**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ**

**федеральное ГОСУДАРСТВЕННОЕ бюджетное ОБРАЗОВАТЕЛЬНОЕ**

**УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ**

**ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В.Г.ШУХОВА»**

**(БГТУ им. В.Г. Шухова)**

**Кафедра программного обеспечения вычислительной техники и автоматизированных систем**

**Курсовая работа**

**Разработка и реализация библиотеки для работы с матрицами на языке программирования С**

Выполнил:

Студент группы: ВТ – 231

Немыкин С. С.

Принял:

Лукьянов А.М.

Белгород

2024

**СОДЕРЖАНИЕ**

|  |  |
| --- | --- |
| Введение..................................................................................................................... | 3 |
| Постановка задачи..................................................................................................... | 3 |
| Описание решения задачи......................................................................................... | 4 |
| Техническая информация.......................................................................................... | 4 |
| Основная часть........................................................................................................... | 5 |
| Вывод……………………………………………………………………………….. | 16 |

**Введение**

Современная программная инженерия требует от разработчиков создания эффективных и универсальных решений для обработки данных различных типов. Одним из таких типов являются матрицы – двумерные массивы чисел, широко используемые в математике, физике, машинном обучении и других областях науки и техники. Язык программирования Си предоставляет мощные средства для работы с памятью и низкоуровневыми структурами данных, что делает его идеальным выбором для реализации высокопроизводительных библиотек, работающих с матрицами.

Данная курсовая работа имеет практическое значение, так как разработанная матричная библиотека может стать основой для более сложных проектов, связанных с обработкой больших объемов числовых данных. Кроме того, она позволит углубить знания в области языка программирования Си и получить опыт в разработке низкоуровневых алгоритмов и структур данных.

**ПОСТАНОВКА ЗАДАЧИ**

Разработать матричную библиотеку на языке программирования Си, включающую следующие функции:

**Инициализация и освобождение памяти**:

my\_create\_matrix — создание новой матрицы заданного размера.

my\_remove\_matrix — освобождение памяти, занятой под матрицу.

**Основные операции над матрицами**:

my\_eq\_matrix — проверка равенства двух матриц.

my\_sum\_matrix — сложение двух матриц.

my\_sub\_matrix — вычитание двух матриц.

my\_mult\_number — умножение матрицы на число.

my\_mult\_matrix — умножение двух матриц.

my\_transpose — транспонирование матрицы.

**Дополнительные операции**:

my\_calc\_complements — вычисление матрицы алгебраических дополнений.

my\_determinant — вычисление определителя квадратной матрицы.

my\_inverse\_matrix — обращение квадратной матрицы (если возможно).

**Проверочные тесты**:

Написать набор тестов для проверки корректности всех реализованных функций.

**ОПИСАНИЕ РЕШЕНИЯ ЗАДАЧИ**

Для реализации матричной библиотеки на языке программирования Си были созданы три основных файла: my\_matrix.c, my\_matrix.h и файл с тестовыми сценариями, который находится в папке tests. В этих файлах содержится вся необходимая логика для работы с матрицами, а также тесты для проверки корректности выполнения операций.

**ТЕХНИЧЕСКАЯ ИНФОРМАЦИЯ**

Матричная библиотека предназначена для выполнения основных математических операций над матрицами. Она написана на языке программирования Си и включает в себя набор функций для создания, удаления, сложения, вычитания, умножения, транспонирования, вычисления определителя и обращения матриц. Библиотека разрабатывается с целью обеспечения высокой производительности, модульности и кросс-платформенности.

**ОСНОВНАЯ ЧАСТЬ**

**Импорт библиотек и создание заголовочного файла**

Проект начинается с подключения стандартных библиотек языка Си (<float.h>, <math.h>, <stdio.h>, <stdlib.h>). Затем определяются коды ошибок с помощью перечисления (OK, MATRIX\_ERROR, CALCULATION\_ERROR) и макросы для малых положительных чисел (EPSILON), успеха (SUCCESS) и неудачи (FAILURE).

#include <float.h>

#include <math.h>

#include <stdio.h>

#include <stdlib.h>

enum { OK, MATRIX\_ERROR, CALCULATION\_ERROR };

#define EPSILON 1e-7

#define SUCCESS 1

#define FAILURE 0

**Создание структуры матрицы**

Далее создается структура matrix\_t, представляющая матрицу, которая содержит указатель на двумерный массив типа double, количество строк и столбцов. Эта структура будет использоваться для хранения и манипулирования матрицами в программе.

typedef struct matrix\_struct {

double \*\*matrix;

int rows;

int columns;

} matrix\_t;

**Функции**

После определения структуры matrix\_t и необходимых макросов, я перехожу к объявлению функций для работы с матрицами.

// Создает новую матрицу размером rows x columns и записывает её в result

int my\_create\_matrix(int rows, int columns, matrix\_t \*result);

// Освобождает память, выделенную под матрицу A

void my\_remove\_matrix(matrix\_t \*A);

// Проверяет равенство двух матриц A и B

int my\_eq\_matrix(matrix\_t \*A, matrix\_t \*B);

// Складывает две матрицы A и B и записывает результат в result

int my\_sum\_matrix(matrix\_t \*A, matrix\_t \*B, matrix\_t \*result);

// Вычитает матрицу B из матрицы A и записывает результат в result

int my\_sub\_matrix(matrix\_t \*A, matrix\_t \*B, matrix\_t \*result);

// Умножает матрицу A на число number и записывает результат в result

int my\_mult\_number(matrix\_t \*A, double number, matrix\_t \*result);

// Перемножает две матрицы A и B и записывает результат в result

int my\_mult\_matrix(matrix\_t \*A, matrix\_t \*B, matrix\_t \*result);

// Транспонирует матрицу A и записывает результат в result

int my\_transpose(matrix\_t \*A, matrix\_t \*result);

// Вычисляет матрицу алгебраических дополнений для матрицы A и записывает результат в result

int my\_calc\_complements(matrix\_t \*A, matrix\_t \*result);

// Вычисляет определитель квадратной матрицы A и записывает его в result

int my\_determinant(matrix\_t \*A, double \*result);

// Находит обратную матрицу для квадратной матрицы A (если существует) и записывает результат в result

int my\_inverse\_matrix(matrix\_t \*A, matrix\_t \*result);

**Реализация Функций**

После написания заголовочного файла с обьявлением всех необходимых функций переходим к написанию логики каждой.

Функция my\_check\_matrix принимает указатель на структуру matrix\_t и проверяет, является ли переданный объект допустимой матрицей. Если указатель на матрицу или сам массив элементов матрицы равен NULL, либо количество строк или столбцов меньше единицы, то возвращается код ошибки MATRIX\_ERROR. В противном случае возвращается код OK, означающий, что матрица корректна.

int my\_check\_matrix(const matrix\_t \*A) {

return (A == NULL || A->matrix == NULL || A->rows < 1 || A->columns < 1)

? MATRIX\_ERROR

: OK;

}

Функция my\_create\_matrix создает новую матрицу размером rows x columns, выделяет необходимую память и инициализирует все элементы матрицы нулями. Если возникают ошибки при выделении памяти или переданные параметры неверны, функция возвращает код ошибки MATRIX\_ERROR, в противном случае — код успешной операции OK.

int my\_create\_matrix(int rows, int columns, matrix\_t \*A) {

if (!A) return MATRIX\_ERROR;

if (rows < 1 || columns < 1) {

A->rows = 0;

A->columns = 0;

A->matrix = NULL;

return MATRIX\_ERROR;

}

A->matrix = (double \*\*)malloc(rows \* sizeof(double \*));

if (!A->matrix) {

A->rows = 0;

A->columns = 0;

return MATRIX\_ERROR;

}

for (int i = 0; i < rows; i++) {

A->matrix[i] = (double \*)malloc(columns \* sizeof(double));

if (!A->matrix[i]) {

for (int j = 0; j < i; j++)

free(A->matrix[j]);

free(A->matrix);

A->rows = 0;

A->columns = 0;

return MATRIX\_ERROR;

}

for (int j = 0; j < columns; j++) {

A->matrix[i][j] = 0.0;

}

}

A->rows = rows;

A->columns = columns;

return OK;

}

Функция my\_remove\_matrix освобождает всю выделенную под матрицу память, проходя по каждой строке и освобождая её, а затем освобождая массив указателей на строки. После этого указатель на матрицу устанавливается в NULL, а размеры матрицы сбрасываются до нуля, делая структуру готовой для повторного использования.

void my\_remove\_matrix(matrix\_t \*A) {

if (my\_check\_matrix(A) == OK) {

for (int i = 0; i < A->rows; i++) {

free(A->matrix[i]);

}

free(A->matrix);

A->matrix = NULL;

A->rows = 0;

A->columns = 0;

}

}

Функция my\_eq\_matrix сравнивает две матрицы A и B на равенство, проверяя совпадение их размеров и элементов с точностью до 1e-6, и возвращает SUCCESS, если они равны, или FAILURE в противном случае.

int my\_eq\_matrix(matrix\_t \*A, matrix\_t \*B) {

if (A == NULL || B == NULL) return FAILURE;

if (A->rows != B->rows || A->columns != B->columns) return FAILURE;

for (int i = 0; i < A->rows; i++) {

for (int j = 0; j < A->columns; j++) {

if (fabs(A->matrix[i][j] - B->matrix[i][j]) > 1e-6) return FAILURE;

}

}

return SUCCESS;

}

Функция my\_sum\_matrix суммирует две матрицы A и B одинакового размера и записывает результат в матрицу result. Если размеры исходных матриц не совпадают, функция возвращает ошибку CALCULATION\_ERROR. Если возникает проблема с созданием результирующей матрицы, возвращается ошибка MATRIX\_ERROR. В противном случае элементы результирующей матрицы рассчитываются поэлементно как сумма соответствующих элементов исходных матриц, и функция возвращает OK.

int my\_sum\_matrix(matrix\_t \*A, matrix\_t \*B, matrix\_t \*result) {

if (A->rows != B->rows || A->columns != B->columns) {

return CALCULATION\_ERROR;

}

if (my\_create\_matrix(A->rows, A->columns, result) != OK) {

return MATRIX\_ERROR;

}

for (int i = 0; i < A->rows; i++) {

for (int j = 0; j < A->columns; j++) {

result->matrix[i][j] = A->matrix[i][j] + B->matrix[i][j];

}

}

return OK;

}

Функция my\_sub\_matrix вычитает матрицу B из матрицы A и записывает результат в матрицу result. Если размеры исходных матриц не совпадают, функция возвращает ошибку CALCULATION\_ERROR. Если возникает проблема с созданием результирующей матрицы, возвращается ошибка MATRIX\_ERROR. В противном случае элементы результирующей матрицы рассчитываются поэлементно как разность соответствующих элементов исходных матриц, и функция возвращает OK.

int my\_sub\_matrix(matrix\_t \*A, matrix\_t \*B, matrix\_t \*result) {

if (A->rows != B->rows || A->columns != B->columns) {

return CALCULATION\_ERROR;

}

if (my\_create\_matrix(A->rows, A->columns, result) != OK) {

return MATRIX\_ERROR;

}

for (int i = 0; i < A->rows; i++) {

for (int j = 0; j < A->columns; j++) {

result->matrix[i][j] = A->matrix[i][j] - B->matrix[i][j];

}

}

return OK;

}

Функция my\_mult\_matrix перемножает две матрицы A и B и записывает результат в матрицу result. Если количество столбцов первой матрицы не равно количеству строк второй, функция возвращает ошибку CALCULATION\_ERROR. Если возникает проблема с созданием результирующей матрицы, возвращается ошибка MATRIX\_ERROR. В противном случае элементы результирующей матрицы рассчитываются путем суммирования произведений соответствующих элементов строк первой матрицы и столбцов второй, и функция возвращает OK.

int my\_mult\_matrix(matrix\_t \*A, matrix\_t \*B, matrix\_t \*result) {

if (A->columns != B->rows) {

return CALCULATION\_ERROR;

}

if (my\_create\_matrix(A->rows, B->columns, result) != OK) {

return MATRIX\_ERROR;

}

for (int i = 0; i < A->rows; i++) {

for (int j = 0; j < B->columns; j++) {

result->matrix[i][j] = 0;

for (int k = 0; k < A->columns; k++) {

result->matrix[i][j] += A->matrix[i][k] \* B->matrix[k][j];

}

}

}

return OK;

}

Функция my\_mult\_number умножает каждый элемент матрицы A на число number и записывает результат в матрицу result. Если возникает проблема с созданием результирующей матрицы, функция возвращает ошибку MATRIX\_ERROR. В противном случае элементы результирующей матрицы рассчитываются как произведение соответствующего элемента исходной матрицы на указанное число, и функция возвращает OK.

int my\_mult\_number(matrix\_t \*A, double number, matrix\_t \*result) {

if (my\_create\_matrix(A->rows, A->columns, result) != OK) {

return MATRIX\_ERROR;

}

for (int i = 0; i < A->rows; i++) {

for (int j = 0; j < A->columns; j++) {

result->matrix[i][j] = A->matrix[i][j] \* number;

}

}

return OK;

}

Функция my\_transpose транспонирует матрицу A и записывает результат в матрицу result. Если одна из передаваемых матриц равна NULL, функция возвращает ошибку MATRIX\_ERROR. Если возникает проблема с созданием результирующей матрицы, функция возвращает ошибку MATRIX\_ERROR. В противном случае элементы результирующей матрицы заполняются так, что строки исходной матрицы становятся столбцами результирующей, и наоборот, и функция возвращает OK.

int my\_transpose(matrix\_t \*A, matrix\_t \*result) {

if (A == NULL || result == NULL) {

return MATRIX\_ERROR;

}

if (my\_create\_matrix(A->columns, A->rows, result) != OK) {

return MATRIX\_ERROR;

}

for (int i = 0; i < A->rows; i++) {

for (int j = 0; j < A->columns; j++) {

result->matrix[j][i] = A->matrix[i][j];

}

}

return OK;

}

Функция get\_minor\_matrix вычисляет минор элемента матрицы A на позиции (i, j). Минор элемента матрицы — это определитель подматрицы, полученной удалением строки и столбца, содержащих данный элемент. Если размер матрицы равен 1, то минором считается само значение единственного элемента матрицы. В остальных случаях функция создает уменьшенную матрицу B путём исключения строки и столбца, соответствующих индексам i и j, и рекурсивно вычисляет определитель этой меньшей матрицы. Результат сохраняется в переменной minor, после чего временная матрица B освобождается, и функция возвращает значение минора.

int get\_minor\_matrix(matrix\_t \*A, int i, int j) {

int n = A->rows;

double minor;

matrix\_t B;

if (n == 1) {

return A->matrix[0][0];

}

my\_create\_matrix(n - 1, n - 1, &B);

for (int row = 0, b\_row = 0; row < A->rows; row++) {

if (row == i) {

continue;

}

for (int col = 0, b\_col = 0; col < A->columns; col++) {

if (col == j) {

continue;

}

B.matrix[b\_row][b\_col++] = A->matrix[row][col];

}

b\_row++;

}

my\_determinant(&B, &minor);

my\_remove\_matrix(&B);

return minor;

}

Функция my\_calc\_complements вычисляет матрицу алгебраических дополнений для матрицы A и сохраняет результат в матрице result. Матрица алгебраических дополнений получается из миноров элементов исходной матрицы, взятых со знаком, зависящим от индексов элемента. Если матрица A не квадратная, функция возвращает ошибку CALCULATION\_ERROR. Для каждого элемента матрицы алгебраического дополнения рассчитывается минор соответствующего элемента исходной матрицы, и знак зависит от суммы индексов строки и столбца. Если сумма индексов чётная, знак остаётся положительным, если нечётная — отрицательным.

int my\_calc\_complements(matrix\_t \*A, matrix\_t \*result) {

if (A == NULL || result == NULL) {

return MATRIX\_ERROR;

}

if (!result) {

return MATRIX\_ERROR;

}

if (my\_eq\_matrix(A, result)) {

return MATRIX\_ERROR;

}

if (A->rows != A->columns) {

return CALCULATION\_ERROR;

}

int n = A->rows;

my\_create\_matrix(n, n, result);

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

double minor = get\_minor\_matrix(A, i, j);

double sign = ((i + j) % 2 == 0) ? 1.0 : -1.0;

result->matrix[i][j] = sign \* minor;

}

}

return OK;

}

Функция calculate\_determinant\_2x2 вычисляет определитель для матрицы размером 2x2. Она проверяет размеры матрицы и возвращает ошибку MATRIX\_ERROR, если матрица не является квадратной матрицей размером 2x2. В этом случае она вычисляет определитель как разность между произведениями диагональных элементов матрицы.

static int calculate\_determinant\_2x2(matrix\_t \*A, double \*result) {

if (A->rows != 2 || A->columns != 2) {

return MATRIX\_ERROR;

}

\*result = A->matrix[0][0] \* A->matrix[1][1] - A->matrix[0][1] \* A->matrix[1][0];

return OK;

}

Основная функция my\_determinant вычисляет определитель для любой квадратной матрицы. Если матрица не является квадратной, она возвращает ошибку CALCULATION\_ERROR. Если матрица имеет размер 1x1, то определителем является сама ячейка матрицы. Если матрица имеет размер 2x2, вызывается функция calculate\_determinant\_2x2. Для матриц большего размера функция рекурсивно вычисляет определители для подматриц, получаемых исключением одной строки и одного столбца, и суммирует результаты с учётом знаков.

**int my\_determinant(matrix\_t \*A, double \*result) {**

if (A->rows != A->columns) {

return CALCULATION\_ERROR;

}

\*result = 0.0;

if (A->rows == 1) {

\*result = A->matrix[0][0];

} else if (A->rows == 2) {

return calculate\_determinant\_2x2(A, result);

} else {

for (int i = 0; i < A->rows; i++) {

double sign = (i % 2 == 0) ? 1.0 : -1.0;

double minor = get\_minor\_matrix(A, 0, i);

\*result += sign \* A->matrix[0][i] \* minor;

}

}

return OK;

}

Функция is\_matrix\_all\_zero проверяет, являются ли все элементы матрицы равными нулю. Она проходит по всем элементам матрицы и проверяет, встречаются ли ненулевые элементы. Если такой элемент найден, функция возвращает 0, что означает, что матрица не является полностью нулевой. Если функция доходит до конца, не обнаруживая ненулевых элементов, она возвращает 1, что свидетельствует о том, что матрица действительно состоит исключительно из нулей.

int is\_matrix\_all\_zero(matrix\_t \*A) {

for (int i = 0; i < A->rows; i++) {

for (int j = 0; j < A->columns; j++) {

if (A->matrix[i][j] != 0) {

return 0;

}

}

}

return 1;

}

Функция my\_inverse\_matrix вычисляет обратную матрицу для матрицы A и сохраняет результат в матрице result. Если матрица A пустая или неквадратная, функция возвращает ошибку CALCULATION\_ERROR. Если определитель матрицы A равен нулю, функция также возвращает ошибку CALCULATION\_ERROR, так как невозможно обратить матрицу с нулевым определителем. Если матрица содержит только нули, функция возвращает ошибку MATRIX\_ERROR, так как такая матрица не подлежит обращению.

int my\_inverse\_matrix(matrix\_t \*A, matrix\_t \*result) {

if (!A) {

return MATRIX\_ERROR;

}

if (!result) {

return CALCULATION\_ERROR;

}

if (A->rows != A->columns) {

return CALCULATION\_ERROR;

}

if (is\_matrix\_all\_zero(A)) {

return MATRIX\_ERROR;

}

double det = 0.0;

int det\_status = my\_determinant(A, &det);

if (det\_status != OK || det == 0) {

return CALCULATION\_ERROR;

}

my\_create\_matrix(A->rows, A->columns, result);

if (A->rows == 1) {

result->matrix[0][0] = 1.0 / A->matrix[0][0];

} else {

matrix\_t compliments;

my\_calc\_complements(A, &compliments);

matrix\_t transposed\_compliments;

my\_transpose(&compliments, &transposed\_compliments);

for (int i = 0; i < transposed\_compliments.rows; i++) {

for (int j = 0; j < transposed\_compliments.columns; j++) {

result->matrix[i][j] = transposed\_compliments.matrix[i][j] / det;

}

}

my\_remove\_matrix(&transposed\_compliments);

my\_remove\_matrix(&compliments);

}

return OK;

}

**Тестирование**

Тестирование — это ключевой этап разработки программного обеспечения, направленный на проверку его работоспособности и соответствия заявленным требованиям. Этот процесс включает выявление ошибок, багов и уязвимостей, которые могут повлиять на качество продукта. Благодаря тестированию удается повысить надежность системы, улучшить пользовательский опыт и минимизировать риски, связанные с эксплуатацией программы. Раннее обнаружение и устранение дефектов значительно сокращает затраты на доработку и поддержку готового продукта.

Кроме того, тестирование помогает оценить производительность системы, ее устойчивость к высоким нагрузкам и способность работать в различных условиях. Оно также играет важную роль в обеспечении безопасности данных, предотвращая потенциальные угрозы и утечки информации. Таким образом, тестирование является неотъемлемым элементом успешного создания и внедрения качественного программного обеспечения.

**Основные этапы тестирования:**

Данный фрагмент кода представляет собой юнит-тест, созданный с использованием фреймворка Check. Он предназначен для проверки корректной работы функции s21\_create\_matrix, отвечающей за создание матрицы определенного размера. В начале теста объявляется переменная типа matrix\_t, которая затем передается в качестве аргумента функции s21\_create\_matrix. Эта функция должна создать матрицу размером 5x5 и вернуть статус выполнения операции (OK).

После вызова функции выполняется проверка результата с помощью макроса ck\_assert\_int\_eq, который сравнивает возвращенное значение с ожидаемой константой OK. Если сравнение успешно, тест считается пройденным. В завершение теста вызывается функция s21\_remove\_matrix, освобождающая память, занятую матрицей. Такой подход позволяет автоматизировать тестирование отдельных компонентов программы и своевременно выявлять возможные ошибки в их работе.

START\_TEST(s21\_create\_matrix\_test\_1) {

matrix\_t A = {0};

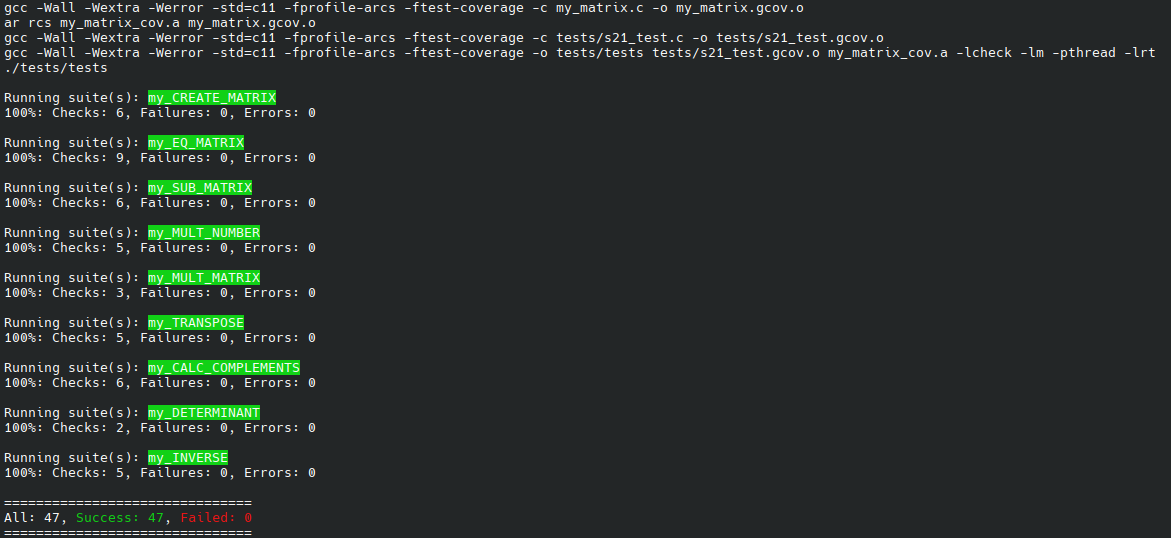
ck\_assert\_int\_eq(s21\_create\_matrix(5, 5, &A), OK);

s21\_remove\_matrix(&A);

}

END\_TEST

Применение check.h в процессе разработки способствует повышению качества программного продукта, позволяя выявлять ошибки на ранних этапах жизненного цикла проекта. Кроме того, наличие автоматизированного набора тестов значительно упрощает рефакторинг и внесение изменений в код, поскольку разработчики могут быстро убедиться, что внесенные изменения не нарушают существующую функциональность.



Проведя тестирование своего проекта, я убедился в правильности выполненной работы, так как все тесты, написанные мной были выполненны идеально.

**Вывод**

Разработка библиотеки матричных функций на языке Си является значимым шагом в изучении основ программирования и создании надёжных программных решений. В рамках данной курсовой работы была создана библиотека, включающая ключевые операции над матрицами, такие как вычисление определителя, транспонирование, сложение, умножение и инвертирование матриц. При реализации каждой функции учитывались требования к производительности и безопасности, что обеспечило создание качественного продукта.

Особенное внимание было уделено этапу тестирования, который сыграл важную роль в достижении высокого уровня надёжности и устойчивости библиотеки. Для автоматизации процесса тестирования использовался инструмент Check, позволяющий создавать широкий спектр тестов, покрывающих разнообразные варианты использования функций. Это позволило убедиться в корректной работе всех функций даже в сложных и нетипичных условиях.

Проведённые тесты продемонстрировали высокую степень надёжности разработанной библиотеки, которая теперь готова к использованию в реальных проектах. Таким образом, цель курсовой работы — разработка эффективной и надёжной библиотеки матричных функций — была успешно достигнута.

Приобретённый опыт и навыки, полученные в ходе выполнения данного проекта, окажутся полезными в дальнейшей учебной и профессиональной деятельности. Практика работы с инструментами тестирования, такими как Check, формирует правильный подход к созданию программного обеспечения, ориентированного на высокое качество и надёжность.

Таким образом, курсовая работа достигла своей цели. Она позволила разработать качественную и надёжную библиотеку матричных функций, которую можно использовать в реальных проектах.

Кроме того, выполнение этой курсовой работы способствовало расширению моих знаний и углублению понимания принципов программирования. Я узнал много нового о методах разработки надёжного и эффективного программного обеспечения, а также получил ценный опыт работы с инструментами тестирования.