
TASK 2 - Understanding Design Patterns in C

---

defs.h

```
#ifndef _QUADTREE_INCLUDE_MATRIX_DEFS_H
#define _QUADTREE_INCLUDE_MATRIX_DEFS_H

//error codes
#define MATRIX_STATUS_OK 0
#define MATRIX_STATUS_ERROR 1
#define MATRIX_STATUS_INVALID_ARG (MATRIX_STATUS_ERROR +1)
#define MATRIX_STATUS_NOT_ENOUGH_MEMORY (MATRIX_STATUS_INVALID_ARG +1)

#endif // !_QUADTREE_INCLUDE_MATRIX_DEFS_H
```

types.h

```
#ifndef _QUADTREE_INCLUDE_MATRIX_TYPES_H
#define _QUADTREE_INCLUDE_MATRIX_TYPES_H

#include <stdint.h>

#ifdef __cplusplus
extern "C" {
#endif

    typedef struct matrix matrix_t;
    typedef int32_t status_t;


#ifdef __cplusplus
}
#endif

#endif //!_QUADTREE_INCLUDE_MATRIX_TYPES_H
```

api.h

```
#ifndef _QUADTREE_INCLUDE_MATRIX_API_H
#define _QUADTREE_INCLUDE_MATRIX_API_H

#include <matrix/types.h>
```

```
#ifdef __cpluscplus
extern "C" {
#endif //cplusplus

    matrix_t* matrix_new(int row, int col, status_t* error);
    status_t matrix_delete(matrix_t* matrix);
    int getRowCount(const matrix_t* matrix);
    int getColumnCount(const matrix_t* matrix);
    const uint8_t* getElement(const matrix_t* matrix);

#ifdef __cplusplus
}
#endif //cplusplus

#endif //!_QUADTREE_INCLUDE_MATRIX_API_H
```

matrix.c

```
#include <matrix/api.h>
#include <matrix/defs.h> //for error codes
#include <stdlib.h> //malloc and free

struct matrix
{
    int row;
    int col;
    uint8_t* data;
};

// function to create a new matrix, it returns an error code if the inputs arguments are invalid:
// if row and column are both less than 1 then it returns MATRIX_STATUS_INVALID_ARG;
// if the size of the variable is larger than available memory then it will return
// MATRIX_STATUS_NOT_ENOUGH_MEMORY;

matrix_t* matrix_new(int row, int col, status_t* error)
{
    status_t status = MATRIX_STATUS_OK;
    matrix_t* result = NULL;

    if (row < 1 || col < 1)
    {
        status = MATRIX_STATUS_INVALID_ARG;
    }
    else
    {
        result = (matrix_t*)malloc(sizeof(matrix_t));
        if (result != NULL)
        {
            uint8_t* elements = (uint8_t)malloc(static_cast<unsigned long long>(row) * col *
sizeof(uint8_t));
            if (elements != NULL)
            {
                result->data = elements;
                result->row = row;
                result->col = col;
            }
            else
            {
                free(result);
                result = NULL;
                status = MATRIX_STATUS_NOT_ENOUGH_MEMORY;
            }
        }
        else
        {
            status = MATRIX_STATUS_NOT_ENOUGH_MEMORY;
        }
    }

    if(NULL != error)
    {
        *error = status;
    }
    return result;
}

// delete function to delete a matrix, if the matrix is not NULL and the data is not NULL then it returns
// MATRIX_STATUS_INVALID_ARG;

status_t matrix_delete(matrix_t* matrix)
{
    if (matrix != NULL || matrix->data != NULL)
    {
        free(matrix->data);
        free(matrix);
        return MATRIX_STATUS_OK;
    }
    return MATRIX_STATUS_INVALID_ARG;
}

//counts the number of rows
int getRowCount(const matrix_t* matrix)
{
    if (matrix != NULL && matrix->data != NULL)
    {
        return matrix->row;
    }
    return 0;
}

// counts the number of columns
int getColumnCount(const matrix_t* matrix)
{
    if (matrix != NULL && matrix->data != NULL)
    {
        return matrix->col;
    }
    return 0;
}

//get the element of the matrix
const uint8_t* getElement(const matrix_t* matrix)
{
    if (matrix != NULL && matrix->data != NULL)
    {
        return matrix->data;
    }
    return 0;
}
```