7.3 循環節の謎に近寄る

私たちは、計算可能な循環節を手に入れることができた。[paracalc.cpp] に手を加えて、掛け算によって循環節が巡回する様子を見てみよう。まずはプログラムの提示からだ。

programming list [cyclical.cpp]

```
#include <iostream>

int p[1000], q[1000];

void disp(int *q, int c, int n) {
    printf("(%2d) ", n);
    while(c--) {
        std::cout << *q++;
    } std::cout << std::endl;
}

void recurf(int p[], int c, int n) {
    int i;
    for(i = 0; i < c; i++) {
        q[i] = p[i] * n;
    }
    for(i = c-1; i >= 0; i--) {
        q[i-1] = q[i-1] + q[i] / 10;
        q[i] = q[i] % 10;
    }
}

int main() {
    int b, r = 1, cyc = 0, num = 1;
    do {
        std::cout << "input b of 1/b : "; std::cin >> b;
    } while(!(b % 2) || !(b % 5));
    do {
        p[cyc++] = (r * 10) / b;
        r = (r * 10) % b;
    } while(r != 1);
    do {
        recurf(p, cyc, num);
        disp(q, cyc, num);
    } while(num++ < cyc);

    return 0;
}
```

ちょっと手を加えるつもりが、これまでにない長さのプログラムになってしまった。原因は、掛け算を配列でしなくてはならないことにある。以前やったことと同じなのだが、当時は常に 2 だけを掛けていたことを覚えているかい？　それに対して今回は、分母が 999 の整数なら、最悪 998

を掛ける必要に迫られる。そのため処理が複雑になった。まあ、とやかく言うよりも順に見ていこうじゃないか。

プログラムはいきなり 3:行目の `int p[1000], q[1000];` で始まっている。今までは変数や配列の宣言は、必ず関数内部でなされていたはずだ。なのに、ここでは関数の外部で宣言されている。どう違うのだろう。

まず、関数内部で用意される変数は**ローカル変数**と呼ばれる。ローカルとは "局所的" と考えてもらっていい。つまりは関数内部でのみ有効な変数だ。プログラムが動いているときは、大抵そのとき必要な関数が呼ばれている。そして、別の関数が必要になれば、一旦今の関数は用なしになる。このとき同時に変数も用なしになる。この仕組みのお陰で、同じ変数名であっても、使う関数が違えば平気で使って構わないのである。

一方、関数の外部で用意される変数は**グローバル変数**と呼ばれる。グローバルとは "大域的" ということだ。別の観点では、グローバル変数はどの関数からも参照できる変数ということになる。だから、どの関数が使われていようとも、グローバル変数は用なしにならない。今回のプログラムは、いくつかの関数で配列の値を共有したいので、それらを関数外部で用意した次第である。

EX. グローバル変数はどこからでも参照できて便利に思える。しかし、どの関数からでも参照できるとなると不都合も生じる。何が不都合なのか、ローカル変数との兼ね合いで考えてみよ。

まず、`main` 関数に目をやることにしよう。今回の `main` 関数で `[paracalc.cpp]` の `main` と違う部分は、33:行目の関数 `recurf()` だけである。もちろん他にもいくつかの違いが目に付くが、それらは `recurf` 関数のせいで必要になっているだけだ。例えば 24:行目の `num = 1` がそうであり、また、33:–34:行目が `do{}while()` 文で囲まれているのがそうである。

`[cyclical.cpp]` では、ただひとつの循環節を表示するだけでなく、そこに一定の数を掛けた状況を示したい。一定の数とは、1 から循環の回数までのすべての数である。1/17 なら循環節は 16 なので、1 から 16 までの数を掛けた状況が示されるようにするのだ。そのための `do{}while()` 文であるが、35:行目の条件がそのことを表している。

33:行目の `recurf` 関数は、掛け算をするための関数である。そのためには、29:行目で得た `p[0]` のデータ—といっても、`p[0]` の値が格納されているアドレスだ—と、循環がどれぐらい続くかのデータ `cyc`、それに今いくつの数を掛けているかを示す値 `num` を渡してあげなければならない。これらがきちんと渡せれば、あとは `recurf()` が処理をしてくれる。

では、`recurf()` が何をしているか探っておこう。値はポインタではなく、配列として受け取っているけど、どちらが分かりやすいかは各自の好みにもよるだろう。

`p[0]` のアドレスと循環回数と掛ける数を受け取った `recurf` 関数は、14:–16:行目にかけて、各配列を定数倍している。そして定数倍した値を、配列 `p` ではなく `q` に代入している。なぜそうし

ているのかといえば、配列 p は定数倍するための基準値として固定しておきたいからだ。ちなみに、配列 p, q は int 型で、今のところ 1,000 桁の循環までに対応しているので、掛けられる定数は 1,000 未満だ。よって、桁あふれの心配はない。

しかし、結果の表示は 1 桁ずつだから、余分な値は上の配列に繰り延べていかなくてはならない。それが 17:–20:行目の処理だが、この方法はすでに登場した [powerof2.cpp] の焼き直しに過ぎない。

EX. [powerof2.cpp] では桁をまたぐ処理が p[i+1] = p[i+1] + p[i] 云々だったが、ここでは q[i-1] = q[i-1] + q[i] 云々である。配列の繰り延べ方が逆になっている理由を考えよ。

実は 14:–20:行目までの 2 つの for 文は、ひとつにまとめて書くことが可能だ。だが、プログラムは短ければよいという代物ではない。多少長めでも、処理が自然に追えるほうが分かりやすいものだ。

関数 recurf() が void 型なのは、この関数が掛け算の結果を配列に納めるだけだからである。値を返す代わりに、グローバル変数を使って、別の関数に処理を委ねるのだ。

処理を委ねられたのが disp 関数だ。本当は [paracalc.cpp] での disp 関数と同じように、disp(q, cyc) だけでよかったのだけれど、掛けている数を表示させたいために num の値も渡している。そのため、while 文に入る前に、6:行目によって掛けている数の表示を済ませている。7:–9:行目の処理内容は、改行させている以外は同じである。std::cout << std::endl; を disp 関数内に書いたのは、単に気分の問題である。

TRY! 早速 1/17 や 1/61 などの循環節について調べてみよ。1/17 や 1/61 は 1/7 と同じ性質を持つが、他の分数はどんな性質になっているだろうか。

TRY! プログラムでは、小数点以下第 1 位の桁が p[0]、第 2 位の桁が p[1]、... というように、桁番号と配列番号がずれて格納される。24:行目の cyc = 0 を cyc = 1 で始めておけば、小数点以下第 1 位の桁が p[1]、第 2 位の桁が p[2]、... となって分かりやすい。ただし、そのようにすると変更しなくてはならない箇所がたくさんある。勇気があれば、もれなく変更をした上でプログラムを走らせてみよ。