



これまでの復習と中間テストの講評

if ()

printf(“n と m は等しい¥n”);

else if ()

printf(“n は m より大きい¥n”);

printf(“n と m より小さい¥n”);

(1)	n==m
(2)	n>m
(3)	else



```
switch( n ) {  
    case 1: printf("caseA¥n");  
            break;  
    case 2:  
    case 3: printf("caseB¥n");  
            break;  
    case 4: printf("caseC¥n");  
            break;  
    default: printf("caseD¥n");  
            break;  
}
```

n	表示
1	caseA
2	caseB
3	caseB
4	caseC
6	caseD

int型の変数n, mとdouble型の変数a, bがあり
n=2, m=3, a=1.5, b=2.0と初期化されている

<code>printf("%d", m/n);</code>	1
<code>printf("%.3f", (double)(m/n));</code>	1.000
<code>printf("%.3f", (double)m/n);</code>	1.500
<code>printf("%.3f", a/n);</code>	0.750
<code>printf("%.3f", a/b);</code>	0.750

■ `%.3f` は小数点以下を3桁

■ `printf("%.3f", (double)(m/n) );`

と

`printf("%.3f", (double)m/n );`

の違いは,

`(double)(m/n)`は, `m/n`を`int`で計算した結果の1を`double`型にしているので, 1.000

`(double)m/n`は, `m`を`double`型にした3.0を`n(2)`で割っているなので, 結果も`double`で1.5



論理式の反転（否定）

$x == y$

$x != y$

$x > y$

$x <= y$

$A \&\& B$

Aの反転 **||** Bの反転

$A || B$

Aの反転 **&&** Bの反転

論理式の反転

	反転
$x > y$	$x < \textcolor{red}{=} y$
$(x \geq y) \&\& (y \neq z)$	$(x < y) \mid \mid (y == z)$
$(x < y) \mid \mid (a \geq b)$	$(x \geq y) \&\& (a < b)$



scanf

- intは%d
- doubleは%lf
- 変数には&をつける

キャスト

- (double)x
- キャストは関数とは違うのでかっこをつける部分に注意

外部変数

- 定義する場所
 - main やその他の関数の外
- 有効範囲
 - プログラム全体

自動変数

- 定義する場所
 - main やその他の関数の内
- static 修飾子がついていない
- 有効範囲
 - 関数の中
 - 関数が呼ばれるたびに生成され,
return するときに消える
 - main関数も他の関数も同じ

静的変数

- 定義する場所
 - main やその他の関数の内
- static 修飾子がついている
- 有効範囲
 - プログラム全体
 - プログラム開始時に生成され,
プログラム終了時に消滅する

シラバス

17,18. 多次元配列, 多重繰り返し

多次元配列とそれを扱うための多重繰り返し処理の実現方法の理解

事前学修: 授業スライド「第9回 多次元配列, 多重繰り返し」と教科書p.124~p.128の熟読。(60分)

事後学修: 授業スライドの用語とその意味の理解。多次元配列の利点と, C言語における2次元配列の宣言と初期化, 添え字によるデータの扱い方, 二重ループによる簡潔な計算方法を説明できること。(60分)

演習課題のプログラムの完成と提出。(180分)

プログラミングの基礎及び演習

日本大学 工学部 情報工学科



プログラミングの基礎 第9回

■ 配列(3)

□ 多次元配列

□ 多次元配列と多重ループ

□ 教科書 p.124～p.128

多次元配列

- 配列は「多次元」にできる
 - 「1次元」の配列だけではない

[0]	[1]	[2]	[3]	[4]
-----	-----	-----	-----	-----

1次元配列

「1次元配列」は「ベクトル」
「2次元配列」は「行列」
をイメージすると良い

[0][0]	[0][1]	[0][2]	[0][3]	[0][4]
[1][0]	[1][1]	[1][2]	[1][3]	[1][4]
[2][0]	[2][1]	[2][2]	[2][3]	[2][4]
[3][0]	[3][1]	[3][2]	[3][3]	[3][4]

2次元配列

注意： [2, 3]のようには書かない

3次元以上も可能

2次元配列

■ 添字を2つ使った配列

- 2つの添字で2次元を表わす
- 画像など,2次元のデータを扱うために用いる

j $\text{data}[i][j]$

$i \downarrow$	$[0][0]$ 1	$[0][1]$ 2	$[0][2]$ 3	$[0][3]$ 4
	$[1][0]$ 5	$[1][1]$ 6	$[1][2]$ 7	$[1][3]$ 8
	$[2][0]$ 9	$[2][1]$ 10	$[2][2]$ 11	$[2][3]$ 12

2次元配列の捉えかた

■ 1次元配列データの配列

- 下の2次元配列dataの場合,
4要素の1次元配列

{1, 2, 3, 4} , {5, 6, 7, 8} , {9, 10, 11, 12}
が並んだ配列

```
int data[3][4] = { {1, 2, 3, 4},  
                   {5, 6, 7, 8},  
                   {9, 10, 11, 12} };
```

2次元配列の宣言と初期化 (1)

■ 2次元配列を宣言した後に初期化する場合

2次元配列の宣言 :

```
int data[3][4];
```

2次元配列の初期化 (例) :

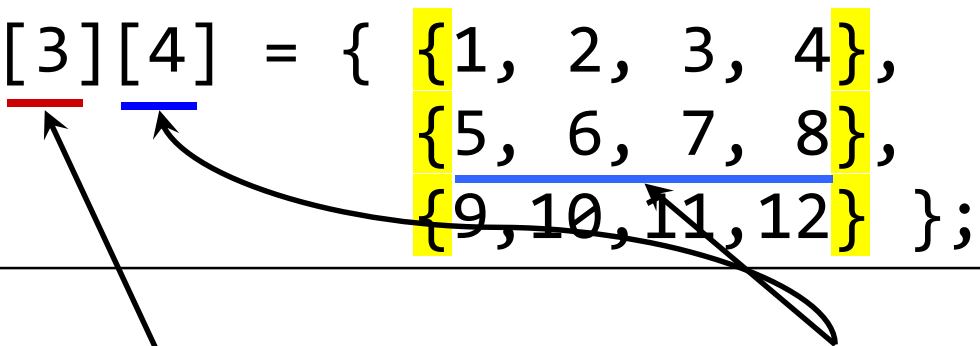
```
int i, j, cnt = 1;
for(i = 0; i < 3; i++) {
    for(j = 0; j < 4; j++) {
        data[i][j] = cnt;
        cnt++;
    }
}
```

		j data[i][j]			
i					
		[0][0]	[0][1]	[0][2]	[0][3]
		1	2	3	4
		[1][0]	[1][1]	[1][2]	[1][3]
		5	6	7	8
		[2][0]	[2][1]	[2][2]	[2][3]
		9	10	11	12

2次元配列の宣言と初期化 (2)

- 2次元配列の宣言と同時に初期化する場合
(初期化子を用いた場合)

```
int data[3][4] = { {1, 2, 3, 4},  
                  {5, 6, 7, 8},  
                  {9, 10, 11, 12} };
```



↑
↓
1つ目の要素数は
省略可能

2つ目の要素数は省略不可
初期化する各データは
その個数を書く必要がある

```
int data[][4] = { {1, 2, 3, 4},  
                 {5, 6, 7, 8},  
                 {9, 10, 11, 12} };
```



プログラミングの基礎 第9回

■ 配列(3)

- 多次元配列

- 多次元配列と多重ループ

- 教科書 p.124～p.128

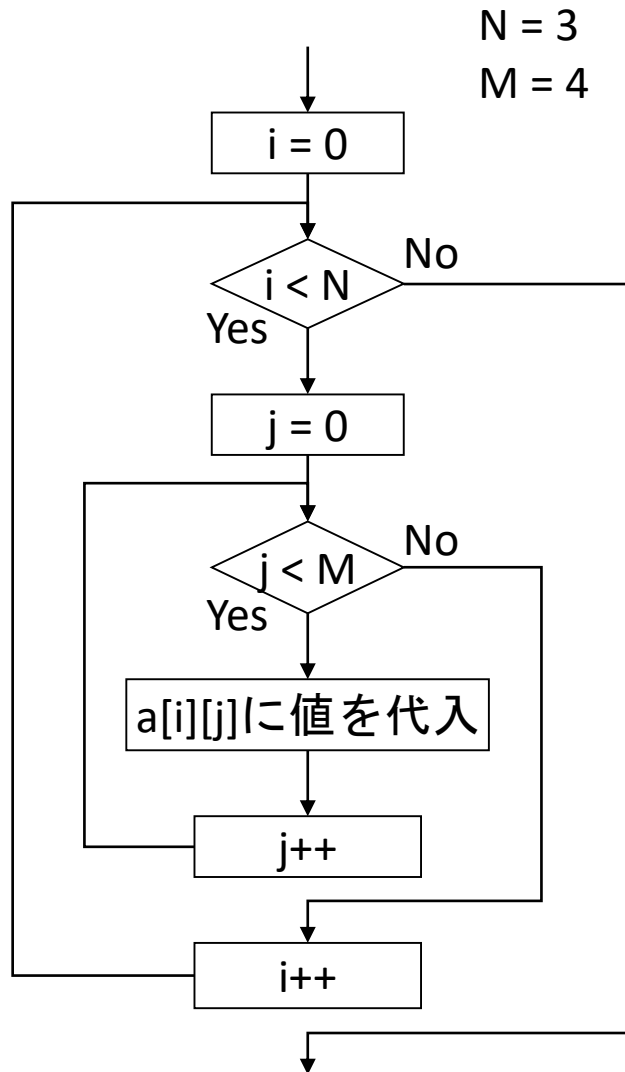
2次元配列と2重ループ

2次元配列のデータをキーボードから読み取るときなどは、下記のように2重ループを使う

```
for (i = 0; i < NOFLINE; i++) {  
    for (j = 0; j < NOFCOLUMN; j++) {  
        printf("a[%d][%d] = ", i, j);  
        scanf("%d", &a[i][j]);  
    }  
}
```



二重ループ (二重反復処理) (1)

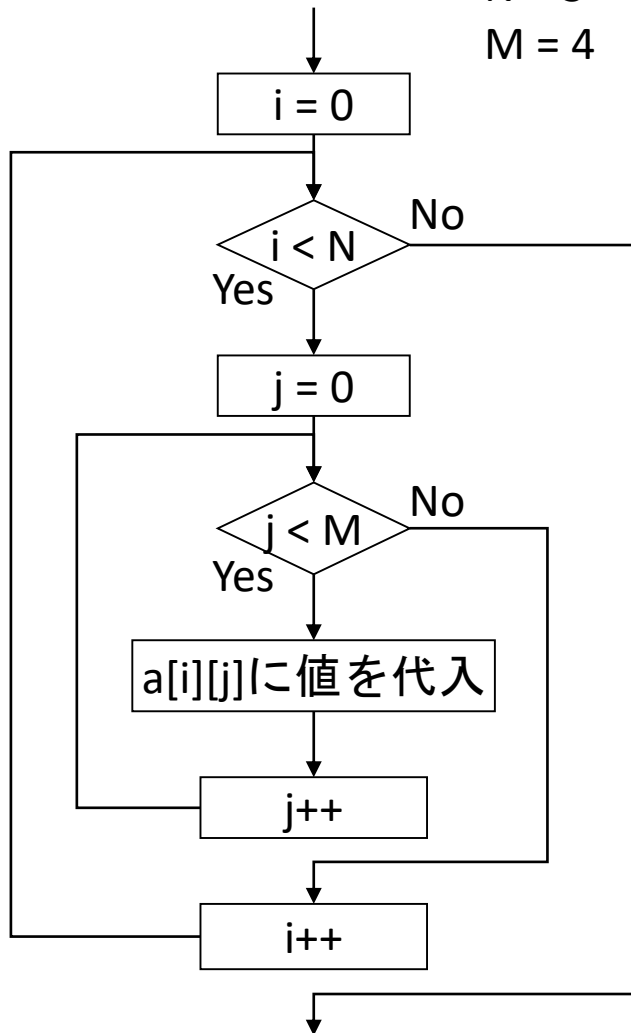


$i = 0, j = 0$
a[0][0]に値を代入
 $j++$
 $i = 0, j = 1$
a[0][1]に値を代入
 $j++$
 $i = 0, j = 2$
a[0][2]に値を代入
 $j++$
 $i = 0, j = 3$
a[0][3]に値を代入
 $j++$
 $i = 0, j = 4$
 $j < M$ を満たさない
 $i++$
 $i = 1, j = 0$
a[1][0]に値を代入
 $j++$
以下略...

$i = 2, j = 3$
a[2][3]に代入
 $j++$
 $i = 2, j = 4$
 $j < M$ を満たさない
 $i++$
 $i = 3$
 $i < N$ を満たさない
終了

二重ループ (二重反復処理) (2)

$N = 3$
 $M = 4$



```
for( i = 0; i < N; i++ ) {  
    for( j = 0; j < M; j++ ) {  
        a[i][j]に値を代入;  
    }  
}
```

三重以上のループも
同様に実現可能
(多重ループ)

二重ループの例

```
int keisan( int a[N][N] );  
  
int main(void) {  
    int i, j, x, a[N][N], sum;  
    for (i = 0; i < N; i++) {  
        for (j = 0; j < N; j++) {  
            printf("a[%d][%d] = ", i, j);  
            scanf("%d", &a[i][j]);  
        }  
    }  
    sum = keisan( a );  
    printf("sum = %d\n", sum);  
    return 0;  
}
```

例題9.4

二重ループ

プロトタイプ宣言

「配列全体」を受け取る

```
int keisan(int a[N][N]) {  
    int i, j, sum = 0;  
    for (i = 0; i < N; i++) {  
        for (j = 0; j < N; j++) {  
            sum += a[i][j];  
        }  
    }  
    return sum;  
}
```

関数keisan

・ ・ ・ 2次元配列の全要素
の総和を求める関数

「配列全体」を関数keisanに渡す
1次元配列と同様に、配列名だけを書く



3重ループ

3次元以上の
配列も可能

3重ループの例

```
int keisan3d(int a[N][N][N]) {  
    int i, j, k, sum = 0;  
    for (i = 0; i < N; i++) {  
        for (j = 0; j < N; j++) {  
            for (k = 0; k < N; k++) {  
                sum += a[i][j][k];  
            }  
        }  
    }  
    return sum;  
}
```



関数に多次元配列をそのまま渡す

```
#define N 3
int keisan2d( int b[][N] );

int main(void) {
    int i, sum;
    int a[N][N] = {{1, 2, 3},
                   {4, 5, 6},
                   {7, 8, 9}};

    sum = keisan2d( a );
    printf("sum = %d", sum);
    return 0;
}
```

プロトタイプ宣言

```
int keisan2d(int b[][N]) {
    int i, j, sum = 0;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            sum += b[i][j];
        }
    }
    return sum;
}
```

関数keisan2d

- ・ ・ ・ 2次元配列の全要素の総和を求める関数

配列をそのまま関数keisan2dに渡す



関数に多次元配列の一部を渡す

```
#define N 3
int keisan1d( int b[N] );  ← 例題9.5

int main(void) {
    int i, sum;
    int a[N][N] = {{1, 2, 3},
                   {4, 5, 6},
                   {7, 8, 9}};
    for (i = 0; i < N; i++) {
        sum = keisan1d( a[i] );
        printf("sum = ", sum);
    }
    return 0;
}
```

プロトタイプ宣言

1次元配列として受け取る

```
int keisan1d(int b[N]) {
    int j, sum = 0;
    for (j = 0; j < N; j++) {
        sum += b[j];
    }
    return sum;
}
```

1次元配列の総和を求める関数
keisan1dを2次元配列の各行
ごとの総和を求めるのに使用

N=3, i=1
のとき

「配列のi行目」を関数keisan1dに渡す
i番目の1次元配列a[i]を引数として与える

[0][0]	[0][1]	[0][2]
[1][0]	[1][1]	[1][2]
[2][0]	[2][1]	[2][2]

この行
を渡す



関数の仮引数の多次元のサイズの省略

- 省略できるのは第1要素のみ
- 関数には、関数を呼ぶ側の実引数の配列の先頭に関する情報が渡される
- 2次元配列で考えると、行の長さが第2要素のサイズ（列の本数と同じ）
これがわからないと、どこで行が変わるかが、関数側で判断できないから
- 第1要素のサイズは、引数の多次元配列の全要素数を知るためには必要だが、プログラムに**誤りがなければ**、全要素数の確認は重要ではない

【演習課題】

- 演習課題提出システムの「第9回」演習課題を実施
- 演習課題9-1, 9-2, 9-3, 9-4
 - 演習時間に，設計後に演習環境にリモートログインしプログラムを作成
 - 演習環境でコンパイル，テストし，完成
 - ソースファイル（9-x.c）をWinSCP等でPCへ転送
 - 課題提出システムへSubmitし各課題100点を目指す
 - 時間内に終了しない場合 ⇒ 宿題
- 授業翌週水曜日の午後5時までに必ず提出
- レポート，今回はなし