Principled Programming

Introduction to Coding in Any Imperative Language

Tim Teitelbaum

Emeritus Professor

Department of Computer Science

Cornell University

Cellular Automata

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

The simulation as a whole is implemented as method main.

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.

- 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 */
```

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 LAST_GEN additional 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 LAST_GEN additional 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 LAST_GEN additional generations.*/
         for (generation=1; generation<=LAST_GEN; generation++) {</pre>
            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++) {</pre>
            NextGeneration();
            Display();
      } /* main */
    /* CellularAutomaton */
```

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.

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

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.

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

```
/* A cellular automaton. */
class CellularAutomaton {
    ...
    static final int M = 5;
    static final int N = 20;
    static int old[][] = new int[M][N];
    static int next[][] = new int[M][N];
    static final int LAST_GEN = 40;
    static int generation;
    // 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.

```
Specify and declare the data representation.
```

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

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

```
// Height of Universe.
// Width of Universe.
// old Universe.
// next Universe.
// Last generation.
// Generation number.
```

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;
    static final int N = 20;
    static int old[][] = new int[M][N];
    static int next[][] = new int[M][N];
    static final int LAST_GEN = 40;
    static int generation;
    // 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;
    static final int N = 20;
    static int old[][] = new int[M][N];
    static int next[][] = new int[M][N];
    static final int LAST_GEN = 40;
    static int generation;
    ...
} /* 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.

Display the current generation of the Universe.

Display the current generation of the Universe.

```
/* 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 */</pre>
```

```
/* 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 */</pre>
```

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

```
/* 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 */</pre>
```

Variables r and c are local variables of method Display.

```
/* 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 */</pre>
```

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 */
```

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: 1

Generation: 2

Etc.

What has been validated?

- Generation counting
- Array creation and initialization
- Formatting of Universe

Instance of the standard compute-use pattern.

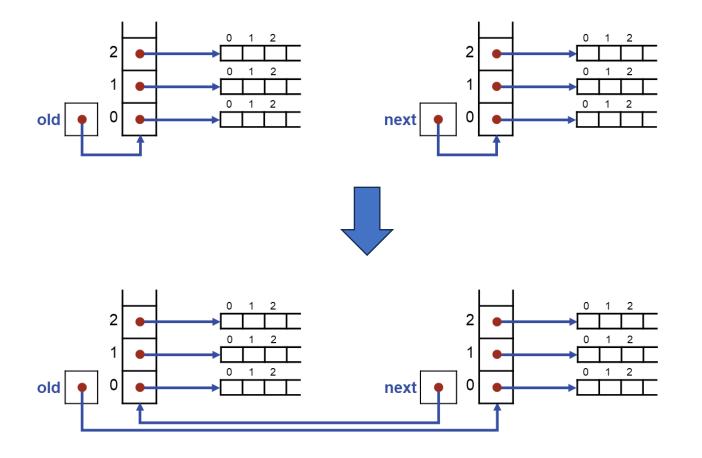
```
/* Update Universe to be the next generation. */
static void NextGeneration() {
    /* Determine the states of next[][] as F(old[][] states). */
    /* Swap old[][] and next[][] Universes. */
    } /* NextGeneration */
```

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

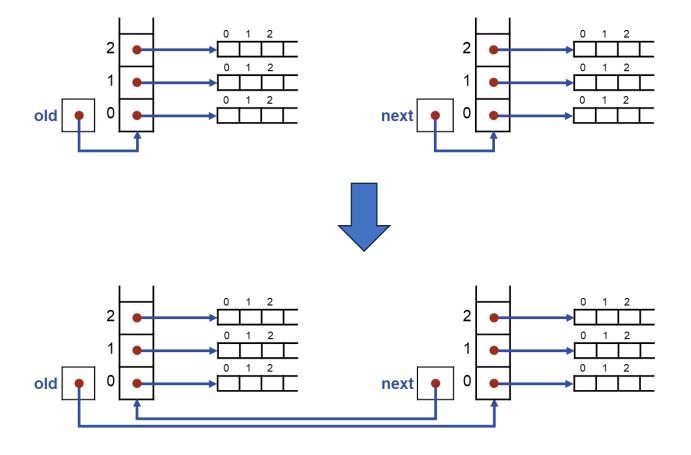
Standard row-major-order traversal for determining new states of each cell of next.

Standard code for swap

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



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.

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 */</pre>
```

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: 1

111111111111111111111

Generation: 2

22222222222222222

2222222222222222222

222222222222222222

Etc.

What has been validated?

- Generation counting
- Array creation and initialization
- Formatting of Universe
- Creation of next Universe from old Universe
- Swapping of old and next

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;
    static final int N = 20;
    static boolean old[][] = new boolean [M][N]; // old Universe.
    static boolean next[][] = new boolean [M][N]; // next Universe.
    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 */</pre>
```

Implement Game of Life rules.

Instance of the standard compute-use pattern.

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 # of alive cells around old[r][c]. */
            /* Set next[r][c] based on old[r][c] and liveNeighbors. */
               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 # of alive cells 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] based on old[r][c] and liveNeighbors. */
  /* 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.

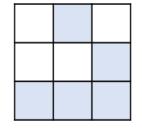
ubscripts

```
/* 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 # of alive cells 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] based on old[r][c] and liveNeighbors. */
  /* Swap old[][] and next[][] Universes. */
      int temp[][] = old; old = next; next = temp;
   } /* NextGeneration */
```

subscripts

ounds

```
/* 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 # of alive cells 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[Math.floorMod(r+dr,M)]
                             [Math.floorMod(c+dc,N)] )
                        liveNeighbors++;
            /* Set next[r][c] based on old[r][c] and liveNeighbors. */
   /* Swap old[][] and next[][] Universes. */
      int temp[][] = old; old = next; next = temp;
   } /* NextGeneration */
```



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

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

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

And presto ...

Generat	tion	: 0	
_X			
X			
XXX			

And	presto	

X_X		
XX		
_X		

Λnd	presto	
Allu	hiesto	•••

And	presto	•••
-----	--------	-----

_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	•••
-----	--------	-----

And presto		
Generation:	6	
X		

__XX__

And presto		
Generation:	7	
XXX		
^^		

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

___XX__

And presto	And	presto	•••
------------	-----	--------	-----

Generation:	10	
XX		
X		
X_X		
Generation:	11	_

And presto		
Generation:XX	11	
X		
XX		

12	
	12

And	presto	•••

Generation:	13	
XX		
X		
X X		

And	presto	
,	p. 0000	

Generation:	14	
X_X		
XX		
X		

And presto		
Generation: XX	15	
XX		

____X__

And presto		
Generation: X	16	
XXX		

And	presto	

Generation:	17	
X_X		
XX		
X		

And	presto	

Generation:	18	
X		
X_X		
XX		

And	presto	
	p. 0000	

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!

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().

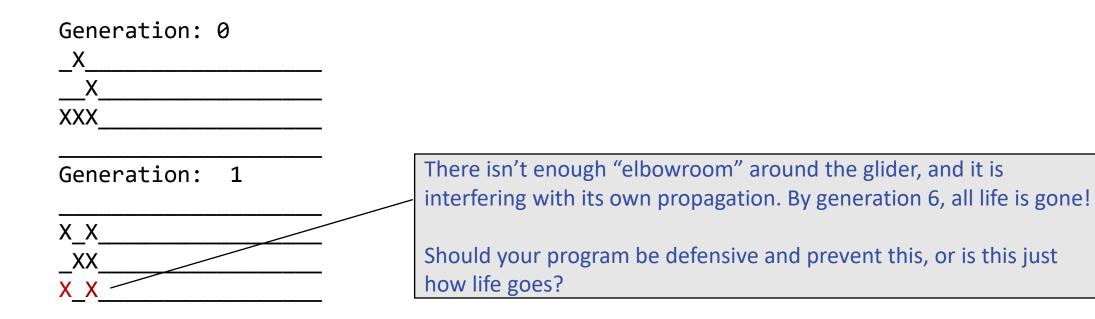
			•			_
Ger	ndn	っす	- 7	Λn	•	И
UCI	ıcı	aι		OH		U

_X __X XXX

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

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().



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.