2.3 大きな数の扱い方

指数関数の計算をしてみると分かることだが、計算結果は爆発的に大きくなってしまう。一般にコンピュータでは、ひとつの変数が扱える整数は数万程度の大きさしかないから、すぐに桁あふれを起こしてしまうのだ。$y = 2^x$ といった単純な関数でさえ、コンピュータはついて行けない。$x = 20$ でもコンピュータは悲鳴をあげるだろう $x = 40$ ともなればお手上げだ。

コンピュータで大きな数を扱うことは不可能なのだろうか？　もちろんそんなことはない。解決のヒントは、私たちが何気なく使っている数に隠れている。私たちは機械と違って、数を記憶するための制限を持たない。だから、10 億でも 1 兆でも扱える。本当にそう思っている？　実は私たちのほうがコンピュータより貧弱だ、と言ったらどう考えるだろうか。だが、これは事実なのである。コンピュータはひとつの変数で数万程度の大きさしか扱えないと言ったが、私たちはひとつの桁で 9 までの数しか扱えないことを知っているだろうか。

そろそろ私の言いたいことが見えてきたことと思う。つまり、こういうことだ。私たちは 0〜9 までの 10 種類—今後のためにあえて言うが、1〜0 までの 10 種類ではない—の数しか扱えないが、位取りをしているために望むだけ大きな数を扱えるのである。これをコンピュータにも応用しよう。ひとつの変数で数万の数しか扱えなくても、位取りの考えを持ち込むことで大きな数を扱うことができるのである。普通コンピュータは 2 進数で演算していると言われるが、ひとつの変数をひとつの桁にすれば数万進数で演算することも可能なのだ。私たちが貧弱と言ったのは、そういう意味である。

では、指数関数の計算に入る前に**配列**について話しておこう。

programming list [array.cpp]

```
1: #include <iostream>
2:
3: int main() {
4:     int f[3];
5:     f[0] = 2, f[1] = 5, f[2] = 6;
6:     std::cout << "  a first num : " << 100*f[0] + 10*f[1] + f[2]
                                                          << std::endl;
7:     std::cout << "a reverse num : " << 100*f[2] + 10*f[1] + f[0]
                                                          << std::endl;
8:
9:     return 0;
10: }
```

配列による変数を用意するには `f[]` や `array[]` のように、変数名に `[]` を付ける。`f[3]` と書いたら 3 つの変数が、`array[10]` と書いたら 10 個の変数が用意されるのだが、注意しなくてはならないことがある。それは、`f[3]` で用意される 3 つの変数は `f[0]`, `f[1]`, `f[2]` の 3 つであり、`f[10]`

で用意される 10 個の変数は f[0], f[1], ..., f[9] の 10 個である点だ。つまり A[n] という配列は、A[0] から A[n-1] までの n 個の変数になるのである。このことは、n 進法で 0 から $n-1$ までの n 個の数が使われることと合致している。さっき 0〜9 までの 10 種類の数を扱うと強調したのは、ここでの説明を見越してのことだ。

さて、プログラムは 3 桁の整数を模している。4:行目の f[3] によって 3 桁分の "位" を用意した。位というのは少し変だが、この場合の処理ではそうしている。つまり f[0] の位、f[1] の位、f[2] の位が用意されたことになる。ちなみに、変数名の f[] は "数字の桁（a figure）" の意味で用いた。でも、d[]（a digit）のほうがよかったかもしれない。

5:行目で位に使われる数字をそれぞれ 2, 5, 6 にしたところだ。だからと言ってこの時点で、256 の数ができたわけではないことを注意しておこう。

6:行目の std::cout で 3 桁の数を表示しているが、f[0], f[1], f[2] の順に百の位、十の位、一の位としている。従って、ここではじめて 256 という数ができたことになる。

7:行目の std::cout でも 3 桁の数を表示しているが、今度は各位を逆順にした数になっていることに気付くだろう。従って、ここでは 652 という数ができたのだ。

プログラムはこれだけだが、このネタをもう少し調理しよう。

2 進数というのはコンピュータではよく使われる。1001011 や 11111011 といった数のことだ。私たちは普段 10 進数を使っているので、これらの数が 77 や 251 だということに気付かないものだ。コンピュータは 1 バイトという単位を使うが、1 バイトは 8 ビットである。え？　何を言ってるの分からないって？　つまりこういうことだ。1 ビットとは 0 か 1 だけが使える桁のことである。だから 8 ビットとは 0 か 1 が使える桁が 8 つあることを指している。そしてこれが 1 バイトになる。要するに 1 バイトで 00000000〜11111111 までの数が扱えるのだ。プログラム例は 1 バイトの 2 進数がいくつになるかを調べるものだ。

programming list [bintodec.cpp]

```
 1: #include <iostream>
 2:
 3: int main() {
 4:     int f[8], dec;
 5:     f[0] = 1, f[1] = 1, f[2] = 1, f[3] = 1, f[4] = 1,
                                                f[5] = 0, f[6] = 1, f[7] = 1;
 6:     dec = 2*(2*(2*(2*(2*(2*(2*f[0]+f[1])+f[2])+f[3])+f[4])+
                                                        f[5])+f[6])+f[7];
 7:     std::cout << "decimal : " << dec << std::endl;
 8:
 9:     return 0;
10: }
```

プログラムは一部が美しくないけれど、正確な値を出してくれる。美しくないのは 5:行目で、ご

丁寧にもひとつひとつの配列変数に代入している。この代入によって 2 進数の 11111011 がセットされたことになる。実はこのセットの仕方も美しくない原因になっているのだが気にしないでおこう。

さて、10 進数なら十の位が 10^1、百の位が 10^2、千の位が 10^3、... である。2 進数では十の位とは呼ばないので、下の位から順に 2^0 の位、2^1 の位、2^2 の位、... である。従って 11111011 を 10 進数にするには $2^7\mathrm{f}[0]+2^6\mathrm{f}[1]+\cdots+2^0\mathrm{f}[7]$ の計算—ほらね、美しくないでしょ。だって指数と配列番号が一致してないもの—をすればよい。しかしコンピュータはべき乗の計算を得意としていないのだ。だから $2^7\mathrm{f}[0]+\cdots$ の計算をさせたければ、`dec = 2*2*2*2*2*2*2*f[0]+` $\cdots$ と書かなくてはならない。

そこで 6:行目のような、中途半端に 2 をくくり出した式を使うことになるのだけれど、これがコンピュータにとって実に好都合なのだ。

EX. `2*(2*(2*(2*(2*(2*(2*f[0]+` $\cdots$ `)+f[7]` が `2*2*2*2*2*2*2*f[0]+` $\cdots$ `+f[7]` に等しいことを確認せよ。

`2*(2*(2*(2*(2*(2*(2*f[0]+` $\cdots$ `) +f[7]` と `2*2*2*2*2*2*2*f[0]+` $\cdots$ `+f[7]` を較べてほしい。掛け算をする回数がかなり減っていることが分かるはずだ。実際、前者は 7 回の掛け算で済んでいるが後者は 28 回も掛け算をしている。コンピュータは掛け算に時間がかかってしまうものである。この程度の計算では恩恵が目立たないが、6:行目に使われた式のほうが、掛け算の回数が少ないから計算時間が短縮されるのだ。

TRY! 5 進数 $\to$ 10 進数にするプログラムにしてみよ。

ところで話題をプログラム `[bintodec.cpp]` に戻すことにしよう。ここでは 2 進数の 11111011 を 10 進数に直しているが、実行すると 251 が返ってくるはずだ。だが、一般に `int` 型が `unsigned` でなければ、最高位の 1 は負の数であることを示している。そのため `int` 型の 11111011 は -5 を表すのだ。8 桁では分かりにくいだろうから、4 桁の 2 進数を例にとろう。

最高位が 0 か 1 によって正の数と負の数の区別がつくので、0111 は正の数で 1111 は負の数である。しかし 1111 は、正の数 0111 にマイナスの意味で最高位の 1 を付けているわけではない。1111 は -1 である。なぜなら $1111+1=0000$ だから、$1111=-1$ であることが分かるというものだ。2 進数は 1111, 1110, 1101, 1100, 1011, ... という具合に減るから、これらは 10 進数で -1, -2, -3, -4, -5, ... を表している。

もし 4 桁の 2 進数が用意されれば、全部で $2^4=16$ 通りの数が表せる。その内、0xxx で表される正の数が 8 通り、1xxx で表される負の数が 8 通りだ。0xxx の正の数には 0 も含まれるので、正

の数は 0 から 7 までだが、1xxx の負の数は -1 から -8 までとなる。お分かりかな？　よって 4 桁の 2 進数が扱える数は -8 から 7 までである。

ちょっと前に、2 バイト—これは 16 桁の 2 進数に相当している—なら $256^2 (= 2^{16}) = 65536$ 通りの数が使えるから、-32767〜32767 の整数が扱えると言った。混乱を避けるためにそう言ったのだが、65536 通りの半分は 32768 通りである。それを正負の数が使うけれど、正の数には 0 が含まれているので、実際に扱える数は -32768〜32767 なのだ。