5 基本的データ型

各々の記憶領域に記録されたビット列がどういう内部表現方式に従っているかは、ビット列自身の中に明示的な情報として含まれている訳ではなく、その記憶領域を使う側 (プログラム側) が決める。

⇨ プログラムはその中で使用される各々の変数や配列の中のデータがどういう内部表現方式に従うかの情報を含む。

C言語でこれらの情報を明示しているのは変数や配列の宣言で、

`int a;` と宣言すると
例えば32ビット, 負数は2の補数で整数を表す方式が、

`float a;` と宣言すると
例えば32ビット, IEEE規格754で実数を表す方式が、

`double a;` と宣言すると
例えば64ビット, IEEE規格754で実数を表す方式が、

変数 a の計算機内部の表現方式として想定される。

これらの、計算機内部のデータ表現方式に対応する

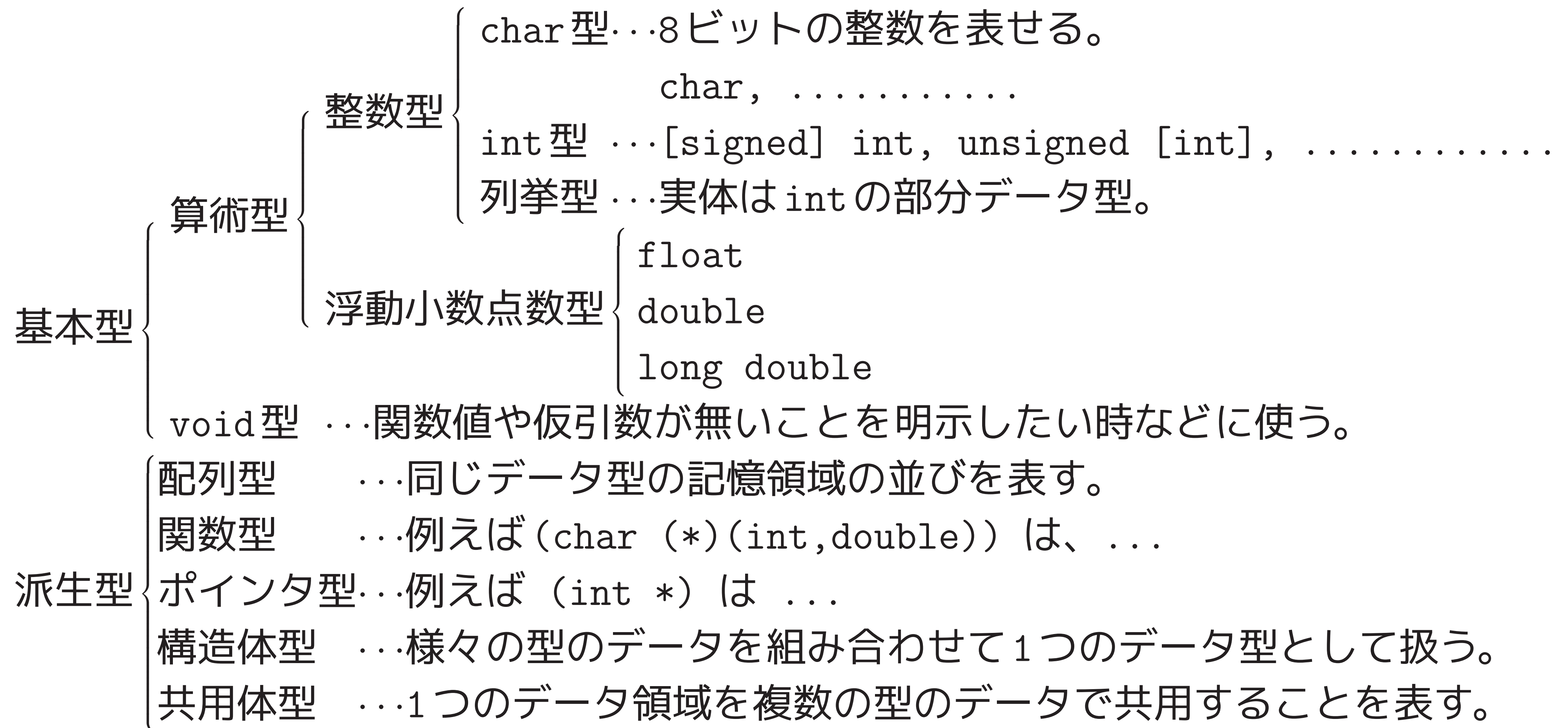
`int, float, double, ...`

といったものを、プログラム上ではデータ型と呼ぶ。

プログラミング言語によっては
複数のデータをまとめて1つの変数として扱う仕組みも用意されている。

⇨ この様な変数の内部構成もやはりプログラム内に宣言されていて、計算機内部のデータ表現の方式に対応するので、データ型と考える。

データ型の分類


基本型
算術型
整数型
char型···8ビットの整数を表せる。
char, ...
int型 ···[signed] int, unsigned [int], ...
列挙型 ···実体はintの部分データ型。
浮動小数点数型
float
double
long double
void型 ···関数値や仮引数が無いことを明示したい時などに使う。
派生型
配列型 ···同じデータ型の記憶領域の並びを表す。
関数型 ···例えば(char (*)(int,double)) は、...
ポインタ型···例えば (int *) は ...
構造体型 ···様々の型のデータを組み合わせて1つのデータ型として扱う。
共用体型 ···1つのデータ領域を複数の型のデータで共用することを表す。

5–1 C言語における文字の扱い---整数型 char--

- C言語では、文字は小さな整数として扱われる。例えば a という文字を表したい時にはプログラムの中では文字定数 'a' を用いるが、これは内部では 97 という整数として扱われる。

  各々の文字の番号は次のASCIIコード表に基づいて決められている。

	上位3ビット							
下位4ビット	0	1	2	3	4	5	6	7
0	ヌル文字	伝送制御拡張	空白	0	@	P	`	p
1	ヘディング開始	装置制御1	!	1	A	Q	a	q
2	テキスト開始	装置制御2	"	2	B	R	b	r
3	テキスト終了	装置制御3	#	3	C	S	c	s
4	伝送終了	装置制御4	$	4	D	T	d	t
5	問合せ	否定応答	%	5	E	U	e	u
6	肯定応答	同期信号	&	6	F	V	f	v
7	ベル	伝送ブロック終結	'	7	G	W	g	w
8	後退	取消	(	8	H	X	h	x
9	水平タブ	媒体終端	)	9	I	Y	i	y
A	改行	置換文字	*	:	J	Z	j	z
B	垂直タブ	拡張	+	;	K	[	k	{
C	書式送り	ファイル分離文字	,	<	L	\	l	|
D	復帰	グループ分離文字	-	=	M	]	m	}
E	シフトアウト	レコード分離文字	.	>	N	^	n	~
F	シフトイン	ユニット分離文字	/	?	O	_	o	抹消

抹消 . . . 機能文字

(列0・1) 機能文字

例えば文字 "K" の場合は、

上位3ビットが 8進の"4"、
下位4ビットが16進の"B"
⟹ 2進コード "100 1011" で表される。

主要な文字の番号は次の通り。

印字可能文字の場合					
文字定数	'a'	'b'	'c'	·······	'z'
(8進表記)	'\141'	'\142'	'\143'	·······	'\172'
(16進表記)	'\x61'	'\x62'	'\x63'	·······	'\x7A'
文字の番号	97	98	99	·······	122
文字定数	'A'	'B'	'C'	·······	'Z'
文字の番号	65	66	67	·······	90
文字定数	'0'	'1'	'2'	·······	'9'
文字の番号	48	49	50	·······	57
文字定数	'&'	'*'	'+'	·············	
文字の番号	38	42	43	·············	

機能文字の場合			
文字定数	’\0’(ナル文字)	’\a’(警告)	’\b’(後退)
(8進表記)	’\000’	’\007’	’\010’
(16進表記)	’\x00’	’\x07’	’\x0’
文字の番号	0	7	8
文字定数	’\t’(水平タブ)	’\n’(改行)	’\v’(垂直タブ)
文字の番号	9	10	11
文字定数	’\f’(紙送り)	’\r’(復帰)	’\"’(2重引用符)
文字の番号	12	13	34
文字定数	’\’’(引用符)	’\\’ (バックスラッシュ)	
文字の番号	39	92	

- 文字の番号(小さな整数)を記憶するためのデータ型として char 型が用意されている。

- char 型は次のいずれかと同等。(コンパイラ次第)

$$\begin{cases}\texttt{signed char} \\ \texttt{unsigned char}\end{cases}$$

- 文字定数 'a', 'b', ... は実は int 型。

- char型変数は1バイトの領域を占める。

  例えば 定数 'a' (と同じ値のchar型データ) は次の様に表される。

7	6	5	4	3	2	1	0
0	1	1	0	0	0	0	1

⇨ 文字 a の番号= $2^6 + 2^5 + 2^0 = 97$

一般に 、ビット列

$b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$

は、unsigned char型データとしては

$\sum_{i=0}^{7} b_i \times 2^i$

という値を持ち、signed char型データとしては

$-b_7 \times 2^7 + \sum_{i=0}^{6} b_i \times 2^i$

という値を持つ。

負数は2の補数で表す。

⇨ 表せる整数の範囲は 、

unsigned char型なら 0～255、

signed char型なら −128～127。

- C言語におけるchar型は整数型の一種に他ならないので、記憶した整数値を数字列で出力することは勿論出来る。ただ、char型変数に文字を記憶させたい場合のために、入力した文字からその番号を割り出したり、記憶した整数番号の文字を出力したり出来るようになっている。[実は、こちらの方がデータ変換が無くて単純。]

⇨ ◇ int型(またはshort型,long型)でも文字を表せる。
◇ char型変数は小さな整数値を保持するためにも使える。

例5．1（C言語における文字の扱い）

```
[motoki@x205a]$  nl datatype-char-Kelley.c
  1  #include <stdio.h>
  2  int main(void)
  3  {
  4    char c='a';
  5    printf("%c %c %c\n", c, c+1, c+2);
  6    printf("%d %d %d\n", c, c+1, c+2);
  7    return 0;
  8  }
[motoki@x205a]$  gcc datatype-char-Kelley.c
[motoki@x205a]$  ./a.out
a b c
97 98 99
[motoki@x205a]$
```

与えられた番号の文字を出力

整数値を10進表記で出力

5–2 1文字入出力 ---getchar() と putchar()

- 文字をそのまま入出力するための関数として getchar とputchar が用意されている。

  int getchar(void)

  ··· 標準入力stdinから文字を読み込み、その文字番号を値とする。

  int putchar(int)

  ··· 引数で指定された値を番号とする文字を標準出力stdoutに書き出す。

補足：

ともに、charで表せる保証の無い EOF を値とすることもあるので、関数値の型は char でなく int に設定されている。

実際、通常は #define EOF (-1) と定義されることが多いが、一般にはEOFの値はそれぞれの処理系の中でどの文字コードとも重ならないという制約を満たせば良いだけなので、charで表せる保証は無い。

- getcharの関数値の型はintなので、
  getcharで読み込んだデータは intで型の変数に格納するのが無難。

- 関数の引数結合は"値呼出し"で行われるので、
  putcharの引数としては int型だけでなく、任意の整数型の式が許される。[文字番号として可能な値を持てば良い。]

- getcharもputcharも<stdio.h>の中で定義されているマクロ。

例題5．2（英小文字→大文字）　文字を読み込みそれが英小文字なら大文字に直して出力する、という作業を繰り返すCプログラムを作成せよ。

例題5．2（英小文字→大文字）　文字を読み込みそれが英小文字なら大文字に直して出力する、という作業を繰り返すCプログラムを作成せよ。

（考え方）　p.131の文字番号の表を見るとアルファベット順に英小文字の番号は 97(=’a’)～122(=’z’)，英大文字の番号は 65(=’A’)～90(=’Z’）が割り当てられているので、読み込んだ文字の番号 c が ’a’以上’z’以下なら　c−’a’+’A’　が大文字に変換後の文字の番号となる。

（プログラミング）

```
[motoki@x205a]$  nl datatype-lower2upper-Kelley.c
     1 #include <stdio.h>

     2 int main(void)
     3 {
     4   int  c;
```

```
  5   while ((c = getchar()) != EOF)
  6     if ('a' <= c && c <= 'z')
  7       putchar(c - 'a' + 'A');
  8     else
  9       putchar(c);
 10   return 0;
 11 }
[motoki@x205a]$  gcc datatype-lower2upper-Kelley.c
[motoki@x205a]$  ./a.out
a
A
B
B
Ctrl-d
[motoki@x205a]$
```

例題5．3（8bitでの整数表現）　長さが8のビット列(すなわち $\mathbf{0}$ と $\mathbf{1}$ の数字列)

$$\mathbf{b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0}$$

を入力して、

① このビット列を **unsigned char** 型データと見た時に表す(非負)整数値 $\sum_{\mathbf{i=0}}^{\mathbf{7}} \mathbf{b_i \times 2^i}$、
およびび

② このビット列を **signed char** 型データと見た時に表す整数値 $-\mathbf{b_7 \times 2^7} + \sum_{\mathbf{i=0}}^{\mathbf{6}} \mathbf{b_i \times 2^i}$

を出力するCプログラムを作成せよ。

(考え方)　読み込んだ文字 '0', '1' の番号 c からその数字の表す数値に変換するには c−'0' を計算すれば良い。 また、次の様に計算すれば効率的に計算できる。

$$\sum_{i=0}^{7} b_i \times 2^i = ((\ldots((b_7\times2+b_6)\times2+b_5)\ldots)\times2+b_1)\times2+b_0$$

$$-b_7\times2^7+\sum_{i=0}^{6} b_i \times 2^i = ((\ldots((-b_7\times2+b_6)\times2+b_5)\ldots)\times2+b_1)\times2+b_0$$

(プログラミング)　読み込む文字データを一時的に格納するために c という名前のint型変数を、読み込んだ b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0 の数としての値を格納するためにint型配列 bit[7]~bit[0] を用意し、また $\sum_{i=0}^{7} b_i \times 2^i$, $-b_7\times2^7+\sum_{i=0}^{6} b_i \times 2^i$ という累算値を計算するためにそれぞれ unsigned_val, signed_val という名前の変数を用意する。

```
[motoki@x205a]$ cat -n datatype-bit-string-as-char.c
     1 /* 長さが8の 0 と 1 の数字列を入力して                          */
     2 /* (1)このビット列を unsigned char 型データと見た時に */
     3 /*     表す(非負)整数値、および                               */
     4 /* (2)このビット列を signed char 型データと見た時に     */
     5 /*     表す整数値                                             */
     6 /* を出力するCプログラム                                       */
     7
     8 #include <stdio.h>
     9 #include <stdlib.h>
    10
    11 int main(void)
    12 {
    13   int  c, bit[8], i;
    14   unsigned char  unsigned_val;
    15   signed   char  signed_val;
    16
```

```
17    printf("Input a bit string of length 8: ");
18    for (i=7; i >= 0; ){
19      switch (c = getchar()) {
20      case '0': case '1':
21        bit[i--] = c-'0';
22        break;
23      case ' ': case '\n': case '\t':
24        break;
25      default:
26        printf("Invalid character appears.\n");
27        exit(EXIT_FAILURE);
28      }
29    }
30
31    unsigned_val = bit[7];
32    signed_val   = -bit[7];
33    for (i=6; i>=0; i--) {
```

```
34      unsigned_val = unsigned_val*2 + bit[i];
35      signed_val   = signed_val*2 + bit[i];
36    }
37
38    printf("\n==>The input bit string can be interpreted
39            "   have a value %d as an unsigned char data,
40            "   have a value %d as a signed char data.\n",
41            unsigned_val, signed_val);
42    return 0;
43  }
[motoki@x205a]$  gcc datatype-bit-string-as-char.c
[motoki@x205a]$  ./a.out
Input a bit string of length 8:  1111 1010

==>The input bit string can be interpreted to
   have a value 250 as an unsigned char data, and
   have a value -6 as a signed char data.
```

```
[motoki@x205a]$  ./a.out
Input a bit string of length 8:  0 1 0 1
0 111

==>The input bit string can be interpreted to
   have a value 87 as an unsigned char data, and
   have a value 87 as a signed char data.
[motoki@x205a]$
```

5–3 データ型 int

- 整数値を表すための最も標準的なデータ型。

- 普通、int型データは1ワードに格納される。

  (ワード： コンピュータ／CPUが一度に処理するデータの単位のこと。今はどれも 1ワード=32ビット以上。最近では 1ワード=64ビットもある。)

- 1ワードが32ビットの計算機の場合 、int型データは長さが32のビット列で表される。

  ⇨ 2^{32}個の整数を表すことが可能。

  ⇨ 普通の計算機だと、

$$
\begin{aligned}
-2^{31} &= -2147483648 \text{ 以上、} \\
2^{31}-1 &= 2147483647 \text{ 以下}
\end{aligned}
$$

  の整数を表せる。

- ごく普通の計算機の場合 、

1ワード=32ビットであり、長さが32のビット列

$$b_{31}\ b_{30}\ b_{29} \cdots\cdots\ b_2\ b_1\ b_0$$

は、int型データとしては

$$-b_{31} \times 2^{31} + \sum_{i=0}^{30} b_i \times 2^i$$

という整数値を持つ。

- 普通のC言語処理系だと
  オーバーフローのチェックはしてくれない。

```
[motoki@x205a]$  cat datatype-int-overflow-Kelley.c
#include <stdio.h>
#define  BIG  2000000000
int main(void)
{
   int  a, b=BIG, c=BIG;
   a = b + c;
   printf("a=%d  b=%d  c=%d\n", a, b, c);
   return 0;
}
[motoki@x205a]$  gcc datatype-int-overflow-Kelley.c
[motoki@x205a]$  ./a.out
a=-294967296  b=2000000000  c=2000000000
[motoki@x205a]$
```

⇨ C言語では、オーバーフローしない様にするのはプログラマの責任。

- 8進整数を10進整数と混同しないこと。例えば、

456 は 10進定数、

0456 は 8進定数
(10進で $4\times8^2+5\times8^1+6\times8^0=302$ に相当)

0x456 は 16進定数
(10進で $4\times16^2+5\times16^1+6\times16^0=1110$ に相当)

0xaBc は 16進定数
(10進で $10\times16^2+11\times16^1+12\times16^0=2748$ に相当)

5–4 整数型：char, short, int, long, unsig

- 整数値を表すためのデータ型としては char, short, int, long（および、各々を unsigned にしたもの）が用意されているが、ANSI規格で定められているのは次のことだけ。（あとはコンピュータ／処理系に依存。）

char のビット数 = 8ビット
long のビット数 ≥ int のビット数
≥ short のビット数
≥ 16ビット
long のビット数 ≥ 32ビット

- 標準ヘッダファイル <limits.h> の中に、扱える最大整数値などの情報が入っている。

マクロ名		意味
CHAR_BIT	···	charのビット数
INT_MIN	···	intの最小値
INT_MAX	···	intの最大値
LONG_MIN	···	longの最小値
LONG_MAX	···	longの最大値

······················

表せる範囲：

8ビット、16ビット、32ビットで表せる最大整数、最小整数は. . .

ビット数		表せる最小整数	表せる最大整数
signed	8	−128	127
	16	−32768	32767
	32	−2147483648	2147483647
unsigned	8	0	255
	16	0	65535
	32	0	4294967295

整数定数：

- 普通の整数定数は int, long, またはunsigned long型 のデータとして扱われる。 （表せる最小の型が選ばれる。）
- 整数定数の型を long, unsigned, ... などに指定できる。例えば、

  37u, 37U は unsigned 型
  37l, 37L は long型
  37ul, 37UL は unsigned long型

整数の内部表現形式：

- unsignedの場合の整数データの記憶方式は全て同じ。すなわち、長さが n のビット列

$$b_{n-1}\ b_{n-2}\ b_{n-3}\cdots\cdots\ b_2\ b_1\ b_0$$

が符号なし整数を表すと見た場合、その値は

$$\sum_{i=0}^{n-1} b_i \times 2^i$$

である。

- signedの場合の整数データの記憶方式は、ほぼ全て同じ。すなわち、普通の計算機の場合、長さが n のビット列

$$b_{n-1}\ b_{n-2}\ b_{n-3}\cdots\cdots\ b_2\ b_1\ b_0$$

が符号付き整数を表すと見た場合、その値は

$$-b_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} b_i \times 2^i$$

である。

5–5 自習 列挙型

- 有限集合の中の個々の要素に記号の名前（ 列挙定数 という）を付け、プログラム内でそれらの名前を自由に使える様にするための機構
- 計算機内部では、個々の要素には整数の識別番号が付けられる。
- 列挙定数の宣言は次のように行なう。

宣言 · · · 説明
`enum boolean {NO, YES, FALSE=0, TRUE};` · · · 列挙定数 NO, YES, FALSE, TRUE の値（識別番号）がそれぞれ 0, 1, 0, 1 に設定される。

`enum month {JAN=1,FEB,MAR,APR,MAY,JUN,`
`JUL,AUG,SEP,OCT,NOV,DEC};`
· · · 列挙定数 JAN, FEB, MAR, . . ., DEC の値（識別番号）がそれぞれ 1, 2, 3, . . ., 12 に設定される。

- 列挙型の変数を確保するには 、次の様に書く。

```
enum boolean sw;
```

5–6 自習 ビット演算

論理演算子：

- `~` 整数式 …各ビットを反転
- 整数式1 `&` 整数式2 …ビット位置毎に論理積
- 整数式1 `^` 整数式2 …ビット位置毎に排他的論理和
- 整数式1 `|` 整数式2 …ビット位置毎に論理和

シフト演算子：

- 整数式1 `<<` 整数式2 … 左シフト。

  (整数式1のビット表現が整数式2ビット分だけ左にシフトされる。)

- 整数式1 `>>` 整数式2 … 右シフト。

  (整数式1のビット表現が整数式2ビット分だけ右にシフトされる。)

例題５．４（１の補数,２の補数,２のべき乗倍） 整数データを $\mathbf{1}$ 個入力してその $\mathbf{1}$ の補数,$\mathbf{2}$ の補数,$\mathbf{2}$ の $\mathbf{5}$ 乗倍,$\mathbf{2}$ の $\mathbf{-5}$ 乗倍の値を出力する $\mathbf{C}$ プログラムを作成せよ。

（考え方）

整数データの １の補数を求めるには 、
その整数データを表す２進内部表現の各ビットを反転すれば良い。

２の補数を求めるには 、
２進内部表現の各ビットを反転した後に１を加算すれば良い。

また、整数データを ２の５乗倍,２の−５乗倍するには 、
その整数データを表す２進内部表現のビット列をそれぞれ5ビットだけ左シフト，５ビットだけ右シフトすれば良い。

（プログラミング）

```
[motoki@x205a]$  cat -n datatype-complement-2pow.c
```

```
 1 #include <stdio.h>
 2
 3 int main(void)
 4 {
 5   int a;
 6
 7   scanf("%d", &a);
 8   printf("1's complement of %d = %d\n"
 9          "2's complement of %d = %d\n"
10          "%d * (2 raised to the 5 power)  = %d\n"
11          "%d * (2 raised to the -5 power) = %d\n",
12          a, ~a, a, ~a+1, a, a<<5, a, a>>5);
13   return 0;
14 }
[motoki@x205a]$  gcc datatype-complement-2pow.c
[motoki@x205a]$  ./a.out
1234567
```

```
1's complement of 1234567 = -1234568
2's complement of 1234567 = -1234567
1234567 * (2 raised to the 5 power)  = 39506144
1234567 * (2 raised to the -5 power) = 38580
[motoki@x205a]$
```

例題5．5（int型データのビット表現） int型データを1個入力してそのビット表現を 0 と 1 の文字列として出力するCプログラムを作成せよ。

（考え方） int型が32ビットで構成されているのだとすると、式 1<<31 の値の2進内部表現は "10000000 00000000 00000000 00000000" である。それゆえ、整数データの2進内部表現と1<<31の2進内部表現の間でビット毎の論理積を取れば、その結果は整数データの2進内部表現の最上位ビットが1であるかどうかによって

"10000000 00000000 00000000 00000000"

または

"00000000 00000000 00000000 00000000"

(整数としての値は 0)

になる。従って、読み込んだint型データ a と 1<<31 の間で&(ビット毎の論理積)演算を行った結果が 0 になれば a のビット表現の最上位ビットは '0'、そうでなければ a のビット表現の最上位ビットは '1'

であることが分かる。

データ a を構成する残りのビットも上位から順に調べたければ、a のビット表現を1ビットずつ左にシフトしながらその最上位ビットを同じ要領で調べて行けば良い。

(プログラミング)

```
[motoki@x205a]$  cat -n datatype-bit-rep-of-int.c
     1 #include <stdio.h>
     2
     3 int main(void)
     4 {
     5   int a, i, mask=1<<31;
     6
     7   scanf("%d", &a);
     8   printf("Integer %d is internally represented "
                                  "by the bit sequence\n"
```

```
 9              "      ", a);
10
11    for (i=0; i<32; i++) {
12      if (i%8 == 0)
13        putchar(' ');
14      putchar((a&mask)==0 ? '0' : '1');
                                /* a&mask==0 と書くとダメ */
15      a = a<<1;
16    }
17    printf(".\n");
18    return 0;
19 }
```

```
[motoki@x205a]$ gcc datatype-bit-rep-of-int.c
[motoki@x205a]$ ./a.out
1024
Integer 1024 is internally represented by the bit sequence
      00000000 00000000 00000100 00000000.
```

```
[motoki@x205a]$  ./a.out
-1
Integer -1 is internally represented by the bit sequence
      11111111 11111111 11111111 11111111.
[motoki@x205a]$
```

5–7 浮動小数点数型

浮動小数点数型：

- 精度の保証された広範囲の実数値を表すためのもの。
  ⇨ 誤差に注意。

- `float`(単精度),
  `double`(倍精度，C言語では標準),
  `long double`(4倍精度)
  } の3つが用意されている。

- データ領域の大きさはコンピュータに依存している。
  `float`の精度 ≤ `double`の精度 ≤ `long double`の精度

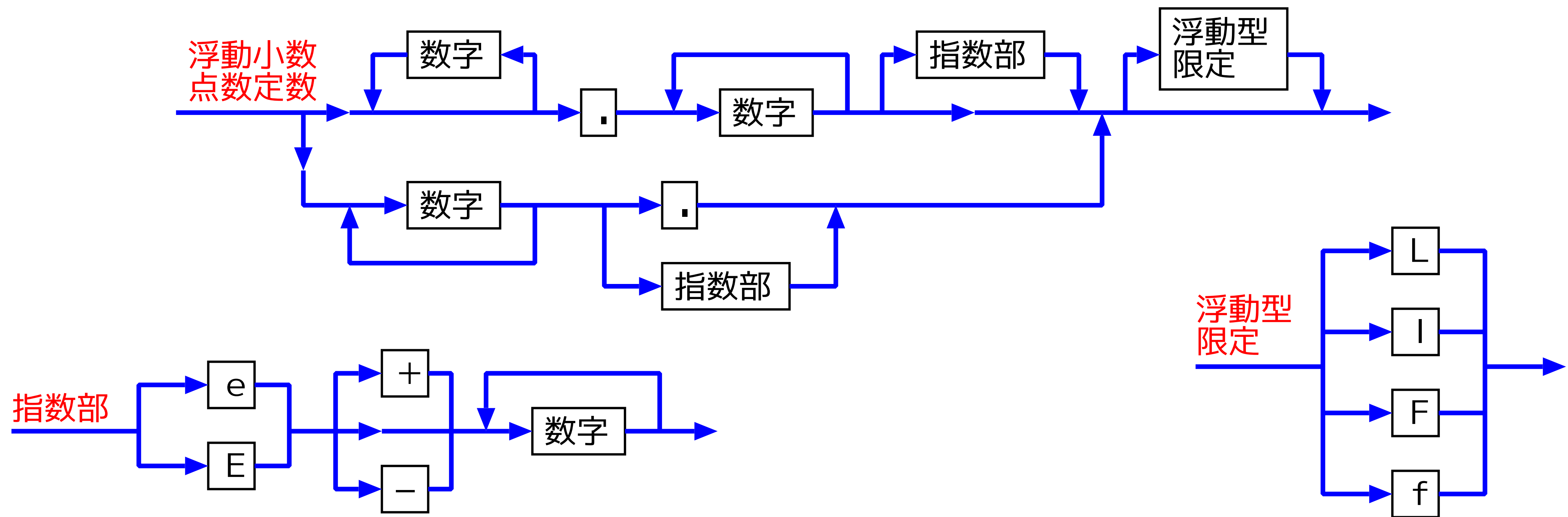
- 標準ヘッダファイル `<float.h>` の中に、扱える最大の浮動小数点数などの情報が入っている。

マクロ名		意味
`FLT_MAX`	···	最大の`float`型浮動小数点数
`DBL_MAX`	···	最大の`double`型浮動小数点数

....

浮動小数点数定数：

- 123.4, 123., .4, 123.4e5, .4E+5, 123e-5, ... といった書き方が出来る。 これらはdouble型の定数になる。
- 定数をfloat型にしたければ、最後に f または F という接尾語を付ける。 例えば、123.4f, .4E+5F, ... 。
- 定数をlong double型にしたければ、最後に l または L という接尾語を付ける。 例えば、123.4l, .4E+5L, ... 。



精度と指数部の表現可能な範囲：

- ごく普通の計算機の場合、float型，double型データは、各々4バイト，8バイトの領域を占め、10進で各々約6桁，約15桁の精度を持つ。また、指数部の表現可能な範囲は、およそ、各々 $-38\sim+38$，$-308\sim+308$ となる。［指数部の表現可能な範囲は計算機のアーキテクチャに大きく依存する。］

IEEE規格754

- 単精度、倍精度、4倍精度における指数部、仮数部のビット数等は次の様に定められている。

	符号部	指数部	仮数部	全部で
単精度	1ビット	8ビット	23ビット（10進で6～7桁）	32ビット
倍精度	1ビット	11ビット	52ビット（10進で15～16桁）	64ビット
4倍精度	1ビット	15ビット	112ビット（10進で33～34桁）	128ビット

- 単精度の場合 、32ビットの列

$$\underbrace{s}_{\text{符号部}}\ \underbrace{e_7e_6\cdots e_1e_0}_{\text{指数部}}\ \underbrace{d_1d_2\cdots d_{22}d_{23}}_{\text{仮数部}}$$

によって、

$$\begin{cases} (-1)^s\times(1+M)\times 2^E & \text{if } -127<E<128 \\ (-1)^s\times M\times 2^{E+1} & \text{if } E=-127 \\ \text{Inf(無限大)} & \text{if } E=128,\ M=0 \\ \text{NaN(非数,Not a Number)} & \text{if } E=128,\ M\neq 0 \end{cases}$$

という実数を表す。但し、

$$M = \sum_{i=1}^{23} d_i\times 2^{-i}$$

(仮数部から $d_0=1$ というビットが省かれていると暗に仮定し、

$$1.d_1d_2\cdots d_{22}d_{23}$$

という2進小数を仮数部が表すと考える。)

$$E = \sum_{i=0}^{7} e_i\times 2^i - 127$$

では、例えば 26 はIEEE規格754(単精度)でどう表されるのか?

⇨ 26.0 を $(-1)^{s}\times(1+M)\times 2^{E}$ という形に変形してみる。

$$
\begin{aligned}
26.0 &= 11010.0 \ (2\text{進数}) \\
&= 1.10100 \times 2^{4} \\
&= (-1)^{0}\times(1+0.10100)\times 2^{4}
\end{aligned}
$$

⇨ 2つを見比べて、s=0, M=0.10100, E=4

⇨ $s e_7 e_6 \cdots e_1 e_0 d_1 d_2 \cdots d_{22} d_{23}$ で 26 が表されるとすると、

$M = \sum_{i=1}^{23} d_i \times 2^{-i} = 0.10100$

なので、

$d_1 = 1$

$d_2 = 0$

$d_3 = 1$

$d_4 = 0$

$d_5 = 0$

....

$E = \sum_{i=0}^{7} e_i \times 2^{i} - 127 = 4$ より、

$\sum_{i=0}^{7} e_i \times 2^{i} = 131$

$= 10000011 \ (2\text{進数})$

それゆえ、

$e_7 = 1$

$e_6 = e_5 = \cdots = e_2 = 0$

$e_1 = e_0 = 1$

実数計算における誤差の必然性 ($1 \div 10 \times 10 \neq 1$?):

実数データを扱う場合には、常に

　　記憶された数値が表そうとした数値の近似にすぎない

ことに注意しなければならない。

例えば、IEEE規格754(単精度)の表現法の場合

10進数 1 は X'3F800000' と表され, $\cdots 2^0 \times 1.0$(2進)

10進数 10 は X'41200000' と表される。 $\cdots 2^3 \times 1.010$(2進)

これら2数の間で割り算$(2^0 \times 1.0) \div (2^3 \times 1.010)$ を行うと、

$0.0\dot{0}011\dot{1}$(2進小数)$=2^{-4} \times 1.\dot{1}001\dot{1}$(2進小数) 循環小数，下記補足

⇨ 10進実数の割算 $1 \div 10$ の結果は X'3DCCCCCC' と表される。

誤差発生

更に、この誤差付きの除算結果

0.0001 1001 1001 1001 1001 1001 100 (2進) = 0.1999998(16進)

に 10(10進数) = 1010(2進数) = A(16進数) を掛けると、

0.FFFFFF0(16進小数)$=2^{-1} \times$1.FFFFFE(16進小数) 下記補足

⇨ 10進の式 $1 \div 10 \times 10$ の結果は X'3F7FFFFF' と表される。

⇨ 実数表現の世界では、

10進の式 $1 \div 10 \times 10$ の計算結果は10進数 1 と等しくはならない。

補足（1÷10）：

$$
\begin{array}{r}
0.0001\dot{1}001\dot{1} \\
1010\ \overline{)\ 1.0000} \\
1010 \\
\hline
1100 \\
1010 \\
\hline
1000
\end{array}
$$

補足(16進の掛け算 0.1999998×A)：

$$
\begin{array}{r}
0.1999998 \\
\times \qquad\qquad A \\
\hline
0.FFFFFF0
\end{array}
\qquad
\begin{cases}
1\times A = A & \text{（16進の「九九」）} \\
9\times A = 5A & \text{（16進の「九九」）} \\
8\times A = 50 & \text{（16進の「九九」）}
\end{cases}
$$

5–8 実数計算に伴って発生する誤差について

計算機を用いて数値計算を行う際、次の様な誤差／現象が起こります。[この内、計算機特有のものは①基数変換に伴う誤差だけであり、残りの3種は手計算の際も起こる。]

① 基数変換（i.e.2進↔10進変換）に伴う丸め：

例えば、10進小数 0.1 は2進法では $0.0\dot{0}01\dot{1}$ という循環小数になる。それゆえ、各数値を2進有限固定長（普通32ビット）で記憶する計算機としては、表し切れない下位の桁を 丸め （四捨五入，切り捨て，または切り上げ）ることになり、10進小数 0.1 を正確に記憶することはできない。従って、計算機内部で 0.1×10.0 の計算をしても結果は 1 にはならない。

一方、10進数 2^{-20} は計算機内部では実数データとして正確に記憶されるが、この値を10進表記（i.e.2進→10進変換）すると $9.5367431640625\times 10^{-7}$ ということになる。この数値は有限小数には違いないが、これを10進7桁の精度で出力すると8桁目以降は捨てられ誤差が発生する。

② 演算に伴う丸め：

例えば、有効桁3桁同士の乗算 1.23×4.56 を行うと

1.23×4.56 = 5.60 88

となり、10^{-3}の位以下が丸め(四捨五入,切り捨て,または切り上げ)られる。この種の誤差に対処するには、式の簡素化などにより演算回数をできるだけ少なくするしかない。

---

③ 情報落ち (情報埋没)：

これは②の誤差の一種である。絶対値の大きさが桁違いに違う2数を加減算すると小さい方の下位の桁が失われてしまう。例えば、次の加算では下線部が失われる。

```
   1.234567
+  0.04321098
-------------
   1.27777798
```

(下線部: 98)

数回の加減算では大した誤差は累積しないが、大量の実数値データを累算する場合は誤差が大きく累積することもある。

この情報落ち誤差が大きく累積しない様にするためには、

多数の実数データの累算は絶対値の小さいものから順に行う

様に心掛ける。

[実数データを累算する毎に誤差が少しずつ溜まってゆくので、次の④桁落ちほど気を付ける必要はない。]

---

④ 桁落ち：

大きさのほぼ等しい2数を減算する時，あるいはそれと同等の加算をする時、有効桁が大きく失われてしまう。 例えば、次の通り。

```
   1.234567 ··· 7桁の有効数字
 − 1.234566 ··· 7桁の有効数字
 ----------
   0.000001 ··· 1桁の有効数字
```

実数計算においては精度が重要であるから、最終結果に影響を及ぼす様な桁落ちは絶対に避けなければならない。

［厳密に言うと、桁落ちにおいては新たな（絶対）誤差が発生する訳ではない。上位の桁が失われてしまうために、それまで累積していた誤差部分の（以後の計算における）影響度／注目度が大きくなるのである。］

5–9 sizeof 演算子

sizeof 演算子：

- 与えられたデータ型またはデータの大きさ(バイト数)を調べるための演算子。
- 次の様に書く。

```
sizeof( データ型 ),
sizeof( 式 ),       sizeof 式 ,
sizeof( 配列 ),     sizeof 配列 ,     ···配列全体の占めるバイト数
sizeof( 構造体 ),   sizeof 構造体 ,
   ································
```

- 関数呼出しと類似の表記法だが、単項演算子。
- 他の単項演算子(e.g. 符号反転の -, ++)と同じ優先順位、結合性(右から左)を持つ。
- 演算結果は普通 unsigned。

例5．6（基本データ型の大きさ, 配列の大きさ）基本データの大きさが実際にどうなっているかは<limits.h> を調べれば分かりますが、これは次の様にプログラムの中で調べることも出来ます。

```
[motoki@x205a]$  cat -n datatype-sizeof.c
     1  #include <stdio.h>
     2  int main(void)
     3  {
     4    int a[]={0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
     5
     6    printf("      char:%3d byte \n", sizeof(char));
     7    printf("     short:%3d byte \n", sizeof(short));
     8    printf("       int:%3d byte \n", sizeof(int));
     9    printf("      long:%3d byte \n", sizeof(long));
    10    printf("  unsigned:%3d byte \n", sizeof(unsigned));
    11    printf("     float:%3d byte \n", sizeof(float));
    12    printf("    double:%3d byte \n", sizeof(double));
```

```
13   printf("long double:%3d byte \n\n", sizeof(long doubl
14
15   printf("  array a[]:%3d byte \n", sizeof(a));
16   return 0;
17 }
[motoki@x205a]$  gcc datatype-sizeof.c
[motoki@x205a]$  ./a.out
       char:  1 byte
      short:  2 byte
        int:  4 byte
       long:  4 byte
   unsigned:  4 byte
      float:  4 byte
     double:  8 byte
long double: 12 byte

  array a[]: 40 byte
```

```
[motoki@x205a]$
```

5–10 型変換とキャスト

算術計算の際の自動型変換：

- 型の異なるデータ間で算術計算を行う際（等）には、2つのデータの型を揃えたりするために、内部では次の順に強制的に型変換が行われる。

  ① char 型や short 型のデータは int 型に変換される。

  ② データ型間の次の順序に従って、下位（i.e. 左）の方の型が上位の型に変換される。（変換後の型が演算結果の型になる。）

  int $<$ unsigned $<$ long $<$ unsigned long
  $<$ float $<$ double $<$ long double

  ⇨ char 型同士の加算結果は char ではなく int 型。

- 整数 ↔ 実数 間の型変換が実際にどう行われるかについては計算機に依存する。

代入の際の自動型変換：

- 代入 変数等 = 式 において 両辺の型が違えば 、
  式の値は変数等の型に強制的に変換される。

キャスト演算子：

- 明示的に型変換を行うことが出来る。
- 式の値をデータ型という型に変換したければ、次の様に書く。
  （ データ型 ） 式
- キャストは単項演算子。
- 他の単項演算子(e.g.符号反転の-,++)と同じ優先順位、結合性(右から左)を持つ。

例５．７（キャスト演算の優先順位）

(float) i+3　は　((float)i) + 3　と同等。

5–11 自習 16進定数と8進定数

16進数：

- 0〜 9，A〜 F(各々10〜 15の代わり)の16個の数字を用いて数を表したもの。

- 16進数字の列

$$h_{n-1}h_{n-2}\cdots h_2h_1h_0$$

によって、

$$\sum_{i=0}^{n-1} h_i \times 16^i$$

という数を表す。

例えば、16進数 A0F3C は 次の10進数を表す。

$$\underbrace{10}_{A} \times 16^4 + 0\times 16^3 + \underbrace{15}_{F} \times 16^2 + 3\times 16^1 + \underbrace{12}_{C} \times 16^0$$
$$= 659260$$

- 非負整数 x を表す16進数を求めるには

$$h_i = \mathrm{mod}(\lfloor x/16^i \rfloor, 16)$$
$$= (\lfloor x/16^i \rfloor \text{を16で割った余り})$$

を計算し、これらを並べればよい。
例えば、x=659260 の場合には次のように計算する。

６５９２６０を１６で割っている

```
16) 659260
16)  41203   余り 12
16)   2575   余り  3
16)    160   余り 15
16)     10   余り  0
         0   余り 10
```

A 0 F 3 C

答

8進数 ・・・（16進数の場合と同様）

16進定数と8進定数： C言語においては、

- 0[0〜7の数字列]
  という形の字句は整数を8進表記したものと見なされる。

- 10〜15を表す16進数字としては、英大文字 A〜F と同様に英小文字 a〜f も許され、
  0x[16進数字の列] または 0X[16進数字の列]
  という形の字句は整数を16進表記したものと見なされる。

- 変換指定を %x または %#x として書式付き出力(e.g.printf)を行えば、整数を16進表記で出力できる。また、変換指定を %x として書式付き入力(e.g.scanf)を行えば、16進表記の整数を入力できる。

- 変換指定を %o または %#o として書式付き出力(e.g.printf)を行えば、整数を8進表記で出力できる。また、変換指定を %o として書式付き入力(e.g.scanf)を行えば、8進表記の整数を入力できる。

- unsigned や long の指定を行いたければ、10進の場合と同様、それぞれ u(またはU) やl(またはL) を定数の最後に付ける。

例5．8（16進定数,8進定数の扱い） 16進定数, 8進定数, %x変換, %#x変換, %o変換, %#o の使用例を次に示す。

```
[motoki@x205a]$  nl datatype-hexadecimal-const-Kelley.c
     1 #include <stdio.h>

     2 int main(void)
     3 {
     4   printf("%d %#x %#o\n", 19, 19, 19);

     5   printf("%d %x %o\n", 19, 19, 19);
     6   printf("%d %x %o\n", 0x1c, 0x1c, 0x1c);
     7   printf("%d %x %o\n", 017, 017, 017);

     8   printf("%d\n", 11 + 0x11 + 011);
     9   printf("%d\n", 2097151);
    10   printf("%d\n", 0x1FfFFf);
    11   return 0;
```

```
  12 }
[motoki@x205a]$  gcc datatype-hexadecimal-const-Kelley.c
[motoki@x205a]$  ./a.out
19 0x13 023
19 13 23
28 1c 34
15 f 17
37
2097151
2097151
[motoki@x205a]$
```