### **Rational (2 балла)**

*В C++ существует множество встроенных типов данных для работы с целочисленной арифметикой и арифметикой над числами с плавающей точкой. Однако при вычислениях над последними возможны ошибки округления, которые могут накапливаться при сложных (состоящих из множества операций) вычислениях. Если при работе с иррациональными числами округлений избежать не получится, то в программах, требующих манипуляций только с рациональными числами, все вычисления можно свести к точной целочисленной арифметике. Именно это вам и предстоит проделать в данном задании.*

#### **Детали реализации**

Необходимо реализовать класс для работы с рациональными числами Rational.

Рациональное число представляется двумя **взаимно простыми** целыми числами - числителем и **положительным** знаменателем. Этот инвариант необходимо поддерживать при выполнении любых операций над объектами класса Rational. Публичный интерфейс класса должен включать:

* Конструктор по умолчанию, который инициализирует число нулем;
* Неявный конструктор преобразования от целых чисел (должен работать следующий код Rational x = 5;). Не забудьте добавить комментарий // NOLINT (см. замечания);
* Конструктор от двух целых чисел (числитель и знаменатель). При этом **не** гарантируется, что числа взаимно простые и ненулевые (об этом далее);
* Методы GetNumerator() и GetDenominator(), возвращающие числитель и знаменатель соответственно;
* Методы SetNumerator(int) и SetDenominator(int), устанавливающие числитель и знаменатель в требуемые значения;
* Бинарные арифметические операции (+, -, /, \*) и их присваивающие версии (+=, -=, /=, \*=);
* Унарные операции (+, -), а также префиксные и постфиксные инкременты и декременты (++, --);
* Операции сравнения;
* Операции ввода из потока и вывода в поток (>>, <<);

При выводе числа в поток необходимо выводить его в виде несократимой дроби в формате <числитель>/<знаменатель> либо <числитель>, если знаменатель равен 1. Ввод осуществляется в том же формате (но дробь на входе необязательно несократима).

Пример:

Rational x = 5; // дробь 5/1

std::cout << x; // вывод: 5

x.SetDenominator(15); // дробь 1/3 (5/15 сократилась)

std::cout << x; // вывод: 1/3

std::cin >> x; // ввод: 14/-6, дробь: -7/3 (общий множитель 2; знаменатель > 0)

#### **Деление на 0**

При делении на 0 (в операции деления или при установке знаменателя в 0) программа должна сообщать об ошибке. Для этих целей в C++ есть механизм *исключений*. Исключение - объект некоторого класса, который унаследован от стандартного класса ошибки библиотеки C++. Если не все ясно сейчас, то это ok, если не все понятно перед зачетом или экзаменом - у вас проблемы.

Класс ошибки RationalDivisionByZero написан за вас, вам лишь требуется в момент, когда происходит деление на ноль, сообщить об ошибке с помощью строки:

throw RationalDivisionByZero{};

**Замечания.**

1. Решение должно состоять из файлов rational.h (объявления) и rational.cpp (определения). Если для решения вам требуются другие компилируемые файлы с реализацией, их стоит добавить в переменную RATIONAL\_SRC в файле CMakeLists.txt.
2. Хорошей идеей будет вынести логику сокращения дроби в отдельный метод.
3. std::gcd.
4. Гарантируется, что числитель и знаменатель в любой момент помещаются в int32\_t.
5. Кодстайл запрещает неявные конструкторы преобразования (и правильно делает), но в этой задаче неявное преобразование из целых чисел в рациональное вполне осмысленно. Чтобы чекер проигнорировал проверку строки с объявлением такого конструктора, добавьте комментарий // NOLINT к этой строке.

### **Array (2 балла)**

*C-style массивы эффективней в использовании чем массивы в динамической памяти, из-за известного на этапе компиляции фиксированного размера и более эффективного расположения в памяти (стековая/автоматическая память). Однако едва ли C-массивы можно назвать удобными в использовании - они спонтанно приводятся к указателям, их не так просто передать в функцию, не работает привычная операция присваивания и т.д. В этой задаче предлагается написать ООП обертку над C-style массивом, который обладает всеми его преимуществами, нивелируя при этом описанные недостатки.*

#### **std::array (C++11)**

Начиная с C++11 в стандартной библиотеке (заголовочный файл <array>) появился шаблонный класс std::array<T, N>, параметризованный типом хранимых элементов T и размером N. Этот класс содержит единственное поле типа "массив T из N элементов" и предоставляет методы для удобного доступа к информации (размер, элементы и т.д), а также изменения элементов массива. Класс std::array в полной мере обеспечивает функционал обычных массивов, не теряя при этом эффективности. Поэтому в современном C++ принято использовать именно std::array вместо C-style массивов. [Подробнее.](https://en.cppreference.com/w/cpp/container/array)

#### **Детали реализации**

Вам необходимо реализовать шаблонный класс Array - упрощенный аналог std::array. Ваш класс должен быть параметризован типом хранимых элементов, а также размером массива. Как было сказано ранее, этот класс должна иметь ровно одно поле - C-style массив требуемого типа и размера. Важным моментом является то, что это поле должно быть публичным! В этом случае, как известно, становится доступна агрегатная инициализация вашего массива:

int arr[4]{1, 2, 3}; // массив {1, 2, 3, 0}

Array<int, 4> my\_arr{1, 2, 3}; // если поле публично, то {1, 2, 3, 0}, иначе - CE

Для корректного прохождения тестов ваш класс должен реализовывать следующие методы (подумайте, какие методы должны быть константными, какие - неконстантными, а какие должны иметь обе версии):

* Операция [] для доступа к элементу массива по индексу;
* Метод At(size\_t idx), который обеспечивает безопасный (с проверкой границ) доступ к элементу по индексу. В случае выхода за границы необходимо бросать исключение типа ArrayOutOfRange, которое уже написано за вас (см. замечания);
* Методы Front() и Back() для доступа к первому и последнему элементу соответственно;
* Метод Data(), возвращающий указатель на начало массива;
* Методы Size() и Empty();
* Метод Fill(const T& value), который заполняет массив значениями value;
* Метод Swap(Array<T, N>& other), обменивающий содержимое массивов одинакового размера.

**Замечания.**

1. Проверка устроена таким образом, что требует от вас жесткого следования принятым (выше) сигнатурам и именованиям сущностей (то есть никакие MyArray, \_\_array\_, back, superSolver3000 не пройдут). Если вы реализовали требуемый функционал не полностью или интерфейс отличается от заявленного, в ответ вы получите ошибку компиляции.
2. "Бросить исключение типа E" значит - написать строку throw E{};. В этот момент выполнение функции прекращается и, если исключение не будет обработано, программа завершится аварийно. Тестирующий код корректно обработает эту ошибку, вам этого делать не нужно.

### **Дополнительная часть (1 балл)**

С помощью механизма перегрузки шаблонов реализуйте функции для извлечения свойств C-style массивов:

* GetSize(array) должна возвращать число элементов в массиве array и 0, если array не является C-style массивом
* GetRank(array) должна возвращать число координат многомерного массива array
* GetNumElements(array) должна возвращать *общее* число элементов в многомерном массиве array

Пример

int x;

int a[3];

int b[3][2][1];

std::cout << GetSize(x) << ' ' << GetSize(a) << ' ' << GetSize(b) << '\n'; // 0 3 3

std::cout << GetRank(x) << ' ' << GetRank(a) << ' ' << GetRank(b) << '\n'; // 0 1 3

std::cout << GetNumElements(x) << ' ' << GetNumElements(a) << ' ' << GetNumElements(b) << '\n'; // 1 3 6

*Важно:* если вы выполнили это задание, добавьте в файл array.h следующую строку, чтобы в тесты была включена проверка этого задания.

#define ARRAY\_TRAITS\_IMPLEMENTED

### **MatrixArray (3 балла)**

*В этом задании вам необходимо реализовать шаблонный класс для работы с матрицами фиксированного размера. Последнее означает, что размеры матрицы должны передаваться в виде шаблонных параметров-целых чисел (non-type template parameters), и для конкретного объекта не могут быть изменены в течении работы программы. С одной стороны, это накладывает ограничения - размер матриц должен быть жестко задан на этапе компиляции и пользователь не имеет возможности его менять. Кроме того, изменения параметров матриц будут требовать повторной компиляции исходного кода, использующего эти матрицы. С другой стороны, так как размеры матрицы - константы времени компиляции, мы имеем возможность использовать нединамические массивы (что заметно ускоряет работу с матрицами), компилятор имеет больше возможностей для оптимизации кода, а неправильная работа с матрицами (умножение матриц неподходящих размеров, обращение неквадратных матриц и т.п.) будет детектироваться еще на этапе сборки программы, а не во время исполнения.*

#### **Детали реализации**

Требуется реализовать шаблонный класс Matrix, принимающий в качестве шаблонных параметров тип хранимых элементов, число строк матрицы и число столбцов. Так как шаблонные параметры - это константы времени компиляции, элементы матрицы можно хранить в двумерном C-style массиве (std::array в этом задании пока запрещен). Это поле-массив должно быть публичным для корректной работы агрегатной инициализации.

Для работы с матрицей предлагается реализовать следующие методы:

* RowsNumber() и ColumnsNumber(), возвращающие число строк и столбцов соответственно;
* Операция () ("круглые скобки") для доступа к элементам матрицы и возможности их изменения (подумайте, почему не []);
* Метод At, аналогичный предыдущему пункту, но бросающий исключение MatrixOutOfRange в случае выхода за границы матрицы;
* Функцию GetTransposed(matrix), возвращающую транспонированную версию переданной матрицы;
* Операции сложения, вычитания умножения матриц и их присваивающие версии. Обратите внимание, что эти операции применимы не к любым парам матриц!;
* Операции умножения (слева и справа) и деления на число, а также их присваивающие версии;
* Операции сравнения на равенство и неравенство двух матриц одинакового размера;
* Операции ввода из потока и вывода в поток.

Пример:

Matrix<int, 2, 3> a{{{1, 2, 3}, {4, 5, 6}}};

Matrix<int, 3, 2> b = GetTransposed(a);

// a + b CE

a + a; // Ok

// a \* a CE

a \* b; // Ok

// a \*= b // CE - в матрицу a нельзя присвоить матрицу другого размера

b \*= Matrix<int, 2, 2>{1, 2, 3, 4}; // Ok

std::cout << a;

1 2 3

4 5 6

**Замечания.**

1. Решение должно состоять из одного файла matrix.h.
2. Помните о константных и неконстантных методах.
3. "Бросить исключение типа E" значит - написать строку throw E{};. В этот момент выполнение функции прекращается и, если исключение не будет обработано, программа завершится аварийно. Тестирующий код корректно обработает эту ошибку, вам этого делать не нужно.
4. Методы этого и дополнительного задания (ниже) необязательно реализовывать наиболее эффективным способом. В первую очередь проверяется правильность реализации.

### **Дополнительная часть (2 балла)**

Для работы с квадратными матрицами есть множество методов не применимых к матрицам произвольного размера. Тем не менее так как размеры матриц известны заранее, мы можем на этапе компиляции проверять внутри них корректность размера.

Дополнительно реализуйте:

* Функцию Transpose(matrix), который транспонирует *исходную* матрицу, изменяя ее (ясно, что транспонировать без изменения размера можно только квадратные матрицы);
* Функцию Trace() - подсчет следа матрицы;
* Функцию Determinant() - детерминант матрицы;
* Функции GetInversed() и Inverse() для обращения матриц (первый метод возвращает обратную, а второй изменяет исходную матрицу).

При попытке обращения вырожденной матрицы необходимо бросать исключение MatrixIsDegenerateError.

*Важно:* если вы выполнили это задание, добавьте в файл matrix.h следующую строку, чтобы в тесты была включена проверка этого задания.

#define MATRIX\_SQUARE\_MATRIX\_IMPLEMENTED