

Учреждение образования
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ
УНИВЕРСИТЕТ»

СТРУКТУРЫ ДАННЫХ
Лабораторный практикум
для студентов IT-специальностей

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. **Динамические структуры данных.**

Лабораторная работа №1

Перечисление. Объединение. STL контейнеры. Массив.

Дек. Однодвух связный список. Вектор;

Цель работы: ознакомление и практическое применение основных операций с контейнерами в стандартной библиотеке C++, таких как перечисление, объединение и работа с различными типами контейнеров: массивами, деками, одно- и двусвязными списками, векторами

1. Теоретическое сведение

Контейнеры в C++ представляют собой структуры данных, которые позволяют хранить и организовывать коллекции объектов. Стандартная библиотека C++ (STL) предоставляет широкий набор контейнеров, которые могут быть использованы в различных ситуациях и задачах.

1. Перечисление:

Перечисление (enumeration) - это пользовательский тип данных в C++, который позволяет определять набор именованных значений, которые могут быть использованы в программе в качестве констант. Перечисление упрощает написание кода, делая его более читаемым и легко поддерживаемым.

Перечисление может быть определено с помощью ключевого слова `enum` и следующего за ним списком значений, разделенных запятыми, заключенными в фигурные скобки:

```
enum Color { Red, Green, Blue };
```

Каждому элементу автоматически присваивается целочисленное значение, начиная с 0 для первого элемента, и увеличиваясь на 1 для каждого последующего элемента. Таким образом, Red будет иметь значение 0, Green - 1, Blue - 2. Также, возможна такая ситуация:

```
enum Color { Red = 1, Green = 2, Blue = 4 };
```

Имена перечислителей (элементов перечисления) должны быть уникальными. Значения же могут совпадать:

```
enum Color { Red = 1, Green = 1, Blue = 4 };
```

Мы можем использовать перечисления в своей программе, используя их имена вместо числовых значений. Например:

```
Color color = Red;
if (color == Blue) {
    // ...
}
```

Если создается переменная типа перечисления – она может принять значение только одного из своих элементов:

```
Color cvet = 2222; //Ошибка
Color cvet = Green;
```

Записать число в переменную типа Color нельзя. Можно записать только именованную константу, которая объявлена во время определения enum.

Перечисления могут быть полезны при работе с кодом, в котором нужно иметь набор именованных констант. Они улучшают читаемость кода и помогают избежать ошибок, связанных с использованием числовых значений напрямую.

2. Объединение:

Объединение (union) - это пользовательский тип данных в C++, который позволяет объединять несколько переменных разных типов в одну общую область памяти. При этом переменные разделяют одну область памяти и имеют одинаковые адреса. Размер Union равен размеру наибольшего элемента. Как и в случае структуры, члены объединения могут иметь любой тип за исключением типов void.

Достоинствами union является экономия памяти. Недостаток – в каждый момент времени в переменной типа union может храниться значение только одного поля из этих типов данных и возможно использовать лишь значение этого элемента (компонента). Определяется это тем, что все поля начинаются с одного адреса и их значения перекрываются.

Объединение определяется с помощью ключевого слова union и следующего за ним списком переменных, которые нужно объединить, заключенных в фигурные скобки:

```
union myUnion {           // объявление объединения с именем myUnion
    int intValue;          // поле типа int с именем intValue
    float floatValue;      // поле типа float с именем floatValue
};
```

Мы можем использовать любую из переменных объединения, но только одну за раз. Например:

```

union myUnion {           // объявление объединения с именем myUnion
    int intValue;         // поле типа int с именем intValue
    float floatValue;     // поле типа float с именем floatValue
};

myUnion myValue;          // создание переменной myValue типа myUnion
myValue.intValue = 10;     // установка значения поля intValue в 10

cout << myValue.intValue << endl; // вывод значения поля intValue на экран
cout << myValue.floatValue << endl; // вывод значения поля floatValue на экран
// здесь значение будет неопределенным,
// т.к. мы не устанавливали его явно

```

Объединения также могут быть использованы для битового доступа к памяти. Это позволяет получить доступ к каждому биту переменной в объединении. Для этого можно определить структуру с помощью union, где каждая переменная имеет размер 1 байт, а затем обращаться к отдельным битам через поля этих переменных.

Например, можно определить такую структуру:

```

union myBits {           // объявление объединения с именем myBits
    struct {              // объявление вложенной структуры
        unsigned char bit0 : 1; // бит 0
        unsigned char bit1 : 1; // бит 1
        unsigned char bit2 : 1; // бит 2
        unsigned char bit3 : 1; // бит 3
        unsigned char bit4 : 1; // бит 4
        unsigned char bit5 : 1; // бит 5
        unsigned char bit6 : 1; // бит 6
        unsigned char bit7 : 1; // бит 7
    } bits;               // переменная bits типа struct
    unsigned char byte;    // переменная byte типа unsigned char
};

// Данное объединение позволяет работать с битами и байтом одной переменной.
// В объединении определена структура с полями, каждое из которых представляет собой один бит числа.
// Также определена переменная byte типа unsigned char, которая занимает те же 8 бит, что и структура bits.
// При изменении значения одного из битов в структуре bits, будет изменяться значение переменной byte,
// а при изменении значения переменной byte, будут изменяться все 8 битов в структуре bits.

```

И мы можем получить доступ ко всем битам переменной:

```
myBits myValue;  
myValue.byte = 0xAA;  
cout << myValue.bits.bit0 << endl; // Выведет 0  
cout << myValue.bits.bit1 << endl; // Выведет 1  
cout << myValue.bits.bit2 << endl; // Выведет 0  
cout << myValue.bits.bit3 << endl; // Выведет 1  
cout << myValue.bits.bit4 << endl; // Выведет 0  
cout << myValue.bits.bit5 << endl; // Выведет 1  
cout << myValue.bits.bit6 << endl; // Выведет 0  
cout << myValue.bits.bit7 << endl; // Выведет 1
```

Здесь мы создаем переменную **myValue** типа **myBits**, присваиваем ей значение **0xAA**, которое в двоичном представлении имеет вид **10101010**. Затем мы обращаемся к отдельным битам через поля структуры **bits**.

Объединения не поддерживают наследование, поэтому они не могут быть использованы в качестве базовых классов. Также, как и структуры объединения нельзя сравнивать. Структуры – когда удобно хранить одновременно несколько вещей вместе в одной коробке, объединения – когда в коробке в каждый момент хранится только одна из этих вещей.

3. Контейнеры STL:

STL (Standard Template Library) - это библиотека шаблонов, которая предоставляет набор контейнеров и алгоритмов для работы с данными в языке программирования C++.

Контейнеры STL предоставляют реализации различных абстрактных структур данных, таких как массивы, списки, множества, карты и т.д. Они обладают высокой производительностью и эффективностью в использовании благодаря оптимизированным алгоритмам и встроенным функциям. Они также имеют стандартизированный интерфейс, что позволяет легко использовать их в различных проектах.

Одним из главных отличий контейнеров STL от других реализаций является стандартизированный интерфейс. Это позволяет легко переключаться между различными контейнерами в зависимости от требований проекта, не меняя основного кода. Также, STL предоставляет богатый набор алгоритмов для работы с контейнерами, что упрощает обработку данных.

Кроме того, STL предоставляет контейнеры с высокой производительностью и оптимизированными алгоритмами. Например, контейнер `vector` использует непрерывную память для хранения данных, что

обеспечивает быстрый доступ к элементам массива. Контейнер `map` использует сбалансированные деревья для хранения пар ключ-значение, что обеспечивает быстрый доступ к элементам и поддерживает автоматическую сортировку.

Также, STL предоставляет удобные инструменты для работы с итераторами, что упрощает обход и обработку данных в контейнерах. Это также позволяет легко использовать контейнеры STL с алгоритмами STL, что повышает эффективность их использования.

4. Массив:

Массив - это структура данных, которая содержит фиксированное количество элементов одного типа, расположенных в памяти последовательно. Каждый элемент в массиве имеет уникальный индекс, который позволяет обращаться к нему напрямую.

Массивы являются одним из наиболее распространенных и важных структур данных в программировании. Они используются для хранения данных, которые могут быть обработаны в цикле или другом повторяющемся процессе. Массивы могут быть использованы для хранения различных типов данных, таких как целые числа, дробные числа, символы и другие объекты.

Создание массива в C++ осуществляется следующим образом:

```
тип_элементов имя_массива[размер_массива];
```

Например, чтобы создать массив из 10 целых чисел, мы можем написать:

```
int myArray[10];
```

Изменение значения элемента массива происходит путем присваивания значения элементу. Например, чтобы изменить значение первого элемента массива на 10, нужно написать:

```
myArray[0] = 10;
```

Если необходимо, чтобы нельзя было изменять значения элементов массива, то такой массив можно определить как константный с помощью ключевого слова `const`:

```
const int numbers[4]{ 1,2,3,4 };  
// numbers[1] = 23; // Ошибка - значения элементов массива изменить нельзя
```

Длина массива не всегда бывает известна. Однако может потребоваться получить ее. Первый способ представляет применение оператора `sizeof`:

```
#include <iostream>  
int main() {  
    int numbers[]{ 10, 20, 30, 40 };  
    std::cout << "Length: " << sizeof(numbers) / sizeof(numbers[0]) << std::endl; // Length: 4  
}
```

Второй способ представляет применение встроенной библиотечной функции `std::size()`:

```
#include <iostream>

int main() {
    int numbers[]{ 10, 20, 30, 40 };
    int count = std::size(numbers);
    std::cout << "Length: " << count << std::endl; // Length: 4
}
```

Массивы могут быть использованы в программировании для хранения больших объемов данных, которые можно обрабатывать с помощью циклов. Например, мы можем использовать массив для хранения оценок студентов и производить расчеты среднего балла.

Однако массивы также имеют свои недостатки. Во-первых, размер массива должен быть определен заранее, что может привести к проблемам, если необходимо хранить большое количество данных и заранее неизвестен их размер. Во-вторых, массивы могут занимать много места в памяти, если используются для хранения больших объемов данных. Наконец, добавление или удаление элементов в массив может быть сложным процессом, так как это потребует перемещения всех элементов в массиве, расположенных после измененного элемента.

Тем не менее, массивы по-прежнему являются важным элементом в программировании и используются в различных алгоритмах и структурах данных. В C++ есть и другие контейнеры, такие как векторы, динамические массивы и списки, которые предлагают решение некоторых проблем массива.

5. Дек:

Дек (deque) - это структура данных, которая представляет собой двустороннюю очередь. Она подобна вектору, но позволяет эффективно добавлять и удалять элементы не только с конца, но и с начала контейнера.

Дек может быть реализован как массив, который автоматически расширяется или сокращается при добавлении или удалении элементов. Однако часто используется реализация дека в виде двусвязного списка.

В стандартной библиотеке C++ дек реализован как контейнер `std::deque` в заголовочном файле `<deque>`. Он предоставляет удобные методы для работы с деком, такие как `push_front()`, `push_back()`, `pop_front()`, `pop_back()`, `at()`, `front()`, `back()`, `empty()`, `size()` и т.д.

Классическим примером использования дека является задача обработки последовательности элементов, где необходимо добавлять и удалять элементы как с начала, так и с конца контейнера.

Рассмотрим, например, задачу обработки запросов на добавление и удаление элементов из начала и конца списка. Предположим, что нам нужно реализовать структуру данных, которая позволяет быстро выполнять следующие операции:

- добавление элемента в начало списка
- добавление элемента в конец списка
- удаление элемента из начала списка
- удаление элемента из конца списка

Для решения этой задачи мы можем использовать дек. Например, следующий код демонстрирует, как можно использовать дек для решения этой задачи:

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

int main() {
    deque<int> mydeque;

    // Добавление элемента в начало списка
    mydeque.push_front(10);

    // Добавление элемента в конец списка
    mydeque.push_back(20);

    // Удаление элемента из начала списка
    mydeque.pop_front();

    // Удаление элемента из конца списка
    mydeque.pop_back();

    // Выводим содержимое дека на экран
    for (auto it = mydeque.begin(); it != mydeque.end(); ++it) {
        cout << *it << " ";
    }
    cout << endl;
    return 0;
}
```

Здесь мы создаем пустой дек `mydeque` и выполняем операции добавления и удаления элементов. Для добавления элементов в начало и конец списка мы используем методы `push_front()` и `push_back()`, а для удаления элементов из начала и конца списка - методы `pop_front()` и `pop_back()`. Кроме того, мы выводим содержимое дека на экран с помощью цикла `for`.

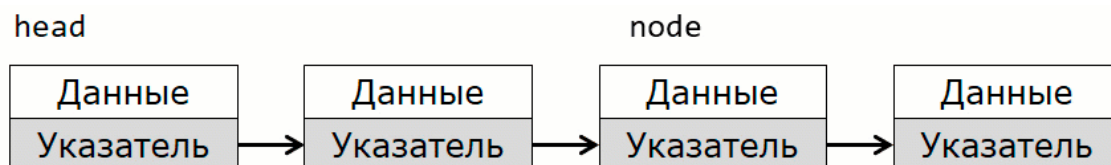
Как видно из примера, дек предоставляет удобный интерфейс для работы с двусторонней очередью, который позволяет быстро выполнять операции добавления и удаления элементов как с начала, так и с конца списка.

Также дек поддерживает операцию доступа к элементам по индексу, аналогично массиву или вектору, с помощью оператора `[]`. Однако, поскольку дек не гарантирует непрерывное расположение элементов в памяти, операция доступа по индексу может иметь худшую производительность, чем в массиве или векторе. Поэтому рекомендуется использовать дек в первую очередь для операций добавления и удаления элементов в начало и конец контейнера.

6. Одно- и двусвязные списки:

В C++ существует несколько различных способов реализации списков, но самым распространенным является односвязный список.

Односвязный список - это структура данных, которая состоит из узлов (или элементов), связанных друг с другом однонаправленной связью. Каждый узел содержит две вещи: данные и ссылку на следующий узел в списке. Первый узел списка называется головой списка, а последний узел не имеет ссылки на следующий элемент и обозначается как "конец" списка.



Для доступа к элементам списка, обычно начинают с головы списка и переходят по ссылкам на следующий элемент до тех пор, пока не дойдут до нужного узла. Таким образом, каждый узел содержит только ссылку на следующий узел, а не на предыдущий.

Пример реализации класса для односвязного списка может выглядеть следующим образом:

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

```

class Node {
public:
    int data;           // Значение узла
    Node* next;         // Указатель на следующий элемент
};

class LinkedList {
public:
    LinkedList() {      // Конструктор класса LinkedList
        head = NULL;   // Устанавливаем указатель на голову списка в NULL
    }
    void insert(int value) { // Метод класса для вставки нового элемента в список
        Node* newNode = new Node; // Создаем новый узел списка
        newNode->data = value;     // Задаем значение для нового узла
        newNode->next = head;      // Новый узел ссылается на текущую голову списка
        head = newNode;           // Устанавливаем новый узел как новую голову списка
    }
    void printList() {      // Метод класса для печати списка на экран
        Node* temp = head;   // Создаем временный указатель на голову списка
        while (temp != NULL) { // Пока не достигнут конец списка
            cout << temp->data << " "; // Выводим значение текущего узла списка на экран
            temp = temp->next; // Переходим к следующему узлу списка
        }
    }
private:
    Node* head;           // Указатель на голову списка
};

int main() {
    LinkedList myList;    // Создаем новый объект класса LinkedList
    myList.insert(5);      // Добавляем новый элемент со значением 5 в список
    myList.insert(10);     // Добавляем новый элемент со значением 10 в список
    myList.insert(15);     // Добавляем новый элемент со значением 15 в список
    myList.printList();    // Печатаем список на экран
    return 0;             // Завершаем программу
}

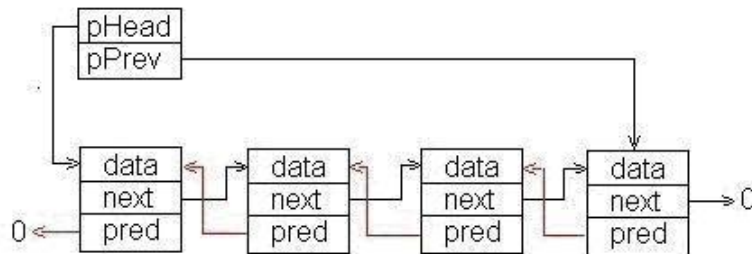
```

Здесь Node представляет узел списка, а LinkedList - сам список. Метод insert добавляет новый элемент в начало списка, а метод printList выводит все элементы списка на экран. В методе main мы создаем новый список, добавляем в него три элемента и выводим список на экран.

Двунаправленный (двусвязный) список – это динамическая структура данных, состоящая из последовательности элементов, каждый из которых содержит:

поле – информационное;

два поля с указателями на соседние элементы: одно поле содержит ссылку на следующий элемент, другое поле – ссылку на предыдущий элемент.



При этом два соседних элемента должны содержать взаимные ссылки друг на друга.

Наличие ссылок на следующее звено и на предыдущее позволяет двигаться по списку от каждого звена в любом направлении: от звена к концу списка или от звена к началу списка, поэтому такой список называют двунаправленным.

Двусвязные списки могут использоваться в программах для ускорения операций вставки, удаления и поиска элементов, поскольку предыдущий узел каждого элемента также известен. Это позволяет быстро обращаться к соседним элементам, не перебирая весь список.

Реализация двусвязного списка на языке программирования C++:

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

// Описание класса для узла двусвязного списка
class Node {
public:
    int data; // Значение узла
    Node* next; // Указатель на следующий узел
    Node* prev; // Указатель на предыдущий узел
};

// Описание класса для двусвязного списка
class DoublyLinkedList {
public:
    // Конструктор класса, инициализирует указатели головы и хвоста списка
    DoublyLinkedList() {
        head = NULL;
        tail = NULL;
    }
};
```

```

}
// Функция для добавления элемента в начало списка
void insert(int value) {
    Node* newNode = new Node; // Выделяем память для нового узла
    newNode->data = value; // Задаем значение нового узла
    newNode->next = head; // Устанавливаем указатель на следующий узел нового узла
    newNode->prev = NULL; // Устанавливаем указатель на предыдущий узел нового
узла в NULL
    if (head != NULL) {
        head->prev = newNode; // Если список не пуст, то указатель на предыдущий узел
головного элемента
        // устанавливаем на новый узел
    }
    head = newNode; // Устанавливаем указатель на головной элемент на новый узел
    if (tail == NULL) {
        tail = head; // Если список пуст, то устанавливаем указатель на хвостовой элемент
на головной элемент
    }
}

// Функция для печати списка
void printList() {
    Node* temp = head; // Указатель на головной элемент
    while (temp != NULL) { // Проходимся по списку, пока не дойдем до конца
        cout << temp->data << " "; // Выводим значение текущего узла на экран
        temp = temp->next; // Переходим к следующему узлу
    }
}

private:
    Node* head; // Указатель на головной элемент списка
    Node* tail; // Указатель на хвостовой элемент списка
};

// Главная функция
int main() {
    DoublyLinkedList myList; // Создаем новый двусвязный список
    myList.insert(5); // Добавляем элемент со значением 5 в начало списка
    myList.insert(10); // Добавляем элемент со значением 10 в начало списка
    myList.insert(15); // Добавляем элемент со значением 15 в начало списка
    myList.printList(); // Выводим список на экран
    return 0; // Завершаем программу
}

```

Здесь Node представляет узел списка, а DoublyLinkedList - сам список. Метод insert добавляет новый элемент в начало списка и обновляет ссылки на предыдущий и следующий узлы. Метод printList выводит все элементы списка

на экран. В методе `main` мы создаем новый список, добавляем в него три элемента и выводим список на экран.

Подводя итог, можно сказать, что односвязный список имеет ссылку только на следующий элемент, что делает его простым и легковесным. Вставка и удаление элементов в односвязном списке происходят быстро, но поиск элементов может занимать время, так как нужно последовательно перебрать все элементы от головы списка до искомого элемента.

Двусвязный список имеет ссылки на следующий и предыдущий элементы, что позволяет быстро перемещаться как вперед, так и назад по списку. Он более гибкий, чем односвязный список, и может использоваться для реализации более сложных структур данных, например, очередей с двойным концом. Однако, двусвязный список занимает больше памяти, чем односвязный список, и требует больше усилий для реализации.

Выбор между односвязным и двусвязным списком зависит от задачи, которую необходимо решить, и от того, какие операции будут чаще выполняться над списком. Если необходимо быстро вставлять и удалять элементы в списке, и поиск элементов не является основной задачей, то можно выбрать односвязный список. Если требуется быстрый доступ к элементам и перемещение по списку как вперед, так и назад, то двусвязный список может быть лучшим выбором.

7. Вектор:

Вектор - это динамический массив в языке программирования C++. Он представляет собой последовательность элементов одного типа, которые хранятся в памяти непрерывно и могут изменять свой размер в процессе выполнения программы.

Основные преимущества векторов в C++:

1. Быстрый доступ к элементам по индексу;
2. Возможность динамического изменения размера;
3. Эффективная работа с памятью благодаря использованию непрерывного блока памяти;
4. Поддержка большинства стандартных алгоритмов, таких как сортировка, поиск, вставка, удаление и т.д.;
5. Поддержка итераторов, которые позволяют перебирать элементы вектора.

Чтобы создать вектор в C++, необходимо объявить объект класса `std::vector` и указать тип хранимых элементов. Например:

```
#include <vector>
std::vector<int> myVector; // объявление пустого вектора целых чисел
```

Чтобы получить доступ к элементам вектора, можно использовать оператор `[]`:

```
int x = myVector[0]; // получение первого элемента вектора
```

Чтобы получить текущий размер вектора, можно использовать метод `size()`:

```
int size = myVector.size(); // получение размера вектора
```

Одним из важных свойств векторов является возможность динамического изменения их размера (в отличие от массива). Для этого векторы предоставляют несколько методов, например, методы `resize()` и `reserve()`.

Метод `resize()` изменяет размер вектора, добавляя новые элементы, если нужно, или удаляя избыточные элементы:

```
std::vector<int> myVector; // объявление пустого вектора
myVector.resize(10); // изменение размера вектора на 10 элементов
```

Метод `reserve()` выделяет память для заданного количества элементов, но не изменяет размер самого вектора:

```
std::vector<int> myVector; // объявление пустого вектора
myVector.reserve(10); // выделение памяти на 10 элементов
```

Кроме того, векторы в C++ поддерживают различные алгоритмы и операции для работы с элементами. Например, методы `insert()` и `erase()` позволяют вставлять и удалять элементы из вектора:

```
std::vector<int> myVector{ 1, 2, 3, 4, 5 };
myVector.insert(myVector.begin() + 2, 50); // вставка элемента со значением 50 в позицию
с индексом 2
myVector.erase(myVector.begin() + 3); // удаление элемента с индексом 3
```

Кроме того, векторы могут использоваться для реализации других структур данных, таких как стеки и очереди.

Векторы в C++ являются мощным инструментом, который упрощает работу с динамическими массивами и обеспечивает эффективное управление памятью.

Однако, векторы также имеют свои ограничения. Векторы хранят элементы в непрерывном блоке памяти, что означает, что при изменении

размера вектора может произойти перераспределение памяти и перемещение элементов в новый блок памяти. Это может привести к нежелательному поведению при работе с указателями и ссылками на элементы вектора. Также, если вектор содержит большое количество элементов, то его создание и обработка может занять много времени и занимать много памяти.

2. Задания к лабораторной работе №1

1. Написать программу, которая создает массив из 10 целых чисел и выводит на экран только четные числа в обратном порядке.
2. Написать программу, которая запрашивает у пользователя 5 чисел и помещает их в дек, а затем выводит числа в порядке, обратном тому, в котором они были введены.
3. Написать программу, которая создает односвязный список из 5 элементов, содержащих целые числа, и удаляет первый и последний элементы.
4. Написать программу, которая создает двусвязный список из 5 элементов, содержащих целые числа, и заменяет каждый элемент на сумму его значения и значения следующего элемента в списке.
5. Написать программу, которая создает вектор из 10 строковых значений и удаляет из него все элементы, начинающиеся на букву "а".
6. Написать программу, которая создает массив из 10 целых чисел и проверяет, все ли они являются простыми числами.
7. Написать программу, которая создает дек из 10 элементов, содержащих целые числа, и выводит на экран каждый второй элемент.
8. Написать программу, которая создает односвязный список из 5 элементов, содержащих целые числа, и меняет местами значения первого и последнего элементов.
9. Написать программу, которая создает двусвязный список из 5 элементов, содержащих целые числа, и находит сумму элементов, находящихся между первым и последним элементами списка.
10. Написать программу, которая создает вектор из 10 элементов, содержащих целые числа, и находит сумму всех отрицательных элементов.
11. Написать программу, которая создает массив из 10 целых чисел и выводит на экран только те числа, которые делятся на 3 без остатка.
12. Написать программу, которая создает дек из 10 строковых значений и выводит на экран все элементы, содержащие букву "о".

13. Написать программу, которая создает односвязный список из 5 элементов, содержащих целые числа, и удваивает значения всех элементов списка.
14. Написать программу, которая создает двусвязный список из 5 элементов, содержащих целые числа, и удаляет каждый второй элемент списка.

Замечание. Выполнить необходимо все задания.

3. Контрольные вопросы к лабораторной работе №1

1. Какие особенности и преимущества имеют STL контейнеры по сравнению с другими типами контейнеров?
2. Как использовать перечисление для описания константных значений в программе?
3. Как работает структура объединения и в каких случаях ее использование может быть полезным?
4. В чем разница между деком и вектором, и какой из них лучше использовать в каких случаях?
5. Как осуществить доступ к элементам массива и какие операции можно выполнять с массивом?
6. Как реализовать двухсвязный список и какие операции можно выполнять с его элементами?
7. Как реализовать односвязный список и как он отличается от двухсвязного списка

Лабораторная работа №2

Схема БД. Меню программы

Цель работы: изучить принцип проектирования схемы базы данных (БД) для программы, которая имеет меню с несколькими разделами функционалом; изучить способ создания консольного меню для программы.

1. Теоретическое сведение

1. Схема БД:

Термин «схема базы данных» может означать как наглядное представление базы данных, так и набор правил, которым она подчиняется, либо полный комплект объектов, принадлежащих конкретному пользователю.

Создание хорошей схемы базы данных (БД) является важным этапом при разработке любого приложения, использующего базу данных. Ниже приведены несколько полезных теоретических принципов, которые помогут создать эффективную схему БД:

Нормализация данных:

Нормализация - это процесс структурирования данных в базе данных в соответствии с набором правил, известных как нормальные формы. Нормализация помогает устранить избыточность данных и повысить эффективность запросов. При проектировании схемы БД рекомендуется использовать нормальные формы, чтобы избежать проблем с целостностью данных.

Использование первичных ключей:

Первичный ключ - это уникальный идентификатор каждой записи в таблице. Использование первичных ключей помогает сделать таблицы более структурированными и повысить производительность запросов. Рекомендуется использовать целочисленные значения в качестве первичных ключей, а также убедиться, что они автоинкрементируются для предотвращения коллизий.

Использование связей между таблицами:

Рекомендуется использовать связи между таблицами для связывания данных из разных таблиц в базе данных. Это позволяет избежать избыточности данных и повысить производительность запросов. Однако не

следует увлекаться созданием слишком сложных связей между таблицами, поскольку это может привести к проблемам с производительностью и сложности обслуживания базы данных.

Использование индексов:

Индексы - это структуры данных, которые позволяют быстро находить записи в таблице по заданному критерию. Использование индексов может существенно ускорить запросы к базе данных, но также может привести к избыточности данных и снижению производительности при вставке и обновлении данных. Рекомендуется использовать индексы только в тех случаях, когда они действительно необходимы для ускорения выполнения запросов.

Использование правильных типов данных: При проектировании схемы БД рекомендуется использовать подходящие типы данных для каждого поля таблицы. Например, для целых чисел следует использовать целочисленные типы данных, а для дат - типы данных даты и времени.

На рисунке 1.1 изображена примерная схема БД для ввода данных:

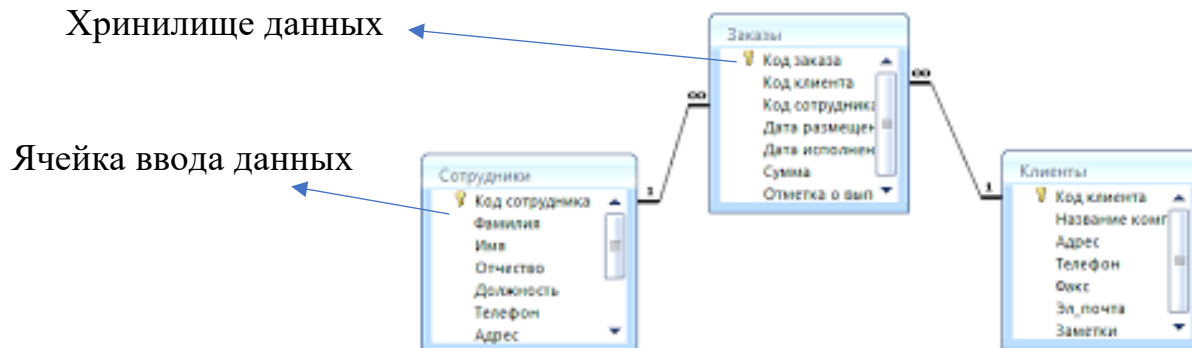


Рисунок 1.1 – Схема БД для ввода данных о сотрудниках и клиентах

Вот еще пример схемы БД для ввода данных и обращения к хранилищу этих данных представленный на рисунке 1.2:

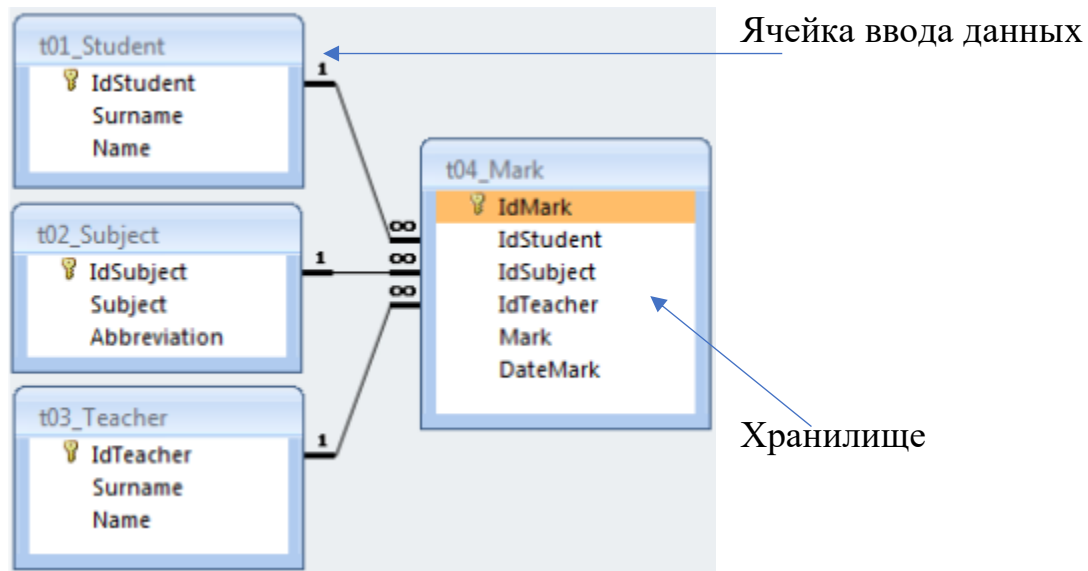


Рисунок 1.2 – Схема ввода данных о студентах и т.п., и хранилища этих данных

2. Меню программы:

Написание меню консольной программы включает в себя несколько шагов:

1. Определение функций, которые будут выполняться при выборе опций меню.
2. Создание структуры меню, содержащей список опций и соответствующих им функций.
3. Написание цикла, который будет отображать меню и ожидать ввода пользователя.
4. Обработка ввода пользователя и вызов соответствующей функции.
5. Повторение цикла до тех пор, пока пользователь не выберет опцию выхода из программы.

Пример кода для меню по схемам, приведенным выше:

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

void Menu() {
    int choice;
```

```
cout << "Меню:" << endl;
cout << "1. Ввод данных." << endl;
cout << "2. Вывод данных." << endl;
cout << "3. Редактирование данных." << endl;
cout << "4. Удаление данных." << endl;
cout << "5. Выход из программы." << endl;
cout << "Ваш выбор: ";
    cin >> choice;
}

void input() {
    cout << "Выбран пункт ввода данных." << endl;
}

void editor() {
    cout << "Выбран пункт редактирования данных." << endl;
}

void output() {
    cout << "Выбран пункт вывода данных." << endl;
}

void del() {
    cout << "Выбран пункт удаления данных." << endl;
}

int main() {
    setlocale(LC_ALL, "RUS");
    do {
        Menu();
        switch (choice) {
            case 1:
                system("cls"); // очистка консоли
                input();
                system("pause"); // задержка консоли
                system("cls");
                break;
            case 2:
                system("cls");
                output();
                system("pause");
                system("cls");
                break;
            case 3:
                system("cls");
                editor();
```

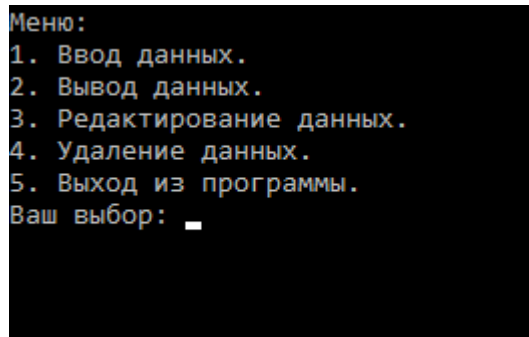
```

        system("pause");
        system("cls");
        break;
    case 4:
        system("cls");
        del();
        system("pause");
        system("cls");
        break;
    case 5:
        system("cls");
        cout << "Выход...." << endl;
        break;
    default:
        cout << "Неверный ввод. Попробуйте еще раз..." << endl;
        break;
    }
} while (choice != 5);

return 0;
}

```

Результат работы программы:



```

Меню:
1. Ввод данных.
2. Вывод данных.
3. Редактирование данных.
4. Удаление данных.
5. Выход из программы.
Ваш выбор: _

```

Данное меню реализовано с помощью switch-case и do-while . Однако оно может быть реализовано иными способами.

2. Задание к лабораторной работе №2

Разработать консольное меню программы и составить схему базы данных для составленного меню по определенным данным. Разработать функционал ввода, вывода, редактирования и удаления данных с использованием STL контейнеров.

Замечание. От исполнителя на данном этапе не требуется создания полностью готового программного продукта, только рабочий интерфейс в

котором присутствует весь базовый функционал. Функционал программы будет редактироваться и улучшаться в последующих лабораторных работах.

Набор случаев для каждого варианта (вариант соответствует номеру в подгруппе):

1. Вы являетесь менеджером в IT-компании, где разрабатываются программные продукты. Для учета клиентов и заказов вы создаете базу данных, в которой хранятся данные клиентов (имя, фамилия, номер телефона, адрес электронной почты) и информация о заказах (тип продукта, его функциональные требования, стоимость). Ваша компания имеет несколько проектов, и вы хотите видеть статистику по каждому проекту.

2. Вы работаете в фитнес-клубе, где предоставляются услуги тренажерного зала, групповых занятий и персональных тренировок. Для учета клиентов и проведенных услуг вы создаете базу данных, в которой хранятся данные клиентов (имя, фамилия, номер телефона, адрес электронной почты) и записи о проведенных тренировках (вид услуги, дата, продолжительность, стоимость). Вы хотите видеть статистику по каждому виду услуг.

3. Вы являетесь менеджером в компании по продаже электроники, где предоставляются услуги консультации и монтажа техники. Для учета клиентов и выполненных услуг вы создаете базу данных, в которой хранятся данные клиентов (имя, фамилия, номер телефона, адрес электронной почты) и записи о выполненных услугах (вид услуги, дата, стоимость). У вас есть несколько филиалов, и вы хотите видеть статистику по каждому филиалу.

4. Вы работаете в автосервисе, где предоставляются услуги по ремонту и техническому обслуживанию автомобилей. Для учета клиентов и выполненных работ вы создаете базу данных, в которой хранятся данные клиентов (имя, фамилия, номер телефона, адрес электронной почты) и записи о проведенных работах (вид работ, дата, стоимость). Вы хотите видеть статистику по каждому типу работ.

5. Вы являетесь врачом в частной клинике, где проводятся медицинские консультации и диагностические и лечебные процедуры. Для учета пациентов и проведенных процедур вы создаете базу данных, в которой хранятся данные пациентов (имя, фамилия, дата рождения, номер телефона) и записи о проведенных процедурах (вид процедуры, дата, результаты, стоимость). Вы хотите видеть статистику по каждому виду процедур.

6. Вы являетесь руководителем проекта по разработке сайта для интернет-магазина. Для учета работ и коммуникации с заказчиком вы создаете базу данных, в которой хранятся данные о заказчике (имя, фамилия, номер телефона, адрес электронной почты) и записи о выполненных работах (вид работ, дата, стоимость). Вы хотите видеть статистику по каждому этапу проекта.

7. Вы работаете в ресторане, где предоставляются услуги кулинарии и обслуживания гостей. Для учета посетителей и выполненных услуг вы создаете базу данных, в которой хранятся данные о клиенте (имя, фамилия, номер телефона, адрес электронной почты) и записи о выполненных услугах (вид услуги, дата, стоимость). Вы хотите видеть статистику по каждому блюду.

8. Вы являетесь финансовым аналитиком в компании, занимающейся продажей товаров по интернету. Для учета покупок и клиентов вы создаете базу данных, в которой хранятся данные о покупателе (имя, фамилия, номер телефона, адрес электронной почты) и записи о покупках (вид товара, дата, стоимость). Вы хотите видеть статистику по каждому товару.

9. Вы являетесь руководителем проекта по разработке мобильного приложения. Для учета работ и коммуникации с заказчиком вы создаете базу данных, в которой хранятся данные о заказчике (имя, фамилия, номер телефона, адрес электронной почты) и записи о выполненных работах (вид работ, дата, стоимость). Вы хотите видеть статистику по каждому этапу проекта.

10. Вы работаете в транспортной компании, предоставляющей услуги по перевозке грузов. Для учета клиентов и проведенных перевозок вы создаете базу данных, в которой хранятся данные о клиенте (имя, фамилия, номер телефона, адрес электронной почты) и записи о проведенных перевозках (тип груза, дата, маршрут, стоимость). Вы хотите видеть статистику по всем клиентам и перевозкам.

11. Вы являетесь управляющим ресторана быстрого питания. Для учета продуктов и сотрудников вы создаете базу данных, в которой хранятся данные о продуктах (наименование, ингредиенты, стоимость) и сотрудниках (имя, фамилия, должность, контактные данные). Вы хотите видеть статистику по каждому блюду.

12. Вы работаете в агентстве недвижимости и занимаетесь продажей квартир. Для учета клиентов и продаж вы создаете базу данных, в которой хранятся данные о клиенте (имя, фамилия, номер телефона, адрес электронной почты) и записи о продажах (тип квартиры, дата, стоимость). Вы хотите видеть статистику по каждому типу квартир.

13. Вы являетесь менеджером в компании, занимающейся техническим обслуживанием оборудования. Для учета клиентов и проведенных работ вы создаете базу данных, в которой хранятся данные о клиенте (имя, фамилия, номер телефона, адрес электронной почты) и записи о выполненных работах (вид работ, дата, стоимость). Вы хотите видеть статистику по каждому виду работ.

14. Вы работаете в клинике, предоставляющей медицинские услуги. Для учета пациентов и проведенных процедур вы создаете базу данных, в которой хранятся данные о пациенте (имя, фамилия, номер телефона, адрес электронной почты) и записи о проведенных процедурах (вид процедуры, дата, результаты, стоимость). Вы хотите видеть статистику по каждому виду процедур.

15. Вы являетесь руководителем команды разработчиков программного обеспечения. Для учета работ и коммуникации в команде вы создаете базу данных, в которой хранятся данные о сотрудниках (имя, фамилия, должность, контактные данные) и записи о выполненных работах (вид работ, дата, стоимость). Вы хотите видеть статистику по каждому сотруднику и этапу проекта.

16. Вы являетесь владельцем интернет-магазина и хотите создать базу данных, в которой хранятся данные о продуктах (наименование, описание, цена) и покупателях (имя, фамилия, адрес доставки, контактные данные). Вы хотите видеть статистику по каждому продукту и покупателю, а также по количеству продаж и общей выручке.

3. Контрольные вопросы к лабораторной работе №2

1. Что понимают под «Схема БД»?
2. Какие существуют принципы построения схемы БД?
3. Что такое нормализация?
4. Что такое первичный ключ?
5. Для чего нужна связь между таблицами БД?
6. Какие типы данных нужно использовать для схем?
7. Как реализовать консольное меню?

Лабораторная работа №3

Двоичные и текстовые файлы

Цель работы: ознакомление с основными операциями чтения и записи данных в файлы в двоичном и текстовом форматах, а также сравнение их особенностей и преимуществ; создание, открытие, запись и считывание данных из текстовых файлов.

1. Теоретическое сведение

Файлы - это один из наиболее распространенных способов хранения и передачи данных в компьютерных системах. В этой лабораторной работе мы будем изучать два основных типа файлов: текстовые и двоичные.

Текстовые файлы - это файлы, содержащие данные в читаемом формате, которые могут быть открыты и прочитаны любым текстовым редактором. Текстовые файлы могут содержать символы ASCII, Unicode или другие кодировки, которые используются для представления текстовой информации.

Двоичные файлы - это файлы, содержащие данные в бинарном формате, которые не могут быть прочитаны простым текстовым редактором. Двоичные файлы могут содержать любые данные, включая изображения, звуковые файлы, видео, архивы и другие бинарные данные.

1. Текстовые файлы:

Для работы с текстовыми файлами в C++ используются классы `std::ifstream` и `std::ofstream` из библиотеки `<fstream>`. Класс `std::ifstream` используется для чтения из файла, а класс `std::ofstream` используется для записи в файл.

В языке C для работы с файлами не используются операторы, вместо этого применяются функции, включенные в стандартную библиотеку. Прототипы функций для работы с файлами находятся в заголовочном файле `<stdio.h>`. В операционной системе C существуют три стандартных потока: стандартный поток вывода `stdout`, стандартный поток ввода `stdin` и стандартный поток вывода ошибок `stderr`.

Для открытия текстового файла в C++ используется функция `open()` класса `std::ifstream` или `std::ofstream`. При этом необходимо указать имя файла

и режим открытия (для чтения, записи или обоих). Режим открытия текстового файла может быть задан следующим образом:

- `std::ios::in` для открытия файла для чтения;
- `std::ios::out` для открытия файла для записи;
- `std::ios::app` для открытия файла для записи в конец файла;
- `std::ios::ate` для открытия файла и установки указателя в конец файла;
- `std::ios::trunc` для открытия файла и обрезания его до нулевой длины.

Например, чтобы открыть файл «file.txt» для чтения, можно использовать следующий код:

```
#include <fstream>
#include <iostream>

int main() {
    setlocale(LC_ALL, "rus");
    std::ifstream file("file.txt");

    if (!file.is_open()) {
        // Обработка ошибки
        std::cerr << "Ошибка\n";
    }

    file.close();

    return 0;
}
```

Место расположения файла, который вы хотите прочитать в программе на C++, зависит от того, как вы обращаетесь к нему в вашем коде.

Если вы используете абсолютный путь, то файл может находиться в любом месте на жестком диске. Абсолютный путь - это полный путь к файлу начиная от корневого каталога файловой системы.

Если вы используете относительный путь, то файл должен быть расположен в той же директории, где находится запущенный исполняемый файл, или в поддиректории.

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

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

Функция `getline()` используется для чтения строки из файла. Функция принимает два параметра: объект типа `std::istream&` и объект типа `std::string&`, в который будет записана прочитанная строка. Например, чтобы прочитать строки из файла, можно использовать следующий код:

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

int main() {
    setlocale(LC_ALL, "rus");
    std::ifstream file("file.txt"); // открываем файл для чтения

    if (!file.is_open()) { // проверяем, успешно ли открыт файл
        std::cerr << "Ошибка!\n";
        return 1;
    }

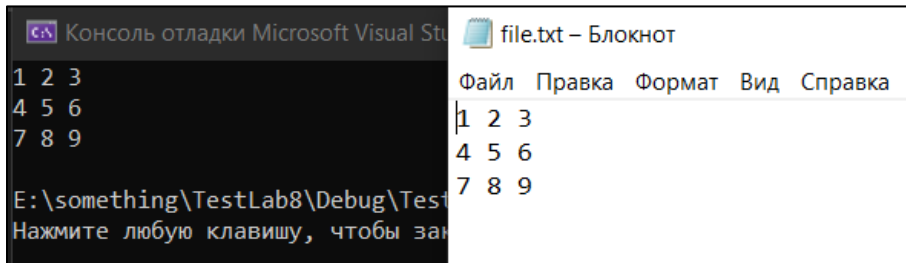
    std::string line;

    while (std::getline(file, line)) { // читаем файл построчно
        std::cout << line << '\n'; // выводим каждую строку на консоль
    }

    file.close(); // закрываем файл

    return 0;
}
```

Результат работы программы и вывод данных в файл:



The screenshot displays two windows side-by-side. The left window, titled 'Консоль отладки Microsoft Visual Studio', shows the program's output: three lines of numbers '1 2 3', '4 5 6', and '7 8 9'. The right window, titled 'file.txt – Блокнот', shows the contents of the file, which are identical to the console output: '1 2 3', '4 5 6', and '7 8 9'.

В этом примере мы открываем файл `file.txt` для чтения с помощью объекта `std::ifstream`. Затем мы проверяем, был ли файл успешно открыт с помощью метода `is_open()`. Если файл был успешно открыт, мы читаем файл построчно с помощью функции `std::getline()`, сохраняем каждую строку в объекте `std::string` и выводим ее на консоль. Наконец, мы закрываем файл с

помощью метода `close()`. Если файл не был успешно открыт, мы выводим сообщение об ошибке на стандартный поток ошибок (`std::cerr`).

Функция `put()` используется для записи символа в файл. Функция принимает символ и возвращает объект типа `std::ostream&`. Например, чтобы записать символ в файл, можно использовать следующий код:

```
#include <fstream>
#include <iostream>

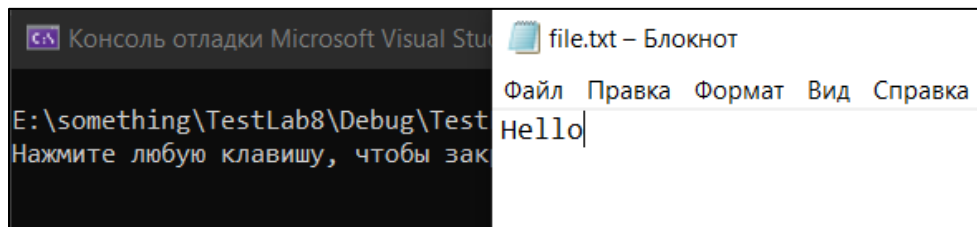
int main() {
    setlocale(LC_ALL, "rus");
    std::fstream file("file.txt", std::ios::out); // открываем файл для записи

    if (!file.is_open()) { // проверяем, успешно ли открыт файл
        std::cerr << "Ошибка!\n";
        return 1;
    }

    file.put('H'); // записываем символ 'H' в файл
    file.put('e'); // записываем символ 'e' в файл
    file.put('l'); // записываем символ 'l' в файл
    file.put('l'); // записываем символ 'l' в файл
    file.put('o'); // записываем символ 'o' в файл

    file.close(); // закрываем файл
    return 0;
}
```

Результат работы программы и вывод данных в файл:



Если же нужно записать целиком строку в файл, вы можете использовать функцию `std::fstream::write()`, которая записывает блок данных заданного размера в файл:

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

int main() {
```

```

setlocale(LC_ALL, "rus");
std::string str = "Hello, World!";

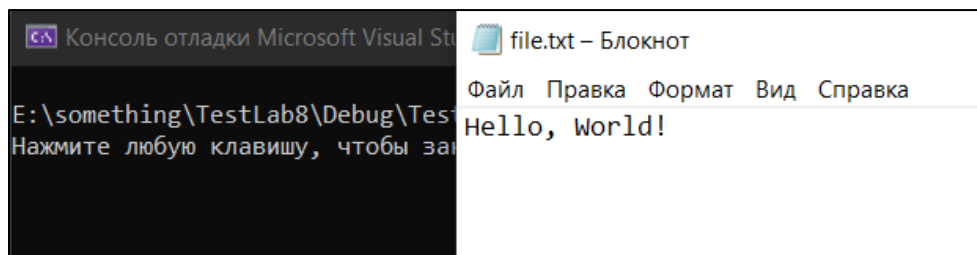
std::fstream file("file.txt", std::ios::out | std::ios::binary); // открываем файл для записи в
бинарном режиме

if (!file.is_open()) { // проверяем, успешно ли открыт файл
    std::cerr << "Ошибка!\n";
    return 1;
}

file.write(str.c_str(), str.size()); // записываем строку в файл
file.close(); // закрываем файл
return 0;
}

```

Результат работы программы и вывод данных в файл:



В этом примере мы создаем объект `std::string` с содержимым "Hello, World!". Затем мы открываем файл `example.txt` для записи в бинарном режиме с помощью объекта `std::fstream`. Мы проверяем, был ли файл успешно открыт с помощью метода `is_open()`. Если файл был успешно открыт, мы записываем строку в файл с помощью функции `write()`, передавая ей указатель на начало строки (`str.c_str()`) и размер строки (`str.size()`). Наконец, мы закрываем файл с помощью метода `close()`. Если файл не был успешно открыт, мы выводим сообщение об ошибке на стандартный поток ошибок (`std::cerr`).

2. Бинарные файлы:

Бинарный режим при работе с файлами в C++ используется для работы с данными в бинарном формате, то есть для записи данных в файл без какой-либо обработки или изменения.

При открытии файла в бинарном режиме с помощью флага `std::ios::binary`, все операции чтения и записи выполняются в режиме байтовых блоков, без какой-либо интерпретации содержимого файла. Это означает, что

данные, которые вы записываете в файл, будут сохраняться в файле без какой-либо дополнительной обработки, а данные, которые вы читаете из файла, будут возвращаться точно в том виде, в котором они были записаны в файл.

Некоторые типы данных в C++ (например, строки и числа с плавающей запятой) имеют различный внутренний формат хранения, который может быть изменен при записи в текстовый файл. При работе в бинарном режиме данные будут сохраняться в файл в их оригинальном бинарном формате, что может быть полезно при сохранении или передаче данных в бинарном формате, когда сохранение точной формы данных является важным условием.

Бинарный режим может использоваться совместно с другими флагами, такими как `std::ios::in` и `std::ios::out`, которые управляют режимами чтения и записи файлов.

2. Задание к лабораторной работе №3

Написать программу, которая считывает данные из текстового файла "input.txt" и записывает их в двоичный файл "output.bin". Каждая строка текстового файла должна быть записана в двоичный файл как последовательность байт, при этом каждая строка должна оканчиваться нулевым байтом.

После того, как все строки были записаны в двоичный файл, программа должна считать данные из "output.bin" и вывести их на экран в текстовом формате, при этом каждая строка должна быть выведена на новой строке.

3. Контрольные вопросы к лабораторной работе №3

1. В чем заключается главное отличие между двоичными и текстовыми файлами?
2. Как с помощью языка программирования открыть двоичный файл?
3. Как записать данные в двоичный файл в C++? Какая библиотека используется для этого?
4. Каким образом можно считать данные из двоичного файла? Как обрабатывать непредвиденные ситуации, такие как наличие ошибок в процессе чтения файла?
5. Каким образом происходит запись и чтение текстовых файлов? Каким образом можно модифицировать содержимое текстового файла?

Лабораторная работа №4

Индексирование записей. Простой/сложный индекс

Цель работы: изучение принципов и методов индексирования записей в структурах вашей БД, а также практическое овладение созданием простых и сложных индексов для оптимизации производительности запросов; изучение основных операций над структурами и ее определением.

1. Теоретическое сведение

1. Определение структуры:

Структура в C++ представляет собой производный тип данных, который представляет какую-то определенную сущность. Структура может определять переменные, функции, конструкторы, деструкторы. Однако обычно структуры служат для хранения каких-то общедоступных данных в виде публичных переменных.

Для определения структуры применяется ключевое слово `struct`, а сам формат определения выглядит следующим образом:

```
struct имя_структуры {  
    компоненты_структуры  
};
```

Имя_структуры представляет произвольный идентификатор, к которому применяются те же правила, что и при наименовании переменных. После имени структуры в фигурных скобках помещаются компоненты структуры - переменные и функции.

Например, определим простейшую структуру:

```
#include <iostream>  
using namespace std;  
  
struct person {  
    unsigned age;  
    string name;  
};  
  
int main() {  
    person tom;  
    tom.name = "Tom";
```

```
tom.age = 34;
cout << "Name: " << tom.name << "\tAge: " << tom.age << endl;
}
```

Здесь определена структура person, которая имеет две переменных: name (представляет тип string) и age (представляет тип unsigned).

Результат работы программы:

```
Name: Tom      Age: 34
```

2. Операции со структурами:

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	&

Операции со структурами:

- Присваивание полю структуры значение того же типа;
- Можно получить адрес структуры. Операция взятия адреса (&);
- Можно обращаться к любому полю структуры;
- Для того, чтобы определить размер структуры, можно использовать операцию sizeof().

Рассмотрим пример структуры student, описание которой выглядит так:

```
struct student {
    char name[25];
    int id, age;
}
```

Указатель new_student определен как:

```
struct student* new_student;
```

Предположим, что память выделена таким образом, чтобы new_student указывал на объект student. Тогда на компоненты этого объекта можно ссылаться следующим образом:

```
(*new_student).name
(*new_student).id
(*new_student).age
```

Поскольку указатели часто используются для указания на структуры, в языке Си специально для ссылок на компоненты таких структур введен оператор выбора стрелка вправо `->`. Например, ссылки на вышеприведенные компоненты структуры можно записать с использованием оператора стрелки вправо `->` как:

```
new_student->name  
new_student->id  
new_student->age
```

3. Индексирование:

Под индексом понимают средство ускорения операции поиска записей в таблице, а следовательно, и других операций, использующих поиск: извлечение, модификация, сортировка и т. д. Структуру, для которой используется индекс, называют индексированной.

Индексирование данных — это техника, повышающая скорость и эффективность запросов к базе данных. Она создаёт отдельную структуру данных, находящуюся в одном поле с данными структуры, что позволяет быстро находить строки по конкретному запросу без необходимости обхода всей структуры. Применяются разные типы индексов, однако они занимают пространство и должны обновляться при изменении данных.

Простой индекс представляет собой структуру данных, которая содержит список ссылок на записи в структуре. Простой индекс может быть создан для любой структуры, которая часто используется в запросах. Для создания простого индекса необходимо определить тип индекса и поле в структуре, по которому индекс будет создан.

Пример кода использования простого индекса:

```
#include <iostream>  
#include <vector>  
#include <algorithm>  
  
using namespace std;  
  
struct Person { // определяется структура Person, которая представляет информацию о человеке  
    int id;  
    string name;  
    int age;  
};
```

```

struct SimpleIndex { // определяется структура SimpleIndex, которая представляет простой
индекс для поиска по заданному полю
    vector<int> index;

    SimpleIndex( vector<Person>& people, string field) { // создается конструктор , который
заполняет вектор index информацией о человеке и поле
        if (field == "id") {
            for (int i = 0; i < people.size(); i++) {
                index.push_back(people[i].id);
            }
        }
        else if (field == "age") {
            for (int i = 0; i < people.size(); i++) {
                index.push_back(people[i].age);
            }
        }
    }

    vector<int> find(int value) { // создается метод для поиска элемента со значением value в
векторе index
        vector<int> result;
        for (int i = 0; i < index.size(); i++) {
            if (index[i] == value) {
                result.push_back(i);
            }
        }
        return result;
    }
};

int main() {
    vector<Person> people = {
        {1, "Alice", 25},
        {2, "Bob", 30},
        {3, "Charlie", 35},
        {4, "David", 40},
    };
    SimpleIndex id_index(people, "id");
    vector<int> ids = id_index.find(3);
    for (int i = 0; i < ids.size(); i++) {
        cout << people[ids[i]].name << endl;
    }
    return 0;
}

```

Результат работы программы:

```
Charlie
Для продолжения нажмите любую клавишу . . .
```

В данном примере создается простой индекс по полю "id". Индекс хранит список идентификаторов людей и позволяет быстро найти людей по их идентификаторам.

Сложный индекс представляет собой структуру данных, которая содержит несколько полей из структуры и ссылки на соответствующие записи. Он может быть создан для нескольких полей, что позволяет создавать более точные запросы. Для создания сложного индекса необходимо определить тип индекса и поля структуры, которые будут включены в индекс.

Пример кода с использованием сложного индекса в структуре:

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

using namespace std;

struct Person { // определяется структура Person, которая представляет информацию о человеке
    int id;
    string name;
    int age;
};

struct ComplexIndex { // определяется структура SimpleIndex, которая представляет простой
    индекс для поиска по заданному полю
    vector<vector<int>> index;
    vector<Person>& people;

    ComplexIndex(vector<Person>& people) : people(people) { // создается конструктор ,
        который заполняет вектор index информацией о человеке и поле
        for (int i = 0; i < people.size(); i++) {
            vector<int> index_element;
            index_element.push_back(people[i].age);
            index_element.push_back(people[i].id);
            index.push_back(index_element);
        }
        sort(index.begin(), index.end());
    }

    vector<int> find(string name, int age) { // создается метод для поиска элементов со
        значением value в векторе index
```

```

    vector<int> result;
    for (int i = 0; i < index.size(); i++) {
        if (people[i].name == name && index[i][0] == age) {
            result.push_back(index[i][1]);
        }
    }
    return result;
}
};

int main() {
    vector<Person> people = {
        {1, "Alice", 25},
        {2, "Bob", 30},
        {3, "Charlie", 35},
        {4, "David", 40},
    };
    ComplexIndex index(people);
    vector<int> ids = index.find("Bob", 30);
    for (int i = 0; i < ids.size(); i++) {
        cout << ids[i] << endl;
    }
    return 0;
}

```

Результат работы программы:

```

2
Для продолжения нажмите любую клавишу . . .

```

В данном примере мы создаем сложный индекс для списка людей. Индекс представляет собой вектор "имя" и "возраст" каждого человека, который сортируется по значению "имени". Затем мы можем использовать метод `find` для поиска людей, чье имя и возраст соответствуют заданным значениям.

4. Рассмотрим в качестве примера таблицу с данными о студентах и запрос о выборе всех студентов из некоторого города N :

Если не использовать никаких специальных ухищрений, и если записи отношения не упорядочены в соответствии с алфавитным порядком значений поля Город, то для решения данной задачи должны быть последовательно просмотрены *все* записи таблицы и из них отобраны те, у которых значения атрибута Город равны заданному в условии выборки значению « N ». При этом, реальное количество отобранных записей может быть существенно меньше общего числа просмотренных при выполнении запроса записей.

Выполнение описанной задачи может быть значительно ускорено, если создать, как это показано на рис. 1.4 дополнительную структуру данных, так называемый *индексный файл* городов, или просто индекс (*index* – указатель). В этом файле представлены все значения поля Город файла, соответствующего таблице Студент, но уже физически упорядоченные по алфавиту, с указателями на соответствующие записи файла таблицы Студент. Поиск нужного города в индексном файле может быть осуществлен существенно быстрее, чем в исходной таблице.

Во-первых, из-за упорядоченного по алфавиту расположения наименований городов, благодаря чему не нужно просматривать все до одной записи файла.

Во-вторых, физические размеры индексного файла существенно меньше файла таблицы Студент, для его размещения требуется меньше физических страниц и чтение его записи с диска будет происходить существенно быстрее.

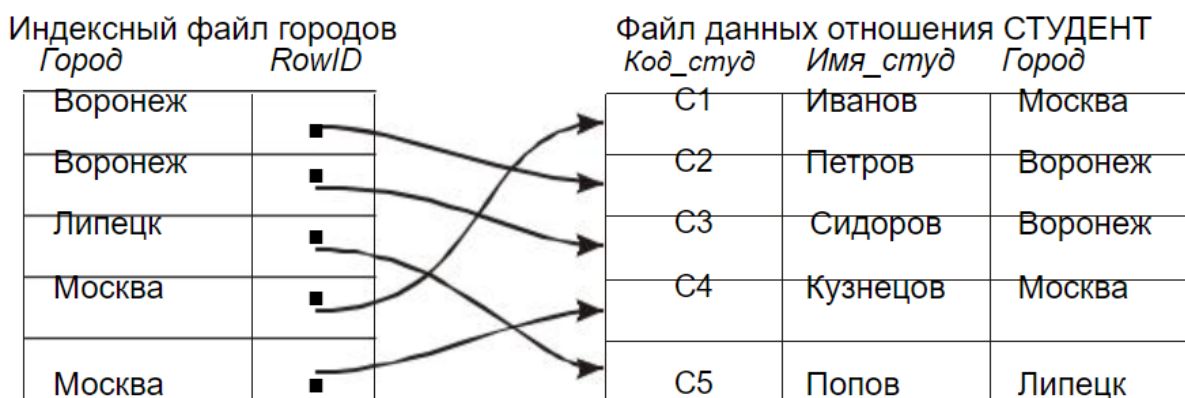


Рисунок 1- Использование индексирования для ускорения доступа к записям о студентах

Недостатком рассмотренной выше и представленной на рис.1.4 структуры индекса является то, что эффективность такого индекса будет падать с ростом числа записей структуры. В частности из-за того, что размер индексного файла также будет увеличиваться и, в конце концов, занимать больше места в памяти.

Таким образом использование индексов может значительно повысить производительность запросов, особенно при работе с большими объемами данных. Однако, создание индексов также может замедлить процесс добавления и обновления данных, поэтому необходимо балансировать

использование индексов в зависимости от конкретных потребностей приложения.

2. Задание к лабораторной работе №4

Для вашей БД (из лабораторной работы №2) для каждой структуры создать поле для индекса данных. Разработать функционал линейного поиска данных по индексу структуры. Сделать ввод индекса автоинкрементным.

3. Контрольные вопросы к лабораторной работе №4

1. Что такое индекс?
2. Что такое индексирование?
3. Какие бывают виды индексирования?
4. Что из себя представляет простой и сложный индекс?
5. Что такое структура? Как она инициализируется?
6. Какие операции существуют над структурами?
7. Какие бывают указатели на поля структуры? Чем они отличаются?

Лабораторная работа №5

Запись/чтение массива структур в файл

Цель работы: изучить работу с массивами структур; изучить функционал записи/чтения в файл/из файла массива структур; изучить вложенность структур; разобрать примеры структур в виде аргументов функции

1. Теоретическое сведение

1. Массивы структур:

Массив структур – это массив, каждый элемент которого является структурой. В памяти элементы массива структур размещаются последовательно.

Из элементов структурного типа можно организовать массивы также как из элементов стандартного типа. Для объявления массива структур следует сначала определить структуру, а затем объявить массив переменных данного типа. Как и массивы переменных, массивы структур индексируются с нуля.

Определяются массивы структур так же, как и массивы других типов данных. Естественное отличие - служебное слово `struct` в названии структурного типа (если обозначение типа не введено с помощью `typedef`). Для введенных выше структурных типов можно определить:

```
struct person people[10];
```

В данном случае определен массив структур `person` из 10 элементов.

Инициализация структуры в языке C (C++) происходит так же, как и при инициализации массива.

Используем массив структур в программе:

```
#include <stdio.h>

struct person {
    int age;
    char name[20];
};

int main(void) {
    struct person people[] = { {23, "Tom"}, {32, "Bob"}, {26, "Alice"}, {41, "Sam"} };
}
```

```

int n = sizeof(people) / sizeof(people[0]);
for (int i = 0; i < n; i++) {
    printf("Name:%s \t Age: %d \n", people[i].name, people[i].age);
}
return 0;
}

```

В массиве people определено 4 объекта person. При инициализации данные каждой отдельной структуры заключаются во вложенные фигурные скобки:

```
{ {23, "Tom"}, {32, "Bob"}, {26, "Alice"}, {41, "Sam"} }
```

Обращение к элементам массива структур происходит по индексу people[0]. А чтобы обратиться к элементу структуры из массива, после индекса указывается имя элемента структуры: people[i].name

Результат работы программы:

```

Name:Tom      Age: 23
Name:Bob      Age: 32
Name:Alice    Age: 26
Name:Sam      Age: 41

```

И также как с массивами других типов с массивами структур можно использовать указатели:

```

#include <iostream>
#include <cstring>

using namespace std;

struct Person {
    int age;
    char name[20];
};

int main() {
    Person people[] = { {23, "Tom"}, {32, "Bob"}, {26, "Alice"}, {41, "Sam"} };
    int n = sizeof(people) / sizeof(people[0]);

    for (Person* p = people; p < people + n; p++) {
        cout << "Name: " << p->name << "\tAge: " << p->age << endl;
    }

    return 0;
}

```

Здесь в массиве `people` те же 4 элемента `person`. Для их просмотра создан указатель `*p`, который устанавливается на начало массива `people`. И в цикле получаем элементы структур через этот указатель. После завершения каждой итерации указатель увеличивается на единицу, то есть перемещается в памяти на количество байт, которые занимает одна структура. И эти действия продолжаются пока указатель не дойдет до конца массива, который можно получить через выражение `people+n`.

Чтобы получить доступ к конкретной структуре в массиве структур, необходимо индексировать имя структуры. Например, чтобы отобразить на экране содержимое члена `name` третьей структуры, используют следующую инструкцию:

```
cout << people[2].name;
```

Подобно всем переменным массивов, у массивов структур индексирование начинается с нуля. При определении элементов массива структур применяются те же правила, что используются для отдельных структур: сопровождаем имя структуры [индекс] операцией получения элемента «`.`» и именем элемента:

`people [0].name`, где `name` – первый элемент массива.

`people [4].name`, где `name` – пятый элемент массива.

Члены структуры могут быть не только стандартный типов (`char`, `int`, `float` и т.д.), но также и описанных ранее структурных типов. Например, у всех людей есть анкетные данные. Есть они и у студента, помимо информации о группе и, например, на каком курсе учится.

Можно было бы описать две структуры – `struct man` и `struct student`. В первой описать поля с анкетными данными, а во второй – тоже поля с анкетными данными, а также о группе и курсе. Но это нерационально. Удобнее при декларации `struct student` ввести член типа описанной ранее `struct man`, а в последней имеются анкетные данные.

```
#include<stdio.h>

void main() {
    struct man {
        char name[15];
        char adres[20];
        int age;
    };
};
```

```
struct student {  
    struct man anketa;  
    char group[10];  
    int curs;  
};  
  
struct man x;  
struct student y;
```

Обращение к полю вложенной структуры:

имя_переменной_структурного_типа.поле_структуры.поле_вложенной_структуры.

2. Использование структур в качестве аргументов функции:

Единственно возможные операции над структурами – это их копирование, присваивание, взятие адреса с помощью & и осуществление доступа к ее элементам. Копирование и присваивание также включают в себя передачу функциям аргументов и возврат ими значений. Структуры нельзя сравнивать. Инициализировать структуру можно списком константных значений ее элементов, автоматическую структуру также можно инициализировать присваиванием.

Возможна передача членов структур функциям и передача целых структур функциям. При передаче функции члена структуры передается его значение, притом не играет роли то, что значение берется из члена структуры.

Например, пусть задана структура следующего вида:

```
struct fred {  
    char x;  
    int y;  
    float z;  
    char str[10]; // с учетом символа окончания строки  
} mike; // экземпляр структуры
```

Тогда каждый член этой структуры можно передать функции:

```
func(mike.x); // передается символьное значение x  
func2(mike.y); // передается целое значение y  
func3(mike.z); // передается значение с плавающей точкой z  
func4(mike.str); // передается адрес строки str[10]  
func(mike.str[2]); // передается символьное значение str[2]
```

Если же нужно передать адрес отдельного члена структуры, то перед именем структуры должен находиться оператор&. Для рассмотренных примеров будем иметь:

```
func(&mike.x); // передается адрес символа x
func2(&mike.y); // передается адрес целого y
func3(&mike.z); // передается адрес члена z с плавающей точкой
func4(mike.str); // передается адрес строки str
func(&mike.str[2]); // передается адрес символа в str[2]
```

Когда в качестве аргумента функции используется структура, для передачи целой структуры используется обычный способ вызова по значению. Это означает, что любые изменения в содержании параметра внутри функции не отразятся на той структуре, которая передана в качестве аргумента.

При использовании структуры в качестве параметра надо помнить, что тип аргумента должен соответствовать типу параметра.

3. Запись массива в файл:

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

Пример с использованием динамического vector массива:

```
#include <iostream>
#include <cstring>
#include <fstream>
#include <vector>
#include <Windows.h>

using namespace std;

struct Person {
    int age;
    string name;
};

vector<Person> people;
Person temp_doc;
int action;

void SavingData() {
    // создается поток для записи
    // открывает fileName и делает так, чтобы он был пустой
    ofstream record("data.txt", ios::out);
    // условие: если файл открылся
```

```

    if (record) {
        for (int i = 0; i < people.size(); i++) {
            record.write((char*)&people[i], sizeof Person); // переводим string в массив
            СИМВОЛОВ И ЗАПОЛНЯЕМ файл ЭТИМ МАССИВОМ

        }
        cout << "Данные записаны!" << endl;
    }
    else
        cout << "Ошибка открытия файла!" << endl;
    record.close();
}

void GettingData() {
    fstream file("data.txt");
    people.clear();

    while (file.read((char*)&temp_doc, sizeof Person)) { // переводим string в массив
        СИМВОЛОВ И ЧИТАЕМ ИЗ файла ЭТОТ МАССИВ
        people.push_back(temp_doc);
    }
    cout << "Массив получен! \n";
    cout << "Список данных: \n";
    for (const auto& person : people) {
        cout << "Age: " << person.age << ", Name: " << person.name << endl;
    }
}

void Menu() {
    cout << "1. Ввод массива в файл. \n"
        << "2. Вывод массива из файла. \n";
    cin >> action;
    cin.ignore(); // очистка буфера ввода от символа новой строки
}

int main() {
    SetConsoleCP(1251); // установка кодовой страницы win-cp 1251 в поток ввода
    SetConsoleOutputCP(1251); // установка кодов страницы win-cp 1251 в поток вывода
    setlocale(LC_ALL, "rus");
    Person pers1;
    pers1.age = 15;
    pers1.name = "Vadim";
    people.push_back(pers1);
    Person pers2;
    pers2.age = 30;
    pers2.name = "Andrew";
}

```

```

people.push_back(pers2);
Menu();
switch (action) {
case 1:
    SavingData();
    Menu();
    break;
case 2:
    GettingData();
    Menu();
    break;
default:
    cout << "Вы выбрали не тот пункт!";
    break;
}
return 0;
}

```

Результат работы программы:

```

1. Ввод массива в файл.
2. Вывод массива из файла.
2
Массив получен!
Список данных:
Age: 15, Name: Vadim
Age: 30, Name: Andrew
Для продолжения нажмите любую клавишу . . .

```

2. Задание к лабораторной работе №5

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.

3. Контрольные вопросы к лабораторной работе №5

1. Что такое массив структур? Чем он отличается от обычного массива?
2. Каким образом создается массив структур?
3. Как получить доступ к конкретной структуре?
4. В чем заключается вложенность структур?
5. Как связаны структуры с аргументами функций?
6. Как используются массивы структур?

7. С помощью чего происходит запись массива структур в файл/из файла?

Лабораторная работа №6

Редактирование файлов: удаление, изменение поля

Цель работы: изучить принцип работы с файлами; узнать о способах редактирования файла и изменении полей данных файла;

1. Теоретическое сведение

1. Инструкция typedef:

Язык C (C++) позволяет определять имена новых типов данных с помощью ключевого слова `typedef`. На самом деле здесь не создается новый тип данных, а определяется новое имя существующему типу. Объявления `typedef` можно использовать для создания более коротких и содержательных имен стандартным типам данных, определенных в языке, или для типов, которые объявили пользователи. Пример:

```
typedef int integer;
```

где `int` – это тип, которому присваиваем новое имя, а `integer` – это то обозначение, которое будет использоваться вместо `int`.

Имена `typedef` разделяют пространство имен с обычными идентификаторами. Поэтому программа может иметь имя `typedef` и идентификатор локальной области с одним и тем же именем.

Структура является типом данных созданным пользователем, к ней можно применять такие же операции, как и к встроенным типам.

2. Бинарные файлы:

Бинарные файлы обычно используются для обработки данных, состоящих из структур, чтение и запись которых удобно выполнять блоками.

Функция `fread (p, size, n, f)` выполняет чтение из файла «f» «n» блоков размером «size» байтов каждый в область памяти, адрес которой «p». В случае успеха функция возвращает количество считанных блоков.

Функция `fwrite (p, size, n, f)` выполняет запись в файл «f» «n» блоков размером «size» байтов каждый из области памяти, с адресом «p».

3. Позиционирование в файле:

Каждый открытый файл имеет скрытый указатель на текущую позицию в нем. При открытии файла этот указатель устанавливается на позицию, определенную режимом, и все операции в файле будут выполняться с данными, начинающимися в этой позиции. При каждом чтении (записи) указатель смещается на количество прочитанных (записанных) байтов – это последовательный доступ к данным.

С помощью функции `fseek` можно выполнить чтение или запись данных в произвольном порядке. Доступ к файлу с использованием этой функции называют произвольным доступом.

Функция `fseek (f, size, code)` выполняет смещение указателя файла `f` на `size` байтов в направлении `code` :

- Смещение от начала;
- Смещение от текущей позиции;
- Смещение от конца файла.

Смещение может быть как положительным, так и отрицательным, но нельзя выходить за пределы файла. В случае успеха функция возвращает 0, при ошибке возвращает 1, например, при выходе за пределы файла.

Пример удаления данных из файла:

```
file.seekp(pos); // перейти на позицию, с которой начнется удаление
file.write("", len); // записать пустые данные на нужную длину
```

Пример изменения поля в файле:

```
file.seekp(pos); // перейти на позицию, которую нужно изменить
file.write(new_value, len); // записать новое значение поля на нужную длину
```

Здесь `pos` - позиция в файле, где нужно удалить или изменить данные, `len` - длина данных, которые нужно удалить или записать, `new_value` - новое значение поля. Позиция в файле можно задавать в байтах с помощью методов `seekg()` и `seekp()` библиотеки `fstream`.

Рассмотрим некоторые полезные функции:

1) `ftell (f)` – определяет значение указателя на текущую позицию в файле, `-1` в случае ошибки;

2) `fileno (f)` – определяет значение дескриптора (`fd`) файла `f`, т.е. число, определяющее номер файла;

3) `fseek(fd, 0, SEEK_END)` – определяет длину файла в байтах, имеющего дескриптор `fd`;

4) `ftruncate(fd, pos)` – выполняет изменение размера файла, имеющего номер `fd`, признак конца файла устанавливается после байта с номером `pos`;

5) `feof(f)` – возвращает `true` при достижении конца файла.

6) `remove(имя_файла)` – удаляет файл с именем «имя_файла».

7) `rename(имя_файла, новое_имя_файла)` – переименовывает файл.

Вот пример кода, который демонстрирует запись структуры в текстовый файл и возможности редактирования файлов:

```
#include <iostream>
#include <fstream>
#include <cstring>

using namespace std;

struct Person {
    char name[50];
    int age;
    double salary;
};

int main() {
    // создаем структуру Person и заполняем ее данными
    Person p = { "John Doe", 30, 1000.0 };

    // записываем структуру в файл
    ofstream fout("person.txt");
    if (fout.is_open()) {
        fout << p.name << endl;
        fout << p.age << endl;
        fout << p.salary << endl;
        fout.close();
    }
    else {
        cout << "Error opening file" << endl;
        return -1;
    }

    // изменяем значение поля salary
    p.salary = 1500.0;

    // открываем файл для чтения и записи
```

```

fstream file("person.txt", ios::in | ios::out);
if (file.is_open()) {
    // переходим на позицию поля salary
    file.seekp(strlen(p.name) + sizeof(int) + 1);

    // записываем новое значение поля
    file.write(reinterpret_cast<const char*>(&p.salary), sizeof(p.salary));

    // закрываем файл
    file.close();
}
else {
    cout << "Error opening file" << endl;
    return -1;
}

// удаляем данные из файла
file.open("person.txt", ios::in | ios::out);
if (file.is_open()) {
    // переходим на позицию поля age
    file.seekp(strlen(p.name) + 1);

    // удаляем данные поля age
    file.write("", sizeof(int));

    // закрываем файл
    file.close();
}
else {
    cout << "Error opening file" << endl;
    return -1;
}
return 0;
}

```

Результатом программы будет текстовый файл с содержимым:

```

John Doe
p-@

```

Пример кода для удаления файла с записанной в него структурой:

```

#include <iostream>
#include <cstdio>

```

```

#pragma warning(disable : 4996)

using namespace std;

struct Person {
    char name[50];
    int age;
    double salary;
};

int main() {
    // создаем структуру Person и заполняем ее данными
    Person p = { "John Doe", 30, 1000.0 };

    // записываем структуру в файл
    FILE* file = fopen("person.txt", "wb");
    if (file != nullptr) {
        fwrite(&p, sizeof(p), 1, file);
        fclose(file);
    }
    else {
        cout << "Error opening file" << endl;
        return -1;
    }

    // удаляем файл
    if (remove("person.txt") == 0) {
        cout << "File deleted successfully." << endl;
    }
    else {
        cout << "Error deleting file." << endl;
        return -1;
    }

    return 0;
}

```

Результатом программы будет сообщение об успешном/неуспешном удалении файла:

```

File deleted successfully.
Для продолжения нажмите любую клавишу . . .

```

2. Задание к лабораторной работе №6

Написать функции удаления (выборочное и полное) и изменения массива структур для вашей БД (из лабораторной работы №2).

3. Контрольные вопросы к лабораторной работе №6

1. Какие новые имена типов данных можно создать с помощью typedef в языке C++?
2. Для чего обычно используются бинарные файлы?
3. Как устанавливается текущая позиция в открытом файле?
4. Что происходит при чтении из файла с помощью функции fread?
5. Как происходит запись в файл с помощью функции fwrite?
6. Как с помощью функции fseek выполнить чтение или запись данных в произвольном порядке?
7. Какие значения может принимать второй аргумент функции fseek, и что они означают?
8. Каковы преимущества использования функции ftell при работе с файлами?
9. Как можно задать позицию в файле для удаления или изменения данных с помощью библиотеки fstream?
10. Какие полезные функции для работы с файлами есть в языке C++?

Лабораторная работа №7

Сортировка записей

Цель работы: изучить различные виды сортировки и реализовать их для структуры данных.

1. Теоретическое сведение

1. Определение сортировки

Под сортировкой понимается процесс перегруппировки элементов массива, приводящий к их упорядоченному расположению относительно ключа.

Цель сортировки – облегчить последующий поиск элементов. Метод сортировки называется устойчивым, если в процессе перегруппировки относительное расположение элементов с равными ключами не изменяется. Основное условие при сортировке массивов – это не вводить дополнительных массивов, т.е. все перестановки элементов должны выполняться в исходном массиве. Сортировку массивов принято называть внутренней, а сортировку файлов – внешней.



2. Методы сортировок:

Методы внутренней сортировки классифицируются по времени их работы. Хорошей мерой эффективности может быть число операций сравнений ключей и число пересылок (перестановок) элементов.

Ключ — это часть информации, определяющая порядок элементов. Таким образом, ключ участвует в сравнениях, но при обмене элементов происходит перемещение всей структуры данных. Для простоты в

нижеследующих примерах будет производиться сортировка массивов, в которых ключ и данные совпадают.



2.1 Сортировка выбором:

Сортировка выбором (англ. selection sort) — алгоритм сортировки, относящийся к неустойчивым алгоритмам сортировки. На массиве из n элементов имеет время выполнения в худшем, среднем и лучшем случае $O(n^2)$, предполагая что сравнения делаются за постоянное время.

Сложность по времени

Худшее время: $O(n^2)$

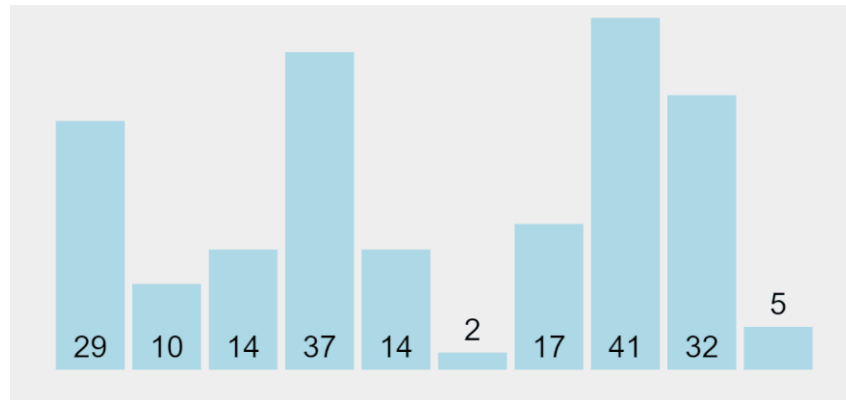
Среднее время: $O(n^2)$

Лучшее время: $O(n^2)$

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

Идея метода состоит в том, чтобы создавать отсортированную последовательность путем присоединения к ней одного элемента за другим в правильном порядке.

Алгоритм состоит из n последовательных шагов, начиная от нулевого и заканчивая $(n-1)$ -м. На i -м шаге выбираем наименьший из элементов $a[i] \dots a[n]$ и меняем его местами с $a[i]$. Последовательность шагов при $n=5$ изображена на рисунке ниже.



Пример кода для сортировки выбором:

```
for (int i = 0; i < N; i++) {  
    min = i;  
  
    for (int j = i + 1; j < N; j++)  
        min = ( a[j] < a[min] ) ? j : min;  
  
    if (i != min) {  
        buf = a[i];  
        a[i] = a[min];  
        a[min] = buf;  
    }  
}
```

2.2 Сортировка вставками:

Сортировка вставками (англ. insertion sort) — простой алгоритм сортировки. Хотя этот метод сортировки намного менее эффективен, чем более сложные алгоритмы (такие как быстрая сортировка), у него есть ряд преимуществ:

- Прост в реализации
- Эффективен на небольших наборах данных

- Эффективен на наборах данных, которые уже частично отсортированы
- Это устойчивый алгоритм сортировки (не меняет порядок элементов, которые уже отсортированы)
- Может сортировать список по мере его получения

Сложность по времени

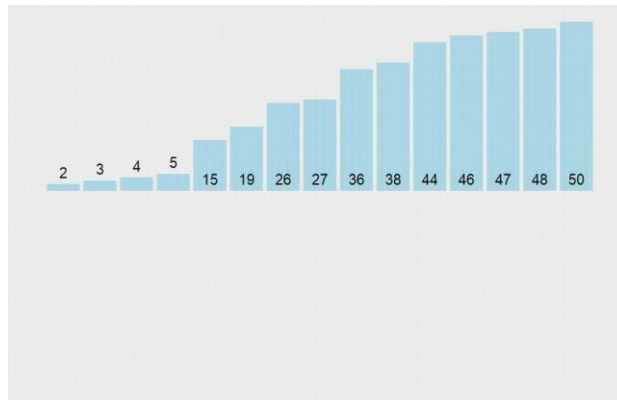
Худшее время: $O(n^2)$ для сравнений и перестановок

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

Аналогичным образом делаются проходы по части массива, и аналогичным же образом в его начале "вырастает" отсортированная последовательность. Массив постепенно перебирается слева направо. При этом каждый последующий элемент размещается так, чтобы он оказался между ближайшими элементами с минимальным и максимальным значением.



Пример кода для сортировки вставками:

```
for (i = 1; i < N; i++) {
    buff = a[i];
    for (j = i - 1; j >= 0 && a[j] > buff; j--)
        a[j + 1] = a[j];

    a[j + 1] = buff;
}
```

2.3 Сортировка слиянием:

Сортировка слиянием (англ. merge sort) — алгоритм сортировки, который упорядочивает списки (или другие структуры данных, доступ к элементам которых можно получать только последовательно, например — потоки) в определённом порядке. Эта сортировка — хороший пример использования принципа «разделяй и властвуй». Сначала задача разбивается на несколько подзадач меньшего размера. Затем эти задачи решаются с помощью рекурсивного вызова или непосредственно, если их размер достаточно мал. Наконец, их решения комбинируются, и получается решение исходной задачи.

Сложность по времени

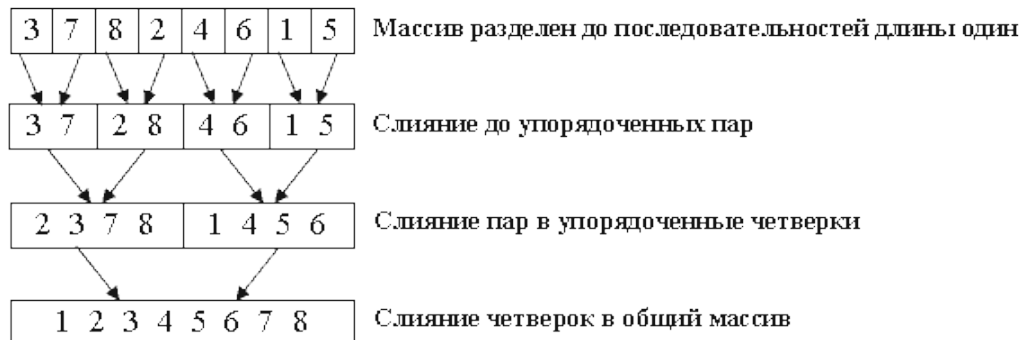
Худшее время: $O(n \log n)$

Среднее время: $O(n \log n)$

Лучшее время: $O(n \log n)$

Затраты памяти: $O(n)$ вспомогательной

Другими словами, он делит исходный массив на более мелкие массивы, пока каждый маленький массив не будет содержать всего одну позицию, а затем сливает маленькие массивы в более крупный и отсортированный.



Пример кода для сортировки слиянием:

```
void Merge(int* A, int first, int last) { //функция, сливающая массивы
    int middle, start, final, j;
    int* mas = new int[100];
    middle = (first + last) / 2; //вычисление среднего элемента
    start = first; //начало левой части
    final = middle + 1; //начало правой части
```

```

    for (j = first; j <= last; j++) //выполнять от начала до конца
        if ((start <= middle) && ((final > last) || (A[start] < A[final]))) {
            mas[j] = A[start];
            start++;
        }
        else {
            mas[j] = A[final];
            final++;
        }
    for (j = first; j <= last; j++) A[j] = mas[j]; //возвращение результата в список
    delete[] mas;
};

void MergeSort(int* A, int first, int last) { //рекурсивная процедура сортировки
    if (first < last)
    {
        MergeSort(A, first, (first + last) / 2); //сортировка левой части
        MergeSort(A, (first + last) / 2 + 1, last); //сортировка правой части
        Merge(A, first, last); //слияние двух частей
    }
};

```

2.4 Быстрая сортировка:

Быстрая сортировка подобно алгоритму сортировки слиянием, этот алгоритм также использует подход «разделяй и властвуй».

Шаги в быстрой сортировке:

- Выбираем в массиве некоторый элемент, который будем называть опорным элементом.
- Операция деления массива: реорганизуем массив таким образом, чтобы все элементы, меньшие или равные опорному элементу, оказались слева от него, а все элементы, большие опорного — справа от него.
- Рекурсивно упорядочиваем подпоследовательности, лежащие слева и справа от опорного элемента.

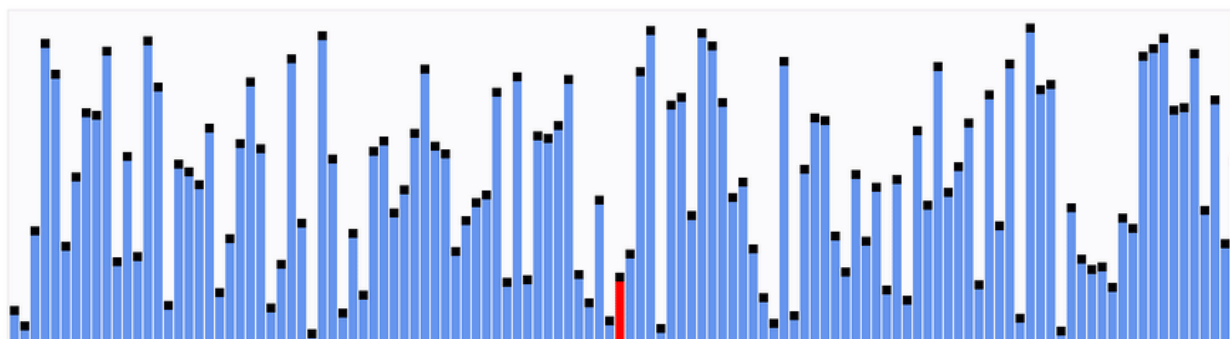
Сложность по времени

Худшее время: $O(n^2)$
 Среднее время: $O(n \log n)$
 Лучшее время: $O(n)$

Затраты памяти: $O(n)$



Базой рекурсии являются списки, состоящие из одного или двух элементов, которые уже упорядочены. Алгоритм всегда завершается, поскольку за каждую итерацию он ставит по крайней мере один элемент на его окончательное место.



Пример кода для быстрой сортировки:

```
void quicksort(int* mas, int first, int last) { //функция сортировки
    int mid, count;
    int f = first, l = last;
    mid = mas[(f + l) / 2]; //вычисление опорного элемента
    do {
        while (mas[f] < mid) f++;
        while (mas[l] > mid) l--;
        if (f <= l) { //перестановка элементов
            count = mas[f];
            mas[f] = mas[l];
            mas[l] = count;
            f++;
            l--;
        }
    } while (f < l);
    if (first < l) quicksort(mas, first, l);
    if (f < last) quicksort(mas, f, last);
}
```

2.5 Пузырьковая сортировка:

Сортировка пузырьком (англ. bubble sort) — простой алгоритм сортировки. Для понимания и реализации этот алгоритм — простейший, но эффективен он лишь для небольших массивов.

Сложность по времени

Худшее время: $O(n^2)$

Среднее время: $O(n^2)$

Лучшее время: $O(n)$

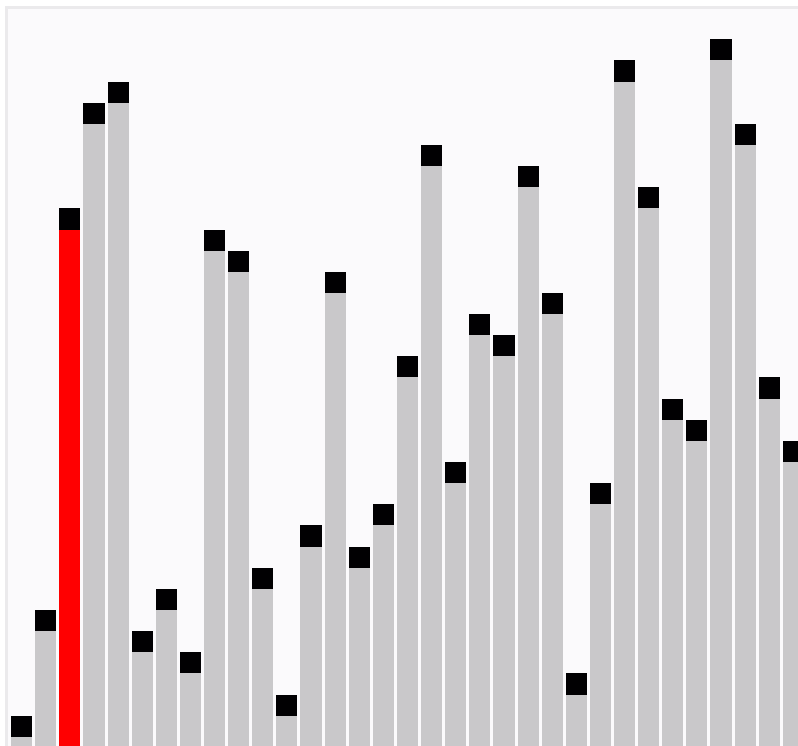
Затраты памяти: $O(1)$

Идея метода: шаг сортировки состоит в проходе снизу вверх по массиву. По пути просматриваются пары соседних элементов. Если элементы некоторой пары находятся в неправильном порядке, то меняем их местами.



После нулевого прохода по массиву «вверх» оказывается самый «легкий» элемент - отсюда аналогия с пузырьком. Следующий проход делается до второго сверху элемента, таким образом второй по величине элемент поднимается на правильную позицию...

Делаем проходы по все уменьшающейся нижней части массива до тех пор, пока в ней не останется только один элемент. На этом сортировка заканчивается, так как последовательность упорядочена по возрастанию.



Пример кода для пузырьковой сортировки:

```
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]);  
            }  
        }  
    }  
}
```


2.6 Шейкерная сортировка:

Шейкерная сортировка отличается от пузырьковой тем, что она двунаправленная: алгоритм перемещается не строго слева направо, а сначала слева направо, затем справа налево.

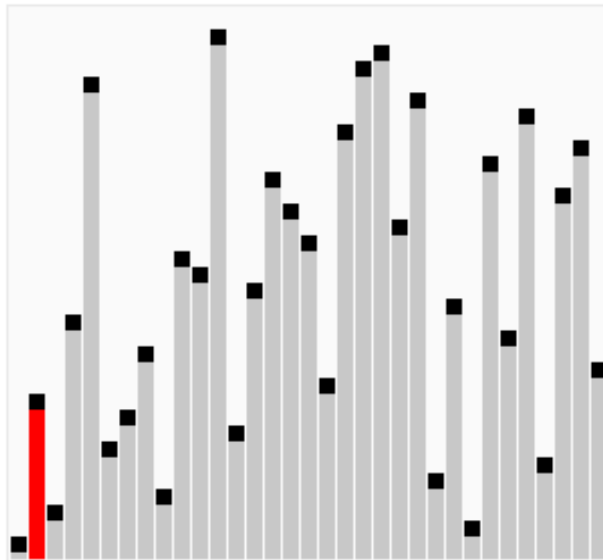
Сложность по времени

Худшее время: $O(n^2)$

Среднее время: $O(n^2)$

Лучшее время: $O(n)$

Затраты памяти: $O(1)$



Пример кода для шейкерной сортировки:

```
void ShakerSort(int* a, int n) {  
    int left, right, i;  
    left = 0;  
    right = n - 1;  
    while (left <= right) {  
        for (i = right; i >= left; i--) {  
            if (a[i - 1] > a[i]) {  
                swap(a[i - 1], a[i]);  
            }  
        }  
        left++;  
        for (i = left; i <= right; i++) {
```

```
        if (a[i - 1] > a[i]) {  
            swap(a[i - 1], a[i]);  
        }  
    }  
    right--;  
}  
}
```

2. Задание к лабораторной работе №7

В соответствии со своей БД из лабораторной работы №2, реализовать все вышеперечисленные сортировки (например, по возрастанию) по каким-либо параметрам.

3. Контрольные вопросы к лабораторной работе №7

1. Что такое сортировка?
2. Что такое устойчивый метод сортировки и почему это важно?
3. Какое основное условие при сортировке массивов?
4. Чем отличается внутренняя сортировка массивов от внешней сортировки файлов?
5. Какую хорошую меру эффективности можно использовать при оценке работы метода сортировки?
6. Что такое ключ в контексте сортировки массива?
7. Как работает метод сортировки выбором?
8. Как работает метод сортировки вставками и какие преимущества у него есть?
9. Какой подход использует быстрая сортировка и как он похож на алгоритм сортировки слиянием?
10. Какие шаги применяются в быстрой сортировке?
11. Что такое сортировка пузырьком и как она работает?
12. Для каких массивов эффективна сортировка пузырьком?
13. Чем отличается шейкерная сортировка от пузырьковой?

Лабораторная работа №8

Фильтрация данных.

Цель работы: изучить фильтрацию информации в некоторой структуре данных.

1. Теоретическое сведение

Фильтр - это быстрый и легкий способ поиска подмножества данных и работы с ними в списке. В отфильтрованном списке отображаются только строки, отвечающие условиям.

1. Классический поиск подстроки в строке:

Рассмотрим пример, позволяющий понять, как работает поиск подстроки в строке. Допустим у нас есть две строки, например, «Hello world» и «lo».

1-й шаг цикла	H	E	L	L	O		W	O	R	L	D
	L										
2-й шаг цикла	H	E	L	L	O		W	O	R	L	D
		L									
3-й шаг цикла	H	E	L	L	O		W	O	R	L	D
			L	O							
4-й шаг цикла	H	E	L	L	O		W	O	R	L	D
				L	O						

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.

На третьей итерации совпал только первый символ искомой строки, а вот второй уже не совпадает. Первый цикл продолжает работу. Четвертая итерация дает необходимые результаты – совпадают все символы искомой строки с частью исходной строки. Работа алгоритма закончена.

Пример программы, находящий все вхождения подстроки «с» в строку «s» и выводящий позиции, с которых начинаются эти подстроки.

```
#include <iostream>

// Функция для поиска подстроки в строке + поиск позиции, с которой начинается подстрока
int pos(const char* s, const char* c, int n) {
    //s - указатель на исходную строку, c - указатель на подстроку, n - вхождения строки
    int i, j; // Счетчики для циклов
    int lenC, lenS; // Длины строк
    // Находим размеры строки искомой и строки исходника
    for (lenC = 0; c[lenC]; lenC++);
    for (lenS = 0; s[lenS]; lenS++);

    // Перебираем все возможные позиции, где может находиться искомая подстрока
    for (i = 0; i <= lenS - lenC; i++) {
        // Проверяем, совпадают ли символы в строке искомой и строке исходника
        for (j = 0; j < lenC && s[i + j] == c[j]; j++);

        // Если символы совпали по длине искомой строки и указанному номеру, то
        // возвращаем позицию
        if (j == lenC && !(n - 1))
            return i;

        // Если символы совпали по длине искомой строки, но не по указанному номеру,
        // продолжаем поиск
        if (j == lenC)
            if (n - 1)
                n--;
            else
                return i;
    }
    return -1; // Иначе вернем -1 как результат отсутствия подстроки
}

// Основная функция
int main() {
    const char* s = "рагарара"; // Исходная строка
    const char* c = "ра"; // Искомая подстрока
    int i, n = 0;

    // Поиск всех вхождений искомой подстроки в исходной строке
```

```

for (i = 1; n != -1; i++) {
    n = pos(s, c, i); // Вызов функции поиска позиции i-ого вхождения подстроки c в строке s
    if (n >= 0)
        std::cout << n << std::endl; // Вывод позиции вхождения искомой подстроки
    }

    system("pause");
    return 0;
}

```

Результат работы программы:

```

0
4
6
Для продолжения нажмите любую клавишу . . .

```

Алгоритм поиска всех вхождений подстроки в строку, реализованный в функции `pos()`, не является оптимальным, так как он осуществляет последовательный поиск в цикле, что может занять много времени, особенно если в строке много вхождений. Более эффективные алгоритмы поиска подстроки могут значительно уменьшить время поиска в больших строках с множеством вхождений.

2. Функция `find()`:

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

```

#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

    if (text.find("banana") == std::string::npos) {
        std::cout << "Not found" << std::endl;
    }
}

```

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 (как видно из примера, число 18446744073709551615). И при поиске мы можем проверять результат функции `find()` на равенство этой константе.

Функция `find` имеет ряд дополнительных версий. Так, с помощью второго параметра мы можем указать индекс, с которого надо вести поиск:

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

int main() {
    std::string text{ "A friend in need is a friend indeed." };
    // поиск с 10-го индекса
    std::cout << text.find("friend", 10) << std::endl;    // 22
}
```

Используя эту версию, мы можем написать программу для поиска количества вхождений строки в тексте, то есть выяснить, сколько раз строка встречается в тексте:

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

int main() {
    std::string text{ "A friend in need is a friend indeed." };
    std::string word{ "friend" };
    unsigned count{ 0 };    // количество вхождений
    for (unsigned i{ 0 }; i <= text.length() - word.length(); ) {
        // получаем индекс
        size_t position = text.find(word, i);
        // если не найдено ни одного вхождения с индекса i, выходим из цикла
        if (position == std::string::npos) break;
        // если же вхождение найдено, увеличиваем счетчик вхождений
        ++count;
        // переходим к следующему индексу после position
        i = position + 1;
    }
    std::cout << "The word is found " << count << " times." << std::endl; // The word is found 2
    times.
}
```

Здесь в цикле пробегаемся по тексту, в котором надо найти строку, пока счетчик `i` не будет равен `text.length() - word.length()`. С помощью функции `find()` получаем индекс первого вхождения слова в тексте, начиная с индекса `i`. Если таких вхождений не найдено, то выходим из цикла. Если же найден индекс, то счетчик `i` получает индекс, следующий за индексом найденного вхождения.

В качестве альтернативы мы могли бы использовать цикл while:

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

int main() {
    std::string text{ "A friend in need is a friend indeed." };
    std::string word{ "friend" };
    unsigned count{};    // количество вхождений
    size_t index{}; // начальный индекс
    while ((index = text.find(word, index)) != std::string::npos) {
        ++count;
        index += word.length(); // перемещаем индекс на позицию после завершения слова в
        тексте
    }
    std::cout << "The word is found " << count << " times." << std::endl;
}
```

Еще одна версия позволяет искать в тексте не всю строку, а только ее часть. Для этого в качестве третьего параметра передается количество символов из искомой строки, которые программа будет искать в тексте:

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

int main() {
    std::string text{ "A friend in need is a friend indeed." };
    std::string word{ "endless" };
    // поиск с 10-го индекса 3 первых символов слова "endless", то есть "end"
    std::cout << text.find("endless", 10, 3) << std::endl;    // 25
}
```

Стоит отметить, что в этом случае искомая строка должна представлять строковый литерал или строку в С-стиле (например, символьный массив с концевым нулевым байтом).

2. Задание к лабораторной работе №8

Для вашей БД из лабораторной работы № 2 реализовать фильтрацию, используя классический поиск подстроки в строке и функцию find() (например, по фамилии).

3. Контрольные вопросы к лабораторной работе №8

1. Что такое фильтр?
2. Как работает классический поиск подстроки в строке?
3. Как реализуется функция `find()`? Какие 3 аргумента она может принимать?

Лабораторная работа №9

Поиск записи по значению/индексу

Цель работы: ознакомление с различными методами поиска, такими как линейный, бинарный и интерполирующий поиски.

1. Теоретическое сведение

В языке C++ есть несколько способов поиска записей по значению или индексу, в зависимости от того, какой тип данных используется для хранения записей и какой метод поиска наиболее подходит для конкретной задачи. Рассмотрим некоторые из них:

1. Линейный поиск либо последовательный поиск:

Линейный поиск является наиболее простым и наименее эффективным алгоритмом поиска в массиве. Он работает путем последовательного перебора элементов массива и сравнения их со значением, которое нужно найти. При обнаружении искомого значения функция возвращает его индекс в массиве, в противном случае возвращается специальное значение (обычно -1). В силу своей простоты, линейный поиск может быть полезен в некоторых ситуациях, но для больших массивов он может быть слишком медленным.

Пример функции, реализующий линейный поиск в массиве:

```
int linear_search(int arr[], int size, int x) {  
    for (int i = 0; i < size; i++) {  
        if (arr[i] == x) {  
            return i; // Возвращаем индекс элемента, если он найден  
        }  
    }  
    return -1; // Возвращаем -1, если элемент не найден  
}
```

Приведем пример программы, реализующую поиск значения в массиве методом линейного поиска. Программа заполняет массив случайными числами в диапазоне от 1 до 50 и предлагает пользователю ввести число, которое нужно найти в массиве. Затем происходит поиск этого числа в массиве методом линейного поиска, и найденные индексы элементов, содержащих это число, выводятся на экран.

```
#include <iostream>  
#include <iomanip>
```

```

#include <ctime>
using namespace std;

// Прототипы функций
int linSearch(int arr[], int requiredKey, int size, int indexes[]); // Линейный поиск
void showArr(int arr[], int size); // Показ массива

int main() {
    setlocale(LC_ALL, "rus");
    const int arrSize = 50;
    int arr[arrSize];
    int requiredKey = 0; // Искомое значение (ключ)
    int indexes[arrSize]; // Массив для хранения индексов
    int nElements = 0; // Количество найденных элементов
    srand(time(NULL));

    // Запись случайных чисел в массив от 1 до 50
    for (int i = 0; i < arrSize; i++) {
        arr[i] = 1 + rand() % 50;
    }

    showArr(arr, arrSize);

    cout << "Какое число необходимо искать? ";
    if (!(cin >> requiredKey)) {
        cout << "Ошибка: введено некорректное значение!" << endl;
        return 1; // Завершаем программу с кодом ошибки
    }

    // Поиск искомого числа и запись номера элемента
    nElements = linSearch(arr, requiredKey, arrSize, indexes);

    if (nElements > 0) {
        // Если в массиве найдено искомое число - выводим индексы элементов на
        экран
        cout << "Значение " << requiredKey << " найдено в ячейках с индексами: ";
        for (int i = 0; i < nElements; i++) {
            cout << indexes[i] << " ";
        }
        cout << endl;
    } else {
        // Если в массиве не найдено искомое число
        cout << "В массиве нет такого значения" << endl;
    }
    return 0;
}

```

```

// Вывод массива на экран
void showArr(int arr[], int arrSize) {
    for (int i = 0; i < arrSize; i++) {
        cout << setw(4) << arr[i];
        if ((i + 1) % 10 == 0) {
            cout << endl;
        }
    }
    cout << endl << endl;
}

// Линейный поиск
int linSearch(int arr[], int requiredKey, int arrSize, int indexes[]) {
    int count = 0;
    for (int i = 0; i < arrSize; i++) {
        if (arr[i] == requiredKey) {
            indexes[count] = i;
            count++;
        }
    }
    return count;
}

```

Результат работы программы:

```

13 41 34 26 4 6 26 18 13 12
17 50 43 31 29 40 45 17 43 49
49 33 45 48 46 28 34 1 20 44
20 12 39 14 10 4 20 18 46 44
14 36 5 30 50 30 7 30 26 13

Какое число необходимо искать? 13
Значение 13 найдено в ячейках с индексами: 0 8 49

```

Линейный поиск обычно используется для поиска единственного элемента в небольшом несортированном массиве. Если массив большой или содержит множество элементов, то эффективнее будет отсортировать его и применить другие алгоритмы поиска, такие как двоичный (бинарный) поиск.

2. Двоичный или бинарный поиск:

Двоичный (бинарный) поиск по значению - это алгоритм поиска элемента в упорядоченном массиве данных. Он использует подход «разделяй и властвуй», разбивая массив на две половины на каждом шаге и проверяя, в какой половине находится искомый элемент. Если элемент найден, то

возвращается его индекс. Если элемент не найден, то возвращается специальное значение, обычно -1.

Пример функции, реализующий бинарный поиск в массиве:

```
int binarySearch(int arr[], int left, int right, int x) {  
    while (left <= right) { // Пока в массиве есть как минимум один элемент, который  
        еще не был проверен  
        int middle = (left + right) / 2; // Вычисление индекса середины отрезка  
  
        if (arr[middle] == x) {  
            return middle;  
        }  
  
        if (arr[middle] < x) {  
            left = middle + 1; // Сдвигаем левую границу поиска, чтобы  
            продолжить поиск в правой части массива  
        } else {  
            right = middle - 1; // Сдвигаем правую границу поиска, чтобы  
            продолжить поиск в левой части массива  
        }  
    }  
    return -1;  
}
```

Предположим, что массив из 12-ти элементов отсортирован по возрастанию.

		ind 0	ind 1	ind 2	ind 3	ind 4	ind 5	ind 6	ind 7	ind 8	ind 9	ind 10	ind 11	
исходный массив	///	1	2	3	4	5	6	7	8	9	10	11	12	///

Пользователь задает искомое значение. Допустим 4. На первой итерации массив делится на две части (ищем средний элемент – middle): $(0 + 11) / 2 = 5$ (0.5 отбрасываются). Сначала проверяется значение среднего элемента массива. Если оно совпадает со значением – алгоритм прекратит работу и программа выведет сообщение, что значение найдено. В нашем случае, ключ не совпадает со значением среднего элемента.

		ind 0	ind 1	ind 2	ind 3	ind 4	ind 5	ind 6	ind 7	ind 8	ind 9	ind 10	ind 11	
I шаг	///	1	2	3	4	5	6	7	8	9	10	11	12	///
		left					midd						right	

Если ключ меньше значения среднего элемента, алгоритм не будет проводить поиск в той половине массива, которая содержит значения больше

ключа (т.е. от среднего элемента до конца массива). Правая граница поиска сместится ($\text{midd} - 1$). Далее снова деление массива на 2.

		ind 0	ind 1	ind 2	ind 3	ind 4	ind 5	ind 6	ind 7	ind 8	ind 9	ind 10	ind 11	
II war	///	1	2	3	4	5	///	///	///	///	///	///	///	
		left		midd		right								

Ключ снова не равен среднему элементу. Он больше него. Теперь левая граница поиска сместится ($\text{midd} + 1$).

		ind 0	ind 1	ind 2	ind 3	ind 4	ind 5	ind 6	ind 7	ind 8	ind 9	ind 10	ind 11	
III war	///	///	///	///	4	5	///	///	///	///	///	///	///	
					left / midd	right								

На третьей итерации средний элемент – это ячейка с индексом 3: $(3 + 4) / 2 = 3$. Он равен ключу. Алгоритм завершает работу.

Приведем пример программы, реализующий алгоритм двоичного поиска, чтобы найти индекс элемента в упорядоченном массиве, который соответствует введенному пользователем значению ключа. Программа заполняет и показывает массив целых чисел от 1 до 12, затем запрашивает у пользователя значение ключа и выполняет двоичный поиск в массиве. Если ключ найден в массиве, программа выводит индекс ячейки массива, в которой находится значение ключа.

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

// Функция с алгоритмом двоичного поиска
int binarySearch(int arr[], int left, int right, int key) {
    while (left <= right) { // Пока левая граница не пересечется с правой
        int middle = (left + right) / 2; // Находим индекс среднего элемента

        if (arr[middle] == key) { // Если ключ найден в среднем элементе
            return middle; // Возвращаем его индекс
        }

        if (arr[middle] < key) { // Если ключ больше среднего элемента
            left = middle + 1; // Ищем в правой половине массива
        } else {
            right = middle - 1; // Ищем в левой половине массива
        }
    }
}
```

```

    }
    return -1; // Если ключ не найден, возвращаем -1
}

int main() {
    setlocale(LC_ALL, "rus");

    const int SIZE = 12;
    int array[SIZE] = {}; // Объявляем и инициализируем пустой массив
    int key = 0; // Переменная для хранения ключа поиска
    int index = 0; // Индекс ячейки с искомым значением

    for (int i = 0; i < SIZE; i++) { // Заполняем и показываем массив
        array[i] = i;
        cout << array[i] << " | ";
    }

    cout << "\n\nВведите любое число: ";
    cin >> key;
    index = binarySearch(array, 0, SIZE, key);

    if (index >= 0) { // Если индекс найденного элемента не равен -1
        cout << "Указанное число находится в ячейке с индексом: " << index <<
"\n\n";
    } else {
        cout << "В массиве нет такого числа!\n\n";
    }
    return 0;
}

```

Результат работы программы:

```

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
Введите любое число: 11
Указанное число находится в ячейке с индексом: 11

```

Один из главных недостатков бинарного поиска заключается в том, что он требует, чтобы данные, с которыми он работает, были упорядочены. Это означает, что если данные не упорядочены, то нужно будет потратить время на их сортировку перед применением алгоритма. Кроме того, бинарный поиск не может работать с такими структурами данных, как неупорядоченные массивы или связанные списки, потому что он требует постоянного времени доступа к элементам данных. Это означает, что если алгоритму нужно

перебирать элементы для поиска нужного, то это будет занимать значительно больше времени, чем простой доступ к элементу по его индексу.

3. Интерполирующий поиск:

Интерполирующий поиск - это алгоритм поиска элемента в упорядоченном массиве, который работает на основе линейной интерполяции.

Пример функции, реализующий интерполирующий поиск в массиве:

```
int interpolationSearch(int arr[], int size, int x) {
    int left = 0;
    int right = size - 1;
    int middle = 0;

    // Пока диапазон поиска не будет пустым или пока не найден искомый элемент
    while (left <= right && x >= arr[left] && x <= arr[right]) {
        // Вычисляем позицию оценки элемента в массиве
        int middle = left + ((x - arr[left]) * (right - left)) / (arr[right] - arr[left]);

        // Если элемент найден, то возвращаем его позицию в массиве
        if (arr[middle] == x)
            return middle;

        // Если оценка элемента меньше искомого, то продолжаем поиск в правой
        // части массива
        if (arr[middle] < x)
            left = middle + 1;

        // Если оценка элемента больше искомого, то продолжаем поиск в левой
        // части массива
        else
            right = middle - 1;
    }

    // Если элемент не найден, то возвращаем -1
    return -1;
}
```

Интерполяционный поиск является улучшением бинарного поиска для равномерно распределенных отсортированных массивов. Вместо использования фиксированного шага, он использует линейную интерполяцию для оценки позиции искомого элемента в массиве.

Идея заключается в следующем: если элементы в массиве равномерно распределены, то можно оценить приблизительную позицию искомого элемента, используя формулу:

```
middle = left + ((requiredKey - arr[left]) * (right - left)) / (arr[right] - arr[left]);
```

Здесь x - искомый элемент, $arr[left]$ и $arr[right]$ - первый и последний элементы в диапазоне поиска соответственно.

После вычисления оценки позиции, интерполяционный поиск сравнивает искомый элемент со значением в оцененной позиции. Если элемент найден, то он возвращается. Если оцененный элемент меньше искомого, то поиск продолжается в правой части массива, иначе в левой.

Алгоритм продолжает поиск, пока диапазон поиска не станет пустым или пока не будет найден искомый элемент. Если элемент не найден, то функция возвращает -1.

Преимущество интерполяционного поиска заключается в том, что он работает быстрее, чем бинарный поиск для равномерно распределенных отсортированных массивов, особенно когда искомый элемент находится ближе к началу массива.

Пример программы, реализующий алгоритм интерполяционного поиска. Программа генерирует массив из 50 случайных чисел в диапазоне от 1 до 50, сортирует массив с помощью пузырьковой сортировки, а затем выполняет поиск с интерполяцией, чтобы найти указанный пользователем ключ. Программа также отображает исходный и отсортированный массивы, а также индексы всех вхождений ключа в массив.

```
#include <iostream>
#include <iomanip>
#include <ctime>

using namespace std;

// Прототипы функций
int interpolationSearch(int arr[], int arrSize, int requiredKey, int indexes[]); //
Интерполяционный поиск
void bubbleSort(int arr[], int arrSize); // Сортировка пузырьком
void showArr(int arr[], int arrSize); // Показ массива
void showSortedArr(int arr[], int arrSize); // Вывод отсортированного массива

int main() {
    setlocale(LC_ALL, "rus");
    const int arrSize = 50;
    int arr[arrSize];
    int requiredKey = 0; // Искомое значение (ключ)
    int indexes[arrSize]; // Массив для хранения индексов
```



```

int nElements = 0; // Количество найденных элементов
srand(time(NULL));

// Запись случайных чисел в массив от 1 до 50
for (int i = 0; i < arrSize; i++) {
    arr[i] = 1 + rand() % 50;
}

showArr(arr, arrSize);

cout << "Какое число необходимо искать? ";
if (!(cin >> requiredKey)) {
    cout << "Ошибка: введено некорректное значение!" << endl;

    return 1; // Завершаем программу с кодом ошибки
}
// Сортировка массива
bubbleSort(arr, arrSize);
cout << "Отсортированный массив:" << endl;
showSortedArr(arr, arrSize);

// Интерполяционный поиск
nElements = interpolationSearch(arr, arrSize, requiredKey, indexes);

// Вывод результатов поиска
if (nElements == 0) {
    cout << "Элементов со значением " << requiredKey << " не найдено." << endl;
}
else {
    cout << "Найдено элементов со значением " << requiredKey << ": " << nElements <<
endl;
    cout << "Индексы найденных элементов: ";
    for (int i = 0; i < nElements; i++) {
        cout << indexes[i] << " ";
    }
    cout << endl;
}

return 0;
}

// Интерполяционный поиск
int interpolationSearch(int arr[], int arrSize, int requiredKey, int indexes[]) {
    int left = 0;
    int right = arrSize - 1;
    int middle = 0;

```

```

int nElements = 0;

while (left <= right && requiredKey >= arr[left] && requiredKey <= arr[right]) {
    middle = left + ((requiredKey - arr[left]) * (right - left)) / (arr[right] - arr[left]);
    if (arr[middle] == requiredKey) {
        indexes[nElements] = middle;
        nElements++;
    }
    if (arr[middle] < requiredKey) {
        left = middle + 1;
    }
    else {
        right = middle - 1;
    }
}

// Сортировка индексов
for (int i = 0; i < nElements - 1; i++) {
    for (int j = 0; j < nElements - i - 1; j++) {
        if (indexes[j] > indexes[j + 1]) {
            swap(indexes[j], indexes[j + 1]);
        }
    }
}

return nElements;
}

// Сортировка пузырьком
void bubbleSort(int arr[], int arrSize) {
    for (int i = 0; i < arrSize - 1; i++) {
        for (int j = 0; j < arrSize - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}

// Показ массива
void showArr(int arr[], int arrSize) {
    cout << "Исходный массив:" << endl;
    for (int i = 0; i < arrSize; i++) {
        cout << setw(2) << arr[i] << " ";
        if ((i + 1) % 10 == 0) {
            cout << endl;
        }
    }
}

```

```

    }
}
cout << endl;
}

// Вывод отсортированного массива
void showSortedArr(int arr[], int arrSize) {
    for (int i = 0; i < arrSize; i++) {
        cout << setw(2) << arr[i] << " ";
        if ((i + 1) % 10 == 0) {
            cout << endl;
        }
    }
    cout << endl;
}
}

```

Результат работы программы:

```

Исходный массив:
23 39 43  1 47 16 34 44 21  6
 7 36 29  3  5 20  4 12 50 24
39 32  9 17 16 44  8 15 26 46
30 29 14 24 13 33 17 26 26 44
23 21 26 21 43 35 29 31 46 50

Какое число необходимо искать? 44
Отсортированный массив:
 1  3  4  5  6  7  8  9 12 13
14 15 16 16 17 17 20 21 21 21
23 23 24 24 26 26 26 26 29 29
29 30 31 32 33 34 35 36 39 39
43 43 44 44 44 46 46 47 50 50

Найдено элементов со значением 44: 2
Индексы найденных элементов: 42 43

```

Несмотря на то, что интерполяционный поиск обычно работает быстрее, чем бинарный поиск в случае равномерно распределенных данных, он также имеет ряд недостатков:

1. Интерполяционный поиск не подходит для неупорядоченных данных. Для его работы необходимо, чтобы данные были отсортированы.
2. В случае не равномерного распределения данных интерполяционный поиск может работать хуже, чем бинарный поиск.
3. При наличии повторяющихся элементов в массиве, интерполяционный поиск может не находить все вхождения искомого элемента.

2. Задание к лабораторной работе №9

В соответствии с содержанием своей БД (из лабораторной работы № 2) реализовать алгоритм линейного и итерполирующего поиска индекса в вашем массиве структур и бинарный поиск по любому из типов данных (например, по фамилии).

3. Контрольные вопросы к лабораторной работе №9

1. Что такое линейный поиск? Как он реализуется?
2. Что такое бинарный поиск? Как он реализуется?
3. Что такое интерполирующий поиск? Как он реализуется?
4. Какой из поисков является наиболее эффективным? Почему?
5. Какой из поисков является наименее эффективным? Почему?
6. Назовите недостатки всех поисков, представленных в данной лабораторной работе.
7. Назовите преимущества всех поисков, представленных в данной лабораторной работе.

Лабораторная работа №10

Динамические структуры данных

Цель работы: научиться работать с выделением динамической памяти в структурах данных.

1. Теоретическое сведение

1. Выделение памяти:

Различают два типа памяти: статическую и динамическую. В статической памяти размещаются локальные и глобальные данные при их описании в функциях. Для временного хранения данных в памяти ЭВМ используется динамическая память, или heap. Размер этой памяти ограничен, и запрос на динамическое выделение памяти может быть выполнен далеко не всегда.

В C++ для операций выделения и освобождения памяти можно также использовать встроенные операторы `new` и `delete`.

Оператор `new` имеет один операнд. Оператор имеет две формы записи:

```
[::] new [(список_аргументов)] имя_типа  
[(инициализирующее_значение)]
```

```
[::] new [(список_аргументов)] (имя_типа)  
[(инициализирующее_значение)]
```

Для разрушения объекта, созданного с помощью оператора `new`, необходимо использовать в программе оператор `delete`.

Оператор `delete` имеет две формы записи:

```
[::] delete переменная_указатель    // для указателя на один элемент  
[::] delete [] переменная_указатель // для указателя на массив
```

Единственный операнд в операторе `delete` должен быть указателем, возвращаемым оператором `new`. Если оператор `delete` применить к указателю, полученному не посредством оператора `new`, то результат будет непредсказуем.

Использование оператора `delete` вместо `delete[]` по отношению к указателю на массив может привести к логическим ошибкам. Таким образом,

освобождать память, выделенную для массива, необходимо оператором delete [], а для отдельного элемента – оператором delete.

Пример кода использования операторов выше:

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

struct A {
    int i; // компонента-данное структуры A

    A() {} // конструктор структуры A

    ~A() {} // деструктор структуры A
};

int main() {
    A* a, * b; // описание указателей на объект структуры A

    float* c, * d; // описание указателей на элементы типа float

    a = new A; // выделение памяти для одного объекта структуры A

    b = new A[3]; // выделение памяти для массива объектов структуры A

    c = new float; // выделение памяти для одного элемента типа float

    d = new float[4]; // выделение памяти для массива элементов типа float

    delete a; // освобождение памяти, занимаемой одним объектом

    delete[] b; // освобождение памяти, занимаемой массивом объектов

    delete c; // освобождение памяти одного элемента типа float

    delete[] d; // освобождение памяти массива элементов типа float
}
```

Для динамических данных память выделяется и освобождается в процессе выполнения программы, а не в момент ее запуска. Так, например, если в программе объявлен массив из 100 элементов, то при запуске программы резервируется память для всех ста элементов, даже если в процессе работы программы всего будут использованы первые 10 элементов массива. С другой стороны, при использовании в программе динамических типов память под них заранее не выделяется. Лишь когда поступают новые данные,

вызывается специальная функция, которая выделяет память, куда эти данные записываются.

Тут появляется проблема. Для динамических типов данных не объявляются переменные, иначе память бы выделялась под переменные. Как тогда обращаться к данным, записанным неизвестно где в памяти?

Проблема решается путем использования структур. Допустим, мы пишем программу, позволяющую вводить данные на сотрудников организации. Количество сотрудников неизвестно. Можно было бы создать массив записей с запасом. Однако, если данных о каждом сотруднике много, то каждая запись занимает много памяти; получается, что мы будем расходовать много памяти в пустую, если сотрудников мало.

2. Практическое использование динамической памяти со структурами данных:

Работа с динамической памятью в контексте структуры данных может быть необходима, когда вы не знаете заранее количество элементов, которые будут храниться в вашей структуре данных. Например, вы можете захотеть создать массив структур данных, но не знать, сколько элементов будет в этом массиве.

Для структур данных, которые содержат переменное количество элементов, вы можете использовать указатели и операторы `new` и `delete`, чтобы динамически выделять и освобождать память.

Например, вот как можно создать структуру данных, которая хранит переменное количество элементов:

```
struct MyDataStructure {  
    int* data; // указатель на массив элементов  
    int size; // размер массива  
  
    MyDataStructure(int n) { // конструктор  
        size = n;  
        data = new int[size]; // выделение памяти под массив элементов  
    }  
  
    ~MyDataStructure() { // деструктор  
        delete[] data; // освобождение памяти  
    }  
};
```

В этом примере мы создали структуру данных `MyDataStructure`, которая содержит указатель на массив элементов и переменную `size`, которая

указывает на количество элементов в массиве. В конструкторе мы выделяем память для массива элементов, используя оператор new, и в деструкторе мы освобождаем эту память, используя оператор delete[].

Теперь мы можем создать объект этой структуры данных и хранить в нем массив элементов переменной длины:

```
int n = 10; // количество элементов
MyDataStructure ds(n); // создаем объект структуры данных

for (int i = 0; i < n; i++) {
    ds.data[i] = i; // инициализируем элементы массива
}
```

Обратите внимание, что мы используем оператор new в конструкторе структуры данных и оператор delete[] в деструкторе для выделения и освобождения памяти под массив элементов.

Работа с динамической памятью может быть удобным способом создания структур данных переменной длины, но также требует более осторожного программирования, чтобы избежать утечек памяти. Для этого нужно всегда освобождать выделенную память, когда она больше не нужна, а также убедиться, что не происходит обращение к уже освобожденной памяти.

Пример использования динамической памяти в структурах данных:

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

using namespace std;

struct Person {
    string name;
    int age;
};

int main() {
    // Выделяем память для одного объекта структуры Person
    Person* p = new Person;

    // Инициализируем поля структуры через указатель
    p->name = "John";
    p->age = 30;

    // Выводим данные на экран
```



```

cout << "Name: " << p->name << ", Age: " << p->age << endl;

// Освобождаем выделенную память
delete p;

// Выделяем память для массива объектов структуры Person
int n = 3;
Person* arr = new Person[n];

// Инициализируем поля структур через указатель
arr[0].name = "Jane";
arr[0].age = 25;

arr[1].name = "Bob";
arr[1].age = 40;

arr[2].name = "Alice";
arr[2].age = 20;

// Выводим данные на экран
for (int i = 0; i < n; i++) {
    cout << "Name: " << arr[i].name << ", Age: " << arr[i].age << endl;
}

// Освобождаем выделенную память
delete[] arr;

system("pause");
return 0;
}

```

Результат работы программы:

```

Name: John, Age: 30
Name: Jane, Age: 25
Name: Bob, Age: 40
Name: Alice, Age: 20
Для продолжения нажмите любую клавишу . . .

```

2. Задание к лабораторной работе №10

В соответствии со своей БД из лабораторной работы №2, реализовать функционал выделения и освобождения памяти по любому из параметров.

3. Контрольные вопросы к лабораторной работе №10

1. Какие два типа памяти различаются?
2. Какие данные хранятся в статической памяти?
3. Для чего используется динамическая память?
4. Как ограничен размер динамической памяти?
5. Какие встроенные операторы в C++ можно использовать для операций выделения и освобождения памяти?
6. Какие формы записи имеет оператор new?
7. Какие формы записи имеет оператор delete?
8. Что произойдет, если оператор delete применить к указателю, полученному не посредством оператора new?
9. В чем разница между использованием оператора delete и delete[]?
10. Как можно решить проблему обращения к данным, записанным в динамической памяти?