1.4 コラッツの問題

有名な数列はフィボナッチ数列だけではない。フィボナッチ数列は不思議な数列だが、他にも未だ未解決の数列がある。それは次の規則で作られる。

始めに勝手な自然数を用意する。次の項は『前の項が偶数なら 2 で割り、奇数なら 3 倍して 1 を足す』というものだ。例えば 50 から始めると

$$50,\ 25,\ 76,\ 38,\ 19,\ 58,\ 29,\ 88,\ 44,\ 22,\ 11,\ 34,\ \ldots$$

のような数列が出来上がる。数列はどこまでも続くように思える。しかし、そうではないのだ。簡単なプログラムを組んで、いろいろな数で試してみよう。

programming list [collatz.cpp]

```
 1: #include <iostream>
 2: #define SUP 25
 3:
 4: int main() {
 5:     int n, i;
 6:     std::cout << "input a seed numnber : "; std::cin >> n;
 7:     std::cout << n << " ";
 8:     for(i = 1; i <= SUP; i++) {
 9:         if(n % 2 == 0) {
10:             n = n / 2;
11:         } else {
12:             n = 3 * n + 1;
13:         }
14:         std::cout << n << " ";
15:     }
16:     std::cout << std::endl;
17:
18:     return 0;
19: }
```

ここではコンピュータプログラムにおける、重要な分岐処理を理解してほしい。コンピュータは繰り返し同じことをするのに苦痛を感じないが、思い通りの処理を自動的にしてくれるほど気が利いているわけではない。コラッツの問題では、偶数と奇数の判断をさせなくてはならない。

プログラムは 5:行目で、必要な変数 `n` と `i` を用意している。整数しか扱わないので `int` 型だ。

キーボードから任意の数を読み込むために 6:行目で `std::cin` を用いている。もちろん、何を入力すればよいか知らせるために、直前で `std::cout` を用いて入力を要請していることに注意しよう。

7:行目で、入力された数を `std::cout` で表示している理由は、入力直後に 2 で割ったり 3 倍して 1 足す操作を始めると、入力した数は捨てられ、新たな数が `n` に代入されてしまう。だから、始

めの数が捨てられる前に表示させたに過ぎない。入力した数を見る必要がないと思えば、この行は不要である。

8:–15:行目が for 文で囲まれているのは、2 で割るか 3 倍して 1 足す操作を繰り返す必要があるからである。従って、偶数と奇数の区別は for(){}の内部に書かれた部分でしていることになる。

ここから新たな命令の登場だ。分岐処理は if 文で行われる。if 文の仕組みは、() 内に記述された条件に合致すれば直後の {} 内の処理が行われ、条件に合致しなければ else 後の {} 内の処理が行われるのである。このプログラムでは {} 内は単文であるが、複数の文を書くこともできる。また、単文は {} で囲む必要はないのだが、対応を明確にする目的と、後で複文になってもいいように、冗長になることを承知で {} を書いている。

さて、if() に記述された条件であるが、これは「n を 2 で割った余りが 0 に等しい」と読んでおく。前に話した通り n % 2 は n を 2 で割った余りを求める計算式である。== と等号が連（つら）なっていることに注目してほしい。今は左辺と右辺が等しいかどうかを調べているのだが、数学の感覚で n % 2 = 0 などと書いてしまうと、とんでもないことになる。n % 2 = 0 と書くとコンピュータは、n を 2 で割った余りに 0 を代入しようとして混乱することになるだろう。慣れるまでは = と == の使い分けには注意が必要になるものだ。

そして、n が 2 で割れれば、コンピュータは直後の {} 内の n = n / 2; を実行し、else{}は実行しないで次の std::cout へと移る。また、n が 2 で割れなければ、コンピュータは n = n / 2; の文を実行しないで、else{}内の n = 3 * n + 1; を実行して次の std::cout へと移る。つまり二者択一で処理が行われるのである。

この部分が for(){}で囲まれているので、最初に設定した上限 SUP まで繰り返されるのだ。

TRY! “input a seed number : ” に対して色々な数を入力して、どのような結果になるか確かめよ。

色々な数に対して、コラッツの問題の操作を繰り返せば、結果の予想がつくだろう。その予想は正しい。しかしコラッツの問題は予想であって証明ではない。現在でも、どうやら確からしいという程度の予測はできても、未だ証明は得られていない。ところで、SUP が 25 になったところでプログラムが終了するのでは、コラッツの操作が終了する前にプログラムが終わることもある。それでは困る。そこで上限を設けるのではなく、n が 1 になったところでプログラムが終了するように改良しておこう。

改良は実に簡単だ。

programming list [collatz2.cpp]

```
1: #include <iostream>
2:
3: int main() {
4:     int n, i;
```

```
 5:     std::cout << "input a seed numnber : "; std::cin >> n;
 6:     std::cout << n << " ";
 7:     while(n != 1) {
 8:         if(n % 2 == 0) {
 9:             n = n / 2;
10:         } else {
11:             n = 3 * n + 1;
12:         }
13:         std::cout << n << " ";
14:     }
15:     std::cout << std::endl;
16:
17:     return 0;
18: }
```

見てのとおり [collatz.cpp] の 7:行目、`for(i = 1; i < SUP; i++)` を `while(n != 1)` に変えただけだ。もちろん上限は必要ないので、`#define SUP 25` はなくしてある。

では、`while(n != 1)` は何をする命令だろうか。以前 `while(1)` で、永久に繰り返しをさせたことを覚えているね。永久に繰り返したのは `()` 内が `1` だからだ。この辺の詳しい話は後回しとさせてもらうが、今は `()` 内が `n != 1` である。実は `while` 文は、`()` 内の条件が**真**である限り `while(){}` の内容を繰り返すのである。そして `!=` は、左辺と右辺が等しくないことを意味する**関係演算子**で、数学なら $\neq$ を使うところだ。従って `while(n != 1)` は、`n` が `1` でない限り `{}` 内を繰り返す。逆の意味に取れば、`n` が `1` になったところで `while` 文が終わるのだ。

TRY! 改良したプログラムで、"input a seed number : " に対して 27 を入力してみよ。

TRY! コラッツの問題においては、7:行目の `while(n != 1)` は `whlile(n > 1)` の方がよい。なぜか？　理由がわからないときは、`while(n != 1)` のプログラムで極端に大きい数を入力してみよ。何が起こるか？

コラッツの問題を体験するために用意したプログラムは、変数に `int n` を用いている。決して `unsigned` ではない。`unsigned` を気楽に用いた場合、**桁あふれ**の対処に問題が生じる。コンピュータにとっては桁あふれは避けて通れないものだ。それは、変数が `unsigned` であってもなくても変わらない。変数に正・負の区別があれば桁あふれを把握しやすいが、そうでないと困る例がある。

簡単のため、`int` なら -50〜50、`unsigned` なら 0〜100 だけが扱えるとして話を進めよう。このとき、正の値をとる数列が、計算によって $20 \to 72 \to 121 \to 95 \to \ldots$ と変化すると仮定すれば、`int` では 72 で桁あふれを起こし、おそらく -29 と解釈される。一方、`unsigned` では 121 で桁あふれを起こすが、おそらく 20 と解釈されてループに陥る。いずれの場合にもプログラムの修正が必要なことは明らかだが、桁あふれとループでは対処の仕方が違ってくる。

少々作り過ぎの例だが、あながち的はずれではない。