**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра «Информационные Системы и Технологий (ИСиТ)»**

Курсовая РАБОТА

**по дисциплине «Программирование»**

Тема: Основы алгоритмизации и программирования на языке С++

|  |  |  |
| --- | --- | --- |
| Студент гр. 0324 |  | Зимацкий С. Н. |
| Преподаватель |  | Глущенко А.Г. |

Санкт-Петербург

2020

**ЗАДАНИЕ**

**на курсовую работу**

|  |  |  |
| --- | --- | --- |
| Студент Зимацкий С. Н. | | |
| Группа 0324 | | |
| Тема работы: Основы алгоритмизации и программирования на языке С++ | | |
| Исходные данные:  Необходимо объединить все 4 лабораторные работы в единый проект. Нужно добавить инфраструктуру переключения между заданиями (интерактивное меню). | | |
|  | | |
| Дата выдачи задания: 21.09.2020 | | |
| Дата сдачи реферата: 12.18.2020 | | |
| Дата защиты реферата: 12.18.2020 | | |
| Студент гр. 0324 |  | Зимацкий С. Н. |
| Преподаватель |  | Глущенко А.Г. |

**Аннотация**

Все лабораторные работы совмещены в одном .cpp файле, внутри файла логически разделены на большие функции, каждая из которых содержит лабораторную работу, с помощью оператора switch осуществлён переход между лабораторными работами, проект залит на GitHub. В результате возможно исполнить программу на компьютере, ниже приложены скриншоты работы программы и код.

**содержание**

|  |  |  |
| --- | --- | --- |
|  | Введение | 5 |
| 1. | Типы данных и их внутреннее представление в памяти | 6 |
| 1.1. | Цель работы. | 6 |
| 1.2. | Основные теоретические положения. | 6 |
| 1.3. | Экспериментальные результаты. | 8 |
| 2. | Одномерные статические массивы | 10 |
| 2.1. | Цель работы. | 10 |
| 2.2. | Основные теоретические положения. | 12 |
| 2.3. | Экспериментальные результаты. | 19 |
| 3. | Арифметика указателей и матрицы | 23 |
| 3.1. | Цель работы. | 23 |
| 3.2. | Основные теоретические положения. | 24 |
| 3.3. | Экспериментальные результаты. | 31 |
| 4. | Текстовые строки как массивы символов, организация работы с файлами | 36 |
| 4.1. | Цель работы. | 36 |
| 4.2. | Основные теоретические положения. | 37 |
| 4.3. | Экспериментальные результаты.  Заключение  Список использованных источников  Приложение А. Код программы. | 45  48  49  50 |
|  |  |  |

**введение**

Объединение 4-х лабораторных работ в одну с помощью интерфейса.

Ниже в каждом разделе описаны все лабораторные работы.

**1.** Типы данных и их внутреннее представление в памяти

**1.1. Цель работы.**Разработать алгоритм и написать программу, которая позволяет:

1) Вывести, сколько памяти (в байтах) на вашем компьютере отводится под различные типы данных со спецификаторами и без: int, short int, long int, float, double, long double, char и bool.

2) Вывести на экран двоичное представление в памяти (все разряды) целого числа. При выводе необходимо визуально обозначить знаковый разряд и значащие разряды отступами или цветом.

3) Вывести на экран двоичное представление в памяти (все разряды) типа float. При выводе необходимо визуально обозначить знаковый разряд мантиссы, знаковый разряд порядка (если есть), мантиссу и порядок.

4) Вывести на экран двоичное представление в памяти (все разряды) типа double. При выводе необходимо визуально обозначить знаковый разряд мантиссы, знаковый разряд порядка (если есть), мантиссу и порядок. (\*)  
  
(\*) - Задание повышенной сложности. Подобные задания могут использовать методы и структуры следующих модулей.

**1.2. Основные теоретические положения.**

Внутреннее представление величин целого типа – целое число в двоичном коде. При использовании спецификатора signed старший бит числа интерпретируется как знаковый (0 – положительное число, 1 – отрицательное). Для кодирования целых чисел со знаком применяется прямой, обратный и дополнительный коды.

Представление положительных и отрицательных чисел в прямом, обратном и дополнительном кодах отличается. В прямом коде в знаковый разряд помещается цифра 1, а в разряды цифровой части числа – двоичный код его абсолютной величины. Прямой код числа −3 (для 16- разрядного процессора):



Обратный код получается инвертированием всех цифр двоичного кода абсолютной величины, включая разряд знака: нули заменяются единицами, единицы – нулями. Прямой код можно преобразовать в обратный, инвертировав все значения всех битов (кроме знакового). Обратный код числа −3:



Дополнительный код получается образованием обратного кода с последующим прибавлением единицы к его младшему разряду. Дополнительный код числа −3:

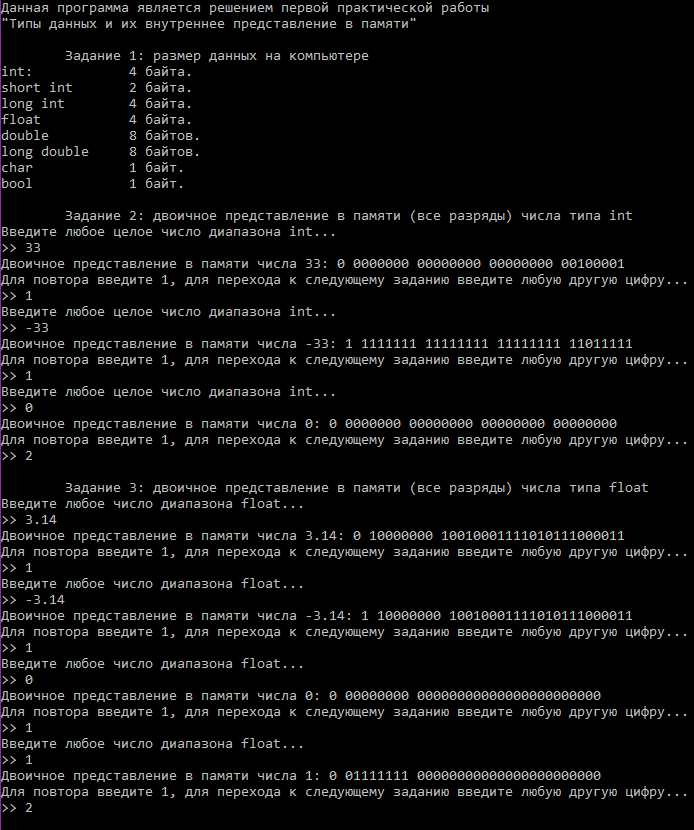


Вещественные типы данных хранятся в памяти компьютера иначе, чем целочисленные. Внутреннее представление вещественного числа состоит из двух частей – мантиссы и порядка.

Для 32-разрядного процессора для float под мантиссу отводится 23 бита, под экспоненту – 8, под знак – 1. Для double под мантиссу отводится 52 бита, под экспоненту – 11, под знак – 1:



**1.3. Экспериментальные результаты.**

Скриншоты работы программы: 

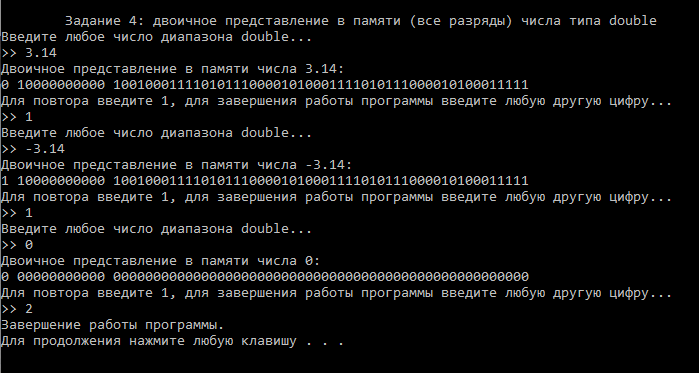


Рисунок 1 – Результат работы программы (полный код программы представлен в приложении А)

**2.** Одномерные статические массивы

**2.1. Цель работы.**

Ознакомиться с работой одномерных массивов, базовыми операциями с массивами, различным типам сортировок массивов.

Необходимо написать программу, которая:

1)    Создает целочисленный массив размерности *N* = 100. Элементы массивы должны принимать случайное значение в диапазоне от -99 до 99.

2)    Отсортировать заданный в пункте 1 элементы массива […] сортировкой (от меньшего к большему). Определить время, затраченное на сортировку, используя библиотеку chrono.

3)    Найти максимальный и минимальный элемент массива. Подсчитайте время поиска этих элементов в отсортированном массиве и неотсортированном, используя библиотеку chrono.

4)    Выводит среднее значение (если необходимо, число нужно округлить) максимального и минимального значения. Выводит индексы всех элементов, которые равны этому значению, и их количество.

5)    Выводит количество элементов в отсортированном массиве, которые меньше числа *a*, которое инициализируется пользователем.

6)    Выводит количество элементов в отсортированном массиве, которые больше числа *b*, которое инициализируется пользователем.

7)    Выводит информацию о том, есть ли введенное пользователем число в отсортированном массиве. Реализуйте алгоритм бинарного поиска. Сравните скорость его работы с обычным перебором. (\*)

8)     Меняет местами элементы массива, индексы которых вводит пользователь. Выведите скорость обмена, используя библиотеку chrono.

Должна присутствовать возможность запуска каждого пункта многократно.

Таблица – Варианты сортировок

|  |  |
| --- | --- |
| Номер варианта | Название сортировки |
| 1 | Bubble sort (пузырьковая сортировка) |
| 2 | Shaker sort (шейкер-сортировка) |
| 3 | Comb sort (сортировка расчёской) |
| 4 | Insert sort (сортировка вставками) |
| 5 (\*) | Quick sort (быстрая сортировка) |

**2.2. Основные теоретические положения.**

* Понятие массива

При использовании простых переменных каждой области памяти для хранения данных соответствует свое имя. Если с группой величин одинакового типа требуется выполнить однообразные действия, им дают одно имя, а различают по порядковому номеру (индексу). Это дает возможность компактно записать множество операций с использованием циклов.

Массив представляет собой индексированную последовательность однотипных элементов с заранее определенным количеством элементов. Наглядно одномерный массив можно представить, как набор пронумерованных ячеек, в каждой из которых содержится определенное значение.

Все массивы можно разделить на две группы: одномерные и многомерные. Описание массива в программе отличается от объявления обычной переменной наличием размерности массива, которая задается в квадратных скобках после имени.

Элементы массива нумеруются с нуля. При описании массива используются те же модификаторы (класс памяти, const и инициализатор), что и для простых переменных.

Аналогом одномерного массива из математики может служить последовательность некоторых элементов с одним индексом: a\_i*ai*​ при  i = 0, 1, 2, … n – одномерный вектор. Каждый элемент такой последовательности представляет собой некоторое значение определенного типа данных. Наглядно одномерный массив можно представить как набор пронумерованных ячеек, в каждой из которых содержится определенное значение:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 3.02 | 1.5 | 7.0 | -2.3 | 12.0 |

Это пример одномерного массива из 5 элементов, каждый из которых представляет собой некоторое вещественное значение и каждое из этих значений имеет индекс от 0 до 4.

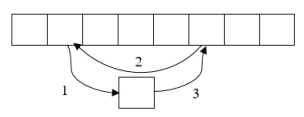
А вот пример одномерного массива из десяти элементов, представляющих собой одиночные символы:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| ‘a’ | ‘b’ | ‘c’ | ‘+’ | ‘1’ | ‘2’ | ‘!’ | ‘#’ | ‘@’ | ‘&’ |

Каждый элемент в этих массивах определяется значением индекса элемента. Например, в последнем массиве элемент с индексом 5 равен символу  ‘2’.

* Обмен местами элементов массива

Обмен элементов массива осуществляется через буферную переменную либо через функцию swap(a, b).

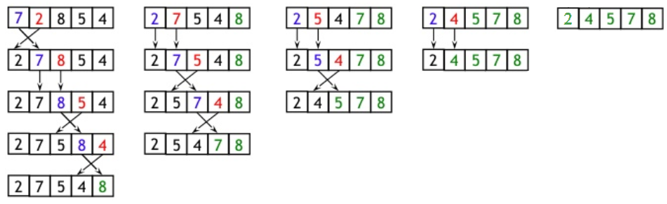


* Пузырьковая сортировка массива (bubble sort)

Сортировка – процесс размещения элементов заданного множества объектов в определенном порядке. Когда элементы отсортированы, их проще найти, производить с ними различные операции. Сортировка напрямую влияет на скорость алгоритма, в котором нужно обратиться к определенному элементу массива.

Простейшая из сортировок – сортировка обменом (пузырьковая сортировка). Вся суть метода заключается в попарном сравнении элементов и последующем обмене. Таким образом, если следующий элемент меньше текущего, то они меняются местами, максимальный элемент массива постепенно смещается в конец массива, а минимальный – в начало. Один полный проход по массиву может гарантировать, что в конце массива находится максимальный элемент.

Затем процесс повторяется до тех пор, пока вся последовательность не будет упорядочена. Важно заметить, что после первого прохода по массиву, уже имеется один упорядоченный элемент, он стоит на своем месте, и менять его не надо. Таким образом на следующем шаге будут сравниваться *N*-1 элемент.



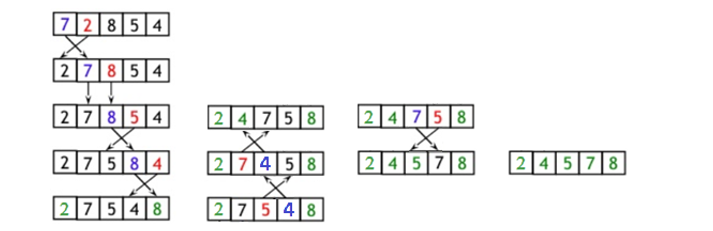
Очевидно, что хуже всего алгоритм будет работать, когда на вход подается массив, отсортированный в обратную сторону (от большего к меньшу). Быстрее же всего алгоритм работает с уже отсортированным массивом.

Но стандартный алгоритм пузырьковой сортировки предполагает полный циклический проход по массиву. Если изначально подается упорядоченная последовательность, то работа алгоритма все равно продолжиться. Исправить это можно, добавив условие проверки: если на текущей итерации ни один элемент не изменил свой индекс, то работа алгоритма прекращается.

* Шейкер-сортировка массива (shaker sort)

Shaker sort – модификация пузырьковой сортировки. Принцип работы этой сортировки аналогичен bubble sort: попарное сравнение элементов и последующий обмен местами. Но имеется существенное отличие. Как только максимальный элемент становится на свое место, алгоритм не начинает новую итерацию с первого элемента, а запускает сортировку в обратную сторону. Алгоритм гарантирует, что после выполнения первой итерации, минимальный и максимальный элемент будут в начале и конце массива соответственно.

Затем процесс повторяется до тех пор, пока массив не будет отсортирован. За счет того, что сортировка работает в обе стороны, массив сортируется на порядок быстрее. Очевидным примером этого был бы случай, когда в начале массива стоит максимальный элемент, а в конце массива – минимальный. Shaker sort справится с этим за 1 итерацию, при условии, что другие элементы стоят на правильном месте.



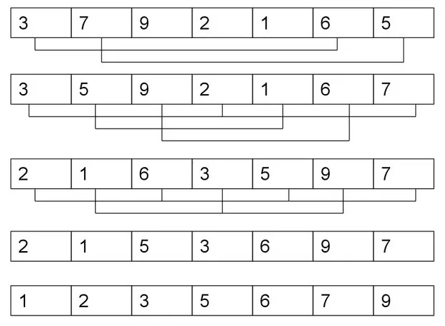
Кажется, что bubble sort теряет свою эффективность по сравнению с shaker sort. Сортировка проходит в массиве в обоих направлениях, а не только от его начала к концу. Но в работе с большими массивами преимущество шейкер-сортировки уменьшается как раз из-за использования двух циклов.

* Сортировка массива расчёской (comb sort)

Очевидный недостаток bubble и shaker sort заключается в том, что элементы переставляются максимум на одну позицию.

Comb sort (сортировка расческой) – ещё одна модификация сортировки пузырьком. Алгоритм был разработан специально для случаев, когда минимальные элементы стоят слишком далеко, или максимальные – слишком близко к началу массива. В сортировке расческой переставляются элементы, стоящие на расстоянии.

Оптимально изначально взять расстояние равным длине массива , а далее уменьшать его на определенный коэффициент, который примерно равен 1.247. Когда расстояние станет равно 1, выполняется обычная сортировка пузырьком.



Сортировка расческой работает намного быстрее, чем bubble или shaker sort, в некоторых ситуациях comb sort работает быстрее quick sort. Но данная сортировка обладает одним очевидным минусом – неустойчивость.

* Сортировка массива вставками (insert sort)

Сортировка вставками (insert sort) – алгоритм сортировки, в котором элементы массива просматриваются по одному, и каждый новый элемент размещается в подходящее место среди ранее упорядоченных элементов.

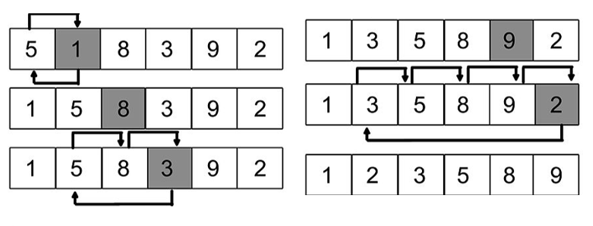
Общая суть сортировки вставками такова:

1)    Перебираются элементы в неотсортированной части массива.

2)    Каждый элемент вставляется в отсортированную часть массива на то место, где он должен находится.

Сортировка вставками делить массив на 2 части – отсортированную и неотсортированную. С каждым новым элементом отсортированная часть будет увеличиваться, а неотсортированная уменьшаться. Причем найти нужное место для очередного элемента в отсортированном массиве достаточно легко.

Рассмотрим самый простой способ (рис. 3.5). Необходимо пройти массив слева направо и обработать каждый элемент. Слева будет наращиваться отсортированная часть массива, а справа – уменьшаться неотсортированная. В отсортированной части массива ищется точка вставки для очередного элемента. Сам элемент отправляется в буфер, что освобождает место в массиве и позволяет сдвинуть элементы и освободить точку вставки.



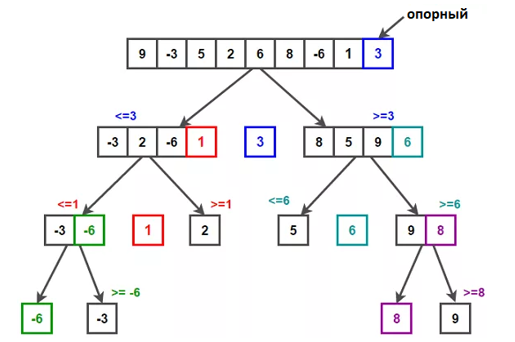
Существует множество модификаций сортировки вставками, некоторые из них затрагивают именно способ вставки элемента в отсортированную часть. Одна из самых лучших модификаций – сортировка простыми вставками с бинарным поиском. Бинарный поиск будет описан позже.

Лучше всего сортировка вставками работает при обработке почти отсортированных массивов. В таком случае insert sort работает быстрее других сортировок.

* Быстрая сортировка массива (quick sort)

Быстрая сортировка (quick sort) – одна из самых быстрых сортировок. Эта сортировка по сути является существенно улучшенной версией алгоритма пузырьковой сортировки.

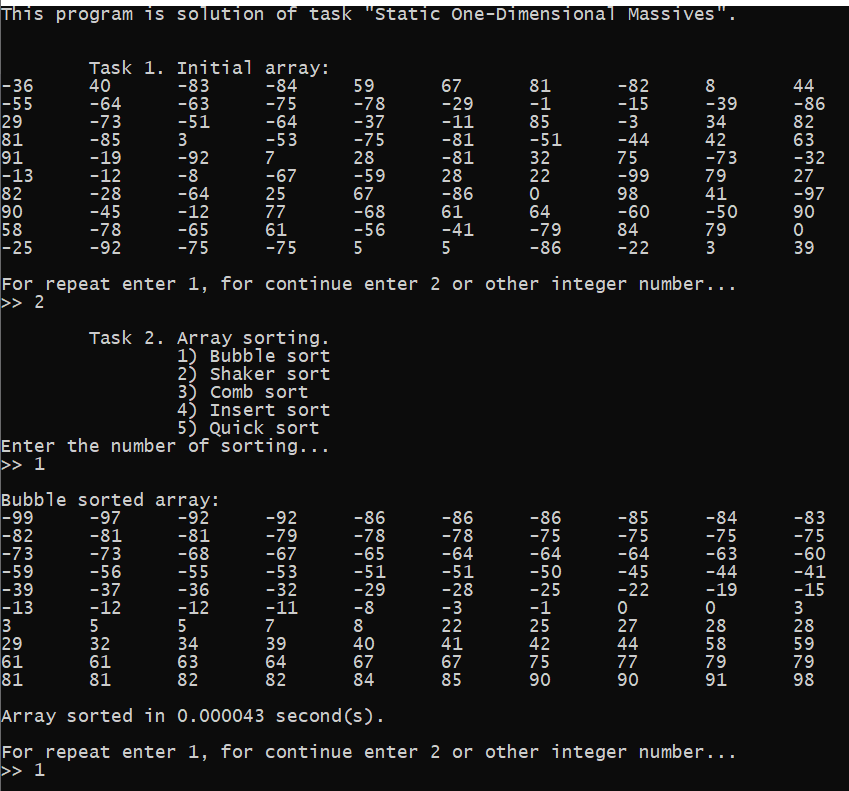
Общая идея алгоритма состоит в том, что сначала выбирается из массива элемент, который называется опорным. От выбора опорного элемента не зависит корректность алгоритма, но в отдельных случаях может сильно зависеть его эффективность. Затем необходимо сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на три непрерывных отрезка, следующие друг за другом: меньше опорного, раны опорному и больше опорного. Для меньших и больших значений необходимо выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

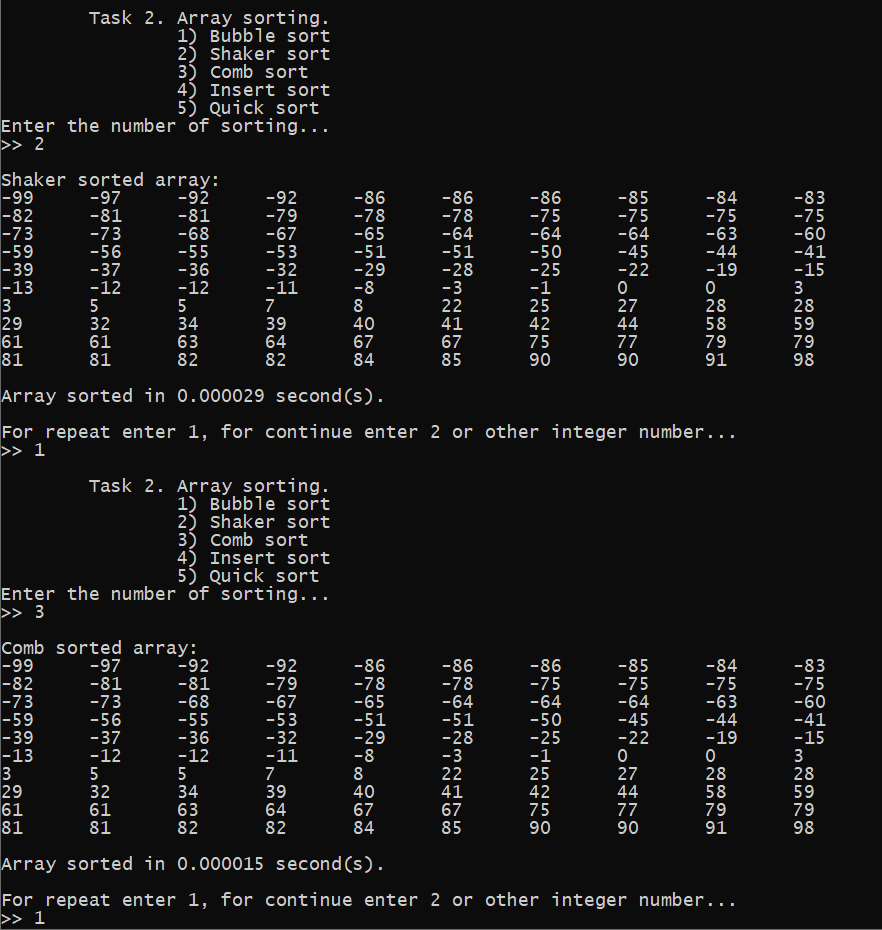


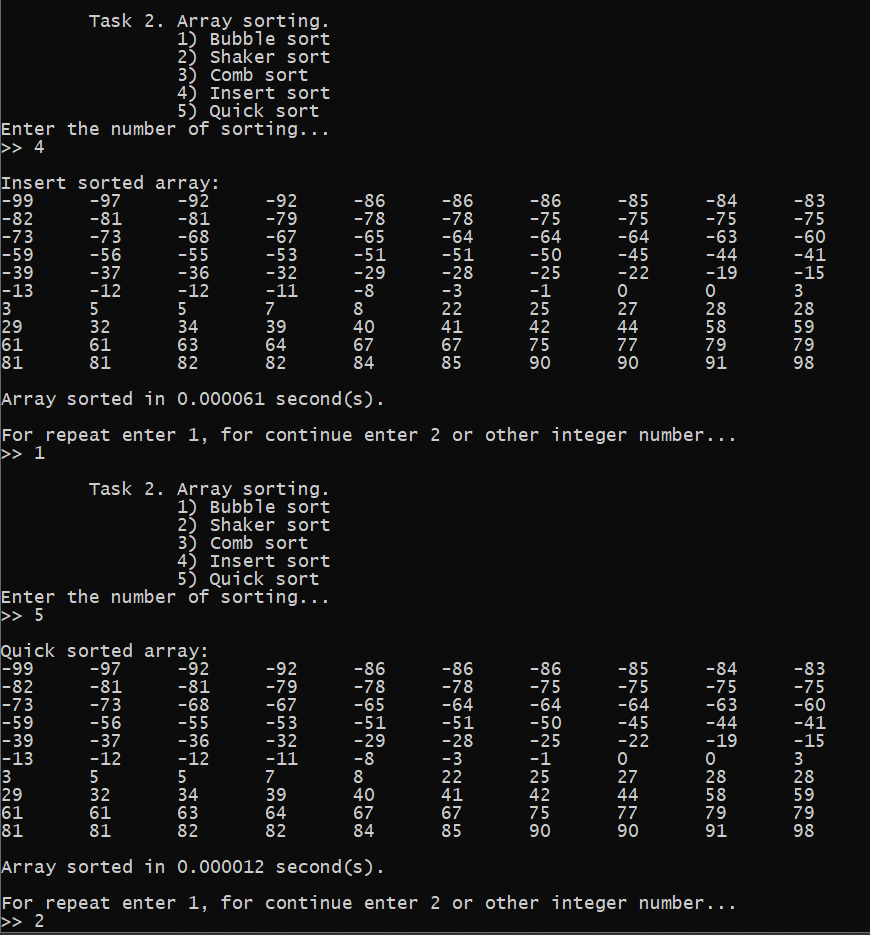
На практике массив обычно делят на две части: «меньше опорного» и «равные и большие» или «меньше опорного или равные» и «большие». Такой поход в общем случае эффективнее, ведь упрощается алгоритм разделения.

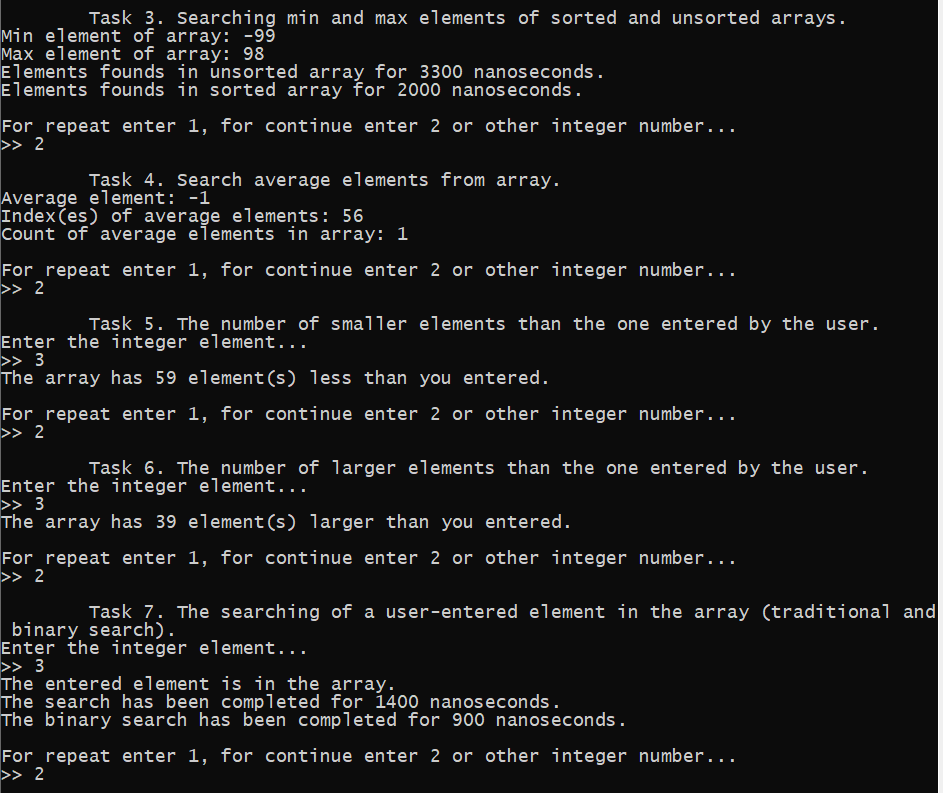
При том, что это один из самых быстродействующих из алгоритмов, данный алгоритм сортировки неустойчив, а прямая реализация в виде функции с двумя рекурсивными вызовами может привести к ошибке переполнения стека.

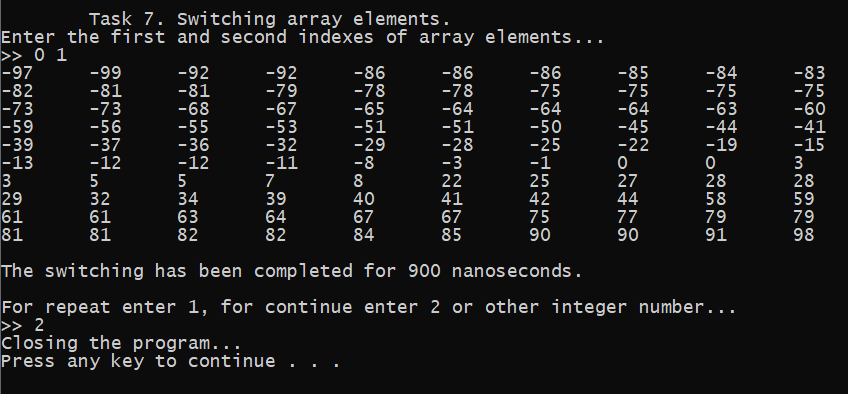
**2.3. Экспериментальные результаты.**









  
Рисунок 2 – Результат работы программы (полный код программы представлен в приложении А)

**3.** Арифметика указателей и матрицы

**3.1. Цель работы.**

Изучить арифметику указателей, указатели массивов и анимацию вывода квадратных матриц в консоли.

Необходимо написать программу, которая:

1)    Используя арифметику указателей, заполняет квадратичную целочисленную матрицу порядка *N* (6,8,10) случайными числами от 1 до  N\*N согласно схемам, приведенным на рисунках. Пользователь должен видеть процесс заполнения квадратичной матрицы.



2)    Получает новую матрицу, из матрицы п. 1, переставляя ее блоки в соответствии со схемами:



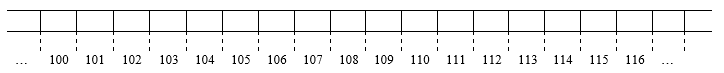
3)    Используя арифметику указателей, сортирует элементы любой сортировкой.

4)    Уменьшает, увеличивает, умножает или делит все элементы матрицы на введенное пользователем число.

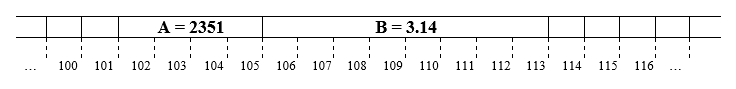
**3.2. Основные теоретические положения.**

Указатели и ссылки являются одними из самых важных и достаточно сложных для понимания и использования средств языка программирования. Они ориентированы на прямую работу с памятью компьютера. С помощью этих средств реализуется работа с динамической памятью и динамическими объектами, возвращение из функций измененных данных и многое другое. К использованию указателей и ссылок мы будем неоднократно возвращаться в последующих разделах.

Все данные (переменные, константы и др.) хранятся в памяти. Память представляет собой непрерывную последовательность ячеек (байтов), каждая из которых имеет свой номер – адрес:



При определении, например, некоторой переменной, она располагается в памяти по определенному адресу и занимает столько ячеек, сколько требует тип этой переменной. Пусть, например, имеется переменные **int A = 2351** и **double B = 3.1** и пусть они располагаются в памяти так:



Говорят, что переменная **А** располагается по адресу 101 и занимает 4 байта, а переменная **B** имеет адрес 105 и занимает 8 байт памяти.

Для получения адреса какого-либо программного объекта используется оператор **&**. Например, если выполнить фрагмент следующей программы (в предположении, что переменные A и B располагаются в памяти, как это показано на предыдущем рисунке):

int A = 2351;

double B = 3.14;

cout <<  “Значение переменной А: ” << A << endl;

cout <<  “Адрес переменной А: ” << &A << endl;

cout <<  “Значение переменной В: ” << В << endl;

cout <<  “Адрес переменной В: ” << &В << endl;

получим следующий результат:

Значение переменной А: 2351

Адрес переменной А: 101

Значение переменной В: 3.14

Адрес переменной В: 105

Правда, значения адресов переменных будут выведены в шестнадцатеричном  формате.

**Указатели** – это тоже обычные переменные, но они **служат для хранения адресов памяти**.

Указатели определяются в программе следующим образом:

**<тип данных> \*<имя переменной>**

 Здесь <**тип данных**> определяет так называемый **базовый тип указателя**.

**<Имя переменной>**является идентификатором переменной-указателя.

Признаком того, что это переменная указатель, является символ \*, располагающийся между базовым типом указателя и именем переменной-указателя.

Например:

int \*p1;

double \*p2;

Здесь определены две переменные-указатели (или просто – два указателя). Указатель **p1** является переменной-указателем на базовый тип **int**(или, как говорят, переменная **p1** указывает  на **int**- значение), а указатель **p2** указывает на **double**– значение.

Иными словами, переменная **p1** предназначена для хранения адресов участков памяти, размер которых соответствует типу **int**(4 байта), а переменная **p2** - для хранения адресов участков памяти, размер которых соответствует типу **double**(8 байт).

Формально указатели представляют собой обычные целые значения типа **int**и занимают в памяти 4 байта не зависимо от базового типа указателя. Значения указателей при их выводе на экран представляются как целые значения в шестнадцатеричном формате.

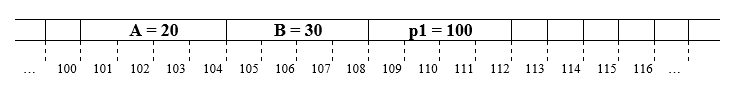
К указателям можно применять некоторые арифметические операции. К таким операциям относятся:  **+**,**-**, **++**, **--**. Результаты выполнения этих операций по отношению к указателям существенно отличаются от результатов соответствующих арифметических операций, выполняющихся с обычными числовыми данными.

Рассмотрим следующий пример:

int A = 20, B = 30;

int \*p1 = &A;

Пусть переменные **A**и **B** расположены в памяти, например, так, как это показано на следующем рисунке:



Указатель **p1** содержит адрес переменной **A**, который равен 100 и \***p1** будет равно значению переменной **A**, то есть 20. Выполним следующую операцию:

p1 = p1 + 1;

или, что то же самое:

p1++;

Значение указателя изменится и станет равным 104, а не 101, как, наверное, ожидалось. То есть теперь указатель ссылается уже на переменную **B**и значение \***p1**будет равно 30.

Таким образом, добавление или вычитание 1 из указателя приводит к изменению его значения на размер базового типа указателя. В общем случае, например, при выполнении следующей операции:

p1 = p1 + N;  //  N – некоторое целое значение

значение указателя увеличится на **sizeof(<базовый тип указателя>) \* N** и в нашем случае это приращение будет равно **sizeof(int) \* N = 4 \* N**. Так, если N = 4, а p1= 100, то значение указателя **p1** увеличится на 16 и станет равно 116, и указатель будет  ссылаться на данные, расположенные по адресу 116.

**Внимание.** Добавлять к указателям или вычитать из указателей можно только целые значения.

Поскольку упомянутые арифметические операции выполняются по-разному при их применении к указателям и обычным арифметическим типам данных, а также учитывая высший приоритет операции \*, при использовании указателей в составе выражений следует внимательно обращаться со скобками. Например, выражения (см. предыдущий рисунок)

\*(p1 + 1)  и \*p1 + 1

имеют совершенно разный смысл. Первое выражение даст значение 30, а второе выражение будет равно 21 (в первом выражении сначала изменяется адрес, а затем осуществляется обращение в память по этому измененному адресу; во втором выражении мы обращаемся по старому адресу и к значению, хранящемуся по этому адресу добавляем 1).

В изучаемых нами языках программирования между массивами и указателями имеется очень тесная связь.

Кода мы определяем в программе некоторый массив, например,

int Arr[10];

переменная **Arr** без индексов представляет собой указатель на первый элемент массива в данном случае из 10 целых чисел (содержит адрес первого элемента массива). Если вывести на экран значение переменной **Arr**

cout << Arr:

мы увидим некоторое целое значение в шестнадцатеричном формате, соответствующее адресу первого элемента этого массива.

**Замечание**. Именно по этой причине в языке C++ отсутствует операция присвоения сразу всех значений одного массива другому (в некоторых других языках, например, в Pascal такая возможность имеется). Действительно, если имеются два массива

int A1[10], A2[10];

то попытка выполнить присвоение **A1 = A2**привела бы к тому, что переменная **A1** стала бы указывать на ту же область памяти, что и переменная **A2** (мы скопировали адрес из **A2**в **A1**, а не содержимое одного массива в другой).  Адрес, который хранился ранее в переменной **A1,** был бы утерян, что привело бы к утечке памяти (для десяти элементов массива **A1** в памяти было выделено место, но теперь мы “забыли”, где оно находится, то есть потеряли память). По этой причине подобные операции с массивами в языке C++ запрещены. Более того, запрещены любые изменения значения переменной массива.

Указателю, имеющему такой же базовый тип, как и элементы массива, можно присвоить массив следующим образом:

int Arr[10];

int \*p;

p = Arr;

Но обратное присвоение выполнить невозможно:

Arr = p;  // Ошибка

Такое присвоение невозможно, поскольку переменная массива – это константа, изменение которой запрещено.

Так как переменная массива является указателем на первый элемент массива, появляются дополнительные возможности по работе с массивами на основе использования арифметики указателей. Например, чтобы получить 5–й элемент массива **Arr**можно воспользоваться одним из следующих выражений:

**Arr[4]**или   \*(**Arr + 4)**или**\*( p + 4)**

Первое выражение – это пример обычной индексации элементов массива. Во втором и третьем выражениях мы использовали арифметику указателей и с помощью операции + получили адрес пятого элемента массива. Затем с помощью операции \* взяли значение по этому адресу и получили значение 5-го элемента массива. Обратите внимание на скобки в этих выражениях, если их не поставить и написать \***Arr + 4**или **\*p + 4**, то эти выражения будут равны значению первого элемента массива увеличенного на 4, так как операция \* имеет больший приоритет, чем операция +.

Вот пример фрагмента программы для работы с массивом с помощью обычной индексации элементов массива. Этот фрагмент обеспечивает ввод элементов целочисленного массива с клавиатуры, вычисление квадратов значений элементов массива, а затем вывод элементов массива на экран:

int A[10];

for (int i = 0; i < 10; ++ i)

{

cin >> A[i];

A[i] = A[i] \* A[i];

}

for (int i = 0; i < 10; ++ i)

cout << A[i] << “  “;

cout << endl;

…..

А вот тот же фрагмент, но с использованием арифметики указателей:

int A[10];

for (int \*Next = A, \*End = Next + 9; Next <= End; ++ Next)

{

cin >> \*Next;

\*Next = \*Next \* \*Next;    // \*Next = (\*Next) \* (\*Next);

}

for (int \*Next = A, \*End = Next + 9; Next <= End; ++ Next)

cout << \*Next << “  “;

cout << endl;

…

Использование арифметики указателей при работе с массивами приводит обычно к уменьшению объема генерируемого кода программы и к уменьшению времени ее выполнения, то есть к увеличению быстродействия.

Поскольку указатель и имя массива, в большой степени, взаимозаменяемы, указатели можно индексировать, как обычные массивы:

int A[10],  \*P  =  A;

for (int i = 0; i < 10; ++ i)

cout << P[i] << “  “;

Можно создавать и массивы указателей. Например:

int a = 1, b = 2, c = 3, \*M[3];

M[0] = & a;  // Элементам массива М присваиваются адреса переменных a, b и c

M[1] = & b;

M[2] = & c;

for  (int i  = 0; i  < 3; ++ i)

cout << \*M[i] << “  ”;

cout << endl;

Массив **M** – это трехэлементный массив указателей на целые значения, то есть каждый элемент этого массива представляет собой указатель на целое.

С помощью массивов указателей можно моделировать различные интересные конструкции данных. Например, пусть имеется квадратная матрица размерности 5 х 5 симметричная относительно главной диагонали. Для ее однозначного представления достаточно хранить в памяти  не все 25 элементов этой матрицы, а только 15 (например, элементы под главной диагональю вместе с элементами главной диагонали). Для этого можно предложить следующую конструкцию:

int A1[1], A2[2], A3[3], A4[4], A5[5], \*A[5] = { A1, A2, A3, A4, A5 };

// Вводим 15 целых значений - элементы под  главной диагональю и диагональные

// элементы матрицы

for (int i = 0; i < 5; ++i)

for (int j = 0; j <= i; ++ j)

cin >> A[i][j];

cout << endl;

// Выводим симметричную матрицу 5 на 5 на экран

for (int i = 0; i < 5; ++i)

{

for (int j = 0; j <= i; ++ j)

cout << A[i][j] << "  ";

for (int j = i + 1; j < 5; ++ j)

cout << A[j][i] << "  ";

cout << endl;

}

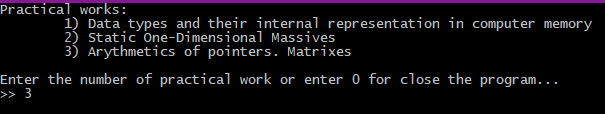
cout << endl;

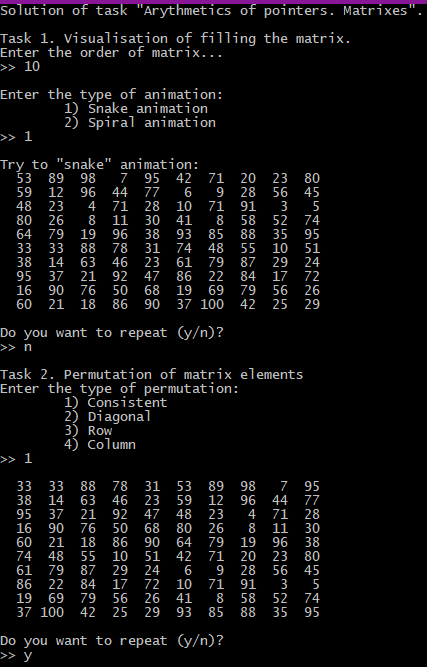
А это пятиэлементный массив указателей на символы, инициализированный некоторыми текстовыми строками:

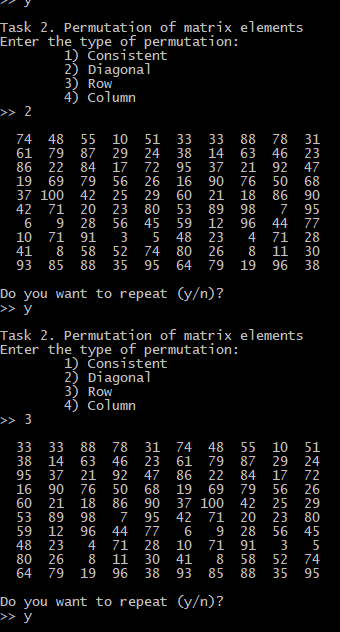
char  \* Words[5]  = { "Слово1", "Слово2", "Слово3", "Слово4", "Слово5" }

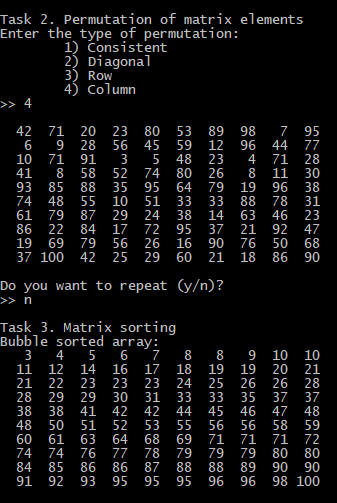
Как это работает: когда компилятор встречает в программе некоторый текст, заключенный в кавычки, в памяти создается символьный массив соответствующей этому тексту длины и адрес этого символьного массива присваивается соответствующему элементу – указателю массива **Words**.

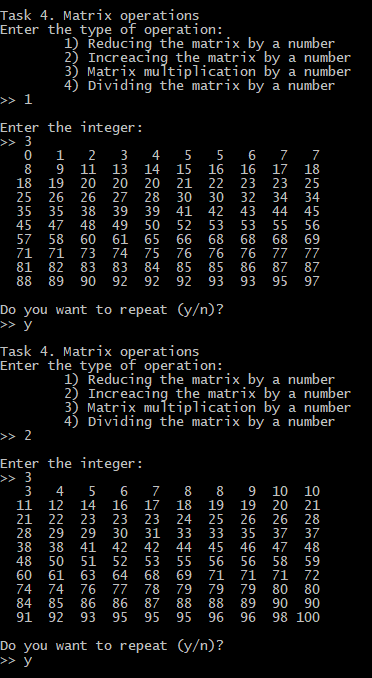
**3.3. Экспериментальные результаты.**

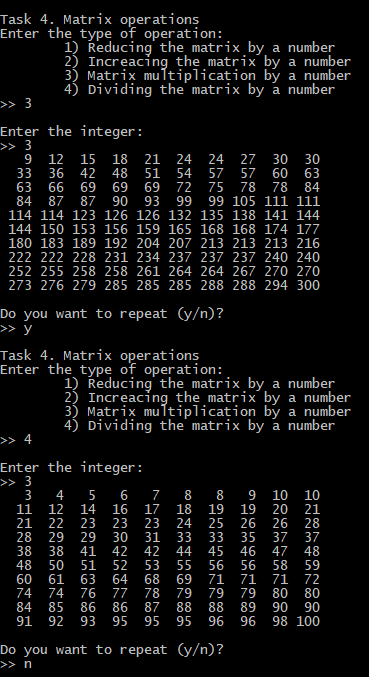


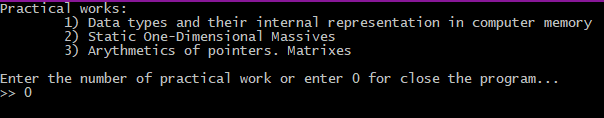










  
Рисунок 3 – Результат работы программы (полный код программы представлен в приложении А)

**4.** Текстовые строки как массивы символов, организация работы с файлами

**4.1. Цель работы.**

Необходимо написать программу, которая реализует поставленную задачу:

1)    С клавиатуры или с файла (\*) (пользователь сам может выбрать способ ввода) вводится последовательность, содержащая от 1 до 50 слов, в каждом из которых от 1 до 10 строчных латинских букв и цифр. Между соседними словами произвольное количество пробелов. За последним символом стоит точка.

2)    Необходимо отредактировать входной текст:

·        удалить лишние пробелы;

·        удалить лишние знаки препинания (под «лишними» подразумевается несколько подряд идущих знаков (обратите внимание, что «…» - корректное использование знака) в тексте);

·        исправить регистр букв, если это требуется (пример некорректного использования регистра букв: пРиМЕр);

3) Вывести на экран слова последовательности в обратном порядке.

4) Вывести на экран  все слова последовательности в две или три колонки (в зависимости от количества слов) с выравниванием слов по правой границе колонки.

5)  Необходимо найти подстроку, которую введёт пользователь в имеющейся строке. Реализуйте два алгоритма: первый алгоритма – Линейный поиск, а второй алгоритм – Бойера-Мура. (\*)

**4.2. Основные теоретические положения.**

Текстовые строки представляются с помощью одномерных массивов символов. В языке C++ текстовая строка представляет собой набор символов, обязательно заканчивающийся нулевым символом (‘\0’). Поэтому, если вы хотите создать текстовый массив для хранения 10 (N) символов, нужно выделить память под 11(N+1) символов.

Объявленный таким образом массив может использоваться для хранения текстовых строк, содержащих не более 10 символов. Нулевой символ позволяет определить границу между содержащимся в строке текстом и неиспользованной частью строки.

При определении строковых переменных их можно инициализировать конкретными значениями с помощью строковых литералов:

char S1[15] = “This is text”;

char S2[] = “Пример текста”;

Последние два элемента переменной  просто не используются, а строка  автоматически подстраивается под длину инициализирующего текста.

При работе со строками можно обращаться к отдельным символам строки как в обычном одномерном массиве с помощью индексов:

cout << S1[0]; // На экране будет выведен символ ‘T’

Если строка формируется при помощи цикла (или иного способа), то необходимо в ее конец обязательно записать нулевой символ '\0'.

* При выводе строк можно использовать форматирование (манипуляторы или функции потока вывода). Вывод текстовых строк на экран крайне простая задача:

char Str[21] = “Это пример текста”;

cout  <<  Str << endl;

cout  <<  “Это текстовый литерал.” << endl;

* Ввод текста с клавиатуры можно осуществлять разными способами, каждый из которых имеет определенные особенности.
* Непосредственное чтение текстовых строк из потока вывода осуществляется до первого знака пробела.
* Такой способ чтения обеспечивает ввод символов до первого пробельного символа (не до конца строки). Остальные символы введенного с клавиатуры остаются в потоке ввода и могут быть прочитаны из него следующими операторами >>.
* Для того чтобы прочесть всю строку полностью, можно воспользоваться одной из функций gets или gets\_s (для этого в программу должен быть включен заголовочный файл <stdio.h>).
* Функция gets имеет один параметр, соответствующий массиву символов, в который осуществляется чтение. Вторая функция (gets\_s) имеет второй параметр, задающий максимальную длину массива символов .
* Ввод текста, длина которого (вместе с нулевым символом) превышает значение второго параметра (то есть длины символьного массива ), приводит к возникновению ошибки при выполнении программы
* Предпочтительно использование функции потока ввода cin.getline:

const int N = 21;

char Str [N];

cin.getline (Str, N);      // Пусть введена строка “Это пример текста”

cout  <<  Str << endl;  // На экран будет выведено “ Это пример текста”

* Если длина введенного с клавиатуры текста превышает максимальную длину массива , в него будет записано (в нашем примере) 20 символов вводимого текста и нулевой символ. Остальные символы введенного текста остаются во входном потоке и могут быть взяты из него следующими инструкциями ввода.
* Функция cin.getline может иметь третий параметр, задающий символ, при встрече которого чтение строки из потока прекращается:

cin.getline (Str, N,  ‘.’);

* Иногда чтение из потока невозможно (например, попытка считать слишком длинный текст). Для того чтобы продолжить чтение из потока, необходимо восстановить его нормальное состояние. Этого можно достигнуть с помощью функции потока cin.clear(), которая сбрасывает состояние потока в нормальное. Если забирать остатки данных из потока ввода не надо, то следует очистить его с помощью функции cin.sync().

Класс string предназначен для работы со строками типа char, которые представляют собой строчку с завершающим нулем (символ ‘\0’). Класс string был введен как альтернативный вариант для работы со строками типа char.

Чтобы использовать возможности класса string, нужно подключить библиотеку <string> и пространство имен std. Объявление же переменной типа string осуществляется схоже с обычной переменной:

string S1; // Переменная с именем s1 типа string

string S2 = “Пример”; // объявление с инициализацией

Создание нового типа string было обусловлено недостатками работы с строками символов, который показывал тип char. В сравнении с этим типом string имеет ряд основных преимуществ:

·        возможность использования для обработки строк стандартные операторы С++(=,+,<,==,>,+=,!=,<=,>=,[])(=,+,<,==,>,+=,!=,<=,>=,[]). Использование типа char приводило требовало написание чрезмерного программного кода;

·        обеспечение лучшей надежности программного кода;

·        обеспечение строки, как самостоятельного типа данных.

/\* string.cpp: Этот файл содержит функцию "main". Здесь начинается и заканчивается выполнение программы \*/

#include <iostream>

#include <string>

using namespace std;

int main()

{

setlocale(LC\_ALL, "Russian");

string A = "Пример"; // Объявляем и инициализируем строку А

string B; // Объявляем строку В

string C = "текста"; // Объявляем и инициализируем строку С

B = A; // Копируем текст из строки А в строку В

C = B + " " + C;

if(C == "Пример текста") // Проверяем корректно ли произошло присоединение строк

cout << C;

return 0;

}

Результат выполнения программы – вывод в консоль сообщения:

Пример текста

С string можно использовать оператор индексации и получать значения символа. Принцип действия такой же, как и у типа char:

string s1 = “hello!”;

cout << s1[0]; // Будет выведен символ ‘h’

Класс string обладает широким функционалом:

·        функция compare() сравнивает одну часть строки с другой;

·        функция length() определяет длину строки;

·        функции find() и rfind() служат для поиска подстроки в строке (отличаются функции лишь направлением поиска);

·        функция erase() служит для удаления символов;

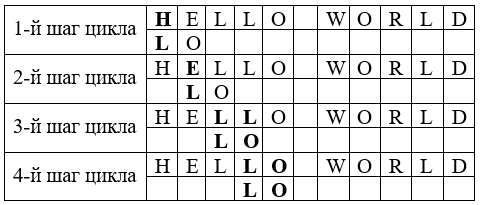
·        функция replace() выполняет замену символов;

·        функция insert() необходима, чтобы вставить одну строку в заданную позицию другой строки;

Но весь функционал string накладывает и свой негативный отпечаток. Основным недостатком string в сравнении с типом char является замедленная скорость обработки данных.

При работе со строками часто будет возникать потребность в поиске набора символа или слов (поиска подстроки в строке). При условии, что текст может быть крайне большим, хочется, чтобы алгоритм поиска подстроки работал быстро.

Самый простой способ подстроки в строке – Линейный поиск – циклическое сравнение всех символов строки с подстрокой. Действительно, этот способ первый приходит в голову, но очевидно, что он будет самым долгим.



На первых двух итерациях цикла сравниваемые буквы не будут совпадать. На третьей же итерации, совпал символ ‘L’, это означает, что теперь нужно сравнивать следующий символ подстроки со следующим символом строки. Видно, что символы отличаются, поэтому алгоритм продолжает свою работу. На четвертой же итерации подстрока была найдена.

Если представить, что исходная строка непорядок больше и подстрока находится в конце строки (или вовсе отсутствует), то сразу видны минусы данного алгоритма.

Одним из самых популярных алгоритмов, который работает быстрее, чем приведенный выше алгоритм, является алгоритм Кнута-Морриса-Пратта (КМП). Идея заключается в том, что не нужно проходить и сравнивать абсолютно все символы строки, если известны символы, которые есть и в строке, и в подстроке.

Суть алгоритма: дана подстрока S и строка T. Требуется определить индекс, начиная с которого образец S содержится в строке T. Если S не содержится в T, необходимо вернуть индекс, который не может быть интерпретирован как позиция в строке.



Хоть алгоритм и работает быстрее, по-прежнему необходимо сначала пройти всю строку, чтобы определить префиксы или суффиксы (вхождение (индексы) символов).

Алгоритм Бойера-Мура в отличие от КМП полностью не зависим и не требует заранее проходить по строке. Этот алгоритм считается наиболее быстрым среди алгоритмов общего назначения, предназначенных для поиска подстроки в строке.

Преимущество этого алгоритма в том, что ценной некоторого количества предварительных вычислений над подстрокой (но не над исходной строкой, в которой ведётся поиск), подстрока сравнивается с исходным текстом не во всех позициях (пропускаются позиции, которые точно не дадут положительный результат).

Поиск подстроки ускоряется благодаря созданию таблиц сдвигов. Сравнение подстроки со строки начинается с последнего символа подстроки, а затем происходит прыжок, длина которого определяется по таблице сдвигов. Таблица сдвигов строится по подстроке так чтобы перепрыгнуть максимальное количество символов строки и не пропустить вхождение подстроки в строку.

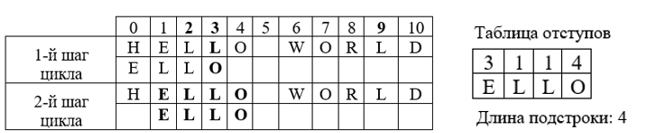
Правила построения таблицы сдвигов:

1)    Значение элемента таблицы равно удаленности соответствующего символа от конца шаблона (подстроки).

2)    Если символ встречается более одного раза, то применятся значение, соответствующее символу, наиболее близкому к концу шаблона.

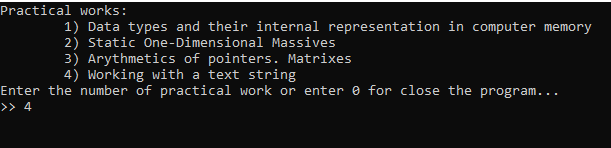
3)    Если символ в конце шаблона встречается 1 раз, ему соответствует значение, равное длине образа; если более одного раза – значение, соответствующее символу, наиболее близкому к концу образа.

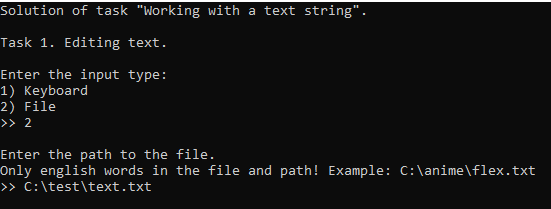
4)    Для символов, отсутствующих в образе, применяется значение, равное длине шаблона.

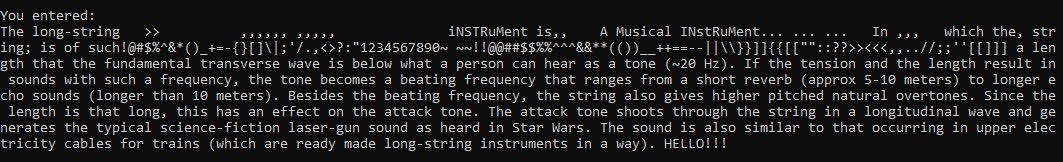


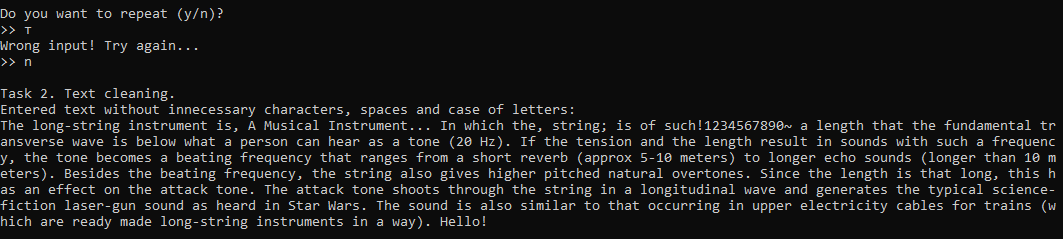
Сначала была построена таблица отступов и подсчитана длина подстроки. Затем начинается алгоритм поиска подстроки в строке. Сравнивает символ ‘L’ строки и ‘O’ подстроки. Элементы не совпадают, поэтому необходимо определить длину отступа. Символ ‘L’ присутствует в таблице отступа, длина отступа равняется 1. Подстрока смещается на 1 символ вперед. На следующей итерации подстрока найдена.

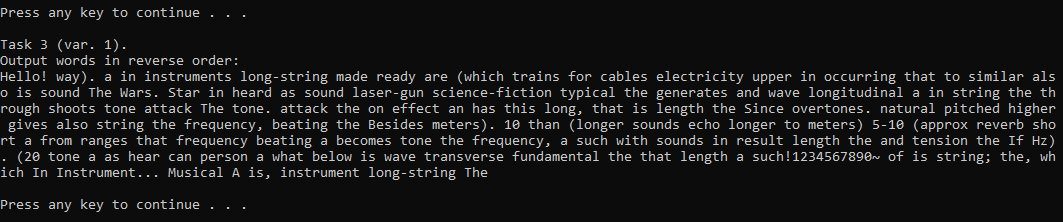
**4.3. Экспериментальные результаты.**

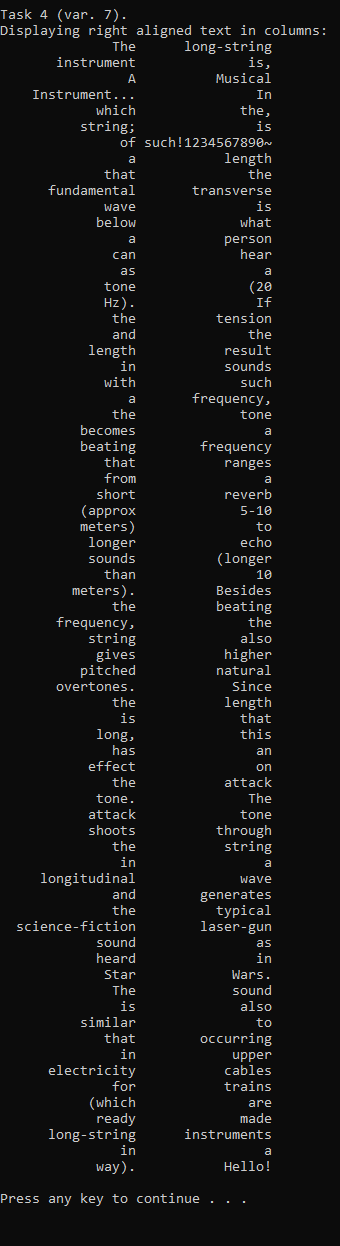


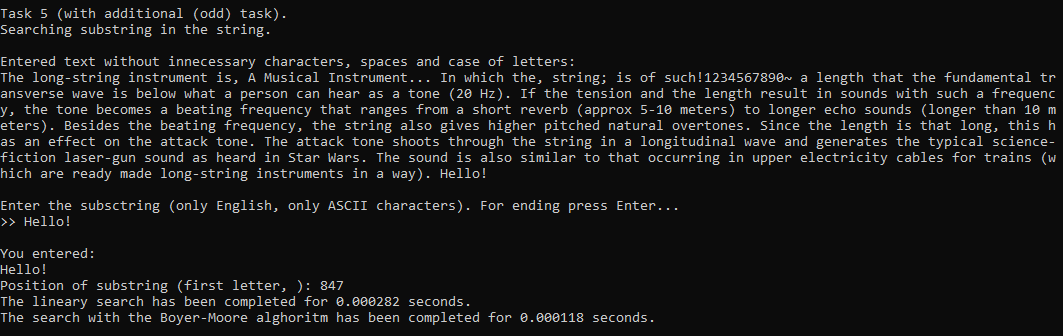


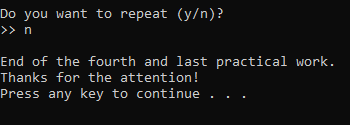










  
Рисунок 5 – Результат работы программы (полный код программы представлен в приложении А)

**заключение**

Лабораторные работы объединены в единую программу с помощью интерфейса, результат продемонстрирован и может быть получен вновь с помощью кода из Git репозитория.

**список использованных источников**

1. [Курс на ресурсе Stepik.org ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ С++](https://stepik.org/course/82772)

**приложение А**

**Код программы**

#include <iostream>

#include <ctime> // Нужно для рандома

#include <chrono> // Нужно для засечения времени

#include <Windows.h> // Нужно для получения координат консоли и их изменения

#include <iomanip> // нужно для std::setw

#include <string>

#include <fstream> // нужно для чтения файлов

using namespace std::chrono; // Нужно для засечения времени

using namespace std;

void practicalWork1();

void practicalWork2();

void practicalWork3();

void practicalWork4();

// Функции, использующиеся в первой практической работе

void intToBinary(int);

void floatToBinary(int);

void doubleToBinary(int, int);

int readInt();

// Функции, использующиеся во второй практической работе

void createAndPrintRandomArray(int[]);

void bubbleSort(int[]);

void shakerSort(int[]);

void combSort(int[]);

void insertSort(int[]);

void quickSort(int[], int, int);

void Merge(int[], int, int, int, int[]);

void InternalMergeSort(int[], int, int, int[]);

void MergeSort(int[], int, int);

int getMinArrayElement(int[]);

int getMaxArrayElement(int[]);

int binarySearch(int[], int, int, int);

void copyArray(int[], int[]);

void printArray(int[]);

bool choiseNextAction();

float stopSecondsTimer(time\_point<steady\_clock>);

long long stopNanoSecondsTimer(time\_point<steady\_clock>);

// Функции, использующиеся в третьей практической работе

void snakeAnimation(int[], int, short cellSize = 4, short delay = 100);

void spiralAnimation(int[], int, short cellSize = 4, short delay = 100);

void consistentPermutation(int[], int);

void diagonalPermutation(int[], int);

void rowPermutation(int[], int);

void columnPermutation(int[], int);

// Функции, использующиеся в четвёртой практической работе

char \* readFile(string);

string textCleaner(string);

bool isSymbol(char);

string deleteChar(string, int, int);

void reverseOutput(string);

int countOfWords(string);

void wordOutput(string, int, int);

int longestWord(string);

int linearSubscringSearch(string, string);

int bouyerMooreSubstringSearch(string, string);

void badCharHeuristic(string, int, int[]);

int main()

{

while (true) {

system("CLS");

cout << "Practical works: \n"

<< "\t1) Data types and their internal representation in computer memory \n"

<< "\t2) Static One-Dimensional Massives \n"

<< "\t3) Arythmetics of pointers. Matrixes \n"

<< "\t4) Working with a text string \n"

<< "Enter the number of practical work or enter 0 for close the program... \n>> ";

int input;

cin >> input;

switch (input) {

case 1:

system("CLS");

practicalWork1();

break;

case 2:

system("CLS");

practicalWork2();

break;

case 3:

system("CLS");

practicalWork3();

break;

case 4:

system("CLS");

practicalWork4();

break;

default:

goto Exit;

}

}

Exit:

cout << "\nClosing the program... \n";

system("pause");

return 0;

}

void practicalWork1() {

cout << "Solution of task \"Data types and their internal representation in computer memory\". \n\n";

do {

cout << "\tTask 1. Size of data types. \n"

<< "int: \t\t" << sizeof(int) << " bytes. \n"

<< "short int \t" << sizeof(short int) << " bytes. \n"

<< "long int \t" << sizeof(long int) << " bytes. \n"

<< "float \t\t" << sizeof(float) << " bytes. \n"

<< "double \t\t" << sizeof(double) << " bytes. \n"

<< "long double \t" << sizeof(long double) << " bytes. \n"

<< "char \t\t" << sizeof(char) << " bytes. \n"

<< "bool \t\t" << sizeof(bool) << " bytes. \n";

} while (choiseNextAction());

system("CLS");

cout << "\tTask 2. Binary representation of integer number. \n\n";

int input;

do {

cout << "Enter the integer number... \n>> ";

// Считывание числа с проверкой на принадлежность к целым числам.

input = readInt();

cout << "Binary representation of number " << input << ": ";

intToBinary(int(input));

} while (choiseNextAction());

system("CLS");

cout << "\tTask 3. Binary representation of float number. \n";

union {

float inputFloat;

int inputInt1;

};

do {

cout << "Enter the float number... \n>> ";

cin >> inputFloat;

cout << "Binary representation of number " << inputFloat << ": ";

floatToBinary(inputInt1);

} while (choiseNextAction());

system("CLS");

cout << "\tTask 4. Binary representation of double number. \n";

union {

double inputDouble;

int inputInt2[2] = { NULL, NULL };

};

do {

cout << "Enter the double number... \n>> ";

cin >> inputDouble;

cout << "Binary representation of number " << inputDouble << ":\n";

doubleToBinary(inputInt2[1], inputInt2[0]);

} while (choiseNextAction());

cout << "\nEnd of the first practical work. \n";

system("pause");

}

void intToBinary(int value) {

int mask = 1 << 31; // Маска представляет собой 2^32, то есть 10...00, где нулей 31.

for (int i = 1; i <= 32; i++) {

/\*

Побитовое сравнение введённого числа с маской.

Если у числа и маски есть хотя бы одна общая единица (в данном случае бОльший бит числа), то выводится "1", иначе "0".

Далее побитовый сдвиг введённого числа влево и так 32 раза (именно столько бит определяется под int).

Например:

0 1000000 00000000 00000000 00000000 - двоичное представление в памяти компьютера введённого числа 2^31

1 0000000 00000000 00000000 00000000 - маска (2^32)

Вывод: 0

Затем побитовый сдвиг введённого числа влево:

1 0000000 00000000 00000000 00000000

1 0000000 00000000 00000000 00000000 - маска (2^32)

Вывод: 1

Далее следуют нули, в выводе: 0 1000000 00000000 00000000 00000000

\*/

putchar(value & mask ? '1' : '0');

value <<= 1;

if (i % 8 == 0 || i == 1)

putchar(' '); // отделение друг от друга байтов, содержащих значение числа, а также знакового бита

}

}

void floatToBinary(int value) {

int mask = 1 << 31;

for (int i = 1; i <= 32; i++) {

putchar(value & mask ? '1' : '0');

value <<= 1;

if (i == 1 || i == 9)

putchar(' '); // отделение друг от друга битов мантиссы и порядка, а также знакового бита

}

}

void doubleToBinary(int firstPart, int secondPart) {

int mask = 1 << 31;

for (int i = 1; i <= 32; i++) {

putchar(firstPart & mask ? '1' : '0');

firstPart <<= 1;

if (i == 1 || i == 12)

putchar(' '); // отделение друг от друга битов мантиссы и порядка, а также знакового бита

}

for (int i = 1; i <= 32; i++) {

putchar(secondPart & mask ? '1' : '0');

secondPart <<= 1;

}

}

int readInt() {

double input;

while (true) {

cin >> input;

// Проверка на целое число

if (static\_cast<int>(input) != input) {

cout << "Need enter the integer number! Try again...\n";

continue;

}

return int(input);

}

}

const int sizeOfArray = 100;

void practicalWork2() {

srand((unsigned)time(NULL));

time\_point<steady\_clock> startTimer = steady\_clock::now();

float stopTimeInSeconds = stopSecondsTimer(startTimer);

long long stopTimeInNanoSeconds = stopNanoSecondsTimer(startTimer);

int count = 0;

cout << "Solution of task \"Static One-Dimensional Massives\". \n\n";

// РЕШЕНИЕ ПЕРВОГО ЗАДАНИЯ

int mainArray[sizeOfArray];

do {

cout << "\tTask 1. Initial array:" << endl;

createAndPrintRandomArray(mainArray);

} while (choiseNextAction());

do { // РЕШЕНИЕ ВТОРОГО ЗАДАНИЯ

cout << "\n\tTask 2. Array sorting." << endl

<< "\t\t1) Bubble sort" << endl

<< "\t\t2) Shaker sort" << endl

<< "\t\t3) Comb sort" << endl

<< "\t\t4) Insert sort" << endl

<< "\t\t5) Quick sort" << endl

<< "\t\t6) Merge sort" << endl

<< "Enter the number of sorting... \n>> ";

int input;

cin >> input;

int copiedArray[sizeOfArray];

copyArray(mainArray, copiedArray); // Копирование массива в другой массив для возможности в дальнейшем на том же массиве проверить другие типы сортировки.

switch (input) {

case 1: // РЕШЕНИЕ 2.1 ЗАДАНИЯ (bubble sort)

startTimer = steady\_clock::now(); // Старт отсчёта времени

bubbleSort(copiedArray);

stopTimeInSeconds = stopSecondsTimer(startTimer);

cout << "\nBubble sorted array: \n";

printArray(copiedArray);

cout << "Array sorted in " << fixed << stopTimeInSeconds << " second(s). \n";

break;

case 2: // РЕШЕНИЕ 2.2 ЗАДАНИЯ (shaker sort)

startTimer = steady\_clock::now();

shakerSort(copiedArray);

stopTimeInSeconds = stopSecondsTimer(startTimer);

cout << "\nShaker sorted array: \n";

printArray(copiedArray);

cout << "Array sorted in " << fixed << stopTimeInSeconds << " second(s). \n";

break;

case 3: // РЕШЕНИЕ 2.3 ЗАДАНИЯ (comb sort)

startTimer = steady\_clock::now();

combSort(copiedArray);

stopTimeInSeconds = stopSecondsTimer(startTimer);

cout << "\nComb sorted array: \n";

printArray(copiedArray);

cout << "Array sorted in " << fixed << stopTimeInSeconds << " second(s). \n";

break;

case 4: // РЕШЕНИЕ 2.4 ЗАДАНИЯ (insert sort)

startTimer = steady\_clock::now();

insertSort(copiedArray);

stopTimeInSeconds = stopSecondsTimer(startTimer);

cout << "\nInsert sorted array: \n";

printArray(copiedArray);

cout << "Array sorted in " << fixed << stopTimeInSeconds << " second(s). \n";

break;

case 5: // РЕШЕНИЕ 2.5 ЗАДАНИЯ (quick sort)

startTimer = steady\_clock::now();

quickSort(copiedArray, sizeOfArray - 1, 0);

stopTimeInSeconds = stopSecondsTimer(startTimer);

cout << "\nQuick sorted array: \n";

printArray(copiedArray);

cout << "Array sorted in " << fixed << stopTimeInSeconds << " second(s). \n";

break;

case 6: // РЕШЕНИЕ ДОПОЛНИТЕЛЬНОГО ЗАДАНИЯ (merge sort)

startTimer = steady\_clock::now();

MergeSort(copiedArray, 0, sizeOfArray - 1);

stopTimeInSeconds = stopSecondsTimer(startTimer);

cout << "\nMerge sorted array: \n";

printArray(copiedArray);

cout << "Array sorted in " << fixed << stopTimeInSeconds << " second(s). \n";

break;

}

} while (choiseNextAction());

int copiedArray[sizeOfArray]; // Создание отсортированного массива, с которым будет производиться работа в дальнейших заданиях

copyArray(mainArray, copiedArray);

quickSort(copiedArray, sizeOfArray - 1, 0);

do { // РЕШЕНИЕ ТРЕТЬЕГО ЗАДАНИЯ

cout << "\n\tTask 3. Searching min and max elements of sorted and unsorted arrays. \n";

startTimer = steady\_clock::now(); // Поиск в неотсортированном массиве

int max = getMaxArrayElement(mainArray);

int min = getMinArrayElement(mainArray);

stopTimeInNanoSeconds = stopNanoSecondsTimer(startTimer);

cout << "Min element of array: " << min << endl

<< "Max element of array: " << max << endl

<< "Elements founds in unsorted array for " << stopTimeInNanoSeconds << " nanoseconds. \n";

startTimer = steady\_clock::now(); // Поиск в отсортированном массиве

max = getMaxArrayElement(copiedArray);

min = getMinArrayElement(copiedArray);

stopTimeInNanoSeconds = stopNanoSecondsTimer(startTimer);

//cout << "Min: " << min << "\tMax: " << max << endl; // Отладка

cout << "Elements founds in sorted array for " << stopTimeInNanoSeconds << " nanoseconds. \n";

} while (choiseNextAction());

do { // РЕШЕНИЕ ЧЕТВЁРТОГО ЗАДАНИЯ

cout << "\n\tTask 4. Search average elements from array. \n";

int averageElement = int(abs(getMaxArrayElement(copiedArray)) - abs(getMinArrayElement(copiedArray)));

cout << "Average element: " << averageElement << endl;

bool isExists = false; // Проверка на то, существует ли вообще в массиве подходящий средний элемент

for (int i = 0; i < sizeOfArray; i++)

if (copiedArray[i] == averageElement) {

isExists = true;

break;

}

if (isExists) {

cout << "Index(es) of average elements: ";

int count = 0;

for (int i = 0; i < sizeOfArray; i++)

if (copiedArray[i] == averageElement) {

cout << i << " ";

count++;

}

cout << "\nCount of average elements in array: " << count << endl;

}

else

cout << "Element not found in array. \n";

} while (choiseNextAction());

do { // РЕШЕНИЕ ПЯТОГО ЗАДАНИЯ

cout << "\n\tTask 5. The number of smaller elements than the one entered by the user. \n"

<< "Enter the integer element...\n>> ";

int input;

cin >> input;

int count = 0;

for (int element : copiedArray)

if (element < input)

count++;

cout << "The array has " << count << " element(s) less than you entered. \n";

} while (choiseNextAction());

do { // РЕШЕНИЕ ШЕСТОГО ЗАДАНИЯ

cout << "\n\tTask 6. The number of larger elements than the one entered by the user. \n"

<< "Enter the integer element...\n>> ";

int input;

cin >> input;

int count = 0;

for (int element : copiedArray)

if (element > input)

count++;

cout << "The array has " << count << " element(s) larger than you entered. \n";

} while (choiseNextAction());

do { // РЕШЕНИЕ СЕДЬМОГО ЗАДАНИЯ

cout << "\n\tTask 7. The searching of a user-entered element in the array (traditional and binary search). \n"

<< "Enter the integer element...\n>> ";

int input;

cin >> input;

startTimer = steady\_clock::now();

bool isExists = false;

for (int element : copiedArray) // Поиск обычным перебором

if (element == input) {

isExists = true;

break;

}

stopTimeInNanoSeconds = stopNanoSecondsTimer(startTimer);

if (isExists)

cout << "The entered element is in the array. \n";

else

cout << "The entered element is not in the array. \n";

cout << "The search has been completed for " << stopTimeInNanoSeconds << " nanoseconds. \n";

startTimer = steady\_clock::now();

int searchedElement = binarySearch(copiedArray, input, 0, sizeOfArray - 1);

stopTimeInNanoSeconds = stopNanoSecondsTimer(startTimer);

/\* Отладка

if (searchedElement != -1)

cout << "Searched element (binary search): " << searchedElement << endl;

else cout << "Element not searched. \n"; \*/

cout << "The binary search has been completed for " << stopTimeInNanoSeconds << " nanoseconds. \n";

} while (choiseNextAction());

do { // РЕШЕНИЕ ВОСЬМОГО ЗАДАНИЯ

cout << "\n\tTask 8. Switching array elements. \n"

<< "Enter the first and second indexes of array elements...\n>> ";

int input1, input2;

cin >> input1 >> input2;

startTimer = steady\_clock::now();

//if (((input1 && input2) >= 0) && ((input1 && input2) < sizeOfArray)) { // Ошибка компилятора о небезопасном условии. Разобраться, почему.

if ((input1 >= 0) && (input2 >= 0) && (input1 < sizeOfArray) && (input2 < sizeOfArray)) {

int t = copiedArray[input1];

copiedArray[input1] = copiedArray[input2];

copiedArray[input2] = t;

}

else {

cout << "Wrong input! Try again...\n\n>> ";

continue;

}

stopTimeInNanoSeconds = stopNanoSecondsTimer(startTimer);

printArray(copiedArray); // Отладка

cout << "The switching has been completed for " << stopTimeInNanoSeconds << " nanoseconds. \n";

} while (choiseNextAction());

cout << "\nEnd of the second practical work. \n";

system("pause");

}

void createAndPrintRandomArray(int inputArray[]) {

for (int i = 0; i < sizeOfArray; i++) {

inputArray[i] = -99 + rand() % 199; // В массив пишутся случайные целые числа диапазона -99..99 включительно. При желании данную функцию можно расширить возможностью выбирать граничные значения.

cout << inputArray[i] << "\t";

if ((i + 1) % 10 == 0) // Вывод по 10 элементов для удобочитаемости

cout << endl;

}

}

void bubbleSort(int inputArray[]) {

bool swapped;

do {

swapped = false;

for (int i = 0; i < sizeOfArray - 1; i++) {

if (inputArray[i] > inputArray[i + 1]) { // Соседние элементы массива меняются местами

int t = inputArray[i];

inputArray[i] = inputArray[i + 1];

inputArray[i + 1] = t;

swapped = true;

}

}

} while (swapped); // Если за последний проход цикла никакие элементы не поменялись местами, цикл прерывается.

}

void shakerSort(int inputArray[]) {

bool swapped;

do {

swapped = false;

for (int i = 0; i < sizeOfArray - 1; i++)

if (inputArray[i] > inputArray[i + 1]) {

int t = inputArray[i];

inputArray[i] = inputArray[i + 1];

inputArray[i + 1] = t;

swapped = true;

}

for (int i = sizeOfArray - 1; i > 0; i--)

if (inputArray[i] < inputArray[i - 1]) {

int t = inputArray[i];

inputArray[i] = inputArray[i - 1];

inputArray[i - 1] = t;

swapped = true;

}

} while (swapped);

}

void combSort(int inputArray[]) {

int swap;

float k = 1.247F, sortingRange = sizeOfArray - 1;

while (sortingRange >= 1)

{

for (int i = 0; i + sortingRange < sizeOfArray; i++)

{

if (inputArray[i] > inputArray[int(i + sortingRange)])

{

swap = inputArray[int(i + sortingRange)];

inputArray[int(i + sortingRange)] = inputArray[i];

inputArray[i] = swap;

}

}

sortingRange /= k;

}

bool swapped;

do {

swapped = false;

for (int i = 0; i < sizeOfArray - 1; i++) {

if (inputArray[i] > inputArray[i + 1]) { // Соседние элементы массива меняются местами

int t = inputArray[i];

inputArray[i] = inputArray[i + 1];

inputArray[i + 1] = t;

swapped = true;

}

}

} while (swapped == true); // Если за последний проход цикла никакие элементы не поменялись местами, цикл прерывается.

}

void insertSort(int inputArray[]) {

int extractedElement, currentPosition;

for (int i = 1; i < sizeOfArray; i++)

{

extractedElement = inputArray[i];

currentPosition = i;

while ((currentPosition >= 1) && (extractedElement < inputArray[currentPosition - 1]))

{

inputArray[currentPosition] = inputArray[currentPosition - 1];

currentPosition--;

}

inputArray[currentPosition] = extractedElement;

}

}

void quickSort(int inputArray[], int end, int begin) {

int middleElement;

int f = begin;

int l = end;

middleElement = inputArray[(f + l) / 2]; // Определяется опорный элемент

while (f < l) { // Массив сортируется так, что по окончании цикла будет представлять собой две части: элементы меньше либо равные опорному (левая), элементы больше либо равные опорному (правая)

while (inputArray[f] < middleElement) f++;

while (inputArray[l] > middleElement) l--;

if (f <= l)

{

int swap = inputArray[f];

inputArray[f] = inputArray[l];

inputArray[l] = swap;

f++;

l--;

}

}

if (begin < l) quickSort(inputArray, l, begin); // Где l - правая граница левой части разделённого пополам массива

if (f < end) quickSort(inputArray, end, f); // Где f - левая граница правой части разделённого пополам массива

}

void Merge(int array[], int first, int middle, int last, int temp[])

{

int idx = first;

int begin1 = first, end1 = middle;

int begin2 = middle + 1, end2 = last;

for (; begin1 <= end1 && begin2 <= end2; )

{

if (array[begin1] < array[begin2])

temp[idx++] = array[begin1++];

else

temp[idx++] = array[begin2++];

}

for (; begin1 <= end1; )

temp[idx++] = array[begin1++];

for (; begin2 <= end2; )

temp[idx++] = array[begin2++];

for (idx = first; idx <= last; idx++)

array[idx] = temp[idx];

}

void InternalMergeSort(int array[], int first, int last, int buffer[])

{

if (first < last)

{

int m = (first + last) / 2;

InternalMergeSort(array, first, m, buffer);

InternalMergeSort(array, m + 1, last, buffer);

Merge(array, first, m, last, buffer);

}

}

void MergeSort(int array[], int first, int last)

{

int \*buffer = new int[sizeOfArray];

InternalMergeSort(array, 0, sizeOfArray - 1, buffer);

delete[] buffer;

}

int getMinArrayElement(int inputArray[]) {

int minElement = INT\_MAX;

for (int i = 0; i < sizeOfArray; i++)

if (inputArray[i] < minElement)

minElement = inputArray[i];

return minElement;

}

int getMaxArrayElement(int inputArray[]) {

int maxElement = INT\_MIN;

for (int i = 0; i < sizeOfArray; i++)

if (inputArray[i] > maxElement)

maxElement = inputArray[i];

return maxElement;

}

int binarySearch(int inputArray[], int inputElement, int lowPos, int highPos) {

while (lowPos <= highPos) {

int mid = lowPos + (highPos - lowPos) / 2;

if (inputArray[mid] == inputElement)

return mid;

if (inputArray[mid] < inputElement)

lowPos = mid + 1;

else

highPos = mid - 1;

}

return -1;

}

void copyArray(int originalArray[], int resultArray[]) {

for (int i = 0; i < sizeOfArray; i++)

resultArray[i] = originalArray[i];

}

void printArray(int inputArray[]) {

for (int i = 0; i < sizeOfArray; i++) {

cout << inputArray[i] << "\t";

if ((i + 1) % 10 == 0) // Вывод по 10 элементов для удобочитаемости

cout << endl;

}

cout << endl;

}

bool choiseNextAction() {

cout << "\nDo you want to repeat (y/n)?" << "\n>> ";

char input;

while (true) {

cin >> input;

if (input == 'y' || input == 'Y')

return true;

else if (input == 'n' || input == 'N')

return false;

else

cout << "Wrong input! Try again...\n>> ";

}

}

float stopSecondsTimer(time\_point<steady\_clock> startTimer) {

using fseconds = duration<float>; // Определение кастомного интервала времени для отображение дробных секунд в таймере.

time\_point<steady\_clock> endTimer = steady\_clock::now(); // Остановка времени

fseconds sortingTime = duration\_cast<fseconds>(endTimer - startTimer); // Вычисление разницы между финальным и стартовым временем

return sortingTime.count();

}

long long stopNanoSecondsTimer(time\_point<steady\_clock> startTimer) {

time\_point<steady\_clock> endTimer = steady\_clock::now();

nanoseconds sortingTime = duration\_cast<nanoseconds>(endTimer - startTimer);

return sortingTime.count();

}

void practicalWork3() {

int order;

cout << "Solution of task \"Arythmetics of pointers. Matrixes\". \n\n"

<< "Task 1. Visualisation of filling the matrix. \n"

<< "Enter the order of matrix... \n>> ";

bool inputIsCorrected = true;

do {

cin >> order;

if (order > 0)

inputIsCorrected = false;

else {

cout << "Wrong input! Try again...\n>> ";

}

} while (inputIsCorrected);

srand((unsigned)time(NULL));

int \*ptrarray = new int[order\*order]; // Объявление динамического одномерного массива размерности двумерного

for (int i = 0; i < order; i++)

for (int j = 0; j < order; j++) // В массив пишутся случайные числа от 1 до N\*N, где N - порядок матрицы.

\*(ptrarray + i \* order + j) = 1 + rand() % (order \* order);

do {

cout << "\nEnter the type of animation:"

<< "\n\t1) Snake animation"

<< "\n\t2) Spiral animation \n>> ";

inputIsCorrected = true;

do {

int input;

cin >> input;

switch (input) {

case 1:

snakeAnimation(ptrarray, order);

inputIsCorrected = false;

break;

case 2:

spiralAnimation(ptrarray, order);

inputIsCorrected = false;

break;

default:

cout << "Wrong input! Try again...\n\n>> ";

}

} while (inputIsCorrected);

} while (choiseNextAction());

int \*copiedArr = new int[order \* order]; // Копирование массива в другой массив для возможности повторно произвести перестановку элементов.

do {

for (int i = 0; i < order; i++)

for (int j = 0; j < order; j++)

\*(copiedArr + i \* order + j) = \*(ptrarray + i \* order + j);

cout << "\nTask 2. Permutation of matrix elements"

<< "\nEnter the type of permutation:"

<< "\n\t1) Consistent"

<< "\n\t2) Diagonal"

<< "\n\t3) Row"

<< "\n\t4) Column \n>> ";

inputIsCorrected = true;

do {

int input;

cin >> input;

switch (input) {

case 1:

consistentPermutation(copiedArr, order);

inputIsCorrected = false;

break;

case 2:

diagonalPermutation(copiedArr, order);

inputIsCorrected = false;

break;

case 3:

rowPermutation(copiedArr, order);

inputIsCorrected = false;

break;

case 4:

columnPermutation(copiedArr, order);

inputIsCorrected = false;

break;

default:

cout << "Wrong input! Try again...\n\n>> ";

}

} while (inputIsCorrected);

cout << endl; // Вывод получившегося массива.

for (int i = 0; i < order; i++) {

for (int j = 0; j < order; j++)

cout << setw(4) << \*(copiedArr + i \* order + j);

cout << endl;

}

} while (choiseNextAction());

cout << "\nTask 3. Matrix sorting"

<< "\nBubble sorted array: \n";

bool swapped;

do {

swapped = false;

for (int i = 0; i < order \* order - 1; i++) {

if (\*(copiedArr + i) > \*(copiedArr + i + 1)) { // Соседние элементы массива меняются местами

int t = \*(copiedArr + i);

\*(copiedArr + i) = \*(copiedArr + i + 1);

\*(copiedArr + i + 1) = t;

swapped = true;

}

}

} while (swapped);

for (int i = 0; i < order; i++) {

for (int j = 0; j < order; j++)

cout << setw(4) << \*(copiedArr + i \* order + j);

cout << endl;

}

do {

cout << "\nTask 4. Matrix operations"

<< "\nEnter the type of operation:"

<< "\n\t1) Reducing the matrix by a number"

<< "\n\t2) Increacing the matrix by a number"

<< "\n\t3) Matrix multiplication by a number"

<< "\n\t4) Dividing the matrix by a number \n>> ";

/\*

Если бы данная программа использовалась для чего-то большего, чем написание курсовой, то её однозначно стоило бы доработать на предмет возможности работы с нецелыми числами, т. к. в случае деления чисел матрицы на введённое пользователем число очень часто могут получаться нецелые числа в ячейках матрицы, а в данном случае мы будем видеть искажённый результат.

\*/

inputIsCorrected = true;

int input;

do {

cin >> input;

if (input == 1 || input == 2 || input == 3 || input == 4)

inputIsCorrected = false;

else {

cout << "Wrong input! Try again...\n\n>> ";

break;

}

} while (inputIsCorrected);

int number;

cout << "\nEnter the integer: \n>> ";

cin >> number;

switch (input) {

case 1:

for (int i = 0; i < order; i++) {

for (int j = 0; j < order; j++) {

\*(copiedArr + i \* order + j) -= number;

cout << setw(4) << \*(copiedArr + i \* order + j);

}

cout << endl;

}

break;

case 2:

for (int i = 0; i < order; i++) {

for (int j = 0; j < order; j++) {

\*(copiedArr + i \* order + j) += number;

cout << setw(4) << \*(copiedArr + i \* order + j);

}

cout << endl;

}

break;

case 3:

for (int i = 0; i < order; i++) {

for (int j = 0; j < order; j++) {

\*(copiedArr + i \* order + j) \*= number;

cout << setw(4) << \*(copiedArr + i \* order + j);

}

cout << endl;

}

break;

case 4:

for (int i = 0; i < order; i++) {

for (int j = 0; j < order; j++) {

\*(copiedArr + i \* order + j) /= number;

cout << setw(4) << \*(copiedArr + i \* order + j);

}

cout << endl;

}

break;

}

} while (choiseNextAction());

cout << "\nEnd of the third practical work. \n";

system("pause");

}

void practicalWork4() {

cout << "Solution of task \"Working with a text string\". \n\n"

<< "Task 1. Editing text. \n";

string sourceStr;

do {

cout << "\nEnter the input type: \n"

<< "1) Keyboard \n"

<< "2) File \n>> ";

int input;

bool inputIsCorrected = true;

do {

cin >> input;

if (input == 1 || input == 2)

inputIsCorrected = false;

else {

cout << "Wrong input! Try again...\n>> ";

}

} while (inputIsCorrected);

switch (input) {

case 1:

cout << "\nEnter the string for edit (only English). For ending press Enter... \n>> ";

cin.ignore(32767, '\n');

getline(cin, sourceStr);

cout << "\nYou entered: \n" << sourceStr << endl;

break;

case 2:

ifstream file;

do {

cout << "\nEnter the path to the file. \n"

<< "Only english words in the file and path! Example: C:\\anime\\flex.txt \n>> ";

string path;

cin.ignore(32767, '\n');

getline(cin, path);

file.open(path);

if (!file.is\_open()) {

cout << "Error opening file! Please restart the program! \n";

continue;

}

getline(file, sourceStr);

} while (!file.is\_open());

cout << "\nYou entered: \n" << sourceStr << endl;

file.close();

break;

}

} while (choiseNextAction());

cout << "\nTask 2. Text cleaning. \n"

<< "Entered text without innecessary characters, spaces and case of letters: \n";

sourceStr = textCleaner(sourceStr);

cout << sourceStr << endl << endl;

system("pause");

cout << "\nTask 3 (var. 1)."

<< "\nOutput words in reverse order: \n";

reverseOutput(sourceStr);

cout << endl;

system("pause");

/\*

Данный алгоритм может показаться избыточным в реализации,

т. к. изначально он разрабатывался под вывод слов в колонках по левому выравниванию и был переработан минимальными усилиями - создан небольшой массив размера длины самого длинного слова в тексте, на вывод подаётся этот массив, т. к. cout умеет выравнивать целые слова. Таким образом не приходится задумываться над тем, как выравнивать по правому краю слова посимвольно.

Его можно переработать более оптимизированно/изящно следующими путями:

1) Поместив текст пословно в двумерный массив, где каждая строка/столбец (кому как удобно) содержит отдельно слово, на вывод соответственно будет подаваться строка/столбец целиком, форматировать в таком случае будет проще.

В текущем же алгоритме схожая реализация.

2) Без создания доп. массивов с помощью каретки выводить слова по правому выравниванию.

Этот вариант предпочтительнее, но требует больше трудозатрат.

Варианты не единственные, можно придумать более изящные.

\*/

cout << "\nTask 4 (var. 7)."

<< "\nDisplaying right aligned text in columns: \n";

int arrSize = sourceStr.size(); // Определение количества символов в тексте, очищенном от мусора

int lWord = longestWord(sourceStr); // Определение самого длинного слова для корректного вывода в ячейках далее

int pos = 0; // Текущий индекс, по которому передаётся на вывод элемент массива с текстом

char \*word = new char[lWord]; // Массив, в который будет записываться текущее слово

for (int t = 0; t < lWord; t++)

word[t] = 0;

// Согласно постановке задачи, в зависимости от чётности-нечётности текст выводится либо в 2, либо в 3 колонки.

if ((countOfWords(sourceStr) % 2) == 0) {

while (pos < arrSize) {

for (int i = 0; i < 2; i++) { // Вывод по 2 слова в строку

short j = 0; // Текущая позиция внутри массива текущего слова (word)

while ((sourceStr[pos] != ' ') && (pos < arrSize)) {

word[j] = sourceStr[pos];

pos++;

j++;

}

word[j] = 0; // Установка конца слова для корректного вывода

cout << setw(lWord + 1) << right << word;

pos++; // Пропуск пробела в тексте

for (int t = 0; t < lWord; t++)

word[t] = 0; // Заполнение нулями для очистки, чтобы можно было отобразить следующее слово

}

cout << endl;

}

} else

while (pos < arrSize) { // Ниже всё аналогично тому, что было выше.

for (int i = 0; i < 3; i++) { // Вывод по 3 слова в строку

if (!(pos < arrSize))

break;

short j = 0;

while ((sourceStr[pos] != ' ') && (pos < arrSize)) {

word[j] = sourceStr[pos];

pos++;

j++;

}

word[j] = 0;

cout << setw(lWord + 1) << right << word;

pos++;

for (int t = 0; t < lWord, word[t] != 0; t++)

word[t] = 0;

}

cout << endl;

}

cout << endl;

system("pause");

time\_point<steady\_clock> startTimer = steady\_clock::now(); // Нужно для таймера

float stopTimeInSeconds = stopSecondsTimer(startTimer);

cout << "\nTask 5 (with additional (odd) task)."

<< "\nSearching substring in the string.\n";

cout << "\nEntered text without innecessary characters, spaces and case of letters: \n";

cout << sourceStr << endl ; // Выводим ещё раз очищенный от мусора текст, чтобы пользователь понимал, где и что ищется

do {

cout << "\nEnter the subsctring (only English, only ASCII characters). For ending press Enter... \n>> ";

string substring;

cin.ignore(32767, '\n');

getline(cin, substring);

cout << "\nYou entered: \n" << substring << endl;

// Используем таймер для засечения разницы времени между разными методами поиска подстроки в строке

startTimer = steady\_clock::now();

int subStrPos = linearSubscringSearch(sourceStr, substring);

stopTimeInSeconds = stopSecondsTimer(startTimer);

if (subStrPos != -1)

cout << "Position of substring (first letter, ): " << subStrPos << endl

<< "The lineary search has been completed for " << fixed << stopTimeInSeconds << " seconds. \n";

else

cout << "Substring was not found!\n"

<< "The lineary search has NOT been completed for " << fixed << stopTimeInSeconds << " seconds! \n";

subStrPos = -1;

startTimer = steady\_clock::now();

subStrPos = bouyerMooreSubstringSearch(sourceStr, substring);

stopTimeInSeconds = stopSecondsTimer(startTimer);

if (subStrPos != -1)

cout << "The search with the Boyer-Moore alghoritm has been completed for " << stopTimeInSeconds << " seconds. \n";

else

cout << "The search with the Boyer-Moore alghoritm has NOT been completed for " << stopTimeInSeconds << " seconds! \n";

} while (choiseNextAction());

cout << "\nEnd of the fourth and last practical work."

<< "\nThanks for the attention! \n";

system("pause");

}

void snakeAnimation(int arr[], int order, short cellSize, short delay) {

/\* Функция принимает на вход:

1) Указатель на первый элемент массива-матрицы (квадратной) (required)

2) Порядок матрицы (required)

3) Размер текстовой ячейки, определяемой под одно число матрицы (минимально это количество цифр в максимальном числе + 1) (опционально)

4) Скорость вывода элементов матрицы, в миллисекундах (опционально)

Функции других анимаций работают аналогично.

\*/

cout << "\nTry to \"snake\" animation: \n";

HANDLE hConsole = GetStdHandle(STD\_OUTPUT\_HANDLE); // Нужно для получения данных курсора консоли

CONSOLE\_SCREEN\_BUFFER\_INFO bi;

int column = 0;

while (column < order) {

for (int row = 0; row < order; row++) { // Вывод вниз

GetConsoleScreenBufferInfo(hConsole, &bi); // Получение координат курсора

bi.dwCursorPosition.Y += 1; // Вычисление новых координат курсора

bi.dwCursorPosition.X -= cellSize;

SetConsoleCursorPosition(hConsole, bi.dwCursorPosition); // Установка положения курсора

cout << setw(cellSize) << \*(arr + row \* order + column);

Sleep(delay); // Задержка

}

column++;

if (column < order) {

for (int row = order - 1; row >= 0; row--) { // Вывод вверх

cout << setw(cellSize) << \*(arr + row \* order + column);

Sleep(delay);

GetConsoleScreenBufferInfo(hConsole, &bi);

bi.dwCursorPosition.Y -= 1;

bi.dwCursorPosition.X -= cellSize;

SetConsoleCursorPosition(hConsole, bi.dwCursorPosition);

}

column++;

GetConsoleScreenBufferInfo(hConsole, &bi);

bi.dwCursorPosition.X += (cellSize \* 2);

SetConsoleCursorPosition(hConsole, bi.dwCursorPosition);

}

else break;

}

if (order % 2 == 0)

for (int i = 0; i < order + 1; i++)

cout << endl;

}

void spiralAnimation(int arr[], int order, short cellSize, short delay) {

cout << "\n\nTry to spiral animation: \n";

HANDLE hConsole = GetStdHandle(STD\_OUTPUT\_HANDLE); // Нужно для получения данных курсора консоли

CONSOLE\_SCREEN\_BUFFER\_INFO bi;

int length = order; // Длина текущего вывода

for (int i = 0; i < length; i++) { // Вывод вправо

cout << setw(cellSize) << \*(arr + (order - length) \* order + i);

Sleep(delay);

}

length--;

int upperRightBorder = order - 1;

int lowerLeftBorder = 0;

while (length > 0)

{

for (int i = 0, t = lowerLeftBorder + 1; i < length; i++, t++) { // Вывод вниз

GetConsoleScreenBufferInfo(hConsole, &bi);

bi.dwCursorPosition.Y += 1;

bi.dwCursorPosition.X -= (cellSize);

SetConsoleCursorPosition(hConsole, bi.dwCursorPosition);

cout << setw(cellSize) << \*(arr + t \* order + upperRightBorder);

Sleep(delay);

}

if (!(length > 0))

break;

for (int i = 0, t = upperRightBorder - 1; i < length; i++, t--) { // Вывод влево

GetConsoleScreenBufferInfo(hConsole, &bi);

bi.dwCursorPosition.X -= (cellSize \* 2);

SetConsoleCursorPosition(hConsole, bi.dwCursorPosition);

cout << setw(cellSize) << \*(arr + upperRightBorder \* order + t);

Sleep(delay);

}

length--;

upperRightBorder--;

if (!(length > 0))

break;

for (int i = 0, t = upperRightBorder; i < length; i++, t--) { // Вывод вверх

GetConsoleScreenBufferInfo(hConsole, &bi);

bi.dwCursorPosition.Y -= 1;

bi.dwCursorPosition.X -= (cellSize);

SetConsoleCursorPosition(hConsole, bi.dwCursorPosition);

cout << setw(cellSize) << \*(arr + t \* order + lowerLeftBorder);

Sleep(delay);

}

lowerLeftBorder++;

if (!(length > 0))

break;

for (int i = 0, t = lowerLeftBorder; i < length; i++, t++) { // Вывод вправо

cout << setw(cellSize) << \*(arr + lowerLeftBorder \* order + t);

Sleep(delay);

}

length--;

}

for (int i = 0; i <= (order / 2); i++)

cout << endl;

}

void consistentPermutation(int arr[], int order) {

// Определение порядка "рабочей четверти" матрицы, элементы которой (четверти) далее будут перебираться (разное поведение для матриц чётного и нечётного порядка)

int halfOrder = ceil(static\_cast<float>(order) / 2);

for (int i = 0; i < order / 2; i++) {

for (int j = 0; j < order / 2; j++) {

// Определение индекса текущего опорного элемента, далее будет производиться работа над элементами с таким же индексом, только в других четвертях матрицы

int \*p = (arr + i \* order + j);

int t = \*p;

// \*p - элемент левой верхней четверти, "рабочей четверти" матрицы

\*p = \*(p + halfOrder \* order);

// \*(p + halfOrder \* order) - элемент левой нижней четверти матрицы

\*(p + halfOrder \* order) = \*(p + halfOrder \* order + halfOrder);

// \*(p + halfOrder \* order + halfOrder) - элемент правой нижней четверти матрицы

\*(p + halfOrder \* order + halfOrder) = \*(p + halfOrder);

// элемент правой верхней четверти матрицы

\*(p + halfOrder) = t;

}

}

}

void diagonalPermutation(int arr[], int order) {

int halfOrder = ceil(static\_cast<float>(order) / 2);

for (int i = 0; i < order / 2; i++) {

for (int j = 0; j < order / 2; j++) {

int \*p = (arr + i \* order + j);

int t = \*p;

\*p = \*(p + halfOrder \* order + halfOrder);

\*(p + halfOrder \* order + halfOrder) = t;

t = \*(p + halfOrder);

\*(p + halfOrder) = \*(p + halfOrder \* order);

\*(p + halfOrder \* order) = t;

}

}

}

void rowPermutation(int arr[], int order) {

int halfOrder = ceil(static\_cast<float>(order) / 2);

for (int i = 0; i < order / 2; i++) {

for (int j = 0; j < order / 2; j++) {

int \*p = (arr + i \* order + j);

int t = \*p;

\*p = \*(p + halfOrder \* order);

\*(p + halfOrder \* order) = t;

t = \*(p + halfOrder);

\*(p + halfOrder) = \*(p + halfOrder \* order + halfOrder);

\*(p + halfOrder \* order + halfOrder) = t;

}

}

}

void columnPermutation(int arr[], int order) {

int halfOrder = ceil(static\_cast<float>(order) / 2);

for (int i = 0; i < order / 2; i++) {

for (int j = 0; j < order / 2; j++) {

int \*p = (arr + i \* order + j);

int t = \*p;

\*p = \*(p + halfOrder);

\*(p + halfOrder) = t;

t = \*(p + halfOrder \* order);

\*(p + halfOrder \* order) = \*(p + halfOrder \* order + halfOrder);

\*(p + halfOrder \* order + halfOrder) = t;

}

}

}

char \* readFile(string fileName) {

ifstream file;

file.open(fileName);

if (!file.is\_open()) {

cout << "Error opening file! \n";

cin.ignore(32767, '\n');

cin.clear();

return 0;

}

char \*pointerArr = new char[sizeof(file)];

while (!file.eof())

{

file.getline(pointerArr, sizeof(file));

}

cout << pointerArr;

return pointerArr;

}

string textCleaner(string inputStr) {

int size = inputStr.size();

bool isChanged;

do {

isChanged = false;

for (int i = 1; i < size; i++) {

if ( // Удаляем лишние символы

(isSymbol(inputStr[i])) && (isSymbol(inputStr[i - 1]))

&& (inputStr[i] != '.')

) {

inputStr = deleteChar(inputStr, size, i);

size--;

isChanged = true;

}

if ( // Редкие сочетания типа "пробел-знак препинания-пробел"? кроме тире

(i >= 2)

&& (inputStr[i] == ' ')

&& (isSymbol(inputStr[i - 1]))

&& (inputStr[i - 1] != '-')

&& (inputStr[i - 2] == ' ')

) {

inputStr = deleteChar(inputStr, size, i - 1);

size--;

isChanged = true;

}

}

for (int i = 2; i < size; i++) { // Редкие случаи с точками

if (

(inputStr[i] == '.')

&& ((isSymbol(inputStr[i - 1])) || (inputStr[i - 1] == ' '))

&& (inputStr[i - 1] != '.')

&& (inputStr[i - 1] != ')')

&& (inputStr[i - 1] != '"')

&& (inputStr[i - 1] != ']')

&& (inputStr[i - 1] != '}')

&& (inputStr[i - 1] != '>')

) {

inputStr = deleteChar(inputStr, size, i);

size--;

isChanged = true;

}

}

for (int i = 1; i < size; i++) {

if ((inputStr[i] == ' ') && (inputStr[i - 1] == ' ')) { // Удаляем лишние пробелы

inputStr = deleteChar(inputStr, size, i);

size--;

isChanged = true;

}

}

} while (isChanged);

for (int i = 1; i < size; i++) { // Исправление регистра

if (

(inputStr[i] >= 65) && (inputStr[i] <= 90)

&& (

((inputStr[i - 1] >= 97) && (inputStr[i - 1] <= 122))

|| ((inputStr[i - 1] >= 65) && (inputStr[i - 1] <= 90))

)

)

inputStr[i] = inputStr[i] + 32; // Замена прописной буквы на строчную

}

return inputStr;

}

bool isSymbol (char c) { // Если элемент является знаком препинания

if (

((c >= 33) && (c <= 47))

|| ((c >= 58) && (c <= 64))

|| ((c >= 91) && (c <= 96))

|| (c >= 123)

)

return true;

else

return false;

}

string deleteChar(string inputStr, int size, int pos) { // Удаление элемента путём смещения массива влево. В теории можно реализовать более эффективный алгоритм, смещая до ближайшего вхождения условия, но ещё нескольких вечеров на перелопачивание алгоритма нет, да и задачи на написание максимально эффективного алгоритма нет.

for (int i = pos; i < (size - 1); i++)

inputStr[i] = inputStr[i + 1];

inputStr.erase(size - 1);

return inputStr;

}

void reverseOutput(string inputStr) {

int size = inputStr.size();

for (int i = size - 1; i >= 0; i--) {

if (inputStr[i] == ' ') {

for (int j = i + 1; (j < size) && (inputStr[j] != ' '); j++)

cout << inputStr[j];

cout << ' ';

}

else

if (i == 0)

for (int j = i; inputStr[j] != ' '; j++)

cout << inputStr[j];

}

cout << endl;

}

int countOfWords(string inputStr) {

int size = inputStr.size();

int count = 1;

for (int i = 0; i < size; i++)

if (inputStr[i] == ' ')

count++;

return count;

}

void wordOutput(string inputStr, int size, int pos) {

for (int i = pos; (i < size) && (inputStr[i] != ' '); i++) {

cout << inputStr[i];

}

}

int longestWord(string inputStr) {

int arrSize = inputStr.size();

int maxWordSize = 0;

int wordSize;

for (int i = 0; i < arrSize; i++) {

wordSize = 0;

do {

wordSize++;

if (wordSize > maxWordSize)

maxWordSize = wordSize;

i++;

} while ((inputStr[i] != ' ') && (i < arrSize));

}

return maxWordSize;

}

int linearSubscringSearch(string inputStr, string substr) {

int mainSize = inputStr.size();

int subSize = substr.size();

bool isFounded = false;

for (int i = 0; i < mainSize; i++) {

int pos = i;

for (int j = 0; ((inputStr[i] == substr[j]) && (j < subSize)); j++, i++) {

if (j == (subSize - 1)) {

isFounded = true;

return pos;

}

}

}

if (!isFounded)

return -1;

}

int bouyerMooreSubstringSearch(string mainStr, string subStr)

{

int mainSize = subStr.size();

int subSize = mainStr.size();

int badchar[256];

badCharHeuristic(subStr, mainSize, badchar);

int s = 0;//s is shift of the pattern with respect to text

while (s <= (subSize - mainSize))

{

int j = mainSize - 1;

while (j >= 0 && subStr[j] == mainStr[s + j])

j--;

if (j < 0)

{

return s;

}

else

s += max(1, j - badchar[mainStr[s + j]]);

}

return -1;

}

void badCharHeuristic(string str, int size, int badchar[256])

{

int i;

//Initialize all occurrences as -1

for (i = 0; i < 256; i++)

badchar[i] = -1;

//Fill the actual value of last occurrence of a character

for (i = 0; i < size; i++)

badchar[(int)str[i]] = i;

}