2024年　6月　13 日

ポインタについて　　５つの記号を理解　　＊　　＆　　[]　　.　　->

[]は配列に、　. と ->　は構造体に用いる。

おなじ意味になる表記がいくつか有り、極論すると　[]　と　->　等は使わなくてもプログラムできる。（使ったほうが簡潔に書ける）

はじめに変数の型とサイズの復習

整数型　int 4 byte　　単精度浮動小数点実数型　float 4 byte 倍精度浮動小数点実数型 double 8 byte

文字型　char 1 byte　長い整数型　long int 8 byte　それぞれのポインタ型　8 byte

printf文でそれぞれの型の変数を出力する際の変換子

整数型　int %d　　単精度浮動小数点実数型 float　%f　倍精度浮動小数点実数型 double %lf

文字型　char %c　倍精度整数型　long int %ld　それぞれのポインタ型　%p　　文字列型　%s

符号なし整数型　unsigned int %u

16進数について

printf 文の　%p　変換でポインタ変数のアドレス値を16進数で出力できる

16進数の１桁は　0 ~ f　の16種の文字で　0~15の16種の数値を表せる

0~9の数字に加えて、a:10, b:11, c:12, d:13, e:14, f:15 のアルファベットを使う

16進数の２桁は　00 ~ ff　で16x16 = 256：　0~255　の256種類の数値を表せる

つまり、16進数の２桁が、ちょうど１バイトに相当する

PC上で、

```
#include <stdio.h>
int main(void)
{
int i=0;
printf("%p \n",&i);
}
```

として、コンパイルし、３回実行してみて出力値を下に記入しなさい。

例：　　0x7ffc55df3138　　１回目：　　２回目：　　３回目：

単に34などと記述した場合十進数の三十四なのか16進数なのか区別できないので16進数には左端に0x　（ゼロエックス）をつけて、0x34と表し、単に34と書いた場合には十進数であるとみなす。

16進数について、の加減を考えます、ここでは下１桁に1,4,8 を加えた場合の差分がわかればokとします

0x3130　+　4　=

0x3134　+　4　=

0x3138　+　4　=

0x313c　+　4　=

以下の16進数アドレス間の差が読み取れるでしょうか

0xb0c8

0xb0cc

0xb0e0

0xb0e4

0xb0e8

変数宣言	値を格納するために確保されるメモリ
`int i;`	4
`int j;`	4
`float f;`	4
`double d;`	8
`char c;`	1
`int a[3];`	12
`char s[20];`	20

これらの変数宣言により、i,j,f,d,c,a,s　などの変数名に、ひとつのアドレスが割り当てられ、

① **<u>＆変数名、でそのアドレス値を参照できる</u>。**

同時に、変数の値を入れる場所として、それぞれの変数の方に応じたサイズ（右端に表記）のメモリが上記アドレスの場所に確保され、単に、変数名、でその値が参照できる。

② **<u>それぞれの型について、型名の後に＊をつけることでポインタ変数を宣言できる</u>**。ポインタ変数にはアドレス値を代入できる。

`int*  pi;`	8
`float* pf;`	8
`double* pd;`	8
`char* pc;`	8

このとき 変数名 はアドレス値を示し、　＊変数名 はそのアドレスに入っている変数の値を示す。
上の1つ目の例では、pi はアドレス値、 *pi はそのアドレスに履いている整数の値。

③ **<u>アドレス値を保持したポインタ変数の前に＊を付けると、そのアドレスにある変数の値を参照できる。</u>**
そこにある変数の型は、ポインタ変数宣言時の型、である。

④ **<u>ポインタ変数に n を加えると、ポインタの型のサイズ、に n をかけた数だけ ポインタ変数の値が増える。</u>**
int* pi; で宣言した pi にアドレス値 10 が入っているとすると pi ++; で piの値は14になる。

⑤ **<u>配列宣言を行うと、[]を付けない変数名に、１つ目の要素（変数名[0]）の開始アドレスが割り振られる。</u>**
int　ai[3]; と配列を宣言したとき ai は int* ai; としたときとおなじアドレス値のはいるポインタ変数となる。整数４バイト３つ分の12バイトaiに確保されたメモリ領域の最初のアドレスを ai は指し、n番目の要素の開始アドレスは　変数名　＋　n * sizeof（int） で与えられる。プログラム上では、上記ルール④により * sizeof（int）は省略される。ai+2でaiに8を加えた、３番目の要素のアドレス値、になる。

5' **<u>配列参照時の [] はポインタ表記で置き換えられる。　a[n] = *(a + n);</u>**
ここでもルール④により実際のアドレス値としては、nに型のサイズを掛けたものが加えられている。ai+0が1番目、ai*1が2番目のアドレス値（aiに４を足した値）ai+2とすれば三番目の要素のアドレス値(aiに８を足した値）になり、*(ai+2)で３番目の要素の整数値を参照できる。これは、 ai[2]と等価。

⑥ **<u>構造体の要素を取り出す場合には構造体変数のあとに、　.　ピリオドをつけてから要素名を記す。</u>**
以下の構造体city_infoについて city_info cities[3]; と要素３つ分を宣言すれば、
２番めの要素の人口は、cities[1].pop で参照できる。

```
typedef struct city_info
 {
 char  *name;              // 市の名前　ポインタ宣言でそれぞれの長さ分をファイルを見てから確保
 int   pop;               // 人口
 float area;              // 面積（平方キロメートル）
 float population_density;       // 人口密度
 } city_info;
```

⑦ **構造体ポインタ変数の指す位置にある構造体の要素を取り出すには、構造体ポインタ変数の後に、-> アロー演算子を付けてから要素名を記す。**

上記のcitiesについて、１つ目の要素の面積を参照するには、cities->area とする。
また、２番めの要素の人口については、２番めの構造体の先頭アドレスが cities + 1 となるので、
(cities + 1)->pop で参照できる。これはcities[1].popと等価。

7' **アロー演算子は ＊を使って構造体変数を参照することで ． に置き換えられる。 p -> pop = (*p).pop;**

括弧の位置に注意
上記の ２番目の要素の人口は、(*(cities + 1)).pop とも表せる。
この場合、おそらく一番簡潔な表記は cities[1].pop

ポインタ宣言 ＋ malloc では、前ページの 配列宣言とほぼ同じメモリ構造を作ることができる。
記法としては、配列宣言のほうが簡単なのに malloc を使う理由としては、

① malloc を使えば要素数、3 のところに変数を用いることができる。配列宣言の要素数には変数を入れられない

② malloc を使えばヒープ領域にメモリが割り当てられ、より大きな配列（多くの要素数）を宣言できる

演算子の優先順位

自信がない場合は明示的にカッコを使う

優先度	演算子	機能	結合の向き
1	()	関数呼び出し	左から右
	[]	配列の要素	
	->	ポインタからの構造体メンバアクセス	
	.	構造体メンバアクセス	
	++	後置インクリメント	
	–	後置デクリメント	
	(type) {…}	複合リテラル	
2	!	論理否定	右から左
	~	ビット否定	
	++	前置インクリメント	
	–	前置デクリメント	
	+	符号	
	-	符号を反転させる	
	*	ポインタの間接参照	
	&	メモリアドレス	
	sizeof	変数や型の大きさを取得	
	_Alignof	(C11) アラインメント値を取得	
3	(型名)	キャスト	右から左
4	*	乗算	左から右
	/	除算	
	%	剰余	
5	+	加算	左から右
	-	減算	
6	<<	左シフト	左から右
	>>	右シフト	
7	<	左の方が小さい	左から右
	<=	左が右以下	
	>	左の方が大きい	
	>=	左が右以上	
8	==	等しい	左から右
	!=	等しくない	
9	&	ビット積	左から右
10	^	ビット排他的論理和	左から右
11	|	ビット和	左から右
12	&&	論理積	左から右
13	||	論理和	左から右
14	? :	条件演算子	右から左
15	=	代入	右から左
	+=	加算代入	
	-=	減算代入	
	*=	乗算代入	
	/=	除算代入	
	%=	剰余代入	
	<<=	左シフト代入	
	>>=	右シフト代入	
	&=	ビット積代入	
	|=	ビット和代入	
	^=	ビット排他的論理和代入	
16	,	カンマ	左から右

講義の説明で行うように、変数宣言をした場合のメモリ内の様子を右側に描いてみてください

```
int  i = 5;
```

```
int  *p;
```

```
p = &i;
```

```
*p = 3;
```

```
double ad[3];
```

```
typedef struct city_info
 {
 char name[20];
 int pop;
 double area;
 } city_info;

city_info cities[2]
```

```
double* pd;
pd = (double *)malloc(sizeof(double) * 3);
```

double ad[3]; と見比べてみよう

```
city_info* pcities;
pcities = (city_info *)malloc(sizeof(city_info) * 2);
```

前ページの cities[2]; と見比べてみよう