Andreas Vasberg, Patrik Viitala

Lab 3 - Group 5

Task 1: Array Storage

This lab was very smooth, and this time, we had no trouble at the beginning with the setup. The only problem we had was that we didn't know how to implement a general version of two_d_store and two_d_fetch. It took some time to figure out that we could use void pointers.

```
char* two_d_alloc(int rows, int columns, int size)
{
    if(rows < 1 || columns < 1 || size < 1)
    {
        printf("Rows, columns and size have to be at least 1.\n");
        char* array = (char*)calloc(1, 1);
        return array;
    }
    else
    {
        char* array = (char*)calloc(rows*columns, size);
        return array;
    }
}

void two_d_dealloc(char* array)
{
    free(array);
    array = NULL;
}
```

In two_d_alloc we take as arguments the number of rows and columns we want and also the size of the picked datatype. For example int have a bit-size of 4. We then allocate a char pointer array with the length of rows*columns*size.
If the rows, columns or the size are below 1, the function prints an error message and allocates a 1x1 matrix with a size of 1.
Two_d_dealloc just deallocates the given array.

```
void two_d_store(int element, char array[], int size, int row, int column,
int rows, int columns)
{
    if(row > rows || column > columns || row < 1 || column < 1 || rows < 1
|| columns < 1)
    {
        printf("Index exceeds array bounds.\n");
```

```
    }
    else
    {
        int position = size*(row - 1)*rows + size*(column - 1);
        memcpy(array + position, &element, size);
    }
}

int two_d_fetch(char array[], int size, int row, int column, int rows, int
columns)
{
    if(row > rows || column > columns || row < 1 || column < 1 || rows < 1
|| columns < 1)
    {
        printf("Index exceeds array bounds.\n");
    }
    else
    {
        int position = size*(row - 1)*rows + size*(column - 1);
        int val;
        memcpy(&val, array + position, size);
        return val;
    }
}
```

Two_d_store takes the given element and puts it at the selected position in the array. The if-statement only ensures the given position exists and isn't negative. Else, the position in the array is calculated with a function that translates the row, column and size into one number. Two_d_fetch works the same way as two_d_store except it only return the value of the given selected position instead of storing a new element.

```
void two_d_store_column(int element, char array[], int size, int row,
int column, int rows, int columns)
{
    if(row > rows || column > columns || row < 1 || column < 1 || rows <
1 || columns < 1)
    {
        printf("Index exceeds array bounds.\n");
    }
    else
    {
        int position = size*(column - 1)*columns + size*(row - 1);
        memcpy(array + position, &element, size);
    }
}
```

```c
int two_d_fetch_column(char array[], int size, int row, int column, int
rows, int columns)
{
    if(row > rows || column > columns || row < 1 || column < 1 || rows <
1 || columns < 1)
    {
        printf("Index exceeds array bounds.\n");
    }
    else
    {
        int position = size*(column - 1)*columns + size*(row - 1);
        int val;
        memcpy(&val, array + position, size);
        return val;
    }
}
```

two_d_store_column and two_d_fetch_column behave in the same way as the previous two functions. The difference being how the position of the index is calculated. This function stores and fetches values in column-major form instead of row-major form.

```c
void two_d_store_ptr(int* element, char array[], int size, int row, int
column, int rows, int columns)
{
    if(row > rows || column > columns || row < 1 || column < 1 || rows <
1 || columns < 1)
    {
        printf("Index exceeds array bounds.\n");
    }
    else
    {
        int position = size*(row - 1)*rows + size*(column - 1);
        memcpy(array + position, element, sizeof(int*));
    }
}

char* two_d_fetch_ptr(char array[], int size, int row, int column, int
rows, int columns)
{
    if(row > rows || column > columns || row < 1 || column < 1 || rows <
1 || columns < 1)
    {
        printf("Index exceeds array bounds.\n");
    }
    else
```

```
    {
        int position = size*(row - 1)*rows + size*(column - 1);
        return array + position;
    }
}
```

Two_d_store_ptr and two_d_fetch_ptr behave in the same way as two_d_store and two_d_fetch except they are now storing and fetching the addresses to the selected value. The address is found in the pointer that is passed into the function.

```
void two_d_store_general(void* element, char array[], int size, int row, int column,
int rows, int columns)
{
    if(row > rows || column > columns || row < 1 || column < 1 || rows < 1 ||
columns < 1)
    {
        printf("Index exceeds array bounds.\n");
    }
    else
    {
        int position = size*(row - 1)*rows + size*(column - 1);
        memcpy(array + position, element, size);
    }
}

char* two_d_fetch_general(char array[], int size, int row, int column, int rows, int
columns)
{
    if(row > rows || column > columns || row < 1 || column < 1 || rows < 1 ||
columns < 1)
    {
        printf("Index exceeds array bounds.\n");
    }
    else
    {
        int position = size*(row - 1)*rows + size*(column - 1);
        return array + position;
    }
}
```

two_d_store_general behaves the same as two_d_store_ptr but stores the address of a void pointer instead of an int pointer. This enables it to store any data type. To access the stored value, the address must be cast to a pointer of the type that has been stored and then dereferenced.

```c
int main()
{
    // Step 1-3
    printf("Step 1-3 start:\n\n");

    char* test_array = two_d_alloc(2, 2, 4);
    two_d_dealloc(test_array);

    printf("\nStep 1-3 end.\n\n");

    // Step 4-6
    printf("Step 4-6 start:\n\n");

    char* row_array = two_d_alloc(3, 3, 4);
    int k = 1000;
    for(int i = 1; i < 4; i++)
    {
        for(int j = 1; j < 4; j++)
        {
            two_d_store(k, row_array, 4, i, j, 3, 3);
            k = k + 1000;
        }
    }
    for(int i = 1; i < 4; i++)
    {
        for(int j = 1; j < 4; j++)
        {
            printf("Index [%d, %d] = ", i, j);
            printf("%d\n", two_d_fetch(row_array, 4, i, j, 3, 3));
        }
    }

    printf("\nStep 4-6 end.\n\n");

    // Step 7
    printf("Step 7 start:\n\n");

    // Attempt to allocate empty array
    printf("Empty array test:\n");
    char* empty_array = two_d_alloc(0, 1, 2);
    free(empty_array);
    // Allocate boundary testing array
    char* boundary_array = two_d_alloc(4, 4, 4);
    // Attempt to store and fetch out of bounds
```

```
    printf("Out of bounds test:\n");
    two_d_store(10000, boundary_array, 4, 5, 1, 4, 4);
    two_d_fetch(boundary_array, 4, 3, 8, 4, 4);
    // Attempt to store and fetch negative indices
    printf("Negative and zero index test:\n");
    two_d_store(10000, boundary_array, 4, -1, 1, 4, 4);
    two_d_fetch(boundary_array, 4, 0, 1, 4, 4);

    printf("\nStep 7 end.\n\n");

    // Step 8
    printf("Step 8 start:\n\n");

    printf("Stored column-major, fetched row-major:\n");
    char* column_array = two_d_alloc(3, 3, 4);
    k = 1000;
    for(int i = 1; i < 4; i++)
    {
        for(int j = 1; j < 4; j++)
        {
            two_d_store_column(k, column_array, 4, i, j, 3, 3);
            k = k + 1000;
        }
    }
    for(int i = 1; i < 4; i++)
    {
        for(int j = 1; j < 4; j++)
        {
            printf("Index [%d, %d] = ", i, j);
            printf("%d\n", two_d_fetch(column_array, 4, i, j, 3, 3));
        }
    }

    printf("\nStored row-major, fetched column-major:\n");
    k = 1000;
    for(int i = 1; i < 4; i++)
    {
        for(int j = 1; j < 4; j++)
        {
            two_d_store(k, column_array, 4, i, j, 3, 3);
            k = k + 1000;
        }
    }
    for(int i = 1; i < 4; i++)
    {
```

```c
        for(int j = 1; j < 4; j++)
        {
            printf("Index [%d, %d] = ", i, j);
            printf("%d\n", two_d_fetch_column(column_array, 4, i, j, 3, 3));
        }
    }

    printf("\nStored column-major, fetched column-major:\n");
    k = 1000;
    for(int i = 1; i < 4; i++)
    {
        for(int j = 1; j < 4; j++)
        {
            two_d_store_column(k, column_array, 4, i, j, 3, 3);
            k = k + 1000;
        }
    }
    for(int i = 1; i < 4; i++)
    {
        for(int j = 1; j < 4; j++)
        {
            printf("Index [%d, %d] = ", i, j);
            printf("%d\n", two_d_fetch_column(column_array, 4, i, j, 3, 3));
        }
    }

    printf("\nStep 8 end.\n\n");

    // Step 9
    printf("Step 9 start:\n\n");

    char* ptr_array = two_d_alloc(3, 3, 4);
    int x = 1000; int y = 2000; int z = 3000;
    int* row1 = &x;
    int* row2 = &y;
    int* row3 = &z;
    for(int i = 1; i < 4; i++)
    {
        two_d_store_ptr(row1, ptr_array, 4, 1, i, 3, 3);
        two_d_store_ptr(row2, ptr_array, 4, 2, i, 3, 3);
        two_d_store_ptr(row3, ptr_array, 4, 3, i, 3, 3);
    }
    for(int i = 1; i < 4; i++)
    {
        for(int j = 1; j < 4; j++)
```

```
        {
            printf("%p = ", two_d_fetch_ptr(ptr_array, 4, i, j, 3, 3));
            printf("%d\n", *(int*)two_d_fetch_ptr(ptr_array, 4, i, j, 3, 3));
        }
    }

    printf("\nStep 9 end.\n\n");
    // Step 10
    printf("Step 10 start:\n\n");

    printf("Int storage in general array:\n");
    char* general_array = two_d_alloc(3, 3, 4);
    int a = 1000; int b = 2000; int c = 3000;
    void* column1 = &a;
    void* column2 = &b;
    void* column3 = &c;
    for(int i = 1; i < 4; i++)
    {
        two_d_store_general(column1, general_array, 4, i, 1, 3, 3);
        two_d_store_general(column2, general_array, 4, i, 2, 3, 3);
        two_d_store_general(column3, general_array, 4, i, 3, 3, 3);
    }
    for(int i = 1; i < 4; i++)
    {
        for(int j = 1; j < 4; j++)
        {
            printf("%p = ", two_d_fetch_general(general_array, 4, i, j, 3, 3));
            printf("%d\n", *(int*)two_d_fetch_general(general_array, 4, i, j,
3, 3));
        }
    }

    printf("\nDouble storage in general array:\n");
    char* general_array2 = two_d_alloc(3, 3, 8);
    double d = 1000.0; double e = 2000.0; double f = 3000.0;
    void* double1 = &d;
    void* double2 = &e;
    void* double3 = &f;
    for(int i = 1; i < 4; i++)
    {
        two_d_store_general(double1, general_array2, 8, i, 1, 3, 3);
        two_d_store_general(double2, general_array2, 8, i, 2, 3, 3);
        two_d_store_general(double3, general_array2, 8, i, 3, 3, 3);
    }
    for(int i = 1; i < 4; i++)
```

```
    {
        for(int j = 1; j < 4; j++)
        {
            printf("%p = ", two_d_fetch_general(general_array2, 8, i, j, 3,
3));
            printf("%lf\n", *(double*)two_d_fetch_general(general_array2, 8, i,
j, 3, 3));
        }
    }

    printf("\nStep 10 end.\n\n");
    free(row_array);
    free(boundary_array);
    free(column_array);
    free(ptr_array);
    free(general_array);
    free(general_array2);

    return 0;
}
```

```
patvih161@ltse01:~/datateknik$ ./a.out
Step 1-3 start:


Step 1-3 end.

Step 4-6 start:

Index [1, 1] = 1000
Index [1, 2] = 2000
Index [1, 3] = 3000
Index [2, 1] = 4000
Index [2, 2] = 5000
Index [2, 3] = 6000
Index [3, 1] = 7000
Index [3, 2] = 8000
Index [3, 3] = 9000

Step 4-6 end.
```

In step 1-3, the array is successfully allocated and deallocated.
In step 4-6, the integers from 1000 to 9000 are stored in row-major form and fetched from the indices shown in the output.

```
Step 7 start:

Empty array test:
Rows, columns and size have to be at least 1.
Out of bounds test:
Index exceeds array bounds.
Index exceeds array bounds.
Negative and zero index test:
Index exceeds array bounds.
Index exceeds array bounds.

Step 7 end.
```

In step 7, the program tries and fails to create an array with 0 rows. The program tries to store values out of bounds and also to fetch values from indices that does not exist (-1 and 0).

```
Step 8 start:

Stored column-major, fetched row-major:
Index [1, 1] = 1000
Index [1, 2] = 4000
Index [1, 3] = 7000
Index [2, 1] = 2000
Index [2, 2] = 5000
Index [2, 3] = 8000
Index [3, 1] = 3000
Index [3, 2] = 6000
Index [3, 3] = 9000

Stored row-major, fetched column-major:
Index [1, 1] = 1000
Index [1, 2] = 4000
Index [1, 3] = 7000
Index [2, 1] = 2000
Index [2, 2] = 5000
Index [2, 3] = 8000
Index [3, 1] = 3000
Index [3, 2] = 6000
Index [3, 3] = 9000

Stored column-major, fetched column-major:
Index [1, 1] = 1000
Index [1, 2] = 2000
Index [1, 3] = 3000
Index [2, 1] = 4000
Index [2, 2] = 5000
Index [2, 3] = 6000
Index [3, 1] = 7000
Index [3, 2] = 8000
Index [3, 3] = 9000

Step 8 end.
```

In step 8, the matrices are used to store and fetch values using different functions. This is done to illustrate that the column-major functions work as they should.

```
Step 9 start:

0x22024f0 = 1000
0x22024f4 = 1000
0x22024f8 = 1000
0x22024fc = 2000
0x2202500 = 2000
0x2202504 = 2000
0x2202508 = 3000
0x220250c = 3000
0x2202510 = 3000

Step 9 end.
```

In step 9, the array stores addresses instead of values. The addresses can be cast to integer pointers and dereferenced to reveal the values.

```
Step 10 start:

Int storage in general array:
0x2202520 = 1000
0x2202524 = 2000
0x2202528 = 3000
0x220252c = 1000
0x2202530 = 2000
0x2202534 = 3000
0x2202538 = 1000
0x220253c = 2000
0x2202540 = 3000

Double storage in general array:
0x2202550 = 1000.000000
0x2202558 = 2000.000000
0x2202560 = 3000.000000
0x2202568 = 1000.000000
0x2202570 = 2000.000000
0x2202578 = 3000.000000
0x2202580 = 1000.000000
0x2202588 = 2000.000000
0x2202590 = 3000.000000

Step 10 end.
```

In step 10, the array is able to store void pointers pointing to an arbitrary data type. As in step 9, the void pointer can be cast to the correct type and dereferenced for the value.