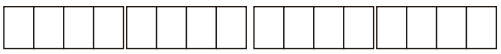
コンピュータの基礎：　変数とポインタ　　2021年5月17日

変数

計算機の中のデータはすべて、０と１の並び（２進数）で表されていることを以前話しました。文字についてはASCIIと呼ばれる文字コードにより128種類の英数字などの文字を、１バイト（=８bit）の情報量で表しています。大文字の A には十進数の65 （16進数の41）が割り当てられていてこれは２進数では、0100 0001になります（４桁ごとにスペースをいれました）。また数字の０と記号？にはそれぞれ十進数の 48 と 63 （それぞれ16進では30と3f）が割り当てられていて、これは２進数では 0011 0000 と 0011 1111 になります。したがって、計算機上で A0? という文字列は 0100 0001 0011 0000 0011 1111 という、24bit （＝３バイト）の0/1の並びで表現されることになります。

練習問題１

小文字の a は十進数の 97、16進数の61、 記号の @は十進数の64、16進数の40です。 文字列 @a の計算機の内部表現を 0 と 1 の並びで記しなさい



計算機上の整数値は、数値を２進数で表し、正負符号の情報を入れます。最も単純に考えると１つのビットを正か負に割り当てれば良さそうですが、実際には補数と呼ばれる表現を取ることが殆どですが、ここでは、簡単のため、左端のビットを正（0）か負（１）とし、残りのビットを２進数として考えてみましょう。現在使われる多くの環境では整数値の記憶のために４バイト（32bit）を取ります。数字の１は、0000 0000 0000 0000 0000 0000 0000 0001 と表せます。数字の -1は、1000 0000 0000 0000 0000 0000 0000 0001 となります。小さな数値を表すにはだいぶ無駄な感じがしますね。この記述の仕方では、表現できる数値の最大値が決まってしまうのがわかるでしょうか？ 最も大きな整数値は、0111 1111 1111 1111 1111 1111 1111 1111 になります。これは２の３１乗から1を引いた数になるので、21億4748万3647 となります。符号なしの整数の場合には４バイトで表せる最大の値は、2の32乗-1で、42億9496万7295となります。

練習問題２

初期のコンピューターでは記憶領域が今ほど豊富に使えなかったため、整数には２バイトを割り当てるのが普通でした。この場合、表せる最大の数値を、符号あり、符号なしについてそれぞれ計算してください。

符号あり　＿＿＿＿＿＿＿＿＿＿＿＿　符号なし　＿＿＿＿＿＿＿＿＿＿＿＿

Processingや、C言語でプログラムを書く時には文字や、数値を入れる**変数**という入れ物を作ります。この時、入れるものによって型、という区別をします。たとえば整数値の入れ物として a という変数 をひとつ作るときには、

```
int  a;
```

という変数宣言を行います。 int （integer = 整数）というのが整数を表す**型**です。こうして宣言（作成）した変数に、

```
a = 1;
```

のようにして数値を入れることが出来ます。さらに入った数値を操作することで様々な計算が出来ます。
同様に、文字を一つ入れる入れ物 c をひとつ作るためには、

```
char  c;
```

という変数宣言を行います。char （character = 文字）というのが整数を表す**型**です。
この時、変数の型によって、必要な入れ物のサイズが変わってきます。文字(char)なら１バイトですし、整数（int）なら4バイトになります。C言語ではその他にも、浮動小数点（単精度）実数（float: 4バイト）、倍精度実数（double: ８バイト）といった変数の型をよく使うので、これらのサイズと内部表現を頭に入れておきましょう。
計算機の、データを記憶する場所としてプログラムから操作できる場所に メインメモリー があります。現在使われているPCでは、通常、4GB、8GBや16GBといったものが主流です。８GBとは、８ギガバイト、のことでギガとは３桁ご

とに増える、キロ（x1000）、メガ（x1000,000）、ギガ（1000,000,000）の係数で、９桁すなわち10億であり、８Gバイトは（およそ）80億バイトになります。

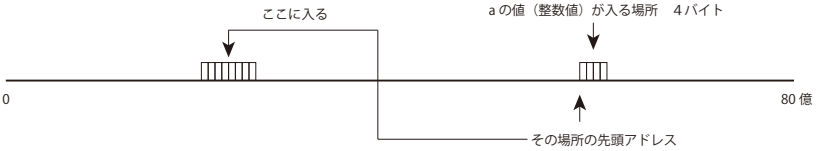
ポインタ

先程の変数宣言に戻り、整数型の変数をひとつ宣言するときを考えます。

```
int a;
```

と宣言すると、整数値を入れる4バイトを記憶する場所が、80億バイトあるメモリのどこかに確保されます。**「確保する」と言うのはこの場所は、a という変数を記憶するために使うので他の変数などが使わないようにして下さいね、と宣言することで、この場所が保護されることになります。**

アドレス、というのはこの記憶場所が、0から80億の何バイト目に位置するかという数値です。先程、計算したように4バイトの符号なし整数で表現できる数値の上限は約42億なので、４ギガバイトを超えるメモリを持つ計算機のアドレスは、４バイトの整数値で表せない場合がでてくるため、メモリ上の任意の場所のアドレスを保存するためには８バイトの**ポインタ**と呼ばれる記憶領域を確保します。



ここまでをもう一度、整理すると、int a; という整数型の変数を宣言すると。メモリ上に、aの整数値が入る４バイトと、a を記憶するその4バイトのメモリ上の位置（アドレス）を記憶するためのポインタと呼ばれる８バイトの場所、が確保され、4バイトの先頭アドレスがその中に書き込まれます。数字の１を記憶するために、合計12バイト（96ビット）も記憶場所を確保するのはいかにも無駄なようですが、様々な理由でそうなっています。ここまでの説明だと、「では、ポインタの先頭アドレスは考慮しなくて良いのか」と思われるかもしれません。宣言した変数名は直接ポインターの先頭アドレスに対応していて、コンピューター（プログラム）は、変数 a の値を操作する時に、まずポインタを見て、その中身が指している場所にある値を**変数 aの値**、として参照する、と考えてください。

ここからが少しややこしいのですが、int　a; と宣言した時、aの値は a で参照でき、aのアドレスを示すポインタは &a で取り出すことが出来ます（実際に数値を見ることは少ないですが、この値を操作することはあります）。&aの使い方を一つ考えてみます。ポインタ変数だけの宣言ということが出来て、たとえば、

```
int *b;
```

とすると、 整数値が入る4バイトは出来ずに、アドレスを入れる８バイトの場所だけが確保されます。先程の int a; と違うのは、a の場合には変数値の４バイトが確保されるのでその先頭アドレスが８バイトのポインタ変数にすぐに書き込まれたのに対し、int *b; ではポインタ変数だけが確保されたので、その値は「未定」となります。

アドレスは、メモリが８GBなら80億以上の数値を表す必要があるため、変数値のサイズは型によって変わりますが、ポインタのサイズはどの型でも同じになります。

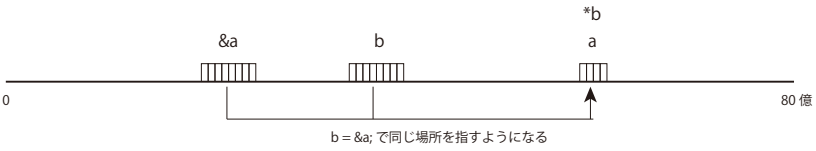
練習問題３

文字型の変数 c を、 char　c; の様に宣言した時に、メモリ上に確保される記憶領域は全部で何バイトでしょう？

ここで、

```
int a;
a = 1;
int *b;
b = &a;
```

としてみると、



図の様に、ポインタ変数が２つでき、整数の変数値 1 を持つ同じ場所を、両方（&aとb）のポインタが指し示す様になります。このとき変数名 a を使って整数値 a の値を１から２に書き換えようとする場合には、

```
a = 2;
```

としますが、変数名 b を使って同じ操作をする場合には

```
*b = 2;
```

とします。
同じ場所を２つのポインタが指し示す必要性が今は理解できないかもしれませんが、いずれ、大きな、プログラムを書いていく上で関数という複数のモジュールに分ける必要がでてきます。その時、異なる関数の間で、同じ値を参照する必要が出てきた時にこのポインタの考え方が重要になってきます。

& や * といった記述の仕方は使ってみないと覚えにくいと思います。プログラミングに慣れていく上で、順次覚えていっていただければよいのですが、今回は特に、コンピューターの中で、実際に数値やデータを記憶する場所、と、その場所のアドレスを記述した場所、が別にあるということだけでも認識しておいていただければ、と思います。

配列

たくさんの変数を使いたい場合に、配列というものが使われます。たとえば５個の整数を使いたい時に、

```
int  a0, a1, a2, a3, a4;
```

とする代わりに、

```
int  a[5];
```

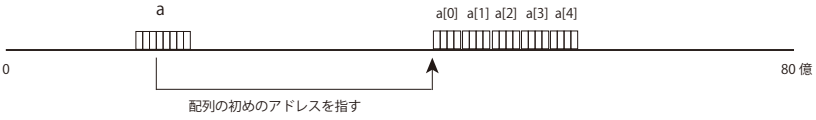
とすれば、a[0], a[1], a[2], a[3], a[4] という５個の変数が使えすようになります。配列のカギカッコ […] の間にはゼロから始まる整数値（添字と呼びます）がはいり、また、変数を使うこともできるので便利です。たとえば、別々に５つの変数名を宣言した上の方法ではそれぞれの値を０に（初期化）したいときには、

```
a0 = 0; a1 = 0; a2 = 0; a3 = 0; a4 = 0;
```

の様にひとつひとつ値を詰める必要がありますが、配列を使って下のように宣言してあれば、

```
int i;
for(i=0;i<5;i++)
  {
  a[i] = 0;
  }
```

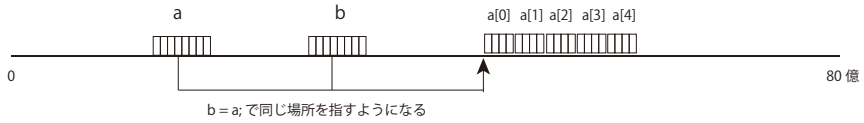
とできます。５個位なら上でいいやと思われるかもしれませんが、100万のオーダーになると上のやり方ではプログラムが不可能になってしまいます。また、上の方法で５個の変数を宣言すると、ポインタ分８バイトと整数値分4バイトの和の５倍で、60バイトの記憶容量を必要としますが、配列で宣言すると下図の様に、8+4x5 = 28バイトで済みます。



コンピュータは、例えば３番目の要素である a[2] の値を取り出すためには、添字の値２に整数型のサイズ４バイトを掛けた8バイトを、ポインタ a の値に足した場所を見れば良い、と判断します。
記法として注意していただきたいのは、先程、int a;と宣言したときにはポインタ変数は &a と参照したのに対して、配列で int a[5]; と宣言したときには、ただ a とすればポインタ変数が参照できます。&を使った書き方をしようと思えば、一番目の要素のアドレスという意味で、&a[0] とすれば a と同じ意味になります。同様に３番目の要素のアドレスは &a[2] と参照できます。
ここで先程と同様にもう一つの整数型のポインタを準備してみます

```
int i;
int a[5];
for(i=0;i<5;i++)
 {
 a = i+1;
 }
int *b;
b = a;
```

先程と同様、&aとbが同じ場所を指すようになります



先ほどと少し違うのは、aもbも&を付けずにアドレスを意味する全く等価な変数になっていることです。したがって、a[2]とb[2]は同じ場所にある３という数値を意味するようになります。
ポインタに整数値を足すと、そのポインタ型のサイズのバイト数が足されます。たとえば上の図の状況で a[2] の要素を参照したい時、bに２を加えて、その値 *b を取り出すことで同じように出来ます。

```
b = b+2;

printf(“%d\n”,a[2]);
printf(“%d\n”,*b);
```

下の二行、printfという文はそれぞれ a[2]と*bの値を画面に表示する関数で、いずれも２が出ます。
わかりにくいと思うので、いま仮にa[0]の先頭アドレスが100番地だったとしましょう。a と bにはいずれも100というアドレスの値が入ります。ここで、b=b+2; とすると bの値はいくつになるでしょう？ 普通に考えると102ですが、bは整数型のポインタで宣言されているために、足す整数値は整数型のサイズ４バイトの値が掛けられます。2x4=8なので、bの値は108になります。108番地はa[2]の要素が入っている場所なので *b とすることでa[2]の値が参照できるのです。

構造体

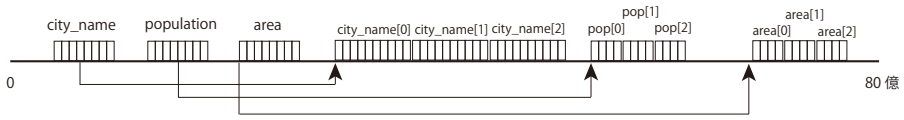
データを効率よく記憶するための仕組みとして、配列と同様に重要なものに構造体（structure）があります。下の表の様な、市の名前、人口、面積のデータを記憶する場合を考えてみましょう。

```
------------------------------------------------------------
City            Population      Area(km^2)
Maebashi        342678          311.64
Takasaki        371127          459.51
Saitama         1232577         217.49
------------------------------------------------------------
```

最も単純には、市の数が3なので、

```
char  city_name[3][10];
int   pop[3];
float area[3];
```

の様に人口と面積を別々の配列で表しておけばそれぞれの人口と面積を記憶する場所が確保できます。この時のメモリ上のデータの配置は以下のようになります。

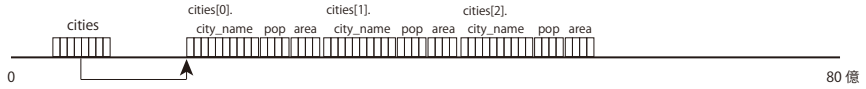


これに対して、市の情報をひとまとめにする構造体というものを以下のように定義できます。

```
struct city_info
 {
 char  city_name[10];
 int   pop;
 float area;
 };
```

こうして定義した構造体を使って以下の様に構造体変数を宣言すると、メモリ上の情報の配置は下図のように整理されます。

```
struct  city_info  cities[3];
```



この記法で、二番目の市の人口にアクセスするには cities[1].pop のようにピリオドを使います。また、先程のように別にポインタ型変数を定義し、同じ二番目の市の人口にアクセス（画面に表示）するには、以下のようにします。

```
struct  city_info  *p;
p = p + 1;
printf (“%d\n”,p -> pop);
```

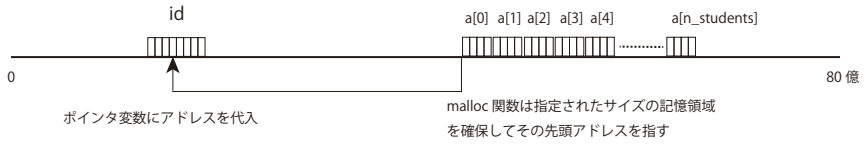
動的メモリ割り当て（dynamic memory allocation）　malloc

変数がポインタを使って構成されている大きな理由の一つは、先程述べたように、プログラムの異なる箇所から同じ変数値を参照したい場合、がありますが、もう一つの大きな要素は動的なメモリの割当にあります。プログラムを記述する段階で、必要な変数の数を特定できない場合があります。たとえば、任意のファイルを読み込んでその内容をすべてメモリ上に置きたい場合には、ユーザーがファイルを開いてみるまでそのファイルにいくつのデータが書き込まれているかを知ることが出来ません。あるいは、学生ひとりひとりに関する情報を処理するプログラムを作成する場合、学生の数は学校によって異なりますし、同じ学校でも翌年には人数が変わっているかもしれません。そこで、プログラムの実行時にデータ（学生）の数をカウントして n_students といった変数に格納し、その数だけの変数をたとえば、int id[n_students]; の様に宣言して確保できれば便利です。実際にはC言語では、この様に配列宣言の際の要素数に変数を用いることは許されていません。代わりにまず変数のポインタ宣言を行って、mallocという関数を用いる手順が取られます。このように、プログラムを実行する時に必要なメモリを確保する手法を　動的メモリ割り当て　と呼びます。

```
int  *id;
```
でポインタだけを宣言

プログラム中で要素数 n_students をカウントし、必要なメモリの量 （整数型 4バイト x n_students）をmallocという関数に渡せば、mallocはメモリ領域を確保して、その先頭アドレスを返してくれるので、id に代入すると、id[0], id[1], … id[n_student] と言った要素にアクセスできるようになります。

```
id = malloc ( 4 * n_students );
```



この動的メモリ割り当ての手法もプログラミング演習IVで実際に使ってみないとなかなか意味がわからないかもしれませんが、データの種類はわかっているけど、いくつのデータを処理しなければならないかプログラミング時にはわからない場合に、ポインタでデータの型だけを決めておいて、実際にデータを置く場所の確保はプログラムが動き始めてから行う方法だととりあえず理解して置けば、後の学習がすこしわかりやすくなるかと思います。

コンピュータの基礎：　２進数・８進数・１０進数・１６進数　　2019年5月27日

ASCII コード表

数値	文字記号	数値	文字記号	数値	文字記号	数値	文字記号
0	NUL	32	SP	64	@	96	`
1	SOH	33	!	65	A	97	a
2	SXT	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DEL	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
29	GS	61	EQ	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL