CS1010E Programming Methodology
Semester 1 2016/2017

Week of 17 October – 21 October 2016
Tutorial 8 Suggested Answers
Composite Data Types
Structures and Arrays

1. We would like to develop a program to perform addition of two positive fractions.

   (a) Define a structure named `Fraction` with integer members `num` and `den` representing the numerator and denominator of the fraction respectively.

   (b) Write a function `addFraction` that takes as argument two fractions f and g and returns the fraction sum $f + g$.

   ```
   Fraction addFraction(Fraction f, Fraction g);
   ```

   $$\mathcal{F}_+(f, g) = \frac{f_n}{f_d} + \frac{g_n}{g_d} = \frac{(f_n \times g_d) + (g_n \times f_d)}{f_d \times g_d}$$

   (c) Write a `main` function to request two fractions from the user, adds the two fractions, and outputs the resulting fraction. A sample run of the program is given below. User input is underlined.

   ```
   Enter two fractions: 1/2 1/3
   Their sum is 5/6.
   ```

   Use the `scanf` conversion specifier `"%d/%d"` to read one fraction. What is the result when 1/3 and 1/3 are added together?

   (d) In order to simplify a function, e.g. 2/4 to 1/2, both numerator and denominator must be divided with their greatest common divisor (gcd). The function $gcd(m, n)$ can be computed using Euclid's algorithm.
   Write a function `gcd` (recursive or otherwise) to compute the greatest common divisor of two given integers m and n, where $m \geq 0$ and $n > 0$.

   ```
   int gcd(int a, int b);
   ```

   Note that the greatest common divisor returned will always be positive given the constraints on the arguments m and n.

   (e) By making a suitable call to the gcd function, modify the `addFraction` function such that the fraction returned is in its simplest form.

```c
#include <stdio.h>

typedef struct {
   int num, den;
} Fraction;

int gcd(int a, int b);
Fraction addFraction(Fraction f1, Fraction f2);

int main(void) {
   Fraction f, g, ans;

   printf("Enter two fractions: ");
   scanf("%d/%d", &(f.num), &(f.den));
   scanf("%d/%d", &(g.num), &(g.den));

   ans = addFraction(f,g);
   printf("The sum is %d/%d.\n", ans.num, ans.den);

   return 0;
}

/*
   Function addFraction takes in two fraction arguments f and g,
   and returns the fraction sum of f + g
*/
Fraction addFraction(Fraction f, Fraction g) {
   Fraction ans;
   int div;

   ans.num = f.num*g.den + f.den*g.num;
   ans.den = f.den*g.den;
   div = gcd(ans.num,ans.den); // gcd(num,den)
   ans.num = ans.num / div;    // divide num with the gcd
   ans.den = ans.den / div;    // divide den with the gcd

   return ans;
}

/*
   Function gcd returns the greatest common divisor of two arguments
   a and b using Euclid's algorithm.
*/
int gcd(int m, int n) {
   return n == 0 ? m : gcd(n,m%n);
}
```

2. What is printed by the following program? You are advised to hand-trace the program to obtain the answer before running the program to verify.

```
11  22  33  44  55
11  22  33  44  55
99  99  77  99  88
11  22  33  44  55
```

```
#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);

   /* list1[0] evaluated to 11 and passed-by-value */
   passElement(list1[0]);
   printArray(list1, 5);

   /* list2 array is referenced by changeElements */
   changeElements(list2);
   printArray(list2, 5);

   /* list2 and list1 are referenced by copyArray */
   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;
}
```

3. 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 program that receives an integer n ($n > 1$) from the user and outputs a list of primes from 2 to n inclusive. Use a boolean array `prime`, i.e. an array of boolean values, such that `prime[i]` is `true` when i is prime; otherwise it is `false`.

   (a) Assuming that the user input n will not exceed 10000, declare a boolean array `prime` of a suitable size within the `main` function.

   (b) Define a function `initPrimes` that takes in the array `prime` together with a value n, and initializes all 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.

   ```
   void initPrimes(bool prime[], int n);
   ```

   (c) Define a function `sievePrimes` that takes in array `prime` together with a value n, and performs the sieve as described above.

   ```
   void sievePrimes(bool prime[], int n);
   ```

   (d) Write a `main` function that receives as user input the value n, initializes the prime table, performs the sieve, and outputs the primes from 2 to n. Define a function `printPrimes` that takes in the array `prime` (after sieving) together with a value n, to assist in the output task.

   ```
   void printPrimes(bool prime[], int n);
   ```

   For example, calling the function `printPrimes(prime,50)` results in the output:

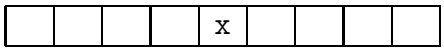
   ```
   2
   3
   5
   7
   11
   13
   17
   19
   23
   29
   31
   37
   41
   43
   47
   ```

```
#include <stdio.h>
#include <stdbool.h>
#define MAX 10001

void initPrimes(bool prime[], int n);
void sieve(bool prime[], int n);
void printPrimes(bool prime[], int n);

int main(void) {
   bool prime[MAX] = {false};
   int n;

   printf("Enter a number from 2 to %d: ", MAX);
   scanf("%d", &n);
   initPrimes(prime,n);
   sieve(prime, n);
   printPrimes(prime, n);
   return 0;
}

/*
   Function initPrimes sets the first n elements
   of the prime array to TRUE.
*/
void initPrimes(bool prime[], int n) {
   int i;

   for (i = 2; i <= n; i=i+1)
      prime[i] = true;
   return;
}

/*
   Function sieve performs prime-sieving until n.
   prime[i] is TRUE if i is prime and FALSE otherwise.
*/
void sieve(bool prime[], int n) {
   int i, j;

   for (i = 2; i < n; i=i+1) { /* or i < ceil(sqrt(n)) */
      if (prime[i]) {
         for (j = i*2; j <= n; j=j+i) {
            prime[j] = false;
         }
      }
   }
   return;
}
```

```
/*
   Function printPrimes prints a list of prime numbers from 2 to n.
   printPrimes must be called after sieve.
*/
void printPrimes(bool prime[], int n) {
   int i;

   for (i = 2; i <= n; i=i+1) {
      if (prime[i]) {
         printf("%d\n", i);
      }
   }
   return;
}
```

4. 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.

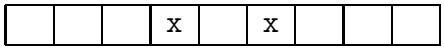
   A population can be represented by a row of cells where each cell represents an organism. Each cell may be alive or dead. Assume a population of nine cells with only one living organism (marked 'x') in the initial state (or generation).



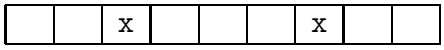
   For each generation, a cell is alive or dead depending on its own previous state and the previous states of the two neighbour cells. We adopt the following rules:

   - If a cell is alive in one generation, it will be dead in the next.
   - 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 generate the first 32 generations for a population size of 63, starting with a single organism `*` in the middle.

   Use the following as a guide to implement your program.

   (a) Declare an `int` array `cell` of an appropriate size with all elements set to zero.

   (b) Populate the middle of the array with an organism by setting the element value to `1`.

   (c) Define a function `printArray` to output the n values of the array in terms of blank or asterisk `*` characters.

```
void printArray(int cell[], int size);
```

   (d) Define the function `gameOfLife` to "play" the game of life

```
void gameOfLife(int cell[], int size, int numGen);
```

   For every generation, call the `printArray` function to output the array.

   (e) You are encouraged to experiment with different sizes and number of generations.

```
#include <stdio.h>
#define MAX 100

void printArray(int cell[], int size);
void gameOfLife(int cell[], int size, int numGen);

int main(void) {
   int cell[MAX] = {0}, size, numGen;

   printf("Enter population size and number of generations: ");
   scanf("%d %d", &size, &numGen);
   gameOfLife(cell,size,numGen);
   return 0;
}

void gameOfLife(int cell[], int size, int numGen) {
   int temp[MAX] = {0};
   int i, j, liveN;

   cell[size/2] = 1;

   printArray(cell, size);
   for (i = 2; i <= numGen; i++) {
      for (j = 0; j < size; j++) {
         if(cell[j] == 1) {
            temp[j] = 0;
         } else {
            liveN = 0;
            if ((j-1 >= 0) && cell[j-1] == 1) {
               liveN++;
            }
            if ((j+1 < size) && cell[j+1] == 1) {
               liveN++;
            }
            if (liveN == 1) {
               temp[j] = 1;
            } else {
               temp[j] = 0;
            }
         }
      }
      for (j = 0; j < size; j++) {
         cell[j] = temp[j];
      }
      printArray(cell, size);
   }
   return;
}
```

```
void printArray(int cell[], int size) {
   int i;
   for (i = 0; i < size; i++) {
      if (cell[i] == 1) {
         printf("*");
      } else {
         printf(" ");
      }
   }
   printf("\n");
   return;
}
```