Strings

Problem 1: (20%) Escriba un programa que realice las siguientes operaciones sobre cadenas de caracteres (strings); no usar memoria dinámica:

a) **int longitud_str(str)**: Función que regrese la longitud de un string `str` (sin contar el caracter nulo);

b) **char *copia_str(dst, src)**: copia el string `src` a `dst`, incluyendo el caracter nulo `\0`; regresa un apuntador a `dst`. **Nota:** La función debe verificar que los strings no estén superpuestos y si hay suficiente espacio en `dst` para copiar `src`; en caso que no, incluir solo hasta donde sea posible almacenar en `dst`;

c) **int compara_str(str1, str2)**: Función que compare lexicográficamente dos strings y regrese un número mayor que, igual a o menor que cero, si `str1` es mayor que, igual a o menor que `str2`. El valor que regrese debe coincidir con la diferencia en magnitud del caracter actual comparado;

d) **int concatena_str(str1, str2, str3)**: Función que concatene tres strings (en el orden dado) separados por un espacio y alamacene en `str1`; debe regresar la nueva longitud de `str1`; **Nota**: La función debe verificar que los strings no estén superpuestos y si hay suficiente espacio en `str1` para almacenar todos los strings; en caso que no, incluir solo hasta donde sea posible almacenar en `str1`;

e) **int encuentra_str(str1, str2)**: Función que busque un sub-string `str1` en `str2` y regrese el número de veces que lo encuentra.

f) **int **frecuencia(str)**: Encuentre la frecuencia de cada elemento de `str`, y regrese un arreglo bidimensional donde se almacene la letra (1er columna) y frecuencia (2da columna). **Salida**: w — 1, r — 5, s — 4, etc... **Nota**: En este ejercicio puede usar memora estática o dinámica.

g) **char *sin_repetir(str)**: Función que encuentre las palabras en `str` que no tengan letras repetidas y las imprima (dentro de la misma función por simplicidad).

Solution (a).

Listing 1: Programa que regresa la longitud de un string.

```
int longitud_str(const char *str) {
  int len = 0;
  while (*(str++)) {
    len++;
  }
  return len;
}
```

◇

Solution (b).

Listing 2: Programa que regresa la longitud de un string.

```
/*

We assume source is a null ending string.
*destiny is any char pointer.
dst_len is the length of destiny,
and we assume that the number passed is consistent
with the memory assigned to it.

 */

char *copia_str(char *destiny, const char *source, int dst_len) {
  char *init = destiny; // store dst's starting memory address

  if (dst_len < longitud_str(source)) {
    fprintf(stderr, "destiny does not have\
---- enough space to copy source.\n");
    exit(1);
  }

  if ((destiny < source + longitud_str(source) && destiny > source) ||
      (destiny < source && source < destiny + dst_len)) {
    fprintf(stderr, "Strings overlap in memory. Exiting program.\n");
    exit(2);
  }
  do {
    *destiny = *source;

    // do this while any of them are not the null char. By this
    // line, we are sure that source fits into destiny.
  } while (*(source++) | *(destiny++));
  return init;
}
```

◇

Solution (c).

Listing 3: Programa que regresa la longitud de un string.

```
int compara_str(const char *str1, const char *str2) {
  do {
    if (*str1 > *str2) {
      return *str1 - *str2;
    }
    if (*str1 < *str2) {
      return *str1 - *str2;
    }
    // loop while both are not '\0'. Note that we do not use &&
    // to prevent short-circuiting the statement, allowing us
    // to always move both pointers
  } while (*++str1 & *++str2);
  // if both have been equal until one (or both) reaches the char
  // '\0', we return  the distance of these two last chars
  return *str1 - *str2;
}
```

◇

Solution (d).

Listing 4: Programa que regresa la longitud de un string.

```
int concatena_str(char *str1, const char *str2, const char *str3) {
  // const pointer to start of str1, we will
  // return this after all concatenations are done
  char const *str1_copy = str1;
  // set pointer to null character of str1
  while (*(str1++))
    ;
  str1--; // go back so we can replace null character with space char
  *str1 = ' ';

  str1++;
  do { // set characters on str1 until null char of str2 is reached
    *(str1++) = *(str2++);
  } while (*str2);

  *str1 = ' ';
  str1++;

  do {
    *(str1++) = *(str3++);
  } while (*str3);

  *str1 = '\0'; // end str1 with null character so that
                // we can print it without
                // having memory problems

  // return length using pointer to start of str1
  return longitud_str(str1_copy);
}
```

◇

Solution (e).

Listing 5: Programa que regresa la longitud de un string.

```
int encuentra_str(const char *str1, const char *str2) {
  if (*str1 == '\0' ||
      *str2 == '\0') // If either of the strings are empty, return 0
    return 0;
  int occurrences = 0; // occurrence counter
  char const *cpy_str1 = str1;
  // constant pointer to where str1 starts, so we may reset str1
  // pointer after making a character match in the
  // following nested while loop
  while (*str2) {
    // while str2 does not point to the null terminating
    // character, we continue searching for occurrences of str1

    // this initializes a loop once we find that the first
    // character of str1 matches some character of str2
    while (*str1 == *str2) {
      str1++;
      str2++;
      // if we reach the end of str1, we have found one occurrence
      if (*str1 == '\0') {
        occurrences++;
        str2--; // go back a position in str2, as we will add it back when
                // exiting this loop
        str1 = cpy_str1; // reset str1 pointer position for next possible
                         // character match
        break;
      }
    }
    str2++;
  }
  return occurrences;
}
```

◇

Solution (f).

Listing 6: Programa que regresa la longitud de un string.

```
int **frecuencia(const char *str) {
  if (*str == '\0')
    return 0;

  // assuming 26 lower-case letters a-z (no ~n)
  int **freq = malloc(26 * sizeof(*freq));
  for (int i = 0; i < 26; i++) {
    // allocate space for each letter and its repetition value
    int *char_freq = malloc(2 * sizeof(int));
    *(char_freq) = 'a' + i;  // initializing array, a = 0th index
    *(char_freq + 1) = 0;    // initialize all counts to 0
    *(freq + i) = char_freq; // point freq[i] to each array
  }

  while (*str) {
    // str points to lowercase char
    if (*str - 'a' >= 0 && *str - 'a' < 26)
      *(*(freq + (*str - 'a')) + 1) += 1; // storing it in freq[str - a
          ][1]
    str++;
  }

  return freq;
}
```

◇

Solution (g).

Listing 7: Programa que regresa la longitud de un string.

```
char *sin_repetir(const char *str) {
  char word[90];     // storage for current word being analyzed.
  char *ptr = word; // pointer to start of array.
                    // This will be the object we
                    //  will be iterating to set the word array contents

  while (*str) {
    *ptr = *str;
    ptr++;
    str++;
    // We enter when a word is formed (that is, a space
    // character is found, or string str has ended).

    if (*str == '-' || *str == '\0') {
      *ptr = '\0'; // End string so it can be processed by frecuencia
          function

      // we then reset pointer to initial position, so that the pointer
          passed
      // in to frecuencia is properly analyzed from start to finish
      ptr = word;
      int **freq = frecuencia(ptr); // char frequency array
      for (int i = 0; i < 26; i++) {
        if (freq[i][1] > 1) // if one char is repeated, end loop
          break;
        if (i == 25) { // if none have been repeated, print word
          printf("%s\n", word);
        }
      }
    }
  }
  return ptr;
}
```

◇

Arreglos Bidimensionales

Problem 2: (10%) Escribir un programa que genere una caminata aleatoria en una matriz de 10x10. El arreglo debe contener inicialmente puntos '.', y debe recorrerse basado en el residuo de un número aleatoria (usar **srand()** y **rand()**) cuyos resultado puede ser 0 (arriba), 1 (abajo), 2 (izq), 3 (der), que indican la dirección a moverse. A) Verificar que el movimiento no se salga del arreglo de la matriz, y B) No se puede visitar el mismo lugar más de una vez. Si alguna de estas condiciones se presenta, intentar moverse hacia otra dirección definida; si todas las posiciones están ocupadas, finalizar el programa e imprimir el resultado.

Solution.

Listing 8: Programa que regresa la longitud de un string.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

#define N 10
#define M 10

#define x_0 0
#define y_0 0

char board[N][M];

void initialize_board(char board[N][M]) {
  char *p, *END = &board[0][0] + M * N;
  p = &board[0][0];
  int i;

  for (p = board[0], i = 0; p < END; p++, i++) {
    *p = '.';
  }
}

void print_board() {
  printf("\033[H");
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
      printf("%c ", board[i][j]);
    }
    printf("\n");
    fflush(stdout);
  }
}

int main() {
  initialize_board(board);
  printf("\033[2J");

  srand(time(NULL));
  char *current = &board[x_0][y_0];
  unsigned char letter = '.' + 1; // avoid '.' char
  *current = letter++; // set first tile to first character, then
      increment char
                       // for next usage

  char *POS_0M = &board[0][M], *POS_N0 = &board[N][0]; // helper pointers
  short int iter = 0;                                  // iteration count
  // pos_x is used to keep track of column position in matrix of current
      block.
  // We use this to check if we can go left or right
  short int pos_x = x_0;
```

```
  unsigned char direction, blocked = 0;
  /*
  we use 4 bits (half a byte) to describe directions:
  0000 0001 = up
  0000 0010 = down
  0000 0100 = left
  0000 1000 = right

  blocked represents the directions that have been proven to be blocked
 */

  while (iter < 1000) {
    if (blocked == 15) {
      break; // if blocked == 00001111 (all directions are blocked), exit
          loop
    }
    iter++;
    direction = 1 << (rand() % 4);
    if (direction & blocked) { // if direction randomly chosen has at
        least one
                               // bit in common with blocked choices, try
                                  again
      continue;
    }
    switch (direction) {
    case 1:
      if (current < POS_0M ||
          *(current - N) != '.') // if current position is in first row or
              north
                                 // character is not a '.', skip
      {
        blocked |= direction; // add blocked direction to blocked variable
        break;
      } else {
        current -= N; // go back N * sizeof(char) = N bytes. This places
            us +1
                      // position north
        *current = letter++; // set the new current position to letter,
            then
                             // increment the letter.

        // after moving, we block the square we come from. For
        // example, if we move north (this case), then we block
        // south, which is 0000 0010 (direction << 1)
        blocked = direction << 1;
        print_board();
        usleep(200000);
        break;
      }
    case 2:
      if (current > POS_N0 || *(current + N) != '.') {
        blocked |= direction;
        break;
```

```
      } else {
        current += N; // go 1 block south in matrix
        *current = letter++;
        blocked = direction >> 1; // since we moved south, blocked is 0000
            0001
        print_board();
        usleep(200000);
        break;
      }
    case 4:
      if (pos_x == 0 || *(current - 1) != '.') {
        blocked |= direction;
        break;
      } else {
        current--;                  // go left
        *current = letter++;        // set current position to letter
        blocked = direction << 1; // since we moved left, blocked is 0000
            1000
        pos_x--;                    // update column position
        print_board();
        usleep(200000);
        break;
      }
    case 8:
      if (pos_x == M - 1 || *(current + 1) != '.') {
        blocked |= direction;
        break;
      } else {
        current++;                  // go right
        *current = letter++;        // set current char to letter, then
            update
        blocked = direction >> 1; // since we moved right, blocked is 0000
            01000
        pos_x++;                    // update column position
        print_board();
        usleep(200000);
        break;
      }
    default:
      fprintf(stderr, "Invalid direction.\n");
      exit(1);
    }
  }
}
```

◇

Problem 3: (10%)

a) Dado un arreglo bidimensional de enteros $M \times N$, encontrar el máximo valor para cada columna y cada renglón:

	9	6	8	1
8	3	6	8	1
4	2	4	3	1
9	9	5	2	1

Nota: Recorre el arreglo (columnas y renglones) en forma eficiente.

b) Cuente el número de bytes del arreglo bidimensional con valor 0 (recuerde que cada entero está representado por 4 bytes).

Solution.

◇

Memoria Dinámica

Problem 4: (20%) Dado una lista de nombres (string) de N personas (apellido_paterno, apellido_materno y nombre(s)), escribir una función que ordene los nombres alfabéticamente usando un arreglo de apuntadores:

```
char **crea_arreglo(char **arr, ...)
char **ordena(char **arr,...)
```

Los nombres pueden tener distinta longitud, pero la memoria que ocupan debe ser la justa (sin desperdicio); cuando un nombre sea prefijo de otro, considerar al nombre más corto como menor. El ordenamiento debe ser a través de una función que reciba el arreglo de apuntadores.

Solution.

◇

Problem 5: (20%) Separe un string en tokens de acuerdo a un caracter especial dado como entrada (puede ser espacio, /, %, etc.) y que regrese un arreglo que apunte a cada uno de los tokens esperados:

```
char **tokens(char *str, char ch)
```

o NULL en caso de no encontrar algún token.

Solution.

◇

Problem 6: Escriba una función que reciba N arreglos de enteros ordenados de menor a mayor, y mezcle los arreglos en un solo arreglo ordenado de igual forma.
Prototipo de la función: **int *merge(int **arr, int N, int *dim)** donde **arr** tiene la siguiente estructura:
N es el número total de arreglos y **dim** es un arreglo de enteroscon la dimensión de cada uno de los arreglos de entrada. La función regresa un apuntados hacia el arreglo mezclado y generado dinámicamente dentro de la función **merge()**.
Nota: Generar dinámicamente todos los arreglos necesarios, e inicialice cada arreglo con valores aleatorios mediante la función **rand()**.

Solution.

◇