Introduction to Computing
III - Arrays, Files and Random Numbers in C

Jonathan Mascie-Taylor
(Slides originally by Quentin CAUDRON)

Centre for Complexity Science,
University of Warwick

Outline

Arrays

Arrays are a series of elements of the same type that are accessed from the same variable via an index :

```
int myArray[10];
```

Arrays

Arrays are a series of elements of the same type that are accessed from the same variable via an index :

```
int myArray[10];
```

This is an *array of integers*, ten integers long. In general, arrays are declared like this :

```
type name[elements];
```

Arrays can have most types. You can even declare a custom data structure and make arrays from those.

Initialising Arrays

In order to populate an array with data, you need to iterate through it.

```
1 int evens[10];
2
3 int i;
4 for(i = 0; i < 10; i++)
5 {
6   evens[i] = i * 2;
7 }
```

Accessing Array Elements

You can access the elements of an array as if it were a normal variable, but with the index after it :

```
printf("%d", evens[3]);
```

Result : `6`

Accessing Array Elements

You can access the elements of an array as if it were a normal variable, but with the index after it :

```
printf("%d", evens[3]);
```

Result : `6`

The elements of an array can only be accessed one by one, unlike Matlab's range (`:`) notation. Therefore, we need to iterate :

```
1 int i;
2 for(i = 0; i < 10; i++)
3 {
4   printf("%d, ", evens[i]);
5 }
```

Result : `0, 2, 4, 6, 8, 10, 12, 14, 16, 18,`

Exercise : Reverse your Input

Write some code that will take in n integer values from user input, and repeat them back to the user in reverse order.

Exercise : Reverse your Input

Write some code that will take in n integer values from user input, and repeat them back to the user in reverse order.

- Ask the user for the value of n
- Use a `for` loop to populate your array
- Use another `for` loop to print the values in the array

Multidimensional Arrays

Using multidimensional arrays is very intuitive, as they are simply "arrays of arrays".

```
1  int twoD[5][3];
2
3  int i, j;
4  for(i = 0; i < 5; i++)
5  {
6    for(j = 0; j < 3; j++)
7    {
8      twoD[i][j] = i*j;
9    }
10 }
```

	0	1	2
0	0	0	0
1	0	1	2
2	0	2	4
3	0	3	6
4	0	4	8

Writing to Files

```
1   // Declare a file handle pointer
2   FILE * myfile;
3
4   // Open a file in Write Text mode
5   myfile = fopen("results.txt", "wt");
6
7   for(i = 0; i < 5; i++)
8   {
9       for(j = 0; j < 3; j++)
10      {
11          fprintf(myfile, "%d, ", twoD[i][
                j]);
12      }
13
14      fprintf(myfile, "\n");
15  }
16
```

Reading from Files

```
1  // Declare a file handle pointer
2  FILE * myfile;
3
4  // Open a file in Read Text mode
5  myfile = fopen( "results.txt", "rt" );
6
7  int x;
8
9  while( fscanf( myfile, "%d, ", &x ) !=
       EOF)
10 {
11     printf("%d\n", x);
12 }
13
14 fclose(myfile);
```

Reading from Files

```
1  // Declare a file handle pointer
2  FILE * myfile;
3
4  // Open a file in Read Text mode
5  myfile = fopen( "results.txt", "rt" );
6
7  int x;
8
9  while( fscanf( myfile, "%d, ", &x ) !=
       EOF)
10 {
11     printf("%d\n", x);
12 }
13
14 fclose(myfile);
```

`EOF` is *end of file*. This is a special character sequence that defines the

Exercise : Indirect Addition

Modify your previous code such that it :

- Takes in input, then outputs everything in the array to a file
- Closes the file
- Reopens the file with a new (read) stream
- Reads the integers, summing them into a new variable
- Prints the result of the addition to screen

Random Numbers - Basics

Random number generators, as we saw in Matlab, do not generate truly random numbers. They are generated by algorithms which produce long sequences of apparently random numbers that are, in fact, determined by a shorter initial value, called a *seed*. The output is deterministic.

Random Numbers - Basics

Random number generators, as we saw in Matlab, do not generate truly random numbers. They are generated by algorithms which produce long sequences of apparently random numbers that are, in fact, determined by a shorter initial value, called a *seed*. The output is deterministic.

In C, we generally use the number of seconds since 1 January, 1970. This value is easily obtainable using `time(0)`, as this date is defined as the current "calendar time".

Random Numbers - Basics

Random number generators, as we saw in Matlab, do not generate truly random numbers. They are generated by algorithms which produce long sequences of apparently random numbers that are, in fact, determined by a shorter initial value, called a *seed*. The output is deterministic.

In C, we generally use the number of seconds since 1 January, 1970. This value is easily obtainable using `time(0)`, as this date is defined as the current "calendar time".

We have to `#include <time.h>` in order to use the `time` function, and we also need to `#include <stdlib.h>` to get access to the random number generation function, `rand()`. This function takes no arguments.

Generating Random Numbers

In order to seed the random number generator, we call the seeding function with the calendar time as argument : `srand(time(0))`. Then, we can call `rand()` to generate a random *integer* between 0 and some large integer, called `RAND_MAX`.

Generating Random Numbers

In order to seed the random number generator, we call the seeding function with the calendar time as argument : `srand(time(0))`. Then, we can call `rand()` to generate a random *integer* between 0 and some large integer, called `RAND_MAX`.

Therefore, if we want to generate random floating-point numbers distributed uniformly $\in [0, 1]$, we have to divide our random number by `RAND_MAX`. However, dividing an integer by an integer results in an integer, and so we get 0. We need to tell the program that our random number is a double.

Generating Random Numbers

```
1  // Seed the random number generator
2  srand(time(0));
3
4  printf("Here are some random doubles :\n");
5
6  // Generate ten random numbers
7  int i;
8  for(i = 0; i < 10; i++)
9  {
10   // Cast rand() as a double
11   printf("%lf \n", (double) rand() / RAND_MAX);
12 }
```

Cellular Automata : a Definition

Cellular automata consist of a regular grid of cells, each have a finite number of possible states.

Cellular Automata : a Definition

Cellular automata consist of a regular grid of cells, each have a finite number of possible states.

An initial state configuration is selected at $t = 0$ by assigning a state to each cell, and then the system propagates forward in discrete time, with a new generation being created at every timestep by a set of system-wide rules.

Cellular Automata : a Definition

Cellular automata consist of a regular grid of cells, each have a finite number of possible states.

An initial state configuration is selected at $t = 0$ by assigning a state to each cell, and then the system propagates forward in discrete time, with a new generation being created at every timestep by a set of system-wide rules.

The rules for the system that governs the state of each cell at a specific timestep is based on the state of the cell and its neighbouring cells in the previous timestep.

Cellular Automata : a Definition

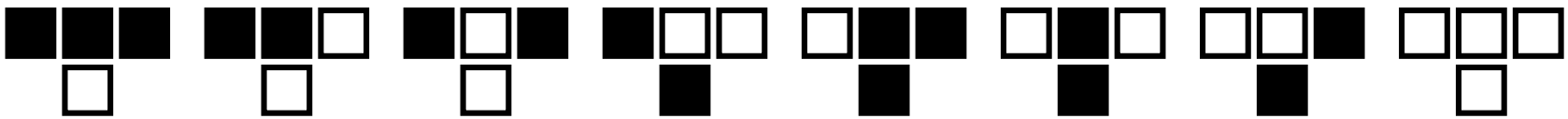
Cellular automata consist of a regular grid of cells, each have a finite number of possible states.

An initial state configuration is selected at $t = 0$ by assigning a state to each cell, and then the system propagates forward in discrete time, with a new generation being created at every timestep by a set of system-wide rules.

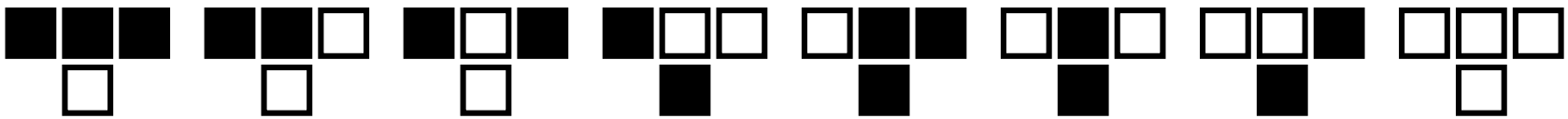
The rules for the system that governs the state of each cell at a specific timestep is based on the state of the cell and its neighbouring cells in the previous timestep.

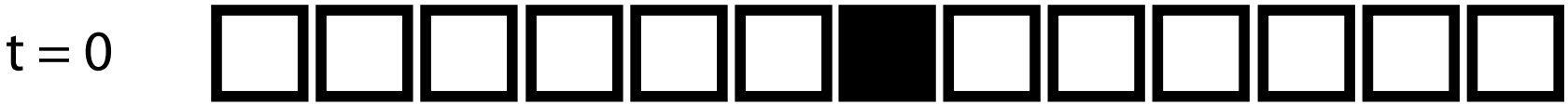
The simplest automata are one-dimensional (on a line) and have two states : 0 or 1.

The Rule 30 Cellular Automaton

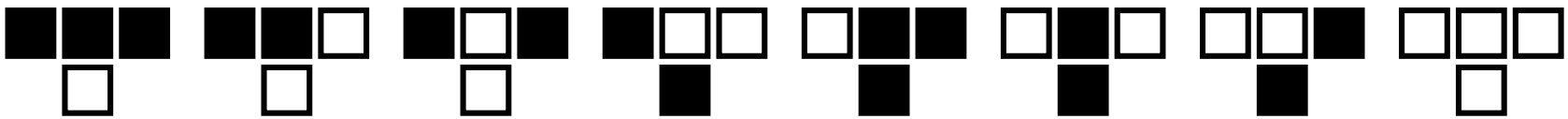


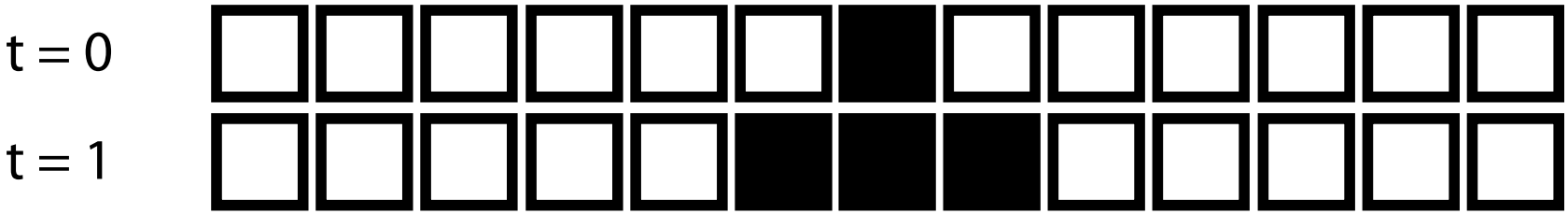
The Rule 30 Cellular Automaton



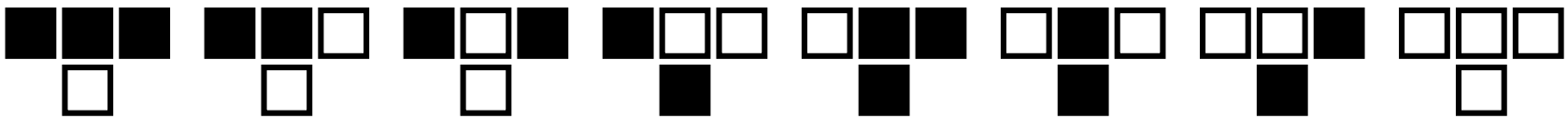


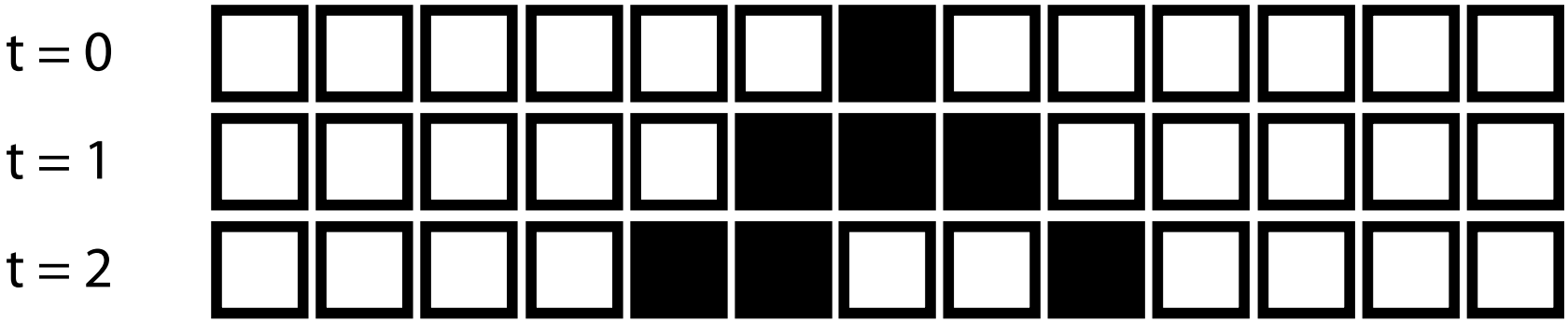
The Rule 30 Cellular Automaton



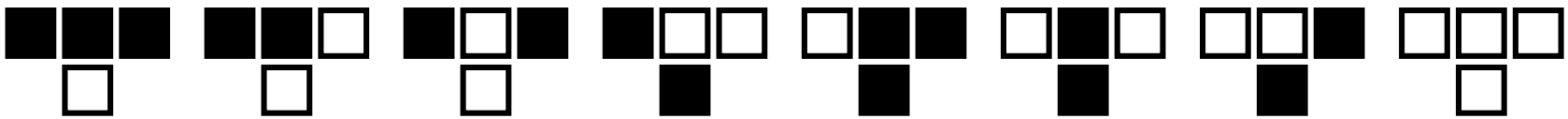


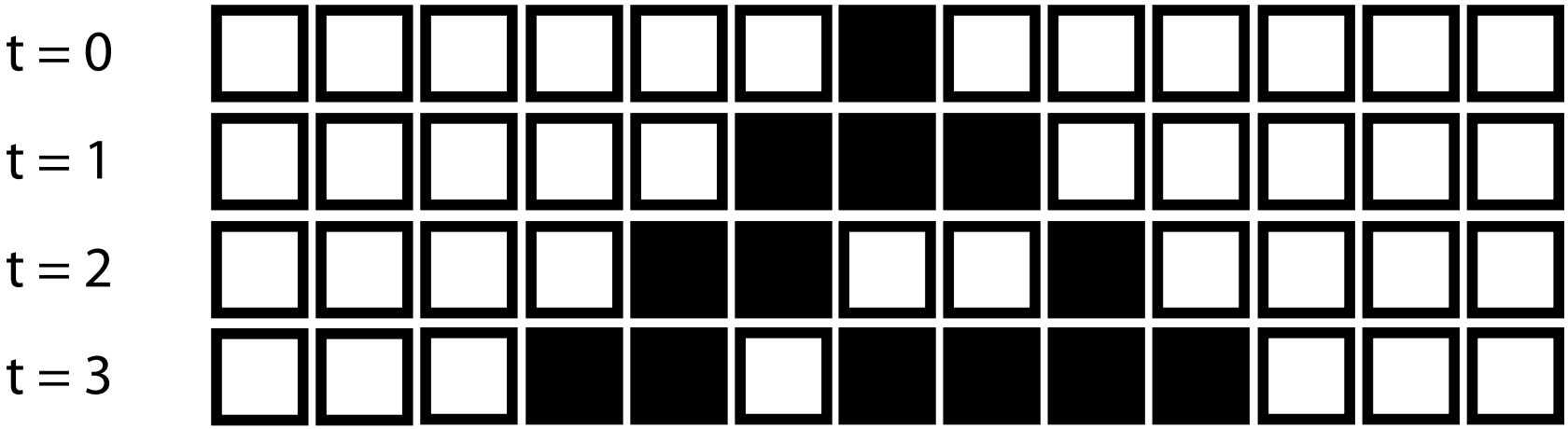
The Rule 30 Cellular Automaton



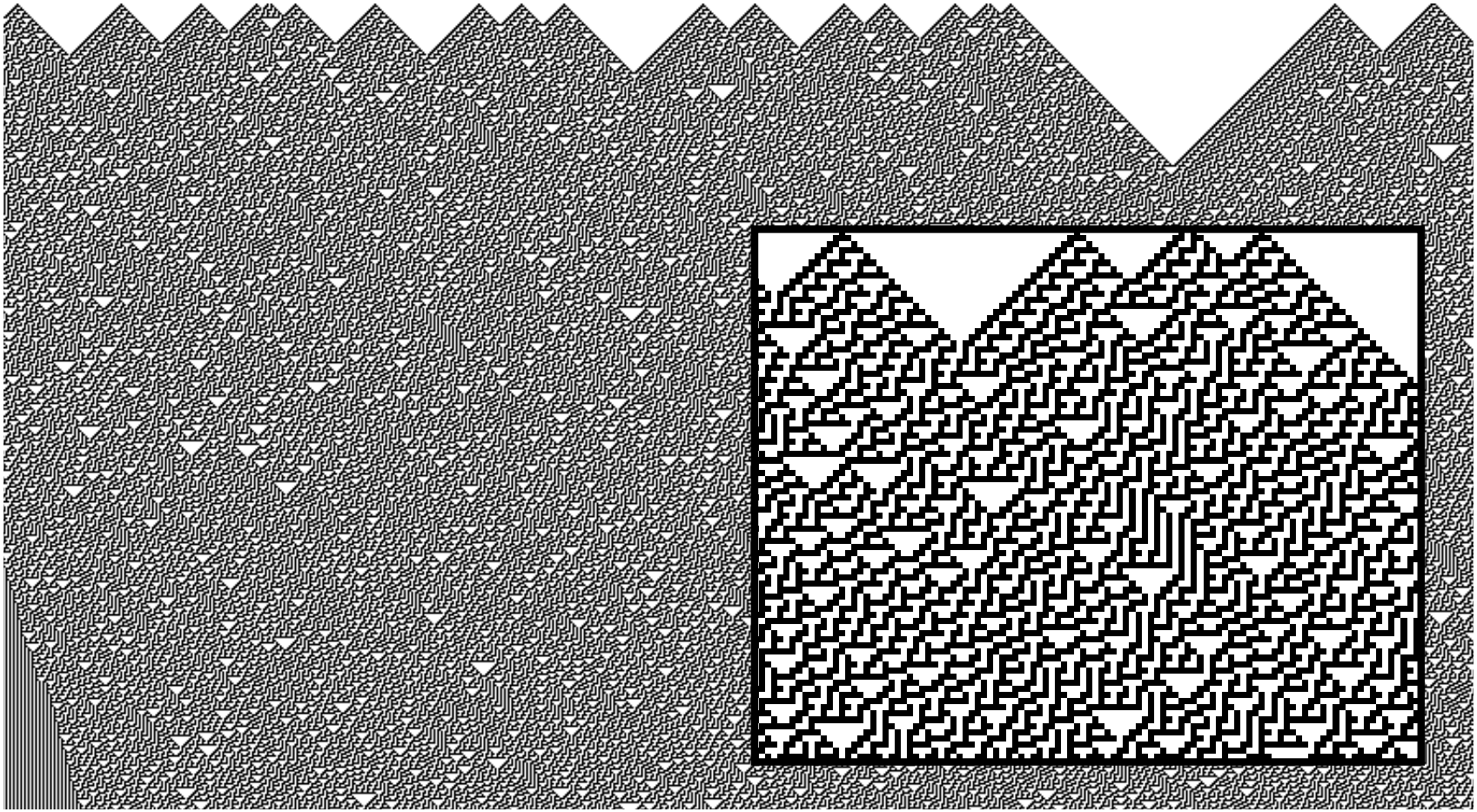


The Rule 30 Cellular Automaton





The Rule 30 Cellular Automaton



Updating

The update value of a cell at $t = n$ is dependent on its value at $t = n - 1$ as well as the value of its neighbours at $t = n - 1$. However, we can calculate a "neighbourhood update value". For example, if we have lattice sites $L_{\text{left}} = 1, L_{\text{middle}} = 0, L_{\text{right}} = 1$, then we can consider this a binary number 101, and hence a decimal number 5.

In order to update a lattice site, we can just calculate its neighbourhood update value at its previous timestep, and use a single `if` statement, instead of comparing the lattice site and its neighbours independently (if $L_l = 1$, then if $L_m = 0$, then if $L_r = 1$, then L_m updates to 0...)

Exercise : Rule 30

A challenging exercise : code up the Rule 30 automaton.

- Declare two 1D integer arrays of size 1002, one to represent the lattice, and the other as a temporary buffer
- Populate the first lattice with mostly zeros and a few ones
- Open a write-file stream for your results
- Iterate 500 times, using a for loop
- Within this loop, iterate over lattice positions *from 1 to 1000*, calculating update values and updating the buffer array accordingly
- Then, iterate over the buffer, copying it into the main array
- As you do so, output the value to file, separated by a space, and enter a new line after completing the loop
- Also try experimenting with periodic boundary conditions (where $L_1 \equiv L_{1001}$)
- To view the results import them in to MATLAB and use `spy`