

数値解析

Numerical Analysis



配列
Array

電気・電子情報工学系 市川 周一
Shuichi Ichikawa, Dept. EEIE

本日の内容

Index

- C言語の(やや)進んだ話題 Advanced topics in C lang.
- 変数宣言 Variable declarations
 - 大域変数, 局所変数, 引数
- アドレスとポインタ Address and Pointer
 - 参照渡しと値渡し
- 配列 Array
 - 配列の定義, 利用
- 線形代数演算 Linear algebra
 - 配列を用いたプログラム例

C言語プログラムの構造

Structure of a C file

□ 変数宣言 Variable declaration

- データ型 変数名

変数が複数あってもよい

- 例: `int count;`

□ 関数宣言 Function declaration

- 関数の型 関数名(引数の型 引数名)

- 例: `double exp(double x);`

- 本体は別に定義

□ 関数 Functions

- 少なくとも`main()`は必要

- 各関数の構造は後ほど説明する

File: **xxx.c**

Variable declarations
Function declarations

Function

⋮

Function

大域変数と局所変数

Global variable, local variable

□ 大域変数 (Global variable)

- 関数の外で定義された変数
Variables defined out of functions
- 各関数から利用できる
Visible from each functions
 - 同じ名前の局所変数があれば, 局所変数が優先される
- 明示的に初期化しなければ, 値は0になる

□ 局所変数 (Local variable)

- 関数内で定義された変数
Variables defined in functions
- その関数内でだけ有効
Visible only in that function
 - 従って, 他の関数内の局所変数と同じ名前でも問題ない
- 明示的に初期化しない限り, 値は不定

大域変数の定義

ファイル xxx.c

```
double x;
```

局所変数の定義

```
int main() {  
    double y;
```

大域変数
xに代入

```
    x = 10.0;  
}
```

局所変数

```
void func() {  
    double x, y;
```

局所変数
xに代入

```
    x = 10.0;  
}
```

用途, 利害得失

Advantages, disadvantages

□ 大域変数 Global variable

- 多くの関数で使う変数, 実行環境など
 - 全てを引数で渡すと, 引数が多くなりすぎる
- 全ての関数から見える
 - どの関数で書き変わるか分からない → デバッグが難しくなりがち
 - 関数間の相互作用が増える → プログラムの複雑性が増す

□ 局所変数 Local variables

- その関数内でだけ使う変数
 - 他の関数から見えない → 他所から直接影響を受けない
 - 情報を隠蔽する(カプセル化) → ソフトウェアの構造化
- 再帰呼び出し (recursion) に必須
 - 非常に大切なことだが, 本日は省略

引数の種類

Parameter and Argument

□ 仮引数 parameter

- 関数定義の際に、値を受け取るために定義された引数
- 局所変数的一种
 - 仮引数を書き換えてもよい

□ 実引数 argument

- 関数を呼び出すときに、値を渡すための引数
- 渡すのは“値”
 - 式や定数を指定 → その値
 - 変数を指定 → その値

Return以外で値を返す方法はないのか？

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

```
int func(int a, int b)
{
    int v;

    v = a + b;
    a = 11;
    b = 22;
    return v;
}
```

仮引数

仮引数を書き換えてみる

```
int main()
{
```

```
    int x = 1;
    int y = 2;
    int z;
```

実引数

```
    z = func(x, y);
    printf("(x,y,z) = (%d,%d,%d)\n",
           x, y, z);
```

```
    return 0;
}
```

実引数x, yは書き変わらない

```
$ ./a.out
(x,y,z) = (1,2,3)
```

例： 大域変数, 局所変数

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

```
int n;
```

Global var.

```
void func()
```

```
{
    for ( ; n > 0; n--) {
        printf("func: n = %d\n", n);
    }
}
```

```
int main()
```

```
{
    n = 2;
    printf("main: n = %d\n", n);
    func();
    printf("main: n = %d\n", n);

    return 0;
}
```

```
$ ./a.out
```

```
main: n = 2
```

```
func: n = 2
```

```
func: n = 1
```

```
main: n = 0
```

値が変わった

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

```
void func(int n)
```

parameter

```
{
    for ( ; n > 0; n--) {
        printf("func: n = %d\n", n);
    }
}
```

```
int main()
```

```
{
```

```
    int n;
```

Local var.

```
    n = 2;
    printf("main: n = %d\n", n);
    func(n);
    printf("main: n = %d\n", n);
```

```
    return 0;
```

```
}
```

```
$ ./a.out
```

```
main: n = 2
```

```
func: n = 2
```

```
func: n = 1
```

```
main: n = 2
```

値は不変

復習：メモリ

Memory

- 順番に番地 (address) がついている
- アドレスを指定してデータ (数値) を読み書きする
- 通常, 各番地ごとに1バイト
 - 1バイト = 8ビット
- データの基本単位
 - バイト byte
 - ビット bit (2進1桁)

address data

0	00101100
1	
2	
:	
:	

変数のアドレス

The address of a variable

□ C言語: `&var`

- 変数`var`のアドレス(番地)
- 変数名はアドレスに対応

□ アドレスを変数で覚えることもできる

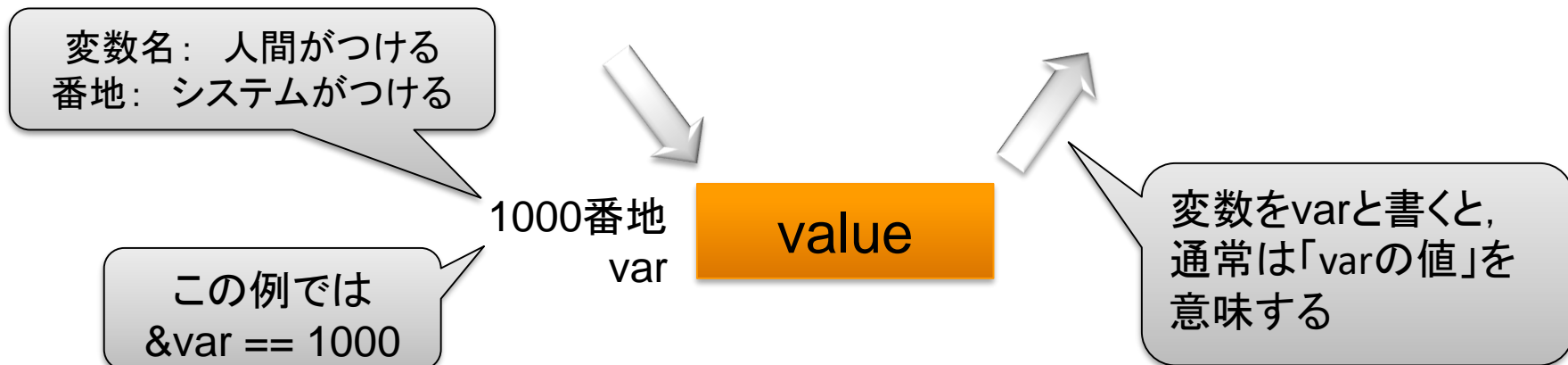
- そういう変数を“ポインタ”という

Pointer = a variable that stores an address

【復習】 `scanf`

```
scanf("%d", &a);
```

```
scanf("%d %d", &a, &b);
```



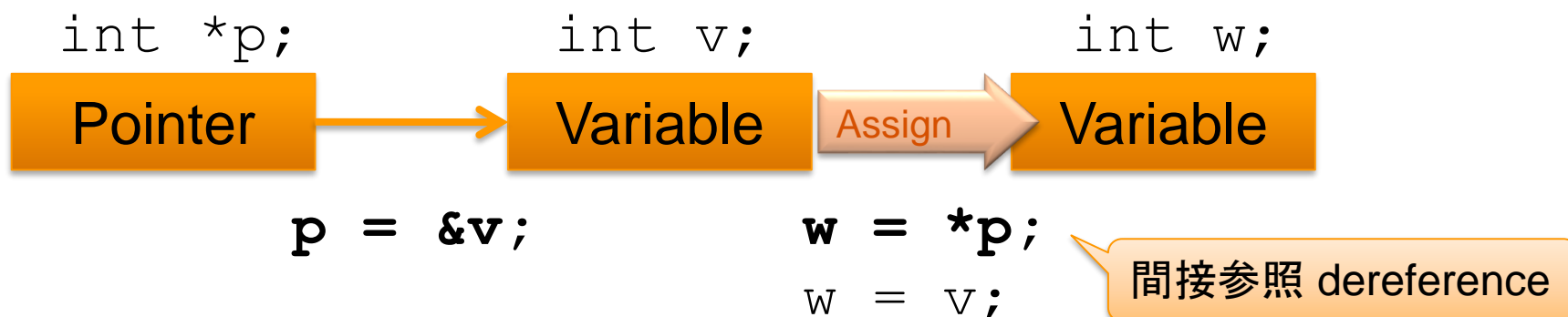
[Important] Address and Pointer

■ 変数なので、演算や代入をすることができる

■ ポインタの値は全て“アドレス”

- ただしポイントが指す変数には、それぞれ型がある

□ 例: int型変数を指すポインタ → (int *)型のポインタ



ポインタの使用例（ちょっと恣意的）

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

```
int main()  
{
```

ポインタ

```
    int *p;  
    int x = 1;  
    int y = 2;
```

```
    printf("x = %d¥n", x);  
    p = &y;  
    x = *p;  
    printf("x = %d¥n", x);  
    return 0;  
}
```

x = y; と同じ

```
$ ./a.out  
x = 1  
x = 2
```

変数yの値になった

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

```
int main()  
{
```

ポインタ

```
    int *p;  
    int x = 1;  
    int y = 2;
```

```
    printf("x = %d¥n", x);  
    p = &x;  
    *p = y;  
    printf("x = %d¥n", x);  
    return 0;  
}
```

x = y; と同じ

```
$ ./a.out  
x = 1  
x = 2
```

変数yの値になった

参照渡し(call by reference)と 値渡し(call by value)

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

ポインタ

```
void func(int *p)
{
    for ( ; *p > 0; *p = *p-1 ) {
        printf("func: n = %d\\n", *p);
    }
}
```

```
int main()
{
```

局所変数

```
    int n;

    n = 2;
    printf("main: n = %d\\n", n);
    func(&n);
    printf("main: n = %d\\n", n);

    return 0;
}
```

```
$ ./a.out
main: n = 2
func: n = 2
func: n = 1
main: n = 0
```

値を変えた！

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

仮引数

```
void func(int n)
{
    for ( ; n > 0; n-- ) {
        printf("func: n = %d\\n", n);
    }
}
```

```
int main()
{
```

局所変数

```
    int n;

    n = 2;
    printf("main: n = %d\\n", n);
    func(n);
    printf("main: n = %d\\n", n);

    return 0;
}
```

```
$ ./a.out
main: n = 2
func: n = 2
func: n = 1
main: n = 2
```

値が不変

復習：関数scanfの使用例

```
#include <stdio.h>


main()
{
    int a, b, c;

    printf("a > ");
    scanf("%d", &a);
    printf("a = %d\n", a);

    printf("a b > ");
    scanf("%d %d", &a, &b);
    printf("a = %d, b = %d\n", a, b);

    printf("a.b > ");
    scanf("%d.%d", &a, &b);
    printf("a = %d, b = %d\n", a, b);

    printf("a:b:c > ");
    scanf("%d:%d:%d", &a, &b, &c);
    printf("a = %d, b = %d, c = %d\n", a, b, c);
}
```



参照渡し
値を貰うため

The diagram shows an orange box with the text '参照渡し 値を貰うため' (Reference passing, to receive the value). Four orange arrows point from this box to the memory addresses (&a, &b, &b, &c) in the scanf calls within the C code.

実行例(太字部分を入力)

```
$ ./a.out
a > 10
a = 10
a b > 11 12
a = 11, b = 12
a.b > 12.34
a = 12, b = 34
a:b:c > 12:23:45
a = 12, b = 23, c = 45
```

本日の内容

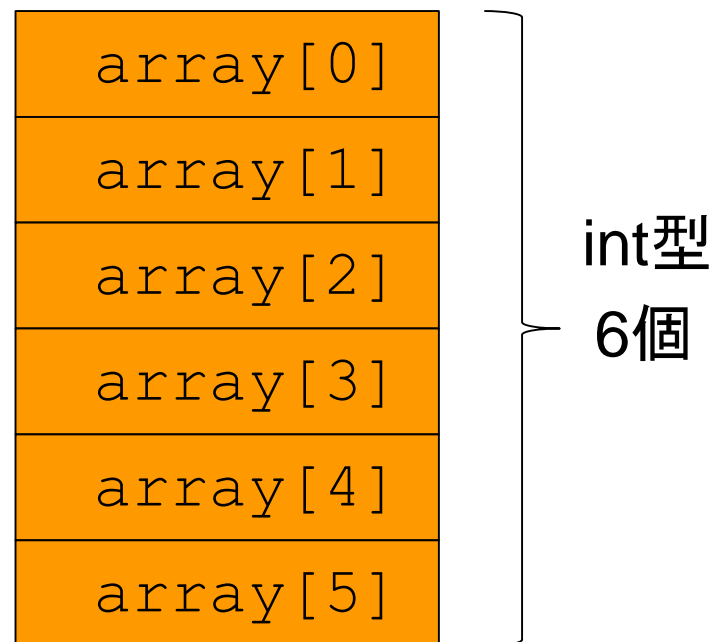
Index

- C言語の(やや)進んだ話題 Advanced topics in C lang.
- 変数宣言 Variable declarations
 - 大域変数, 局所変数, 引数
- アドレスとポインタ Address and Pointer
 - 参照渡しと値渡し
- 配列 Array
 - 配列の定義, 利用
- 線形代数演算 Linear algebra
 - 配列を用いたプログラム例

配列 Array

- 同じ型のデータを複数用いる場合のデータ構造
 - 右図では int 型
 - 連続した記憶場所に配置
- 記憶場所全体を“配列”という
 - 右図の `array[0] ~ [5]`
- 配列の名前を“配列名”
 - 右図の `array`
- 配列要素の番号を“添字”
 - `[]` 内の数字が添字

```
int array[6];
```



配列の定義

Definition on an array

□ 型 配列名[個数]

添え字は0～46

- 例: `int x[47];`

- 例: `float y[47];`

□ 他の変数と同時に定義しても良い

- 例: `int i, j, array[47];`

□ 初期値を与えることもできる

- 例: `int sample[3] = { 2, 1, 3 };`

- 結果: `sample[0]=2, sample[1]=1, sample[2]=3`

- 重要: 初期化しないと…

- 局所変数の場合 → 値は不定

- 大域変数の場合 → 値は0

- 配列でない変数と同じ

配列の入力・出力

Input / output an array

- 一度に全要素を入出力することはできない
 - 正確には, 「まだ教えていない」
- 繰り返し構文で, 要素を一つずつ入出力する

```
for (j = 0; j < todofuken; j++) {  
    printf("Input tcode and kion > ");  
    scanf("%d %f", &tcode[j], &kion[j]);  
}
```

```
for (j = 0; j < todofuken; j++) {  
    sa = kion[j] - heikin;  
    printf("%3d %.3f %.3f¥n", tcode[j], kion[j], sa);  
}
```

添字は変数でも
式でも良い

【重要】 配列で良くある間違い

[Important] Typical programming error

□ 範囲外を読みだすとどうなるか

```
#include <stdio.h>

int main()
{
    int a[10];
    int i;

    // writing a[0]--a[9]
    for (i = 0; i < 10; i++) a[i] = i*11;

    // OK ::: printing a[0]--a[9]
    // BUG::: printing a[10]--a[19]
    for (i = 0; i < 20; i++) {
        printf("a[%d] = %d\n", i, a[i]);
    }

    return 0;
}
```

コンパイルエラー
は出ない

存在しないa[10]~a[19]を出力

```
$ cc -Wall array-0.c
$ ./a.out
a[0] = 0
a[1] = 11
a[2] = 22
a[3] = 33
a[4] = 44
a[5] = 55
a[6] = 66
a[7] = 77
a[8] = 88
a[9] = 99
a[10] = 4195334
a[11] = 11
a[12] = -6928
a[13] = 32767
a[14] = 4195566
a[15] = 0
a[16] = 0
a[17] = 0
a[18] = 0
a[19] = 0
```

不定値を
返した

実行エラーを出
す可能性もある

【重要】 配列で良くある間違い

□ 範囲外に書き込むとどうなるか

```
#include <stdio.h>

int main()
{
    int a[10];
    int i;

    // OK ::: writing to a[0]--a[9]
    // BUG::: writing to a[10]--a[19]
    for (i = 0; i < 20; i++) a[i] = i;

    for (i = 0; i < 20; i++) {
        printf("a[%d] = %d\n", i, a[i]);
    }

    return 0;
}
```

コンパイルエラー
は出ない

存在しないa[10]~a[19]へ代入

```
$ cc -Wall array-1.c
$ ./a.out
a[0] = 0
a[1] = 1
a[2] = 2
a[3] = 3
a[4] = 4
a[5] = 5
a[6] = 6
a[7] = 7
a[8] = 8
a[9] = 9
a[10] = 10
a[11] = 11
a[12] = 12
a[13] = 13
a[14] = 14
a[15] = 15
a[16] = 16
a[17] = 17
a[18] = 18
a[19] = 19
Segmentation fault
```

実行エラー！

【重要】 配列で良くある間違い

```
#include <stdio.h>

int main()
{
    int x[10], a[10];
    int i;

    // OK :: writing to x[0]--x[9]
    for (i = 0; i < 10; i++) x[i] = -i;
    // OK :: checking x[0]--x[9]
    for (i = 0; i < 10; i++) {
        printf("x[%d] = %d\n", i, x[i]);
    }

    // BUG:: writing to a[10]~a[19]
    for (i = 0; i < 20; i++) a[i] = i;

    // checking x[0]--x[9]
    // x[] was not written, but modified
    for (i = 0; i < 10; i++) {
        printf("a[%d] = %d    x[%d] = %d\n", i, a[i], i, x[i]);
    }

    return 0;
}
```

← 配列xの書込

存在しないa[10]~a[19]へ代入

← 配列aの書込

```
$ cc -Wall array-2.c
```

```
$ ./a.out
```

```
x[0] = 0
```

```
x[1] = -1
```

```
x[2] = -2
```

```
x[3] = -3
```

```
x[4] = -4
```

```
x[5] = -5
```

```
x[6] = -6
```

```
x[7] = -7
```

```
x[8] = -8
```

```
x[9] = -9
```

```
a[0] = 0
```

```
a[1] = 1
```

```
a[2] = 2
```

```
a[3] = 3
```

```
a[4] = 4
```

```
a[5] = 5
```

```
a[6] = 6
```

```
a[7] = 7
```

```
a[8] = 8
```

```
a[9] = 9
```

← 元
の値

書き込んでいない
のに、
値が変わった

```
x[0] = 12
```

```
x[1] = 13
```

```
x[2] = 14
```

```
x[3] = 15
```

```
x[4] = 16
```

```
x[5] = 17
```

```
x[6] = 18
```

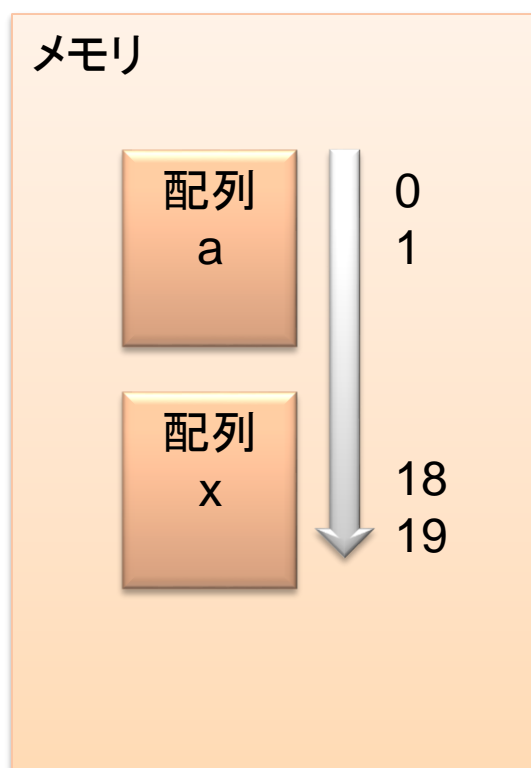
```
x[7] = 19
```

```
x[8] = -8
```

```
x[9] = -9
```

分析： 配列添字の誤り

- メモリ上には, 変数を含め各種の情報が存在する
 - 配列外へのアクセス
→ 他の情報へのアクセス
 - 配置はシステムが決定する
- 配列外の読み出し
 - 予測できない値
- 配列外への書き込み
 - データの破壊
 - プログラムの異常終了



マクロ展開

Macro expansion

- マクロ展開とは
 - 入力の中の“特定の文字列”を“別の文字列”に置き換えること
 - 例: `#define ABC abc`
- 前処理プログラム (cpp) が処理する
 - `#define`だけでなく, `#include`(ファイルの包含)もcppが行う
 - コメントもcppが除去する
- 使用目的
 - 同じ値を複数回使うとき, `define`してから使う (変更が1回で済む)
 - より理解しやすいプログラムになる (例: `x < 100` より `x < WIDTH`)



使用例: define

- 配列のサイズをdefineしてから使った
 - サイズを変えるときは, defineを変えて, 再コンパイルするだけ

```
#include <stdio.h>

int main()
{
    int a[10];
    int i;

    // OK ::: writing to a[0]--a[9]
    // BUG::: writing to a[10]--a[19]
    for (i = 0; i < 20; i++) a[i] = i;

    for (i = 0; i < 20; i++) {
        printf("a[%d] = %d¥n", i, a[i]);
    }

    return 0;
}
```

```
#include <stdio.h>
#define N 10

int main()
{
    int a[N];
    int i;

    // OK ::: writing to a[0]--a[9]
    for (i = 0; i < N; i++)
        a[i] = i;

    for (i = 0; i < N; i++) {
        printf("a[%d] = %d¥n", i, a[i]);
    }

    return 0;
}
```

プリプロセッサの仕事

- C プリプロセッサは、以下の4つの機能を提供します。(cppマニュアルより)
 - ヘッダファイルを読み込みます。これは プログラムに組み込まれる (C 言語の)宣言の入ったファイルです。
 - C 言語の任意の部分の省略形として マクロを定義し、C プリプロセッサがプログラム内の全てのマクロを その定義で置き換えます。
 - 条件文の処理をします。専用のプリプロセッサコマンドを用いて、いろいろな条件にしたがってプログラムの一部を含めたり除外したりできます。
 - 行番号の制御をします。ソースファイルと コンパイルされた中間ファイルとを組み合わせたり再アレンジしたりするプログラムを 用いる場合、コンパイラにオリジナルのソースの何行目であるかを知らせるための、行番号制御のプリプロセッサコマンドを利用できます。

```
% cpp
#define NAME Ichikawa

My name is NAME.
# 1 "<stdin>"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 160 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "<stdin>" 2

My name is Ichikawa.
%
```

```
% cpp
#include <stdio.h>
# 1 "<stdin>"
(長いので途中省略)
int fgetc(FILE *);
int fgetpos(FILE * restrict, fpos_t * restrict);
char *fgets(char * restrict, int, FILE * restrict);
FILE *fopen(const char * restrict, const char * restrict);
int fprintf(FILE * restrict, const char * restrict, ...);
int fputc(int, FILE *);
(長いので途中省略)
# 483 "/usr/include/stdio.h" 3 4
extern int __isthreaded;
# 2 "<stdin>" 2

%
```


線形代数の演算

Linear algebra

- 数値計算では非常に多くの線形代数演算が行われる
 - 行列やベクタ → 配列を用いて表現される
- C言語では、行列演算などの(直接の)サポートはない
 - データを配列で表現
 - 演算を繰り返し構文で実現
- 多くの場合、既製品の演算ライブラリを用いる
 - 例: LAPACK (Linear Algebra PACKage)
 - 線型計算のための数値解析ソフトウェアライブラリ
 - 線型方程式, 固有値, など
 - 例: BLAS (Basic Linear Algebra Subprograms)
 - ベクトルや行列に対する, 基本演算ライブラリ

行列サポートのある言語もある
例: FORTRAN90 → $C = A * B$

- 計算センターで提供
- メーカーが商品を販売

ベクトルと行列

Vector and Matrix

- n 次元ベクトル \rightarrow n 要素の一次元配列
- $m \times n$ 行列 \rightarrow $(m \times n)$ 要素の二次元配列

C言語で、どのように表現し、どのように扱うか？

$$\vec{x} = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix} \quad A = \begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{pmatrix}$$

例：ベクトルの加算

Vector addition

- ベクトルは、配列で表現
 - 3つの配列を変数として定義
- 関数vaddの引数として、配列のアドレスを渡す
 - 値を返してもらうため
 - 参照渡し
- 配列名＝最初の要素のアドレス

$$\vec{x} = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{pmatrix}, \vec{y} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{pmatrix}, \vec{z} = \begin{pmatrix} z_0 \\ z_1 \\ \vdots \\ z_{N-1} \end{pmatrix},$$
$$\vec{z} = \vec{x} + \vec{y}$$

```
#define N      3

double x[N], y[N], z[N];

// 途中省略

vadd(z, x, y);
```

同じ意味

```
vadd(&z[0], &x[0], &y[0]);
```

```
void vadd(double a[N], double b[N], double c[N])
{
    int i;

    for (i = 0; i < N; i++) a[i] = b[i] + c[i];
}
```

結果はこうなる

```
for (i = 0; i < N; i++) z[i] = x[i] + y[i];
```

例：ベクトルの加算

```
void vadd(double a[N], double b[N], double c[N])  
{  
    int i;  
  
    for (i = 0; i < N; i++) a[i] = b[i] + c[i];  
}
```

変数a, b, cはポインタ



```
void vadd(double *a, double *b, double *c)  
{  
    int i;  
  
    for (i = 0; i < N; i++) a[i] = b[i] + c[i];  
}
```

これでも全く同じように動く

2次元配列

2-dimensional array

□ 行列 matrix

- 2次元の配列
- 添字の順番に注意！
 - 特に正方行列でない場合

□ 初期化も可能

- 右図参照

```
double a[M][N];
```

 M行N列

$$A = \begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,N-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,N-1} \\ \vdots & \vdots & & \vdots \\ a_{M-1,0} & a_{M-1,1} & \cdots & a_{M-1,N-1} \end{pmatrix}$$

```
double a[2][3] = {  
    {1, 2, 3},  
    {4, 5, 6}  
};
```

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

例： 行列の加算

Matrix addition

```
double x[M][N], y[M][N], z[M][N];  
  
// 途中省略. 行列x, yの値は設定されているとする  
  
MatAdd(z, x, y);
```

```
void MatAdd(double a[M][N], double b[M][N], double c[M][N])  
{  
    int i, j;  
  
    for (i = 0; i < M; i++) {  
        for (j = 0; j < N; j++) {  
            a[i][j] = b[i][j] + c[i][j];  
        }  
    }  
}
```

補足1

- 2次元配列のメモリ上の構造
 - 1次元のメモリ上に、多次元の配列を実現している

2次元配列のメモリ上の表現

少し進んだ話題

```
#include <stdio.h>
#define N      3
```

```
int main()
{
```

```
    double x[N][N] = {
        { 1, 2, 3 },
        { 4, 5, 6 },
        { 7, 8, 9 }
    };
```

```
    int i, j;
    double *p = x;
```

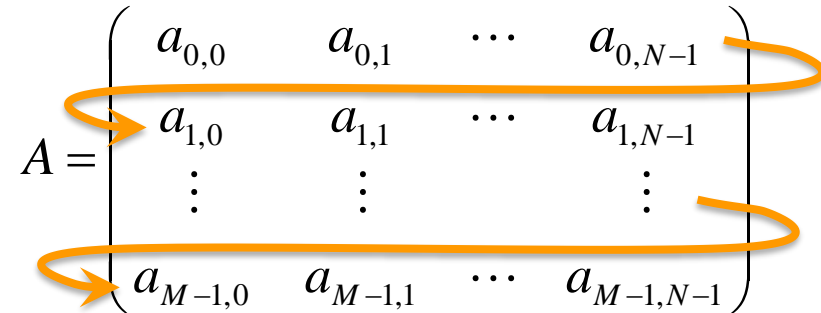
```
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++)
            printf("%.1f ", p[i*N+j]);
```

```
    }
    printf("\n");
    return 0;
}
```

ポインタpはx[0][0]
を指している

1次元配列

```
double a[M][N];
```



メモリ上の配置順序

左のプログラムで確認できる

$a_{0,0}, a_{0,1}, \dots, a_{0,N-1}, a_{1,0}, \dots, a_{1,N-1}, a_{N-1,0}, \dots, a_{N-1,N-1}$

0行目

1行目

... N-1行目

```
$ cc -Wall m.c
m.c: In function 'main':
m.c:12: warning: initialization from incompatible pointer type
$ ./a.out
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0
```

警告: 2次元配列を無理に1次元配列として使ったため
(この警告を消す方法はある)
(明示的型変換: 今回は省略)

2次元配列のメモリ上の表現

Representation of 2D array

少し進んだ話題

```
#include <stdio.h>
#define N      3

int main()
{
    double x[N][N] = {
        { 1, 2, 3 },
        { 4, 5, 6 },
        { 7, 8, 9 }
    };
    int i, j;
    double *p = x;

    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++)
            printf("%.1f ", p[i*N+j]);
        printf("\n");
    }
    return 0;
}
```

このように書くと
警告が出ない

```
#include <stdio.h>
#define N      3

int main()
{
    double x[N][N] = {
        { 1, 2, 3 },
        { 4, 5, 6 },
        { 7, 8, 9 }
    };
    int i, j;
    double *p = (void *)x;

    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++)
            printf("%.1f ", p[i*N+j]);
        printf("\n");
    }
    return 0;
}
```

```
$ cc -Wall m.c
m.c: In function 'main':
m.c:13: warning: initialization from incompatible pointer type
$ ./a.out
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0
```

```
$ cc -Wall mm.c
$ ./a.out
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0
```

補足2

- 数値シミュレーションなどで、乱数は(実用上)重要
- 数値解析の演習でも、ランダムな初期値が欲しい場合

乱数生成

□ シミュレーション等では乱数が必要になる

- 乱数の品質が悪いと、結果に影響を及ぼす

□ 乱数の種類

- 真性乱数：物理現象から生成される乱数列

- 再現性がない

- セキュリティ応用等に必須
- デバッグが難しい

- 乱数性を持ったハードウェアが必要

機種や環境に依存

- 疑似乱数：アルゴリズムで生成される数列

- 再現性がある

- ソフトウェア開発には有利

- ソフトウェアで生成できる

- 簡単に生成できる、大量に生成できる

色々なアルゴリズムがある。用途次第

疑似乱数生成

- シミュレーション等では必須の作業
 - 色々なライブラリ関数が存在する
 - 以下, Unix系OSの例を説明する
- 注意
 - 以下の関数は単純だが, 乱数品質は良くない
 - 本格的なシミュレーションでは, 別の関数を用いること
- `long lrand48()`
 - 0 と $2^{31} - 1$ の間で一様分布する 非負のロング整数を返す
- `double drand48()`
 - 区間 $[0.0, 1.0)$ で 一様分布する非負の倍精度浮動小数点実数値を返す

関数lrand48, drand48の例

確認せよ

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

```
int main()
```

```
{
```

```
    int i;
```

```
    for (i = 0; i < 10; i++) {
        printf("%ld¥n", lrand48());
```

```
    }
```

```
    return 0;
```

```
}
```

10個の乱数

Long型

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

```
int main()
```

```
{
```

```
    int i;
```

```
    for (i = 0; i < 10; i++) {
        printf("%e¥n", drand48());
```

```
    }
```

```
    return 0;
```

```
}
```

Double型

```
$ cc -Wall lrand48.c
```

```
$ ./a.out
```

```
851401618
```

```
1804928587
```

```
758783491
```

```
959030623
```

```
684387517
```

```
1903590565
```

```
33463914
```

```
1254324197
```

```
342241519
```

```
824023566
```

再現性がある。
→ a.outを何度実行し
ても同じ乱数列になる

数列自体は実行環境
によって異なる可能性
がある

```
$ cc -Wall drand48.c
```

```
$ ./a.out
```

```
3.964648e-01
```

```
8.404854e-01
```

```
3.533361e-01
```

```
4.465834e-01
```

```
3.186928e-01
```

```
8.864284e-01
```

```
1.558285e-02
```

```
5.840902e-01
```

```
1.593686e-01
```

```
3.837159e-01
```

$0 \leq \text{drand48()} < 1$
なる乱数列を返す

数列自体は実行環境
によって異なる可能性
がある

異なる乱数列を生成したい

確認せよ

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

```
int main()
{
```

```
    int i;
```

```
     srand48(1);
```

```
    for (i = 0; i < 10; i++) {
        printf("%ld¥n", lrand48());
    }
    return 0;
```

```
}
```

乱数の種 (seed)
から乱数列を生成

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

```
int main()
{
```

```
    int i;
```

```
     srand48(2);
```

```
    for (i = 0; i < 10; i++) {
        printf("%ld¥n", lrand48());
    }
    return 0;
```

```
}
```

種を変更

```
$ cc -Wall lrand48-3.c
```

```
$ ./a.out
```

```
89400484
976015093
1792756325
721524505
1214379247
3794415
402845420
2126940991
1611680321
786566648
```

```
$ cc -Wall lrand48-3.c
```

```
$ ./a.out
```

```
1959434203
341627945
1231072447
1721222818
1189008653
467581419
1466698862
1314478799
1346665927
2120537405
```

種を変えれば
乱数列も変わる

実行毎に異なる乱数列を生成したい

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    int i;

    // for your information
    printf("=== pid = %d ===\n", getpid());

    // seed the pid
    srand48(getpid());

    for (i = 0; i < 10; i++) {
        printf("%ld\n", lrand48());
    }
    return 0;
}
```

getpid()に必要な

プロセス番号は、プロセスごとに異なる
a.outを起動するごとに変わる

```
$ cc -Wall lrand48-4.c
$ ./a.out
=== pid = 12636 ===
1350338436
2050864414
202937104
410885407
631993926
1617204528
1090938575
1638988828
1081825680
2055405522
$ ./a.out
=== pid = 12637 ===
1072888506
1416477267
1788736875
1410583719
606623333
2080991533
7308370
826526637
816811287
1241892630
```

異なることを
確認せよ

乱数列の値の範囲を限定したい

□ 下のプログラムを理解し、実行して確認せよ

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

int main()
{
    int i;
    long int r;

    for (i = 0; i < 10; i++) {
        r = lrand48() % 201 - 100;
        printf("%ld¥n", r);
    }
    return 0;
}
```

```
$ cc -Wall lrand48-2.c
$ ./a.out
90
-57
-51
27
-96
66
-73
79
-75
47
```

$$0 \leq \text{lrand48}() \% 201 \leq 200$$
$$-100 \leq \text{lrand48}() \% 201 - 100 \leq 100$$

出現確率を厳密に同じにしたければ、
もっと丁寧なプログラムを組むこと

発展課題： 0～10に均等分布する乱数(整数)を
1000個生成し、それぞれの値が何回出現したか
数えて出力するプログラムを書け。
(条件: 配列を使うこと)

行列を乱数で初期化したい

内容を理解し、
動作を確認せよ

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

#define M      3
#define N      4

void RandomMatrix(double a[M][N])
{
    int i, j;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            a[i][j] = drand48();
}

void DumpMatrix(double x[M][N])
{
    int i, j;

    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j++) {
            printf("%e ", x[i][j]);
        }
        printf("\n");
    }
}
```

```
int main()
{
    double x[M][N];

    RandomMatrix(x);
    DumpMatrix(x);

    return 0;
}
```

```
$ cc -Wall RandomMatrix.c
$ ./a.out
3.964648e-01 8.404854e-01 3.533361e-01 4.465834e-01
3.186928e-01 8.864284e-01 1.558285e-02 5.840902e-01
1.593686e-01 3.837159e-01 6.910044e-01 5.885891e-02
$
```

区間[0, 1)の乱数.
値自体は実行環境で
異なるかも知れない