10 . . . の旅

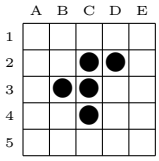
10.1　ライフゲーム

『十年ひとむかし』という言葉がある。いまや時代の変化は早いというものの、“昔” をしのぶには 10 年でも短いぐらいだ。あるものが一般に浸透し始めたと思えたら、あるものは間違いなくひと昔もふた昔も前に産声をあげている。「それ、10 年前にはなかったよね。」っていうのは、おそらく世間に疎（うと）い人が言うことばだ。

ライフゲームというシミュレーションがある。「なにそれ、10 年前にあった？」と言いたくなるかもしれないが、ライフゲームはコンウェイ[1]が 1970 年に考案したセル・オートマトン[2]的なゲームだ。10 年どころじゃないよね。たっぷり半世紀以上前だ。ライフゲームはコンピュータと親和性が高い。もちろん当時もコンピュータはあったけれど、計算速度は現在よりはるかに遅く、リアルタイムに表示できるディスプレイはなかった。じゃあ、どうやってシミュレーションしたのかって？　もちろん紙に出力したのだ。

Terminal を使えば、当時の雰囲気が味わえる。この地では、ライフゲームに取り組んでみよう。と言っても、ほんの小さなライフゲームもどきを作るだけである。Terminal では画面で動きのあるシミュレーションはできないので、紙に出力するように、変化を順次表示させるのだ。

その前に、ライフゲームについて簡単に説明しておこう。



ライフゲームは、格子状のマスに置かれた “石” や “空きマス” を、周りの状況によって変化させるシミュレーションである。その際の規則は以下の 3 種類だけである。

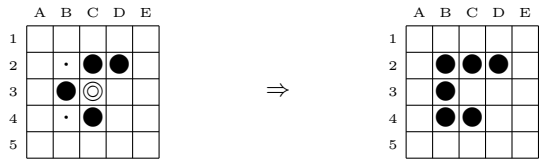
- 石があるマスの周囲 8 マスについて、周囲に石が 1 個以下、もしくは石が 4 個以上のときはマスの石は取り除かれる。たとえば 3C のマスにある石は、周りに 4 個の石があるので取り除かれる対象である。

[1] ジョン・ホートン・コンウェイ (1937–2020)：イギリスの数学者。
[2] cellular automaton とも言う。セル（格子状のマス）と単純な規則による、離散的計算モデル。

- 石があるマスの周囲 8 マスについて、周囲に石がちょうど 2 個、もしくは石がちょうど 3 個のときはマスの石はそのまま残る。たとえば 4C のマスにある石は、周りにちょうど 2 個の石があるので残る対象である。

- 石がないマスの周囲 8 マスについて、周囲に石がちょうど 3 個のときはマスに新たに石が置かれる。たとえば 4B の空いたマスは、周りにちょうど 3 個の石があるので新たに石を置く対象である。

ただし、石を取り除いたり新たに置いたりするのは、すべての石と空きマスを調査したとき同時に行うものとする。さて上の図において、1A から 5E までの 25 マスについて規則を当てはめてみよう。取り除く対象の石は “◎” に、残る対象の石は “●” のままに、新たに石を置く対象のマスは “・” にすると



のように変化するのである。そして、次の配置においても同じ規則を当てはめていくと、次々と配置が変化していく。

最初の置き石の状態によって、ライフゲームの盤面の状況は多彩である。あるときは平面に石が一つもなくなって終了するかもしれない。また、あるときは同じ配置が繰り返されるようになるかもしれない。場合によっては、際限なく石が増えることもある。

ライフゲームは、規則を変えれば様々な亜種ができることだ。しかし、周囲に石がちょうど 3 個のときにマスに新たに石が置かれる規則が、絶妙な規則になっていることは興味深い。

いきなりライフゲームの完成形にあたるプログラムを提示するのは大変なので、順を追って見ていこう。

programming list [Life1st.cpp]

```
#include <iostream>

class Lifegame {
    int p[6][6];

public:
    Lifegame(int s[][6], int rows) {
        for(int x = 0; x < rows; x++) {
            for(int y = 0; y < 6; y++) {
                p[x][y] = s[x][y];
             }
        }
```

```
13:     }
14:
15:     void disp() {
16:         for(int x = 1; x < 5; x++) {
17:             for(int y = 1; y < 5; y++) {
18:                 std::cout << p[x][y];
19:             }
20:             std::cout << std::endl;
21:         }
22:         std::cout << std::endl;
23:     }
24: };
25:
26: int main() {
27:     int s[][6] = {{0, 0, 0, 0, 0, 0},
28:                   {0, 0, 1, 0, 0, 0},
29:                   {0, 0, 0, 1, 0, 0},
30:                   {0, 1, 1, 1, 0, 0},
31:                   {0, 0, 0, 0, 0, 0},
32:                   {0, 0, 0, 0, 0, 0}};
33:     Lifegame cells = Lifegame(s, 6);
34:
35:     cells.disp();
36:
37:     return 0;
38: }
```

なにもクラスを使うことはなかったのだけれど、万が一にも君たちが Terminal ライフゲームを育てて楽しもうとするなら、専用のクラスがあった方がよいかと思ったのだ。ただし、ここではクラスとしては不十分なものしかできない。本気で役立つクラスにするなら、汎用（はんよう）的な仕様にしなくてはならない。

このプログラムで何をしたいかというと、初期配置を設定して、それを画面で見ることである。規則による変化はあとで仕込むことにする。

まず、`main()` 中 27:–32:行目で初期配置を設定している。`int s[][]` は 2 次元配列と呼ばれる記述で、たとえば `s[4][6]` と書けば、4 要素の 1 次元配列—つまり $\{0,\ 1,\ 2,\ 0\}$ のようなもの—が 6 要素集まって

$$\begin{array}{cccc} 0 & 1 & 2 & 0 \\ \cdots & & & \\ \cdots & & & \\ \cdots & & & \\ \cdots & & & \\ 4 & 1 & 2 & 6 \end{array}$$

のような縦横のデータを想定している。`int` 型なので、要素は整数値に限られる。

27:–32:行目はデータの格納状況が一目で分かるように、インデントの位置を揃えて書いてある。これを 33:行目のように `Lifegame` コンストラクタで初期化するのだが、`s[6][6]` ではなく、`s[][6]` として `Lifegame(s, 6)` のように二度手間と思える方法で行っていることに注目してもらいたい。なぜそうするかというと、引数は 1 次元で渡す必要があるからだ。そして、8:–12:行目で 1 次元配列を 6 層分積み上げる。

2 次元配列の引数の渡し方の詳細は各自で調べてもらおう。とりあえず、これで 6×6 のマスに `1` と `0` が設定され、以下の状態を表したことになった。

-	-	-	-	-	-
-		●			-
-			●		-
-	●	●	●		-
-					-
-	-	-	-	-	-

図で "-" を打ったマスは空きマスなのだが、一般の空きマスと区別するためにそうしている。区別した理由はあとで説明することになる。図の配置はライフゲームでは主役級の "グライダー" と呼ばれる形である。規則を一回適用することを "1 世代進む" と表現すると、グライダーは世代が進むとまさに滑るように移動する。

EX. ライフゲームの練習として、手作業で 4 世代先まで変化させてみよ。グライダーの名前に恥じない結果を目にするだろう。

言い忘れたが、35:行目で画面への出力を単に `disp();` ではなく `cells.disp();` としているのは、`disp()` 関数が `Lifegame` クラスのメンバ関数の一つの機能として働くからだ。前に複素数周りを周遊したとき、`Complex` クラスで宣言した `c` から、`c.real()` で複素数 `c` の実部を取得しただろう。それとは少し異なるのだが、使い方は似ている。役に立つクラスというのは、多くのプロパティやメソッドを持っているものである。