Задача 1(Key-Value storage)

Вася разрабатывает свою структуру — базу данных «ключ-значение». Эта структура данных должна хранить значение, ассоциированное с ключом, и она будет делать это супер-эффективно. Пока для простоты Вася выбрал за основу `std::unordered_map`, но потом он это переделает.

Какие операции должно поддерживать такое хранилище? Правильно: вставка элемента, удаление элемента и поиск элемента. Вася написал прототипы функций `Insert`, `Remove` и `Find`, но функция `Find` почему-то не работает. Помогите Васе её исправить. Вот код Васи:

```
#include <unordered_map>

template <typename Key, typename Value>
class KeyValueStorage {
private:
      std::unordered_map<Key, Value> data;

public:
    void Insert(const Key& key, const Value& value) {
        data[key] = value;
    }

    void Remove(const Key& key) {
        data.erase(key);
    }

    bool Find(const Key& key, Value* const value = nullptr) const;
};


// Почему-то не работает...
//
// template <typename Key, typename Value>
// bool KeyValueStorage<Key, Value>::Find(const Key& key, Value* const value) const {
//     auto it = std::find(data.begin(), data.end(), key);
//     auto val = *it;
//     if (value != nullptr)
//         value = &val;
//     return it != data.end();
// }

// Ваша реализация функции KeyValueStorage::find будет вставлена сюда:
#include "your_version_of_find.h"
```

Ваша версия функции `Find` будет вставлена в конце этого кода. Её заголовок должен быть таким же, как в закомментированной части.

Функция `Find` по задумке должна возвращать `true`, если ключ был найден, и `false` в противном случае. Если второй аргумент функции `Find` отличен от `nullptr` и ключ найден, то функция должна записать найденное значение в тот объект, на который ссылается этот аргумент (предполагается, что новая структура данных сможет быстро определять наличие ключа, но само значение будет извлекаться дорого, и делать это нужно лишь при необходимости). Использовать эту функцию предполагается примерно так:

```
#include "key_value_storage.h"

#include <string>

int main() {
    KeyValueStorage<std::string, int> kv;
    kv.Insert("hello", 42);
    kv.Insert("bye", -13);
    int value = 123;
    auto res = kv.Find("wrong", &value);  // должно вернуться false, а value не должен
меняться
    res = kv.Find("bye", &value);  // должно вернуться true, в value должно быть -13
    res = kv.Find("hello", nullptr);  // должно вернуться true
}
```

Задача 2(Deque)

В этой задаче вам надо будет написать свой дек. Писать его по-честному долго и сложно, поэтому мы пошли вам навстречу: вам нужно написать упрощенную версию дека без итераторов, и умеющую только добавлять элементы в начало и конец. Поддерживать удаление элементов из дека не требуется.

В отличие от стандартного дека возьмите за основу два вектора, растущих каждый в свою сторону. Предлагаем такой прототип — а вам нужно реализовать указанные функции:

```
#include <cstddef>
#include <vector>

template <typename T>
class Deque {
private:
    std::vector<T> head, tail;

public:
    bool Empty() const;

    size_t Size() const;

    void Clear();

    const T& operator [] (size_t i) const;

    T& operator [] (size_t i);

    const T& At(size_t i) const;  // throws std::out_of_range on incorrect index

    T& At(size_t i);  // throws std::out_of_range on incorrect index

    const T& Front() const;

    T& Front();

    const T& Back() const;

    T& Back();

    void PushFront(const T& elem);

    void PushBack(const T& elem);
};
```

Задача 3(MathVector)

Математический *вектор* (не путать с `std::vector`!) – структура линейной алгебры, определяющаяся набором упорядоченных чисел (*координат*). Обозначается как . Число n в таком случае называется *размерностью* вектора.

В качестве примера можно рассмотреть вектора размерности два с координатами в вещественных числах. В таком случае вектор $(1,2)$ будет задавать знакомый нам со школы геометрический вектор с началом в координате $(0,0)$ и концом в $(1,2)$.

Также заметим, что координаты вектора необязательно вещественные числа. Это могут быть рациональные, комплексные или любые другие математические объекты, обладающие набором базовых операций сложения и умножения. (например математические матрицы)

Над математическим вектором можно проводить две операции:

1. Сложение двух векторов одинаковой размерности
2. Умножение вектора на число (тип числа должен быть одинаковым с типом чисел координат у вектора):

Вам дан шаблонный класс `MathVector<T>`, представляющий собой математический вектор с координатами типа `T`:

```
#include <iostream>
#include <vector>

template <typename T>
class MathVector {
 private:
    std::vector<T> data;

 public:
    // Храним в `data` нулевой вектор длины `n`
    MathVector(size_t n) {
        data.resize(n);
    }

    template <typename Iter>
    MathVector(Iter first, Iter last) {
        while (first != last) {
            data.push_back(*first);
        }
    }

    size_t Dimension() const {
         return data.size();
    }

    T& operator [] (size_t i) {
        return data[i];
    }

    const T& operator [] (size_t i) const {
        return data[i];
    }
};

// Output format: (1, 2, 3, 4, 5)
template <typename T>
std::ostream& operator << (std::ostream& out, const MathVector<T>& v) {
    out << '(';
    for (size_t i = 0; i != v.Dimension(); ++i) {
        if (i > 0) {
            out << ", ";
        }
        out << v[i];
```

```
    }
    out << ')';
    return out;
}

template <typename T>
MathVector<T>& operator *= (MathVector<T>& v, const T& scalar) {
    for (size_t i = 0; i != v.Dimension(); ++i) {
        v[i] *= scalar;
    }
    return v;
}

template <typename T>
MathVector<T> operator * (const MathVector<T>& v, const T& scalar) {
    auto tmp(v);
    tmp *= scalar;
    return tmp;
}

template <typename T>
MathVector<T> operator * (const T& scalar, const MathVector<T>& v) {
    return v * scalar;
}
```

Вам требуется исправить ошибки в коде этого класса и дописать операторы += и + для сложения векторов. Считайте, что складываться друг с другом всегда будут только векторы одинаковой размерности.

Задача 4(Многочлены)

Многочлен от одной переменной – алгебраическое выражение, состоящие из суммы нескольких произведений числовых коэффициентов на переменную в натуральной степени.

Так же как и в задаче о математическом векторе, числами здесь могут являться любые объекты со стандартным набором базовых математических операций (сложение, вычитание, умножение, деление), например дробные, вещественные или комплексные числа, а так же математические матрицы и другие алгебраические объекты.

Реализуйте шаблонный класс `Polynomial` (многочлен от одной переменной) на основе контейнера `std::vector`. Тип коэффициентов многочлена передавайте в качестве параметра шаблона. Хранение коэффициентов должно быть плотным (то есть должны храниться все коэффициенты, в том числе и промежуточные нулевые).

Сделайте следующее:

1. Напишите конструкторы, которые
    - создают многочлен по заданному вектору коэффициентов (коэффициенты задаются по возрастанию степени).
    - создают многочлен по заданному коэффициенту (многочлен нулевой степени), который равен значению по умолчанию параметра шаблона.
    - создают многочлен по заданным итераторам на начало и следующий за концом последовательности коэффициентов (аналогично, по возрастанию степени).
2. Перегрузите операторы `==` и `!=`. Ваш код должен быть очень простым. Операторы должны работать и в том случае, когда один из аргументов является скалярной величиной.
3. Перегрузите операторы `+`, `-` и `*`, а также соответствующие операторы `+=`, `-=` и `*=`. Учтите, что должны быть определены и такие арифметические операции, в которых один из аргументов является скалярной величиной.
4. Перегрузите оператор `[]` для получения коэффициента многочлена перед заданной степенью переменной. Достаточно константной версии этого оператора. Оператор должен работать для любых степеней (в том числе больше текущей максимальной). Напишите также метод `Degree` для вычисления степени многочлена (считайте, что у нулевого многочлена степень равна -1).
5. Перегрузите оператор `()` для вычисления значения многочлена в точке. В качестве аргумента этот оператор принимает значение того типа, от которого создан многочлен. Постарайтесь написать эффективный код.
6. Перегрузите оператор `<<` для печати многочлена в поток вывода. Для простоты будем выводить коэффициенты через пробел от **старшей степени к младшей**.
7. Предусмотрите методы `begin()` и `end()` для доступа к константным итераторам, позволяющим перебрать коэффициенты многочлена (это могут быть просто итераторы вектора). При этом ведущие нули коэффициентами не считаются. Итерация должна происходить по возрастанию степени.