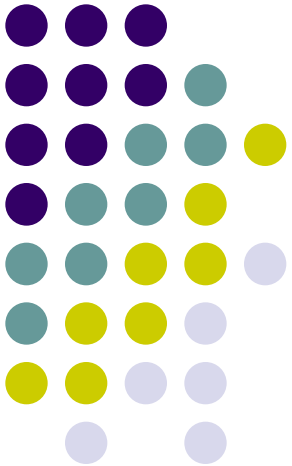
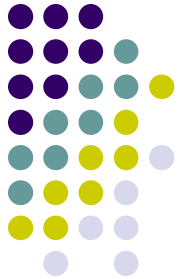
第二回補助スライド

C言語のおさらい
特にメモリ・ポインタ関連

遠藤敏夫
スライド協力：滝澤真一朗

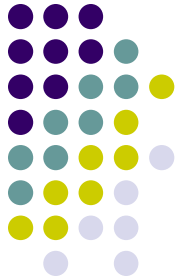


C言語



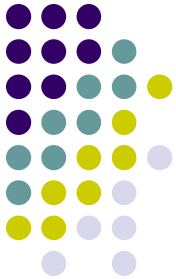
- 手続き型言語
  - 手続き（ルーチン、関数、Javaのメソッドに相当）の組み合わせでプログラムを実装
- ポインタを用いてメモリにアクセス可能
- 用途
  - システムプログラミング
  - デスクトップアプリケーション
  - Webアプリケーション　　など多岐にわたる

CとJavaの比較



- 類似点
  - 基本データ型の種類
    - char, short, int, long, float, double, void など
  - 演算子
    - =, ==, +, -, *, /, %, ++, --, &&, ！ など
  - 制御構文
    - for, while, switch, break, continue, return　など
- 相違点
  - boolean型が無い
    - ライブラリとしてはある
    - 条件判断には、falseは0、trueは0以外を使用
  - 例外処理が無い
    - 関数の戻り値などを細かくチェック
  - ポインタ型・演算子
    - *, &, ., ->

プログラム比較(1/2)



- 1から10までの和を表示するプログラム

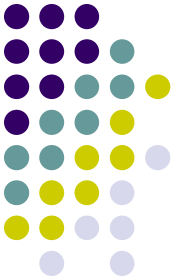
Java

```
class Sum {
  /* 足し算の和を計算 */
  public static void main(String[] args) {
    int sum = 0;
    for (int i = 1; i <= 10; i++) {
      sum = sum + i;
    }
    System.out.println("sum=" + sum);
  }
}
```

C言語

```
#include <stdio.h>

/* 足し算の和を計算 */
int main() {
  int i;
  int sum = 0;
  for (i = 1; i <= 10; i++) {
    sum = sum + i;
  }
  printf("sum=%d¥n", sum);
  return 0;
}
```

プログラム比較(2/2)



- ライブラリで提供されている関数を使う場合には「#include」でヘッダーファイルを読込む
  - printf関数はCの標準ライブラリで提供されている関数で、関数の情報(型情報)がヘッダーファイル stdio.h で宣言されている
  - Javaのimportに相当
- ローカル変数はブロックの先頭でしか宣言できない
  - ブロック:「{...}」で囲まれた部分
  - 「{...}」で囲ってブロックを意図的に作っても良い
- コメントは /* … */

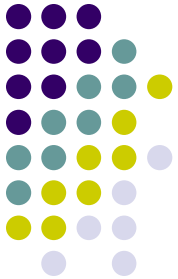
printf関数

- 文字列をフォーマットして出力する関数
  - int printf(“フォーマット文字列”, 変数1, 変数2, ...)

```
int i      = 100;
double d   = 0.01;
char c     = 'x';
char *str = "ABC";

printf("%d¥n", i);
printf("%f¥n", d);
printf("%c¥n", c);
printf("%s¥n", str);
printf("%d, %lf¥n", i, d);
```

%d	整数(int)
%ld	整数(long)
%f	浮動小数点数(float)
%lf	浮動小数点数(double)
%c	文字(char)
%s	文字列(char*)

配列



- 定義方法

```
int a[5];
int b[3]  = {1,2,3};
int c[]   = {1,2,3};
int d[10] = {1,2};
```

← サイズ5の配列を作成、初期化なし
← サイズ3の配列を作成、各要素を初期化
← 同上
← サイズ10の配列を作成
1番、2番要素のみ1, 2 で初期化、後は 0

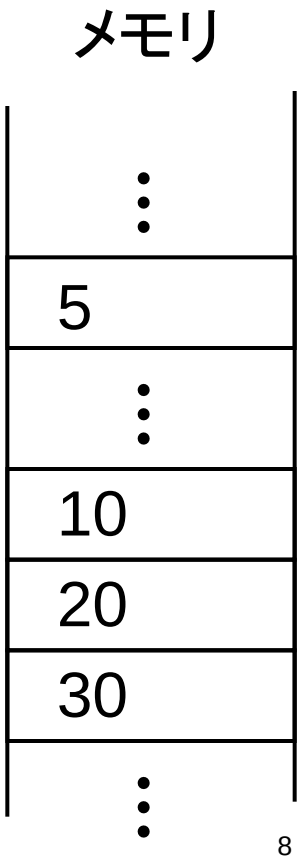
- アクセス方法
  - i 番目の要素は a[i]
  - 最初の要素の添字は 0
  - a[2] = 5 などで書き換え可能
  - a = a+1などはエラー

```
// d
int i;
for (i = 0; i < 10; i++) {
  printf("%d ", d[i]);
}
printf("¥n");
```

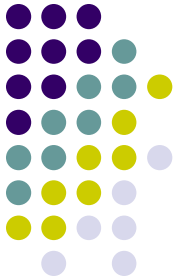
変数を宣言すると，計算機内では何が起こるか？

- int a = 5;
  ⇒ aは，整数(int)型の変数
- int b[3] = {10, 20, 30};
  ⇒ bは，整数(int)型のサイズ3の配列型の変数

- コンパイラは，各変数のためにメモリ領域を確保
  - 最適化の結果，メモリに置かれずレジスタのみに置かれるケースはあるが，今は考えない
- 配列型変数の場合は，b[0], b[1], b[2]は連続したメモリ領域に置かれる



C言語では，配列の長さを動的に決めることができない



- int a[100];　　OK
- int a[100][50];　OK(二次元配列)
- int a[b];　　NG
- int a[b][c];　　NG

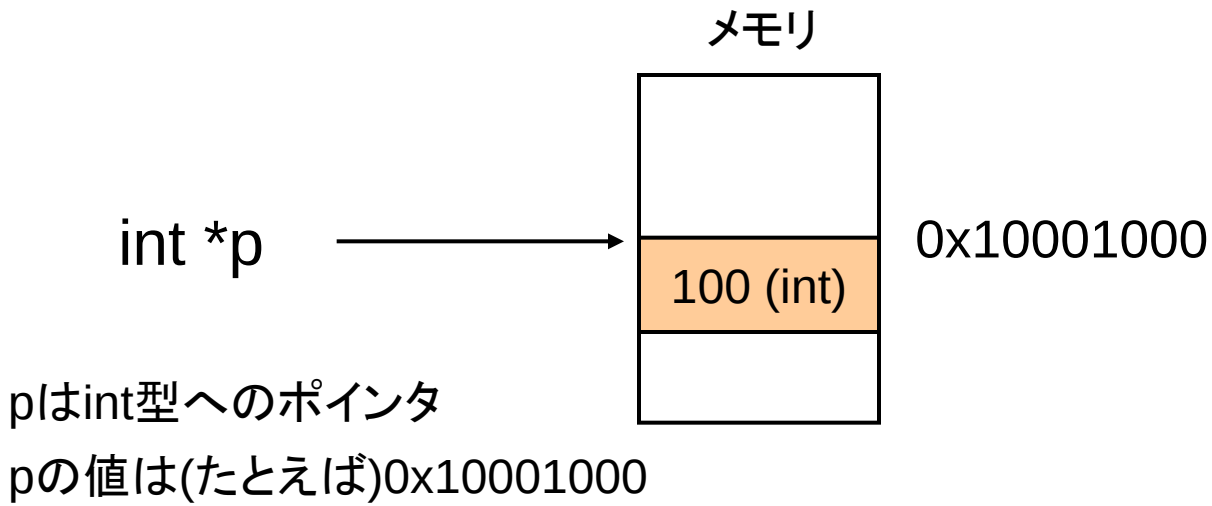
しかし，プログラム実行が始まってからデータサイズを決めたい場合も多い

⇒　ポインタとmalloc関数で同等のことは可能

多次元配列のときはtrickが必要(後述)

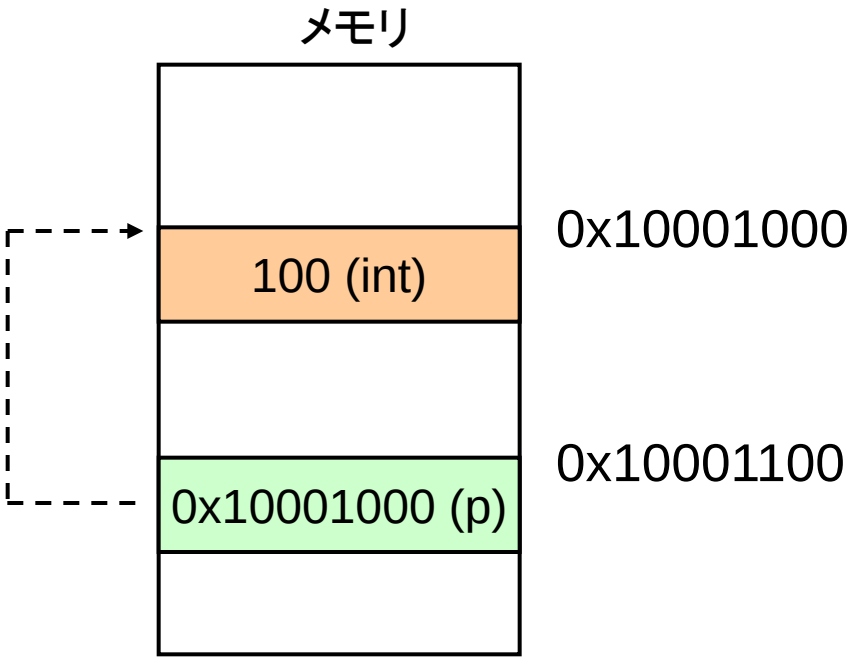
ポインタとは(1)

- メモリ上の変数のアドレスを保持する型
  - ポインタ型のサイズは8バイト（64 bit systemの場合）
  - Type型の変数のアドレスを保持するポインタのことを、「Type型へのポインタ」と言う
  - Javaにおける、オブジェクトへの「参照」



ポインタとは(2)

- ポインタ自身も変数なので、(多くの場合)メモリ上に置かれている



ポインタ演算

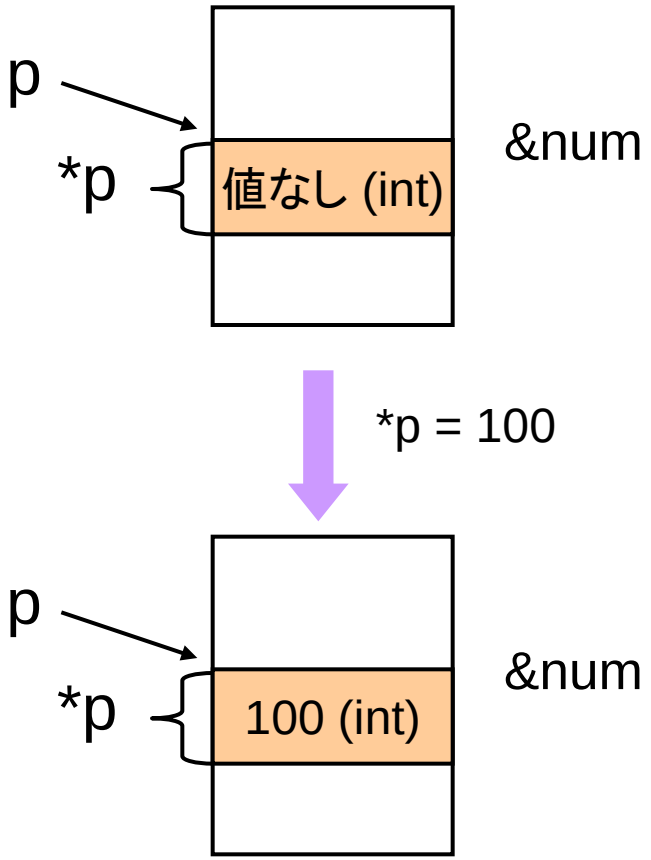
- ポインタ型の宣言
  - int *p;
    - int型へのポインタ p を宣言
- 変数のアドレスを取得
  - &num
    - アドレス演算子
    - numのアドレスを取得
    - int型へのポインタとして扱える
- ポインタ型への代入
  - p = &num
    - pはnumを指す
- ポインタが指すメモリの内容にアクセス
  - *p
    - 間接演算子

```
int num = 100;
int num2;
int *p;
p = &num;
num2 = *p;

/* 共に100を表示 */
printf("num  = %d¥n", num);
printf("num2 = %d¥n", num2);
```

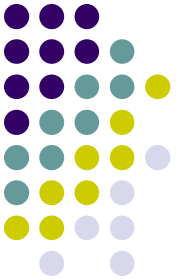
間接演算子による値の代入

- 間接演算子*を用いてポインタ経由で値の代入ができる

```
int num, *p;

p = &num;
*p = 100;
// 100を表示
printf("num  = %d¥n", num);
*p = 200;
// 200を表示
printf("num  = %d¥n", num);
// 200を表示
printf("*p   = %d¥n", *p);
```



NULLポインタ（空ポインタ）



- 「NULL」を代入されたポインタ

```
int *p = NULL;
```

  - 有用なデータを指していない
  - ポインタを返す関数で、戻り値が無いときにも使用
- NULLポインタへのアクセスは実行時エラー

```
int num = 100;
int *p = &num;

printf("%d¥n", *p);
p = NULL;
printf("%d¥n", *p);
```

←― 100を表示

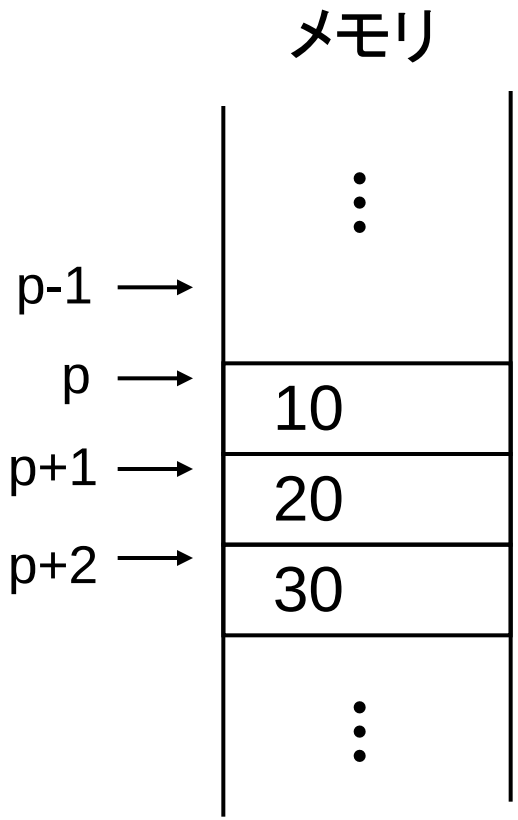
←― 実行時エラー

ポインタ演算(続き)

- ポインタと整数の加減算が可能
  - int *p
  - p+1　pのint1個分先のポインタ
    (1byte先ではない)
- *(p+1)で，「pのint1個分先のポインタのメモリ内容」を示す･･･配列に似ている？

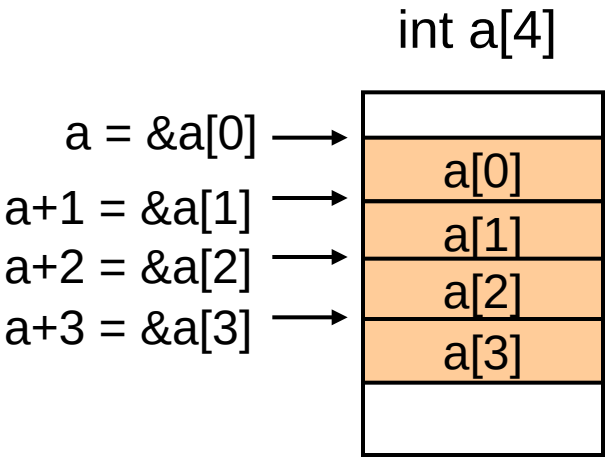
⇒ 配列 ≒ ポインタ

*(p+i) と，p[i] は同等



配列とポインタ

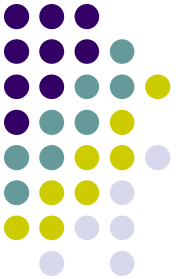
- (添字演算子[ ] を伴わない)配列名は、配列の先頭要素へのポインタとなる
  - ポインタ，配列とも，[ ]によるアクセス，加減算可能
- ポインタを用いた配列へのアクセス




```
int i;
int a[4] = {1,2,3,4};
int *p = a;
for (i = 0; i < 4; i++) {
  printf("%d ", *(p+i));
}
printf("¥n");
```

←　*(a+i)，a[i]，p[i]でも同等

配列とポインタの違い



- 配列は宣言と同時にメモリ領域が用意される
- ポインタは宣言後に，領域を何とかする必要
  - 他の変数の&を取る
  - malloc関数(後述)
- ポインタに対する代入は可能
- 配列(添字なし)に対する代入はエラー
  - 配列は「左辺値」ではない

```
int a[4] = {1,2,3,4};
int *p;
*(p+1) = 5;
p = a+1;
*(p+1) = 5;
a = a+1;
```

→ pはまだ無効なポインタなので違反

→ a[2]=5と同じ

→ 配列には代入できないのでエラー

配列を引数に取る関数

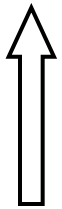
- 関数宣言

```
int func1(int a[], int size);
int func2(int *a, int size);
```

  - コンパイラでは同等に解釈される

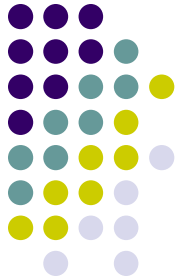
- 関数呼び出し

```
int a[] = {1,2,3,4};
func1(a, 4);
func2(a, 4);
```



配列名は先頭要素へのポインタ
(アドレス &a[0])になる
&a[0]の型は int *

ポインタを使った，動的長さの配列



- int a[n]; (これはエラー)を実現するには？
- 重要な関数：void *malloc(size_t size);

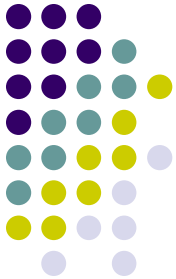
  ⇒sizeバイトのメモリをヒープ領域より確保し，そのポインタを返す．
- 領域が不要になったら，free関数で解放する．

固定長さの場合

```
int a[5];

…この間は，a[i]を
自由に使える…
```

```
int *a;
a = (int *)malloc(sizeof(int)*n);
if (a == NULL) {
  printf( “メモリ不足¥n” );
  exit(0);
}

…この間は，a[i]を自由に使える…

free(a);
```

多次元配列の場合



int a[m][n]; (これはエラー)を実現するには？

素直にはできないので，以下のいずれか

- ポインタのポインタにする

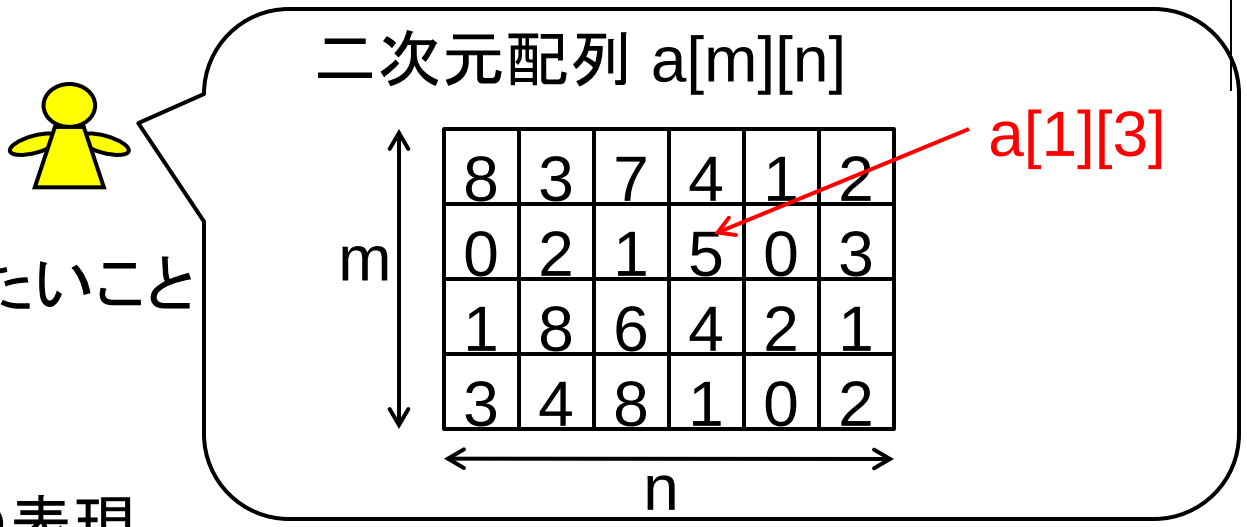
  各行の一次元配列をmallocし，それらへのポインタをまとめた動的配列をmallocする

- あきらめて，長さm×nの一次元配列にする

  (後述のサンプルプログラムではこちらを採用)

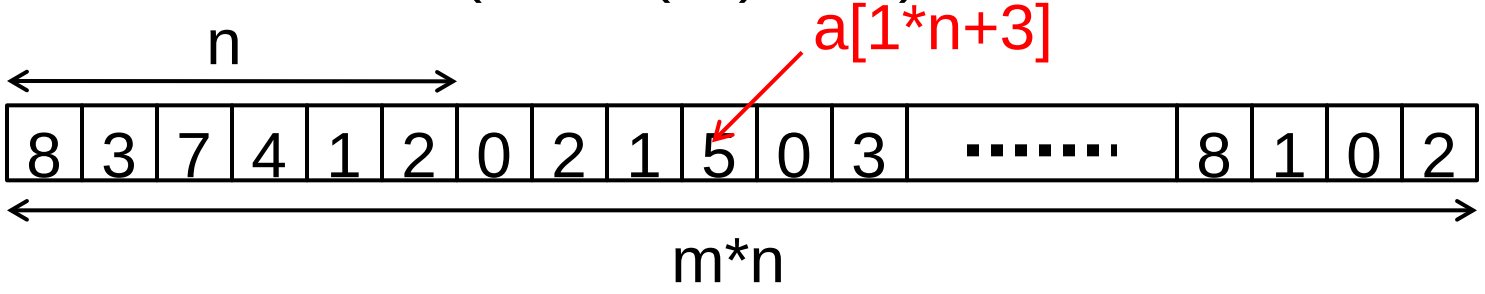
  a[i][j] の代わりに，a[i*n+j]とする

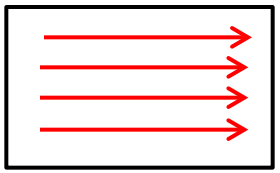
二次元配列の一次元配列による表現

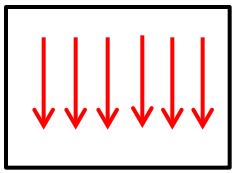


C言語での表現

```
int *a;   a = malloc(sizeof(int)*m*n);
```



Row major 

Column major 

Fortran, BLAS で利用