Exercise 3 — Pointers

Andreas Nüchter, Fabian Arzberger, Jasper Zevering

Winter 2025/2026

Goals: In this assignment you will write a program to simulate a simple one-dimensional cellular automaton, which are related to (but are not the same as) two-dimensional cellular automata like Conway's game of Life. We will use this as an example to work with pointers and pointer arithmetic, as well as dynamic memory allocation. Pointers are the most confusing topic in C. . ..

Prerequisites:

- pointers
- dynamic memory allocation
- the sizeof operator
- type casts

Program to write: One-dimensional cellular automata

A cellular automaton (CA) consists of a collection of cells which can hold data values, generally organized into some regular geometric arrangement (usually a square grid or a line). A one-dimensional CA (1dCA) has all the cells arranged in a line. The values in the cells change over time. Each cell's value at the next time interval depends on its current value as well as the value of its two immediate neighbors. We set the first and last cells to be always empty. In this case you will be implementing a 1dCA with two states (full and empty, which you should implement as 1 and 0).

The 1dCA you will implement has the following update rule:

1. If the current cell is empty, and one of the two adjacent cells is full (but not both), set the current cell to be full.
2. Otherwise set the current cell to be empty.

You have to iterate through all the cells to generate the new pattern (the next "generation"). You need to have two arrays to store the data, because you are not allowed to modify one generation until the next generation has been completely computed. The 1dCA should be randomly seeded with 1's and 0's to start with (try having about 50% of the cells 1s initially). If successive generations are printed on a screen, one generation per line, with (say) a "." for 0s and a "*" for 1s, you will get a chaotic-looking pattern that has a fractal-like quality to it and is quite pretty.

Be sure you get the correct update rule in your program; every year, some students get some part of it wrong. If the pattern generated by your program doesn't look cool, you've almost certainly done something wrong.

Command-line arguments

Your program should take two command-line arguments: the numbers of cells in the 1dCA and the number of generations to compute. The program should print out this many generations to the screen.

Use dynamic memory allocation to create the arrays you need to hold the 1dCA data. Make sure you check the return value of every malloc or calloc call and exit with an error message if the call returns NULL. Also make sure you free your newly-allocated data before your program exits.

NOTE: if your program doesn't get the correct number of arguments, it should exit with a usage message as usual.

Note that you can do this entire program without using pointer arithmetic. To give you practice in using pointer arithmetic, write the function which updates the board in two ways:

1. only using array operations;
2. only using pointer operations.

Define two separate programs for these two cases (with different names). The programs will share all of their code except for the board updating function. For the second program, you don't have to use explicit pointer arithmetic in any functions other than the update function; use array operations everywhere else. Also note that for both programs you have to use dynamic memory allocation when allocating the arrays (because you don't know the size of the arrays in advance).

Don't just translate the array operations into equivalent pointer operations; instead, use the iteration idiom shown above, suitably modified to work with your program. Specifically, you should use three pointers to iterate through the array: for any given value of an index variable `i`, one points to `cell[i-1]`, one to `cell[i]` and one to `cell[i+1]` (assuming that the array is called cell). Each of the three pointers will get incremented once for each new cell value that is being computed. In addition, you'll need another pointer into another array which is the array you're writing into. In this way you won't have to use array operations at all, and the only pointer arithmetic you'll need is to increment by one (`++`). This requirement applies throughout the entire function; even if you have to copy an array at the end, use pointer operations to do this. Once again, don't just translate the array code into the equivalent pointer code.

As always, try to decompose your program into small, easy-to-understand functions instead of having a few huge functions. Make each function do one thing only (don't mix printing with updating, for instance).

Don't forget to include the memory leak checking code in your file. Include the "memcheck.h" header in your program, compile the corresponding source code and link it to your program. The assignment will not be graded if your program has any memory leaks.

Other stuff to do

Write your own Makefile for this lab, based on the examples you've seen before. Each of the two programs you have to write should have a different name. You should have separate targets to compile each program, as well as an "`all`" target that compiles both programs (i.e. you type "`make all`" and both programs are built). It's convenient to make the "`all`" target the first target in the program, so just typing "`make` will build both programs. Note that you can write the "`all`" target in a single line (hint: what are its dependencies?). You should also have a "`clean`" target that removes all compiled files (object files and executables), and a "`check`" target that calls the style checker on both programs (note that the style checker can take multiple filenames as arguments). Finally, make sure that you compile the memcheck.c memory leak checker and link it into both programs.

Name your programs anything you want, except don't call them "life" or some variant of the word "life". "Life" refers to a specific kind of two-dimensional cellular automaton, not the one-dimensional cellular automaton we're writing here.

Submission: The source code for the two versions of your program, along with your `Makefile`.

Please send the programs to `fabian.arzberger@uni-wuerzburg.de`
and `jasper.zevering@uni-wuerzburg.de` until November 05, 2025 23.59. Include `[CS4SE]` in your subject line and state the names of all team members in the message body. Please work in groups of two students. One submission per team is sufficient. Late homework won't be accepted!