C 言語の補習(2)

- ポインタの基礎

ポインタ(pointer)

- メモリ上の番地を使って変数にアクセスするもののこと
  - 変数の読み書きは，その変数の名前を使うのが最も簡単な方法
  - ポインタでは，名前の代わりに番地(格納場所)を使う

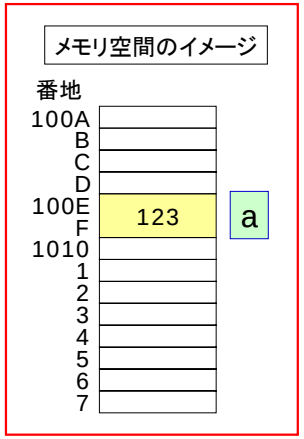
簡単な例

- 次のプログラムでは，変数 a に 123 が代入される

① `int a;`
② `a = 123;`

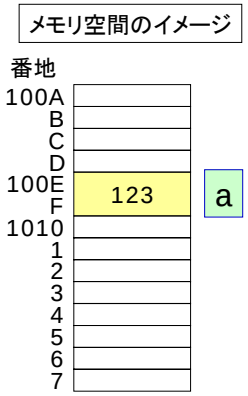
① int 型の変数をメモリ上に確保する．
右の例では 100E 番地から2バイト分の領域で，これを「a」と名付ける．

② 「a」に対して 123 を代入する．



名前の代わりに番地を

- この例の場合，変数 a は 100E 番地の変数という見方もできる



- 人間に例えると
  - 「愛大 太郎」
  - ⇕
  - 「松山市文京町3番に住んでいる人」

これがポインタの考え方

番地の調べ方

- 変数が格納される領域の番地は自動的に割り当てられる
  - 常に固定されているわけではない
- 変数名の前に＆(アンパサンド)を付けると，その番地を調べることができる

```
&a
```

ただ，これだけだと使いようがない．．．
番地を記録するための変数が必要

ポインタ変数

- 番地を格納するための変数
- 名前の前に＊(アスタリスク)を付けて宣言

```
int  a;
int* p;

a = 123;
p = &a;
```

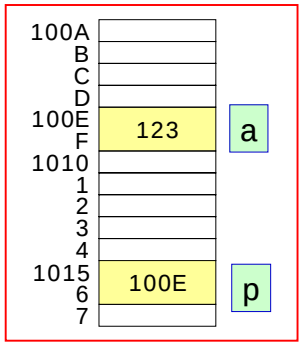
```
int*   p;
int   *p;
```
の2種類の書き方があるがどちらも可

変数 p には変数 a の番地が格納される

改めてメモリ空間のイメージ図

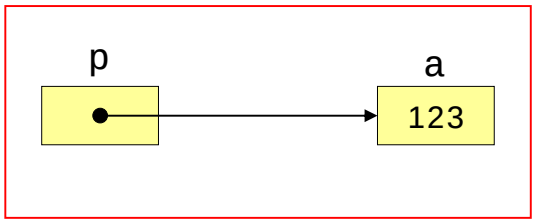
- int 変数 a とポインタ変数 p のイメージ
  - p にも自動的にその領域が割り当てられる
  - p の内容は a が置かれている番地となっている

```
int  a;
int* p;
a = 123;
p = &a;
```



よく使われるイメージ図

- 変数 a の格納番地がポインタ変数 p に代入されている場合



p の内容は a の番地(場所)を指しているという意味

※もともと「ポインタ」という言葉は「指し示すもの」という意味

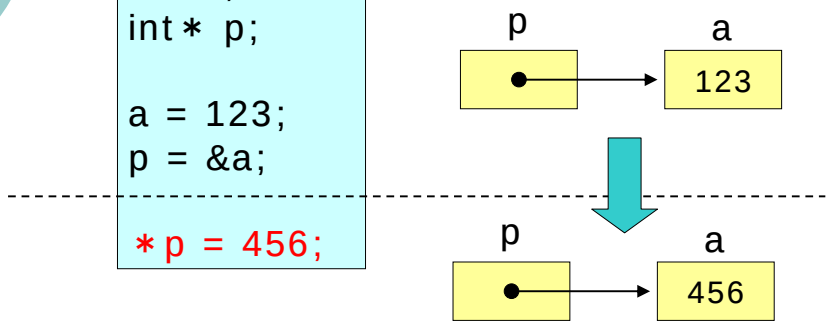
ポインタを使ったアクセス

- ポインタ変数の前に＊を付けると，参照先の内容になる
- つまり

ポインタを使ったアクセス例

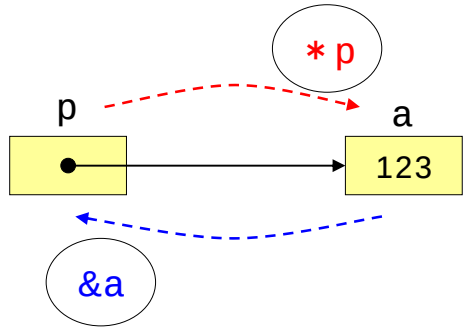
- 簡単な例

```
int  a;
int＊ p;

a = 123;
p = &a;

＊p = 456;
```



＊ と ＆ が持つ働き

- ＊は矢印の向きに従ってたどっていく
- ＆は矢印の向きに逆らってたどっていく



ポインタ変数の型

- 参照先の変数の型(int, double 等)を明確にしておく必要がある

```
int＊   p;
```

ポインタ変数 p は，
int 型変数へのアクセス専用となる

見方を変えると，＊p が int 型変数を参照することになる

```
int   ＊p;
```

メモリ管理の注意点

- ポインタ変数を宣言しただけでは，肝心のデータ領域が存在しない！

```
int* p;
*p = 3;
```
← 間違い！

この時点では，変数 p には何が入っているか分からない

つまり，メモリ上のどこを指しているかは不明なのに，
その指定先に「3」を代入しようとしている！

メモリエラーを引き起こす
Segmentation fault（セグメントエラー）

正しいメモリ管理（1）

- 自動変数（変数宣言されたもの）を使う

```
int* p;
int a;
p = &a;
*p = 3;
```


正しいメモリ管理（1）

- 自動変数（変数宣言されたもの）を使う

```
int* p;
int a;
p = &a;
*p = 3;
```

p

？？

正しいメモリ管理（1）

- 自動変数（変数宣言されたもの）を使う

```
int* p;
int a;
p = &a;
*p = 3;
```

p

？？

a

？？

正しいメモリ管理(1)

- 自動変数(変数宣言されたもの)を使う

```
int* p;
int  a;
p = &a;
*p =  3;
```

正しいメモリ管理(1)

- 自動変数(変数宣言されたもの)を使う

```
int* p;
int  a;
p = &a;
*p =  3;
```

正しいメモリ管理(2)

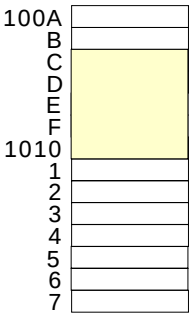
- malloc 関数により自分でメモリを確保する

malloc(バイト数)

指定されたバイト(byte)数だけ
メモリが確保される.
戻り値は, その先頭番地.

正しいメモリ管理(2)

(例) malloc(5) を実行



空き領域から適当に
(連続した)5バイト分
の領域が確保される

malloc(5) の実行が
正しく行われると, その
先頭番地(この例では
100C)が戻り値になる.

正しいメモリ管理(2)

- int や double がそれぞれ何バイト必要なのかを調べる: **sizeof(...)** を使う
  - 例: sizeof(int), sizeof(double)
- int 型変数のためのメモリ確保は

```
malloc( sizeof(int) )
```


正しいメモリ管理(2)

- int 型変数の確保とポインタによるアクセス

```
int* p;
p = (int*)malloc( sizeof(int) );
*p = 3;
```

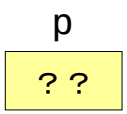
malloc の戻り値は番地であるが,
参照先が int 型なのかそれとも他の型(double 等)なのかがこのままでは不明

明示的に int 型へのポインタ(int*)であることを指定している. これをキャストという.

正しいメモリ管理(2)

- malloc を使う

```
int* p;
p = (int*)malloc( sizeof(int) );
*p = 3;
```



正しいメモリ管理(2)

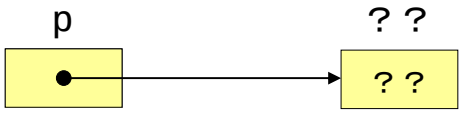
- malloc を使う

```
int* p;
p = (int*)malloc( sizeof(int) );
*p = 3;
```

正しいメモリ管理(2)

- malloc を使う

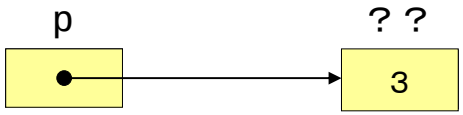
```
int* p;
p = (int*)malloc( sizeof(int) );
*p = 3;
```



正しいメモリ管理(2)

- malloc を使う

```
int* p;
p = (int*)malloc( sizeof(int) );
*p = 3;
```



malloc を使った場合の注意点

- 自分で確保したメモリは，自分で解放する
- **free 関数**を使う

```
int* p;
p = (int*)malloc( sizeof(int) );
*p = 3;
.....
free(p);
```


まとめ

- ポインタは番地を使って変数にアクセスする
- 番地を調べる: &変数名　（例） &a
- アクセスする : *ポインタ　（例）*p
- 自分でメモリを確保する: malloc

```
型名* p = (型名*)malloc( sizeof(型名) );
```

（例） `int* p = (int*)malloc( sizeof(int) );`

最後は自分でメモリを解放する: `free(p);`