n)       ! 初期値のサイズを変数 n に代入する
call random_seed(put=seed)     ! 初期値を整数配列 seed(n) に格納した値に設定する
call random_seed(get=seed)     ! 初期値を整数配列 seed(n) に代入する
```

random_seed は，引数を同時に 2 個以上指定することができないので注意して下さい．random_seed の使い方は少し煩雑なので，ここでは省略します．

[42] ここで a(1:10) は，配列要素 a(1)〜a(10) を取り出した部分配列です．部分配列については 7.2 節で説明しています．

演 習 問 題 4

(4-1) 変数の値を交換するサブルーチン

2 個の実数引数 x と y を持ち，その引数にそれぞれ実変数を与えると，その 2 個の実変数の内容が交換されて戻り値となるようなサブルーチンを作成せよ．

また，2 個の 1 次元実数配列 a(n) と b(n) および整数 n を引数に持ち，その引数にそれぞれ 1 次元配列とその要素数を与えると，その 2 個の配列の内容が全て交換されるようなサブルーチンを作成せよ．

(4-2) 2 次方程式の解 (サブルーチン利用)

3 個の実変数 a，b，c を引数とし，それから，

$$ax^2 + bx + c = 0$$

の 2 解 x_1，x_2，および判別式 $D = b^2 - 4ac$ を計算して戻り値にするサブルーチンを作成せよ．ただし，2 実解または重解になる場合は，x_1，x_2 をそれぞれ戻り値変数 x1 と x2 に代入し，2 虚解になる場合は，実部を x1 に，虚部を x2 に代入するようにせよ．

次にそれを使って問題 (3-1) の解答をサブルーチンを使う形に修正せよ．

(4-3) 統計計算 (サブルーチン利用)

要素数 n の 1 次元配列 a(n) と要素数を引数とし，

$$\text{平　均}\ \ \bar{A} = \frac{1}{n}\sum_{i=1}^{n} A_i$$

$$\text{標準偏差}\ \ \sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(A_i - \bar{A})^2}$$

を計算して戻り値とするサブルーチンを作成し，問題 (2-4) の解答をサブルーチンを使う形に修正せよ．

(4-4) モンテカルロ法

世の中の動きは，運動方程式のような微分方程式を用いて確実に予測できるとは限らない．例えば，さいころを投げて次にどの数字が出るかで進み方を決めるすごろくのように，進む先が確率的にしか決められない事象の積み重ねを計算することで未来を統計的に予測する方法をモンテカルロ法という．モンテカルロ法により円周率 π を計算するプログラムを作成してみよう．

コンピュータでさいころの代わりをするものは 4.8.5 節で説明した乱数である．まず，乱数発生用サブルーチン random_number を使って 2 個の乱数 x と y を発生させて 2 次元座標点 (x, y) を作る．この乱数の座標点を N 個作り，その中で原点から半径 1 の円より内側にある点の数 N_1 を数えて $p = \dfrac{4N_1}{N}$ を計算する．座標点数 N が増えると，この p が π に近づくことを確かめよ．

第 5 章　データ出力の詳細とデータ入力

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，最も単純な 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，本章では好みに応じて数値の出力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．

5.1　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，最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” で表示します．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，データの意味を並記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．また，変数や整数値を結果とす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[43]．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．この例のように，*form* に “*” を与えれば，print 文と同じく，データの数値型に応じた標準形式で出力します．

なお，原則として $nd \geqq 10$ にして下さい．これは，Fortran コンパイラの仕様によって，1 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 文と同じ標準出力の指定になるので，write(6,*) と書けば，ファイルへの出力はなく，画面に表示されます[44]．

なお，print 文や write 文において，“データ” の記述がなくてもエラーではありません．すなわち，

```
print form
write(nd, form)
```

などのように書くこともできます．これは，5.3 節で説明する format の中に文字列だけを入れて出力する場合や，改行だけを行いたいときに使うことができます．例えば，

```
print *
```

[43] “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．ファイル名は 5.6 節で説明する open 文を使えば変更することができます．また，コンパイラによっては，open 文を使わなくても装置番号に対応する環境変数の指定で，任意の名前を持つファイルに変更することが可能です．

[44] *nd* として，“*” を記述することもでき，これも標準出力指定です．すなわち，Fortran での標準出力には “print *form*,”，“write(6,*form*)”，“write(*,*form*)” の 3 通りがあります．

と書けば，改行だけ行う出力文になります．

5.2 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．同様に，

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

というプログラムを実行すると，最後の print 文では 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．

この配列名だけを print 文に書く単純な出力は，配列要素が少ないときには良いのですが，配列要素が多いと適当に改行を入れて出力されるのであまりお勧めできません．

そこで，次のように do 文を使って要素ごとに出力する方がよいでしょう．

```
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[45]．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
```

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 ずつ増えていき，その変数で指定されるデータが横に並んで出力されます．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)
```

[45] format を利用すれば続けて書くことは可能です．5.3 節を参照して下さい．

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,(i,a(i),b(i),i=1,3)
```

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

do 文と同様に，3 番目の数として増分値を付加することもできます．

```
（データ 1, データ 2, ..., 整数型変数 = 初期値，終了値，増分値）
```

例えば，

```
print *,(a(i),i=10,1,-2)
```

は，“print *,a(10),a(8),a(6),a(4),a(2)” 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整数型変数 (この例では j) 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，do 文のカウンタ変数とも重複しないようにしなければなりません．

5.3 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はできないし，結果の確認だけなら，それほど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，同じ形式の出力をくり返しても，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することができます．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．1 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 は format 文の文番号
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 (3.2 節) 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 に従ってデータが整形されて出力されます．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 文ごとに異なる数字をつけなければなりません[46]．

[46] 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慣習です．また，3.2 節で説明した goto 文のジャンプ先を示す文番号とは，最初の数字や桁数を変えて，系統が異なることを明示するようにします．

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．

```
  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 文は，書式を示すだけで，コンピュータの動作はありません．このため，記述の位置には無関係で，非実行文より後でさえあれば，指定する print 文や write 文より前に書くことも可能です．

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
  write(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 2 個の「’」で囲むのが基本ですが，「"」も使えるので，format 内部に文字列が入っているときには 2 個の「"」で囲みます．

この 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 6.4 節で説明します．

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 5.1 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，表 5.1 で斜体文字 (w, m, d) は整定数で指定します．また，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[47]．

表 5.1　主要な編集記述子

編集指定	数値の型	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 より小さい時には，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
ES$w.d$	実数	幅 w で実数を浮動小数点形式で出力する (d は E 編集と同じ) 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 (d は E 編集と同じ) 0 以外の数値を出力すると，仮数部の整数は 1 以上 1000 未満となり，指数部は 3 で割り切れる数になる
G$w.d$	整数／実数	整数の場合は I 編集を使って出力する 実数の場合は F 編集を使った固定小数点形式で出力するが， 指数部の絶対値が大きくて，幅 w では出力できないときには， E 編集に切り替えて浮動小数点形式で出力する
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
Lw	論理	幅 w で論理型値を出力する (T または F)

* I 編集の I の代わりに B を使えば 2 進数表示になり，Z を使えば 16 進数表示になります．

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 個無駄になります．そこで，ES 編集や EN 編集を使う方がよいでしょう．例えば，

[47]なお，i0 のように，幅指定数 w を 0 にすると，数値の桁数に応じた幅で出力されます．ただし，コンパイラによって解釈が異なるので，どのコンピュータでも同じ表示になるとは限らないようです．

```
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と同数の編集記述子を書かなければなりません．このとき，同じ編集記述子をくり返すならば，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，整数のようなくり返しのときには，かっこで囲んで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,'  ',i7))",x,m,y,n
```

この print 文の format は，“ f10.5,' ',i7,f10.5,' ',i7 ”と書くことと同等です.

複素数は，“実部，虚部”という実数のペアであり，コンピュータ内部的には要素数 2 の 1 次元配列と同型です．このため，複素数を出力するときは，複素数 1 個あたり，実数の編集記述子を 2 個並べる必要があります．例えば，以下のように書きます.

```
complex c
c = (1.0,-2.0)
print "(4f8.3)",c,c**2
```

この出力結果は,

```
    3.000  -2.000   5.000 -12.000
```

となります．しかし，これでは複素数という感じが出ないので，少し工夫して,

```
complex c
c = (1.0,-2.0)
print "(2(f8.3,' + (',f8.3,')i   '))",c,c**2
```

などとしてみればよいでしょう．この出力結果は,

```
    3.000 + (  -2.000)i      5.000 + ( -12.000)i
```

となります.

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，残りの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．そこで，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．例えば,

```
real a(4)
integer m
a(1) = 2.25
a(2) = 30.2
a(3) = 400.7
a(4) = 5000.6
print "(10(f10.2,'cm  '))",(a(m),m=1,4)          !  反復指定は 10
```

のように，反復指定を 10 回にしておいても，出力結果は，

```
    2.25cm       30.20cm       400.70cm      5000.60cm
```

となります．

文字列のように出力文中のデータ値との対応がない編集記述子もあります．代表的なものを表 5.2 に示します．ここで，r は整定数で指定します．

表 5.2　出力文中のデータ値との対応がない編集記述子

編集指定	編集の意味
/	改行する（“r/”のような反復指定も可)
X	スペースを挿入する（“rX”のような反復指定も可)
$	print 文や write 文終了時の改行を抑制する
:	出力文中の数値の出力が終わった時点で format 中の以後の出力を打ち切る

これらの記号を使うときには，コンマで区切る必要はありません[48]．例えば，2 次元配列 a(3,3) の配列要素を 3 行 3 列の行列のように 1 行あたり 3 個ずつ出力するときは，スラッシュ (/) 編集を使って，

```
real a(3,3)
 .......
print "(3(3f12.5/))",((a(i,j),j=1,3),i=1,3)
```

のように書くことができます．この format は，内側の 3f12.5 による 3 個の実数値出力の後で改行する，という動作を 3 回くり返すことを意味します．ただし，print 文は出力の最後に改行を行うので，このままだと無駄な改行が 1 回入ることに留意しておく必要があります．

ドル ($) 編集は，改行コードを出力しない指定です．通常，print 文や write 文で出力すると，出力後に改行をするため，複数の print 文や write 文を使って 1 行に出力することはできません．これは，指定した出力文字に加えて改行コードを出力しているためです．ドル ($) 編集を使えば改行コードを出力しないので，複数の print 文や write 文を使って 1 行に文字を続けて出力することができます．

例えば，5.2 節で説明した 2 次元配列の出力においては，次のように do ブロックの中に入れた要素ごとの print 文を使っても，列要素を横に並べて出力することができます．

```
do i = 1, 3
  do j = 1, 3
    print "(f12.5$)",a(i,j)
  enddo
  print *
enddo
```

ここで，内側の enddo の次に “*”だけ記述した print 文がありますが，これは，1 行出力した後で改行するためです (5.1 節)．この print 文を省略すると，全ての要素が横に並んで出力されます．

[48]ただし，反復指定をするときは，反復数の前にコンマを入れた方が良いでしょう．

先ほどのスラッシュ (/) 編集の例では余分な改行が 1 回入りますが，次のように最後にドル ($) を書いておけばこの改行を避けることができます．

```
print "(3(3f12.5/)$)",((a(i,j),j=1,3),i=1,3)
```

最後のコロン (:) 編集はわかりにくいので，例を使って説明します．先ほど反復指定に大きめの数値を与えておくことができる例を示しましたが，このとき数字の前に文字列を付けようとすると問題が起こります．例えば数値の前に $ 記号を付けたくて，

```
real a(4)
integer m
a(1) = 2.25
a(2) = 30.2
a(3) = 400.7
a(4) = 5000.6
print "(10('    $',f10.2))",(a(m),m=1,4)        ! 反復指定は 10
```

と書くと，出力結果は，

```
 $      2.25    $     30.20    $    400.70    $   5000.60    $
```

となります．すなわち，最後に余分な 5 個目の “$”が出力されるのです．これは，数値と編集記述子が対応しなくなった時点で出力が終了するのですが，5 個目の “$”の時点ではまだ終了するかどうか未定だからです．これを防ぐために使うのがコロン (:) 編集です．上記の format を修正して，

```
print "(10('    $',f10.2:))",(a(m),m=1,4)       ! f10.2 の後にコロンを挿入
```

のように f10.2 の後に “:”を入れておけば，対応しなくなった時点からさかのぼって，“:”の場所で出力が打ち切られます．これならば，

```
 $      2.25    $     30.20    $    400.70    $   5000.60
```

のように，“$”は 4 個しか出力されません．

5.4 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，コンパイルして作成した実行形式のプログラムは，その動作だけを利用する “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．

5.4.1 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[49]．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 $nd \geqq 10$ 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

[49] 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．ファイル名は 5.6 節で説明する open 文を使えば変更することができます．また，コンパイラによっては，open 文を使わなくても装置番号に対応する環境変数の指定で，任意の名前を持つファイルに変更することが可能です．

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．これに対し，read 文が要求する数値よりもファイルに書かれている数値の方が多い場合は，read 文に記述されている全ての変数に数値が代入された時点で入力が終了します．この後，別の read 文で再び入力を実行すると，最後に読み込んだ行の次の行から入力を再開します．例えば，fort.20 という名前のファイルに，

```
     5.2    1.5
     10     20
```

と書いて保存しておき，プログラム中に，

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

と書けば，この 2 回の read 文実行後，x = 5.2，y = 1.5，m = 10，n = 20 となります．read 文の処理は行単位で行われるので，最後に読み込んだ行に余分な数値が書かれている場合は無視されます．例えば，上記の fort.20 を書き換えて，

```
     5.2    1.5    30    40
     10     20
```

のように 1 行目に数字を余分に書いても，2 回目の read 文の結果は変わりません．

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，入力要素数やく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[50]．

```
  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．そこで，read 文の前に入力を促すような文字を出力する print 文を入れることをお勧めします．例えば，

[50]標準入力の装置番号は 5 です．よって，“read *,”と書く代わりに，“read(5,*)”と書くこともできます．また装置番号を*にして，“read(*,*)”と書くこともできます．

```
print *,'Input X and Y :'
read *,x,y
```

と書いておけば，入力のタイミングもわかるし，どの変数へ入力するための数値を要求されているかもわかります．

5.4.2 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．この強制終了を防ぐため，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，  ...
```

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，その行から実行を継続します．

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[51]．両方入れてもかまいません．入力データ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よいでしょう．

5.4.3 ネームリストを用いた入力

便利なデータ入力手段として，**ネームリスト** 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覚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，ネームリスト入力では，入力データを“変数＝データ値”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．

ネームリスト入力を使うときは，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
```

[51] コンパイラによっては err 指定でのファイルの終了検知ができないことがあります．そのときも end 指定を使って下さい．

と書きます．ローカル変数だけではなく，use 文で指定されたモジュール中で宣言されているグローバル変数も登録可能です．また，ネームリスト名の異なる複数のネームリストを作成することも可能です．

namelist 文を使ってネームリスト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．これを **ネームリスト入力文** といいます．ネームリスト入力文は変数を指定しません．必要ならば，

```
read(nd, ネームリスト名,err=num)
```

のように，文番号 num を使った “err=num” を追加してエラー発生時の処理をすることも可能です．例えば，ネームリスト名が option ならば，

```
read(15,option,err=999)
```

などのように書きます．

ネームリスト入力文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 namelist 文の登録順とは無関係なので，自由に並べることができます．

```
&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 “/” までがネームリスト入力文 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．入力を開始すると，ネームリスト入力終了の記号 “/” を読み込むまで入力を続けるので，次のように 1 行ずつ書くこともできます[52]．

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

ネームリスト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，ネームリスト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)
```

[52] 入力ファイルに同じ変数への代入文を複数記述することもできます．この場合は最後に代入した値が有効になります．

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．

ネームリストに登録された変数の内容は，次の **ネームリスト出力文** 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，ネームリスト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に限定して使った方がよいと思います．

5.5 書式なし入出力文によるバイナリ形式の利用

これまで，print 文・write 文による出力や read 文による入力には，文字を使って表現した数値を用いていました．これは我々人間の都合によるものです．コンピュータ内部の数値は 2 進数で表現されていて，“10”とか，“1.000e20”とかの文字ではありません．しかし，2 進数の並びでは我々が理解しにくいため，出力するときは“2 進数→ 10 進数文字表現”というデータ変換を行って画面に表示したりファイルに保存し，入力するときは，“10 進数文字表現→ 2 進数”というデータ変換を行って所定の変数に数値を代入するのです．この文字で表現した形式を **テキスト形式** といいます．

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，倍精度実数の 2 進数表現は 8byte ですが，実数を有効数字 15 桁で出力するには 15 文字 (15byte) 以上必要です．そこで，大量のデータを精度を落とさずに保存するときは，内部表現のままで保存するのが有効です．この内部表現を **バイナリ形式** といいます．Fortran でバイナリ形式の入出力を行うときは，write 文や read 文において *form* を省略した書式を用います．これを **書式なし入出力文** といいます．これに対し，これまで説明してきた *form* を指定する write 文や read 文は **書式付き入出力文** です．

書式なし write 文は，次のように装置番号だけをかっこ内に指定した write 文です．

```
write(nd) データ 1, データ 2,  ...
```

これに対し，書式なし read 文は，装置番号だけをかっこ内に指定した read 文です．

```
read(nd) 変数 1, 変数 2,  ...
```

書式なし read 文は，書式付き read 文と同様に，文番号 *num* を使って，

```
read(nd,err=num) 変数 1, 変数 2,  ...
```

のように，“err=*num*”や“end=*num*”を追加したエラー発生やデータ終了時の処理をすることもできます．

書式なし write 文で作成したバイナリ形式ファイルから，書式なし read 文を使ってデータを読み込むときにはいくつか注意が必要です．まず，write 文で出力したときのデータと同じ数値型の変数を同じ順番で read 文に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
```

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数値を入力するには,

```
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
```

のように入力すると，出力と数値型の異なる n と x には正しい値が代入されません.

また，テキスト出力のような“改行”はありませんが，write 文 1 回ごとに印 (ヘッダ) が付くので，write 文 1 回の出力に対し，read 文 1 回で入力しなければなりません[53]．ただし，write 文 1 回で書き込まれたデータ数よりも read 文 1 回で入力する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．例えば，上記の fort.20 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すると，1 行目の read 文実行時における 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.

逆に，上記の write 文を修正して,

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

という 2 行で出力したファイルから,

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

のように，read 文 1 回で読み込むこともできません.

書式なし write 文は，1 回で出力できるデータ数に特に制限はありません．このため，大きな配列を 1 回で出力することも可能です．例えば,

```
real p(10000),q(10000),r(10000)
write(30) p,q,r
```

のように，1 回の write 文で複数の配列の全要素を出力することができます.

もっとも，出力用のプログラムと入力用のプログラムのメンテナンスを考えれば，出力する配列要素数を次のようにあらかじめ出力しておいた方がよいと思います.

```
real p(10000),q(10000),r(10000)
write(30) 10000
write(30) p,q,r
```

[53] このように，書式なし write 文を使って保存したバイナリ形式ファイルには，データだけでなく Fortran 独自のヘッダが付加されています．このため，C 言語など，他のプログラミング言語で作成したプログラムから読み込む場合や，データ解析ソフトを使って解析するときにはうまく入力できない場合があります．そのような利用が目的でデータを保存するときは，テキスト形式で保存した方が良いでしょう.

これを入力するときは，出力数を考慮して

```
real p(10000),q(10000),r(10000)
integer i,imax
read(30) imax
read(30) (p(i),i=1,imax),(q(i),i=1,imax),(r(i),i=1,imax)
```

のように書きます．こうしておけば，出力用のプログラムを修正して要素数を減らしても，入力用のプログラムの変更は不要です．

5.6 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，任意の名前を持つ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．この動作を「**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] )
```

ここで，角かっこ，“［”と“］”は，この中が省略可能であるという意味で使っているだけなので，角かっこ自体は書かないで下さい．それぞれの記述 (**制御指定子**) の意味を表 5.3 に示します．

表 5.3 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 (整数型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“text.out”という名にするときは，

```
open(10,file='text.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“binary.dat”という名にするときは，

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

と書きます．ただし，既存の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．上書きを防ぎたいときには，

```
open(30,file='binary.dat',form='unformatted',status='new',err=999)
```

のように，status='new' と err を指定します．status に'new' を指定すると，ファイルが存在しなければ新しく作成し，存在すればエラーになります．そこで，エラーの処理を err で指定した文番号 999 の行に用意しておけば上書きを防ぐことができます．

status の指定を省略して open 文を実行したとき，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．このとき，そのファイルが read 文の入力用であれば，データが入っていないので read 文実行時にエラーになりますが，プログラム終了後も作成された空のファイルが残ってしまいます．そこで，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よいでしょう．例えば，バイナリ形式ファイル "binary.inp" を入力ファイルとして装置番号 20 に指定するときには，

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

のように書きます．この open 文では，オープンした時点でファイルが存在しなければ，err=999 で指定した文番号 999 の行へジャンプします．

ファイルへ出力する場合，write 文実行時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，次の **close 文** で完了動作を指定します．

```
close(nd)
```

この動作を「**ファイルをクローズする**」といいます．ここで nd はクローズするファイルの装置番号です．

例えば，装置番号 30 のファイルをクローズするときは，

```
close(30)
```

のように書きます．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切れます．このため，同じ装置番号に新たに別のファイルを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[54]．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．配列を使って計算しようとするとメモリオーバーになってしまうほど大量のデータを処理する場合などに利用して下さい．

[54]「巻き戻す」という言葉を使うのは，磁気テープが昔のコンピュータにおける大容量記録媒体の一つだったからです．しかし，最近は音楽用カセットテープでさえ使わなくなったのですから，あまり適切な言葉とは言えないかもしれませんね．

演 習 問 題 5

(5-1) 2 次方程式の解の出力

問題 (3-1) で作成したプログラムを修正し，read 文を使って 3 個の実変数 a，b，c をキーボードから入力すれば，それらを用いて，

$$ax^2 + bx + c = 0$$

の解を計算して，format を利用して以下の数値や文字列を出力するプログラムにせよ．

(1) 実変数 a，b，c の値（小数点以下 3 桁の固定小数点形式出力）
(2) 2 次方程式の解が，「2 実解」か，「重解」か，「2 虚解」かを示す文字列
(3) (2) の分類に応じた解の数値出力（小数点以下 7 桁の浮動小数点形式出力）

(5-2) くり返し出力

$n = 100$，$h = 0.01$ として，i=0〜n の整数に対し，1 行に以下の数値を並べて出力するプログラムを作成せよ．

$$i \qquad hi \qquad \sin(2\pi hi) \qquad \cos(2\pi hi) \qquad \sin(3\pi hi) \qquad \cos(3\pi hi)$$

ここで，i は文字幅 5 の整数で，hi は，小数点以下 2 桁の固定小数点形式で，sin，cos の値は，小数点以下 7 桁の浮動小数点形式で出力せよ．なお，π の値はできるだけ正確な値 (有効数字 15 桁程度) を使うこと．

(5-3) 行列の表示

m 行 n 列の行列 A を，2 次元実数配列 a(m,n) で表すとする．この配列と整数 m，n を引数に与えると，配列要素を次のように出力するサブルーチンを作成せよ．

(1) 1 行目に，1〜n までの整数を並べて出力

$$1 \qquad 2 \qquad 3 \qquad \ldots \qquad n$$

(2) 2 行目から $m+1$ 行目までは，$i+1$ 行目が次のように行番号と行列要素を並べて出力

$$i \qquad a_{i1} \qquad a_{i2} \qquad a_{i3} \qquad \ldots \qquad a_{in}$$

ここで，(1) の整数の出力と (2) の対応する行列の列要素の出力が縦に並ぶようにすること．

このサブルーチンを使って，5 行 4 列の行列 A を表現する配列，a(5,4) の各要素に適当な数値を代入して出力せよ．

(5-4) open 文，rewind 文，close 文によるバイナリファイルの利用

まず，open 文を使って，“test.dat”というバイナリ形式のファイルをオープンする．

次に，1〜100000 の整数を，要素数 1000 の整数配列に 1000 個づつ代入した後，その全ての配列要素を先ほどオープンしたバイナリファイルに書式なし write 文を使って書き込むという動作をくり返すことで，全てファイルに保存せよ．

次に，rewind 文を使ってファイルを巻き戻し，書式なし read 文を使って，1000 個づつ数値を読み込んで，その全ての値の合計と平均を計算せよ．

計算終了後は，close 文を使って，ファイルをクローズすること．

なお，合計計算の途中で整数の最大値を超える可能性があるので，合計を計算するときに使用する変数は実数型にすること．

第 6 章　文字列の活用

これまで print 文や format 文に記述して，数値の意味などを表示するのに使ってきた文字列ですが，この他にも様々な用途があります．また，固定した文字の並びだけではなく，文字列変数を使って条件に応じてその内容を変更することもできるし，文字列と文字列を連結したり，文字列の一部を取り出したりするなどの “文字列演算”もできます．本章では文字列をより積極的に活用する手法を紹介します．

6.1　文字列定数と文字列変数

Fortran における文字列とは 2 個の「’」または「"」で囲んだ文字の並びのことです．例えば，

```
'abc'  'Taguchi␣T.'  "(123.5678+X^2)"  "漢字も書けます"
```

等が文字列です．文字列にはスペースや記号を入れることもできます[55]．「’」と「"」は同等なので，どちらを使うかは自由です．例えば，通常は「’」で囲み，文字列の中に「’」を使いたいときは，「"Teacher's"」のように全体を「"」で囲む，というように決めておけばよいでしょう[56]．

コンピュータの「文字」は文字コードという整数値と対応していて，文字列は文字コードを表す整数の配列で実現されています．このため，整数定数の「1」と文字列の「’1’」は全く異なるものです．文字コードは半角英数字なら 1 文字あたり 1byte の整数，全角の漢字やひらがななら 1 文字あたり 2byte の整数です．半角英数字のコードは，現在ほとんど全てのコンピュータで ASCII コードが使われています[57]．このため，半角英数字だけでプログラムを書けば移植性の問題はありません．しかし，全角文字は OS や利用環境によって文字コードが違う可能性があるので，プログラム中に日本語を記述すると，別のコンピュータに移したときに文字化けすることがあります[58]．Fortran において日本語が使えるのはコメント文とテキスト表示くらいなので，それほど重要ではありません．以下，文字列中に記述した「文字」は 1byte の ASCII コードであると仮定して説明をします．

「’」または「"」で囲んだ文字列は，**文字列定数** ですが，文字列を代入して保存する **文字列変数** を作ることもできます．文字列変数を作るには **character 宣言** を使います．character 宣言は，以下のような書式です．

```
character 文字列変数 1*文字数 1，文字列変数 2*文字数 2，...
```

文字数とは，文字列変数に代入可能な最大の文字数です．指定できる最小の文字数は 1 です．また，拡張宣言文にして属性などを付加することも可能です．例えば，

```
character c1*10,c2*20,cc*1
character chr(20)*30,chs(100,200)*50
```

と宣言すると，文字列変数 c1 には 10 個まで，c2 には 20 個までの文字を代入することができます．これに対し cc には 1 文字しか入りません[59]．また chr や chs のように文字列の配列を宣言することも可能です．chr は 30 文字まで入る 1 次元配列で，chs は 50 文字まで入る 2 次元配列です．

同じ文字数の文字列変数を複数個宣言するときは，以下の 2 種類の書式で宣言することもできます．

```
character(文字数)  文字列変数 1，文字列変数 2，...
character(len=文字数)  文字列変数 1，文字列変数 2，...
```

[55]本章では半角スペース記号を明記するときに “␣”を使います．

[56]なお，「’」で囲んだ文字列の中に，「’’」のように 2 個の連続した「’」を入れると，1 個の「’」として扱われます．

[57]ASCII コード表は付録 E にあります．

[58]パソコンで使われているのは Shift JIS コードが多く，Linux などの UNIX 系 OS で使われているのは EUC コードが多いようです．最近は，コード体系をより統一化した Unicode を使う OS が増えてきました．

[59]文法的には “*文字数”を省略して変数名だけで宣言すると，文字数が 1 の文字列になります．しかし，プログラムがわかりにくくなるので，常に “*1”を付加して文字数 1 を陽に指定した方がよいでしょう．

ここで，「文字列変数」には，変数名か，配列名とその要素数を記述します．例えば，文字数 10 の文字列変数や文字列配列を宣言するときは，

```
character(10) cs1,ds1,cas1(100)
character(len=10) cs2,ds2,cas2(100)
```

などと書きます．2 行目の書式は，入力に手間がかかりますが「文字数」という意味を明示しているところが利点です．これらの書式では，次のように parameter 変数 (4.8.2 節) で文字数を指定することもできます．

```
integer, parameter :: kp = 10
character(kp) ch2
character(len=kp) ch3
```

文字列変数に文字列を代入するには，数値の代入と同じで，イコール「=」を使います．例えば，上記の 10 文字の文字列変数 c1 に文字列定数を代入するには，

```
c1 = 'abc'
```

のように書きます．このとき，代入される文字 'abc' は 3 文字なので，10 文字の変数 c1 の先頭から順に 1 文字ずつ代入され，残りの領域は半角スペースが代入されます．スペースも文字ですから，c1 の文字数は，代入した文字列の文字数にかかわらず 10 のままです．図に描けば，以下のようなイメージです．

c1	a	b	c	␣	␣	␣	␣	␣	␣	␣

逆に，代入する文字列の文字数の方が多い場合には，その文字列の先頭から代入できる最大文字数までが代入され，残りは切り捨てられます．拡張宣言文 (4.8.1 節) を使えば，次のようにあらかじめ文字列定数を代入した文字列変数を用意することも可能です．

```
character :: chr*10='abcde'
```

この場合，文字列変数の文字数が 10 文字なのに代入しているのは 5 文字ですから，後の 5 文字はスペースが代入されています．

6.2 部分文字列と文字列演算

文字列変数に代入された文字列は部分的に取り出すことができます．これを **部分文字列** といいます．部分文字列は，文字列変数の先頭から数えて何番目から何番目という範囲を「:」を使って指定します．例えば，c1 という文字列変数の n1 番目から n2 番目の文字を取り出すには，

```
c1(n1:n2)
```

と指定します．この文字列は，n2-n1+1 文字の文字列として扱われます．例えば，c1='abcdefg' ならば，c1(3:5) は，'cde' です．n1 と n2 を等しくすることで 1 文字を取り出すこともできます．例えば，

```
c1 = 'abcdefg'
do i = 1, 7
  print *,c1(i:i)
enddo
```

とすれば，1 文字ずつ縦に出力されます．

文字列配列の部分文字列を取り出す場合には，要素指定を先に，部分文字列指定を後に書きます．例えば，1 次元の文字列配列 chr に対し，k 番目の要素の部分文字列は，

```
chr(k)(n1:n2)
```

のように指定します．かっこが連続するので，順番に気を付けて下さい．

文字列は連結することもできます．文字列の連結には演算子「//」を用います．例えば，

```
c1 = 'abc'//'xyz'
```

と書くと，c1 には'abcxyz' という文字列が代入されます．文字列の連結は，文字列定数と文字列変数，文字列変数と文字列変数という組み合わせでも可能です．ただし，文字列変数に代入された文字列を連結するときには注意が必要です．例えば，

```
character c1*10,c2*20
c1 = 'abc'
c2 = c1//'xyz'
```

と書いた場合，c2 に'abcxyz' という文字列が代入されると思ったら間違いです．正しくは

```
'abc␣␣␣␣␣␣␣xyz'
```

が代入されます．これは，上記のように 10 文字の文字変数 c1 に 3 文字の'abc' を代入しても，c1 の文字数は変わらないからです．末尾の不要なスペースを削除するには，部分文字列を使う必要があります．

```
character c1*10,c2*20
c1 = 'abc'
c2 = c1(1:3)//'xyz'
```

このプログラムならば，c2 に'abcxyz' が代入されます．

しかし，この方法は変数に代入されている文字数が不明のときには使えません．そこで，関数 trim が用意されています．trim は末尾のスペースを除去した文字列を返す組み込み関数です．これを使えば，

```
character c1*10,c2*20
c1 = 'abc'
c2 = trim(c1)//'xyz'
```

と書くことができます．この結果も c2 には'abcxyz' が代入されます．

文字列をサブルーチンの引数にするときは，「(*)」を指定して宣言します．例えば，

```
subroutine csubr1(chr)
  implicit none
  character(*) chr          ! 文字数は不要
   .......
```

の chr のように宣言します．この宣言文は，次のように書くこともできます．

```
  character chr*(*)         ! 文字数は不要
```

Fortran の文字列には，文字コードの並びだけではなく，文字数の情報も含まれています．このため，(*) 指定のように文字数が明記されていなくても，コール側で指定した文字数の文字列として使用することができます．例えば，

```
program ctest1
  implicit none
  call csubr1('abcde')
end program ctest1

subroutine csubr1(chr)
  implicit none
  character(*) chr
  print *,chr,len(chr)      ! 文字列 'abced' と文字数 5 が出力される
end subroutine csubr1
```

のようなプログラムにおいて，サブルーチン csubr1 中の print 文の出力は，文字列 'abcde' と文字数 5 になります．ここで，関数 len は文字列の文字数を取得する関数です．文字列に関する組み込み関数は，6.6 節で説明します．

6.3 文字列の大小関係

文字列には大小関係があり，これを利用して if 文で条件分岐をすることもできます．2 個の文字列を比較するときは，以下の手順で行います．

(1) 文字数の長さが異なるときは，短い方の文字列の末尾にスペースを追加して文字数を等しくする
(2) 2 個の文字列を先頭から 1 文字ずつ比較していって，全て同じならば，「等しい (==)」
(3) 異なる文字があれば，最初に異なる文字の ASCII コードを比較して，コード値の大小が文字列の「大 (>)」または「小 (<)」

付録 E の表からわかるように，ASCII コードは，“スペース”＜“数字”＜“英大文字”＜“英小文字”の順で大きくなり，個々の文字は次のような順序になっています．

```
    '␣' < '0' < '1' <...< '9' < 'A' < 'B' <...< 'Z' < 'a' < 'b' <...< 'z'
```

例えば，文字列の比較を使って次のようなプログラムを書くことができます．

```
  character c1*10,c2*20
  c1 = 'abcde'                          !  3 番目が 'c'
  c2 = 'abdce'                          !  3 番目が 'd'
  if (c1 < c2)  print *,'c1 < c2'
  if (c1 == c2) print *,'c1 == c2'
  if (c1 > c2)  print *,'c1 > c2'
```

この出力結果は，「c1 < c2」です．なぜなら，3 番目の文字が初めて異なり，c1 は 'c'，c2 は 'd' ですが，'c'<'d' だからです．なお，c1 は 10 文字，c2 は 20 文字ですが，後ろにスペースを補うので，この例の結果には無関係です．

6.4 出力における文字列の利用

最もよく文字列を使う場面は，print 文や write 文の中に入れて表示の補助に使うことでしょう．例えば，

```
  real spd,pres
  spd = 10.0
  pres = 960.0
  print *,' Wind Speed = ',spd,'  Pressure = ',pres
```

と書けば，出力が

```
   Wind Speed =    10.0000000000000        Pressure =    960.000000000000
```

のようになって，どの数値がどの変数の値なのかが一目でわかります．

ここで，文字列はその長さに合わせて出力されています．これは，print 文が文字列に入っている文字数の情報に合わせて出力するからです．この機能は，format で文字列の出力形式を指定するときにも使うことができます．文字列の出力指定には，A 編集 (表 5.1) を用いますが，A 編集は，「a」だけ書くと，文字列の文字数が出力幅になります．例えば上記の print 文を，

```
  print 600,' Wind Speed = ',spd,' m/s'
  print 600,' Pressure = ',pres,' hPa'
600 format(a,f8.2,a)
```

と書けば，次のような文字幅に合わせた出力結果が得られます．

```
   Wind Speed =    10.00 m/s
   Pressure =   960.00 hPa
```

5.3 節の format の説明で紹介しましたが，出力形式指定を print 文や write 文の中に埋め込む書式も，文字列の利用方法の一つです．例えば，

```
   print 600,x
   write(10,600) x
600 format(' x = ',es12.5)
```

というプログラムは次のように書き換えることができます．

```
   print "(' x = ',es12.5)",x
   write(10,"(' x = ',es12.5)") x
```

すなわち，format 文のかっこ以下を，両端のかっこを含めて文字列にして print 文や write 文の書式指定の位置 (*form*) に書き込みます．このとき，文字列変数を使えば，複数の場所で同じ format を使うときに便利です．

```
   character :: form*20="(' x = ',es12.5)"
    .......
   print form,x
   write(10,form) x
```

文字列変数を利用すれば，出力内容に応じて実行時に format を変更することも可能です．例えば，

```
   character form*20
    .......
   if (abs(x) >= 1.e5) then
     form = "('x = ',es12.5)"
    else
     form = "('x = ',f10.5)"
   endif
   print form,x
```

とすれば，x の絶対値が 10^5 以上のときには es12.5 編集で，さもなくば f10.5 編集で出力されます．

また，部分文字列を利用して変更することも考えられます．例えば，

```
   character form*20
    .......
   form = "('x = ',f10.5)"
   if (x >= 100.0) form(10:13)='12.3'        ! 10.5 から 12.3 に変更
   print form,x
```

のように書けば，x が 100.0 以上のときは f12.3 編集で，さもなくば f10.5 編集で出力されます．

6.5 数値・文字列変換

ここまでは write 文や read 文の「書式指定」に文字列を使用していましたが，「装置番号」の位置に文字列を記述することもできます．これは「文字列」から「数値」への変換，またはその逆変換を行うときに使用します．6.1 節で述べたように，「123」という数値と「'123'」という文字列はコンピュータ内部の表現が異なりますが，処理の過程で 123 という数値からそれに相当する文字を作ったり，逆に'123'という文字列を数値として計算に利用したい場合があります．そもそも，書式付き write 文は「コンピュータ内部の 2 進数」を「文字で表現された 10 進数」に変換して出力する動作であり，書式付き read 文はファイルなどから「文字で表現された 10 進数」を入力して「コンピュータ内部の 2 進数」に変換す

る動作です．装置番号の位置に文字列を利用すると，その変換機能だけを使うことができるのです．

「数値」→「文字列」変換をするには **write 文** を用います．例えば，

```
real x
character ch*20
x = 123.5
write(ch,"(f10.5)") x
```

と書けば，文字列変数 ch に’ 123.50000’ という文字列が代入されます．ただし，文字列に変換するときは，出力文字数以上の長さを持つ文字列変数を用意しておく必要があります．

open 文 (5.6 節) を使ってファイルをオープンするとき，ファイル名は文字列で指定する必要があります．そこで，file01.dat，file02.dat，...，file10.dat という通し番号の付いた 10 個のファイルが存在して，それぞれを装置番号 11〜20 としてオープンするときは，次のように書きます．

```
character name*20
integer m
do m = 1, 10
  write(name,"('file',i2.2,'.dat')") m
  open(10+i,file=name)
enddo
```

ここで，整数 m の出力編集を i2.2 にしているのは，スペースを ’0’ で埋めて，整数の出力を 01，02，... のようにするためです．

逆に，「文字」→「数値」変換をするには **read 文** を用います．例えば，

```
real x
character ch*20
ch = '12345.0'
read(ch,*) x
```

と書けば，実数型変数 x に 12345.0 という「数値」が代入されます．

6.6 文字列に関する組み込み関数

文字列に関する組み込み関数は，6.2 節で紹介した trim や len の他にも色々あります．代表的なものを表 6.1 に示します．本節ではこれらの関数の中で比較的良く使うものの使用方法について説明します．

表 6.1 文字列に関する組み込み関数

文字列関数	引数の型	関数値の型	関数の意味
trim(c)	文字列	文字列	末尾の空白を削除
len(c)	文字列	整数	文字列の文字数
len_trim(c)	文字列	整数	末尾の空白を削除した文字列の文字数
adjustl(c)	文字列	文字列	文字列を左にそろえた文字列
adjustr(c)	文字列	文字列	文字列を右にそろえた文字列
index(c,cp)	2 個の文字列	整数	文字列 c 中で文字列 cp が含まれていれば， その開始位置を返す　無ければ 0 を返す
repeat(c,n)	文字列と整数	文字列	文字列 c を n 個連結した文字列
char(n)	整数	1 文字の文字列	ASCII コード値を与えると， それに対応する半角英数文字を返す
ichar(c)	1 文字の文字列	整数	半角英数 1 文字を与えると， それに対応する ASCII コード値を返す

まず，6.2 節にも出てきた関数 len は，文字列の文字数を取得するための関数ですが，使用の際は注意が必要です．例えば，

```
character c1*10
integer m,n
m  = len('abc')           ! m = 3
c1 = 'abc'
n  = len(c1)              ! n = 10
```

とすると，m の値は 3 ですが，n の値は 10 になります．これは，文字列変数の文字数が宣言時の文字数だからです．末尾のスペースを除去した文字列の長さを取得するには，関数 trim を使って除去してから関数 len で調べるか，関数 len_trim を用います．例えば，上のプログラムで，最後の n への代入文を

```
n=len(trim(c1))  または  n=len_trim(c1)
```

で置き換えれば，n = 3 になります．

最後の char と ichar の使用例を示します．この二つの関数を組み合わせれば，read 文や write 文を使わなくても，文字と数値の変換をすることができます．例えば，2 桁の整数を 3 桁の 8 進数で表示するプログラムは以下のようになります．

```
integer zero,num
character c1*10
num = 12
c1 = '000'
zero = ichar('0')                          ! '0' の文字コードを整数値に変換
c1(3:3) = char(zero+mod(num,8))            ! num の 1 の位の文字
c1(2:2) = char(zero+mod(num/8,8))          ! num の 8 の位の文字
c1(1:1) = char(zero+mod(num/64,8))         ! num の 64 の位の文字
print *,'decimal(',num,') = octal( ',trim(c1),' )'
```

ASCII コードでは，'0','1','2',...,'8','9' の順で値が 1 ずつ増加しています．そこで，先頭の'0' のコード値に 1 桁の数字を加えれば，その数字のコード値が得られます．同様に，大文字の英数字や，小文字の英数字も 1 ずつ増加して並んでいるので，先頭の文字である'A' や'a' のコード値に，その文字の順番の差の数字を加えて逆変換すれば，対応する文字を得ることができます．

演 習 問 題 6

(6-1) 通し番号付き名前の生成

整数 n を与えれば，適当に決めた文字列の末尾に 1〜n までの通し番号を付けて n 要素の文字列配列に代入して戻るサブルーチンを作れ．例えば，20 個の名前として

```
sample01
sample02
   :
sample20
```

を生成するサブルーチンを作成せよ．このとき，数値・文字列変換の `write` 文を使い，2 桁の整数で先頭に 0 を付加する `i2.2` 編集を利用すればよい．

(6-2) 名簿の先頭と末尾

適当な要素数を持つ 10 文字の文字列配列を用意して，それに適当な名前をローマ字で代入しておく．例えば，

```
character(10) city(9)
city(1) = 'Miyagi'
city(2) = 'Kyoto'
city(3) = 'Osaka'
city(4) = 'Tokyo'
city(5) = 'Hokkaido'
city(6) = 'Aichi'
city(7) = 'Okinawa'
city(8) = 'Fukuoka'
city(9) = 'Ehime'
```

のように文字列を文字列配列に代入しておき，配列要素をアルファベット順に並べ替えるプログラムを作成せよ．並べ替えには問題 (3-4) のバブルソートアルゴリズムを利用すればよい．

(6-3) 16 進数表示

引数に整数を与えると，その数を 16 進数で表した文字列に変換して戻り値に代入するサブルーチンを作成せよ．次に，このサブルーチンを用いて 0〜255 の整数を 16 進数表示で 1 行あたり 16 個づつ並べて出力するプログラムを作成せよ．

(6-4) 図形の描画

文字列を使って図形を表示してみよう．例えば，80 文字の文字列変数を用意して，次式で与えられる整数 m で決まる位置の文字を'*' にし，それ以外の文字をスペースにして出力する，という動作を $i = 0$〜n について行えば正弦波が表示される．

$$m = 40 \sin(hi) + 40$$

ただし，$m = 0$ のときには，$m = 1$ に修正する．例えば，$n = 100$，$h = 0.0628$ にして図形を描け．

第 7 章　配列計算式

第 6 章で説明した文字列の処理は，それ以前に説明した数値の計算と少し毛色が違います．数値計算が 1 個の数値と 1 個の数値の間の演算を基本としているのに対し，文字列の処理では，“文字列”という文字コードの集合をひとまとめにして取り扱い，文字列と文字列の連結や文字列変数への代入を一つの式で実行することができます．文字列とは文字コードの配列なのですから，これを数値計算的に処理するならば，本来は 1 文字ずつ処理しなければなりません．文字列の演算が可能なのは，Fortran コンパイラが，“文字列”という数値集合の処理を，内部で 1 文字ずつ処理するプログラムに変換してくれるからです．

Fortran では，この概念を一般の配列に拡張した **配列演算** が可能です．配列演算とは，数値の集合を配列名で代表させ，配列要素と配列要素の演算を，配列名と配列名の演算の形で記述するものです．配列演算を使えば，do 文を使わずに配列を使った計算を記述することができます．これを **配列計算式** と呼びます．本章では，この配列計算式の使い方について説明します．

7.1　基本的な配列計算式

配列演算は，配列の全要素に対して，全て同じパターンの演算をするのが基本です．このため，1 行の配列計算式で用いる配列は，次元と各次元の要素数が全て等しい **同型の配列** でなければなりません．その点に注意すれば，配列の全ての要素に同じ定数を代入するときや，2 個の配列の対応する要素間で全て同じ四則演算をするときに，あたかも配列名が一つの変数であるかのような記述をすることができます．例えば，次のような記述が可能です．

```
real x(10,10),y(10,10),z(10,10)
x = 1.0
y = 2.0
z = x + 3.5*x*y**2
```

このプログラムを実行すると，配列 x の全ての要素は 1.0 になり，配列 y の全ての要素は 2.0 になり，配列 z の全ての要素は 15.0 $(=1.0+3.5\times1.0\times2.0^2)$ になります．このプログラムは do 文を使った以下のプログラムと同じ結果になります[60]．

```
real x(10,10),y(10,10),z(10,10)
integer i,j
do j = 1, 10
  do i = 1, 10
    x(i,j) = 1.0
    y(i,j) = 2.0
    z(i,j) = x(i,j) + 3.5*x(i,j)*y(i,j)**2
  enddo
enddo
```

配列計算式中の定数は，全要素に対して共通の値として計算します．

次のような，組み込み関数を使った配列計算式も可能です．

```
real a(10,10),b(10,10),c(10,10)
 .......
a = sqrt(b*sin(c)/(3.2*c + 1.5e-3))
```

この配列計算式を do 文で表すと，次のようになります．

[60]ただし，ループの処理はコンパイラが自動的に生成するので，実際にこの通りの順序で計算するかどうかはわかりません．あくまでも，このプログラムと同じ結果になるという意味だと考えてください．

```
  do j = 1, 10
    do i = 1, 10
      a(i,j) = sqrt(b(i,j)*sin(c(i,j))/(3.2*c(i,j) + 1.5e-3))
    enddo
  enddo
```

配列計算式においては，式中に含まれる全ての配列が同型でなければならないので，異なる次元や要素数の配列が混じっているとエラーになります．ただし，下限は異なってもかまいません．例えば，

```
  real a(3,3),r(-1:1,0:2)
   .......
  a = 3.14*r**2
```

のように計算することができます．ただし，この配列計算式を do 文で表せば，

```
do j = 1, 3
  do i = 1, 3
    a(i,j) = 3.14*r(i-2,j-1)**2
  enddo
enddo
```

のように，対応する要素の位置がずれるので注意して下さい．

このように，配列計算式を使うとプログラムがシンプルになります．逆に，単一変数の計算式と配列計算式との区別がなくなるので，両者が入り交じるとエラーが見つけにくくなるという欠点もあります．

なお，サブルーチンの引数配列を使って配列計算をするときには注意が必要です．例えば，

```
  subroutine subr(a,b)
    implicit none
    real a(*),b(*)
    a = b**2              ! これはエラーになる
```

のようなプログラムでは，配列計算式はエラーになります．なぜなら，“*”を入れて配列宣言をした場合は，要素数が不定だからです．これに対し，

```
  subroutine subr(a,b,n)
    implicit none
    real a(n),b(n)
    integer n
    a = b**2              ! この場合は OK
```

のように，整合配列を使うと，配列計算が可能になります．

7.2 部分配列

配列名だけを使った配列計算式では全ての要素について計算をしますが，常に全要素の計算が必要とは限りません．そこで，配列の一部を取り出した配列，**部分配列** を指定することができます．部分配列は“:”(コロン) を使って要素番号の範囲を指定します．例えば，1 次元配列 a に対して，

```
  a(n1:n2)
```

と書けば，a(n1)〜a(n2) という n2-n1+1 個の要素から構成された 1 次元配列として配列計算式に使うことができます．また，2 次元配列 b に対して，

```
  b(n11:n12,n21:n22)
```

と書けば, b(n11,n21)〜b(n12,n21)〜b(n11,n22)〜b(n12,n22) という (n12-n11+1)×(n22-n21+1) の 2 次元配列として配列計算式に使うことができます．また，ある次元の要素番号を固定して，

```
b(n,n21:n22)
```

と書けば，b(n,n21)〜b(n,n22) という n22-n21+1 個の要素から構成された 1 次元配列として配列計算式に使うことができます．

逆に，要素番号に “:”だけを記述すると，“その次元の全要素”という意味になります．例えば，

```
b(:,n21:n22)
```

と書けば，第 1 次元は全要素で，第 2 次元が n21〜n22 の範囲の要素から構成された 2 次元配列として配列計算式に使うことができます．このため，全ての要素番号を “:”にすると，配列全体を代表することになります．そこで，全配列要素を使った配列計算式を書くときにも “:”を使って配列であることを明示することができます．例えば，7.1 節の 2 次元配列計算式，

```
a = sqrt(b*sin(c)/(3.2*c + 1.5e-3))
```

は，次のように書くことができます．

```
a(:,:) = sqrt(b(:,:)*sin(c(:,:))/(3.2*c(:,:) + 1.5e-3))
```

この方が，単一変数の計算式と区別できるので良いと思います．本書でもこれ以降はこの書式を使って全要素の配列計算式を記述します．

要素範囲の指定に “:整数値”を追加して，do 文のような増分値を指定することもできます．例えば，1 次元配列 a に対して，

```
a(n1:n2:n3)
```

と書けば，a(n1),a(n1+n3),a(n1+2*n3),... という要素からなる 1 次元配列として配列計算式に使うことができます．この場合，終了要素は n2 とは限らないので注意して下さい．例えば，次のように書くことができます．

```
real a(10),b(5)
 .......
b(1:5) = a(1:10:2)
```

この結果は，b(1)=a(1)，b(2)=a(3)，b(3)=a(5)，b(4)=a(7)，b(5)=a(9) という 5 個の代入文と等価になります．増分値は負数を与えることも可能なので，次のように逆向きに代入させることも可能です．

```
real a(10),c(10)
 .......
c(1:10) = a(10:1:-1)
```

この結果は，c(1)=a(10)，c(2)=a(9)，...，c(10)=a(1) という 10 個の代入文と等価になります．

配列計算式で部分配列を使うときは，その部分配列がその他の配列と同型であれば良いので，宣言文での次元や要素数が一致している必要はありません．例えば 1 次元配列なら，

```
real a(10),b(5)
 .......
a(3:5) = b(1:3)**2
```

のように書くことができます．また，2 次元配列の部分配列を使って，

```
real a(10,10),b(5,4),c(10)
 .......
a(3:5,6:8) = b(1:3,1:3)**2
```

などと書くことができます．この配列計算式を do 文で表せば，

```
do j = 6, 8
  do i = 3, 5
    a(i,j) = b(i-2,j-5)**2
  enddo
enddo
```

となります．ある次元の要素番号を固定した 2 次元配列は 1 次元配列として扱えるので，2 次元配列と 1 次元配列が混在した計算も可能です．例えば，

```
real a(10,10),c(10)
 .......
c(3:8) = a(3,3:8)**2
```

と書くことができます．この配列計算式を do 文で表せば，

```
do j = 3, 8
  c(j) = a(3,j)**2
enddo
```

となります．

なお，部分配列を使った配列計算式で，左辺と右辺に同じ配列を使う場合には，単純に do 文で置き換えた動作と結果が異なる場合があるので注意が必要です．例えば，

```
real a(10)
do i = 1, 10
  a(i) = i
enddo
a(2:10) = a(1:9)            ! この配列計算式は単純な do 文で置き換えられない
print *,(a(i),i=2,10)
```

というプログラムを考えます．この中の配列計算式 a(2:10)=a(1:9) を do 文にすると，

```
do i = 1, 9
  a(i+1) = a(i)
enddo
```

と置き換えられそうですが，実際にやってみると print 文の出力結果は異なります．配列計算式の場合は 1 から 9 までの異なる数字が出力されるのに対し，この do ループで置き換えると全て 1 が出力されます．これは，配列計算式が do ループのように 1 要素ずつ計算と代入をくり返すのではなく，まず右辺の配列要素計算を全て行って結果を補助配列に保存しておき，その後で保存した補助配列から左辺の配列への代入を行うためです．

このため，配列代入機能を利用すると，次のように配列要素の順番を逆転させるのも簡単です．

```
a(1:10) = a(10:1:-1)
```

これと同じ動作をするプログラムを 1 回の do ループで書くのは難しいです．この代入機能は，配列計算式を使う利点の一つです[61]．

部分配列は要素が等間隔に並んでいますが，整数の 1 次元配列を使って要素指定をすれば，任意の順番で指定した配列を作ることができます．例えば，次のような指定ができます．

[61] ただし，配列要素数が非常に大きくて，コンピュータのメモリ利用可能限界に近くなると，補助配列の生成のおかげでメモリオーバーになる可能性もあります．

```
real a(10)
integer m(3),i
do i = 1, 10
  a(i) = i
enddo
m(1) = 7;  m(2) = 2;  m(3) = 5
print *,a(m)
```

最後の出力結果は,

```
7.00000000000000  2.00000000000000  5.00000000000000
```

となります．すなわち，a(m) とは，a(m(1)),a(m(2)),a(m(3)) という要素数 3 の配列のことです．このように整数配列で要素指定をすると，要素指定配列の要素数を持つ配列になります．a(m(2:3)) のように部分配列を使って要素指定をしたり，7.4 節の配列構成子を使って要素指定をすることもできます．

配列計算式では，計算結果を補助配列に一時保存してから一度に代入するので，部分配列の代入文では注意が必要である，という話をしましたが，これはサブルーチンの引数に部分配列を与えるときにも言えます．4.5 節で，サブルーチンの引数に配列を与えるとき，配列名を与えることは，その配列の先頭要素アドレスを与えることであり，配列要素を与えることは，その配列要素を先頭アドレスとする配列を与えることなので，どちらも元の配列における並びが重要であるという話をしました．

これに対し，部分配列を引数に与えた場合には，次のような処理を行います．

```
部分配列の要素    →  同型の補助配列に一度代入する
                  →  その補助配列の先頭アドレスをサブルーチンに与える
```

このため，サブルーチン側で受け取る配列は，必ずしも元の配列の並びと同じになるとは限りません．

例えば，b(10,10) という宣言をした 2 次元配列の 1 次元部分配列 b(5,3:5) を，次のように 4.5 節のサブルーチン sub に与えた場合を考えます．

```
call sub(b(5,3:5))
```

この結果は，b(5,3),b(5,4),b(5,5) という要素数 3 の 1 次元配列を与えたものと同じになります．すなわち，元の配列における並びとは無関係です．

このため，2 次元配列 b(10,10) の j 列目を与えるための call 文,

```
call sub(b(1,j))
```

と，部分配列を使った call 文,

```
call sub(b(:,j))
```

とは，結果は同じですが，内部の動作が異なることがあります．

また，i 行目の要素，b(i,1)，b(i,2)，...，b(i,10) を処理したい場合には,

```
call sub(b(i,:))
```

と書くことができます．

7.3 where 文による条件分岐

配列演算は便利ですが，全ての要素について同じ形の計算をするので，要素の条件に応じて異なる処理をさせることができません．そこで，配列要素の条件に応じて動作を分岐させるための **where 文** が用意されています．where 文は，以下のような形式です．

```
where (配列条件) 配列計算式
```

これは，“配列条件”に合った要素に対してのみ，“配列計算式”を実行するという意味です．例えば，

```
real a(10,10),b(10,10)
 .......
where (a(:,:) > 0) b(:,:) = a(:,:)**2
```

のように書きます．この where 文の部分を do 文で表せば，次のようになります．

```
do j = 1, 10
  do i = 1, 10
    if (a(i,j) > 0) b(i,j) = a(i,j)**2
  enddo
enddo
```

where 文はブロック構文にすることもできます．**ブロック where 文**では，次のように **else where 文**を追加して，条件に一致しない場合の記述をすることもできます．

```
where (配列条件 1)
  配列計算式 1
   .......
 else where (配列条件 2)
  配列計算式 2
   .......
 else where
  配列計算式 0
   .......
endwhere
```

この場合，配列条件 1 を満足する要素は配列計算式 1 のブロックを実行し，配列条件 1 を満足せず，配列条件 2 を満足する要素は配列計算式 2 のブロックを実行し，... という具合に続いて，全ての条件を満足しない要素は配列計算式 0 のブロックを実行するという動作になります．中間の else where に関するブロックは省略可能です．if 文と違って，“then”が不要なこと，“全ての条件以外”を表す文が“else where”であること，ブロックの最後に **endwhere** 文を付加すること，などに注意して下さい[62]．

例えば，

```
real a(10,10),b(10,10)
 .......
where (3.0*b(:,:) > 5.0)
  a(:,:) = b(:,:)**2
 else where
  a(:,:) = b(:,:)**3
endwhere
```

のように書くことができます．このブロック where 文を do 文で表せば，次のようになります．

```
do j = 1, 10
  do i = 1, 10
    if (3.0*b(i,j) > 5.0) then
      a(i,j) = b(i,j)**2
     else
      a(i,j) = b(i,j)**3
    endif
  enddo
enddo
```

[62] endwhere は“end where”と離して書いても OK です．

where ブロック中で使用する配列は，全て同型でなければなりません．このため，where ブロックの中に，単一変数の計算式や，次元や要素数の異なる配列計算式を入れることはできません．

7.4 配列構成子

配列演算を使えば，do 文を使わなくても配列要素の計算ができますが，これまでの書式では配列要素ごとに異なる定数を代入することはできません．そこで，1 次元配列だけですが，定数や変数などを並べて明示的に配列要素を与える **配列構成子** が用意されています．n 個の要素からなる 1 次元の配列構成子は以下の形式です．

```
(/数値 1, 数値 2,..., 数値 n/)
```

この時，n 個の数値は全て同じ型の数値型でなければなりません．例えば，

```
real a(5)
a(:) = (/1.0,2.0,3.0,4.0,5.0/)
```

と書けば，右辺は実数型の 1 次元配列構成子であり，この配列計算式は，a(1)=1.0, a(2)=2.0, a(3)=3.0, a(4)=4.0, a(5)=5.0 という 5 個の代入文を実行したことに相当します．

数値の位置には変数や計算式を書くこともできます．例えば，

```
real a(5),b
b    = 2.5
a(:) = (/b,2*b,3*b,4*b,5*b/)
```

と書けば，この配列計算式は，a(1)=b, a(2)=2*b, a(3)=3*b, a(4)=4*b, a(5)=5*b という 5 個の代入文を実行したことに相当します．

部分配列で指定することも可能です．例えば，

```
real a(5),c(5)
a(:) = (/1.0,2.0,3.0,4.0,5.0/)
c(:) = (/a(3:5),a(1:2)/)
```

と書けば，この配列計算式は，c(1)=a(3), c(2)=a(4), c(3)=a(5), c(4)=a(1), c(5)=a(2) という 5 個の代入文を実行したことに相当します．

この配列構成子は，数値が全て定数であれば，次のように拡張宣言文の初期値代入に使うこともできます．

```
real :: a(5)=(/1.0,2.0,3.0,4.0,5.0/)
```

しかし，配列要素が多いと，数値を並べて配列構成子を記述するのに手間がかかります．そこで，5.2 節で説明した **do 型並び** を使って数値の設定をすることができます．do 型並びには，増分値なしと増分値付きの 2 種類があります．増分値を省略した時の増分値は 1 です．

```
(データ 1, データ 2,..., 整数型変数=初期値, 終了値)              ! 増分値なし
(データ 1, データ 2,..., 整数型変数=初期値, 終了値, 増分値)      ! 増分値付き
```

この形式を配列構成子中に記述すると，まず整数型変数 (カウンタ変数) を初期値にして，データ 1，データ 2，... と並べ，次に，カウンタ変数に増分値を加えて，再度，データ 1，データ 2，... と並べ，カウンタ変数が終了値以下の間にくり返して得られる数値を並べたことに相当します．

例えば，

```
(/(n, n=1,5)/)
```

という do 型並びを使った配列構成子は，整数型の配列構成子，

```
(/1,2,3,4,5/)
```

と同じです．また，

```
(/(n, n**2, n=1,5)/)
```

という do 型並びを使った配列構成子は，整数型の配列構成子，

```
(/1,1,2,4,3,9,4,16,5,25/)
```

と同じです．増分値を指定して，

```
(/(n, n**3, n=1,8,2)/)
```

という do 型並びを使った配列構成子は，整数型の配列構成子，

```
(/1,1,3,27,5,125,7,343/)
```

と同じです．

複数の do 型並びを並べたり，通常の数値と混在させることも可能です．例えば，

```
(/(real(n), n=0,2),1.5,1.8,(real(n), n=3,5)/)
```

という実数型の配列構成子は，

```
(/0.0,1.0,2.0,1.5,1.8,3.0,4.0,5.0/)
```

と同じです．この時，型変換関数の real(n) を単に n と書くとエラーになるので注意して下さい．これは，n が整数型なので，n だけを書くと一つの配列構成子の中に実数型と整数型が混在するからです．

do 型並びは多重にすることも可能です．例えば，

```
(/((i+j,i=1,3),j=1,4)/)
```

という do 型並びを使った整数型の配列構成子は，

```
(/2,3,4,3,4,5,4,5,6,5,6,7/)
```

と同じです．多重にした場合，カウンタ変数は内側が先に進みます．また，do 型並びを do ブロックの中に入れる時には，カウンタ変数が do 文のカウンタ変数とも重複しないようにしなければなりません．

本節最初の例は，次のように do 型並びを使って書くことができます．

```
real a(5)
a(:) = (/(n, n=1,5)/)
```

ここで，右辺は整数型の配列で，左辺は実数型の配列ですが，この場合は全要素に対して整数型から実数型への変換が行われて代入されるのでエラーにはなりません．なお，do 型並びは拡張宣言文での初期値代入に使うこともできますが，カウンタ変数はその宣言文より前に宣言されている必要があります．

配列構成子は 1 次元配列しか用意されていません．このため，2 次元以上の配列計算で使う時は，1 次元ごとの部分配列に代入するか，配列の形状を変換する組み込み関数 reshape を使います．reshape については 7.5 節で説明します．

7.5 配列に関する組み込み関数

配列を引数にする組み込み関数も色々用意されています．まず，配列要素を利用した計算に関する関数を表 7.1 に示します．

表 7.1 配列要素の計算に関する組み込み関数

配列関数	引数の型	関数値の型	関数の意味
sum(a)	配列*	配列要素の型	全ての配列要素の和
product(a)	配列*	配列要素の型	全ての配列要素の積
minval(a)	配列*	配列要素の型	全ての配列要素の最小値
maxval(a)	配列*	配列要素の型	全ての配列要素の最大値
dot_product(a,b)	2 個の 1 次元配列[63]	配列要素の型	2 個の 1 次元配列の内積
matmul(a,b)	2 個の 2 次元配列[64]	2 次元配列	2 個の 2 次元配列の行列積

例えば，

```
real a(5)
integer i
a(:) = (/(i, i=1,5)/)
print *,sum(a)                  ! 配列 a の全要素の合計
```

と書けば，配列 a の全要素の合計 (15.0) が出力されます．部分配列を利用すれば，指定した範囲の配列要素だけを使った計算も可能です．

同様に，

```
real a(10)
integer i,n
a(:) = (/(i, i=1,10)/)
n = 5
print *,product(a(1:n))         ! 部分配列の要素 a(1)〜a(n) の積
```

と書けば，a(1) から a(n) までの積 (120.0) が出力されます．

表 7.1 で引数の型に “*”の付いた関数を使う場合，引数配列の後に整数型の引数を追加すると，その整数値が指定する次元に関してのみ計算をすることができます．例えば，

```
real b(3,4),x,y(4),z(3)
integer i,j
do j = 1, 4
  b(:,j) = (/(sin(0.5*(i+j)), i=1,3)/)
enddo
x = maxval(b)                   !  配列 b の全要素の最大値
y(:) = maxval(b,1)              !  配列 b の第 1 次元方向の要素の最大値
z(:) = maxval(b,2)              !  配列 b の第 2 次元方向の要素の最大値
```

と書けば，x には 2 次元配列 b の全要素の最大値が代入されますが，1 次元配列 y には，b の第 2 次元を固定して第 1 次元方向に要素を比較したときの最大値がそれぞれ代入されます．また，1 次元配列 z には，b の第 1 次元を固定して第 2 次元方向に要素を比較したときの最大値がそれぞれ代入されます．すなわち，次元を指定すると，その次元を抜いた要素数の配列が結果になります．

次に，配列情報に関する関数を表 7.2 に示します．表 7.2 で，引数の型に “*”の付いた関数は，引数配列の後に次元を示す整数型の引数を追加して，その次元に関する値を取り出すこともできます．

[63] a の要素数と b の要素数は等しくなければなりません．

[64] 行列積の計算上，a の第 2 要素数と b の第 1 要素数は等しくなければなりません．なお，a か b のどちらかは 1 次元配列でも計算できます．この場合，結果は 1 次元配列になります．

表 7.2　配列情報に関する組み込み関数

配列関数	引数の型	関数値の型	関数の意味
minloc(a)	配列	整数型配列	全ての配列要素の最小値の位置
maxloc(a)	配列	整数型配列	全ての配列要素の最大値の位置
lbound(a)	配列*	整数型配列	配列の各次元の最小要素番号リスト
ubound(a)	配列*	整数型配列	配列の各次元の最大要素番号リスト
shape(a)	配列	整数型配列	配列の各次元の要素数リスト
size(a)	配列*	整数	配列の全要素数
reshape(a,s)	配列と整数型配列	a の型の配列	配列 a を配列 s で指定された型の配列に変換する
allocated(a)	配列	論理	配列 a が割り付けられていれば真，さもなくば偽 (4.8.4 節)

例えば，

```
real array(3,4)          !  3 × 4 の配列
integer m,n1,n2
m  = size(array)         ! 全要素数 3 × 4=12
n1 = size(array,1)       ! 第 1 次元要素数 3
n2 = size(array,2)       ! 第 2 次元要素数 4
```

と書けば，m には，array の全要素数である 12 が代入され，n1 には array の第 1 次元の要素数である 3 が，n2 には array の第 2 次元の要素数である 4 が代入されます．

また，表 7.2 で関数値の型が “整数型配列” になっている関数は，引数に与えた配列の次元を要素数に持つ 1 次元整数型配列が戻り値です．その際，その 1 次元配列の各要素には，引数配列の各次元の情報が代入されています．例えば，下限と上限を取得するための関数，lbound と ubound は以下のように使います．

```
real bb(-10:10,4)          !  2 次元配列
integer blow(2),bupp(2)    !  2 次元なので要素数 2 の配列
blow(:) = lbound(bb)
bupp(:) = ubound(bb)
print *,blow,bupp
```

ここで，2 次元配列 bb は，第 1 次元の下限が −10，上限が 10，第 2 次元の下限が 1，上限が 4 ですから，配列構成子で書けば，blow は (/-10,1/)，bupp は (/10,4/) になります．このとき，ubound(bb,2) のように，次元に関する引数を加えると，その結果は 1 個の整数 (この例では 4) になります．

reshape は，配列の形状を変換する関数です．そもそも，どんな次元の配列もメモリ上では 1 次元的に並んでいるので，2 次元配列 a(3,4) と 1 次元配列 b(12) のように，全要素数が等しい配列は同じように取り扱えるはずです．しかし，配列演算は次元および各次元の要素数が等しい同型の配列間でしか許可されていません．そこで，a(3,4) と b(12) の間で演算をするには，形式上，同型になるような変換が必要です．これを実行するのが関数 reshape です．reshape には，最初の引数に変換したい配列を与え，2 番目の引数に変換後の形状を表す 1 次元配列を与えます．ここで形状を表す 1 次元配列とは，変換後に，k1×k2×...×kn の n 次元配列にする場合，

```
(/k1,k2,...,kn/)
```

で与えられる要素数 n の 1 次元整数型配列のことです．例えば，

```
real a(3,4)
integer i,j
a(:,:) = reshape((/((sin(0.5*(i+j)), i=1,3), j=1,4)/), (/3,4/))
```

と書くことができます．reshape の第 1 引数は do 型定数なので 1 次元配列ですが，これを第 2 引数の (/3,4/) で 3×4 の 2 次元配列に変換するよう指定しているわけです．

この配列の形状を表す 1 次元配列は，関数 shape を使って取得することができます．例えば，上記の a(3,4) という宣言をした配列に対し，shape(a) は，1 次元整数型配列，(/3,4/) になります．そこで，上記の配列計算式は次のように書くことができます．

```
a(:,:) = reshape((/((sin(0.5*(i+j)),i=1,3),j=1,4)/),shape(a))
```

しかし，これではまだ do 型変数における i や j の範囲指定が定数のままです．そこで，ここは先ほど説明した関数 size で書き直すと良いでしょう．

```
a(:,:) = reshape((/((sin(0.5*(i+j)),i=1,size(a,1)),j=1,size(a,2))/),shape(a))
```

これなら，宣言文の要素数を変更するときに，この配列計算式を変更する必要はありません．

3.3.4 節で紹介した論理型のデータを配列にしておき，表 7.3 のような組み込み関数を利用すると，大量の論理演算を一度に実行することができます．

表 7.3　論理型の配列に関する組み込み関数

配列関数	引数の型	関数値の型	関数の意味
all(a)	論理型配列*	論理型	全ての配列要素の AND
any(a)	論理型配列*	論理型	全ての配列要素の OR
count(a)	論理型配列*	整数型	配列中の “真”の数

これらの関数は，引数の型に “*”があることからわかるように，引数配列の後に次元を示す整数型の引数を追加して，その次元に関する論理演算値を取り出すこともできます．

例えば，1 次元配列に代入されている数値に対し，それらが全て 50 より大きいかどうかを一度に判定するには，以下のように書くことができます．

```
real fdata(100)
logical(1) flag(100)
...
flag(:)  = fdata(:) > 50        ! 論理演算も配列計算をすることができる
if (all(flag)) print *,'All Data > 50'
```

このように，論理演算も配列計算によって一度に判定することができ，その判定結果を代入した論理型の配列を使えば，それらを一度に判定することが可能になります．

演 習 問 題 7

(7-1) 正多角形の面積

外接円の半径が 1 の正多角形の面積を，正 3 角形から正 n 角形まで配列計算式を使って全て計算せよ．ここで，外接円の半径が r のとき，正 n 角形の面積 S_n は次式で与えられる．

$$S_n = nr^2 \cos\frac{\pi}{n} \sin\frac{\pi}{n}$$

計算は，まず $n-2$ 要素の実数型 1 次元配列 `a(3:n)` に 3〜n の整数を代入し，上式を配列計算式にして別の $n-2$ 要素の実数型配列 `s(3:n)` に代入すれば良い．

(7-2) 奇数の合計

n 要素の実数型 1 次元配列 `a(n)` に 1,3,...,$2n-1$ という奇数を代入し，それぞれの平方根を別の n 要素の実数型配列 `b(n)` に代入するプログラムを配列計算式だけで書け．また，関数 `sum` を使って，配列 `a(n)` と `b(n)` の総和をそれぞれ計算せよ．

(7-3) 2 次元配列の指定した行と行を入れ替えるサブルーチン

問題 (4-1) の実数配列 `a(n)` と `b(n)` の要素を全て交換するサブルーチンを，`do` 文は使わずに配列計算で実行する形式にせよ．その時に必要な補助配列は，引数で宣言する動的割り付けを利用せよ．

このサブルーチンを使って，n 行 n 列の 2 次元配列 `a(n,n)` に対し，

$$a_{ij} = i + n(j-1)$$

のように値を代入して，行番号 i_1 と i_2 の行要素を交換するプログラムを部分配列指定を使って記述せよ．

(7-4) モンテカルロ法による体積計算

モンテカルロ法を使った球の体積計算を，配列計算式を使った以下の手順で行え[65]．

(1) まず，3 個の n 要素の 1 次元実数型配列，`x(n)`，`y(n)`，`z(n)` を用意して，これらの全要素に 4.8.5 節で説明したサブルーチン `random_number` を使って乱数を代入する．
(2) 次に，(`x(n)`，`y(n)`，`z(n)`) を 3 次元座標として，それぞれの原点からの距離を実数型配列 `r(n)` に代入する．
(3) 次に，配列 `r(n)` 中で 1 以下か否かを調べた結果を論理型配列 `s(n)` に代入する．
(4) 次に，論理型配列 `s(n)` と関数 `count` を使って，配列 `r(n)` が 1 以下になるような要素の数 n_1 を計算する．

最後に得られた n_1 と要素数 n の比率が，半径 1 の球の体積の 1/8 (=$\pi/6$) に近くなることを確かめよ．

[65] モンテカルロ法の原理は，問題 (4-4) 参照．

付録 A　gfortran を用いたコンパイルから実行までの手順

gfortran は Fortran95 の文法で書いたプログラムをコンパイルできる代表的なフリーの Fortran です．多くの Linux ディストリビューションに付属しているし，Windows 版や Mac OS 版も配布されているので，お手持ちのパソコンにインストールすれば，手軽に Fortran を利用することができます．ここでは gfortran を使用したコンパイルから実行までの手順について述べます．

まず，Fortran プログラムを作るときには，ファイル名の最後に “.f90”または “.f95”を付けます．すなわち，

```
文字列.f90　または　文字列.f95
```

という名前にします．このファイル名の最後に付加した “ドット (.)+文字”の部分を拡張子と呼びます．作成したプログラムファイルを計算機で実行させるには **コンパイル** と **リンク** という二つの過程が必要です．

“コンパイル”というのは，Fortran や C 言語など，人間が理解できる言語で書かれたプログラムをコンピュータが理解できる機械語に翻訳することです．コンパイルするアプリケーションを **コンパイラ** といいます．フリー Fortran コンパイラの代表が gfortran です．

gfortran でプログラムをコンパイルをするときは，gfortran コマンドを使って，

```
gfortran プログラムファイル名
```

と入力します．例えば，“test1.f90”というファイル名のプログラムをコンパイルするときは

```
gfortran test1.f90
```

と入力します．コンパイル時に文法エラーなどが見つかると，エラーメッセージを出力して終了します．

gfortran コマンドは，コンパイルが成功すると，引き続きリンクを行います．プログラムはコンパイルしただけでは実行できません．コンパイラはプログラミング言語を機械語に翻訳するだけであり，複数のルーチンを結合して OS 上で起動可能な形式にする作業までは行わないからです．この結合処理が “リンク”です．単にプログラム中のルーチンのリンクだけではありません．入出力文の `read` や `write`，組み込み関数の `sin` や `log` 等は，標準ライブラリルーチンとして用意されているので，必要に応じてこれらのリンクも行います[66]．

リンクにも成功すると，最後に “a.out”という名前のファイルが作成されます．これが OS 上で直接実行させることができる機械語で書かれたファイルで，これを **実行形式ファイル** といいます．もしリンクに失敗した場合は，エラーメッセージを出力して終了します．この時 a.out は作成されません．

実行形式ファイルは，一般のコマンドのようにファイル名を入力することでプログラムを実行することができます．ただし，次のように頭に “./”を付ける必要があります[67]．

```
./a.out
```

プログラムが正常に動作すれば，それに書かれた一連の計算を行って終了します．計算の途中に `print` 文の記述があれば，結果を画面に出力するし，`write` 文の記述があれば，指定したファイルに書き出します．また，標準入力の `read` 文の記述があれば，その時点でプログラムが一時停止して入力待ちの状態になります．この状態で必要な数値をキーボードから入力すれば計算は再開します．

以上が gfortran を用いた最も単純なコンパイルからプログラム実行までの手順ですが，gfortran コマ

[66] コンパイラである gfortran 自体はリンクする機能を持っていませんが，gfortran 内部から OS に付属しているリンク用アプリケーション (リンカ) を呼び出すことができるので，コンパイル後にリンクを実行してくれます．

[67] “./”は “現在のディレクトリにあるファイル”という意味です．このあたりの詳細は，Linux などの解説書を見て下さい．使用しているコンピュータの設定によっては不要な場合もあります．

ンドにファイル名を与えただけの命令では，どんなプログラムをコンパイルしても a.out という同じ名前の実行形式ファイルになってしまうので不便です．また，Fortran プログラムは計算速度が重要なので，最適化をしてできるだけ高速に処理する方が良いでしょう．さらに 1.6 節で述べたように数値計算をするときは倍精度実数を使うべきなので，自動倍精度化オプションも必要です．

このため，gfortran でプログラムをコンパイル・リンクする時は，以下のようなオプションを付けることをお勧めします．

```
gfortran -O -fdefault-real-8 test1.f90 -o test1
```

`-O` が最適化のオプション，`-fdefault-real-8` が自動倍精度化のオプションです．また，`-o` のオプションの次の文字列 (ここでは `test1`) は実行形式ファイル名指定です．大文字の`-O` と小文字の`-o` を間違わないようにして下さい．

この場合，コンパイルとリンクが正常に終了すると，`-o` オプションで指定した “test1” という名の実行形式ファイルが作成されます．このため，プログラムを実行するには，

```
./test1
```

と入力することになります．

Linux を使っている環境ならば，以下のような内容のスクリプトを作成しておけば便利だと思います．

```
gfortran -O -fdefault-real-8 $1.f90 -o $1
```

付録 B　エラー・バグへの対処法

プログラムを開発する際の最大の問題は **エラー** (Error) の発生です．コンパイル・リンク・実行という一連の過程の中でそれぞれエラーが発生する可能性があります．エラーはプログラム開発にとって付き物であり，避けては通れません．エラーの出ないように慎重にプログラムを書くことはもちろんですが，出たら出たでそれらに如何に素早く対処できるかが腕の見せ所です．ここではエラーをまとめて，それぞれの対処法を示します．なお，具体的なエラーメッセージは gfortran の出力を利用していますが，他のコンパイラでも形式が違うだけで出力情報の内容はあまり変わらないと思います．

(1) コンパイルエラー

コンパイルとは，作成したプログラムを文法に従って機械語に翻訳することです．このため，コンパイルエラーとは文法的な間違いのことです．プログラムに文法的間違いがあれば，コンパイラがエラーメッセージを出力して終了します[68]．コンパイラのエラーメッセージ形式はコンパイラによって違いますが，gfortran では次のような形式で出力されます．この例は，“test1.f90” というプログラムファイルをコンパイルしたときのエラー出力です．

```
test1.f90:15.7:
   do i = 1, 100
      1
Error: Symbol 'i' at (1) has no IMPLICIT type
```

[68] コンパイラが出すメッセージには “Warning”(警告) と “Error”(エラー) があります．Warning は「このままでも実行できるけど，ひょっとすると予期しない結果が起こる可能性があります」という意味なので，必ずしも修正する必要はありません．このため，メッセージが Warning だけのときは強制終了せずにリンクの作業に移ります．しかし，文法的には間違いがなくても，作成者の意図とは違ったプログラムになっている可能性があるので，念のためにプログラムを確認した方が良いでしょう．

以下に，この出力の読み方を示します．

(1) 1 行目はエラーのあるプログラムファイル名 (この例では，test1.f90) とエラーの位置
　この例では，エラーの位置が 15.7 となっていますが，15 はプログラムの行番号，7 は左から数えた文字の位置を示す数値です．
(2) 2 行目はエラーのあるプログラム文のコピー
(3) 3 行目は “1”を上向き矢印のように使って，2 行目の文のエラーが起きている位置を指定
(4) 4 行目は 3 行目が示した部分の具体的なエラーの説明
　このメッセージを頼りにエラーを見つけてプログラムを修正します．

この例の 4 行目のメッセージは「3 行目の 1 が示している変数 i は，暗黙の型がない」という意味です．これは，変数 i が integer 文などで宣言されていないことが原因です．

コンパイルエラーは，エラーメッセージに従って修正すればいいので，それほどの手間ではありません．ただし，ある文に文法エラーがあると，その文が存在しない状態で引き続きコンパイルを行うため，その波及効果で下方のプログラムがエラーになることがあります．例えば宣言文に誤りがあると，その宣言文で宣言している変数がコンパイラにとっては宣言されていないことになり，その変数を使っている文が全てエラーになります．エラーメッセージが多い時は，一度に全部チェックしないで，ある程度修正したら再度コンパイルして，エラーが出るかどうかを確認した方が良いと思います．

(2) リンクエラー

リンクとは，別々に翻訳されたルーチンを結合して実行形式ファイルにする作業のことです．よって，リンクエラーは用意されていないサブルーチンや関数を記述したときに起こります．例えば，

```
program test1
  implicit none
  real x,y
  x = 5
  y = 100
  call subr(x,y)          ! subr が用意されていないとリンクエラーになる
  print *,x,y
end program test1
```

のようなプログラムを作って，これだけを単独でコンパイル・リンクすると

```
/tmp/ccy3lcdo.o: In function ‘MAIN__’:
test1.f90:(.text+0x6d): undefined reference to ‘subr_’
collect2: ld returned 1 exit status
```

のようなメッセージが出力されます．以下に，この出力の読み方を示します．

(1) 1 行目はエラーが起こっているルーチンの表示
　この例では “MAIN”，すなわちメインプログラム中で起こっていることが示されています．
(2) 2 行目はエラーの原因
　このメッセージを頼りにエラーを見つけてプログラムを修正します．

この例の 2 行目のメッセージには「subr という名への参照が定義されていない」とあります．これは，メインプログラム中でコールしている subr という名のサブルーチンが用意されていないのが原因です[69]．

標準の組み込み関数が用意されていないことはまずないので，リンクエラーのほとんどはプログラム中で呼び出したサブルーチン名や関数名の書き間違いが原因です．リンクエラーのメッセージにはコン

[69] メッセージ中では MAIN__とか，subr_のように名前の後ろにアンダースコアが付いていますが，これはコンパイラが自動的に付加するものなので無視して下さい．リンクは gfortran がするのではなく，ld というリンカがするのですが，リンカは OS によって異なるので，gfortran を使っていてもこの形式でエラーメッセージが出るとは限りません．

パイルエラーのようなエラー行の表示はありませんが，エラーメッセージの中に見つからなかったサブルーチンや関数の名前が表示されているので，エディタの検索機能を使えば，問題のある call 文などを特定するのはさほど困難ではありません．

(3) 実行時エラー

一番厄介なのが実行時のエラーです (こういうプログラムは“バグ (虫)”があるといいます)．なぜなら，コンパイルエラーやリンクエラーのようなエラーの場所を特定する情報が出力されないので，どこでエラーが起こっているのかを探す作業が大変だからです．実行時エラーを探すには，プログラムを詳細にチェックするしかありません (この作業を「バグを取る」という意味で“デバッグ”といいます)．

それでも，答えが合っているか否かは別として，プログラムがなんとか終了すれば良いのですが，バグによっては実行した後でうんともすんともいわなくなってしまう場合があります．これは無限ループに入ってしまったか，暴走したかのどちらかだと考えられます．通常，このような状態で実行を強制的に終了させるには，“コントロールキー”を押しながら“C キー”を押します．まず間違いなく止まります．

もう一つ，以下のようなメッセージを出して強制的に終了する場合があります．

```
Segmentation fault
```

これはプログラムではなく OS が出力しているメッセージで，主として実行中のプログラムがそれ以外の実行中のプログラムのメモリ領域を破壊しようとした時に起こります．例えば，次のように配列宣言の範囲外の要素に値を代入しようとすると起こることがあります[70]．

```
real a1(10)
integer i
do i = 1, 100000
   a1(i) = i
enddo
```

この例では，10 個しか用意されていない配列に，100000 番目まで値を代入しようとしているのですから，どこのメモリに値を代入しようとしているのか不明なのが問題なのです．

また，サブルーチンの引数に型の合っていない数値を与えたり，戻り値を与える引数に定数を与える，などの不注意なサブルーチンコールでも Segmentation fault が起こる可能性があります．Segmentation fault は，実行時に問題が出た段階で起こるので，適当に print 文を入れて，どこまでが出力されて，どこからが出力されないかを確認することでエラーの起こっている場所を特定することができます．

しかし，このような実行時エラーはコンピュータの動作が正常でないという形で表面化するので，まだ良い方です．原因を特定するのに時間がかかるかもしれませんが，修正しなければ動かないのだから実害はありません．実を言うと，一番やっかいなエラーとは，ちゃんと実行してはいるのだけど，計算結果が何となくおかしい，というものです．

筆者は何年も Fortran のプログラムを書いてきましたが，実行時のエラーほど見つけにくいものはありません．それでも見つかれば良い方で，場合によってはバグの存在を知らずに結果を出し，学会直前に気がついた，なんてこともありました．大規模なシミュレーションプログラムを作るとき，長いプログラムを書くのに時間がかかるのはもちろんですが，取りあえず動いたプログラムの計算結果が正しいかどうかを確認する作業にもかなりの時間が必要です．本書で紹介した「エラーの出にくいプログラムの書き方」は，この確認作業にかかる時間を少しでも減らすためのものです．しかし，最後に必要なのは「地道な努力と忍耐」なのです．

[70] 実際にはこの程度の簡単なプログラムでは起こるとは限らないのですが，だからといってこんなプログラムを書いてはいけません．

付録C　自動倍精度化オプション

計算機シミュレーションでは大量の数値を使って複雑な計算をくり返し，必要に応じてそれらを集積する作業を行います．このとき問題となるのが，1.11.2 節で述べた“桁落ち”です．桁落ちを減らすための工夫をいくつか紹介しましたが，究極的には演算精度を上げるしかありません．

Fortran プログラムにおいて標準的に取り扱える実数には，単精度実数型と倍精度実数型がありますが，数値計算をするには倍精度実数型を使うべきです．倍精度実数計算は単精度実数計算と比べてそれほど実行時間はかかりません．これは最近の CPU が倍精度実数計算用のハードウェアを装備しているからです．

Fortran の仕様では，デフォルトの実数型が単精度なので，`real` 宣言をした実数は単精度実数型です．このため，基本仕様のままで倍精度実数計算をするには，変数宣言をする時に `real*8` や `real(8)` などの 8byte 指定が必要です (付録 D 参照)．

例えば，デフォルト仕様のままでも，

```
real*8 x,a1(10)
```

と宣言すれば，変数 `x` や配列 `a1` は倍精度になります．しかし，宣言文を変更するだけでは不完全です．`1.0` や `1.23e5` などの実定数も基本仕様では単精度なので，倍精度実定数に書き換える必要があります．以下に，単精度実定数を倍精度実定数にする変更例を示します[71]．

```
1.23e5      →   1.23d5        ! e を d で置き換える
4.56e-15    →   4.56d-15      ! e を d で置き換える
3.14        →   3.14d0        ! 末尾に d0 をつける
1.0         →   1.0d0         ! 末尾に d0 をつける
```

`1.23e5` のような指数部を付加した単精度実定数は，`e` の代わりに `d` を使って書けば倍精度実定数になるのですから，慣れればそれほど問題ではありません．しかし `1.0` や `3.14` のような指数のない定数は，自然な書き方なので間違いに気付かない可能性があります．しかし，基本仕様ではこれらは単精度実定数なので，倍精度にするには `d0` を付けなければなりません．

そもそも，ハードウェア的にも倍精度計算が基本なのに，常に倍精度を意識した書式を利用しなければならないのは，Fortran が古い計算機用に開発された言語だったときの名残りです．そこで最近の Fortran コンパイラのほとんどが自動倍精度化オプションを装備していることを幸いに，本書では単精度実数型と倍精度実数型を使い分けることはせず，単に“実数型”として説明しました．自動倍精度化オプションを付けてコンパイルすれば，`real` だけの宣言文で倍精度実数変数を宣言できるし，`1.0` や `1.23e5` のような定数はそのままで倍精度実定数を意味します．

表 C.1 に筆者が使ったことのある Fortran コンパイラの自動倍精度化オプションを示します．この他のコンパイラについてはそれぞれのマニュアルをチェックして下さい．ほとんどで用意されていると思います．

表 C.1　コンパイラに応じた自動倍精度化オプション

コンピュータ	コンパイラ	ベンダー	自動倍精度化オプション
パソコン	`gfortran`	GNU	`-fdefault-real-8`
パソコン	`ifort`	Intel	`-r8`
パソコン	`f95`	Absoft	`-N113`
スパコン	`sxf90`	NEC	`-Wf,-A dbl4`

自動倍精度化は，コンパイルする時にオプションを加えれば良いだけなので，プログラムを変更することから考えれば楽なものです．ぜひ利用して下さい．

[71] デフォルト実数が単精度のときは，実数化の組み込み関数 `real` の結果も単精度実数型になります．このため，倍精度実数化が必要なときには関数 `dble` を使います．

付録 D　数値型の精度指定

1.6 節で，基本的な数値型には整数型と実数型があり，実数型には単精度実数型と倍精度実数型があるという話をしました．Fortran のデフォルト実数型は単精度ですが，コンパイラの自動倍精度化機能 (付録 C) を使えばデフォルトを倍精度実数型に変更できるので，本書では 2 種類の実数型を区別せず，単に“実数型”と表現しています．しかし，大量の実数型データをバイナリ形式で保存するときには，保存容量を圧縮するために精度を落とすことが考えられますし，**4 倍精度実数** という倍精度よりも有効桁数の多い実数型を使って，より高精度の計算をすることも可能です[72]．ここでは，定数や変数の精度を変更する書式について説明します．

変数の精度を指定するには，以下の 3 種類の方法があります．ここでは，基本的な宣言文だけを紹介しますが，4.8.1 節で述べた属性や数値代入を含む拡張宣言も可能です．

```
型指定*精度数　変数 1，変数 2，...
型指定 (精度数)　変数 1，変数 2，...
型指定 (kind=精度数)　変数 1，変数 2，...
```

ここで，“精度数”のところには，数値型の byte 数を整定数で指定します．実数型の場合，単精度なら 4，倍精度なら 8，4 倍精度なら 16 です．例えば，単精度実数型変数を宣言するには，

```
real*4 xs1,ys1,as1(100)
real(4) xs2,ys2,as2(100)
real(kind=4) xs3,ys3,ds3(100,100)
```

などと書きます．また，通常の整数型は 4byte ですが，最近は **倍精度整数** という 8byte の整数型 (使用可能な範囲は -2^{63}〜$2^{63}-1$) を使うことができるので，これを利用するときには，

```
integer*8 n81,m81,k81(100)
integer(8) n82,m82,k82(100)
integer(kind=8) n83,m83,ka83(100,100)
```

などと書きます．

3 種類の書式の中でどれが良いかは一概には言えません．一番目の“*”を使った書式は，古い Fortran から使われているので，コンパイラが古い場合や，昔のプログラムを継承する可能性がある場合には覚えておいた方が良いと思います．しかし，これからプログラムを書き始める場合には，二番目か，精度数の意味を明記した三番目の方が良いと思います．これらは，次のように parameter 変数 (4.8.2 節) を使って精度数を指定することができます．

```
integer, parameter :: kp=16
real(kp) xq2,yq2,aq2(100)
real(kind=kp) xq3,yq3,dq3(100,100)
```

これに対し，“*”を使った書式では必ず数字を使って指定しなければなりません．parameter 変数で精度指定をしておけば，計算機環境に応じて精度を変更する必要があるときに便利です．

ただし，高精度の計算をする場合は，定数も高精度にしなければなりません．例えば，“1.23”と書けばデフォルトの実数型になるので，本書の暗黙指定では倍精度実数型です．よって，このまま 4 倍精度の計算に使うと精度が落ちてしまいます．$A\times10^B$ で表される実数を 4 倍精度で表現するときは“AqB”と書きます．このため，“1.23”は“1.23q0”と書かなければなりません．もし 4 倍精度実数型変数に“1.2”のような実数型定数を代入すると，倍精度が保証する有効数字 15 桁までは正しい数字が入りますが，それ以降にどんな数字が入るかは不定です．

[72]ただし，4 倍精度実数型は，必ず実装しなければならない規格ではないようで，コンパイラによっては使えない場合があります．gfortran でも，バージョン 4.8 では使えますが，4.2 では使えませんでした．

しかし，parameter 変数で変数の精度を指定しているときに，e や q のような文字を使って定数の精度を指定していると，精度変更の時に定数の変更が別途必要になります．そこで，数値の後にアンダースコア “_”と精度数を付けて定数の精度を指定することができます．例えば，倍精度の “1.23”は “1.23_8” と書け，4 倍精度の “1.23q0”は，“1.23_16”と書くことができます．この精度指定には parameter 変数を使うことができるので，次のように書くことができます．

```
integer, parameter :: kp=16
real(kp) xx,yy
xx = 1.23_kp
yy = 1.2345e-15_kp*xx**3
```

最後の行のように，1.2345e-15 のような指数指定の数値の後にも精度数を付加することができます．この場合，精度数が 16 なので，4 倍精度定数になります．

付録 E　ASCII コード

文字列の比較などに使われる半角英数字の文字コード (ASCII コード) を表 E.1 に示します．

表 E.1　ASCII コード表

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	␣	!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

半角英文字は 1byte(8bit) 文字であり，左端の数字が上位 4bit，上端の数字 (16 進数) が下位 4bit です．例えば，“T”の文字は 16 進数の 54，“<”は 3C，スペース “␣”は 20 です．1F 以下と 7F は制御コードに割り当てられていて文字としては定義されていません．

なお，5C のバックスラッシュ “\”は，日本語フォントでは “¥”に割り当てられています．

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.

[2] “Fortran90 入門”，新井親夫，森北出版，1998.

[3] “Fortran90 プログラミング”，冨田博之，培風館，1999.

[4] “Numerical Recipes in Fortran 77, Second Edition”, W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, Cambridge University Press, 1996.

[5] “Numerical Recipes in Fortran 90”, W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, Cambridge University Press, 1996.

[6] “Fortran ハンドブック”，田口俊弘，技術評論社，2015.

索 引

な

は

ま

ら