

# 6- Практическое задание – Массивы, Структуры, Указатели, Перегрузка операторов

## Table of Contents

Цель .....	2
1. Краткое объяснение массивов .....	2
2. Реализация класса Matrix для базовых операций .....	5
3. Работа со структурами и массивами .....	8
4. Введение в указатели в классах .....	10
5. Перегрузка оператора для пользовательского класса .....	13
<b>Что такое static в контексте?</b> .....	15
Самостоятельная задача: реализовать класс полинома .....	16

# Цель

Цель этой практической сессии - обеспечить студентам всестороннее понимание основных концепций объектно-ориентированного программирования на C++, с акцентом на массивы, структуры, указатели и перегрузку операторов. Через серию задач студенты будут участвовать в практических упражнениях, охватывающих широкий спектр тем.

## 1. Краткое объяснение массивов

В C++ массив — это коллекция элементов, хранящихся в смежных ячейках памяти. Это делает его эффективным для хранения данных одного типа. Когда мы используем массивы внутри класса, мы инкапсулируем данные, то есть упаковываем их вместе с методами, которые работают с этими данными. Наша цель - создать класс, который может:

- Хранить ежедневные показания температуры в массиве.
- Добавлять показание температуры.
- Вычислять среднюю температуру.
- Находить максимальное и минимальное показания температуры.

```
#include <iostream>
#include <limits> // Для numeric_limits

class TemperatureLogger {
private:
    static const int MAX_READINGS = 365; // Предполагаем максимум 365 показаний
    int temperatures[MAX_READINGS]; // Массив для хранения показаний температуры
    int numReadings; // Текущее количество показаний

public:
    // Конструктор
    TemperatureLogger() : numReadings(0) {
        for (int i = 0; i < MAX_READINGS; ++i) {
            temperatures[i] = 0;
        }
    }

    // Метод для добавления показания температуры
    void addReading(int temp) {
        if (numReadings < MAX_READINGS) {
            temperatures[numReadings++] = temp;
        } else {
            std::cout << "Достигнуто максимальное количество показаний, добавление невозможно." << std::endl;
        }
    }

    // Метод для расчета средней температуры
    double calculateAverage() const {
        if (numReadings == 0) return 0; // Предотвращаем деление на ноль
        int sum = 0;
        for (int i = 0; i < numReadings; ++i) {
            sum += temperatures[i];
        }
        return static_cast<double>(sum) / numReadings;
    }

    // Метод для поиска максимального показания температуры
    int findMaxTemperature() const {
        int maxTemp = std::numeric_limits<int>::min(); // Инициализируем наименьшим возможным значением
        for (int i = 0; i < numReadings; ++i) {
            if (temperatures[i] > maxTemp) {
                maxTemp = temperatures[i];
            }
        }
    }
}
```

```

    }
    return maxTemp;
}

// Метод для поиска минимального показания температуры
int findMinTemperature() const {
    int minTemp = std::numeric_limits<int>::max(); // Инициализируем наибольшим возможным значением
    for (int i = 0; i < numReadings; ++i) {
        if (temperatures[i] < minTemp) {
            minTemp = temperatures[i];
        }
    }
    return minTemp;
}
};

int main() {
    TemperatureLogger logger;
    // Пример использования
    logger.addReading(23);
    logger.addReading(25);
    logger.addReading(22);
    logger.addReading(26);
    logger.addReading(24);

    std::cout << "Средняя температура: " << logger.calculateAverage() << std::endl;
    std::cout << "Максимальная температура: " << logger.findMaxTemperature() << std::endl;
    std::cout << "Минимальная температура: " << logger.findMinTemperature() << std::endl;

    return 0;
}

```

## Объяснение каждого шага

### Определение класса и частные члены:

- Константа MAX\_READINGS определяет максимальное количество показаний температуры, которое мы можем хранить.
- Массив temperatures содержит показания температуры.
- numReadings отслеживает текущее количество добавленных показаний температуры.

### Инициализация конструктора:

- Инициализирует numReadings значением 0 и устанавливает все значения в массиве temperatures равными 0.

### Метод addReading:

- Добавляет новое показание температуры в массив, если есть место, и увеличивает numReadings. Если массив заполнен, выводит сообщение о том, что добавить больше показаний нельзя.

### Метод calculateAverage:

- Вычисляет среднюю температуру из сохраненных показаний. Возвращает 0, если показаний не было добавлено, чтобы предотвратить деление на ноль.

### Методы findMaxTemperature и findMinTemperature:

- Эти методы перебирают массив temperatures, чтобы найти максимальное и минимальное показания температуры соответственно. Для инициализации сравнительных значений используются std::numeric\_limits<int>::min() и std::numeric\_limits<int>::max().

### Главная функция:

- Демонстрирует использование класса TemperatureLogger путем добавления некоторых показаний, а затем вывода средней, максимальной и минимальной температур.

Эта задача вводит концепцию массивов в классе и демонстрирует инкапсуляцию, ограничивая доступ к массиву напрямую и вместо этого предоставляя методы для взаимодействия с данными. Через этот пример студенты учатся управлять данными в классе и выполнять операции с этими данными контролируемым образом.



Строка `TemperatureLogger() : numReadings(0)` в конструкторе класса `TemperatureLogger` является примером списка инициализации в C++. В C++ список инициализации — это способ непосредственной инициализации переменных членов класса в момент создания объекта класса. Давайте разберем, что это означает и почему это используется:

#### Синтаксис и Назначение

1. **`TemperatureLogger()`** — это конструктор класса `TemperatureLogger`.
2. **После имени конструктора двоеточие:** вводит список инициализации.
3. **`numReadings(0)`** инициализирует член класса `numReadings` значением 0. Это означает, что когда объект `TemperatureLogger` создается, `numReadings` сразу устанавливается в 0 до выполнения тела конструктора (если оно есть).

#### Преимущества Использования Списков Инициализации

- A. **Эффективность:** для некоторых типов инициализация членов через список инициализации может быть более эффективной, чем присваивание значений внутри тела конструктора. Это особенно верно для объектов классов, имеющих конструкторы с параметрами. Это позволяет избежать создания временного объекта и его последующего копирования в целевой объект.
- B. **Необходимость для `const` или Ссылочных Членов:** если класс имеет `const` члены или ссылки, эти члены должны быть инициализированы с использованием списка инициализации, поскольку им нельзя присвоить значения в теле конструктора.
- C. **Порядок Инициализации:** Члены инициализируются в порядке их объявления в классе, а не в порядке, указанном в списке инициализации. Список инициализации учитывает этот порядок, обеспечивая предсказуемый способ инициализации членов.

#### Объяснение Примера

В контексте класса `TemperatureLogger` использование `numReadings(0)` в списке инициализации гарантирует, что член `numReadings` устанавливается в 0 как только объект `TemperatureLogger` создается. Это критически важно для логики класса, поскольку он полагается на `numReadings` для отслеживания количества добавленных показаний температуры, начиная с нуля.

Использование списков инициализации является общепринятой и рекомендуемой практикой в программировании на C++ для инициализации членов класса. Оно обеспечивает четкую, эффективную и безопасную инициализацию для всех типов переменных членов.

## 2. Реализация класса Matrix для базовых операций

Многомерные массивы в C++ — это массивы, содержащие более одного индекса или измерения. Они особенно полезны для представления таблиц, матриц или любых данных, организованных в строки и столбцы. Когда мы инкапсулируем многомерный массив в класс, мы можем определить методы для выполнения операций над данными, таких как сложение или вычитание матриц, на чем и будет сосредоточено внимание в этой задаче.

### Цель:

Цель состоит в том, чтобы создать класс Matrix, который инкапсулирует 2D массив и предоставляет методы для базовых операций с матрицей: сложение, вычитание и отображение матрицы.

```
#include <iostream>
using namespace std;

class Matrix {
private:
    static const int ROWS = 3; // Количество строк
    static const int COLS = 3; // Количество столбцов
    int data[ROWS][COLS]; // 2D массив для данных матрицы

public:
    // Конструктор для инициализации матрицы нулями
    Matrix() {
        for (int i = 0; i < ROWS; ++i) {
            for (int j = 0; j < COLS; ++j) {
                data[i][j] = 0;
            }
        }
    }

    // Функция для ввода данных матрицы пользователем
    void inputMatrix() {
        cout << "Введите данные матрицы (" << ROWS << "x" << COLS << "):\n";
        for (int i = 0; i < ROWS; ++i) {
            for (int j = 0; j < COLS; ++j) {
                cin >> data[i][j];
            }
        }
    }

    // Функция для сложения двух матриц
    Matrix add(const Matrix& m) const {
        Matrix result;
        for (int i = 0; i < ROWS; ++i) {
            for (int j = 0; j < COLS; ++j) {
                result.data[i][j] = data[i][j] + m.data[i][j];
            }
        }
        return result;
    }

    // Функция для вычитания двух матриц
    Matrix subtract(const Matrix& m) const {
        Matrix result;
        for (int i = 0; i < ROWS; ++i) {
            for (int j = 0; j < COLS; ++j) {
                result.data[i][j] = data[i][j] - m.data[i][j];
            }
        }
        return result;
    }
}
```

```

// Функция для отображения матрицы
void display() const {
    for (int i = 0; i < ROWS; ++i) {
        for (int j = 0; j < COLS; ++j) {
            cout << data[i][j] << " ";
        }
        cout << "\n";
    }
}

};

int main() {
    Matrix m1, m2, result;

    cout << "Ввод данных для Матрицы 1:\n";
    m1.inputMatrix();

    cout << "Ввод данных для Матрицы 2:\n";
    m2.inputMatrix();

    cout << "Матрица 1:\n";
    m1.display();

    cout << "Матрица 2:\n";
    m2.display();

    result = m1.add(m2);
    cout << "Результат сложения:\n";
    result.display();

    result = m1.subtract(m2);
    cout << "Результат вычитания:\n";
    result.display();

    return 0;
}

```

## Объяснение каждого шага

### Определение класса и приватные члены:

- Константы ROWS и COLS определяют размер матрицы. В этом примере используется матрица 3x3.
- data — это 2D массив, который хранит целые числа матрицы.

### Конструктор:

- Инициализирует все элементы массива data значением 0. Это делается с помощью вложенных циклов для итерации по каждому элементу массива.

### Метод inputMatrix:

- Позволяет пользователю вводить значения для матрицы из консоли. Использует вложенные циклы для получения ввода для каждого элемента в массиве data.

### Метод add:

- Принимает другой объект Matrix в качестве параметра и возвращает новый объект Matrix, который является результатом сложения текущей матрицы с параметром матрицы. Сложение выполняется поэлементно.

### Метод subtract:

- Похож на метод add, но выполняет вычитание. Возвращает новый объект Matrix, который является результатом вычитания матрицы-параметра из текущей матрицы.

**Метод display:**

- Выводит матрицу в консоль. Этот метод итерирует через массив data и выводит каждый элемент, форматируя вывод так, чтобы он выглядел как матрица.

**Главная функция:**

- Демонстрирует использование класса Matrix, создавая две матрицы, позволяя пользователю ввести их данные, а затем отображая результаты их сложения и вычитания.

Эта задача вводит концепцию многомерных массивов в классе и демонстрирует, как инкапсулировать данные и операции над этими данными в классе. Через этот пример студенты учатся выполнять базовые операции с матрицами структурированным и объектно-ориентированным способом.

### 3. Работа со структурами и массивами

Структуры в C++ предоставляют способ группировать переменные различных типов под одним именем, что упрощает управление и представление сложных данных. Сочетание структур с массивами позволяет создавать коллекции структурированных данных, что особенно полезно для управления списками записей с одинаковыми атрибутами, но разными значениями, например, список студентов.

#### Цель

Цель состоит в том, чтобы определить структуру Student, содержащую атрибуты, такие как имя, идентификатор и оценки. Мы создадим массив структур Student и напишем функции для заполнения, отображения и расчета средней оценки для каждого студента.

```
#include <iostream>
#include <string>
using namespace std;

// Определение структуры Student
struct Student {
    string name;
    int id;
    float grades[5]; // Массив для хранения 5 оценок студента
};

// Функция для заполнения записи студента
void populateStudent(Student& s) {
    cout << "Введите имя студента: ";
    cin >> s.name;
    cout << "Введите ID студента: ";
    cin >> s.id;
    for (int i = 0; i < 5; ++i) {
        cout << "Введите оценку " << (i + 1) << ": ";
        cin >> s.grades[i];
    }
}

// Функция для отображения записи студента
void displayStudent(const Student& s) {
    cout << "Имя: " << s.name << ", ID: " << s.id << ", Оценки: ";
    for (int i = 0; i < 5; ++i) {
        cout << s.grades[i] << " ";
    }
    cout << endl;
}

// Функция для расчета и отображения средней оценки студента
void displayAverageGrade(const Student& s) {
    float sum = 0;
    for (int i = 0; i < 5; ++i) {
        sum += s.grades[i];
    }
    cout << "Средняя оценка для " << s.name << ": " << (sum / 5) << endl;
}

int main() {
    const int NUM_STUDENTS = 3; // Количество студентов
    Student students[NUM_STUDENTS]; // Массив структур Student

    // Заполнение и отображение данных каждого студента
    for (int i = 0; i < NUM_STUDENTS; ++i) {
        cout << "Студент " << (i + 1) << endl;
        populateStudent(students[i]);
        displayStudent(students[i]);
        displayAverageGrade(students[i]);
        cout << endl;
    }
}
```



```
    return 0;
}
```

## Объяснение каждого шага

### Определение структуры Student:

- Структура Student определена с name (строка), id (целое число) и grades (массив из 5 чисел с плавающей точкой) для хранения оценок студента.

### Функция populateStudent:

- Принимает ссылку на структуру Student в качестве параметра.
- Предлагает пользователю ввести имя, ID и 5 оценок для студента, которые сохраняются непосредственно в переданной структуре Student.

### Функция displayStudent:

- Принимает структуру Student в качестве параметра (по константной ссылке, чтобы предотвратить модификацию).
- Выводит в консоль имя студента, ID и все оценки.

### Функция displayAverageGrade:

- Также принимает структуру Student по константной ссылке.
- Рассчитывает среднее из 5 оценок, суммируя их и деля на 5, затем выводит среднюю оценку.

### Главная функция:

- Определяет NUM\_STUDENTS как количество студентов (3 в этом примере) и создает массив структур Student для хранения их данных.
- Проходит через массив students, используя функции populateStudent, displayStudent и displayAverageGrade для обработки данных каждого студента.
- Демонстрирует полный процесс заполнения, отображения и расчета средних оценок для небольшой группы студентов.

Эта задача демонстрирует, как структуры и массивы могут быть использованы вместе для структурированного управления сложными данными. Создавая структуру для записей студентов и массив для хранения нескольких записей, мы эффективно управляем и манипулируем данными, связанными с группой студентов. С помощью функций, которые работают с этими структурами, мы инкапсулируем логику для заполнения, отображения и анализа данных студентов, придерживаясь принципов структурированного и модульного программирования.

## 4. Введение в указатели в классах

Указатели — это переменные, которые хранят адрес памяти другой переменной. Они являются мощной функцией в C++, позволяющей управлять динамической памятью, передавать ссылки на функции и более эффективно манипулировать массивами и объектами. Когда указатели используются в классах, они могут управлять динамическими данными, обмениваться ресурсами и облегчать полиморфизм.

Указатели достигают своих возможностей и работают за кулисами за счет прямого доступа к адресам памяти. Когда указатель используется для динамического выделения памяти (например, с использованием оператора `new` в C++), он получает адрес начала блока памяти, выделенного в куче, что позволяет программе эффективно управлять этой памятью во время выполнения. Это дает программам гибкость в управлении размером и структурой данных, позволяя изменять размер массивов или объектов "на лету" и обращаться к данным через их памятные адреса, что невозможно с использованием статического выделения памяти.

Цель

Модифицировать класс `TemperatureLogger` из Задачи 1 для использования указателя на массив, который хранит показания температуры. Это включает динамическое выделение памяти для массива и обеспечение правильного управления памятью для предотвращения утечек.

```
#include <iostream>
using namespace std;

class TemperatureLogger {
private:
    int *temperatures; // Указатель на массив температур
    int capacity; // Максимальное количество показаний
    int numReadings; // Текущее количество показаний

public:
    // Конструктор
    TemperatureLogger(int cap) : capacity(cap), numReadings(0) {
        temperatures = new int[capacity]; // Динамическое выделение памяти
        for (int i = 0; i < capacity; ++i) {
            temperatures[i] = 0;
        }
    }

    // Деструктор
    ~TemperatureLogger() {
        delete[] temperatures; // Освобождение выделенной памяти
    }

    // Метод для добавления показания температуры
    void addReading(int temp) {
        if (numReadings < capacity) {
            temperatures[numReadings++] = temp;
        } else {
            cout << "Достигнута максимальная емкость, добавить больше показаний невозможно." << endl;
        }
    }

    // Метод для расчета средней температуры
    double calculateAverage() const {
        if (numReadings == 0) return 0; // Предотвращение деления на ноль
        int sum = 0;
        for (int i = 0; i < numReadings; ++i) {
            sum += temperatures[i];
        }
        return static_cast<double>(sum) / numReadings;
    }

    // Другие методы (findMaxTemperature, findMinTemperature) остаются аналогичными и опущены для краткости
};
```

```
int main() {
    TemperatureLogger logger(365); // Создание объекта logger с емкостью для 365 показаний

    // Пример использования
    logger.addReading(23);
    logger.addReading(25);
    // Добавление дополнительных показаний по мере необходимости...

    cout << "Средняя температура: " << logger.calculateAverage() << endl;

    return 0;
}
```

## Объяснение каждого шага

### Указатель и динамическое выделение памяти:

- `int *temperatures;` объявляет указатель на целое число, который будет использоваться для указания на массив температур.
- В конструкторе `temperatures = new int[capacity];` динамически выделяет память для массива целых чисел заданного размера `capacity`, который представляет максимальное количество показаний температуры, которое может хранить логгер.

### Конструктор и инициализация:

- `TemperatureLogger(int cap) : capacity(cap), numReadings(0)` инициализирует `capacity` значением `cap` и `numReadings` значением 0 с использованием списка инициализации.
- Конструктор также инициализирует динамически выделенный массив `temperatures` нулями с помощью цикла.

### Деструктор для управления памятью:

- `~TemperatureLogger()` — это деструктор, который отвечает за освобождение динамически выделенной памяти, когда экземпляр `TemperatureLogger` уничтожается. `delete[] temperatures;` гарантирует, что память, выделенная для массива `temperatures`, освобождается, предотвращая утечки памяти.

### Метод `addReading`:

- Добавляет новое показание температуры в массив, если есть место, проверяя `numReadings < capacity`. После добавления температуры в массив увеличивает `numReadings`.

### Метод `calculateAverage`:

- Рассчитывает и возвращает среднюю температуру всех хранящихся показаний. Суммирует температуры и делит на `numReadings`, обрабатывая случай, когда `numReadings` равно 0, чтобы избежать деления на ноль.

### Демонстрация в главной функции:

- Демонстрирует создание объекта `TemperatureLogger` с определенной емкостью (365 показаний в данном случае).
- Показывает добавление показаний и расчет средней температуры.

Эта задача вводит указатели в классах, сосредоточив внимание на динамическом выделении и управлении памятью. Используя указатели для массива показаний температуры, мы обеспечиваем динамический размер и эффективное использование памяти, демонстрируя фундаментальный аспект управления ресурсами в C++. Правильное использование конструкторов и деструкторов гарантирует, что ресурсы корректно инициализируются и очищаются, предотвращая утечки памяти и способствуя созданию надежного программного обеспечения.



Указатель в Задаче 4 используется для динамического выделения памяти, что решает несколько проблем:

1. **Масштабируемость:** Используя указатель и динамически выделяя память, класс `TemperatureLogger` может обрабатывать любое количество показаний температуры вплоть до указанной емкости во

время выполнения. Это более гибко по сравнению с Задачей 1, где размер массива должен быть определен во время компиляции, ограничивая количество показаний фиксированным количеством.

2. **Эффективное использование памяти:** Динамическое выделение памяти позволяет программе использовать память более эффективно. Если мы придерживаемся статического массива, как в Задаче 1, мы выделяем память для максимального количества показаний, даже если нам нужно хранить только несколько показаний, потенциально растрачивая ресурсы памяти.

3. **Адаптивность:** с помощью указателей и динамической памяти класс TemperatureLogger можно легче адаптировать к различным требованиям, таким как корректировка емкости на основе ввода пользователя или условий во время выполнения. Без указателей, как в Задаче 1, емкость класса фиксирована, что снижает его адаптивность к различным сценариям использования.

В итоге, указатель в Задаче 4 решает проблемы масштабируемости, эффективного использования памяти и адаптивности, возникающие при использовании массивов фиксированного размера, определенных во время компиляции, как видно в Задаче 1.

## 5. Перегрузка оператора для пользовательского класса

Перегрузка операторов позволяет пользовательским типам (классам) в C++ переопределять поведение операторов (например, +, -, \*, /) при их использовании с экземплярами этих классов. Эта возможность позволяет объектам пользовательских классов использоваться более интуитивно, подобно встроенным типам, повышая читаемость и удобство поддержки кода.

Цель

Демонстрация перегрузки оператора на примере улучшения класса Matrix для поддержки сложения двух матриц с использованием оператора +. Это включает в себя определение функции-члена в классе Matrix, реализующей логику сложения элементов двух матриц.

```
#include <iostream>
using namespace std;

class Matrix {
private:
    static const int ROWS = 2; // Упрощено для демонстрации
    static const int COLS = 2; // Упрощено для демонстрации
    int data[ROWS][COLS]; // 2D массив для хранения данных матрицы

public:
    // Конструктор для инициализации элементов матрицы нулями
    Matrix() {
        for (int i = 0; i < ROWS; ++i) {
            for (int j = 0; j < COLS; ++j) {
                data[i][j] = 0;
            }
        }
    }

    // Функция для ввода данных матрицы
    void inputMatrix() {
        cout << "Введите данные матрицы (" << ROWS << "x" << COLS << "):\n";
        for (int i = 0; i < ROWS; ++i) {
            for (int j = 0; j < COLS; ++j) {
                cin >> data[i][j];
            }
        }
    }

    // Перегрузка оператора + для сложения двух матриц
    Matrix operator+(const Matrix& other) const {
        Matrix result;
        for (int i = 0; i < ROWS; ++i) {
            for (int j = 0; j < COLS; ++j) {
                result.data[i][j] = this->data[i][j] + other.data[i][j];
            }
        }
        return result;
    }

    // Функция для отображения матрицы
    void display() const {
        for (int i = 0; i < ROWS; ++i) {
            for (int j = 0; j < COLS; ++j) {
                cout << data[i][j] << " ";
            }
            cout << endl;
        }
    }
};
```

```
int main() {
    Matrix m1, m2, result;

    cout << "Ввод данных для Матрицы 1:\n";
    m1.inputMatrix();

    cout << "Ввод данных для Матрицы 2:\n";
    m2.inputMatrix();

    result = m1 + m2; // Использование перегруженного оператора +

    cout << "Результат сложения:\n";
    result.display();

    return 0;
}
```

## Объяснение каждого шага

### Определение класса Matrix:

- Класс Matrix инкапсулирует 2D массив data, который хранит элементы матрицы, с фиксированными размерами ROWS и COLS для упрощения.

### Конструктор:

- Инициализирует все элементы матрицы 0, чтобы обеспечить чистое состояние перед вводом любых данных.

### Метод inputMatrix:

- Позволяет пользователю вводить данные матрицы. Итерирует через элементы матрицы, запрашивая у пользователя ввод значений для каждого.

### Перегрузка оператора (+):

- Оператор + перегружен для сложения двух матриц. Функция Matrix operator+(const Matrix& other) const принимает другую матрицу в качестве параметра и возвращает новый объект Matrix, представляющий сумму двух матриц. Сложение выполняется поэлементно.

### Метод display:

- Выводит матрицу в консоль, итерируя через каждый элемент и отображая его в форматированном виде, представляющем структуру матрицы.

### Рабочий процесс в главной функции:

- Демонстрирует создание двух объектов Matrix и ввод их данных.
- Использует перегруженный оператор + для сложения матриц, сохраняя результат в новом объекте Matrix.
- Отображает результирующую матрицу, чтобы продемонстрировать эффект сложения матриц.

Эта задача иллюстрирует, как можно использовать перегрузку операторов для расширения функциональности пользовательских классов в C++, позволяя им вести себя более похоже на встроенные типы. Перегружая оператор + для класса Matrix, мы обеспечиваем интуитивное и естественное использование сложения матриц, делая код более читаемым и легким для поддержки.



### Что такое static в контексте?

В C++ ключевое слово `static` имеет несколько применений, но когда оно используется в определении класса для переменной, это указывает на то, что переменная ассоциирована с самим классом, а не с каким-либо экземпляром класса. Это означает, что существует только одна копия переменной, общая для всех экземпляров класса, а не отдельная копия для каждого экземпляра.

В контексте класса `Matrix`:

```
static const int ROWS = 2;  
static const int COLS = 2;
```

`ROWS` и `COLS` объявлены как `static const int`. Это означает:

- `static`: Существует только одна копия этих переменных, независимо от того, сколько объектов `Matrix` создано. Они принадлежат самому классу, а не какому-то конкретному объекту.
- `const`: Их значения постоянны; после инициализации они не могут быть изменены. Это важно для определения характеристик, которые не должны изменяться, таких как размеры матриц в этой конкретной реализации.
- `int`: Указывает тип переменных как целочисленный.

### Зачем используется static?

Ключевое слово `static` используется для `ROWS` и `COLS` по нескольким причинам:

- Эффективность использования памяти: Поскольку все матрицы в данном контексте имеют фиксированный размер, нет необходимости, чтобы у каждого экземпляра была своя собственная копия `ROWS` и `COLS`. Единая общая копия снижает использование памяти.
- Информация на уровне класса: Это подчеркивает, что размер матриц является характеристикой самого класса `Matrix` в целом, а не чем-то, что варьируется между экземплярами.
- Простота: Это упрощает доступ к этим значениям, поскольку они могут быть доступны напрямую через имя класса (например, `Matrix::ROWS`) без необходимости экземпляра объекта.

### Какая проблема решается?

Использование `static` для `ROWS` и `COLS` решает потенциальную проблему избыточного использования памяти и логического несоответствия, когда эти размеры варьируются между экземплярами класса `Matrix`, в то время как дизайн предполагает, что все матрицы должны иметь одинаковые размеры. Это гарантирует, что размеры универсально фиксированы для всех экземпляров класса, соответствуя намерению дизайна, что все матрицы имеют одинаковый фиксированный размер. Такой выбор дизайна упрощает реализацию и использование класса `Matrix`, делая ясным, что эти размеры являются внутренними свойствами самого класса, а не свойствами, которые могут отличаться от одного экземпляра к другому.

# Самостоятельная задача: реализовать класс полинома

Полиномы — это математические выражения, состоящие из переменных и коэффициентов, представленных в виде  $a_n * x^n + a_{n-1} * x^{(n-1)} + \dots + a_1 * x + a_0$ . Реализация класса полинома на C++ включает хранение коэффициентов в динамическом массиве и определение операций, таких как сложение и вычисление. Перегрузка операторов позволяет улучшить класс, позволяя выполнение этих операций с использованием стандартных операторов (например, + для сложения).

## Цель

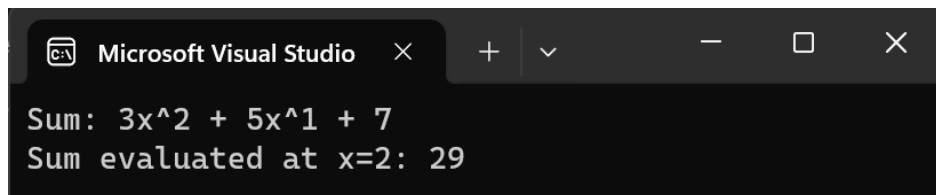
Создать класс `Polynomial`, который представляет полином с использованием массива коэффициентов. Класс будет реализовывать сложение двух полиномов с помощью перегрузки оператора (+), а также метод для вычисления значения полинома при заданном значении  $x$ .

Реализовать класс полинома на C++ со следующими возможностями:

- ✓ Определить конструктор для инициализации полинома с заданной степенью.
- ✓ Реализовать метод для установки коэффициентов для каждого члена полинома.
- ✓ Перегрузить оператор + для выполнения сложения двух полиномов.
- ✓ Создать метод для вычисления значения полинома для заданного значения  $x$ .
- ✓ Отобразить полином в удобочитаемом формате.

Студентам необходимо написать полную реализацию класса `Polynomial`, включая конструктор, деструктор, методы для установки коэффициентов, перегрузку оператора +, вычисление полинома и отображение полинома.

## Пример вывода:



```

Microsoft Visual Studio
Sum: 3x^2 + 5x^1 + 7
Sum evaluated at x=2: 29
  
```