

Учреждение образования
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ
УНИВЕРСИТЕТ»

Структуры данных

Лабораторный практикум

Минск 2023

ПРЕДИСЛОВИЕ

Практикум содержит задания для выполнения лабораторных работ на основе приложения **Microsoft Visual Studio**. В каждой работе имеются краткие теоретические сведения по рассматриваемым вопросам.

Преподаватель определяет, какие лабораторные работы должны выполнять студенты и в каком объеме. Предполагается, что выполнение большинства лабораторных работ занимает у студентов два академических часа.

Задания для выполнения лабораторных работ содержат кнопки, при нажатии на которые открываются тесты, предназначенные для контроля знаний студентов. Тестирование происходит по команде преподавателя и занимает несколько минут.

Для работы тестирующих программ предварительно в приложении Word надо разрешить использование макросов. При этом тексты ответов на формах располагаются каждый раз случайным образом, и ответить на вопросы можно только один раз, так как после нажатия на кнопку «Результаты» форма с вопросами и вариантами ответов исчезает.

При **оформлении отчетов по лабораторным работам** необходимо использовать приложение **Word**. Каждый отчет должен содержать название работы, условия задач, блок-схемы алгоритмов, тексты разработанных программ, скриншоты результатов выполнения программ.

В верхнем колонтитуле записывается фамилия студента и номер группы, в нижнем ☐ номера страниц. Шрифт ☐ 10 или 12, интервал ☐ одинарный, поля ☐ по 1,5 см. Все отчеты сохраняются в **одном** файле.

ОГЛАВЛЕНИЕ

Лабораторная работа №1 Перечисление. Объединение. STL контейнеры. Массив. Дек. Однодвух связный список. Вектор;

Лабораторная работа №2 Схема БД. Меню программы;

Лабораторная работа №3 Двоичные и текстовые файлы;

Лабораторная работа №4 Индексирование записей. Простой/сложный индекс;

Лабораторная работа №5 Запись/чтение массива структур в файл;

Лабораторная работа №6 Редактирование файлов: удаление, изменение поля;

Лабораторная работа №7 Сортировка записей;

Лабораторная работа №8 Фильтрация данных;

Лабораторная работа №9 Поиск записи по значению/индексу.

Лабораторная работа №10. Динамическое программирование: поиск наибольшей общей подпоследовательности,

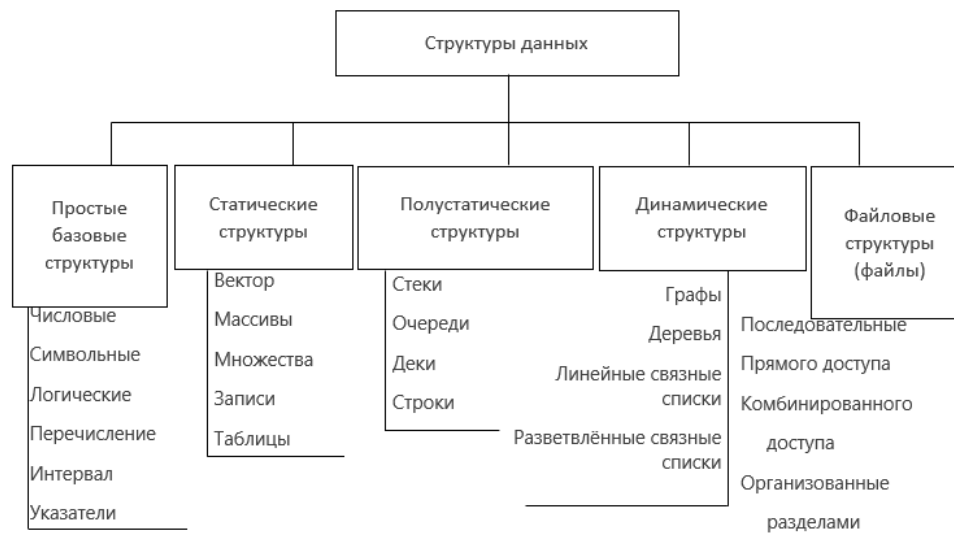
последовательность Фибоначчи, задача о рюкзаке, задача о коммивояжёре.

Лабораторная работа №1

Перечисление. Объединение. STL контейнеры.

Массив. Дек. Map. Set. Одно/двух связный список. Вектор

Структуры данных - это специальные методы организации и хранения данных, которые позволяют эффективно обрабатывать и манипулировать с ними. Они являются ключевыми компонентами в разработке программного обеспечения, так как позволяют эффективно решать различные задачи, связанные с обработкой данных.



В C++, **контейнер** - это абстрактный тип данных (ADT), который представляет собой объект, хранящий коллекцию элементов определенного типа. Контейнеры предоставляют удобный интерфейс для добавления, удаления и доступа к элементам коллекции.

Контейнеры в C++ могут быть реализованы как классы или шаблоны классов. Стандартная библиотека C++ предоставляет множество контейнеров, таких как вектор, список, дерево, хэш-таблица и другие.

Контейнеры могут быть разделены на две категории: последовательные и ассоциативные. Последовательные контейнеры, такие как вектор и список, хранят элементы в порядке их добавления и обеспечивают доступ к элементам по индексу или итератору. Ассоциативные контейнеры, такие как множество и карта, хранят элементы в отсортированном порядке и обеспечивают быстрый поиск элементов по ключу.



Контейнеры в C++ являются важной частью стандартной библиотеки и предоставляют эффективные и удобные средства для работы с коллекциями данных.

Обычно, функционал классов-контейнеров языка C++ следующий:

1. Создание пустого контейнера (через конструктор).
2. Добавление нового объекта в контейнер.
3. Удаление объекта из контейнера.
4. Просмотр количества объектов, находящихся на данный момент в контейнере.
5. Очистка контейнера от всех объектов.
6. Доступ к сохраненным объектам.
7. Сортировка объектов/элементов (не всегда).

Ниже будут рассмотрены самые популярные и частоиспользуемые контейнеры.

Перечисление (Enumeration) - это процесс перебора элементов множества или коллекции. В программировании перечисление используется для перебора элементов в массивах, списках или других структурах данных.

```
#include <iostream>

using namespace std;

enum Colors {
    RED,
    GREEN,
    BLUE,
};

int main()
{
```

```

setlocale(LC_ALL, "rus");
Colors car = RED;
switch (car) {
case RED:
    cout << "Машина красного цвета" << endl;
    break;
case GREEN:
case BLUE:
    cout << "Машина зеленая или синяя" << endl;
    break;
default:
    cout << "Неизвестный цвет машины" << endl;
    break;
}
}

```

```
Машина красного цвета
```

Объединение (Union) - это тип данных, который позволяет хранить разные типы данных в одной и той же памяти. Объединение может содержать несколько элементов, но только один элемент может быть активным в любой момент времени.

```

#include <iostream>

using namespace std;

union NewUnion {
    short int a;
    int b;
    double c;
};

int main()
{
    setlocale(LC_ALL, "rus");
    NewUnion un;
    un.a = 5; // занимает 2 бита
    cout << "a = " << un.a << endl;
    un.b = 15412241; // занимает 4 бита
    cout << "a = " << un.a << " - мусор" << endl;
    cout << "b = " << un.b << endl;
    un.c = 141.14; // занимает 8 битов
    cout << "c = " << un.c << endl;
}

```

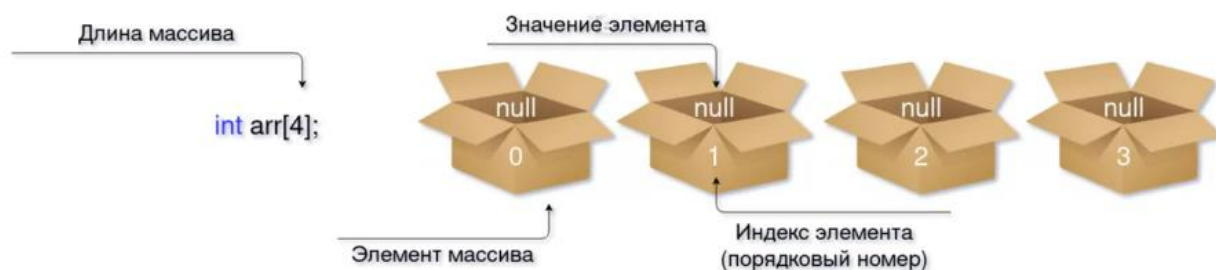
```

a = 5
a = 11281 - мусор
b = 15412241
c = 141.14

```

STL (Standard Template Library) контейнеры - это набор шаблонных классов и функций в стандартной библиотеке C++, которые предоставляют реализацию различных структур данных, таких как массивы, списки, деревья и т.д. В STL реализовано множество алгоритмов, таких как сортировка, поиск, генерация случайных чисел, перемещение элементов и т.д.

Массив (Array) - это структура данных, которая хранит фиксированное количество элементов одного типа. Элементы массива располагаются в памяти последовательно, и доступ к ним осуществляется по индексу.



```
#include <iostream>
#include <array>
using namespace std;

int main()
{
    // создание массива целых чисел
    array<int, 3> arr = { 10, 20, 30 };

    // вывод элементов на экран
    for (int i = 0; i < arr.size(); i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    // изменение элемента массива
    arr[1] = 25;

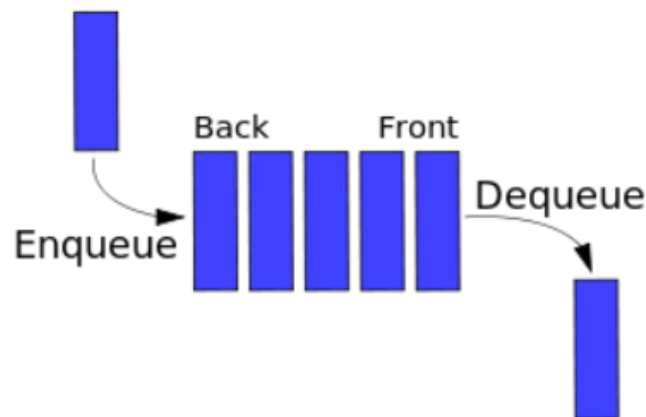
    // вывод элементов на экран
    for (int i = 0; i < arr.size(); i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```



```
10 20 30
10 25 30
```

Дек (Deque) - это двусторонняя очередь, которая позволяет добавлять и удалять элементы как в начале, так и в конце очереди. Дек реализован как двусвязный список. Недостаток: нельзя использовать указатели на элементы deque, так как контейнер может перемещать элементы в памяти при изменении размера, что может приводить к ошибкам при работе с указателями.



```
#include <iostream>
#include <deque>
using namespace std;

int main()
{
    // создание двусторонней очереди целых чисел
    deque<int> dq;

    // добавление элементов в начало и конец очереди
    dq.push_front(10);
    dq.push_back(20);
    dq.push_front(5);

    // вывод элементов на экран
    for (int i = 0; i < dq.size(); i++) {
        cout << dq[i] << " ";
    }
    cout << endl;

    // удаление элементов из начала и конца очереди
    dq.pop_front();
    dq.pop_back();
}
```

```

// вывод элементов на экран
for (int i = 0; i < dq.size(); i++) {
    cout << dq[i] << " ";
}
cout << endl;

return 0;
}

```

```

5 10 20
10

```

Map - представляет собой ассоциативный контейнер, который хранит элементы в виде пар "ключ-значение". Он обеспечивает быстрый доступ к элементам по ключу и автоматически сортирует их по возрастанию ключа.

Ключи и значения могут быть любых типов, которые можно сравнивать оператором <. Ключи должны быть уникальными, то есть каждому ключу соответствует только одно значение.

	KEYS	VALUES	
	Jan	327.2	
	Feb	368.2	
	Mar	197.6	
	Apr	178.4	
	May	100.0	
	Jun	69.9	
	Jul	32.3	
Aug →	Aug	37.3	→ 37.3
	Sep	19.0	
	Oct	37.0	
	Nov	73.2	
	Dec	110.9	
	Annual	1551.0	

```

#include <map>
#include <string>
#include <iostream>
using namespace std;

int main() {
    setlocale(LC_ALL, "rus");
    map<string, int> myMap;

    myMap["one"] = 1;
    myMap["two"] = 2;
    myMap["three"] = 3;

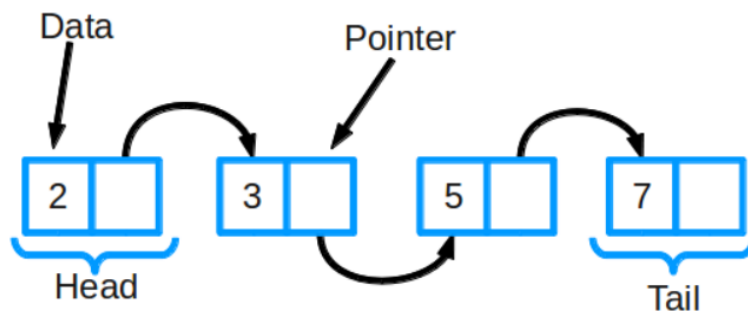
    cout << "Значение второго ключа: " << myMap["two"] << endl;

    return 0;
}

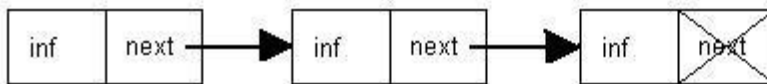
```

Значение второго ключа: 2

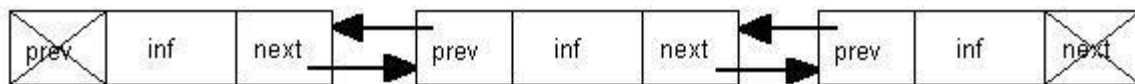
Одно/двух связный список (Singly/Doubly Linked List) - это структура данных, которая состоит из узлов, каждый из которых содержит значение элемента и указатель на следующий (или предыдущий в случае двусвязного списка) узел. Список может быть произвольной длины, и элементы могут добавляться и удаляться из него в любой момент времени. Обычно для перебора элементов двусвязного списка используются итераторы. (однонаправленный связный список = односвязный список)



Связный список



Двусвязный список



В односвязном списке для доступа к предыдущему элементу необходимо обойти весь список с начала. В двусвязном списке же можно легко перемещаться как вперед, так и назад по списку, что делает его более удобным для определенных задач, например, для удаления элементов из середины списка. Однако, двусвязный список требует большего количества памяти для хранения ссылок на предыдущий элемент.

Основные функции, применяемые к `forward_list`:

1. `begin()` - возвращает итератор на первый элемент контейнера

2. *end()* - возвращает итератор на элемент, следующий за последним элементом контейнера
3. *empty()* - возвращает true, если контейнер пуст, и false в противном случае
4. *size()* - возвращает количество элементов в контейнере
5. *clear()* - удаляет все элементы из контейнера
6. *push_front(value)* - добавляет элемент value в начало контейнера
7. *insert_after(iterator, value)* - добавляет элемент value после элемента, на который указывает итератор iterator
8. *pop_front()* - удаляет первый элемент из контейнера
9. *erase_after(iterator)* - удаляет элемент, следующий за элементом, на который указывает итератор iterator
10. *remove(value)* - удаляет все элементы со значением value
11. *front()* - возвращает ссылку на первый элемент контейнера
12. *begin()* - возвращает итератор на первый элемент контейнера
13. *end()* - возвращает итератор на элемент, следующий за последним элементом контейнера

Основные функции, применяемые к list(очень много схожих с forward_list, выделим отличающиеся):

1. *push_back(value)* - добавляет элемент value в конец контейнера
2. *insert(iterator, value)* - добавляет элемент value перед элементом, на который указывает итератор iterator
3. *pop_back()* - удаляет последний элемент из контейнера
4. *erase(iterator)* - удаляет элемент, на который указывает итератор iterator
5. *back()* - возвращает ссылку на последний элемент контейнера
6. *end()* - возвращает итератор на элемент, следующий за последним элементом контейнера

Пример односвязного списка:

```
#include <iostream>
#include <forward_list>

using namespace std;

int main()
{
    setlocale(LC_ALL, "rus");
    forward_list<int> newlist = { 6, 2, 8, 4, 5 };
```

```

cout << "1-ый член массива: " << newList.front() << endl;
// newList.back() не работает, так как в односвязном списке напрямую можно получить доступ только к 1-
ому элементу
auto iter = newList.begin(); // итератор указывает на 1-ой элемент
newList.emplace_after(iter, 15); // добавляем после 1-ого элемента
for (int n : newList)
{
    cout << n << "\t";
}
}

```

```

1-ый член массива: 6
6      15      2      8      4      5

```

Пример двусвязного списка:

```

#include <iostream>
#include <list>

using namespace std;

int main()
{
    setlocale(LC_ALL, "rus");
    list<int> newList = { 6, 2, 8, 4, 5 };
    cout << "1-ый член массива: " << newList.front() << endl;
    cout << "n-ый член массива: " << newList.back() << endl;
    auto iter = ++newList.begin(); // итератор указывает на 2-ой элемент
    newList.emplace(iter, 15); // добавляем после 1-ого элемента
    for (int n : newList)
    {
        cout << n << "\t";
    }
}

```

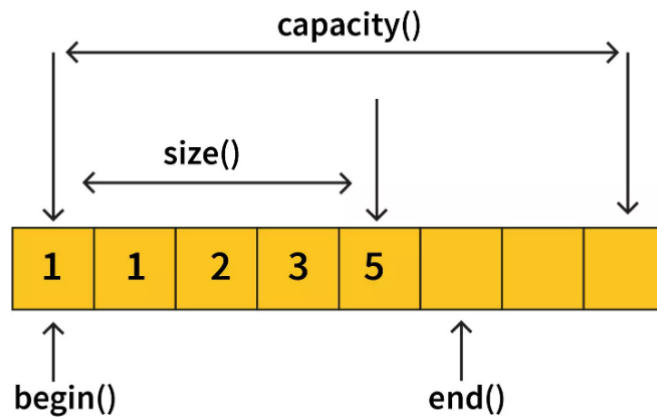
```

1-ый член массива: 6
n-ый член массива: 5
6      15      2      8      4      5

```

Вектор (Vector) - это динамический массив, который может изменять свой размер во время выполнения программы. Vector использует память очень эффективно, поскольку все его элементы хранятся в непрерывной области памяти. Кроме того, размер контейнера vector автоматически увеличивается, когда вы добавляете новые элементы, что делает его более эффективным, чем статические массивы. Vector поддерживает случайный доступ, что означает,

что вы можете обращаться к элементам в любом порядке, не обязательно последовательно.



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

int main() {
    std::vector<int> numbers; // создаем vector для хранения чисел

    // добавляем числа в vector
    numbers.push_back(5);
    numbers.push_back(1);
    numbers.push_back(3);
    numbers.push_back(2);
    numbers.push_back(4);

    // сортируем элементы vector
    std::sort(numbers.begin(), numbers.end());

    // выводим числа на экран
    for (int number : numbers) {
        std::cout << number << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

```
1 2 3 4 5
```

Контейнер **Set** - это ассоциативный контейнер, который используется для хранения упорядоченного набора уникальных элементов. Каждый элемент в set является уникальным и автоматически сортируется в порядке возрастания.

Set реализован как красно-черное дерево, которое гарантирует логарифмическую сложность для основных операций, таких как вставка, удаление и поиск элементов.

Контейнер **multiset** в C++ похож на контейнер **set**, но в отличие от него, **multiset** позволяет хранить дубликаты элементов. То есть, если элемент уже присутствует в **multiset**, при добавлении нового элемента с тем же значением, он будет добавлен в контейнер.

multiset также реализован как красно-черное дерево и поддерживает логарифмическую сложность для основных операций, таких как вставка, удаление и поиск элементов.

Пример использования Set:

```
#include <iostream>
#include <set>
int main() {
    setlocale(LC_ALL, "rus");
    std::set<int> mySet;
    // Добавление элементов в set
    mySet.insert(5);
    mySet.insert(3);
    mySet.insert(1);
    mySet.insert(4);
    mySet.insert(2);
    // Вывод элементов set в порядке возрастания
    for (auto it = mySet.begin(); it != mySet.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;
    // Поиск элемента в set
    auto it = mySet.find(3);
    if (it != mySet.end()) {
        std::cout << "Элемент найден: " << *it << std::endl;
    }
    else {
        std::cout << "Элемент не найден" << std::endl;
    }
    // Удаление элемента из set
    mySet.erase(4);
    // Вывод элементов set в порядке возрастания
    for (auto it = mySet.begin(); it != mySet.end(); ++it) {
        std::cout << *it << " ";
    }
}
```

```
std::cout << std::endl;  
return 0;  
}
```

```
1 2 3 4 5  
Элемент найден: 3  
1 2 3 5
```

В этом примере мы создаем пустой set и добавляем в него несколько элементов. Затем мы выводим элементы set в порядке возрастания, используя итераторы begin() и end(). Мы также ищем элемент со значением 3 в set с помощью метода find(). Затем мы удаляем элемент со значением 4 из set с помощью метода erase(). Наконец, мы выводим элементы set снова для проверки, что элемент 4 был удален.

Задания к лабораторной работе:

1. Написать программу, которая, в зависимости от выбора пользователя, в консоли должна выводить названия стран мира, с использованием перечислений.
2. Написать программу, в которой в конец/в начало массива добавляется n элементов и вывести на экран.
3. Написать программу, в которой нужно найти среднее арифметическое элементов вектора и вывести его на экран.
4. Написать программу, в которой нужно найти максимальный/минимальный элемент дэки и добавить его в конец/в начало, с выводом на экран.
5. Написать программу, в которой нужно удалить все элементы односвязного списка и добавить n новых, с выводом на экран.
6. Написать программу, в которой в начало/в конец двусвязного списка добавляется n элементов и вывести на экран.
7. Написать программу, которая будет искать введенное пользователем число в map.

Дополнительные задания(задания располагаются по уровням сложно):

- 1.Объедините два массива целых чисел в один и выведите его элементы, используя вектор. (3 балла)

2. Создайте дек перечислений, представляющий месяцы года, и добавьте в него несколько элементов. Затем удалите первый и последний элементы и выведите содержимое дека.(3 баллов)

3. Создайте массив строк и отсортируйте его по длине строк. Если длина строк одинакова, то сортируйте их в лексикографическом порядке. Затем выведите содержимое отсортированного массива. (3 баллов)

Вопросы к лабораторной работе:

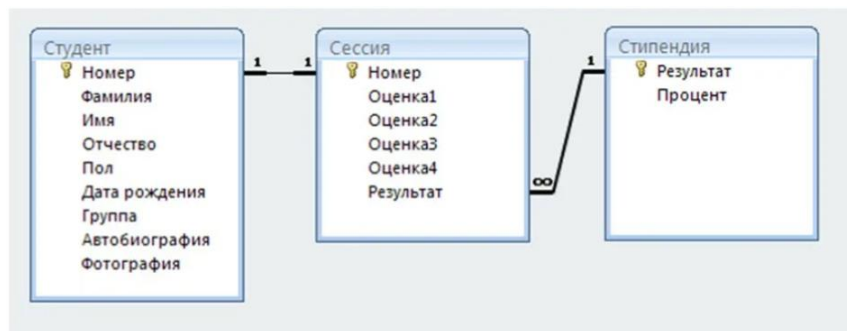
1. Дайте определение структурам данных?
2. Какими способами можно реализовать контейнеры в C++?
3. На какие категории разделяются контейнеры в C++?
4. Дайте определение перечислению и объединению?
5. Дайте определение map и назовите его отличия от массивов и векторов.
6. В чем отличие вектора(vector) и массива(array)?
7. Что предоставляет реализовать различные структуры данных, такие как массивы, списки, деревья?
8. Можно ли удалять элементы из одно/двух связного списка в любой момент времени?
9. Что происходит с вектором, если его текущий размер недостаточен?
10. Какие контейнеры хранят элементы в отсортированном порядке и обеспечивают быстрый поиск элементов по ключу?
11. Сколько элементов может содержать объединение?

Лабораторная работа №2 Схема БД. Меню программы

Порой, чтобы продемонстрировать работу консольной программы бывает удобно воспользоваться меню — элементом пользовательского интерфейса, позволяющим выбрать одну из нескольких перечисленных опций программы. Иногда же наличие меню является обязательным условием задания по программированию.

Схема БД (Базы Данных) - это структурное представление данных, которое определяет, как данные будут храниться, организовываться и связываться между собой в базе данных. Схема БД определяет таблицы, поля, связи между таблицами, ограничения и индексы.

Создание схемы БД - это важный этап проектирования базы данных, который позволяет определить основную структуру и связи между объектами данных. При проектировании схемы БД следует учитывать требования к хранению и обработке данных, а также предусмотреть возможность расширения и изменения схемы в будущем.



Но так как изучение баз данных будет подробно изучаться как отдельная дисциплина в следующем семестре, то бд будет заменяться файловой системой. Однако составление даже самой простой схемы бд значительно упростит понимание дальнейшей файловой системы(из каких файлов и какие данные получать, записывать и тд). Файлы не будут связаны между собой как в случае с таблицами в базе данных.

Меню программы - это пользовательский интерфейс, который позволяет пользователю выбирать определенные функции или операции, которые необходимо выполнить в программе. Меню программы может содержать различные элементы, такие как кнопки, выпадающие списки, чекбоксы и т.д., которые позволяют пользователю выбирать и настраивать определенные параметры программы.

Кроме того, при проектировании меню программы следует учитывать различные возможности для настройки и персонализации пользовательского интерфейса, такие как изменение цветовой схемы, выбор языка и т.д. Это позволяет пользователю настроить программу под свои потребности и предпочтения.

Составление меню программы в C++ включает в себя несколько шагов:

1. Определение опций меню: определите список опций, которые вы хотите включить в меню программы. Каждая опция должна иметь уникальный идентификатор и краткое описание.
2. Отображение меню: напишите код, который отображает меню на экране. Обычно это включает в себя использование цикла для вывода списка опций на экран и ожидание ввода пользователя.
3. Обработка выбора пользователя: когда пользователь выбирает опцию из меню, программа должна выполнить соответствующую функцию или выполнять определенные действия. Напишите код для обработки выбора пользователя и выполнения соответствующих функций.
4. Повторение цикла: после выполнения выбранной опции программа должна вернуться к началу цикла и отобразить меню снова, чтобы пользователь мог выбрать другую опцию или выйти из программы.

Существуют разные способы реализации консольного меню, один из самых простых и популярных использовать **switch-конструкцию**.

```
#include <iostream>
#include <windows.h>

using namespace std;

int _stateMenu;

void Menu() {
    cout << "Выберите действие:" << endl
         << "(1) Ввод данных" << endl
         << "(2) Вывод" << endl
         << "(3) Удаление" << endl
         << "Ваш выбор: ";
    cin >> _stateMenu;
}

int main()
{
    setlocale(LC_ALL, "rus");
    Menu();
    while (_stateMenu != 0) {
```

```

switch (_stateMenu)
{
case 1:
    system("cls"); // очистка консоли
    cout << "Данные введены!" << endl;
    system("pause"); // задержка консоли
    system("cls");
    Menu();
    break;
case 2:
    system("cls");
    cout << "Данные выведены!" << endl;
    system("pause");
    system("cls");
    Menu();
    break;
case 3:
    system("cls");
    cout << "Данные удалены!" << endl;
    system("pause");
    system("cls");
    Menu();
    break;
default:
    system("cls");
    cout << "Неверно введен номер действия!" << endl;
    system("pause");
    system("cls");
    Menu();
    break;
}
}
}

```

Выберите действие:

(1) Ввод данных

(2) Вывод

(3) Удаление

Ваш выбор: 1_

Данные введены!

Для продолжения нажмите любую клавишу . . . _

Выберите действие:

(1) Ввод данных

(2) Вывод

(3) Удаление

Ваш выбор: 2_

```
Данные выведены!  
Для продолжения нажмите любую клавишу . . .
```

Есть различные способы реализации меню в C++, помимо использования конструкции switch-case. Рассмотрим несколько из них:

1. Использование массива функций: создайте массив указателей на функции, которые будут выполнять определенные действия, и используйте цикл для вывода меню и вызова соответствующей функции при выборе пользователем опции.

Пример:

```
#include <iostream>
using namespace std;

void option1() {
    cout << "выбрано 1" << endl;
}

void option2() {
    cout << "выбрано 2" << endl;
}

void option3() {
    cout << "выбрано 3" << endl;
}

int main() {
    setlocale(LC_ALL, "rus");
    void (*options[3])() = { option1, option2, option3 };
    int choice;

    do {
        cout << "Menu:" << endl;
        cout << "1. Option 1" << endl;
        cout << "2. Option 2" << endl;
        cout << "3. Option 3" << endl;
        cout << "4. Exit" << endl;
        cout << "Ваш выбор: ";
        cin >> choice;

        if (choice >= 1 && choice <= 3) {
            (*options[choice - 1])();
        }
    }
}
```

```

else if (choice == 4) {
    cout << "выходим из программы..." << endl;
}
else {
    cout << "неправильный выбор" << endl;
}
} while (choice != 4);

return 0;

```

```

Menu:
1. Option 1
2. Option 2
3. Option 3
4. Exit
Ваш выбор: 1
выбрано 1
Menu:
1. Option 1
2. Option 2
3. Option 3
4. Exit
Ваш выбор: 2
выбрано 2
Menu:
1. Option 1
2. Option 2
3. Option 3
4. Exit
Ваш выбор: 3
выбрано 3
Menu:
1. Option 1
2. Option 2
3. Option 3
4. Exit
Ваш выбор: 4
выходим из программы...
}

```

2. Использование функциональных объектов: вместо массива указателей на функции можно использовать объекты. (способ может показаться тяжёлым, т.к. в нём используются лямбда выражения) Пример:

```

#include <iostream>
#include <functional>
using namespace std;

class Option1 {
public:
    void operator()() {
        cout << "выбрано 1" << endl;
    }
};

class Option2 {
public:
    void operator()() {
        cout << "выбрано 2" << endl;
    }
};

```

```

    }
};
class Option3 {
public:
    void operator()() {
        cout << "выбрано 3" << endl;
    }
};
int main() {
    setlocale(LC_ALL, "rus");
    Option1 option1;
    Option2 option2;
    Option3 option3;
    std::function<void()> options[3] = { [&]() { option1(); }, [&]() { option2(); }, [&]() { option3(); } };
    int choice;
    do {
        cout << "Menu:" << endl;
        cout << "1. Option 1" << endl;
        cout << "2. Option 2" << endl;
        cout << "3. Option 3" << endl;
        cout << "4. Exit" << endl;
        cout << "Enter your choice: ";
        cin >> choice;
        if (choice >= 1 && choice <= 3) {
            options[choice - 1]();
        }
        else if (choice == 4) {
            cout << "выходим из программы..." << endl;
        }
        else {
            cout << "неправильный выбор" << endl;
        }
    } while (choice != 4);

    return 0;
}

```

```
Menu:
1. Option 1
2. Option 2
3. Option 3
4. Exit
Enter your choice: 1
выбрано 1
Menu:
1. Option 1
2. Option 2
3. Option 3
4. Exit
Enter your choice: 3
выбрано 3
Menu:
1. Option 1
2. Option 2
3. Option 3
4. Exit
Enter your choice:
2
выбрано 2
Menu:
1. Option 1
2. Option 2
3. Option 3
4. Exit
Enter your choice: 4
выходим из программы...
```

Однако использование конструкции `switch-case` интуитивно понятно и упрощает чтение кода, поэтому рекомендуем использовать её.

Так же есть несколько способов реализации **графического меню** в консоли на C++. Рассмотрим некоторые из них:

1. Использование библиотеки `ncurses`:

Библиотека `ncurses` является стандартной библиотекой для создания текстовых пользовательских интерфейсов в UNIX-подобных операционных системах. Она предоставляет набор функций для работы с окнами, кнопками, полями ввода, списками и другими элементами интерфейса. На C++ для работы с `ncurses` можно использовать, например, библиотеку `PDCurses`.

2. Использование ANSI Escape Sequences:

ANSI Escape Sequences - это последовательности символов, которые можно использовать для управления цветом текста, перемещения курсора и других действий в консоли. С помощью этих последовательностей можно реализовать простые графические элементы, такие как рамки, кнопки и т.д. Однако, для создания более сложных интерфейсов, потребуется написание большого количества кода.

3. Использование графических библиотек:

Если требуется создать более сложный графический интерфейс, можно использовать графические библиотеки, такие как SDL или OpenGL. Однако, для работы с такими библиотеками потребуется использовать оконный менеджер, такой как X Window System.

Как мы видим, в основном надо устанавливать дополнительные библиотеки или компиляторы, однако можно использовать ANSI Escape Sequences, которая не является библиотекой, а является последовательностью символов, т.е. не требует дополнительных установок и манипуляций.

Основные последовательности символов для манипуляций с цветом текста:

- `"\033[0m"` - сброс стилей и цвета текста;
- `"\033[30m"` - установка цвета текста на чёрный;
- `"\033[31m"` - установка цвета текста на красный;
- `"\033[32m"` - установка цвета текста на зелёный;
- `"\033[33m"` - установка цвета текста на жёлтый;
- `"\033[34m"` - установка цвета текста на синий;
- `"\033[35m"` - установка цвета текста на фиолетовый;
- `"\033[36m"` - установка цвета текста на голубой;
- `"\033[37m"` - установка цвета текста на белый;
- `"\033[41m"` - установка цвета фона на красный;
- `"\033[42m"` - установка цвета фона на зелёный;
- `"\033[43m"` - установка цвета фона на жёлтый;
- `"\033[44m"` - установка цвета фона на синий;
- `"\033[45m"` - установка цвета фона на фиолетовый;
- `"\033[46m"` - установка цвета фона на голубой;
- `"\033[47m"` - установка цвета фона на белый.

Для использования ANSI Escape Sequences в коде на языке C++ можно использовать специальные символы-экранирующие последовательности, такие как `\033`. Например, чтобы установить цвет текста на красный, можно использовать следующий код:

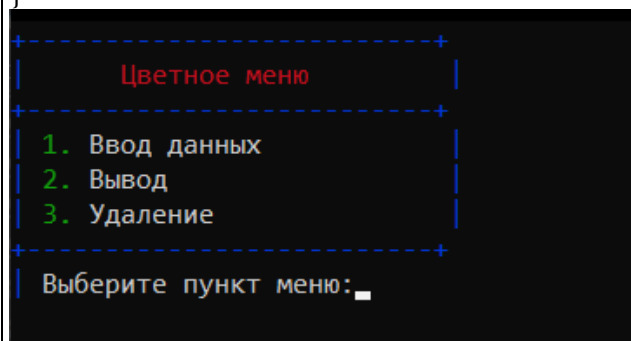
```
std::cout << "\033[31m" << "Red text" << "\033[0m" << std::endl;
```

Этот код устанавливает цвет текста на красный, выводит строку "Red text" и сбрасывает цвет и стили текста до значений по умолчанию.


```

switch (_stateMenu)
{
case 1:
    system("cls"); // очистка консоли
    cout << "Данные введены!" << endl;
    system("pause"); // задержка консоли
    Menu();
    break;
case 2:
    system("cls");
    cout << "Данные выведены!" << endl;
    system("pause");
    Menu();
    break;
case 3:
    system("cls");
    cout << "Данные удалены!" << endl;
    system("pause");
    Menu();
    break;
default:
    system("cls");
    cout << "Неверно введен номер действия!" << endl;
    system("pause");
    Menu();
    break;
}
}
return 0;
}

```



Для более подробного изучения ANSI Escape Sequences: [ANSI escape sequences \(pierai.github.io\)](https://github.com/pierai/ansi-escape-sequences)

Задания к лабораторной работе (брать задания в соответствии со своим вариантом), написать консольное меню (по желанию использовать графическое оформление), вывод содержимого каждого пункта меню должен быть реализован как отдельная функция(для лучшего понимания модифицировать меню в следующих лабах):

ПРИМЕЧАНИЕ! От исполнителя на данном этапе не требуется создания функционального программного продукта, только рабочий интерфейс в котором учтены все поставленные условия!

1. В гостинице есть разные номера с разной *вместимостью, уровнем комфорта и ценой*, которые клиенты могут арендовать на определенный срок. Мы собираем информацию о клиентах, включая их *фамилии, имена, отчества и комментарии*. Когда клиенты заселяются, мы выдаем им свободный номер, который соответствует их требованиям. Мы записываем *дату заселения*, а когда они выезжают, мы записываем *дату освобождения номера*.
2. Вы работаете в страховой компании и следите за ее финансовой деятельностью. Компания имеет много филиалов в разных местах страны, каждый с названием, адресом и телефоном. Люди обращаются к вам, чтобы заключить договор о страховании, например, страхование автомобилей от угона или страхование домашнего имущества. Вы фиксируете дату заключения договора, страховую сумму, вид страхования, тарифную ставку и филиал, где договор был заключен. Договоры заключают страховые агенты, и вы храните информацию об агентах, а также о том, в каком филиале они работают. Заработная плата агентов зависит от процента от страхового платежа, который определяется по виду страхования, по которому заключен договор.
3. Вы работаете в ломбарде и отслеживаете финансовую сторону работы компании. Люди обращаются к вам, чтобы получить деньги в займы под залог своих вещей. Вы запрашиваете их *паспортные данные* и оцениваете *стоимость* заложенного товара. Затем вы решаете, *сколько денег* получит клиент и какую *комиссию* возьмете с него, и договариваетесь *о сроке возврата*. Если клиент согласен, вы составляете документ, выдаете деньги и храните заложенный товар. Если клиент не возвращает деньги вовремя, то товар становится вашей собственностью. После этого вы можете продавать товары по цене, которая может быть выше или ниже, чем цена, которую клиент заявлял при залоге.

4. Вы работаете в компании, которая продает товары оптом и в розницу. Вам нужно следить за финансовой стороной работы компании. Компания торгует разными товарами, каждый из которых имеет свое *наименование, оптовую и розничную цены, а также дополнительную информацию*. Когда покупатели приходят в компанию, вы записываете их данные (*имя, адрес, телефон, контактное лицо*) и создаете документ, в котором указываете, какие товары они купили и когда. Часто покупатели покупают несколько товаров за один раз.
5. Вы работаете в бюро, которое помогает людям найти работу и работодателям найти работников. Когда работодатель обращается к вам, вы записываете его данные (*название, вид деятельности, адрес, телефон*) в базу данных. Когда ищущий работу обращается к вам, вы записываете его данные (*фамилия, имя, квалификация, профессия и т.д.*) в базу данных. Когда работодатель и ищущий работу находят друг друга, вы создаете документ, в котором указываете *их данные, должность и комиссионные* (доход бюро).
6. Вы работаете в нотариальной конторе и ваша работа связана с финансовой стороной компании. Компания предоставляет клиентам различные *услуги*, которые вы описали в списке. При обращении клиента к вам, вы записываете его данные (*название, вид деятельности, адрес, телефон*) в базу данных. После оказания услуги клиенту, вы создаете документ, в котором указываете *описание услуги, стоимость сделки, комиссионные* (доход конторы) *и описание сделки*. Клиент может получить несколько услуг в рамках одной сделки, а стоимость каждой услуги определена заранее.
7. Вы работаете в компании, которая продает запчасти для автомобилей. Ваша задача - отслеживать финансовую сторону работы компании. Основная часть работы - это работа с поставщиками. У компании есть список поставщиков, у каждого из которых есть *название, адрес и телефон*. Вы покупаете детали у этих поставщиков, которые имеют уникальный номер (*артикул*) и *цену*. Некоторые поставщики могут

поставлять одинаковые детали. Вы записываете каждую покупку деталей вместе с *датой покупки и количеством деталей*. Цена на детали может меняться от поставки к поставке, и поставщики сообщают вам *о дате изменения цены и ее новом значении заранее*.

8. Вы работаете в учебном заведении и организуете курсы для улучшения знаний студентов. Вам известна информация *о группах студентов*, которые сформированы в зависимости от их *специальности и отделения*. Каждая группа имеет определенное *количество студентов*, а проведение занятий обеспечивается штатом преподавателей. У каждого преподавателя есть анкетные данные (*фамилия, имя, отчество, телефон*) и *стаж работы*. Вы получаете информацию о том, *сколько часов* каждый преподаватель проводит занятий с соответствующими группами, а также какие виды занятий он проводит (*лекции, практики*), какой *предмет* и сколько он получает *оплату за 1 час*.
9. Вы руководите информационно-аналитическим центром в коммерческом банке, который выдает кредиты юридическим лицам. Ваша задача - отслеживать работу кредитного отдела. Каждый кредит имеет свое *название* и отличается по *условиям получения, процентной ставке и сроку возврата*. Чтобы получить кредит, юридическое лицо должно предоставить информацию о себе, включая *название, вид собственности, адрес, телефон и контактное лицо*. Каждый раз, когда банк выдает кредит, он фиксирует *сумму кредита, клиента и дату выдачи* в своей системе.
10. Вы - коммерческий директор театра, и ваша работа связана с организацией финансовых вопросов, привлечением актеров и заключением контрактов. В театре каждый год ставятся различные спектакли, каждый из которых имеет свой бюджет. Вы нанимаете актеров для конкретных ролей в спектаклях и заключаете с ними контракты на определенную сумму. Каждый актер имеет *свой стаж работы* и может быть награжден за *свои достижения*. Несколько актеров могут играть одну и ту же роль в рамках одного спектакля.

Контракт определяет *базовую зарплату* актера, а после отыгрыша спектакля актер может получить *премию*.

11. Вы - руководитель службы планирования платной поликлиники, и ваша работа заключается в отслеживании финансовых показателей работы поликлиники. В поликлинике работают врачи *разных специальностей и уровней квалификации*, а каждый день туда обращаются пациенты. При регистрации каждого пациента заносятся его данные в базу данных (*фамилия, имя, отчество, год рождения*). Каждый пациент может обращаться в поликлинику несколько раз за различной медицинской помощью. Все обращения пациентов заносятся в базу данных, указывается *диагноз, стоимость лечения и дата обращения*. Пациенты проходят обследование и лечение у разных специалистов, и *общая стоимость лечения* зависит от того, какие *консультации и процедуры* им назначают.
12. Вы работаете в большом торговом центре, который сдаёт в аренду свои торговые помещения другим компаниям. Ваша задача - следить за финансовой стороной работы торгового центра. В торговом центре есть несколько торговых помещений, которые можно сдавать в аренду. Для каждого помещения важны такие данные, как *этаж, площадь, наличие кондиционера и стоимость* аренды в день. Вы собираете данные о потенциальных клиентах (*название компании, адрес, телефон, реквизиты, контактное лицо*) и показываете им свободные помещения. Если они соглашаются арендовать помещение, вы заключаете договор и записываете *информацию о помещении, клиенте и сроке аренды* в базу данных.
13. Вы являетесь руководителем коммерческой службы телекомпании. Ваша задача - следить за финансовой стороной работы компании, связанной с рекламой в телевизионном эфире. Заказчики обращаются к вам, чтобы разместить свою рекламу *в определенной передаче в определенный день*. Каждый рекламный ролик имеет определенную *продолжительность*. У каждого заказчика есть *банковские реквизиты*,

телефон и контактное лицо для переговоров. Передачи имеют свой *рейтинг*, и для каждой передачи известна *стоимость минуты* рекламы (определяется коммерческой службой, исходя из рейтинга передачи и других факторов).

14. Вы работаете в коммерческом отделе компании, которая продает различные товары через интернет. Ваша задача - следить за финансовой стороной работы компании. На сайте компании есть товары, которые продаются. Каждый товар имеет свое *название, цену и единицу измерения* (штуки, килограммы, литры). Вы пытаетесь собирать данные с клиентов для анализа и оптимизации работы компании. Для этого вы собираете стандартные данные клиентов, такие как *имя* и контактную информацию (*телефон и адрес электронной почты*). При каждой продаже вы записываете информацию *о клиенте, товаре, количестве, дате продажи и дате доставки*.
15. Вы работаете парикмахером в парикмахерской, где стригут клиентов в соответствии с их пожеланиями и каталогом различных видов стрижки. Каждая стрижка имеет свое *название, пол (мужской или женский) и стоимость*. Чтобы сохранять порядок, вы создаете базу данных клиентов, запоминая их *фамилии, имена и отчества*. После завершения работы, в кассе фиксируются данные о стрижке, клиенте и *дате выполнения работы*. Теперь у вашей парикмахерской есть филиал, и вы хотите *видеть статистику по каждому филиалу отдельно*.
16. Вы работаете в химчистке, которая принимает вещи от людей для удаления пятен. Чтобы сохранять порядок, вы создаете базу данных клиентов, запоминая их *фамилии, имена и отчества*. Все услуги, которые вы предоставляете, разделены на виды, каждый вид имеет свое *название, тип и стоимость*, которая зависит от *сложности* работы. Когда клиент приходит, вы определяете *объем работ, вид услуги и ее стоимость*. Если клиент согласен, он оставляет свою вещь (в этот момент записываются данные *об услуге, клиенте и дате приема*), а забирает ее после обработки (в этот момент записывается *дата*

возврата). Теперь у вашей химчистки есть филиал, и вы хотите видеть статистику по каждому филиалу отдельно.

Дополнительные задания(т.к. по теме второй лабораторной работе нельзя придумать интересные дополнительные задания, то в качестве доп. заданий будут алгоритмические задачи):

1. Напишите программу, которая будет находить наибольший общий делитель двух чисел. (3 балла)
2. Напишите программу, которая будет находить медиану вектора чисел. (3 балла)
3. Напишите программу, которая будет находить количество путей в ориентированном графе от одной вершины до другой. (3 балла)

Вопросы к лабораторной работе:

1. Дайте определение меню.
2. В каких случаях удобно воспользоваться меню?
3. Как вы понимаете процесс создания схемы БД (Базы Данных)?
4. Дайте определение схемы БД (Базы Данных).
5. Что определяет схема БД (Базы Данных)?
6. Что необходимо учитывать при проектировании схемы БД?
7. Какие элементы может содержать меню программы?
8. Перечислите способы реализации консольного меню?
9. Что следует учитывать при проектировании меню программы?
10. Расскажите, что происходит в примере к лабораторной работе 2?

Лабораторная работа №3. Двоичные и текстовые файлы

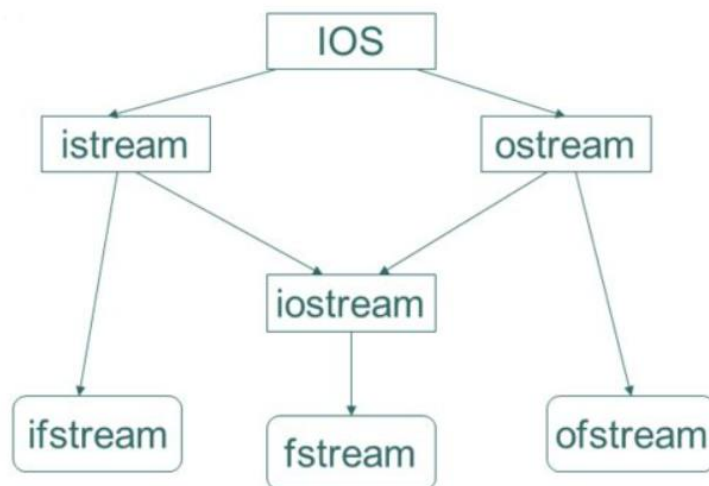
Файлы - это наборы данных, которые хранятся на диске или другом устройстве хранения информации. Файлы можно разделить на два типа: текстовые и двоичные.

Текстовый поток — это последовательность символов. Они организуются по строкам, каждая из которых заканчивается символом «конца строки». Конец самого файла обозначается символом «конца файла». При записи информации в текстовый файл, просмотреть который можно с помощью любого текстового редактора, все данные преобразуются к символьному типу и хранятся в символьном виде. При передаче символов из потока на экран, часть из них не выводится (например, символ возврата каретки, перевода строки). В текстовом режиме каждый разделительный символ строки автоматически преобразуется в пару (возврат каретки – переход на новую строку).

Текстовые файлы могут быть созданы и изменены с помощью любого текстового редактора, такого как Microsoft Word, Notepad и т.д. Каждый символ в текстовом файле кодируется в соответствующий ASCII-код (American Standard Code for Information Interchange).

Двоичный поток — это последовательность байтов, которые однозначно соответствуют тому, что находится на внешнем устройстве. В двоичных файлах информация считывается и записывается в виде блоков определенного размера, в которых могут храниться данные любого вида и структуры.

Двоичные файлы могут быть созданы и изменены с помощью специальных программ, таких как бинарные редакторы. Каждый бит в двоичном файле имеет определенное значение, которое может быть интерпретировано в соответствии с определенной схемой кодирования.



Различия между текстовыми и двоичными файлами:

Основные различия между текстовыми и двоичными файлами заключаются в способе хранения и интерпретации данных.

1) В текстовых файлах данные представлены в виде символов, в то время как в двоичных файлах данные представлены в виде битов. Текстовые файлы могут быть открыты и изменены любым текстовым редактором, в то время как для работы с двоичными файлами требуются специальные программы.

2) Текстовые файлы могут быть отформатированы и структурированы для лучшей читаемости, в то время как двоичные файлы обычно не содержат форматирования или структуры.

3) Текстовые файлы занимают меньше места на диске, чем двоичные файлы, потому что каждый символ занимает только один байт, в то время как каждый бит в двоичном файле занимает один бит.

4) Наконец, текстовые файлы могут быть легко читаемы и интерпретируемы человеком, в то время как двоичные файлы обычно требуют специальных программ и знаний для их интерпретации.

Режимы открытия файлов устанавливают характер использования файлов. Для установки режима в классе `ios_base` предусмотрены константы, которые определяют режим открытия файлов.

Константа	Описание
<code>ios_base::in</code>	открыть файл для чтения
<code>ios_base::out</code>	открыть файл для записи
<code>ios_base::ate</code>	при открытии переместить указатель в конец файла
<code>ios_base::app</code>	открыть файл для записи в конец файла
<code>ios_base::trunc</code>	удалить содержимое файла, если он существует
<code>ios_base::binary</code>	открытие файла в двоичном режиме

Обратите внимание на то, что флаги `ate` и `app` по описанию очень похожи, они оба перемещают указатель в конец файла, но флаг `app` позволяет производить запись, только в конец файла, а флаг `ate` просто переставляет флаг в конец файла и не ограничивает места записи.

Режимы открытия файлов можно устанавливать непосредственно при создании объекта или при вызове функции `open()`.

```
ofstream fout("study.txt", ios_base::app); // открываем файл для добавления
информации к концу файла
fout.open("study.txt", ios_base::app); // открываем файл для добавления
информации к концу файла
```

Режимы открытия файлов можно комбинировать с помощью поразрядной логической операции **или** `|`, например: `ios_base::out | ios_base::trunc` — открытие файла для записи, предварительно очистив его.

Объекты класса `ofstream`, при связке с файлами по умолчанию содержат режимы открытия файлов `ios_base::out | ios_base::trunc`.

`ofstream` и `ifstream` - это классы стандартной библиотеки C++, которые используются для работы с файлами.

`ofstream` - это класс, который используется для записи данных в файл. Он наследуется от класса `ostream`, который определяет операторы `<<` и `>>` для вывода и ввода данных. Для создания файла и записи данных в него необходимо создать объект класса `ofstream`. При создании объекта можно указать имя файла и режим открытия файла.

Вот пример кода, демонстрирующего использование класса `ofstream` :

```

#include <iostream>
#include <fstream>

using namespace std;

int main() {
    ofstream file("example.txt");

    if (file.is_open()) {
        file << "Hello, world!" << endl;
        file.close();
    }
    else {
        cout << "Ошибка открытия файла" << endl;
    }

    return 0;
}

```



example – Блокнот

Файл Правка Формат Вид Справка

Hello, world!

В данном примере мы создаем объект класса `ofstream` с именем "example.txt" и записываем в него строку "Hello, world!" с помощью оператора `<<`. Затем мы закрываем файл с помощью функции `close()`.

`ifstream` - это класс, который используется для чтения данных из файла. Он наследуется от класса `istream`, который также определяет операторы `>>` и `<<` для ввода и вывода данных. Для чтения данных из файла необходимо создать объект класса `ifstream`. При создании объекта можно указать имя файла и режим открытия файла.

Вот пример кода, демонстрирующего использование класса `ifstream`:

```


#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
    ifstream file("example.txt");
    string line;

    if (file.is_open()) {
        while (getline(file, line)) {
            cout << line << endl;
        }
    }
}

```

```
}  
file.close();  
} else {  
    cout << "Ошибка открытия файла" << endl;  
}  
  
return 0;  
}
```

A terminal window with a black background and white text displaying "Hello, world!".

Hello, world!

В данном примере мы создаем объект класса `ifstream` с именем "example.txt" и читаем данные из файла с помощью функции `getline()`. Затем мы выводим прочитанные данные на экран с помощью функции `cout`. В конце мы закрываем файл с помощью функции `close()`.

1. Абсолютный путь

Абсолютный путь указывает полный путь к файлу от корневой директории файловой системы. Например:

```
ifstream file("/home/user/Documents/example.txt");
```

2. Относительный путь

Относительный путь указывает путь к файлу относительно текущей директории. Например, если текущая директория - это директория `Documents`, а файл находится в поддиректории `example`, то относительный путь будет выглядеть так:

```
ifstream file("example/example.txt");
```

3. Использование переменных окружения

Вы можете использовать переменные окружения, такие как `HOME` или `USERPROFILE`, чтобы задать путь к файлу. Например:

```
string homeDir = getenv("HOME");  
ifstream file(homeDir + "/Documents/example.txt");
```

Для того, чтобы удалить файл, нужно использовать функцию `remove()` ("название файла"):

```
remove("study.txt");
```

Задания к лабораторной работе:

Написать функции:

1. Создание и запись данных в файл, название которого вводится с клавиатуры;
2. Чтение данных из файла файла (название файла не нужно вводить с клавиатуры);
3. Удаление содержимого из файла;
4. Удаление файла.

Дополнительные задания:

1. Напишите программу, которая считывает список файлов из указанной директории и выводит на экран их размеры в байтах.(3 балла)
2. Напишите программу, которая открывает двоичный файл, ищет в нем все вхождения некоторой последовательности байтов и заменяет их на другую последовательность байтов.(3 балла)
3. Напишите программу, которая создает двоичный файл, записывает в него некоторые данные (например, числа) и затем читает эти данные из файла.(3 балла)

Вопросы к лабораторной работе:

1. Что из себя представляет файл?
2. В чем разница между file.txt и file.dat?
3. Дайте определение текстовому и двоичному потоку?
4. Назовите различия между текстовым и двоичным потоком?
5. Какие основные операции выполняются с файлами?
6. Какие заголовочные файлы необходимо подключить для работы с файлами?
7. Аналогичен ли файловый ввод/вывод стандартному вводу/выводу? Почему?
8. Какие существуют способы чтения из файла?
9. Какие вы знаете константы для установки режима в классе ios_base, которые определяют режим открытия файлов?
10. Для чего нужны переменные окружения?
11. Расскажите про абсолютный и относительный путь?
12. Разница флагов ate и app?
13. Можно ли комбинировать режимы открытия файлов?

14. Какую функцию используют для удаления файла?
15. Для чего нужна функция `open()`?

Лабораторная работа №4. Индексирование записей.

Простой/сложный индекс

Struct (структура) - это тип данных в C++, который позволяет объединять несколько переменных разных типов в одну единицу данных. В структуре можно объединить переменные разных типов, такие как целые числа, числа с плавающей точкой, символьные строки, указатели и другие структуры.

Структуры объявляются с помощью ключевого слова "struct", за которым следует имя структуры и фигурные скобки, в которых указываются переменные, которые должны быть включены в структуру. Например, вот как объявляется простая структура с двумя полями:

```
struct Point {  
    int x;  
    int y;  
};
```

В этом примере мы объявили структуру "Point", которая состоит из двух целочисленных полей "x" и "y". После объявления структуры можно создать переменную этого типа и обращаться к полям структуры, как показано ниже:

```
Point p1 = { 1, 2 };  
p1.x = 3;  
p1.y = 4;
```

В этом примере мы создали переменную "p1" типа "Point" и задали значения ее полей "x" и "y". Затем мы обратились к полям структуры отдельно, чтобы изменить их значения.

Структуры могут быть использованы для создания сложных типов данных, таких как списки, деревья, графы и другие структуры данных. Они также могут быть использованы для передачи нескольких значений в функцию вместо передачи каждой переменной отдельно.

Массив структур - это массив, который содержит элементы структурного типа данных. Такой массив можно использовать для хранения набора структур, каждая из которых содержит набор переменных различных типов.

Чтобы объявить массив структур в C++, нужно сначала определить структуру, которая будет использоваться для создания элементов массива. Например, следующий код определяет структуру "Person", которая содержит два поля:

```
struct Person {  
    std::string name;  
    int age;  
};
```

Затем можно создать массив структур типа "Person", как показано ниже:

```
Person people[3];
```

К структурам данных в C++ применимы следующие операции:

Название операции	Знак операции
выбор элемента через имя (селектор)	. (точка)
выбор элемента через указатель (селектор)	-> (минус и знак больше)
присваивание	=
взятие адреса	&

Индексирование записей - это способ доступа к отдельным элементам записи (struct) в C++. Запись является типом данных, в котором связанные переменные объединяются в один объект.

Для доступа к отдельным полям внутри записи, можно использовать оператор доступа к членам (member access operator) - точку ".". Индексирование записей может использоваться для доступа к элементам в массиве записей или для доступа к элементам внутри одной записи.

Для индексирования записей в C++ используется оператор квадратных скобок "[]". Например, если у вас есть массив записей, вы можете обратиться к отдельным элементам массива, используя индексацию, как показано в следующем примере:

```
#include <iostream>  
#include <fstream>  
#include <string>  
using namespace std;  
  
int main() {  
    struct Person {  
        string name;  
        int age;  
    };
```

```

};

Person people[3];
people[0].name = "Alice";
people[0].age = 25;
people[1].name = "Bob";
people[1].age = 30;
people[2].name = "Charlie";
people[2].age = 35;

cout << people[1].name << " is " << people[1].age << " years old." << endl;
return 0;
}

```

```

Bob is 30 years old.

```

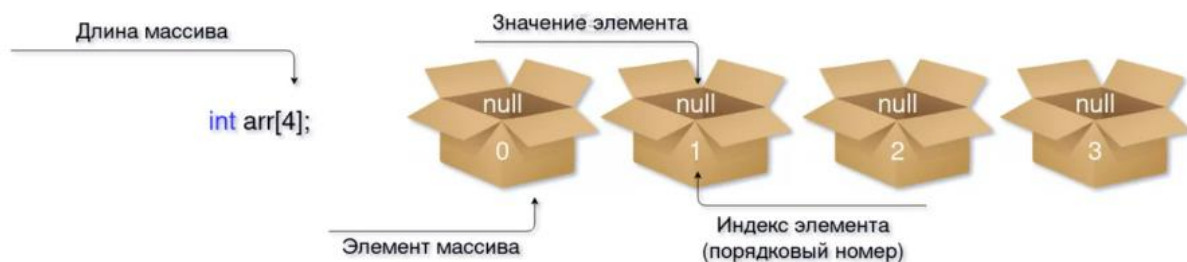
В этом примере мы создали структуру "Person" с полями "name" и "age". Затем мы объявили массив "people" из трех элементов, каждый из которых является записью типа "Person". Затем мы задали значения для полей каждого элемента массива и вывели на экран значения полей второго элемента массива.

Для структур данных, объявленных в программе, память выделяется автоматически средствами системы программирования либо на этапе компиляции, либо при активизации процедурного блока, в котором объявляются соответствующие переменные.

Главное заключается в том, что независимо от используемого языка программирования, имеющиеся в программе структуры данных не появляются "из ничего", а явно или неявно объявляются операторами создания структур. В результате этого всем структурам программы выделяется память для их размещения.

Простой индекс и сложный индекс - это две разные концепции в C++, связанные с обращением к элементам массива.

Простой индекс - это индекс, который используется для обращения к элементу массива, который находится на определенной позиции в массиве.



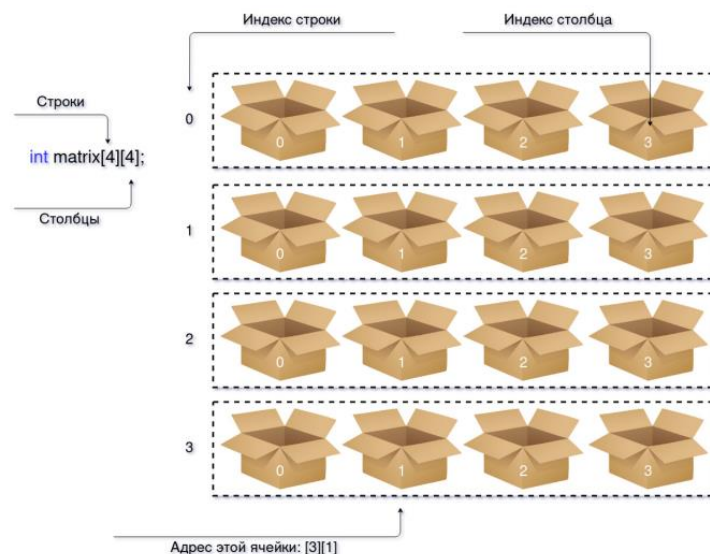
Можно дать так же второе определение: Простой индекс - это структура данных, которая содержит список значений ключа и ссылок на соответствующие записи в файле данных. Простой индекс может быть создан для любого поля в файле данных, которое может быть использовано для быстрого поиска. Простой индекс может быть создан для единственного поля или для нескольких полей в файле данных.

Например, если у вас есть массив чисел, то простой индекс используется для обращения к определенному элементу массива, как в следующем примере:

```
int arr[5] = { 1, 2, 3, 4, 5 };  
cout << arr[2]; // выведет 3, так как 3 находится на позиции 2 в массиве
```

В этом примере мы объявили массив "arr" из 5 элементов и вывели на экран значение элемента, который находится на позиции 2 (индекс 2) в массиве.

Сложный индекс - это индекс, который используется для обращения к элементу в многомерном массиве.



Можно дать так же второе определение: Сложный индекс - это структура данных, которая содержит несколько полей и ссылок на соответствующие записи в файле данных. Сложный индекс может быть создан для любой комбинации полей в файле данных, которая может быть использована для быстрого поиска. Сложный индекс может быть создан для нескольких полей, которые используются вместе для поиска.

Если у вас есть, например, двумерный массив, то для доступа к его элементам вам нужно использовать два индекса: первый индекс обозначает номер строки, а второй индекс - номер столбца.

Например:

```
int matrix[3][3] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };  
cout << matrix[1][2]; // выведет 6, так как 6 находится во второй строке и третьем столбце
```

В этом примере мы объявили двумерный массив "matrix" из 3 строк и 3 столбцов и вывели на экран значение элемента, который находится во второй строке и третьем столбце.

Основные операции, применимые к struct:

1. Присваивание (=):

Структуры могут быть присвоены друг другу с помощью оператора присваивания (=). Например:

```
struct Person {  
    std::string name;  
    int age;  
};  
Person p1 = { "Alice", 25 };  
Person p2 = p1; // присваивание p2 значений p1
```

2. Сравнение (==, !=, <, >, <=, >=):

Структуры могут быть сравниваемы между собой, если они содержат сравнимые поля. Например:

```
Person p1 = { "Alice", 25 };  
Person p2 = { "Bob", 30 };  
  
if (p1.age < p2.age) {  
    std::cout << p1.name << " is younger than " << p2.name << std::endl;  
}
```

3. Доступ к полям ("."):

Поля структуры можно получить, используя оператор доступа к полям (".") с именем структуры и именем поля. Например:

```
Person p = { "Alice", 25 };  
std::cout << p.name << " is " << p.age << " years old." << std::endl;
```

4. Инициализация:

Структуры могут быть инициализированы при объявлении или во время выполнения программы. Например:

```
Person p1 = { "Alice", 25 };  
Person p2;  
p2.name = "Bob";  
p2.age = 30;
```

5. Передача аргументов в функции:

Структуры могут быть переданы в функцию как параметры и возвращены из функции как результат. Например:

```
// Определение структуры Person с полями name и age  
...  
void printPerson(Person p);  
  
// Функция createPerson, которая создает новый объект Person с заданным именем и возрастом  
Person createPerson(string name, int age) {  
    Person p;  
    p.name = name;  
    p.age = age;  
    return p;  
}  
  
// Функция printPerson, которая выводит информацию о Person  
void printPerson(Person p) {  
    cout << p.name << " is " << p.age << " years old." << endl;  
}  
  
int main() {  
    Person p1 = createPerson("Alice", 25);  
    Person p2 = createPerson("Bob", 30);  
    printPerson(p1);  
    printPerson(p2);  
}
```

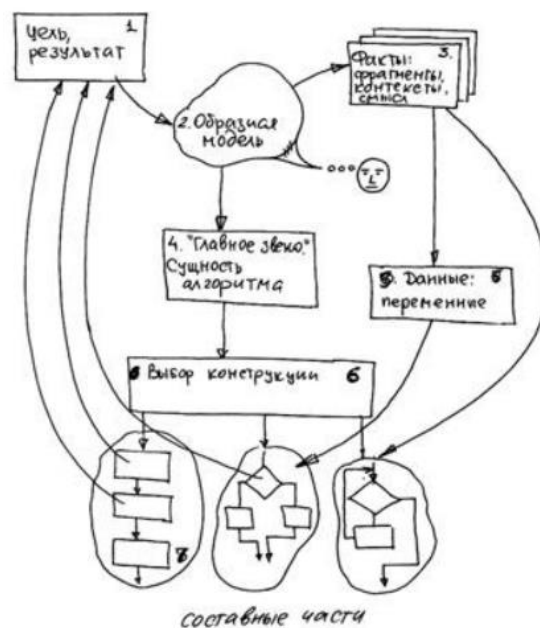
```
Alice is 25 years old.  
Bob is 30 years old.
```

Структурность данных и технология программирования:

При структурировании больших программных изделий возможно применение подхода, основанного на структуризации алгоритмов и известного как

"нисходящее" проектирование или "программирование сверху вниз", или подхода, основанного на структуризации данных и известного как "восходящее" проектирование или "программирование снизу вверх".

В первом случае структурируют прежде всего действия, которые должна выполнять программа. Большую и сложную задачу, стоящую перед проектируемым программным изделием, представляют в виде нескольких подзадач меньшего объема. Таким образом, модуль самого верхнего уровня, отвечающий за решение всей задачи в целом, получается достаточно простым и обеспечивает только последовательность обращений к модулям, реализующим подзадачи. На первом этапе проектирования модули подзадач выполняются в виде "заглушек". Затем каждая подзадача в свою очередь подвергается декомпозиции по тем же правилам. Процесс дробления на подзадачи продолжается до тех пор, пока на очередном уровне декомпозиции получают подзадачу, реализация которой будет вполне обозримой. В предельном случае декомпозиция может быть доведена до того, что подзадачи самого нижнего уровня могут быть решены элементарными инструментальными средствами (например, одним оператором выбранного языка программирования).



Чрезвычайно продуктивным технологическим приемом, связанным со структуризацией данных, является инкапсуляция. Смысл ее состоит в том, что сконструированный новый тип данных - "строительный блок" - оформляется таким образом, что его внутренняя структура становится недоступной для программиста-пользователя этого типа. Программист, использующий этот тип

данных в своей программе (в модуле более высокого уровня), может оперировать с данными этого типа только через вызовы функций, определенных для этого. Новый тип данных представляется для него в виде "черного ящика", для которого известны входы и выходы, но содержимое - неизвестно и недоступно. Инкапсуляция чрезвычайно полезна и как средство преодоления сложности, и как средство защиты от ошибок. Первая цель достигается за счет того, что сложность внутренней структуры нового типа данных и алгоритмов выполнения операций над ним исключается из поля зрения программиста-пользователя. Вторая цель достигается тем, что возможности доступа пользователя ограничиваются лишь заведомо корректными входными точками, следовательно, снижается и вероятность ошибок.

Указатель на структуру и массив структур:

Указатель на структуру - это переменная, которая содержит адрес в памяти структуры. Она позволяет работать с полями структуры, используя адрес этой структуры.

Для определения указателя на структуру необходимо сначала определить саму структуру.

```
struct student {  
    char name[50];  
    int age;  
    float gpa;  
};
```

Чтобы создать указатель на структуру student, необходимо использовать синтаксис указателей. Например:

```
struct student* ptr;
```

Этот код создает указатель ptr на тип struct student. Затем можно использовать оператор & для получения адреса структуры. Например:

```
struct student s;  
ptr = &s;
```

Чтобы обратиться к полям структуры через указатель, используйте операторы * и ->. Например:

```
(*ptr).age = 20;  
ptr->gpa = 3.5;
```


Эти операторы используются для доступа к полям структуры через указатель. Первый оператор (*ptr) разыменовывает указатель и возвращает значение структуры, а затем обращается к полю age. Второй оператор ptr->gpa автоматически разыменовывает указатель и обращается к полю gpa.

Указатель на массив структур - это способ обращения к элементам массива структур через указатель на первый элемент.

```
struct student {  
    char name[50];  
    int age;  
    float gpa;  
};  
  
struct student class[10];
```

Этот код определяет массив структур class из 10 элементов типа struct student. Каждый элемент массива содержит поля name, age и gpa.

Чтобы создать указатель на структуру student, необходимо использовать синтаксис указателей. Например:

```
struct student* ptr;
```

Этот код создает указатель ptr на тип struct student. Затем можно использовать оператор & для получения адреса структуры. Например:

```
struct student s;  
ptr = &s;
```

Этот код создает переменную s типа struct student и присваивает ей значения полей. Затем он создает указатель ptr на тип struct student и присваивает ему адрес структуры s.

Чтобы обратиться к полям структуры через указатель, используйте операторы * и ->. Например:

```
(*ptr).age = 20;  
ptr->gpa = 3.5;
```

Пример со списком рассылки:

```
// объявление массива структур  
....
```

```

void init_list(void)
{
    for (int t = 0; t < MAX; ++t) {
        addr_list[t].name = ""; // очистка полей структуры
        addr_list[t].street = "";
        addr_list[t].city = "";
        addr_list[t].state = "";
        addr_list[t].zip = 0;
    }
}

int menu_select(void)
{
    string s;
    int c;
    cout << "1. Введите имя\n";
    cout << "2. Удалите имя\n";
    cout << "3. Выведите список\n";
    cout << "4. Выход\n";
    do {
        cout << "\nВведите номер нужного пункта: ";
        getline(cin, s);
        c = stoi(s); // считывание номера выбранного пункта
    } while (c < 0 || c > 4); // проверка на корректность ввода
    return c;
}

int find_free(void)
{
    int t{}; // инициализация переменной t
    for (int t = 0; t < MAX && !addr_list[t].name.empty(); ++t); // поиск свободной структуры
    if (t == MAX) return -1; /* свободных структур нет */
    return t; // возврат номера свободной структуры
}

void enter(void)
{
    int slot;
    string s;

    slot = find_free(); // поиск свободной структуры
    if (slot == -1) {
        cout << "\nСписок заполнен";
        return;
    }
    cout << "Введите имя: ";
    getline(cin, addr_list[slot].name); // считывание поля имени
    cout << "Введите улицу: ";
    getline(cin, addr_list[slot].street); // считывание поля улицы
    cout << "Введите город: ";
    getline(cin, addr_list[slot].city); // считывание поля города
    cout << "Введите штат: ";
    getline(cin, addr_list[slot].state); // считывание поля штата
    cout << "Введите почтовый код: ";
    getline(cin, s);
    addr_list[slot].zip = stoul(s); // считывание поля почтового кода
}

void del(void)

```

```

{
    int slot;
    string s;
    cout << "Введите № записи: ";
    getline(cin, s);
    slot = stoi(s); // считывание номера записи для удаления
    if (slot >= 0 && slot < MAX)
        addr_list[slot].name = ""; // удаление поля имени
}

void list(void)
{
    for (int t = 0; t < MAX; ++t) {
        if (!addr_list[t].name.empty()) { // проверка, что структура не пустая
            cout << addr_list[t].name << endl;
            cout << addr_list[t].street << endl;
            cout << addr_list[t].city << endl;
            cout << addr_list[t].state << endl;
            cout << addr_list[t].zip << endl << endl; // вывод полей структуры
        }
    }
    cout << "\n\n";
}

int main(void)
{
    // инициализация списка
    ....
    // выбор пункта меню
    ....
    // добавление записи
    ....
    // удаление записи
    ....
    // вывод списка
    ....
    // выход из программы
}
}
return 0;
}

```

```
1. Введите имя
2. Удалите имя
3. Выведите список
4. Выход
```

```
Введите номер нужного пункта: 1
Введите имя: Kirill
Введите улицу: Beloruskaya
Введите город: Minsk
Введите штат: Belarus
Введите почтовый код: 342338
```

```
1. Введите имя
2. Удалите имя
3. Выведите список
4. Выход
```

```
Введите номер нужного пункта: 3
Kirill
Beloruskaya
Minsk
Belarus
342338
```

```
1. Введите имя
2. Удалите имя
3. Выведите список
4. Выход
```

```
Введите номер нужного пункта: 0
1. Введите имя
2. Удалите имя
3. Выведите список
4. Выход
```

```
Введите номер нужного пункта: 2
Введите № записи: 0
1. Введите имя
2. Удалите имя
3. Выведите список
4. Выход
```

```
Введите номер нужного пункта: 3
1. Введите имя
2. Удалите имя
3. Выведите список
4. Выход
```

```
Введите номер нужного пункта: 4
```

Задание к лабораторной работе:

Основываясь на вашей схеме базы данных из лабораторной работы №2, реализовать структуры, создать индексы для каждой структуры, а так же реализовать вывод структур, линейный поиск данных по индексу (при вводе индекса, выводило/удаляло всю структуру).

Дополнительные задания:

1. Создайте простой индекс для поля "номер телефона" в файле контактов. Напишите программу для поиска контактов по номеру телефона с использованием этого индекса. (3 балла)
2. Создайте сложный индекс для полей "имя" и "город" в файле клиентов. Напишите программу для поиска клиентов по имени и городу с использованием этого индекса. (3 балла)
3. Создайте индекс для нескольких файлов данных и напишите программу для поиска записей в этих файлах с использованием этого индекса. (3 балла)

Вопросы к лабораторной работе:

1. Что такое структура в C++?
2. Какие операции применимы к структурам данных в C++?
3. С помощью какого ключевого слова объявляются структуры?
4. Для чего могут быть использованы структуры?
5. Дайте определение массиву структур?
6. Как объявить массив структур?
7. Что такое индексирование записей?
8. Какой оператор используется для индексирования записей в C++?
9. На каком этапе выделяется память для структур данных?
10. Что такое простой индекс?
11. Что такое сложный индекс?
12. Какие основные операции, применимые к struct, вы знаете?
13. Расскажите про структурность данных и про технологию программирования?

Лабораторная работа №5. Запись/чтение массива структур в файл

Для начала рассмотрим вложенные структуры и инициализацию структур.

Инициализация структур

Инициализация структур путем присваивания значений каждому члену по порядку — занятие довольно громоздкое (особенно, если этих членов много), поэтому в языке C++ есть более быстрый способ инициализации структур — с помощью **списка инициализаторов**. Он позволяет инициализировать некоторые или все члены структуры во время объявления переменной типа struct:

```
struct Employee
{
    short id;
    int age;
    double salary;
};

Employee john = { 5, 27, 45000.0 }; // john.id = 5, john.age = 27, john.salary = 45000.0
Employee james = { 6, 29 }; // james.id = 6, james.age = 29, james.salary = 0.0 (инициализация по умолчанию)
```

Если в списке инициализаторов не будет одного или нескольких элементов, то им присвоятся значения по умолчанию (обычно, 0). В примере, приведенном выше, члену james.salary присваивается значение по умолчанию 0.0, так как мы сами не предоставили никакого значения во время инициализации.

Вложенные структуры

Одни структуры могут содержать другие структуры. Например:

```
struct Employee
{
    short id;
    int age;
    double salary;
};

struct Company
{
    Employee CEO; // Employee - это структура внутри структуры Company
    int numberOfEmployees;
```

```
};
```

```
Company myCompany;
```

В этом случае, если бы мы хотели узнать, какая зарплата у CEO (исполнительного директора), то нам бы пришлось использовать оператор выбора членов дважды:

```
myCompany.CEO.salary = 2000;
```

Сначала мы выбираем поле CEO из структуры myCompany, а затем поле salary из структуры Employee.

Вы можете использовать вложенные списки инициализаторов с вложенными структурами:

```
Company myCompany = { { 3, 35, 55000.0f }, 7 };
```

Теперь рассмотрим массив структур: структуры часто образуют массивы.

Например, чтобы объявить 100-элементный массив структур типа addr, который был определен ранее, напишите следующее:

```
Company myCompany[100];
```

Это выражение создаст 100 наборов переменных, каждый из которых организован так, как определено в структуре Company.

Чтобы получить доступ к определенной структуре, указывайте имя массива с индексом. Например, чтобы вывести зарплату из третьей структуры, напишите следующее:

```
cout << myCompany[2].CEO.salary;
```

Как и в других массивах переменных, в массивах структур индексирование начинается с 0.

Запись/чтение массива структур в файл - это процесс сохранения массива структур в файл и последующего чтения из него. Для записи массива структур в файл необходимо создать файл и последовательно записывать каждую структуру в файл. Для чтения из файла необходимо открыть файл и последовательно считывать каждую структуру из файла.

№	Имя	Возраст	Рост	Вес	Объемы			Английский
1	Olga	17	172	55	87	60	88	1
2	Elena	23	175	54	90	65	89	0
...
7	Katya	19	171	50	89	62	92	1

Нюансы и тонкости при записи/чтении массива структур в файл

При записи и чтении массива структур в файл важно учитывать, что структуры могут содержать различные типы данных, в том числе и пользовательские типы данных. При записи и чтении, необходимо обеспечить правильное преобразование типов данных, чтобы сохранить и восстановить значения полей структуры.

Также важно учитывать порядок записи и чтения полей структуры. При записи структуры в файл, следует записывать поля в том же порядке, в котором они объявлены в структуре. При чтении структуры из файла, необходимо читать поля в том же порядке, в котором они были записаны.

Для удобства чтения и записи структур в файл, можно использовать специальные разделители, например, символы перевода строки \n или запятые ,. Эти разделители позволяют разделять поля структур и упрощают чтение и запись данных.

Также можно использовать двоичный формат записи файлов, который позволяет записывать данные в более компактном и быстром формате, чем текстовые файлы.

Для чтения из файла можно использовать оператор >> или функцию getline(). Для записи в файл используется оператор <<. Для открытия файла в режиме чтения и записи используется объект класса fstream.

Для сохранения массива структур в файл можно использовать несколько подходов, например:

1. Запись структур в файл последовательно, как один непрерывный блок данных.
2. Запись каждой структуры в файл по отдельности, с использованием разделителя между структурами.

3. Запись структур в файл в виде таблицы, где каждая строка представляет одну структуру, а каждый столбец - одно поле структуры.

В зависимости от требований к эффективности работы с файлами и удобства чтения и записи данных, можно выбрать подходящий формат записи массива структур в файл.

Пример:

Для того, чтобы записать массив структур в файл или же считать его с файла, используем библиотеку `fstream` из Лабораторной работы №3. Пример:

```
// Создаем структуру из элементов типа string
....
// Функция для чтения данных из файла
void ReadingData(Initial* (&d), int& n, string fileName)
{
    setlocale(LC_ALL, "rus");
    // Создаем поток для чтения из файла
    ifstream reading(fileName);
    // Условие: если файл открылся
    if (reading) {
        reading >> n; // Считываем количество записей из файла
        d = new Initial[n]; // Создаем динамический массив структур размером n
        for (int i = 0; i < n; i++) {
            // Считываем данные из файла и записываем в соответствующие поля структуры
            reading >> d[i].surname;
            reading >> d[i].name;
            reading >> d[i].patronymic;
        }
        cout << "Данные считаны!" << endl;
    }
    else
        cout << "Ошибка открытия файла!" << endl;
    // Закрываем поток чтения
    reading.close();
}

// Функция для записи данных в файл
void SavingData(Initial* d, int n, string fileName)
{
    setlocale(LC_ALL, "rus");
    // Создаем поток для записи в файл
    // Открываем файл и делаем его пустым
    ofstream record(fileName, ios::out);

    // Условие: если файл открылся
    if (record) {
        record << n << endl; // Записываем количество записей в файл

        for (int i = 0; i < n; i++) {
            // Записываем данные структуры в файл, разделяя их переводом строки
            record << d[i].surname << endl;
            record << d[i].name << endl;
        }
    }
}
```

```

        record << d[i].patronymic << endl;
    }
    cout << "Данные записаны!" << endl;
}
else
    cout << "Ошибка открытия файла!" << endl;
// Закрываем поток записи
record.close();
}

int main()
{
    setlocale(LC_ALL, "rus");
    // Массив структур
    ....
    // Читаем данные из файла "Input.txt"
    ....
    // Записываем данные в файл "Output.txt"
    ....
    // Освобождаем память, выделенную под массив структур
    ....
}

```

Input – Блокнот

Файл Правка Формат Вид Справка

аааааааа
 бббббббб
 вввввввв
 гггггггг
 дддддддд
 ееееееее

Консоль отладки Microsoft Visual Studio

Данные считаны!
 Данные записаны!

Output – Блокнот

Файл Правка Формат Вид Справка

2

Первая функция ReadingData отвечает за чтение данных из файла. В нее передаются ссылки на массив структур Initial* (&d), количество записей int&n и имя файла string fileName. Функция открывает файл, считывает из него количество записей, создает динамический массив структур размером n и последовательно считывает данные из файла, записывая их в поля структуры. После чтения данных из файла функция закрывает поток чтения.

Вторая функция SavingData отвечает за запись данных в файл. В нее передаются ссылки на массив структур Initial* d, количество записей int n и имя файла string fileName. Функция открывает файл для записи, записывает в

него количество записей, а затем последовательно записывает данные из массива структур в файл, разделяя их переводом строки.

Задание к лабораторной работе:

В соответствии со своими заданиями из лабораторной работы №2, составить массив структур для своей БД и реализовать функции считывания с файла и вывод структуры на экран, записи в файл и добавления данных в массив. Для функций считывание и записи должен быть выбор (в виде меню), в котором пользователь может сам ввести название файла с клавиатуры либо использовать уже заготовленный файл.

Дополнительные задания:

1. Создайте структуру с полями "имя", "возраст" и "адрес". Создайте массив из 4 структур и заполните его данными. Запишите массив структур в файл. (3 балла)
2. Реализуйте функцию поиска в массиве структур по заданному критерию (например, возрасту или адресу). Напишите программу, которая находит все структуры, соответствующие заданному критерию. (3 балла)
3. Напишите программу, которая считывает массив структур из файла. Выведите на экран содержимое считанного массива структур. (3 балла)

Вопросы к лабораторной работе:

1. Какой самый быстрый способ инициализации структур?
2. Могут ли структуры содержать другие структуры?
3. Какие вы знаете нюансы и тонкости при записи/чтении массива структур в файл?
4. Назовите функции, которые можно использовать для записи и чтения двоичных файлов
5. Какие вы знаете подходы для сохранения массива структур в файл?
6. Какое значение присвоится, если в списке инициализаторов не будет одного или нескольких элементов?
7. Могут ли одни структуры содержать другие?
8. Можно ли использовать вложенные списки инициализаторов с вложенными структурами?
9. Как получить доступ к определенной структуре?
10. С какого элемента в массивах структур начинается индексирование?
11. Могут ли структуры содержать различные типы данных?

12. Следует ли записывать поля в том же порядке, в котором они объявлены в структуре?

Лабораторная работа №6. Редактирование файлов: удаление, изменение поля

Наиболее понятным и эффективным способом удаления и редактирования данных в C++ можно считать следующие функции:

Функция выборочного удаления:

Функция выборочного удаления в C++ - это функция, позволяющая удалить из файла определенную строку или группу строк, не изменяя содержимое других строк в файле.

Для реализации функции выборочного удаления можно использовать следующий алгоритм:

1. Открыть файл для чтения и создать временный файл для записи.
2. Считать строки из оригинального файла и записывать их во временный файл.
3. При считывании строк проверять, соответствует ли текущая строка той строке, которую нужно удалить. Если да, то не записывать ее во временный файл.
4. Закрыть оба файла и переименовать временный файл в оригинальный файл.

Использование функции выборочного удаления может быть полезным при работе с большими файлами, когда необходимо удалить только часть информации из файла, не затрагивая остальное содержимое.

Нюансы при использовании функции выборочного удаления:

1. Изменение индексов элементов. При удалении элемента из массива, индексы всех последующих элементов смещаются на одну позицию влево. Это может привести к ошибкам, если в программе используются жестко закодированные индексы элементов массива. Чтобы избежать этой проблемы, можно использовать итераторы или ссылки на элементы массива.
2. Выделение и освобождение памяти. При удалении элемента из массива, необходимо выделить новый массив, который будет содержать оставшиеся элементы. При этом надо учитывать, что выделение и освобождение памяти может занимать много времени и ресурсов, особенно если массив большой. Чтобы избежать этой проблемы, можно использовать другие структуры данных, которые позволяют эффективно добавлять и удалять элементы, например, связный список.
3. Нужна проверка наличия элемента. Прежде чем удалять элемент из массива, необходимо проверить, существует ли он в массиве. Если элемента нет в массиве, программа может завершиться аварийно или удалить неправильный элемент.
4. Необходимость сохранения порядка элементов. Если порядок элементов в массиве важен, необходимо убедиться, что он сохраняется при

удалении элементов. В некоторых случаях может быть лучше использовать другую структуру данных, которая позволяет сохранять порядок элементов, например, упорядоченный массив или дерево.

5. Необходимость правильной реализации копирования. При копировании элементов массива необходимо использовать правильную реализацию копирования, чтобы избежать утечек памяти или неожиданного поведения программы.

Пример:

```
....  
  
struct Initial {  
    string surname,  
        name,  
        patronymic;  
};  
  
struct Date {  
    int day,  
        month,  
        year;  
};  
  
struct Data {  
    Initial _initial;  
    Date _date;  
};  
  
void Copy(Data* (&d_n), Data* (&d_o), int n)  
{  
    for (int i = 0; i < n; i++) {  
        d_n[i] = d_o[i];  
    }  
}  
  
void DeleteData(Data* (&d), int& n)  
{  
    int _n;  
    cout << "Введите номер элемента (от 1 до " << n << "): ";  
    cin >> _n;  
    _n--;  
    system("cls");  
  
    if (_n >= 0 && _n < n) {  
        // временный массив  
        Data* buf = new Data[n];  
  
        Copy(buf, d, n);  
  
        // выделяем новую память  
        --n;  
        d = new Data[n];  
    }
```

```

        int q = 0;

        // заполняем неудаленные данные
        for (int i = 0; i <= n; i++) {
            if (i != _n) {
                d[q] = buf[i];
                ++q;
            }
        }

        system("cls");
        delete[] buf;
        cout << "Данные удалены!" << endl;
    }
    else
        cout << "Номер введен неверно!" << endl;
}
int main()
{
    ....
    // переменная хранящая кол-во структур массива
    ....
    // массив данных
    ....
    //ввод из файла
    ....
}

```

В данном и последующих примерах данные вводятся через функцию ReadingData() (из лабораторной работы №5) из файла Input.txt.

Для удаления данных из массива используется функция DeleteData, которая принимает ссылку на массив структур и количество элементов в массиве. Функция запрашивает у пользователя номер элемента, который нужно удалить, после чего создает временный массив, копирует в него все элементы из исходного массива, кроме того, который нужно удалить, выделяет новую память для исходного массива, копирует все элементы из временного массива в исходный массив и освобождает память, занимаемую временным массивом.

Также в программе используется функция Copy, которая копирует данные из одного массива в другой.

Функция полного удаления:

Для того, чтобы полностью удалить данные нам не нужна новая функция, для этого достаточно в main очистить динамический массив с помощью delete. И создать новый динамический массив.

```

amountOfData = 0;
delete[] d;
d = new Data[amountOfData];

```

Функция изменения:

```
....

struct Initial {
    string surname,
        name,
        patronymic;
};

struct Date {
    int day,
        month,
        year;
};

struct Data {
    Initial _initial;
    Date _date;
};

void Copy(Data* (&d_n), Data* (&d_o), int n)
{
    for (int i = 0; i < n; i++) {
        d_n[i] = d_o[i];
    }
}

void DataChange(Data* (&d), int n)
{
    int _n;
    cout << "Введите номер элемента (от 1 до " << n << "): ";
    cin >> _n;
    _n--;
    system("cls");

    // проверка, что ввели правильное значение
    if (_n >= 0 && _n < n) {
        cout << "Введите ФИО: ";
        cin >> d[_n]._initial.surname;
        cin >> d[_n]._initial.name;
        cin >> d[_n]._initial.patronymic;

        cout << "Введите дату: ";
        cin >> d[_n]._date.day;
        cin >> d[_n]._date.month;
        cin >> d[_n]._date.year;
    }
}
```



```

        system("cls");

        cout << "Данные изменены!" << endl;
    }
    else
        cout << "Номер введён неверно!" << endl;
}

int main()
{
    // переменная хранящая кол-во структур массива
    ....
    // массив данных
    ....
    //ввод из файла
    ....
}

```

в данном коде есть функция DataChange, которая позволяет изменять данные в массиве структур Data. Эта функция принимает в качестве параметра указатель на массив структур Data и количество элементов в этом массиве. Затем функция запрашивает у пользователя номер элемента, который нужно изменить, и запрашивает новые значения для ФИО и даты. После ввода новых данных функция выводит сообщение об успешном изменении данных.

Задание для лабораторной работы №6:

Написать функции выборочного и полного удаления массива структур для вашей бд, а также функцию изменения массива структур для вашей БД (из лабораторной работы №2).

Дополнительные задания:

1. Создайте файл с именем "test.txt" и запишите в него несколько строк текста. Откройте файл в режиме чтения и записи, замените одну из строк на новую и сохраните изменения. (3 балла)
2. Создайте файл с именем "numbers.txt" и запишите в него несколько чисел. Откройте файл в режиме чтения и записи и замените каждое число на новое, увеличенное на 10. (3 балла)
3. Создайте файл с именем "words.txt" и запишите в него список слов. Откройте файл в режиме чтения и записи и удалите все слова, начинающиеся на определенную букву. (3 балла)

Вопросы к лабораторной работе:

1. Какой наиболее понятный и эффективный способ удаления и редактирования данных для баз данных в C++?
2. Какой алгоритм можно использовать для реализации функции выборочного удаления?
3. Нюансы при использовании функции выборочного удаления?
4. Для чего нужна функция полного удаления?
5. Для чего нужна функция изменения?
6. Что происходит с индексами элементов при удалении?
7. Нужно ли проверять наличие элемента в массиве, прежде чем удалять этот элемент из массива?
8. Расскажите про необходимость правильной реализации копирования
9. При каких условиях может быть полезно использование функции выборочного удаления?
10. Почему важно сохранение порядка элементов?

Лабораторная работа №7. Сортировка записей

Было подсчитано, что до четверти времени централизованных компьютеров уделяется сортировке данных. Это потому, что намного легче найти значение в массиве, который был заранее отсортирован. В противном случае поиск немного похожит на поиск иголки в стоге сена.

Алгоритмы сортировки — это алгоритмы для упорядочивания элементов в массиве. Упорядочивание элементов в массиве необходимо для анализа данных или для дальнейшей работы с этим массивом уже других алгоритмов, например, алгоритм бинарного поиска работает только в заранее отсортированном массиве.

Алгоритмы сортировки оцениваются по скорости выполнения и эффективности использования памяти.

Для сортировки записей **в файле** необходимо считать данные из файла в память, выполнить сортировку и записать отсортированные данные обратно в файл. Для чтения из файла можно использовать оператор `>>` или функцию `getline()`. Для записи в файл используется оператор `<<`.

Сложность алгоритмов оценивают с помощью нотации O большое (Big O). O большое показывает, как сильно растут затраты на алгоритм, при увеличении количества элементов в массиве.

В анализе алгоритмов, мы часто рассматриваем лучший, худший и средний случаи времени выполнения. Давайте разберем каждый из этих случаев:

Лучший случай (Best Case)

Лучший случай времени выполнения алгоритма определяется как минимальное количество времени, которое потребуется алгоритму для выполнения задачи на входных данных. Этот случай редко рассматривается при анализе алгоритмов, так как он обычно представляет собой оптимистичный сценарий, который может не иметь практической ценности.

Пример: В алгоритме линейного поиска лучший случай происходит, когда искомый элемент находится в первой позиции массива. В этом случае алгоритму потребуется выполнить только одну операцию.

Худший случай (Worst Case)

Худший случай времени выполнения алгоритма определяется как максимальное количество времени, которое потребуется алгоритму для выполнения задачи на входных данных. Анализ худшего случая является важным, так как он предоставляет верхнюю границу на время выполнения алгоритма, что позволяет оценить, насколько плохо алгоритм может работать в самых неблагоприятных условиях.

Пример: В алгоритме линейного поиска худший случай происходит, когда искомый элемент находится в последней позиции массива или его нет в

массиве. В этом случае алгоритму потребуется выполнить N операций, где N - размер массива.

Средний случай (Average Case)

Средний случай времени выполнения алгоритма определяется как среднее время выполнения алгоритма на всех возможных входных данных. Анализ среднего случая часто является наиболее полезным, так как он предоставляет представление о том, как алгоритм будет работать в "типичных" условиях.

Пример: В алгоритме линейного поиска средний случай можно оценить как выполнение $N/2$ операций, если предположить, что искомый элемент равновероятно может быть на любой позиции массива.

Биг-Омега (Omega) и Биг-Тетта (Theta) - это два других понятия, которые используются для оценки времени и сложности алгоритмов в C++.

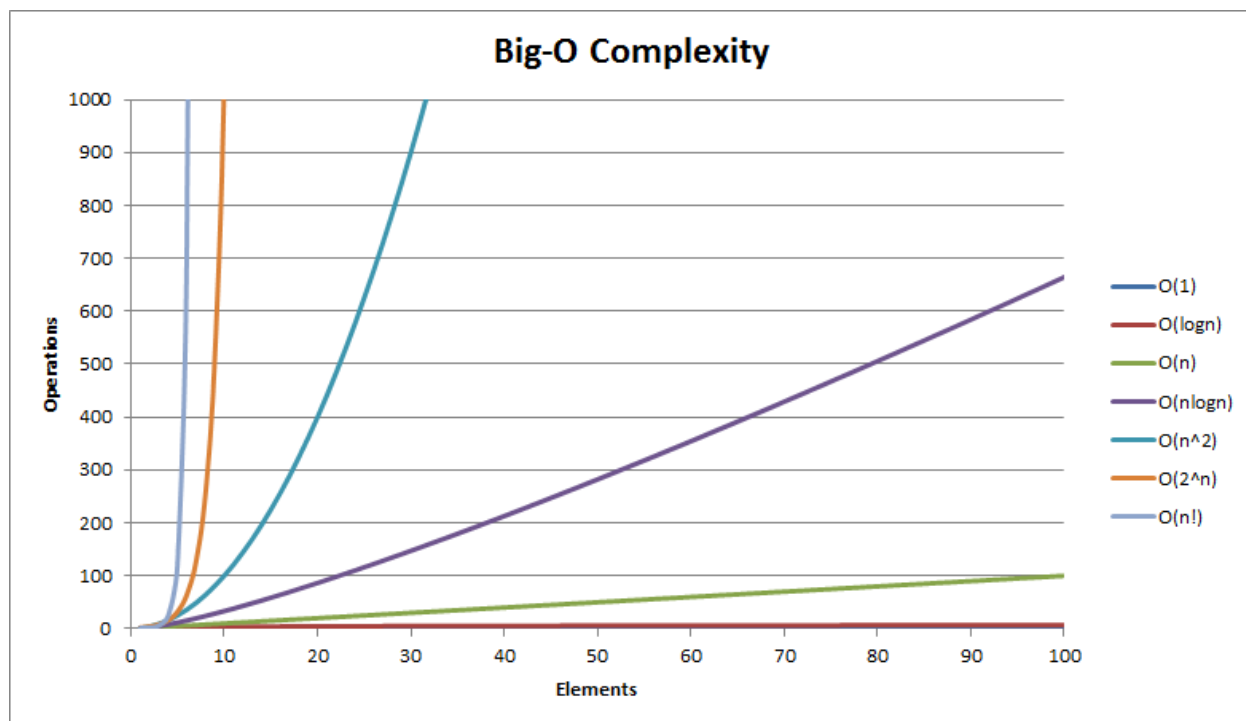
Биг-Омега (Omega) говорит о нижней границе времени выполнения алгоритма. Если мы говорим, что время выполнения алгоритма составляет $\Omega(n)$, то это означает, что в наихудшем случае алгоритм будет выполняться не быстрее, чем n единиц времени.

Биг-Тетта (Theta) говорит о верхней границе времени выполнения алгоритма. Если мы говорим, что время выполнения алгоритма составляет $\Theta(n)$, то это означает, что в наихудшем случае алгоритм будет выполняться не медленнее, чем n единиц времени и не быстрее, чем n единиц времени.

Важно отметить, что биг-Омега и биг-Тетта могут быть одинаковыми для одного и того же алгоритма, но они также могут быть разными. Поэтому, когда мы говорим о времени выполнения алгоритма, мы должны учитывать и биг-Омега, и биг-Тетта.

Например, если мы знаем, что алгоритм имеет сложность $\Omega(n)$ и $\Theta(n^2)$, это означает, что в наихудшем случае алгоритм будет выполняться не медленнее, чем n единиц времени, но может быть медленнее, чем n^2 единиц времени. В то же время, в лучшем случае алгоритм будет выполняться не быстрее, чем n^2 единиц времени, но может быть быстрее, чем n^2 единиц времени.

На картинке ниже рассмотрены различные способы роста алгоритмов.



Самыми популярными и частоиспользуемыми алгоритмами сортировки являются: пузырьковая сортировка, быстрая сортировка, сортировка выбором, сортировка вставками, сортировка слиянием.

Ниже представлены лучшие, худшие и средние случаи времени выполнения алгоритмов.

Алгоритм	Временная сложность		Пространственная сложность	
	Наилучший случай	Средний случай	Наихудший случай	Наихудший случай
Быстрая сортировка	$O(n \log(n))$	$O(n \log(n))$	$O(n^2)$	$O(\log(n))$
Сортировка слиянием	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	$O(n)$
Timsort	$O(n)$	$O(n \log(n))$	$O(n \log(n))$	$O(n)$
Пирамидальная сортировка	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	$O(1)$
Сортировка пузырьком	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Сортировка вставками	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Сортировка выбором	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Сортировка Шелла	$O(n)$	$O((n \log(n))^2)$	$O((n \log(n))^2)$	$O(1)$
Блочная сортировка	$O(n+k)$	$O(n+k)$	$O(n^2)$	$O(n)$
Поразрядная сортировка	$O(nk)$	$O(nk)$	$O(nk)$	$O(n+k)$

Так же есть шейкерная сортировка, сортировка расчёской, сортировка Шелла, гномья сортировка и многие другие, но используются намного реже, чем представленные выше. (по этой ссылке можно подробнее почитать про другие алгоритмы сортировок: [Описание алгоритмов сортировки и сравнение их производительности / Хабр \(habr.com\)](https://habr.com/ru/articles/444444/))

Принцип «разделяй и властвуй»

При подходе «разделяй и властвуй» задача делится на мелкие подзадачи, каждая из которых решается независимо. При их делении на еще более мелкие подзадачи в конце концов настает момент, когда дальнейшее деление невозможно. Эти мельчайшие «атомарные» подзадачи и решаются. Решения всех подзадач в итоге объединяются, и получается решение исходной задачи.

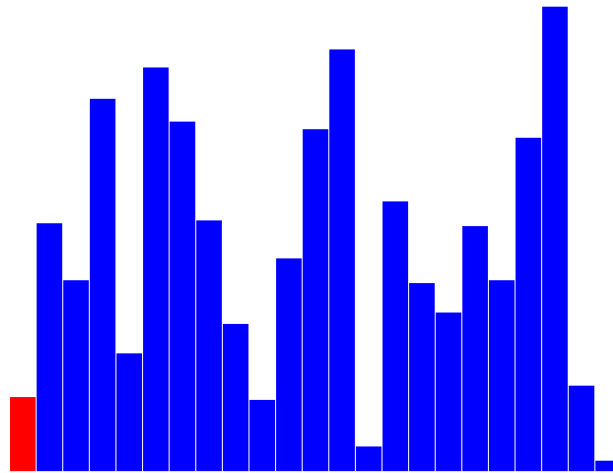
В целом подход «разделяй и властвуй» рассматривается как трехэтапный процесс:

1. Программа разбивается на независимые подзадачи(части), которые не пересекаются между собой.
2. Каждая подзадача решается отдельно и независимо от других частей.
3. Из отдельных решений подзадач строится решение исходной задачи.

Рассмотрим основные алгоритмы сортировки:

Сортировка выбором (Selection sort)

Для того, чтобы отсортировать массив в порядке возрастания, следует на каждой итерации найти элемент с наибольшим значением. С ним нужно поменять местами последний элемент. Следующий элемент с наибольшим значением становится уже на предпоследнее место. Так должно происходить, пока элементы, находящиеся на первых местах в массиве, не окажутся в надлежащем порядке.



(нажми два раза, чтобы оживить картинку)

```
void SelectionSort(int* a, int n)
{
    int smallest_id;

    for (int i = 0; i < n; i++) {
        smallest_id = i;
        for (int j = i + 1; j < n; j++) {
            if (a[j] < a[smallest_id])
                smallest_id = j;
        }
        //меняем местами элементы
        swap(a[smallest_id], a[i]);
    }
}
```

Пузырьковая сортировка (Bubble sort)

При пузырьковой сортировке сравниваются соседние элементы и меняются местами, если следующий элемент меньше предыдущего. Требуется несколько проходов по данным. Во время первого прохода сравниваются первые два элемента в массиве. Если они не в порядке, они меняются местами и затем сравниваются элементы в следующей паре. При том же условии они так же меняются местами. Таким образом сортировка происходит в каждом цикле пока не будет достигнут конец массива.

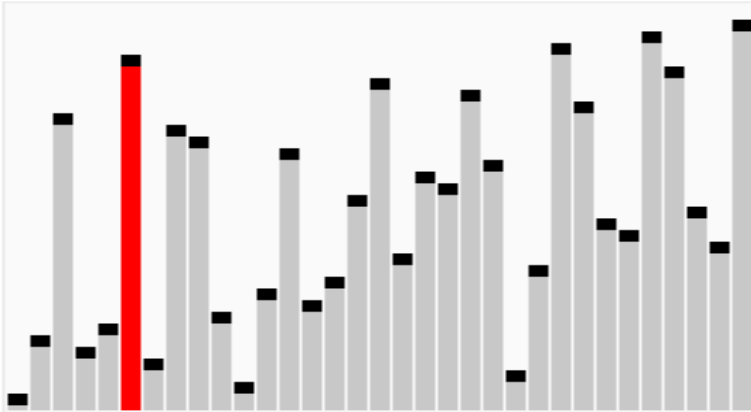
Сложность по времени

Худшее время: $O(n^2)$

Среднее время: $O(n^2)$

Лучшее время: $O(n)$

Затраты памяти: $O(1)$



(нажми два раза, чтобы оживить картинку)

```
void BubbleSort(int* a, int n)
{
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (a[i] > a[j]) {
                //меняем местами элементы
                swap(a[i], a[j]);
            }
        }
    }
}
```

Сортировка вставками (Insertion sort)

При сортировке вставками массив разбивается на две области: упорядоченную и неупорядоченную. Изначально весь массив является неупорядоченной областью. При первом проходе первый элемент из неупорядоченной области изымается и помещается в правильное положение в упорядоченной области. На каждом проходе размер упорядоченной области возрастает на 1, а размер неупорядоченной области сокращается на 1. Основной цикл работает в интервале от 1 до $N-1$. На j -й итерации элемент $[i]$ вставлен в правильное положение в упорядоченной области. Это сделано путем сдвига всех элементов упорядоченной области, которые больше, чем $[i]$,

на одну позицию вправо. $[i]$ вставляется в интервал между теми элементами, которые меньше $[i]$, и теми, которые больше $[i]$.

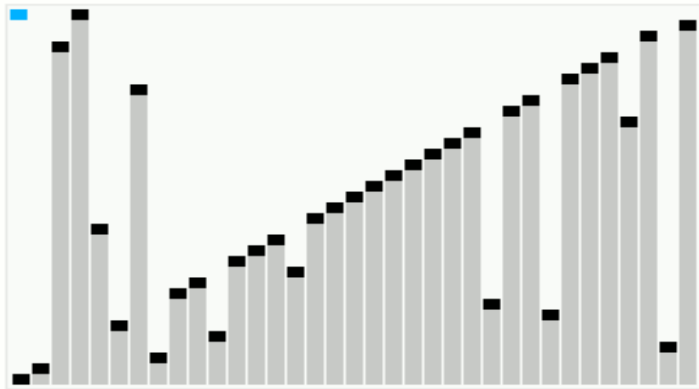
Сложность по времени

Худшее время: $O(n^2)$ для сравнений и перестановок

Среднее время: $O(n^2)$ для сравнений и перестановок

Лучшее время: $O(n)$ для сравнений, $O(1)$ для перестановок

Затраты памяти: $O(n)$ основной, $O(1)$ дополнительной



(нажми два раза, чтобы оживить картинку)

```
void insertionSort(int data[], int lenD)
{
    int key = 0;
    int i = 0;
    for (int j = 1; j < lenD; j++) {
        key = data[j];
        i = j - 1;
        while (i >= 0 && data[i] > key) {
            data[i + 1] = data[i];
            i = i - 1;
            data[i + 1] = key;
        }
    }
}
```

Сортировка слиянием (Merge sort)

При рекурсивной сортировке слиянием массив сначала разбивается на мелкие кусочки - на первом этапе - на состоящие из одного элемента. Затем эти

кусочки объединяются в более крупные кусочки - по два элемента и элементы при этом сравниваются, а в результате в новом кусочке меньший элемент занимает место слева, а больший - справа. Далее происходит слияние в ещё более крупные кусочки и так до конца алгоритма, когда все кусочки будут объединены в один, уже отсортированный массив.

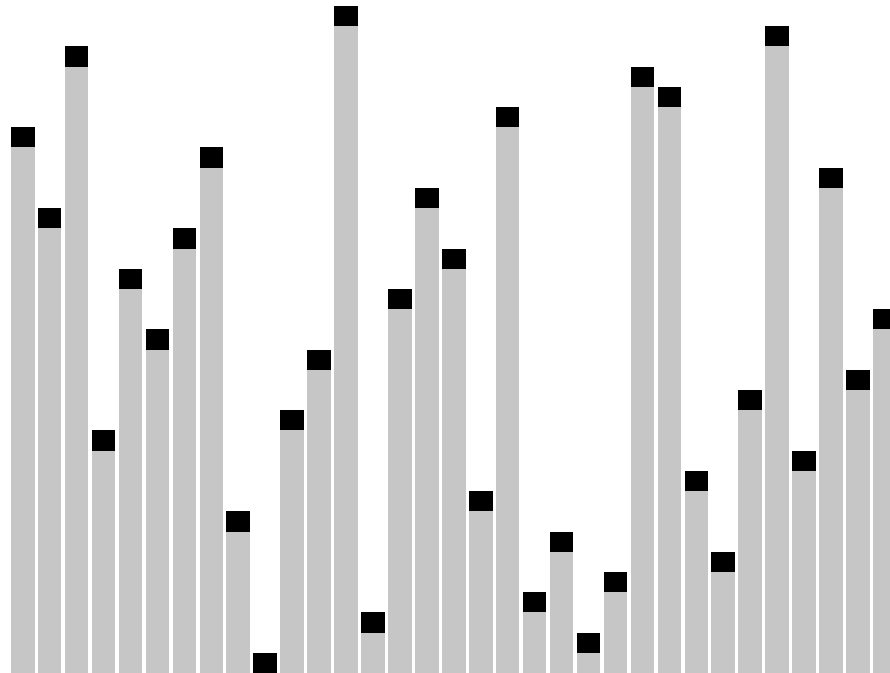
Сложность по времени

Худшее время: $O(n \log n)$

Среднее время: $O(n \log n)$

Лучшее время: $O(n \log n)$

Затраты памяти: $O(n)$ вспомогательной



(нажми два раза, чтобы оживить картинку)

```
void mergeSort(int data[], int lenD)
{
    if (lenD > 1) {
        int middle = lenD / 2;
        int rem = lenD - middle;
        int* L = new int[middle];
        int* R = new int[rem];
        for (int i = 0; i < lenD; i++) {
            if (i < middle) {
                L[i] = data[i];
            }
        }
    }
}
```

```

    }
    else {
        R[i - middle] = data[i];
    }
}
mergeSort(L, middle);
mergeSort(R, rem);
merge(data, lenD, L, middle, R, rem);
}
}
void merge(int merged[], int lenD, int L[], int lenL, int R[], int lenR) {
    int i = 0;
    int j = 0;
    while (i < lenL || j < lenR) {
        if (i < lenL & j < lenR) {
            if (L[i] <= R[j]) {
                merged[i + j] = L[i];
                i++;
            }
            else {
                merged[i + j] = R[j];
                j++;
            }
        }
        else if (i < lenL) {
            merged[i + j] = L[i];
            i++;
        }
        else if (j < lenR) {
            merged[i + j] = R[j];
            j++;
        }
    }
}
}

```

Быстрая сортировка (Quick sort)

Быстрая сортировка использует алгоритм "разделяй и властвуй". Она начинается с разбиения исходного массива на две области. Эти части находятся слева и справа от отмеченного элемента, называемого опорным. В конце процесса одна часть будет содержать элементы меньше, чем опорный, а другая часть будет содержать элементы больше опорного.

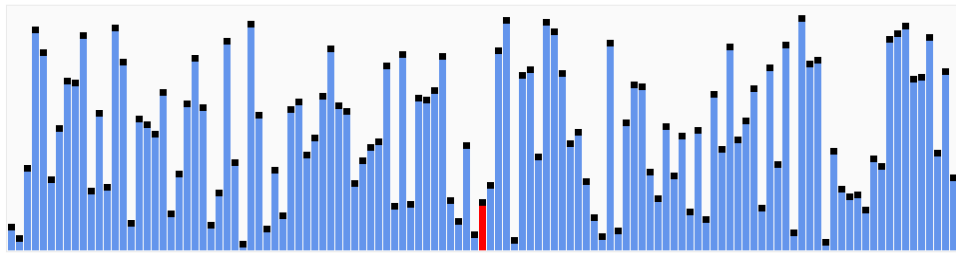
Сложность по времени

Худшее время: $O(n^2)$

Среднее время: $O(n \log n)$

Лучшее время: $O(n)$

Затраты памяти: $O(n)$



(нажми два раза, чтобы оживить картинку)

```
void QuickSort(int* a, int left, int right)
{
    int    l_hold = left, // левая граница
           r_hold = right, // правая граница
           pivot = a[left]; // разрешающий элемент
    while (left < right) // пока границы не сомкнутся
    {
        while ((a[right] >= pivot) && (left < right))
            right--; // сдвигаем правую границу пока элемент [right] больше [pivot]
        if (left != right) // если границы не сомкнулись
        {
            a[left] = a[right]; // перемещаем элемент [right] на место разрешающего
            left++; // сдвигаем левую границу вправо
        }
        while ((a[left] <= pivot) && (left < right))
            left++; // сдвигаем левую границу пока элемент [left] меньше [pivot]
        if (left != right) // если границы не сомкнулись
        {
            a[right] = a[left]; // перемещаем элемент [left] на место [right]
            right--; // сдвигаем правую границу вправо
        }
    }
    a[left] = pivot; // ставим разрешающий элемент на место
    pivot = left;
    left = l_hold;
    right = r_hold;
    if (left < pivot) // Рекурсивно вызываем сортировку для левой и правой части массива
        QuickSort(a, left, pivot - 1);
    if (right > pivot)
        QuickSort(a, pivot + 1, right);
}
```

Задания к лабораторной работе:

Реализовать функцию с выбором сортировки, которая будет использоваться для сортировки вашего массива структур по каким-либо параметрам(т.е. сортировать, например, по цене), выбор будет происходить среди следующих алгоритмов: *пузырьковая сортировка, быстрая сортировка, сортировка выбором, сортировка вставками, сортировка слиянием.*

Дополнительные задания(использовать структуры и массивы структур для упорядочивания данных):

1. Создайте файл с именем "data.txt" и запишите в него несколько строк текста. Откройте файл в режиме чтения и записи, считайте данные в память, отсортируйте их в алфавитном порядке и сохраните отсортированные данные в файле. (3 балла)
2. Создайте файл с именем "employees.txt" и запишите в него данные о сотрудниках, включая их имена и зарплаты. Откройте файл в режиме чтения и записи, считайте данные в память, отсортируйте их по зарплатам и сохраните отсортированные данные в файле. (3 балла)
3. Реализуйте алгоритм быстрой сортировки для сортировки записей в файле(содержимое файла выберите сами). Сохраните отсортированные данные в новом файле с именем "sorted_data.txt". (3 балла)

Вопросы к лабораторной работе:

1. Важно ли использовать сортировку данных?
2. Дайте определение алгоритмам сортировки?
3. Как оценить сложность алгоритмов?
4. Какая сортировка самая быстрая?
5. Какая сортировка является самой медленной? Почему?
6. Расскажите про принцип «разделяй и властвуй».
7. Расскажите про основные алгоритмы сортировки.
8. Какой принцип действия сортировки выбором/слиянием/вставками?
9. В чем заключается преимущество быстрой сортировки?
11. Какие виды сортировки вы знаете?

Лабораторная работа №8. Фильтрация данных.

Фильтрация данных - это процесс поиска и извлечения нужных данных из набора данных. Фильтрация может осуществляться на основе различных критериев, таких как значения полей, дата, время и т.д.

Для фильтрования данных в файле необходимо выполнить следующие шаги:

1. Открыть файл с исходными данными для чтения.
2. Прочитать данные из файла и сохранить их в переменной.
3. Применить фильтр на основе заданных критериев.

4. Записать отфильтрованные данные в новый файл.

Далее мы разберём на примерах, как может выглядеть алгоритм поиска подстроки в строке. Примеры будут основываться на функциях стандартных библиотек, ведь именно в таких функциях и проявляются все удобства написания программ. А вот классический разбор алгоритма, основанный на циклах и сравнениях, также достаточно примечателен. Поэтому мы его рассмотрим в этом же уроке.

Сам алгоритм в принципе очень прост. Есть две строки. Например "Hello world" и "lo".

Работать будем в два цикла:

1. Первый будет выполнять проход по всей строке, и искать местоположение **первой буквы** искомой строки ("l").
2. Вторым, начиная с найденной позиции первой буквы – сверять, какие буквы стоят после неё и сколько из них подряд совпадают.

Проиллюстрируем поиск подстроки в строке:

1-й шаг цикла	Н	Е	Л	Л	О		W	O	R	L	D
	Л										
2-й шаг цикла	Н	Е	Л	Л	О		W	O	R	L	D
		Л									
3-й шаг цикла	Н	Е	Л	Л	О		W	O	R	L	D
			Л	О							
4-й шаг цикла	Н	Е	Л	Л	О		W	O	R	L	D
				Л	О						

На первых двух итерациях цикла сравниваемые буквы не будут совпадать (выделено красным). На третьей итерации искомая буква (первый символ

искомого слова) совпала с символом в строке, где происходит поиск. При таком совпадении в работу включается второй цикл.

Он призван отсчитывать количество символов после первого в искомой строке, которые будут совпадать с символами в строке исходной. Если один из следующих символов не совпадает – цикл завершает свою работу. Нет смысла гонять цикл впустую, после первого несовпадения, так как уже понятно, что искомого тут нет.

На третьей итерации совпал только первый символ искомой строки, а вот второй уже не совпадает. Придется первому циклу продолжить работу. Четвертая итерация дает необходимые результаты – совпадают все символы искомой строки с частью исходной строки. А раз все символы совпали – подстрока найдена. Работу алгоритма можно закончить.

Сколько раз искомое слово встречается в строке и в каких местах? Именно это и призван контролировать третий параметр – `int n` – номер вхождения в строку. Если поставить туда единицу – он найдет первое совпадение искомого. Если двойку, он заставит первый цикл пропустить найденное первое, и искать второе. Если тройку – искать третье и так далее. С каждым найденным искомым словом, этот счетчик вхождений уменьшается на единицу.

Посмотрим, как выглядит классический код поиска подстроки в строке в C++:

```
// Функция для поиска подстроки в строке
// + поиск позиции, с которой начинается подстрока
int pos(const char* s, const char* c, int n)
{
    int i, j;           // Счетчики для циклов
    int lenC, lenS;     // Длины строк
    //Находим размеры строки исходника и искомого
    for (lenC = 0; c[lenC]; lenC++);
    for (lenS = 0; s[lenS]; lenS++);
    for (i = 0; i <= lenS - lenC; i++) // Пока есть возможность поиска
    {
        for (j = 0; s[i + j] == c[j]; j++); // Проверяем совпадение посимвольно
        // Если посимвольно совпадает по длине искомого
        // Вернем из функции номер ячейки, откуда начинается совпадение
        // Учитывать 0-терминатор ( '\0' )
        if (j == lenC && !(n - 1))
            return i;
        if (j == lenC)
            if (n - 1)
                n--;
            else
                return -1;
    }
}
```



```

        return i;
    }
    //Иначе вернем -1 как результат отсутствия подстроки
    return -1;
}
int main()
{
    const char* s = "парарапа";
    const char* c = "ра";
    ....
}
0
4

```

Два цикла выполняют каждый свою задачу. Один топает по строке в надежде найти “голову” искомого слова (первый символ). Второй выясняет, есть ли после найденной “головы” “тело” искомого. Причем проверяет, не лежит ли это “тело” в конце строки. Т.е. не является ли длина найденного слова на единицу больше длины искомой строки, если учитывать, что в эту единицу попадает нуль-терминатор (`'\0'`).

Мы видим, что программа нашла начало подстроки **ра** в ячейках символьного массива с индексом 0 и 4. Но почему? Ведь в слове **парарапа** 3 таких подстроки. Все дело в `'\0'` .

В целом смысл самого алгоритма на этом заканчивается. Больше никаких сложностей кроме нуля в конце строки нет.

Алгоритм Кнута-Морриса-Пратта

Кнута-Морриса-Пратта (КМП) - это алгоритм поиска подстроки в строке, который использует метод сравнения шаблона с текстом, но в отличие от простого метода сравнения, КМП использует информацию о предыдущих сравнениях для ускорения процесса.

Алгоритм КМП основан на использовании префикс-функции, которая вычисляется для шаблона. **Префикс-функция для шаблона** - это массив значений, где $pi[i]$ - это максимальная длина суффикса шаблона $p[0...i]$, который одновременно является его префиксом. Массив pi можно вычислить за линейное время $O(m)$, где m - длина шаблона.

Затем, используя префикс-функцию, КМП алгоритм сравнивает шаблон с текстом, начиная с начала строки. Если текущий символ текста не

соответствует текущему символу шаблона, то алгоритм сдвигает шаблон на количество символов, равное разности текущей позиции и значения префикс-функции для предыдущей позиции. Это позволяет избежать повторных сравнений символов, которые уже были проверены и не соответствуют искомой подстроке.

Алгоритм продолжает сравнивать символы шаблона с текстом, пока не будет найдено первое вхождение подстроки в строку или пока не достигнут конец текста.

	$i \rightarrow i \rightarrow i \rightarrow i \rightarrow i \rightarrow i \rightarrow i \rightarrow i$													
	↓	↓	↓	↓	↓	↓	↓	↓						
Строка	A	B	C	A	B	C	A	A	B	C	A	B	D	
Подстрока	A	B	C	A	B	D								
		A	B	C	A	B	D							
			A	B	C	A	B	D						
				A	B	C	A	B	D					
					A	B	C	A	B	D				
						A	B	C	A	B	D			
							A	B	C	A	B	D		
								A	B	C	A	B	D	

Асимптотическая сложность алгоритма Кнута-Морриса-Пратта - $O(n+m)$, где n - длина текста, а m - длина шаблона. Он обладает лучшей производительностью, чем метод посимвольного сравнения, особенно для длинных строк и подстрок.

Для более подробного и углубленного изучения формулы и нюансов алгоритма КМП: [MAXimal :: algo :: Префикс-функция. Алгоритм Кнута-Морриса-Пратта \(e-maxx.ru\)](http://e-maxx.ru/algo/Prefix-function)

Пример поиска подстроки в строке с использованием алгоритма Кнута-Морриса-Пратта:

```
// Функция вычисления префикс-функции для шаблона
vector<int> compute_prefix_function(const string& pattern) {
    int m = pattern.length();
    vector<int> pi(m); // массив для хранения значений префикс-функции
```

```

pi[0] = 0; // префикс-функция для первого символа всегда равна 0
int k = 0; // длина максимального суффикса, совпадающего с префиксом
for (int i = 1; i < m; i++) {
    while (k > 0 && pattern[k] != pattern[i]) {
        k = pi[k - 1]; // корректируем значение k, используя префикс-функцию для предыдущего
        символа
    }
    if (pattern[k] == pattern[i]) {
        k++; // увеличиваем значение k, если символы совпадают
    }
    pi[i] = k; // сохраняем значение префикс-функции для i-го символа шаблона
}
return pi; // возвращаем массив значений префикс-функции
}

// Функция поиска подстроки в тексте с использованием алгоритма КМП
int kmp_search(const string& pattern, const string& text) {
    int m = pattern.length(); // длина шаблона
    int n = text.length(); // длина текста
    vector<int> pi = compute_prefix_function(pattern); // вычисляем префикс-функцию для шаблона
    int k = 0; // длина максимального суффикса, совпадающего с префиксом шаблона
    for (int i = 0; i < n; i++) {
        while (k > 0 && pattern[k] != text[i]) {
            k = pi[k - 1]; // корректируем значение k, используя префикс-функцию для предыдущего
            символа шаблона
        }
        if (pattern[k] == text[i]) {
            k++; // увеличиваем значение k, если символы совпадают
        }
        if (k == m) { // если k равно длине шаблона, то мы нашли подстроку
            return i - m + 1; // возвращаем позицию, с которой начинается подстрока
        }
    }
    return -1;
}

int main() {
    setlocale(LC_ALL, "rus");
    string pattern = "abc"; // задаем шаблон
    string text = "abababc"; // задаем текст
    ....
}

```

Подстрока найдена в позиции 4

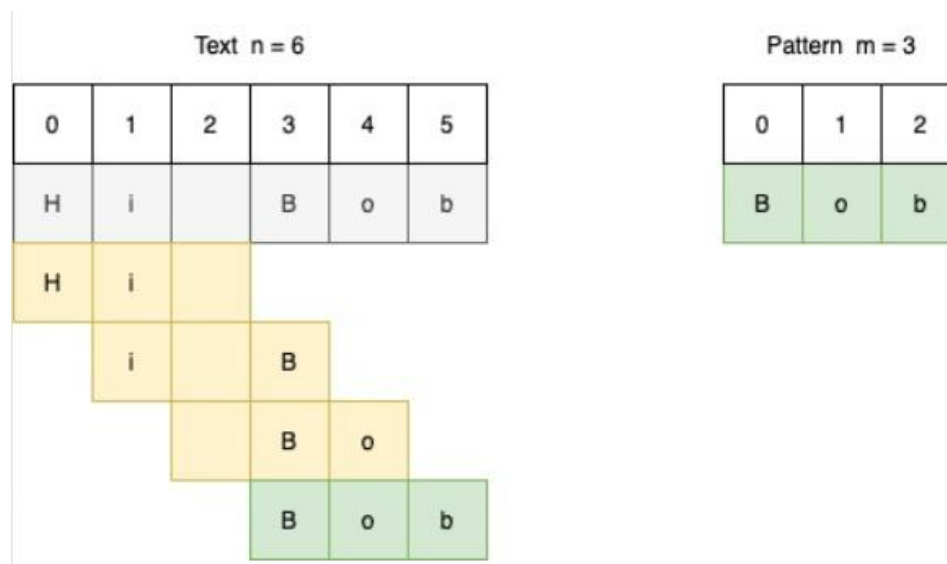
Алгоритм Рабина-Карпа

Алгоритм Рабина-Карпа (RK) - это алгоритм поиска подстроки в строке, который основан на хэшировании строк. Для каждого окна в строке, которое имеет длину подстроки, вычисляется хэш-значение. Затем сравнивается хэш-

значение окна со значением хэша подстроки. Если они совпадают, то выполняется дополнительная проверка на совпадение каждого символа.

Наивный алгоритм поиска строки сравнивает заданный шаблон со всеми позициями в тексте. Это приводит к далеко не идеальной сложности времени исполнения $O(nm)$, где n = длина текста, а m = длина шаблона.

Алгоритм Рабина-Карпа совершенствует этот подход за счёт того, что сравнение хешей двух строк выполняется за линейное время: для поиска совпадения это гораздо эффективнее, чем сравнение отдельных символов этих строк. Таким образом, алгоритм показывает лучшее время исполнения $O(n+m)$.



С применением хеш-функции связаны два нюанса.

Во-первых, алгоритм хорош настолько, насколько хороша его хеш-функция. Если при использовании хеш-функции имеют место многочисленные ложные срабатывания, то сравнение символов будет выполняться слишком часто. В этом случае очень сложно считать этот метод более эффективным, чем наивный алгоритм.

Во-вторых, каждый раз, когда подстрока проходит по тексту, вычисляется новый хеш, что крайне неэффективно, ведь в этом случае производительность такая же, если не хуже, как у наивного алгоритма.

Обе эти проблемы решаются с помощью полиномиального хеша с операциями сложения и умножения. И хотя это не какой-то эксклюзив алгоритма Рабина-Карпа, которого нет в других алгоритмах, здесь он работает так же хорошо.

Алгоритм Рабина-Карпа работает следующим образом:

1. Вычисляем хэш-значение для подстроки, которую мы ищем.
2. Вычисляем хэш-значения для всех подстрок строки, которые имеют ту же длину, что и искомая подстрока.
3. Сравниваем хэш-значения подстрок. Если хэш-значения совпадают, то выполняем дополнительную проверку на совпадение каждого символа.

Для вычисления хэш-значения используется функция хэширования, которая преобразует строку в число. Обычно используется хэш-функция, которая основана на арифметических операциях над ASCII-кодами символов строки. Однако, хэш-функция должна быть такой, чтобы вероятность коллизий (ситуация, когда две разные строки имеют одно и то же хэш-значение) была минимальной.

Стоит отметить, что при использовании алгоритма Рабина-Карпа возможны ложные срабатывания, когда хэш-значения двух разных строк совпадают. Для уменьшения вероятности ложных срабатываний можно использовать несколько хэш-функций или дополнительно проверять совпадение каждого символа при совпадении хэш-значений.

Алгоритм Рабина-Карпа может быть эффективен при поиске нескольких подстрок в одной строке, так как при этом можно повторно использовать уже вычисленные хэш-значения.

Пример программы на C++, которая ищет подстроку в строке с помощью алгоритма Рабина-Карпа:

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

// Хэш-функция, которая суммирует ASCII-коды символов
int hash_function(string str) {
    int hash_value = 0;
    for (int i = 0; i < str.length(); i++) {
        hash_value += (int)str[i];
    }
}
```

```

    return hash_value;
}
// Функция для поиска подстроки в строке с помощью алгоритма Рабина-Карпа
vector<int> rabinkarp(string str, string substr) {
    vector<int> matches; // массив для хранения индексов найденных подстрок
    int substr_hash = hash_function(substr); // хэш-значение искомой подстроки
    for (int i = 0; i <= str.length() - substr.length(); i++) {
        string current_substr = str.substr(i, substr.length());
        int current_hash = hash_function(current_substr); // хэш-значение текущей подстроки
        if (current_hash == substr_hash && current_substr == substr) {
            matches.push_back(i); // добавляем индекс найденной подстроки в массив
        }
    }
    return matches;
}

int main() {
    setlocale(LC_ALL, "rus");
    string str = "Hello, world!";
    string substr = "wor";
    vector<int> matches = rabinkarp(str, substr);
    if (matches.size() == 0) {
        cout << "Подстрока не найдена" << endl;
    }
    else {
        cout << "Подстрока найдена в следующих позициях: ";
        for (int i = 0; i < matches.size(); i++) {
            cout << matches[i] << " ";
        }
        cout << endl;
    }
    return 0;
}

```

```
Подстрока найдена в следующих позициях: 7
```

Однако в виду возможных неточностей при поиске выполнение поиска при помощи этого алгоритма в лабораторной работе мы не требуем.

Более подробная информация про алгоритм Рабина-Карпа: [Алгоритм Рабина-Карпа с полиномиальным хешем и модульной арифметикой | by Андрей Шагин | NOP::Nuances of Programming | Medium](#)

Использование `string::find()`

Класс `string` в C++ снабжен методом `find()`, возвращающий номер ячейки, с которого начинается тело искомой строки в исходной строке.

Функция `find()` возвращает индекс первого вхождения подстроки или отдельного символа в строке:

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

int main()
{
    std::string text{ "A friend in need is a friend indeed." };
    std::cout << text.find("ed") << std::endl;    // 14
    std::cout << text.find("friend") << std::endl; // 2
    std::cout << text.find('d') << std::endl;     // 7
    std::cout << text.find("apple") << std::endl; // 18446744073709551615
}
```

Если строка или символ не найдены (как в примере выше в последнем случае), то возвращается специальная константа `std::string::npos`, которая представляет очень большое число. И при поиске мы можем проверять результат функции `find()` на равенство этой константе.


Функция `find()` принимает вторым параметром номер символа, с которого начать поиск. Т.е. найдя первое вхождение, его значение увеличивается на единицу и `find()` продолжает поиск со следующего символа.

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

using namespace std;

int main()
{
    string s = "parapapa";
    int i = 0;

    for (i = s.find("pa", i++); i != string::npos; i = s.find("pa", i + 1))
        cout << i << endl;
}
```



В качестве третьего параметра передается количество символов из искомой строки, которые программа будет искать в тексте:

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

```
using namespace std;

int main()
{
    string text{ "A friend in need is a friend indeed." };
    string word{ "endless" };
    // поиск с 10-го индекса 3 первых символов слова "endless", то есть "end"
    cout << text.find("endless", 10, 3) << endl;
}

25
```

Функция **rfind()** работает аналогично функции **find()**, принимает те же самые параметры, только ищет подстроку в обратном порядке - с конца строки:

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

int main()
{
    std::string text{ "A friend in need is a friend indeed." };
    std::cout << text.rfind("ed") << std::endl;    // 33
    std::cout << text.rfind("friend") << std::endl; // 22
    std::cout << text.rfind('d') << std::endl;    // 34
    std::cout << text.rfind("apple") << std::endl; // 18446744073709551615
}

33
22
34
18446744073709551615
```

Пара функций - *find_first_of()* и *find_last_of()* позволяют найти соответственно первый и последний индекс любого из набора символов.

Если нам надо найти позиции символов, которые НЕ представляют собой любой символ из набора, то мы можем использовать функции *find_first_not_of()* (первая позиция) и *find_last_not_of()* (последняя позиция).

Задание к лабораторной работе:

Для вашей БД из лабораторной работы № 2 реализовать 3 вида поиска: стандартный алгоритм поиска подстроки (в лекции рассматривался как первый

алгоритм), поиск подстроки с помощью алгоритма Кнута-Морриса-Пратта, поиск подстроки с использованием `string::find()`. Поиск производить по одному типу данных, например, по цене.

Дополнительные задания(все строки считывать из файлов):

1. Напишите программу, которая принимает на вход две строки и находит все вхождения первой строки во вторую строку. Выведите на экран по-зиции начала каждого вхождения.(3 балла)
2. Измените свою программу для поиска подстроки в строке так, чтобы она искала только первое вхождение. Если подстрока не найдена, про-грамма должна выводить сообщение об этом.(3 балла)
3. Напишите программу, которая находит наиболее длинную общую под-строку двух заданных строк. Выведите найденную подстроку на экран.(3 балла)
4. Измените свою программу для поиска наиболее длинной общей под-строки так, чтобы она работала с произвольным количеством строк. Выведите найденную подстроку на экран.(3 балла)

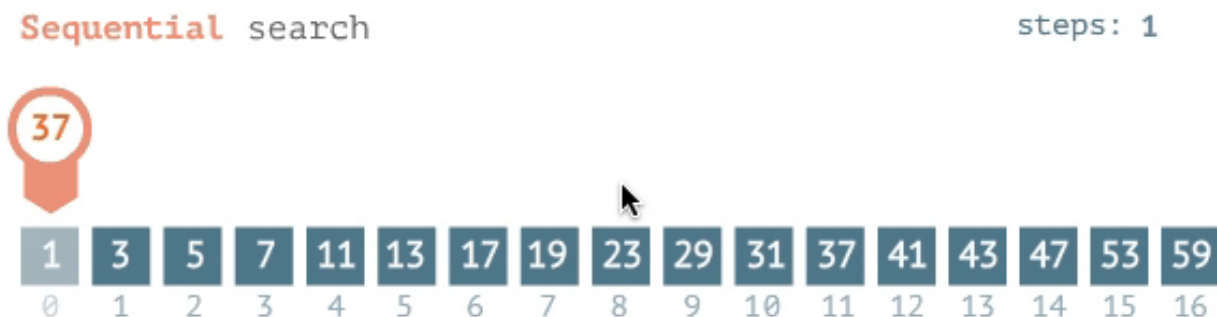
Вопросы к лабораторной работе:

1. В чем суть алгоритма Кнута-Морриса-Пратта?
2. Что такое префикс-функция для шаблона?
3. Для чего нужен метод `find()`?
4. В чем `rfind()` отличается от `find()`?
5. Что выполняют функции `find_first_of()` и `find_last_of()`?
6. В чем преимущества алгоритма Кнута-Морриса-Пратта?
7. Как реализовывается поиск подстроки в строке?
8. Что представляет специальная константа `std::string::npos`?
9. Чем простой метод сравнения отличается от КМП?
10. Что позволяет избежать повторных сравнений символов, которые уже были проверены и не соответствуют искомой подстроке?
11. Назовите особенности алгоритма Рабина-Карпа.
12. В чём заключается возможная неточность алгоритма Рабина-Карпа.

Лабораторная работа №9. Поиск записи по значению/индексу.

100% программистов, во время обучения, рано или поздно столкнутся с необходимостью проверить наличие в массиве определённого значения. Существует несколько общеизвестных алгоритмов поиска в языках программирования.

Сейчас мы рассмотрим самый простой из них (но далеко не самый эффективный) – линейный поиск либо последовательный поиск. **Линейный поиск** - это простой алгоритм поиска элемента в массиве или списке, который заключается в том, чтобы последовательно перебирать все элементы и сравнивать их с целевым элементом до тех пор, пока не будет найден элемент или не будут пройдены все элементы.



(нажми два раза, чтобы оживить картинку)

Основные преимущества линейного поиска:

1. Простота реализации: линейный поиск является одним из самых простых алгоритмов поиска и может быть легко реализован на любом языке программирования без использования дополнительных структур данных или библиотек.
2. Универсальность: линейный поиск может быть применен к любому типу данных и любой форме контейнера, включая неупорядоченные массивы, списки, файлы и т.д.
3. Надежность: линейный поиск всегда найдет целевой элемент, если он присутствует в контейнере.

Основные недостатки линейного поиска:

1. Низкая эффективность: в худшем случае, когда целевой элемент находится в конце контейнера или отсутствует в нем, линейный поиск должен пройти через все элементы, что может занять значительное время для больших контейнеров.
2. Неэффективность при частом поиске: если требуется выполнить множество поисковых операций в большом контейнере, линейный поиск будет неэффективен, поскольку каждый поиск потребует просмотра всех элементов.
3. Непригодность для упорядоченных контейнеров: если контейнер упорядочен, можно использовать более эффективные алгоритмы поиска, такие как двоичный поиск, которые могут быть значительно быстрее линейного поиска.

Обычно линейный поиск применяется для одиночного поиска в небольшом массиве, который не отсортирован. В других случаях, лучше и эффективней сначала отсортировать массив и применять другие алгоритмы поиска. Например двоичный (бинарный) поиск либо другие.

```
....  
  
int linearSearch(int arr[], int n, int x) {  
    for (int i = 0; i < n; i++) {  
        if (arr[i] == x) {  
            return i;  
        }  
    }  
    return -1;  
}  
  
int main() {  
    setlocale(LC_ALL, "rus");  
    int arr[] = { 45, 12, 78, 23, 56, 89 };  
    ....  
}
```

Элемент найден в позиции 3

В этой программе определена функция `linearSearch`, которая принимает массив `arr`, его размер `n` и целевой элемент `x`. Функция последовательно перебирает элементы массива и сравнивает их с целевым элементом. Если элемент найден, функция возвращает его позицию в массиве; если элемент не найден, функция возвращает `-1`.

Предположим, вы ищете фамилию человека в телефонной книге. Она начинается с буквы «К». Конечно, можно начать с самого начала и перелистывать страницы, пока вы не доберетесь до буквы «К». Но скорее всего для ускорения поиска лучше раскрыть книгу на середине: ведь буква «К» должна находиться где-то ближе к середине телефонной книги. Теперь допустим, что вы вводите свои данные при входе на Facebook. При этом Facebook необходимо проверить, есть ли у вас учетная запись на сайте. Для этого ваше имя пользователя нужно найти в базе данных. Допустим, вы выбрали себе имя пользователя “karlnageddon”. Facebook может начать с буквы А и проверять все подряд, но разумнее будет начать с середины.

Перед нами типичная задача поиска. И во всех этих случаях для решения задачи можно применить один алгоритм: бинарный поиск.

Бинарный поиск - это алгоритм поиска элемента в упорядоченном массиве или списке, который заключается в последовательном делении массива на половины и проверке, находится ли целевой элемент в левой или правой половине. Алгоритм повторяется, пока не будет найден целевой элемент, или пока не будет проверено, что его в массиве нет.

Binary search

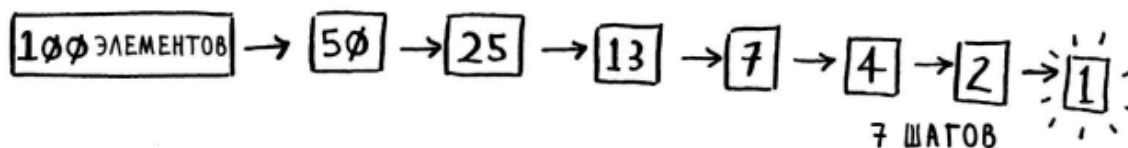
steps: 0



(нажми два раза, чтобы оживить картинку)

Рассмотрим пример того, как работает бинарный поиск. Я загадал число от 1 до 100. Вы должны отгадать мое число, используя как можно меньше попыток. Предположим, вы начинаете перебирать все варианты подряд: 1, 2, 3, 4... Это плохой способ угадать число. Это пример простого поиска. При каждой догадке исключается только одно число. Если я загадал число 99, то, чтобы добраться до него, потребуется 99 попыток!

Существует другой, более эффективный способ. Начнем с 50. Все эти числа слишком малы, но вы только что исключили половину чисел! Теперь вы знаете, что все числа 1-50 меньше загаданного. Следующая попытка: 75. На этот раз перелет... Но вы снова исключили половину оставшихся чисел. С бинарным поиском вы каждый раз загадываете число в середине диапазона и исключаете половину оставшихся чисел. Следующим будет число 63 (по середине между 50 и 75). Так работает бинарный поиск.



При бинарном поиске каждый раз исключается половина чисел

Сколько времени экономит применение бинарного поиска? Если список состоит из 100 чисел, может потребоваться до 100 попыток, используя линейный поиск. Для списка из 4 миллиардов чисел потребуется до 4 миллиардов попыток. Таким образом, максимальное количество попыток совпадает с размером списка. С бинарным поиском дело обстоит иначе. Если список состоит из 100 элементов, потребуется не более 7 попыток. Для списка из 4 миллиардов элементов потребуется не более 32 попыток. Впечатляет, верно? Бинарный поиск выполняется за логарифмическое время.



Несмотря на то, что бинарный поиск является очень эффективным алгоритмом поиска элемента в упорядоченном массиве, у него **есть некоторые недостатки**:

1. Бинарный поиск требует, чтобы массив был упорядочен. Если массив не упорядочен, то перед началом поиска потребуется произвести сортировку, что может занять значительное время.
2. Бинарный поиск не работает с динамическими структурами данных, такими как связанные списки, потому что доступ к элементам в них не может быть осуществлен быстро.
3. Бинарный поиск занимает больше памяти, чем линейный поиск, потому что требуется дополнительная переменная для хранения среднего индекса.
4. Если массив содержит повторяющиеся элементы, то бинарный поиск может вернуть любой из них, а не обязательно первый или последний. Это может быть нежелательным в некоторых случаях.
5. В худшем случае, сложность бинарного поиска может достигать $O(n)$, когда все элементы массива одинаковы.
6. Бинарный поиск неэффективен для поиска элемента в небольших массивах, так как в таком случае линейный поиск может быть быстрее.

Основные шаги бинарного поиска:

1. Определить границы поиска в массиве, обычно это начало и конец массива.

2. Вычислить средний индекс, как сумму границ, деленную на 2. Этот индекс будет использоваться для разделения массива на две части: левую и правую.
3. Сравнить целевой элемент со средним элементом в массиве. Если целевой элемент меньше, чем средний элемент, то дальнейший поиск будет осуществляться только в левой половине массива, иначе - в правой.
4. Если целевой элемент найден, то поиск завершается, иначе алгоритм повторяется для соответствующей половины массива.
5. Если границы поиска сомкнулись, то элемент не найден в массиве.

Пример:

```
// функция с алгоритмом двоичного поиска
int Search_Binary(int arr[], int left, int right, int key)
{
    int midd = 0;
    while (1)
    {
        midd = (left + right) / 2;

        if (key < arr[midd])    // если искомое меньше значе-ния в ячейке
            right = midd - 1;  // смещаем правую границу по-иска
        else if (key > arr[midd]) // если искомое больше значе-ния в ячейке
            left = midd + 1;    // смещаем левую границу поиска
        else                  // иначе (значения равны)
            return midd;        // функция возвращает индекс ячейки
        if (left > right)       // если границы сомкнулись
            return -1;
    }
}

int main()
{
    setlocale(LC_ALL, "rus");
    ....
}
```

```
1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
```

Введите любое число: 7

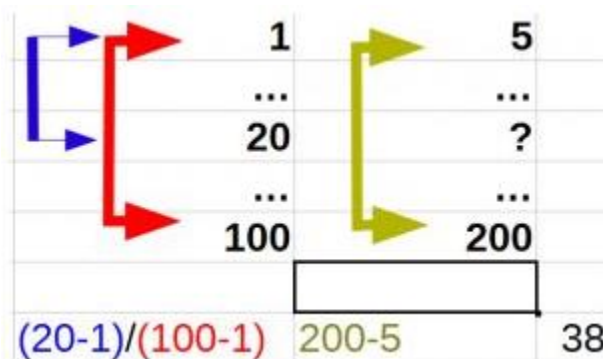
Указанное число находится в ячейке с индексом: 6

Интерполяционный поиск - это алгоритм поиска элемента в упорядоченном массиве, который использует формулу интерполяции для оценки приблизительного местоположения целевого элемента в массиве и выбирает соответствующий интервал для дальнейшего поиска.

Простыми словами, интерполяционный поиск работает так:

1. Определить границы поиска в массиве, обычно это начало и конец массива.
2. Оценить приблизительное местоположение целевого элемента в массиве с помощью формулы интерполяции. Формула может быть разной в зависимости от конкретной реализации, но обычно она использует значения начального и конечного элементов массива, а также значение целевого элемента.
3. Сравнить целевой элемент со значением, найденным по формуле интерполяции. Если целевой элемент меньше, чем значение из формулы, то дальнейший поиск производится в левой части массива, иначе - в правой части массива.
4. Повторять шаги 2-3 до тех пор, пока целевой элемент не будет найден или пока не будет установлено, что он отсутствует в массиве.

Интерполяционный поиск может быть эффективным, если элементы в массиве распределены равномерно и значения между ними линейно увеличиваются или уменьшаются. Однако, если распределение элементов не является равномерным, то интерполяционный поиск может работать медленнее, чем бинарный поиск.



Примерно так это будет выглядеть в виде картинки

Формула интерполяции достаточно проста — вычисляется длина между номерами первого элемента и искомого (задаваемого точнее). Такая же длина

считается между первым и последним номерами. Длины между собой делятся, как раз и получая вычисление подобия. То же самое происходит со значениями элементов – так же вычисляется расстояние между граничными значениями в массиве.

Линейная интерполяция может быть выражена следующей формулой:

$$\text{position} = \text{low} + ((\text{high} - \text{low}) / (\text{arr}[\text{high}] - \text{arr}[\text{low}])) * (\text{x} - \text{arr}[\text{low}]),$$

где: low - индекс начального элемента интервала поиска; high - индекс конечного элемента интервала поиска; arr - упорядоченный массив; x - значение целевого элемента, которое мы ищем в массиве; position - приблизительное местоположение целевого элемента в массиве.

Пример:

```
int main()
{
    //Массив значений в котором пойдет поиск
    int MyArray[] { 1, 2, 4, 6, 7, 89, 123, 231, 1000, 1235 };
    int x = 0; //Текущая позиция массива, с которым сравнивается искомое
    int a = 0; //Левая граница области, где ведется поиск
    int b = 9; //Правая граница области, где ведется поиск
    int WhatFind = 123; //Значение, которое нужно найти
    bool found; //Переменная-флаг, принимающая True если искомое найдено

    /***** Начало интерполяции *****/
    //Цикл поиска по массиву, пока искомое не найдено
    //или пределы поиска еще существуют
    for (found = false; (MyArray[a] < WhatFind) && (MyArray[b] > WhatFind) && !found; )
    {
        //Вычисление интерполяцией следующего элемента, который будет сравниваться с
        //искомым
        x = a + ((WhatFind - MyArray[a]) * (b - a)) / (MyArray[b] - MyArray[a]);
        //Получение новых границ области, если искомое не найдено
        if (MyArray[x] < WhatFind)
            a = x + 1;
        else if (MyArray[x] > WhatFind)
            b = x - 1;
        else
            found = true;
    }

    /***** Конец интерполяции *****/

    //Если искомое найдено на границах области поиска, показать на какой границе оно
    if (MyArray[a] == WhatFind)
```

```

        cout << WhatFind << " founded in element " << a << endl;
    else if (MyArray[b] == WhatFind)
        cout << WhatFind << " founded in element " << b << endl;
    else
        cout << "Sorry. Not found" << endl;

    return 0;
}

```

```
123 founded in element 6
```

Таким образом сам цикл просто вычисляет по формуле область массива, где может находиться искомое используя этот самый принцип интерполяции в C++, подбирая подобия так сказать. Если вычисленное не равно искомому, значит нужно сдвинуть границы области, где проходит вычисление. Если вычисленное больше – сдвигается правая граница области поиска, если меньше – левая. Так отрезая (как в бинарном поиске) кусок массива за куском постепенно достигается нужная ячейка массива, ну или границы области поиска сужаются до таких величин, в пределах которого уже искать нечего, когда дистанция между границами равна 1 (т.е. между точкой А и В нет более элементов для вычисления) решение говорит о том, что значение в массиве не найдено.

Задание к лабораторной работе:

В соответствии с содержанием своей БД (из лабораторной работы № 2) реализовать алгоритмы: линейного поиска, бинарного поиска, интерполяционного поиска. Поиск выполнять по любому типу данных, например, по цене.

Дополнительные задания(массивы структур считывать из файла):

1. Реализуйте алгоритм линейного поиска для поиска записи в массиве структур, содержащих информацию о студентах (имя, фамилия, возраст, средний балл). (3 балла)
2. Реализуйте алгоритм двоичного поиска для поиска записи в отсортированном массиве структур, содержащих информацию о телефонах (модель, производитель, цена). (3 балла)

3. Реализуйте интерполяционный поиск для поиска записи в массиве структур, содержащей информацию о товарах (название, цена, производитель, описание). (3 балла)

Вопросы к лабораторной работе:

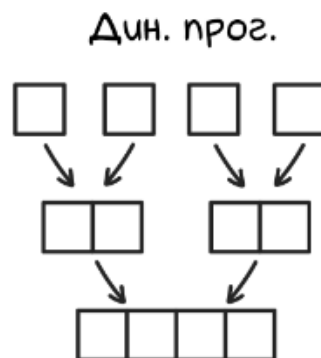
1. Дайте определение линейному поиску
2. Какие основные преимущества линейного поиска вам известны?
3. Назовите недостатки линейного поиска
4. Что такое бинарный поиск?
5. Перечислите недостатки бинарного поиска?
6. Перечислите основные шаги бинарного поиска?
7. Дайте определение интерполяционному поиску
8. Как работает интерполяционный поиск?
9. Назовите формулу интерполяции
10. Дайте сравнительную характеристику линейному, бинарному и интерполяционному поискам

Лабораторная работа №10. Динамическое программирование: поиск наибольшей общей подпоследовательности, последовательность Фибоначчи, задача о рюкзаке, задача о коммивояжёре.

Концепция динамического программирования заключается в разбиении сложной задачи на более простые подзадачи и сохранении результатов этих подзадач для последующего использования. Это позволяет избежать повторных вычислений и ускорить процесс решения задачи.

Во многих задачах по программированию решение с помощью рекурсии или полного перебора требует выполнения очень большого числа операций. Попытка решить такие задачи, например, полным перебором, приводит к превышению времени выполнения.

Однако среди переборных и некоторых других задач можно выделить класс задач, обладающих одним хорошим свойством: имея решения некоторых подзадач (например, для меньшего числа n), можно практически без перебора найти решение исходной задачи.



Такие задачи решают методом динамического программирования, а под самим динамическим программированием понимают сведение задачи к подзадачам.

Последовательности

Последовательность представляет собой упорядоченный набор элементов. Строка — это частный случай последовательности, дальнейшие примеры будут для простоты рассматривать именно строки, но без изменений их можно использовать и для произвольного текста или чего-нибудь последовательного еще.

Классической задачей на последовательности является следующая.

Последовательность Фибоначчи F_n задается формулами: $F_1 = 1$, $F_2 = 1$,

$F_n = F_{n-1} + F_{n-2}$ при $n > 1$. Необходимо найти F_n по номеру n .

последовательность Фибоначчи выглядит так: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987 и так далее. Каждое число из ряда Фибоначчи, разделенное на последующее, имеет значение, стремящееся к уникальному показателю, которое составляет 1,618. Первые числа ряда Фибоначчи не дают настолько точное значение, однако по мере нарастания, соотношение постепенно выравнивается и становится все более точным.

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}	F_{17}	F_{18}	... F_n
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597	2584	

↓
(0+1 = 1)
↓
(1+1 = 2)
↓
(1+2 = 3)
↓
(2+3=5)

Как видно, каждое число в последовательности представляет собой сумму двух предыдущих чисел.
Таким образом, последовательность Фибоначчи может быть суммирована с использованием этой формулы:

$$F_n = (F_{n-1}) + (F_{n-2})$$

Один из способов решения, который может показаться логичным и эффективным, — решение с помощью рекурсии:

```
int F(int n) {  
    if (n < 2) return 1;  
    else return F(n - 1) + F(n - 2);  
}
```

Но как можно заметить, такая, казалось бы, простая программа уже при $n = 40$ работает заметно долго. Это связано с тем, что одни и те же промежуточные данные вычисляются по несколько раз — число операций нарастает с той же скоростью, с какой растут числа Фибоначчи — экспоненциально.

Для данной задачи применимо более простое решение:

```
F[0] = 1;  
F[1] = 1;  
for (i = 2; i < n; i++) F[i] = F[i - 1] + F[i - 2];
```

Такое решение можно назвать решением «с начала» — мы первым делом заполняем известные значения, затем находим первое неизвестное значение (F_3), потом следующее и т.д., пока не дойдем до нужного.

Именно такое решение и является классическим для динамического программирования: мы сначала решили все подзадачи (нашли все F_i для $i < n$), затем, зная решения подзадач, нашли ответ ($F_n = F_{n-1} + F_{n-2}$, F_{n-1} и F_{n-2} уже найдены).

Поиск наибольшей общей подпоследовательности (LCS)

Поиск наибольшей общей подпоследовательности (LCS) является классической задачей в информатике и используется во многих областях, таких как биоинформатика, обработка естественного языка, компьютерное зрение и другие.

Общей подпоследовательностью для x и y считаем такую последовательность z , которая является одновременно подпоследовательностью x и подпоследовательностью y .

Максимальная общая подпоследовательность — это общая подпоследовательность с максимальной длиной. В качестве примера, пусть $x = \text{HABRANABR}$, $y = \text{HARBOUR}$, в этом случае $\text{LCS}(x, y) = \text{HARBR}$. Можно уже переходить непосредственно к алгоритму вычисления LCS, но, хорошо бы понять, для чего нам может это может понадобиться.

Метод динамического программирования может быть использован для решения задачи поиска LCS. Идея заключается в том, чтобы создать матрицу размера $(m+1) \times (n+1)$, где m и n — длины двух строк, которые мы сравниваем. Каждый элемент матрицы (i, j) содержит длину LCS для подстроки первой строки длиной i и подстроки второй строки длиной j .

Заполнение матрицы происходит с помощью следующего алгоритма:

1. Инициализируем первую строку и первый столбец матрицы нулями.
2. Проходимся по каждому элементу матрицы (i, j) со строкой i от 1 до m и столбцом j от 1 до n .
3. Если символы в строках с индексами i и j совпадают, то LCS для этих подстрок будет на 1 больше, чем LCS для подстрок $i-1$ и $j-1$. Значение в ячейке (i, j) равно значению в ячейке $(i-1, j-1)$ плюс 1.
4. Если символы не совпадают, то LCS для этих подстрок будет максимальным из LCS для подстрок $i-1$ и j и LCS для подстрок i и $j-1$. Значение в ячейке (i, j) равно максимальному значению из ячеек $(i-1, j)$ и $(i, j-1)$.

После заполнения матрицы, LCS может быть восстановлена обратным проходом по матрице, начиная с ячейки (m, n). Если значение в ячейке (i, j) равно значению в ячейке (i-1, j-1) плюс 1, то добавляем символ первой строки с индексом i в LCS и переходим к ячейке (i-1, j-1). Если значение в ячейке (i, j) равно значению в ячейке (i-1, j), то переходим к ячейке (i-1, j). Если значение в ячейке (i, j) равно значению в ячейке (i, j-1), то переходим к ячейке (i, j-1).

X = <A, B, C, B, D, A, B>;
Y = <B, D, C, A, B, A>

		B	D	C	A	B	A
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2	3	4	4

Z = <B, C, B, A>

Временная сложность алгоритма поиска LCS с использованием динамического программирования составляет $O(m*n)$, где m и n - длины двух строк.

Пример задачи и её решения:

Даны две последовательности A и B, нужно найти их наибольшую общую подпоследовательность.

```
int lcs(string A, string B, int m, int n) {
    int** L = new int* [m + 1];
    for (int i = 0; i <= m; i++) {
        L[i] = new int[n + 1];
        memset(L[i], 0, (n + 1) * sizeof(int));
    }
    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
```

```

        if (A[i - 1] == B[j - 1])
            L[i][j] = L[i - 1][j - 1] + 1;
        else
            L[i][j] = max(L[i - 1][j], L[i][j - 1]);
    }
}
int result = L[m][n];
for (int i = 0; i <= m; i++) {
    delete[] L[i];
}
delete[] L;
return result;
}
int main() {
    setlocale(LC_ALL, "rus");
    string A = "ABCDGH";
    string B = "AEDFHR";
    ....
}

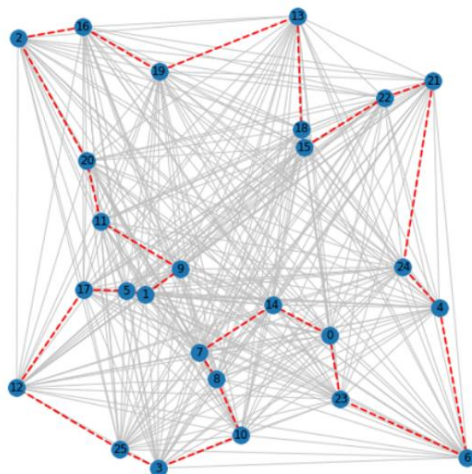
```

длина LCS: 3

Функция lcs принимает на вход строки A и B, а также их длины m и n соответственно. Функция создает двумерный динамический массив L размером (m+1) x (n+1) с помощью оператора new. Затем она заполняет этот массив значениями длин наибольших общих подпоследовательностей для всех подстрок A и B путем перебора всех возможных подстрок и сравнения символов.

Задача о коммивояжёре

Задача о коммивояжере (Travelling Salesman Problem, TSP) - это задача нахождения кратчайшего возможного маршрута, проходящего через все заданные города и возвращающегося в исходный город.



Известны только три метода нахождения точного решения:

1. Перебор грубой силой (brute force).
2. Ветвления и границ с полным перебором.
3. Динамическое программирование.

Решение методом грубой силы не подходило из-за вычислительной сложности. Прикинем на глаз, для 26 городов в симметричном варианте как верхний предел. Получаем $25!/2 = 7755605021665492992000000$ маршрутов.

Даже если проверять по миллиарду маршрутов в секунду, то солнце потухнет раньше, чем мы дождёмся результата.

Отличный метод «Метод ветвления и границ с полным перебором», хотя не прост для понимания, но на большинстве наборов данных даёт отличные результаты. Очень нестабильный по времени алгоритм, в некоторых ситуациях разница доходила более 1000 раз от среднего на типовых маршрутах. А самое печальное: нет возможности узнать, как долго ещё будет идти обработка, секунды или дни, всё случайно.

Решение задачи о коммивояжере методом динамического программирования основывается на следующей идее: мы строим таблицу размером $2^n \times n$ (где n - количество городов), где каждая строка представляет собой подмножество городов. Значение в ячейке (S, i) представляет собой длину кратчайшего пути, начинающегося в городе 1, посещающего все города из множества S и заканчивающегося в городе i . Затем мы заполняем эту таблицу значениями, используя уже вычисленные значения для более маленьких подмножеств городов. Наконец, мы выбираем наименьшее значение из последнего столбца

таблицы, соответствующее длине кратчайшего пути, проходящего через все города.

Алгоритм можно описать следующим образом:

1. Инициализируем таблицу значений D размером $2^n \times n$, где $D[S][i]$ представляет собой длину кратчайшего пути, начинающегося в городе 1, посещающего все города из множества S и заканчивающегося в городе i .
2. Заполняем первый столбец таблицы, устанавливая $D[\{1\}, 1]$ равным 0, а все остальные элементы равными бесконечности.
3. Для каждого подмножества S городов размера k от 2 до n :
 - Для каждого города i в S :
 - Вычисляем значение $D[S][i]$ как минимум из значений $D[S - \{i\}][j] + d(j, i)$, где j - все города из множества S , кроме i , а $d(j, i)$ - длина ребра (j, i) .
4. Выбираем минимальное значение в последнем столбце таблицы D и восстанавливаем маршрут, который соответствует этому значению.

Решение этой задачи довольно сложное и даже используя метод динамического программирования получается очень большое время выполнения, так что в этом случае обойдёмся без примера реализации и не будем испытывать на прочность наши компьютеры.

ГОРОДА	ОПЕРАЦИИ
6	72φ
7	5φ4φ
8	4φ32φ
...	...
15	13φ7674368φφφ
...	...
3φ	265252859812191,068436308489000000

Количество операций стремительно растёт

Задача о рюкзаке

Задача о рюкзаке — это классическая задача комбинаторной оптимизации, которая заключается в том, чтобы выбрать из заданного множества предметов

некоторое подмножество, которое помещается в рюкзак заданной вместимости, и при этом максимизировать суммарную стоимость выбранных предметов.



Формально, задача о рюкзаке может быть сформулирована следующим образом: дано n предметов с весами w_1, w_2, \dots, w_n и стоимостями v_1, v_2, \dots, v_n , а также рюкзак с вместимостью W . Требуется выбрать некоторое подмножество предметов так, чтобы их суммарный вес не превышал W , а суммарная стоимость была максимальной.

Метод динамического программирования может быть использован для решения этой задачи. Идея заключается в том, чтобы построить таблицу $dp[i][j]$, содержащую максимальную стоимость, которую можно получить, выбирая предметы из первых i предметов и имея рюкзак с вместимостью j . Значение в ячейке $dp[i][j]$ можно вычислить с помощью следующей рекуррентной формулы:

$$dp[i][j] = \max(dp[i-1][j], dp[i-1][j-w_i] + v_i), \text{ если } j \geq w_i$$
$$dp[i][j] = dp[i-1][j], \text{ если } j < w_i$$

здесь w_i и v_i — вес и стоимость i -го предмета соответственно. Первое слагаемое в формуле соответствует случаю, когда мы не берем i -й предмет, а второе слагаемое соответствует случаю, когда мы берем i -й предмет. Если $j < w_i$, то i -й предмет не может быть взят в рюкзак, поэтому значение $dp[i][j]$ равно значению $dp[i-1][j]$.

Итак, чтобы решить задачу о рюкзаке с помощью метода динамического программирования, мы должны заполнить таблицу dp с помощью указанной выше формулы и вернуть значение $dp[n][W]$, которое содержит максимальную стоимость, которую можно получить, выбирая предметы из всех n предметов и имея рюкзак с вместимостью W .

Пример кода для решения задачи о рюкзаке методом динамического программирования:

```
int knapsack(int n, int W, const vector<int>& w, const vector<int>& v) {
    vector<vector<int>>> dp(n + 1, vector<int>(W + 1, 0)); // создаем двумерный вектор для хранения
    значений функции
    // dp[i][j] - максимальная стоимость, которую можно получить, выбирая первые i предметов и
    имея рюкзак вместимости j.
    for (int i = 1; i <= n; ++i) { // цикл по предметам
        for (int j = 1; j <= W; ++j) { // цикл по вместимости рюкзака
            if (w[i - 1] <= j) { // если предмет i помещается в рюкзак вместимостью j
                dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - w[i - 1]] + v[i - 1]); // значение функции для i-го
                предмета и j-ой вместимости
                // max - выбираем максимальное значение из двух вариантов:
                // 1) предмет i не включаем в рюкзак, тогда максимальная стоимость не изменится и будет
                равна dp[i - 1][j]
                // 2) предмет i включаем в рюкзак, тогда максимальная стоимость будет равна стоимости
                i-го предмета + максимальной стоимости,
                // которую можно получить, выбирая первые i-1 предметов и имея рюкзак вместимости j-
                w[i-1]. Таким образом, максимальная
                // стоимость будет равна dp[i - 1][j - w[i - 1]] + v[i - 1].
            }
            else { // если предмет i не помещается в рюкзак вместимостью j
                dp[i][j] = dp[i - 1][j]; // максимальная стоимость не изменится и будет равна максимальной
                стоимости,
                // которую можно получить, выбирая первые i-1 предметов и имея рюкзак вместимости j.
            }
        }
    }
    return dp[n][W]; // возвращаем максимальную стоимость, которую можно получить, выбирая
    первые n предметов и имея рюкзак вместимости W.
}

int main() {
    setlocale(LC_ALL, "rus"); // устанавливаем локаль для корректного отображения кириллицы
    int n = 3; // количество предметов
    int W = 50; // вместимость рюкзака
    vector<int> w = { 10, 20, 30 }; // веса предметов
    vector<int> v = { 60, 100, 120 }; // стоимости предметов
    int ans = knapsack(n, W, w, v); // вычисляем максимальную стоимость
    cout << "Максимальная стоимость: " << ans << endl; // выводим результат
    return 0;
}
```

Максимальная стоимость: 220

В данном коде реализован решение задачи о рюкзаке методом динамического программирования. Функция `knapsack` принимает на вход количество предметов `n`, вместимость рюкзака `W` и два вектора - `w` и `v`, содержащие веса

и стоимости предметов соответственно. Функция возвращает максимальную стоимость, которую можно получить, выбирая предметы для помещения в рюкзак.

В целом, динамическое программирование является очень эффективным и универсальным подходом к решению задач, который может быть применен в различных областях, включая алгоритмы машинного обучения, оптимизацию и анализ данных, биоинформатику и многое другое.

Таким образом, динамическое программирование - это мощный инструмент, который может значительно повысить эффективность и точность алгоритмов и решить многие сложные задачи.

Задания для лабораторной работы:

Добавьте в ваше меню функции для следующих заданий:

1. Считайте из файла ваш массив структур и реализуйте поиск наибольшей общей последовательности, введенной пользователем, для одного из полей структур, например, название услуги.
2. Напишите функцию, которая будет находить F_n по номеру n . N вводится пользователем.(ряд Фибоначчи)
3. Начался пожар в офисе вашей компании(основываясь на ЛР №2), вам нужно собрать рюкзак и вынести из офиса вещей на самую большую сумму, чтобы их спасти. Напишите функцию подсчёта этой суммы, основываясь на «задаче о рюкзаке».

Во всех заданиях использовать методы динамического программирования.

Дополнительные задания:

1. Задача о максимальной сумме подмассива. Дан массив из N целых чисел. Необходимо найти непустой подмассив этого массива, который имеет наибольшую сумму. Решите задачу с помощью динамического программирования.(3 балла)
2. Задача о разбиении числа. Дано натуральное число N . Необходимо найти количество способов разбить это число на сумму натуральных чисел. Решите задачу с помощью динамического программирования.(3 балла)
3. Задача о кратчайшем пути в графе. Дан взвешенный ориентированный граф с N вершинами и M ребрами. Необходимо найти кратчайший путь от вершины

S до вершины T. Решите задачу с помощью динамического программирования.(3 балла)

Вопросы к лабораторной работе:

1. В чем заключается концепция динамического программирования?
2. Дайте определение последовательности?
3. Дайте определение последовательности Фибоначчи
4. Максимальная общая последовательность – это?
5. Что такое поиск наибольшей общей последовательности?
6. В чем суть задачи о коммивояжёре?
7. Какие методы решения задачи о коммивояжёре вы знаете?
8. В чем суть задачи о рюкзаке?
9. Что такое динамическое программирование?
10. В чем преимущества динамического программирования?