Lab 3: Memory Organization

Task 1:

Row-major-form:

Begin by allocating a char byte array of size m*n*s, were m=number of columns, n=number of rows, s=size of each element.

Make element (0,2) and (1,0) 15.

Store first checks if row & columns are in the (m,n) bounds. If true, copy the value into position (r,c) in the matrix. The corresponding spot in the array is calculated by A + m*r*s + c*s, where A is the memory address for the array.

Row-major-form explanation: m is the number of columns in the matrix, or the amounts of elements in every row. So every row is a m*s sized chunk of memory and r tells you what chunk to take. s is the size of every element, so c*s tells you what element in a row to use.

For the fetching, we first check if the row & columns are in the (m,n) bounds. If true, return the value in position (r,c) in the matrix. The corresponding spot in the array is calculated by m*r*s + c*s. And if false, return -1.

We also made a function to print out the matrix. It loops over every row and column, and then prints value in that position.

End by deallocating the allocated memory with “free”.

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>


/* Row Major Form explanation
 * m is the number of culomns in the matrix
 * or the amount of elements in every row.
 * So every row is a m * s sized chunk of memory
 * and r tells you which chunk to take.
 * s is the size of every element so c * s
 * tells you what element in a row to use.
 * Which gives you your row and column. */

char* two_d_alloc(int n, int m, int s) {
    // Allocates array of size n * m* s
    char* array = malloc(n*m*s);
    // Fills array with 0
    for(int i = 0; i < n*m*s; i++)
        array[i] = 0;
    // Makes the end 127 for no apparent reason
    array[n*m*s - 1] = 127;
    return array;
}

int byteCount(char* a){
    // Made for testing
    int i = 1;
    while(a[i - 1] != 127)
        i++;
```

```
    return i;
}

void two_d_dealloc(char* c){
    free(c);
}

void two_d_store(int n, int m, int r, int c, int s, int v, char* A){
    // If the row r and column c is within the bounds
    if(r < n && c < m)
        // Copies from input v to A(r,c), with row major form
        memcpy(A + m*r*s + c*s, &v, s);
    printf("%d\n", m*r*s + c*s);
}

int two_d_fetch(int n, int m, int r, int c, int s, char* A){
    // If the row r and column c is within the bounds
    if(r < n && c < m)
        // If true return element A(r,c), with row major form
        return A[m*r*s + c*s];
    // If not return -1
    return -1;
}

void printArray(char* a, int n, int m, int s){
    // Print out array in matrix form
    // Loop over every row
    for(int i = 0; i < n; i++){
        // Loop over every column
        for(int j = 0; j < m; j++){
            // Print value in index
            printf("%d ", a[m*i*s + j*s]);
        }
        // At end of row, newline
        printf("\n");
    }
}
```

Column-major-form:
The difference between column and row-major-form is calculating the spot in the array, is now done by r*s + n*c*s + A.
Row-major-form explanation: n is the number of rows in the matrix, or the amounts of elements in every column. So every column is a n*s sized chunk of memory and c tells you what chunk to take. s is the size of every element, so r*s tells you what element in a column to use.

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

/* Column Major Form explanation
 * n is the number of rows in the matrix
```

```
 * or the amount of elements in every column.
 * So every column is a n * s sized chunk of memory
 * and c tells you which chunk to take.
 * s is the size of every element so r * s
 * tells you what element in a column to use.*/

char* two_d_alloc(int n, int m, int s) {
    // Allocates array of size n * m* s
    char* array = malloc(n*m*s);
    // Fills array with 0
    for(int i = 0; i < n*m*s; i++)
        array[i] = 0;
    return array;
}

int byteCount(char* a){
    // Made for testin
    int i = 1;
    while(a[i - 1] != 127)
        i++;
    return i;
}

void two_d_dealloc(char* c){
    free(c);
}

void two_d_store(int n, int m, int r, int c, int s, int v, char* A){
    // If the row r and column c is within the bounds
    if(r < n && c < m)
        A[r*s + n*c*s] = v;
        // Copies from input v to A(r,c)
    printf("%d\n", r*s + n*c*s);
}

int two_d_fetch(int n, int m, int r, int c, int s, char* A){
    // If the row r and column c is within the bounds
    if(r < n && c < m)
        // If true return element A(r,c)
        return A[r*s + n*c*s];
    // If false return -1
    return -1;
}

void printArray(char* a, int n, int m, int s){
    for(int i = 0; i < n; i++){
        for(int j = 0; j < m; j++){
            printf("%d ", a[i*s + n*j*s]);
        }
```

```
        printf("\n");
    }
}
#include <stdio.h>

extern char* two_d_alloc(int row, int column, int s);
extern void two_d_dealloc(char* buffer);
extern int printArray(char* a, int n, int m, int s);
extern void two_d_store(int n, int m, int r, int c, int s, int v, char* A);
extern int two_d_fetch(int n, int m, int r, int c, int s, char* A);

int main() {
    // Allocate a char array of size n*m*s
    int n = 2;
    int m = 3;
    int s = sizeof(int);
    char* b = two_d_alloc(n, m, s);

    // Add 15 in row 0 column 2 and row 1 column 0
    two_d_store(n, m, 0, 2, s, 15, b);
    two_d_store(n, m, 1, 0, s, 15, b);

    // Fetch within array and print result
    int op = two_d_fetch(n, m, 1, 0, s, b);
    if(op != -1)
        printf("%d\n", op);
    else
        printf("OUT OF BOUNDS\n");

    // Fetch outside array and print result
    op = two_d_fetch(n, m, 10, 10, s, b);
    if(op != -1)
        printf("%d\n", op);
    else
        printf("OUT OF BOUNDS\n");

    // Prints the array in matrix form
    printArray(b, n, m, s);
    two_d_dealloc(b);
    return 0;
}
```

Pointers:
The same procedure as row-major-form, but we work with integer pointers, instead of integers by value.

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

```c
/* Row Major Form explanation
 * m is the number of culomns in the matrix
 * or the amount of elements in every row.
 * So every row is a m * s sized chunk of memory
 * and r tells you which chunk to take.
 * s is the size of every element so c * s
 * tells you what element in a row to use.*/

char* two_d_alloc(int n, int m, int s) {
    // Allocates array of size n * m* s
    char* array = malloc(n*m*s);
    // Fills array with 0
    for(int i = 0; i < n*m*s; i++)
        array[i] = 0;
    // Makes the end 127 for no apparent reason
    array[n*m*s - 1] = 127;
    // Return array
    return array;
}

int byteCount(char* a){
    // UNUSED
    int i = 1;
    while(a[i - 1] != 127)
        i++;
    return i;
}

void two_d_dealloc(char* c){
    free(c);
}

void two_d_store(int* n, int* m, int* r, int* c, int* s, int* v, char* A){
    // Check to see if r and c are within the bounds of the matrix
    if(*r < *n && *c < *m)
        // Store pointers value in array
        A[(*m)*(*r)*(*s) + (*c)*(*s)] = *v;
}

int* two_d_fetch(int* n, int* m, int* r, int* c, int* s, char* A){
    // Check to see if r and c are within the bounds of the matrix
    if(*r < *n && *c < *m)
        // If true return the memory addres of element A(r,c)
        return (int*) &A[(*m)*(*r)*(*s) + (*c)*(*s)];
    // If not true return NULL
    return NULL;
}
```

```c
void printArray(char* a, int n, int m, int s){
    // Prints the entire array in matrix form
    // Loops over every row
    for(int i = 0; i < n; i++){
        // Loops over every column
        for(int j = 0; j < m; j++){
            // Prints the value of the element
            printf("%d ", a[m*i*s + j*s]);
        }
        // End of every row newline.
        printf("\n");
    }
}
```

```c
#include <stdio.h>

extern char* two_d_alloc(int row, int column, int size);
extern void two_d_dealloc(char* buffer);
extern int printArray(char* a, int n, int m, int s);
extern void two_d_store(int* n, int* m, int* r, int* c, int* s, int* v, char*
A);
extern int* two_d_fetch(int* n, int* m, int* r, int* c, int* s, char* A);

int main() {
    // Allocate array of size n * m * s
    int n = 2;
    int m = 3;
    int s = sizeof(int);
    char* b = two_d_alloc(n, m, s);
    // Memes
    int zero = 0;
    int one = 1;
    int two = 2;
    int fifteen = 15;
    int ten = 10;

    // Store 15 in array at pos (0,2) and (1,0)
    two_d_store(&n, &m, &zero, &two, &s, &fifteen, b);
    two_d_store(&n, &m, &one, &zero, &s, &fifteen, b);

    // Fetch within array and print result
    int* op = two_d_fetch(&n, &m, &one, &zero, &s, b);
    if(op)
        printf("%d\n", *op);
    else
        printf("OUT OF BOUNDS\n");

    // Fetch outside array and print result
    op = two_d_fetch(&n, &m, &ten, &ten, &s, b);
```

```
    if(op)
        printf("%d\n", *op);
    else
        printf("OUT OF BOUNDS\n");

    // Print array in matrix form
    printArray(b, n, m, s);
    two_d_dealloc(b);
    return 0;
}
```

Dynamic:
Rewrote functions to deal with dynamic data types by using void pointers.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/* Row Major Form explanation
 * m is the number of culomns in the matrix
 * or the amount of elements in every row.
 * So every row is a m * s sized chunk of memory
 * and r tells you which chunk to take.
 * s is the size of every element so c * s
 * tells you what element in a row to use.*/

char* two_d_alloc(int n, int m, void* s) {
    // Allocates an array of size n * m * sizeof(s)
    int sz = sizeof(*s);
    char* array = malloc(n*m*sz);
    // Fills the array with 0
    for(int i = 0; i < n*m*sz; i++)
        array[i] = 0;
    return array;
}

int byteCount(char* a){
    // UNUSED
    int i = 1;
    while(a[i - 1] != 127)
        i++;
    return i;
}

void two_d_dealloc(char* c){
    free(c);
}

void two_d_store(int n, int m, int r, int c, void* s, int v, char* A){
```

```
    int sz = sizeof(*s);
    // Checks if r and c are in bounds
    if(r < n && c < m)
        // If true copy v into the array at (r,c)
        memcpy(A + m*r*sz + c*sz, &v, sz);
    printf("%d\n", m*r*sz + c*sz);
}

void* two_d_fetch(int n, int m, int r, int c, void* s, char* A){
    int sz = sizeof(*s);
    // Checks if r and c are in bounds
    if(r < n && c < m)
        // Return A(r,c)
        return &A[m*r*sz + c*sz];
    // If not in bounds return -1
    return null;
}

void printArray(char* a, int n, int m, void* s){
    int sz = sizeof(*s);
    // Print entire array in matrix form
    for(int i = 0; i < n; i++){
        for(int j = 0; j < m; j++){
            printf("%d ", a[m*i*sz + j*sz]);
        }
        printf("\n");
    }
}
#include <stdio.h>

extern char* two_d_alloc(int row, int column, void* s);
extern void two_d_dealloc(char* buffer);
extern int printArray(char* a, int n, int m, void* s);
extern void two_d_store(int n, int m, int r, int c, void* s, int v, char* A);
extern void* two_d_fetch(int n, int m, int r, int c, void* s, char* A);

int main() {

    // Allocate a n * m * s sized array
    int n = 2;
    int m = 3;
    double d;
    char* b = two_d_alloc(n, m, &d);

    // Store 15 in position (0,2) and (1,0)
    two_d_store(n, m, 0, 2, &d, 15, b);
    two_d_store(n, m, 1, 0, &d, 15, b);

    // Fetch inside array and print result
```

```
    void* op = two_d_fetch(n, m, 1, 0, &d, b);
    if(op)
        printf("%d\n", *op);
    else
        printf("OUT OF BOUNDS\n");
    // Fetch outside array and print result
    op = two_d_fetch(n, m, 10, 10, &d, b);
    if(op)
        printf("%d\n", *op);
    else
        printf("OUT OF BOUNDS\n");

    // Print the entire array
    printArray(b, n, m, &d);
    two_d_dealloc(b);
    return 0;
}
```

Task 2:

Memory dump:

First we fill our array with the integers 0,1,2,…,n*m*s (their indexes).

To perform the memory dump we first loop through every byte in the array, while checking if it is the beginning of a word. If it is we print the memory address of that word.

Prints all the bytes of the word, and saves them to an array for use later. If at the end of a word, we start looping through the character array and print out the ACSII form of the byte if the ASCII representation exist and is legible, also prints out a new line character. If not, print ‘ö’.

```
#include <stdio.h>

extern char* two_d_alloc(int row, int column, int s);
extern void two_d_dealloc(char* buffer);
extern int printArray(char* a, int n, int m, int s);
extern void two_d_store(int n, int m, int r, int c, int s, int v, char* A);
extern int two_d_fetch(int n, int m, int r, int c, int s, char* A);

void fillArray(char* a, int n, int m, int s){
    // Fill the array with their index
    for(int i = 0; i < n*m*s; i++)
        a[i] = i;
}

void hexDump(char* a, int n, int m, int s){
    // Word sized array for use in printing characters
    char b[4];
    // Loops through every byte in array
    for(int i = 0; i < n*m*s; i++){
        /* If at the beggining of a word (4 bytes)
         * print the memory addres of the word (first byte)
         */
        if(i % 4 == 0)
            printf("mem: %p\n", a+i);
        // Print the byte at index 1
        printf("%02hhx ", a[i]);
        // Save the byte to character array
        b[i%4] = a[i];
        // If at the end of a word
        if(i % 4 == 3 && i != 0){
            // Loop through character array and
            // print out the bytes in ascii form.
            for(int j = 0; j < 4; j++){
                // If within ascii table of characters
                if(i < 130-j && i > 31)
                    printf("%c ", b[j]);
                // If outside ascii table write ö instead
                else
                    printf("ö ");
            }
            // Print endline at end of word
```

```
            printf("\n");
        }
    }
}

int main(){
    // Allocate array of size n * m * s
    int s = sizeof(char);
    int n = 1;
    int m = 200;
    char* str = two_d_alloc(n, m, s);
    // Fills the array
    fillArray(str, n, m, s);
    // Prints the array
    hexDump(str, n, m, s);
    // Deallocates the memory
    two_d_dealloc(str);
    return 0;
}
```

Task 3:

Linked list:

First create a struct for the linked list that holds pointer to the next node and an in value. Then fill array with nodes holding the next node and an integer value. The integer value is the nodes position in the array.

Then perform hexDump. In the hexDump we see both the memory address to the next node but also the value the nodes stores. In this example that value is the nodes position.

Memory address where the fifth node is located:

We start at 0

0xa23590

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

extern char* two_d_alloc(int row, int column, int s);
extern void two_d_dealloc(char* buffer);
extern int printArray(char* a, int n, int m, int s);
extern void two_d_store(int n, int m, int r, int c, int s, int v, char* A);
extern int two_d_fetch(int n, int m, int r, int c, int s, char* A);

// Linked list struct holds a pointer to the next node and an int value
typedef struct linkedList_{
    int d;
    struct linkedList_* next;
} linkedList;

void fillArray(char* a, int n, int m, int s){
```

```
    int zero = 0;
    linkedList* temp = malloc(s);
    // Fills array with nodes holding the next node and an integer value
    // Sets integer value to the nodes position in the array
    for(int i = 0; i < n*m; i++){
        // Allocates memory for linkeList
        linkedList* l = malloc(s);
        // Offset for ascii table
        l->d = i + 32;
        if(i){
            // Copys the nodes address into the previous nodes next
            memcpy(a + (i-1)*s + sizeof(char*), &l, sizeof(char*));
        }
        // Copies the node into the right position in the array
        memcpy(a+i*s, l, s);
    }
    // Sets the last nodes next to null
    memcpy(a+n*m*s - s + sizeof(char*), &zero, sizeof(char*));
}


void hexDump(char* a, int n, int m, int s){
    // Word size for use in printing
    int ws = 8;
    // Array of size word size for storing values for acsiing
    char b[ws];
    // Loops through entire array
    for(int i = 0; i < n*m*s; i++){
        // If at start of word print memory address of word
        if(i % ws == 0)
            printf("mem: %p\n", a+i);

        // Prints out the bytes of the array
        printf("%0hhx ", a[i]);
        // Stores bytes into secondary array for use in ascci representation
        b[i%ws] = a[i];
        // If at the end o a word start writing out ascii representation of wo
rd
        if(i % ws == ws-1 && i != 0){
            // Prints seperator between ascii and hex
            printf("| ");
            // Loops through secondary array
            for(int j = 0; j < ws; j++){
                // If the value is within ascii table
                if(b[j] < 128 && b[j] > 31)
                    // Prints ascii value
                    printf("%c ", b[j]);
                else
                    // If not print non ascii character.
```

```
                    printf("ö ");
            }
            // Print newline at end of word
            printf("\n");
        }
    }
}

int main(){
    // Allocate memory of size s * n * m
    int s = sizeof(linkedList);
    int n = 1;
    int m = 10;
    char* str = two_d_alloc(n, m, s);
    // Fills array
    fillArray(str, n, m, s);
    // Prints array
    hexDump(str, n, m, s);
    // Frees array
    two_d_dealloc(str);
    return 0;
}
```