

Cuprins

Fișiere comune	2
Alocarea dinamica a memoriei. Tipuri specifice.	16
Laborator 1	16
Laborator 2	19
Tablouri	22
Laborator 3	22
Liste liniare simplu înlănțuite. Stive și cozi	25
Laborator 4	25
Laborator 5	26

Fișiere comune

utils.h

```
1  #pragma once
2  #include <initializer_list>
3  #include <iostream>
4
5  constexpr size_t getSize(std::initializer_list<double> l) {
6      size_t n = 0;
7      auto it = l.begin();
8      auto end = l.end();
9      while (it++ != end) ++n;
10     return n;
11 }
12
13 constexpr void getSize(std::initializer_list<
14     std::initializer_list<double>> list,
15     int& s1, int& s2) {
16     constexpr int invalid = -1;
17     s1 = 0;
18     s2 = invalid;
19
20     for (const auto& l: list) {
21         int i = getSize(l);
22         ++s1;
23         if (s2 != invalid && i != s2) {
24             //std::cerr << i << "!=" << s2 << "\n";
25             throw std::logic_error("Fixed line size expected");
26         }
27         s2 = i;
28     }
29     //std::cerr << "size: " << s1 << ", " << s2 << "\n";
30 }
31 constexpr int MAX_SZ = 256;
32 inline size_t readSize(const char* name, int a = 1, int b = MAX_SZ) {
33     int res;
34     do {
35         std::cout << name << ": ";
36         std::cin >> res;
37     } while (res < a || res > b);
38     return res;
39 }
```

```

40
41
42 inline void assert(bool cond, const char* msg) {
43     if (!cond) throw std::logic_error(msg);
44 }
45 template<typename T>
46 inline T read(const char* name) {
47     T res;
48     std::cout << name << ": ";
49     std::cin >> res;
50     return res;
51 }

```

matrix.h

```

1  #pragma once
2  #include "utils.h"
3
4  #include <iostream>
5  #include <cmath>
6  #include <cstring>
7  #include <utility>
8
9  struct Mat {
10     int m, n;
11     double *data;
12     Mat() : m(0), n(0), data(nullptr) {}
13     Mat(int m, int n) : m(m), n(n) { data = new double[m*n]; }
14     //template<int m, int n>
15     Mat(std::initializer_list<std::initializer_list<double>> list) {
16         getSize(list, m, n);
17         data = new double[m*n];
18
19         auto it = begin();
20         for (const auto& l : list) {
21             for (const auto& v : l) *(it++) = v;
22         }
23     }
24     Mat(Mat&& rhs) noexcept :
25         m(rhs.m), n(rhs.n),
26         data(std::exchange(rhs.data, nullptr)) {
27         //rhs.n = rhs.m = 0;
28     }
29
30     Mat(const Mat& rhs) : Mat(rhs.m, rhs.n) {
31         std::memcpy(data, rhs.data, sizeof(double) *m*n);
32     }
33     Mat& operator=(const Mat& rhs) {
34         setSize(rhs.m, rhs.n);
35         std::memcpy(data, rhs.data, sizeof(double) *m*n);

```

```

36         return *this;
37     }
38     Mat& operator=(Mat&& rhs) noexcept {
39         this->~Mat();
40         m = rhs.m;
41         n = rhs.n;
42         data = std::exchange(rhs.data, nullptr);
43
44         return *this;
45     }
46
47     ~Mat() {
48         delete[] data;
49     }
50     double& at(int i, int j) {
51         assert(i < m && j < n, "out of range");
52         return data[i * n + j];
53     }
54     double at(int i, int j) const { return data[i * n + j]; }
55     double* begin() { return data; }
56     const double* begin() const { return data; }
57     double* end() { return data + m*n; }
58     const double* end() const { return data + m*n; }
59
60     void setSize(int m, int n) {
61         if (this->m == m && this->n == n) return;
62         this->~Mat();
63         new (this) Mat(m, n);
64     }
65
66     static Mat read() {
67         Mat res(readSize("m"), readSize("n"));
68         for (auto& i : res)
69             std::cin >> i;
70         return res;
71     }
72     void print(const char* name) const {
73         std::cout << name << " = Mat " << m << "x" << n << "{\n";
74
75         for (int i = 0; i < m; ++i) {
76             for (int j = 0; j < n; ++j) {
77                 std::cout << at(i, j) << " ";
78             }
79             std::cout << "\n";
80         }
81         std::cout << "}\n";
82     }
83
84     friend std::ostream& operator<<(std::ostream& s, Mat& m) {
85         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) {

```

```

136         double x = 0;
137         for (int i = 0; i < sz2; ++i) {
138             x += std::abs(type == Type::Col? at(j, i) : at(i, j));
139         }
140         if (x > max) max = x;
141     }
142     return max;
143 }
144 };
145
146 Mat& add(const Mat& a, const Mat& b, Mat& res) {
147     assert(a.m == b.m && a.n == b.n, "Sizes don't match, can't add");
148     res.setSize(a.m, a.n);
149     for (int i = 0; i < a.m; ++i)
150         for (int j = 0; j < a.n; ++j)
151             res.at(i, j) = a.at(i, j) + b.at(i, j);
152     return res;
153 }
154
155 Mat& mul(double a, const Mat& b, Mat& res) {
156     res.setSize(b.m, b.n);
157
158     for (int i = 0; i < res.m; ++i)
159         for (int j = 0; j < res.n; ++j)
160             res.at(i, j) = a * b.at(i, j);
161     return res;
162 }
163
164 Mat& neg(const Mat& a, Mat& res) {
165     return mul(-1, a, res);
166 }
167
168 Mat& sub(const Mat& a, const Mat& b, Mat& res) {
169     return add(a, neg(b, res), res);
170 }
171
172 Mat& mul(const Mat& a, const Mat& b, Mat& res) {
173     assert(a.n == b.m, "Sizes don't match, can't multiply");
174     res.setSize(a.m, b.n);
175
176     for (int i = 0; i < res.m; ++i)
177         for (int j = 0; j < res.n; ++j) {
178             res.at(i, j) = 0;
179             for (int k = 0; k < a.n; ++k)
180                 res.at(i, j) += a.at(i, k) * b.at(k, j);
181         }
182     return res;
183 }
184
185 Mat& trans(const Mat& a, Mat& res) {
186     assert(a.data != res.data, "Can't transpose inplace");
187     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();

```

```

42     _begin = std::exchange(rhs._begin, nullptr);
43     _end = std::exchange(rhs._end, nullptr);
44     return *this;
45 }
46
47 ~Vec() { delete[] _begin; }
48
49 constexpr size_t size() const { return _end - _begin; }
50
51 constexpr double& operator[](size_t i) { return _begin[i]; }
52 constexpr double operator[](size_t i) const { return _begin[i]; }
53
54 void setSize(size_t n) {
55     if (size() == n) return;
56     *this = Vec(n);
57 }
58
59 friend std::ostream& operator<<(std::ostream& s, const Vec& v) {
60     s << "(";
61     double* it = v._begin;
62     for (double* end = v._end - 1; it < end; ++it)
63         s << *it << ", ";
64
65     if (it < v._end) s << *it;
66
67     return s << ")";
68 }
69 static Vec read() {
70     Vec res(readSize("n"));
71     for (auto& v: res) std::cin >> v;
72     return res;
73 }
74 double norm() const;
75 };
76 void assertSizes(const Vec& a, const Vec& b) {
77     assert(a.size() == b.size(), "Sizes don't match");
78 }
79
80 Vec& add(const Vec& a, const Vec& b, Vec& res) {
81     assertSizes(a, b);
82     res.setSize(a.size());
83     auto aIt = a.begin();
84     auto bIt = b.begin();
85     for (auto& v: res) v = *(aIt++) + *(bIt++);
86     return res;
87 }
88
89 Vec& mul(double a, const Vec& b, Vec& res) {
90     res.setSize(b.size());
91     auto it = b.begin();

```



```

92     for (auto& v : res) v = a * (*(it++));
93     return res;
94 }
95
96 Vec& neg(const Vec& b, Vec& res) { return mul(-1, b, res); }
97
98 Vec& sub(const Vec& a, const Vec& b, Vec& res) {
99     return add(a, neg(b, res), res);
100 }
101 double dot(const Vec& a, const Vec& b) {
102     double res = 0;
103     assertSizes(a, b);
104     auto bIt = b.begin();
105     for (auto& v : a) res += v * (*(bIt++));
106     return res;
107 }
108 double norm(const Vec& a) {
109     return std::sqrt(dot(a, a));
110 }
111 double Vec::norm() const {
112     return ::norm(*this);
113 }

```

stack.h

```

1  #pragma once
2  #include "utils.h"
3
4  template<typename T>
5  struct Stack {
6      struct Node {
7          T val;
8          Node* next;
9      };
10
11      Stack(const Stack&) = delete;
12      Stack& operator=(const Stack&) = delete;
13      Node* top;
14      Stack(Node* top = nullptr) : top(top) {}
15      ~Stack() {
16          while (top) {
17              Node* n = top;
18              top = top->next;
19              delete n;
20          }
21      }
22      void push(T val) {
23          Node *n = new Node{ val, top };
24          top = n;
25      }

```

```

26 void quickPop() {
27     Node* n = top;
28     top = top->next;
29     delete n;
30 }
31 T pop() {
32     assert(top, "Empty stack");
33     Node* n = top;
34     top = top->next;
35     T res = n->val;
36     delete n;
37     return res;
38 }
39 bool empty() const { return top == nullptr; }
40 friend std::ostream& operator<<(std::ostream& s, const Stack& st) {
41     if (st.empty()) return s << "{}";
42     s << "{";
43     const Node* it = st.top;
44     while (it->next != nullptr) {
45         s << it->val << ", ";
46         it = it->next;
47     }
48
49     return s << it->val << "}";
50 }
51 };

```

queue.h

```

1  #pragma once
2  #include "utils.h"
3
4  template<class T>
5  struct Queue {
6      struct Node {
7          T val;
8          Node* next;
9      };
10     Node* top;
11     Node* bot;
12     Queue(Node* top = nullptr, Node* bot = nullptr) : top(top), bot(bot) {}
13     /*
14     Queue(const Queue&) = delete;
15     Queue& operator=(const Queue&) = delete;
16     ~Queue() {
17         while (top) {
18             Node* n = top;
19             top = top->next;
20             delete n;
21         }

```

```

22     */
23 void push(T val) {
24     Node *n = new Node{ val, nullptr };
25     if (bot == nullptr) {
26         top = n;
27     }
28     else {
29         bot->next = n;
30     }
31     bot = n;
32 }
33
34 T pop() {
35     assert(top, "Empty queue");
36     Node* old = top;
37     T res = old->val;
38     top = top->next;
39     if (top == nullptr)
40         bot = nullptr;
41     delete old;
42     return res;
43 }
44 bool empty() const { return top == nullptr; }
45 static Queue read(const char* msg) {
46     Queue q;
47     std::cout << msg << ":\n";
48     int len1 = readSize("n");
49     for (int i = 0; i < len1; ++i) {
50         T s;
51         std::cin >> s;
52         q.push(s);
53     }
54     return std::move(q);
55 }
56
57 friend std::ostream& operator<<(std::ostream& s, const Queue& st) {
58     if (st.empty()) return s << "{}";
59     s << "{";
60     const Node* it = st.top;
61     while (it->next != nullptr) {
62         s << it->val << ", ";
63         it = it->next;
64     }
65
66     return s << it->val << "}";
67 }
68 };

```

list.h

```

1  #pragma once
2  #include "utils.h"
3
4  #include <type_traits>
5  #include <utility>
6
7
8  // We should never check for (bot == nullptr),
9  // so we don't update it when the list becomes empty.
10 template<typename T>
11 struct List {
12     struct Node {
13         T val;
14         Node* next;
15     };
16     struct It {
17         const Node* n;
18         constexpr It(const Node* n) : n(n) {}
19         constexpr It& operator++() {
20             n = n->next;
21             return *this;
22         }
23         constexpr auto& operator*() { return n->val; }
24         constexpr bool operator==(const It& rhs) { return n == rhs.n; }
25         constexpr bool operator!=(const It& rhs) { return n != rhs.n; }
26     };
27     Node* top;
28     Node* bot;
29     constexpr List(Node* top = nullptr, Node* bot = nullptr)
30         : top(top), bot(bot) {}
31     List(const List&) = delete;
32     List& operator=(const List&) = delete;
33
34     constexpr It begin() const { return top; }
35     constexpr It end() const { return nullptr; }
36
37     ~List() {
38         while (top) {
39             Node* n = top;
40             top = top->next;
41             delete n;
42         }
43     }
44     List(List&& rhs) noexcept
45         : top(std::exchange(rhs.top, nullptr)),
46           bot(std::exchange(rhs.bot, nullptr)) {}
47     List& operator=(List&& rhs) noexcept {
48         this->~List();

```

```

49         this->top = std::exchange(rhs.top, nullptr);
50         this->bot = std::exchange(rhs.bot, nullptr);
51     }
52
53     T& first() { return top->val; }
54     T& last() { return bot->val; }
55     T pop_front() {
56         assert(top, "Empty list");
57         Node* n = top;
58         top = top->next;
59         T res = n->val;
60         delete n;
61         return res;
62     }
63
64     void push_front(T val) {
65         Node *n = new Node{ val, top };
66         top = n;
67     }
68     void push_back(T val) {
69         Node *n = new Node{ val, nullptr };
70         if (top == nullptr) {
71             top = n;
72         }
73         else {
74             bot->next = n;
75
76         }
77         bot = n;
78     }
79     bool empty() const { return top == nullptr; }
80
81     bool operator==(const List& rhs) {
82         for (auto it = top, rit = rhs.top;
83              it != nullptr;
84              it = it->next, rit = rit->next) {
85             if (it->val != rit->val) return false;
86         }
87         return true;
88     }
89
90     static List read(const char* msg) {
91         List q;
92         std::cout << msg << ":\n";
93         int len1 = readSize("n");
94         for (int i = 0; i < len1; ++i) {
95             T s;
96             std::cin >> s;
97             q.push_front(s);
98         }

```

```

99         return std::move(q);
100     }
101
102     friend std::ostream& operator<<(std::ostream& s, const List& st) {
103         if constexpr (std::is_same_v<T, char>) {
104             const Node* it = st.top;
105             while (it != nullptr) {
106                 s << it->val;
107                 it = it->next;
108             }
109             return s;
110         }
111         else {
112             if (st.empty()) return s << "{}";
113             s << "{";
114             const Node* it = st.top;
115             while (it->next != nullptr) {
116                 s << it->val << ", ";
117                 it = it->next;
118             }
119
120             return s << it->val << "}";
121         }
122     }
123
124     void remove(T& val) {
125         remove_if([&] (T& t) { return t == val; });
126     }
127     // removes all elements that satisfy p
128     template<class P>
129     void remove_if(P p) {
130         apply_on(p, [] (Node* n) { delete n; } );
131     }
132     //applies f() on all nodes that satisfy the predicate p()
133     template<class P, class F>
134     void apply_on(P p, F f) {
135         for (;;) {
136             auto* n = top;
137             if (!n) {
138                 return;
139             }
140             if (!p(n->val)) break;
141             top = n->next;
142             f(n);
143         }
144         auto* prev = top;
145         auto* it = prev->next;
146
147         while (it) {
148             if (p(it->val)) {

```

```

149         prev->next = it->next;
150         if (prev->next == nullptr) {
151             bot = prev;
152         }
153         f(it);
154         it = prev;
155     } else {
156         prev = it;
157         it = it->next;
158     }
159 }
160 }
161 // P is a predicate on T
162 template<typename P>
163 Node* find(P p) {
164     auto n = top;
165     for (; n; n = n->next) {
166         if (p(n->val))
167             return n;
168     }
169     return n;
170 }
171 };
172
173 // keeps in l all the elements that don't satisfy the predicate p
174 // and returns pair of:
175 // - a reference to the original list
176 // - a list containing the elements that satisfy p
177 template<typename T, typename P>
178 constexpr auto partition_split(List<T>& l, P p) {
179     struct res_t {
180         List<T>& notSatisfying;
181         List<T> satisfying;
182     } res = { l, {} };
183     List<T>& sat = res.satisfying;
184     auto insertNode = [&sat](auto* n) {
185         n->next = sat.top;
186         sat.top = n;
187         if (sat.bot == nullptr) {
188             sat.bot = n;
189         }
190     };
191     l.apply_on(p, insertNode);
192     return res;
193 }

```

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 specifice¹, 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 linii² sau pe coloane³. Se vor folosi tablouri bidimensionale alocate static.

```
1  #include "utils.h"
2
3  #include <iostream>
4  #include <cmath>
5
6  struct Mat {
7      double data[MAX_SZ][MAX_SZ] {};
8      int m, n;
9
10     Mat() : m(0), n(0) {}
11     Mat(int m, int n) : m(m), n(n) {}
12
13     static Mat read() {
14         Mat res(readSize("m"), readSize("n"));
15
16         for (int i = 0; i < res.m; ++i)
17             for (int j = 0; j < res.n; ++j)
18                 std::cin >> res.data[i][j];
19         return res;
20     }
21     void setSize(int m, int n) {
22         this->m = m;
23         this->n = n;
24     }
25 }
```

¹Dacă $A \in \mathcal{M}_{m \times n}(\mathbb{R})$, atunci $\|A\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^m |a_{ij}|$, $\|A\|_\infty = \max_{1 \leq i \leq m} \sum_{j=1}^n |a_{ij}|$, $\|A\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n a_{ij}^2}$.

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

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


```

25     double& at(int i, int j) { return data[i][j]; }
26     double at(int i, int j) const { return data[i][j]; }
27
28     void print(const char* name) const {
29         std::cout << name << " = Mat " << m << "x" << n << "{\n";
30         for (int i = 0; i < m; ++i) {
31             for (int j = 0; j < n; ++j)
32                 std::cout << at(i, j) << " ";
33             std::cout << "\n";
34         }
35         std::cout << "}\n";
36     }
37 private:
38     enum class Type { Row, Col };
39     template<Type type>
40     bool isStrictlyDiagonallyDominantImpl() const {
41         assert(m == n, "Matrix must be square");
42         for (int i = 0; i < m; ++i) {
43             double val = std::abs(at(i, i));
44             double sum = -val;
45             for (int j = 0; j < m; ++j)
46                 sum += std::abs(type == Type::Col ? at(j, i) : at(i, j));
47             if (sum >= val) return false;
48         }
49         return true;
50     }
51
52     template<Type type>
53     double normImpl(int sz1, int sz2) const {
54         double max = -1;
55         for (int j = 0; j < sz1; ++j) {
56             double x = 0;
57             for (int i = 0; i < sz2; ++i)
58                 x += std::abs(type == Type::Col ? at(j, i) : at(i, j));
59             if (x > max) max = x;
60         }
61         return max;
62     }
63 public:
64     double norm1() const { return normImpl<Type::Row>(n, m); }
65     double normInf() const { return normImpl<Type::Col>(m, n); }
66     double normF() const {
67         double res = 0;
68         for (int i = 0; i < m; ++i)
69             for (int j = 0; j < n; ++j)
70                 res += at(i, j) * at(i, j);
71
72         return std::sqrt(res);
73     }
74     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 tastatură a celor n componente ale unui vector, afișarea componentelor vectorului sub forma unui n -uplu de elemente. Se vor folosi tablouri unidimensionale alocate dinamic.

```
1  #include "utils.h"
2
3  #include <iostream>
4  #include <utility>
5  #include <cmath>
6
7  struct Vec {
8      double *_begin, *_end;
9
10     constexpr double* begin() { return _begin; }
11     constexpr const double* begin() const { return _begin; }
12
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&) = delete;
24     Vec(Vec&& rhs) noexcept
25         : _begin(std::exchange(rhs._begin, nullptr)),
26         _end(std::exchange(rhs._end, nullptr)) {}
27
28     Vec& operator=(const Vec&) = delete;
29     Vec& operator=(Vec&& rhs) noexcept {
30         this->~Vec();
31         _begin = std::exchange(rhs._begin, nullptr);
32         _end = std::exchange(rhs._end, nullptr);
33         return *this;
34     }
35
36     ~Vec() { delete[] _begin; }
37
38     constexpr size_t size() const { return _end - _begin; }
39
40     constexpr double& operator[](size_t i) { return _begin[i]; }
41     constexpr double operator[](size_t i) const { return _begin[i]; }
42
43     void setSize(size_t n) {
```

```

44         if (size() == n) return;
45         *this = Vec(n);
46     }
47
48     friend std::ostream& operator<<(std::ostream& s, const Vec& v) {
49         s << "(";
50         double* it = v._begin;
51         for (double* end = v._end - 1; it < end; ++it)
52             s << *it << ", ";
53
54         if (it < v._end) s << *it;
55
56         return s << ")";
57     }
58     static Vec read() {
59         Vec res(readSize("n"));
60         for (auto& v: res) std::cin >> v;
61         return res;
62     }
63     double norm() const;
64 };
65 void assertSizes(const Vec& a, const Vec& b) {
66     assert(a.size() == b.size(), "Sizes don't match");
67 }
68
69 Vec& add(const Vec& a, const Vec& b, Vec& res) {
70     assertSizes(a, b);
71     res.setSize(a.size());
72     auto aIt = a.begin();
73     auto bIt = b.begin();
74     for (auto& v : res) v = *(aIt++) + *(bIt++);
75     return res;
76 }
77
78 Vec& mul(double a, const Vec& b, Vec& res) {
79     res.setSize(b.size());
80     auto it = b.begin();
81     for (auto& v : res) v = a * (*(it++));
82     return res;
83 }
84
85 Vec& neg(const Vec& b, Vec& res) { return mul(-1, b, res); }
86
87 Vec& sub(const Vec& a, const Vec& b, Vec& res) {
88     return add(a, neg(b, res), res);
89 }
90 double dot(const Vec& a, const Vec& b) {
91     double res = 0;
92     assertSizes(a, b);
93     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.

$$\left\{ \begin{array}{l} 2x_1 - x_2 = 1 \\ -x_1 + 2x_2 - x_3 = 1 \\ -x_2 + 2x_3 - x_4 = 1 \\ \dots\dots\dots \\ -x_{n-2} + 2x_{n-1} - x_n = 1 \\ -x_{n-1} + 2x_n = 1, \quad n \in \mathbb{N}, 2 \leq n \leq 50 \end{array} \right. .$$

```
1  #include "utils.h"
2
3  #include "vector.h"
4  #include "matrix.h"
5
6  #include <iostream>
7  #include <cmath>
8
9  constexpr double eps = 1e-7;
10
11 Vec& mul(const Mat& m, const Vec& v, Vec& res) {
12     assert(v.begin() != res.begin(), "Can't multiply inplace");
13     assert(v.size() == size_t(m.n), "Sizes don't match");
14     res.setSize(m.m);
15
16
17     for (size_t i = 0; i < res.size(); ++i) {
18         res[i] = 0;
19         for (size_t k = 0; k < v.size(); ++k)
20             res[i] += m.at(i, k) * v[k];
21     }
22     return res;
23 }
24
25 // A * X = b
26 struct System {
27     Mat A;
```

```

28 Vec b;
29
30 System(int n, int m) : A(n, m), b(n) {}
31
32 System(std::initializer_list<std::initializer_list<double>> A,
33         std::initializer_list<double> b) : A(A), b(b) {
34     assert(std::size_t(this->A.m) == this->b.size(),
35            "sizes don't match");
36 }
37
38 friend std::ostream& operator<<(std::ostream& s, const System& sys) {
39     s << "System " << sys.A.m << "x" << sys.A.n << ": \n";
40     auto& A = sys.A;
41     for (int i = 0; i < A.m; ++i) {
42         s << "{";
43         for (int j = 0; j < A.n; ++j) {
44             //showpos shows a '+' in front of positive numbers
45             if (std::abs(A.at(i, j)) > eps)
46                 s << std::showpos << A.at(i, j)
47                 << std::noshowpos << "*x" << (j+1) << " ";
48         }
49         s<< "= " << sys.b[i] << "\n";
50     }
51
52     return s;
53 }
54
55 // L_i += f * L_j
56 void addLines(int i, double f, int j) {
57     for (int k = 0; k < A.n; ++k) {
58         A.at(i, k) += f * A.at(j, k);
59     }
60     b[i] += f * b[j];
61 }
62
63 // L_i *= f
64 void multiplyLine(int i, double f) {
65     for (int k = 0; k < A.n; ++k) {
66         A.at(i, k) *= f;
67     }
68     b[i] *= f;
69 }
70
71 Vec solveTriangulated() {
72     for (int i = A.m-1; i > 0; --i) {
73         addLines(i-1, -A.at(i-1, i), i);
74         multiplyLine(i, 1 / A.at(i, i));
75     }
76     return b;
77 }

```

```

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 corespunzătoare până la întâlnirea vocalei citite.

```
1  #include "utils.h"
2  #include "stack.h"
3
4  #include <iostream>
5  #include <cstdlib>
6
7  bool isVowel(char c) {
8      c = tolower(c);
9      return c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u';
10 }
11
12 int main() {
13     std::string str;
14     std::cout << "str: ";
15     std::getline(std::cin, str);
16
17     char v;
18     do {
19         v = read<char>("vowel");
20     } while (!isVowel(v));
21
22     Stack<char> lower;
23     Stack<char> upper;
24     auto printStacks = [&] (const char* s) {
25         std::cout << s;
26         std::cout << "lower:" << lower << "\n";
27         std::cout << "upper:" << upper << "\n";
28     };
29
30     printStacks("Before:\n");
31     for (auto& c : str) {
32         if (islower(c)) lower.push(c);
33         else if (isupper(c)) upper.push(c);
```

```

34     }
35
36     printStacks("After Adding:\n");
37     Stack<char>& stack = isupper(v)? upper: lower;
38     while (!stack.empty()) {
39         if (char c = stack.pop(); c == v) {
40             break;
41         }
42     }
43     printStacks("Result:\n");
44     return 0;
45 }

```

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 corespunzătoare 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 apare în listă.

```

1  #include "utils.h"
2  #include "list.h"
3
4  #include <iostream>
5
6  int main() {
7      auto nums = List<int>::read("numbers");
8
9      auto [evens, odds] = partition_split(nums, [](int n) {return n % 2;});
10     std::cout << "odd:  " << odds << "\n";
11     std::cout << "even: " << evens << "\n";
12
13     int x;
14     do {
15         std::cout << "x (must be even): ";
16         std::cin >> x;
17     } while (x % 2);
18
19     std::cout << "removing...\n";
20     evens.remove(x);
21
22     std::cout << "even: " << evens << "\n";
23     return 0;
24 }

```

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 produs nou, eliminarea unui produs în 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.

fixedPoint.h

```

1  #pragma once
2  #include "utils.h"
3  #include <iostream>
4
5  constexpr char decimalSeparator = '.';
6
7  constexpr long pow(long b, long e) {
8      long res = 1;
9      while (--e >= 0) res *= b;
10     return res;
11 }
12 template<size_t decimals = 2>
13 class FixedPoint {
14     static_assert(decimals < 18);
15     static constexpr long factor = pow(10, decimals);
16     int64_t val;
17     using FP = FixedPoint<decimals>;
18
19 public:
20     static constexpr FP make(int64_t v) {
21         FP r;
22         r.val = v;
23         return r;
24     }
25     constexpr FixedPoint(long double v) : val(v * factor) {}
26     constexpr FixedPoint(double v) : val(v * factor) {}
27     constexpr FixedPoint(float v) : val(v * factor) {}
28     constexpr FixedPoint(long v) : val(v * factor) {}
29     constexpr FixedPoint(int v) : val(v * factor) {}
30     constexpr FixedPoint() : val(0) {}
31
32     constexpr FP operator+(FP rhs) const { return make(val + rhs.val); }
33     constexpr FP operator-(FP rhs) const { return make(val - rhs.val); }
34     constexpr FP operator+() const { return *this; }
35     constexpr FP operator-() const { return make(-val); }
36     constexpr FP operator*(FP rhs) const {
37         return make((val * rhs.val) / factor);
38     }
39     constexpr FP operator/(FP rhs) const {
40         return make((val * factor) / rhs.val);
41     }
42
43     constexpr FP& operator+=(FP rhs) { return *this = *this + rhs; }
44     constexpr FP& operator-=(FP rhs) { return *this = *this - rhs; }
45     constexpr FP& operator/=(FP rhs) { return *this = *this / rhs; }
46     constexpr FP& operator*=(FP rhs) { return *this = *this * rhs; }
47
48     constexpr bool operator< (FP rhs) const { return val < rhs.val; }

```

```

49 constexpr bool operator> (FP rhs) const { return val > rhs.val; }
50 constexpr bool operator<=(FP rhs) const { return val <= rhs.val; }
51 constexpr bool operator>=(FP rhs) const { return val >= rhs.val; }
52 constexpr size_t textLen(bool showSign = false) const {
53     size_t baseline = 1 + decimals + showSign; // 1 for the dot
54     if (val < 0) return make(-val).textLen(true);
55     // a int64_t can only store 19 digits
56     if (val < factor * 1) return baseline + 0;
57     if (val < factor * 10) return baseline + 1;
58     if constexpr (decimals <= 16)
59     if (val < factor * 100) return baseline + 2;
60     if constexpr (decimals <= 15)
61     if (val < factor * 1000) return baseline + 3;
62     if constexpr (decimals <= 14)
63     if (val < factor * 10000) return baseline + 4;
64     if constexpr (decimals <= 13)
65     if (val < factor * 100000) return baseline + 5;
66     if constexpr (decimals <= 12)
67     if (val < factor * 1000000) return baseline + 6;
68     if constexpr (decimals <= 11)
69     if (val < factor * 10000000) return baseline + 7;
70     if constexpr (decimals <= 10)
71     if (val < factor * 100000000) return baseline + 8;
72     if constexpr (decimals <= 9)
73     if (val < factor * 1000000000) return baseline + 9;
74     if constexpr (decimals <= 8)
75     if (val < factor * 10000000000) return baseline + 10;
76     if constexpr (decimals <= 7)
77     if (val < factor * 100000000000) return baseline + 11;
78     if constexpr (decimals <= 6)
79     if (val < factor * 1000000000000) return baseline + 12;
80     if constexpr (decimals <= 5)
81     if (val < factor * 10000000000000) return baseline + 13;
82     if constexpr (decimals <= 4)
83     if (val < factor * 100000000000000) return baseline + 14;
84     if constexpr (decimals <= 3)
85     if (val < factor * 1000000000000000) return baseline + 15;
86     if constexpr (decimals <= 2)
87     if (val < factor * 10000000000000000) return baseline + 16;
88     if constexpr (decimals <= 1)
89     if (val < factor * 100000000000000000) return baseline + 17;
90     if constexpr (decimals <= 0)
91     if (val < factor * 1000000000000000000) return baseline + 18;
92     return baseline + (19 - decimals);
93 }
94
95 //a buffer of size at most 22 is needed
96 //(19 for digits, 1 for the dot, 1 for sign, and 1 for \0)
97 static constexpr size_t MaxBuffSize = 22;
98

```

```

99 // returns the string length (no \0)
100 constexpr size_t toString(char* buf, bool showSign) const {
101     auto len = textLen(showSign);
102     char* p = buf + len;
103     auto i = val;
104     if (val < 0) {
105         *buf = '-';
106         i = -val;
107     }
108     else if (showSign) *buf = '+';
109     *p-- = '\0';
110     char* decimalPoint = p - decimals;
111     for (; p != decimalPoint; --p) {
112         *p = (i % 10) + '0';
113         i /= 10;
114     }
115     *p = decimalSeparator;
116     while (i) {
117         *--p = (i % 10) + '0';
118         i /= 10;
119     }
120     return len;
121 }
122 static constexpr auto isDigit(char c) {
123     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;
174 }
175
176 friend std::ostream& operator<<(std::ostream& s, FP fp) {
177     char buf[MaxBuffSize];
178     fp.toString(buf, s.flags() & s.showpos);
179     return s << buf;
180 }
181
182 friend std::istream& operator>>(std::istream& s, FP& fp) {
183     long double ld;
184     s >> ld;
185     fp = ld;
186     return s;
187 }
188 };

```

inputHelper.h

```

1  #pragma once
2  #include "fixedPoint.h"
3  #include <iostream>
4
5  class MultiInputHelper {
6  public:
7      std::string line;

```

```

8      const char* p;
9
10     public:
11         MultiInputHelper(std::string_view msg = "") {
12             getLine(msg);
13         }
14         void getLine(std::string_view msg = "") {
15             if (msg!=""){
16                 std::cout << msg;
17                 if (msg.back() != ' ') std::cout << ": ";
18             }
19             std::getline(std::cin, line);
20             p = line.c_str();
21         }
22
23         void getLineAfterInvalid(std::string_view msg) {
24             std::cout << "!!";
25             getLine(msg);
26         }
27         void readName(std::string& res, std::string_view msg) {
28             eatWhiteSpace(msg);
29             if (*p == '"') {
30                 const char* beg = ++p;
31                 res = "";
32                 for (;;) {
33                     if (!*p) {
34                         getLine/*AfterInvalid*/(msg);
35                         return readName(res, msg);
36                     } else if (*p++ == '"') {
37                         if (*(p-2) == '\\') {
38                             res += std::string(beg, p-beg-2) + '"';
39                             beg = p;
40                         } else {
41                             res += std::string(beg, p-beg-1);
42                             return;
43                         }
44                     }
45                 }
46             } else {
47                 readString(res, msg);
48             }
49         }
50
51         void readString(std::string& res, std::string_view msg) {
52             eatWhiteSpace(msg);
53             const char* beg = p;
54             while (*p && !isspace(*p)) {
55                 ++p;
56             }
57             auto sz = beg-p;

```

```

58         if (sz == 0) {
59             getLine/*AfterInvalid*/(msg);
60             return readString(res, msg);
61         } else {
62             res = std::string(beg, p - beg);
63         }
64     }
65     char readChar(std::string_view msg = "") {
66         eatWhiteSpace(msg);
67         return *p++;
68     }
69
70     std::string_view readStringView(std::string_view msg) {
71         eatWhiteSpace(msg);
72         const char* beg = p;
73         while (*p && !isspace(*p)) ++p;
74
75         auto sz = beg - p;
76         if (sz == 0) {
77             getLineAfterInvalid(msg);
78             return readStringView(msg);
79         } else {
80             return std::string_view(beg, p - beg);
81         }
82     }
83     template<size_t precision>
84     void readFP(FixedPoint<precision>& res, std::string_view msg) {
85         eatWhiteSpace(msg);
86         auto str = FixedPoint<precision>::fromString(p, res);
87         if (str == nullptr) {
88             getLineAfterInvalid(msg);
89             readFP(res, msg);
90             return;
91         } else {
92             p = str;
93         }
94     }
95
96 private:
97     void eatWhiteSpace(std::string_view message) {
98         for (;;) {
99             if (*p == 0) getLine(message);
100             if (isspace(*p)) ++p;
101             else return;
102         }
103     }
104 };

```


stock.h

```

1  #pragma once
2  #include "fixedPoint.h"
3  #include "list.h"
4
5  #include <iostream>
6  #include <iomanip>
7
8  constexpr size_t precision = 2;
9  using FP = FixedPoint<precision>;
10
11 template<class T>
12 constexpr void setMax(T& a, T b) {
13     if (b > a) a = b;
14 }
15 struct Product {
16     std::string name;
17     FP quantity;
18     std::string unit;
19     FP unitPrice;
20     constexpr auto totalPrice() const {
21         return quantity * unitPrice;
22     }
23 };
24
25 struct Stock {
26     List<Product> products;
27
28     //return value: was the operation was successful
29     bool add(Product p) {
30         auto* res = products.find(
31             [&] (Product& other) { return other.name == p.name; });
32         if (res != nullptr) {
33             auto& v = res->val;
34             std::cout << "Product '" << p.name << "' already exists.\n"
35                 << "Resupplying.\n";
36             if (v.unit != p.unit) {
37                 std::cout << "Units did not match ('"
38                     << v.unit << "' != '" << p.unit << "')\n";
39                 return false;
40             }
41             v.unitPrice += (v.quantity*v.unitPrice + p.quantity*p.unitPrice)
42                 / (v.quantity + p.quantity);
43             v.quantity += p.quantity;
44         } else {
45             products.push_front(p);
46         }
47         return true;
48     }

```

```

49 //return 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     } //TODO: THIS THROWS
57     res->val.quantity += quantity;
58     return true;
59 }
60 //return 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             std::cout << "\n";
139             printLine();
140             std::cout << "| ";
141             printCentered(longest.name, fields[0]);
142             std::cout << " | ";
143             printCentered(longest.quantity + 1 + longest.unit, fields[1]);
144             std::cout << " | ";
145             printCentered(longest.unitPrice, fields[2]);
146             std::cout << " | ";
147             printCentered(longest.totalPrice, fields[3]);
148             std::cout << " |\n";

```

```

149         printLine();
150     }
151     void printFooter(FixedPoint<precision> totalPrice) const {
152         printLine();
153         std::cout << "| ";
154         printPadded(longest.name, "");
155         std::cout << " | ";
156         printPadded(longest.quantity + 1 + longest.unit, "");
157         std::cout << " | ";
158         printPadded(longest.unitPrice, "");
159         std::cout << " | " << std::right;
160         printPadded(longest.totalPrice, totalPrice);
161         std::cout << " |\n";
162         printLine();
163     }
164
165     private:
166         template<typename T>
167         static void printPadded(size_t len, T v) {
168             std::cout << std::setw(len) << v;
169         }
170         static void printCentered(size_t len, std::string_view s) {
171             //we can safely assume that len >= s.len();
172             int total = len - s.size();
173             int left = total / 2;
174             int right = total - left;
175             hline(left, ' ');
176             std::cout << std::setw(0) << s;
177             hline(right, ' ');
178         }
179
180         static void hline(int len, char c) {
181             while (--len >= 0)
182                 std::cout << c;
183         }
184
185         void printLine() const {
186             std::cout << "+";
187             hline(longest.name+2, '-');
188             std::cout << "+";
189             hline(longest.quantity + 3 + longest.unit, '-');
190             std::cout << "+";
191             hline(longest.unitPrice+2, '-');
192             std::cout << "+";
193             hline(longest.totalPrice+2, '-');
194             std::cout << "+\n";
195         }
196     };
197 };

```

stock.cpp

```

1  #include "stock.h"
2
3  #include "inputHelper.h"
4  #include "utils.h"
5  #include "list.h"
6
7  #include <iostream>
8  #include <iomanip>
9
10 struct Command {
11     char shortName;
12     std::string_view name;
13     std::string_view args;
14     bool (*f)(Stock& s, MultiInputHelper& ih); //returns true if should exit
15     std::string_view description;
16     void print() const {
17         std::cout << " " << std::setw(10) << std::left << name
18             << " - " << shortName << " "
19             << std::setw(25) << args << "- " << description << "\n";
20     }
21 };
22 using IH = MultiInputHelper;
23
24 void printHelp();
25 bool printHelp(Stock& s, IH& ih) {
26     printHelp();
27     return true;
28 }
29 bool quit(Stock& s, IH& ih) { return false; }
30 bool print(Stock& s, IH& ih) {
31     s.print();
32     return true;
33 }
34
35 bool add(Stock& s, IH& ih) {
36     Product p;
37     ih.readName(p.name, "name");
38     ih.readFP(p.quantity, "quantity");
39     ih.readString(p.unit, "unit");
40     ih.readFP(p.unitPrice, "unit price");
41     s.add(p);
42     return true;
43 }
44
45 bool sell(Stock& s, IH& ih) {
46     std::string name;
47     FP quantity;
48     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.sell(name, quantity);
62     return true;
63 }
64
65 bool value(Stock& s, IH& 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     //std::getline(std::cin, str);
72     char c;
73     for (;;) {
74         ih.getLine("+ ");
75         //std::cout << "+ ";
76         add(s, ih);
77         ih.getLine("Add more products (y/N)? ");
78         //std::cout << "Add more products (y/N)? ";
79         //std::cin >> c;
80         if (tolower(ih.readChar()) != 'y') break;
81     }
82     return true;
83 }
84 constexpr Command cmds[] = {
85     { 'H', "help", "", printHelp, "Show help" },
86     { 'A', "add", "name quantity unit price", add, "Add product" },
87     { 'I', "init", "", init, "Add multiple elements" },
88     { 'S', "sell", "name quantity", sell, "Sell product" },
89     { 'R', "resupply", "name quantity", resupply, "Resupply product" },
90     { 'V', "value", "", value, "Print stock total value" },
91     { 'P', "print", "", print, "Print a table of products" },
92     { 'Q', "quit", "", quit, "Quit the program" },
93 };
94 void printHelp() {
95     std::cout << "\nFormat of commands: \n"
96               << "name - shortName args - description\n";
97
98     for (size_t i = 0; i < (sizeof(cmds) / sizeof(cmds[0])); ++i)

```

```

99         cmds[i].print();
100     }
101     bool eval(Stock& stock, MultiInputHelper& ih) {
102         auto s = ih.readStringView("> ");
103         if (s.size() == 1) {
104             for (size_t i = 0; i < (sizeof(cmds) / sizeof(cmds[0])); ++i) {
105                 if (cmds[i].shortName == toupper(s[0]))
106                     return cmds[i].f(stock, ih);
107             }
108         }
109         for (size_t i = 0; i < (sizeof(cmds) / sizeof(cmds[0])); ++i) {
110             if (cmds[i].name == s) return cmds[i].f(stock, ih);
111         }
112         std::cout << "Invalid command!\n";
113         printHelp();
114         ih.getLineAfterInvalid("> ");
115
116         return true;
117     }
118
119     int main() {
120         Stock stock;
121         printHelp();
122         MultiInputHelper ih("> ");
123         for (;;) {
124             if (!eval(stock, ih)) break;
125         }
126         return 0;
127     }

```