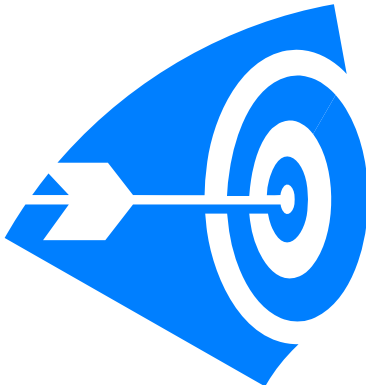
第13回 ポインタ



1

今回の目標

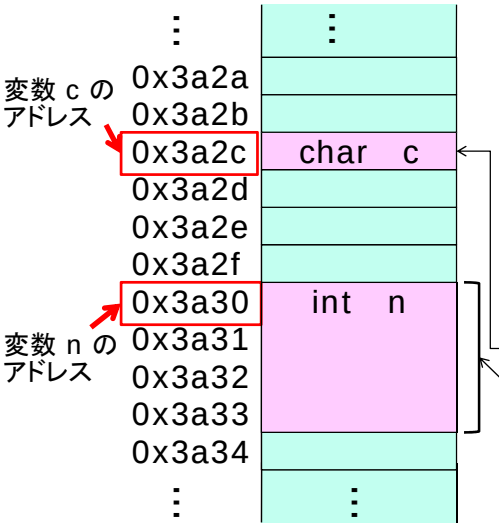
- C言語におけるポインタを理解する。
- 変数のアドレスを理解する。
- ポインタ型を理解する。
- アドレス演算子、間接演算子の効果を理解する。

☆他の関数で定義されたベクトルの和を求める関数を作成する。

2

変数とアドレス

ad･dress〔 ədrés 〕
住所、宛て先

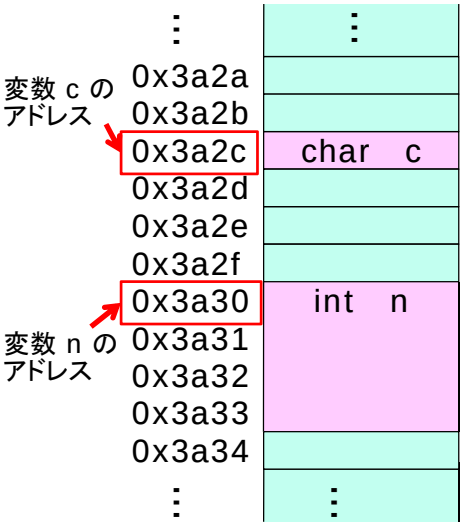


- メインメモリ内では、1バイトごとに一つのアドレスが割り当てられている。
- アドレスは16進数（0x****）で表される。
- ある変数に割り当てられた領域の先頭アドレスを、その変数のアドレスと呼ぶ。

char型は 1 バイト
int型は 4 バイト

アドレス演算子

&：変数に割り当てられたアドレスを求める演算子。
前置の単項演算子。



書式

& 変数名

左の例の場合、
&c は 0x3a2c
&n は 0x3a30

アドレスを表示するprintf文の変換仕様

printf文には、アドレスを表示するための変換仕様がある。

%p ⟷ アドレス

変数 a のアドレスを表示させる書式

```
printf("%p",&a);
```

アドレスを表示するために&をつける

変数 a の型に関わらず、
上の書式で a のアドレスが出力される。

5

練習1

```
/* address.c  アドレス表示実験     */
#include <stdio.h>
int main()
{
    /*変数宣言*/
    int n;
    double x;
    /*変数への値の代入*/
    n=1;
    x=0;
    /*変数のアドレスと値を表示*/
    printf("int型の変数nのアドレス: %p¥n", &n);
    printf("int型の変数nの値: %d¥n", n);
    printf("¥n");

    printf("double型の変数xのアドレス: %p¥n",  &x);
    printf("double型の変数xの値: %f ¥n",   x);
    return 0;
}
```

6

ポインタ

pointer〔pɔ́intə〕
指す人、助言、ヒント

ポインタとは、変数のアドレスを入れる変数である。

ポインタの型は
これまでのどの型(char,int,double)とも異なる。

7

ポインタと記号＊(その1):ポインタの宣言

変数のアドレスを入れるための変数(ポインタ)の用意の仕方。

宣言

```
データ型　＊ポインタの名前;
```

例

```
char *p;
```
文字型の変数のアドレス専用

```
int *q;
```
整数型の変数のアドレス専用

```
double *r;
```
実数型の変数のアドレス専用

ポインター＝変数のアドレスを入れるための入れ物
(ただし、用途別)

pは(char *)型の変数と考えてもよい。
char型と(char *)型は異なる型。

8

ポインタへのアドレスの代入

```
int    n;
int    *p;  /*ポインタ*/

p=(&n);
/*  p には n のアドレスが入る*/
```

ポインタpがある変数nのアドレスを蓄えているとき、
ポインタpは変数nを指すという。

あるいは、pは変数nへのポインタであるという。

9

ポインタと記号＊（その2）:間接演算子

ポインタに、変数のアドレスを代入すると、間接演算子＊でそのポインタが指す変数の値を参照できる。

書式

＊ポインタ

＊：ポインタに格納されているアドレスに割り当てられている変数を求める演算子。
前置の単項演算子

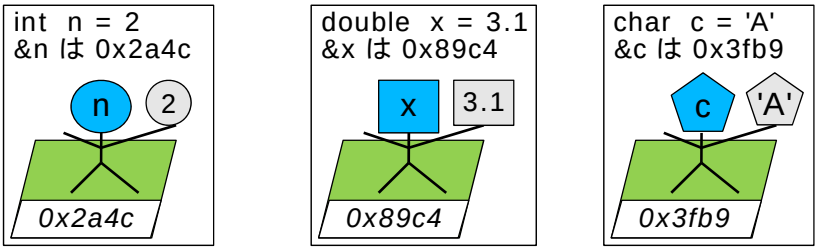
```
char  a;
char  b;
char  *p;

p=(&a);  /* pはaを指す。*/

b=(*p);  /*これは「b=a;」と同じ。*/

(*p)=b;  /*これは「a=b;」と同じ。*/
```

ポインタpがある変数yのアドレスを蓄えているとき、(*p)はあたかも変数yのように振舞う。

10

変数を住居に例えると...

- アドレスは住所
- 変数名は住人の名前
- 変数の値は住人の持ち物




```
n = 3;
```
は、「n さんに値 3 を渡して」

```
p = &n;
(*p) = 3;
```
は、「住所 (&n) の住人に値 3 を渡して」

11

練習2

```
/* test_pointer.c   ポインター実験  コメント省略     */
#include <stdio.h>

int main()
{
    /*変数宣言*/
    int  i;    /*整数が入る変数*/
    int  j;
    int  *p;  /*アドレスが入る変数(ポインタ)*/
    int  *q;

    /* 変数への値の代入 */
    i=1;
    j=2;

/* 次へ続く  */
```

12

```
/*続き*/
    /*実験操作*/
    p=(&i);  /* ポインタpへ変数iのアドレスを代入 */
    q=(&j);  /* ポインタqへ変数jのアドレスを代入 */

    printf("アドレス代入直後¥n");

    printf("iの中身は、%d¥n", i);
    printf("iのアドレスは、%p¥n", &i);
    printf("pの中身は、%p¥n", p);
    printf("pの指す変数の中身は、%d¥n", *p);
    printf("¥n");

    printf("jの中身は、%d¥n" ,j);
    printf("jのアドレスは、%p¥n", &j);
    printf("qの中身は、%p¥n", q);
    printf("qの指す変数の中身は、%d¥n", *q);
    printf("¥n¥n");

/* 次へ続く  */
```

13

```
/*続き*/

    /* ポインタを用いた、変数の値の操作 */
    (*q) = (*q)+(*p);
    printf("(*q)=(*q)+(*p); を実行¥n");
    printf("¥n");

    printf("iの中身は、%d¥n" ,i);
    printf("iのアドレスは、%p¥n", &i);
    printf("pの中身は、%p¥n", p);
    printf("pの指す変数の中身は、%d¥n", *p);
    printf("¥n");

    printf("jの中身は、%d¥n", j);
    printf("jのアドレスは、%p¥n" ,&j);
    printf("qの中身は、%p¥n", q);
    printf("qの指す変数の中身は、%d¥n", *q);

    return 0;
}
```

14

演算子＆と＊の結合力

演算子＆、＊の結合力は、算術演算子よりつよく、
インクリメント演算やデクリメント演算よりよわい。

++ -- ＞ &(アドレス演算子) ＊(間接演算子) ＞ / ＊(算術演算子) ＞ + −

```
*p++;
*p+1;
```

は

```
*(p++);
(*p)+1;
```

の意味

1つの式内でインクリメント演算子と間接演算子を使うときには、
括弧を用いて意図を明確にすること。

15

関数とポインタ

関数の仮引数や戻り値として、アドレスを用いることができる。

書式

```
戻り値の型  関数名(仮引数の型  ＊ 仮引数名)
{
    return  戻り値;
}
```

```
void  func_ref(int  *p)
{
   return;
}
```

アドレスを渡して関数を呼び出す方法。
この場合の仮引数はpで、intへのポインタ型。
(int *)型であって、int型ではないことに注意。

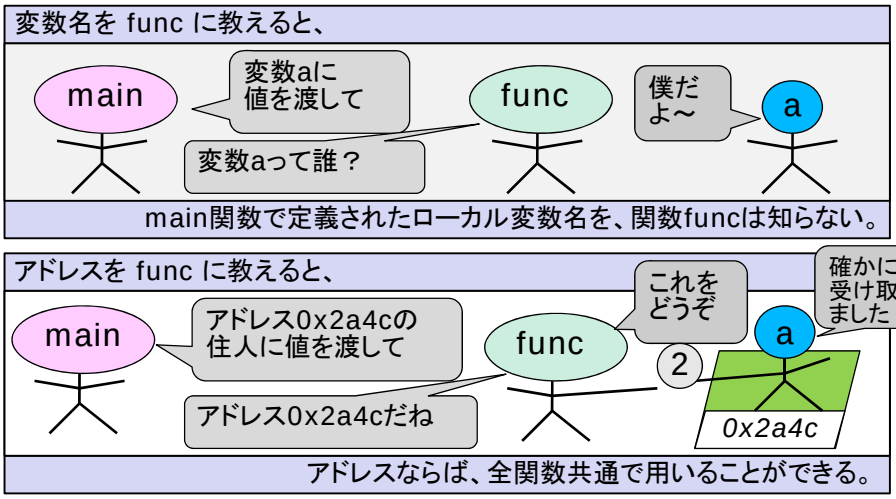
```
int main()
{
   func_ref(&i);
   a=i;
}
```

仮引数がポインタ型の場合、呼び出す際にはアドレスを指定しなければならない。

16

イメージ

main関数で定義された変数を、他の関数（func）で書き換えたい。



アドレスを受け取ることで、
他の関数内のローカル変数の内容を変更できる。

17

アドレスと scanf 文

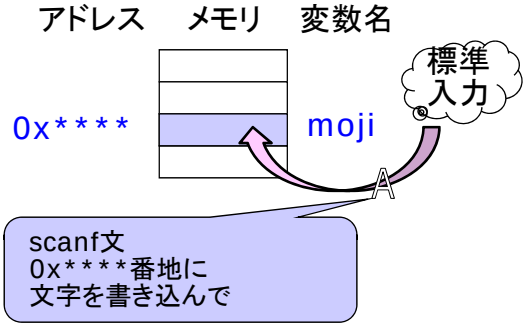
書式

```
scanf("%c", 文字を入れる変数のアドレス);
scanf("%d", 整数を入れる変数のアドレス);
scanf("%lf", 実数を入れる変数のアドレス);
```

例

```
char  moji;
scanf("%c",&moji)
```

&がついているので
アドレス

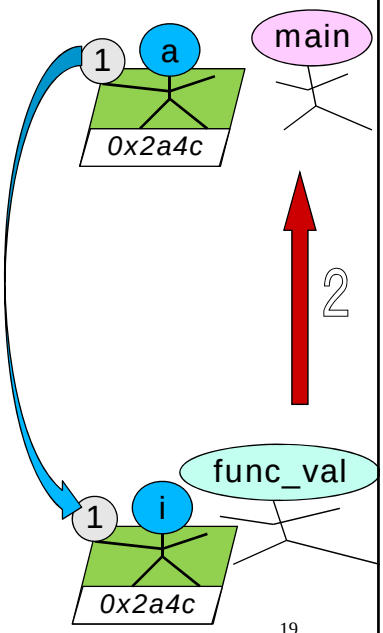


関数 scanf は、アドレスを渡されることで
変数の値を書き換えることができる。

18

値による呼び出し（call by value）

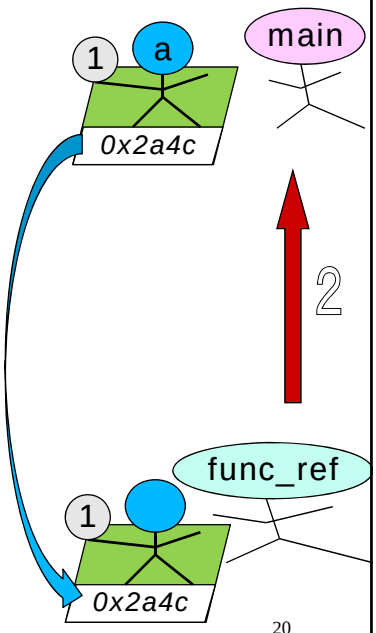
```
/*test_cbv.c 値による呼び出し実験*/
#include <stdio.h>
int  func_val(int);

int  main()
{
    int i=1;
    int j=0;

    printf("i=%d  j=%d¥n", i, j);

    j=func_val(i);

    printf("i=%d  j=%d¥n", i, j);
    return  0;
}

int  func_val(int   i)
{
    i++;
    return  i;
}
```



19

アドレスによる呼び出し（call by address）

```
/*test_cba.c  アドレスによる呼び出し実験*/
#include <stdio.h>
int  func_ref(int *p);

int  main()
{
    int i=1;
    int j=0;

    printf("i=%d  j=%d¥n", i, j);

    j=func_ref(&i);

    printf("i=%d  j=%d¥n", i ,j);
    return 0;
}

int  func_ref(int   *p)
{
    (*p)++;
    return (*p);
}
```



20

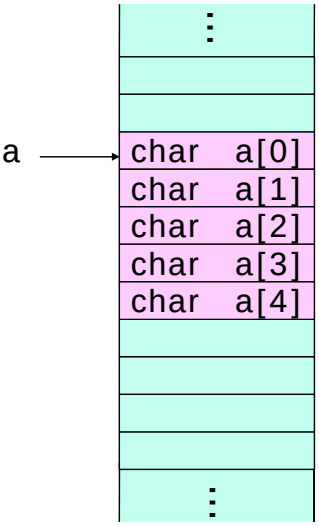
配列とアドレス

C言語では、配列名は先頭の要素のアドレスを指す。

例えば、

```
#define  MAX  5
char  a[MAX];
```

と宣言するとアドレスの連続した
5個のchar変数がメモリ上に確保され、
その先頭のアドレスがaにはいる。
つまり、aには
「&a[0]の値（アドレス）」
が保持されている。

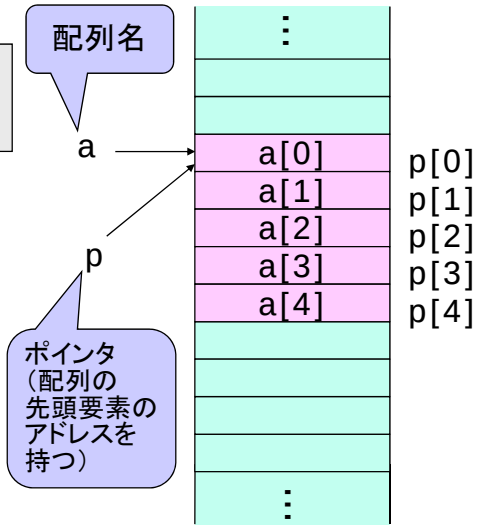


21

ポインタによる配列の別名

```
char a[MAX];
char *p;
p=a;
```

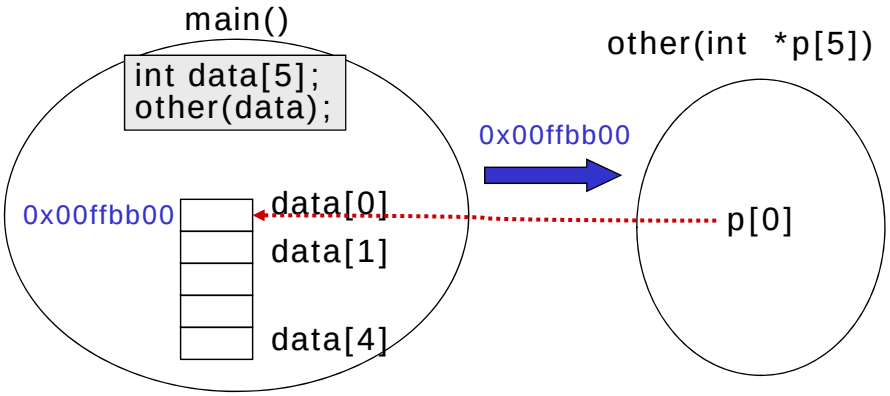
とすると、
a[i]とp[i]は
同じ変数（配列要素）を表す。



22

関数間での配列の引き渡し1

他の関数に配列(data[**])の
先頭要素のアドレス(data,すなわち、&data[0])を渡すことで、
配列全体を参照可能な状態にできる。

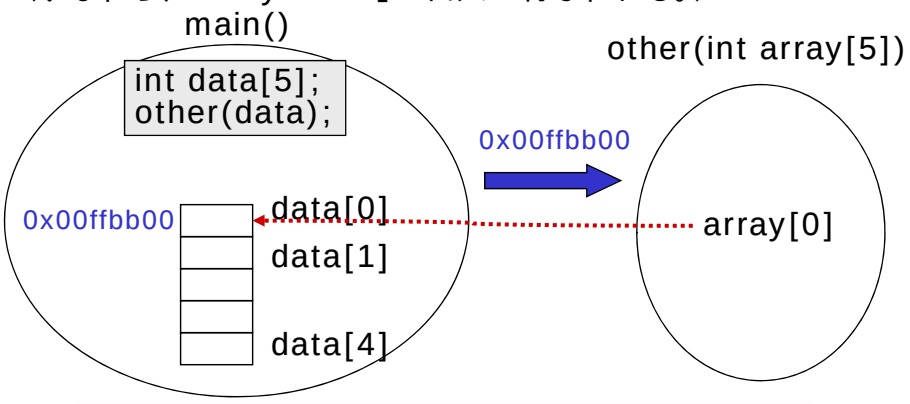


main 関数で定義された配列 data に、
関数 other では別名として p が与えられていると考えてもよい。

23

関数間での配列の引き渡し2

受け取る側では、仮引数に配列を記述しても良い。
この場合、引き渡されたアドレスが、引数の配列要素の先頭
アドレスになる。
(すなわち、「array=data」の代入が行なわれる。)

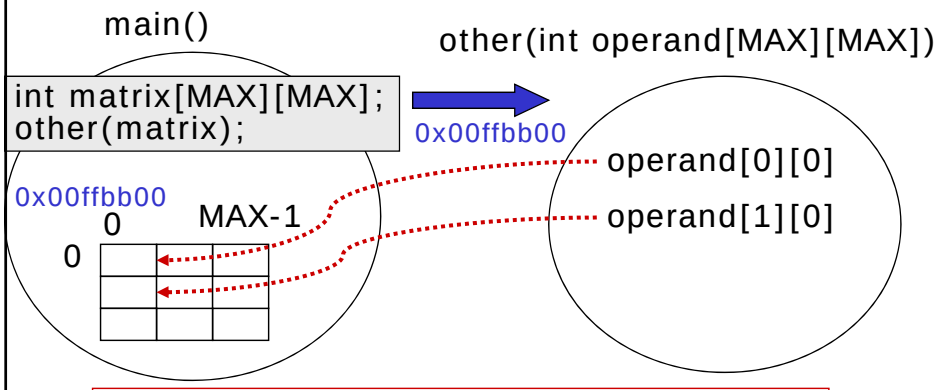


注意:
呼び出し側の配列の内容が書き換わるかもしれない。
十分に注意して関数を設計すること。

24

関数間での2次元配列の引き渡し

2次元配列では、先頭アドレスの他に大きさも指定する必要がある。



注意:
呼び出し側の配列の内容が書き換わるかもしれない。
十分に注意して関数を設計すること。

25

他の関数で定義された配列の和を計算する関数

```
/*
  作成日:yyyy/mm/dd
  作成者:本荘太郎
  学籍番号:b00b0xx
  ソースファイル: add_vector.c
  実行ファイル: add_vector
  説明:2つのN次元ベクトルの値を標準入力から受け取り、その和を出力する。
  入力:標準入力から N 個の実数値を2回受け取り、
       それぞれN次元ベクトル a, b の要素とする。
       aの要素が全て入力されてからbの要素が入力される。
  出力:標準出力に、N次元ベクトル a, b の和を出力する。
*/
#include <stdio.h>
#define  N  3   /* ベクトルの次元数 */

/* 標準入力からN次元ベクトルの値を取り込む関数  */
void  scan_vector(double  *p);
/* 標準出力にN次元ベクトルの値を出力する関数  */
void  print_vector(double  *p);
/* N次元ベクトルの和を計算する関数 */
void  add_vector(double  *p, double  *q, double  *r);

/* 次のページに続く */
```

26

```
/* 続き */
int  main()
{
   /* main関数のローカル変数宣言 */
   double  a[N]; /* 最初に入力されたベクトルの値 */
   double  b[N]; /* 次に入力されたベクトルの値 */
   double  c[N];  /* ベクトルaとベクトルbの和 */

   /* N次元ベクトル a, b の値を入力 */
   printf("%d次元ベクトル a の値を入力してください¥n", N);
   scan_vector(a);
   printf(" %d次元ベクトル b の値を入力してください¥n", N);
   scan_vector(b);

   /* N次元ベクトルの和を計算 */
   add_vector(a, b, c);

   /* 計算結果を出力 */
   printf("a+b=");
   print_vector(c);
   printf("¥n");
   return 0;
}
/* main 関数終了。次に続く。 */
```

27

```
/* 続き */

/* 標準入力からN次元ベクトルの値を入力する関数
  仮引数 p : N次元ベクトルの値を保持する配列の先頭アドレス
  戻り値なし
*/
void  scan_vector(double  *p)
{
   /* ローカル変数の宣言 */
   int  i; /* ループカウンタ、ベクトルを表す配列の添え字 */

   for(i=0; i<N; i++)
    {
      scanf("%lf", &p[i]);
    }
   return;
}

/* 次に続く */
```

28

```
/* 続き */

/* 標準出力にN次元ベクトルの値を出力する関数
  仮引数 p : N次元ベクトルの値を保持する配列の先頭アドレス
  戻り値:なし
*/
void  print_vector(double  *p)
{
   /* ローカル変数の宣言 */
   int  i; /* ループカウンタ、ベクトルを表す配列の添え字 */

   printf("(%6.2f", p[0]);
   for(i=1; i<N; i++)
    {
      printf(",%6.2f",p[i]);
    }
   printf(")");
   return;
}

/* 次に続く */
```

29

```
/* 続き */

/* N次元ベクトルの和を計算する関数
  仮引数 p, q : 被演算項を保持する配列の先頭アドレス
  仮引数 r : 演算結果のベクトルの値を保持する配列の先頭アドレス
  戻り値:なし
*/
void  add_vector(double  *p, double  *q, double  *r)
{
   /* ローカル変数の宣言 */
   int  i; /* ループカウンタ、ベクトルを表す配列の添え字 */

   for(i=0; i<N; i++)
    {
      r[i] = p[i] + q[i];
    }
   return;
}

/* プログラム終了 */
```

30