計算物理学II (第9回)

今回の内容

- 演習(消化できていないところを)
- 第３回レポート(提出締切：12/28)
- 配列の動的割付(C/Fortran)
- 関数の引数について
- ポインタ(C言語)

配列のサイズに変数を使う

- 通常の配列の領域確保では次元を整数値で設定
  複数の同じサイズの配列がある場合に変更が大変

```
program main
  implicit none
  integer :: array(5)

end program main
```

```
#include<stdio.h>

int main(){
  int array[5];

}
```

- 配列のサイズに変数を使う

配列より手前でparameter属性をつけて変数を宣言
parameter属性がついた変数の値は変更できない

```
program main
  implicit none
  integer, parameter :: n = 5
  integer :: array(n)

end program main
```

マクロ定数
(コンパイル時にNは5に置き換えられる)

```
#include<stdio.h>
#define N 5
int main(){
  int array[N];

}
```

変数を使うと複数の同じサイズの配列があるときに書き換える箇所は１箇所で済む

関数/サブルーチンに配列を渡す(Fortran)

サイズも引数で渡す場合

```
program main
  implicit none
  integer, parameter :: n = 5
  integer :: array(n)
  array = 4
  call print_array(n,array)
  call print_array(3,array)
contains
  subroutine print_array(m,array_to_print)
    implicit none
    integer,intent(in) :: m
    integer,dimension(1:m),intent(in) :: array_to_print
    print *, array_to_print(1:m)
  end subroutine print_array
end program main
```

配列のサイズはintent(in)属性とすると
配列の次元の宣言で使える

サイズは関数側で計算する場合

```
program main
  implicit none
  integer, parameter :: n = 5
  integer :: array(n)
  array = 4
  call print_array(array)
  call print_array(array(1:3))
contains
  subroutine print_array(array_to_print)
    implicit none
    integer,dimension(:),intent(in) :: array_to_print
    integer :: size
    size = sizeof(array_to_print(:))/4
    print *, array_to_print(1:size)
  end subroutine print_array
end program main
```

関数内では配列のサイズは指定しないで宣言
sizeofはバイト数を返すのでinteger型なら4バイトで割る

実行結果

```
           4           4           4           4           4
           4           4           4
```

関数に配列を渡す(C)

サイズも引数で渡す必要がある

```
#include<stdio.h>
void array_to_print(int m, int array[m]){
  int i;
  for(i = 0; i<m; i++){
    printf("%d ", array[i]);
  }
  printf("\n");
}
int main(){
  int array[5]={0,1,2,3,4};
  array_to_print(5,array);
  array_to_print(3,array);
}
```

←array[m]はarray[]でもよい

実行結果

```
0 1 2 3 4
0 1 2
```

C言語では配列を関数に渡すとき関数内から要素数がわからない

配列の動的割付(Fortran)

- コンパイル時に配列のサイズが決まらない場合

```
program main
  implicit none
  integer :: n
  integer, allocatable :: array(:)
  write(*,*) "n = ?"
  read(*,*) n
  print *, "allocated? ", allocated(array)
  allocate(array(n))
  print *, "allocated? ", allocated(array)
  array= 3
  print *, array(1:n)
  deallocate(array)
end program main
```

実行結果

```
 n = ?
5          ←標準入力より5を入力
 allocated?  F
 allocated?  T
           3           3           3           3           3
```

allocatable属性を付けて配列を宣言、allocateでメモリに配列の領域を実行時に確保
deallocateでメモリに確保した配列領域を解放
allocatedで配列の割付状況がわかる(.True. or .False.)
すでにallocateされているものを再割付したり、allocateされてないものをdeallocateするとエラーになる

サブルーチンの引数の値を更新する(Fortran)

```
program main
  implicit none
  real*8 :: x, y
  x = 3.0; y = 0.0;
  print *, "main: x = ", x, "y = ", y
  call func(x,y)
  print *, "main: x = ", x, "y = ", y
contains
  subroutine func(x,y)
    real*8, intent(in) :: x
    real*8, intent(out) :: y
    y = x**2
    print *, "func: x = ", x, "y = ", y
    return
  end subroutine func
end program main
```

intent(in)をつけるとサブルーチン内で変数の値を変更できない
intent(out)をつけるとサブルーチン内で変数の値を更新できる

実行結果

```
main: x =    3.0000000000000000       y =    0.0000000000000000
func: x =    3.0000000000000000       y =    9.0000000000000000
main: x =    3.0000000000000000       y =    9.0000000000000000
```

変数のアドレス(C)

宣言された変数はメモリ上に格納する領域が確保され、アドレスが割り当てられる。
変数aのメモリ上のアドレスは &a (&はアドレス演算子)
配列b[3]の先頭の要素b[0]のアドレスは b または &b[0]
%pは変数のメモリ上でのアドレスを表示(16進数)
表示される変数のアドレスは実行するごとに異なる

```c
#include<stdio.h>

int main(){
  int a, b[3];
  double c, d[3];

  printf("&a    = %p\n", &a);     //aのアドレス値
  printf("b     = %p\n", b);      //b[0]のアドレス値
  printf("&b[0] = %p\n", &b[0]);  //b[0]のアドレス値
  printf("&b[1] = %p\n", &b[1]);  //b[1]のアドレス値
  printf("&b[2] = %p\n", &b[2]);  //b[2]のアドレス値

  printf("&c    = %p\n", &c);     //cのアドレス値
  printf("d     = %p\n", d);      //d[0]のアドレス値
  printf("&d[0] = %p\n", &d[0]);  //d[0]のアドレス値
  printf("&d[1] = %p\n", &d[1]);  //d[1]のアドレス値
  printf("&d[2] = %p\n", &d[2]);  //d[2]のアドレス値

}
```

16進数で表示されたメモリ上のアドレス

```
&a    = 0x7ffc12176e74
b     = 0x7ffc12176e84
&b[0] = 0x7ffc12176e84
&b[1] = 0x7ffc12176e88
&b[2] = 0x7ffc12176e8c
&c    = 0x7ffc12176e78
d     = 0x7ffc12176e90
&d[0] = 0x7ffc12176e90
&d[1] = 0x7ffc12176e98
&d[2] = 0x7ffc12176ea0
```

bと&b[0]は同じ

b[1]のアドレスは
4バイト後ろ

d[1]のアドレスは
d[0]の8バイト後ろ

ポインタとは(C)

- ポインタ変数には変数のアドレスを格納 (%pでアドレスを表示できる)
- *(間接演算子)をポインタ変数の手前につけるとポインタの指す先の値にアクセスできる。

```c
#include<stdio.h>

int main(){
  int a = 1, b = 2;
  int *p; // pは整数型へのポインタ変数

  printf("&a = %p\n", &a); // aのアドレス値
  printf("&b = %p\n", &b); // bのアドレス値

  p = &a; // pにaのアドレスを代入
  printf("p = %p\n", p); // ポインタ自体の値(アドレス値)
  printf("a=%d,b=%d,*p=%d\n", a,b,*p); // *pは変数aの値
  a = 3;
  printf("a=%d,b=%d,*p=%d\n", a,b,*p);

  p = &b; // pにbのアドレスを代入
  printf("p = %p\n", p); // ポインタ自体の値(アドレス値)
  printf("a=%d,b=%d,*p=%d\n", a,b,*p);
  a = 0;
  printf("a=%d,b=%d,*p=%d\n", a,b,*p);
  // aの値を変えても*pは変わらない

}
```

```
&a = 0x7ffc48f0fa08
&b = 0x7ffc48f0fa0c
p = 0x7ffc48f0fa08
a=1,b=2,*p=1
a=3,b=2,*p=3
p = 0x7ffc48f0fa0c
a=3,b=2,*p=2
a=0,b=2 *p=2
```

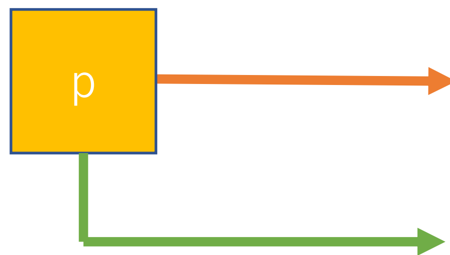
aとbのアドレスは
宣言時に割り当てられる

p=&aによりpにaのアドレスが格納される

*pはaに格納された値を示す

p=&bによりpにbのアドレスが格納される

*pはbに格納された値を示す



アドレス：

a
0x7ffc48f0fa08~
0x7ffc48f0fa0b

b
0x7ffc48f0fa0c~
0x7ffc48f0fa0f

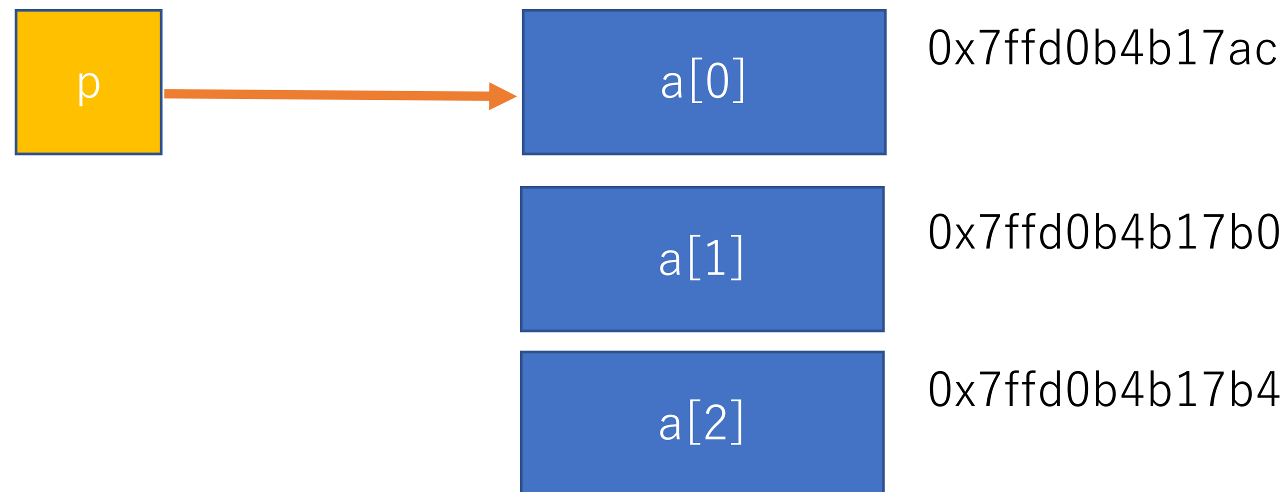
配列とポインタ(C)

配列の任意の要素のアドレスをポインタに格納すれば要素の値も参照できる

```c
#include<stdio.h>

int main(){

  int a[3]; // 3要素の配列
  int *p; // 整数型へのポインタ変数
  int i;

  a[0]=12; a[1]=24; a[2]=36;

  for(i=0; i<3; i++){
    printf("a[%d]=%d, &a[%d] = %p\n", i, a[i], i, &a[i]);
  }

  printf(" a = %p\n", a);
  // aは&a[0]と同じく配列の先頭のアドレス

  p=a; //pに配列の先頭アドレスを代入
  printf(" *p = %d\n", *p);
  p=&a[1]; //pにa[1]のアドレスを代入
  printf(" *p = %d\n", *p);
  p=&a[2]; // pにa[2]のアドレスを代入
  printf(" *p = %d\n", *p);
}
```

```
a[0]=12, &a[0] = 0x7ffd0b4b17ac
a[1]=24, &a[1] = 0x7ffd0b4b17b0
a[2]=36, &a[2] = 0x7ffd0b4b17b4
 a = 0x7ffd0b4b17ac
 *p = 12
 *p = 24
 *p = 36
```

配列(1要素4バイトでアドレスは連続)



ポインタ演算(C)

ポインタは間接参照以外に加減算、等号不等号を用いた値の比較が可能

```
#include<stdio.h>

int main(){

  int a[3]; // 3要素の配列
  int *p; // 整数型へのポインタ変数
  int i;

  a[0]=12; a[1]=24; a[2]=36;

  for(i=0; i<3; i++){
    printf("a[%d]=%d, &a[%d] = %p\n", i, a[i], i, &a[i]);
  }

  printf(" a = %p\n", a);
  // aは&a[0]と同じく配列の先頭のアドレス

  p=a; //pに配列の先頭アドレスを代入
  printf(" *p = %d\n", *p);
  p++; //pのアドレスを1つ(4バイト)すすめる
  printf(" *p = %d\n", *p);
  p++; // pのアドレスを1つ(4バイト)すすめる
  printf(" *p = %d\n", *p);
}
```

```
a[0]=12, &a[0] = 0x7fff85a4c58c
a[1]=24, &a[1] = 0x7fff85a4c590
a[2]=36, &a[2] = 0x7fff85a4c594
 a = 0x7fff85a4c58c
 *p = 12
 *p = 24
 *p = 36
```

整数型へのポインタは1増やすと
アドレスが4バイト増える

倍精度実数型へのポインタは1増やすと
アドレスが8バイト増える

ポインタを使う例：関数との値のやりとり(C)

```c
#include<stdio.h>

void func(double x, double y){
  printf("in func: &x = %p &y = %p\n", &x,&y);
  y = x*x;
  printf("in func; x = %f, y = %f\n", x, y);
}

int main(){
  double x,y;
  x = 2.0; y=0.0;
  printf("in main: &x = %p, &y = %p\n", &x,&y);
  func(x,y);
  printf("in main: x = %f, y = %f\n", x, y);

}
```

```
in main: &x = 0x7ffe423f9c08, &y = 0x7ffe423f9c10
in func: &x = 0x7ffe423f9bd8 &y = 0x7ffe423f9bd0
in func; x = 2.000000, y = 4.000000
in main: x = 2.000000, y = 0.000000
```

関数funcの中でyを計算してもmainに戻ったときに
値を参照できない

関数に値渡しをすると関数側で値がコピーされた新しい領域が用意される
関数の中で計算された変数には呼び出し元のmainから参照できない

ポインタを使う例：関数との値のやりとり(C)

```c
#include<stdio.h>

void func(double x, double *y){ // yはポインタ変数
  printf("in func: &x = %p, y = %p\n", &x,y);
  *y = x*x; // yの指す値にx*xを代入
  printf("in func; x = %f, y = %f\n", x, *y);
}

int main(){
  double x,y;
  x = 2.0; y=0.0;
  printf("in main: &x = %p, &y = %p\n", &x,&y);
  func(x,&y); // yのアドレスを渡す
  printf("in main: x = %f,  y = %f\n", x, y);

}
```

```
in main: &x = 0x7ffc3d66cc68, &y = 0x7ffc3d66cc70
in func: &x = 0x7ffc3d66cc38,  y = 0x7ffc3d66cc70
in func; x = 2.000000, y = 4.000000
in main: x = 2.000000, y = 4.000000
```

mainからfuncにyのアドレスが渡される。
xは値がコピーされるため
mainとfuncでxのアドレスが異なる

関数の中からyの値を更新できる

関数の中で更新したい変数はアドレスを渡す。
関数側ではポインタ変数として宣言する。

ポインタを使う例：配列の動的割付(C)

- コンパイル時にサイズが定まらない配列
- サイズの大きな配列(スタック領域に収まらない場合)

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

int main(){
  int *array; // 整数型へのポインタ
  int n,i;

  scanf("%d", &n); // nの値を標準入力から読み込む

  array = (int *)malloc(n*sizeof(int));
  // mallocを使ってn要素のint型の領域を確保、先頭のアドレスをarrayポインタに代入

  for(i=0;i<n;i++){
    array[i] = i; // *(array+i)=i; と同値
    printf("array[%d]=%i, &array[%d]=%p\n",i,array[i],i,&array[i]);
  }

  free(array); // 確保した領域を解放
}
```

n=10として割り付けた例

```
10
array[0]=0, &array[0]=0xf84420
array[1]=1, &array[1]=0xf84424
array[2]=2, &array[2]=0xf84428
array[3]=3, &array[3]=0xf8442c
array[4]=4, &array[4]=0xf84430
array[5]=5, &array[5]=0xf84434
array[6]=6, &array[6]=0xf84438
array[7]=7, &array[7]=0xf8443c
array[8]=8, &array[8]=0xf84440
array[9]=9, &array[9]=0xf84444
```