Principled Programming

Introduction to Coding in Any Imperative Language

Tim Teitelbaum

Emeritus Professor
Department of Computer Science
Cornell University

Cellular Automata

Introduction

We illustrate two-dimensional arrays, and enumerations over them, using the examples of Cellular Automata and the Game of Life.

Cellular Automata model the Universe as a rectangular grid of *cells*, each in a given *state*. Time progresses in discrete steps. On each clock tick, each cell simultaneously decides what state to enter based on its current state and the current states of its neighbors. Each cell makes its decision independently, but all cells follow the same rules.

The Game of Life is a particular Cellular Automaton that models birth and death.

Systematic top-down development of an entire program is illustrated. Deeply-nested **for**-statements in the code arise naturally as a consequence of stepwise refinement, but are readily understood.

Class `CellularAutomaton` models the notion of a Cellular Automaton, and its simulation.

```
/* A cellular automaton. */
class CellularAutomaton {

   } /* CellularAutomaton */
```

☞ **Aggregate the definitions of related variables and methods in a class.**

The simulation as a whole is implemented as method `main`.

```
/* A cellular automaton. */
class CellularAutomaton {
   ...
   /* Simulate a cellular automaton. */
   static void main() {
      ___________
      } /* main */
   ...
   } /* CellularAutomaton */
```

- ☞ **Program top-down, outside-in.**
- ☞ **Many short procedures are better than large blocks of code.**

A *class method* is defined within a class using the keyword **`static`**.

The simulation as a whole is implemented as method `main`.

```
/* A cellular automaton. */
class CellularAutomaton {
   ...
   /* Simulate a cellular automaton. */
   static void main() {
      ___________
      } /* main */
   ...
   } /* CellularAutomaton */
```

- ☞ **Program top-down, outside-in.**
- ☞ **Many short procedures are better than large blocks of code.**

Adopt the pattern that first initializes, then computes.

```
/* A cellular automaton. */
class CellularAutomaton {
   ...
   /* Simulate a cellular automaton. */
   static void main() {
      /* Initialize. */
      /* Compute. */
      } /* main */
   ...
   } /* CellularAutomaton */
```

☞ **Master stylized code patterns, and use them.**

Instantiate placeholders *Initialize* and *Compute* for the problem at hand.

```
/* A cellular automaton. */
class CellularAutomaton {
   ...
   /* Simulate a cellular automaton. */
   static void main() {
      /* Create the initial Universe and display it. */
      /* Simulate and display remaining generations. */
      } /* main */
   ...
   } /* CellularAutomaton */
```

☞ **Master stylized code patterns, and use them.**

Refine the specifications.

```
/* A cellular automaton. */
class CellularAutomaton {
   ...
   /* Simulate a cellular automaton. */
   static void main() {
      /* Create the initial Universe and display it. */
         Initialize();
         Display();
      /* Simulate and display remaining generations. */
      } /* main */
   ...
   } /* CellularAutomaton */
```

Refine the specifications.

```
/* A cellular automaton. */
class CellularAutomaton {
   ...
   /* Simulate a cellular automaton. */
   static void main() {
      /* Create the initial Universe and display it. */
         Initialize();
         Display();
      /* Simulate and display remaining generations. */
         for (generation=1; generation<=LAST_GEN; generation++) {
            NextGeneration();
            Display();
            }
      } /* main */
   ...
   } /* CellularAutomaton */
```

Initialize, Display, and NextGeneration are other class methods to be defined.

Refine the specifications.

```
/* A cellular automaton. */
class CellularAutomaton {
   ...
   /* Simulate a cellular automaton. */
   static void main() {
      /* Create the initial Universe and display it. */
         Initialize();
         Display();
      /* Simulate and display remaining generations. */
         for (generation=1; generation<=LAST_GEN; generation++) {
            NextGeneration();
            Display();
            }
      } /* main */
   ...
   } /* CellularAutomaton */
```

Top-level Code Structure

Create stubs for the methods that have been introduced, which you can do mindlessly.

```
/* A cellular automaton. */
class CellularAutomaton {
   ...
   /* Create the initial Universe. */
   static void Initialize() { } /* Initialize */

   /* Display the Universe. */
   static void Display() { } /* Display */

   /* Update Universe to be the next generation. */
   static void NextGeneration() { } /* NextGeneration */
   ...
   } /* CellularAutomaton */
```

☞ **Defer challenging code for later; do the easy parts first.**

Our simulation of a cellular automaton models a finite M-by-N Universe of cells. The states of each generation of next cells is determined from the states of the old cells, where generations are numbered from 0 through LAST_GEN. The state of each cell is modeled as an **int**.

Specify and declare the data representation.

```
/* A cellular automaton. */
class CellularAutomaton {
   ...
   static final int M = 5;                              // Height of Universe.
   static final int N = 20;                             // Width of Universe.
   static int old[][] = new int[M][N];                  // old Universe.
   static int next[][] = new int[M][N];                 // next Universe.
   static final int LAST_GEN = 40;                      // Last generation.
   static int generation;                               // Generation number.
   ...
   } /* CellularAutomaton */
```

☞ **Aggregate the definitions of related variables and methods in a class.**

Data Representation

Our simulation of a cellular automaton models a finite M-by-N Universe of cells. The states of each generation of next cells is determined from the states of the old cells, where generations are numbered from 0 through LAST_GEN. The state of each cell is modeled as an **int**.

N.B. The term "state" is overloaded. Each cell of the Universe has a "state".

Specify and declare the data representation.

```
/* A cellular automaton. */
class CellularAutomaton {
   ...
   static final int M = 5;                          // Height of Universe.
   static final int N = 20;                         // Width of Universe.
   static int old[][] = new int[M][N];              // old Universe.
   static int next[][] = new int[M][N];             // next Universe.
   static final int LAST_GEN = 40;                  // Last generation.
   static int generation;                           // Generation number.
   ...
   } /* CellularAutomaton */
```

☞ **Introduce program variables whose values describe "state".**

Our simulation of a cellular automaton models a finite M-by-N Universe of cells. The states of each generation of next cells is determined from the states of the old cells, where generations are numbered from 0 through LAST_GEN. The state of each cell is modeled as an int.

N.B. The term "state" is overloaded. Each cell of the Universe has a "state", and the simulation as a whole has a "state".

Specify and declare the data representation.

```
/* A cellular automaton. */
class CellularAutomaton {
   ...
   static final int M = 5;                          // Height of Universe.
   static final int N = 20;                         // Width of Universe.
   static int old[][] = new int[M][N];              // old Universe.
   static int next[][] = new int[M][N];             // next Universe.
   static final int LAST_GEN = 40;                  // Last generation.
   static int generation;                           // Generation number.
   ...
   } /* CellularAutomaton */
```

☞ **Introduce program variables whose values describe "state".**

Names of variables intended to be constant throughout program execution (`final`) are, by convention, all capital letters.

Declare and specify the data representation.

```
/* A cellular automaton. */
class CellularAutomaton {
   ...
   static final int M = 5;                              // Height of Universe.
   static final int N = 20;                             // Width of Universe.
   static int old[][] = new int[M][N];                  // old Universe.
   static int next[][] = new int[M][N];                 // next Universe.
   static final int LAST_GEN = 40;                      // Last generation.
   static int generation;                               // Generation number.
   ...
   } /* CellularAutomaton */
```

☞ **Minimize use of literal numerals in code; define and use symbolic constants.**

Declare and specify the data representation.

```
/* A cellular automaton. */
class CellularAutomaton {
    ...
    static final int M = 5;                          // Height of Universe.
    static final int N = 20;                         // Width of Universe.
    static int old[][] = new int[M][N];              // old Universe.
    static int next[][] = new int[M][N];             // next Universe.
    static final int LAST_GEN = 40;                  // Last generation.
    static int generation;                           // Generation number.
    ...
    } /* CellularAutomaton */
```

Variables declared at the top-level of a class with the keyword **`static`** are called *class variables*, and are shared among all of the methods of the class.

We now turn to the implementation of the methods

```
/* A cellular automaton. */
class CellularAutomaton {
   ...
   } /* CellularAutomaton */
```

The declarations and their initializations already did so.

Anything that needs to be at first can done in `initialize`. For now, nothing.

```
/* Create the initial Universe. */
static void Initialize() {
    ____________
   } /* Initialize */
```

Display

Display the current generation of the Universe.

```
/* Display Universe old[][] as an M-by-N grid. */
static void Display() {
  ____________
  } /* Display */
```

Display

Display the current generation of the Universe.

```
/* Display Universe old[][] as an M-by-N grid. */
static void Display() {
   System.out.println( "Generation: " + generation );
   ___________
   } /* Display */
```

Use a standard row-major-order traversal, and output newlines at row ends.

```
/* Display Universe old[][] as an M-by-N grid. */
static void Display() {
   System.out.println( "Generation: " + generation );
   for (int r=0; r<M; r++) {
      for (int c=0; c<N; c++) System.out.print( old[r][c] + " " );
      System.out.println();
      }
   } /* Display */
```

☞ **Master stylized code patterns, and use them.**

Use a standard row-major-order traversal, and output newlines at row ends.

```
/* Display Universe old[][] as an M-by-N grid. */
static void Display() {
   System.out.println( "Generation: " + generation );
   for (int r=0; r<M; r++) {
      for (int c=0; c<N; c++) System.out.print( old[r][c] + " " );
      System.out.println();
      }
   } /* Display */
```

Means: Don’t go to the beginning of the next line after printing.

☞ **Master stylized code patterns, and use them.**

Display

Use a standard row-major-order traversal, and output newlines at row ends.

```
/* Display Universe old[][] as an M-by-N grid. */
static void Display() {
   System.out.println( "Generation: " + generation );
   for (int r=0; r<M; r++) {
      for (int c=0; c<N; c++) System.out.print( old[r][c] + " " );
      System.out.println();
      }
   } /* Display */
```

Variables `r` and `c` are local variables of method `Display`.

Display

Use a standard row-major-order traversal, and output newlines at row ends.

```
/* Display Universe old[][] as an M-by-N grid. */
static void Display() {
   System.out.println( "Generation: " + generation );
   for (int r=0; r<M; r++) {
      for (int c=0; c<N; c++) System.out.print( old[r][c] + " " );
      System.out.println();
      }
   } /* Display */
```

Variables generation, M , N, and old are class variables.

The existing stub for `NextGeneration` suffices for an initial test.

```
/* Update Universe to be the next generation. */
static void NextGeneration() { } /* NextGeneration */
```

☞ **Write degenerate program stubs that allow partial programs to execute.**

Test early and often.

To try it out, invoke `CellularAutomaton.main()`.

☞ **Test programs incrementally.**

☞ **Never be (very) lost. Don’t stray far from a correct (albeit, partial) program.**

Output:

```
Generation: 0
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
Generation: 1
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
Generation: 2
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
Etc.
```

What has been validated?

- Generation counting
- Formatting of Universe

☞ **Validate output thoroughly.**

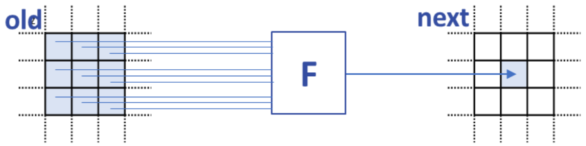
Instance of the standard compute-use pattern.

```
/* Update Universe to be the next generation. */
static void NextGeneration() {
   /* Determine the states of next[][] as F(old[][] states). */
   /* Swap old[][] and next[][] Universes. */
   } /* NextGeneration */
```

☞ **Master stylized code patterns, and use them.**

Standard row-major-order traversal for determining new states of each cell of next.

```
/* Update Universe to be the next generation. */
static void NextGeneration() {
   /* Determine the states of next[][] as F(old[][] states). */
      for (int r=0; r<M; r++)
         for (int c=0; c<N; c++)
```




```
   /* Swap old[][] and next[][] Universes. */
   } /* NextGeneration */
```

Standard row-major-order traversal for determining new states of each cell of next.

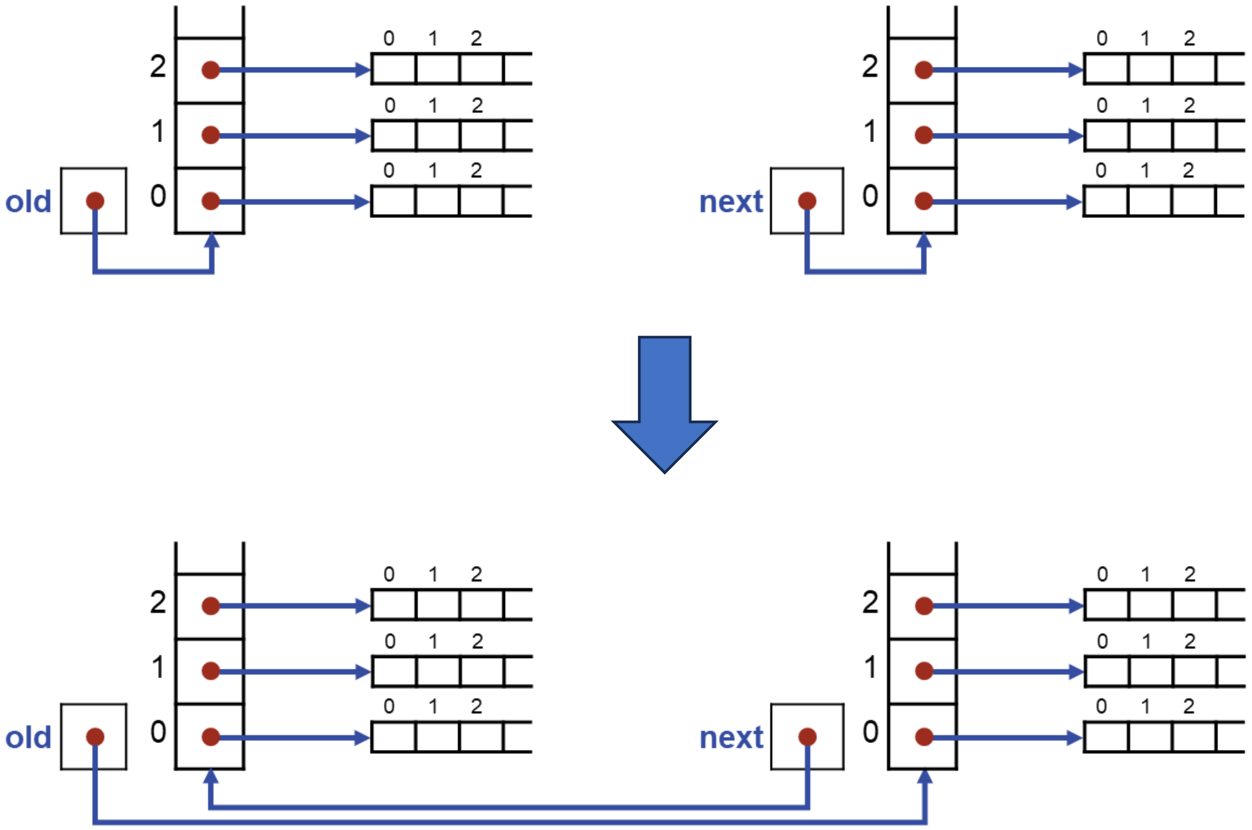
```
/* Update Universe to be the next generation. */
static void NextGeneration() {
   /* Determine the states of next[][] as F(old[][] states). */
      for (int r=0; r<M; r++)
         for (int c=0; c<N; c++)
            /* next[r][c] = F( old[r][c] and its neighbors ); */
   /* Swap old[][] and next[][] Universes. */
   } /* NextGeneration */
```

Standard code for swap

```
/* Update Universe to be the next generation. */
static void NextGeneration() {
   /* Determine the states of next[][] as F(old[][] states). */
      for (int r=0; r<M; r++)
         for (int c=0; c<N; c++)
            /* next[r][c] = F( old[r][c] and its neighbors ); */
   /* Swap old[][] and next[][] Universes. */
      int temp[][] = old;  old = next;  next = temp;
   } /* NextGeneration */
```

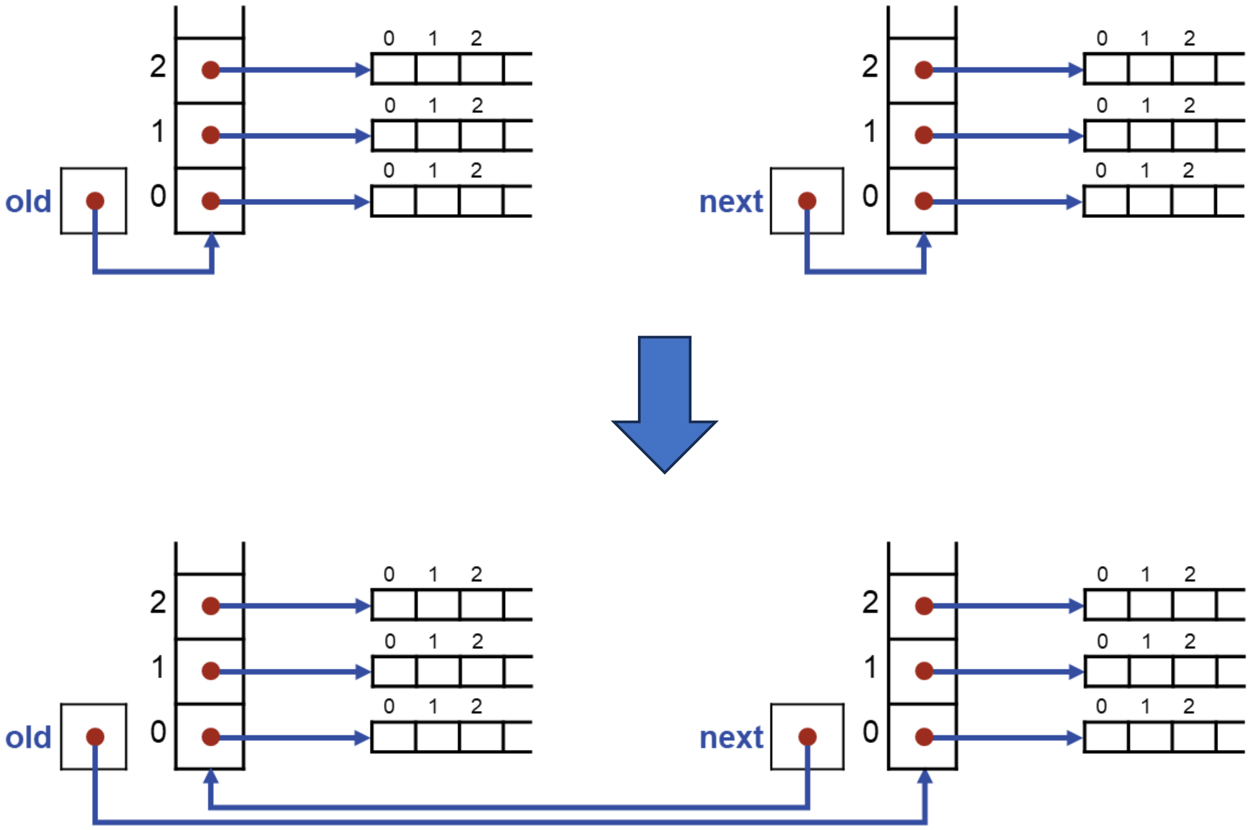
☞ **Master stylized code patterns, and use them.**

Notice that swap is a constant-time operation, independent of the size of the Universes.



NextGeneration

Notice that swap is a constant-time operation, independent of the size of the Universes.*



*C/C++ Constant-time swap is not available for C-style arrays in C/C++. Rather, this can be read as describing one of the alternatives to C-style arrays that are available in C++.

Completed `next_generation` for a generic cellular automaton.

```
/* Update Universe to be the next generation. */
static void NextGeneration() {
   /* Determine the states of next[][] as F(old[][] states). */
      for (int r=0; r<M; r++)
         for (int c=0; c<N; c++)
            /* next[r][c] = F( old[r][c] and its neighbors ); */
   /* Swap old[][] and next[][] Universes. */
      int temp[][] = old;  old = next;  next = temp;
   } /* NextGeneration */
```

For easy incremental testing, let each cell increment its state on each generation.

```
/* Update Universe to be the next generation. */
static void NextGeneration() {
   /* Determine the states of next[][] as F(old[][] states). */
      for (int r=0; r<M; r++)
         for (int c=0; c<N; c++)
            next[r][c] = old[r][c] + 1;
   /* Swap old[][] and next[][] Universes. */
      int temp[][] = old;  old = next;  next = temp;
   } /* NextGeneration */
```

☞ **Test programs incrementally.**

Test early and often.

To try it out again, invoke `CellularAutomaton.main()`.

☞ **Test programs incrementally.**

☞ **Never be (very) lost. Don't stray far from a correct (albeit, partial) program.**

Output:

```
Generation: 0
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
00000000000000000000
Generation: 1
11111111111111111111
11111111111111111111
11111111111111111111
11111111111111111111
11111111111111111111
Generation: 2
22222222222222222222
22222222222222222222
22222222222222222222
22222222222222222222
22222222222222222222
Etc.
```

What has been validated?

- Generation counting
- Formatting of Universe
- Creation of `next` Universe from `old` Universe
- Swapping of `old` and `next`

☞ **Validate output thoroughly.**

Game of Life. A Cellular Automaton in which each cell is either dead or alive.

In each generation:

- Each live cell with 2 or 3 live neighbors lives on to the next generation (life) otherwise it dies (death).
- Each dead cell with 3 live neighbors comes alive in the next generation (birth) otherwise it remains dead.

Each cell is either dead or alive, so specialize the Universes as Boolean 2-D arrays.

```
/* A cellular automaton. */
class CellularAutomaton {
   ...
   static final int M = 6;                               // Height of Universe.
   static final int N = 20;                              // Width of Universe.
   static boolean old[][] = new boolean [M][N];  // cell true iff alive.
   static boolean next[][] = new boolean [M][N]; // cell true iff alive.
   static final int LAST_GEN = 50;                       // Last generation.
   static int generation;                                // Generation number.
   ...
   } /* CellularAutomaton */
```

☞ **Choose representations that by design don’t have nonsensical configurations.**

Specialize the output by compactly rendering dead as "_" and alive as "X".

```
/* Display Universe old[][] as an M-by-N grid. */
static void Display() {
   System.out.println( "Generation: " + generation );
   for (int r=0; r<M; r++) {
      for (int c=0; c<N; c++)
         if ( old[r][c] ) System.out.print( "X" );
         else System.out.print( "_" );
      System.out.println();
      }
   } /* Display */
```

Implement Game of Life rules.

```
/* Update old[][] to be the next generation of the Universe. */
static void NextGeneration() {
   /* Determine the states of next[][] as Life(old[][] states). */
      for (int r=0; r<M; r++)
         for (int c=0; c<N; c++)
            /* Set next[r][c] = according to Game of Life rules. */
   /* Swap old[][] and next[][] Universes. */
      int temp[][] = old;  old = next;  next = temp;
   } /* NextGeneration */
```

Instance of the standard compute-use pattern.

```
/* Update old[][] to be the next generation of the Universe. */
static void NextGeneration() {
   /* Determine the states of next[][] as Life(old[][] states). */
      for (int r=0; r<M; r++)
         for (int c=0; c<N; c++) {
            /* Compute. */
            /* Use. */
            }
   /* Swap old[][] and next[][] Universes. */
      int temp[][] = old;  old = next;  next = temp;
   } /* NextGeneration */
```

☞ **Master stylized code patterns, and use them.**

Instantiate placeholders *Compute* and *Use* for the problem at hand.

```
/* Update old[][] to be the next generation of the Universe. */
static void NextGeneration() {
   /* Determine the states of next[][] as Life(old[][] states). */
      for (int r=0; r<M; r++)
         for (int c=0; c<N; c++) {
            /* Let liveNeighbors be number alive around old[r][c]. */
            /* Set next[r][c] according to the birth and death rules. */
            }
   /* Swap old[][] and next[][] Universes. */
      int temp[][] = old;  old = next; next = temp;
   } /* NextGeneration */
```

☞ **Master stylized code patterns, and use them.**

```
/* Update old[][] to be the next generation of the Universe. */
static void NextGeneration() {
   /* Determine the states of next[][] as Life(old[][] states). */
      for (int r=0; r<M; r++)
         for (int c=0; c<N; c++) {
            /* Let liveNeighbors be number alive around old[r][c]. */
            /* Set next[r][c] according to the birth and death rules. */
               if ( old[r][c] ) /* Currently live. */
                  if ( liveNeighbors==2 || liveNeighbors==3 )
                     next[r][c] = true;
                  else next[r][c] = false;
               else /* Currently dead. */
                  if ( liveNeighbors==3 ) next[r][c] = true;
                  else next[r][c] = false;
            }
   /* Swap old[][] and next[][] Universes. */
      int temp[][] = old;  old = next;  next = temp;
   } /* NextGeneration */
```

Use is a structured four-way case analysis.

```
/* Update old[][] to be the next generation of the Universe. */
static void NextGeneration() {
   /* Determine the states of next[][] as Life(old[][] states). */
      for (int r=0; r<M; r++)
         for (int c=0; c<N; c++) {
            /* Let liveNeighbors be number alive around old[r][c]. */
               int liveNeighbors = 0;
               for (int dr=-1; dr<=+1; dr++)
                  for (int dc=-1; dc<=+1; dc++)
                     if ( !((dr==0)&&(dc==0)) && old[r+dr][c+dc] )
                        liveNeighbors++;
            /* Set next[r][c] according to the birth and death rules. */
               ...
            }
   /* Swap old[][] and next[][] Universes. */
      int temp[][] = old;  old = next;  next = temp;
   } /* NextGeneration */
```

Compute is a 3x3 row-major-order traversal, incrementing `liveNeighbors` as appropriate.

Game of Life

```
/* Update old[][] to be the next generation of the Universe. */
static void NextGeneration() {
   /* Determine the states of next[][] as Life(old[][] states). */
      for (int r=0; r<M; r++)
         for (int c=0; c<N; c++) {
            /* Let liveNeighbors be number alive around old[r][c]. */
               int liveNeighbors = 0;
               for (int dr=-1; dr<=+1; dr++)
                  for (int dc=-1; dc<=+1; dc++)
                     if ( !((dr==0)&&(dc==0)) && old[r+dr][c+dc] )
                        liveNeighbors++;
            /* Set next[r][c] according to the birth and death rules. */
               ...
            }
   /* Swap old[][] and next[][] Universes. */
      int temp[][] = old;  old = next;  next = temp;
   } /* NextGeneration */
```

To prevent the subscripts going out of bounds

Game of Life

```
/* Update old[][] to be the next generation of the Universe. */
static void NextGeneration() {
   /* Determine the states of next[][] as Life(old[][] states). */
      for (int r=0; r<M; r++)
         for (int c=0; c<N; c++) {
            /* Let liveNeighbors be number alive around old[r][c]. */
               int liveNeighbors = 0;
               for (int dr=-1; dr<=+1; dr++)
                  for (int dc=-1; dc<=+1; dc++)
                     if ( !((dr==0)&&(dc==0)) && old[(r+dr)%M][(c+dc)%N] )
                        liveNeighbors++;
            /* Set next[r][c] according to the birth and death rules. */
               ...
            }
   /* Swap old[][] and next[][] Universes. */
      int temp[][] = old;  old = next;  next = temp;
   } /* NextGeneration */
```

To prevent the subscripts going out of bounds, simulate on a torus.

Create some life, which will glide diagonally down and to the right.

```
/* Establish original configuration in old. */
static void Initialize() {
   /* Glider */
      old[0][1] = old[1][2] = old[2][3] = old[2][1] = old[2][2] = true;
   } /* Initialize */
```

old	0	1	2	…
0		T		…
1			T	…
2	T	T	T	…
	…	…	…	…

To let it rip, invoke `CellularAutomaton.main()` yet again.

And presto …

```
Generation: 0
_X__________________
__X_________________
XXX_________________
____________________
____________________
```

And presto …

```
Generation: 1
____________________
X_X_________________
_XX_________________
_X__________________
____________________
```

Game of Life

And presto …

```
Generation: 2

____________________
__X_________________
X_X_________________
_XX_________________
____________________
```

Game of Life

And presto …

```
Generation: 3

____________________
_X__________________
__XX________________
_XX_________________
____________________
```

And presto …

```
Generation: 4
____________________
__X_________________
___X________________
_XXX________________
____________________
```

Back to the same configuration as Generation 0, but shifted down and one cell to the right.

And presto …

```
Generation:  5
____________________
____________________
_X_X________________
__XX________________
__X_________________
```

And presto …

```
Generation:  6
____________________
____________________
___X________________
_X_X________________
__XX________________
```

And presto …

```
Generation:  7
____________________
____________________
__X_________________
___XX_______________
__XX________________
```

And presto …

```
Generation:  8
____________________
____________________
___X________________
____X_______________
__XXX_______________
```

Back to the same configuration as Generation 1, but shifted down and right one cell.

And presto …

```
Generation:  9
___X________________
____________________
____________________
__X_X_______________
___XX_______________
```

Whoa! What’s going on? Oh, I forgot, we are on a torus.

And presto …

```
Generation:  10
___XX_______________
____________________
____________________
____X_______________
__X_X_______________
Generation:  11
```

And presto …

```
Generation:  11
___XX_______________
____________________
____________________
___X________________
____XX______________
```

And presto …

```
Generation:  12
___XXX______________
____________________
____________________
____X_______________
_____X______________
```

Testing

And presto …

```
Generation:  13
____XX______________
____X_______________
____________________
____________________
___X_X______________
```

And presto …

```
Generation:  14
___X_X______________
____XX______________
____________________
____________________
_____X______________
```

And presto …

```
Generation:  15
_____XX_____________
____XX______________
____________________
____________________
____X_______________
```

And presto …

```
Generation:  16
______X_____________
____XXX_____________
____________________
____________________
_____X______________
```

And presto …

```
Generation:  17
____X_X_____________
_____XX_____________
_____X______________
____________________
____________________
```

And presto …

```
Generation:  18
______X_____________
____X_X_____________
_____XX_____________
____________________
____________________
```

And presto …

```
Generation:  19
_____X______________
______XX____________
_____XX_____________
____________________
____________________
```

And presto …

```
Generation:   20
______X____________
_______X___________
_____XXX___________
___________________
___________________
```

Back to the same configuration as Generation 0, but shifted right several cells. The glider is coiling around the donut!

Boundary Conditions

What are the boundary conditions for this problem, and did we forget them?

For example, what if the height of the Universe were only 4? To try it out, change N, and invoke `CellularAutomaton.main()`.

```
Generation: 0
_X__________________
__X_________________
XXX_________________
____________________
```

☞ **Boundary conditions. Dead last, but don't forget them.**

Boundary Conditions

What are the boundary conditions for this problem, and did we forget them?

For example, what if the height of the Universe were only 4? To try it out, change N, and invoke `CellularAutomaton.main()`.

```
Generation: 0
_X__________________
__X_________________
XXX_________________
____________________
Generation:  1
____________________
X_X_________________
_XX_________________
X_X_________________
```

There isn't enough "elbowroom" around the glider, and it is interfering with its own propagation. By generation 6, all life is gone!

Should your program be defensive and prevent this, or is this just how life goes?

☞ **Boundary conditions. Dead last, but don't forget them.**

Summary:

The notion of a **class** has been introduced as a means for aggregating variables and methods.

Many standard precepts, patterns, and recommended coding techniques have been illustrated.

And the Game of Life itself is fascinating.