ポインタの裏話

岡﨑　直観

okazaki at ecei.tohoku.ac.jp

http://www.chokkan.org/

@chokkanorg

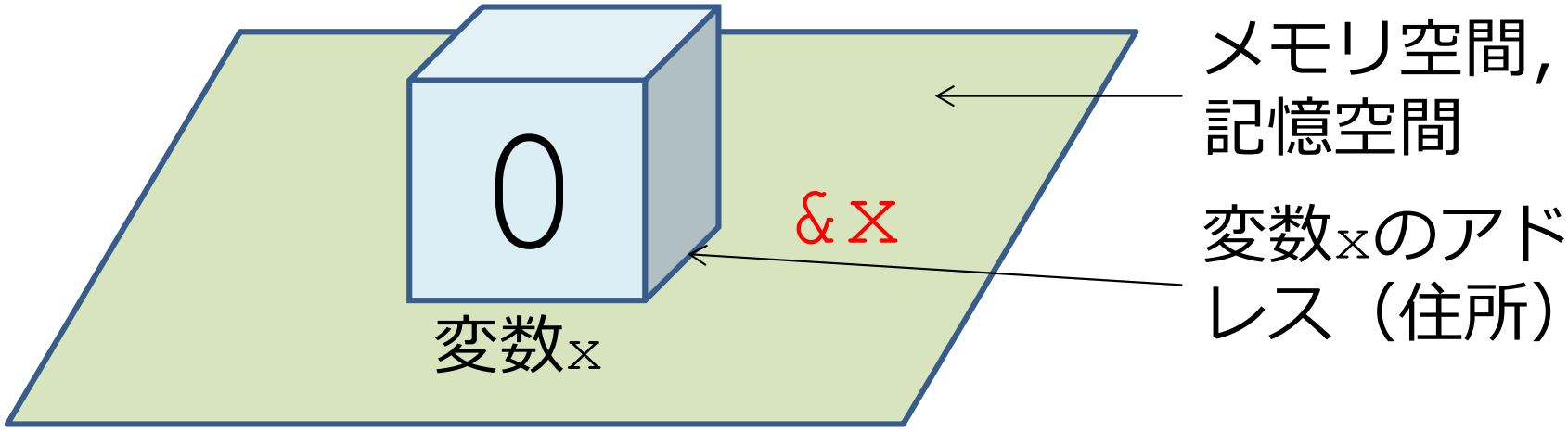
このような説明を覚えていますか？

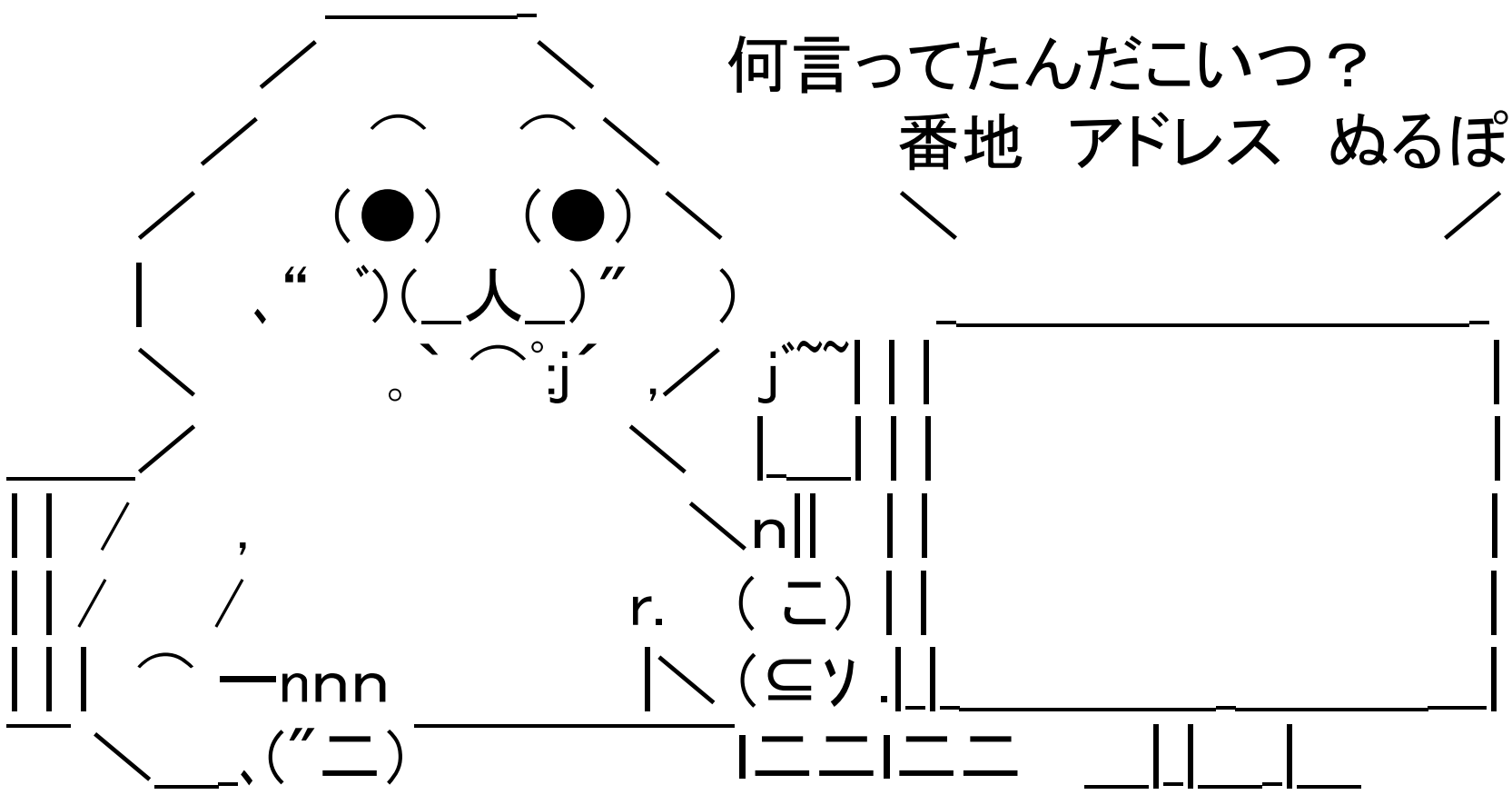
知らなくても全く問題ありません

`int x = 0;`

&xは変数xの「アドレス」

「番地」「住所」を返す

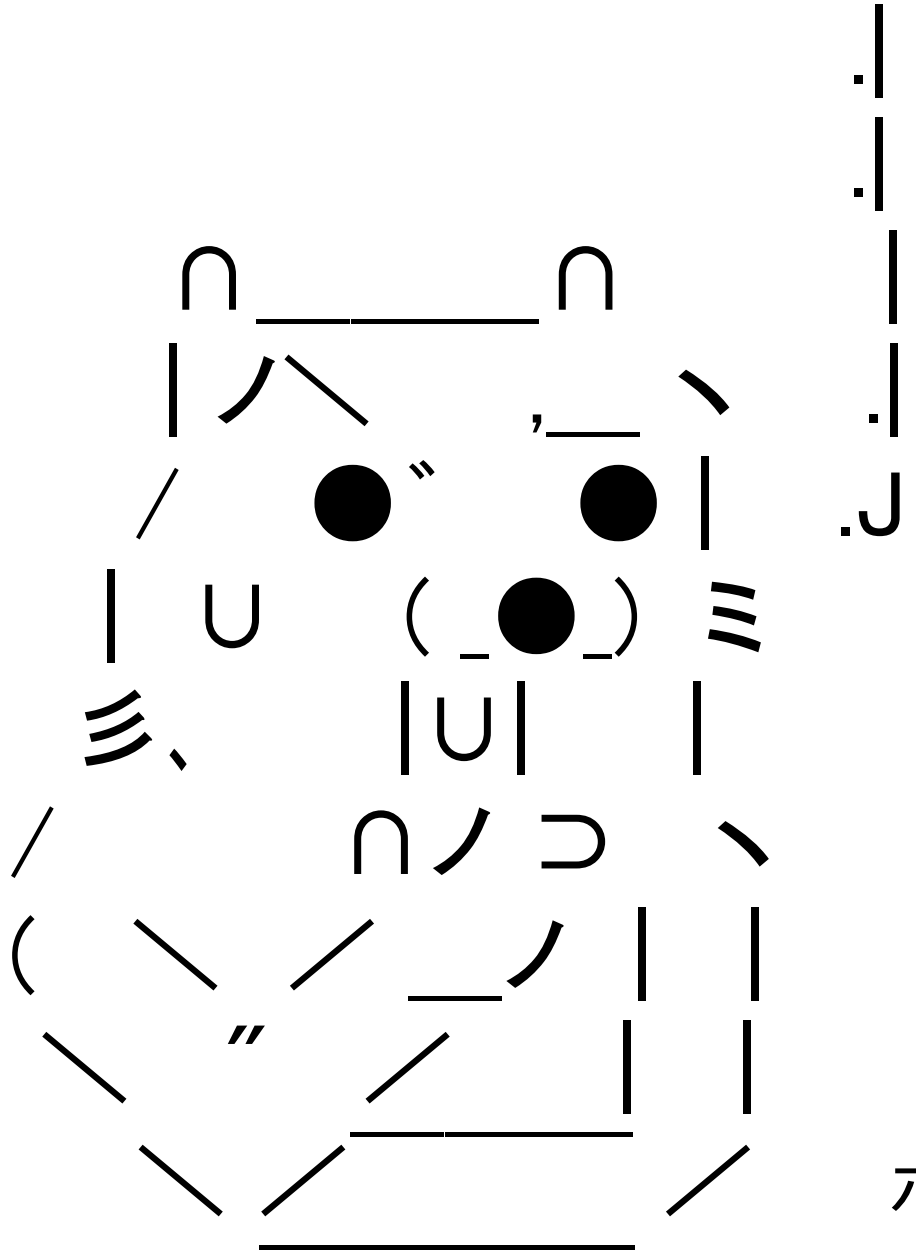



何言ってたんだこいつ？
番地　アドレス　ぬるぽ

別の例

```
int x = 0;
int *p = &x;
*p = 1;
```

変数xの値が1になるのは分かる

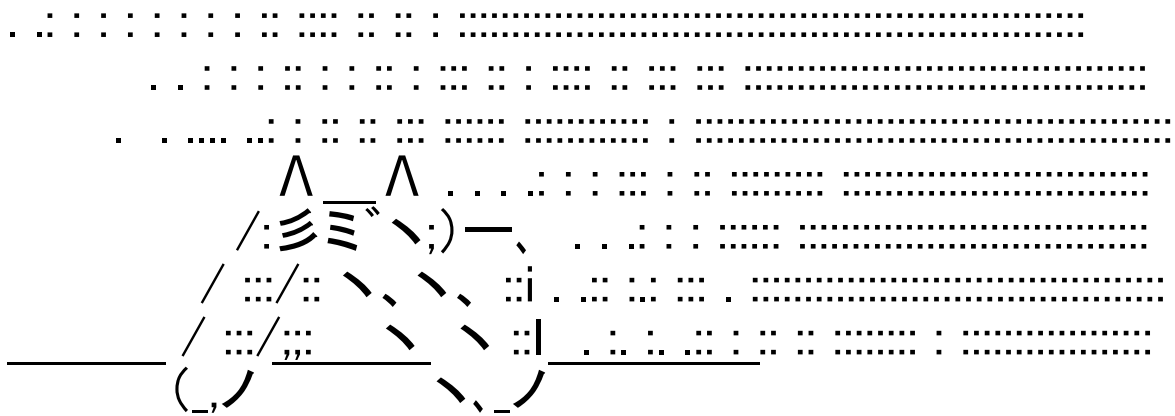


何故こんな回りくどいことを？

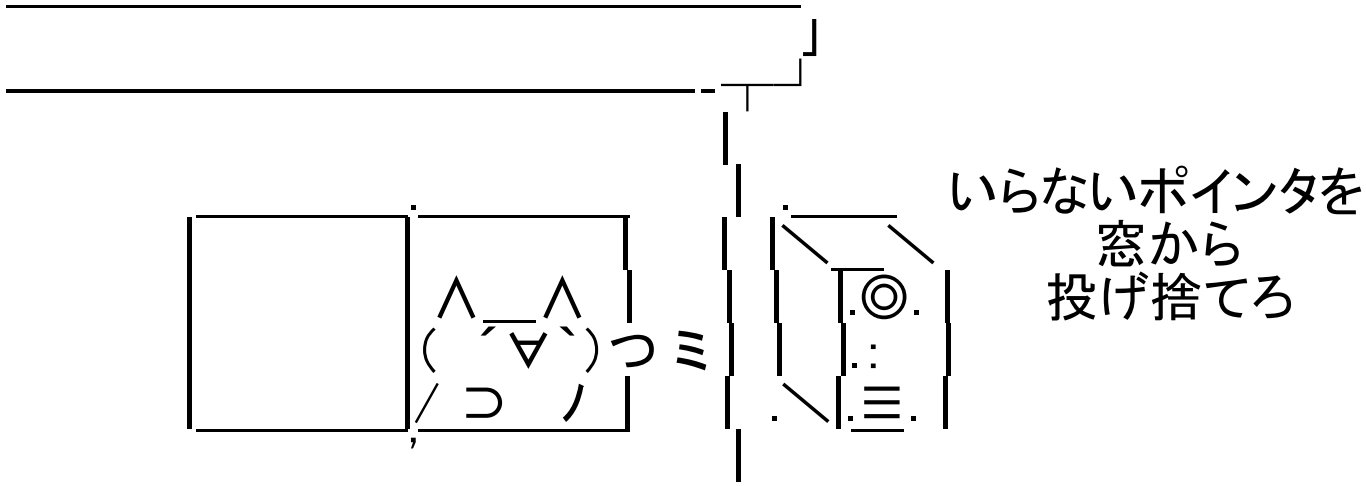
ポインタの必然性が感じられない

ポインタが分からない原因

計算機のメモリの仕組みを知らない



ポインタの必要性が分からない



今回はポインタに関する
知識を手加減せずに
すべて説明する

何故ポインタが要るのか？

変数があれば十分じゃね？

その答えは
「機械語」と
C言語の「立ち位置」

CPUが理解できる唯一の言語
それが・・・

機械語

人間の読むものではありません

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

int main(int argc, char *argv[])
{
    int x = atoi(argv[1]);
    printf("x = %d, &x = %08X¥n", x, &x);
    return 0;
}
```

機械語への翻訳例
（コンパイル結果）

cc
gcc

※Visual C++ 2010を用い
Intel x86系CPUの機械語に変換
※主要コード部のみ表示

```
55 8b ec 51 8b 45 0c 8b 48 04 51 ff 15 a4 20 39
00 83 c4 04 89 45 fc 8d 55 fc 52 8b 45 fc 50 68
18 30 39 00 ff 15 9c 20 39 00 83 c4 0c 33 c0 8b
e5 5d c3
```

機械語の世界には

変数がない

ポインタしかない

機械語のプログラムは１６進数の羅列

CPUのレジスタは一時変数のようなもの

C言語のコンパイル結果

機械語

アセンブリ言語

```
int main(int argc, char *argv[])
{
55                      push        ebp
8B EC                   mov         ebp,esp
51                      push        ecx
    int x = atoi(argv[1]);
8B 45 0C                mov         eax,dword ptr [ebp+0Ch]
8B 48 04                mov         ecx,dword ptr [eax+4]
51                      push        ecx
FF 15 A4 20 2F 01       call        dword ptr ds:[012F20A4h]
83 C4 04                add         esp,4
89 45 FC                mov         dword ptr [ebp-4],eax
    printf("x = %d, &x = %08X¥n", x, &x);
8D 55 FC                lea         edx,[ebp-4]
52                      push        edx
8B 45 FC                mov         eax,dword ptr [ebp-4]
50                      push        eax
68 18 30 2F 01          push        12F3018h
FF 15 9C 20 2F 01       call        dword ptr ds:[012F209Ch]
83 C4 0C                add         esp,0Ch
    return 0;
33 C0                   xor         eax,eax
}
8B E5                   mov         esp,ebp
5D                      pop         ebp
C3                      ret
```

強引に解釈すると
ebp-4が変数xへ
のポインタです

C言語は機械語のポインタの存在を隠蔽してくれる

一方で，

C言語は計算機を直接操る手段を残している

実行時間が速い

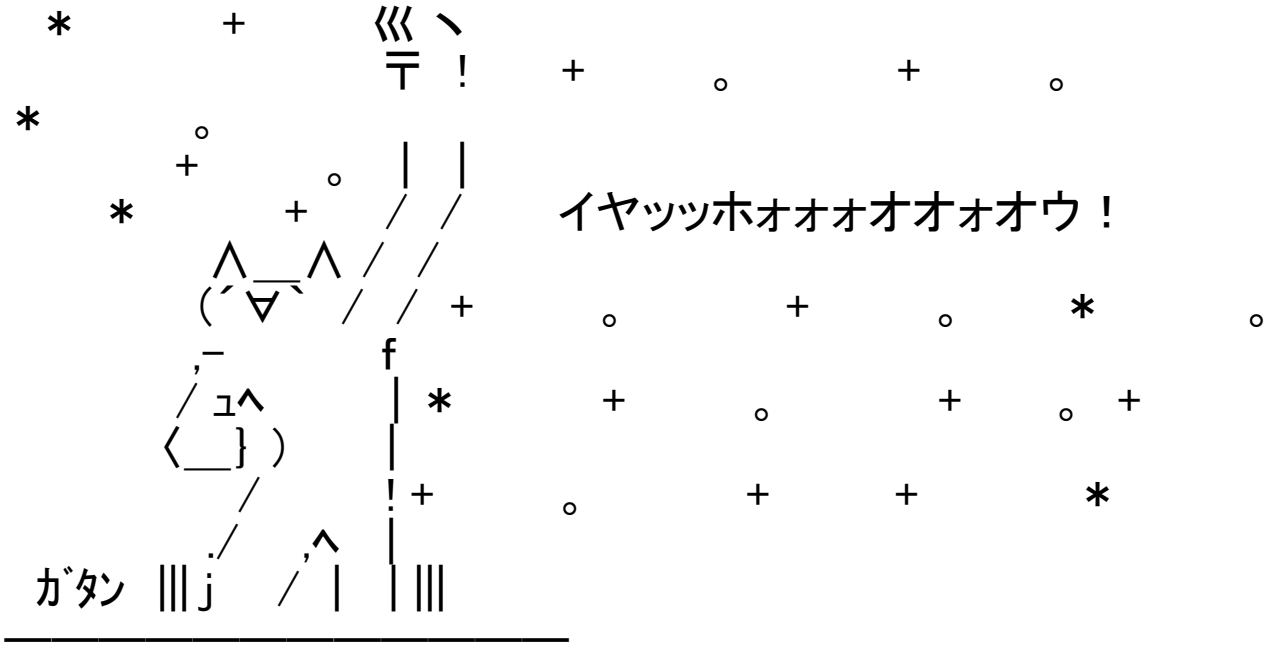
計算機の能力を最大限に引き出せる

文字列や配列など一部でポインタの概念を残している

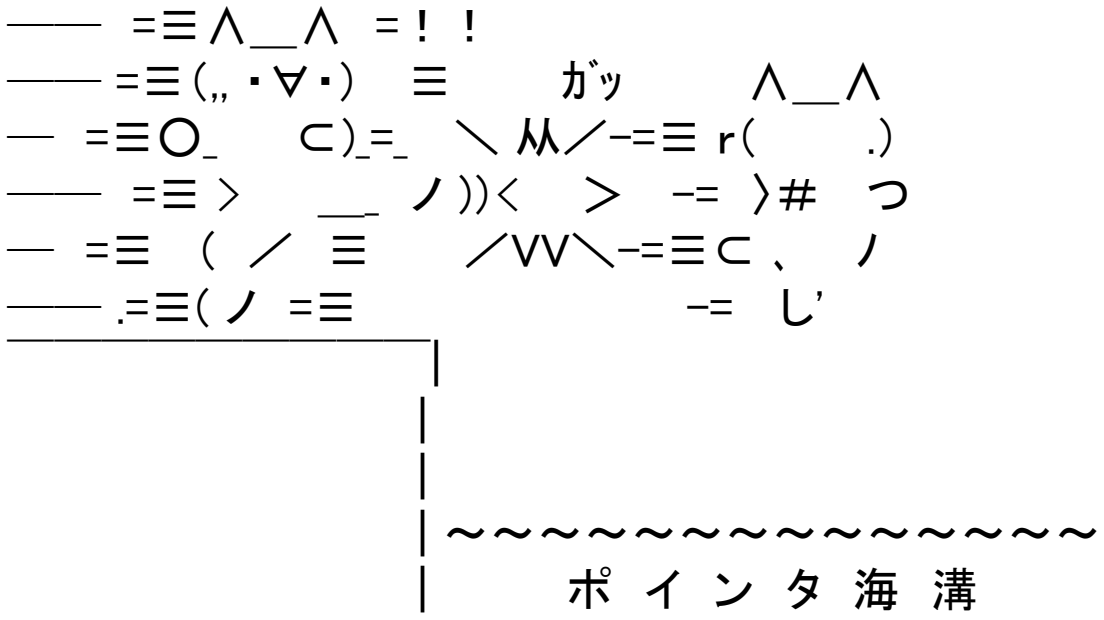
ポインタを隠蔽しきれず
ちょっと中途半端

結論としては

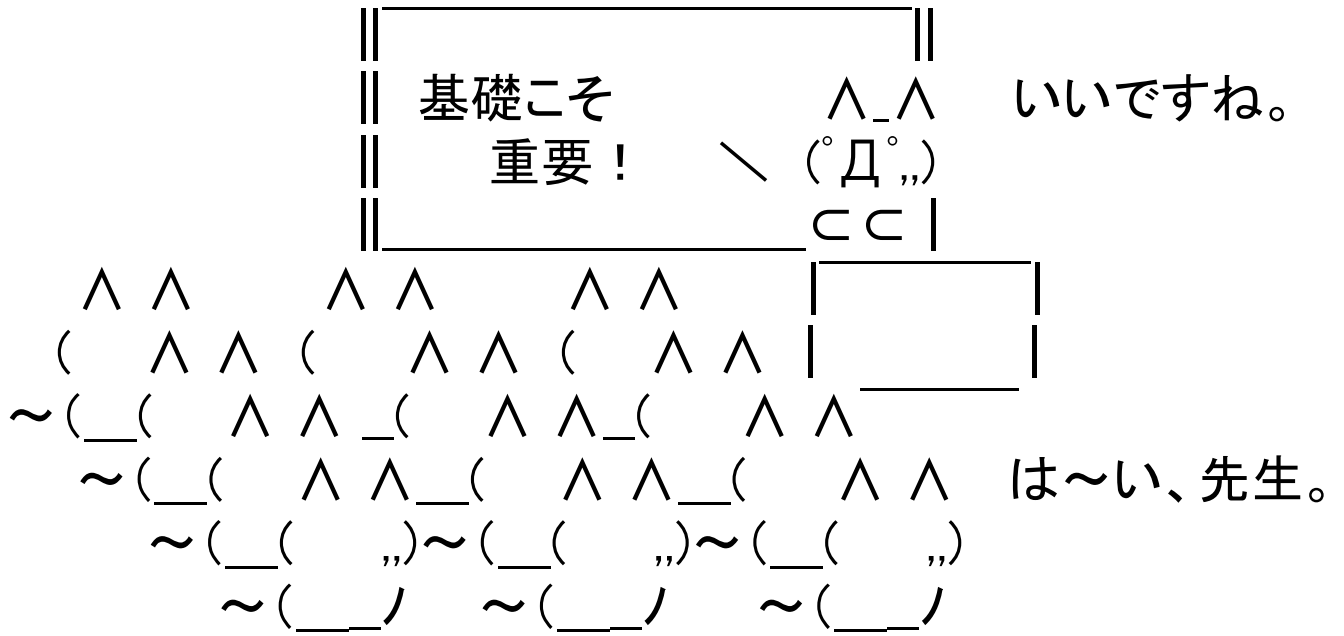
普段はポインタを使う必要がない



特定の用途にはポインタが必要

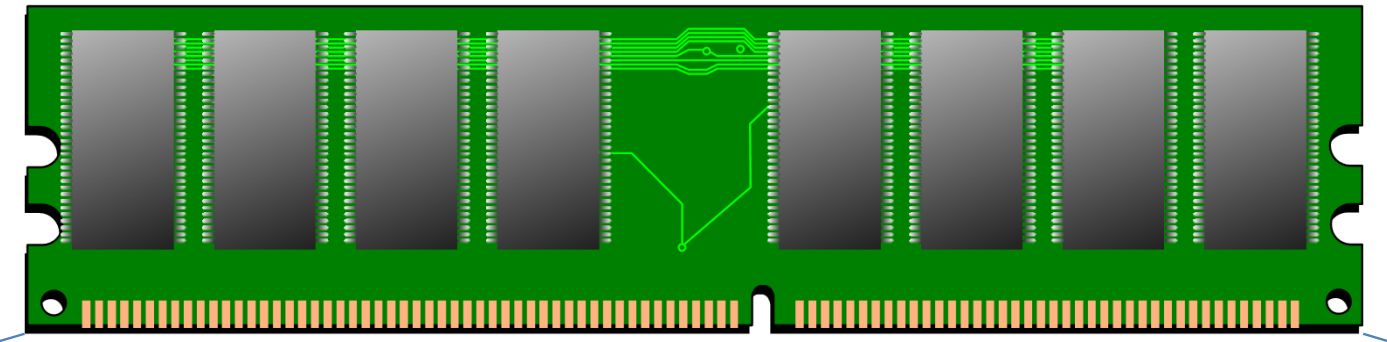


やはり計算機の知識は必要



計算機の記憶のしくみ

メモリ（主記憶）の構造



アドレス	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00000000	7F	45	4C	46	01	02	01	06	01	00	00	00	00	00	00	00
0x00000010	00	02	00	12	00	00	00	01	00	01	08	90	00	00	00	34
0x00000020	00	00	1A	A4	00	00	01	00	1A	34	00	20	00	05	00	28
0x00000030	00	1A	00	19	00	00	00	06	00	00	00	34	00	01	00	34
0x00000040	00	00	00	00	00	00	00	A0	00	00	00	A0	00	00	00	05
0x00000050	00	00	00	00	00	00	00	03	00	00	00	D4	00	01	00	D4
0x00000060	00	00	00	00	00	00	00	11	00	00	00	11	00	00	00	04
0x00000070	00	00	00	00	00	00	00	01	00	00	00	00	00	01	00	00
…																
…																
…																
0xFFFFFFF0																

主記憶の記憶状態の表現

01111111010001010100110001000110000000010000001000000……

アドレス	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00000000	7F	45	4C	46	01	02	01	06	01	00	00	00	00	00	00	00
0x00000010	00	02	00	12	00	00	00	01	00	01	08	90	00	00	00	34
0x00000020	00	00	1A	A4	00	00	01	00	1A	34	00	20	00	05	00	28
0x00000030	00	1A	00	19	00	00	00	06	00	00	00	34	00	01	00	34
0x00000040	00	00	00	00	00	00	00	A0	00	00	00	A0	00	00	00	05
…																
0xFFFFFFF0																

- 8ビット（=1バイト）をまとめて一つの升で表現
  - ひとつの升は（0x00～0xFF; 0～255）の値を記憶できる
- 計算機が持っている升の数: 主記憶の量（バイト）
  - この場合，0x100000000バイト = 4294967296バイト ≒ 4GB
- 16個の升をまとめて１行で示すのが慣例（← いろいろ便利なので）
  - このため，番地（アドレス）を表現するときに16進数を使う
  - 行と列を結合したものが番地（アドレス）：例えば0x00000028番地の値は0x1A

記憶内容の読み出し／書き込み

- 番地（位置）と型（大きさ）を指定する
  - 00010100番地に対して…
    - `char` (1バイト): `0x41`
    - `short` (2バイト): `0x4142`
    - `int` (4バイト): `0x41424344`
    - `double` (8バイト): `2393736.0`
    - `char[3]` (3バイト): `{0x41, 0x42, 0x43}`
    - `char*`: “ABCD”

※演習室の計算機のCPUはビッグエンディアン

アドレス	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
…																
0x00010100	41	42	43	44	00	00	00	00	00	00	00	00	00	00	00	00
…																

C言語の変数の裏で番地と型が管理されている

主記憶の仕組みを知らなくても
変数や配列が使えている

ポインタの基礎

ポインタ: 記憶空間を直接読み書きする手段

0x00000028番地のデータを読んでみよう

char（1バイト）として読む

```
// p: 0x00000028番地からchar（1バイト）にアクセスする手段
char *p = (char *)0x00000028;
// pが指している番地（0x00000028）の値を読み込む
v = *p;
```

vに0x1A（= 26）が代入される

アドレス	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00000000	7F	45	4C	46	01	02	01	06	01	00	00	00	00	00	00	00
0x00000010	00	02	00	12	00	00	00	01	00	01	08	90	00	00	00	34
0x00000020	00	00	1A	A4	00	00	01	00	1A	34	00	20	00	05	00	28
0x00000030	00	1A	00	19	00	00	00	06	00	00	00	34	00	01	00	34
0x00000040	00	00	00	00	00	00	00	A0	00	00	00	A0	00	00	00	05
0x00000050	00	00	00	00	00	00	00	03	00	00	00	D4	00	01	00	D4
0x00000060	00	00	00	00	00	00	00	11	00	00	00	11	00	00	00	04
0x00000070	00	00	00	00	00	00	00	01	00	00	00	00	00	01	00	00
…																
…																
…																
0xFFFFFFF0																

short（2バイト）として読む

```
// p: 0x00000028番地からshort（2バイト）にアクセスする手段
short *p = (short *)0x00000028;
// pが指している番地（0x00000028）の値を読み込む
v = *p;
```

vに0x1A34 (= 6708) が代入される

アドレス	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00000000	7F	45	4C	46	01	02	01	06	01	00	00	00	00	00	00	00
0x00000010	00	02	00	12	00	00	00	01	00	01	08	90	00	00	00	34
0x00000020	00	00	1A	A4	00	00	01	00	1A	34	00	20	00	05	00	28
0x00000030	00	1A	00	19	00	00	00	06	00	00	00	34	00	01	00	34
0x00000040	00	00	00	00	00	00	00	A0	00	00	00	A0	00	00	00	05
0x00000050	00	00	00	00	00	00	00	03	00	00	00	D4	00	01	00	D4
0x00000060	00	00	00	00	00	00	00	11	00	00	00	11	00	00	00	04
0x00000070	00	00	00	00	00	00	00	01	00	00	00	00	00	01	00	00
…																
…																
…																
0xFFFFFFF0																

※ビックエンディアンを仮定

int（4バイト）として読む

```
// p: 0x00000028番地からint（4バイト）にアクセスする手段
int *p = (int *)0x00000028;
// pが指している番地（0x00000028）の値を読み込む
v = *p;
```

vに0x1A340020 (= 439615520) が代入される

アドレス	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00000000	7F	45	4C	46	01	02	01	06	01	00	00	00	00	00	00	00
0x00000010	00	02	00	12	00	00	00	01	00	01	08	90	00	00	00	34
0x00000020	00	00	1A	A4	00	00	01	00	1A	34	00	20	00	05	00	28
0x00000030	00	1A	00	19	00	00	00	06	00	00	00	34	00	01	00	34
0x00000040	00	00	00	00	00	00	00	A0	00	00	00	A0	00	00	00	05
0x00000050	00	00	00	00	00	00	00	03	00	00	00	D4	00	01	00	D4
0x00000060	00	00	00	00	00	00	00	11	00	00	00	11	00	00	00	04
0x00000070	00	00	00	00	00	00	00	01	00	00	00	00	00	01	00	00
…																
…																
…																
0xFFFFFFF0																

※ビックエンディアンを仮定

double（8バイト）として読む

```
// p: 0x00000028番地からdouble（8バイト）にアクセスする手段
double *p = (double *)0x00000028;
// pが指している番地（0x00000028）の値を読み込む
v = *p;
```

vに0x1A3400200050028を浮動小数点で解釈したもの（$1.882796 \times 10^{-182}$ ）を代入

アドレス	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00000000	7F	45	4C	46	01	02	01	06	01	00	00	00	00	00	00	00
0x00000010	00	02	00	12	00	00	00	01	00	01	08	90	00	00	00	34
0x00000020	00	00	1A	A4	00	00	01	00	1A	34	00	20	00	05	00	28
0x00000030	00	1A	00	19	00	00	00	06	00	00	00	34	00	01	00	34
0x00000040	00	00	00	00	00	00	00	A0	00	00	00	A0	00	00	00	05
0x00000050	00	00	00	00	00	00	00	03	00	00	00	D4	00	01	00	D4
0x00000060	00	00	00	00	00	00	00	11	00	00	00	11	00	00	00	04
0x00000070	00	00	00	00	00	00	00	01	00	00	00	00	00	01	00	00
…																
…																
…																
0xFFFFFFF0																

※ビックエンディアンを仮定

こんなことも出来ます（voidポインタ）

```
// p: 0x00000028番地を指す手段（アクセス方法は指定しない）
void *p = (void *)0x00000028;
// 読み込み方法が指定されていないので，下のコードはコンパイルできない
v = *p;
// アクセス方法を一時的にintに指定（キャスト）してアクセス
v = *(int *)p;
```

×

○

vに0x1A340020 (= 439615520) が代入される

アドレス	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00000000	7F	45	4C	46	01	02	01	06	01	00	00	00	00	00	00	00
0x00000010	00	02	00	12	00	00	00	01	00	01	08	90	00	00	00	34
0x00000020	00	00	1A	A4	00	00	01	00	1A	34	00	20	00	05	00	28
0x00000030	00	1A	00	19	00	00	00	06	00	00	00	34	00	01	00	34
0x00000040	00	00	00	00	00	00	00	A0	00	00	00	A0	00	00	00	05
0x00000050	00	00	00	00	00	00	00	03	00	00	00	D4	00	01	00	D4
0x00000060	00	00	00	00	00	00	00	11	00	00	00	11	00	00	00	04
0x00000070	00	00	00	00	00	00	00	01	00	00	00	00	00	01	00	00
…																
…																
…																
0xFFFFFFF0																

ポインタ: 番地＋アクセス方法の組

```
char *p = (char *)0x00000028;
```

番地: *pでアクセスしたい番地

pの値 → 0x00000028

アクセス方法: *pとしたときの型

*pの値 → 0x1A （char型）

つまり，ポインタ変数（例えばp）とは

- 基本的には番地を記憶する変数

```
char *p = (char *)0x00000028;
```

  - このときpの値は0x00000028
  - つまりint型の普通の変数と同じと思えばよい

- ただし，以下の特殊機能が用意されている
  - *p: pの番地の値にアクセス
  - p[i]: pからiだけ離れた番地の値にアクセス
  - p+i: pからiだけ離れた番地を計算する

配列の正体

p[i]とp+iについて詳しく見てみよう

char*に対する操作

```
char *p = (char *)0x00000028;
```

```
v = *p;
```

```
++p;
v = *p;
```

p→0x00000029

```
v = p[1];
```

```
v = *(p+2);
```

v → 0x1A　　v → 0x34　　v → 0x34　　v → 0x00

アドレス	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00000000	7F	45	4C	46	01	02	01	06	01	00	00	00	00	00	00	00
0x00000010	00	02	00	12	00	00	00	01	00	01	08	90	00	00	00	34
0x00000020	00	00	1A	A4	00	00	01	00	1A	34	00	20	00	05	00	28
0x00000030	00	1A	00	19	00	00	00	06	00	00	00	34	00	01	00	34
0x00000040	00	00	00	00	00	00	00	A0	00	00	00	A0	00	00	00	05
0x00000050	00	00	00	00	00	00	00	03	00	00	00	D4	00	01	00	D4
0x00000060	00	00	00	00	00	00	00	11	00	00	00	11	00	00	00	04
0x00000070	00	00	00	00	00	00	00	01	00	00	00	00	00	01	00	00
…																
…																
…																
0xFFFFFFF0																

short*に対する操作

```
short *p = (short *)0x00000028;
```

```
v = *p;
```

```
++p;
v = *p;
```

p→0x0000002A

```
v = p[1];
```

```
v = *(p+2);
```

v → 0x1A34

v → 0x0020

v → 0x0020

v → 0x0005

アドレス	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00000000	7F	45	4C	46	01	02	01	06	01	00	00	00	00	00	00	00
0x00000010	00	02	00	12	00	00	00	01	00	01	08	90	00	00	00	34
0x00000020	00	00	1A	A4	00	00	01	00	1A	34	00	20	00	05	00	28
0x00000030	00	1A	00	19	00	00	00	06	00	00	00	34	00	01	00	34
0x00000040	00	00	00	00	00	00	00	A0	00	00	00	A0	00	00	00	05
0x00000050	00	00	00	00	00	00	00	03	00	00	00	D4	00	01	00	D4
0x00000060	00	00	00	00	00	00	00	11	00	00	00	11	00	00	00	04
0x00000070	00	00	00	00	00	00	00	01	00	00	00	00	00	01	00	00
…																
…																
…																
0xFFFFFFF0																

連続したshort型（２バイト）のデータにアクセスしやすいように，
ポインタ変数の加減算の移動単位が自動的に2になる仕様！

int*に対する操作

```
int *p = (int *)0x00000028;
```

```
v = *p;
```

```
++p;
v = *p;
```

```
v = p[1];
```

```
v = *(p+2);
```

p→0x0000002C

v→0x1A340020　　v→0x00050028　　v→0x00050028　　v→0x001A0019

アドレス	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00000000	7F	45	4C	46	01	02	01	06	01	00	00	00	00	00	00	00
0x00000010	00	02	00	12	00	00	00	01	00	01	08	90	00	00	00	34
0x00000020	00	00	1A	A4	00	00	01	00	1A	34	00	20	00	05	00	28
0x00000030	00	1A	00	19	00	00	00	06	00	00	00	34	00	01	00	34
0x00000040	00	00	00	00	00	00	00	A0	00	00	00	A0	00	00	00	05
0x00000050	00	00	00	00	00	00	00	03	00	00	00	D4	00	01	00	D4
0x00000060	00	00	00	00	00	00	00	11	00	00	00	11	00	00	00	04
0x00000070	00	00	00	00	00	00	00	01	00	00	00	00	00	01	00	00
…																
…																
…																
0xFFFFFFF0																

連続したint型（４バイト）のデータにアクセスしやすいように，
ポインタ変数の加減算の移動単位が自動的に4になる仕様！

ポインタの番地に対する加減算

```
int *p = (int *)0x00000028;
```

⇨ ++p; pの値 → 0x0000002C

⇨ --p; pの値 → 0x00000024

⇨ p += 2; pの値 → 0x00000030

C言語の仕様により，自動的にポインタの型を考慮し，加減算が実行される

配列とポインタの関係

```
int *p = (int *)0x00000028;
```

⇨ p[0]; 0x00000028番地の値

⇨ p[1]; 0x0000002C番地の値

⇨ *(p+1); 0x0000002C番地の値

p[i] == *(p+i)

配列は加減算付きのポインタへのアクセス

ポインタの実際

```
char *p = (char *)0x00000018;
```

みたいなコードは最近見かけない

実際に実行すると セグメンテーション フォルトが発生

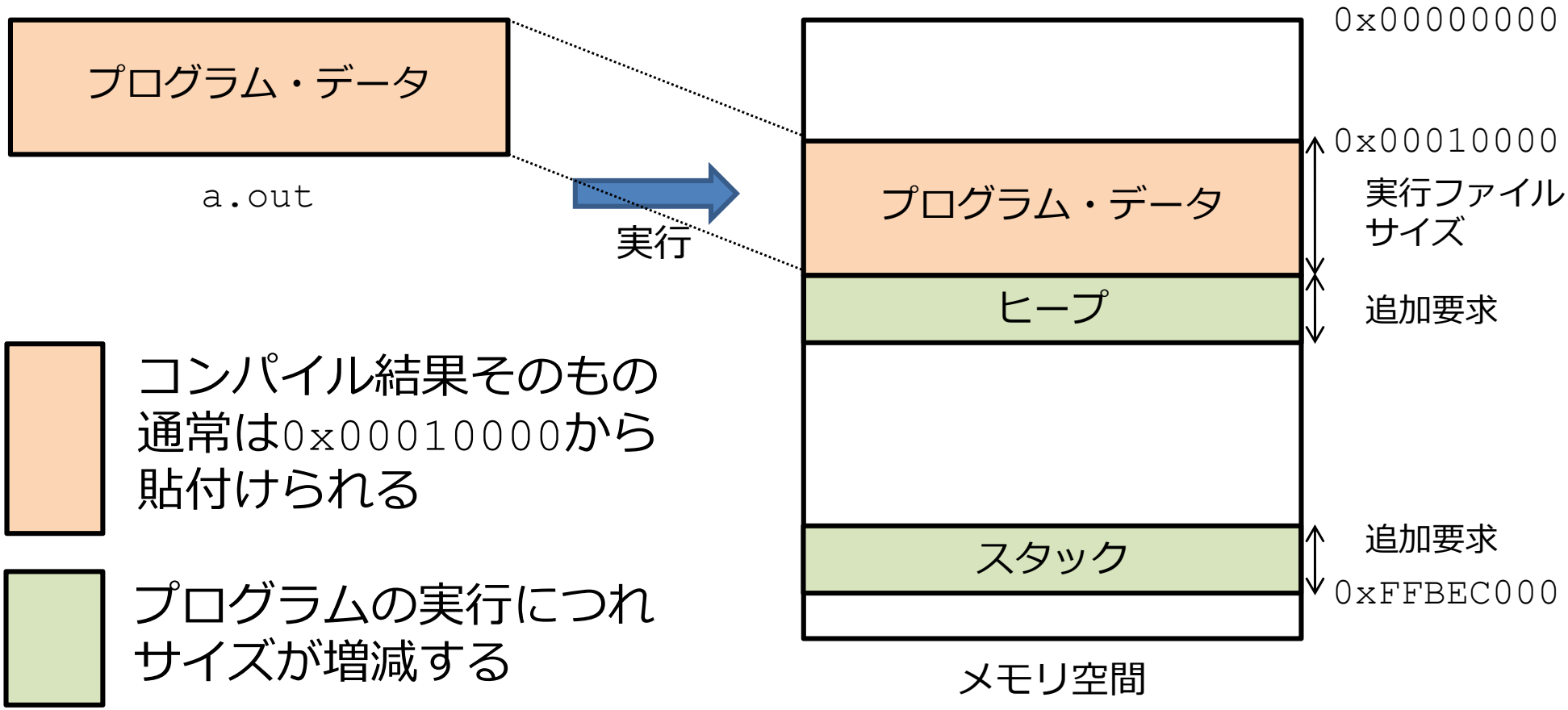
そっか、ぬるぽか！

何故か？

プログラムはＯＳから割り当てられた番地のみ利用できる

プログラムがどの
番地を利用できるか
実行するまで不明

プログラムを実行する流れ



コンパイル結果そのもの
通常は0x00010000から
貼付けられる

プログラムの実行につれ
サイズが増減する

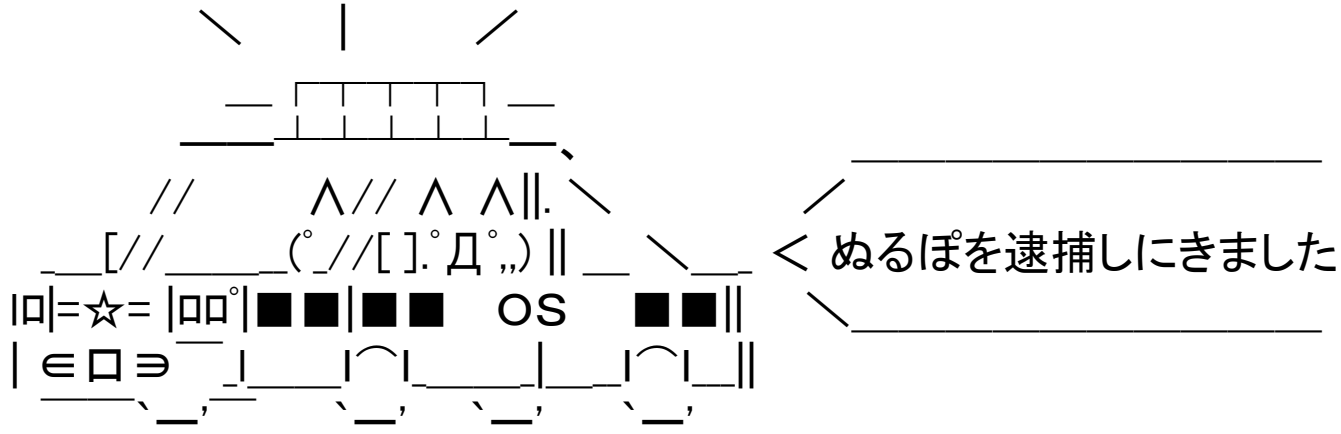
プログラムがアクセスして
はいけない領域

※32ビットSPARC上で動作する
Solarisの場合
※状況に応じてアドレスが変わる

参考: プログラムの読み込み (http://docs.oracle.com/cd/E19253-01/819-0391/chapter6-34713/index.html)

セグメンテーション違反 (Segmentation Fault)

≒ 許可されない（予期せぬ）番地への読み書き（バグ）



つまり

ポインタの番地を決めるのはＯＳやコンパイラの仕事

ポインタの番地を直接指定する機会は殆ど無い

ポインタの番地を正しくセットしないと Segmentation Fault

C言語における ポインタの 使い道

C言語におけるポインタの使い道

1. 配列
   - 配列とポインタは等価
2. 文字列
   - char型の文字の特殊な配列として表現される
3. 関数の引数の参照渡し
   - 引数の変数の値を関数内で更新して返す場合
4. メモリ領域の動的な確保
   - 配列や構造体を必要なサイズだけ確保する
5. 関数ポインタ
   - 高等テクニックなので説明は省略

①配列としてのポインタ

配列を宣言するとどうなるか

```
int data[6] = {0,1,1,0,1,0};
```

- コンパイラは，以下の処理を自動で行う
  - 6 (個) × 4 (byte/int) = 24 (byte) の記憶領域を予約

アドレス	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
…																
0x00010220	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD
0x00010230	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD
0x00010240	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD
0x00010250	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD
0x00010260	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD
…																
…																
0xFFFFFFF0																

配列を宣言するとどうなるか

```
int data[6] = {0,1,1,0,1,0};
```

- コンパイラは，以下の処理を自動で行う
  - 6 (個) × 4 (byte/int) = 24 (byte) の記憶領域を予約
  - （指定された初期化値を確保した記憶領域にセット）
  - メモリ領域の先頭の番地をポインタ変数dataにセット
    - この例では，`int *data = (int *)0x00010234;`

アドレス	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
…																
0x00010220	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD
0x00010230	CD	CD	CD	CD	00	00	00	00	00	00	00	01	00	00	00	01
0x00010240	00	00	00	00	00	00	00	01	00	00	00	00	CD	CD	CD	CD
0x00010250	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD
0x00010260	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD
…																
…																
0xFFFFFFF0																

このように
配列のメモリ管理は
自動で行われる

配列がポインタであることを意識せずに `data[i]` と書ける

②文字列を表現するポインタ

文字列を宣言するとどうなるか

```
char *msg = "hello";
```

- コンパイラは文字 (char) 配列に自動で変換

```
char str[] = {'h','e','l','l','o',0};
           = {0x68,0x65,0x6C,0x6C,0x6F,0};
```

  - 文字列の終端を示す0が追加される（文字列特有の仕様）
  - この例では，`msg = (char *)0x00010250;`

アドレス	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
…																
0x00010220	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD
0x00010230	CD	CD	CD	CD	00	00	00	00	00	00	00	01	00	00	00	01
0x00010240	00	00	00	00	00	00	00	01	00	00	00	00	CD	CD	CD	CD
0x00010250	68	65	6C	6C	6F	00	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD
0x00010260	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD	CD
…																
…																
0xFFFFFFF0																

実は""で文字列を使う度に，裏で文字配列が生成される

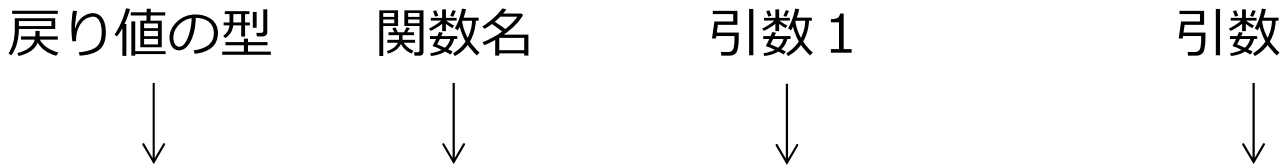
```
printf("hello!");
```

でも文字の配列が
メモリ上に確保される

③関数への参照渡し

関数とは

- C 言語の関数
  - 0 個以上の値を引数として受け取る
  - 0 個または 1 個の値を戻り値として返す




```
double func(double x, double y)
{
    return sqrt(x * x + y * y);
}
```

func(x, y)の戻り値: $\sqrt{x^2 + y^2}$

  - 使い方

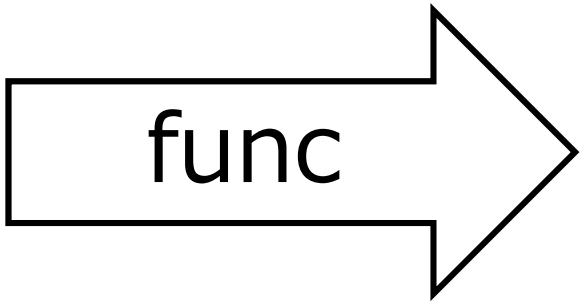
```
d = func(3, 4);
```

```
if (func(x, y) < 1)
```

関数から２つ以上の値を返すには？

(x, y)

直交座標系



(r, θ)

極座標系

陥りやすいダメな例（cf. 確認問題）

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

void func(double x, double y, double r, double th)
{
    r = sqrt(x * x + y * y);
    th = atan2(y, x);
}

int main()
{
    double x = 1, y = 1, r = 0, th = 0;
    func(x, y, r, th);
    printf("(r, th) = (%f, %f)¥n", r, th);
}
```

変数rとthはfunc関数内でのみ有効．main関数側に変更が反映されない！

```
$ ./a.out
(r, th) = (0.00000, 0.00000)
```

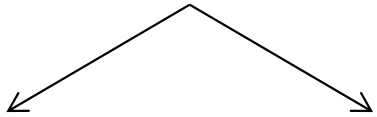
関数の呼び出し元の変数の値を更新する

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

void func(double x, double y, double *r, double *th)
{
    *r = sqrt(x * x + y * y);
    *th = atan2(y, x);
}

int main()
{
    double x = 1, y = 1, r = 0, th = 0;
    func(x, y, &r, &th);
    printf("(r, th) = (%f, %f)¥n", r, th);
}
```

更新したい変数をポインタに



ポインタ変数が指している値を更新する ←③

更新したい変数の番地を渡す

←① ←② ←④

```
$ ./a.out
(r, th) = (1.4142, 0.7854)
```

どんなトリックなのか（1/3）

①: `double x = 1, y = 1, r = 0, th = 0;`

コンパイラは，main関数内でdouble型（８バイト）の変数，x，y，r，thの値を保持するメモリを予約

```
1.0: 0x3FF0000000000000
0.0: 0x0000000000000000
```

←― ※浮動小数点のややこしい仕様です．気にしないで下さい．

②: `func(x, y, &r, &th);`

```
func(
    0x3FF0000000000000,0x3FF0000000000000,
    0x00010270, 0x00010278);
```

※対比のため，浮動小数点を無理矢理8バイトで表記しています．

アドレス	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
…	x								y							
0x00010260	3F	F0	00	00	00	00	00	00	3F	F0	00	00	00	00	00	00
0x00010270	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
…	r								th							

どんなトリックなのか（2/3）

③: `*r = sqrt(x * x + y * y);`
`*th = atan2(y, x);`

$\sqrt{1^2 + 1^2} = 1.4142\ldots$
$\tan^{-1} 1 = \pi/4 = 0.7854\ldots$

②によりmain関数の変数r,thのメモリ番地は,
r=0x00010270, th=0x00010278と伝えられている

`*r = 1.4142…`
`*th = 0.7854…`

⟺

00010270番地に1.4142…
00010278番地に0.7854…

```
1.4142…: 0x3FF6A09E667F3BCD
0.7854…: 0x3FE921FB54442D18
```

※浮動小数点のややこしい仕様です．気にしないで下さい．

アドレス	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
…	x								y							
0x00010260	3F	F0	00	00	00	00	00	00	3F	F0	00	00	00	00	00	00
0x00010270	3F	F6	A0	9E	66	7F	3B	CD	3F	E9	21	FB	54	44	2D	18
…	r								th							

どんなトリックなのか（3/3）

変数r,thの値を格納しているメモリの内容が書き換えられたので，r = 1.4142…, th = 0.7854…

つまり，main関数が所有している変数の値を，
func関数から直接変更していたことになる

④: `printf("(r, th) = (%f, %f)¥n", r, th);`

```
$ ./a.out
(r, th) = (1.4142, 0.7854)
```

アドレス	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
…	x								y							
0x00010260	3F	F0	00	00	00	00	00	00	3F	F0	00	00	00	00	00	00
0x00010270	3F	F6	A0	9E	66	7F	3B	CD	3F	E9	21	FB	54	44	2D	18
…	r								th							

関数内で呼び出し元の変数の値を更新するには，ポインタ化

参照渡しの有名な例

scanf関数は複数の値の同時読み込みが可能になっている

このような仕様のため

```
scanf("%d %d", &x, &y);
```

のように&をつける

④メモリ領域の動的な確保

メモリを使うには，
OSに割り当てて
もらう必要がある

プログラム執筆時点で要素数が確定している配列・文字列の確保はコンパイラの仕事

プログラムを書いた時点で必要な要素数が分からない場合，手動でメモリ確保する必要

例えば・・・

ワードプロセッサに何文字入力されるか分からない

受信するメールの大きさが分からない

こういう時は・・・

400文字格納できる文字列用のメモリを下さい

「原稿用紙をもう一枚下さい！」

3,487,394バイトの添付ファイルを保存・閲覧するためのメモリを下さい

メモリ管理関数の抜粋

- `void *malloc(size_t size);`
  - sizeバイトの連続したメモリ領域を確保し，そのメモリ領域の先頭の番地を返す
- `void *calloc(size_t n, size_t size);`
  - メモリ領域を0に初期化しながら確保（詳細は省略）
- `void *realloc(void *p, size_t size);`
  - `malloc`や`calloc`で確保されたメモリ領域（先頭の番地がp）のサイズをsizeに拡張・縮小し，新しいメモリ領域の先頭の番地を返す
- `char *strdup(const char *str);`
  - 文字列strを格納できるメモリ領域を確保した後，文字列strを新しいメモリ領域にコピーし，
- `void free(void *p);`
  - 上に挙げた関数で確保されたメモリを解放する

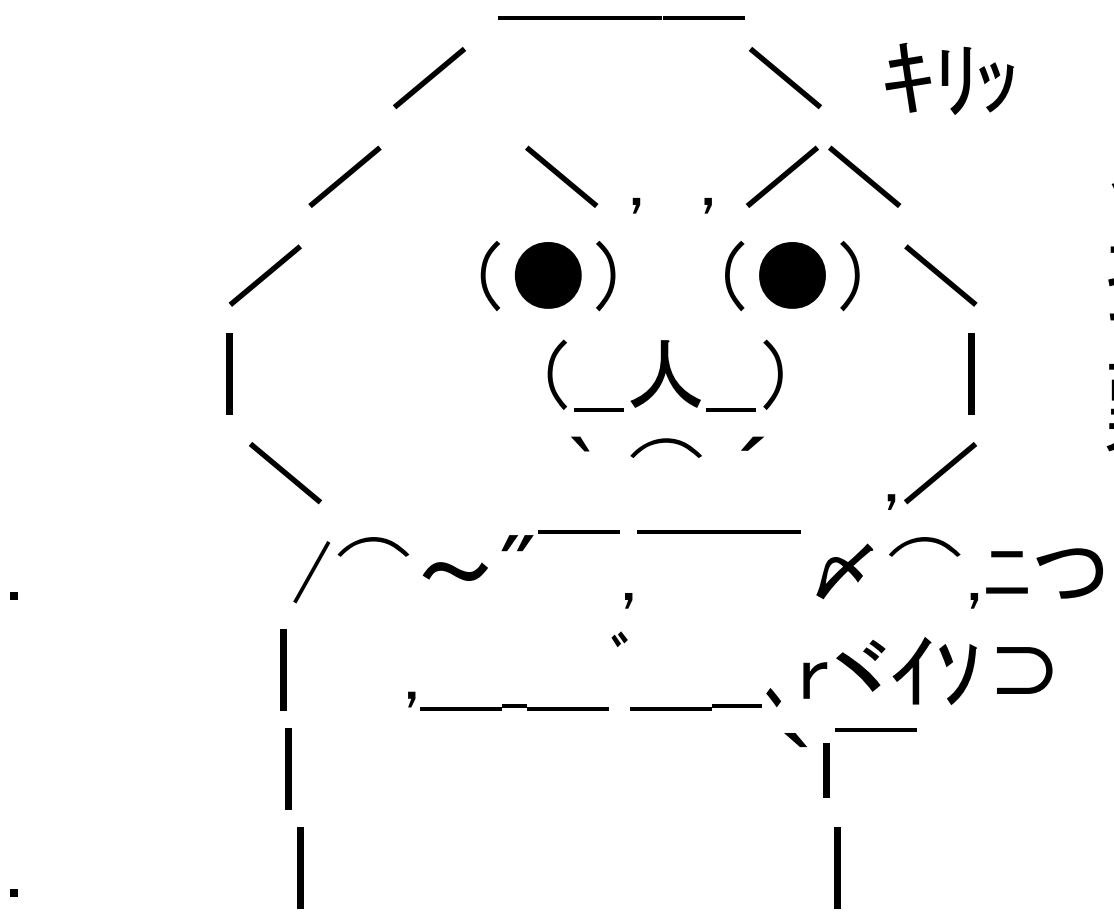
400文字格納できる文字列用のメモリを下さい

```
char *p = (char *)malloc(400);
```

メモリが不要になったのでお返しします

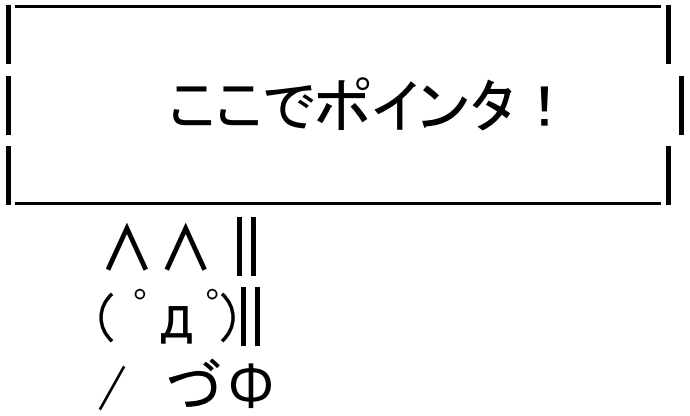
```
free(p);
```

まとめ



メモリのイメージが掴めればポインタは難しくない！

ポインタの用途は限られているので実例を見て慣れよ



今回扱わなかった内容

- malloc/freeの実例
- 二次元配列
- ポインタのポインタ
- constなどの修飾子
- 関数ポインタ
- 構造体のポインタ