計算物理学II (第10回)

今回の内容

- C言語
  - 多次元配列の割付
  - 構造体
- Fortran
  - Namelist
  - ポインタ
  - 派生型(構造体)
  - その他の話題

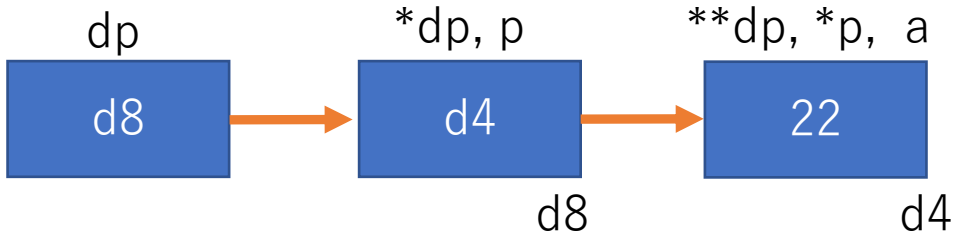
多重間接参照(ダブルポインタ) (C)

ポインタ変数もメモリ上に格納されており、アドレスがある
ポインタ変数を指すポインタを用いることができる(多重間接参照)

```c
#include<stdio.h>

int main(){
  int **dp; // ポインタのポインタ変数
  int *p;
  int a=22;

  printf("a = %d, &a = %p\n", a,&a); //変数aのアドレス
  p=&a;
  printf("*p = %d,  p = %p\n", *p, p); // pはaを指す
  printf("&p = %p\n", &p); //ポインタ変数pのアドレス
  dp=&p; //dpはポインタpを指す
  printf("**dp= %d, *dp = %p, dp = %p\n",**dp, *dp, dp);

}
```

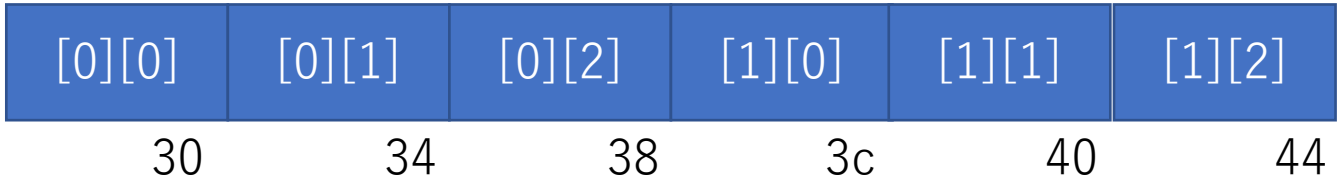
```
a = 22, &a = 0x7ffc117b80d4
*p = 22,  p = 0x7ffc117b80d4
&p = 0x7ffc117b80d8
**dp= 22, *dp = 0x7ffc117b80d4, dp = 0x7ffc117b80d8
```

dpはポインタpのアドレス(&p)を格納
*dpでpに格納されている値(aのアドレスを表示)
**dpでpに格納されている値が指している値(a)を表示



二次元配列(C)

array[2][3]は



の順にメモリ上に配置される

```c
#include<stdio.h>
#include<stdlib.h>

int main(){
  int array[2][3];
  int i,j;

  for(i=0;i<2;i++){
    for(j=0;j<3;j++){
      printf("&array[%d][%d]= %p\n", i,j,&array[i][j]);
      // array[i][j]要素のアドレスを表示
    }
  }
  printf("\n");
  for(i=0;i<2;i++){
    for(j=0;j<3;j++){
      printf("array[%d]+%d= %p\n", i,j, array[i]+j );
      // array[i]+jと&array[i][j]は同じ
    }
  }
  printf("\n");
  for(i=0;i<2;i++){
    printf("array[%d] = %p\n", i, array[i]);
    // array[i]には行ごとの先頭のアドレスが入っている
  }
  printf("\n");
  printf("array = %p\n", array);
  // arrayには配列の先頭のアドレスが入っている

}
```

```
&array[0][0]= 0x7ffd1822c930
&array[0][1]= 0x7ffd1822c934
&array[0][2]= 0x7ffd1822c938
&array[1][0]= 0x7ffd1822c93c
&array[1][1]= 0x7ffd1822c940
&array[1][2]= 0x7ffd1822c944

array[0]+0= 0x7ffd1822c930
array[0]+1= 0x7ffd1822c934
array[0]+2= 0x7ffd1822c938
array[1]+0= 0x7ffd1822c93c
array[1]+1= 0x7ffd1822c940
array[1]+2= 0x7ffd1822c944

array[0] = 0x7ffd1822c930
array[1] = 0x7ffd1822c93c

array = 0x7ffd1822c930
```

配列要素の前に&をつけると
要素のアドレス

[i] はアドレスをiすすめる演算

array[0]は二次元配列の
先頭アドレス
array[1]は [1][0]要素のアドレス

配列名arrayは先頭のアドレス

二次元配列の動的割付(C)

Cの2次元配列はメモリ上に1次元に配置される
→2次元配列を使わず1次元配列としてプログラム上で宣言して扱うのが一番簡単

array[2][3]

別の配列array1D[6]とみなす

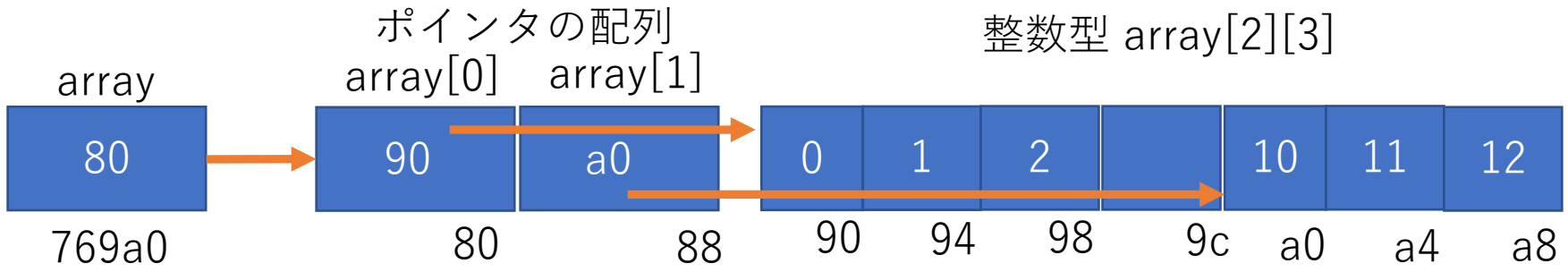
配列array[n][m]の要素array[i][j]は1次元配列ではarray1D[m*i+j]

二次元配列の動的割付(C)

```c
#include<stdio.h>
#include<stdlib.h>
int main(){

  int n, m,i,j;
  int **array; //ポインタ変数へのポインタ
  //配列の次元を標準入力より読み込む
  scanf("%d %d", &n, &m);
  // ポインタの配列を割付
  array = (int **)malloc(n*sizeof(int*));
  // ポインタの配列要素のアドレスそれぞれに整数型を割り当て
  for(i=0;i<n;i++){
    array[i] = (int *)malloc(m*sizeof(int));
    // array[i]の値(アドレス)
    printf("array[%d] = %p\n", i, array[i]);
    // array[i]のアドレス
    printf("&array[%d] = %p\n", i, &array[i]);
  }
  // array自身のアドレス
  printf("&array = %p \n", &array);
  // arrayに格納されているアドレス
  printf("array = %p \n", array);
  // arrayが指すアドレスが格納しているアドレス
  printf("*array = %p \n", *array);
  for(i=0;i<n;i++){   // 値の代入、アドレスの表示など
    for(j=0;j<m;j++){
      array[i][j] = i*10+j;
      printf("array[%d][%d]=%d, &array[%d][%d]=%p\n", i,j,array[i][j],i,j,&array[i][j]);
    }
  }
  for(i=0;i<n;i++){
    free(array[i]);  // 領域の解放は割付と逆順に行う
  }
  free(array);
}
```

```
2 3
array[0] = 0x7fe9bb504090
&array[0] = 0x7fe9bb504080
array[1] = 0x7fe9bb5040a0
&array[1] = 0x7fe9bb504088
&array = 0x7ffeece769a0
array = 0x7fe9bb504080
*array = 0x7fe9bb504090
array[0][0]=0, &array[0][0]=0x7fe9bb504090
array[0][1]=1, &array[0][1]=0x7fe9bb504094
array[0][2]=2, &array[0][2]=0x7fe9bb504098
array[1][0]=10, &array[1][0]=0x7fe9bb5040a0
array[1][1]=11, &array[1][1]=0x7fe9bb5040a4
array[1][2]=12, &array[1][2]=0x7fe9bb5040a8
```

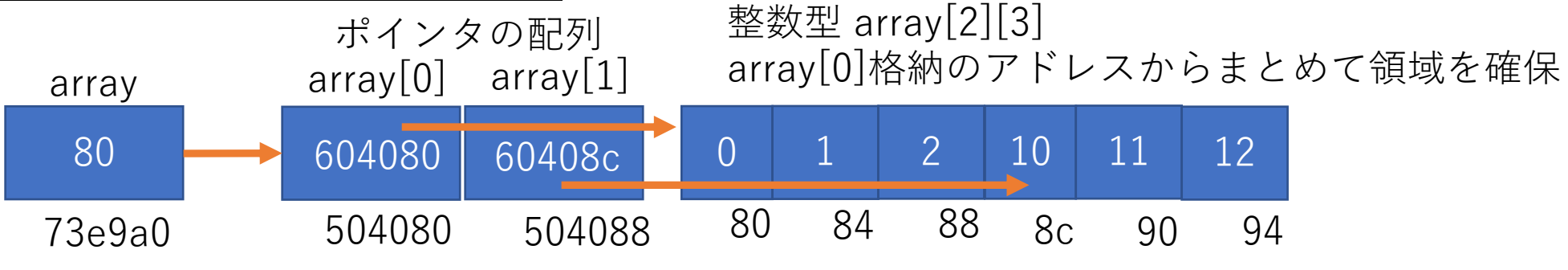
array[0][2]とarray[1][0]の間は連続にならないことがある



二次元配列の動的割付(C)

```c
#include<stdio.h>
#include<stdlib.h>
int main(){

  int n, m,i,j;
  int **array; //ポインタ変数へのポインタ
  //配列の次元を標準入力より読み込む
  scanf("%d %d", &n, &m);
  // ポインタの配列を割付
  array = (int **)malloc(n*sizeof(int*));
  // すべてのメモリを割り当て
  array[0]=(int *)malloc(n*m*sizeof(int));
  printf("&array[0] = %p\n", &array[0]);
  // ポインタの配列要素にアドレスを代入
  for(i=1;i<n;i++){
    array[i] = array[0] + i*m;
    printf("array[%d] = %p\n", i, array[i]);
    // array[i]のアドレス
    printf("&array[%d] = %p\n", i, &array[i]);
  }
  // array自身のアドレス
  printf("&array = %p \n", &array);
  // arrayに格納されているアドレス
  printf("array = %p \n", array);
  // arrayが指すアドレスが格納しているアドレス
  printf("*array = %p \n", *array);
  for(i=0;i<n;i++){   // 値の代入、アドレスの表示など
    for(j=0;j<m;j++){
      array[i][j] = i*10+j;
      printf("array[%d][%d]=%d, &array[%d][%d]=%p\n", i,j,array[i][j],i,j,&array[i][j]);
    }
  }
  free(array[0]);
  free(array);
}
```

```
2 3
&array[0] = 0x7ff465504080
array[1] = 0x7ff46560408c
&array[1] = 0x7ff465504088
&array = 0x7ffeeb73e9a0
array = 0x7ff465504080
*array = 0x7ff465604080
array[0][0]=0, &array[0][0]=0x7ff465604080
array[0][1]=1, &array[0][1]=0x7ff465604084
array[0][2]=2, &array[0][2]=0x7ff465604088
array[1][0]=10, &array[1][0]=0x7ff46560408c
array[1][1]=11, &array[1][1]=0x7ff465604090
array[1][2]=12, &array[1][2]=0x7ff465604094
```



構造体(C)

- 構造体とは：複数の種類の変数をまとめて一つにしたもの

```c
#include<stdio.h>
#include<stdlib.h>
int main(){
  struct xline{
    int n; // 分割点
    double xmin, xmax, length;
    double *x;
  };

  struct xline x1, x2;
  int i;

  x1.xmin = 0.0; x1.xmax = 1.0;
  x1.length = (x1.xmax-x1.xmin);
  x1.n = 5;
  x1.x = (double *)malloc(x1.n*sizeof(double));
  x1.x[0] = x1.xmin;
  printf("x1.x[0]=%f \n", x1.x[0]);
  for(i=1; i<x1.n-1; i++){
    x1.x[i] = x1.xmin+(x1.xmax-x1.xmin)/(x1.n-1)*i;
    printf("x1.x[%d]=%f \n", i, x1.x[i]);
  }
  x1.x[x1.n-1] = x1.xmax;
  printf("x1.x[%d]=%f \n", x1.n-1,x1.x[x1.n-1]);
  x2=x1;
  printf("x2.x[%d]=%f \n", 2,x2.x[2]);
  free(x1.x);
}
```

一つの構造体xlineの中に
整数、倍精度実数、倍精度実数へのポインタなどまとめられる

xline型の構造体としてx1とx2を宣言 (struct 構造体名 変数名;)

構造体のメンバには.でアクセスできる

構造体の中のポインタに1次元配列を割当

minからxmaxまでの間をn-1分割した座標点の値を代入

同じ型の構造体はコピー(代入)できる

x2.xはポインタでx1.xと同じアドレスなのでfreeはx1.xだけでよい

実行結果

```
x1.x[0]=0.000000
x1.x[1]=0.250000
x1.x[2]=0.500000
x1.x[3]=0.750000
x1.x[4]=1.000000
x2.x[2]=0.500000
```

構造体(C)

```c
#include<stdio.h>
#include<stdlib.h>
int main(){
  typedef struct xline_{
    int n; // 分割点
    double xmin, xmax, length;
    double *x;
  } xline;

  xline x1;
  int i;

  x1.xmin = 0.0; x1.xmax = 1.0;
  x1.length = (x1.xmax-x1.xmin);
  x1.n = 5;
  x1.x = (double *)malloc(x1.n*sizeof(double));
  x1.x[0] = x1.xmin;
  printf("x1.x[0]=%f \n", x1.x[0]);
  for(i=1; i<x1.n-1; i++){
    x1.x[i] = x1.xmin+(x1.xmax-x1.xmin)/(x1.n-1)*i;
    printf("x1.x[%d]=%f \n", i, x1.x[i]);
  }
  x1.x[x1.n-1] = x1.xmax;
  printf("x1.x[%d]=%f \n", x1.n-1,x1.x[x1.n-1]);
  free(x1.x);
}
```

typedef struct 構造体名 {構造体のメンバの宣言} 別名;

とすると
別名 構造体変数名;
だけで構造体変数の宣言ができる

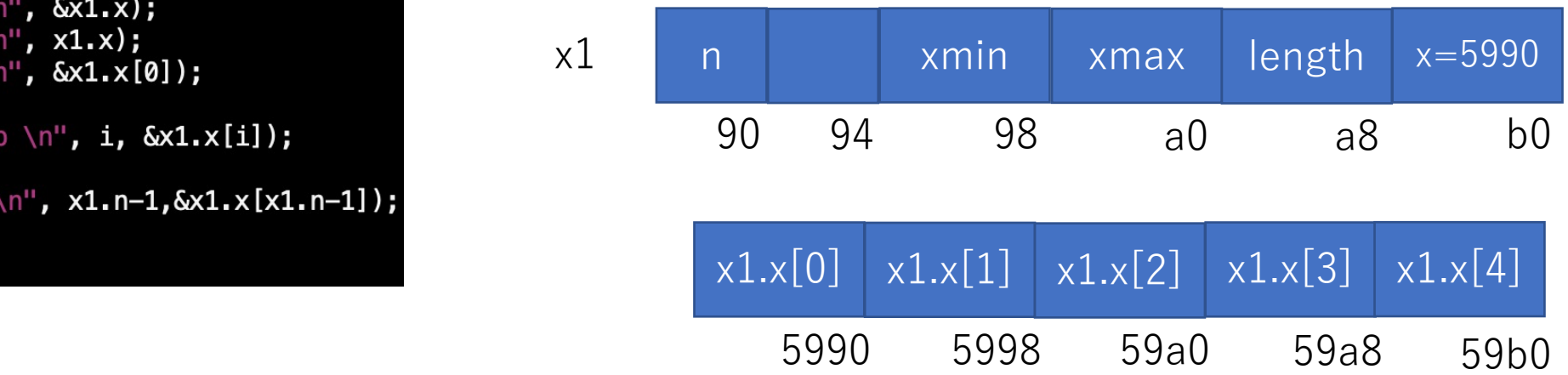
構造体のアドレス(C)

```c
#include<stdio.h>
#include<stdlib.h>
int main(){
  typedef struct xline_{
    int n; // 分割点
    double xmin, xmax, length;
    double *x;
  } xline;

  xline x1;
  int i;

  printf("&x1         = %p\n", &x1);
  x1.n = 5;
  printf("&x1.n       = %p\n", &x1.n);
  printf("&x1.xmin    = %p\n", &x1.xmin);
  printf("&x1.xmax    = %p\n", &x1.xmax);
  printf("&x1.length = %p\n", &x1.length);
  x1.x = (double *)malloc(x1.n*sizeof(double));
  printf("&x1.x       = %p \n", &x1.x);
  printf("x1.x        = %p \n", x1.x);
  printf("&x1.x[0]    = %p \n", &x1.x[0]);
  for(i=1; i<x1.n-1; i++){
    printf("&x1.x[%d]    = %p \n", i, &x1.x[i]);
  }
  printf("x1.&x[%d]    = %p \n", x1.n-1,&x1.x[x1.n-1]);
  free(x1.x);
}
```

実行結果

```
&x1         = 0x7ffedfe3c990
&x1.n       = 0x7ffedfe3c990
&x1.xmin    = 0x7ffedfe3c998
&x1.xmax    = 0x7ffedfe3c9a0
&x1.length = 0x7ffedfe3c9a8
&x1.x       = 0x7ffedfe3c9b0
x1.x        = 0x7fb716405990
&x1.x[0]    = 0x7fb716405990
&x1.x[1]    = 0x7fb716405998
&x1.x[2]    = 0x7fb7164059a0
&x1.x[3]    = 0x7fb7164059a8
x1.&x[4]    = 0x7fb7164059b0
```

構造体変数のアドレス=構造体の先頭の変数のアドレス

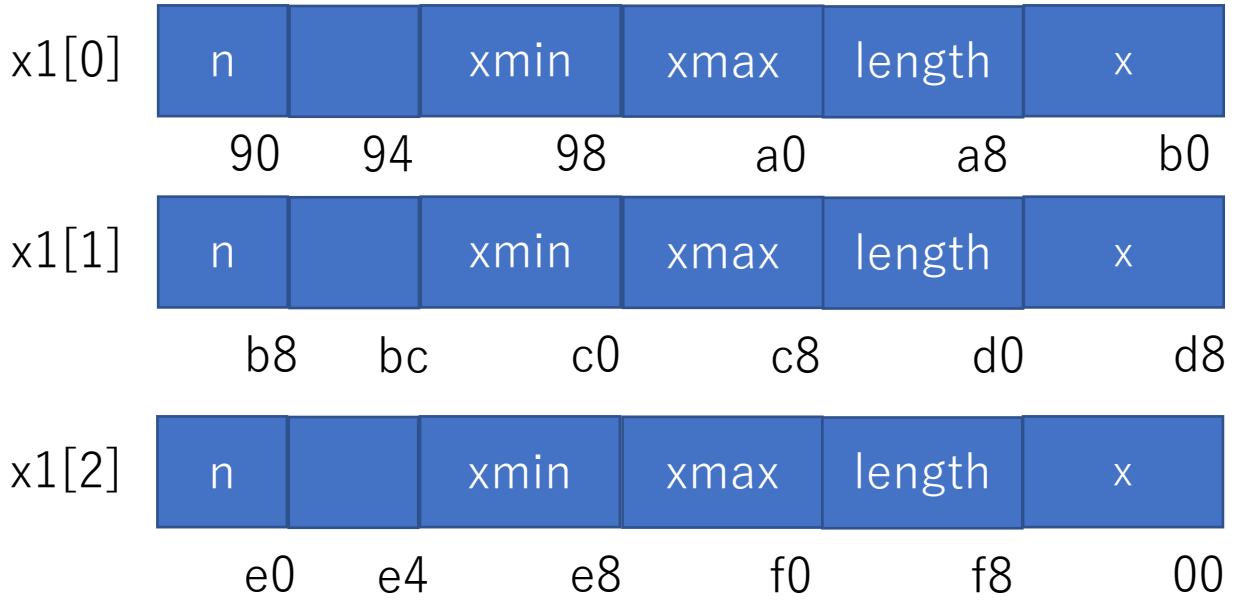


構造体へのポインタ(C)

```c
#include<stdio.h>
#include<stdlib.h>
int main(){
  typedef struct xline_{
    int n; // 分割点
    double xmin, xmax, length;
    double *x;
  } xline;

  xline *x1; xline型へのポインタ変数
  int i;
                    5つのxline型の領域を確保
  x1 = malloc(5*sizeof(xline));
  for(i=0;i<5;i++){
    printf("&x1[%d]        = %p\n", i,&x1[i]);
  }
  x1[0].n=5; x1[1].n=10;

  free(x1);
}
```

実行結果

```
&x1[0]        = 0x7fcfc1c05990
&x1[1]        = 0x7fcfc1c059b8
&x1[2]        = 0x7fcfc1c059e0
&x1[3]        = 0x7fcfc1c05a08
&x1[4]        = 0x7fcfc1c05a30
```



構造体のポインタ(C)

```c
#include<stdio.h>
#include<stdlib.h>
int main(){
  typedef struct xline_{
    int n; // 分割点
    double xmin, xmax, length;
    double *x;
  } xline;

  xline *x1, x2;
  int i;

  x1 = &x2;
  x1->n=5;

  printf("x2.n = %d\n", x2.n);
  printf("x1->n = %d\n", x1->n);
}
```

x1は構造体へのポインタ、x2は構造体

x1にx2のアドレスを代入

x1が指している構造体のメンバの値を更新(x2.nに5が代入される)

構造体のポインタが他の構造体を指している場合
メンバには -> (アロー演算子) でアクセスする

実行結果

```
x2.n = 5
x1->n = 5
```

Namelist(Fortran)

ファイルからの初期パラメータの読み込み

```
program main
  implicit none

  integer :: a, b
  real*8 :: c

  open(10,file="input.txt", action="read")

  read(10,*) a, b, c

  close(10)

end program main
```

input.txtにはreadと同じ順番で数値を書いておく

```
10 10 2.0
```

入力パラメータの数が増えると分かりづらい

```
program main
  implicit none
  integer :: a, b
  real*8 :: c
  namelist /input/a,b,c
  open(10,file="input.txt", action="read")
  read(10,nml=input)
  close(10)
  print *, "a = ", a, "b = ", b, "c = ", c
end program main
```

a,b,cをnamelist ”input”としてまとめる

input.txtにはnamelist形式で記述する。記述の順番は任意

```
&input
c = 2.0,
a = 10,
b = 10,
/
```

複数のnamelistを定義することもできる。
一つのファイルに複数のnamelistを書いておいてもよい(namelistの数だけreadする)

ポインタ(Fortran)

ポインタは変数を指すことができる

```
program main
implicit none

  integer, pointer :: ptr    ! 整数型へのポインタ
  real*8, pointer :: ptr2(:) ! 倍精度実数型の1次元配列へのポインタ

  integer, target :: j=2  ! ポインタの指示先にはtarget属性をつけて宣言する
  real*8, target :: a(5)

  integer :: i
  do i = 1, 5
     a(i) = i/2.0d0
  end do
  ptr=>j ! ptrは整数型jを指す
  print *, "ptr = ", ptr
  ptr = ptr + 1 !ポインタの指す変数の値も変更できる
  print *, "ptr = ", ptr
  print *, "j   = ", j
  ptr2=>a ! ptr2は配列aを指す
  print *, "ptr2(3) = ", ptr2(3)

end program main
```

ポインタはpointer属性をつけて宣言
ポインタの指示先はtarget属性をつけて宣言
配列へのポインタはサイズを指定しない

ポインタの指示先を設定するには
ポインタ代入 => を使う

実行結果

```
 ptr =                2
 ptr =                3
 j   =                3
 ptr2(3) =    1.5000000000000000
```

ポインタ(Fortran)

```
program main
  implicit none

  integer, pointer :: ptr    ! 整数型へのポインタ
  real*8, pointer :: ptr2(:) ! 倍精度実数型の1次元配列へのポインタ

  integer, target :: j=2  ! ポインタの指示先にはtarget属性をつけて宣言する
  real*8, target :: a(5)

  ptr=>null()  ! ポインタを初期化
  nullify(ptr,ptr2) ! nullifyでまとめて初期化してもよい
  print *, "ptr = ", ptr !初期化されているとゼロを返す
  ! 初期化されて指示先が設定されていなポインタは.False.を返す
  print *, "associated(ptr) = ", associated(ptr)

  ptr=>j ! ptrに指示先を設定
  ! 指示先が設定されると.True.を返す
  print *, "associated(ptr) = ", associated(ptr)
  print *, "ptr = ", ptr

end program main
```

ポインタの初期化には
null()を指示先に設定するか、
nullify関数を用いる

associated関数でポインタの
指示先が設定されているかどうかを
調べることができる

実行結果

```
ptr =                    0
associated(ptr) =  F
associated(ptr) =  T
ptr =                    2
```

ポインタ(Fortran)

```
program main
  implicit none

  integer, pointer :: ptr     ! 整数型へのポインタ
  real*8, pointer :: ptr2(:) ! 倍精度実数型の1次元配列へのポインタ
  integer :: i
  allocate(ptr, ptr2(5)) ! 指示先にメモリ領域を割り当てることができる

  ptr = 2
  print *, "ptr = ", ptr

  do i = 1, 5
     ptr2(i) = i/2.0d0
  end do
  print *, "ptr2(3) = ", ptr2(3)

  deallocate(ptr,ptr2) !割り当てた領域を解放

end program main
```

ポインタはallocateで指示先に
領域を割り当てることができる

割り当てたあとは通常の変数として
使用できる

使い終わったらdeallocateで
割り当てた領域を解放する

実行結果

```
ptr =            2
ptr2(3) =    1.5000000000000000
```

派生型(構造体)（Fortran)

- 構造体とは：複数の種類の変数をまとめて一つにしたもの

```
program main
  implicit none

  type xline
     integer :: n
     real*8 :: xmin, xmax, length
     real*8, allocatable :: x(:)
  end type xline

  type(xline) :: x1, x2 ! xline型の変数の宣言
  integer :: i
  x1%xmin = 0.0d0; x1%xmax = 1.0d0
  x1%length = x1%xmax - x1%xmin
  x1%n = 5
  allocate(x1%x(5))
  x1%x(1) = x1%xmin
  print *, "x1%x(1) = ", x1%x(1)
  do i = 2, x1%n-1
     x1%x(i) = x1%xmin + x1%length/(x1%n-1)*(i-1)
     print *, "x1%x(",i,")= ", x1%x(i)
  end do
  x1%x(x1%n) = x1%xmax
  print *, "x1%x(",x1%n,")= ", x1%x(x1%n)
  print *, "sizeof(x2%x) = ", sizeof(x2%x)
  x2 = x1 ! 構造体のコピーを他の構造体変数に代入
  print *, "sizeof(x2%x) = ", sizeof(x2%x) !5要素=40バイト
  print *, "x2%x(",x2%n,")= ", x2%x(x2%n)

  deallocate(x1%x)
  deallocate(x2%x)

end program main
```

type 派生型名
　派生型のメンバの宣言
end type 派生型名

派生型変数の宣言は type(派生型名) :: 変数名

派生型のメンバには 派生型名%変数名 でアクセスできる

同じ型の他の派生型変数に代入してコピーを作成できる
allocatableな配列もx1と同じサイズにallocateされる

実行結果

```
x1%x(1) =    0.0000000000000000
x1%x(           2 )=   0.25000000000000000
x1%x(           3 )=   0.50000000000000000
x1%x(           4 )=   0.75000000000000000
x1%x(           5 )=    1.0000000000000000
sizeof(x2%x) =                           8
sizeof(x2%x) =                          40
x2%x(           5 )=    1.0000000000000000
```

変数の初期化(Fortran)

宣言時の変数の初期化は初回しか行われない(save属性)

```
program main
  implicit none
  call func
  call func
  call func
contains
  subroutine func
    implicit none
    integer :: i = 0
    i = i + 1
    print *, "i = ", i
    return
  end subroutine func
end program main
```

実行結果。２回目にサブルーチンが呼ばれたとき
内部変数iの値は前回の値が保持されている

```
 i =           1
 i =           2
 i =           3
```

```
program main
  implicit none
  call func
  call func
  call func
contains
  subroutine func
    implicit none
    integer :: i
    i = 0
    i = i + 1
    print *, "i = ", i
    return
  end subroutine func
end program main
```

実行結果。サブルーチンが呼ばれるごとに
i=0, i=i+1が実行されている

```
 i =           1
 i =           1
 i =           1
```

引数のoptional属性(Fortran)

```
program main
  implicit none
  real*8 :: a=1.0d0, b=2.0d0, c=4.0d0
  print *, add(a,b,c)
  print *, add(a,b)
contains
  function add(x,y,z)
    real*8, intent(in) :: x,y
    real*8, intent(in), optional :: z
    real*8 :: add
    if(present(z)) then !zが引数として渡されている場合
       add = x + y + z
    else ! zが引数にない場合
       add = x + y
    end if
    return
  end function add
end program main
```

optional属性をつけた変数は引数になくてもよい

present関数は引数として渡されていれば.True.
渡されていなければ.False.を返す

実行結果

```
 7.0000000000000000
 3.0000000000000000
```

a+b+cが実行される
a+ b が実行される

変数のキーワード引数(Fortran)

通常は関数・サブルーチンの引数の順番は定義と合わせる必要があり、optional変数は最後にまとめる。
キーワード引数を使うと呼び出すときに任意の順番で変数を記述できる

```
program main
  implicit none
  real*8 :: a=1.0d0, b=2.0d0, c=4.0d0

  print *, subtract(x=a,y=b) ! 1.0-2.0が実行される
  print *, subtract(y=a,x=b) ! 2.0-1.0が実行される
  print *, subtract(z=c,x=a,y=b) ! 1.0-2.0-4.0が実行される

contains
  function subtract(x,y,z)
    real*8, intent(in) :: x, y
    real*8, intent(in), optional :: z
    real*8 :: subtract
    if(present(z))then
       subtract = x - y - z
    else
       subtract = x - y
    end if
    return
  end function subtract
end program main
```

実行結果

```
-1.0000000000000000
 1.0000000000000000
-5.0000000000000000
```

モジュールへのアクセス制限(Fortran)

```
module var
  implicit none
  private ! デフォルトをprivate(モジュール外から参照不可能)にする
  integer, public :: a, b  !モジュール外から参照可能
  integer, private :: c, d !モジュール外から参照不可能
  public :: set_cd ! 関数、サブルーチンなどもprivate/publicの指定ができる
contains
  subroutine set_cd
    implicit none
    a = a + 1; b = b + 1
    c = 4; d = 5
    !モジュール内のサブルーチンからはprivate属性の変数の値を更新できる
    print *, "set_cd: a = ", a , "b = ", b
    print *, "set_cd: c = ", c , "d = ", d
  end subroutine set_cd
end module var
program main
  use var, only: a,e=>b,set_cd
  ! only以下にリストした変数、関数、サブルーチン等だけが参照可能
  ! e=>bでモジュールのbという変数をeという名前で使用する
  implicit none
  integer :: c=0, d=0 !モジュールのc,dとは別の変数
  a=1; e=2 !モジュールの変数の値の更新ができる
  print *, "main: a = ", a, "b = ", e
  call set_cd !モジュール内のset_cdサブルーチンを呼び出す
  print *, "main: a = ", a, "b = ", e !set_cdで値が変更されている
  print *, "main: c = ", c, "d = ", d
end program main
```

モジュールの変数
public属性をつけるとモジュール外から参照可能、
private属性のものはモジュール内の関数、
サブルーチン副プログラムからのみ参照可能

use モジュール名, only: …とすることで
module内の特定の変数、関数などのみを参照可能とできる

実行結果

```
main: a =           1 b =           2
set_cd: a =           2 b =           3
set_cd: c =           4 d =           5
main: a =           2 b =           3
main: c =           0 d =           0
```