

	もくじ	今回 (#08) の内容：シラバスでの該当部分
<div>#08 ポインタと配列 2022 年度 / プログラミング及び実習 III</div> <div>角川裕次 龍谷大学 先端理工学部</div> <div>1 / 28</div>	<div>1 第 10-3 節 ポインタと配列</div> <div>2 / 28</div>	<div>小テーマ: ポインタと配列</div> <div>第 12 回：配列型とポインタ型 第 13 回：ポインタの演算 第 14 回：配列を引数とする関数</div> <div>3 / 28</div>
重要概念リスト	今回の実習・課題 (manaba へ提出)	きょう特に心がけてほしいこと
<div>■ ポインタ演算 <math>p + i</math> ■ ポインタのインクリメント <math>p++</math> ■ ポインタのデクリメント <math>p--</math> ■ 配列 <math>a</math> に対する <math>\text{sizeof}(a)</math> と <math>\&amp;a</math> ■ 配列を引数とする関数呼出し</div> <div>4 / 28</div>	<div>実習内容と課題内容は講義途中に提示します (作成したファイル類は manaba に提出)</div> <div>5 / 28</div>	<div>プログラムコードを見て 以下のようなビジュアルをイメージできるようになること</div> <div><pre>int ary_set(int *v, int n, int val) {     ... } int main(void) {     int a[] = { ... };     ary_set(a, 5, 99);     ... }</pre><div>ソースコード</div><div><math>v</math> <math>n</math> <math>val</math> <math>v[0] \ a[0]</math> <math>v[1] \ a[1]</math> <math>v[2] \ a[2]</math> <math>v[3] \ a[3]</math> <math>v[4] \ a[4]</math></div></div> <div>ポインタの理解で重要</div> <div>■ ビジュアルをイメージできなければ理解できてないと思われ</div> <div>6 / 28</div>

## 第 10-3 節 ポインタと配列

### ポインタと配列 p.292

重要

#### 配列名とポインタ

配列 `a[ ]` に対して  
配列名 `a` によりその配列の先頭要素 `a[0]` へのポインタを表す

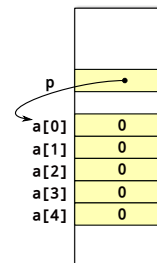
例

```
int a[5];
int *p = a;
```

`p = a` は `p = &a[0]` に等しい

- C 言語ではそういうおやくそく
- `p` の型は `int*` だから

注意：配列全体を指すのではない

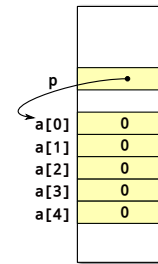


### 配列と各要素を指すポインタ

重要

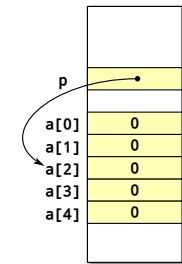
```
int a[5];
int *p = a;
```

`p` は `a[0]` を指す



```
int a[5];
int *p = &a[2];
```

`p` は `a[2]` を指す



7 / 28

8 / 28

9 / 28

### ポインタの演算

重要

#### ポインタ `p` への演算

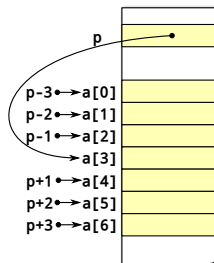
ポインタ `p` が配列の要素 `e` を指す時に以下の規則が成立

- ・ `p + i` : 要素 `e` の `i` 個だけ後方の要素を指す
- ・ `p - i` : 要素 `e` の `i` 個だけ前方の要素を指す

`p+i` と `&p[i]` は等価

例 : `p = &a[3]` のとき

- `p-2` は `a[1]` を指すポインタ値
- `p+2` は `a[5]` を指すポインタ値
- `p+3` は `a[6]` を指すポインタ値



10 / 28

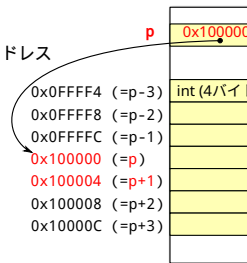
### ポインタの演算：理解のポイント

例 : `p` が整数 (`int`) を指すポインタ, 値が `0x100000` の時

- ・ 誤解 : `p+1 = 0x100001`
- ・ 正確 : `p+1 = 0x100004`
- ・ ただし `int` 型変数に 4 バイトを使用している場合
- ・ `p` が指す `int` 型データの次の `int` 型データのアドレス

**`p+1` の理解** (`p` はポインタ変数)

- ・ 誤解 : `p` の値に 1 バイトを加えたアドレス
- ・ 正確 : `p` の値に要素のサイズ分のバイト数を加えたアドレス



11 / 28

### ポインタのインクリメント

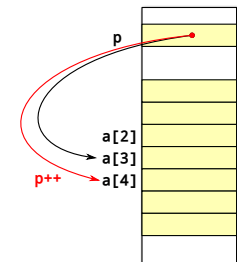
重要

ポインタ `p` のインクリメント `p++`

- 1 つ次の要素を指すように値が変化
- `p = p + 1` と等価
- (increment : 増やす)

「1 つ次の要素」:  
配列だと思ったときの 1 つ次の要素

例 : `p = &a[3]` のときの `p++` の結果  
`p = &a[4]` となる



12 / 28

## ポインタのデクリメント

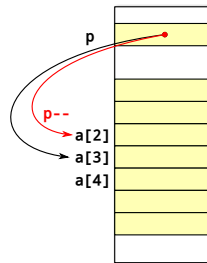
重要

ポインタ  $p$  のデクリメント  $p--$

- 1 つ前の要素を指すように値が変化
- $p = p - 1$  と等価
- (decrement: 減らす)

「1 つ前の要素」:  
配列だと思ったときの 1 つ前の要素

例:  $p = \&a[3]$  のときの  $p--$  の結果:  
 $p = \&a[2]$  となる



13 / 28

## 配列の要素のアドレスの表示

List 10-9

```
#include <stdio.h>
int main(void)
{
    int i;
    int a[5] = {1, 2, 3, 4, 5};
    int *p = a; /* pはa[0]を指す */
    for (i = 0; i < 5; i++)
        printf("&a[%d] = %p    p+%d = %p\n",
               i, &a[i], i, p + i);
    return 0;
}
```

実行例 (Ubuntu での場合; 出力内容はシステム依存)

```
&a[0] = 0x7ffa62c06f0    p+0 = 0x7ffa62c06f0
&a[1] = 0x7ffa62c06f4    p+1 = 0x7ffa62c06f4
&a[2] = 0x7ffa62c06f8    p+2 = 0x7ffa62c06f8
&a[3] = 0x7ffa62c06fc    p+3 = 0x7ffa62c06fc
&a[4] = 0x7ffa62c0700    p+4 = 0x7ffa62c0700
```

14 / 28

## 配列名 $a$ に対する `sizeof` と `&`

重要

`sizeof(a)` は配列全体の大きさを表す

- 注意: `sizeof(a[0])` ではない
- 例: `int a[10];` に対する `sizeof(a)`
- `sizeof(a) = sizeof(int) × 10` を得る
- 要素サイズを個数分、つてことですね

`&a` は配列全体へのポインタを表す

- 注意: `a[0]` へのポインタへのポインタではない
- 例: `int a[10];` に対する `&a`
- 配列全体 `a[10]` へのポインタを表す
- ポインタの型は `int(*)[10]` (要素数 10 の `int` 型配列へのポインタ)

15 / 28

## 間接演算子と添字演算子

p.294

重要

ポインタ  $p$  への演算結果に対する間接演算子

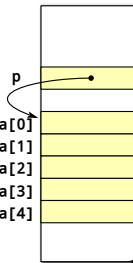
ポインタ  $p$  が配列の要素  $e$  を指す時に以下の規則が成立

- ・  $*(p + i)$  は  $p[i]$  と表記できる
- ・  $*(p - i)$  は  $p[-i]$  と表記できる

例:  $p = \&a[0]$  の場合に以下が成立

- ポインタ値に関して  
 $\&a[1] = a+1 = \&p[1] = p+1$
- 同じオブジェクトを指すのだから値は同一  
 $a[1] = *(a+1) = p[1] = *(p+1)$

```
*(a+0) = p[0] = *(p+0) = a[0]
*(a+1) = p[1] = *(p+1) = a[1]
*(a+2) = p[2] = *(p+2) = a[2]
*(a+3) = p[3] = *(p+3) = a[3]
*(a+4) = p[4] = *(p+4) = a[4]
```



16 / 28

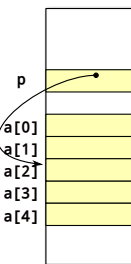
## 配列要素とポインタ

$p = \&a[2]$  の場合

- このとき  $*p$  は  $a[2]$  のエイリアス (別名)

```
*(a+0) = p[-2] = *(p-2) = a[0]
*(a+1) = p[-1] = *(p-1) = a[1]
*(a+2) = p[0] = *(p+0) = a[2]
*(a+3) = p[1] = *(p+1) = a[3]
*(a+4) = p[2] = *(p+2) = a[4]
```

$*(a+0)$  は  $*a$  と簡潔に書ける



17 / 28

## 同じ要素へアクセスする方法いろいろ

2 つは同一の要素を表す

- $a[i]$
- $*(a+i)$

(配列先頭から  
 $i$  個後ろの要素)

2 つは同一のポインタを表す

- $\&a[i]$
- $a+i$

(配列先頭から  
 $i$  個後ろの要素へのポインタ)

18 / 28

## ポインタの演算

できること

- ポインタと整数の加減算 (例:  $p+4$ )
- ポインタからポインタの減算 (例:  $p-q$ )

できないこと

- ポインタとポインタの加算 (例:  $p+q$ )

例: ポインタ演算

```
int a[10];
p1=&a[1];
p6=&a[6];
```

- ポインタと整数の加算:  $p1+7 = \&p[8]$
- ポインタと整数の減算:  $p6-2 = \&p[4]$
- ポインタからポインタの減算:  $p6-p1 = 6 - 1 = 5$

19 / 28

## ポインタの演算を試してみる

ソースコード (独自)

```
#include <stdio.h>
int main(void)
{
    int a[10];
    int *p0 = &a[0], *p1 = &a[1], *p2 = &a[2];
    printf("%ld, %ld\n", sizeof(a), sizeof(a[0]));
    printf("%p, %p, %p\n", p0, p1, p2);
    printf("%ld, %ld\n", p2-p0, p1-p0);
    return 0;
}
```

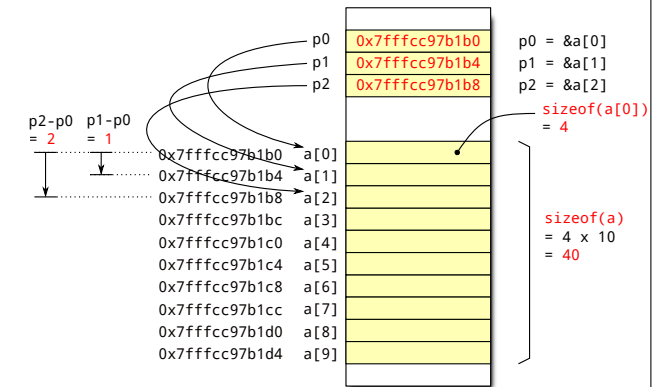
実行例

```
40, 4
0x7fffc97b1b0, 0x7fffc97b1b4, 0x7fffc97b1b8
2, 1
```

(実行するシステムにより実行結果が違う場合あり)

20 / 28

## ポインタの演算を試してみる: 超図解



21 / 28

## 配列の要素の値とアドレスを表示

List 10-10 (全体)

```
#include <stdio.h>
int main(void)
{
    int a[5] = {1, 2, 3, 4, 5};
    int *p = a;
    for (int i = 0; i < 5; i++)
        printf("a[%d] = %d *(a+%d) = %d p[%d] = %d *(p+%d) = %d\n",
               i, a[i], i, *(a+i), i, p[i], i, *(p+i));
    for (int i = 0; i < 5; i++)
        printf("&a[%d] = %p a+%d = %p &p[%d] = %p p+%d = %p\n",
               i, &a[i], i, (a+i), i, &p[i], i, (p+i));
    return 0;
}
```

配列 `a` の初期値を覚えておこう

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

- $a[0]=1, a[1]=2, a[2]=3, a[3]=4, a[4]=5$

22 / 28

## 配列の要素の値とアドレスを表示; 実行例 1/1

List 10-10 前半, 表示する値 4 種 ( $i=0,1,\dots,4$ )

```
a[i]      *(a + i)    p[i]      *(p + i);
```

出力: それぞれ同じ値

```
a[0] = 1 *(a+0) = 1 p[0] = 1 *(p+0) = 1
a[1] = 2 *(a+1) = 2 p[1] = 2 *(p+1) = 2
a[2] = 3 *(a+2) = 3 p[2] = 3 *(p+2) = 3
a[3] = 4 *(a+3) = 4 p[3] = 4 *(p+3) = 4
a[4] = 5 *(a+4) = 5 p[4] = 5 *(p+4) = 5
```

23 / 28

## 配列の要素の値とアドレスを表示; 実行例 2/2

List 10-10 後半, 表示する値 4 種 ( $i=0,1,\dots,4$ )

```
&a[i]      a+i
&p[i]      p+i
```

出力: それぞれ同じポインタ値

```
&a[0] = 0x7ffd6c4e1e60 a+0 = 0x7ffd6c4e1e60
&p[0] = 0x7ffd6c4e1e60 p+0 = 0x7ffd6c4e1e60

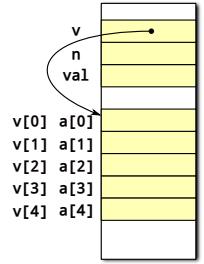
&a[1] = 0x7ffd6c4e1e64 a+1 = 0x7ffd6c4e1e64
&p[1] = 0x7ffd6c4e1e64 p+1 = 0x7ffd6c4e1e64

&a[2] = 0x7ffd6c4e1e68 a+2 = 0x7ffd6c4e1e68
&p[2] = 0x7ffd6c4e1e68 p+2 = 0x7ffd6c4e1e68

&a[3] = 0x7ffd6c4e1e6c a+3 = 0x7ffd6c4e1e6c
&p[3] = 0x7ffd6c4e1e6c p+3 = 0x7ffd6c4e1e6c

&a[4] = 0x7ffd6c4e1e70 a+4 = 0x7ffd6c4e1e70
&p[4] = 0x7ffd6c4e1e70 p+4 = 0x7ffd6c4e1e70
```

24 / 28

配列とポインタの相違点 p.296	配列の受け渡し p.298	関数呼び出しによるポインタの受け渡し
<p>配列とポインタ：似ていることも多いが相違点もあり</p> <p>規則：配列名を代入式の左オペランドに出来ない</p> <p>できる</p> <pre>int *p; int y[5]; p = y;</pre> <ul style="list-style-type: none"><li>ポインタ変数への代入は可能</li></ul> <p>できない (コンパイルエラー)</p> <pre>int a[5]; int b[5]; a = b;</pre> <ul style="list-style-type: none"><li>配列まるごと一気に代入はできない</li><li>C 言語でのおやくそく</li></ul> <p>25 / 28</p>	<p>関数間での配列の受け渡し：ポインタにより受け渡す</p> <ul style="list-style-type: none"><li>要素数の情報は伝わらない</li><li>要素数は引数で明示的に渡す</li></ul> <p>List 10-11：配列の受け渡し</p> <pre>#include &lt;stdio.h&gt; void ary_set(int v[], int n, int val) {     for (int i = 0; i &lt; n; i++)         v[i] = val; } int main(void) {     int a[] = {1, 2, 3, 4, 5};     ary_set(a, 5, 99);     for (int i = 0; i &lt; 5; i++)         printf("a[%d] = %d\n", i, a[i]);     return 0; }</pre> <p>26 / 28</p>	<p>配列 a へのポインタを引数にして関数呼出し</p> <ul style="list-style-type: none"><li>仮引数名：v</li></ul> <p>関数内：v[0] により a[0] をアクセスできる</p> <pre>int ary_set(int *v,             int n,             int val) {     ... } int main(void){     int a[] = { ... };     ary_set(a, 5, 99);     ... }</pre> <p>ソースコード</p>  <p>27 / 28</p>
<p>おわり</p> <p>28 / 28</p>		