入門講習会 ビット/変数のサイズ,符号

1. 2 進数

コンピューターは, 内部ではあらゆる値を 2 進数で扱っている。なぜなら電気で動くコンピューターにとっては, 「電気が流れているなら 1, 流れていないなら 0」のように値を表現したほうが扱いやすいからである。

人によっては高校数学の「n 進法」という単元で 2 進法についても学んだかもしれないが, 一応復習しておこう。

2 進法とは, 数字を 0 と 1 の 2 種類だけで表現する方法である。対して, 0,1,2,3,4,5,6,7,8,9 の 10 種類で数字を表現する方法は 10 進法と呼ばれる。2 進法, 10 進法で表された数のことをそれぞれ 2 進数, 10 進数と呼ぶ。これら 2 つを区別するために, 数の右下に(進数)と記述することがある。

[10 進数, 2 進数の例]

10 進数: $24_{(10)}$

2 進数: $1010010_{(2)}$

10 進法と比較しながら, 2 進法についてみていこう。

まず 10 進法は 9 に 1 を加えると繰り上がりが起こる。一方 2 進法では 1 に 1 を加えると繰り上がりが起こる。

$9_{(10)} + 1_{(10)} = 10_{(10)}$

$1299_{(10)} + 1_{(10)} = 1300_{(10)}$

$1_{(2)} + 1_{(2)} = 10_{(2)}$

$1011_{(2)} + 1_{(2)} = 1100_{(2)}$

10 進法では各桁の数に 10 の累乗の重みを掛けた数の和で表すことができる。一方 2 進法では各桁の数に 2 の累乗の重みを掛けた数の和で表すことができる。その和の結果は 10 進数となる。

$1299_{(10)} = 1\times10^3 + 2\times10^2 + 9\times10^1 + 9\times10^0 = 1299_{(10)}$

$1101_{(2)} = 1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0 = 13_{(10)}$

10 進法の各桁を取り出すための方法に, 「10 で割った余りで 1 の位を求めた後, それを 10 で割って切り捨てを行う操作を繰り返す」というものを以前説明した(第 3 回, 第 4 回ループ例参照)。一方 2 進法では, 「2 で割った余りで 1 の位を求めたあと, それを 2 で割って切り捨てを行う操作を繰り返す」方法によって, 各桁を取り出すことができる。つまり, 10 進法から 2 進法への変換ができる。

$1299_{(10)}$から各桁を 10 進法で取り出す:

1299 % 10 = $\underline{9}$　　1299 ÷ 10 = 129

129 % 10 = 9　　129 ÷ 10 = 12
12 % 10 = 2　　12 ÷ 2 = 1
1 % 10 = 1　　1 ÷ 10 = 0

余りの部分を下から順に見ていくと, 1, 2, 9, 9, つまり $1299_{(10)}$が復元できる。

$13_{(10)}$から各桁を 2 進法で取り出す:

13 % 2 = 1　　13 ÷ 2 = 6
6 % 2 = 0　　6 ÷ 2 = 3
3 % 2 = 1　　3 ÷ 2 = 1
1 % 2 = 1　　1 ÷ 2 = 0

余りの部分を下から順に見ていくと, 1, 1, 0, 1, つまり $1101_{(2)}$が 13 の 2 進数表現となる。

2. ビット

2 進数の一つの桁には名前がついており, 「ビット」と呼ばれる。例えば $10100100_{(2)}$という 2 進数があったとしよう。これには 8 つのビットがあるので, 8 ビットの 2 進数と呼ぶことがある。ビットの順番を数えるとき, 「右/左から○ビット目」と言うことがある。

ある桁のビットが 1 であるとき, 「ビットが立っている」と表現することがある。例えば $10100100_{(2)}$という 2 進数は, 右から 3, 6, 8 ビット目が立っている。

コンピューターでは値を 2 進数で表現すると前項で話した。2 進数によって単に「数値」だけを表現できるわけではない。

各ビットは 0 と 1 しか値をとりえないことから, 「ある状態が真か偽か」という情報を 2 進数に入れることができると考えられる。

例えば, 太郎君, 次郎君, 三郎君がある授業に出席しているか否かについて, 次のように 3 ビットの 2 進数だけで情報を表現することができる。

右から 1 ビット目が立っていれば, 太郎君は出席している。
右から 2 ビット目が立っていれば, 次郎君は出席している。
右から 3 ビット目が立っていれば, 三郎君は出席している。

これによって, 101 は「太郎君と三郎君は出席している」, 111 は「3 人とも出席している」, 000 は「誰も出席していない」のように情報を表すことができる。

「ある状態が真か偽か」という情報を表現するのに 2 進数は親和性が高い。しかし 0 や 1 だけでなく, もっと幅の広い情報を各桁に入れたいと思うときがあるかもしれない。この場合は, 複数のビットの塊として情報を表現すればよい。

例えば, 太郎君, 次郎君, 三郎君が昼食に食べたパンの個数(0 個以上 3 個以下)につい

て，次のように 9 ビットの 2 進数で表現できる。
　右から 1,2 ビット目が太郎君の食べたパンの個数を表す。
　右から 3,4 ビット目が次郎君の食べたパンの個数を表す。
　右から 5,6 ビット目が三郎君の食べたパンの個数を表す。
これによって，111010 は，「太郎君は 2 個，次郎君は 2 個，三郎君は 3 個のパンを食べた」，111111 は，「3 人とも 3 個パンを食べた」のように情報を表すことができる。

最後に，ビットの数によって何通りの情報が保持できるのかを考えてみよう。3 ビットの場合をまず見てみる。愚直に数え上げると，000,001,010,011,100,101,110,111 の 8 通り。計算で求めるなら，各桁に対して 0 か 1 かの 2 通りがあり，それが 3 ビットあるのだから，2^3=8 通りとなる。
一般に，n ビットの 2 進数には 2^n 通りの情報が表現可能であることが分かるだろう。

いままで使ってきた整数型，文字型，実数型はみんな 2 進数で表現されている。しかしその表現方法やビットの数がそれぞれ異なる。このあたりはやや込み入った話になるため，割愛する。

3. 変数のサイズ
ビットは情報の表現できる範囲を表すから，ビットの数はデータの容量(サイズ)を意味する。8 つのビットをまとめて 1 バイトという。つまり，1 バイトで 2^8=256 通りの情報が表現できる。
変数にはサイズがある。例えば，int 型で 100000000000000000 などのとても大きな数は表現できない。
変数のサイズはビット/バイトで表現する。実はサイズは C 言語のコンパイラによって様々なのだが，その一例を示す。筆者の環境では以下のようになっている。

型	サイズ(バイト)
`char`	1
`int`	4
`long long`	8
`double`	8

例えば，int 型の変数のサイズは 4 バイト=32 ビットだから，2^{32} 通りの情報を表現できる。実際，負号も表現できることに注意すると，-2^{31}-1 から 2^{31} までの値が表現できる。

4. 符号なし変数
少し話はそれるのだが，後の説明のために，符号なしの型について説明する。
通常，int 型は符号付き整数値を表す。つまり，負数も表現できる。char 型が符号を持つ

か持たないかは処理系による。

unsigned という単語を変数の int または char 型の前につけることで，符号なし変数を宣言できる。例えば，符号なし整数型の変数 x を宣言するときは，次のように書けばよい。

```
unsigned int x;
```

符号なし整数型の場合に限って，次のように int を省略して書ける。

```
unsigned x;
```

さて，符号なし整数型を宣言すると何が嬉しいのかというと，一つは「負号を表現する必要がない場合に，より大きな数が表現できる」という点である。例えば int 型の場合は$-2^{31}-1$ から 2^{31} までの変数が表現できたが, unsigned int 型の場合は負号の部分を整数として扱えるため, 0 から 2^{32} までの値が表現できる。

二つ目は，シフト演算子が正常に行える，という点である。詳しくは後述する。

符号付き整数型と符号なし整数型は，勿論どちらもビットを用いて表現されているのだが，その表現方法は異なる。このあたりの話についてはやや込み入った話になるため，割愛する。

さて符号付きと符号なしでは表現方法が異なるため，printf で出力するときもその違いを明示しなければならない。つまり，出力変換指定子が%d ではない。符号なし整数を printf で出力するときは, %u を用いる。

[補足]

符号なし実数型(unsigned double)なるものは存在しない。符号付きか符号なしかを決められるのは, int(あるいは long long int)と char のみ。

char が符号を持つかどうかは処理系に依存する。明示的に符号付き変数を宣言したい場合は，次のように型の前に signed をつければよい。

```
signed char a;
```

5. ビット演算子

ビットを操作するための演算子がある。どれも真偽における「かつ」や「または」と性質が似ていることに注意しよう。

まず, 1 ビットのみのビット演算について説明しよう。

(1) not(否定)

真偽における「否定」に相当する演算子。ビットを反転させる。

0 ならば 1, 1 ならば 0 を返す。

X	not X
0	1
1	0

このように，あるビットとその演算との関係を表したものを真理値表という。

(2) or(論理和)

真偽における「または」に相当する演算子。

二つのビットを比較して，二つのどちらか一方でも 1 なら結果として 1 を返す。

X	Y	X or Y
0	0	0
0	1	1
1	0	1
1	1	1

(3) and(論理積)

真偽における「かつ」に相当する演算子。

二つのビットを比較して，二つのどちらとも 1 なら結果として 1 を返す。

X	Y	X and Y
0	0	0
0	1	0
1	0	0
1	1	1

(4) xor(排他的論理和)

「どちらか一方のみ」を意味する演算子。

二つのビットを比較して，二つのどちらか一方のみが 1 なら 1 を返す。

X	Y	X xor Y
0	0	0
0	1	1
1	0	1
1	1	0

and, or, xor には結合/交換法則が成り立ち，例えば X xor Y = Y xor X, (X xor Y) xor Z=X xor (Y xor Z)である。

さて，これら(1)〜(4)の操作をビットの各桁について行うための演算子が存在し，それぞれ not 演算子, or 演算子, and 演算子, xor 演算子と呼ばれている。

C 言語での 4 つの演算子の記号は次のようになっている。

ビット演算子	記号
not X	~X
X or Y	X | Y
X and Y	X & Y
X xor Y	X ^ Y

例を示そう。次のプログラムは，入力整数値 x, y について, not x, x or y, x and y, x xor y の結果を 10 進法の符号無し整数値で出力するプログラムである。

```c
#include <stdio。h>

int main(void)
{
    unsigned x, y;

    scanf("%u %u", &x, &y);
    printf("%u¥n%u¥n%u¥n%u¥n%u¥n", ~x, x | y, x & y, x ^ y);

    return 0;
}
```

実行結果は次のようになる。

```
210（入力）
118（入力）
4294967085
246
82
164
```

unsigned int 型のサイズが 32 ビットだとする。すると，

$x = 210 = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101\ 0010_{(2)}$

$y = 118 = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111\ 0110_{(2)}$

である(見やすさのために 4 桁ごとに空白を入れている)。よって，

$\sim x = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 0010\ 1101_{(2)}$

$x \mid y = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1111\ 0110_{(2)}$

$x \& y = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0101\ 0010_{(2)}$

$x \hat{}\ y = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1010\ 0100_{(2)}$

となり，この 10 進数表記での結果が出力された。

not 演算子の場合は変数のサイズによって値が変わってしまうことに注意しよう。例えば，別の環境では unsigned int 型のサイズが 16 ビットなら，

$x = 210 = 0000\ 0000\ 1101\ 0010_{(2)}$

より，

$\sim x = 1111\ 1111\ 0010\ 1101_{(2)} = 65325$

となる。

次に，「ビットシフト」と呼ばれる操作を紹介する。これは，ビット桁を左や右にずらず操作で，それぞれ「左シフト」「右シフト」と呼ばれている。

10 進法の場合では,10 倍したり 10 で割ったりすると，桁が左や右にずれる。これは 2 進法でも同様で，ビットの値をずらすという操作は，元の値を 2 倍したり 2 で割ったりする操作と同等である。

実はシフトには「論理シフト」と「算術シフト」の二種類がある。算術シフトによって符号を維持しながら 2 倍したり 2 で割ったりできるのだが，詳しくは割愛する。C で確実に使えるのは論理シフトなので，論理シフトのみ焦点を当てて説明する。

(5) 右シフト

桁を右にずらす。ずらしたことによって左端に隙間ができるが，そこは 0 で埋められる。シフトによってはみ出たビットは消える。

（例）

11011001 を 3 ビット右シフトすると, 000**11011** となる。

(6) 左シフト

桁を左にずらす。ずらしたことによって右端に隙間ができるが，そこは 0 で埋められる。シフトによってはみ出たビットは無視される。

（例）

110**11001** を 3 ビット左シフトすると, **11001**000 となる。

さて,(5)(6)の操作を行うための演算子が存在し，それぞれ「右シフト演算子」「左シフト演算子」と呼ばれる。C 言語では以下のような記号でシフトを書く。

シフト演算子	記号
X を n ビット右シフト	X >> n
X を n ビット左シフト	X << n

基本的には，シフト演算子を利用するとき,X は符号なし変数,n は非負整数でないといけない。この条件を満たさなかった場合，正常な動作は保証されないので注意しよう。

さて，いままで紹介してきた演算子には複合代入演算子が存在する。単にいままでの演算子の後に=をつければ，複合代入演算子となる。例えば，「X を 2 ビット左にシフトしたものを X に代入」という操作は `X <<= 2` と書ける。

6. ビット演算の応用

ビットの応用は多種多様である。ここではそのうちの一部を紹介する。

[特定のビットを立てる]

or 演算を用いると，特定のビットを 1 にすることができる。

第 2 項で述べた「太郎君，次郎君，三郎君がある授業に出席しているか否か」を例にしてみよう。この情報を格納する変数を X とする。X は 0 で初期化しておく。

さて, X は次のようなルールで情報を表現している。

右から 1 ビット目が立っていれば，太郎君は出席している。

右から 2 ビット目が立っていれば，次郎君は出席している。

右から 3 ビット目が立っていれば，三郎君は出席している。

そこで, X に「太郎君が授業に出席している」情報を表現するためには，次のように書けばよい。

```
X = 1;
```

$4 = 001_{(2)}$ だから, X は右から 1 ビット目のみが立っている状態になる。ここに「三郎君が出席している」情報を付加するためには，それぞれ次のように書けばよい。

```
X |= 4;
```

|=は|の複合代入演算子だから，これは `X = X | 1` と同義である。$X = 001_{(2)}, 4 = 100_{(2)}$ の or 演算を取ることによって, $X = 101_{(2)}$ となる。

「太郎君と三郎君が出席している」という情報を表現する方法として，次のようにも書ける。

```
X = 1 | 4;
```

4 は「太郎君が出席している」ことを表す成分，1 は「三郎君が出席している」ことを表す成分である。これらの二つは独立な状態なので， or 演算で組み合わせて「太郎君と次郎君が出席している」を表現できる。

今回は状態の数が少ないため， 単に 4 や 2 をそのまま書いていたが，状態の数が多くなると，2 進数と 10 進数の変換をせずに，シフト演算子を用いて次のように「太郎君/次郎君/三郎君出席フラグ」を作っておいたほうが分かりやすくなる。

```
unsigned TARO = 1;
unsigned JIRO = 1 << 1; /*001 を 2 ビット左シフトして 010*/
unsigned SABURO = 1 << 2; /*001 を 2 ビット左シフトして 100*/
X = TARO | SABURO;
```

[特定のビットを消す]

and 演算は，「二つのビットを比較して，両方とも 1 なら 1 を返す」演算である。つまり，次の性質が成り立つ。

`m = 11111…111`$_{(2)}$ **(全ビットが 1 の数)としたとき，** `X & m = X`

もし二つのうち一方が 0 であるなら, 0 を返す。よって，次の性質が成り立つ。

`m = 0`$_{(2)}$ **(全ビットが 0 の数)としたとき，** `X & m = 0`

この性質を利用すると，ある数の任意のビットを 0 にすることができる。

例えば，太郎君の出席を取り消したいときは, not と and を用いて次のように書く。

```
X &= ~TARO;
```

複合演算子を使っており，これは `X = X & ~TARO` と同義である。not 演算によって，TARO が出席しているかどうかを表すビットが 0 になり，他の全てのビットは 1 になる。実際, ~TARO = 111… $11\underline{\mathbf{110}}_{(2)}$となる。and 演算は，一方が 0 なら 0 を返すのだから, X と~TARO の and 演算を取ることによって, X 右から 1 ビット目のみを 0 にし，他のビットはそのまま保持することができる。

「太郎も三郎も欠席」という情報に変えたい場合は次に書けることも分かるだろう。

```
X &= ~(TARO | SABURO);
```

TARO | SABURO = $101_{(2)}$だったから, ~(TARO | SABURO) = 111…$11\underline{\mathbf{010}}_{(2)}$である。右から 1 ビット目と 3 ビット目が 0 になっているので，これと X で and 演算をとることによって, X の 1,3 ビット目を 0 に変えることができる。

[特定のビットを取り出す]

特定のビットを取り出すためには，そのビットが下位 1 ビット目に来るようにシフトした後に, 1 と and 演算をとればよい。

例えば，三郎が出席しているかどうかを見るときは，次のように書く。

```
(X >> 2) & 1
```

三郎が出席しているなら 1，出席していないなら 0 が返る。

三郎が出席している状態，例えば X = $101_{(2)}$だったとする。X>>2=$001_{(2)}$だから，これと 1 で and 演算をとれば 1 が返ってくる。出席していない状態，例えば X=$010_{(2)}$だったとする。X>>2=$000_{(2)}$だから，これと 1 で and 演算をとれば 0 が返ってくる。

返ってくるのは計算結果なので，例えば if 文と組み合わせて次のように書ける。

```
if ((X >> 2) & 1) {
    三郎が出席していた場合の処理
}
```

[ビットをずらしながら探索]

特定のビットを取り出す方法を応用すると，ループ文を使って，すべてのビットが 1 か 0 かを調べることができる。

例を示そう。太郎君・次郎君・三郎君の出席の情報を整数値(0 以上 8 未満)として入力する。出席しているなら o，していないなら x を，太郎君・次郎君・三郎君の順に表示するプログラムである。

```
#include <stdio.h>

int main(void) {
    unsigned x;
    int i;
```

```
    scanf("%u", &x);
    for(i = 0; i < 3; i++) {
        if((x >> i) & 1) printf("o ");
        else printf("x ");
    }
    printf("¥n");

    return 0;
}
```

7. xor の性質

最後に, xor 演算の細かい性質について述べる。

xor は「2 つのビットを比較して，どちらか一方のみが 1 なら 1 を返す演算子」である。つまり，「2 つのビットを比較して，どちらとも同じ数なら, 0 を返す」ことになる。このことから次の性質がいえる。

(1) X xor X = 0

次の視点を持つことも大切である。

(2)1 ビットの数 X があって，X と 0 で xor をとると X はそのまま，X と 1 で xor をとると X のビットが反転する。

これは, xor の真理値表を次のように並び替えてみれば分かる。

X	Y	X xor Y
0	0	0
1	0	1
0	1	1
1	1	0

このことから，次の性質がいえる。

(3) X xor 0 = X

(4) X xor $1111\cdots11_{(2)}$ = not X

(4)の性質を上手く利用すると，特定のビットを反転させることができる。例えば

$0100\ \underline{1001\ 0010}\ 0110_{(2)}$ xor $0000\ \underline{1111\ 1111}\ 0000_{(2)} = 0100\ \underline{0110\ 1101}\ 0110_{(2)}$

さらに，次が成り立つ。

(5) (X xor Y) xor Y = X

(6) (X xor Y) xor X = Y

これは次の真理値表を見れば分かるし, (3)と結合/交換法則からも導ける。

X	Y	X xor Y	(X xor Y) xor X	(X xor Y) xor Y
0	0	0	0	0
0	1	1	1	0
1	0	1	0	1
1	1	0	1	1

性質(5)(6)は，

xor によって 2 つの値 1 つの値として保持できる。2 つの値が保持されている数を X とすれば，一方の値を取り出すためには，もう一方の数と X とで xor をとればよい。

ということを意味する。

この面白い例として，「変数 x と y の値を入れ替える」処理を示そう。まず，普通に書くなら，次のように一旦値を退避しておくための変数 tmp を利用して，次のように書く。

```
tmp = x;
x = y;
y = tmp;
```

xor を巧みに使うと，次のように書くことができる。

```
x ^= y;
y ^= x;
x ^= y;
```

このままでは非常に分かりづらいので，複合代入演算子を使わない形に書き換える。さらに，変数 x の値を a，変数 y の値を b とする。=を代入演算子としてでなく，等式として式の流れを詳しく見てみると，次のようになる。

```
x = a
y = b
x = x ^ y = a ^ b
y = x ^ y = (a ^ b) ^ b = a
x = x ^ y = (a ^ b) ^ a = b
```

最後の 2 行で，性質(5)と(6)を用いている。

この変数入れ替え処理は非常に面白いのだが，初見の人はいったい何をやっているか分からない。ソースコードの読みやすさのため，このようなトリッキーなコードは書かない方が良いと思われる。