はじめての Fortran90(その四)

浅岡 香枝 *　平野 彰雄 **

1　はじめに

これまで「はじめての Fortran90」と題し、 Fortran90 の新しい機能について解説してきました。

今回は文字型データの扱いと入出力について紹介します。

2　文字型データ

2.1　文字コード

Fortran での文字型データの扱いをお話する前に、計算機内部での文字データはどのように扱われるかについて説明します。

計算機では、プログラムを記述する英数文字どの文字データは、決められた規則に従って符号化され表現されています。これを**文字コード**と呼び、英数字の表現に用いられる代表的なコード系としては **EBCDIC**(Extended Binary Coded Decimal Interchange Code) と **ASCII**(American Standard Code for Information Interchange) があります。

EBCDIC はもともと IBM が作った８ビット単位符号で表す文字コード系です。一方、 **ASCII** は **ISO**(International Organization for Standardization: 国際標準化機構) が制定した７ビット単位符号に基づくアメリカ標準の文字コード系であり、日本の JIS ７コードとは兄弟関係にあるものです。これらの文字コード系の比較を表1に示します。

表 1. ASCII と EBCDIC コードの比較

	ASCII	EBCDIC
文字コード域 (10 進)	0 から 128	0 から 255
英字 A の値 (16 進)	41	C1
処理系	UNIX	MSP

2.2　文字型変数の宣言

文字型変数の宣言は次の形式です。

```
CHARACTER([LEN=]length [,KIND=kind]) &
          [,attribute] :: var_name
```

文字型データには、種別パラメータと長さパラメータの二つがあります。

種別パラメータ (KIND=) は、もともと日本語コードであるとか、他の文字コードを扱うために導入されています。しかし、現状では各処理系が扱う英数字の文字コード系を示す KIND=1 しかありません。

一方、長さパラメータ (LEN=) は、文字型変数の長さを指定します。

```
CHARACTER                :: char    ! (1)
CHARACTER(LEN=5)         :: char5   ! (2)
CHARACTER,DIMENSION(5) :: c_array ! (3)
```

この例で char は 1 文字の文字変数、 char5 は 5 文字の領域を持つ文字変数、 c_array は 1 文字の領域を持つ 5 要素の文字配列の宣言です。

2.3　文字定数の宣言

文字定数は、次のように文字列を'(**シングルクォート**) または"(**ダブルクォート**) で囲って表します。

例えば、次の二つは同じ文字定数を表します。

```
"Kyoto"   !(1)
'Kyoto'   !(2)
```

'(**シングルクォート**) と"(**ダブルクォート**) の使い分けは、それぞれの記号を文字データとして扱いたい場合に使い分けます。

```
"DON'T"    ---> DON'T
'DON"T'    ---> DON"T
```

文字定数で種別パラメータ値を指定する場合は、次のようになります。

```
1_"Kyoto"   !(1)
1_'Kyoto'   !(2)
```

* あさおか かえ ,** ひらの あきお (京都大学大型計算機センター)

他のデータ型では種別パラメータ値を後ろに指定しますが、文字定数では前に指定することになっています。

また、長い文字定数を書きたい場合には、次のように&(**アンパサンド**) を用いて、複数行に書くことができます。

```
"ABCDEFGHIJKLMN&
   &OPQRSTUVWXYZ"
```

これは、次のように書いたのと同じ文字定数として解釈されます。

```
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

ここで、1 行目の& と 2 行目の& を重ね合わせる形で継続される事に注意してください。例えば、次のように空白を置いてしまうと、空白を含む文字定数として解釈されます。

```
"ABCDEFGHIJKLMN    &
   &OPQRSTUVWXYZ"       !(1)
"ABCDEFGHIJKLMN&
  &    OPQRSTUVWXYZ"    !(2)
```

これらは、次のように書いたのと同じ文字定数として解釈されます。

```
"ABCDEFGHIJKLMN    OPQRSTUVWXYZ"
```

2.4 文字型データの演算

2.4.1 文字変数への代入

文字データも数値データと同じく= を使って代入できます。文字データの代入は左辺の文字型変数の長さを基本とした処理となります。

次のプログラムで式 **(2)** の代入は、変数 **char_var** の長さが 5 であるために、6 以降の数字は代入されていません。

```
CHARACTER(LEN=5) :: char_var
char_var = '12'              !(1)
WRITE(*,*) "'",char_var,"'"
char_var = '1234567890'      !(2)
WRITE(*,*) "'",char_var,"'"

[実行結果]
'12   '
'12345'
```

2.4.2 文字型データの連結

文字型データの連結は// で表し、これを連結演算子と呼びます。

連結演算子を用いたプログラムを次に示します。

```
CHARACTER(LEN=5)  :: c1,c2
CHARACTER(LEN=10) :: c3
c1="12345"; c2="67890"
write(*,*) c1," ",c2
c3=c1//c2
write(*,*) c3," ",c2//c1//"A"

[実行結果]
 12345 67890
 1234567890 6789012345A
```

2.4.3 文字部分列

文字型データの一部を部分配列と同じような表記で、部分文字列を指定できます。

```
CHARACTER(LEN=10) :: line  &
 = "1234567890"
WRITE(*,*) line(2:4)
WRITE(*,*) line(2:)
WRITE(*,*) line(:4)
WRITE(*,*) "abcdefg"(2:4)

[実行結果]
 234
 234567890
 1234
 bcd
```

このプログラム例で示すように、文字型データ (変数、定数) の後に括弧で囲んで **I:J** と書いた場合、指定された文字列の **I** 番目から **J** 番目の部分文字列が取り出されます。また、**I** を省略すると 1 と解釈され、**J** を省略すると文字列の長さが指定されたと解釈されます。

また、文字型配列変数に対して部分文字列を指定する場合は、次のように先に配列要素を指定し、後に部分文字列を指定します。

```
CHARACTER(LEN=5), DIMENSION(2) :: lines
lines(1) =  "12345"
lines(2) =  "67890"
WRITE(*,*) lines(2:2)(2:4)

[実行結果]
 789
```

2.4.4 文字の比較演算

文字データも関係演算式 (<,<=,==,>=,>) を用いて比較することができます。文字データの比較は、先頭の一文字同士を比較します。もし等価であれば、その次の文字を比較、それでも等価であればその次の文字というように順々に比較していきます。もし、比較する文字列の長さが異なれば、短い方に空白を補ってから比較します。

図1は、渡された文字配列tableをソートして返すサブルーチンです。

```
SUBROUTINE sort_bubble(table)
CHARACTER(LEN=*), DIMENSION(:) :: table
CHARACTER(LEN=LEN(table))      :: tmp
INTEGER  :: i,j,n
n=SIZE(table)
DO i=1, n-1, 1
 DO j=i+1, n, 1
  IF(table(i) > table(j)) THEN
   tmp = table(j)
   table(j) = table(i)
   table(i) = tmp
  END IF
 END DO
END DO
END SUBROUTINE sort_bubble
```

図 1. ソートプログラム (その一)

このプログラムで **LEN** はtableの長さを返す組込み関数で、 **SIZE** は配列の大きさを返す組込み関数です。

関係演算式を用いた比較は処理系の文字コードで行われます。しかし、代表的なEBCDIC、 ASCIIの文字コードでも英数字の配置が次のように異なるので注意が必要です。

EBCDIC

"空白 "

<"A"<"B"<"C"< ··· <"Z"

<"a"<"b"<"c"< ··· <"z"

<"0"<"1"<"2"< ··· <"9"

ASCII

"空白 "

<"0"<"1"<"2"< ··· <"9"

<"A"<"B"<"C"< ··· <"Z"

<"a"<"b"<"c"< ··· <"z"

Fortranでは関係演算式とは別に、表2に示すような常にASCII順序で比較を行う組込み関数が用意されています。

表 2. 関係演算子と組込み関数の対応

関係演算子	>	>=	<	<=
関数名	LGT	LGE	LLT	LLE

図1のプログラムを図2に示すように関係演算子から組込み関数を用いた比較に直すことで、文字データ処理においても移植性に優れたプログラムになります。

```
SUBROUTINE sort_bubble(table)
CHARACTER(LEN=*), DIMENSION(:) :: table
CHARACTER(LEN=LEN(table))      :: tmp
INTEGER  :: i,j,n
n=SIZE(table)
DO i=1, n-1, 1
 DO j=i+1, n, 1
  IF( LGT(table(i),table(j)) ) THEN
   tmp = table(j)
   table(j) = table(i)
   table(i) = tmp
  END IF
 END DO
END DO
END SUBROUTINE sort_bubble
```

図 2. ソートプログラム (その二)

2.5 文字処理のための組込み関数

2.5.1 文字と整数の変換

組込み関数 **ICHAR** は引数に文字を与えるとそのコードを整数で返します。逆に **CHAR** は引数に与えた整数値に対応する文字を返す組込み関数です。

ICHAR,**CHAR** は処理系のコードに対応する値を返しますが、同じような機能でASCIIコードに対応する組込み関数 **IACHAR**,**ACHAR** が用意されています。

使用例

```
WRITE(*,*) ICHAR('A'),CHAR(ICHAR('A'))
WRITE(*,*) IACHAR('A'),ACHAR(IACHAR('A'))
[MSP(EBCDIC) での実行結果]
 193 A
 65 A
[UNIX(ASCII) での実行結果]
 65 A
 65 A
```

2.5.2 その他の文字処理のための組込み関数

1) 長さを求める関数には **LEN**、 **LEN_TRIM** があります。

LEN は文字データ長を返し、 **LEN_TRIM** は後ろの空白を除いた長さを返します。

使用例

```
CHARACTER(LEN=10) :: c = " 123456   "
WRITE(*,*) LEN(c),LEN_TRIM(c)
[実行結果]
 10 7
```

2) 文字列内に指定した文字を探す関数には、 **INDEX**、 **VERIFY**、 **SCAN** があります。

INDEX は文字列から部分文字列を探し位置を返します。 **SCAN** および **VERIFY** は指定され

た文字列の個々の文字を探します。 **SCAN** は指定された文字が見つかった位置を返し、 **VERIFY** は指定された文字以外を見つけた位置を返します。これらの関数は見つからなかった場合はゼロを返します。また、文字列のサーチは先頭から行いますが、オプション **BACK=.true.** を指定すると後ろから逆にサーチします。

使用例

```
WRITE(*,*) INDEX(" abcdef ","cd")
WRITE(*,*) SCAN(" abcdef "," 1")
WRITE(*,*) VERIFY(" abcdef "," 1")
WRITE(*,*) INDEX(" abcdef ","cd",&
               BACK=.true.)
WRITE(*,*) SCAN(" abcdef "," 1",&
               BACK=.true.)
WRITE(*,*) VERIFY(" abcdef "," 1",&
               BACK=.true.)
```

[実行結果]

```
 4
 1
 2
 4
 8
 7
```

3) 文字列の整形をする関数には、 **TRIM**、**ADJUSTL**、 **ADJUSTR** があります。また、**REPEAT** は文字列と繰返し数を指定して文字列を生成します。

使用例

```
CHARACTER(LEN=5) :: c = " abc "
WRITE(*,*) LEN(TRIM(c)),LEN_TRIM(c)
WRITE(*,*) LEN_TRIM(ADJUSTL(c))
WRITE(*,*) LEN_TRIM(ADJUSTR(c))
WRITE(*,*) REPEAT("+*",5)
```

[実行結果]

```
 4 4
 3
 5
 +*+*+*+*+*
```

3 入出力

入出力操作は処理系のファイル管理機構に依存します。したがって、この解説で処理系に関連する項目は UNIX での標準的な扱いを基本に説明します。MSP とは異なる部分もあるので注意してください。

入出力形式は、次のように分類されます。

1) アクセス方式

- **順次入出力**
- **直接入出力**

2) データ形式

- **テキスト入出力**
  - **変数並び入出力**
  - **書式付き入出力**
  - **ネームリスト入出力**
- **バイナリ入出力**
  - **書式無し入出力**

入出力操作は基本的に **READ 文**,**WRITE 文**、および **OPEN 文**,**CLOSE 文**などの補助入出力文で行います。

3.1 装置番号と入出力

Fortran では入出力文と実際の入出力装置 (ファイルや端末) の対応は**装置番号**で管理します。この装置番号は 0 から 99 までの値です。入出力装置と装置番号の結合と解放は、それぞれ **OPEN 文**、 **CLOSE 文**で行います。また、次の装置番号は **OPEN 文**とは関係なくプログラムの開始時に結合されています。これを**事前結合**と呼びます。

装置番号	入出力装置
5	**標準入力 (STDIN)**
6	**標準出力 (STDOUT)**
0	**標準エラー出力 (STDERR)**

3.2 OPEN 文と CLOSE 文

3.2.1 OPEN 文

OPEN 文は、装置番号とファイルの結合およびファイル属性を与えます。基本形式は次のようになっています。

```
OPEN([UNIT=]unit [,FILE=file] &
    [,STATUS=status] [,ACCESS=access] &
    [,FORM=form] [,ACTION=action] &
    [,RECL=recl] [,IOSTAT=iostat])
```

各オペランドの意味と指定値は、次のようになっています。

- **UNIT=unit**

  装置番号を指定します。 unit には整変数あるいは整定数が指定できます。

- **FILE=file**

  ファイルを指定します。 file にはファイル名を文字型で指定します。

- **STATUS=status**

  ファイルの状態を指定します。 status には次の文字列が指定できます。

  ”old”, ”new”, ”replace”, ”scratch”, ”unknown”

- **ACCESS=access**

  アクセス方式を指定します。 access には次の文字列が指定できます。

  ”sequential”,”direct”

- **FORM=form**

  ファイルのデータ形式を指定します。 form には次の文字列が指定できます。

  "formatted" ,"unformatted"

- **ACTION=action**

  入出力の許可を指定します。 action には次の文字列が指定できます。

  "read", "write", "readwrite"

- **RECL=recl**

  レコードの長さを指定します。 recl には整変数あるいは整定数が指定できます。

- **IOSTAT=iostat**

  iostat にはステータスを返すための整変数を指定します。正常に実行されればゼロ、それ以外は正の値となります。

3.2.2 CLOSE 文

CLOSE 文は装置番号とファイルの結合を解除します。基本形式は次のようになっています。

```
CLOSE([UNIT=]unit [,STATUS=status] &
     [,IOSTAT=iostat])
```

各オペランドの意味と指定値は次のようになっています。

- **UNIT=unit**

  装置番号を指定します。 unit には整変数あるいは整定数が指定できます。

- **STATUS=status**

  CLOSE 文の実行後のファイルの状態を指定します。 status には次の文字列が指定できます。

  "keep","delete"

- **IOSTAT=iostat**

  iostat にはステータスを返すための整変数を指定します。正常に実行されればゼロ、それ以外は正の値となります。

3.3 並び入出力

Fortran で最も簡単な入出力文の形式で、このシリーズの解説記事でも基本的に並び入出力を使って解説してきました。

並び入出力文は、次のように **FMT=*** を指定し、入出力項目 (変数、定数) を,(**カンマ**) で区切って並べると、指定された項目のデータ型に応じた書式で処理します。

基本形式は、次のような文です。

```
READ(UNIT=u,FMT=*)  項目並び
WRITE(UNIT=u,FMT=*) 項目並び
```

標準入出力装置を介した並び入出力のプログラム例を次に示します。

なお、このプログラムの READ,WRITE 文で最初の ***(アスタリスク)** は UNIT= を省略した表記で、後の * は FMT= を省略した表記です。

```
INTEGER   :: int
REAL      :: real
CHARACTER :: char
LOGICAL   :: logical
READ(*,*)  int,real,char,logical
WRITE(*,*) int,real,char,logical
```

このプログラムを実行させ、入力データに

```
1 1.0 b .true.
```

を与えると

```
1 1.00000000 b T
```

と出力されるはずです。

なお、出力の書式は処理系依存とされているので、実行結果は異なることもあります。

3.4 書式付き入出力

並び入出力文は手軽に使うことができますが、Fortran の入出力は文単位に 1 レコードを構成するので、配列を並びに書いてしまうと非常に長いレコードになり、プリンタ出力では途中で切れてしまうといったことが起ります。このようなことを避けるためには書式付き入出力文で項目の書式やレコードの長さを指定します。

基本形式は、次のように **FMT=fmt** を指定し、fmt には**編集記述子**を用いた入出力の書式の指定を文字型で記述します。

```
READ(UNIT=u,FMT=fmt)  項目並び
WRITE(UNIT=u,FMT=fmt) 項目並び
```

簡単な書式付き WRITE 文のプログラム例を図 3に示します。

また、書式付き入出力文で使える主な編集記述子を表 3に示します。

```
CHARACTER(10)  :: a="string"
REAL(8)        :: d=1.23d-8
REAL           :: e=3.4E+34
REAL           :: f=5.6
INTEGER        :: i=789
LOGICAL        :: l=.true.
CHARACTER(200) :: fmt = &
   "('a:  ',a12,   1x,g12.5,/,&
   &'d:  ',d12.5,  1x,g12.5,/,&
   &'e:  ',e12.5, 1x,g12.5,/,&
   &'f:  ',f12.5,  1x, g12.5,/,&
   &'i:  ',i12,    1x, g12.5,/,&
   &'l:  ',l12,    1x,g12.5)"
WRITE(*,FMT=fmt) a,a,d,d,e,e,f,f,i,i,l,l
```

[実行結果]

```
a:     string       string
d:   0.12300D-07  0.12300E-07
e:   0.34000E+35  0.34000E+35
f:        5.60000   5.6000
i:            789          789
l:              T            T
```

図 3. 書式付き WRITE 文を用いたプログラム

表 3. 編集記述子

データ型	記述子名	形式
整数型	I 型編集記述子	rIw
	B 型編集記述子	rBw
	O 型編集記述子	rOw
	Z 型編集記述子	rZw
実数型	F 型編集記述子	rF$w.d$
	E 型編集記述子	rE$w.d$Ee
	EN 型編集記述子	rEN$w.d$Ee
	ES 型編集記述子	rES$w.d$Ee
文字型	A 型編集記述子	rAw
論値型	L 型編集記述子	rLw
すべての型	G 型編集記述子	rG$w.d$
空白	X 型編集記述子	rX
改行		/

この表の英小文字の意味は、次のようなものです。

- r: 繰り返し数
- w: 入出力文字の桁数
- d: 小数点以下の桁数
- e: 指数部の桁数

3.5 停留入出力

書式付き入出力文に **ADVANCE=”no”** を指定するとレコードの構成を制御することができます。これを停留入出力といい、この指定があると次のレコードでは無く、次のバイトに位置づけられます。

停留入出力の READ,WRITE 文は、次のように指定します。

```
READ(UNIT=u,FMT=fmt,&
            ADVANCE="no")  項目並び
WRITE(UNIT=u,FMT=fmt,&
            ADVANCE="no")  項目並び
```

図 4は、停留入出力を用いた動作を示すプログラム例です。

```
INTEGER :: no
WRITE(*,FMT="(a)") " Enter number := "
READ(*,*) no
WRITE(*,*) " Input number := ",no
WRITE(*,FMT="(a)",ADVANCE="no") &
                    " Enter number := "
READ(*,*) no
WRITE(*,*) " Input number := ",no
```

[実行結果]

```
 Enter number :=
 10
 Input number := 10
 Enter number := 100
 Input number := 100
```

図 4. 停留入出力を用いたプログラム

ADVANCE=”no” を指定すると改行されていないことがわかると思います。

```
PROGRAM lower_to_upper
  IMPLICIT NONE
  INTEGER :: i,iostat
  CHARACTER :: char
  CHARACTER(26) ::  lower = &
   "abcdefghijklmnopqrstuvwxyz"
  CHARACTER(26) ::  upper = &
   "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
  loop: DO
   READ(*,FMT="(a1)",&
     ADVANCE="no",IOSTAT=iostat) char
   IF( iostat /= 0) EXIT
   i=INDEX(lower, char)
   IF( i > 0 ) char=upper(i:i)
   WRITE(*,FMT="(a1)",&
     ADVANCE="no") char
  END DO loop
END PROGRAM lower_to_upper
```

図 5. 英大文字に変換するプログラム

また、図 5は標準入力に与えられたファイルを英大文字に変換し標準出力に書き出すプログラム例です。

このプログラムで **IOSTAT=iostat** の指定は READ 文の実行結果の状態が変数 **iostat** に設定されます。正常にはゼロ、 EOF は負の値、エラーは正の値が設定されます。

3.6 ネームリスト (変数群) 入出力

ネームリスト入出力とは、変数名付きで入出力を行うものです。ネームリスト入出力を使うには、まず、**NAMELIST** 文で入出力の対象となる変数の並びを指定します。

NAMELIST 文は、次のような形式です。

```
NAMELIST / group_name / var1, var2, ...
```

そして、定義した NAMELIST 文の **group_name** を READ 文、 WRITE 文の NML 指定子で次のように指定します。

```
READ(UNIT=u,NML=group_name)
WRITE(UNIT=u,NML=group_name)
```

一方、ネームリスト入力に対応するデータは、次のような形式で指定します。

```
&group_name var1=value1,var2=value2,... /
```

すなわち、先頭の& に続いて group_name を書き、次に値を設定する変数に対して**変数名= 値**という形で記述します。区切りは空白あるいは,(**カンマ**) です。最後は /(**スラッシュ**) を書きます。指定する変数と値のペアは **NAMELIST** 文での変数の宣言の順序とは関係なく、また、複数行に書くことも許されます。

図 6にネームリスト入出力を使った簡単なプログラム例を示します。

```
PROGRAM ex_namelist
  IMPLICIT NONE
  REAL    :: alpha=1.0
  INTEGER :: l=1,m=10,n=100
  NAMELIST / group / l,m,n,alpha
  OPEN(UNIT=10,FILE="name.dat")
  WRITE(*,NML=group)
  READ(10,NML=group)
  WRITE(*,NML=group)
  CLOSE(UNIT=10)
END PROGRAM ex_namelist
```

図 6. ネームリストを用いたプログラム

また、ファイル **name.dat** のデータを次に示します。

```
&group  n=2,
        l=20,
        m=200,/
```

実行結果は、次のようになります。

```
&GROUP L=1,M=10,N=100,ALPHA=1.00000000/
&GROUP L=20,M=200,N=2,ALPHA=1.00000000/
```

3.7 書式無し入出力

書式無し入出力とは、指定された項目並びをバイナリの形でファイルに対して入出力を行うものです。他のテキスト形式での入出力文と異なり、指定された入出力項目のメモリの内容がそのままファイルに書かれるのでファイルの容量も小さくなり、また、テキストに変換する際の 10 進変換誤差も生じないので、プログラム間でのデータの受け渡しには大変有効です。

書式無し READ,WRITE 文には、装置番号と入出力する項目並びだけを指定します。

一つの入出力文で 1 レコードを構成するので、同じファイルに対する READ 文と WRITE 文の順序、各入出力文の項目並びの順序、データ型、配列の大きさなどは、一致する必要があります。

書式無し入出力を行うファイルに対しては、次のような OPEN 文を指定します。

```
OPEN(UNIT=unit,FILE=file,&
                FORM="unformatted")
```

READ、 WRITE 文は、次のようになります。

```
READ(UNIT=unit) 項目並び
WRITE(UNIT=unit) 項目並び
```

図 7と図 8に書式無し入出力を用いたプログラム例を示します。

図 7のプログラムでは **unformat.dat** というファイルにデータを書出し、図 8のプログラムでデータを読込んで処理しています。図 7と図 8の書式なし READ,WRITE 文の順序、入出力並びが同じになっていることに注意してください。

```
PROGRAM output
 IMPLICIT NONE
 INTEGER,PARAMETER :: na = 100
 INTEGER :: n,i,j
 REAL(8),DIMENSION(na,na) :: array
 n=na
 OPEN(UNIT=10,FILE="unformat.dat", &
   FORM="unformatted")
 DO i=1,n
  DO j=1,n
   array(j,i) = sqrt(real(j+i))
  END DO
 END DO
 WRITE(10) n
 DO i=1,n
  WRITE(10) array(:,i)
 END DO
 CLOSE(UNIT=10)
END PROGRAM output
```

図 7. 書式無し出力プログラム

```
PROGRAM input
 IMPLICIT NONE
 INTEGER,PARAMETER :: na = 100
 INTEGER :: n,i,j
 REAL(8),DIMENSION(na,na) :: array
 REAL(8) :: sum
 OPEN(UNIT=10,FILE="unformat.dat", &
   STATUS="old",FORM="unformatted")
 READ(10) n
 DO i=1,n
  READ(10) array(:,i)
 END DO
 CLOSE(UNIT=10)
 sum=0.0
 DO i=1,n
  DO j=1,n
   sum = sum + array(j,i)
  END DO
 END DO
 WRITE(*,*) n,sum
END PROGRAM input

[実行結果]
100 98066.13009440899
```

図 8. 書式無し入力プログラム

3.8 直接入出力

直接入出力とはレコード番号を指定して書出し、読込み、更新を行うものです。

直接入出力ファイルを使用するには、 OPEN 文で次のように指定します。

```
OPEN(UNIT=unit,FILE=file, &
  ACCESS="direct",RECL=recl,FORM=form)
```

ここで ACCESS=”direct”は直接入出力指定、また、 recl には 1 レコードの長さの最大値を整数型で指定です。

form には ”formatted”,”unformatted”が指定できます。省略時は ”unformatted”です。

一方、 READ、 WRITE 文には、次のように処理するレコード番号を REC= で指定します。先頭のレコード番号は 1 です。

```
READ(UNIT=unit,REC=rec)
WRITE(UNIT=unit,REC=rec)
READ(UNIT=unit,FMT=fmt,REC=rec)
WRITE(UNIT=unit,FMT=fmt,REC=rec)
```

図 9に直接入出力を用いたプログラムと実行結果を示します。

このプログラムでは、まず、最初 10 レコードを初期化し、次に、レコード番号と対応データの入力を待ちます。データ入力の最後はレコード番号 0 でデータは end です。入力が終わるとデータを印刷します。なお、同じレコード番号が指定されると後のデータで更新されます。

```
PROGRAM direct_access
  IMPLICIT NONE
  INTEGER :: i,rec,max_rec=10
  CHARACTER(10) :: data
  OPEN(UNIT=10,FILE="direct.dat"&
  ,ACCESS="direct",RECL=14)
  init: DO i=1,max_rec
   WRITE (UNIT=10,REC=i) 0,'0'
  END DO init
  WRITE(*,*) "Enter rec_no,data_string"
  WRITE(*,*) "For end ('0 end')"
  input: DO
   READ(*,*) rec,data
   IF (rec <= 0 .OR. max_rec < rec) EXIT
   WRITE(UNIT=10,REC=rec) rec,data
  END DO input
  WRITE(*,*) "Print all input data"
  print: DO i=1,max_rec
     READ(UNIT=10,REC=i) rec,data
     IF ( rec /= 0 ) &
        WRITE(*,*) rec,data
  END DO print
  CLOSE(UNIT=10)
END PROGRAM direct_access

   [実行結果]
  Enter rec_no,data_string
  For end ('0 end')
 3 apple
 8 orange
 5 lemon
 3 melon
 0 end
  Print all input data
  3 melon
  5 lemon
  8 orange
```

図 9. 直接入出力を用いたプログラム

4 おわりに

今回は文字型データの扱いと入出力について解説しました。昨年 8 月から連載してきた「はじめてのFortran90」シリーズは、これで終わります。このシリーズの広報 Vol.,No. と解説した内容は、次のようになっています。

Vol.,No.	解 説 記 事
Vol.30,No.4	基礎と配列演算
Vol.30,No.5	再帰関数, 構造型とポインタ処理
Vol.31,No.1	モジュールおよびプロシジャ
Vol.31,No.2	文字データ型および入出力

この連載では、 Fortran90 の規格で取入れられた新しい機能を中心に解説しました。このシリーズを読んで頂いて、少しでも Fortran90 に対しての理解を深めていただければ幸いです。

これからも皆様の役に立つ解説を書いていきたいと思います。よろしくお願いします。