National University of Singapore
School of Continuing and Lifelong Education

TIC1001: Introduction to Computing and Programming I
Semester I, 2019/2020

Tutorial 5
Strings and Arrays

1. What is printed by the following program? You should try to hand-trace the program to obtain the answer before running the program to verify.

```c
#include <stdio.h>

void printArray(int list[], int numElem);
void passElement(int num);
void changeElements(int list[]);
void copyArray(int list1[], int list2[], int numElem);

int main(void) {
    int list1[5] = {11, 22, 33, 44, 55};
    int list2[5] = {99, 99, 99, 99, 99};
    printArray(list1, 5);
    passElement(list1[0]);
    printArray(list1, 5);
    changeElements(list2);
    printArray(list2, 5);
    copyArray(list2, list1, 5);
    printArray(list2, 5);
    return 0;
}

void printArray(int list[], int numElem) {
    int i;
    for (i = 0; i < numElem; i++)
        printf("%d ", list[i]);
    printf("\n");
    return;
}

void passElement(int num) {
    num = 1234;
    return;
}

void changeElements(int list[]) {
    list[2] = 77;
    list[4] = 88;
    return;
```

```
}

void copyArray(int list1[], int list2[], int numElem) {
    int i;
    for (i = 0; i < numElem; i++)
        list1[i] = list2[i];
    return;
}
```

2. The sieve of Eratosthenes is a simple algorithm for making tables of primes. The idea comes from a simple observation: multiples of a prime are not prime themselves. So start with a list of numbers from 2 to n. As 2 is the smallest prime number, proceed to cross out all other multiples of 2 (i.e. even number) in the list. The smallest remaining number is 3, which is the next prime. Cross out all other multiples of 3 will yield 5 as the next prime. Keep crossing all multiples of primes until no numbers can be crossed out. The remaining numbers are all primes within 2 to n.

   Write a function that receives an integer n ($n > 1$) and prints a list of primes from 2 to n inclusive. A boolean array `primes` can be used to represent the sieve, i.e., an array of boolean values, such that `primes[i]` is true when i is prime; otherwise it is false. We will accomplish this in several steps:

   (a) Define a function `void init_primes(bool primes[], int n)` that takes in the array primes together with its size, and initialize all the numbers from 2 to n to be prime. That is to say, since no sieve is performed yet, we assume all numbers to be primes in the beginning.

   (b) Define a function `void sieve_primes(bool primes[], int size, int n)` that takes in array primes together with its size and a number n, and performs the sieve as described above with n, i.e., it "crosses put" all multiples of n.

   (c) Define the function `print_primes(int n)` which takes in an integer n ($n > 1$) and prints a list of prime numbers from 2 to n inclusive.

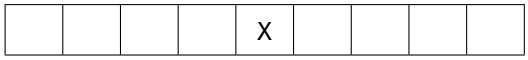
   Within this function, you should declare a boolean array `primes` that is suitably sized to accommodate numbers up to and including the input n, and use the above-defined functions to achieve the task.

   For example, calling `print_primes(50)` will result in the output:

   ```
   2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
   ```

3. Conway's Game of Life is a cellular automaton devised by the British mathematician John Horton Conway in 1970 that has attracted much interest, because of the surprising ways in which patterns can evolve. In this exercise, we shall look at a one-dimensional version of the game of life.

   A population can be represented by a row of cells where each cell represents an organism. Each cell may be alive or dead. Assume a population of nine cells with only one living organism (marked 'X') in the initial state (or generation).



For each generation, a cell is alive or dead depending on its own previous state and the previous states of the two neighbour cells. We adopt the following rules:

(a) If a cell is alive in one generation, it will be dead in the next.

(b) If a cell is dead in one generation, but has one and only one live neighbour cell, it will be alive in the next generation.

Applying the above rules to the initial generation above, we obtain the next generation

Applying the rules again will generate

Your task is to write a function `game_of_life(bool cells[], int size, int num_gen)`, which takes as inputs an array $cells$ of length $size$ representing the row of cells, and the number of generations to play. The function will print the output of each generation from 1 to n using underscore (_) as blank and `X` as a cell.

You might wish to define the following functions to help your implementation:

(a) `print_cells` to print a given row of cells.

(b) `next_gen` which takes in a row of cells and generate the next generation.

Experiment with different sizes and initial starting state and see the results.