はじめての Fortran90(その二)

浅岡 香枝 *　平野 彰雄 **

1　はじめに

前回から連載の「はじめての Fortran90」シリーズの二回目です。

前回は、 Fortran90 の新しい機能について述べ、プログラムの形式として自由形式、データ型の宣言、手続き、配列演算機能について解説しました。

今回は、

- 再帰手続き
- 構造体
- ポインタ
- 構造型とポインタを用いた動的なデータ構造

について解説します。

2　再帰手続き

再帰手続きとは、ある手続きが直接あるいは間接的に自分自身を呼び出すような構造になっている手続きをいいます。

2.1　再帰手続きの宣言

FORTRAN77 では再帰手続きを書くことは許されていませんでしたが、 Fortran90 から次のように FUNCTION 文、 SUBROUTINE 文の前にキーワード **RECURSIVE** をつけることで再帰手続きが宣言できます。

```
RECURSIVE SUBROUTINE name(arg,..)
          :
END SUBROUTINE name
RECURSIVE FUNCTION name(arg,..) &
                      [RESULT(name)]
          :
END FUNCTION name
```

2.2　再帰関数と RESULT 句

FORTRAN77 の関数宣言では、結果を返す式は関数名への代入式として表現していました (図 1)。

```
REAL FUNCTION add(arg1,arg2)
  IMPLICIT NONE
  REAL, INTENT(IN) :: arg1,arg2
  add = arg1+arg2
END FUNCTION add
```

図 1. RESULT 句の無い関数

しかし、これは手続き名への代入というプログラムは読みにくものです。 Fortran90 では **RESULT 句**で関数の結果を返す変数名を指定できます (図 2)。

```
FUNCTION add(arg1,arg2) RESULT(result)
  IMPLICIT NONE
  REAL,INTENT(IN) :: arg1,arg2
  REAL :: result
  result = arg1+arg2
END FUNCTION add
```

図 2. RESULT 句を指定した関数

ただし、 **RESULT 句**を指定した関数は、 FUNCTION 文にデータ型は指定できません。また、直接自分自身を呼び出す再帰関数の宣言では **RESULT 句**の指定が必須になります。

入力された数の階乗を求めるプログラム factrial を再帰関数として宣言すると図 3のようになります。

```
PROGRAM fact
  IMPLICIT NONE
  INTEGER :: n
  READ(*,*) n
  WRITE(*,*) n,factrial(n)
  CONTAINS
  RECURSIVE FUNCTION factrial(n)  &
                RESULT(result)
   INTEGER,INTENT(IN) :: n
   INTEGER :: result
   IF ( n > 0 ) THEN
    result = n*factrial(n-1)
   ELSE
    result = 1
   END IF
  END FUNCTION factrial
END PROGRAM fact
```

図 3. 再帰関数の例

* 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)

2.3 再帰サブルーチン

階乗プログラムのサブルーチン版を図 4に示します。

```
PROGRAM fact
  IMPLICIT NONE
  INTEGER :: n,result
  READ(*,*) n
  CALL fact(n,result)
  WRITE(*,*) n,result
  CONTAINS
  RECURSIVE SUBROUTINE factrial(n,result)
    INTEGER,INTENT(IN) :: n
    INTEGER,INTENT(INOUT) :: result
    IF ( n > 0 ) THEN
     CALL factrial(n-1,result)
     result = n*result
    ELSE
     result = 1
    END IF
  END SUBROUTINE factrial
END PROGRAM fact
```

図 4. 再帰サブルーチンの例

2.4 再帰手続き VS 繰返し

アルゴリズムには、再帰的なプログラム表現になるものが多くあり、簡潔に表現できるので再帰手続きが宣言できることは大変便利です。しかし、計算機にとって手続きの呼出し処理は多くのコストを要するものです。したがって、高速な演算のためには、再帰的な処理を繰返しに置き換えることも必要です。図 5に階乗プログラムの DO ループ版を示します。

```
PROGRAM fact
  IMPLICIT NONE
  INTEGER :: n,result
  READ(*,*) n
  CALL factrial(n,result)
  WRITE(*,*) n,result
  CONTAINS
  SUBROUTINE factrial(n,result)
    INTEGER,INTENT(IN) :: n
    INTEGER,INTENT(INOUT) :: result
    INTEGER :: i
    result = 1
    DO i = 1, n
    result = result*i
    END DO
  END SUBROUTINE factrial
END PROGRAM fact
```

図 5. 階乗の繰返しプログラム例

3 構造体

Fortran には、組込みデータ型として整数型、実数型、複素数型、論理型、文字型の五つがあります。 Fortrn90 では、この五つの組込みデータ型に加えて、利用者が新たにデータ型を定義し、拡張する枠組みを備えています。この利用者定義のデータ型を構造型といい、構造型で定義された変数を構造体といいます。

3.1 構造型の定義と宣言

構造体を宣言するには、まず、構造型の定義が必要です。構造型定義の形式は、次のようになっています。

```
TYPE derived_type_name
    component_define
END TYPE derived_type_name
```

ここで、 **derived_type_name** は、利用者定義の構造型の名前です。また、 **component_define** には、定義する構造型の成分をデータ型の宣言文と同じ形式で並べたものを指定します。構造型の成分には、五つの組込みデータ型とそれ自身を含む構造型を宣言することができます。

例えば、 X-Y 座標系での一点を表す構造型 point の定義は、次のようになります。

```
TYPE point
   REAL :: x,y
END TYPE point
```

ここで、構造型 point は、 x,y という実数型の二つの成分を持つデータ型として定義されます。一方、構造型を用いた構造体宣言の形式は、次のように指定します。

```
TYPE(derived_type_name)[,attribute] &
                  :: name,...
```

したがって、構造型 point の構造体 p1,p2 の宣言は、次のようになります。

```
TYPE point
   REAL :: x,y
END TYPE point
    :
TYPE(point) :: p1,p2
```

3.2 構造体の演算と成分構成子 (%)

利用者が宣言した構造体を直接演算式に用いることはできません。構造体の演算式は、構造型の個々の成分を指定した演算として記述する必要があり

ます。構造体の各成分を指定するには、成分構成子 %(パーセント) を用います。前に宣言した構造体変数 p1,p2 の各成分は、次のように表します。

点 p1 の x 座標	**p1%x**
点 p1 の y 座標	**p1%y**
点 p2 の x 座標	**p2%x**
点 p2 の y 座標	**p2%y**

また、演算式は次のように表します。

```
p1%y=1.0
p1%x=p1%x+p2%x
x=p1%x*p2%x
```

ここで、 x は実数型の変数です。

3.3 構造体の代入と構造型の定数

構造体の演算式は、個々の成分を指定しないと表せませんが、同じ構造型を持つ構造体変数同士であれば代入できます。

```
p1=p2                    !(1)
p1%x=p2%x; p1%y=p2%y     !(2)
```

この式 (1) と式 (2) は同じ処理を表し、構造型の対応する成分がそれぞれ代入されます。また、構造型の定数は、次のような形式で表します。

```
derived_type_name(value,...)
```

すなわち、先頭に構造型名を指定し、各成分の与える定数を順に,(カンマ) で区切り、全体を括弧で囲んで表します。これを構造体構成子といいます。

```
p1%x=3.0; p1%y=7.0     !(1)
p1=point(3.0,7.0)      !(2)
```

式 (1) の成分構成子を使った代入式は、式 (2) のように構造体構成子を用いて表します。また、構造体構成子は構造体の宣言で初期値として指定できます。

3.4 構造体の入出力

入出力文に構造体変数名を直接書くことができます。この場合構造型の定義で書いた順に各成分が処理されます。すなわち、 次の式 (1) と式 (2) の結果は同じになります。

```
WRITE(*,*) p1,p2                  !(1)
WRITE(*,*) p1%x,p1%y,p2%x,p2%y    !(2)
```

3.5 構造体を用いたプログラム例

構造体を用いたプログラム例として X-Y 座標上の二点 p1 と p2 を与え、その二点と原点 (0,0) とで囲まれる三角形の面積を求めるプログラムを図6に示します。

```
PROGRAM s_triangle
  IMPLICIT NONE
  TYPE point
   REAL :: x,y
  END TYPE point
  TYPE(point) :: p1,p2
  REAL :: s
  READ(*,*) p1,p2
  s=ABS(p1%x*p2%y-p1%y*p2%x)/2.0
  WRITE(*,*) s
END PROGRAM s_triangle
```

図 6. 構造体を用いたプログラム例

3.6 色々な構造体の例

1) 構造体を成分に含む構造体

図 7は、 X-Y 座標の一点を表す point という構造型を定義し、三角形の各頂点 a,b,c を成分とする構造型 triangle の定義と構造体変数 t の宣言です。

```
TYPE point
   real :: x,y
END TYPE point
TYPE triangle
   TYPE(point) :: a,b,c
END TYPE triangle
TYPE(triangle) :: t
```

図 7. 構造体を成分に含む構造体の例

この場合、 triangle 型の変数 t の成分 a,b,c は、それぞれ構造型 point であるために演算式では、t%a%x,t%a%y ように表します。

t%a%x のように示すものを構造体の末端成分といい、演算式は末端成分で表します。また、この構造型 triangle の構造体構成子は、次のように表します。

```
t=triangle(point(0.0,0.0), &
  point(1.0,4.0),point(4.0,1.0))
```

2) 構造型の配列

図 8のプログラムの構造体 p1,p2 は、 100 個の構造型 point を要素とする配列の宣言です。このプログラムは、 p1,p2 の対になったデータを読み、それぞれの面積を求めています。

```
PROGRAM s_triangle
  IMPLICIT NONE
  INTEGER, PARAMETER :: n=100
  TYPE point
   REAL :: x,y
  END TYPE point
  TYPE(point),DIMENSION(n) :: p1,p2
  REAL,DIMENSION(n) :: s
  INTEGER :: i
  DO i=1,n
   READ(*,*)  &
    p1(i)%x,p1(i)%y,p2(i)%x,p2(i)%y
  END DO
  s=ABS(p1%x*p2%y-p1%y*p2%x)/2.0
  WRITE(*,*) s
END PROGRAM s_triangle
```

図 8. 構造型の配列

3) 配列を成分にもつ構造体

図 9プログラムの構造型 group は、成分として整数型の識別子 id とその値の持つ 100 個の実数型配列 value から構成されています。このプログラムは、 id がゼロになるまでデータを読み、 id とその総和を出力します。

```
PROGRAM group_sum
 IMPLICIT NONE
 TYPE group
  INTEGER :: id
  REAL,DIMENSION(100) :: value
 END TYPE group
 TYPE(group) :: work
 DO
  READ(*,*) work
  IF (work%id == 0) EXIT
  WRITE(*,*) work%id,sum(work%value(:))
 END DO
END PROGRAM group_sum
```

図 9. 配列を成分に含む構造体の例

4 ポインタ

ポインタとは、 Fortran90 で新たに導入されたもので、図 10に示すように他の変数領域を指す変数です。ポインタは型宣言文で POINTER 属性を指定して宣言します。

ポインタが指すことができる領域には、次の二種類があります。

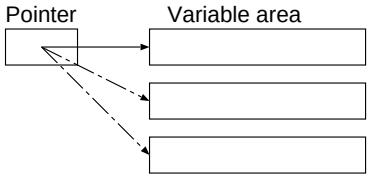

図 10. ポインタの概念

1) 型宣言文で TAGET 属性が指定された変数。

2) ポインタ変数を指定した ALLOCATE 文で動的に割付けられた領域。

4.1 ポインタの宣言

ポインタ変数とその指示先であるターゲット変数は、それぞれ変数の型宣言で **POINTER 属性**、**TARGET 属性**を指定します。

```
REAL,POINTER :: p
REAL,TARGET  :: t
```

ここで、

- ポインタ変数は、 TERGET 属性を持たない変数は指せません。
- ポインタ変数は、同じデータ型のターゲット変数でないと指せません。

4.2 ポインタとターゲットの結合

宣言されたポインタ変数は、何も指示していない状態であり、

```
p => t
```

のような式でターゲット変数と結合し、指示しない限り演算式に指定できません。この式をポインタ代入といい=> で表します。

4.3 ポインタの操作

4.3.1 ポインタ代入

図 11のプログラムは、式 (1) の操作でポインタ変数 p1 はターゲット変数 t1 と結合し、式 (2) の操作でポインタ変数 p2 はターゲット変数 t2 と結合します。

式 (1),(2) の結果ポインタ変数 p1,p2 は、図 12のようにそれぞれターゲット変数 t1,t2 の領域を指します。

```
PROGRAM pointer
  IMPLICIT NONE
  REAL,POINTER :: p1,p2
  REAL,TARGET  :: t1=1.0,t2=2.0
  p1=>t1                    !(1)
  p2=>t2                    !(2)
  WRITE(*,*) p1,t1,p2,t2    !(3)
END PROGRAM pointer
```

図 11. プログラム例 1

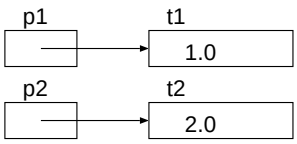

図 12. ポインタ代入の結果

式 (3) の WRITE 文の結果は、

```
  1.0   1.0    2.0   2.0
```

となります。すなわち、ポインタ変数 p1,p2 は、ターゲット変数 t1,t2 の別名として機能しています。

ここで、図 11のプログラムの後に、次のようなポインタ変数に対する演算を加えると

```
  p2=>t1                    !(4)
  WRITE(*,*) p1,t1,p2,t2    !(5)
```

式 (4) のポインタ代入の結果、ポインタ変数 p2 の指示先はターゲット変数 t1 に変更され、 (5) の WRITE 文の結果は、

```
  1.0   1.0    1.0   2.0
```

となります。

すなわち、式 (4) の実行後のポインタ変数 p2 も図 13のようにターゲット変数 t1 の領域を指します。

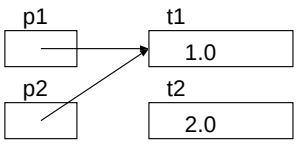

図 13. 指示先の変更

ここで、式 (4) のポインタ代入は

```
  p2=>p1                    !(4)
  WRITE(*,*) p1,t1,p2,t2    !(5)
```

のように、ポインタ変数同士をポインタ代入しても同じ結果になります。

4.3.2 ポインタ変数へ代入

これまでポインタ代入について説明しましたが、次はポインタ変数への代入操作について説明します。

図 14のプログラムは、図 11のプログラムとは異なり、ターゲット変数は初期値を持っていません。式 (1) は、ポインタ変数 p1,p2 とターゲット変数 t1,t2 をそれぞれ結合し、指示する操作です。ここでターゲット変数 t1,t2 の値は未定義であり、t1,t2,p1,p2 を WRITE しても結果は不定です。式 (2) で p1,p2 に対する代入で指示先、 t1、 t2 にそれぞれ 1.0,2.0 が代入されます。

```
PROGRAM pointer
  IMPLICIT NONE
  REAL,POINTER :: p1,p2
  REAL,TARGET  :: t1,t2
  p1=>t1 ; p2=>t1           !(1)
  p1=1.0 ; p2=2.0           !(2)
  WRITE(*,*) p1,t1,p2,t2    !(3)
END PROGRAM pointer
```

図 14. プログラム例 2

式 (3) の WRITE 文の結果は、図 11のプログラムと同じで、次のようになります。

```
  1.0   1.0    2.0   2.0
```

ここで、図 14のプログラムの後に、次のようなポインタ変数同士の代入加えると、

```
  p2=p1                     !(4)
  WRITE(*,*) p1,t1,p2,t2    !(5)
```

式 (5) の WRITE 文の結果は、次のようになります。

```
  1.0   1.0    1.0   1.0
```

すなわち、 p1 の指示先の t1 の値 1.0 が p2 の指示先の t2 に代入されます。 式 (4) の実行後の各変数は、図 15のようになっています。

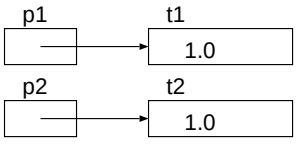

図 15. ポインタ変数同士の代入の結果

4.4 ポインタ配列

配列宣言で **POINTER 属性**を指定するとポインタ配列が宣言できます。多次元のポインタ配列も宣言できますが、個々の次元の寸法を指定することはできません。多次元のポインタ配列は、配列の次元寸法の代わりに:(コロン) を,(コンマ) で区切って、次元数だけを指定します。例えば、一次元、二次元のポインタ配列は、次のように宣言します。

```
REAL,POINTER,DIMENSION(:)   :: ptr_1d
REAL,POINTER,DIMENSION(:,:) :: ptr_2d
```

ポインタ配列と結合する配列は、 **TERGRT 属性**を持つ必要があります。

```
REAL,TARGET,DIMENSION(5)    :: t_1d
REAL,TARGET,DIMENSION(5,5)  :: t_2d
```

```
PROGRAM pointer_array
 IMPLICIT NONE
 INTEGER  :: i
 REAL,POINTER,DIMENSION(:)   :: ptr_1d
 REAL,POINTER,DIMENSION(:,:) :: ptr_2d
 REAL,TARGET,DIMENSION(5,5)  :: t_2d &
  = RESHAPE((/ (i,i=1,25) /), (/5, 5/))
 ptr_2d => t_2d                    !(1)
 DO i=1,5
    WRITE(*,*) (ptr_2d(i,:))       !(2)
 END DO
 ptr_1d => t_2d(:,1)               !(3)
 WRITE(*,*) ptr_1d                 !(4)
 ptr_1d => t_2d(1,:)               !(5)
 WRITE(*,*) ptr_1d                 !(6)
 ptr_2d => t_2d(1:2,1:2)           !(7)
 WRITE(*,*) ptr_2d                 !(8)
END PROGRAM pointer_array
```

図 16. ポインタ配列の結合

4.4.1 ポインタ配列の結合

ポインタ配列は、同じデータ型、同じ次元数を持つターゲット配列とだけ結合し、ポインタ配列の各要素がターゲット配列の各要素を指示します。ポインタ配列の各次元寸法は、結合したターゲット配列の各次元寸法になります。

図 16のプログラムは、ポインタ配列が部分配列式で指定されたターゲット配列と結合し、指示する例です。式 (1) のポインタ代入は、ポインタ、ターゲットともに二次元の配列であるので、ptr_2d の各要素がt_2d の各要素を指すように結合されます。式 (2) の WRITE 文の結果は、次のようになります。

```
 1.0   6.0  11.0  16.0  21.0
 2.0   7.0  12.0  17.0  22.0
 3.0   8.0  13.0  18.0  23.0
 4.0   9.0  14.0  19.0  24.0
 5.0  10.0  15.0  20.0  25.0
```

式 (3) は、t_2d の一列目の全要素を一次元のポインタ配列ptr_1d で指しています。このようにポインタ代入文と部分配列式を用いると配列の一部を指すことができます。式 (4) の WRITE 文の結果は、次のようになります。

```
 1.0   2.0   3.0  4.0  5.0
```

式 (5) は、t_2d の一行目の全要素を一次元のポインタ配列ptr_1d で指しています。式 (6) の WRITE 文の結果は、次のようになります。

```
 1.0   6.0   11.0  16.0  21.0
```

式 (7) は、配列t_2d のを特定の部分を二次元のポインタ配列ptr_2d で指しています。式 (8) の WRITE 文の結果は、次のようになります。

```
 1.0   2.0   6.0  7.0
```

式 (7) の部分配列式の結果は、二次元の配列となるのでポインタ配列も二次元ptr_2d になります。

4.5 ポインタ引数

手続きの引数にポインタを指定することができます。呼出し元の実引数がポインタの場合、手続き側の仮引数には、次の二種類があります。

- 仮引数もポインタ
- 仮引数はポインタ以外

4.5.1 仮引数はポインタ

仮引数をポインタとする場合、実引数もポインタであり、ポインタ配列の場合次元も一致しなければなりません。この場合手続きの呼出し時の結合状態が手続きに渡されます。ただし、ポインタ仮引数には **INTENT 属性**は指定できません。また、ポインタ仮引数を外部手続きで宣言する場合、呼出し元に引用仕様の宣言が必須です。図 17にポインタを仮引数とするプログラム例を示します。

```
PROGRAM pointer_arg1
 IMPLICIT NONE
 INTEGER  :: i
 REAL,POINTER,DIMENSION(:) :: p
 REAL,TARGET,DIMENSION(5)  :: t &
  = (/ (i,i=1,5) /)
 p => t
 call sub(p)
 WRITE(*,*) t
 CONTAINS
  SUBROUTINE sub(v)
  REAL,POINTER,DIMENSION(:) :: v
    v = v + 100.0
  END SUBROUTINE sub
END PROGRAM pointer_arg1
```

図 17. ポインタ仮引数の例

4.5.2 仮引数はポインタ以外

ポインタ実引数は、ポインタ以外の仮引数と対応することも可能です。この場合ポインタは結合していなければならず、指示先が仮引数に渡ります。図18にポインタ以外を仮引数とするプログラム例を示します。

```
PROGRAM pointer_arg2
 IMPLICIT NONE
 INTEGER  :: i
 REAL,POINTER,DIMENSION(:) :: p
 REAL,TARGET,DIMENSION(5)  :: t &
  = (/ (i,i=1,5) /)
 p => t
 call sub(p)
 WRITE(*,*) t
 CONTAINS
  SUBROUTINE sub(v)
  REAL,DIMENSION(:),INTENT(INOUT) :: v
    v = v + 100.0
  END SUBROUTINE sub
END PROGRAM pointer_arg2
```

図 18. ポインタ以外の仮引数の例

4.6 ポインタの動的割付け

ポインタは、**TARGET 属性**を持つ変数を指示するだけでなく、**ALLOCATE 文**にポインタを指定すると動的に割付けた領域と結合できます。図19のプログラムは、ポインタの動的割付けの例です。

図18のプログラムは、式 (1) で入力するデータ個数を読込み、式 (2) の **ALLOCATE 文**で必要な大きさの領域を割付け、式 (3) でデータを読込み、式 (4) で総和を取り、出力しています。式 (5) の **DEALLOCATE 文**は、**ALLOCATE 文**で確保した領域を解放します。

ここで、**TERGET 属性**を持つ変数とポインタ変数との結合は、ポインタ代入を用いて結合する必

```
PROGRAM alloc_pointer
 IMPLICIT NONE
 INTEGER  :: n
 REAL,POINTER,DIMENSION(:) :: p
 READ(*,*)  n               !(1)
 ALLOCATE(p(n))             !(2)
 READ(*,*)  p               !(3)
 WRITE(*,*) SUM(p)          !(4)
 DEALLOCATE(p)              !(5)
END PROGRAM alloc_pointer
```

図 19. ポインタの動的割付け例

要がありましたが、式 (2) のポインタ配列を指定した **ALLOCATE 文**は、n 個の一次元の実数配列領域を確保し個々の要素がポインタ配列 p と結合していることに注意してください。また、式 (5) の **DEALLOCATE 文**で領域が解放され、ポインタ配列 p の結合も解かれます。

4.7 動的割付け配列 VS ポインタ配列

図19のプログラムは、**ALLOCATABLE** 属性を指定した動的割付け配列を用いて表せます (図20)。

```
PROGRAM alloc_array
 IMPLICIT NONE
 INTEGER  :: n
 REAL,ALLOCATABLE,DIMENSION(:) :: a
 READ(*,*)  n               !(1)
 ALLOCATE(a(n))             !(2)
 READ(*,*)  a               !(3)
 WRITE(*,*) SUM(a)          !(4)
 DEALLOCATE(a)              !(5)
END PROGRAM alloc_array
```

図 20. 割付け配列の例

動的割付け配列とポインタ動的割付けの違いは、動的割付け配列が名前を持った領域を確保し、効率の良いプログラムを実現するのに対して、ポインタ動的割付けは、無名の領域を割付け、線形リストなど柔軟な動的データ構造を持つプログラムを表現できることです。

4.8 ポインタの結合状態と文

ポインタ割付けを使った動的なデータ構造を説明する前に、ポインタの結合状態とポインタの状態に密接に関連する **ALLOCATE 文**、**DEALLOCATE 文**、**NULLIFY 文**および状態を調べるための **ASSOCIATED 組込み関数**の機能について説明します。

ポインタは、基本的に次の三つの状態を持ちます。

1) 未定義状態

未定義状態とは、ポインタが宣言されいるだけで何とも結合せず、指示してない状態をいいます。

未定義状態のポインタをプログラムで参照することはできません。

2) 結合状態

結合状態とは、そのポインタが指示している状態をいいます。

未定義状態のポインタが結合状態になるのは、ポインタ代入で **TARGET** 属性を持つ変数と結合し指示した場合と、 **ALLOCATE** 文で割付け領域と結合した場合です。

3) 空状態

空状態とは、このポインタ変数が何も指示していない状態をいいます。

NULLIFY 文に指定されたポインタ変数は、空状態になります。なお、 **ALLOCATE** 文で割付けられたポインタ変数は、 **DEALLOCATE** 文で解放すると、指定されたポインタも空状態になります。

4.8.1 ALLOCATE 文

ALLOCATE 文にポインタを指定した場合、指定された大きさの領域を割付け、ポインタがその領域と結合されます。

ALLOCATE 文は、次のような形式であり、複数のポインタ変数が指定できます。 stat は整変数で割付けが成功した場合、ゼロが設定されます。

```
ALLOCATE(ptr_var,...[,STAT=stat])
```

ただし、 ALLOCATE 文はポインタ変数の領域を割付け、結合して返すだけです。

したがって、次のプログラムのように既に割付けられたものと同じポインタ変数を ALLOCATE 文に指定すると、前の領域は他のポインタから指示されていない限り参照不能になるので注意が必要です。

```
PROGRAM alloc_pointer
 IMPLICIT NONE
 REAL,POINTER :: p1,p2
 ALLOCATE(p1)
 p1=1.0
 p2=>p1
 ALLOCATE(p1)
 p1=2.0
 WRITE(*,*) p1,p2
END PROGRAM alloc_pointer
```

4.8.2 DEALLOCATE 文

DEALLOCATE 文は、ポインタ指示先を解放し、ポインタを空状態にします。

DEALLOCATE 文は、次のような形式であり、複数のポインタ変数が指定できます。 stat は整変数であり解放が成功した場合、ゼロが設定されます。

```
DEALLOCATE(ptr_var,...[,STAT=sta])
```

ただし、 DEALLOCATE 文は指定されたポインタ変数の指示の領域を解放し、ポインタ変数を空状態にするだけです。

したがって、次のプログラムのように他のポインタが指示する領域を DEALLOCATE 文で解放するとそのポインタの状態は不定 (未定義) になってしまいます。

```
PROGRAM dealloc_pointer
 IMPLICIT NONE
 REAL,POINTER :: p1,p2
 ALLOCATE(p1)
 p1=1.0
 p2=>p1
 DEALLOCATE(p1)
END PROGRAM dealloc_pointer
```

4.8.3 NULLIFY 文

NULLIFY 文は、次のような形式であり、指定されたポインタ変数を空状態にします。

```
NULLIFY(ptr_var,...)
```

また、空状態のポインタ変数を他のポインタ変数に、ポインタ代入するとそのポインタ変数も空状態になります。

4.8.4 組込み関数 ASSOCIATED

ポインタ変数の結合状態は、組込み関数 ASSOCIATE で知ることができます。

組込み関数 ASSOCIATED は、次のような形式で指定されたポインタ変数の結合状態を調べ、結果を論理値で返します。

```
ASSOCIATED(pointer [,target])
```

引数に未定義なポインタ変数を指定することはできません。また、引数の指定により次のような結果を返します。

1) target を省略した場合

- pointer が結合状態のときは真 (.TRUE.)
- pointer が空状態のときは偽 (.FALSE.)

2) target が TARGET 属性を持つ変数の場合

- pointer が target を指示していれば真 (.TRUE.)
- 上記以外は偽 (.FALSE.)

3) target が POINTER 属性を持つ変数場合

- pointer が target を指示していれば真 (.TRUE.)
- 上記以外は偽 (.FALSE.)

また、引数 pointer,target のいずれかが空状態の場合、 ASSOCIATE は偽 (.FALSE.) を返します。

```
PROGRAM pointer_list
 IMPLICIT NONE
 TYPE list
  INTEGER :: val
  TYPE(list),POINTER :: next
 END TYPE list
 TYPE(list), POINTER :: root,work
 INTEGER :: data
 ALLOCATE(root)
 NULLIFY(root%next)
 work => root
 create_list: DO
  READ(*,*) data
  IF ( data /= 0 ) THEN
   work%val = data
   ALLOCATE(work%next)
   work=>work%next
   NULLIFY(work%next)
  ELSE
   EXIT create_list
  END IF
 END DO create_list
 work => root
 print_list: DO
  IF (ASSOCIATED(work%next)) THEN
   WRITE(*,*) work%val
   work=>work%next
  ELSE
   EXIT print_list
  END IF
 END DO print_list
 work => root
 free_list: DO
  IF (ASSOCIATED(work%next)) THEN
   root => work
   work=>work%next
   DEALOCATE(root)
  ELSE
   EXIT free_list
  END IF
 END DO free_list
END PROGRAM pointer_list
```

図 21. リスト構造プログラム

4.9 リスト構造プログラム

図 21に、ポインタの動的割付けの例として線形リスト構造プログラムを示します。

4.10 木構造プログラム

図 22に、ポインタの動的割り付けの例として木構造リストプログラムを示します。

```
PROGRAM tree_list
IMPLICIT NONE
 TYPE node
  INTEGER :: val
  TYPE(node),POINTER :: left,right
 END TYPE node
 TYPE(node), POINTER :: tree
 INTEGER :: data
 NULLIFY(tree)
 loop: DO
  READ(*,*) data
  IF ( data == 0  ) EXIT loop
  CALL insert(tree,data)
 END DO loop
 CALL print_tree(tree)
 CONTAINS
  RECURSIVE SUBROUTINE insert(t, data)
   TYPE(node), pointer :: t
   INTEGER, INTENT(IN) :: data
   IF(.not. ASSOCIATED(t)) THEN
    ALLOCATE(t)
    t%val=data
    NULLIFY(t%left)
    NULLIFY(t%right)
   ELSE IF (data < t%val) THEN
    CALL insert(t%left, data)
   ELSE
    CALL insert(t%right, data)
   END IF
  END SUBROUTINE insert
  RECURSIVE SUBROUTINE print_tree(t)
   TYPE(node), pointer :: t
   IF(ASSOCIATED(t)) THEN
    CALL print_tree(t%left)
    WRITE(*,*) t%val
    CALL print_tree(t%right)
   END IF
  END SUBROUTINE print_tree
END PROGRAM tree_list
```

図 22. 木構造プログラム

5 おわりに

連載記事「はじめての Fortran90」シリーズの二回目です。今回は、再帰手続き、利用者定義のデータ型、構造型、ポインタについて解説しました。前号でお約束していたモジュールの解説まで、たどり着けませんでした。

次回は、文字列処理、入出力、モジュールを取り上げたいと考えています。それではまた。