Лабораторная работа №5. Основны применения стандартных контейнеров. Ввод и вывод данных.

Необходимо написать многофайловую программу, в которой применяются алгоритмы обработки некорректного пользовательского ввода, а для хранения данных используются такие контейнеры как `std::array`, `std::vector` и `std::string`. Предусмотрены возможности считывания/вывода данных из/в файл(а).

Теоретическая часть

Рассмотрим общие представления о работе с такими контейнерами как `std::array`, `std::vector` и `std::string`.

std::array

- Представленный в C++11, `std::array` — это фиксированный массив, который не распадается в указатель при передаче в функцию.
- std::array определяется в заголовочном файле `array`, внутри пространства имён `std`.

Объявление переменной `std::array` следующее:

```
#include <array>
std::array<int, 4> myarray; // объявляем массив типа int длиной 4
```

- Длина `std::array` должна быть установлена во время компиляции.
- `std::array` можно инициализировать с использованием списка инициализаторов или `uniform`-инициализации:

```
std::array<int, 4> myarray = { 8, 6, 4, 1 }; // список инициализаторов
std::array<int, 4> myarray2 { 8, 6, 4, 1 }; // uniform-инициализация

std::array<int, 4> myarray;
myarray = { 0, 1, 2, 3 }; // ок
myarray = { 8, 6 }; // ок, элементам 2 и 3 присвоен нуль!
myarray = { 0, 1, 3, 5, 7, 9 }; // нельзя, слишком много элементов в списке инициализаторов!
```

- В `std::array` вы не можете пропустить (не указывать) длину массива:

```
std::array<int, > myarray = { 8, 6, 4, 1 }; // нельзя, должна быть указана длина массива
```

- Доступ к значениям массива через оператор индекса осуществляется как обычно.
- Оператор индекса не выполняет никаких проверок на диапазон.
- Если указан недопустимый индекс, то произойдёт сбой.

```
std::cout << myarray[1];
myarray[2] = 7;
```

- `std::array` поддерживает вторую форму доступа к элементам массива — функция `at()`.
- функция `at()` осуществляет проверку диапазона.

```
std::array<int, 4> myarray { 8, 6, 4, 1 };
```

```
myarray.at(1) = 7; // элемент массива под номером 1 - корректный, присваиваем ему значение 7
myarray.at(8) = 15; // элемент массива под номером 8 - некорректный, получим ошибку (выбрасывается исключение типа std::out_of_range), которую можно обработать
```

- Поскольку проверка диапазона выполняется, то функция `at()` работает медленнее, чем оператор `[]`.
- `std::array` автоматически делает все очистки после себя, когда выходит из области видимости, поэтому нет необходимости прописывать это вручную.

Размер и сортировка

- С помощью функции `size()` можно узнать длину массива:

```
#include <iostream>
#include <array>
int main()
{
	std::array<double, 4> myarray{ 8.0, 6.4, 4.3, 1.9 };
	std::cout << "length: " << myarray.size();
	return 0;
}
```

Результат:

length: 4

- Поскольку `std::array` не распадается в указатель при передаче в функцию, то функция `size()` будет работать, даже если её вызвать из другой функции:

```
#include <iostream>
#include <array>
void printLength(const std::array<double, 4> &myarray)
{
    std::cout << "length: " << myarray.size();
}
int main()
{
    std::array<double, 4> myarray { 8.0, 6.4, 4.3, 1.9 };
    printLength(myarray);
    return 0;
}
```

Результат тот же:

length: 4

- Стандартная библиотека C++ использует термин «размер» для обозначения длины массива — не путайте это с результатами выполнения оператора `sizeof` с обычным фиксированным массивом, когда возвращается фактический размер массива в памяти (размер элемента * длина массива).
- В примере мы передаём `std::array` по (константной) ссылке. Это делается по соображениям производительности для того, чтобы компилятор не выполнял копирование массива при передаче в функцию.
- Всегда передавайте `std::array` в функции по обычной или по константной ссылке.

Поскольку длина массива всегда известна, то циклы `foreach` также можно использовать с `std::array`:

```
std::array<int, 4> myarray { 8, 6, 4, 1 };
for (auto &element : myarray)
    std::cout << element << ' ';
```

Вы можете отсортировать `std::array`, используя функцию `std::sort()`, которая находится в заголовочном файле `algorithm`:

```
#include <iostream>
#include <array>
#include <algorithm> // для std::sort
int main()
{
    std::array<int, 5> myarray { 8, 4, 2, 7, 1 };
    std::sort(myarray.begin(), myarray.end()); // сортировка массива по
возрастанию
//    std::sort(myarray.rbegin(), myarray.rend()); // сортировка массива по
убыванию
    for (const auto &element : myarray)
        std::cout << element << ' ';

    return 0;
}
```

Результат:

```
1 2 4 7 8
```

Функция сортировки использует итераторы, которые мы ещё не рассматривали. О них мы поговорим несколько позже.

std::vector

В стандартной библиотеке C++ вектором (`std::vector`) называется динамический массив, обеспечивающий быстрое добавление новых элементов в конец и меняющий свой размер при необходимости. Вектор гарантирует отсутствие утечек памяти.

- Для работы с вектором нужно подключить заголовочный файл `vector`.
- Элементы вектора должны быть одинакового типа, и этот тип должен быть известен при компиляции программы. Он задаётся в угловых скобках после `std::vector`: например, `std::vector<int>` — это вектор целых чисел типа `int`, а `std::vector<std::string>` — вектор строк.
- Само имя `std::vector` не является типом данных: это шаблон, в который требуется подставить нужные параметры (тип элемента), чтобы получился конкретный тип данных.

Рассмотрим пример программы, которая заполняет вектор элементами и печатает их через пробел:

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

int main() {
      std::vector<int> data = {1, 2, 3, 4, 5};
      for (int elem : data) {
            std::cout << elem << " ";
      }
```

```
	std::cout << "\n";
}
```

Здесь мы инициализируем вектор через список инициализации, в котором элементы перечислены через запятую. Другой способ инициализации вектора — указать число элементов и (при необходимости) образец элемента:

```
#include <string>
#include <vector>

int main() {
	std::vector<std::string> v1; // пустой вектор строк
	std::vector<std::string> v2(5); // вектор из пяти пустых строк
	std::vector<std::string> v3(5, "hello"); // вектор из пяти строк "hello"
}
```

Обращение к элементам

- Вектор хранит элементы в памяти последовательно, поэтому по индексу элемента можно быстро найти его положение в памяти.
- Индексация начинается с нуля.
- Отрицательные индексы не допускаются.

```
std::vector<int> data = {1, 2, 3, 4, 5};
int a = data[0]; // начальный элемент вектора
// последний элемент вектора (в нём пять элементов)
int b = data[4];
data[2] = -3; // меняем элемент 3 на -3
```

- Общее количество элементов в векторе, можно воспользоваться функцией `size`:

```
std::cout << data.size() << "\n";
```

- Когда мы обращаемся по индексу через квадратные скобки, проверки его корректности не происходит.
- Если в корректности индекса вы не уверены, то можно использовать функцию `at()`.

```
std::vector<int> data = {1, 2, 3, 4, 5};
// неопределённое поведение: может произойти всё что угодно
std::cout << data[42] << "\n";
std::cout << data.at(0) << "\n"; // напечатается 1
// произойдёт исключение std::out_of_range — его можно
//будет перехватить и обработать
std::cout << data.at(42) << "\n";
```

Рассмотрим функции вектора `front` и `back`, которые возвращают его первый и последний элемент без использования индексов:

```
std::vector<int> data = {1, 2, 3, 4, 5};
// то же, что data[0]
std::cout << data.front() << "\n";
// то же, что data[data.size() - 1]
std::cout << data.back() << "\n";
```

- Важно учитывать, что вызов этих функций на пустом векторе приведёт к неопределённому поведению.

- Для проверки вектора на пустоту вместо сравнения `data.size() == 0` принято использовать функцию empty, которая возвращает логическое значение:

```
if (!data.empty()) {
// вектор не пуст, с ним можно работать
}
```

Итерация по индексам

Так сложилось, что в стандартной библиотеке индексы и размеры контейнеров имеют беззнаковый тип. Вместо `unsigned int` или `unsigned long int` для него используется традиционный псевдоним `size_t` (а точнее, `std::vector<T>::size_type`). Тип `size_t` на самом деле совпадает с `uint32_t` или `uint64_t` в зависимости от битности платформы. Его использование в программе дополнительно подчёркивает, что мы имеем дело с индексами или с размером.
Итерацию по элементам `data` с помощью индексов можно записать так:

```
for (size_t i = 0; i != data.size(); ++i) {
      std::cout << data[i] << " ";
} // Это каноническая форма записи такого цикла.
```

Беззнаковость типа возвращаемого значения функции `size` порождает следующую проблему. По правилам, унаследованным ещё от языка C, результат арифметических действий над беззнаковым и знаковым типами приводится к беззнаковому типу. Поэтому выражение `data.size() - 1`, например, тоже будет беззнаковым. Если `data.size()` окажется нулём, то такое выражение будет вовсе не минус единицей, а самым большим беззнаковым целым (для 64-битной платформы это $2^{64}-1$).
Рассмотрим следующий ошибочный код, который проверяет, есть ли в векторе дубликаты, идущие подряд:

```
// итерация по всем элементам, кроме последнего:
for (size_t i = 0; i < data.size() - 1; ++i) {
      if (data[i] == data[i + 1]) {
            std::cout << "Duplicate value: " << data[i] << "\n";
      }
}
```

Эта программа будет некорректно работать на пустом векторе.

- Условие `i < data.size() - 1` на первой итерации окажется истинным, и произойдёт обращение к элементам пустого вектора.
- Правильнее было бы переписать это условие через `i + 1 < data.size()`.
- Можно воспользоваться внешней функцией `std::ssize`, которая появилась в C++20. Она возвращает знаковый размер вектора.

```
for (std::int64_t i = 0; i < std::ssize(data) - 1; ++i) {
      if (data[i] == data[i + 1]) {
            std::cout << "Duplicate value: " << data[i] << "\n";
      }
}
```

Добавление и удаление элементов

В вектор можно эффективно добавлять элементы в конец и удалять их с конца. Для этого существуют функции `push_back` и `pop_back`. Рассмотрим программу, считывающую числа с клавиатуры в вектор и затем удаляющую все нули в конце:

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

int main() {
	int x;
	std::vector<int> data;
	while (std::cin >> x) { // читаем числа, пока не закончится ввод
		data.push_back(x); // добавляем очередное число в вектор
	}

	while (!data.empty() && data.back() == 0) {
		// Пока вектор не пуст и последний элемент равен нулю
		data.pop_back(); // удаляем этот нулевой элемент
	}
}
```

- Добавление элементов в другие части вектора или их удаление неэффективно, так как требует сдвига соседних элементов.
- Отдельных функций, например, для добавления или удаления элементов из начала у вектора нет.
- Это можно сделать с помощью общих функций `insert/erase` и итераторов.
- Удалить все элементы из вектора можно с помощью функции `clear`.

Резерв памяти

- Вектор хранит элементы в памяти в виде непрерывной последовательности, друг за другом. При этом в конце последовательности резервируется дополнительное место для быстрого добавления новых элементов. Когда этот резерв заканчивается, при вставке очередного элемента происходит реаллокация: элементы вектора копируются в новый, более просторный блок памяти.
- Реаллокация — довольно дорогая процедура, но если она происходит достаточно редко, то её влияние незначительно. Можно доказать, что если размер нового блока выбирать в два раза больше предыдущего размера, то амортизационная сложность добавления элемента будет константной.
- Текущий резерв вектора можно узнать с помощью функции `capacity` (не путайте её с функцией `size`).

Соотношение резерва вектора и его размера показано на рисунке 5.1.

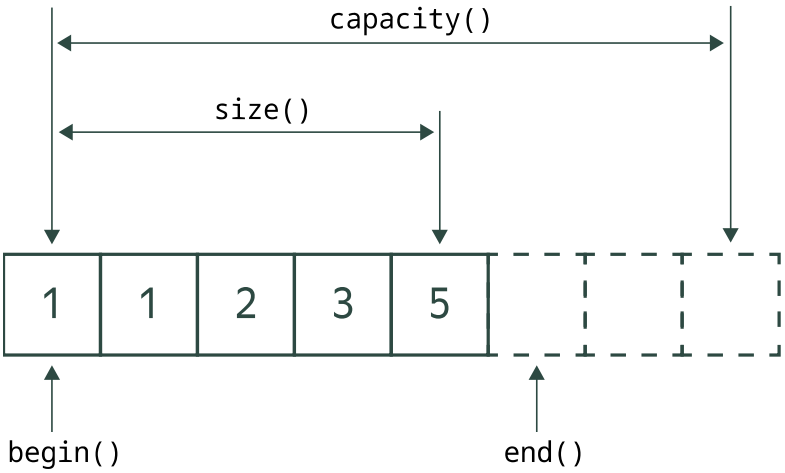

Рис. 5.1 Соотношение резерва вектора и его размера

Рассмотрим программу, в которой в вектор последовательно добавляются элементы и после каждого шага печатается размер и резерв:

```
#include <iostream>
```

```
#include <vector>

int main() {
	std::vector<int> data = {1, 2};
	std::cout << data.size() << "\t" << data.capacity() << "\n";

	data.push_back(3);
	std::cout << data.size() << "\t" << data.capacity() << "\n";

	data.push_back(4);
	std::cout << data.size() << "\t" << data.capacity() << "\n";

	data.push_back(5);
	std::cout << data.size() << "\t" << data.capacity() << "\n";
}
```

Вот вывод этой программы:

```
2	2
3	4
4	4
5	8
```

- Видно, что размер вектора увеличивается на единицу, а резерв удваивается после исчерпания.
- При добавлении четвёрки используется имеющаяся в резерве память, а при добавлении тройки и пятёрки происходит реаллокация.

Демонстрация одной из процедур реаллокации для программы из предыдущего листинга приведена на рисунке 5.2.

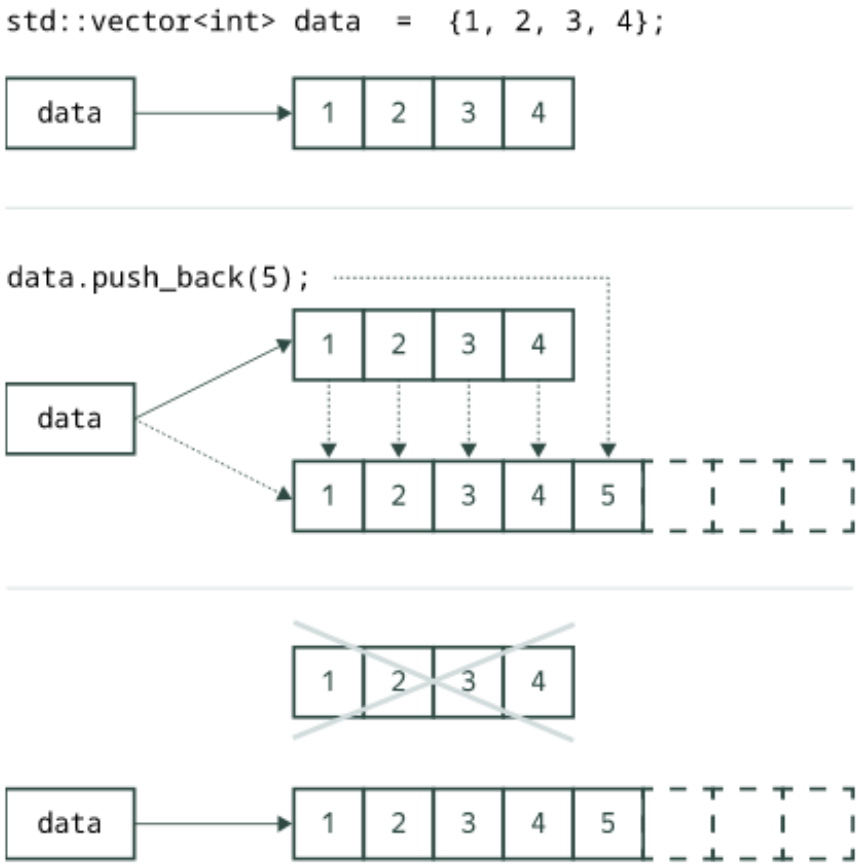

Рис. 5.2 Пример процедуры реаллокации

Иногда требуется заполнить вектор элементами, причём число элементов известно заранее. В таком случае можно сразу зарезервировать нужный размер памяти с помощью функции `reserve`, чтобы при добавлении элементов не происходили реаллокации. Пусть, например, нам сначала задаётся число слов, а потом сами эти слова, и нам требуется сложить их в вектор:

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

int main() {
	std::vector<std::string> words;

	size_t words_count;
	std::cin >> words_count;

	// Размер вектора остаётся нулевым, меняется только резерв:
	words.reserve(words_count);

	for (size_t i = 0; i != words_count; ++i) {
		std::string word;
		std::cin >> word;
		// Все добавления будут дешёвыми, без реаллокаций:
		words.push_back(word);
	}
}
```

- Если передать в `reserve` величину меньше текущего резерва, то ничего не поменяется — резерв останется прежним.
- Функцию `reserve` не следует путать с функцией `resize`, которая меняет количество элементов в векторе.
- Если аргумент функции `resize` меньше текущего размера, то лишние элементы в конце вектора удаляются. Если же он больше текущего размера, то при необходимости происходит реаллокация и в вектор добавляются новые элементы с дефолтным значением данного типа.

```
#include <vector>

int main() {
	std::vector<int> data = {1, 2, 3, 4, 5};
	// поменяли резерв, но размер вектора остался равным пяти
	data.reserve(10);

	data.resize(3); // удалили последние элементы 4 и 5
	data.resize(6); // получили вектор 1, 2, 3, 0, 0, 0
}
```

Многомерные векторы

Воспользуемся вектором векторов, чтобы сохранить матрицу (таблицу) целых чисел. Пусть на вход программы сначала поступают число строк и число столбцов матрицы, а потом — сами элементы построчно:

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

int main() {
	size_t m, n;
```

```
	std::cin >> m >> n; // число строк и столбцов

	/* создаём матрицу matrix из m строк, каждая из которых — вектор из n
нулей */
	std::vector<std::vector<int>> matrix(m, std::vector<int>(n));

	for (size_t i = 0; i != m; ++i) {
		for (size_t j = 0; j != n; ++j) {
			std::cin >> matrix[i][j];
		}
	}
	// напечатаем матрицу, выводя элементы через табуляцию
	for (size_t i = 0; i != m; ++i) {
		for (size_t j = 0; j != n; ++j) {
			std::cout << matrix[i][j] << "\t";
		}
		std::cout << "\n";
	}
}
```

Сортировка вектора

В стандартной библиотеке в заголовочном файле `algorithm` есть готовая функция `sort`. Гарантируется, что сложность её работы в худшем случае составляет $O(n \cdot \log n)$, где `n` — число элементов в векторе. Типичные реализации используют алгоритм сортировки `Introsort`.

```
#include <algorithm>
#include <vector>

int main() {
	std::vector<int> data = {3, 1, 4, 1, 5, 9, 2, 6};

	// Сортировка диапазона вектора от начала до конца
	std::sort(data.begin(), data.end());

	// получим вектор 1, 1, 2, 3, 4, 5, 6, 9
}
```

- В функцию `sort` передаются так называемые итераторы, ограничивающие рассматриваемый диапазон.
- В нашем случае мы передаём диапазон, совпадающий со всем вектором, от начала до конца.
- Соответствующие итераторы возвращают функции `begin` и `end`.
- Итераторы можно считать обобщёнными индексами (но они могут быть и у контейнеров, не допускающих обычную индексацию).
- Для сортировки по убыванию можно передать на вход обратные итераторы `rbegin` и `rend`, представляющие элементы вектора в перевёрнутом порядке.

```
std::sort(data.rbegin(), data.rend()); // 9, 6, 5, 4, 3, 2, 1, 1
```

std::string

Контейнер `std::string` можно рассматривать как особый случай вектора символов `std::vector<char>`, имеющий набор дополнительных функций. В частности, у строки есть все те же рассмотренные нами функции, что и у вектора (например, `pop_back` или `resize`).

Рассмотрим некоторые специфические функции строки:

```
#include <iostream>
#include <string>

int main() {
	std::string s = "Some string";

	// приписывание символов и строк
	s += ' '; // добавляем отдельный символ в конец, это аналог push_back
	s += "functions"; // добавляем строку в конец
	std::cout << s << "\n"; // Some string functions

	// выделение подстроки
	// подстрока "string" из 6 символов начиная с 5-й позиции
	std::string sub1 = s.substr(5, 6);
	// подстрока "functions" с 12-й позиции и до конца
	std::string sub2 = s.substr(12);

	// поиск символа или подстроки
	size_t pos1 = s.find(' '); // позиция первого пробела, в данном случае 4
	size_t pos2 = s.find(' ', pos1 + 1); // позиция следующего пробела (11)
	size_t pos3 = s.find("str"); // вернётся 5
	size_t pos4 = s.find("#"); // вернётся std::string::npos
}
```

- Вставку, замену и удаление подстрок можно сделать через указание индекса начала и длины подстроки:

```
#include <iostream>
#include <string>

int main() {
	std::string s = "Some string functions";

	// вставка подстроки
	s.insert(5, "std::");
	std::cout << s << "\n"; // Some std::string functions

	// замена указанного диапазона на новую подстроку
	s.replace(0, 4, "Special");
	std::cout << s << "\n"; // Special std::string functions

	// удаление подстроки
	s.erase(8, 5); // Special string functions
}
```

- Аналогичные действия для других контейнеров (например, для того же вектора) можно сделать через итераторы.

Практическая часть

1. Ознакомится с теоретическим текстовым материалом изложенным в пояснении к лабораторной работе.
2. Получить задание у преподавателя (два номера задач).
3. Обеспечить в разрабатываемых программах дружественный консольный интерфейс. Программа должна предлагать пользователю вводить исходные данные (с пояснением того, что нужно ввести), а также при выводе результат её работы должен сопровождаться пояснением (комментарием). Например, при вводе программа запрашивает число строк и столбцов матрицы:
   `n = 2 `\n`
   m = 3 `\n`
   Введите через разделитель элементы матрицы `\n`
   1 2 3 1 0 1 `\n`,`
   а при выводе — номер строки с наибольшим элементом заданной матрицы:
   `Наибольший элемент располагается в 1-ой строке`.
4. Обеспечить в разрабатываемых программах защиту от некорректного пользовательского ввода. При этом следует продумать возможные случаи такого некорректного ввода. Перечислить в отчёте примеры возможных вариантов некорректного пользовательского ввода. Продемонстрировать то, каким образом разрабатываемые программы обрабатывают некорректный пользовательский ввод. Достаточно привести по одному примеру на один случай некорректного пользовательского ввода.
5. Обеспечить в разрабатываемых программах возможность многократного ввода исходных данных необходимых для решения поставленной задачи. Программа должна спрашивать примерно следующее: «Хотите повторить ввод исходных данных? Да — 1, Нет — 0.» Также в некотором виде должен формироваться запрос(ы), определяющие откуда поступят исходные данные и куда будет осуществлён вывод результата.
6. Обеспечить ввод исходных данных из файла / консоли. Смотрите таблицу 5.1.
7. Обеспечить вывод результатов в файл / консоль. Смотрите таблицу 5.1.
8. В случае ввода данных из файла программа должна завершаться (не предлагать повторно ввести исходные данные). Смотрите таблицу 5.1.
9. Обеспечить ввод данных из файла таким образом, что, при этом путь к файлу задаётся с консоли, если он не указывается, то используется путь к файлу ввода по умолчанию. То есть, например, программа выдаёт такой запрос: «Укажите файл для ввода исходных данных для работы программы `[~/rez/in.txt]`:». Если при этом просто нажать Enter, то файлом для ввода исходных данных будет файл `~/rez/in.txt`.
10. Обеспечить вывод результатов работы программы в файл таким образом, что, при этом путь к файлу вывода задаётся с консоли, если он не указывается, то используется путь к файлу по умолчанию. То есть, например, программа выдаёт такой запрос: «Укажите файл для вывода результатов работы программы `[~/rez/out.txt]`:».

Если при этом просто нажать `Enter`, то файлом вывода будет файл `~/rez/out.txt`).

11. Продемонстрировать инициализацию переменных с использованием `uniform`-инициализации.

12. Обеспечить работу со строками произвольной длины, то есть считать, что заранее не известно то, какой именно длины будет последовательность символов, задаваемая пользователем. Не следует обращать внимание на то, что в индивидуальном задании указывается количество символов входной последовательности. Считать такое указание одним из многих вариантов размерности, заданной опосредованно пользователем.

13. Обеспечить работу с матрицами произвольной размерности. Пользователь может задать любое количество строк и столбцов. Если в индивидуальном задании указывается какая-то конкретная размерность для матрицы, то считать такое указание одним из многих вариантов размерности, заданной пользователем.

14. Для хранения данных использовать только контейнеры, при этом: для нечётных номеров заданий — `str::array` и `std::string`, для чётных номеров заданий — `std::vector` и `std::string`.

15. При передаче в функции контейнеров с данными использовать передачу по ссылке. Если функция не изменяет данные, то использовать передачу по константной ссылке.

16. Выполнить по заданию преподавателя одну указанную задачу из диапазона 251 — 270.[18] При выполнении задачи осуществить обработку последовательности символов, длина последовательности заранее неизвестна. Номер задачи должен отличаться от номера задачи, выполненной в лабораторной работе №4.

17. Выполнить по заданию преподавателя одну указанную задачу из диапазона 367 — 423. При выполнении задачи выполнить манипуляции с матрицей, используя вложенные циклы. Номер задачи должен отличаться от номера задачи, выполненной в лабораторной работе №4.

18. Задания с графикой давать не нужно.

19. Оформить программу как многофайловую (`все следующие в данном пункте требования – отмеченные * – должны быть выполнены`).
    * Используемые константы расположить в отдельном заголовочном файле `*.h`.
    * Дополнительно поместить используемые константы в отдельное пространство имён.
    * Прототипы пользовательских функций расположить в отдельном заголовочном файле(файлах) `*.h`.
    * Пользовательские функций расположить в отдельном файле(файлах) `*.cpp`.
    * В заголовочных файлах использовать механизм `Header guards` или `#pragma once`.
    * Таким образом, в главном файле проекта должна располагаться только функция `main`, в которой подключается ряд пользовательских заголовочных файлов и осуществляется вызов пользовательских функций.

18 «Задачи по программированию». Авторы: С.А. Абрамов, Г.Г. Гнездилова, Е.Н. Капустина, М.И. Селюн. Компьютерный набор и оформление: Е.А. Гречникова. Лежит в ББ.

20. Оформить по выполненной лабораторной работе в рабочей тетради письменный отчёт.[19]

Табл. 5.1. Пояснение к пунктам 6 - 8

Входой поток	**Выходной поток**	**Завершение / Вопрос о продолжении**
Консоль	Консоль	Вопрос о продолжении
Консоль	Файл	Вопрос о продолжении
Файл	Консоль	Завершение
Файл	Файл	Завершение

19 **Содержание отчёта:**
1. Задания на лабораторную работу.
2. Листинги программ.
3. Примеры выполнения программы (исходные данные + результаты).
4. Перечисление примеров возможных вариантов некорректного пользовательского ввода (начиная с лабораторной работы №3).

Примечание: Рабочая тетрадь с отчётами подписывается с указанием ФИО и номера учебной группы.