プログラミング第一同演習

慶應義塾大学 理工学部 情報工学科
講義担当: 河野 健二
演習担当: 杉浦 裕太

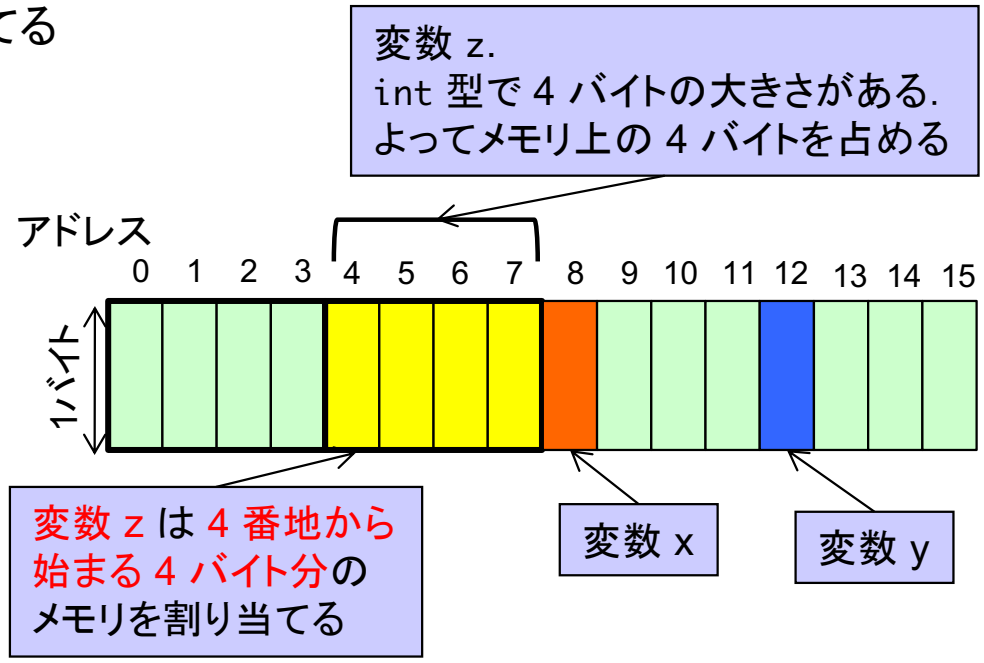
本日の内容

- ポインタと配列の関係
  - ポインタと配列の復習（ポインタ，忘れてないよね？）
  - 1 次元配列とポインタの関係
  - `sizeof` 再び
- 配列を引数とした関数呼び出し
  - call-by-reference のように見えるので注意
- 2 次元配列についても少しだけ
  - 慣れないと難しいので，徐々に慣れればよい
- 雑多な話題：いくつかの型，ビット演算とシフト演算

復習：変数とメモリの関係

- 変数には値を覚えておくためのメモリ領域が割り当てられる
  - char 型の変数は 1 バイト
  - int 型の変数は 4 バイト
    - int 型は 32 bit の整数．すなわち 4 (= 32 / 8) バイトとなる
    - 4 バイト分のメモリを割り当てる

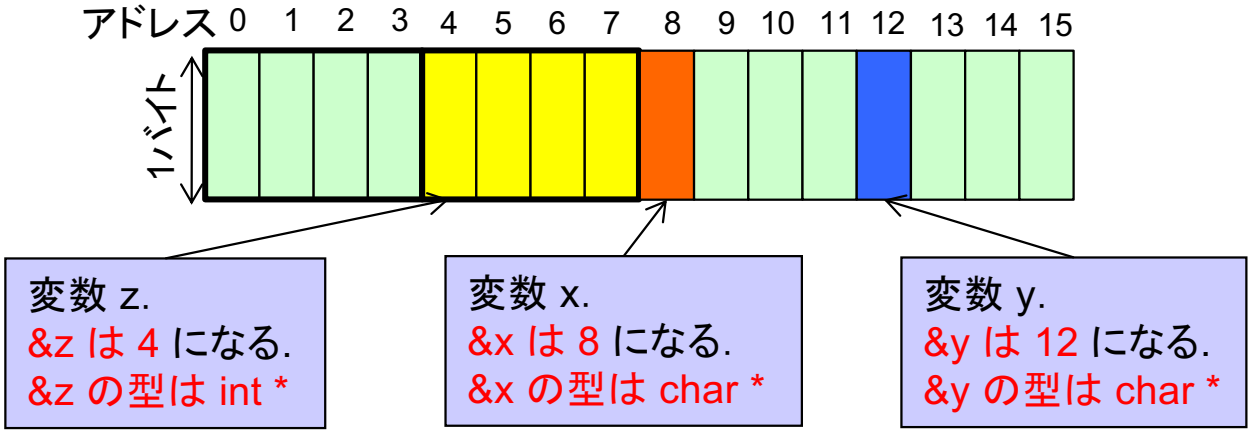
```
int main()
{
    char x;
    char y;
    int z;
    …
}
```



復習：ポインタとポインタ型

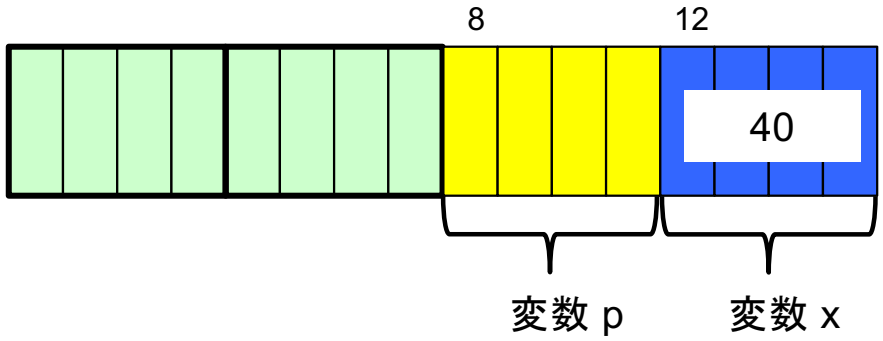
- ポインタとは変数が格納されているアドレスのことである
- C 言語では，変数が格納されているアドレスを取得できる
  - &x と書くと，変数 x が格納されているアドレスが得られる
    この例では，&x の値は 8 になる．&x の型は char * になる
  - &z と書くと，変数 z が格納されているアドレスが得られる
    この例では &z の値は 4 になる．&z の型は int * になる

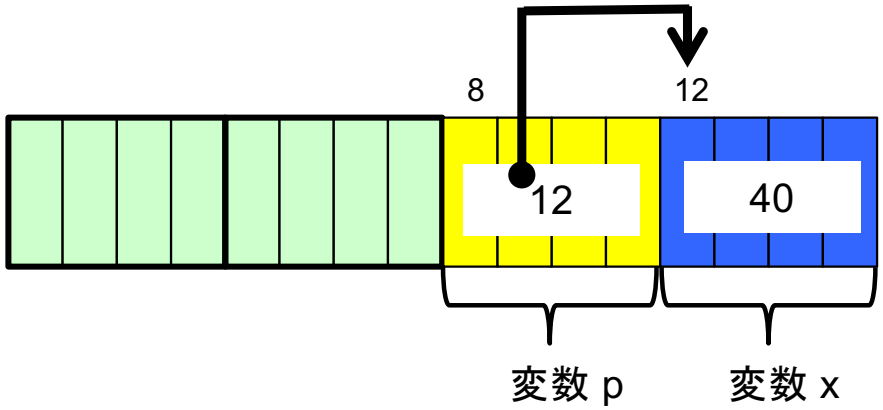
```
int main()
{
    char x;
    char y;
    int z;
    …
}
```

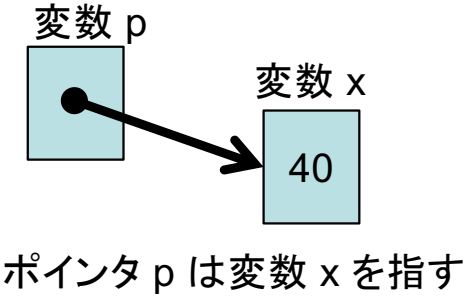


復習：ポインタを使ったプログラム例

- `int x = 40, *p;`
  - メモリのどこかに変数 x と 変数 p が割り当てられる
    - この例ではアドレス 12 番地に x, アドレス 8 番地に p.
  - 変数 x の値は 40 に初期化される
- `p = &x;`
  - 変数 p に変数 x のアドレスが入る
  - ポインタ p は変数 x を指すとイメージする



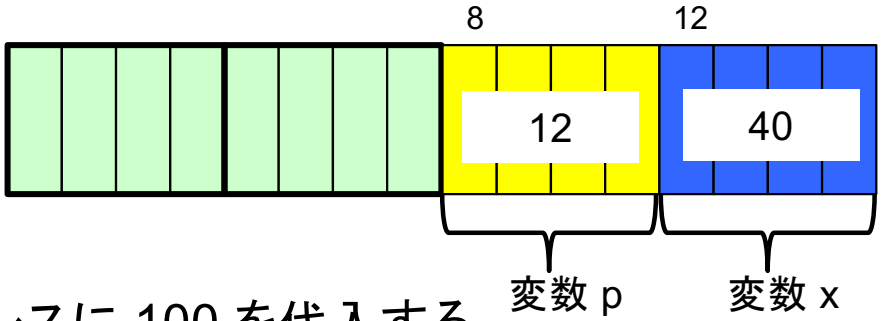


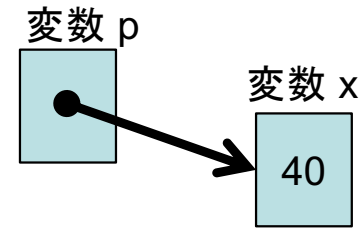


ポインタ p は変数 x を指す

復習：ポインタを使ったプログラム例

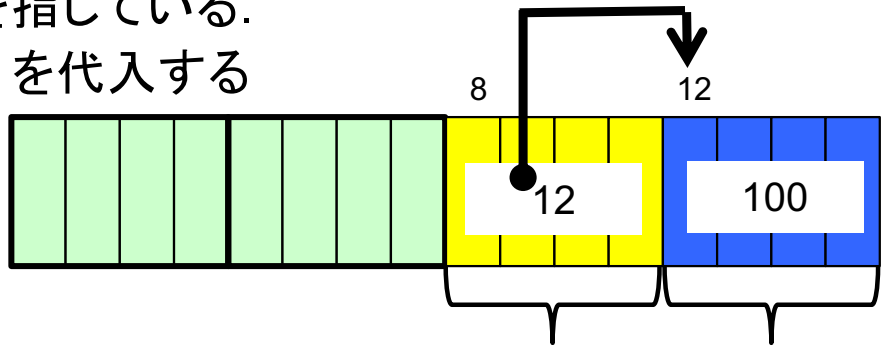
- printf("%d¥n", *p);
  - 変数 p に入っているアドレスに置かれている値を表示する
  - ポインタ p は変数 x を指している．よって，変数 x の値を表示する

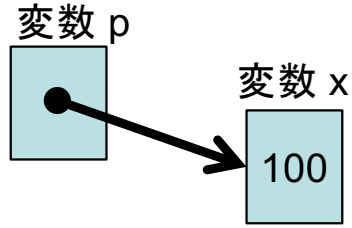




ポインタpは変数xを指す

- *p = 100;
  - 変数 p に入っているアドレスに 100 を代入する
  - ポインタ p は変数 x を指している．
  - よって，変数 x に 100 を代入する





ポインタ p は変数 x を指す

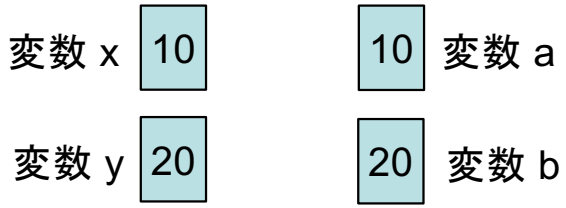
復習：変数の値の入れ替え

- 変数の値を入れ替えることをスワップ (swap) という

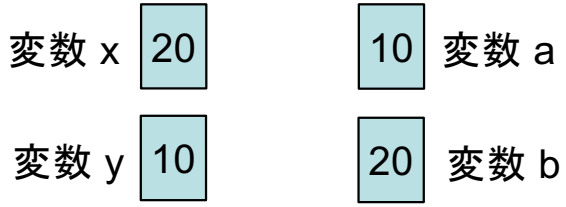
```
void swap(int x, int y)
{
    int t;
    t = x;
    x = y;
    y = t;
}

int main()
{
    int a = 10,  b = 20;
    swap(a, b);
    printf( “a = %d, b = %d¥n” , a, b);
    return 0;
}
```

swap() が呼び出された時点では，次のようになっている
変数 x, y には変数 a, b のコピーが入る



swap() の中で変数 x, y の値を入れ替えても
変数 a, b の値は変わらない



C は call-by-value (値呼び出し)だったことを思いだそう

復習：変数の値の入れ替え

- swap() には変数 a, b へのアドレスを渡すようにする

```
void swap(int *xp, int *yp)
{
    int t;
    t = *xp;
    *xp = *yp;
    *yp = t;
}

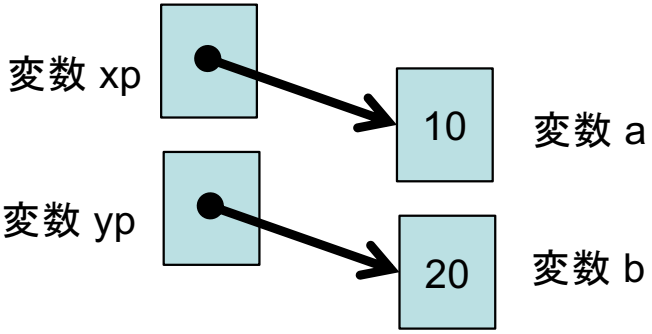
int main()
{
    int a = 10,  b = 20;
    swap(&a, &b);
    printf("a = %d, b = %d¥n", a, b);
    return 0;
}
```

変数 a のアドレス，
変数 b のアドレスを
受け取るようにする．

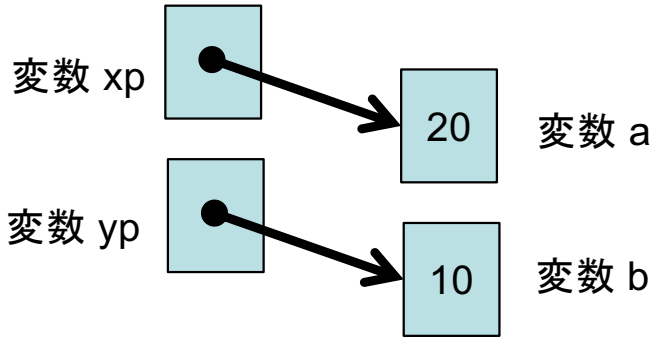
変数 a のアドレス，
変数 b のアドレスを
swap() に渡す

a = 20, b = 10
と表示される．

swap() が呼び出された時点では，次のようになっている



swap() 終了時には，次のようになる



Quiz: プログラムの実行結果は？

```
int main()
{
    int x = 40, *p;
    p = &x;
    x = 100;
    printf("%d", *p);
    return 0;
}
```

```
int main()
{
    int x = 40, *p, *q;
    p = &x;
    q = p;
    *p = 200;
    printf("%d", *q);
    return 0;
}
```

```
void foo(int *p)
{
    *p = 54;
}

int main()
{
    int x = 40;
    foo(&x);
    printf("%d", x);
    return 0;
}
```

Quiz: プログラムの実行結果は？

```
void bar(int *p)
{
    int *q = p;
    *q = 54;
}

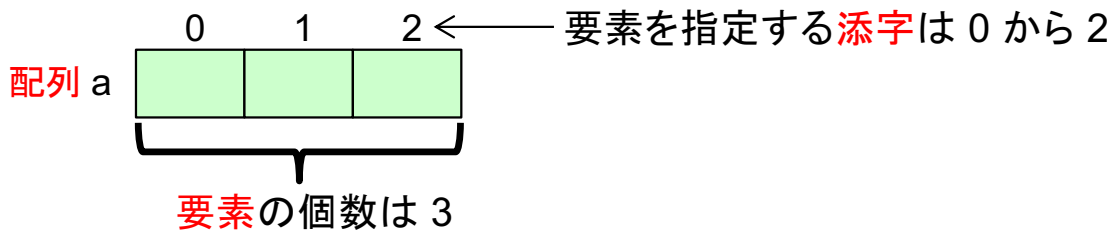
int main()
{
    int x = 40;
    bar(&x);
    printf("%d", x);
    return 0;
}
```

```
void f(int *p, int *q)
{
    *p = 10;
    *q = 20;
}

int main()
{
    int x = 40;
    f(&x, &x);
    printf("%d", x);
    return 0;
}
```

復習: 配列 (1)

- 同じ型の変数をまとめて定義，利用する仕組み
- 例: int 型の変数を 3 つまとめて宣言する
  - `int a[3];`
    “int 型の配列 a, 要素の数は 3 つ” を宣言



  - 数学の数列 $\{a_i\}$ $(0 \leqq i \leqq 2)$ にちょっと似ている

復習：配列 (2)

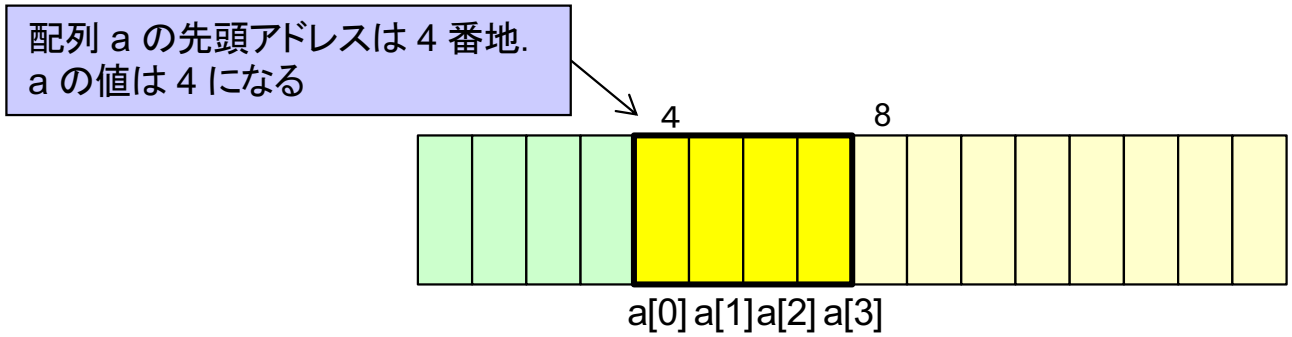
- `int a[3];`
  - `a[0]`, `a[1]`, `a[2]` という 3 つの変数を宣言したような感じ
    - `a[0] = 1;`
      `a[1] = 3;`
      `a[2] = 2;`
      } 配列のそれぞれの要素は `int` 型の変数となる

```
for (i = 0; i < 3; i++) {
        printf("a[%d] = %d¥n", i, a[i]);
}
```
← for 文ですべての要素を表示

- Python のリストのようなもの？
  - まったく違う！

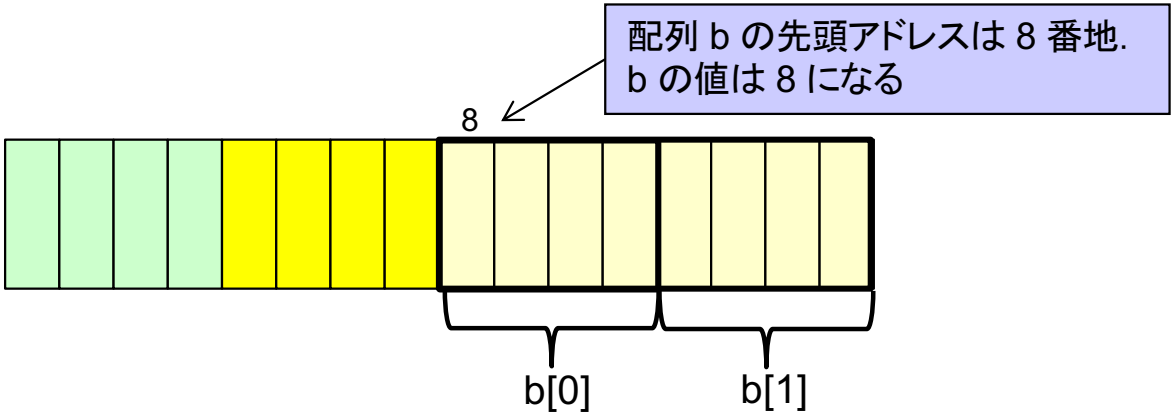
配列とメモリの関係 (1)

- 配列もメモリに領域がとられる
- char a[4];
  - 要素数 4 の char 型の配列．当然，メモリに場所がとられる
    - char 型は 1 バイト．よって a[4] は 4 バイトの大きさになる
  - a の値は配列の先頭アドレスになる
    - この例では a の値は 4 になる



配列とメモリの関係 (2)

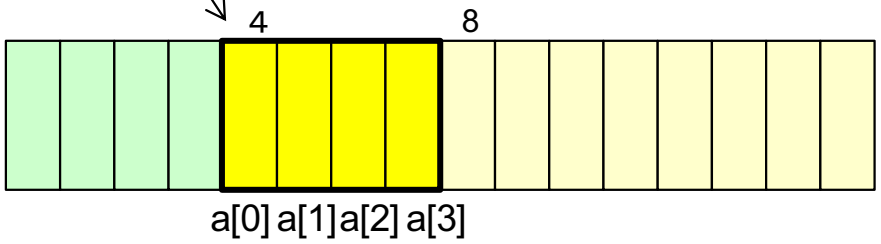
- 配列もメモリに領域がとられる
- `int b[2];`
  - 要素数 2 の `int` 型の配列
    - `int` 型は 4 バイト. よって `b[2]` は 8 バイトの大きさになる
  - b の値は配列の先頭アドレスになる
    - この例では b の値は 8 になる



配列とポインタの関係 (1/6)

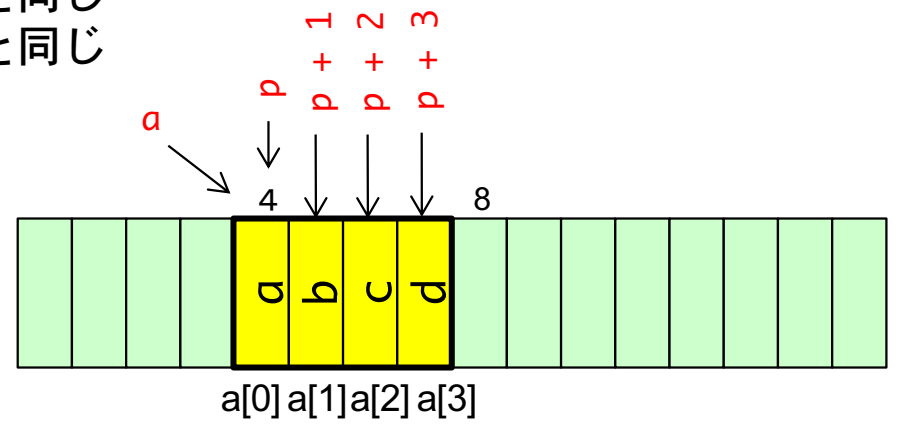
- char a[4];
- a の値は配列 a の先頭アドレス．したがって，ポインタとして扱える
  - char a[4];
    char *p;　　// a は char 型の変数 a[0] を指すアドレス
    p = a;　　// したがって，a は char * 型の変数に代入できる

配列 a の先頭アドレスは 4 番地．
char * 型の p の値も 4 になる



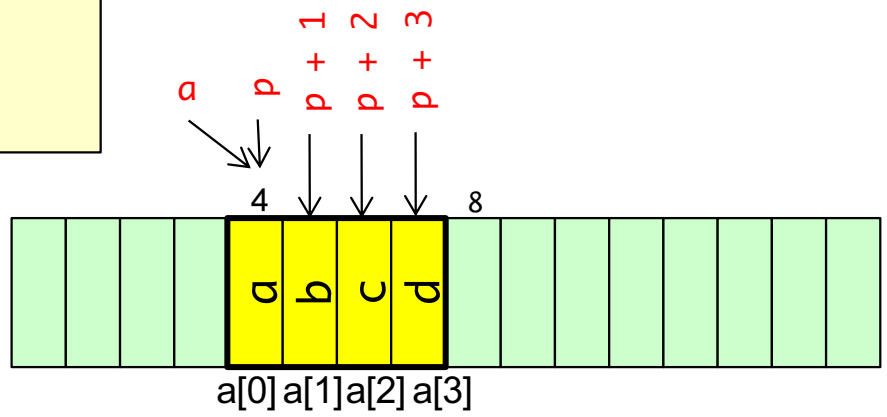
配列とポインタの関係 (2/6)

- ポインタを使って配列の各要素にアクセスできる
- ポインタに対して加算・減算ができる
  - ポインタの中身はアドレス
  - そのアドレスを増やしたり，減らしたりできる
    - ```
      char *p, a[] = {'a', 'b', 'c', 'd'};
      p = a;                  // p は a[0] を指す
      *p = 'A';               // a[0] = 'A'; と同じ
      *(p + 1) = 'B';         // a[1] = 'B'; と同じ
      ```


      ```

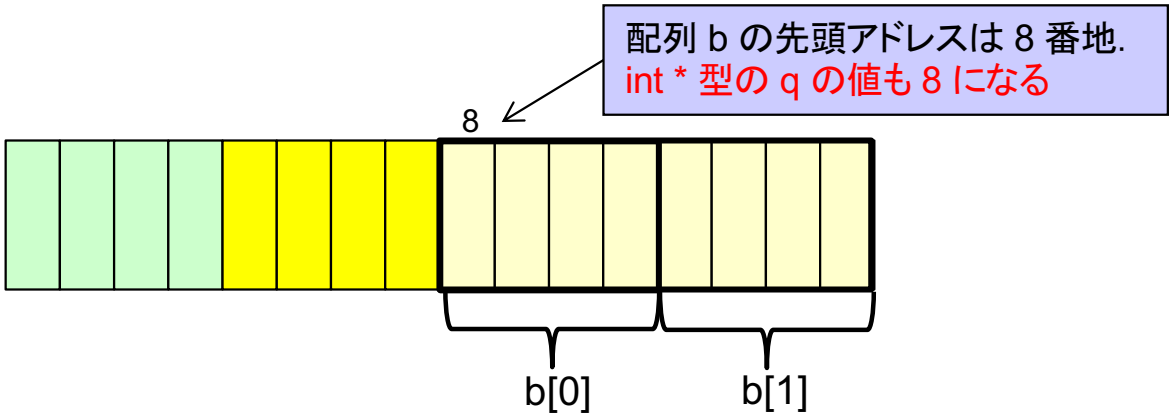
配列とポインタの関係 (3/6)

```
int main()
{
    char *p, a[ ] = {'a', 'b', 'c', 'd'};
    p = a;
    printf("%c", *p);          // 'a' を表示
    p++;
    printf("%c", *p);          // 'b' を表示
    printf("%c", *(p - 1));    // 'a' を表示
    return 0;
}
```



配列とポインタの関係 (4/6)

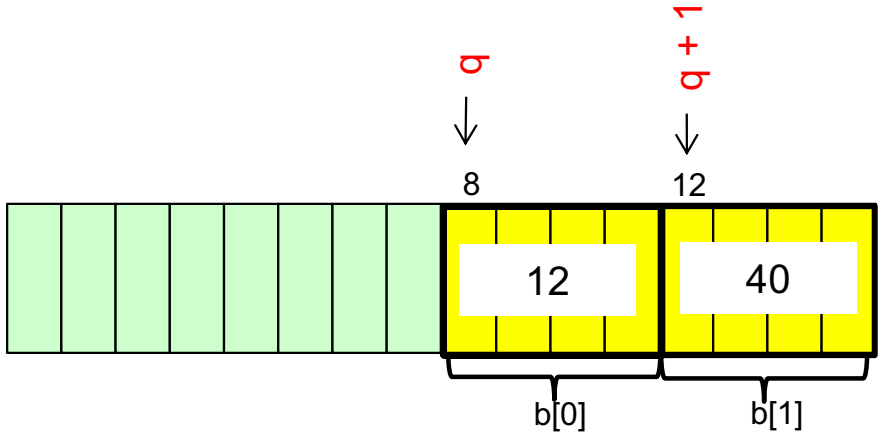
- `int b[2];`
- b の値は配列 b の先頭アドレス．したがって，ポインタとして扱える
  - `int b[2];` `// b は int 型の変数 b[0] を指すアドレス`
    `int *q;`
    `q = b;` `// したがって，b は int* 型の変数に代入できる`



配列とポインタの関係 (5/6)

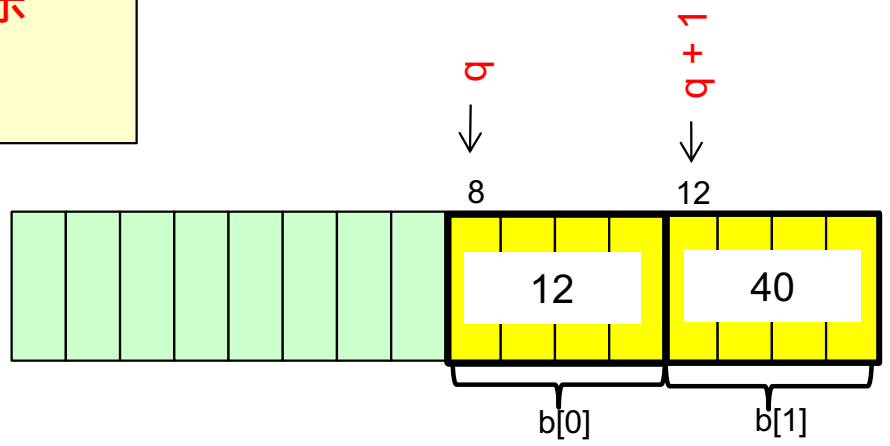
- char 型の配列以外では，ポインタ演算に注意
  - `int *q, b[ ] = {12, 40};`
    `q = b;`　　`// q は b[0] を指す．下図では q の値は 8`

- q + 1 の意味は？
  - b[1] を指すポインタになる
    - q は int へのポインタ
    - int 型は 4 バイト
    - q + 1 は 12 (= 8 + 4) になる



配列とポインタの関係 (6/6)

```
int main()
{
    int *q, b[ ] = {12, 40};
    q = b;
    printf("%d", *q); // 12 を表示
    q++;
    printf("%d", *q); // 40 を表示
    printf("%d", *(q - 1)); // 12 を表示
    return 0;
}
```



sizeof (1/3)

sslab system software

- ある型の変数がメモリ上で占めるバイト数を返す
- 型名を引数にすると，その型のバイト数を返す
  - `sizeof(char)` → 1
  - `sizeof(int)` → 4
- 変数名を引数にすると，その変数のバイト数を返す
  -
```
char a;
int b;
sizeof(a) → 1
sizeof(b) → 4
```

sizeof (2/3)

- ポインタ変数の増減は，ポインタの指す型のサイズ分だけ増減する

```
int *ip1, *ip2;
char *cp;
...
ip1 = ip2;          ←――― これはOK
ip2 = cp;           ←――― これはエラー
ip2 = (int *)cp;    ← これはOK
...
ip1++;  ←――――――― sizeof(int) 増える
cp++;   ←――――――― sizeof(char) 増える
ip1 += 2;←――――――― sizeof(int) * 2 増える
cp += 2; ←――――――― sizeof(char) * 2 増える
```

sizeof (3/3)

- ちょっと面白い使い方
  - 配列の要素数を得ることができる（時がある）

- 例：

```
int a[ ] = {1, 2, 3, 4, 5};
sizeof(a) / sizeof(a[0]) → 5 になる
```

  - sizeof(a) は配列 a がメモリ中で占めるバイト数
  - sizeof(a[0]) は配列 a の 1 要素がメモリに占めるバイト数
  - よって，sizeof(a) / sizeof(a[0]) は配列の要素数になる

配列の要素数が明示されている場合のみ

配列を引数とする関数呼び出し (1/5)

sslab system software

- 配列を引数とする関数の宣言:
  - 仮引数は “要素の型 仮引数名[ ]” と宣言する

```
// a は char 型の配列.
void show(char a[ ], int n)
{
      …
}
```

- 呼び出し方:
  - 配列の先頭アドレスを渡す

```
char b[3];
b[0] = 'a'; b[1] = 'b'; b[2] = 'c';
…
show(b, 3);
```

配列を引数とする関数呼び出し (2/5)

変数 a は char 型の配列

変数 n は配列の要素数.
引数の a から配列の要素数はわからない.
そのため，要素数を引数として渡している

```
void show(char a[ ], int n)
{
    int i;
    for (i = 0; i < n; i++) {
          printf("%c", a[i]);
    }
    printf("¥n");
}

int main()
{
     char b[ ] = {'a', 'b', 'c'};
     show(b, 3);
     return 0;
}
```

配列として使える

配列の先頭アドレスを渡す.
show() の仮引数 a には
配列 b の先頭アドレスが入る

配列を引数とする関数呼び出し (3/5)

sslab system software

- 仮引数には配列の先頭アドレスが渡される
  - 仮引数の型をポインタ型にしてもよい

```
// a は char へのポインタ型
void show(char *a, int n)
{
    …
}
```

- 同じように呼び出せる
  - 配列の先頭アドレスを渡す

```
char b[3];
b[0] = 'a'; b[1] = 'b'; b[2] = 'c';
…
show(b, 3);
```

配列を引数とする関数呼び出し (4/5)

変数 a は char 型へのポインタ

```
void show(char *a, int n)
{
    int i;
    for (i = 0; i < n; i++) {
        printf("%c", *(a + i));
        printf("%c", a[i]);
    }
    printf("¥n");
}

int main()
{
    char b[ ] = {'a', 'b', 'c'};
    show(b, 3);
    return 0;
}
```

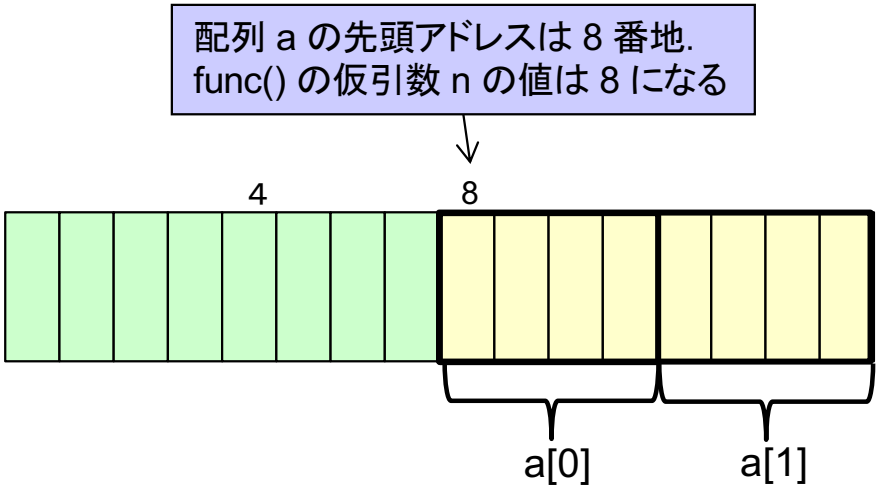
ポインタを使って配列にアクセス

配列としても使える

配列の先頭アドレスを渡す.
show() の仮引数 a には
配列 b の先頭アドレスが入る

配列を引数とする関数呼び出し (5/5)

- 仮引数には配列の先頭アドレスが渡される
  - 右のプログラムで・・・
    - n[0] = 10 は *n = 10,
      n[1] = 15 は *(n + 1) = 15 と同等
    - よって, main() 内の a[0], a[1] の値も書き換わる




```
void func(int[ ]);

int main()
{
    int a[2];
    func(a);
    printf("%d, %d", a[0], a[1]);
}

void func(int n[ ])
{
    . . .
    n[0] = 10;
    n[1] = 15;
    . . .
}
```

10, 15 と表示される

*n = 10;
*(n + 1) = 15;
と同じ

例：配列の要素の合計を求める

```
#define N 10

int sum(int [], int);

int main()
{
    int i, x[N], r;

    for (i = 0; i < N; i++)
        x[i] = i + 1;

    r = sum(x, N);
    printf("sum = %d¥n", r);
    return 0;
}

int sum(int a[ ], int n)
{
   int i, total = 0;
   for (i = 0; i < n; i++)
         total += a[i];

   return total;
}
```

配列を引数とする関数のプロトタイプ宣言.
・仮引数名を書かなくてよい

配列を引数とする関数 sum() に,
配列 x の先頭アドレスを渡す

配列を引数とする関数の宣言

普通に使える

例題：文字の出現度数

- 標準入力から文字を順次読み込んで，数字および英字の出現度数を数え，その結果を表示しなさい．ただし，英字の大文字と小文字は区別しないものとする

getchar() の使い方

- int getchar()
  - 文字をひと文字読み込んで，その文字の ASCII コードを返す
    - ASCII コードとは，文字につけられた番号のこと
    - コンピュータ内部では文字は ASCII コードで表現する
  - 入力がなかったら EOF という特別な値を返す

- 例：
  - 'a' という文字を読み込んだら，'a' に対応する ASCII コードである 97 を返す
  - '0' という文字を読み込んだら，'0' に対応する ASCII コードである 48 を返す

- C 言語では文字と ASCII コードを同一視する
  - 'a' と書いたら，それは数字の 97 と同等である
  - '0' と書いたら，それは数字の 48 と同等である

ASCIIコード表

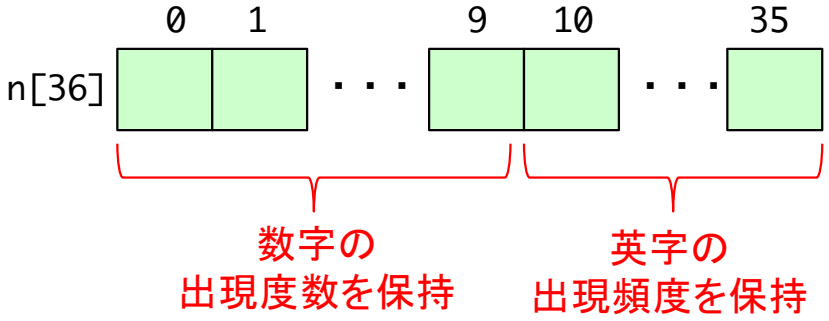
	0	16	32	48	64	80	96	112
+0	NULL	DLE	SP	0	@	P	`	p
+1	SOH	DC1	!	1	A	Q	a	q
+2	STX	DC2	“	2	B	R	b	r
+3	ETX	DC3	#	3	C	S	c	s
+4	EOT	DC4	$	4	D	T	d	t
+5	ENQ	NAK	%	5	E	U	e	u
+6	ACK	SYN	&	6	F	V	f	v
+7	BEL	ETB	‘	7	G	W	g	w
+8	BS	CAN	(	8	H	X	h	x
+9	HT	EM	)	9	I	Y	i	y
+10	NL	SUB	*	:	J	Z	j	z
+11	VT	ESC	+	;	K	[	k	{
+12	NP	FS	,	<	L	¥	l	|
+13	CR	GS	-	=	M	]	m	}
+14	SO	RS	.	>	N	^	n	~
+15	SI	US	/	?	O	_	o	DEL

赤字: 制御文字
青地: スペース
黒字: 通常文字

例題: 考え方

- 文字 '0' ～ '9' と 'a' ～ 'z' の出現度数を保持する変数が必要
  - (int 型の変数を36個) → 煩雑

- 配列を利用する
  - int n[36];



- 入力された文字の ASCII コードを c とすると，添え字 i は次のように求められる
  - '0' ～ '9': i = c – '0' (i は 0～9)
  - 'a' ～ 'z': i = c – 'a' + 10 (i は 10～35)
  - 'A' ～ 'Z': i = c – 'A' + 10 (i は 10～35)

例題: 配列の初期化

```
#include <stdio.h>

void count(int [ ]);

int main()
{
        int n[36], c, i;

        for (i = 0; i < 36; i++)
                n[i] = 0;
```

• n[0] ～ n[35] に 0 を代入 (初期化)

```
        count(n);




        return 0;
}
```

例題: count()

```
#include <stdio.h>

void count(int n[])
{
    int c, i;

    while ((c = getchar()) != EOF) {

    }
}
```

- c に入力した値を保持
- EOFになるまでループ

(c = getchar()) != EOF の意味:

1. まず, getchar() を呼び出す
2. その返り値を c に代入する
3. (c = getchar()) 全体の値は c の値となる
4. (c = getchar()) 全体の値が EOF と等しくない, すなわち, c の値が EOF と等しくないという意味

例題：入力された文字に応じて場合分け

```
#include <stdio.h>

void count(int n[ ])
{
        int c, i;

        while ((c = getchar()) != EOF) {
                if (c >= '0' && c <= '9')
                        i = c - '0';
                else if (c >= 'a' && c <= 'z')
                        i = c - 'a' + 10;
                else if (c >= 'A' && c <= 'Z')
                        i = c - 'A' + 10;
                else
                        continue;
                n[i]++;
        }
}
```

if文の範囲
- 入力が数字か英字のときは
  配列の添字 i を求める.
  添字の求め方はすでに説明した通り

入力がそれ以外のときは
ループの先頭へ

文字に対応する度数を1増やす
n は配列なので，main() の中の n[ ] の
要素が更新される

例題: 英字の出現度数の表示

```
#include <stdio.h>

int main()
{
	int n[36], c, i;

	for (i = 0; i < 36; i++)
		n[i] = 0;

	count(n);

	printf("char  freq¥n");
	for (i = 0, c = '0'; i < 10; i++, c++)
		printf("   %c : %3d¥n", c, n[i]);
	for (c = 'a'; i < 36; i++, c++)
		printf("   %c : %3d¥n", c, n[i]);
	return 0;
}
```

- カンマ演算子 (,) を使うと式を並べられる.
  並べた式を順に評価する
- `i = 0, c = '0'` は
  `i = 0` を実行後, `c = '0'` を実行するという意味
- `i++, c++` は
  `i++` を実行後, `c++` を実行するという意味

- 最後の for 文に入ってきたとき i の値は 10
- i が 10〜35 の間、ループ本体が実行される

- c の初期値は ‘a’
- c++ を実行すると c の値は ‘b’ に

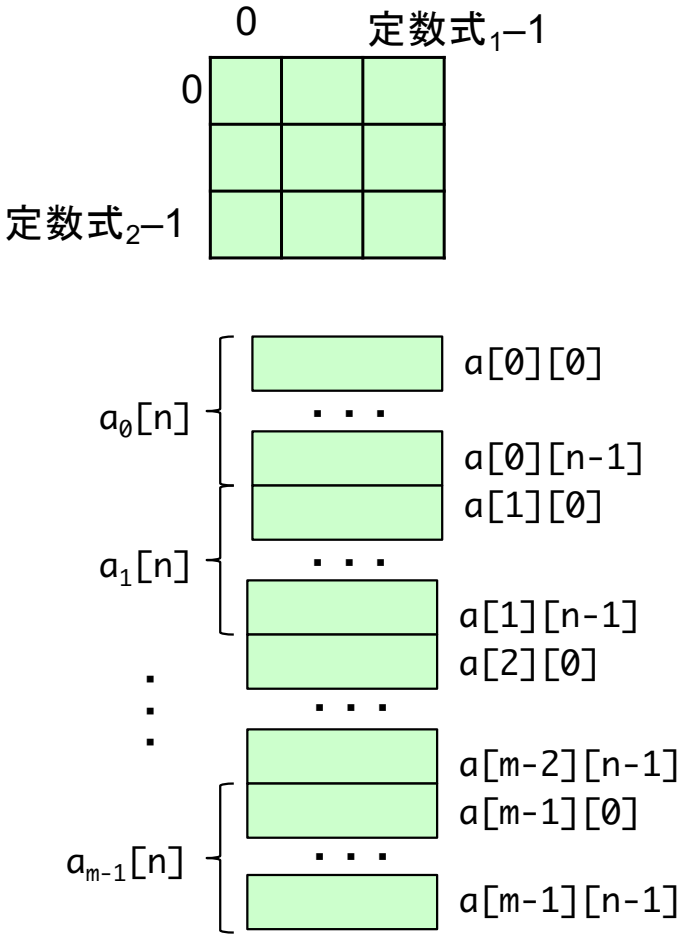
2次元配列

- 2次元配列の宣言の一般形

  型名 配列名 [定数式$_2$] [定数式$_1$]

  - 正しい例: `int a[5][10];`
  - 誤りの例: `int a[5,10];`

- 2次元配列の要素
  - 例: `int a[m][n]` → `a[n]` が `m` 個連なっている
  - 右図はメモリイメージ



2次元配列の仮引数宣言

- 仮引数宣言の一般形　型名 配列名 [[定数式$_2$]] [定数式$_1$]
  - 定数式$_1$ は必須
  - 例: `int func(int m[ ][10]) { . . . }`

- プロトタイプ宣言の一般形
  - 配列名、定数式$_2$は省略可能　型名 [配列名] [[定数式$_2$]] [定数式$_1$]
  - 定数式$_1$ は必須
  - 例: `int func(int[ ][10]);`

これまでに学んできた型 (type)

sslab system software

- 組込み型 (*built-in type* あるいは *primitive type*)
  - プログラミング言語であらかじめ用意されている型

- これまでに紹介したもの
  - `char` 型
    - 8 bit 幅. すなわち -128 ～ 127 の範囲の整数を表せる
  - `int` 型
    - 32 bit 幅. すなわち −2,147,483,648 ～ 2,147,483,647 の範囲の整数を表せる
  - `float` 型
    - 単精度の浮動小数点数

組込みの整数型（理工学 ITC の場合）

sslab system software

- char 型
  - 8 bit 幅．すでに紹介済
- short 型
  - 16 bit 幅．-2^{15} ～ $2^{15} - 1$ の範囲の整数を表す
- int 型
  - 32 bit 幅．すでに紹介済
- long 型
  - int 型と同じ
- long long 型
  - 64 bit 幅．-2^{63} ～ $2^{63} - 1$

符号付整数と符号なし整数

- 通常は符号付の整数となる
  - すなわち，マイナスの数を表現できる
  - 内部的には 2 の補数表現
    - 「計算機基礎」で学習済

- unsigned という修飾子をつけると「符号なし」となる
  - 非負整数として解釈される
  - 例：

```
unsigned char x;
                // x は 0 ～ 255 の符号なし 8 bit の整数
unsigned int y;
                // y は 0 ～ 2^32 – 1 の符号なし 32 bit の整数
```

定数の宣言：const

- 変数の値が定数であることを宣言する
  - 初期化したあとは，値の変更ができない
- 型名の前に const をつける
  - 例：

```
/* 変数 a は定数．宣言するときに初期化 */
const int a = 4;
a = 3;              /* エラー */

/* 変数 PI は定数 */
const double PI = 3.14;
PI = 3.3;           /* エラー */
```

ビット演算

- 整数をビット単位で操作することができる
  - ビット単位で否定，AND, OR, XOR が計算できる

名称	演算子	一般形	意味
ビット否定	~	~e	e の各ビットを反転させる 例: ~1101 1001 = 0010 0110
ビットAND	&	e_1 & e_2	e_1 と e_2 の各ビットについて AND をとる 例: 1101 1001 & 1111 0000 = 1101 0000
ビットOR	|	e_1 | e_2	e_1 と e_2 の各ビットについて OR をとる 例: 1101 1001 | 1111 0000 = 1111 1001
ビットXOR	^	e_1 ^ e_2	e_1 と e_2 の各ビットについて XOR をとる 例: 1101 1001 ^ 1111 0000 = 0010 1001

シフト演算 (1/2)

- 整数に対してビットシフトを行うことができる

名称	演算子	一般形	意味
右シフト	>>	e_1 >> e_2	e_1 を右に e_2 ビットだけシフトする
左シフト	<<	e_1 << e_2	e_1 を左に e_2 ビットだけシフトする

  - 左シフトの例:

```
int x = 0xD;   // 2 進数で 1101. 10 進数で 13.

printf("%d", x << 2);
```

左に 2 ビットシフトする
2 進数で 110100,
10 進数で 52 となる

シフト演算 (2/2)

- 右シフトの例：
  - 符号付き整数の場合，算術シフトとなる
    - 符号ビットと同じ値が挿入される

```
int x = 0xF0F0A1A1;   // 2 進数で 1111 0000 1111 0000 1010 0001 1010 0001

printf("%d", x >> 2);
```

右に 2 ビット算術シフトする．符号ビットが 1 なので・・・
1111 1100 0011 1100 0010 1000 0110 1000

  - 符号なし整数の場合，論理シフトとなる
    - 常に 0 が挿入される

```
unsigned int x = 0xF0F0A1A1;   // 2 進数で 1111 0000 1111 0000 1010 0001 1010 0001

printf("%d", x >> 2);
```

右に 2 ビット論理シフトする．
0011 1100 0011 1100 0010 1000 0110 1000

まとめ

sslab system software

- 配列とポインタの関係
  - C 言語では配列とポインタはほぼ同一視する
  - ポインタ演算とその意味
  - `sizeof`
- 配列を引数とした関数呼び出し
  - 配列の先頭アドレスを call-by-value で渡す
  - call-by-reference のように振る舞うので注意
- 2 次元配列についても少しだけ
  - ポインタで扱うには慣れがいる
- 雑多な話題：いくつかの型，ビット演算とシフト演算