

CS 211, Winter 2004
Lab Assignment L2: Arrays, Pointers, and Strings
Assigned: Monday, Jan. 19, Due: Monday, Feb. 2, 11:00 AM

Brian M.Dennis, Instructor
Bin Lin, Tom Lechner, Rachel Goldsborough, Teaching Assistants

Aims and Goals

The purpose of this assignment is to make you proficient with C++ pointers, arrays, and strings. After you complete this lab, you should be able to do the following:

- Declare and define pointer variables
- Dereference and store into pointers
- Declare and define array variables
- Dereference and store into arrays
- Iterate through the elements of an array using integer indices or pointer arithmetic
- Avoid dangling pointer errors
- Create and manipulate null-terminated strings
- Generate random numbers for a program's input
- Use command line arguments in a program

So far arrays are our only composite data structure. Remember, they're homogeneous collections of typed objects, sequentially indexed, but no bounds checking is done for us, and there's no way to determine the number of objects in an array.

To motivate the usage of arrays, we'll warm up with some basic pointer and array manipulations, then do some slightly more involved tasks, and wrap up with an exploration of cellular automata.

Basic Pointer Manipulation (10 Points)

Write two versions each of the `strcat`, `strncat`, `strcmp`, and `strncmp` procedures, as described in chapter 5 of Deitel & Deitel. The first version of each procedure should use array subscripting, and the second should use pointers and pointer arithmetic.

We will provide skeleton code for each of the functions and a driver program. You will only have to fill in the various functions.

Basic String Manipulation (10 Points)

Write a program that uses random-number generation to create sentences. The program should use four arrays of `char*` pointers called `article`, `noun`, `verb`, and `preposition`. The program should create a sentence by selecting a word at random from each array in the following order: `article`, `noun`, `verb`, `preposition`, `article` and `noun`. As each word is picked, it should be concatenated to the previous words in an array that is large enough to hold the entire sentence. The words should be separated by spaces. When the final sentence is output, it should start with a capital letter and end with a period. The program should generate sentences.

The arrays should be filled as follows:

```
article : "the" "a" "one" "some" "any"
noun    : "boy" "girl" "dog" "town" "car"
verb    : "drove" "jumped" "ran" "walked" "skipped"
preposition : "to" "from" "over" "under" "on"
```

Basic Array Manipulation (10 Points)

Figure 3.10 of Deitel & Deitel plays the dice game craps. Write a modified version of that program to play 1000 games of craps. The program should keep track of the following statistics and answer the following questions:

- How many games are **won** on the first roll, second roll, third roll ...twentieth roll, and after the twentieth roll?
- How many games are **lost** on the first roll, second roll, third roll ...twentieth roll, and after the twentieth roll?
- What are the chances of winning at craps? (*Note:* You should discover that craps is one of the fairest casino games. What does this mean?)
- What is the average length of a game of craps?
- Do the chances of winning improve with the length of the game?

For this part of the homework, we will provide code that turns the program of Figure 3.10 into a procedure. Then you will have to implement the machinery to run the trials, keep track of the statistics, and answer the questions.

Advanced Pointer Manipulation (20 Points)

Problem 5.24 from Deitel & Deitel describes the *quicksort* algorithm for sorting an array of values, in this case integers. Implement a program that generates an array of 100 random integers between 0 and 1000, runs quicksort on the array to sort it, and then checks to make sure that the array is actually sorted.

We will provide skeleton code to drive the program. You will have to implement the `partition` and `qsort` functions as well as the code to generate the random array and the sorting test.

1-dimensional Cellular Automata (20 Points)

Note, if you are not familiar with base 2, or binary number representations, you should read the first four sections of Appendix C in Deitel & Deitel. We will briefly discuss binary numbers in class though.

Implement a program that can read command line input and execute a 1-dimensional elementary cellular automaton.

To remind you what such a system is, for the automaton we have a 1-dimensional grid of n cells in which each cell can hold the value 0 or 1. The system is given some initial set of values and then stepped sequentially through time. On each time step, every cell changes its values based upon its current value and the values of its left neighbor and right neighbor. How it changes can be defined by a set of rules that might be written like so:

```
111 110 101 100 011 010 001 000
--- --- --- --- --- --- --- ---
 0  1  0  1  1  0  1  0
```

On the top row we have a sequence of cells, with the cell we want to update in the middle, surrounded by its two neighbors. The bottom row indicates the new value of the middle cell.

Finally, on each step of the machine, every cell updates simultaneously in parallel. Thus, we have to be careful about keeping the old values of cells around since their neighbors need those values to calculate their own values.

We can treat each segment of the upper row as a binary number. Converting to decimal our rule looks like the following:

```
 7  6  5  4  3  2  1  0
--- --- --- --- --- --- --- ---
 0  1  0  1  1  0  1  0
```

Which leads to a natural representation where we keep the transformation rules as an array. To use this array, we convert a cell and its neighbors into an index into this array to determine a cell's next value. We can also treat the bottom row in its entirety as a binary number. So in our example, the row would equal decimal 90. This means that a single decimal number between 0 and 255 completely specifies our rules table.

Here is what your program should do:

- Take from the command line arguments, an integer representing the rules in decimal, and the number of steps to run in decimal.
- Convert the decimal rules number into a rules array.
- Iterate the automaton for the number of steps given on the command line, using the rules array to update the cells appropriately.
- On each iteration print out the current grid, by printing out a 1 if the cell holds one, or a blank space if the grid holds zero.
- Run on a grid of size $n + 3$. Where n is an even number. This leaves two cells on either end, which should always be 0, and 1 cell in between two $n/2$ halves. The center cell will be initialized to 1. For the purposes of this assignment, set $n = 40$, although for testing you may want to make it smaller.

Below is a sample run of the intended output, with $n = 20$.

```
bmd@BMD-LT-3A[/cygdrive/c/bmd/211-Winter-2004/lab2]$ ./ca 90 10
Rules: [0 1 0 1 1 0 1 0 ]
0      [                1                ]
1      [                1  1                ]
2      [                1    1                ]
3      [                1  1  1  1                ]
4      [                1      1                ]
5      [                1  1      1  1                ]
6      [                1    1    1    1                ]
7      [                1  1  1  1  1  1  1  1                ]
8      [                1      1                ]
9      [                1  1                ]
```

Once your program is working you should see interesting behaviors for rules 30, 90, 54, 122, 126, 182, 250, and 254, among others. We will post output for these rules on the class website so you know what to expect.

Extra Credit (20 Points)

Extend your 1 dimensional cellular automata so that the grid is really a *torus*. The ends conceptually wrap around to touch each other so that the far left end of the grid is the right neighbor of the far right end and vice versa. This means that those end points may change over time.

Also, instead of always initializing the grid with one cell “on” in the middle, initialize all the cells randomly to 1 or 0 with the odds of a 1 being 1 in 5. Then iterate the automaton for 50 cycles and calculate the

percentage of zeroes and ones in the final grid. Do this 10 times (each run should be different since the initialization is random) for rules 182 and 18, and report your results.