Cuprins

Fișiere comume

utils.h

```cpp
#pragma once
#include <initializer_list>
#include <iostream>

constexpr size_t getSize(std::initializer_list<double> l) {
    size_t n = 0;
    auto it = l.begin();
    auto end = l.end();
    while (it++ != end) ++n;
    return n;
}

constexpr void getSize(std::initializer_list<
                           std::initializer_list<double>> list,
                       int& s1, int& s2) {
    constexpr int invalid = -1;
    s1 = 0;
    s2 = invalid;

    for (const auto& l: list) {
        int i = getSize(l);
        ++s1;
        if (s2 != invalid && i != s2) {
            //std::cerr << i << "!=" << s2 << "\n";
            throw std::logic_error("Fixed line size expected");
        }
        s2 = i;
    }
    //std::cerr << "size: " << s1 << ", " << s2 << "\n";
}
constexpr int MAX_SZ = 256;
inline size_t readSize(const char* name, int a = 1, int b = MAX_SZ) {
    int res;
    do {
        std::cout << name << ": ";
        std::cin >> res;
    } while (res < a || res > b);
    return res;
}
```

```

inline void assert(bool cond, const char* msg) {
    if (!cond) throw std::logic_error(msg);
}
template<typename T>
inline T read(const char* name) {
    T res;
    std::cout << name << ": ";
    std::cin >> res;
    return res;
}
```

matrix.h

```
#pragma once
#include "utils.h"

#include <iostream>
#include <cmath>
#include <cstring>
#include <utility>

struct Mat {
    int m, n;
    double *data;
    Mat() : m(0), n(0), data(nullptr) {}
    Mat(int m, int n) : m(m), n(n) { data = new double[m*n]; }
    //template<int m, int n>
    Mat(std::initializer_list<std::initializer_list<double>> list) {
        getSize(list, m, n);
        data = new double[m*n];

        auto it = begin();
        for (const auto& l : list) {
            for (const auto& v : l) *(it++) = v;
        }
    }
    Mat(Mat&& rhs) noexcept :
        m(rhs.m), n(rhs.n),
        data(std::exchange(rhs.data, nullptr)) {
        //rhs.n = rhs.m = 0;
    }

    Mat(const Mat& rhs) : Mat(rhs.m, rhs.n) {
        std::memcpy(data, rhs.data, sizeof(double) *m*n);
    }
    Mat& operator=(const Mat& rhs) {
        setSize(rhs.m, rhs.n);
        std::memcpy(data, rhs.data, sizeof(double) *m*n);
```

```
        return *this;
    }
    Mat& operator=(Mat&& rhs) noexcept {
        this->~Mat();
        m = rhs.m;
        n = rhs.n;
        data = std::exchange(rhs.data, nullptr);

        return *this;
    }

    ~Mat() {
        delete[] data;
    }
    double& at(int i, int j) {
        assert(i < m && j < n, "out of range");
        return data[i *n +j];
    }
    double at(int i, int j) const { return data[i * n + j]; }
    double* begin() { return data; }
    const double* begin() const { return data; }
    double* end() { return data + m*n; }
    const double* end() const { return data + m*n; }

    void setSize(int m, int n) {
        if (this->m == m && this->m == n) return;
        this->~Mat();
        new (this) Mat(m, n);
    }

    static Mat read() {
        Mat res(readSize("m"), readSize("n"));
        for (auto& i : res)
            std::cin >> i;
        return res;
    }
    void print(const char* name) const {
        std::cout << name << " = Mat " << m << "x" << n << "{\n";

        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                std::cout << at(i, j) << " ";
            }
            std::cout << "\n";
        }
        std::cout << "}\n";
    }

    friend std::ostream& operator<<(std::ostream& s, Mat& m) {
        s << "Mat " << m.m << "x" << m.n << "{\n";
```

```
86          for (int i = 0; i < m.m; ++i) {
87              for (int j = 0; j < m.n; ++j) {
88                  s << m.at(i, j) << " ";
89              }
90              s << "\n";
91          }
92          return s<< "}\n";
93      }
94
95      double norm1() const { return normImpl<Type::Row>(n, m); }
96      double normInf() const { return normImpl<Type::Col>(m, n); }
97      double normF() const {
98          double res = 0;
99          for (int i = 0; i < m; ++i)
100             for (int j = 0; j < n; ++j)
101                 res += at(i, j) * at(i, j);
102
103         return std::sqrt(res);
104     }
105     bool isStrictlyRowDiagonallyDominant() const {
106         return isStrictlyDiagonallyDominantImpl<Type::Row>();
107     }
108
109     bool isStrictlyColDiagonallyDominant() const {
110         return isStrictlyDiagonallyDominantImpl<Type::Col>();
111     }
112
113 private:
114
115     enum class Type { Row, Col };
116     template<Type type>
117     bool isStrictlyDiagonallyDominantImpl() const {
118         assert(m == n, "Matrix must be square");
119         for (int i = 0; i < m; ++i) {
120             double val = std::abs(at(i, i));
121             double sum = -val;
122             for (int j = 0; j < m; ++j) {
123                 sum += std::abs(type == Type::Col? at(j, i): at(i, j));
124             }
125             if (sum >= val) {
126                 std::cout << "(" << sum << "," << val << ")";
127                 return false;
128             }
129         }
130         return true;
131     }
132     template<Type type>
133     double normImpl(int sz1, int sz2) const {
134         double max = -1;
135         for (int j = 0; j < sz1; ++j) {
```

```
            double x = 0;
            for (int i = 0; i < sz2; ++i) {
                x += std::abs(type == Type::Col? at(j, i) : at(i, j));
            }
            if (x > max) max = x;
        }
        return max;
    }
};

Mat& add(const Mat& a, const Mat& b, Mat& res) {
    assert(a.m == b.m && a.n == b.n, "Sizes don't match, can't add");
    res.setSize(a.m, a.n);
    for (int i = 0; i < a.m; ++i)
        for (int j = 0; j < a.n; ++j)
            res.at(i, j) = a.at(i, j) + b.at(i, j);
    return res;
}

Mat& mul(double a, const Mat& b, Mat& res) {
    res.setSize(b.m, b.n);

    for (int i = 0; i < res.m; ++i)
        for (int j = 0; j < res.n; ++j)
            res.at(i, j) = a * b.at(i, j);
    return res;
}

Mat& neg(const Mat& a, Mat& res) {
    return mul(-1, a, res);
}
Mat& sub(const Mat& a, const Mat& b, Mat& res) {
    return add(a, neg(b, res), res);
}
Mat& mul(const Mat& a, const Mat& b, Mat& res) {
    assert(a.n == b.m, "Sizes don't match, can't multiply");
    res.setSize(a.m, b.n);

    for (int i = 0; i < res.m; ++i)
        for (int j = 0; j < res.n; ++j) {
            res.at(i, j) = 0;
            for (int k = 0; k < a.n; ++k)
                res.at(i, j) += a.at(i, k) * b.at(k, j);
        }
    return res;
}

Mat& trans(const Mat& a, Mat& res) {
    assert(a.data != res.data, "Can't transpose inplace");
    res.setSize(a.n, a.m);
```

```
186
187     for (int i = 0; i < res.m; ++i)
188         for (int j = 0; j < res.n; ++j)
189             res.at(i, j) = a.at(j, i);
190     return res;
191 }
```

vector.h

```
1  #pragma once
2  #include "utils.h"
3
4  #include <iostream>
5  #include <utility>
6  #include <cmath>
7
8  struct Vec {
9      double *_begin, *_end;
10
11     constexpr double* begin() { return _begin; }
12     constexpr const double* begin() const { return _begin; }
13
14     constexpr double* end() { return _end; }
15     constexpr const double* end() const { return _end; }
16
17     constexpr Vec() : _begin(nullptr), _end(nullptr) {}
18     explicit Vec(size_t n) : _begin(new double[n]), _end(_begin+n) {}
19     Vec(std::initializer_list<double> list) : Vec(getSize(list)) {
20         auto it = _begin;
21         for (const auto& v : list) *(it++) = v;
22     }
23     Vec(const Vec& rhs) : Vec(rhs.size()) {
24         auto it = _begin;
25         for (const auto& v : rhs) *(it++) = v;
26     }
27     Vec(Vec&& rhs) noexcept
28         : _begin(std::exchange(rhs._begin, nullptr)),
29         _end(std::exchange(rhs._end, nullptr)) {}
30
31     Vec& operator=(const Vec& rhs) {
32         if (size() != rhs.size()) {
33             this->~Vec();
34             new (this) Vec(rhs.size());
35         }
36         auto it = _begin;
37         for (const auto& v : rhs) *(it++) = v;
38         return *this;
39     }
40     Vec& operator=(Vec&& rhs) noexcept {
41         this->~Vec();
```

```
        _begin = std::exchange(rhs._begin, nullptr);
        _end = std::exchange(rhs._end, nullptr);
        return *this;
    }

    ~Vec() { delete[] _begin; }

    constexpr size_t size() const { return _end - _begin; }

    constexpr double& operator[](size_t i) { return _begin[i]; }
    constexpr double operator[](size_t i) const { return _begin[i]; }

    void setSize(size_t n) {
        if (size() == n) return;
        *this = Vec(n);
    }

    friend std::ostream& operator<<(std::ostream& s, const Vec& v) {
        s << "(";
        double* it = v._begin;
        for (double* end = v._end - 1; it < end; ++it)
            s << *it << ", ";

        if (it < v._end) s << *it;

        return s << ")";
    }
    static Vec read() {
        Vec res(readSize("n"));
        for (auto& v: res) std::cin >> v;
        return res;
    }
    double norm() const;
};
void assertSizes(const Vec& a, const Vec& b) {
    assert(a.size() == b.size(), "Sizes don't match");
}

Vec& add(const Vec& a, const Vec& b, Vec& res) {
    assertSizes(a, b);
    res.setSize(a.size());
    auto aIt = a.begin();
    auto bIt = b.begin();
    for (auto& v : res) v = *(aIt++) + *(bIt++);
    return res;
}

Vec& mul(double a, const Vec& b, Vec& res) {
    res.setSize(b.size());
    auto it = b.begin();
```

```
    for (auto& v : res) v = a * (*(it++));
    return res;
}

Vec& neg(const Vec& b, Vec& res) { return mul(-1, b, res); }

Vec& sub(const Vec& a, const Vec& b, Vec& res) {
    return add(a, neg(b, res), res);
}
double dot(const Vec& a, const Vec& b) {
    double res = 0;
    assertSizes(a, b);
    auto bIt = b.begin();
    for (auto& v : a) res += v * (*(bIt++));
    return res;
}
double norm(const Vec& a) {
    return std::sqrt(dot(a, a));
}
double Vec::norm() const {
    return ::norm(*this);
}
```

stack.h

```
#pragma once
#include "utils.h"

template<typename T>
struct Stack {
    struct Node {
        T val;
        Node* next;
    };

    Stack(const Stack&) = delete;
    Stack& operator=(const Stack&) = delete;
    Node* top;
    Stack(Node* top = nullptr) : top(top) {}
    ~Stack() {
        while (top) {
            Node* n = top;
            top = top->next;
            delete n;
        }
    }
    void push(T val) {
        Node *n = new Node{ val, top };
        top = n;
    }
```

```
    void quickPop() {
        Node* n = top;
        top = top->next;
        delete n;
    }
    T pop() {
        assert(top, "Empty stack");
        Node* n = top;
        top = top->next;
        T res = n->val;
        delete n;
        return res;
    }
    bool empty() const { return top == nullptr; }
    friend std::ostream& operator<<(std::ostream& s, const Stack& st) {
        if (st.empty()) return s << "{}";
        s << "{";
        const Node* it = st.top;
        while (it->next != nullptr) {
            s << it->val << ", ";
            it = it->next;
        }

        return s << it->val << "}";
    }
};
```

queue.h

```
#pragma once
#include "utils.h"

template<class T>
struct Queue {
    struct Node {
        T val;
        Node* next;
    };
    Node* top;
    Node* bot;
    Queue(Node* top = nullptr, Node* bot = nullptr) : top(top), bot(bot) {}
    /*
    Queue(const Queue&) = delete;
    Queue& operator=(const Queue&) = delete;
    ~Queue() {
        while (top) {
            Node* n = top;
            top = top->next;
            delete n;
        }
```

```
        }*/
    void push(T val) {
        Node *n = new Node{ val, nullptr };
        if (bot == nullptr) {
            top = n;
        }
        else {
            bot->next = n;
        }
        bot = n;
    }

    T pop() {
        assert(top, "Empty queue");
        Node* old = top;
        T res = old->val;
        top = top->next;
        if (top == nullptr)
            bot = nullptr;
        delete old;
        return res;
    }
    bool empty() const { return top == nullptr; }
    static Queue read(const char* msg) {
        Queue q;
        std::cout << msg  << ":\n";
        int len1 = readSize("n");
        for (int i = 0; i < len1; ++i) {
            T s;
            std::cin >> s;
            q.push(s);
        }
        return std::move(q);
    }

    friend std::ostream& operator<<(std::ostream& s, const Queue& st) {
        if (st.empty()) return s << "{}";
        s << "{";
        const Node* it = st.top;
        while (it->next != nullptr) {
            s << it->val << ", ";
            it = it->next;
        }

        return s << it->val << "}";
    }
};
```

list.h

```cpp
#pragma once
#include "utils.h"

#include <type_traits>
#include <utility>


// We should never check for (bot == nullptr),
// so we don't update it when the list becomes empty.
template<typename T>
struct List {
    struct Node {
        T val;
        Node* next;
    };
    struct It {
        const Node* n;
        constexpr It(const Node* n) : n(n) {}
        constexpr It& operator++() {
            n = n->next;
            return *this;
        }
        constexpr auto& operator*() { return n->val; }
        constexpr bool operator ==(const It& rhs) { return n == rhs.n; }
        constexpr bool operator !=(const It& rhs) { return n != rhs.n; }
    };
    Node* top;
    Node* bot;
    constexpr List(Node* top = nullptr, Node* bot = nullptr)
        : top(top), bot(bot) {}
    List(const List&) = delete;
    List& operator=(const List&) = delete;

    constexpr It begin() const { return top; }
    constexpr It end() const { return nullptr; }

    ~List() {
        while (top) {
            Node* n = top;
            top = top->next;
            delete n;
        }
    }
    List(List&& rhs) noexcept
        : top(std::exchange(rhs.top, nullptr)),
          bot(std::exchange(rhs.bot, nullptr)) {}
    List& operator=(List&& rhs) noexcept {
        this->~List();
```

```
        this->top = std::exchange(rhs.top, nullptr);
        this->bot = std::exchange(rhs.bot, nullptr);
    }

    T& first() { return top->val; }
    T& last() { return bot->val; }
    T pop_front() {
        assert(top, "Empty list");
        Node* n = top;
        top = top->next;
        T res = n->val;
        delete n;
        return res;
    }

    void push_front(T val) {
        Node *n = new Node{ val, top };
        top = n;
    }
    void push_back(T val) {
        Node *n = new Node{ val, nullptr };
        if (top == nullptr) {
            top = n;
        }
        else {
            bot->next = n;

        }
        bot = n;
    }
    bool empty() const { return top == nullptr; }

    bool operator== (const List& rhs) {
        for (auto it = top, rit = rhs.top;
            it != nullptr;
            it = it->next, rit = rit->next) {
            if (it->val != rit->val) return false;
        }
        return true;
    }

    static List read(const char* msg) {
        List q;
        std::cout << msg << ":\n";
        int len1 = readSize("n");
        for (int i = 0; i < len1; ++i) {
            T s;
            std::cin >> s;
            q.push_front(s);
        }
```

```
        return std::move(q);
    }

    friend std::ostream& operator<<(std::ostream& s, const List& st) {
        if constexpr (std::is_same_v<T, char>) {
            const Node* it = st.top;
            while (it != nullptr) {
                s << it->val;
                it = it->next;
            }
            return s;
        }
        else {
            if (st.empty()) return s << "{}";
            s << "{";
            const Node* it = st.top;
            while (it->next != nullptr) {
                s << it->val << ", ";
                it = it->next;
            }

            return s << it->val << "}";
        }
    }

    void remove(T& val) {
        remove_if([&] (T& t) { return t ==val; });
    }
    // removes all elements that satisfy p
    template<class P>
    void remove_if(P p) {
        apply_on(p, [] (Node* n) { delete n; } );
    }
    //applies f() on all nodes that satisfy the predicate p()
    template<class P, class F>
    void apply_on(P p, F f) {
        for (;;) {
            auto* n = top;
            if (!n) {
                return;
            }
            if (!p(n->val)) break;
            top = n->next;
            f(n);
        }
        auto* prev = top;
        auto* it = prev->next;

        while (it) {
            if (p(it->val)) {
```

```
                prev->next = it->next;
                if (prev->next == nullptr) {
                    bot = prev;
                }
                f(it);
                it = prev;
            } else {
                prev = it;
                it = it->next;
            }
        }
    }
    // P is a predicate on T
    template<typename P>
    Node* find(P p) {
        auto n = top;
        for (; n; n = n->next) {
            if (p(n->val))
                return n;
        }
        return n;
    }
};

// keeps in l all the elements that don't satisfy the predicate p
// and retuns pair of:
//  - a reference to the original list
//  - a list containing the elements that satisfy p
template<typename T, typename P>
constexpr auto partition_split(List<T>& l, P p) {
    struct res_t {
        List<T>& notSatisfying;
        List<T> satisfying;
    } res = { l, {} };
    List<T>& sat = res.satisfying;
    auto insertNode = [&sat](auto* n) {
        n->next = sat.top;
        sat.top = n;
        if (sat.bot == nullptr) {
            sat.bot = n;
        }
    };
    l.apply_on(p, insertNode);
    return res;
}
```

Alocarea dinamică a memoriei. Tipuri specifice.

Laborator 1

16. Scrieți funcții pentru implemetarea operațiilor specifice pe matrice de numere reale cu m linii și n coloane: suma, diferența și produsul al două matrice, produsul dintre o matrice și un scalar real, transpusa unei matrice, norme matriceale specifice[1], citirea de la tastatură a componentelor unei matrice, afișarea componentelor matricei. Pentru cazul particular al unei matrice patratice de ordin n, să se testeze dacă aceasta satisface criteriul de dominanță pe linii[2] sau pe coloane[3]. Se vor folosi tablouri bidimensionale alocate static.

```cpp
#include "utils.h"

#include <iostream>
#include <cmath>

struct Mat {
    double data[MAX_SZ][MAX_SZ] {};
    int m, n;

    Mat() : m(0), n(0) {}
    Mat(int m, int n) : m(m), n(n) {}

    static Mat read() {
        Mat res(readSize("m"), readSize("n"));

        for (int i = 0; i < res.m; ++i)
            for (int j = 0; j < res.n; ++j)
                std::cin >> res.data[i][j];
        return res;
    }
    void setSize(int m, int n) {
        this->m = m;
        this->n = n;
    }
```

[1] Dacă $A \in \mathcal{M}_{m\times n}(\mathbb{R})$, atunci $||A||_1 = \max\limits_{1\le j\le n} \sum\limits_{i=1}^{m} |a_{ij}|$, $||A||_\infty = \max\limits_{1\le i\le m} \sum\limits_{j=1}^{n} |a_{ij}|$, $||A||_F = \sqrt{\sum\limits_{i=1}^{m}\sum\limits_{j=1}^{n} a_{ij}^2}$.

[2] $A \in \mathcal{M}_n(\mathbb{R})$ este strict diagonal dominantă pe linii dacă $|a_{ii}| > \sum\limits_{\substack{j=1\\ j\neq i}}^{n} |a_{ij}|$, pentru orice $i = 1, ..., n$.

[3] $A \in \mathcal{M}_n(\mathbb{R})$ este strict diagonal dominantă pe colonane dacă $|a_{jj}| > \sum\limits_{\substack{i=1\\ i\neq j}}^{n} |a_{ij}|$, pentru orice $j = 1, ..., n$.

```
    double& at(int i, int j) { return data[i][j]; }
    double at(int i, int j) const { return data[i][j]; }

    void print(const char* name) const {
        std::cout << name << " = Mat " << m << "x" << n << "{\n";
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j)
                std::cout << at(i, j) << " ";
            std::cout << "\n";
        }
        std::cout << "}\n";
    }
private:
    enum class Type { Row, Col };
    template<Type type>
    bool isStrictlyDiagonallyDominantImpl() const {
        assert(m == n, "Matrix must be square");
        for (int i = 0; i < m; ++i) {
            double val = std::abs(at(i, i));
            double sum = -val;
            for (int j = 0; j < m; ++j)
                sum += std::abs(type == Type::Col ? at(j, i) : at(i, j));
            if (sum >= val) return false;
        }
        return true;
    }

    template<Type type>
    double normImpl(int sz1, int sz2) const {
        double max = -1;
        for (int j = 0; j < sz1; ++j) {
            double x = 0;
            for (int i = 0; i < sz2; ++i)
                x += std::abs(type == Type::Col ? at(j, i) : at(i, j));
            if (x > max) max = x;
        }
        return max;
    }
public:
    double norm1() const { return normImpl<Type::Row>(n, m); }
    double normInf() const { return normImpl<Type::Col>(m, n); }
    double normF() const {
        double res = 0;
        for (int i = 0; i < m; ++i)
            for (int j = 0; j < n; ++j)
                res += at(i, j) * at(i, j);

        return std::sqrt(res);
    }
    bool isStrictlyRowDiagonallyDominant() const {
```

```
75          return isStrictlyDiagonallyDominantImpl<Type::Row>();
76      }
77      bool isStrictlyColDiagonallyDominant() const {
78          return isStrictlyDiagonallyDominantImpl<Type::Col>();
79      }
80  };
81
82  Mat& add(const Mat& a, const Mat& b, Mat& res) {
83      assert(a.m == b.m && a.n == b.n, "Sizes don't match, can't add");
84      res.setSize(a.m, a.n);
85      for (int i = 0; i < a.m; ++i)
86          for (int j = 0; j < a.n; ++j)
87              res.at(i, j) = a.at(i, j) + b.at(i, j);
88      return res;
89  }
90  Mat& mul(double a, const Mat& b, Mat& res) {
91      res.setSize(b.m, b.n);
92
93      for (int i = 0; i < res.m; ++i)
94          for (int j = 0; j < res.n; ++j)
95              res.at(i, j) = a * b.at(i, j);
96      return res;
97  }
98  Mat& neg(const Mat& a, Mat& res) { return mul(-1, a, res); }
99  Mat& sub(const Mat& a, const Mat& b, Mat& res) {
100     return add(a, neg(b, res), res);
101 }
102 Mat& mul(const Mat& a, const Mat& b, Mat& res) {
103     assert(a.n == b.m, "Sizes don't match, can't multiply");
104     res.setSize(a.m, b.n);
105
106     for (int i = 0; i < res.m; ++i)
107         for (int j = 0; j < res.n; ++j) {
108             res.at(i, j) = 0;
109             for (int k = 0; k < a.n; ++k)
110                 res.at(i, j) += a.at(i, k) * b.at(k, j);
111         }
112     return res;
113 }
114 Mat& trans(const Mat& a, Mat& res) {
115     assert(a.data != res.data, "Can't calculate the transpose inplace");
116     res.setSize(a.n, a.m);
117
118     for (int i = 0; i < res.m; ++i)
119         for (int j = 0; j < res.n; ++j)
120             res.at(i, j) = a.at(j, i);
121     return res;
122 }
```

Laborator 2

18. Scrieți funcții pentru implementarea operațiilor specifice pe vectori din $\mathbb{R}^n$: suma, diferența și produsul scalar al doi vectori, produsul dintre un vector și un scalar real, negativarea unui vector, norma euclidiană a unui vector, citirea de la tastură a celor n componente ale unui vector, afișarea componentelor vectorului sub forma unui n-uplu de elemente. Se vor folosi tablouri unidimensionale alocate dinamic.

```cpp
#include "utils.h"

#include <iostream>
#include <utility>
#include <cmath>

struct Vec {
    double *_begin, *_end;

    constexpr double* begin() { return _begin; }
    constexpr const double* begin() const { return _begin; }


    constexpr double* end() { return _end; }
    constexpr const double* end() const { return _end; }

    constexpr Vec() : _begin(nullptr), _end(nullptr) {}
    explicit Vec(size_t n) : _begin(new double[n]), _end(_begin+n) {}
    Vec(std::initializer_list<double> list) : Vec(getSize(list)) {
        auto it = _begin;
        for (const auto& v : list) *(it++) = v;
    }
    Vec(const Vec&) = delete;
    Vec(Vec&& rhs) noexcept
        : _begin(std::exchange(rhs._begin, nullptr)),
        _end(std::exchange(rhs._end, nullptr)) {}

    Vec& operator=(const Vec&) = delete;
    Vec& operator=(Vec&& rhs) noexcept {
        this->~Vec();
        _begin = std::exchange(rhs._begin, nullptr);
        _end = std::exchange(rhs._end, nullptr);
        return *this;
    }

    ~Vec() { delete[] _begin; }

    constexpr size_t size() const { return _end - _begin; }

    constexpr double& operator[](size_t i) { return _begin[i]; }
    constexpr double operator[](size_t i) const { return _begin[i]; }

    void setSize(size_t n) {
```

```
        if (size() == n) return;
        *this = Vec(n);
    }

    friend std::ostream& operator<<(std::ostream& s, const Vec& v) {
        s << "(";
        double* it = v._begin;
        for (double* end = v._end - 1; it < end; ++it)
            s << *it << ", ";

        if (it < v._end) s << *it;

        return s << ")";
    }
    static Vec read() {
        Vec res(readSize("n"));
        for (auto& v: res) std::cin >> v;
        return res;
    }
    double norm() const;
};
void assertSizes(const Vec& a, const Vec& b) {
    assert(a.size() == b.size(), "Sizes don't match");
}

Vec& add(const Vec& a, const Vec& b, Vec& res) {
    assertSizes(a, b);
    res.setSize(a.size());
    auto aIt = a.begin();
    auto bIt = b.begin();
    for (auto& v : res) v = *(aIt++) + *(bIt++);
    return res;
}

Vec& mul(double a, const Vec& b, Vec& res) {
    res.setSize(b.size());
    auto it = b.begin();
    for (auto& v : res) v = a * (*(it++));
    return res;
}

Vec& neg(const Vec& b, Vec& res) { return mul(-1, b, res); }

Vec& sub(const Vec& a, const Vec& b, Vec& res) {
    return add(a, neg(b, res), res);
}
double dot(const Vec& a, const Vec& b) {
    double res = 0;
    assertSizes(a, b);
    auto bIt = b.begin();
```

```
94      for (auto& v : a) res += v * (*(bIt++));
95      return res;
96  }
97  double norm(const Vec& a) {
98      return std::sqrt(dot(a, a));
99  }
100 double Vec::norm() const {
101     return ::norm(*this);
102 }
```

Tablouri

Laborator 3

7. Folosind structurile de date `VECTOR` și `MATRICE` definite la curs și funcțiile necesare, rezolvați următorul sistem algebric liniar cu n ecuații și n necunoscute folosind metoda lui Gauß de eliminare.

$$\begin{cases} 2x_1 - x_2 = 1 \\ -x_1 + 2x_2 - x_3 = 1 \\ -x_2 + 2x_3 - x_4 = 1 \\ \cdots\cdots\cdots\cdots \\ -x_{n-2} + 2x_{n-1} - x_n = 1 \\ -x_{n-1} + 2x_n = 1, \quad n \in \mathbb{N}, 2 \leq n \leq 50 \end{cases}.$$

```cpp
#include "utils.h"

#include "vector.h"
#include "matrix.h"

#include <iostream>
#include <cmath>

constexpr double eps = 1e-7;

Vec& mul(const Mat& m, const Vec& v, Vec& res) {
    assert(v.begin() != res.begin(), "Can't multiply inplace");
    assert(v.size() == size_t(m.n), "Sizes don't match");
    res.setSize(m.m);


    for (size_t i = 0; i < res.size(); ++i) {
        res[i] = 0;
        for (size_t k = 0; k < v.size(); ++k)
            res[i] += m.at(i, k) * v[k];
    }
    return res;
}

// A * X = b
struct System {
    Mat A;
```

```
    Vec b;

    System(int n, int m) : A(n, m), b(n) {}

    System(std::initializer_list<std::initializer_list<double>> A,
           std::initializer_list<double> b) : A(A), b(b) {
           assert(std::size_t(this->A.m) == this->b.size(),
                  "sizes don't match");
    }

    friend std::ostream& operator<<(std::ostream& s, const System& sys) {
        s << "System " << sys.A.m << "x" << sys.A.n << ": \n";
        auto& A = sys.A;
        for (int i = 0; i < A.m; ++i) {
            s << "{";
            for (int j = 0; j < A.n; ++j) {
                //showpos shows a '+' in front of positive numbers
                if (std::abs(A.at(i, j)) > eps)
                    s << std::showpos << A.at(i, j)
                      << std::noshowpos << "*x" << (j+1) << " ";
            }
            s<< "= " << sys.b[i] << "\n";
        }

        return s;
    }

    // L_i += f * L_j
    void addLines(int i, double f, int j) {
        for (int k = 0; k < A.n; ++k) {
            A.at(i, k) += f * A.at(j, k);
        }
        b[i] += f * b[j];
    }

    // L_i *= f
    void multiplyLine(int i, double f) {
        for (int k = 0; k < A.n; ++k) {
            A.at(i, k) *= f;
        }
        b[i] *= f;
    }

    Vec solveTriangulated() {
        for (int i = A.m-1; i > 0; --i) {
            addLines(i-1, -A.at(i-1, i), i);
            multiplyLine(i, 1 / A.at(i, i));
        }
        return b;
    }
```

```
78
79      bool checkSolution(const Vec& x) const {
80          Vec r;
81          mul(A, x, r);
82          sub(b, r, r);
83          for (auto& v : r) {
84              if (std::abs(v) > eps) return false;
85          }
86          return true;
87      }
88
89      static Vec solveCustom(int n) {
90          System s(n, n);
91          for (auto& v : s.A) v = 0;
92          s.A.at(0,0) = 2;
93          s.A.at(0,1) = -1;
94          s.b[0] = 1;
95          for (int i = 1; i < n - 1; ++i) {
96              s.b[i] = 1;
97
98              s.A.at(i,i-1) = -1;
99              s.A.at(i,i) = 2;
100             s.A.at(i,i+1) = -1;
101         }
102         s.b[n-1] = 1;
103         s.A.at(n-1, n-2) = -1;
104         s.A.at(n-1, n-1) = 2;
105         s.customTriangulate();
106         return s.solveTriangulated();
107     }
108     void customTriangulate() {
109         multiplyLine(0, 1 / A.at(0, 0));
110         for (int i = 1; i < A.m; ++i) {
111             addLines(i, 1, i-1);
112             multiplyLine(i, 1 / A.at(i, i));
113         }
114     }
115 };
116
117 int main() {
118     try {
119         int n = readSize("n", 2, 51);
120         std::cout << "x = "<< System::solveCustom(n) << "\n";
121     } catch (std::exception& e) {
122         std::cerr << "Error" << e.what() << "\n";
123         return 1;
124     }
125     return 0;
126 }
```

Liste liniare simplu înlănțuite
Stive și cozi

Laborator 4

5. Se citește un text de la tastatura (poate conține orice caracter, inclusiv spații) și se încarcă în două stive: o stivă va conține doar litere mici, iar cealaltă doar litere mari. Se citește de la tastatură o vocală a alfabetului englez (literă mare sau mică). Ștergeți stiva corespunzătore până la întâlnirea vocalei citite.

```cpp
#include "utils.h"
#include "stack.h"

#include <iostream>
#include <cstdlib>

bool isVowel(char c) {
    c = tolower(c);
    return c == 'a' || c== 'e' || c== 'i' || c=='o' || c=='u';
}

int main() {
    std::string str;
    std::cout << "str: ";
    std::getline(std::cin, str);

    char v;
    do {
        v = read<char>("vowel");
    } while (!isVowel(v));

    Stack<char> lower;
    Stack<char> upper;
    auto printStacks = [&] (const char* s) {
                           std::cout << s;
                           std::cout << "lower:" << lower << "\n";
                           std::cout << "upper:" << upper << "\n";
                       };

    printStacks("Before:\n");
    for (auto& c : str) {
        if (islower(c)) lower.push(c);
        else if (isupper(c)) upper.push(c);
```

```
    }

    printStacks("After Adding:\n");
    Stack<char>& stack = isupper(v)? upper: lower;
    while (!stack.empty()) {
        if (char c = stack.pop(); c == v) {
            break;
        }
    }
    printStacks("Result:\n");
    return 0;
}
```

Laborator 5

11. Creați o listă liniară simplu înlănțuită în nodurile căreia sunt memorate numere naturale. Separați numerele naturale memorate în listă, în două liste, una corespunzătore numerelor pare și cealaltă, numerelor impare. Afișați cele două liste. Ștergeți din lista numerelor pare, o valoare pară x, citită de la tastatură, ori de câte ori aprare în listă.

```
#include "utils.h"
#include "list.h"

#include <iostream>

int main() {
    auto nums = List<int>::read("numbers");

    auto [evens, odds] = partition_split(nums, [](int n) {return n % 2;});
    std::cout << "odd:  " << odds << "\n";
    std::cout << "even: " << evens << "\n";

    int x;
    do {
        std::cout << "x (must be even): ";
        std::cin >> x;
    } while (x % 2);

    std::cout << "removing...\n";
    evens.remove(x);

    std::cout << "even: " << evens << "\n";
    return 0;
}
```

17. Modelați printr-o LLSI un stoc de produse caracterizate prin: denumire, unitate de măsură, cantitate și preț unitar. Implementați principalele operații pe stoc: crearea stocului, introducerea unui produse nou, eliminarea unui produs in cazul în care acesta a fost vândut în întregime, modificare informații despre un produs (de exemplu, modificarea cantității unui produs, în cazul vânzării), calculul valorii stocului la un moment dat, listare stoc.

stock.h

```cpp
#pragma once
#include "fixedPoint.h"
#include "list.h"

#include <iostream>
#include <iomanip>

constexpr size_t precision = 2;
using FP = FixedPoint<precision>;

template<class T>
constexpr void setMax(T& a, T b) {
    if (b > a) a = b;
}
struct Product {
    std::string name;
    FP quantity;
    std::string unit;
    FP unitPrice;
    constexpr auto totalPrice() const {
        return quantity * unitPrice;
    }
};

struct Stock {
    List<Product> products;

    //retun value: was the operation was successful
    bool add(Product p) {
        auto* res = products.find(
            [&] (Product& other) { return other.name == p.name; });
        if (res != nullptr) {
            auto& v = res->val;
            std::cout << "Product '"<< p.name << "' already exists.\n"
                "Resuplying.\n";
            if (v.unit != p.unit) {
                std::cout << "Units did not match ('"
                          << v.unit <<"' != '" <<p.unit << "')\n";
                return false;
            }
            v.unitPrice += (v.quantity*v.unitPrice + p.quantity*p.unitPrice)
                / (v.quantity +p.quantity);
            v.quantity += p.quantity;
        } else {
            products.push_front(p);
        }
        return true;
    }
```

```
49      //retun value: was the operation was successful
50      bool resupply(const std::string& name, FP quantity) {
51          auto* res = products.find(
52              [&] (Product& other) { return other.name == name; });
53          if (res == nullptr){
54              std::cout << "Product '" << name << "' not found.\n";
55              return false;
56          }
57          res->val.quantity += quantity;
58          return true;
59      }
60      //retun value: was the operation was successful
61      bool sell(const std::string& name, FP quantity) {
62          auto* res = products.find(
63              [&] (Product& other) { return other.name == name; });
64          if (res == nullptr) {
65              std::cout << "Product '" << name << "' not found.\n";
66              return false;
67          }
68          auto& v = res->val;
69          if (quantity > v.quantity) {
70              std::cout << "Quantity too high. Can't sell. (" << quantity
71                        << " > " << v.quantity << ")\n";
72              return false;
73          }
74          v.quantity -= quantity;
75          return true;
76      }
77      constexpr FP value() const {
78          FP total = 0;
79          for (auto& p : products)
80              total += p.totalPrice();
81          return total;
82      }
83      void print() const {
84          if (products.empty()) {
85              std::cout << "No products\n";
86              return;
87          }
88          auto ph = PrintHelper(*this);
89          ph.printHeader();
90          FP total = 0;
91          for (auto& p : products) {
92              ph.printProd(p);
93              total += p.totalPrice();
94          }
95          ph.printFooter(total);
96      }
97  private:
98
```

```
99      class PrintHelper {
100         static constexpr std::string_view fields[] = {
101             "Name", "Quantity", "Unit Price", "Total Price"
102         };
103         struct Longest {
104             size_t name = fields[0].size();
105             size_t quantity = 0;// fields[1].size();
106             size_t unit = 0;
107             size_t unitPrice = fields[2].size();
108             size_t totalPrice = fields[3].size();
109         } longest;
110     public:
111         constexpr PrintHelper(const Stock& s) {
112             auto& l = longest;
113             for (auto& p : s.products) {
114                 setMax(l.name, p.name.size());
115                 setMax(l.quantity, p.quantity.textLen());
116                 setMax(l.unit, p.unit.size());
117                 setMax(l.unitPrice, p.unitPrice.textLen());
118                 setMax(l.totalPrice, p.totalPrice().textLen());
119             }
120
121             longest.quantity = std::max(fields[1].size() - longest.unit -1,
122                                         longest.quantity);
123         }
124         void printProd(const Product& p) const {
125             std::cout << "| " << std::left;
126             printPadded(longest.name, p.name);
127             std::cout << " | " << std::right;
128             printPadded(longest.quantity, p.quantity);
129             std::cout << " " << std::left;
130             printPadded(longest.unit, p.unit);
131             std::cout << " | " << std::right;
132             printPadded(longest.unitPrice, p.unitPrice);
133             std::cout << " | ";
134             printPadded(longest.totalPrice, p.totalPrice());
135             std::cout << " |\n";
136         }
137         void printHeader() const {
138             printLine();
139             std::cout << "| ";
140             printCentered(longest.name, fields[0]);
141             std::cout << " | ";
142             printCentered(longest.quantity + 1 + longest.unit, fields[1]);
143             std::cout << " | ";
144             printCentered(longest.unitPrice, fields[2]);
145             std::cout << " | ";
146             printCentered(longest.totalPrice, fields[3]);
147             std::cout << " |\n";
148             printLine();
```

```
149         }
150         void printFooter(FixedPoint<precision> totalPrice) const {
151             printLine();
152             std::cout << "| ";
153             printPadded(longest.name, "");
154             std::cout << " | ";
155             printPadded(longest.quantity + 1 + longest.unit, "");
156             std::cout << " | ";
157             printPadded(longest.unitPrice, "");
158             std::cout << " | "<<std::right;
159             printPadded(longest.totalPrice, totalPrice);
160             std::cout << " |\n";
161             printLine();
162         }
163 
164     private:
165         template<typename T>
166         static void printPadded(size_t len, T v) {
167             std::cout << std::setw(len) << v;
168         }
169         static void printCentered(size_t len, std::string_view s) {
170             //we can safely assume that len >= s.len();
171             int total = len - s.size();
172             int left = total / 2;
173             int right = total - left;
174             hline(left, ' ');
175             std::cout << std::setw(0) << s;
176             hline(right, ' ');
177         }
178 
179         static void hline(int len, char c) {
180             while (--len >= 0)
181                 std::cout << c;
182         }
183 
184         void printLine() const {
185             std::cout << "+";
186             hline(longest.name+2, '-');
187             std::cout << "+";
188             hline(longest.quantity + 3 + longest.unit, '-');
189             std::cout << "+";
190             hline(longest.unitPrice+2, '-');
191             std::cout << "+";
192             hline(longest.totalPrice+2, '-');
193             std::cout << "+\n";
194         }
195     };
196 };
```

stock.cpp

```cpp
#include "stock.h"

#include "inputHelper.h"
#include "utils.h"
#include "list.h"

#include <iostream>
#include <iomanip>

struct Command {
    char shortName;
    std::string_view name;
    std::string_view args;
    bool (*f)(Stock& s, MultiInputHelper& ih);//returns true if should exit
    std::string_view description;
    void print() const {
        std::cout << "  " << std::setw(10) << std::left << name
                  << " - " << shortName << " "
                  << std::setw(25) << args << " - " << description << "\n";
    }
};
using IH = MultiInputHelper;

void printHelp();
bool printHelp(Stock&, IH&) {
    printHelp();
    return true;
}
bool quit(Stock&, IH&) { return false; }
bool print(Stock& s, IH&) {
    s.print();
    return true;
}

bool add(Stock& s, IH& ih) {
    Product p;
    ih.readName(p.name, "name");
    ih.readFP(p.quantity, "quantity");
    ih.readString(p.unit, "unit");
    ih.readFP(p.unitPrice, "unit price");
    s.add(p);
    return true;
}

bool sell(Stock& s, IH& ih) {
    std::string name;
    FP quantity;
    ih.readName(name, "name");
```

```
49      ih.readFP(quantity, "quantity");
50
51      s.sell(name, quantity);
52      return true;
53  }
54
55  bool resupply(Stock& s, IH& ih) {
56      std::string name;
57      FP quantity;
58      ih.readName(name, "name");
59      ih.readFP(quantity, "quantity");
60
61      s.resupply(name, quantity);
62      return true;
63  }
64
65  bool value(Stock& s, IH&) {
66      std::cout << "Total stock value: " << s.value() << "\n";
67      return true;
68  }
69  bool init(Stock& s, IH& ih) {
70      std::string str;
71      for (;;) {
72          ih.getLine("+ ");
73          add(s, ih);
74          ih.getLine("Add more products (y/N)? ");
75          if (tolower(ih.readChar()) != 'y') break;
76      }
77      return true;
78  }
79  constexpr Command cmds[] = {
80      { 'H', "help", "", printHelp, "Show help" },
81      { 'A', "add", "name quantity unit price", add, "Add product" },
82      { 'I', "init", "", init, "Add multiple elements" },
83      { 'S', "sell", "name quantity", sell, "Sell product" },
84      { 'R', "resupply", "name quantity", resupply, "Resupply product" },
85      { 'V', "value", "", value, "Print stock total value" },
86      { 'P', "print", "", print, "Print a table of products" },
87      { 'Q', "quit", "", quit, "Quit the program" },
88  };
89  void printHelp() {
90      std::cout << "\nFormat of commands: \n"
91                   "name - shortName args - description\n";
92
93      for (size_t i = 0; i < (sizeof(cmds) / sizeof(cmds[0])); ++i)
94          cmds[i].print();
95  }
96  bool eval(Stock& stock, MultiInputHelper& ih) {
97      ih.getLine("> ");
98      auto s = ih.readStringView("> ");
```

```
    if (s.size() == 1) {
        for (size_t i = 0; i < (sizeof(cmds) / sizeof(cmds[0])); ++i) {
            if (cmds[i].shortName == toupper(s[0]))
                return cmds[i].f(stock, ih);
        }
    }
    for (size_t i = 0; i < (sizeof(cmds) /sizeof(cmds[0])); ++i) {
        if (cmds[i].name == s) return cmds[i].f(stock, ih);
    }
    std::cout << "Invalid command!\n";
    printHelp();
    std::cout << "!";
    return true;
}

int main() {
    Stock stock;
    printHelp();
    MultiInputHelper ih;
    for (;;) {
        if (!eval(stock, ih)) break;
    }
    return 0;
}
```

fixedPoint.h

```
#pragma once
#include "utils.h"
#include <iostream>

constexpr char decimalSeparator = '.';

constexpr long pow(long b, long e) {
    long res = 1;
    while (--e >= 0) res *= b;
    return res;
}
template<size_t decimals = 2>
class FixedPoint {
    static_assert(decimals < 18);
    static constexpr long factor = pow(10, decimals);
    int64_t val;
    using FP = FixedPoint<decimals>;

public:
    static constexpr FP make(int64_t v) {
        FP r;
        r.val = v;
        return r;
```

```
    }
    constexpr FixedPoint(long double v) : val(v * factor) {}
    constexpr FixedPoint(double v) : val(v * factor) {}
    constexpr FixedPoint(float v) : val(v * factor) {}
    constexpr FixedPoint(long v) : val(v * factor) {}
    constexpr FixedPoint(int v) : val(v * factor) {}
    constexpr FixedPoint() : val(0) {}

    constexpr FP operator+(FP rhs) const { return make(val + rhs.val); }
    constexpr FP operator-(FP rhs) const { return make(val - rhs.val); }
    constexpr FP operator+() const { return *this; }
    constexpr FP operator-() const { return make(-val); }
    constexpr FP operator*(FP rhs) const {
        return make((val * rhs.val) / factor);
    }
    constexpr FP operator/(FP rhs) const {
        return make((val * factor) / rhs.val);
    }

    constexpr FP& operator+=(FP rhs) { return *this = *this + rhs; }
    constexpr FP& operator-=(FP rhs) { return *this = *this - rhs; }
    constexpr FP& operator/=(FP rhs) { return *this = *this / rhs; }
    constexpr FP& operator*=(FP rhs) { return *this = *this * rhs; }

    constexpr bool operator< (FP rhs) const { return val < rhs.val; }
    constexpr bool operator> (FP rhs) const { return val > rhs.val; }
    constexpr bool operator<=(FP rhs) const { return val <= rhs.val; }
    constexpr bool operator>=(FP rhs) const { return val >= rhs.val; }
    constexpr size_t textLen(bool showSign = false) const {
        size_t baseline = 1 + decimals + showSign; // 1 for the dot
        if (val < 0) return make(-val).textLen(true);
        // a int64_t can only store 19 digits
        if (val < factor * 1) return baseline + 0;
        if (val < factor * 10) return baseline + 1;
        if constexpr (decimals <= 16)
        if (val < factor * 100) return baseline + 2;
        if constexpr (decimals <= 15)
        if (val < factor * 1000) return baseline + 3;
        if constexpr (decimals <= 14)
        if (val < factor * 10000) return baseline + 4;
        if constexpr (decimals <= 13)
        if (val < factor * 100000) return baseline + 5;
        if constexpr (decimals <= 12)
        if (val < factor * 1000000) return baseline + 6;
        if constexpr (decimals <= 11)
        if (val < factor * 10000000) return baseline + 7;
        if constexpr (decimals <= 10)
        if (val < factor * 100000000) return baseline + 8;
        if constexpr (decimals <= 9)
        if (val < factor * 1000000000) return baseline + 9;
```

```
        if constexpr (decimals <= 8)
        if (val < factor * 10000000000) return baseline + 10;
        if constexpr (decimals <= 7)
        if (val < factor * 100000000000) return baseline + 11;
        if constexpr (decimals <= 6)
        if (val < factor * 1000000000000) return baseline + 12;
        if constexpr (decimals <= 5)
        if (val < factor * 10000000000000) return baseline + 13;
        if constexpr (decimals <= 4)
        if (val < factor * 100000000000000) return baseline + 14;
        if constexpr (decimals <= 3)
        if (val < factor * 1000000000000000) return baseline + 15;
        if constexpr (decimals <= 2)
        if (val < factor * 10000000000000000) return baseline + 16;
        if constexpr (decimals <= 1)
        if (val < factor * 100000000000000000) return baseline + 17;
        if constexpr (decimals <= 0)
        if (val < factor * 1000000000000000000) return baseline + 18;
        return baseline + (19 - decimals);
    }

    //a buffer of size at most 22 is needed
    //(19 for digits, 1 for the dot, 1 for sign, and 1 for \0)
    static constexpr size_t MaxBuffSize = 22;

    // returns the string length (no \0)
    constexpr size_t toString(char* buf, bool showSign) const {
        auto len = textLen(showSign);
        char* p = buf + len;
        auto i = val;
        if (val < 0) {
            *buf = '-';
            i = -val;
        }
        else if (showSign) *buf = '+';
        *p-- = '\0';
        char* decimalPoint = p - decimals;
        for (; p != decimalPoint; --p) {
            *p = (i % 10) + '0';
            i /= 10;
        }
        *p = decimalSeparator;
        while (i) {
            *(--p) = (i % 10) + '0';
            i /= 10;
        }
        return len;
    }
    static constexpr auto isDigit(char c) {
        return c >= '0' && c <= '9';
```

```
124     }
125     // if the return value is nullptr it means we didn't read anything good
126     //else we return a pointer to the end of the read FixedPoint
127     static constexpr const char* fromString(const char* str, FP& res) {
128         long sign = 1;
129         if (*str == '+') ++str;
130         else if (*str == '-') { ++str; sign = -1; }
131         auto p = str;
132 
133         res.val = 0;
134         while (isDigit(*p)) {
135             res.val *= 10;
136             res.val += *p - '0';
137             ++p;
138         }
139         if (*p == '.' || *p == decimalSeparator) {
140             ++p;
141             if (str == p-1)
142                 str = p;
143             [&] {
144                 size_t i = 0;
145                 for (; i < decimals; ++i) {
146                     if (!isDigit(*p)) {
147                         for (;i < decimals; ++i) res.val *= 10;
148                         return;
149                     }
150                     res.val *= 10;
151                     res.val += *p - '0';
152                     ++p;
153                 }
154                 if (isDigit(*p)) {
155                     if (*p - '0' >= 5) ++res.val;
156                     ++p;
157                 }
158                 while (isDigit(*p)) ++p;
159             }();
160         } else {
161             res.val *= factor;
162         }
163         if (p == str) { return nullptr; std::cout << "NULL"; }
164         res.val *= sign;
165         return p;
166     }
167     static constexpr auto fromString(const char* str) {
168         struct {
169             FP res;
170             const char* str;
171         } res;
172         res.str = fromString(str, res.res);
173         return res;
```

```
    }

    friend std::ostream& operator<<(std::ostream& s, FP fp) {
        char buf[MaxBuffSize];
        fp.toString(buf, s.flags() & s.showpos);
        return s << buf;
    }

    friend std::istream& operator>>(std::istream& s, FP& fp) {
        long double ld;
        s >> ld;
        fp = ld;
        return s;
    }
};
```

inputHelper.h

```
#pragma once
#include "fixedPoint.h"
#include <iostream>

class MultiInputHelper {
public:
    std::string line;
    const char* p;

public:
    MultiInputHelper() {}
    void getLine(std::string_view msg = "") {
        if (msg!=""){
            std::cout << msg;
            if (msg.back() != ' ') std::cout << ": ";
        }
        std::getline(std::cin, line);
        p = line.c_str();
    }

    void getLineAfterInvalid(std::string_view msg) {
        std::cout << "!";
        getLine(msg);
    }
    void readName(std::string& res, std::string_view msg) {
        eatWhiteSpace(msg);
        if (*p == '"') {
            const char* beg = ++p;
            res = "";
            for (;;) {
                if (!*p) {
                    getLine/*AfterInvalid*/(msg);
```

```
                    return readName(res, msg);
                } else if (*p++ == '"') {
                    if (*(p-2) == '\\') {
                        res += std::string(beg, p-beg-2) + '"';
                        beg = p;
                    } else {
                        res += std::string(beg, p-beg-1);
                        return;
                    }
                }
            }
        } else {
            readString(res, msg);
        }
    }

    void readString(std::string& res, std::string_view msg) {
        eatWhiteSpace(msg);
        const char* beg = p;
        while (*p && !isspace(*p)) {
            ++p;
        }
        auto sz = beg-p;
        if (sz == 0) {
            getLine/*AfterInvalid*/(msg);
            return readString(res, msg);
        } else {
            res = std::string(beg, p - beg);
        }
    }
    char readChar(std::string_view msg = "") {
        eatWhiteSpace(msg);
        return *p++;
    }

    std::string_view readStringView(std::string_view msg) {
        eatWhiteSpace(msg);
        const char* beg = p;
        while (*p && !isspace(*p)) ++p;

        auto sz = beg-p;
        if (sz == 0) {
            getLineAfterInvalid(msg);
            return readStringView(msg);
        } else {
            return std::string_view(beg, p - beg);
        }
    }
    template<size_t precision>
    void readFP(FixedPoint<precision>& res, std::string_view msg) {
```

```
83          eatWhiteSpace(msg);
84          auto str = FixedPoint<precision>::fromString(p, res);
85          if (str == nullptr) {
86              getLineAfterInvalid(msg);
87              readFP(res, msg);
88              return;
89          } else {
90              p = str;
91          }
92      }
93
94  private:
95      void eatWhiteSpace(std::string_view message) {
96          for (;;) {
97              if (*p == 0) getLine(message);
98              if (isspace(*p)) ++p;
99              else return;
100         }
101     }
102 };
```